

# **Dynamic Task Scheduling and Binding for Many-Core Systems through Stream Rewriting**

**Dissertation**

**zur**

**Erlangung des akademischen Grades**

**Doktor-Ingenieur (Dr.-Ing.)**

**der Fakultät für Informatik und Elektrotechnik**

**der Universität Rostock**

vorgelegt von

Lars Middendorf, geb. am 21.09.1982 in Iserlohn  
aus Rostock

Rostock, 03.12.2014

## Gutachter

- Prof. Dr.-Ing. habil. Christian Haubelt  
Lehrstuhl "Eingebettete Systeme"  
Institut für Angewandte Mikroelektronik und Datentechnik  
Universität Rostock
- Prof. Dr.-Ing. habil. Heidrun Schumann  
Lehrstuhl Computergraphik  
Institut für Informatik  
Universität Rostock
- Prof. Dr.-Ing. Michael Hübner  
Lehrstuhl für Eingebettete Systeme der Informationstechnik  
Fakultät für Elektrotechnik und Informationstechnik  
Ruhr-Universität Bochum

**Datum der Abgabe:** 03.12.2014

**Datum der Verteidigung:** 05.03.2015

## **Acknowledgements**

First of all, I would like to thank my supervisor Prof. Dr. Christian Haubelt for his guidance during the years, the scientific assistance to write this thesis, and the chance to research on a very individual topic. In addition, I thank my colleagues for interesting discussions and a pleasant working environment. Finally, I would like to thank my family for supporting and understanding me.



# Contents

<b>1</b>	<b>INTRODUCTION.....</b>	<b>1</b>
1.1	STREAM REWRITING .....	5
1.2	RELATED WORK .....	7
1.3	CONTRIBUTIONS.....	13
1.4	PUBLICATIONS .....	16
<b>2</b>	<b>STREAM REWRITING .....</b>	<b>18</b>
2.1	STREAM REWRITING MACHINE .....	18
2.2	PARALLEL REWRITING .....	23
2.3	SCHEDULING .....	31
2.4	STREAM COMPRESSION .....	39
2.5	MEMORY ACCESS.....	42
<b>3</b>	<b>SOURCE MODELS .....</b>	<b>48</b>
3.1	FUNCTIONAL PROGRAMS .....	48
3.2	TASK GRAPHS.....	51
3.3	C SOURCE CODE .....	60
<b>4</b>	<b>MULTI-CORE ARCHITECTURES .....</b>	<b>65</b>
4.1	INTRODUCTION .....	65
4.2	SHARED MEMORY ARCHITECTURE .....	67
4.3	STREAM REWRITING NETWORK.....	74
4.4	COMPARISON .....	79
4.5	CONCLUSION.....	82
<b>5</b>	<b>GRAPHICS PROCESSING.....</b>	<b>83</b>
5.1	INTRODUCTION .....	83
5.2	RELATED WORK .....	85
5.3	GENERIC GRAPHICS PIPELINE.....	92
5.4	FUNCTIONAL PROCESSOR.....	96
5.5	SIMT SHADER CORE.....	103
5.6	GENERAL PURPOSE GRAPHICS PROCESSOR.....	114
<b>6</b>	<b>HIGH-LEVEL SYNTHESIS .....</b>	<b>128</b>
6.1	INTRODUCTION .....	128
6.2	RELATED WORK .....	129
6.3	IMPLEMENTATION .....	130
6.4	RESULTS .....	133
<b>7</b>	<b>CONCLUSION.....</b>	<b>137</b>
7.1	RESULTS .....	137
7.2	FUTURE WORK .....	138
<b>8</b>	<b>REFERENCES.....</b>	<b>141</b>



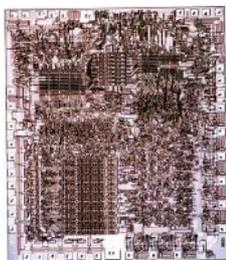
## 1 Introduction

The scalability of modern computer architectures is mostly limited by stagnating clock rates and increased energy consumption. Multi and many-core systems offer the ability to overcome these issues by performing computations on several functional units in parallel. Although a multi-core design usually consumes a larger area than an equivalent single-core design, it can offer a similar computational power at lower clock frequencies and thus helps to reduce thermal heat or power issues [1]. Thus, designing hardware and software architectures consequently towards concurrency and parallelism are currently the most effective and in the long term the best approach of accelerating an application [2].

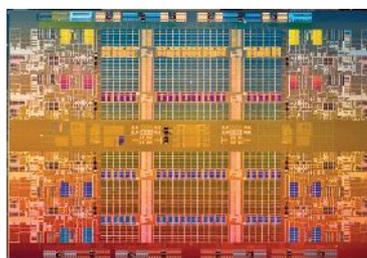
The concept of multiple cores can be recognized in general purpose processors [3] and graphics processing units [4], which are optimized for maximum performance, but also in mobile and embedded systems to perform computationally intensive tasks in resource constrained environments [5]. In particular, the integration of more functional units and their interconnections already involves significant technical problems to be solved at the hardware domain.

For instance, Figure 1 shows three example processors with a complexity ranging from 29K to 7.1B transistors. Although each generation can take advantage of technological improvements and shrunk structures, the simple duplication of arithmetic and logical blocks leads to an unconstrained grow of the power usage [6]. Hence, it is often better to spend the newly won capacity with optimized and therefore more specialized functionality, which results in complex heterogeneous systems. As a result, the integration of different processor types becomes possible but on the other hand, the utilization of these additional resources turns out to be a significant and currently unsolved issue.

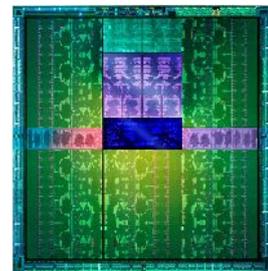
Due to its complexity, the task management is often moved to the software layer, which is now responsible for the binding and scheduling of concurrent work items to processor cores [7] and involves several different programming models (Section 1.2.1). When developing embedded real-time systems, the focus usually lies on meeting hard deadlines that require the completion of certain tasks within a given period of time. For these applications, a static schedule is usually pre-computed at design time to ensure correctness and predictability of the system.



Intel 8088, no cache,  
29K transistors [6]  
1979



Intel Nehalem, 8 cores,  
2.3B transistors [6]  
2010



NVidia GK110,  
7.1B transistors [8]  
2013

Figure 1. Photographs of various processors from the 8088 to modern GPUs.

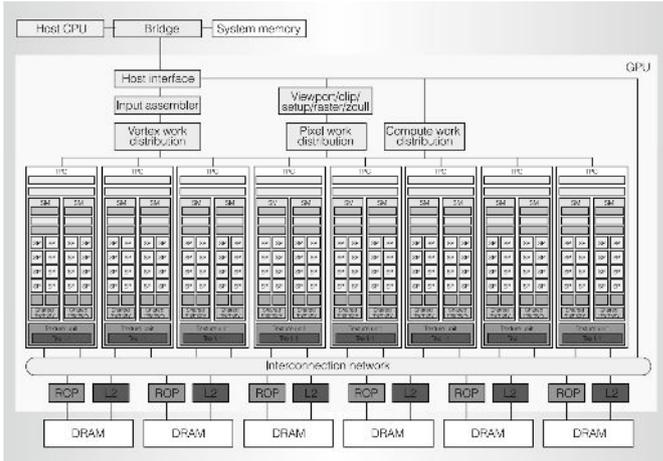


Figure 2. Architecture of the Tesla GPU [9].

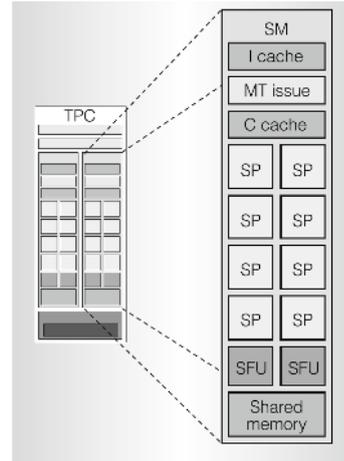


Figure 3. Streaming multiprocessor [9].

However, the creation of a static schedule can take a significant amount of time and is only feasible if the expected workload is known at design time. Hence, the software development for many-core architectures like the SCC processor with 48 cores [10], the 64-core tile processor [11] or graphics processing units [9] becomes challenging in case of dynamic parallelism. Especially for irregular workloads, the efficient communication between cooperating tasks induces several question not yet solved in general [12], so that the resulting schedule is usually not an exact solution [13].

Therefore, it seems remarkable that for the specific use case of graphics processing, many-core architectures with thousands of shader processors have become common and the performance of applications scales accordingly [14]. One reason can be found in the architecture of GPUs, which is adapted for the parallel workloads of graphics processing. For instance, the *Tesla* processor (Figure 2) contains a large array of streaming multiprocessors, which are fed by specialized units for vertex, pixel and compute work. In particular, each of these tasks, like the rasterization of a triangle, can be naturally partitioned into a sufficient number of smaller work items for each processor. For example, the hardware scheduler of the *Tesla* GPU creates a separate thread for each 2x2 quad of pixels. In particular, there are up to 768 threads per multiprocessor, which help to hide the latency of floating-point operations and memory request at moderate costs. On the other hand, the communication between threads is limited and the system actually requires a minimal workload for efficiency. Hence, this architecture is well-suited for data parallel problems with extensive floating-point arithmetic, which apply the same type of computation to a large number of input elements. However, in case of irregular data access and control flow like sorting or a binary searching, a general purpose CPU can outperform a GPU despite its several times higher theoretical arithmetic throughput [15].

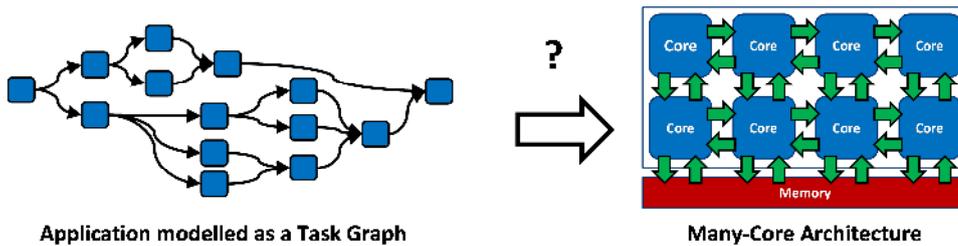


Figure 4. How to map a complex application with dynamic parallelism to many-core architectures?

One reason is the usage of the SIMT (single-instruction multiple-threads) execution model in GPUs, which evaluates the same instruction for a group of threads in parallel. In these architectures shader cores (SP) consist only of an arithmetic unit and share common functionality like the memory interface. Especially the instruction fetch and decode stages are shared with other cores in the same multiprocessor (Figure 3). Thus, in the optimal case, all cores of a multiprocessor are active and are working in parallel on the same instruction but a data dependent branches can lead to different control paths, which cause performance degradation (*incoherent control flow*). Hence, in order to take advantage of current GPUs, a problem must fit into the data parallel execution model.

As a result, the dynamic binding and scheduling of applications with numerous tasks has not been solved for many-core architecture in general (Figure 4). Instead, both general purpose CPUs and graphics processors have their particular strengths, so that the question arises how these contrary concepts can be combined. For this purpose, this thesis presents a novel execution model, which deals with the problem of dynamic binding and scheduling of irregular workload in a highly concurrent environment. Since several concepts from general purpose CPUs and graphics processors are combined, the relation between these architectures is compared in Figure 5.

On the left hand side, the single core CPU exclusively relies on instruction-level parallelism but is also well-suited for irregular data access due to a cache hierarchy. A branch predictor usually ensures that irregular control flow is handled efficiently and dynamic scheduling can be implemented in software. When moving towards the right hand side of this chart, parallelism, throughput, and performance increase but also the programming model becomes more specialized. Thus, a multi-core CPU enables task and thread-level parallelism

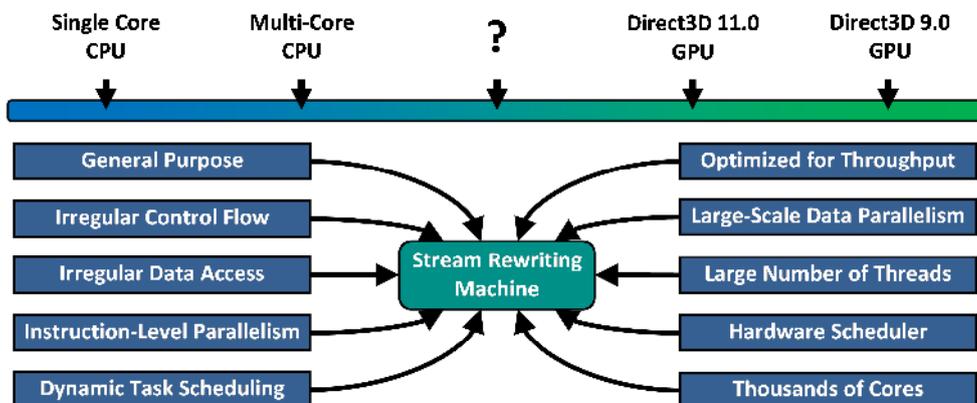


Figure 5. Comparison of CPU and GPU architectures.

but a program must be first partitioned into concurrently running processes. Unlike instruction-level parallelism, which can be automatically extracted using Tomasulo's Algorithm [16], the synchronization of separately running threads must be explicitly specified by the programmer and often adds undesirable complexity to an application.

While general purpose processors have been evolved into multi-core CPUs and are moving towards many-core architectures, graphics processors are already highly parallel and become more flexible instead. Each major version of Direct3D [17] introduces new shader types and adds more capabilities. For instance, the compute shader of Direct3D 11 [18] enables general purpose computations and random memory access. However, the application must be still adapted to the given programming model in order to take advantage of the large-scale data parallelism. Obviously, there is a tradeoff between generality and specialization for a particular problem, so that a GPU built from large general purpose cores would consume unnecessary area and power. However, in this thesis, it is questioned if there could possibly exist an intermediate class of parallel architectures, which inherits features from both CPUs and GPUs. For instance, it would be advantageous if a hardware scheduler, able to handle millions of threads, could be combined with irregular and general purpose computations. Due to the usage of a stream as the main computation model, these architectures are characterized as *stream rewriting machines (SRM)*. In a summary, this thesis deals with the following three questions:

**1. How can dynamic parallelism be modeled?**

A fundamental question is the description of applications in such a way, that they are scalable for many-core architectures. The basic task graph model contains a fixed number of nodes and can therefore not describe dynamic parallelism. Otherwise, C source code is a much more expressive representation but inherently sequential. Data parallel loop programs contain explicit parallelism but usually cannot describe the dynamic creation of individual tasks.

**2. How can the dynamic scheduling and binding of irregular tasks be performed?**

The proposed system should be able to handle a large number of unpredictable and frequently varying tasks as well as their dynamic creation and completion. Also, recursive tasks must be managed efficiently. As a result, the static binding of tasks to computational resources and static scheduling are not feasible. In addition, the method must be simple enough to be implemented in hardware.

**3. How can the hardware architecture support dynamic scheduling and binding?**

An important aspect is the architecture of many-core systems enforcing dynamic parallelism. Should the architecture provide support for task management and how can the interface between hardware and software be defined? Can the flexibility of a software implementation be combined with the throughput of a hardware scheduler?

Before detailing the contributions of this thesis, the proposed execution model of *stream rewriting* is presented in the next section.

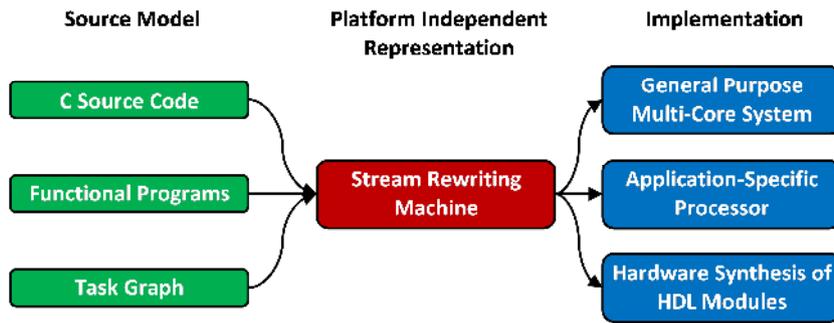


Figure 6. Input models and target platforms of stream rewriting.

## 1.1 Stream Rewriting

This thesis proposes an alternative approach for managing dynamic parallelism at the system level. The central innovation is a technique called *stream rewriting*, which acts as an intermediate and platform independent representation of concurrent programs (Figure 6). On the left hand side, several source models like imperative C code, functional programs and task graphs as shown. A significant part of this thesis deals with the translation of these models in a *stream rewriting machine* (SRM), which is the abstract model for stream rewriting. For example, an SRM can be derived from an imperative C program, functional languages as well as task graphs, which are often used to describe the dependencies in parallel application. Hence, stream rewriting is an abstract model for concurrency, which focuses especially on the aspect of dynamic parallelism and can describe both recursive functions but also dynamic expandable task graphs.

On the right hand side, there are several target platforms for the implementation of a concrete SRM. It can be either compiled into software running on general purpose processors or generic many-core architectures. In addition, the SRM can be synthesized into application-specific hardware modules.

An example design flow for stream rewriting is illustrated in Figure 7. At design time, the given application is modeled as a static or dynamic task graph. This graph can be converted into rewriting rules with the help of the decomposition tree which describes the topology as a structure of sequential and parallel blocks. Eventually, the decomposition tree can be translated into a functional program, which is then compiled into rewriting rules and evaluated at runtime using the stream rewriting machine. A more detailed description of this technique and a comparison of several implementations can be found in Section 3.

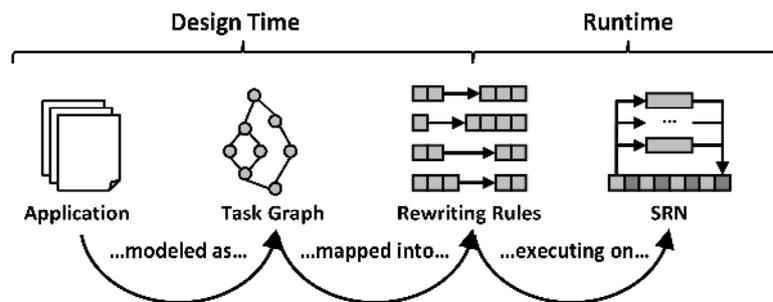


Figure 7. Design flow for stream rewriting.

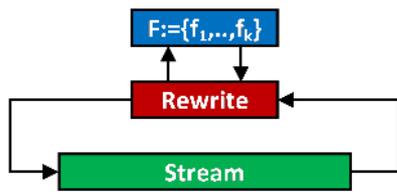


Figure 8. Stream Rewriting Machine.

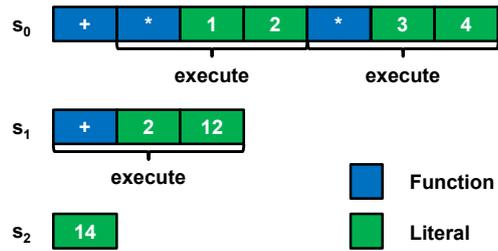


Figure 9. Minimalistic example of stream rewriting.

A stream rewriting machine (SRM) basically consists of a token stream, a rewriting function, and a set of rewriting rules  $F$  shown in Figure 8. While the stream contains the state of the system, the behavior and therefore the implementation of the program is described by the rewriting rules  $F$ . In this model, each rule detects a certain pattern in the stream and replaces the corresponding part with the results of a computation. The stream is circulating in the system, so that all rules are continuously checked and if appropriate, the associated function is evaluated and the results are inserted into the stream.

Although a formal model of the rewriting process is specified in Section 2, a minimalistic example is already presented in Figure 9. Here, the initial stream  $s_0$  encodes the equation  $1 \cdot 2 + 3 \cdot 4$  and consists of literal tokens in green and function tokens drawn in blue. In particular, a rewriting function can execute if it is followed by a sufficient number of arguments, so that their data dependencies have been resolved. In the first rewriting step, only the multiplications are ready and can be evaluated since there are two literals available. The addition has to wait until the step  $s_1 \rightarrow s_2$  to produce the final result 14.

Even in this small example, several important properties of stream rewriting can be studied. First, it describes all scheduling decisions as find-and-replace operations. Since data dependencies are encoded into the stream, they can be automatically resolved via pattern matching. On the other hand, the rewriting of independent patterns like the two multiplications can be performed in parallel because the corresponding sub-streams do not overlap. The result of these two different data paths is synchronized on the stream via the pattern matching of the  $+$  operator. As a consequence, the whole complexity of task management can be reduced into find-and-replace operations on the stream.

In the following sections, a several related models for parallel programming are presented and compared to stream rewriting.

Before dealing with the contribution of this thesis, related work is described first in the next section.

## 1.2 Related Work

There already exist alternative solutions to the problem of managing concurrency in dynamic systems. In this section, their particular strengths and drawbacks are described in comparison to stream rewriting.

### 1.2.1 Programming Models

Several different programming models for parallelism are presented in this sub-section.

#### 1.2.1.1 Data Parallelism

In addition to graphics processing, modern GPUs can be also utilized to accelerate general purpose and numeric computations. Especially data parallel problems can take advantage of hundreds and thousands of processor cores, which provide several TFLOPS of floating-point

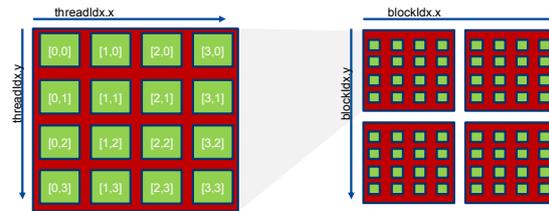


Figure 10. CUDA model of computation [19].

performance [20]. For this purpose, compute APIs like CUDA or OpenCL [21] permit to execute a kernel function on a parallel grid of threads (Figure 10). Each instance runs the same program on a different location in the grid that can be determined by a set of pre-defined variables. Hence, this model of computation is most useful for image processing or numerical simulations, which apply the same function to every element of an array.

In addition, threads are organized in blocks, which run on the same processor and can exchange data via shared memory. Actually, the kernel function is equivalent to the body of a data parallel loop, while the iteration is managed by the hardware scheduler of the GPU. Although two dimensional grids are most common, 1D and 3D domains are also possible. For instance, the following example shows the definition a CUDA kernel function, which adds the two one dimensional floating-point arrays *a* and *b* and stores the result in *c*:

```
__global__ void Add(float *a, float *b, float *c)
{
    uint id = blockIdx.x * blockDim.x + threadIdx.x;
    c[id] = a[id] + b[id]
}
```

First, the global index of the current thread is calculated using the build-in variables *blockIdx*, *blockDim* and *threadIdx*, which specify the coordinates of the current block, its dimensions and the index of the current thread within the block. The kernel is invoked from the C part of a CUDA program using the following syntax, which specifies the number of blocks and number of threads per block to execute:

```
Add<<<blockCount,blockDim>>>(a, b, c)
```

Hence, the total number of executed threads equals to  $blockCount \cdot blockDim$ .

The equivalent C code for this CUDA program is shown below and emulates this behavior using two nested loops:

```

void Add(float *a, float *b, float *c)
{
    for (int blockIdx = 0; blockIdx < blockCount; blockIdx++)
    {
        for (int threadIdx = 0; threadIdx < blockDim; threadIdx++)
        {
            int id = blockIdx * blockDim + threadIdx;
            c[id] = a[id] + b[id];
        }
    }
}

```

For two- and three-dimensional kernels, the corresponding steps are performed accordingly for the y and z coordinates. There also exist several domain-specific compute APIs, like DirectCompute [22], which is integrated in Direct3D, or the C++ language extension C++/AMP [23]. OpenMP [24] is an already established API for general purpose processors, which utilizes pragmas to mark parallel loops. In addition, recent implementations of OpenMP can also employ the graphics processor via CUDA [25].

All of these programming models offer a scalable solution for data parallel problems, which can be described as nested loops, but dependencies between subsequent iterations usually prevent a concurrent evaluation and are therefore not supported. Although it is possible in certain cases, to remove these dependencies by transforming loop indices [26] [27], the pure data parallel model is not suited to handle algorithms with dynamic workloads.

Otherwise, stream rewriting supports all types of data parallelism and can describe especially the dynamic nesting of loops, a variable number of iterations as well as recursive branch-and-bound algorithms. Further, this thesis proposes a technique, that handles loops with a large number of iterations (Section 2.3.4), and introduce specialized writing rules for data parallel loops (Section 2.4.1). As a result, stream rewriting is a true superset of these existing techniques and also permits the irregular and heterogeneous problems described in the next section.

### 1.2.1.2 Data Flow Networks

Data flow process networks offer an abstract model of concurrency, which reduces the complexity and development time of parallel applications [29]. A dataflow graph (DFG) consists of isolated concurrently running processes, which are

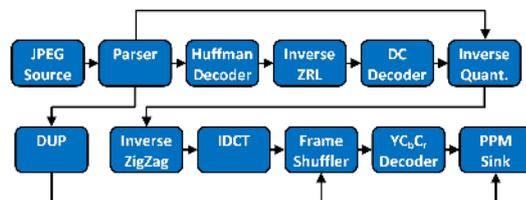


Figure 11. Data flow model of a JPEG Decoder [28].

also called actors and communicate only via dedicated point-to-point connections. There is no central mechanism for synchronization required if local groups of actors are self-scheduled, waiting for the required number of inputs to become available. In addition, large

data-flow graphs can be composed hierarchically by wrapping sub-graphs into actors and an abstract dataflow model can be re-targeted automatically for many different platforms. For instance, the data flow model of the JPEG Decoder (Figure 11) contains several different actors like the *Huffman Decoder*, the *DC Decoder* or the *IDCT*, which are connected via queues and work in parallel.

Hence, an initial implementation of this DFG can be derived by mapping actors directly to software threads or hardware modules [30] [31]. As an optimization, several connected blocks can be merged into more coarse grained actors via clustering to reduce the communication overhead and therefore improve latency and throughput [32].

For synchronous data-flow graphs [33], the rates of each actor are fixed and specified as part of the model. Consequently, an optimal schedule and the maximum size of the connecting buffers can be determined analytically [34]. However, not all problems can be modelled with constant rates and a fully dynamic self-scheduling can be expensive when implemented in software [35]. Thus, it becomes feasible to divide a graph into statically scheduled and dynamic parts to improve the performance [36]. However, in case of embedded systems, which are interacting with dynamically changing environments, often an adaptive mapping [37] is required to account for frequently varying workloads [38].

Stream rewriting provides an alternative approach for the scheduling of complex applications with different types of parallelism. Although its expressiveness is more limited to series-parallel graphs [39] without feed-back loops, this technique allows for varying data rates and also permits the recursive expansion of actors at runtime (Section 3.2.3).

### 1.2.1.3 *StreamIt*

The *StreamIt* programming language is based on Java and has been specially designed to describe data flow networks. Comparable to a class in Java, the basic building block of *StreamIt* is a filter module containing one input and one output channel.

More complex structures can be constructed by using special keywords to combine multiple filters into pipelines, arbiters, and feedback loops. Since all *StreamIt* actors are annotated with production and consumption rates, the compiler can generate an optimal static schedule [40]. In addition, the filters without an internal state can be replicated automatically, so that both data and pipeline parallelism are supported [41]. The abstract high-level description of *StreamIt* makes it possible to transform and optimize the data flow graph automatically for different target platforms [42].

*StreamIt* has been used as a high-level language for a hardware synthesis tool which is able to automatically derive an efficient FPGA implementation [43] [44]. Similarly, stream programs can be also translated into CUDA code that runs on the GPU [45]. In this case, the graph is restructured according to the architectural characteristics of CUDA to exploit as much data parallelism as possible. However, filters containing an internal state cannot be translated, so that these parts of the program remain on the CPU.

*StreamIt* has also been used to model a flexible graphics pipeline on top of a reconfigurable processor grid. Since the fixed data rates of the *StreamIt* filters are incompatible with the varying amount of data produced in the graphics pipeline, the language has been extended to support variable rates as well. For load balancing, actors are replicated and mapped

automatically to cores according to the expected workload but it is also possible to modify the configuration at runtime.

As a central difference, *Stream Rewriting* supports arbitrary recursion and function pointers, which cannot be described by *StreamIt* programs. *Stream Rewriting* also supports dynamic data flow graphs and can adapt itself to varying workloads at runtime without an external reconfiguration. On the other hand, *StreamIt* permits feed-back loops in the data flow graph, which cannot be modelled using *Stream Rewriting*.

### 1.2.2 System Level Parallelism

When designing the software and hardware architecture of many-core systems, a central problem is the scheduling and binding of task to processing units.

#### 1.2.2.1 Task Graphs

At the system level, the parallelism of many-core processors is exploited using multiple threads [46] [47], which communicate via shared memory or message passing. The application is often represented as a task graph [48] [49] and then mapped onto the target architecture [50]. The binding of tasks to processor cores can be either calculated as a preprocessing step [51] or dynamically at runtime [7]. In case of heterogeneous systems, the binding must consider several additional constraints regarding the capabilities of each core [52]. The static approach for task mapping is presented in [53] and uses both pipelining and data parallelism to maximize the utilization of processor cores in an MPSoC (multi-processor system-on-chip). Similarly, a technique for dynamic task mapping on a processor array has been presented in [54] but in contrast to stream rewriting, it requires a central scheduler, which works well for a small number of processors but might create a bottleneck in many-core systems.

While the static approach most certainly produces a better solution if all factors are known at design time, this thesis puts the focus especially on the dynamic case to handle a varying and large number of tasks with unpredictable work load at runtime. In fact, stream rewriting transfers the concept of out-of-order execution to the system level, which is implemented as *score-boarding* [55] or the *Tomasulo algorithm* for individual instructions [16]. However, in contrast to the out-of-order execution of data flow graphs, no global reservation station is required [56].

#### 1.2.2.2 Invasive Computing

*Invasive computing* [57] has been developed for heterogeneous many-core systems and eliminates the central scheduler of previous architectures [12] [58]. Instead, tasks are encapsulated as relocatable work items called *i-lets*. They can be pushed between neighboring cores for load-balancing and thus allow an application to dynamically invade a grid of processors. *Stream Rewriting* pursues a similar goal at a higher level of abstraction since it represents tasks as data tokens on a stream, which are decoupled from the implementation. In addition, stream rewriting also offers a mechanism to synchronize tasks via local pattern matching, while the invasive platform relies on atomic operations or mutexes for this purpose.

### 1.2.2.3 Work Stealing

Contrary, *work stealing* uses the opposite approach of invasive computing by letting processors explicitly fetch or steal tasks from their neighbors [59] [60]. An execution model, based on work stealing, and an extension to the C programming language (Cilk) are presented in [61] and [62]. Similar to the concept of this thesis, they can invoke functions on a new thread via the *spawn* keyword to implement dynamic and also recursive task graphs. For comparison, the *stream rewriting machine* (SRM) creates threads by generating a sequence of control and data tokens. Though, work stealing requires shared memory for synchronization, while stream rewriting is able to join threads using local pattern matching. Hence, the main difference is the choice of the stream as a common data structure.

## 1.2.3 Hardware Architectures

Concurrency and parallelism can be exploited at different abstraction levels in different ways. At the system level, the usage of multiple cores [63] permits to run several distinct tasks simultaneously [64] but also the instruction-set can be designed to utilize explicit data parallelism for SIMD and vector processing.

### 1.2.3.1 Many-Core Systems

In this work, several highly parallel hardware architectures are presented like the many-core system with 128 RISC processors in Section 4 or the graphics processor in Section 5. The *Larrabee* processor has been also designed as a more flexible GPU and it is based on up to 48 x86-compatible cores. The instruction-set of the *Larrabee* has been extended with vector arithmetic for floating-point calculations. Instead of dynamic scheduling, multiple hardware threads utilize the execution units. The chip is essentially an array of general purpose processors with cache coherence, virtual memory, and arbitrary data exchange via a ring-bus. Related designs like the *SCC* architecture [10] and the *tile processor* [11] also contain a grid of processor cores and unlike CUDA, they all support also heterogeneous tasks. However, in contrast to the stream rewriting architectures, there is no hardware support for pipeline parallelism and also the scheduling must be handled in software. As an advantage, stream rewriting provides a solution for distributing tasks at the system-level, which is simplistic enough to be implemented in hardware. In Section 4.3, the responsible hardware component, the stream rewriting network (SRN), is described in more detail.

### 1.2.3.2 Data Flow Machines

In addition, stream rewriting shares some similarities with early *data flow machines* like the *systolic array* and the *Xputer* [65]. They usually consist of a two dimensional array of processing elements with limited control logic for arithmetic operations. Similarly, the execution is also driven by data dependencies and therefore self-timed. The functionality of each core is simple and consists of a single instruction that is applied to a large amount of input data.

The *PipeRench* architecture [66] and its successor the *queue machine* [67] also process a token stream in hardware, but unlike stream rewriting, they do not support control tokens nor functions calls, so that their main use case is the acceleration of a basic block like the

inner body of a loop. Similarly, it is not possible to create new threads at runtime, limiting their applicability to almost data parallel problems and arithmetic operations always work on adjacent tokens so that usually, an expensive reordering is required.

In comparison, these architectures are well suited for data parallel problems but are less useful for general data flow graphs. Contrary, stream rewriting embeds control information into the stream and allows redefining the structure of data flow graph at runtime. As a result, complex control flow like recursion and expandable shader nodes can be modelled using the same rewriting technique based on pattern matching (Section 3.3).

### **1.2.3.3 Instruction Level Parallelism**

The amount of *instruction level parallelism* (ILP) in a program is highly variable and usually depends on the application. For example, algorithms from the fields of numerical computations, image and graphics processing already contain a large degree of intrinsic parallelism that can be directly utilized by the compiler. In particular, binding and scheduling can be performed at design time to identify concurrent operations and data dependencies [68] [69]. For instance, the instruction set of transport triggered architectures [70] consists exclusively of parallel move instructions between functional units, registers and memory. Similar, VLIW designs like [71] allow the compiler [72] to pack several independent instructions into a bundle, which is then executed in parallel [73]. EPIC architectures like the Itanium processor [74] extend this concept by reserving a special bit in the opcode to chain an arbitrary number of concurrent instructions [75].

A more backward compatible approach is the usage of SIMD (single instruction multiple data) extensions like SSE [76] or AVX [77], which introduce additional registers and opcodes for parallel vector computations [78] [79] [80]. Also, classic vector processors [81] distinguish between scalar and vector instructions but usually support much wider registers than the SIMD extensions.

However, many of these techniques must be explicitly considered by the programmer or depend extensively on the quality of static analysis in the compiler, which is usually limited by dynamic control flow like loops and branches [82]. Stream rewriting also resolves data dependencies dynamically at runtime, while the granularity of a rewriting rule spans from a single instruction to entire functions.

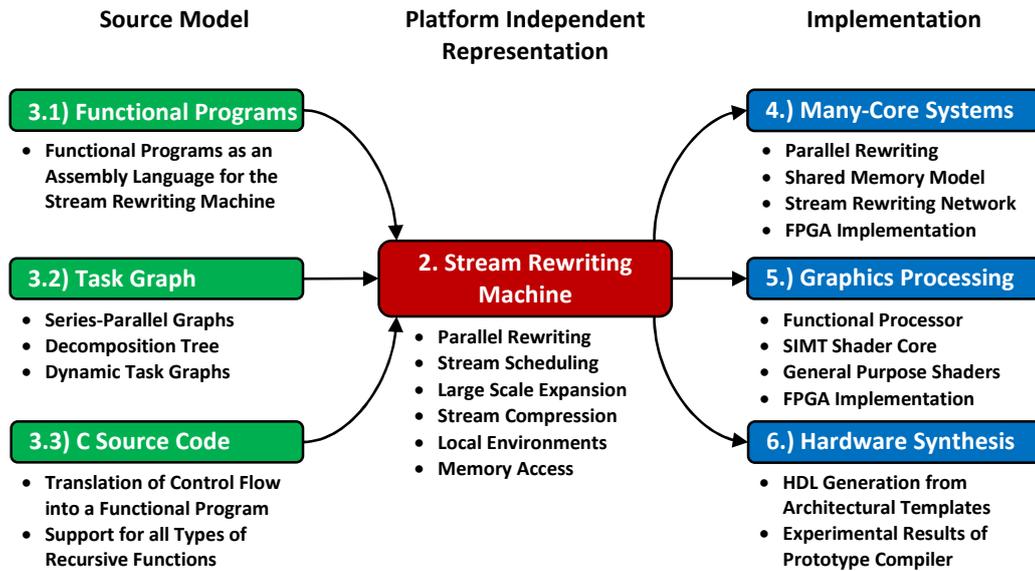


Figure 12. Source models and target platforms for stream rewriting, which are presented in this thesis.

### 1.3 Contributions

This thesis proposes *stream rewriting* as a novel model of computation for the specification, analysis, and implementation of parallel applications. The fundamental contribution is the observation that this model permits to specify dynamic scheduling and binding as rewriting rules to handle a large amount of irregular tasks without a central scheduler. The structure of this thesis and its contributions are presented in Figure 12 and distinguishes between the core model of stream rewriting, the source models, and the implementations.

Based on an abstract model, which is named the *stream rewriting machine* (SRM), the most important challenges like parallel rewriting, scheduling, synchronization, and memory access are discussed from a theoretical point of view in Section 2. In this section, the following contributions are most significant:

- **Stream rewriting is based on local operations**  
 Rewriting operations modify locally constrained and non-overlapping intervals of the stream and can be therefore performed on different regions in parallel. Also the synchronization of concurrent threads is performed via local pattern matching on the stream and does not require shared memory or atomic operations.
- **Stream rewriting is driven by data dependencies**  
 The execution order of tasks is defined implicitly by data dependencies, so that an implementation can choose an optimal schedule at runtime.
- **Concurrent threads are synchronized by dynamic binding**  
 Access to shared resources is serialized by dynamically binding interfering tasks to the same processor and permits complex atomic read-modify-write operations.

- **Stack-based scheduling enables deep recursion**  
The stack-based scheduling automatically balances between a depth-first and a breadth-first traversal of the call tree. This approach avoids an exponential growth of tokens during recursive traversal but still maintains a large degree of concurrency.
- **Local environments embed shared data into the stream**  
Local environments redefine the value of global variables for a restricted sub-stream and therefore provide an efficient mechanism for the precise distribution of shared data without side-effects.

Further, in this thesis three *source models* are analyzed and can be translated into rewriting rules for the stream rewriting machine (Section 3):

1. **Functional Language:** In combination with a grammar and semantics, the stream rewriting machine is actually a hardware interpreter for a minimalistic functional language. Due to the absence of hidden side-effects, the purely functional model is well-suited for modeling complex tasks with interdependences and concurrency.
2. **Task Graphs:** Since task graphs provide an efficient model of computation for specification, analysis, and implementation of concurrent applications, also a novel approach for scheduling the class of series-parallel task graphs is presented. For this purpose, the topology of the graph and the state of the tasks are encoded as a stream of tokens, which is iteratively rewritten at different positions in parallel. Hence, this approach is most useful for compute-intensive applications with varying workload.
3. **C Source Code:** In addition to functional programs, also the translation of C-like language into a stream rewriting machine is described. The presented compilation technique enables the usage of stream rewriting as an intermediate format for the high-level synthesis of hardware modules (Section 6).

In addition, the abstract concept of stream rewriting has been implemented and evaluated on three different target platforms:

1. **Many-Core System:** The utilization of stream rewriting is evaluated for dynamic task binding and scheduling in embedded many- and multi-core systems. In particular, several different FPGA implementations with up to 128 general purpose cores have been exploited. In this case, the rewriting is either performed entirely in software or as a combination of software and custom hardware. In contrast to the classic scheduling of periodic real-time tasks, stream rewriting is more appropriate for handling a large number of dynamic work-items (Section 4).

2. **Graphics Processor:** Since graphics processing usually involves a huge number of threads as well as data and pipeline parallelism, it represents an ideal test case for stream rewriting. Despite the computational power of modern graphics processing units (GPU), a main limitation of these architectures is often the lack of efficient on-chip communication between different shader cores. In this context, stream rewriting enables an efficient communication infrastructure for the creation of highly optimized and application-specific rendering pipelines. As a result, three different architectures could be evaluated using an FPGA. Especially for graphics processing, this thesis presents the following contributions (Section 5):

- **Application-Specific Rendering Pipelines**

The proposed graphics processor supports an arbitrary amount of dynamic and recursive shader stages as well as complex data flow and light-weight synchronization within the rendering pipeline.

- **Unified Model of Computation for Graphics**

A graphics application must be usually split into two parts for GPU and CPU, which are developed using separate languages and optimized according to different rules. The CPU part is either executed sequentially or moderate multi-threaded, while the GPU part of the program is adapted to the more restricted but highly parallel model of the graphics processor. Although the graphics API connects these two totally different environments, the binary decision between CPU and GPU leads to an artificial break and introduces communication overhead. For this purpose, stream rewriting offers a continuous transition between task, pipeline and data parallelism as well as support for individual threads.

- **Complete Prototype System**

Most important, the concept of stream rewriting for graphics processing has been evaluated as part of a complete test system consisting of several applications, an OpenGL driver, a kernel mode driver, and the FPGA prototype connected via PCIe. As a result, the measurements include all costs that would also occur for an ASIC, so that the experiments provide valuable insights for future GPU architectures.

- **OpenGL Extension for Stream Rewriting**

This thesis proposes a novel OpenGL extension (*EXT\_stream\_rewriting*), which integrates stream rewriting into the existing programming model and adds the stream rewriting shader as a new general purpose shader stage.

- **Advanced Rendering Techniques**

Several advanced rendering techniques, like *K-buffering*, *path-tracing* and the *recursive generation of procedural geometry* are implemented on the stream rewriting processor but require significant effort to run efficiently on current GPUs.

3. **Hardware Synthesis:** By generating specialized hardware modules for a particular stream rewriting machine, recursion and dynamic scheduling also become available for high-level synthesis. The compiler is based on the concepts of Section 3.3 and translates a C-program into rewriting rules, which are eventually used to derive a HDL implementation. The tool chain is evaluated using several generic test cases and a more complex ray-tracing example (Section 6).

The paper is concluded by a final review (Section 7) of the stream rewriting technique and a summary of the achieved improvements. In addition, also a list of currently unsolved problems in the concept and further technical limitations are presented in this section.

## 1.4 Publications

Many of the results presented in this thesis have been published previously. The following list contains the most significant publications for stream rewriting and their relation to individual sections of this paper:

- **Hardware synthesis of recursive functions through partial stream rewriting [83]**  
This paper describes a high-level compiler that generates hardware for specialized stream rewriting machines. In contrast to existing tools, the system supports all types of direct and indirect function calls as well as recursion. The SRM in this paper corresponds to the basic model and the results of the experiments are presented in Section 4. Several smaller functions and a more complex ray-tracing example have been synthesized and evaluated. The author of this paper has developed the main concept and the implementation, which are analyzed in [83].
- **Dynamic task mapping onto multi-core architectures through stream rewriting [84]**  
This paper takes the opposite direction and puts the focus on generality. Consequently, *stream rewriting* has been implemented entirely in software on a many-core system and the token stream is stored in shared memory. A comparison to [83] shows an improved scalability and conceptually. The author of this thesis responsible for the concept of mapping task graphs into rewriting rules (Section 3.1) and the experiments, which presented in Section 4.2.
- **A novel graphics processor architecture based on partial stream rewriting [85]**  
In this paper, the author of this thesis presents a graphics processor based on functional programming, which utilizes the concept of stream rewriting for task management (Section 3.1). In particular, this architecture introduces the stack-based scheduling (Section 2.3.3), instancing and environments (Section 2.4). The author of this thesis has developed both the concept and the implementation of this paper.

- **A Programmable Graphics Processor based on Partial Stream Rewriting [86]**  
The graphics processor has been redesigned for improved performances. Similar to existing GPUs, this version is based on SIMT cores and supports a wide data stream, the parallel dispatch of data parallel threads and has been evaluated using a larger set of test cases. In addition, this design establishes the correct ordering of write operations by placing them on the stream (Section 5.4). The author of this thesis has developed the concept, the hardware design of the graphics processor, the required software, and is also responsible for the presented experiments.
- **System Level Synthesis of Many-Core Architectures using Parallel Stream Rewriting**  
This paper extends the concept of stream rewriting for general purpose many-core architectures with up to 128 processors on an FPGA prototype. In order to provide a sufficient band-width, the hierarchical stream rewriting network employs parallel decoding and dispatch of the token stream [87]. The author of this paper has developed the main concept and the implementation.

The following list contains the relevant publications of the author:

- [88] L. Middendorf und C. Haubelt, „Scheduling of Recursive and Dynamic Data-Flow Graphs using Stream Rewriting“ in *Proceedings of Special Edition on Data-flow Programming Models and Machines (MPP'14)*, Paris, France, October 2014.
- [87] L. Middendorf und C. Haubelt, „System Level Synthesis of Many-Core Architectures using Parallel Stream Rewriting“ in *Proceedings of the 2014 Electronic System Level Synthesis Conference (ESLsyn)*, , San Francisco, CA , 2014.
- [89] Christian Haubelt, Florian Ludwig, Lars Middendorf, Christian Zebelein, „Using Stream Rewriting for Mapping and Scheduling Data Flow Graphs onto Many-Core Architectures in *Proceeding of the Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove CA, USA, November 2013
- [85] L. Middendorf und C. Haubelt, „A novel graphics processor architecture based on partial stream rewriting,“ in *Design and Architectures for Signal and Image Processing (DASIP), 2013 Conference on*, Cagliari, 2013.
- [86] L. Middendorf und C. Haubelt, „A Programmable Graphics Processor based on Partial Stream Rewriting,“ *Computer Graphics Forum*, Bd. 32, Nr. 7, pp. 325-334, 2013.
- [84] L. Middendorf, C. Zebelein und C. Haubelt, „Dynamic Task Mapping onto Multi-Core Architectures through Stream Rewriting,“ in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), 2013 International Conference on*, Agios Konstantinos, 2013.
- [83] L. Middendorf, C. Bobda und C. Haubelt, „Hardware synthesis of recursive functions through partial stream rewriting,“ in *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, San Francisco, CA , 2012.

## 2 Stream Rewriting

Stream rewriting is a novel method for dynamic scheduling and binding. In this section, a formal model of computation for stream rewriting (Section 2.1) is defined, which is called the *stream rewriting machine* (SRM). Section 2.2 specifies the *parallel rewriting* of the stream by *partitioning* (Section 2.2.3) and *binding* (Section 2.2.4) different segments to concurrently working processing units. In particular, the *stream scheduling* (Section 2.3) describes the selection of tokens, which are considered for rewriting at a particular instant. *Stream compression* (Section 2.4) is proposed as a technique to reduce redundant information in order to decrease the size of the stream. On top of the pure functional SRM, the usage of *memory access* and *side effects* is specified in Section 2.5.

### 2.1 Stream Rewriting Machine

The *stream rewriting machine* (SRM) is an abstract model of computation.

#### 2.1.1 Definitions

An SRM can be defined as:

**Definition D1:** A stream rewriting machine (SRM) is a 3-tupel  $(s, F, rewrite)$  consisting of the following elements (Figure 8, Page 6):

1. A stream  $s$  of tokens, which stores the state of the system,
2. a set of functions  $F$  representing the behavior of the system, and
3. a function  $s_{n+1} := rewrite(s_n)$  that maps a stream  $s_n$  to a subsequent stream  $s_{n+1}$ .

**1. Stream:** Formally, a stream  $s \in \Sigma^*$  is defined as a word from an alphabet  $\Sigma$  of tokens. In case of a finite stream, the elements of  $s := \langle x_1, \dots, x_n \rangle$  can be accessed as  $x_i$  and the length of the stream is denoted as  $|s| := n$ . In addition, two streams  $s := \langle x_1, \dots, x_n \rangle$  and  $t := \langle y_1, \dots, y_m \rangle$  are concatenated via the associative operator  $+$ :

$$s + t := \langle x_1, \dots, x_n, y_1, \dots, y_m \rangle \text{ with } |s + t| := |s| + |t| \quad (1)$$

Further, the empty stream  $e := \langle \rangle$  has the length  $|e| = 0$  and acts as a neutral element for the concatenation, so that the triple  $(\Sigma^*, +, e)$  specifies a monoid.

**2. Functions:** The program is given by a finite set of functions  $F := \{f_1, \dots, f_k\}$ , each of them mapping  $n_i$  input tokens to  $m_i$  output tokens with  $f_i: \Sigma^{n_i} \rightarrow \Sigma^{m_i}$ .

**3. Rewriting:** The rewriting function  $rewrite: \Sigma^* \rightarrow \Sigma^*$  maps the current state of the system to a successor state by invoking executable tasks from  $F$  and replacing at least one token in the stream if possible. Hence, an infinite sequence of streams or system states is given by:

$$s_{n+1} := rewrite(s_n) \quad (2)$$

This iterative process has been already illustrated by Figure 8 on page 6. In fact, each iteration in the SRM derives a new stream  $s_{n+1}$  from the previous one  $s_n$ .

If the stream does not change between two iterations, so that  $s_{n+1} = s_n$ , a fixed-point has been reached and there will be no further modifications. Hence, the final result is defined as the limiting value of  $s_n$ :

$$result(s_0) := \lim_{n \rightarrow \infty} s_n \quad (3)$$

### 2.1.2 Stream Operations

In this section, additional stream operations are defined, which will be required to specify the behavior and several extensions of the basic SRM in the following sections.

Two streams  $s := \langle x_1, \dots, x_n \rangle$  and  $s' := \langle x'_1, \dots, x'_m \rangle$  are identical if and only if they have the same length and their tokens are equal:

$$s = s' \Leftrightarrow (|s| = |s'|) \wedge (\forall i \in [1, |s|]: x_i = x'_i) \quad (4)$$

Similar to sets, there is a test if a token  $x \in \Sigma$  is contained in a finite stream  $s := \langle x_1, \dots, x_n \rangle$ :

$$x \in s \Leftrightarrow \exists i \in [1, n]: x_i = x \quad (5)$$

Likewise, a sub-stream  $s' \subseteq s$  corresponds to a compact range in the stream  $s$ :

$$\begin{aligned} s' \subseteq s &\Leftrightarrow \exists a, b \in \Sigma^*: s = a + s' + b \\ s' \subset s &\Leftrightarrow (s' \subseteq s) \wedge (s' \neq s) \end{aligned} \quad (6)$$

For instance, a sub-stream  $s[i, j]$  with  $1 \leq i \leq j \leq n$  can be constructed by applying the interval operator  $[]$  on a finite stream  $s := \langle x_1, \dots, x_n \rangle$ :

$$s[i, j] := \langle x_i, \dots, x_j \rangle \subseteq s \quad (7)$$

The result of multiple subsequent rewriting operations is recursively specified as follows:

$$rewrite^n(s) := \begin{cases} s & \text{if } n = 0 \\ rewrite(rewrite^{n-1}(s)) & \text{else} \end{cases} \quad (8)$$

Hence, the set of stream derived from  $s$  is specified as  $s^*$ .

$$s^* := \{rewrite^i(s) : i \in \mathbb{N}_0\} \quad (9)$$

For example, the maximum length of any stream and therefore an upper bound for the required storage is given by  $\sup_{s' \in s^*} |s'|$  if existent. Based on  $s^*$ , the relation  $(\Sigma^+, \preceq)$  determines if a stream  $s$  can be rewritten into  $s'$ :

$$s \preceq s' \Leftrightarrow s' \in s^* \quad (10)$$

According to its definition, the relation is both reflexive and transitive and thus forms a pre-order. For example, the final result is always more derived than the initial stream  $result(s_0) \preceq s_0$  if existent. However,  $(\Sigma^+, \preceq)$  does not conform to a partial order since the rewriting process might run in a cyclic fixed point according to the condition  $(s' = rewrite(s)) \wedge (s = rewrite(s'))$  but  $s' \neq s$ . In this case, it follows that  $s \preceq s'$  and  $s' \preceq s$  for two distinct elements  $s' \neq s$ , which is insufficient for a partial order.

However, a canonical equivalence relation  $(\Sigma^+, \equiv)$  between streams can be defined as:

$$s \equiv s' \Leftrightarrow (s \preceq s') \wedge (s' \preceq s) \quad (11)$$

It follows immediately from the inherited definition of  $(\Sigma^+, \equiv)$  that this relation is reflexive, symmetric, and transitive. Therefore, two streams  $s$  and  $s'$  are equivalent if they are identical or located on a cyclic fixed point, so that both can be derived from each other. Thus, a strict order  $(\Sigma^+, <)$ , comparing the equivalence classes of two streams, is given by:

$$s < s' \Leftrightarrow (s \preceq s') \wedge (s \not\equiv s') \quad (12)$$

As a consequence, the condition  $rewrite(s) < s$  hold true for each valid rewriting step, which does not end in an infinite loop. In addition, the final stream  $result(s)$  is the infimum of all generated streams:

$$result(s) = \inf_{<}(s^*) \quad (13)$$

### 2.1.3 Stream Grammar

After the introduction of the basic data structures, the function  $rewrite$  is now specified in more detail. For this purpose, a minimal alphabet  $\Sigma$  of tokens and a minimal language  $S$  are specified to describe the set of all valid streams. More complex stream rewriting machines with application-specific extensions will be presented in subsequent sections. In particular the stream  $s \in S \subseteq \Sigma^*$  is a word from a language  $S$  with alphabet  $\Sigma := \{CALL\} \cup \mathbb{Z}$ , which is defined by the following grammar in *Extended Backus-Naur Form*:

<b>Definition D2:</b>	$\begin{aligned} \text{stream} &:= \{expr\} \\ \text{expr} &:= \text{const} \mid \text{call} \\ \text{const} &:= \langle \text{literal value } z \in \mathbb{Z} \rangle \\ \text{call} &:= \{expr\} i \text{'CALL'} \end{aligned}$
-----------------------	--

Hence, each token in the stream is either a special CALL symbol or a literal value from  $\mathbb{Z}$ . Likewise, each CALL expression represents the invocation of a function  $f_i: \Sigma^{n_i} \rightarrow \Sigma^{m_i} \in F := \{f_1, \dots, f_k\}$ ,  $n_i, m_i \in \mathbb{N}_0$ , which is identified by the literal  $i \in [1, k]$  and an optional list of arguments. The function  $f_i$  is invoked if there are at least  $n_i$  preceding arguments available on the stream. As a result, the tokens for the invocation are replaced by the results of  $f_i$ .

<b>Definition D3:</b>	<p>A sequence of tokens <math>\langle \dots, x_1, \dots, x_{n_i}, i, CALL, \dots \rangle</math> is called <b>executable</b> if the CALL token referring to <math>f_i: \Sigma^{n_i} \rightarrow \Sigma^{m_i}</math> is preceded by at least <math>n_i</math> literal values, so that <math>x_1, \dots, x_{n_i} \in \mathbb{Z}</math> and <math>1 \leq i \leq k</math>.</p>
-----------------------	---

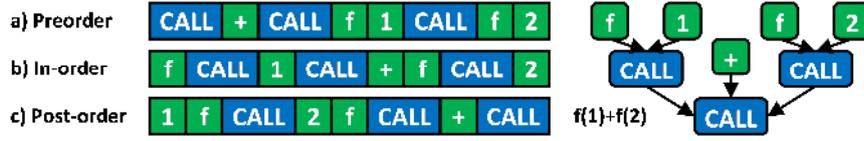


Figure 13. Expression  $f(1) + f(2)$  encoded using alternative stream formats. Instead of the function identifier  $i$ , the function symbols ' $f$ ' and ' $+$ ' are used to improve readability.

As a result, the rewriting process replaces the pattern

$$\langle \dots, x_1, \dots, x_{n_i}, i, CALL, \dots \rangle \text{ with } x_1, \dots, x_{n_i} \in \mathbb{Z} \text{ and } 1 \leq i \leq k \quad (14)$$

by the following sub-stream:

$$\langle \dots, r_1, \dots, r_{m_i}, \dots \rangle \text{ with } (r_1, \dots, r_{m_i}) := f_i(x_1, \dots, x_{n_i}) \quad (15)$$

Essentially, the stream stores a call tree in post-order format with its arguments placed in front of the *CALL* token. Alternatively, a preorder format can be defined by the specifying the *CALL* expression using the following grammar rule:

$$call := 'CALL' i \{expr\} \quad (16)$$

Although preorder and post-order formats are equivalent, each of them offers different advantages. For example, the preorder format requires less effort for decoding since the function  $f_i$  and therefore also the number of expected arguments is known as soon as the identifier  $i$  is read. Important for the post-order format: Both the innermost and also the first invocations are located in the beginning of the stream, which is most useful for the stack-based scheduling (Section 2.3.3). Contrary, the preorder layout contains the executable tasks at the end. In this thesis, both formats are utilized in different sections.

As a third alternative, an in-order storage format is also possible but would require parentheses to associate arguments and *CALL* tokens. It is therefore not discussed further.

For comparison, the expression  $f(1) + f(2)$  is encoded using these three formats and Figure 13 shows the resulting streams as well as the associated expression tree. Here, the set of functions is given by  $F := \{f, +\}$  but for the sake of clarity, the identifier  $i$ , originally defined as an integer  $i \in \{1, 2\}$ , is symbolically described by the name of the function.

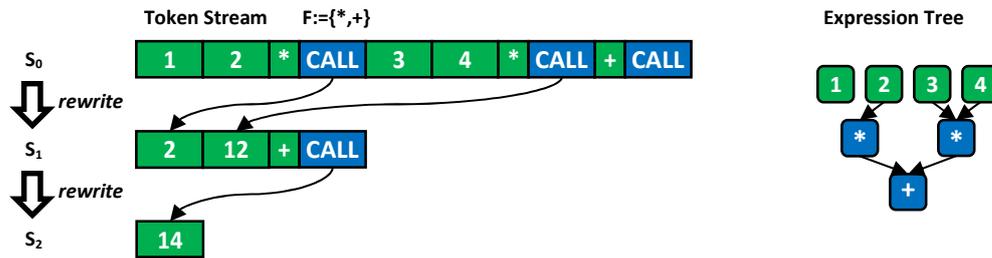


Figure 14. Expression tree evaluated using stream rewriting.

### 2.1.4 Examples

A minimalistic example of stream rewriting, which uses the post-order grammar, is shown in Figure 14. There are two functions defined by  $F := \{*, +\}$  to implement integer multiplication as well as addition. Initially, the first stream  $s_0$  corresponds to the expression  $1 \cdot 2 + 3 \cdot 4$  and the tree on the right. Indicated by the arrows, both multiplications are ready to execute because each *CALL* token and the identifier  $i = 1$  are preceded by two literals. As a result, the corresponding parts of the stream are replaced by  $1 \cdot 2 = 2$  and  $3 \cdot 4 = 12$  to produce the next iteration  $s_1$ .

Most notably, both instances can be executed concurrently because they are working on separate parts of the stream, which is important to process a large amount of tasks in parallel. Also, data dependencies are handled automatically by the pattern matching. Hence, the addition has to wait for the intermediate results of the two multiplication stages, which becomes available in step  $s_1$ . Afterwards, the addition can execute as well to produce the final result in  $s_2$ .

Instead of a single literal, a function can also generate a sub-stream of literal and *CALL* tokens to describe pending invocations. For example, a recursive function might be dynamically expanded into a sub-tree of *CALL* expressions to perform its computation. This method is called continuation-passing style [90], known from functional programming, and maps natively to the execution model of stream rewriting. The example in Figure 15 calculates the forth Fibonacci number function via recursive stream rewriting. In iteration  $s_0$ , the initial call  $f(3)$  is placed on the stream and matching patterns are successive replaced until the final result 2 becomes ready in  $s_5$ .

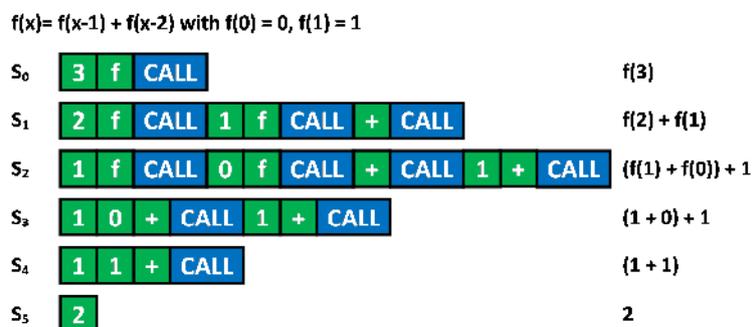


Figure 15. Rewriting of the Fibonacci function  $f(x) = f(x - 1) + f(x - 2)$

In this example, the stream  $s_5 := \langle 2 \rangle$  consists of a single literal value and is therefore a fixed-point, so that  $rewrite(s_5) = s_5$ . In general, streams consisting exclusively of literal values are cannot be rewritten further and thus correspond to final results.

An important observation is that the stream can both grow and shrink during the rewriting process. Hence, the recursive expansion might lead to an exponential increase of the stream size, which can be reduced by the stack-based scheduling (Section 2.3.2) into a linear growth. However, not all programs terminate and some applications like continuously running hardware systems might also never reach a fixed-point by design. In this case, there exists no element  $n$  with  $s_{n+1} = s_n$ , the sequence does not converge, and therefore the limiting value does not exist. Otherwise, the final stream usually contains the result of the computation.

## 2.2 Parallel Rewriting

Based on the abstract model of stream rewriting, the advantages and problems of a parallel rewriting algorithm are analyzed. The SRM has been extended into a *parallel stream rewriting machine*, shown in Figure 16. Instead of a single block, it contains multiple parallel rewriting units, which communicate via the *stream rewriting network* (SRN). In particular, the SRN introduces

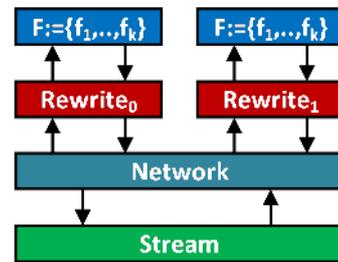


Figure 16. Parallel Stream Rewriting Machine

the concept of *partitioning the stream into segments* (Section 2.2.3), which might be evaluated concurrently. Further, the SRN is responsible for assigning these fragments to rewriting units, which is called *binding* (Section 2.2.4). Before these terms are defined in detail, several conceptual issues with parallel stream rewriting are discussed in Sections 2.2.1 and 2.2.2.

### 2.2.1 Concurrency of Stream Rewriting

Stream rewriting is well-suited for a parallel implementation due to the following properties, which are derived from the formal specification (**Definition D1 on page 18**).

#### 1.) No explicit execution order.

The execution order of rewriting rules is defined implicitly by data dependencies, so that multiple rules without interdependencies can be evaluated in parallel. Thus, an implementation can choose an optimal schedule at runtime in order to maximize utilization of available processing units.

#### 2.) Executable rewriting patterns do not overlap.

According to the stream grammar, the intervals of executable rules are disjoint, which can be shown as follows. Assume that  $u := \langle x_i, \dots, x_j \rangle$  and  $v := \langle x_a, \dots, x_b \rangle$  are two overlapping sub-streams, so that  $s = \langle \dots, x_i, \dots, x_a, \dots, x_j, \dots, x_b, \dots \rangle$  with  $i < a < j < b$ . If both  $u$  and  $v$  are executable, it immediately follows from (**Definition D3 on page 20**) that  $x_i, \dots, x_{j-1}, x_a, \dots, x_{b-1} \in \mathbb{Z}$  and  $x_j, x_b = CALL$ . For this condition to become true,  $x_j$  must

be a *CALL* token within the interval  $\langle x_{a+1}, \dots, x_{b-1} \rangle$  of literals, so that  $v$  cannot be executable, which leads to a contradiction. As a consequence, the pattern matching can be performed in parallel on disjoint sub-streams without synchronization. Hence, if a rewriting pattern is found in one sub-stream, there will be no other executable pattern in the same range.

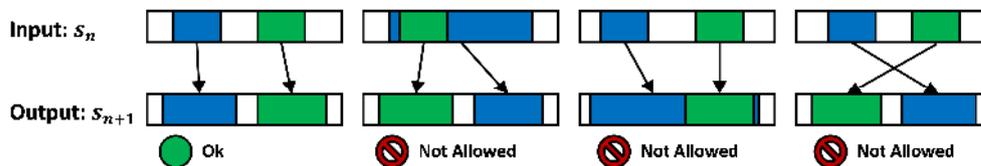


Figure 17. Executable tasks never overlap on the stream.

### 3.) Modifications of the stream are local and disjoint.

According to the definition of the rewriting operation, the input pattern is replaced by the outputs of the associated function  $f_i$  but the rest of the stream remains unmodified. Hence, the execution of a rewriting rule affects only the corresponding input tokens. However, since the input intervals of executable rules are disjoint, two concurrent rewriting rules never modify the same part of the stream in parallel. As a consequence, the rewriting process can be partitioned on multiple cores, while the pattern detection, the evaluation, and the generation of output tokens can be performed atomically without synchronization. Figure 17 illustrates the parallel rewriting of two executable rules marked in blue and green. Only the first case is possible according to the specification. Neither input nor output segments can overlap and in addition, also the relative order of rewritten fragments must be ensured.

### 4.) Relaxed pattern matching is acceptable.

The function *rewrite* requires that at least one matching rule is replaced to guarantee monotonicity of the sequence  $(s_i)$ . Hence, implementations with limited computing resources are free to choose an optimal but non-empty sub-set of rules that are actually executed per iteration. The stack-based scheduling (Section 2.3.2) evaluates only rules, which are located within the first  $b$  tokens of the stream, in order to reduce the total memory requirement. In particular, implementations can stop the pattern matching at any time if their computational resources are fully utilized and invoking more tasks provides little benefit.

### 5.) The stream can be partitioned without exact knowledge of its contents.

Based on the relaxed pattern matching, the stream can be partitioned into arbitrary segments for parallel rewriting on multiple cores. Obviously, *CALL* expressions will be probably split at the border of the segments, but their rewriting can be postponed until there are no other executable tasks available. Hence, a scheduling algorithm can be implemented without accessing the token stream, which might be especially useful for architectures with distributed or non-uniform memory architectures.

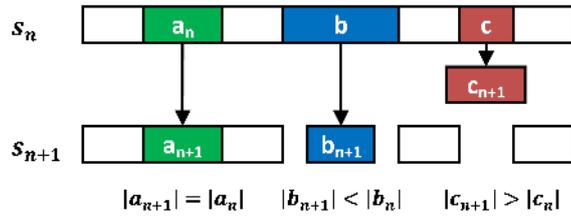


Figure 18. Reassembly of rewritten fragments

## 2.2.2 Implementation Challenges

In addition to the benefits of the parallel stream rewriting, a concurrent implementation has to deal with the following problems:

### 1.) Partitioning of the stream

While the stream can be split at any position without affecting correctness of the final result, not every scheme might lead to an efficient load-balancing. In particular, a coarse-grained partitioning into large blocks reduces the costs of stream management and synchronization since each core can work independently for a longer time. However, a fine-grained splitting of the stream into individual tasks leads to a better load-balancing at the expense of higher communication costs. Several approaches for stream partitioning are compared in Section 2.2.3.

### 2.) Stream Distribution and Scalability

Especially for many-core systems, the distribution of tokens to the processing units represents a possible performance bottleneck because the stream is specified as a central storage with global read and writes access from all cores. In particular, it has to be ensured that the synchronization costs are kept low and the bandwidth is sufficient high enough to keep all cores utilized. While an initial implementation might place the stream into a globally shared memory, alternative hardware architectures for stream management and distributed storage of the stream should be also considered.

### 3.) Efficient reassembly of rewritten fragments

Although both pattern matching and the rewriting of sub-streams can be performed in parallel, the resulting fragments must be eventually reassembled into a single token stream to produce either an intermediate result or the final limiting value of the stream. However, the size of these fragments is unpredictable because it depends on the rewriting process itself. Figure 18 shows the rewriting of the three fragments  $a_n, b_n, c_n$  into  $a_{n+1}, b_{n+1}$  and  $c_{n+1}$ . In case of  $a_n$ , the result  $a_{n+1}$  has the same length and can be stored at the original position. Contrary, fragment  $b_n$  shrinks and fragment  $c_n$  is expanded, so that both do not fit into the original stream.

As a result, the storage format of the stream optimally supports fast insertion and removal of tokens at arbitrary positions to allow an in-place rewriting of the stream. Otherwise, parts of the stream must be copied to fill gaps or to provide additional space. As an alternative solution, the size of the rewritten fragments can be also estimated or constrained. Different approaches for stream management are discussed in Section 2.2.3.

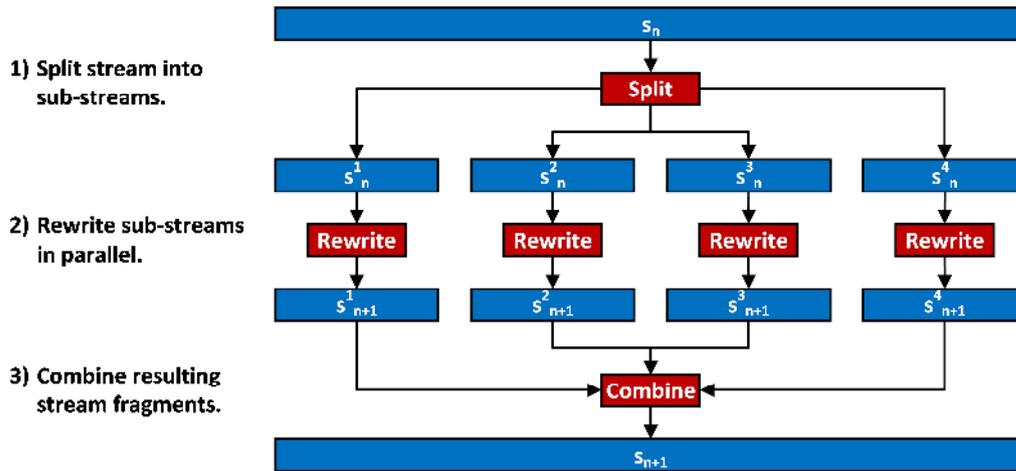


Figure 19. Partitioning and parallel rewriting of the stream.

### 2.2.3 Partitioning

In order to perform a parallel rewriting of a stream  $s_n$ , it must be first partitioned into sub-streams  $s_n^1, \dots, s_n^k$ , which can be evaluated concurrently. Due to the pattern matching, the stream can be subdivided almost arbitrary into non-overlapping intervals. In particular, the partitioning can be performed without looking at the content of the stream. However, not all fragmentations are efficient and feasible. In the worst case, no segment  $s^j$  contains an executable CALL expression (**Definition D3 on page 20**). In contrast to the concept of binding in the next section, no assumption concerning the number and usage of rewriting units are made at this point. Thus, a generic algorithm for the parallel rewriting of a stream  $s_n$  consists of the following three steps, which are also illustrated in Figure 19:

#### 1. Split the given stream $s_n$ into $k$ segments.

First, the stream is partitioned into  $k$  sub-streams  $s_n^1, \dots, s_n^k$  with  $s_n := \sum_{j=1}^k s_n^j$ . The number and size of these segments depends on the exact implementation and multiple different approaches are compared in the next sections. For instance, a system with  $p$  processing units would most likely split the stream into at least  $k \geq p$  parts to ensure complete utilization. Formally, this process is described by a function

$$split: \Sigma^* \times \mathbb{N} \rightarrow \mathbb{N}, \quad (17)$$

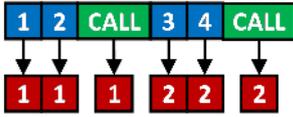
which attaches a segment number to each token of the stream  $s := \langle x_1, \dots, x_n \rangle$ . Hence, each segment  $s^j$  can be described as the sub-stream of tokens which are tagged with number  $j$ .

$$s^j := \sum_{i=1}^{|s|} \begin{cases} \langle x_i \rangle & \text{if } split(s, i) = j \\ \langle \rangle & \text{else} \end{cases} \quad (18)$$

Formally, the assignment for a given stream  $s$  is specified as  $split_s(i) := split(s, i)$ . For instance, in the example, the stream  $s_1$  is partitioned into two segments  $s_1^1$  and  $s_1^2$ :

$$s_1 := \langle 1, 2, CALL, 3, 4, CALL \rangle$$

$$split_{s_1} := \{(1,1), (2,1), (3,1), (4,2), (5,2), (6,2)\}$$

$$s_1^1 := \langle 1, 2, CALL \rangle, s_1^2 := \langle 3, 4, CALL \rangle$$


In addition, the function  $split_s$  must be monotonic, so that:

$$\forall a, b \in \mathbb{N}: a \leq b \Rightarrow split_s(a) \leq split_s(b) \quad (19)$$

As a result, each non-empty segment  $s^j$  starts at the token with index  $p_j := \min\{i: split_s(i) = j\}$  and ends at the token with index  $q_j := \max\{i: split_s(i) = j\}$ . Due to the monotonicity of  $split_s$ , each segment  $s^j$  corresponds to a compact interval of the sub-stream  $s^j := \langle x_{p_j}, \dots, x_{q_j} \rangle$ . Hence, two neighboring tokens  $x_u$  and  $x_{u+1}$  in a sub-stream with  $s^j := \langle \dots, x_u, x_{u+1}, \dots \rangle$ , are also adjacent in  $s$ . As a result, an executable CALL pattern in the sub-stream  $s^j$  is also a valid CALL pattern in  $s$  and can be therefore rewritten. Important to note, the opposite argument is not true.

Figure 20 shows the tradeoffs of several partitioning schemes. In the case (a), the entire stream is assigned to a single segment. The uniform segmentation (b) does not require access to the stream but might cut matching CALL expressions. Contrary, the more precise heuristic in (c) starts new segment only after a potentially matched rewriting rule.

## 2. Parallel Rewriting

In a second step, each segment  $s_n^j$  is rewritten individually into  $s_{n+1}^j$ :

$$s_{n+1}^j := rewrite(s_n^j) \quad (20)$$

## 3. Reassembly of the stream fragments

Hence, the parallel rewriting function can be specified as:

$$partition(s) := \sum_{j=1}^k rewrite(s^j) \quad (21)$$

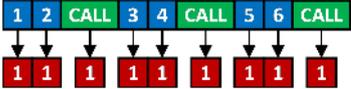
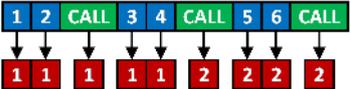
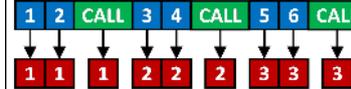
a) Single Segment	b) Uniform Segmentation (k=2)	c) CALL Heuristic
		
$split(s, i) := 1$	$split(s, i) := \left\lfloor \frac{(i-1)k}{ s } \right\rfloor + 1$	$split(s, i) = \begin{cases} split(s, i) & \\ split(s, i-1) + 1 & \text{if } (x_{i-1} = CALL) \\ split(s, i-1) & \text{else} \end{cases}$

Figure 20. Different partition schemes

Obviously, for  $k > 1$ , it has to be considered that  $partition(s)$  is in general not equivalent to  $rewrite(s)$  because *CALL* expressions might be split at the border between two segments. As a result, the complete pattern is not located in any segment and cannot be rewritten. For example, the stream  $s := \{0, CALL\}$ , which contains a call to function  $f_0$ , is uniformly partitioned into  $s^1 := \langle 0 \rangle, s^2 := \langle CALL \rangle$ , which leads to the incorrect fixed-point with  $partition(s) = rewrite(s^1) + rewrite(s^2) = s$ . Hence, in order to avoid these false fixed-points, the stream has to be rewritten at once. As a result, the definition of  $partition$  is extended into the general parallel rewriting function:

$$parallel(s) := \begin{cases} partition(s) & \text{if } partition(s) \neq s \\ rewrite(s) & \text{else} \end{cases} \quad (22)$$

### 2.2.4 Binding

While the *partitioning* defines the splitting of the stream into segments, which might be evaluated concurrently, the term *binding* describes the assignment of work-items to computational resources in a multi-core system. For stream rewriting, the smallest unit of work is a segment  $s^j$ , which is assigned to a rewriting unit of the PSRM (Figure 16, page 23). In particular, the function *bind* assigns a segment  $j$  of the partitioned stream  $s$  to a rewriting unit  $i$ :

$$\begin{aligned} bind: \Sigma^* \times \mathbb{N} &\rightarrow \mathbb{N} \\ (s, j) &\mapsto i \end{aligned} \quad (23)$$

When assuming a system with  $p$  rewriting units, the rewriting of a sub-stream  $s^j$  on a particular rewriting unit  $i \in [1, p]$  is denoted by the indexed function  $rewrite_i: \Sigma^* \rightarrow \Sigma^*$ . Hence, the parallel rewriting function of Section 2.2.3 can be refined into:

$$partition(s) := \sum_{j=1}^k rewrite_{bind(j)}(s^j) \quad (24)$$

Similar to the partitioning schemes, several different binding strategies can be defined.

For instance, the minimalistic binding  $bind(s, n) = 1$ , computes all segments on the same core regardless of the partitioning. A round-robin mechanism, which utilizes all rewriting units, can be specified as:  $bind(s, n) = (n \bmod p) + 1$ . Also, a random binding like  $bind(s, n) = rand([1, p])$  might be possible.

If all rewriting units support the same compute capabilities, there are no further constraints for binding a sub-stream  $s^j$ . However, for heterogeneous systems with limited resources and custom hardware blocks, a specialization of rewrite units is reasonable. For this purpose, the function  $supported(i, j)$  indicates if the rewriting unit  $i$  is able to evaluate  $f_j$ :

$$\begin{aligned} supported(i, j): \mathbb{N} \times \mathbb{N} &\rightarrow \{0, 1\} \\ (i, j) &\mapsto \begin{cases} 1 & \text{if rewriting unit } i \text{ supports function } f_j \\ 0 & \text{else} \end{cases} \end{aligned} \quad (25)$$

The definition of this function depends on the capabilities of an actual implementation. In addition, the function  $required: \Sigma^* \times \mathbb{N} \rightarrow \{0,1\}$  is specified to examine if a stream  $s := \langle x_1, \dots, x_n \rangle$  might contain a call to a particular function  $f_j$ :

$$required(s, j) := \langle j, CALL \rangle \subseteq s \quad (26)$$

By comparing the results of  $supported$  and  $required$ , it can be asserted if a given rewriting unit  $i$  is able to evaluate a segment  $s^j$  of the stream. Hence, the binding in a heterogeneous system must satisfy the following constraint:

$$(bind(s, j) = i) \Rightarrow \forall k \in [1, |F|]: required(s^j, k) \leq supported(i, k) \quad (27)$$

Here, each function  $f_k$  from the set  $F := \{f_1, \dots, |F|\}$  of the SRM, which is required by the sub-stream  $s^j$ , must be also supported by the selected rewriting unit  $i$ .

### 2.2.5 Stream Rewriting Network (SRN)

The stream rewriting network (SRN), which connects the rewriting units and the token stream (Figure 16, page 23), is defined as a tuple of the split **(Equation (17))** and binding **(Equation (23))** functions:

**Definition D4:** A stream rewriting network (SRM) is a tuple consisting of a split and a binding function:

$$SRM := (split, bind)$$

$$split: \Sigma^* \times \mathbb{N} \rightarrow \mathbb{N}$$

$$bind: \Sigma^* \times \mathbb{N} \rightarrow \mathbb{N}$$

The split function must satisfy the following monotonic constraint:

$$\forall a, b \in \mathbb{N}: a \leq b \Rightarrow split(s, a) \leq split(s, b)$$

Three examples for stream rewriting machines are presented in Figure 21, which are based on different variants for  $split$  and  $bind$  from Section 2.2.3 and Section 2.2.4. The abstract network component of (Figure 16, page 23) is now replaced by a concrete topology.

The ring architecture (Figure 21a) with  $SRM_R := (split_R, bind_R)$  concatenates rewriting units in a chain and the results of the first rewriting operation are passed to subsequent cores. Hence, each element of the recursive sequence  $(s_n)$  is partitioned into a single segment that is assigned to a different core.

The example contains two rewriting units ( $p = 2$ ), so that the initial stream  $s_1$  is evaluated on rewritten on unit 1, stream  $s_2$  is rewritten on unit 2 and the third iteration  $s_3$  is again bound to unit 1. For a concrete implementation and reasonable large streams, it can be assumed that both rewriting units are working in parallel for most of the time. An advantage of the ring topology is its simplicity since the network merely consists of queues between the rewriting units.

a) Ring Architecture (p=2)	b) Shared Memory (p=2)	c) Parallel Distribution (p=2)
$split_R(s_n, i) = 1$	$split_M(s, i) = \left\lfloor \frac{(i-1)p}{ s } \right\rfloor + 1$	$split_p(s, i) = \begin{cases} split(s, i-1) + 1 & \text{if } (x_{i-1} = CALL) \\ split(s, i-1) & \text{else} \end{cases}$
$bind_R(s_n, i) = (n \bmod p) + 1$	$bind_M(s_n, i) = i$	$bind_p(s_n, i) = (i \bmod p) + 1$
$SRM_R := (split_R, bind_R)$	$SRM_M := (split_M, bind_M)$	$SRM_p := (split_p, bind_p)$

Figure 21. Different types of stream rewriting networks

The scalability of the ring network has been evaluated for the hardware synthesis of recursive functions (Section 6) and for many-core systems (Section 4). Since the throughput of this chain depends on the slowest rewriting unit, a worst case can occur if the size of stream varies greatly between subsequent iterations. Since each rewriting step from  $s_i$  to  $s_{i+1}$  is performed entirely by a single core, there is one rewriting unit, which is responsible for the most expensive iteration. If the computation time between iterations  $s_i$  of the stream varies frequently, some rewriting units are always over- or underutilized.

The stream rewriting network  $SRM_M$  (Figure 21b) is based on shared memory and divides the stream into as many parts as there are rewriting units. In this example, the stream is partitioned into two segments and bound to two rewriting units ( $p = 2$ ). For  $M_M$ , the network is a crucial part and must be able to handle multiple requests from separate rewriting units in parallel. In addition, the memory of the token stream should provide sufficient bandwidth to keep the cores utilized. Even without considering a concrete implementation, the shared memory SRM most certainly requires more hardware resources than the ring but it performs a parallel rewriting and distributes each step  $s_i \rightarrow s_{i+1}$  equally to all rewriting units.

The third architecture  $SRM_p$  (Figure 21c) contains specialized router components, which switch to another rewriting unit after a CALL token has been detected. As a difference, the routers access the stream although no complete pattern matching is required. In particular, the heuristic can be also performed in parallel on a group of tokens (Section 4.3.5) and provides a more fine-grained load-balancing. As a result, the recombination of stream fragments represents a potential bottleneck.

All three designs have been implemented and a throughput comparison of scalability, performance, and resource usage can be found in Section 4.4.

## 2.3 Scheduling

The scheduling of the stream rewriting process can be distinguished into the two phases. The first one is called the *global scheduling* and takes place before the pattern matching, while the *local scheduling* is responsible for selecting executable *CALL* expressions. Similar to the partitioning, the global scheduling is based on parameters like the size of stream and does not require access to the tokens, while the local scheduling actually has to look into the stream. In this section, both concepts are described according to the post-order format of the stream grammar but can be adapted to the preorder format as well.

### 2.3.1 Local Scheduling

The selection of an executable *CALL* expression (**Definition D3 on page 20**) from the token stream is called *local scheduling*. Data dependencies of concurrent threads are resolved by iteratively applying find-and-replace operations and all modifications of the stream are local. No assumptions concerning the rewriting order on the stream are made and there is no guarantee that a specific *CALL* pattern is evaluated as soon as possible. In contrast to a find-and-replace algorithm, it is not a mistake to ignore some executable expressions. Hence, it is valid to check only a part of the stream and the rewriting can be performed independently or selectively on different sections. Both aspects support the scalability of a hardware implementation since they allow to split the stream and to distribute work items to different rewriting units. In a summary, the advantages of the local scheduling via pattern matching can be described as follows:

- **Automatic scheduling driven by data dependencies**  
The execution of a *CALL* expression can start as soon as there are a sufficient number of preceding literal tokens, which correspond to intermediate results of already completed functions.
- **Light-weight creation and completion of threads**  
New tasks are created by emitting the corresponding *CALL* expressions and do not require interaction with a global scheduler. For implementations, which distinguish the two token types with a single bit, no additional costs are involved.
- **Synchronization using local operations**  
Parallel branches of a call tree can be evaluated in parallel and synchronized via pattern matching as shown for  $f(1)$  and  $f(2)$  of the Fibonacci function in Figure 15 on page 22. Most important, the recombination is a local operation on the stream and always affects a compact range of tokens. As a consequence, the synchronization of threads does not require access to a global memory nor atomic operations.

Although not all applications can benefit from the dynamic scheduling via stream rewriting because their use case requires the predictability of a static schedule, many algorithms from the field of computer graphics fit perfectly into this model of computation (Section 5).

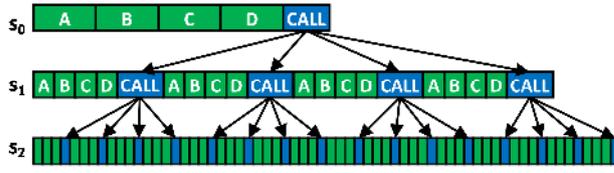


Figure 22. Exponential growth due to recursive expansion

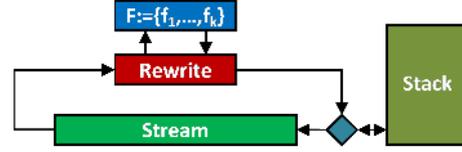


Figure 23. SRM with stack-based scheduling

### 2.3.2 Global Scheduling

Due to the recursive expansion, the stream can grow exponentially as shown in Figure 22. While a large stream potentially contains a huge number of matching rules, which might be evaluated in parallel, its storage requirements can quickly exceed the available memory. Hence, the size of the stream is a trade-off between concurrency and storage capabilities. However, for a particular implementation, it can be assumed that there is a minimum stream size  $b$ , which must be reached to achieve full utilization of all processing cores but above this threshold, little performance benefits are expected.

In this context, the concept of *global stream scheduling* describes, which parts of a stream are considered for rewriting in a particular iteration. For this purpose, the function

$$\text{schedule}: S \rightarrow S \quad (28)$$

selects a sub-stream, invokes *rewrite* on the fragment, and re-assembles the results. For instance, the scanning of the whole stream is described by  $\text{schedule}(s) := \text{rewrite}(s)$  and corresponds to a breadth-first search of the call tree. Hence, for a function, which performs two recursive calls up to a depth of  $d$ , the maximum stream size is bounded by  $O(2^d)$ . Contrary, a depth-first traversal would limit its length to  $O(d)$  but could rewrite at most only one executable rule per iteration.

As a tradeoff between memory consumption and performance, a mixture of depth-first and breadth-first traversal expands only executable CALL tokens within in the first  $b$  tokens of the stream. Hence, for a stream  $s := \langle x_1, \dots, x_n \rangle$ , the scheduling is specified as:

$$\text{schedule} := \begin{cases} \text{rewrite}(x_1, \dots, x_b) + \langle x_{b+1}, \dots, x_n \rangle & \text{if } n > b \\ \text{rewrite}(x_1, \dots, x_n) & \text{else} \end{cases} \quad (29)$$

According to the stream grammar (**Definition D2 on page 20**), this approach first checks the innermost invocations at the beginning of the stream, which are most likely to be ready. For a hardware or software implementation, the inactive remainder can be pushed onto a stack and does not need to be touched in each iteration. As a result, the execution time of an iteration *rewrite* is independent from the total size of the stream and thus also from the total number of managed threads. The data flow model of modified stack-based SRM (Figure 23) contains a switch after the rewriting unit, which redirects the stream to the stack if the limit of  $b$  tokens has been reached. In case of a short stream, tokens are fetched from the stack and appended to the end of the stream accordingly. As a result, the stack works as a buffer for large streams keeps the size of the active part always below or equal to the optimal size of  $b$  tokens.

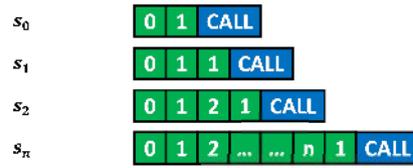


Figure 24. The recursive expansion can produce an arbitrary number ( $n$ ) of literal tokens.

Similar to the partitioning, too small values for  $b$  might cause the SRM to run into a false fixed point. For instance, the stream  $\langle 1, 2, 3, 1, CALL \rangle$  describes the invocation of  $f_1$  using the three parameters  $\langle 1, 2, 3 \rangle$ . Therefore, in case of  $b < 5$ , the SRM never considers the entire CALL expression for rewriting. In addition, the recursive nesting of invocations can produce an arbitrary number of literal tokens at the beginning of the stream. For instance, the following function  $f_1$  is called recursively and produces a prefix of  $b$  literals (Figure 24):

$$f_1(x) := \begin{cases} \langle x, x+1, 1, f, CALL \rangle & \text{if } x < b \\ \langle x \rangle & \text{else} \end{cases} \quad (30)$$

Hence, even well-formed functions can produce intermediate streams with an unbounded amount of literal tokens in the beginning. As a result, there exists no fixed  $b \in \mathbb{N}$ , which is sufficient for all stream rewriting machines. Consequently, further constraints have to be defined in order to guarantee the correctness of the stack-based scheduling. For this purpose, the following parameters are derived from the set of function  $F$ .

1. Let  $p := \max_{f_i \in F} n_i$  be the maximum number of arguments of all functions  $f_i$ . The maximum exists since the set of functions  $F := \{f_1, \dots, f_k\}$  is finite and  $n_i \in \mathbb{N}_0$ .
2. Let  $d$  be the maximum recursion depth of any function in stream

According to the grammar (**Definition D2 on page 20**), also shown below, each invocation is constructed by a sequence of at most  $p$  arguments, the identifier and the CALL token:

$$\begin{aligned} \text{stream} &:= \{expr\} \\ \text{expr} &:= \text{const} | \text{call} \\ \text{const} &:= \langle \text{literal value } z \in \mathbb{Z} \rangle \\ \text{call} &:= \{expr\} i \text{'CALL'} \end{aligned}$$

Hence, a run of literals has a maximum length of  $d \cdot p + 1$  tokens including the identifier  $i$ . By choosing  $b > d \cdot p + 2$ , it can be guaranteed that the prefix  $\langle x_1, \dots, x_b \rangle$  of stream  $s := \langle x_1, \dots, x_b, \dots \rangle$  contains at least one CALL token. Therefore, there exists also a first CALL token  $x_j$  with  $j := \min\{j : x_j = CALL\}$  in the stream, which is only preceded by literals, so that the stream has the following format:

$$s := \langle \underbrace{x_1, \dots, x_{j-2}}_{\in \mathbb{Z}}, i, CALL, \dots, x_b, \dots \rangle \quad (31)$$

As a result, the first CALL is preceded by  $j - 2$  literals and the identifier  $i$ . It becomes executable and can be rewritten if there are at least  $n_i$  arguments. In case of  $j - 2 < n_i$ , the stream is malformed since the requested number of arguments can be never produced from the literals  $x_1, \dots, x_{j-2}$ . Thus, for a well-formed stream, there is always at least one executable CALL within the first  $d \cdot p + 2$  tokens. Also important, the assumptions made in this section require that literal results are removed from the stream as soon as possible.

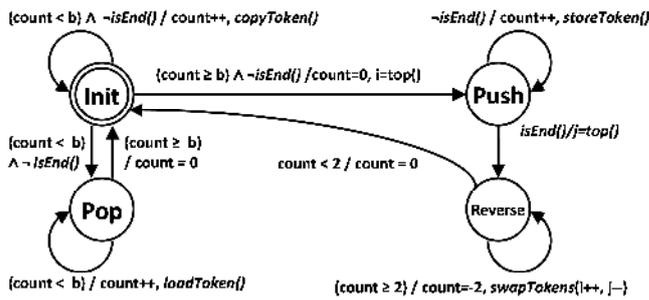


Figure 25. Finite state machine controlling the stack.

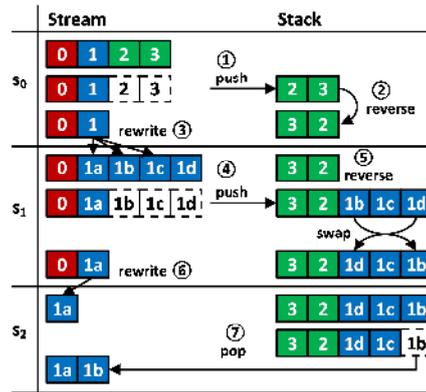


Figure 26. Saving and restoring tokens.

### 2.3.3 Stack Management for Global Scheduling

Formally, the behavior of the stack can be described as a finite state machine presented in Figure 25 with the four states INIT, PUSH, REVERSE, POP, which are specified as follows:

- State: INIT**  
 Initially, tokens are passed through (*copyToken*) and increment a counter (*count*). If the optimal size of *b* tokens has been reached, indicated by  $count \geq b$ , the stack switches into the **PUSH** state to store the remainder of the stream on the stack. Contrary, if the end of the stream is reached (*isEnd*) without the threshold being surpassed yet ( $count < b$ ), the stack switches into the **POP** state to fill up the stream.
- State: PUSH**  
 In this state, the remainder of the stream is stored in memory (*storeToken*). Since the tokens are written sequentially in FIFO order, the newly appended segment must be reversed (Figure 26), so that the state machine goes into the **REVERSE** state, when the end of the stream has been detected.
- State: REVERSE**  
 After the remainder of the stream has been stored on the stack, the REVERSE state swaps pairs of tokens in-place (*swapTokens*) before returning to the **INIT** state. For this purpose, the two pointers *i* and *j* are initialized to the beginning and the end of the newly appended section. Here, *top()* denotes the topmost element of the stack. In each step, the associated tokens are exchanged and the pointers are moved towards each other. This reversal is necessary since the recently pushed token is currently at the top of stack but should be fetched last. The layout of the stack in this state is also illustrated in steps (2) and (5) of Figure 26.
- State: POP**  
 The previously stored tokens are appended to the end of the stream (*loadToken*) until the optimal size is reached or the stack becomes empty. Eventually, the stack returns to the initial state to process the next iteration.

Figure 26 illustrates the process of loading and storing of tokens by showing the contents of both the stack and the stream for three iterations. Here, the threshold is set to  $b = 2$  tokens. As a consequence, the last two of the four tokens in  $s_0$  are pushed onto the stack **(1)** and will be reversed **(2)**. In this example, the blue token 1 is expanded into 1a, 1b, and 1c from  $s_0$  to  $s_1$ . **(3)**. Again, the last two tokens (1b; 1c) are stored **(4)** on the stack and are also reversed **(5)**. Eventually, in  $s_2$ , the stream shrinks **(6)** and the available position is filled-up with token 1b from the stack **(7)**. As a result of the swap, the next token to be fetched is located at the top of the stack. Also, the stored tokens are not touched again until there is enough space available on the stream.

A short example of this scheduling technique for a recursive call tree is shown in Figure 27. Here, the color of each token shows the branch and the number displays its depth. In this example, the threshold  $b$  is chosen as  $b = 4$ , so that the stream is expanded quickly until  $s_2$  when the workload is sufficient to fully utilize the system. From this point, only the first 4 tokens, which correspond to the inner tasks, are rewritten. Eventually, the stream shrinks again, the postponed tokens are fetched from the stack and move back into the active part.

In general, the memory requirements of this technique can be analyzed as follows:

If the call tree has a maximum depth of  $d$  and performs two recursive invocations per node, a breadth-first approach would expand all executable CALL expressions, which leads to a memory requirement of  $O(2^d)$  and  $O(d)$  iterations (Table 1). Contrary, a depth-first traversal expands only the first CALL per iteration and therefore requires  $O(2^d)$  iterations with a maximum stream size of  $O(d)$ . The stack-based scheduling represents a tradeoff between these two approaches:

As long as the system is underutilized with  $|s| < b$ , all executable tasks are expanded to reach the optimal operating point. If there exists an iteration  $i$  with  $|s_i| \geq b$ , this approach is equivalent to  $b$  parallel depth-first traversals with a maximum depth of  $d$ . Thus, the memory requirement is bounded by  $O(bd)$  and the number of iterations can be approximated by  $O(2^d/b)$ . Hence, in comparison to a breadth-first traversal, there are more but smaller iterations, so that the total number of executed tasks remains the same.

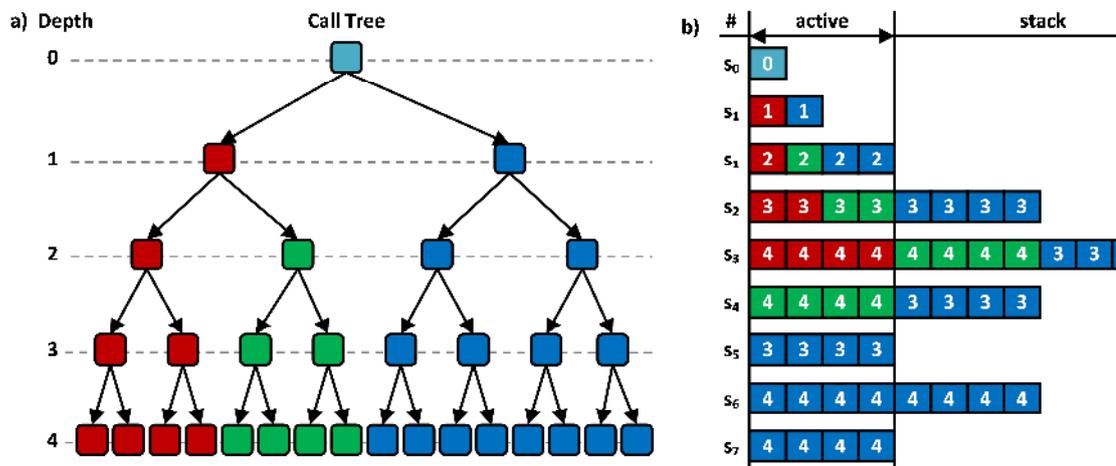


Figure 27. Combination of breadth-first and depth-first evaluation

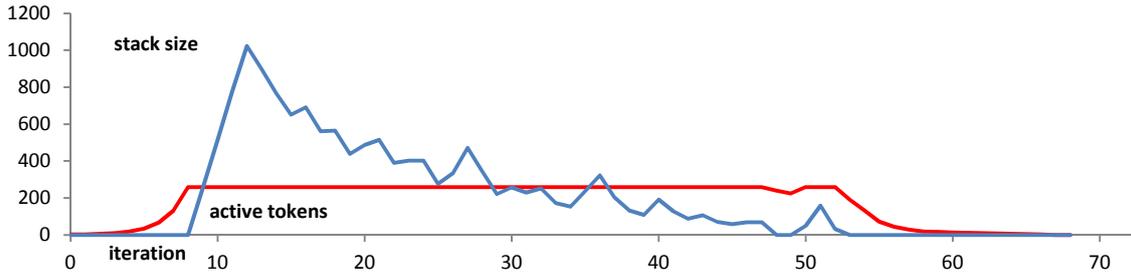


Figure 28. Stack and stream size per iteration for computing the recursive sum from 0 to 100.

For example, a typical branch-and-bound algorithm recursively subdivides a given problem into smaller tasks, which can be solved individually and later recombined. This class of algorithms is in particular well-suited for stream rewriting because it allows evaluating individual tasks in parallel. A generic example for such an algorithm is the recursive function  $rsum(a, b)$ , which accumulates the interval  $[a, b]$ :

$$rsum(a, b) = \begin{cases} rsum\left(a, \left\lfloor \frac{a+b}{2} \right\rfloor\right) + rsum\left(\left\lfloor \frac{a+b}{2} \right\rfloor + 1, b\right) & \text{if } a > b \\ a & \text{else} \end{cases} \quad (32)$$

Each invocation of  $rsum$  creates two additional instances of this function, so that the number of tokens is roughly doubled per iteration (Figure 27). As a result, the total memory requirement of  $rsum(a, b)$  can be approximated as  $O(b - a)$ . For comparison, Figure 28 shows the size of the stack (blue) and the token stream (red), when using the proposed scheduling algorithm with  $b = 256$ . In fact, the length of the stream increases exponentially until the target size of 256 tokens has been reached. Then, the remainder is stored on the stack and the number of tokens grows linearly per iteration. In particular, the length of the active stream remains constant while the stack compensates the varying amount of tokens. At the end of the computation, both the stack and the stream shrink until all tokens are processed.

As a result, the global scheduling algorithm utilizes the stack to reduce the exponential expansion into a linear growth. Important for a possible hardware implementation, the scheduling decision can be made by a relatively simple state machine (Figure 24, Page 33)

Table 1. Comparison of binary tree traversal algorithms.

	Depth-First	Breadth-First	Parallel Stack
Tasks	$O(2^d)$	$O(2^d)$	$O(2^d)$
Memory	$O(d)$	$O(2^d)$	$O(bd)$
Iterations	$O(2^d)$	$O(d)$	$O(2^d/b)$

### 2.3.4 Large Scale Expansion

For many-core systems with more than hundred processor cores, an adequate number of work items or tasks are necessary to fully utilize the available computing resources. Often, it is also more efficient to generate tasks dynamically within the processor instead of loading a large stream from external memory. Hence, a mechanism must be provided, which is able to quickly expand function calls at runtime. In the traditional data-parallel model of OpenCL and CUDA, a kernel function is invoked  $n$  times in parallel on a 1D, 2D or 3D grid. For the 1D case, this behavior can be replicated using the following rewriting rule that invokes the kernel function  $f_i$  on the interval  $[a, b]$ :

$$f(a, b, i) := \langle a, i, CALL, a + 1, i, CALL, \dots, b - 1, i, CALL, b, i, CALL \rangle \quad (33)$$

However, for large values of  $n := b - a + 1$ , this approach is not suitable since it schedules the entire expansion onto a single core. For example, if  $n$  is set to 1,000,000, this particular core will produce 3,000,000 tokens during the execution of  $f$ . Since the evaluation of a rewriting rule is atomic and cannot be interrupted by design, most of the remaining cores will be blocked until all 3,000,000 tokens are written to the stream. While multiple rules can be evaluated in parallel, the resulting sub-streams must be reassembled in the correct order. For this purpose, both hardware and software implementations usually require reorder buffers as a temporary storage. If the capacity of these buffers is exceeded, the evaluation of multiple instances is serialized and therefore reduces the throughput.

For graphics hardware, there exist similar constraints that limit the number of outputs per shader instance. For example, the *geometry shader* of Direct3D can generate up to 1024 floating-point values per invocation and is therefore not suitable for large-scale data amplification. In addition, its performance significantly decreases for larger number of outputs, which has shown by [91] and [92]. More important, the stream would also grow excessively because all 1,000,000 instances are generated at the same time and thus render the scheduling approach of Section 2.3.2 almost useless.

Actually, the parallel rewriting of a token stream faces exactly the same problem because each CALL expression can produce a variable number of outputs. However, it is possible to perform the expansion recursively, so that the number of outputs per instance remains below the critical threshold and takes advantage of the stack-based scheduling. Similar to the recursion sum, the following branch-and-bound algorithm subdivides the interval  $[a, b]$  until the maximum block size of  $T$  has been reached:

$$f(a, b, i) = \begin{cases} f\left(a, \left\lfloor \frac{a+b}{2} \right\rfloor, i\right) + f\left(\left\lfloor \frac{a+b}{2} \right\rfloor + 1, b, i\right) & \text{if } b - a > T \\ \langle a, i, CALL, a + 1, i, CALL, \dots, b - 1, i, CALL, b, i, CALL \rangle & \text{else} \end{cases} \quad (34)$$

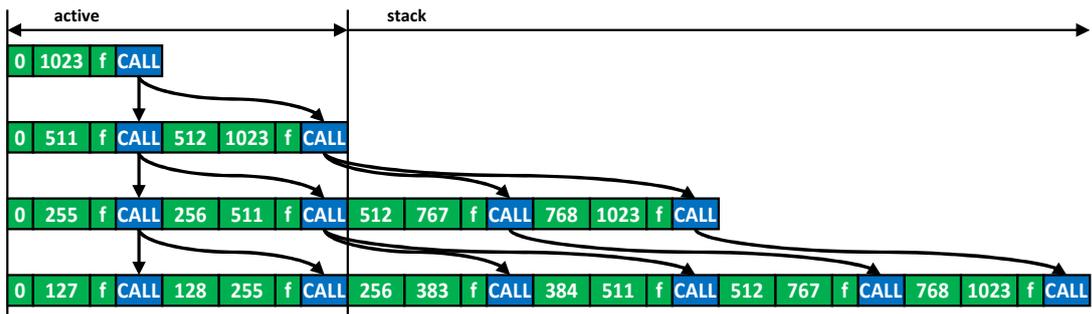


Figure 29. Recursive expansion of 1024 instances.

For example, the recursive expansion of the interval  $[0,1023]$  using this technique is illustrated in Figure 29. Due to the stack-based scheduling, the active part of the stream is sub-divided once per iteration, while the remaining part on the stack is left unmodified. Initially, the top-level call is evaluated on a single core, but starting from the second iteration, the expansion could potentially run on two cores in parallel. As a result, recursion is not only supported by stream rewriting but rather essential to ensure optimal task distribution and also to reduce the maximum size of the stream.

As an alternative, there is also a mechanism for instancing (Section 2.4), which offers a compact encoding for data parallel tasks. However, in contrast to the recursive expansion, it relies requires a separate grammar rule and therefore does not work on the basic SRM. On the other hand, the explicit instancing provides more information to the SRM and can be therefore also better optimized. For example, the graphics processor (Section 5.4) expression takes the size of a two-dimensional region and a hardware scheduler then distributes the entire set of tasks uniformly to the processor cores.

### 2.3.5 Open Systems

The initial stream  $s_0$  might be used to describe the input of a continuously running device like an embedded system or a graphics processor. In this case, there probably exists no upper bound for the number of tokens in  $s_0$ , so that the total length of the stream is either unknown or infinite. Hence, a scheduling algorithm, which requires random access to the stream, is not adequate in such a scenario since only a small part of the stream is accessible at any time. Instead, the system should incrementally rewrite and combine incoming tokens. For this purpose, the model of the SRM is updated according to Figure 30 and contains the novel entry and exit points. In particular, the entry point appends tokens to the stream, while results are extracted at the exit point. In particular, results can be either literal values or more specified specialized *OUT* tokens described in Section 2.5.1.

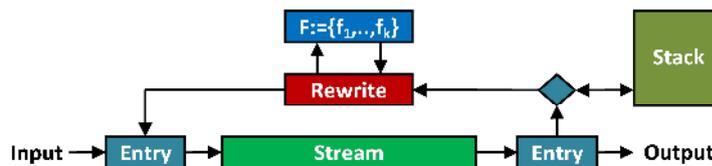


Figure 30. SRM with entry and exit for processing infinite streams.

As a fundamental insight and similar to the stack-based scheduling, it is actually sufficient to rewrite a finite sub-stream of a potential infinite large stream. Hence, the stream  $s \in S$  is partitioned into a finite front part  $u$  and a possible infinite remainder  $v$ :

$$s := u + v \text{ with } |u| < b \quad (35)$$

The first part  $u$  has a maximum length of  $b$  tokens and can be passed to the rewriting function, while the remainder  $v$  is left unmodified:

$$\text{rewrite}_{inf}(s) := \text{rewrite}(u) + v \quad (36)$$

The entry point of the modified SRM (Figure 30) switches between the input stream and the already circulating tokens in the ring. In particular, it appends incoming tokens to the end of the current stream, as long as it is shorter than  $b$  tokens, so that in combination with the stack, the active part of the stream is always kept below this threshold. In Section 2.3.2, it has been shown that a lower bound for  $b$  can be constructed based on the maximum number of arguments and the recursion depth of all rewriting rules. As a result, the rewriting of the infinite stream can be reduced to a finite prefix of  $b$  tokens.

Each optimized hardware or software implementation already defines some limits for these parameters based on the size of internal queues and local storages, so that these additional requirements impose moderate costs. For instance, to handle functions with up to 255 arguments and 32 levels of recursion, the system must be able to store a stream of  $\approx 8192$  tokens, which can be packed into 32Kb for a 32-bit SRM.

## 2.4 Stream Compression

Although the SRM can work on streams of arbitrary lengths, the storage requirements and computation time of a particular implementation should roughly correspond with the number of processed tokens. For instance, a smaller stream generally causes less stack operations, which usually access the slow external memory. Similarly, the performance of the abstract SRM architectures shown in Figure 21 on page 30 depends on the stream processing bandwidth. Also, for computational less expensive tasks with a large number of arguments, the pattern matching can become a bottleneck of the architecture. As an optimization, two mechanisms are proposed, which reduce the size of stream by removing redundant data:

1. The concept of *instancing* improves the throughput of data parallel tasks, which are called several times using almost the same set of arguments.
2. Hierarchical *environments* define a set of shared variables for a branch of the call-tree and can contain different types of tasks.

Both approaches move shared data, which would have been passed as arguments, into a central location and therefore reduce the number of temporary data on the stream.

### 2.4.1 Instancing

The basic model of stream rewriting already supports data parallelism through recursion. For example, the technique for large scale expansion (Section 2.3.4) utilizes the stack to iterate over huge data sets in parallel. However, all invocations must be still written to the stream and thus require bandwidth and computation time. Hence, especially for the medium sized leaves of a recursive expansion, it would be advantageous if a data parallel task could be written once and distributed to a given amount of rewriting cores.

In the context of graphics processing, the term *instancing* usually describes the explicit repetition of a task and is often handled by specialized hardware for performance reasons. In particular, *instancing* can be used to draw a large number of similar objects with minimal CPU usage [17]. For stream rewriting, the basic grammar is extended by the following rule:

**Definition D5:** The evaluation of a CALL expression on the interval  $[0, m) \in \mathbb{Z}$  is specified by the instantiation rule:

$$call\_inst := \{expr\} m i 'CALL\_INST'$$

As a result, the function  $f_i$  is invoked  $m$  times for each element of the interval  $[0, m) \in \mathbb{Z}$  and the index of the current iteration is appended to the call. Formally, the instance expression is expanded into a sequence of calls as shown in the following rewriting pattern:

$$\underbrace{\langle x_1, \dots, x_k, m, i, CALL\_INST \rangle}_{\in \mathbb{Z}} \rightarrow \underbrace{\langle x_1, \dots, x_k, 0, i, CALL \rangle}_{Instance: 0}, \dots, \underbrace{\langle x_1, \dots, x_k, m-1, i, CALL \rangle}_{Instance: m-1} \quad (37)$$

For example, the streams  $s_0$  and  $s_1$  in Figure 31 are equivalent but  $s_0$  requires only a fraction of the memory. Similar to the CALL expressions, the instantiation rule requires all arguments  $x_1, \dots, x_k, m$  to be literal values but the exact instant of the unfolding is implementation-specific. As a result, the SRM can optimize the processing depending on the current size of the stream and in combination with the global scheduling. Hence, the instance expression is formally equivalent to a sequence of calls but allows for additional optimizations at runtime. Therefore, the SRM may switch to a recursive or partial expansion if the length of the stream has already reached the threshold  $b$  for global scheduling.

Likewise, an implementation may dispatch the task to several rewriting units in parallel without storing all instances in memory. Thus, *call\_inst* expressions are comparable to vector instructions and signal explicit data parallelism to the underlying hardware.

Instance expressions can be also nested in order to iterate over multiple dimensions or hierarchical grids. In combination with regular calls, the expansion can be also performed conditionally to skip large groups of instances and therefore accelerate recursive branch-and-bound algorithms.



Figure 31. Instancing can significantly reduce the stream size for data parallel problems.

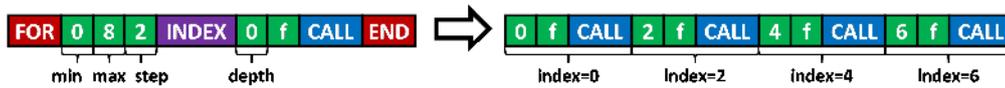


Figure 32. Expansion of a FOR/END block with min=0, max=8 and step=2.

A more general alternative to the instanced calls is the FOR/END block, which replicates the enclosed sub-stream regardless of the contained tokens (Figure 32):

**Definition D6:** FOR loops are described as a block of FOR/END tokens and take three literals to specify *start*, *end*, and *step size* of the loop counter:

$$for := 'FOR' min max step \{expr\} 'END'$$

INDEX tokens within the loop are replaced by the counter of the loop at the given *depth*. A depth of zero refers to the innermost loop.

$$index := 'INDEX' depth$$

At runtime, the inner sub-stream is replicated according to the given range and contained INDEX tokens are replaced by the current iteration number. In addition, loops can be also nested to iterate over multi-dimensional grids. For this purpose, the INDEX token also specifies a relative depth to identify the correct loop. The example shown in Figure 32 iterates from 0 to 8 with a step size of 2, so that the SRM produces the four iterations with index values 0, 2, 4, and 6.

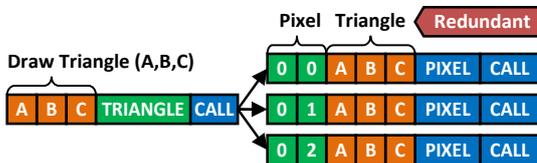


Figure 33. Triangle parameters are replicated.

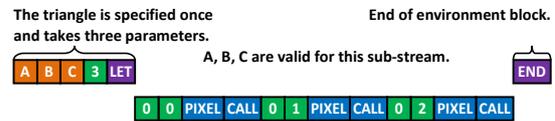


Figure 34. Stream compression using environment.

### 2.4.2 Environments

The second technique for stream compression deals with the automatic management of shared data in a highly concurrent environment. Since stream rewriting supports recursive tasks, shared data can also occur at different levels of the hierarchy.

For example, the rendering of a scene usually starts with one or multiple draw calls, which are successive expanded into triangles and finally rasterized into pixels. Often, each level of this hierarchy requires access to certain parameters of its predecessor steps. Hence, shared data must be replicated and passed as arguments to a large number of subsequent CALL expressions. For instance, the computations of a pixel are based on the coordinates of a triangle or the result of a triangle-setup as shown in Figure 33. In this example, the triangle data must be replicated for each pixel and therefore introduces a significant overhead.

For this purpose, local environments allow specifying a set of shared parameters  $c_1, \dots, c_n$  only once for a sub-stream enclosed into a pair of *LET/END* tokens:

**Definition D7:** An environment block guarantees that the shared parameters  $c_1, \dots, c_n$  are initialized during the evaluation of the inner expression *expr*.

$$env := c_1, \dots, c_n, n, 'LET' \{expr\} 'END'$$

Since the results of an environment might be passed as an argument to another *CALL*, literals at the end of a block are extracted by exchanging them with the *END* token:

$$\langle i \in \mathbb{Z}, END \rangle \rightarrow \langle END, i \in \mathbb{Z} \rangle \quad (38)$$

Finally, empty environments are removed from the stream using the rule:

$$\langle LET, END \rangle \rightarrow \langle \rangle \quad (39)$$

Environment blocks offer an efficient mechanism for the distribution of shared read-only data and support automatic memory management. Neither current graphics nor compute APIs provide a comparable functionality: For instance, CUDA and OpenCL support data-exchange between threads via shared memory, but it must be managed manually and relies on explicit synchronization. On the other hand, the aggregation of shared parameters hinders the parallel distribution of the token stream (Section 2.2.3) because the SRM has to read every environment before the contained *CALL* expressions can be evaluated.

## 2.5 Memory Access

Several applications like graphics processing require both read and write access to large datasets that cannot be stored on the stream due to its size. In addition, random memory access is necessary for some operations but does not fit well into the concept of stream rewriting, which focusses on locality instead. Since shared memory cannot be avoided completely, this section introduces a safe and predictable model for the integration of side-effects into the rewriting process.

According to the formal specification, the functions  $f_i \in F$  are pure, so that the final result of the stream does not depend on the evaluation order. Actually, the scheduling of rewriting rules is based on the partial order defined by their data dependencies. However, a rewriting function  $f_i$  is allowed to read or even modify a shared memory, the partial

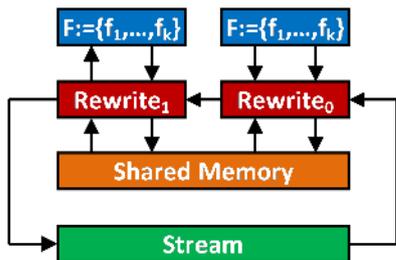


Figure 35. SRM with shared memory

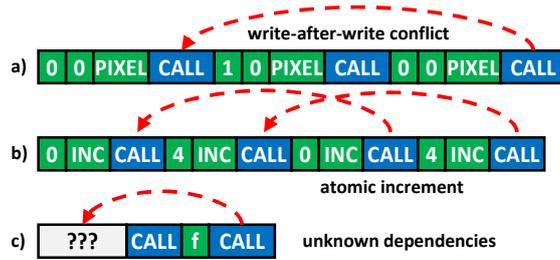


Figure 36. Different types of conflicts on the stream.

ordering may be not sufficient to avoid conflicts. Hence, the extension of the SRM model with shared memory access (Figure 35) effectively introduces global data dependencies that are not encoded into the stream and therefore invisible to the scheduler.

Some potential conflicts are illustrated in Figure 36. For example, writes to the same address or pixel must be performed in order (Figure 36 a) and the execution of atomic increments (Figure 36 b) should not overlap. The worst case is shown in Figure 36 c) and here, the dependencies of  $f$  are unknown because the corresponding tokens have not been expanded yet. In general, the following options are available for handling memory access:

- **Unordered Access**

In the most simplistic case, the order of memory operation is either not relevant or can be ensured by inserting artificial data dependencies into the CALL expressions. This approach has been proposed for the high-level synthesis of recursive functions and the general purpose architecture based on shared memory (Section 4.2). If the SRM is based on shared memory anyway, this model does not require additional rules or hardware components. As an example, the *mandelbrot* and *raytracing* test cases (Section 4.3.6) write each pixel of the screen only once.

- **Platform-dependent Synchronization**

Although, the SRM does not account for memory request, each rewriting function can still use the synchronization capabilities of the underlying implementation like atomic operations or semaphores. In this case, the memory model of the basic SRM would be comparable to CUDA or OpenGL, which provide synchronization primitives for shared memory access but also require the manual synchronization of threads.

- **Shared Environments**

Alternatively, if shared data is written only once and read by multiple subsequent functions, the shared storage can be modeled using environments (Section 2.4.2). As an advantage, no external memory is required since all data is kept on the stream and will be automatically garbage collected. Hence, this technique is most useful for exchanging temporary data between cooperating threads but unfeasible for global results because environments do not persist on the stream.

While the existing capabilities of the SRM provide a reasonable compromise for some applications, they are not sufficient to handle a large number of random memory requests and read-modify-write operations, which are commonly used in graphics processing. Hence, in the next two sections, the synchronization of writes (Section 2.5.1) and the atomic execution of rewriting functions (Section 2.5.2) are specified.

### 2.5.1 Write Ordering

If an application is performing ordered write operations, the final stream can be interpreted as a sequence of address and data tuples that are intended to be stored in memory. The final result of the rewriting process is specified as the limiting value of the sequence:

$$\begin{aligned} result(s_0) &:= \lim_{n \rightarrow \infty} s_n \\ s_{n+1} &:= rewrite(s_n) \end{aligned} \tag{40}$$

By definition, the limiting value consists of tokens, which cannot be rewritten further and therefore remain on the stream. Since all rewriting operations maintain the relative alignment of the stream, the results are automatically ordered. Hence, the stream  $result(s_0)$  offers the possibility to return an output stream of literal values. For simple functions like *Fibonacci* (Section 2.1.4), which compute a scalar value, the output stream also contains a single literal (Figure 15, Page 22):

$$result(\langle 3, fib, CALL \rangle) = \langle 2 \rangle \tag{41}$$

When evaluating multiple different functions in parallel, the results are concatenated:

$$result(\langle 3, fib, CALL, 4, fib, CALL \rangle) = \langle 2, 3 \rangle \tag{42}$$

Since the results correspond to CALL expressions of the initial stream, this technique is in particular useful for rendering algorithms that construct an image in multiple steps and thus are expected to maintain the original draw order. This method is applied in Section 2.5.1 and has been evaluated as part of the stream rewriting network in Section 4.3.3 as well as for graphics processing in Section 5. Especially for the post-order encoding, the pattern matching has to look-ahead a large number of tokens to determine if a literal value is left on the stream or can be possibly applied as an argument to a subsequent CALL. Thus, the basic stream grammar is extended with a special OUT token, which definitively marks the preceding tokens as final:

**Definition D8:** The output token is never rewritten and marks the inner expression *expr* as final result:

$$out := expr 'OUT'$$

The graphics processor in Section 5.4 utilizes a more specialized version of this generic mechanism and supports a PIXEL token, which additionally contains coordinates, color and depth values. Instead of  $(addr, data)$  tuples, the output stream can be also compressed using a run-length encoding to save bandwidth (Section 4.3.3).

This technique enables ordered writes at moderate costs and fits into the functional model of stream rewriting. In addition to literal results and further invocations (CALL), a rewriting function can also return a sequence of memory writes. However, a significant disadvantage of this method is that even non-conflicting write operations are serialized and kept on the stream. Hence, in the following section, a more fine grained approach for synchronizing both read and write operations is presented.

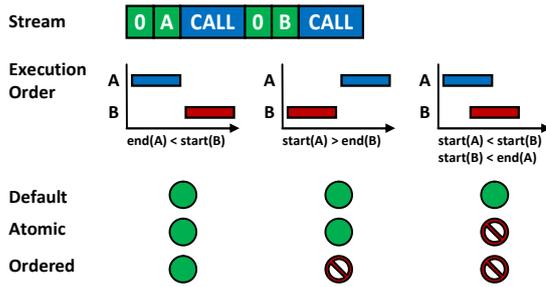


Figure 37. Execution orders used for synchronization.

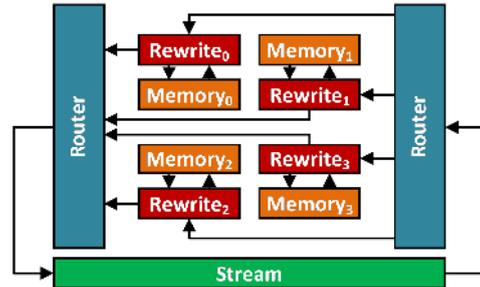


Figure 38. SRM with distributed shared memories.

### 2.5.2 Synchronization

In general, the SRM has to support the synchronization of arbitrary read or write operations, random memory access and atomic operations. High-performance applications like graphics processing usually require a significant bandwidth of random and irregular read-modify-write operations into external memory. For instance, the rendering of a single pixel might generate at least four memory reads for each bi-linear texture fetch as well as two atomic read-modify-write operations of the depth and color buffers. While texture fetches are mostly unordered reads, the framebuffer usually requires atomic and in case of blending also ordered access.

More generally, memory conflicts are side effects and cannot be determined from the stream at this point because calls encode only local data transfers. As a consequence, the grammar of the stream is extended again to explicit provide information of global dependencies. Formally, let  $A$  and  $B$  describe two instances of a function,  $start(A)$ ,  $start(B)$  the starting times of their execution, and  $end(A)$ ,  $end(B)$  their end times. In addition, the functions  $addr(A)$  and  $addr(B)$  specify the memory locations, which are accessed during the execution of  $A$  and  $B$ . In particular, the execution of a rewriting function is ordered according to the following three categories (Figure 37):

- **Default:** The rewriting function can execute without any restrictions. It can perform arbitrary memory operations and emit all types of CALL expressions. Hence, any execution times, which satisfy the following condition, are allowed:

$$start(A) \leq end(A) \wedge start(B) \leq end(B) \quad (43)$$

- **Atomic:** The execution of the rewriting functions  $A$  and  $B$  may not overlap with other instances bound to the same address. It can emit all types of tokens and CALL expressions but the following constraint must be considered:

$$addr(A) = addr(B) \Rightarrow [start(A), end(A)] \cap [start(B), end(B)] = \emptyset \quad (44)$$

- **Ordered:** All functions of the same address are evaluated in the order they appear on the completely expanded stream. Hence, for a stream  $s := X + B + Y$  where any derived stream  $X' \in X^*$  might contain  $A$ , the execution intervals must be disjoint and strictly ordered. In addition, an ordered function cannot emit further calls.

$$\exists X' \in X^*: A \in X' \Rightarrow start(A) \leq end(A) < start(B) \leq end(B) \quad (45)$$

The extended SRM is illustrated in Figure 38 and contains the following two additions: First, the address space of the shared memory is partitioned and each rewriting unit has exclusive access to its own block. Hence, in this example, there are four rewriting units, four blocks of memory and consequently, the address space must be divided into four segments. An actual implementation might also combine the four memory blocks into a single external memory as long as the responsibility of a certain rewriting unit for a specific address range is preserved.

The second addition are the router components, which have been already shown for the parallel distribution of the stream in the generic SRM (Figure 21c, Page 30) and are responsible for assigning executable rewriting functions to rewriting cores. Here, the routers are performing the binding according to the memory location a rewriting function is potentially accessing. For instance, all instances, which might require a synchronized access to the address range of memory block 1, are passed to rewriting unit 1. In order to describe the new semantics, the grammar of the stream is extended with the following rules for atomic and ordered CALL expressions, which explicitly state the utilized address:

**Definition D9:** Atomic and ordered calls are specified by the rules:

$$\begin{aligned} call\_atomic &:= \{expr\} \text{ addr } i \text{ 'ATOMIC' 'CALL' } \\ call\_ordered &:= \{expr\} \text{ addr } i \text{ 'ORDERED' 'CALL' } \end{aligned}$$

Hence, for ordinary CALL expressions, a rewriting function can be bound to any core but atomic functions must be executed on the rewriting unit containing the specified address. Ordered functions additionally have to check if there are any preceding and not yet executed invocations on the stream. Since the address is threatened as an opaque handle by the scheduler, it may point to an actual memory location but can also refer to a memory range, a pixel column or another shared resource.

The most restrictive category is the ordered execution since it guarantees that all preceding instances on the stream with the same address have been completed with  $end(a) < start(b)$ . In the worst case, a rewriting function consequently has to wait until all previous default calls are expanded and the entire set of dependencies has been resolved, which is similar to the mechanism of the R-Buffer [93]. However, due to the recursive expansion of the stream, memory dependencies might be unknown (Figure 36c) for some sub-stream and thus, all subsequent ordered functions must be blocked completely. As a result, the strict execution order represents the most expensive category of synchronization. Though, according to the global scheduling algorithm (Section 2.3.2),

which prefers the beginning of the stream, the waiting instances might be pushed on the stack. As an optimization, a function without an explicit memory address can be also declared as constant to indicate that it will not introduce memory conflicts.

**Definition D10:** A constant function does not perform memory writes and invokes only constant functions.

$$call\_const: = \{expr\} i 'CONST' 'CALL'$$

Since a constant call guarantees that it will never expand into a non-constant invocation, it does not interfere with atomic and ordered operations. For high-level languages, the constant annotation might be either specified as part of the signature like in C/C++ or determined automatically by the compiler as part of the global optimization step.

### 2.5.3 Discussion

The concept for memory synchronization generalizes the approach of *Pomegrate* [94], which contains several reordering networks to reorder to temporary results between subsequent pipeline stages. Otherwise, there also exist alternative techniques to ensure deterministic memory access in a concurrent environment:

For general purpose CPUs in multi-core systems, atomic operations are usually implemented by locking the memory bus or as a combination of *load-linked* and *store-conditional* instructions in the MIPS processor [95]. However, these fine-grained techniques does not provide the required scalability for many-core architectures since there is a large number of pending requests hiding the latency of the external memory.

GPUs usually contain dedicated raster output stages (ROP) for blending and atomic operations [9]. Since the SRM is designed towards a more flexible software-centric approach, specialized hardware components are omitted. As an alternative, atomic operations can be also supported directly by the memory subsystem but usually a large L2 cache is required to hide the latency of external memory like in the Fermi design [96].

The dynamic binding of the SRM ensures the atomic execution of an entire rewriting function and therefore supports all combinations of atomic operations for both integer and floating-point data types. Most important, it also avoids the need for globally locking certain memory locations. However, as a disadvantage, the throughput of both arithmetic and memory operation is coupled more tightly, so that the distribution of the memory addresses can impact the load balancing negatively. For instance, if the majority of memory requests targets an address in block zero, also the arithmetic part of these functions is evaluated by rewriting unit zero (Figure 38, Page 45). However, such a worst case can be prevented by interleaving addresses between blocks and possible splitting functions into arithmetic and memory sections, which are then routed independently.

### 3 Source Models

After specifying the basic concept of the stream rewriting machine, three source models are presented in this section. At the lowest level of abstract, a functional language directly corresponds to the execution model of the stream rewriting machine (Figure 39). In addition, task graphs and also imperative C source code can be layered on top of this functional model.

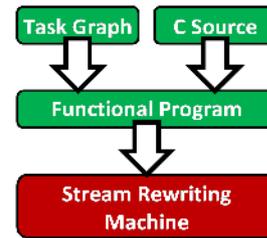


Figure 39. Source Models

#### 3.1 Functional Programs

In this section, a functional language for stream rewriting is proposed, which can be seen as an assembly language for the SRM and is based on the concept of paper [85].

##### 3.1.1 Introduction

Actually, the basic execution model of stream rewriting already defines a minimalistic functional program  $F := \{f_1, \dots, f_k\}$  with a grammar consisting of nested CALL expressions. Similarly to existing languages like *ML* [97] or *Haskell* [98], imperative constructs like loops and branches can be emulated using recursion. For example, the high-level compiler for hardware synthesis (Section 3.3) generates a set of rewriting rules from a C program by transforming basic blocks into separate functions and passing active variables as arguments. Also important in the context of functional programming, the rewriting of a CALL expression is strict because it requires all arguments to be evaluated before a function is entered. Contrary, lazy languages like *Haskell* compute arguments on demand. In the execution model of stream rewriting, the strict evaluation order is the only guarantee, which is given regarding the execution time of a function. Beside random memory access (Section 2.5), rewriting rules are mathematically pure, free of side-effects and therefore well-suited to model concurrent computations. As a result, this model exposes mostly the same properties and constraints as a strict functional programming language. Hence, programming the SRM in a functional style fits well to the underlying concepts of iterative rewriting.

##### 3.1.2 Related Work

The evaluation of functional programs using rewriting of sequences has been already shown for the language JOY [99]. However, since their tokens can contain arbitrary values including lists, this approach is not directly suitable for a hardware implementation. In addition, the work of [100] describes the relation between term rewriting systems and functional languages. Especially for multi-core systems, a stateless functional model can remove the need for atomic operations and memory coherency, which has been evaluated in [101] using a minimalistic C++ based  $\lambda$ -calculus.

Likewise, the natural parallelism of term rewriting and the support for multiple types of concurrency has been emphasized by [102]. In fact, the authors also propose a partitioned term rewriting to automatically utilize multiple processors. The central difference to the SRM is the storage of the rewriting expression. In their system, there is a fixed set of

random accessible cells, which store intermediate results and dependencies. Likewise, there is also a bit to distinguish between reduced and complex terms, which roughly correspond to the literal and CALL expressions. However, the SRM stores expressions as a stream and dependencies are given through relative positioning using a grammar. As an advantage, the storage for terms does not need to be allocated explicitly but instead, any expression can be dynamically inserted into the stream.

The *Reduceron* [103] processor and its successor the *PilGRIM* [104] architecture also execute functional programs, which are based on minimal subset of *Haskell*, directly in hardware. Concurrency is exploited by reducing several independent expressions in parallel, so that a function can be possible evaluated in a single cycle, which basically corresponds to the instruction-level parallelism (ILP) also found in general purpose processors. Since the *Reduceron* processor is not pipelined, it is unclear how efficient floating-point operations might be executed on this architecture. Both systems employ a garbage-collected heap to store active nodes and therefore require random memory access. Contrary, stream rewriting is based on local pattern matching. Shared data is embedded into the stream using *LET/END* environments (Section 2.4.2) and also automatically garbage-collected.

### 3.1.3 Language Definition

Similar to assembly code, the functional language is minimalistic representation of the lowest abstraction and intended as compile target. A function definition consists of a name, a parameter list and the body, containing calls to other functions or build-in operators. The following example defines a function  $f$  that returns the sum of both arguments  $x$  and  $y$ :

```
func f(x,y) = add_i32(x,y);
```

It is also possible to declare a constant variable the global scope using the `const` keyword:

```
const PI = 3.14159265359;
```

Within a function, immutable local variables are supported to store temporary values:

```
var temp = 1;
```

There is also an `if/else` operator to select between two alternative values, used in the following example to return the minimum of  $x$  and  $y$  (`clt_i32` = compare less than):

```
func min(x,y) = if (clt_i32(x,y)) x else y;
```

In addition, tuples are created using the following syntax:

```
var tuple = (1,2,3);
```

Data parallelism is modeled using the FOR expression (Section 2.4.1), which takes a loop variable, a range and a step size. In the following example, the loop is dynamically expanded into 256 writes of zero with a stride of 4 byte:

```
for (addr, 0, 1024, 4)
{
    write(addr, 0);
}
```

The LET statement (Section 2.4.2) evaluates the inner expression under the assumption of the given assignment and redefines a constant for a particular scope. Thus, the function  $g$  and  $h$  are evaluated to  $g(1) = 2$  and  $h(1) = 3$  in the following example:

```
const c = 1;
func f(x) = add_i32(x, c);
func g(x) = let (c = 1) f(x) // g(1) = 2
func h(x) = let (c = 2) f(x) // h(1) = 3
```

The following example shows a part of a more complex ray-tracer (Section 5.4), which calculates the intersection between a ray  $r(t) := s + r \cdot t$  and a fixed ground plane. It returns an intersection tuple containing a function  $f$ , which describes the color of the material, the position on the ray  $t$ , the geometry normal, and further parameters  $c$ . If the plane is not hit, a reference to the background is returned instead.

```
func intersection(f, t, n[3], c[3]) = { f, t, n, c };
func background(v[3],n[3],c[3]) = 0; // background is always black
func plane_color(v[3],n[3], c[3]) = { ... compute color of plane ... }

func intersect_plane(s[3], r[3]) =
{
    var t = neg_f32(div_f32(add_f32(s[1], -16.0), r[1]));

    if (cgt_f32(t, 0.0))
    {
        intersection(plane_color, t, {0.0, -4.0, 0.0}, {1.0, 1.0, 1.0});
    }
    else
    {
        intersection(background, 100000.0, {0.0, 0.0, 0.0}, {1.0, 1.0, 1.0});
    }
};
```

### 3.1.4 Discussion

The functional language describes the basic elements of the stream grammar (Section 2) in a textual form. It can be either used for the low-level programming of a stream rewriting machine, which is evaluated in Section 5.4 for graphics processing, or as an intermediate representation for more high-level compilers.

## 3.2 Task Graphs

Task graphs provide an efficient model of computation for specification, analysis and implementation of concurrent applications. This section deals with the mapping of series-parallel task graphs into rewriting rules by describing both the topology and the state as a stream of tokens. As a result, nodes can be replicated on demand and also the recursive instantiation of sub-graphs is supported. Hence, the presented approach is most useful for compute-intensive applications that must adapt to frequently varying and unpredictable workload at runtime. This section is based on the papers [84] and [87].

### 3.2.1 Introduction

Due to physical restrictions and power consumption, concurrency and parallelism are currently the most effective and in the long term the best way of accelerating a program. In this context, task graphs offer an abstract model of computation which reduces the complexity and development time of parallel applications. Instead of designing explicitly for a specific multi-core architecture or hardware platform, an abstract task graph [49] retains its concurrency and can be retargeted automatically for different environments [50] [105] [106].

A task graph is a directed acyclic graph (DAG) that contains the tasks as nodes and their dependencies as edges. An edge between two nodes specifies that the task associated with the first node must be completed before the second one can be started. Hence, the edges define a partial execution order, so that independent tasks may be evaluated in parallel on different processing units. A central challenge is to provide a mapping between tasks and processing units for each time-step that is optimal in terms of resource usage and correct with respect to the dependency relation. In addition, there are often also several other constraints on communication costs and execution time to fulfill.

In general, there are offline scheduling algorithms [107] [51], which calculate a static schedule during a preprocessing step [108], and online algorithms that generate the schedule dynamically at runtime [7]. Both types have their advantages and drawbacks for a certain kind of application. A static schedule usually requires less computationally overhead at runtime and leads to a more predictable behavior, which might be important for embedded systems with hard real-time requirements. On the other hand, an online scheduling algorithm can react to varying workloads at runtime while an inappropriate static schedule may either waste resources or processing time.

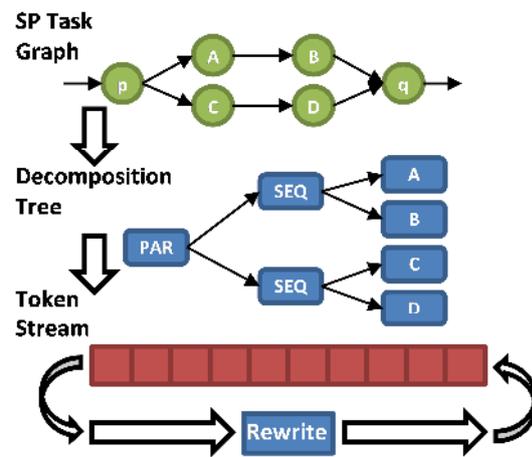


Figure 40. Stream rewriting of a task graph

This thesis presents a method for online scheduling of *series-parallel* (SP) task graphs [39], using the execution model shown in Figure 40. Contrary to arbitrary task graphs, an SP graph has a regular structure and consists of nested fork-join constructs [109]. Most important for stream rewriting, an SP graph can be decomposed into a tree of parallel and sequential computations [110]. The resulting decomposition tree (DT) is an isomorphic representation of the SP task graph but, unlike the original graph, it can be stored uniquely in preorder or post-order form. Hence, the decomposition tree can be used to serialize the graph into a stream of parallel and sequential tasks, which is not possible for graphs with arbitrary topologies, so that this restriction to series-parallel graphs is a reasonable trade-off. In addition, a SP compatible sub-graph corresponds to a compact sub-range of the stream. Thus, it is possible to expand the graph dynamically by inserting the corresponding tokens. Hence, the creation of new tasks is lightweight and does not require explicit book-keeping. This capability is especially valuable for algorithms with dynamically generated workloads like nested data-parallelism and recursive branch-and-bound techniques.

### 3.2.2 Related Work

There already exist several different solutions for online scheduling of dynamic task graphs: A *conditional task graph* [111] has a static topology but individual edges can be enabled or disabled at runtime by the use of predicates. The system in [112] generates separate schedules for each alternative path, which are eventually merged. It combines advantages from static and dynamic scheduling by mapping mutual exclusive tasks on the same processing unit for improved resource sharing and reduced latency. Though, stream rewriting provides a similar functionality by creating sub-graphs at runtime.

The *parametric task graph* [113] [114] is different concept but pursues similar goals. Large task graphs are stored in symbolic form and expanded at runtime in order to save memory and reduce scheduling time. Stream rewriting supports the parametric expansion of the task graphs but do use local pattern matching instead of a central scheduler. By streaming tasks into external memory, it is also possible to handle a large amount of instances, even when not using a symbolic form.

A *series-parallel* (SP) task graph is constructed recursively using serial and parallel composition, which corresponds to the fork-join execution model at the application-level.

The restriction to the subset of series-parallel graphs allows optimizing response time and throughput of a schedule in polynomial time [115]. For stream rewriting, SP graphs have been chosen because they can be represented as a token stream and modified using rewriting rules to enable online scheduling of dynamic SP task graphs.

The stream rewriting machine is a subset and shares some similarities with a *process algebra*, a different modelling technique for parallel behavior [116]. However, the SRM trades the support of general tasks graphs and global events for the ability distribute the execution arbitrary on many-cores architectures.

Although working on synchronous data-flow graphs (SDF), the *out-of-order execution* approach in [56] also deals with the problem of mapping tasks on a multi-core system. Though, the concept of a central buffer station for data-exchange with explicit thread management is quite different from scheduling via streaming and pattern matching.

Also related, the execution model of the *queue machine* evaluates an acyclic data flow graph by iteratively modifying a circulating stream [67]. But unlike the SRM, which embeds control information into the stream, their concept is based on data tokens. As a result, the queue machine does neither support dynamic flow control nor recursions.

In addition, there are approaches which try to reduce the amount of dynamic scheduling by using static analysis [117]. By looking into the implementation of a task, production and consumption rates can be often determined, so that it can be considered synchronously. As a result, at least these parts can be scheduled using quasi-static scheduling. Another approach, which tries to decrease dynamic data-flow, is the introduction of separate modes, which define synchronous data rates for each actor. While modes can be switched dynamically, each mode has its own static schedule [118].

### 3.2.3 Series Parallel Graphs

The initial model of the application is a task graph  $G := (V, E)$  with vertices  $V := \{v_1, \dots, v_n\}$  and edges  $E \subseteq V \times V$ . In addition, the weights  $w: E \rightarrow \mathbb{N}$  for each edge  $e := (x, y) \in E$  describe the number of parameters transmitted from a predecessor task  $x$  to its successor  $y$ . In consistence with the abstract definition of stream rewriting (Section 2.1), each task  $v_i$  of this graph will be associated with a function  $f_i: \mathbb{Z}^{a_i} \rightarrow \mathbb{Z}^{b_i}$  that maps  $a_i$  inputs to  $b_i$  output values. Since each node  $v_i$  is associated with the function  $f_i$ , the sum of incoming and outgoing edge weights must match  $a_i$  and  $b_i$ :

$$\sum_{(x,v_i) \in E} w(x, v_i) = a_i \text{ and } \sum_{(v_i,y) \in E} w(v_i, y) = b_i \quad (46)$$

Since the focus has been put on the class of *series-parallel* (SP) task graphs [109] with a regular structure of nested fork-join constructs, the graph can be uniquely serialized into a stream. Most important, a sub-graph always corresponds to a compact sub-stream, so that all rewriting rules are local operations working on a limited number of tokens. Formally, an SP graph  $G := (V, E, I, O)$  with tasks  $V$ , edges  $E$ , inputs  $I$  and outputs  $O$  can be constructed recursively by the rules shown in Figure 41:

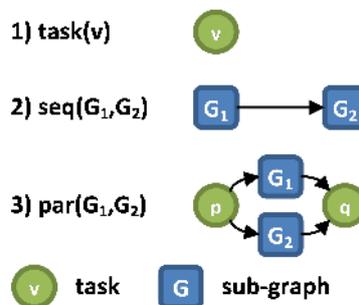


Figure 41. Rules for recursive construction of serial-parallel task graphs.

1.) The SP graph of a single task  $v$  is specified as:

$$task(v) := (\{v\}, \emptyset, \{v\}, \{v\}) \quad (47)$$

2.) Two SP graphs  $G_1$  and  $G_2$  are sequentially composed by inserting edges from  $O_1$  to  $I_2$ :

$$seq(G_1, G_2) := (V_1 \cup V_2, E_1 \cup E_2 \cup (O_1 \times I_2), I_1, O_2) \quad (48)$$

with  $G_1 := (V_1, E_1, I_1, O_1)$  and  $G_2 := (V_2, E_2, I_2, O_2)$

3.) Similarly, parallel composition is defined by the union:

$$par(G_1, G_2) := (V_1 \cup V_2, E_1 \cup E_2, I_1 \cup I_2, O_1 \cup O_2) \quad (42)$$

The conditions  $I_1 \cap I_2 \neq \emptyset$  and  $O_1 \cap O_2 \neq \emptyset$  are explicitly allowed but the constraints  $I_1 \cap O_2 = \emptyset$  and  $I_2 \cap O_1 = \emptyset$  must be true. If both inputs share a common task  $p \in I_1 \cap I_2$ , it is used as a synchronization point. Similar, there might be also one or multiple shared output nodes  $q \in O_1 \cap O_2$ .

Due to this construction, each SP graph can be described as a tree of leaf tasks, parallel, and sequential computations denoted by  $task$ ,  $par$ , and  $seq$  expressions. It is called the *decomposition tree* (DT) of a series-parallel graph  $G$  and retains the original the dependencies between tasks. For example, Figure 42 shows an SP graph and the corresponding decomposition tree, which specified by the term  $par(seq(task(v_1), task(v_2)), par(task(v_3), seq(task(v_4), task(v_5))))$ .

For a given SP graph, the decomposition tree can be derived in  $O(\log n)$  using the parallel algorithm described in [109]. However, the decomposition is not unique, so that multiple equivalent decomposition trees might correspond to a single SP graph. For example, the sequence  $v_1, v_2, v_3, v_4$  of tasks, shown in left Figure 43, can be described by the following decomposition trees:

$$DT_1 := seq(seq(seq(task(v_1), task(v_2)), task(v_3)), task(v_4))$$

$$DT_2 := seq(seq(task(v_1), task(v_2)), seq(task(v_3), task(v_4)))$$

$$DT_3 := seq(task(v_1), seq(task(v_2), seq(task(v_3), task(v_4))))$$

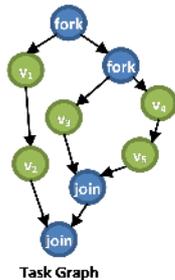


Figure 42. Task graph and decomposition tree (DT).

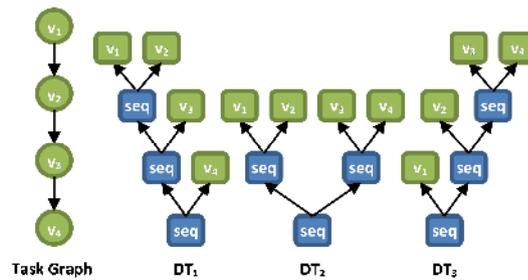


Figure 43. Three equivalent DTs of the same graph.

The choice of a particular tree is implementation dependent but in general, the DT can be characterized according to its depth and the number of temporary nodes, which are required during the rewriting process. In particular, the depth of the DT represents the minimum number of iterations until a fixed-point can be reached and the number of temporary nodes corresponds to the length of the intermediate stream. As a result, there is a tradeoff between the execution time, measured in iterations, and the storage requirements. In general, a fewer number of temporary nodes leads to a smaller stream and a more shallow tree reaches the fixed-point in less iterations.

In this example,  $DT_1$  requires the expansion of the whole tree before the leaf  $v_1$  can be rewritten. Hence, the three sequential nodes are first expanded and then successively reduced while the results of  $v_1, \dots, v_4$  become available. Hence, the stream must provide enough capacity to store all four tasks  $v_1, \dots, v_4$ , which is shown in the following sequence:

$$\begin{aligned} s_0 &:= \langle seq, CALL \rangle \\ s_1 &:= \langle seq, CALL, v_4, CALL \rangle \\ s_2 &:= \langle seq, CALL, v_3, CALL, v_4, CALL \rangle \\ s_3 &:= \langle v_1, CALL, v_2, CALL, v_3, CALL, v_4, CALL \rangle \\ s_4 &:= \langle \dots, v_2, CALL, v_3, CALL, v_4, CALL \rangle \end{aligned}$$

If the same tree is rotated into  $DT_3$ , the task  $v_1$  is emitted in the first iteration by the sequential root node. It can execute in iteration  $s_1 \rightarrow s_2$  and produce its successor:

$$\begin{aligned} s_0 &:= \langle seq, CALL \rangle \\ s_1 &:= \langle seq, CALL, v_1, CALL \rangle \\ s_2 &:= \langle seq, CALL, v_2, CALL \rangle \\ s_3 &:= \langle v_3, CALL, v_4, CALL \rangle \end{aligned}$$

Hence, there is only one leaf task on the stream at any time. Although  $DT_2$  has the lowest depth, it also takes four iterations to finish due to data dependencies, so that in this case,  $DT_3$  is the most efficient solution. Although important, the optimization of the DT is left for future work.

Most important for stream rewriting, the decomposition tree represents a scheme to serialize the task graph on the stream, so that both scheduling and modifications of the topology can be expressed in terms of rewriting rules. For this purpose, the restriction to series-parallel topologies is necessary to ensure that the graph can be transformed into a tree. In particular, there is at most one directed path between two nodes in a tree, so that regardless of the serialization scheme, each node is visited at most once. Especially important for parallel rewriting, each task is located at a single position in the stream and each sub-graph is associated with a continuous region. Hence, local modifications of the graph correspond to local modifications of the stream.

On the other hand, an arbitrary graph may contain several paths towards a single shared node, which complicates the storage format. For instance, it is possible to store the shared node only at the first appearance and embed references into the stream, which has been proposed for the rewriting of binary decision diagrams (BDD) in [119]. However, using references would require a global dictionary of nodes and an appropriate naming scheme, which directly conflicts with the approach of local rewriting rules. In particular, the creation and synchronization of threads would always require global access, while these operations can be currently performed local on a stream fragment. Hence, stream rewriting omits arbitrary topologies in favor of improved concurrency. If arbitrary connections are necessary for a particular use case, they can be also implemented manually as queues in memory. In addition, for many use cases like graphics processing, an SP topology already offers a significant advantage in comparison to current rendering pipelines (Section 5).

### 3.2.4 Functional Model

The SP task graph can be translated into an equivalent decomposition tree (DT) consisting of nested *task*, *seq* or *par* operations. Before specifying rewriting rules, the tree will be first transformed into a functional program.

While the leaf nodes  $task(v_i)$  with  $i = 1 \dots n$  can be immediately associated with functions  $f_i$ , the remaining parallel (*par*) and sequential compositions (*seq*) must be also converted into a functional form. Let  $n = |V|$  be the number of tasks,  $m$  be the number of sequential nodes and  $k$  be the number parallel nodes in the DT. The functional program  $F$  is given as a finite set of functions:

$$F := \left\{ \underbrace{f_1, \dots, f_n}_{n \times task}, \underbrace{f_{n+1}, \dots, f_{n+m}}_{m \times seq}, \underbrace{f_{n+m+1}, \dots, f_{n+m+k}}_{k \times par} \right\} \quad (49)$$

The execution model of stream rewriting is self-timed, so that a sequential ordering must be enforced using data dependencies in case of *seq* and a barrier at the end of a *par* operation. For this purpose, a strict evaluation order is assumed, which guarantees that all arguments of a function must be available before execution is started:

Let  $\varphi$  denote the function of a *task*, *seq* or *par* operation, which will be recursively defined as follows:

#### 1) Leaf Task

A leaf node  $task(v_i)$  with task  $v_i$  is mapped directly to the associated function:

$$\varphi(task(v_i)) := f_i \quad (50)$$

## 2) Sequence

Similarly, a sequential node  $seq(G_1, G_2)$ , with two sub-graphs  $G_1$  and  $G_2$ , corresponds to a functional composition. Due to data dependencies, the function of the second sub-graph  $\varphi(G_2)$  will be evaluated after the first sub-graph  $\varphi(G_1)$ :

$$\varphi(seq(G_1, G_2)) := \varphi(G_2) \circ \varphi(G_1) \quad (51)$$

## 3) Parallel Composition

For the parallel composition  $par(G_1, G_2)$ , two functions  $\lambda_1: \mathbb{Z}^{c_1} \rightarrow \mathbb{Z}^{d_1}$  and  $\lambda_2: \mathbb{Z}^{c_2} \rightarrow \mathbb{Z}^{d_2}$  for both sub-graphs  $G_1$  and  $G_2$  are defined as:

$$\lambda_1 := \varphi(G_1) \text{ and } \lambda_2 := \varphi(G_2) \quad (52)$$

The input vector of the  $par$  node  $x := (x_1, \dots, x_{c_1+c_2})$  contains arguments for  $\lambda_1$  and  $\lambda_2$ , which can be evaluated in parallel to produce the vector  $r := (r_1, \dots, r_{d_1+d_2})$  according to:

$$\begin{aligned} (r_1, \dots, r_{d_1}) &:= \lambda_1(x_1, \dots, x_{c_1}) \\ (r_{d_1+1}, \dots, r_{d_1+d_2}) &:= \lambda_2(x_{c_1+1}, \dots, x_{c_1+c_2}) \end{aligned} \quad (53)$$

Finally, the strict evaluation order synchronizes both data paths via the identity function  $id$ :

$$\varphi(par(G_1, G_2)) := (x_1, \dots, x_{c_1+c_2}) \mapsto id(r_1, \dots, r_{d_1+d_2}) \quad (54)$$

Here, the usage of  $id$  ensures that all  $r_1, \dots, r_{d_1+d_2}$  are evaluated to guarantee that both  $\lambda_1$  and  $\lambda_2$  are completed before the parallel process finished. Hence, the end of the parallel composition enforces a barrier for the computations of both sub-graphs. Important to note, without the strict evaluation order of stream rewriting, this assumption cannot be made. As a result, the application, initially given as an SP task graph, is translated into the functional program  $F$  and can be evaluated via stream rewriting.

### 3.2.5 Rewriting Rules

The basic model of stream rewriting (Section 2.1.1) consists of nested CALL expressions and is therefore already sufficient to represent the call tree of the functional program  $F$ . A more detailed functional language can be constructed using additional tokens. Also important in this context, functions cannot only emit literals but also CALL token themselves to represent pending invocations. This technique is called continuation-passing style [90] and maps natively to the execution model. For instance, a single leaf task  $\varphi(task(v_i)) := f_i$  is mapped trivially to the following sub-stream:

$$\langle \dots, i, CALL, \dots \rangle \quad (55)$$

Likewise, the sequence of tasks  $v_i$  and  $v_j$  is translated into with the functional composition of the corresponding functions can be described by the pattern:

$$\langle \dots, i, CALL, j, CALL, \dots \rangle \quad (56)$$

Here, the outputs of  $v_i$  are supplied as arguments to  $v_j$  and the strict evaluation order ensures that  $v_j$  cannot start execution until  $v_i$  is finished. For example, let the two tasks  $f_1$  and  $f_2$  be defined as  $f_1(x) = x + 1$  and  $f_2(x) = 2x$ . According to the previous definition, the sequential composition  $seq_{f_1, f_2}$  is given as the new task  $f_3$  with:

$$f_3 := seq_{f_1, f_2} = \langle 1, CALL, 2, CALL \rangle \quad (57)$$

The invocation of  $seq_{f_1, f_2}$  with argument  $x = 1$  leads to the following sequence of streams:

$$\begin{aligned} s_0 &:= \langle 1, 3, CALL \rangle \\ s_1 &:= \langle 1, 1, CALL, 2, CALL \rangle \\ s_2 &:= \langle 2, 2, CALL \rangle \\ s_3 &:= \langle 4 \rangle \end{aligned} \quad (58)$$

Similarly, the parallel execution of tasks  $v_i$  and  $v_j$  with  $c_i$  and  $c_j$  arguments, which are eventually joined at node  $v_q$  (Figure 41 on page 53), can be specified by the definition:

$$par_{f_i, f_j}(x_1, \dots, x_{c_i+c_j}) = \langle x_1, \dots, x_{c_i}, i, CALL, x_{c_i+1}, \dots, x_{c_i+c_j}, j, CALL, q, CALL \rangle \quad (59)$$

An example for the parallel composition with  $f_i := f_j := *$  and  $f_q := +$  is shown in Figure 14 on page 22. Here, both multiplications can be computed in parallel and their result is joined by the addition task.

### 3.2.6 Dynamic Task Graphs

For applications with varying amount of workload, static task graphs might either over- or underestimate the actual computational requirements. In addition, recursive algorithms require to start new tasks dynamically and to parallelize the execution of branches. Also, it should be possible to select different tasks at runtime similar to a conditional task graph. Currently, a task is evaluated to a tuple of integers, which are then passed to the next instance via sequential or parallel nodes. By redefining a task as  $v: \mathbb{Z} \rightarrow \Sigma^*$ , it can return any sequence of tokens which may be constants but also entire sub-graphs (Figure 45).

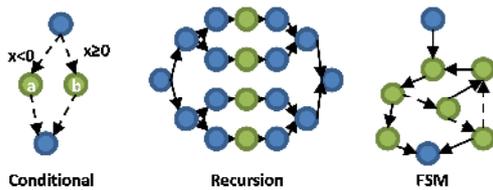


Figure 44. Dynamic task graphs.

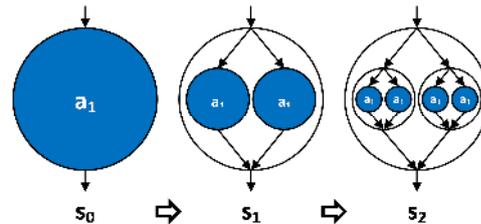


Figure 45. Dynamic expansion of sub-graphs.

Hence, the principles, used for the composition of the initial stream in Section 2, can be also applied dynamically. Although the graph is dynamic, it is still encoded on the stream and remains series-parallel at any time. However, the ability to expand tasks via stream rewriting at runtime leads to the following additional use-cases (Figure 44), which are not possible with static SP task graphs:

- **Conditional Task Graphs**

Similar to literal outputs, which are produced by emitting literals, the id of the CALL token can be calculated dynamically, too. Therefore, it becomes possible to conditionally select one or multiple successor tasks. In the following example, the task  $v$  either invokes  $a$  or  $b$  depending on the value of argument  $x$ :

$$v(x) := \begin{cases} \langle x, a, CALL \rangle & \text{if } x < 0 \\ \langle x, b, CALL \rangle & \text{else} \end{cases} \quad (60)$$

- **Recursion**

In addition, tasks can be also expanded recursively to process hierarchical data structures. Branches of identical depth are created in the same iteration and can be rewritten in parallel. For example, the task  $r$  is either expanded further or replaced by the leaf task  $stop$  depending on the value of  $x$ :

$$\begin{aligned} r(x) &:= \begin{cases} \langle x, expand, CALL \rangle & \text{if } x > 0 \\ \langle x, stop, CALL \rangle & \text{else} \end{cases} & (61) \\ expand(x) &:= \langle x - 1, r, CALL \rangle \\ stop(x) &:= \langle \rangle \end{aligned}$$

- **Finite State Machine**

As an extension of the conditional task graph, the dynamic selection of the successor task can be also used to implement finite state machines (FSM) on the stream. This technique might be in particular useful to virtualize a large and possibly unpredictable number of FSMs. In general, each state is mapped into a task  $state_i$  that invokes the next state  $j$  dynamically if the condition  $c_{ij}$  is true. The following example shows the basic scheme for implementing a FSM with  $n$  states and  $n^2$  transitions:

$$\begin{aligned} state_1(x) &:= \begin{cases} \langle x, state_1, CALL \rangle & \text{if } c_{11} \\ \dots & \dots \\ \langle x, state_n, CALL \rangle & \text{if } c_{1n} \end{cases} & (62) \\ \dots & \\ state_n(x) &:= \begin{cases} \langle x, state_1, CALL \rangle & \text{if } c_{n1} \\ \dots & \dots \\ \langle x, state_n, CALL \rangle & \text{if } c_{nn} \end{cases} \end{aligned}$$

### 3.2.7 Conclusion

In this section, a formal approach for the translation of static and dynamic task graphs into the execution model of stream rewriting has been presented, so that the SRM is appropriate for running a large variety of different applications. In particular, also the support for dynamic task graphs requires only the basic model of stream rewriting consisting of CALL and literal tokens. Therefore, the technique, specified in this section, is compatible to all SRM implementations.

Further, the restriction to series-parallel task graphs represents a tradeoff, which enables dynamic scheduling without central task management. It actually complies with the basic principles of stream rewriting, which aims to improve the processing of unpredictable tasks at runtime. Although this decision excludes several task graphs, which might be trivial to handle using a traditional scheduler, the dynamic and recursive expansion of nodes at runtime facilitates scenarios that cannot be described using a static approach. Several examples are evaluated using three different SRM architectures in Section 4.

## 3.3 C Source Code

In this section, the generation of rewriting rules from a C-like language is specified. It will be used to construct a tool-chain for the high-level synthesis of recursive functions, which is based on a hardware implementation of the stream rewriting machine (Section 6).

### 3.3.1 Introduction

The mapping of a functional language to the execution model of stream rewriting is straightforward because each CALL expression already represents the invocation of a function (Section 3.1). Hence, an imperative language like C can be compiled into rewriting rules by translating it into a functional form.

The two main differences between the imperative and the functional forms are the notion of state and control flow. In contrast to C, the functional language does not allow to modify the value of a local variable. Actually, local declarations are just aliases and not associated with a mutable state or a storage location. Hence, in compliance to the underlying execution model, all modifications occur through rewriting parts of the stream. As a result, the modification of a local variable can be only implemented by passing the new value to another CALL expression.

Immediately linked to the concept of a mutable state, the idea of control flow provides an implicit order for the operations of the program. However, due to the absence of side-effects, the functional model does not need to include an absolute execution order. Instead, the evaluation of expressions is implicitly controlled by data dependencies. In addition, the control flow of the program can be also seen as another mutable state variable. Hence, the translation of the C source code requires to analyze both data and control flow of the program and to map modifications into rewriting operations.

In particular, each basic block of the program can be modeled as the state of a FSM, while each state is translated into a separate rewriting rule according to Equation (62). Similarly, active local variables are passed as parameters. In addition, rewriting rules can be also further sub-divided into more fine-grained operations to fit into the primitives of the target

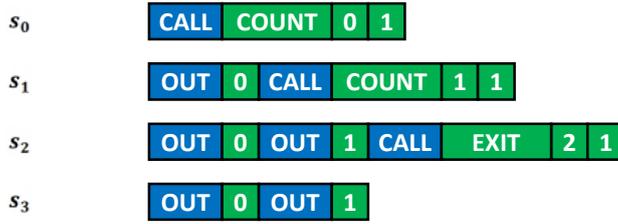


Figure 46. Recursive loop counting from 0 to 1

platform. For example, for hardware synthesis, the compiler automatically derives a set of primitive operations, which are directly implemented in hardware and adapts the rewriting functions accordingly (Section 6.3.2).

The individual steps of the compilation process are described in the following sections. Here, the preorder format of the stream grammar is used in combination with extension for ordered-writes (Section 2.5.1), which adds the OUT token.

### 3.3.2 Functions and Recursion

Function calls are the most basic operation in the SRM and can be represented directly using the CALL expressions. Due to the pattern matching, multiple nested frames persist on the stream until their arguments are computed, so that all types of recursion are supported. Similar to branching, function pointers are implemented by choosing the identifier of the CALL token dynamically.

The example in Figure 46 shows a recursive counter, which outputs the numbers from  $i$  to  $n$ . In order to improve readability, the function identifier  $i \in \mathbb{Z}$  is replaced by the name of the function with COUNT=1 and EXIT=2. Hence, the corresponding SRM using the post-order format is given by:

$$F := \{f_{COUNT}, f_{EXIT}\} \quad (63)$$

$$\begin{aligned} f_{COUNT}: \mathbb{Z}^2 &\rightarrow \Sigma^4 \\ (min, max) &\mapsto \langle OUT, min, CALL, x, min + 1, max \rangle \end{aligned} \quad (64)$$

$$\text{with } x := \begin{cases} (CALL, COUNT) & x < c \\ (CALL, EXIT) & x \geq c \end{cases}$$

The function EXIT is called at the end of the loop and removes both arguments from the stream to terminate the thread:

$$\begin{aligned} f_{EXIT}: \mathbb{Z}^2 &\rightarrow \Sigma^0 \\ (min, max) &\mapsto \langle \rangle \end{aligned} \quad (65)$$

These rewriting rules can be described by the following C program, which consists solely of function calls and arithmetic expressions. In particular, the loop counter is passed as an argument to the recursive call instead being stored in a local variable, which is shown by the following source code on the next page:

```

void exit(int min, int max);

void count(int min, int max)
{
    out(min);
    void (*next)(int, int) = (min < max) ? count : exit;
    next(min+1, max); // modify the counter for the next iteration
}

```

Hence, the next task is to bring an arbitrary C program into this format by removing control flow and assignments to local variables. However, it should be noted that the rewriting system does not define an explicit order of evaluation, so that all calls are asynchronous by default. The actual order is determined by data dependencies, so that inner calls are evaluated first. Synchronous calls and barriers are special cases that can be implemented by introducing artificial dependencies, which has been shown for task graphs in Section 3.2.

### 3.3.3 Branching and Loops

The following C code shows a more high-level description of the counter example from the previous section and the corresponding control flow graph is illustrated in Figure 47.

```

void count(int min, int max)
{
    int i = min
    do
    {
        out(i);
        i = i + 1;
    } while (i < max);
}

```

First, each function is converted into basic blocks by translating branches and loops into conditional jumps and labels. According to the control flow graph, there are three basic blocks in this example named  $Block_1$  to  $Block_3$ . The function starts in  $Block_1$  which directly calls the body of the loop in  $Block_2$ . Similar to the previous example, the current value of the loop counter  $i$  is printed. Likewise, the dependent branch at the end is implemented by selecting between the two different successor blocks  $Block_2$  or  $Block_3$ .

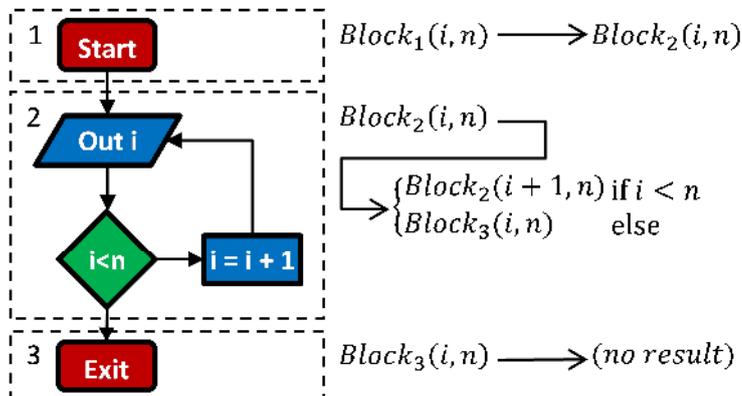


Figure 47. Mapping of flow control into rules.

The last block  $Block_3$  exits the program and therefore does not call any further functions. A verbose C representation after this intermediate step corresponds to the following code:

```
void count(int min, int max)
{
  block1:
    i = min;
    goto block2;

  block2:
    out(i);
    i = i + 1;
    if (i < max)
      goto block2;
    goto block3:
  block3:
}
```

As a result, each basic block is terminated by at least one unconditional jump and one or multiple conditional branches. In order to convert each of these blocks into a rewriting function, the next step eliminates local variables and parameters. For this purpose, the code is brought into a single static assignment format (SSA), where each variable is written exactly once. Therefore, each assignment generates a new variable to track the most recent value. For example, the assignment  $i=i+1$  of  $i$  is stored in the new variable  $i\_1$ :

```
void block1(int min, int max) {
  block2(min, max);
}

void block2(int i, int max) {
  out(i);
  int i_1 = i + 1;
  void (*next)(int, int) = (i < max) ? block2 : block3;
  next(i_1, max);
}

void block3(int i, int max){ }
```

Active local variables are added to the signature of each block. For instance,  $Block_2$  reads and writes the variable  $i$ , so that it has to be included into the parameter list. As a consequence, the predecessors of  $Block_2$  ( $= \{Block_1, Block_2\}$ ) must supply a value for  $i$  when branching to  $Block_2$ . In addition, also the outputs of a block are merged, so that the argument list of all successor blocks is equivalent and the branch at the end can switch between two function pointers. Though, arrays of local variables are not supported since they hinder the data flow analysis and complicate the transformation into the SSA form. Finally, the C program consists exclusively of declarations and function calls. Thus, the code can be directly converted into the functional language by replacing arithmetic operations with build-in macros. In contrast to the initial example, there are three rules but  $Block_1$  and  $Block_2$  are equivalent and could be merged. Also, it should be noted that the conditional  $if/else$  is actually an expression and the return value of  $Block_2$ .

The following functional program represents the result of the compilation process and can be evaluated on the stream rewriting machine according to Section 3.1:

```
func block1(min, max) = block2(min, max);

func block2(i, max) = {
  out(i);
  var i_2 = add_i32(i,1);
  if (c1t_i32(i, max))
    block2(i_2,max)
  else
    block3(i_2,max);
}

func block3(i, max) = {};
```

As consequence, a restricted subset of C can be compiled into the abstract model of the stream rewriting machine and therefore profits from the dynamic parallelism. In order to execute the program on an actual implementation, the functional language must be subsequently translated into a platform dependent format. For example, Section 6 describes a high-level synthesis tool for recursive functions, which performs further optimizations and finally outputs RTL code. Other implementations of the SRM like the general purpose many-core system (Section 4) or the graphics processors (Section 5) have not been evaluated using this approach. However, targeting these architectures would require a similar technique in order to generate microcode for the processor (Section 5.5) or invoke a software library for stream rewriting (Section 4.2).

## 4 Multi-Core Architectures

When designing the software and hardware architecture of many-core systems with hundreds of processors on a single chip, a central problem is the scheduling and binding of work-items to execution units. This section contains a novel synthesis flow for applications with highly dynamic and unpredictable behavior, which is based on the concept of parallel stream rewriting. Complex examples, evaluated using an FPGA prototype, show the effectiveness of stream rewriting architectures. This section is based on [84] and [87].

### 4.1 Introduction

Multi and many-core systems offer numerous benefits like reduced energy consumption and latency, as well as improved throughput for both high-performance and low-power applications. Scalability is achieved by parallelization instead of high clock rates and can therefore overcome

technological limitations like thermal heat or power issues [1]. While functional units and also processor cores can be replicated at the expense of increased area, the utilization of the additional resources remains a more fundamental problem [12]. Hence, beside the design of the actual hardware architecture, also the programming raises several challenges. In particular, the binding and scheduling of tasks to processor cores as well as the efficient communication and synchronization at the system level are not yet solved in general [7]. Hence, task mapping is often a trade-off between an optimal solution and application specific heuristics [13]. Conventionally, parallelism is exploited by partitioning an application into tasks that can run on different processor cores and communicate via shared memory or message passing. In particular, the binding of tasks to processor cores can be either calculated as a preprocessing step [51] or dynamically at runtime.

Usually, the set of tasks contains interdependencies, described as directed graph [49], which must be considered for scheduling and binding the application on the target architecture [50]. For a static application model [120], specified as a graph of task nodes and its dependencies, an optimal schedule can be pre-computed at design time [108]. However, in case of embedded systems, which are interacting with dynamic environments, often an adaptive mapping [37] is required to account for varying workloads.

Figure 48 shows the design flow for many-core systems based on stream rewriting. The application is first modelled as a task graph and translated into a set of rewriting rules and an initial stream, which has been discussed in Section 3.2. In this section, the hardware and software architectures for a stream rewriting machines are designed and evaluated. Since the system must be able to handle a large and unpredictable number of dynamically created tasks, the majority of static optimizations, which are based on an extensive knowledge of the behavior at design time, cannot be applied in this case.

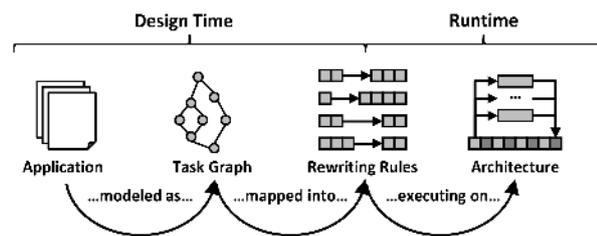


Figure 48. Design flow for many-core architecture

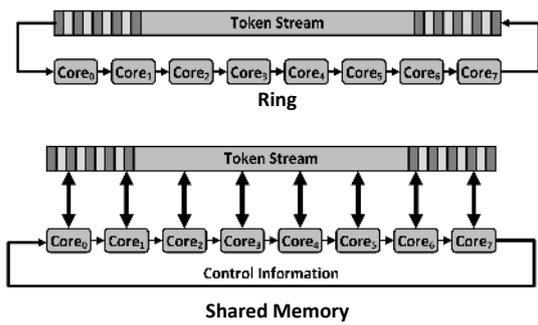


Figure 49. SRM using ring or shared memory.

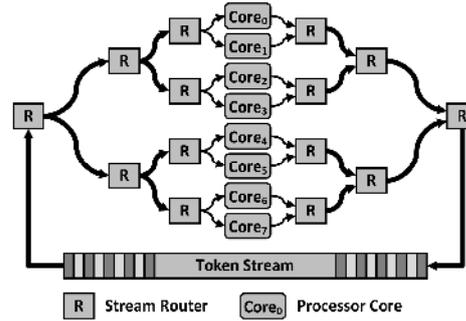


Figure 50. Stream rewriting network with routers (R).

In order to analyze the consequences of different design decisions, three separate architecture templates for many-core system are evaluated. The first design is based on the ring architecture (Figure 49 top), which has been already shown in Section 2.2. The rewriting cores are implemented as general purpose processors rewriting cores and connected via queues storing the token stream.

The second design (Figure 49 bottom) is also based on general purpose processors, but in this case, the token stream is stored in an external shared memory and a small circular communication channel is used only for synchronization. The advantage of the shared memory design is simplicity, generality and portability since it does not require custom hardware components. Especially important for the wide variety of embedded SoCs, it can be ported to different platforms and performs the stream rewriting entirely in software. Despite its flexibility, the shared memory design clearly exhibits a potential bottleneck for the performance of the system. Though, the experiments show scalability for a moderate number of cores and reasonable coarse grained tasks.

On the other hand, the third design employs the *stream rewriting network* (SRN) (Figure 50) to connect a set of general purpose processor cores. It consists of several routers (R) for task distribution and reassembly of the resulting sub-streams. Also, the pattern matching is performed in parallel by specialized hardware components, while the processors are responsible for executing the detected tasks. Therefore, this architecture combines the performance advantages of application-specific hardware with the flexibility of software on off-the-shelf general purpose processors. As a result, the experiments show the scalability of the SRN for up to 128 cores. Thus, the SRN is more likely usable for many-core architectures and represents an alternative to the classic processor arrays connected via a mesh network [11] and dataflow machines [65]. Both architectures use the post-fix format of the stream grammar (Section 2.1.3) without extensions, so that tasks are described as the pattern  $\langle CALL, i, x_1, \dots, x_n \rangle$  with  $x_1, \dots, x_n \in \mathbb{Z}$ . Tasks are implemented as the functions of a C program, whose entry point is given by the identifier  $i$ , so that the original tool-chain of the corresponding processor can be utilized.

In the next sections, the shared memory architecture and the SRN are presented in more detail, while the more simplistic ring is mainly used for comparison (Section 4.4).

## 4.2 Shared Memory Architecture

In this section, an algorithm for parallel stream rewriting using shared memory is described.

### 4.2.1 Parallel Rewriting

In the shared-memory architecture, the token stream is stored in a global memory and can be accessed by a set of  $n$  general purpose processor cores. All cores are running the same software and there is circular channel for synchronization. Hence, the stream can be partitioned into blocks and each core rewrites a specific segment (Section 2.2). A uniform splitting of the stream into blocks of the same size is illustrated in Figure 51. However, in addition to the problem of false fixed-points, which has been already resolved, a concrete implementation has to consider the following challenges:

- **Stream Modification**

The size of the stream naturally grows and shrinks during the rewriting process. Similarly, also the length of a particular segment is most likely different after a rewriting step, so that it does not fit into its previous storage location. However, inserting or deleting a token segment in the middle of a large stream is costly, involves copy operations and would require an explicit synchronization of all cores.

- **Stream Partitioning**

The partitioning of the stream into equally sized segments guarantees optimal load-balancing if the tasks are distributed uniformly. However, for recursive problems, it has been shown that the innermost and therefore executable tasks are either located at the end or the beginning of the stream (Section 2.3.3). If the block size is chosen too large, a certain core might receive all executable tasks, while the rest of the system is only copying non-matching tokens. On the other hand, a block, which is too small might contain only a too few rewriting rules to be efficient. Hence, it can be expected that the optimal block size must be adapted dynamically.

- **Memory Coherence**

Since all processors access non-overlapping regions of the stream, no direct conflict occurs. However, two neighboring tokens, which are written by distinct cores, can belong to the same cache line. When using incoherent caches with a write-back policy, it is possible that different versions of a cache line from separate cores are overwritten by each other. Hence, if the platform does not offer memory coherence, data caches must be configured as write-through and explicitly invalidated after a rewriting step.

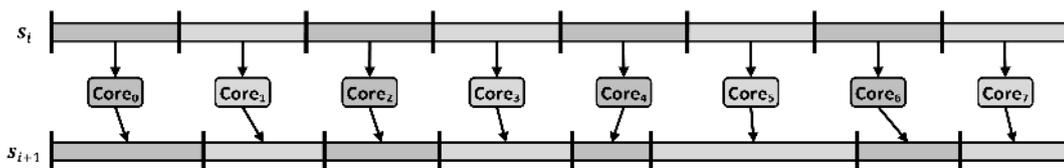


Figure 51. Partitioning and parallel rewriting of the stream.

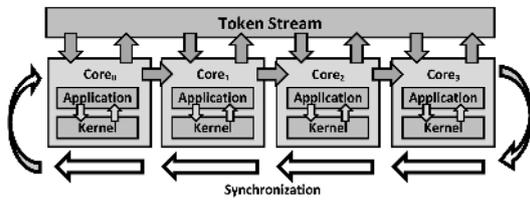


Figure 52. Software and hardware architecture

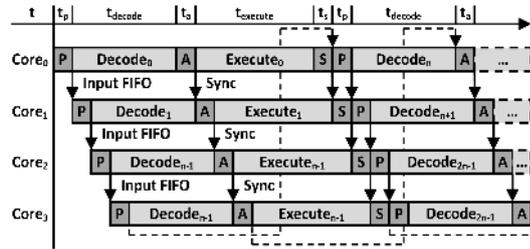


Figure 53. Phases and synchronization of the cores.

The software, running on each core, is split into the application layer, which defined the task and a kernel, which is responsible for the actual stream rewriting (Figure 52). It runs an infinite loop and implements a parallel stream rewriting algorithm consisting of the five steps *Partition(P)*, *Decode(D)*, *Allocate(A)*, *Execute(E)* and *Synchronize(S)*. If a task is detected as executable, the application is invoked to produce the resulting tokens. The software image is exactly the same on all cores in order to keep function pointers interchangeable.

The kernel receives synchronization data from the preceding processors via the communication channel, performs the required computations and forwards the results. In particular, the token stream is organized as a large FIFO in external memory and the circular communication channel is used to exchange its address range as the 4-tuple:  $(start, end, read, write) \in \mathbb{Z}^4$ . Here, *start* and *end* define the dimensions of the FIFO, while *read* and *write* specify the corresponding read and write pointers. Actually, the FIFO tuple is the only shared variable, which is simultaneously access by multiple cores, and then used to allocate non-overlapping regions on the stream that do not require protection. Hence, the tuple acts as a shared token, which owned by exactly one of the cores and passed around for synchronization. The five steps of the algorithm are illustrated in Figure 53 and can be described more detailed in the follow list:

### 1. Partition (P)

In the first step, each core receives the current FIFO tuple and reserves a certain amount of input tokens. For this purpose, the size of the stream is evaluated and the optimal block size  $b$  is determined using a heuristic. If no executable task has been found,  $b$  is increased in the next iteration until one core tries to rewrite the entire stream to handle the special case of incorrect fixed-points. The read pointer is moved forward and wrapped around if necessary, and finally, the modified FIFO is passed to the successor. This phase is performed sequentially, since each core has to wait for the FIFO tuple but due to its simplicity, it takes only a few cycles.

## 2. Decode (D)

After a core has allocated its input range, it can immediately start analyzing the corresponding token stream. It calculates the number of output tokens, which will be produced during the rewriting process, so that the output region can be allocated accordingly. Therefore, the stream is decoded using pattern matching and each task is executed in a special mode in order to retrieve the number of outputs. In contrast to the partitioning step, the decoding phase can be performed in parallel on all cores and is only limited by memory throughput.

## 3. Allocate (A)

At this point, the sub-stream of each core, the executable tasks and the number of tokens, which will be eventually produced during the rewriting process, have been determined. As shown in Figure 51, the resulting fragments must be compacted in memory to build the stream of the next iteration. Hence, the start address of the outputs generated by a particular core depends on the number of tokens written by the preceding processors.

Similar to the partitioning of the input stream, also the allocation of the output parts in the FIFO must be performed in order. Thus, in the allocate phase, each core receives the current FIFO, moves the write pointer and passes the tuple to the next processor. As a consequence, the allocation phases of different cores cannot overlap and must be serialized. However, similar to the partitioning step, the computations are inexpensive in comparison to the stream rewriting.

## 4. Execute (E)

In the execute phase, the tasks marked in the decode phase are invoked and the resulting tokens are written into the previously allocated regions of the global token stream. The rewriting can be performed in parallel but takes most likely the largest amount of time per iteration.

## 5. Synchronize (S)

Finally, the cores are synchronized in the last step to ensure that all cores have written their results before the algorithm starts again with the partitioning and decoding phases. Without the explicit synchronization, a read-after-write conflict is possible between the execute and decode phases of subsequent iterations.

The chronological order of these five phases and the synchronization points between adjacent cores are illustrated in Figure 53. The duration of the partition phase is notated as  $t_p$  and similarly, the other time spans are named accordingly as  $t_d, t_a, t_e$  and  $t_s$ . Based on their timing, two types of phases can be distinguished. First, there are the partition (P), allocate (A) and synchronization (S) phases, which are executed sequentially but always have a fixed-length:

$$t_p + t_a + t_s = \text{const} \quad (66)$$

On the other hand, the duration of the decode (D) phase depends on the block size and can be approximated by  $t_d \in O(b)$ . Similarly, if there exists a maximum delay of  $t$  for each task, the upper bound for the length of the execution phase (E) is given by  $t_e \in O(bt)$ . Hence, in order to achieve scalability, the granularity of tasks and the block size must comply with:

$$t_p + t_a + t_s \ll t_d + t_e \quad (67)$$

Further, let  $x \in R$  be the ratio between both types of phases, so that the following equation hold true:

$$(t_p + t_a + t_s) \cdot x = t_d + t_e. \quad (68)$$

In particular,  $x$  is determined by the granularity of the threads. Coarse grained threads, which perform more work per invocation, lead to a longer execution phase and in general, larger values of  $x$  increase the scalability of the system. The relation between the thread granularity and the number of cores is computed more formally in the following section. For this purpose, the total execution time  $t_i(n, x)$  of an iteration  $i$  using  $n$  cores is approximated as the sum of the parallelizable and non-parallelizable phases:

$$\begin{aligned} t_i(n, x) &:= (t_p + t_a + t_s) \cdot n + \frac{t_d + t_e}{n} & (63) \\ &= (t_p + t_a + t_s) \cdot n + \frac{(t_p + t_a + t_s) \cdot x}{n} \\ &= (t_p + t_a + t_s) \cdot \left(n + \frac{x}{n}\right) \end{aligned}$$

Hence, for a given value of  $x$ , the speedup of  $n$  cores can be computed as:

$$\tau(n) := \frac{t_i(1, x)}{t_i(n, x)} = \frac{1 + x}{n + \frac{x}{n}} \quad (69)$$

By differencing  $\tau(n)$  and solving the equation  $\tau'(n) = 0$ , a local maximum can be found at  $n = \sqrt{x}$ . As a result, for an application with granularity  $x$ , the best speedup is achieved using  $n = \sqrt{x}$  cores. For a larger system, the overhead of the non-parallelizable phases P, A and S becomes significant and for a smaller number of cores, there is still unused concurrency. The speedup for up to 16 cores and  $x \in \{4, 9, 16, 64\}$  is plotted in Figure 54. For  $x = 64$ , the maximum acceleration of  $\approx 4$  is reached with 8 cores.

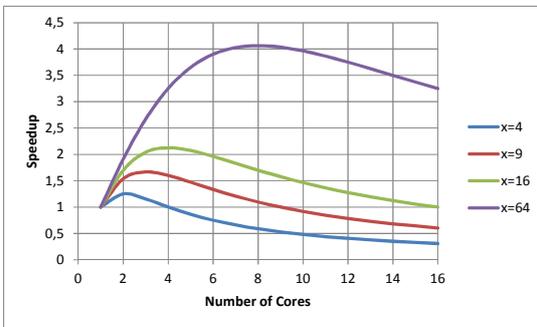


Figure 54. Speedup for different cores.

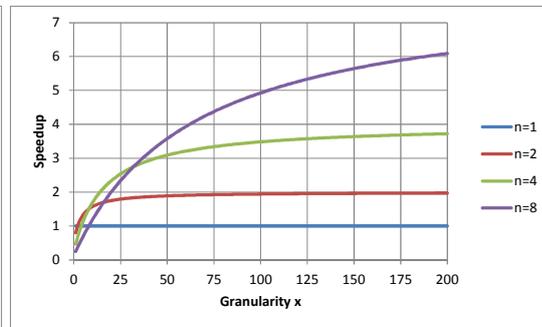


Figure 55. Speedup for thread granularity.

On the other hand, the speedup can be evaluated with respect to  $x$  in order to find the optimal thread granularity for a given system with  $n$  cores. As shown in Figure 55 for  $n \in \{1,2,4,8\}$  cores, larger values of  $x$  provide better scalability. The curves are asymptotically approaching the maximum theoretically speedup given by the limiting value  $\lim_{x \rightarrow \infty} \tau(x) = n$ , which is never reached for a concrete implementation. However, it is possible to determine the thread granularity, which is necessary to achieve a reasonable fraction  $\lambda$  of the maximum speedup. For this purpose, the equation of  $\tau$  is solved for  $x$ :

$$\tau(n) \geq \lambda \cdot n \Leftrightarrow \frac{1+x}{n+\frac{x}{n}} \geq \lambda \cdot n \Leftrightarrow x \geq \frac{\lambda n^2 - 1}{1-\lambda} \quad (70)$$

For instance, in order to reach an efficiency of  $\lambda = 90\%$  using 4 cores, the granularity must be larger than  $x \geq \frac{0.9 \cdot 4^2 - 1}{1 - 0.9} = 134$ . Similarly, for  $\lambda = 50\%$  and a system with 8 cores, the granularity must be  $x \geq \frac{0.5 \cdot 8^2 - 1}{1 - 0.5} = 62$ .

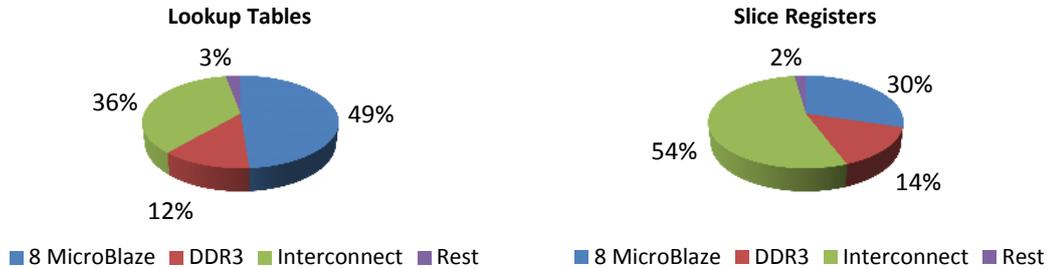


Figure 56. Resource distribution of the test system with eight cores.

#### 4.2.2 Hardware Platform

The proposed architecture has been synthesized as a SoC for the ML605 evaluation board from Xilinx, which contains a Virtex 6 VLX240T FPGA and 512MB of external DDR3 memory. The rewriting cores are implemented as MicroBlaze processors, which are configured with 16KB cache and floating-point units run at 100MHz. All processors can access the token stream in the external memory via an AXI interconnect clocked at 200MHz. The timing has been met for up to eight cores and the resource consumptions of six-input lookup tables (LUT), flip-flops (FF), block rams (BRAM) and DSP blocks (DSP48) are listed in Table 2.

Each core adds roughly 6,000 LUTs to the system but when synthesized individually, a single MicroBlaze requires  $\approx 3,500$  LUTs. Actually, the remaining 2,500 LUTs are spent on the connection between the processor and the external memory. In particular, the distribution of lookup tables and slice registers for a system with eight cores is illustrated in Figure 56. While the memory controller (DDR3) consumes 12% of the lookup tables and 14% of the occupied slice registers, the eight MicroBlaze cores make up half of the system. However, the other half must be dedicated to the AXI interconnect (green), which is responsible for routing memory request between processor cores and DDR3 controller. Hence, the global memory access of each core is relatively expensive although it offers the convenience to implement stream rewriting in software.

Hence, stream rewriting based on shared memory is most useful for existing platforms that already contain a reasonable fast memory interconnect. However, if the synthesis of custom hardware modules is possible, the *stream rewriting network* (SRN), which will be presented in Section 4.3, allows for a much larger number of cores.

Table 2. Resource usage of the shared memory test system.

Component	LUT	FF	BRAM 18K/36K	DSP48E	
<b>System Cores</b>					
1	14,874	19,446	1	29	5
2	20,309	24,795	2	41	10
4	31,494	35,463	4	65	20
6	41,695	46,117	6	89	30
8	52,737	56,772	8	113	40
<b>MicroBlaze</b>	≈ 3,557	≈ 2,814	0	4	5
<b>DDR3 Controller</b>	≈ 7,629	≈ 8,401	0	0	0

### 4.2.3 Recursive Tests

Stream rewriting allows for dynamic scheduling of recursive functions. Therefore, the performance of the system is evaluated using the following three representative recursive functions to simulate the workload of a typical branch-and-bound algorithm:

$$1.) \quad rsum(a, b) = \begin{cases} rsum\left(a, \left\lfloor \frac{a+b}{2} \right\rfloor\right) + rsum\left(\left\lceil \frac{a+b}{2} \right\rceil, b\right) & \text{if } a > b \\ a & \text{else} \end{cases} \quad (71)$$

$$2.) \quad fib(x) = \begin{cases} fib(x-2) + fib(x-1) & \text{if } x \geq 2 \\ x & \text{else} \end{cases} \quad (72)$$

$$3.) \quad ack(n, m) = \begin{cases} m + 1 & n = 0 \wedge n = 0 \\ ack(n-1, 1) & m = 0 \wedge n > 0 \\ ack(n-1, ack(n, m-1)) & m > 0 \wedge n > 0 \end{cases} \quad (73)$$

The runtime of each function and the size of the token stream per iteration are shown in Figure 57 on the next page. Since the actual computations of these functions mostly consist of a few integer operations, a delay of 1,000 cycles was inserted to emulate a coarse grained task. Both the recursive sum and the Fibonacci test show a reasonable scalability of 3.0 on a system with four cores. However, the step from four to eight cores yields only a minor speedup to 4.0-4.5. Contrary, the Ackermann function does not scale at all since mostly one thread at the end of the stream can be replaced per iteration and the rest is just copied. Further explanations can be found when looking at the size of the stream. Actually, *rsum* and *fib* expand exponentially with a peak after ≈15 iterations but *ack* (3,4) is oscillating and takes more than 10.000 iterations to converge from which the only first 200 steps are shown.

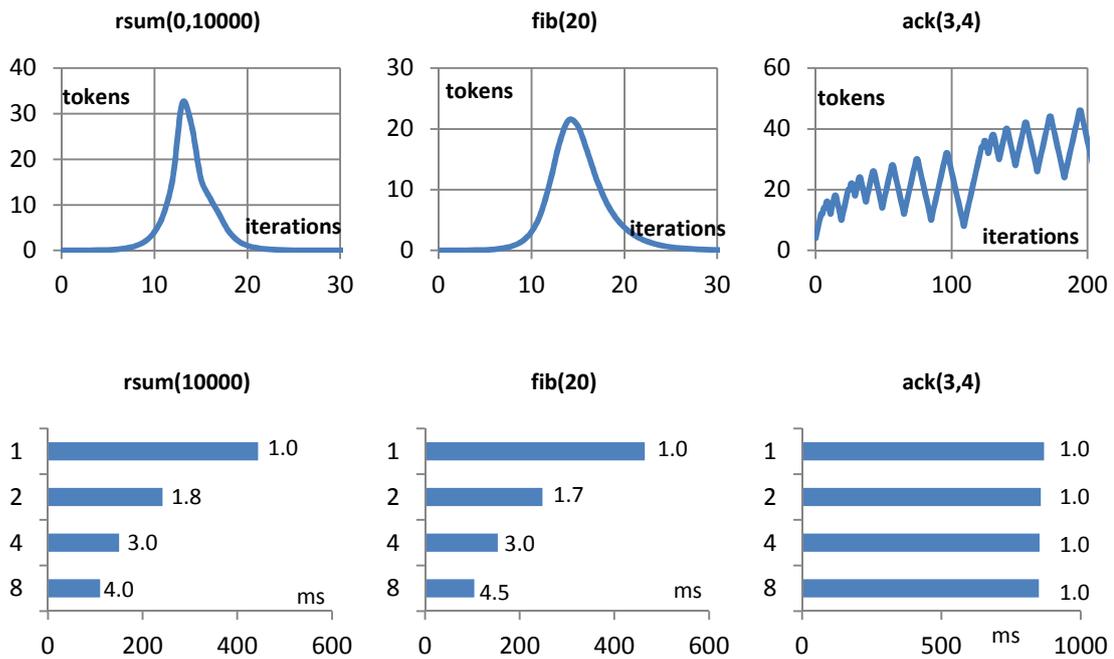


Figure 57. Performance of recursive test cases.

#### 4.2.4 Application Tests

In addition to the generic examples, the three more complex tests *Mandelbrot*, *Bezier3D* and *Raytracing*, which represents different types of concurrency, have been also evaluated. The *Mandelbrot* test represents a typical branch-and-bound algorithm that subdivides the screen (640x480) recursively into small rectangles, whose contents is then computed in parallel. In general, *Mandelbrot* is a classic data parallel problem since each pixel can be evaluated independently. However, in order to improve load-balancing, the recursive expansion method (Section 2.3.4) is utilized and iteratively enlarges the stream in parallel. *Bezier3D* implements a complex rendering pipeline for the rasterization of bezier surfaces and contains both data and pipeline parallelism. First, the surface is subdivided into quads and then into triangles, which are lit and projected on the screen. Each 2D triangle is drawn by recursively determining the set of pixels that are located inside. As an optimization, blocks outside of the triangle are hierarchically skipped.

Similarly, the *Raytracing* test draws a simple geometric scene consisting of a reflecting sphere lying on top of an infinite checkerboard. The color of each pixel is determined by several sub-tasks, which are shooting rays into the scene and calculate the nearest intersection point. Especially the screen area of the sphere is expensive to compute since the reflection requires additional rays. In contrast to a fixed assignment between pixels and cores, the reflecting rays are handled by dynamically created tasks, which are distributed across all cores.

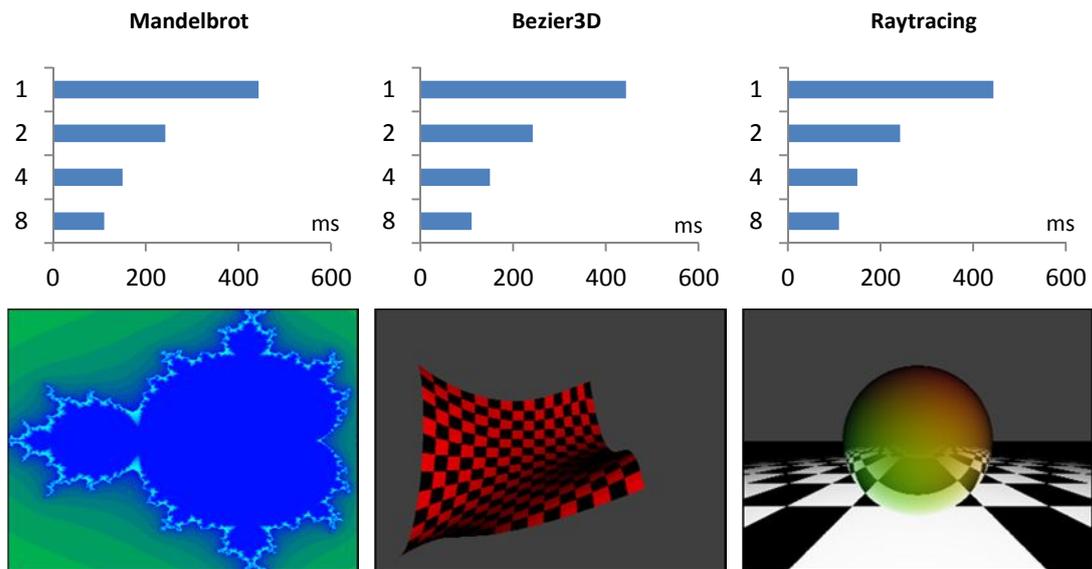


Figure 58. Performance and screenshots of test applications.

The computation time of each test case and the corresponding screenshots are shown in Figure 58. All examples indicate a scalability of the system although the speedup is stagnating at eight cores. Though, it has to be considered that all examples are scheduled dynamically and especially the *Raytracing* test utilizes more than one million threads. As a result, the generic software-based stream rewriting technique represents a viable solution for highly dynamic scheduling and binding in many-core systems.

### 4.3 Stream Rewriting Network

In contrast to the shared memory approach, the *Stream Rewriting Network* (SRN) offers potentially a larger throughput since it decodes and distributes a wide stream in parallel.

#### 4.3.1 Overview

The architecture of the SRN (Figure 50) is refined by the schematic in Figure 59. In this example, it consists of  $n = 2$  groups of  $m = 4$  processors, which is called a  $2 \times 4$  configuration. However, in general, the design can be parameterized from  $1 \times 1$  to  $8 \times 16$  cores and experimental results as well as resource usages are presented in Section 4.3.6.

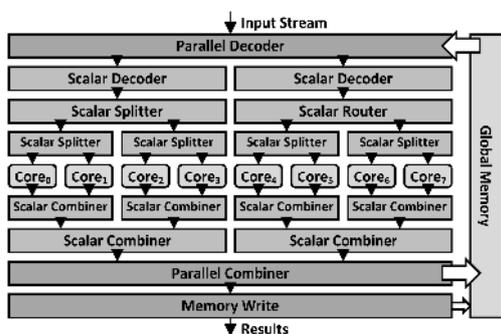


Figure 59. SRN for parallel stream rewriting (2x4).

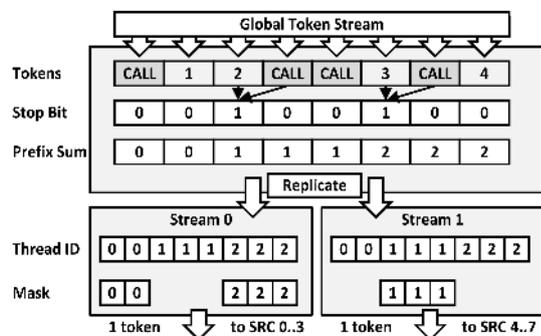


Figure 60. Parallel decoding of the token stream.

Especially for the larger configurations, the distribution of the token stream to the cores might cause a potential bottleneck, since the stream is a global resource. Hence, the interconnection network should provide a sufficient large bandwidth, so that the cores never run empty and are always fully utilized. Through, depending on the granularity of the tasks, the actual throughput of each core is most likely less than one token per cycle, so that resources of the SRN can be safely shared between processors. As a result, the cores are organized in a two-level hierarchy of parallel and scalar data paths, which provides a trade-off between area and performance.

At the top level, the global stream can transmit  $n$  tokens per cycle, where  $n$  is the number of processor groups (limited by memory throughput). In each cycle, these  $n$  tokens are processed by the *parallel decoder* and distributed to the  $n$  processor groups. It used a fast heuristic to ensure that that the tokens of a *CALL* expression are sent to the same processor group. Hence, the partitioning can vary between the case, when all  $n$  tokens are assigned to the same processor group and the other extreme, when each token is assigned to a different group. On average, it can be expected that each group receives one token per cycle, so that the further processing and exact pattern matching at the *scalar decoder* is based on streams, which transmit at most a single token per cycle.

The throughput of each core is less than one token per cycle for moderate complex tasks. Thus, the stream is distributed by the *scalar splitter* to  $m$  processors for rewriting with an average rate of  $\frac{1}{m}$  tokens per cycle. It is reassembled by the *scalar combiner*, widened again and merged by the *parallel combiner*. Finally, the resulting stream is stored as a FIFO in global memory and the *memory write* module is responsible for handling write operations. Since all parameters and results are exchanged using the token stream, the performance of the system is limited by its throughput. However, the memory is accessed in bursts of  $n$  tokens that can be split on multiply banks to ensure the required band-width. Hence, in case of a  $n \times m$  system, a task  $f_i: \mathbb{Z}^{a_i} \rightarrow \mathbb{Z}^{b_i}$  should take at least  $\frac{m}{\max(a_i, b_i)}$  cycles.

### 4.3.2 Stream Decoding

The *parallel decoder* does not need to detect every pattern but instead it is responsible for dividing a parallel stream at *nearly optimal* positions, so that the tokens of matching calls are all sent to the same group for final decoding by the *scalar decoder*. The individual steps of this algorithm are illustrated in Figure 60 on page 65 and work on  $n$  tokens in parallel. Assume that the stream is as  $s := \langle t_1, \dots, t_{|s|} \rangle$  with tokens  $t_i$ . First, possible break points at the transition between literal and *CALL* tokens are marked with a stop bit:

$$stop(i) := \begin{cases} 1 & \text{if } (x_i \neq CALL) \wedge (x_{i+1} = CALL) \\ 0 & \text{else} \end{cases} \quad (74)$$

The prefix sum (Figure 60) of the stop bits yields is used to attach a thread id to each token:

$$thread(i) := \sum_{j=1}^{j<i} stop(j) \quad (75)$$

Threads are bound to consecutive groups, so that the sub-stream for a group  $i$  is constructed by removing unwanted tokens:

$$\text{StreamForGroup}(i) := \langle t_i : \text{thread}(i) \bmod n = i \rangle \quad (76)$$

In particular, the calculation of the *stop* bits and the prefix sum is also pipelined, allowing the decoder to process  $n$  tokens per cycle. The *scalar decoder* accepts  $\approx 1$  token per cycle from the narrow stream and looks for matching *CALL* patterns according to the specification. Each processor reads its sub-stream and in case of an executable *CALL* expression, it immediately jumps to the function  $f_i$  and rewrites the stream.

### 4.3.3 Ordered Writes

In addition to scheduling, the SRN also handles memory writes and is based on the concept described in Section 2.4. While it is possible to handle memory writes directly in the task function, this behavior can lead to race conditions because the SRN does not provide any guarantee at which time and on which resource a task is executed. For instance, the rasterization test in Section 4.3.6 allows multiple overlapping triangles to be computed in parallel as long as the pixel writes are committed in order. As a solution, the final literal values on the stream are interpreted as  $(\text{addr}, \text{data})$  tuples, which should be written into memory. In particular, from a stream  $s := \langle t_1, \dots, t_{2n}, \dots, t_c \rangle$  with  $t_1, \dots, t_{2n} \in \mathbb{Z}$ , the  $2n$  literal values at the beginning can be safely extracted, because these tokens will be never modified again, and perform the  $n$  write operations  $(t_{2i}, t_{2i+1})$  with  $i = 1..n$ . In addition, the tuples are also run-length encoded to save stream and memory band-width.

### 4.3.4 Software Development

In contrast to the fixed pattern matching, the functionality of the tasks is more flexible due to the software implementation. The SRN is compatible to any processor, which has an input and output streaming interface and supports the three opcodes listed in Table 3. In particular, the processor has to provide an opcode *get* for reading a token and an opcode *put* for writing a literal. Further, there is an opcode *call* required that takes a function and its number of argument to emit a *CALL* token. The scalar decoder (Figure 50) already marks matching calls with a special flag, so that the processor only has to invoke the associated task of a *CALL* expression. Here, the identifier  $i$ , which is located immediately after the *CALL* token, is interpreted as the *starting address* of the function.

Since the three opcodes *get*, *put* and *call*, represent a minimal interface between the application and the underlying stream rewriting architecture, their functionality might be also implemented as a runtime library in case of a pure software implementation.

Table 3. Minimal set of opcodes for stream rewriting

OpCode	Description
<b>get(x)</b>	Read literal token from stream.
<b>put(x)</b>	Append literal token to the stream.
<b>call(func, arg_count)</b>	Emit CALL token of function <i>func</i> with <i>argc</i> arguments.

The following complete example shows a C program of the recursive Fibonacci function:

```
void fib()
{
    int x;
    get(x);    // read argument x

    if (x <= 1)
    {
        put(x); // return result
    }
    else
    {
        call(add, 2);    // calculate sum of
        call(fib, 1); put(x-1); // fib(x-1)
        call(fib, 1); put(x-2); // fib(x-2)
    }
}

void add()
{
    int x, y;
    get(x);    // read argument x
    get(y);    // read argument y
    put(x+y);  // return result x+y
}
```

In order to improve the readability of the source code, a set of utility functions has been built on top of these basic instructions. For instance the macros *param\_\** declare and read arguments, the *put\** functions write typed values to the stream, and *call\** performs type-safe invocations. Thus, the example can be written more verbosely as:

```
void fib() {
    param_1i(x);

    if (x <= 1)
    {
        put1i(x); // return result
    }
    else
    {
        call(add, 2);    // calculate sum of
        call1i(fib, x-1); // fib(x-1)
        call1i(fib, x-2); // fib(x-2)
    }
}

void add() {
    param_1i(x);
    param_1i(y);
    put1i(x+y);
}
```

As a further improvement, the high-level compiler (Section 3.3) for hardware synthesis of recursive functions (Section 6) inserts these stream instructions automatically.

Config	Cores	LUT	FF	BRAM18K/36K	DSP48	
1x1	1	19.910	29.768	6	26	3
1x2	2	21.060	30.856	6	30	6
1x4	4	23.023	33.032	6	38	12
1x8	8	27.129	37.380	6	54	24
1x16	16	34.157	46.080	6	86	48
2x16	32	52.519	66.336	3	154	96
4x16	64	88.580	111.270	5	286	192
8x16	128	174.601	225.197	9	550	384

Table 4. Resource usage of the SRN test system.

### 4.3.5 Implementation

First, the scalability of the design is tested, which is followed by a direct comparison to the shared memory architecture presented Section 4.

A prototype of SRN architecture has been implemented using the VC707 evaluation board from Xilinx. The design contains also a memory controller (MIG), a VGA controller and a PCIe interface, so that it can be used as an add-on card in a PC. The processor is a custom implementation of the MicroBlaze from Xilinx, which is slower but smaller and can run at the same frequency of the SRN (200MHz). It has been specially designed for minimal size to test the concept of stream rewriting in a many-core environment. As a result, the processor does not support floating-point operations in hardware but there is a 32-bit multiplier and a barrel shifter, so that floating-point operations can be emulated. Due to the lack of pipelining, each instruction takes at least three cycles to complete.

In order to evaluate the scalability of the architecture, different configurations of the SRN with up to 128 cores have been synthesized. The resource usage measured of look-up tables (LUT), flip-flops (FF), block ram modules (BRAM) with 36/18kbits and multipliers (DSP48) is illustrated in Table 4. There is an almost linear increase of  $\approx 1207$  LUTs per core and a fixed overhead of  $\approx 16640$  LUTs. Since the FPGA contains  $\approx 300.000$  LUTs in total, even the largest 8x16 configuration can fit the onto the device without timing issues. Performance and scalability of the system are tested using both generic and complex test cases. The generic test cases consist of the recursive sum (*rsum*), the Fibonacci (*fib*) and Ackermann (*ack*) functions. In contrast to the balanced call tree of the recursive sum (Figure 61 a), the task *graph* of *fib* is asymmetric (Figure 61 b). To simulate a reasonable work load, each non-leaf task sleeps for approximately 8192 cycles and each leaf task for 4096 cycles. The worst-case for stream rewriting is given by the *Ackermann* function (sleeps 4096 cycles), which executes only the single task at the end of the stream per iteration.

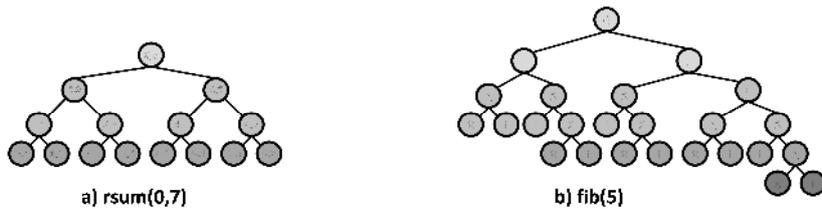


Figure 61. Call trees of the recursive sum and Fibonacci function.

The test is completed by the three application tests *mandelbrot*, *bezier3d* and *raytracing* introduced in Section 4.2.4. Especially the *bezier3d* test can take advantage of the ordered writes because it draws long spans of the same color.

#### 4.3.6 Results

For each configuration, the execution times of every test case have been measured in cycles and can be found in the paper [87]. However, in order to estimate the scalability of the SRN, the relative speedup is more important than the absolute but platform-specific timings. Therefore, the duration of a computation in relation to the 1x1 configuration is shown in Figure 62 on the next page.

It can be seen that the speedup of  $rsum(0,100)$  begins to stagnate with speedup of 15 for 2x16 cores due to insufficient work load. As a result, this test is bound by the latency of the external stream memory. However, when rewriting the larger stream of the  $rsum(0,10^4)$  and  $rsum(0,10^5)$  functions, the performance scales almost linearly with the number of cores. For examples, the 8x16 configuration receives a speedup of 126 for  $rsum(10^5)$  and the Fibonacci function runs 117 times faster. As expected, the Ackermann does not scale well since by construction, only one task can execute per iteration. Though, it still profits from the parallel decoding and the increased band-width in the 2x16, 4x16 and 8x16 configurations. In addition to the generic examples, the *Mandelbrot* and *Bezier3d* tests can be parallelized and achieve large speedups, while the scalability of the *Raytracing* test is mediocre. Hence, the SRN is well-suited for handling large task graphs if there is an adequate amount of concurrency. Also important, the same software has been used for all configurations of a particular test case.

### 4.4 Comparison

The SRN is directly compared to the ring and shared memory techniques using software-based stream rewriting. For this purpose, all three architectures for stream rewriting have been implemented on the ML605 board from Xilinx and utilize the MicroBlaze software with floating-point unit and 32Kb of local memory. Since both the ring and shared memory architectures perform the stream rewriting in software, they can run on the same generic hardware platform shown in Figure 52.

The central difference of the ring design is the usage of the circular communication channel instead of the external memory for passing tokens. However, in the SRN architecture, the stream management is moved into hardware and the ordered writes are placed on the stream as *OUT* tokens, so that the memory interface of the processors can be omitted. The resource consumptions of these two hardware architectures are listed in

Table 5. Although the SRN is slightly smaller for the 4x2 configuration, there is a larger hardware overhead for up to four cores, so that the size of both designs is roughly comparable.

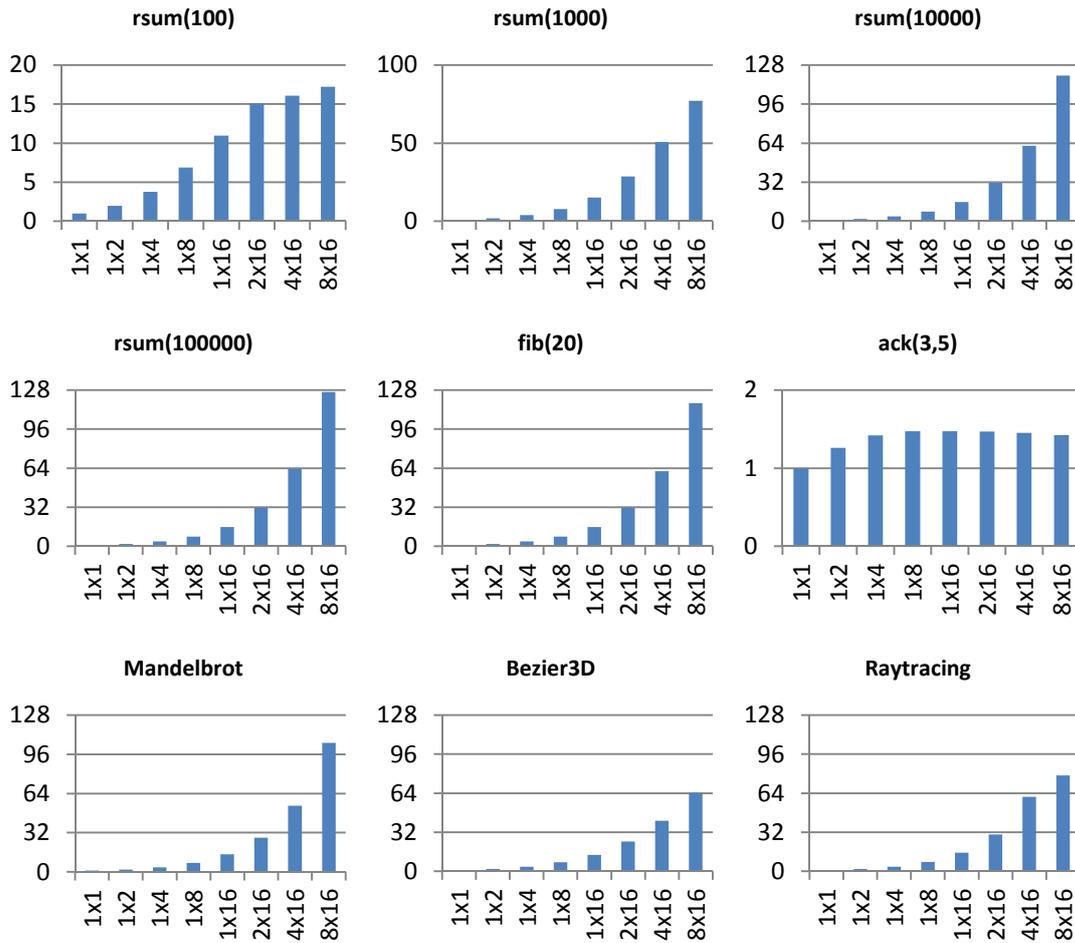


Figure 62. Relative speedup and scalability of the SRN using various test cases.

Table 5. Resource consumption of the stream rewriting architectures on ML605

Configuration	Cores	LUT	FF	BRAM18K/36K		DSP48
<b>Ring, Shared Memory</b> <i>Software stream rewriting</i>	<b>1</b>	14,874	19,446	1	29	5
	<b>2</b>	20,309	24,795	2	41	10
	<b>4</b>	31,494	35,463	4	65	20
	<b>8</b>	52,737	56,772	8	113	40
<b>Stream Rewriting Network</b> <i>Hardware stream rewriting</i>	<b>1x1</b>	16,185	21,266	3	25	5
	<b>2x1</b>	20,823	26,148	3	36	10
	<b>4x1</b>	31,188	39,597	5	63	20
	<b>4x2</b>	42,732	48,114	21	95	40

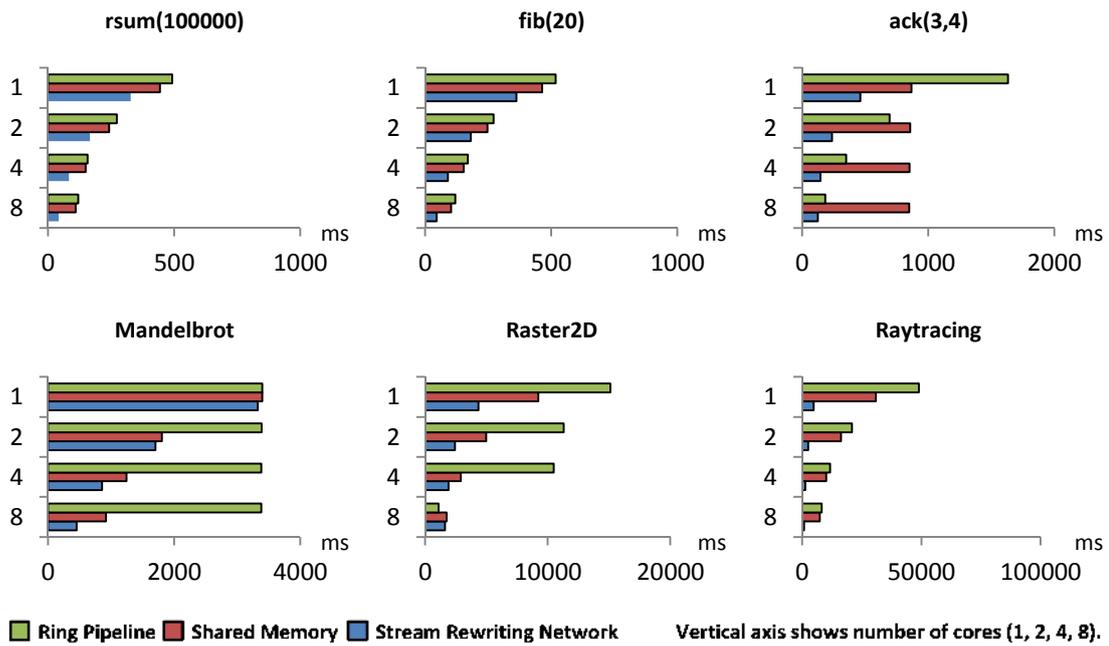


Figure 63. Performance comparison of different stream rewriting architectures.

The performance results of all three platforms and six test cases are illustrated in Figure 63. Given exactly the same compute capabilities, it can be seen that the SRN (blue) provides a performance advantage over the shared memory and ring architectures in nearly all tests. While the difference is mediocre for the compute-intensive tests like *rsum(10<sup>5</sup>)*, *fib(20)*, *Mandelbrot* and *Raster2D*, the speedup of the SRN becomes significant for the bandwidth limited *ack(3,4)* and *Raytracing*. In case of the Ackermann function, one only task is executed per iteration, so that the stream is mostly copied within the system. Likewise, the raytracing test employs several nested tasks per pixel and therefore also benefits from the hardware-based stream distribution.

The ring pipeline is usually slower than the two other topologies for compute-intensive test cases. In particular, consecutive iterations are mapped onto subsequent cores, so that all the expensive leaf tasks of the *Mandelbrot* are assigned to a single core. Consequently, in the eight core configuration, seven processors are waiting for input data, while the last one is busy and responsible for all expensive computations. However, since the stream is only stored in memory between the last and the first core of the chain, less bandwidth is consumed and the ring architecture outperforms the shared memory design in the *Ackermann* test case. Also, the rendering of the *Bezier3D* scene using the ring with eight cores is even slightly faster than the SRN and the best timing for this particular test case. As a result, the scalability of the ring architecture depends heavily on the application and is mostly unpredictable.

The performance of the shared memory is often located between the SRN and the ring architecture. However, the relatively minor improvement from 4 to 8 cores shows that this specific implementation is also limited by memory bandwidth. In particular, for both the shared memory and the ring, there are worst cases, which are not relevant for the SRN.

## 4.5 Conclusion

This section demonstrates two important aspects of stream rewriting:

- 1) It has been shown that stream rewriting can be implemented on generic multi-core systems without special hardware components. Especially the shared memory technique is highly portable and might be integrated into several existing embedded SoCs. Although the relatively slow MicroBlaze processor is mostly limited by memory throughput, the tests show scalability in many cases.
- 2) The performance of stream rewriting has been evaluated for many-core systems with up to 128 processors. A special on-chip network, the stream rewriting network (SRN), has been used both pattern matching and task distribution. The results (Figure 62) show linear scalability for some generic test cases, while the more complex applications like *Bezier3D* and *Raytracing* still achieve a speedup of more than 64 for 128 cores. A direct comparison of all three techniques indicates a performance advantage of the SRN over the shared memory and ring architectures. Since the SRN requires modifications at the system level, the shared memory technique offers an advantage in case of existing designs, for which such a fundamental change might not be feasible.

## 5 Graphics Processing

Despite the computational power and memory bandwidth of modern graphics processing units (GPU), a main limitation of these architectures is often the lack of efficient on-chip communication between different shader cores. Although individual stages of the Direct3D and OpenGL rendering pipeline are programmable, its topology and data flow remain fixed. In this section, novel hardware architectures for software-based rendering are presented, which provide an efficient communication infrastructure for the creation of highly optimized and application-specific rendering pipelines. For this purpose, the state of the pipeline is encoded as a stream of data and control tokens, while the functionality of the shader stages is represented as rewriting operations. As a result, this architecture supports an arbitrary amount of dynamic and recursive shader stages as well as complex data flow and light-weight synchronization within the rendering pipeline. This section is based on paper [86] and further unpublished work.

### 5.1 Introduction

Since the introduction of shaders in Direct3D [17] with version 8.0 and in OpenGL [121] via the *ARB\_vertex\_program* and *ARB\_fragment\_program* extensions, there is a shift from the traditional fixed-function pipeline of the original OpenGL towards a more programmable and software-centric approach [122]. Subsequently, further pipeline stages have been added, so that there are currently up to six different shader types for pixels, vertices, tessellation, per-triangle, and general purpose computations [18].

In recent years, GPUs have evolved from specialized coprocessors for rendering to general purpose computation platforms supporting a wide range of applications [123]. Compute APIs like CUDA [19] [124] and OpenCL [125] offer even a more fine-grained control of the underlying graphics hardware by directly exposing the grid of processing cores. However, their architecture is mainly focused on data parallelism and forces a large number of cores to execute the same kernel. Hence, in order to take advantage of the parallelism achieved by hundreds of identical processing cores [4], problems must be translated into a set of data parallel kernels [126] [127]. In addition, there is little support for the creation of custom pipeline stages, so that queues and even the scheduling must be implemented in software. While this development might eventually lead to a revival of software rendering, current graphics processors consist of fixed, configurable and programmable components, whose complexity is rather exposed than hidden by the main graphics APIs.

On the other hand, even modern GPUs typically do not allow redefining the structure of the graphics pipeline and neither permit the addition of custom shader stages. As a consequence, there is a discrepancy between the computational power of individual shader stages and the flexibility of the graphics pipeline in general. For example, separate shaders for surfaces and lights, linked dynamically on demand, could enforce a more modular material design similar to Pixar's RenderMan [128].

Also, hardware support for recursion is still weak or non-existent, although recursive data structures and algorithms are common for graphics processing. In this context, recursive shaders could enable custom tessellation patterns, procedural geometry, ray-tracing, and the traversal of the scene graph to be implemented entirely on the GPU. Hence, both the limitations of recent graphics hardware and also the fixed programming model hinder the development of more advanced rendering techniques. However, it is possible to implement custom rendering pipelines by manually scheduling shader instances in software and storing work queues in external memory [129]. In fact, this approach effectively emulates a different type of graphics hardware on top of general purpose computing APIs like CUDA or OpenCL.

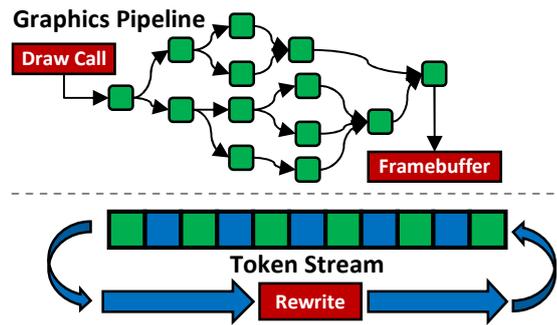


Figure 64. Graphics pipeline based on stream rewriting.

This section presents a much more general solution for dynamic rendering pipelines and evaluates the usage of stream rewriting for graphics processing using three different architectures, which are both based on a modified version of the stream rewriting network (Section 4.3). The first architecture is based on functional programming. The second design utilizes SIMT shader cores and tile-based rendering, while the third prototype employs general purpose processors. The concept of stream rewriting for graphics processing is illustrated in Figure 64 and consists of two views. The rendering pipeline on the top is modelled as a graph of programmable shader nodes, which are invoked by a draw call and eventually produce pixels that are written into the framebuffer. At the hardware level, the state of the pipeline is represented as a token stream and the functionality of the shader stages corresponds to rewriting operations. Initially, the draw call is placed on the stream as a sequence of tokens that refer to an input node of the graph. When the stream becomes empty, the execution of the draw call is completed and at this point all primitives are written into the framebuffer. Beside the general benefits for many-core architectures, stream rewriting offers the following benefits graphics processing:

- **Data and Pipeline Parallelism**

The majority compute APIs are focused on data parallelism and only CUDA supports a very limited type of dynamically nested tasks on recent GPUs. However, graphics processing requires data parallelism to within the same stages and pipeline parallelism between subsequent stages. In addition, even the most recent programmable graphics pipeline does not support programmable communication between shader units beside data exchange via the global memory. Here, stream rewriting provides a unified model of computation, which can describe both types of concurrency.

- **Virtual Rendering Pipeline**

All graphics processors based on stream rewriting support an arbitrary amount of custom pipeline stages defined by the software with full programmability. Therefore, a more flexible application-specific graphics pipeline can be build to better fit the application-specific requirements of a particular rendering algorithm. There is only one generalized type of shader stage, which can be instantiated and connected using queues, feed-back loop as well as fork and join nodes to create complex topologies.

- **Recursive Shaders**

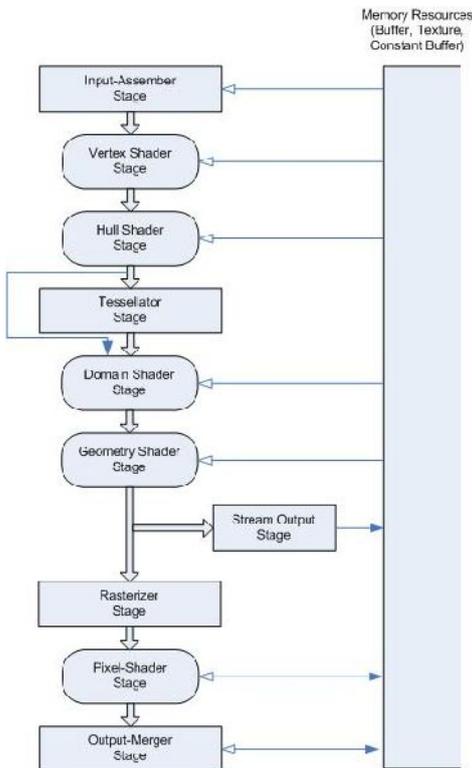
Recursion is commonly used in geometric algorithms but current graphics hardware provides little support for this powerful technique. However, the proposed model directly supports recursion in a general way and also parallelizes the evaluation of concurrent branches via load-balancing on multiple processors.

Before the computational model of stream rewriting for graphics processing is presented, the next sections first contain a discussion of related work.

## 5.2 Related Work

This section describes several different architectures related to graphics processing.

### 5.2.1 Direct3D and OpenGL



The functionality of current GPUs is exposed at the application level through various programming interfaces (API) which can be distinguished into graphics and compute APIs. In graphics mode, the individual steps of the rendering process are represented as stages of a pipeline (Figure 65). An application can use APIs like Direct3D [130] or OpenGL [121] to configure the pipeline and to supply input data. The rectangular stages correspond to fixed blocks of hardware and can be modified by a set of parameters only. In contrast, the rounded blocks can be programmed by custom functions, which are called *shaders*. For instance, the user can load a function that calculates the lighting of vertices or pixels into the corresponding stage. However, the data-flow between these blocks is based on a fixed set of registers, so that the structure of the rendering-pipeline cannot be changed.

As a result, new shader types for specialized tasks have been introduced continuously. While Direct3D 8.0 knew only pixel and vertex shaders,

Figure 65: The Direct3D 11 graphics pipeline [http://msdn.microsoft.com]

the current version 11.1 supports up to six different shader types. In addition, the amount of input and output data per stage is fixed. The only programmable stage that can amplify data is the geometry shader, but it can output a maximum of 1024 values per invocation. Since computing power is growing faster than memory bandwidth, it can be advantageous to create geometry procedurally in the rendering pipeline. Therefore, the tessellator unit, which can generate a large amount of dynamic vertices, has been added recently in Direct3D 11. But it is also a fixed function that works on a unit rectangle and only the stages before and after the tessellator are programmable to compensate for this limitation. Feedback loops are not directly supported but the intermediate results of computation can be streamed out into memory resources and later read back. If the requirements of the application match this scheme, the graphics APIs offer a convenient high-performance interface for rendering.

However, for more sophisticated problems, compute APIs like OpenCL [125] or CUDA [124] must be employed. They offer more flexibility by directly exposing the grid of processing cores and are thus most useful for data-parallel problems. As a result, the creation of custom pipeline structures on the GPU is still an open problem, because there is little hardware support for pipeline parallelism. For example, queues can be implemented in shared memory and threads are synchronized using atomic functions. While this approach has been particularly successful for implementing custom rendering techniques [131], it effectively simulates a completely different type of hardware on top of the data-parallel processor grid. This thesis proposes stream rewriting as a tradeoff between these two modes to enable efficient thread management and scheduling in hardware, while the execution of shaders is handled in software for maximum flexibility.

### 5.2.2 Hardware Architectures

In addition to the GPU architectures from NVIDIA [96] or AMD [132], there exist also several alternative designs, which improve certain aspects of the rendering pipeline:

The *Larrabee* processor [133] has been already mentioned in Section 1 as a many-core system but actually it has been designed as a more flexible GPU based on up to 48 x86-compatible cores. Thus, the instruction set contains additional opcodes for vector arithmetic for floating-point calculations. Instead of dynamic scheduling, the execution units are utilized by multiple hardware threads. In addition, the *Larrabee* supports cache coherence, virtual memory and arbitrary data exchange between the cores to facilitate high-level software development of rendering architectures.

The two relevant rendering techniques are rasterization and ray-tracing. The *ray processing unit* RPU [134] [135] has been developed especially for raytracing and handles irregular control flow and recursion. Shader programs can contain a special *trace* instruction, which performs a scene traversal in hardware and computes the nearest intersection point of a ray. Similarly, the mobile SGRT ray-tracing architecture [136] contains a dedicated tree traversal unit with four pipelines, which handles recursion by storing pending threads in a FIFO that is connected to the input buffer. The concept of stream rewriting is based on a similar mechanism but instead of rays, there is native support for recursive and indirect functions in general.

The *programmable culling unit* (PCU) [137] represents a step towards a programmable but hardware accelerated rasterizer. Unlike the pixel shader, which works on individual pixels, the PCU can speed up rendering by excluding larger blocks of pixel from the rasterization process. In comparison, the fully reconfigurable stream rewriting pipeline could also support a programmable culling stage as part of the rasterizer.

Also related to the token stream, the *F-Buffer* [138] stores the intermediate state of pixel shaders in external memory. However, since the token stream contains control information to encode state and topology of the entire rendering pipeline, it can be seen as a generalized form of the *F-Buffer*.

Order independent transparency requires the fragments for each pixel to be sorted and merged from back to front. There are several software solutions like depth peeling [139], the K+-buffer [140] or linked lists [141]. Alternatively, the *R-Buffer* [93] keeps all active fragments of the current frame in a queue and writes them in order for each pixel. For this purpose, each fragment has to circulate in the *R-Buffer* until all further values at the same location have been written. Hence, the maximum number of iterations corresponds to the depth complexity of the scene.

The *Delay Stream* [142] is actually a relaxed version of the *R-Buffer* and defers the processing of rendering commands until a certain threshold is reached. This allows the rasterizer to look ahead the command stream and cull triangles, which will be overdrawn by subsequent primitives.

The *Pomegranate* design [94] is a scalable graphics architecture with several vertex and pixel pipelines working in parallel. Here, the main challenge is maintaining the original order of triangles from the draw call, so that the rendered image is indistinguishable from a sequential implementation. For this purpose, *Pomegranate* contains a point-to-point network, which sorts the results the triangle fragments according to their screen space position. The stream rewriting network (SRN) introduced in Section 4.3 architecture resembles a generic variant of this design.

### 5.2.3 Software Pipelines

Before introducing the concept in Section 5.3, several different software and hardware architectures are presented, which are related to graphics processing for more flexible graphics pipelines. The creation of custom graphics pipelines on current GPUs has been already researched extensively since it provides the foundation for the implementation of novel rendering techniques [143]. While a rendering pipeline usually contains both data and pipeline parallelism, compute APIs like CUDA or OpenCL are optimized for data parallel kernels with coherent control flow. Also, the communication between processing cores is usually bound to several restrictions. Hence, the central challenge is the dynamic scheduling of heterogeneous work items within the limitations of the CUDA model.

One possible solution for the management of dynamic threads on the GPU is the implementation of the scheduler as a kernel itself [131] [144] and the usage of work stealing for load-balancing [145]. For instance, a hierarchical software rasterizer has been implemented entirely in CUDA [129] but requires significant modifications to fit into the data parallel scheme. Due to the lack of hardware support, queues must be stored as

buffers in global or shared memory, while the read and write pointers are accessed using atomic operations. Though, it is possible to manage queues with multiple concurrent producers using parallel prefix sums [146]. However, most of these software-based scheduling approaches require combining all tasks into a single kernel (mega-kernel) that selects the correct sub-task at runtime through dynamic branching. Therefore, this technique effectively interferes with the single-instruction multiple-thread (SIMT) architecture of modern GPUs, which is optimized executing for groups of threads executing the same control path. As a possible solution, the authors of [147] propose to sort threads according to their kernel and to divide complex shader into more specialized tasks.

A more recent feature of CUDA is called *dynamic parallelism* and permits the recursive invocation of kernels on the GPU [148]. However, the driver has to reserve up to 150MB of external memory for each level of recursion, so that this technique is most useful for large nested grids of kernels. Stream rewriting supports all types of recursion and a more fine-grained creation of individual threads by inserting control tokens into the stream. The generic *GRAMPS* pipeline shares some similarities with this theoretical model and also supports different types of concurrency [149] like pipeline and data parallelism. Shaders are represented as process nodes and connected via ordered and unordered communication queues. In comparison, stream rewriting basically restricts the topology of the shader network to the class of series parallel graphs [109], which are built from sequential and parallel composition.

This restriction allows storing the shader network and the contents of the communication queues as a stream (Section 3.2), so that the inputs of a particular shader instance are always located in a compact range of tokens. There is no explicit management of threads required and therefore also the topology of the network can be modified at runtime.

Another challenge of in rendering pipelines is the parallel processing of multiple geometric primitives and the subsequent reordering of fragments, before they are written into the framebuffer. For this purpose, the authors of the K-Buffer [150] propose a hardware extension for distributing fragments to a specific shader core. Since this functionality does not exist in current GPUs, a workaround to minimize the resulting memory hazards is presented in [151] and a different solution [140] utilizes atomic operations.

#### 5.2.4 Shader and Compute Languages

The first programmable graphics hardware supported only a few selected instructions and registers, so that shader programs were naturally very short and had been optimized manually in assembly language to exploit the limited capabilities. Due to the development of flexible graphics processors with a general purpose instruction set, more complex shaders became available and required the usage of high-level languages like Cg (multi-platform), HLSL (Direct3D) or GLSL (OpenGL).

These languages are variants of C with special data types and optimized for graphics processing. In particular, each shader program replaces the functionality of a previous fixed block like the vertex or fragment processing stages and interacts with the remaining fixed hardware via a set of input and output registers. However, the communication between shader stages is currently an underrated aspect of all OpenGL or Direct3D shading

languages. Several restrictions are lifted due to the further improved flexibility of modern graphics hardware, so that more functionality can be moved into software but on the other hand, modularity and reusability are becoming important issues.

For this purpose, the *Spark* [152] shading language bundles the functionality of an effect, which is often spread across several stages, into a single class. This aspect-oriented approach is orthogonal to the programming model of stream rewriting and could be optionally layered on top of the SRM. Hence, the Spark language could take advantage of the dynamic composition and becomes even more useful as the SRM enables almost arbitrary complex rendering pipelines.

A scene can contain several different types of materials and lights. For instance, there can be point-lights, spot-lights, an ambient environment light or the sun. Some materials may be reflective or interact with the light in different ways. As a result, the calculations for each combination of a material and a light are unique. Offline renderers like, for instance, Pixar's *RenderMan* use different light and material shaders that can invoke each other [128]. Although there is only one active pixel-shader in Direct3D 11, the API still supports a limited form of dynamic linkage, which permits to combine different subroutines at runtime. Though, each variant requires a separate draw call, which is expensive in terms of CPU cycles. For comparison, stream rewriting is even more flexible and enables the dynamic composition of shader fragments without CPU interaction.

Also, the generation of procedural geometry using grammars [153] [154] might map natively to the concept of stream rewriting. For the general purpose *bulk-synchronous GPU programming* language (BSGP) [155], which supports dynamic threads as well as fork and join constructs, the stream rewriting processor (SRP) could be a possible target architecture. Similarly, the *Cilk* programming language extends C with the *spawn* keyword to invoke functions on a new thread and therefore also supports dynamic and recursive tasks. Basically, *spawn* could be implemented on the SRM as a processor opcode that inserts the corresponding control and data tokens into the stream.

The diversity of shading languages, which are often tied to a special rendering architecture, hinders the maintainability and reusability of shader programs. The *AnySL* [156] compiler demonstrates a method that allows a renderer to support several different shading languages through recompilation. In this context, the functional programming model could be also used as an intermediate language that is dynamically mapped to different target platforms. As a result, the stream rewriting language presented in this paper might become also relevant for existing rendering systems.

Similarly, the GRAMPS programming model also deals with the dynamic scheduling of irregular pipeline and data parallelism [149]. However, they employ a more traditional global scheduler that manages task queues and thread instances while the SRM uses local pattern matching.

In addition, several functional languages have been also proposed for GPU programming. For instance, *Vertigo* [157] is based on Haskell and allows composing complex objects from parametric surfaces and geometric operators in the shader. For example, a torus is constructed by rotating a circle using the *revolve* operator and the *displace* function adds a height field onto an existing surface. In addition, the library contains already several basic definitions for computing the interaction between light and materials, which can be applied to these objects as well. Since *Vertigo* targets an old class of graphics hardware (DX8.1), the functional program is compiled into a restrictive pseudo assembly code without flow control. Using the SRM, it might be possible to execute at least a sub-set directly in hardware and therefore retain the modularity of the functional language.

The more recent functional shading language *Renaissance* [158] is conceptually based on *Vertigo* and pursues similar goals. Technically, it does not generate low-level assembly code but utilized the vendor and platform independent OpenGL Shading Language (GLSL). While the underlying hardware usually expects a pair of tightly coupled vertex and fragment shaders, *Renaissance* allows specifying the shader as a single program and attaches different frequencies to variables. For instance, uniform variables never change during the invocation of a draw call, while per-vertex variables are automatically interpolated to the fragment level. The *LET* environment of stream rewriting supports an arbitrary number of user-defined frequencies although there is no automatic interpolation.

The language *Obsidian* [159] is also based on Haskell but targets general purpose computations on GPUs and therefore generates CUDA source code. Consequently, the language provides an abstract array type, which either represent a concrete dataset or references result of a previous computation. There are several standard functions like *map*, *fold* or *zip* for array manipulation that can be combined with user-defined functions by a special composition operator. Hence, by threatening the GPU as a vector machine, the error-prone indexing of individual array elements and the manual handling of threads can be mostly avoided, so that the development of CUDA kernels is greatly simplified. A very similar approach for C++ is also attempted by the library *Thrust* [160], which contains a large number of templates for array handling on the GPU and also supports the fusion of kernel functions.

### 5.2.5 Recursive Shaders

Despite the coarse grained *dynamic parallelism* of recent GPUs [161], general support for recursive functions and functional languages requires significant effort [162]. In particular, the two most important challenges are the storage of the stack for a large number of threads [163] and the SIMT divergence during recursion [164]. Through, there are several important use cases for recursive algorithms and data structures in computer graphics [20]. For instance, by storing the geometry of a scene as a bounding hierarchy tree, visible parts can be queried hierarchically with logarithmic complexity [165]. In addition, point hierarchies allow to dynamically improve the quality at visually important regions like the silhouette [166]. Hence, these recursive techniques help to avoid computations on elements of a scene, which are either not in the field of vision or do not contribute to the quality of the final image.

However, the implementation of recursion on the GPU has not been solved in general. Although recursive shaders can be constructed by a while loop and a stack in local or global memory, this approach raises several issues. Beside the high latency of global memory and the overhead of dynamic branching, it is not possible to create a new thread for each recursive invocation, so that this technique is far from optimal. While recursion can be eliminated in some case like the stack-less raytracing implementation [167], this method visits only one branch of the tree and therefore does not represent a general solution.

For highly detailed scenes, the memory bandwidth required to read the geometry data from external memory can represent a bottleneck. Hence, it is often more efficient to generate the dynamically within the rendering pipeline from a more coarse grained representation like high-order surfaces [168]. For this purpose, Direct3D 11 introduces the *tessellator stage*, which subdivides a square, triangle or line segment, which is then mapped to the corresponding geometry using the *domain shader*. If more flexibility is required, also the *compute shader* can be used to generate geometry, which has been shown for stack-based terrains in [169]. Further, a recursive approach for geometric subdivision has been presented by [170] but requires to load and store the entire geometry for each level of detail, so that they basically perform a deep-first expansion of the stream. Contrary, the stack-based scheduling (Section 2.3.2) employs a combination of depth-first and breast-first traversal and therefore requires significantly less memory bandwidth.

As an alternative, the adaptive tessellation of [171] is written in CUDA and does not require special hardware support for packing the resulting vertices into a continuous stream. Instead, it uses multi-pass approach to recombine the outputs of each patch. First, the number of vertices per patch, which depends on the tessellation level, is computed. Then, the output address of each patch is determined by adding the size of all preceding patches using the fast parallel prefix sum [146]. Finally, the patches are sub-divided and each instance writes its outputs the previously computed address in parallel. However, this technique is not applicable for stream rewriting since it relies on a fixed number of inputs per patch, which fit better onto a data parallel model of a SIMT GPU.

### 5.2.6 Task Queues

Queues are the preferred communication medium between concurrent processes, because they decouple different rates of production and consumption while maintaining the order of elements. For instance, a task queue can be used for distributing non data parallel work items [172]. However, for optimal performance, both reader and writer must be replicated many times, so that multiple elements can be produced and consumed in parallel. Currently there are two possibilities to implement queues on the GPU. The first method uses a buffer in shared memory and atomic operations to update the read and write pointers [145]. The second approach uses prefix sums to assign parts of the queue to distinct threads, before the actual calculation happens. Afterwards, multiple threads can access the memory simultaneously without explicit synchronization. Although this method employs more concurrency, the amount of data, a thread writes into the queue must be known or calculated before, so that the memory regions can be aligned seamlessly [146].

### 5.3 Generic Graphics Pipeline

The execution model of the proposed graphics processor is based on the concept of stream rewriting. In this section, the mapping between a generic rendering pipeline and the rewriting rules of the token stream is described. After the abstract rendering pipeline and its functional representation have been specified, several advanced examples and three different implementations are presented. Similar to the task graphs presented in Section 3.2, the stream holds the state and the topology of the rendering pipeline, while each function, and therefore each shader stage, is modeled as a rewriting rule.

#### 5.3.1 Shader Stages

The basic building block of the rendering pipeline is a generic shader stage (Figure 66) with  $n_i$  input and  $m_i$  output registers. In addition, shaders can be hierarchically combined to create more complex networks using either sequential (*seq*) or parallel (*par*) composition. Similar to the

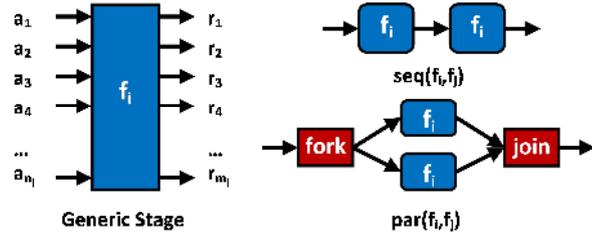


Figure 66. Blocks of the generic rendering pipeline.

mapping of task graphs (Section 3.1), the rendering pipeline is modeled as a functional program  $P$  consisting of  $u$  shaders,  $v$  sequential, and  $w$  parallel nodes:

$$P := \left\{ \underbrace{f_1, \dots, f_n}_{u \times \text{shader}}, \underbrace{f_{n+1}, \dots, f_{n+m}}_{v \times \text{seq}}, \underbrace{f_{n+m+1}, \dots, f_{n+m+k}}_{w \times \text{par}} \right\} \quad (77)$$

The generic stage is specified as a function  $f_i$  that maps the  $n_i$  input values to  $m_i$  results:

$$f_i: \mathbb{Z}^{n_i} \rightarrow \mathbb{Z}^{m_i} \quad (78)$$

Here,  $\mathbb{Z}$  represents a machine word and can contain both integer and floating-point numbers. Similar to Section 3.1, the sequence  $seq(f_i, f_j)$  of two rendering pipelines  $f_i: \mathbb{Z}^{n_i} \rightarrow \mathbb{Z}^{m_i}$  and  $f_j: \mathbb{Z}^{n_j} \rightarrow \mathbb{Z}^{m_j}$  with  $m_i = n_j$  corresponds to the composition:

$$\begin{aligned} seq(f_i, f_j) &:= f_j \circ f_i \in \{f: \mathbb{Z}^{n_i} \rightarrow \mathbb{Z}^{m_j}\} \\ (r_1, \dots, r_{m_j}) &:= f_j(f_i(a_1, \dots, a_{n_i})) \end{aligned} \quad (79)$$

As shown on top of Figure 66 the input values are first processed by  $f_i$  and then passed to  $f_j$  to produce the outputs of the combined stage. Hence, the resulting function has the same signature as a single shader and can be used itself for hierarchical combination.

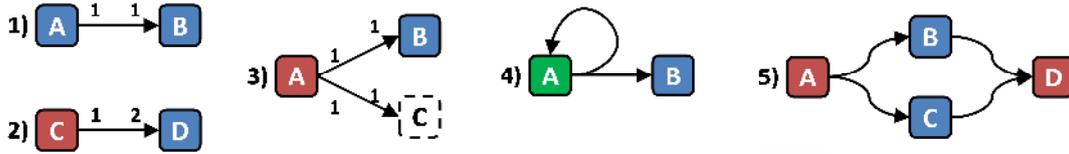


Figure 67. Pipeline fragments modelled via rewriting rules.

The parallel composition  $par(f_i, f_j)$  of  $f_i: \mathbb{Z}^{n_i} \rightarrow \mathbb{Z}^{m_i}$  and  $f_j: \mathbb{Z}^{n_j} \rightarrow \mathbb{Z}^{m_j}$  can be also described in a similar way. Here, an *implicit fork* block distributes the  $n_i + n_j$  input registers  $(a_1, \dots, a_{n_i+n_j})$  of the combined stage to both  $f_i$  and  $f_j$  to produce the *result vector*  $(r_1, \dots, r_{m_i+m_j})$  of the *join* block:

$$\begin{aligned} (r_1, \dots, r_{m_i}) &:= f_i(a_1, \dots, a_{n_i}) \\ (r_{m_i+1}, \dots, r_{m_i+m_j}) &:= f_j(a_{n_i+1}, \dots, a_{n_i+n_j}) \end{aligned} \quad (80)$$

The formal description of the *join* operation uses the identity function  $id$  to synchronize the results of  $f_i$  and  $f_j$ . For this purpose, the strict evaluation order of stream rewriting ensures that both are *completed* before  $id$  is invoked:

$$par(f_i, f_j) := (a_1, \dots, a_{n_i+n_j}) \mapsto id(r_1, \dots, r_{m_i+m_j}) \quad (81)$$

In a summary, the formalism for mapping a generic graphics pipeline into rewriting rules is equivalent to the scheduling of task graphs presented in Section 3.2.

### 5.3.2 Pipeline Fragments

Based on rules for parallel and sequential composition, also the more complex pipeline fragments shown Figure 67 can be derived using a similar approach. The examples presented in this section are based on the post-order format of the stream grammar but could be also described in preorder. Each node represents a single pipeline stage and the listed *fragments* can be combined recursively to create more complex topologies.

Example 1 in Figure 67 shows a minimalistic pipeline built from the two shaders  $A$  and  $B$ . Similar to the behavior of *pixel* and *vertex shaders*, each invocation of  $A$  creates exactly one call to  $B$ . In terms of stream rewriting, shader  $A$  can be specified by the rule:

$$A(a_1, \dots, a_n) := \langle r_1, \dots, r_m, B, CALL \rangle \quad (82)$$

Here,  $a_i$  denotes the inputs and  $r_i$  the outputs of stage  $A$ . However, it is also possible to produce several invocations by emitting multiple *CALL* tokens. In example 2, the stage  $D$  is called twice, so that the rule is extended to:

$$C(a_1, \dots, a_n) := \langle r_1, \dots, r_x, D, CALL, r_y, \dots, r_m, D, CALL \rangle \quad (83)$$

This functionality can be used to describe the geometry shader stage of Direct3D, which emits a variable amount of triangles per invocation. A primitive might be also culled by returning an empty sequence. In addition, it is also possible to choose the target stage dynamically by calculating the stage identifier  $i$  (Example 3 in Figure 67):

$$A(a_1, \dots, a_n) := \begin{cases} \langle r_1, \dots, r_m, B, CALL \rangle & \text{if } a_1 < 0 \\ \langle r_1, \dots, r_m, C, CALL \rangle & \text{else} \end{cases} \quad (84)$$

Here, shader  $A$  either calls  $B$  or  $C$  depending on the value of argument  $a_1$ . Similar to the dynamic branching in Direct3D, the shader can switch between different code paths. In addition, shaders can be also linked dynamically by threatening the stage identifier  $i$  as a function pointer. For example, different materials and lights in a scene could be stored as function pointers and combined *without* intervention from the CPU.

Further, a shader can invoke itself to create a feed-back loop (Example 4 in Figure 67). For instance, a tessellation shader  $T$  can use this capability to create a smooth surface by recursively subdividing a triangle  $t$  into four fragments  $t_0, \dots, t_3$ . First, the shader checks if the maximum detail level has been already reached ( $d = 0$ ) and then either draws the triangle by calling the stage DRAW or performs a further subdivision step ( $d > 0$ ):

$$T(d, t) := \begin{cases} \langle d, t_0, T, CALL, \dots, d, t_3, T, CALL \rangle & \text{if } d > 0 \\ \langle t, DRAW, CALL \rangle & \text{else} \end{cases} \quad (85)$$

Similar to the task graphs, the pipeline can be split and later synchronized by the fork-join construct, which basically corresponds to a sub-routine. In example 5, a complex surface shader  $A$  invokes multiple material layers ( $B, C$ ), whose results that are eventually combined by shader  $D$ :

$$A(\dots) := \langle \dots, B, CALL, \dots, C, CALL, D, CALL, \dots \rangle \quad (86)$$

Hence, the rewriting rules are suitable for modelling a large number of different pipeline configurations. As a result, the flexibility is greatly enhanced in comparison to the fixed rendering pipelines of Direct3D and OpenGL (Section 5.2.1).

### 5.3.3 Dynamic Parallelism and Recursion

Similar to the dynamic task graphs, also the structure of the rendering pipeline can be modified at runtime. For this purpose, a shader returns a sub-stream, which encodes the corresponding rendering pipeline instead of literal values. Several use cases for this approach are shown in Figure 68. For instance, a dynamic  $d$  *fork* shader (Figure 68a) can create  $k$  threads of  $f_i$  with  $x = 1 \dots k$  by emitting a variable number of tokens:

$$fork(a_1, \dots, a_n) := \langle 0, \underbrace{a_1, \dots, a_n, 1, i, CALL}_{x=1}, \dots, \underbrace{a_1, \dots, a_n, k, i, CALL}_{x=k}, join, CALL \rangle \quad (87)$$

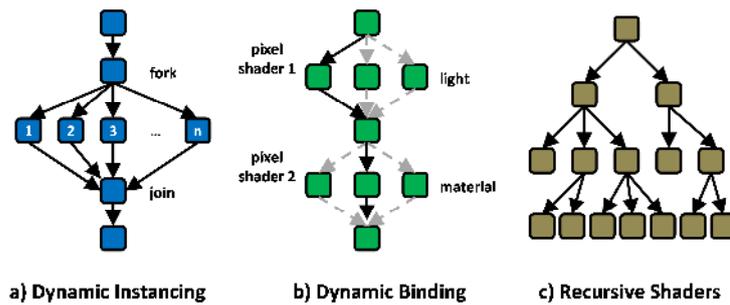


Figure 68. Examples of dynamic and recursive rendering pipelines.

In particular, the *join* shader requires one argument and wait for the zero literal at the beginning for synchronization. As a possible use case, this technique enables a graphics processor to generate a variable number of draw calls within a shader, which is not feasible with the current Direct3D and OpenGL implementations.

The interaction between lighting and materials is usually highly complex and often depends on application-specific requirements to create a particular graphics style. For this purpose, high-quality off-line renderers like RenderMan [128] usually support separate shader programs for lights and materials, which are combined on demand. Similarly, recent GPUs offer dynamic linking to allow modular shaders but it is actually performed on the CPU and therefore does not account for dynamic conditions calculated within a shader. However, since the stage identifier  $i$  for a *CALL* token can be chosen dynamically, different paths in the shader network can be chosen dynamically for each primitive. The example shown in Figure 68b) displays a combination of three materials and three light stages, which would require to a combination of nine statically linked shaders.

In addition, stream rewriting can express recursive shaders to implemented branch-and-bound algorithms (Figure 68 c), which are commonly used as a performance optimization when iterating over large data sets. By conditionally invoking or discarding subsequent stages, parts of the call tree can be cut early to omit unnecessary computations. For instance, the intersection test between a ray and the scene may skip larger blocks based on their bounding volume. Another example for recursion is the tessellation shader illustrated in Figure 69. Here, each invocation of the *patch* rule subdivides the given triangle into four smaller fragments. As a result, the triangular patch of iteration  $s_0$  is subsequently expanded into a set of 16 triangles in  $s_2$ , which are eventually sent to a rasterizer stage. Most notably, the concept of stream rewriting naturally maps to the refinement steps.

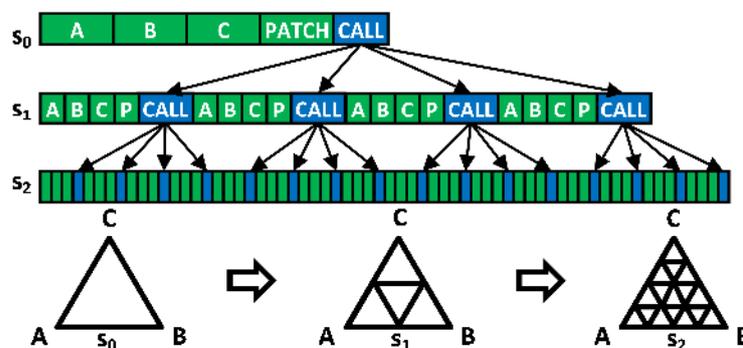


Figure 69. Recursive tessellation using stream rewriting.

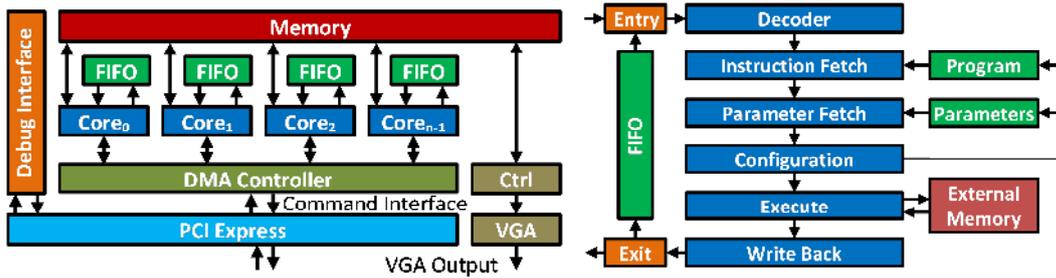


Figure 70. System architecture of the stream processor. Figure 71. Structure of a stream rewriting core

## 5.4 Functional Processor

In this section, the design of a *stream rewriting processor* (SRP), which is based on the functional language from Section 3.1, is implemented and evaluated.

### 5.4.1 Stream Rewriting Processor

The proposed system architecture of the *stream rewriting processor* (SRP) is illustrated in Figure 70. At the top level, the processor contains several rewriting cores (SRC), which are working on different streams in parallel (Figure 71). Each of them has access to a local buffer, which stores the token stream, and the external memory, containing framebuffer and textures. While all cores receive the same sequence, each core is responsible for executing the whole rendering pipeline for a different part of the screen. The partitioning is configured by the host CPU using parameters.

Further, the SRP is equipped with a PCIe interface for communication with the host CPU and a VGA controller displaying the rendered image. The PCIe channel is split into a debug interface and a DMA controller, responsible for sending and receiving token streams.

### 5.4.2 Stream Rewriting Core

Similar to the abstract execution model, the stream rewriting core consists of a ring pipeline processing the token stream. In addition, there is also an entry point, which receives new tokens from the DMA controller, and an exit point, where results can leave the ring. The rewriting rules are implemented in the pipeline stages in the middle, so that one iteration in the ring corresponds to one invocation of the function *rewrite*. The queue between exit and entry is used as a temporary storage to compensate for varying stream lengths.

First, the decoder receives the token stream and performs the pattern matching. It generates a sequence of program addresses that are used in the instruction fetch stage to load a program, which evaluates the rule or bypasses unmatched tokens. Likewise, the parameter fetch stage retrieves global parameters from the parameter memory. LET tokens are interpreted by the configuration stage, which also initializes the program memory. The execution stage evaluates arithmetic operations and is responsible for memory access. Finally, the results are written back on the token stream to proceed with the next iteration or leave the core at the exit point. In contrast to a RISC processor, there are no temporary registers, so that all inputs of an instruction are either arguments or parameters. Hence, the

argument fetch, execute and write back phases of a single rewriting function never overlap. Therefore, several types of pipeline conflicts can be avoided, which leads to considerable simpler hardware. On the other hand, complex functions must be split in a preprocessing step and intermediate results are passed as arguments on the stream.

The *execution unit* (Figure 72) contains several stages and their arrangement has been chosen according to the requirements of graphics processing to chain as many commands as possible. Several stages have additional registers or even a stack to store temporary values. As a result, multiple subsequent instructions can be combined into macro opcodes to reduce the number of dependent rules significantly.

Beside the standard 32-bit integer and single-precision floating-point formats, the pipeline also supports 32-bit packed RGBA colors for framebuffer operations and textures. The first stage can evaluate a 4D dot product, which is a very common operation in geometry computations. An integer variant is also used for address calculation of pixels and texels. After that, the REQUEST stage may initiate a read operation at the memory interface while the result will be collected later in the RECEIVE stage. Similarly, the rest of the stages are responsible for packing floating-point values, performing color operations (RGBA), comparisons, and bitwise logical operations. The IF/ELSE/END\_IF stage conditionally discards values and contains a stack for up to 32 nested blocks. Finally, the last stage may optionally write the value into external memory.

The proposed ALU implements a complex instruction set and requires a large 48-bit opcode but on the other hand, it can perform up to 11 operations per cycle, so that the equivalent functionality would consume a far larger number of RISC instructions. Also important, the intermediate results are stored in small local registers instead of a large register file. If utilized by a large number of threads, the latency of more than 100 cycles can be hidden and especially graphics applications benefit from the improved throughput. For example, the perspective correct rasterization of a pixel with vertex colors takes six cycles and a bilinear texture sample is completed in four cycles. Also, blend operations, consisting of a read, a color operation (RGBA), and a write-back step, can be performed in two cycles.

### 5.4.3 Scheduling

The rewriting process expands and evaluates multiple functions per iteration. It represents a breadth-first search of the call tree, so that the number of tokens in the stream may grow exponentially according to the call depth. Consequently, a mechanism to throttle the expansion of the stream is necessary to keep the length below a threshold  $b$  (Figure 73).

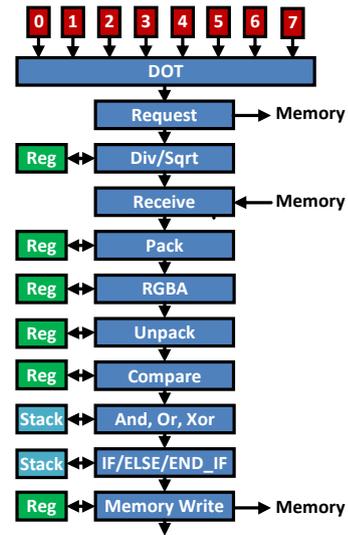


Figure 72. Pipelined execution unit.

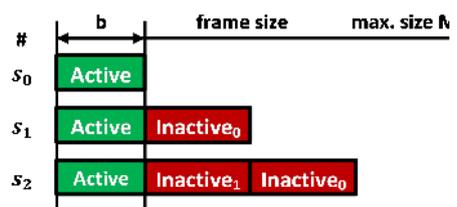


Figure 73. Expansion with constraint  $b$ .

According to the stack-based scheduling (Section 2.3.2), the current sequence is divided into an active part of at most  $b$  tokens and the inactive remainder shown in Figure 73. New tokens are created only in the active part while the inactive part is left unmodified, so that the system combines depth-first and breath-first traversal. When the size of the stream exceeds the threshold  $b$ , tokens at the end of the stream are shifted into the inactive area. Later, when the active part has completed evaluation and shrinks, the tokens at the end are moved back and can continue execution. For example, a FOR loop is interrupted and later resumed by writing the current state on the stream. Similarly, the entry point only appends new tokens to the end of the stream until the threshold size of  $b$  has been reached.

#### 5.4.3.1 *Software Development*

The SRP has a long pipeline with a complex instruction set, which leads to numerous optimization opportunities by chaining appropriate operations, but requires an extensive knowledge of its micro-architecture. Even for the assembly language of Section 3.1.3, a manual mapping into ALU opcodes would be inefficient and difficult. In particular, the assembly contains generic operations like the 32-bit integer addition *add\_i32* or a floating-point multiplication *mult\_f32*, which must be assigned to the corresponding ALU stages. In addition, this architecture does not contain temporary registers, so that complex expressions, which do not fit into the ALU scheme, must be split into multiple passes. Hence, the assembler performs the following optimization steps on the functional program:

1. LET and FOR expressions are moved into separate functions, which are later translated into distinct rewriting rules.
2. Arithmetic expressions like addition or multiplications are optimized to reduce and balance the depth of the expression tree.
3. All functions are mapped into ALU operations using a greedy algorithm.
4. Unused functions and constants are removed from the program.
5. The tree is partitioned, so that each function can be computed in one pass through the ALU and there corresponds to a rewriting function.

As a result, the expression tree of each function has a maximum depth of one node and consists solely of ALU operations, so that the program can be directly translated into the binary instruction format.

#### 5.4.3.2 *Stream Debugger*

Debugging highly concurrent programs often requires a different approach than examining singled-threaded applications. Due to the large amount of threads, the current state of a multi-core machine contains numerous instances of registers, local variables and stack frames, so that it becomes in particular difficult for the user to track the progress of the parallel execution. Similar to the verification of hardware components, the debugger often creates a trace file, which can be evaluated offline [173]. However, there are also combinations like the hardware-aware debugger for the OpenGL shading language [174], which enables source-level debugging at runtime by instrumenting shader programs.

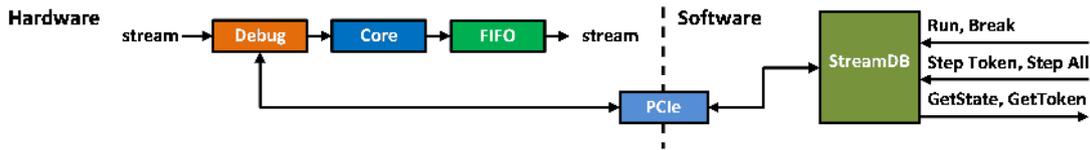


Figure 74. Schematic of the stream rewriting debugger.

For stream rewriting, the debugger (*streamdb*) is implemented as a combination of software library and a hardware component inserted into the token stream (Figure 74). It has to ability to block the stream and return the token at the current position to the software layer via an asynchronous communication channel. The debugger decodes the token into a readable representation and annotates the name of the corresponding function for CALL expressions. In addition, the stream can be advanced to the next control token, to the end of the iteration, or the device can run until the stream becomes empty, which usually marks the end of a larger computation.

The debugger is especially useful to detect dead-locks causes by infinite running threads. It shows the contents of the stream and can be attached or detached at any time. Since the debugger runs as a separate application and uses a different communication channel than the graphics driver, it does not interfere with the stream computations.

#### 5.4.4 Results

In order to demonstrate the capabilities of the proposed processor architecture, it has been prototyped using an FPGA board and connected to a PC via PCI express. Two example pipelines, implementing rasterization and ray-tracing, show the improved configurability compared to existing GPUs. A schematic of the hardware and software components in the test system is shown in Figure 75. The ML605 board

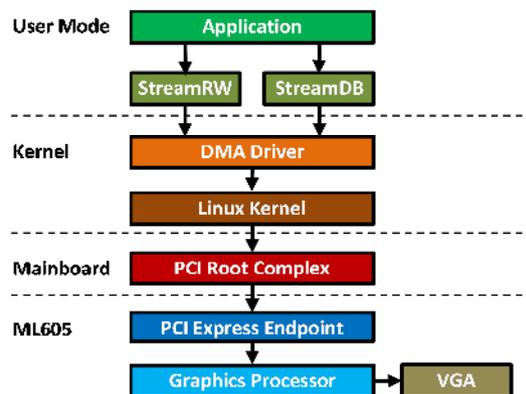


Figure 75. Setup of the complete test system.

is equipped with a PCIe connector and was plugged into a PC with an Intel Core 2 Q9650 running at 3.00GHz on a Dell OptiPlex 780 mainboard. The operation system for this test system is Ubuntu 11.10 with kernel version 3.0.0. The DMA controller shown in Figure 75 utilizes the embedded PCIe core of the FPGA to autonomously transfer data from the mainboard to the FPGA. It provides a bi-directional data stream, which is mapped to a file descriptor using a custom Linux driver, so that applications can communicate with the graphics processor using file IO. At the topmost level, the application uses two libraries *streamrw* and *streamdb* (Debugger) that provide an abstract view of the SRM.

The driver manages two buffers in system memory, which provide the space for the two FIFOs. It uses Linux kernel functions to send the physically addresses of these buffers to the DMA controller on the FPGA. At the hardware level, the PCIe packets are transmitted from the root complex on the mainboard to the embedded PCI express endpoint on the Virtex 6.

The DMA controller itself autonomously sends read request to the mainboard to transfer portions of the FIFO to the stream rewriting processor.

The proposed stream rewriting processor has been synthesized for the Virtex 6 XC6VLX240T-1FFG1156 FPGA and tested using the ML605 evaluation board from Xilinx. The resource usage for a varying number of cores is presented in Table 6. In this case, a LUT corresponds to a 6-input lookup table of the FPGA, a FF represents a single register, DSP48 is an embedded multiply-adder and a BRAM is a fixed 18K or 36K memory block. The design met timing at 200MHz.

Table 6. Resource usage of the FPGA prototype

Cores	LUT	FF	BRAM 18K/36K		DSP48
1	16,230	30,771	26	32	14
2	25,835	48,867	49	44	28
3	35,674	66,963	72	56	42
4	43,328	85,049	95	68	56

#### 5.4.4.1 Rasterization

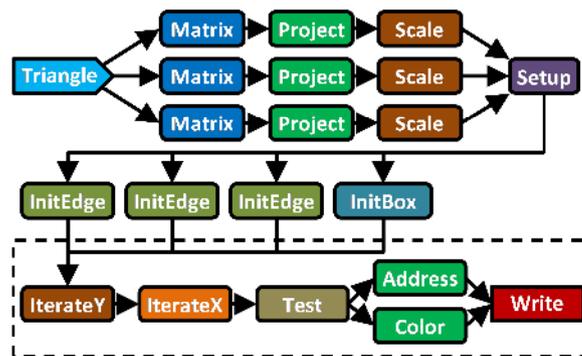


Figure 76. Example pipeline implemented for rasterization.

A simplified rasterization pipeline has been implemented to demonstrate the abilities of the stream rewriting processor Figure 76. It contains LET environments, FOR loops and the stream is split and synchronized several times. In addition, the data-rates between stages differ significantly because the triangle is sub-divided into a much larger number of pixels. The actual drawing is similar to the technique presented in [175] and works by testing pixels against the edge equations of the triangle.

The input of the rasterization pipeline is a sequence of triangles in world space. For each of the three vertices, there is a series of calculations. First, the vertex is transformed into view space using a matrix-vector-multiplication (*Matrix*). Next, it is projected on the screen and finally scaled according to the current viewport. The pipeline is split and later recombined, so that these computations can be performed in parallel.

At this position, the triangle is two-dimensional and fed into the setup stage to calculate its bounding box and three edge equations. The edges will be evaluated at every pixel but are constant for the whole triangle, so that they can be stored as part of an environment (dashed box). For comparison, pixel coordinates are represented by two words, while the

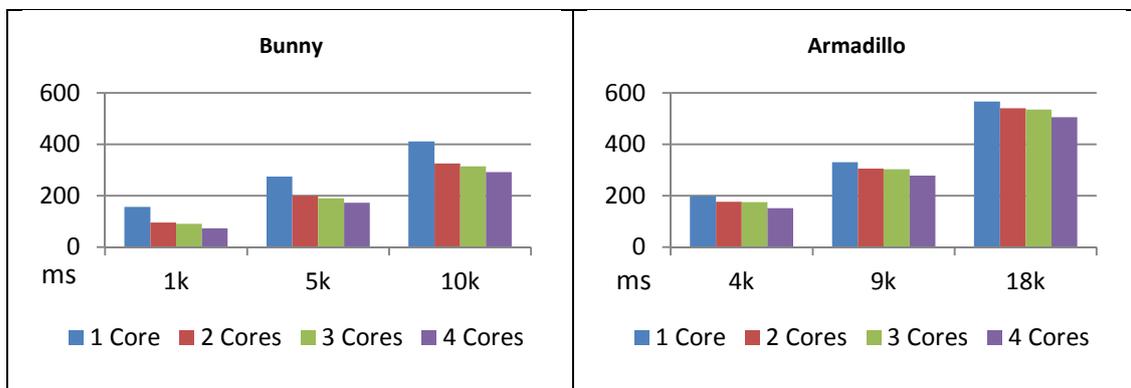
three edge equations take nine words. Hence, without the environment, the data passed between the stages and therefore also the data on the stream would be more than quadrupled. Inside the environment, two nested FOR loops iterate over the pixels of the box (*IterateY*, *IterateX*). The coordinates are tested against the edge equations and pixels outside the triangle are discarded. Color and address are calculated and the pixel is drawn into the framebuffer.

Although it is possible to implement a rasterization pipeline in CUDA, it must be re-engineered to fit into the data-parallel processor grid [176]. Contrary, the SRP architecture allows describing the stages of the rendering pipeline directly as individual functions.

In order to evaluate performance and scalability of the stream rewriting processor, various models from

- <http://simonfuhrmann.de/uni/bthesis.html>
- <http://www-i8.informatik.rwth-aachen.de>
- <http://www-graphics.stanford.edu/data/3Dscanrep/>

have been rendered at a resolution of 640x480 pixels. Figure 77 lists the render time for one frame measured at the application-level. The render time for the same type of model increases less than linear when choosing a more detailed version because most of the work is done at the pixel level, which is mostly independent from the triangle count.



Test	1 Core	2 Cores	3 Cores	4 Cores
<b>Bunny (1k)</b>	156 ms	95ms	91 ms	73 ms
<b>Bunny (5k)</b>	275 ms	200ms	190 ms	173 ms
<b>Bunny (10k)</b>	411 ms	326 ms	314 ms	292 ms
<b>Armadillo (4k)</b>	199 ms	176 ms	175 ms	151 ms
<b>Armadillo (9k)</b>	330 ms	306 ms	303 ms	278 ms
<b>Armadillo (18k)</b>	567 ms	541 ms	535 ms	506 ms
<b>Sphere (1k)</b>	115 ms	92 ms	91 ms	84 ms
<b>Sphere (5k)</b>	222 ms	168 ms	178 ms	161 ms
<b>Sphere (20k)</b>	627 ms	531 ms	544 ms	515 ms

Figure 77. Render times of rasterization test.

### 5.4.4.2 Ray-Tracing

In addition to the rasterization pipeline, also a ray-tracing test case has been implemented that sends rays from the camera into the scene to determine the color of a pixel. Shadows are calculated by tracing secondary rays from the intersection point towards the light source. While the first test uses a constant color, the second renders a textured plane and the last two examples show three lit spheres with optionally shadows (Figure 78).

The performance measurements of these tests are listed in Figure 79 and achieve an almost linear speedup in all cases. In contrast to the rasterizer, the raytracing application requires a significant higher number of per-pixel computations to calculate the intersection between a ray and a sphere. However, the costs of each pixel are roughly the same, so that the partitioning of the screen into separate areas for each core works well. In case of the *shadows* sample, the minor slowdown might be caused by the black background, which does not utilize additional shadow rays. Also, in case of a reflecting sphere, a better mechanism for global load-balancing like the *stream rewriting network* (Section 4.3), might provide an advantage.

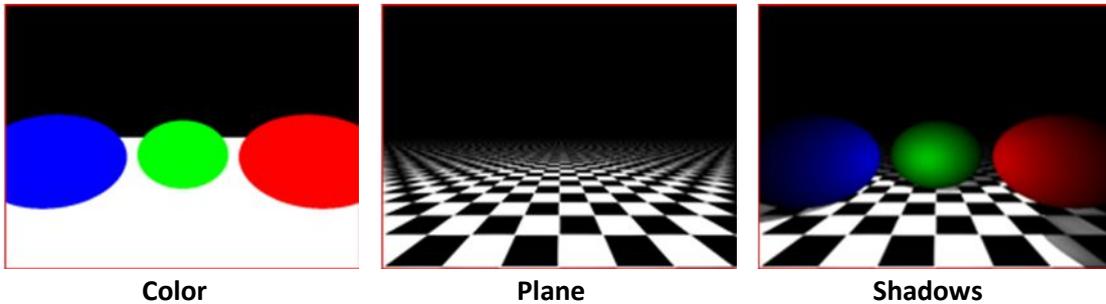
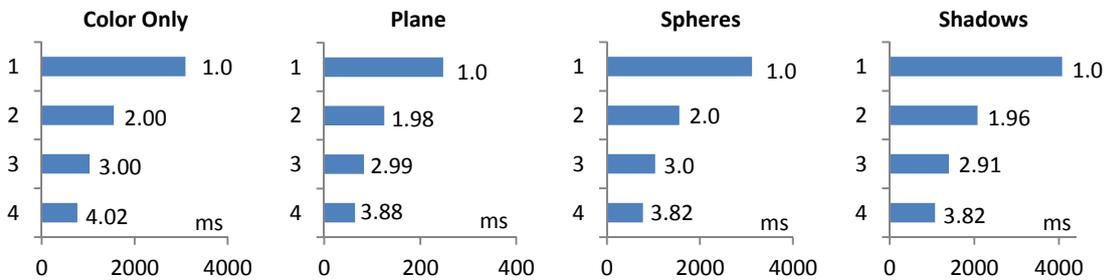


Figure 78. Screenshots of ray-tracing scenes.



Test	1 Core	2 Cores	3 Cores	4 Cores
Color Only	3093 ms	1547 ms	1031 ms	770 ms
Plane	248 ms	125 ms	83 ms	64 ms
Spheres	3117 ms	1559 ms	1039 ms	776 ms
Shadows	4079 ms	2076 ms	1401 ms	1069 ms

Figure 79. Render times of raytracing test.

### 5.4.5 Conclusion

A novel graphics processor based on stream rewriting has been presented that allows describing custom rendering pipelines as a functional program. As a result, the rendering pipeline is virtualized by reducing all functionality to rewriting operations.

Although the FPGA prototype cannot compete with commercial ASIC solutions in terms of performance, the two non-trivial examples show the flexibility, its feasibility and to some extent also the scalability of the functional SRP architecture. Most important, the design has been evaluated as part of a standard computer system, so that the measurements include all costs that would also occur for a commercial graphics device. The experimental results clearly indicate the success of the architecture for an authentic test case. Thus, the concept of stream rewriting via pattern matching has been proven as a viable solution for the problem of scheduling and synchronization in a multi-threaded environment.

The next step for this architecture would be the development of a compiler for a more high-level functional language that could be layered on top the existing assembly. In addition, it would be also interesting to combine the functional concept with the traditional OpenGL pipeline.

### 5.5 SIMT Shader Core

In this section, the architecture of a SIMT based design is presented and similar to the functional approach (Section 5.4), its performance is analyzed using a FPGA prototype. The main innovation of this architecture are the usage of a *global dispatcher* (Figure 78), which redistributes the token stream in each iteration and therefore improves the load-balancing. The shader core (Figure 79) executes a same instruction of four threads in parallel (SIMT) and contains a local register file for intermediate results to reduce the traffic on the stream. The costs of read-modify-write operations are reduced by storing pixels into an on-chip tile buffer (Figure 78) before they are flushed into the global memory. As result, the SIMT architecture achieves better scalability than the previous design and has been evaluated using a larger number of examples (Section 5.5.4).

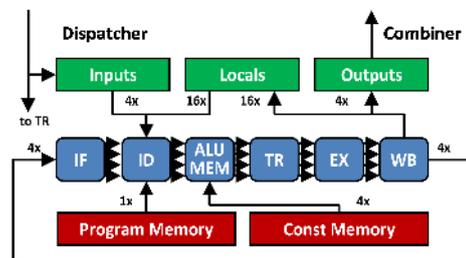
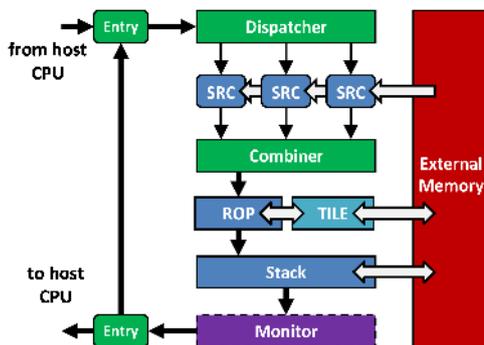


Figure 80. Architecture of the stream rewriting processor. Figure 81. Design of a stream rewriting core (SRC).

### 5.5.1 Overview

The concept of stream rewriting leads to an efficient hardware architecture for graphics processing because all rewriting operations modify only local, compact and non-overlapping regions of the stream. For comparison, one of the general purpose many-core systems presented in Section 4 organizes several rewriting cores (SRC) sequentially as a ring, so that all tokens, even if they are not modified, have to pass through the execution units of a core before they can be accessed by its successor. Therefore, it takes as many cycles to execute a function locally as it would require passing it to another core. The results of the ring architecture show scalability but load-balancing becomes challenging and the architecture mainly benefits from pipelining (Section 4.4).

Contrary, in this architecture (Figure 78), the stream is also circulating in a ring but the stream rewriting cores are arranged in parallel. Only several distinguished hardware units like the entry and exit point, the *raster output stage* (ROP), the *stack*, and a *performance monitor*, are chained. Tokens sent by the host CPU correspond to draw calls and are fetched by the entry point until the ring is filled with a minimum work load. Similarly, the exit point allows results, which should not be written into the framebuffer, to be returned to the CPU. In order to avoid bottlenecks and to keep all components utilized, the data width of the ring is chosen large enough to handle several tokens (here: 3) per cycle.

The actual rewriting is performed by the group of parallel *stream rewriting cores* (SRC). For clarity, the schematic (Figure 78) shows only three cores but actually, the architecture has been evaluated for up to eight cores. A dispatcher creates threads from executable rules, which are then passed to the array of stream rewriting cores. Finally, the results are reassembled in the correct order by the combiner. The ROP stage performs blending and depth-testing on the incoming pixels and stores them in a local memory tile (here 512kb), which is later flushed into external memory. In particular, the pixels are described as three consecutive OUT tokens (2.5.1) storing the address, color and depth values. Also, there is a performance monitor that records the number of executed tasks and tokens per iteration.

Due to the recursive expansion, the size of the stream varies frequently and may quickly exceed the capacity of the available on-chip memory. As specified for global scheduling in Section 2.3.2, the stack stores the last part of long streams in the external memory, so that only the active part of the stream and at most  $b$  tokens are circulating in the ring. Consequently, this architecture utilizes the post-order format of the stream grammar.

### 5.5.2 Stream Rewriting Core

Each stream rewriting core (Figure 81) contains a ring pipeline of several stages, three register files and connections to the dispatcher and the combiner. Executable rules are received from the dispatcher and translated into corresponding tasks, circulating in the pipeline. For non-matching parts of the stream, a pass-through program is executed, which simply copies the input tokens. There are separate register files for inputs, local variables and outputs, so that the decoding of rules, the execution of tasks and the write back of results can occur in parallel. Beside the instruction memory, also a limited set of constants are available to store global parameters like the current camera matrix or light vectors.

Table 7. Instruction Set of the SRC (excerpt).

Name	Function	Description
<b>Integer</b>		
DOT_I32 $r, a, b, c, d$	$r \rightarrow ab + cd$	dot product
ADD_I32 $r, a, b, c, d$	$r \rightarrow a + b + c + d$	sum of four
SEQ_I32 $r, a, b, c, d$	$r \rightarrow a = b ? c : d$	set on equal
SLT_I32 $r, a, b, c, d$	$r \rightarrow a < b ? c : d$	set on less
<b>Float</b>		
DOT_F32 $r, a, b, c, d$	$r \rightarrow ab + cd$	dot product
SEQ_I32 $r, a, b, c, d$	$r \rightarrow a = b ? c : d$	set on equal
SLT_I32 $r, a, b, c, d$	$r \rightarrow a < b ? c : d$	set on less
RCP_F32 $r, a$	$r \rightarrow 1/a$	approximate reciprocal
RSQ_F32 $r, a$	$r \rightarrow 1/\sqrt{a}$	inverse square root [177]
<b>Memory</b>		
READ $r, a$	$r \rightarrow \text{memory}[a]$	read 32-bit word
SAMPLE $r, u, v, tx$	$r \rightarrow \text{sample2d}(tx, u, v)$	sample texture $tx$ at $(u, v)$
<b>Control</b>		
EMIT $a, b$	append token $a$ to stream if $b \neq 0$	conditional write
EXIT	exit program if $a \neq 0$	conditional exit

Table 1 lists some of the instructions supported by the SRC. Each of them can refer up to four operands  $a, b, c, d$  which are either fetched from the input, local or constant register file. The result  $r$  of an ALU operation is stored in the local register file. In addition, there are two control flow instructions: An EMIT instruction appends a token to the output stream and is used to produce the results of a rewriting rule. END either exits the shader conditionally or terminates the execution immediately if operand  $b$  is zero. There are no complex control flow instructions but instead this behavior can be emulated by using the pipeline fragments shown in Section 3.4 to implement branches and feed-back loops similar to functional programming. As an advantage, dynamic flow control can be scheduled globally across all cores.

The actual program execution takes place in the ring pipeline of  $n$  stages (here  $n = 32$ ), which are grouped into the blocks instruction fetch (IF), instruction decode (ID), ALU, execute (EX), write-back (WB) and a stage for thread management (TR). For each instruction of a thread, the IF stage fetches the current command word, which is then decoded in the ID stage that also loads the corresponding inputs, locals or constant values. The next stage is the combined ALU and memory stage, which implements deeply pipelined integer and floating-point operations as well as load instructions and texture fetches. Then, the execute stage (EX) handles control flow and finally, the write-back stage (WB) stores the results either in the local or the output register file.

Each hardware thread is represented by a tuple describing the execution context and the associated resources of a thread. It contains the current instruction pointer ( $ip \in \mathbb{Z}$ ) as well as the number ( $size \in \mathbb{Z}^3$ ) of assigned input, output and local registers:

$$t = (id, ip, addr, size) \in THREADS \subseteq \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}^3 \times \mathbb{Z}^3 \quad (88)$$

The design of the SRC resembles a barrel processor and supports multiple hardware threads for improved utilization and latency. In fact, there are two levels of concurrency. First, each stage can store its own thread, so that the processor can execute up to  $n$  unrelated rules in parallel. In addition, the pipeline also supports SIMT parallelism up to a level of  $m$  (here  $m = 4$ ) to account for the data parallelism of computer graphic tasks. As a consequence, each stage can process up to  $m$  threads in parallel as long as they execute the same rewriting rule. Hence, the maximum number of active threads in the pipeline is  $n \cdot m$  (here 128). For comparison, in terms of CUDA, the warp size of the SRC is  $m$  and its functionality approximately corresponds to  $m$  CUDA cores.

### 5.5.3 Shader Programs

The stream rewriting processor (Figure 80 on page 103) emulates a graphics pipeline by modifying a token stream. Each stage corresponds to a rewriting rule, so that the complete pipeline can be described as a set of functions  $f_i: \mathbb{Z}^n \rightarrow \Sigma^m$  with  $n, m \in \mathbb{N}_0$  (see Section 5.3). Conceptually, the SRP contains a single program memory as an array of instructions:

$$program: \mathbb{Z} \rightarrow INSTRUCTION \tag{89}$$

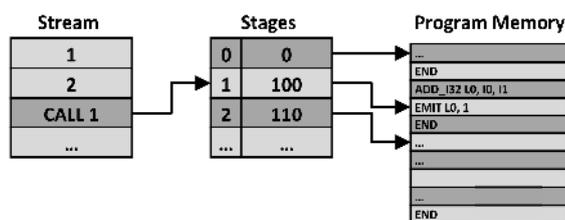


Figure 82. Linkage of CALL expressions and shader stages

A pipeline stage is defined by binding a logical stage identifier  $i \in \mathbb{Z}$ , which is also used in the CALL token, to identify the starting address of the corresponding rewriting rule:

$$stage: \mathbb{Z} \rightarrow \mathbb{Z} \tag{90}$$

Hence, an executable rewriting rule with token CALL  $i$  causes the SRC to create a new thread at address  $stage(i)$ . While the thread executes, it produces a variable amount of output tokens via the EMIT instruction, which are then used to replace the original CALL expression. The relation between the stage identifier  $i$  and the associated shader is also illustrated in Figure 82. Individual shaders are stored in the flat program storage. The start address of each shader is defined by the stage table, while the last instruction of a program is always the *end* opcode. This extra level of indirection permits to rebind logical shader stages without modified the program memory. In the current implementation of the SRP, the minimum size of the stage table contains 64 entries to define up to 64 logical stages. An example is given by the shader program on the next page, which is stored at address 100 and returns the sum of both arguments.

```

100: ADD_I32 L0, I0, I1 # add inputs I0, I1
101: EMIT   L0, 1      # emit local L0
102: EXIT   0          # exit shader

```

By setting  $stage(4) := 100$ , this shared is invoked using the CALL expression with index 4:

$$\langle \dots, 1, 2, 4, CALL, \dots \rangle \rightarrow \langle \dots, 3, \dots \rangle \tag{91}$$

Currently, the system is programmed from a host CPU using a low-level hardware-specific library. Similar to CUDA or Direct3D, calls are submitted using the dispatch function, which takes the id of the stage, an optional rectangle and a list of  $n$  arguments:

$$dispatch(stage, rect, args) \tag{92}$$

Technically, dispatch sends the following stream to the stream rewriting processor:

$$\langle arg_0, \dots, arg_{n-1}, stage, CALL \rangle \tag{93}$$

In the next section, a prototype implementation using this API is presented.

### 5.5.4 Implementation

In order to evaluate the concept of stream rewriting for graphics processing, the SRP has been implemented using Verilog and synthesized for the VC707 prototyping platform from Xilinx, which features a Virtex 7 XC7VX485T FPGA that can be re-configured with a user-defined netlist. Most important, the board also has a PCIe connector and can be plugged into a PC for communication between the SRP and a graphics application. The hardware and software setup of the complete test system consists of a PC with a dual core 3.20GHz Pentium 4 CPU with Ubuntu (kernel 3.2).

The SRP has been evaluated with eight SRCs in a  $8 \times 4$  configuration. The design met timing at 200MHz and required 173,774 6-input lookup tables, 224,757 registers, 396 embedded multiply-adders, 374 BRAM blocks. Individual utilizations are listed in Table 8.

Table 8. Resource usage per component

Component	LUT	Slice Registers	BRAM	DSP48E1
<b>System (8 Cores)</b>	174,438	225,493	374	396
<b>SRC Pipeline 0</b>	17,768	21,812	24	48
<b>Dispatcher</b>	1,144	1,850	0	0
<b>Combiner</b>	3,645	4,368	0	0
<b>Stack</b>	2,101	2,478	2	0
<b>ROP + TILE</b>	3,557	8,785	128	8
<b>MIG</b>	10,156	12,909	0	0
<b>PCIe Interface</b>	1707	2273	14	0
<b>HDMI Interface</b>	480	838	7	0

### 5.5.4.1 Generic Performance Tests

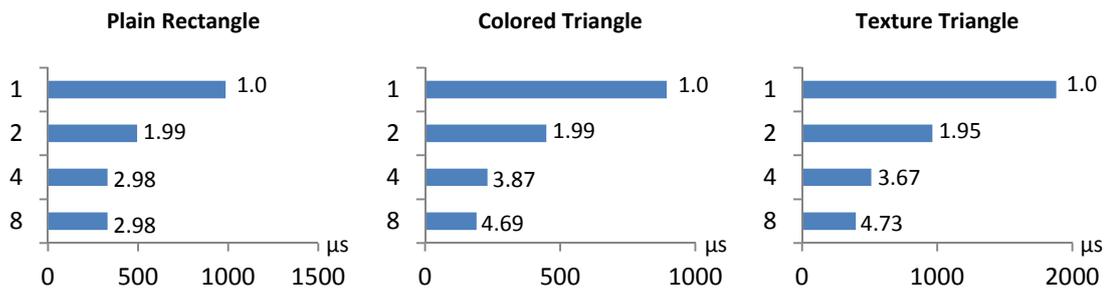
To estimate the raw performance and scalability of the design, three generic test cases are run first, and record the required cycles, processed tokens, executed threads and the maximum stack size. All tests are rendering into a tile of 256x256 pixels and the measurements do not include the effort to initialize or flush the buffer. The tests draw a plain rectangle (256x256) as well as two rectangular triangles with interpolated colors and texturing.

```
DOT_I32 L0, $X, A0, $Y, B0    # edge 0
DOT_I32 L1, $X, A1, $Y, B1    # edge 1
DOT_I32 L2, $X, A2, $Y, B2    # edge 2
SLT_I32 L3, L0, C0, 0, 1      # if L0<C0
SLT_I32 L3, L1, C1, 0, L3     # or L1<C1
SLT_I32 L3, L2, C2, 0, L3     # or L2<C2
EXIT L3                       # the pixel is outside => exit
...
EMIT color, 1                  # write PIXEL
EMIT depth, 1                  # token to the
EMIT PIXEL($X, $Y), 1         # stream
```

Figure 83. Rasterization Shader

The rectangle uses a short shader, which emits a single pixel token per iteration, to measure the throughput of the system. Likewise, the triangle shader (Figure 83) contains arithmetic instructions for attribute interpolation and outputs only pixels, which are contained within the triangle. It uses instancing (Section 2.4.1) to enumerate and optionally discard uncovered 16x16 blocks [176] [178]. The source code shown in Figure 83 checks if a single pixel  $(x, y)$  is inside the triangle by testing the conditions  $A_i \cdot x + B_i \cdot y \geq C_i$  against the three edges ( $i \in Z_3$ ) of the triangle.

The results of these tests are presented in Figure 84. For one core, the plain rectangle takes three cycles per pixel because each core can emit one token per cycle and each pixel is encoded using three OUT tokens (2.5.1) in hardware for address, color and depth. The number of tokens has been recorded as zero because in this minimalistic example, they are immediately processed by the ROP stage and never reach the monitor (Figure 80). Since the global stream has a width of three tokens in this implementation, the rectangle test is bandwidth limited for more than three cores. Contrary, the colored and textured triangles require more work per pixel, so that the ratio between communication and computations is more convenient. Though, the speed-up also drops for more than four cores with 16 pipelines but the more expensive textured triangle scales slightly better.



Cores (SRC)	1	2	4	8
<b>Plain Rectangle</b>				
Cycles	197,185	98,885	66,121	66,121
Tokens	0	0	0	0
Max. Stack	0	0	0	0
Threads	65,536	65,536	65,536	65,536
<b>Colored Triangle</b>				
Cycles	178,904	89,856	46,180	38,154
Tokens	3,345	3,345	3,345	3,345
Max. Stack	246	246	246	246
Threads	38,912	38,912	38,912	38,912
<b>Textures Triangle</b>				
Cycles	376,410	192,919	102,579	79,524
Tokens	4,257	4,257	4,257	4,257
Max. Stack	1,152	1,152	1,152	1,152
Threads	38,912	38,912	38,912	38,912

Figure 84. Performance of generic tests

#### 5.5.4.2 Recursion

In addition, the recursive capabilities and the performance of the stack are evaluated by drawing a rectangular 2D triangle through repeated subdivision. The triangle is recursively split into four sub-triangles until the pixel-level has been reached, so that each iteration roughly quadruples the number of tokens. The results have been recorded for various triangles sized and are shown on the top of Figure 85.

Obviously, the number of executed threads, and therefore also the number of clock cycles, grows exponentially when the edge length of the triangle is doubled. Likewise, multiple cores offer some scalability but cannot account for the exponentially grow of threads. However, the maximum required stack size increases only linearly. Similarly, a trace of the stack size and the number of active tokens is shown at the bottom of Figure 85. Despite the frequently varying size of the stack, the number of circulating tokens remains nearly constant at the requested level (here  $b = 3072$ ).

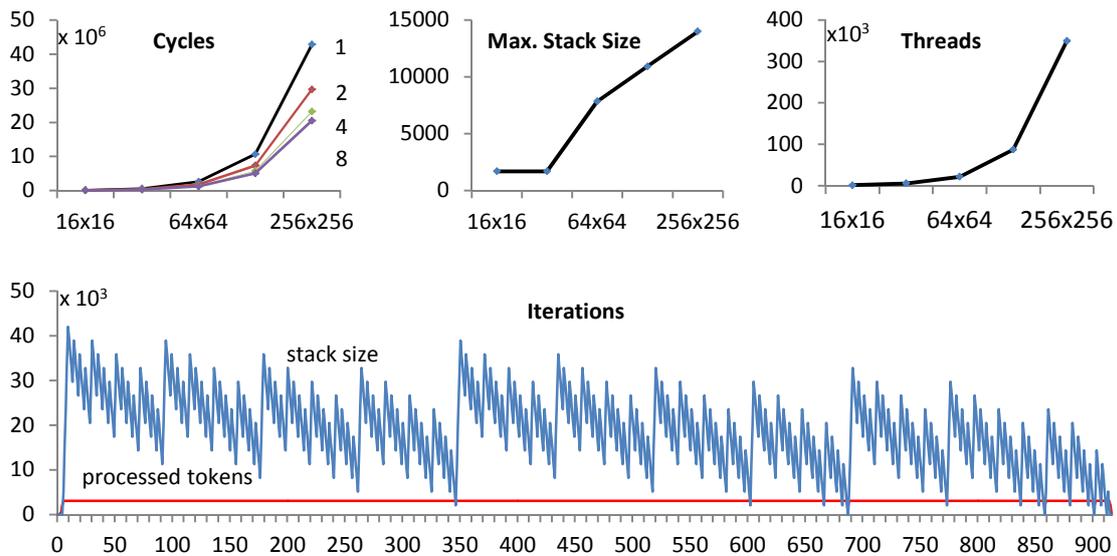


Figure 85. Results of recursive tests.

### 5.5.4.3 Graphics Pipeline

Based on the generic examples, a more complex pipelines is built to estimate performance and flexibility of the system. In addition, the SRP architecture using a global dispatcher and parallel working cores is compared to much simpler ring pipeline, which arranges the SRC sequentially and uses a FIFO instead of a stack. All tests render into a framebuffer of 640x480 pixels and six tiles (256x256). In contrast to the generic test, the complete rendering time of a frame is measured. The results are listed in Figure 86 and Figure 87 for different numbers of cores, while the corresponding screenshots are shown in Figure 88 and Figure 89. The models used is this test, the bunny, dragon and armadillo meshes are taken from the *Stanford 3D Scanning Repository*:

<http://www.graphics.stanford.edu/data/3Dscanrep/>.

First, a rasterization pipeline has been implemented with perspective correct attribute interpolation [179] and triangle setup as custom shader stages. It supports both vertex colors and per-pixel lighting by interpreting the color as a normal vector. In addition, Phong Tessellation [180] is added as a set of recursive rules and interpolates a surface based on the vertex normals. In this example, a cube is subdivided one, two or four times by a recursive shader program. Further, a torus is rendered using vertex colors, per-pixel lighting or texture mapping. In addition, a test using small triangles (3 pixel) is run to estimate these non-pixel-related costs like the triangle setup. It can be observed that most tests scale with the number of cores and that the parallel arrangement is slightly more efficient but the results also expose several scalability issues, which are discussed in the next section.

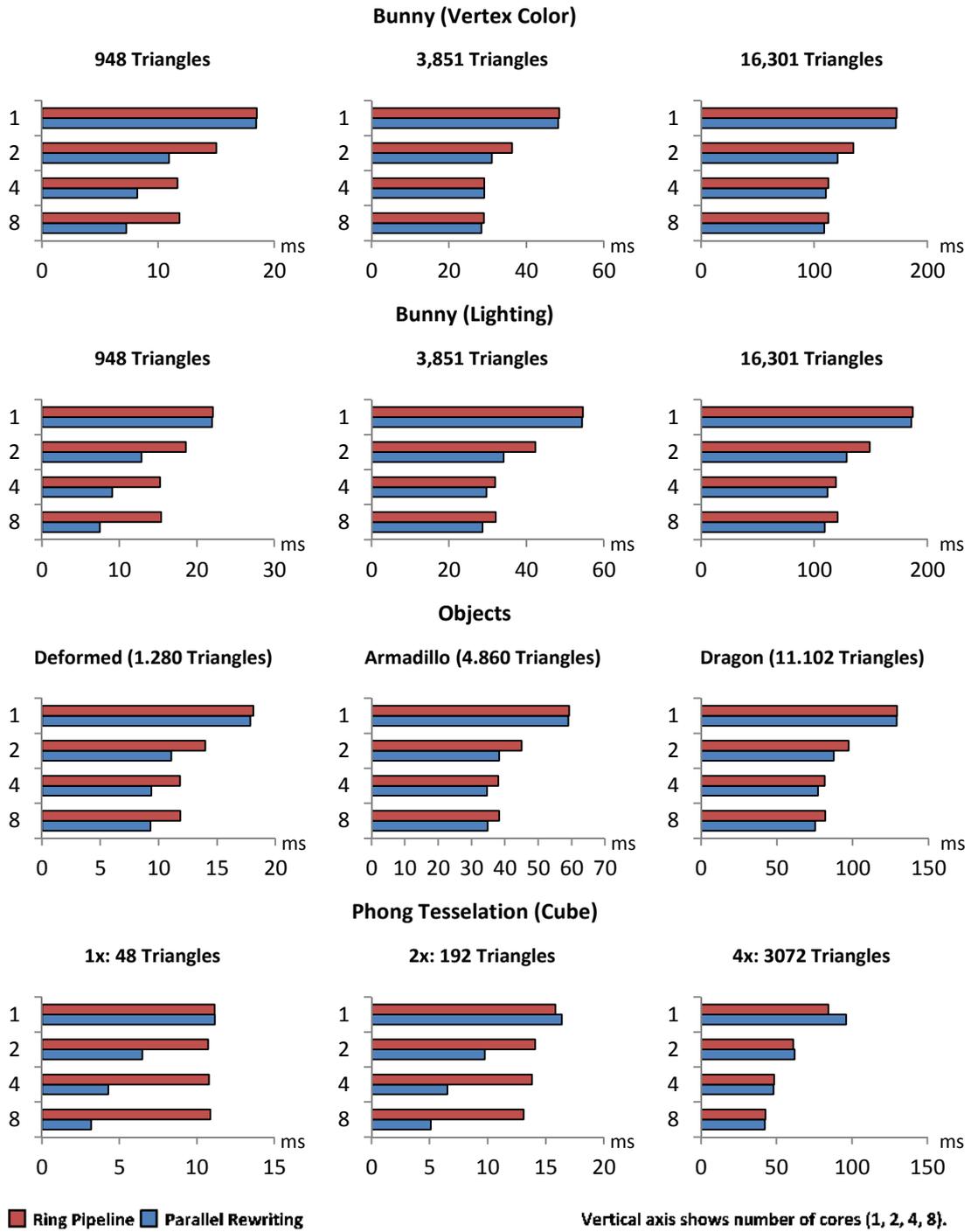


Figure 86. Performance of rendering tests.

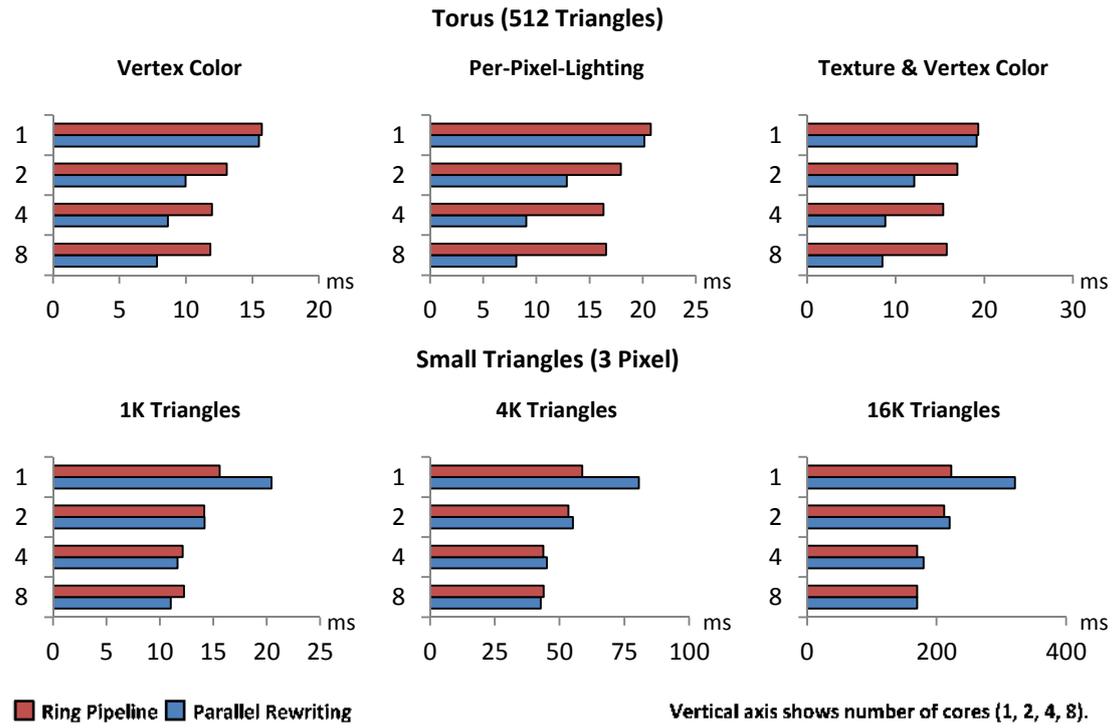


Figure 87. Performance results of rendering tests.

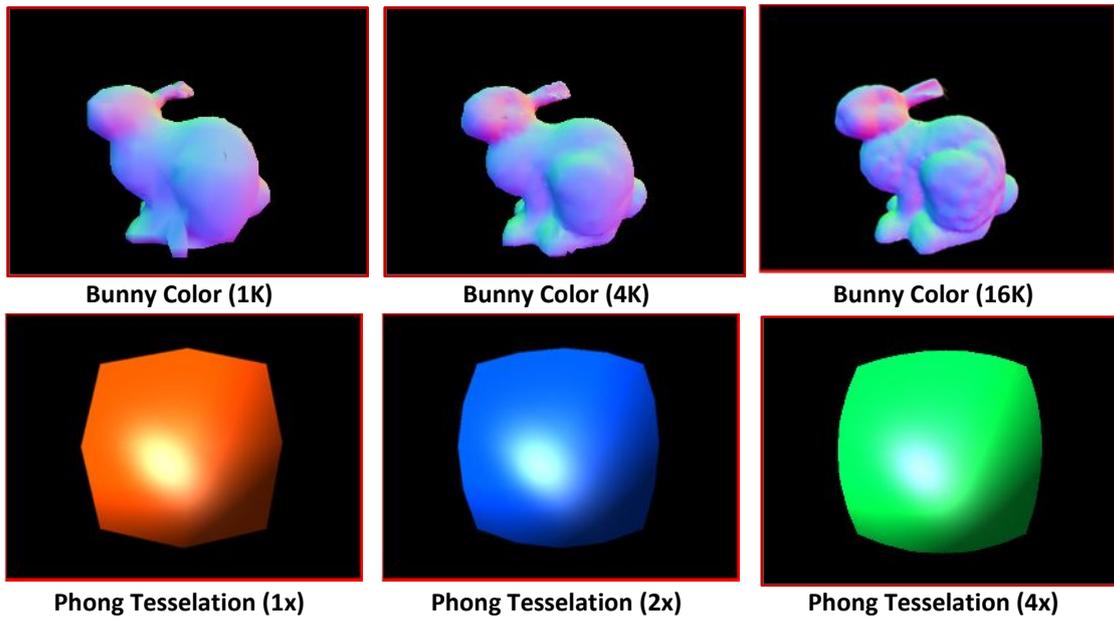


Figure 88. Screenshots (640x480) of rendering tests.

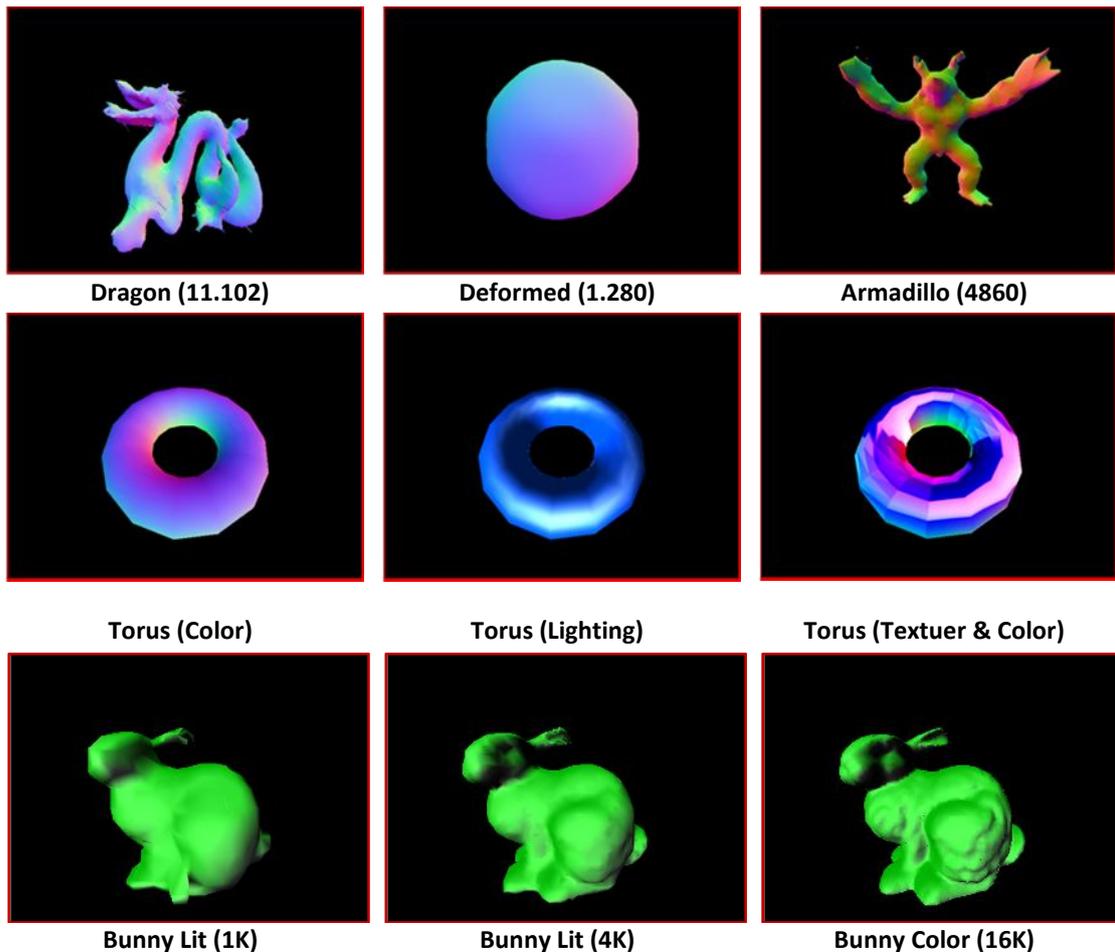


Figure 89. Screenshots (640x480) of rendering tests.

### 5.5.5 Discussion

The performance evaluation of the SRP prototype and indicates some limitations of the proposed architecture and leads to the following results:

- **Stream Management**

The throughput of the stream might easily become a limitation of this architecture, so that it has been made wide in the presented implementation. The combiner (Figure 80) reassembles multiple narrow streams, which are coming from the SRCs, into a wide stream in parallel. Similarly, the ROP unit also reads three tokens at once and can therefore write one pixel per cycle. In addition, its capacity could be easily extended by widening the stream to six or even more tokens. For this prototype, a raw fill rate of 200 MPixel/s is sufficient since the focus has been put on the flexibility of the shader cores (SRC). However, the generic tests in Section 5.5.4.1 show that for simple shaders, which output a constant color, the stream still becomes a bottleneck and that, according to the stream width of three tokens, their speed-up is also limited by three.

- **Stack-based Scheduling**

In comparison to the FIFO, the main advantage of the stack is the reduced memory usage. However, it has a lower raw throughput due to implementation details. Hence, for one core, there is just has the overhead of the stack without the benefits of the improved scheduling, which can be seen in the *Phong Tessellation* test case.

- **Dispatcher**

The dispatcher is currently the slowest component because it has a maximum throughput of one token per cycle. Hence, the usage of instancing provides a significant speed-up because it can reuse input data, which allows creating up to four SIMT threads per cycle. Currently, geometry processing has been implemented via CALL tokens, so that is does not utilize SIMT and also suffers from stream contention. Contrary, the pixel operations are specified as instanced calls (2.4.1) and dispatched at a much higher rate. Therefore, the experiments in Figure 86 and Figure 87 show a better scalability for models using fewer and large triangles.

This section presents the evaluation of stream rewriting for graphics processing that combines a traditional SIMT approach for local threads at each core with a global scheduling algorithm for stream management. As a result, it effectively solves many existing limitations and supports both programmable computations and communication. In the next section, the flexibility of this approach is extended towards the general purpose processors.

## 5.6 General Purpose Graphics Processor

Based on the experimental results of the functional processor and the SIMT design, a third stream rewriting processor (SRP) for graphics is developed in this section. Instead of specialized shader cores and microcode, this architecture utilizes general purpose processors, which can be programmed in ANSI C and are optimized for instruction-level parallelism. The flexibility of this architecture is further

enhanced by a two-level cache hierarchy, custom atomic operations through dynamic binding (Section 2.5.2) and support for stream compression (Section 2.4). In addition, a functional simulator and an OpenGL extension for stream rewriting simplify the development of test cases. As a result, these novel capabilities greatly reduce the complexity of several advanced rendering methods, like order-independent transparency via *K-buffering*, *path-tracing* and the recursive generation of *procedural geometry*. Although these techniques require significant effort to run efficiently on current GPUs, the experiments show that they fit well into the concept of stream rewriting and thus benefit from the flexibility this stream rewriting processor (SRP).

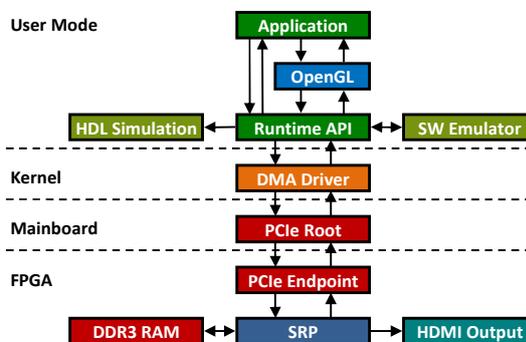


Figure 90. Components of the SRP test system.

### 5.6.1 Overview

The complete test system of the SRP is shown in Figure 90 and consists of several hardware and software layers. A graphics application can program the SRP either via a runtime API or by using a custom OpenGL extension. In addition to the hardware SRP, there are two simulated devices:

1. The HDL simulation is written in Verilog and based on the same source code as the hardware to provide a slow but cycle-accurate model. Hence, it is mainly used to verify correct functionality of the hardware SRP.
2. The software emulator utilizes a much faster but less precise functional model, which is better suited for complex test cases. It is integrated into the driver stack and performs binary translation of the shader at runtime, so that stream rewriting applications can be run at decent speed without modifications.

A hardware prototype of the SRP has been implemented on the VC707 FPGA board from Xilinx and runs at 200MHz. Most important for graphics processing, the board includes 1GB of 800MHz DDR3 memory, a PCIe slot, and a HDMI output. In this configuration, the initial token stream of the application is copied to the SRP via the PCIe interface. Similar to the other PCIe interfaces, a kernel mode driver allocates a ring buffer in system memory and the SRP fetches the commands autonomously via DMA transfers. In addition, read-backs from the host CPU to the SRP are implemented in a similar way.

Table 9. Resource Usage of the FPGA (Relative Usage in %)

#Cores	LUT	REG	BRAM	DSP48E1
<b>1</b>	27,909 (9%)	41,234 (6%)	7/137	10
<b>2</b>	34,834 (11%)	49,051 (8%)	7/147	20
<b>4</b>	49,084 (16%)	65,767 (10%)	8/181	40
<b>8</b>	76,405 (25%)	96,854 (15%)	8/221	80
<b>16</b>	128,583 (42%)	159,033 (26%)	8/301	160
<b>32</b>	240,188 (79%)	282,591 (46%)	7/447	320

The entire system has been synthesized, placed and routed for different configurations of the SRP, which consists of up to 32 stream rewriting cores (SRC). The resource utilization of each setup is listed in Table \ref{tab:resources}. The resources are distinguished into six-input look-up tables (LUT), registers (REG), block rams (BRAM) with 18/36kbits and 25x18 bit multipliers (DSP48E1). In the next sections, the components of this SRP are described in more detail.

### 5.6.2 Stream Rewriting Processor (SRP)

The data flow model of the *stream rewriting processor* (Figure 91) consists of concurrently working hardware blocks that are connected via queues (arrows), so that all steps of the stream rewriting process can be optimally performed in parallel. In general, the structure resembles the abstract design of current GPUs [9] and contains a decoder for work items and several stream rewriting cores (SRC). The stream rewriting network (green) with routers (R) transports the token stream. In addition, each core has also access to the external memory via a separate network (red) with routers (M) and a shared L2 cache (256 KB).

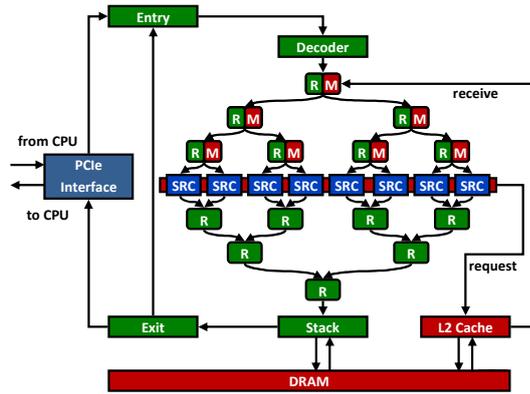


Figure 91. Architecture of the stream rewriting processor.

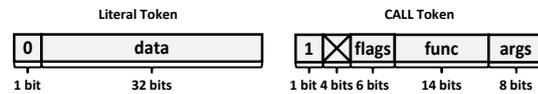


Figure 92. Binary token format of the SRP.

Here, the single most significant difference to established architectures is the global feedback loop at the left side, which allows the SRCs to dynamically generate work-load during the execution of a shader. The rewriting process starts at the PCIe interface with the initial stream  $s_1$  received from the CPU. The entry point ensures that arriving tokens are always appended to the end of the existing stream and truncates the input stream if the optimal size of  $b$  tokens has been reached (Section 2.3). Executable CALL expressions are detected by the decoder and are then rewritten by the tree of stream rewriting cores (Section 5.4.2). The cores are connected through a hierarchical network of routers (R), which split and reassemble the resulting token stream in the correct order. Eventually, the stream is reaching the stack, which implements the scheduling algorithm presented in Section 2.3 and at the exit point, results are removed from the stream and sent back to the CPU. Contrary, pending computations are passed to the entry point to proceed the rewriting process with another iteration. Memory requests from the SRCs are collected using a 512 bit wide ring-bus and routed through a direct-mapped L2 cache in order to reduce the latency of external memory access. The cache utilizes a write-back strategy and contains 4096 blocks of 64 bytes. Eventually, the results of a read operation are sent back to the SRC via second hierarchically network (red) with routers (M).

The binary token format is illustrated in Figure 92 and consists of 33 bits. The most significant bit distinguishes between literal values and CALL tokens, which contain the number of arguments, the address of the corresponding function and additional flags.

### 5.6.3 Stream Rewriting Core (SRC)

While the stream management and decoding is well-suited for dedicated hardware blocks and pipelining, shaders are written in software to combine the performance advantages of fixed hardware with the flexibility of general purpose computations. The *stream rewriting core* (SRC) is designed as a general purpose processor with additional streaming rewriting instructions (Table 10) but in contrast to the CUDA multiprocessor, it is optimized for single-thread performance to

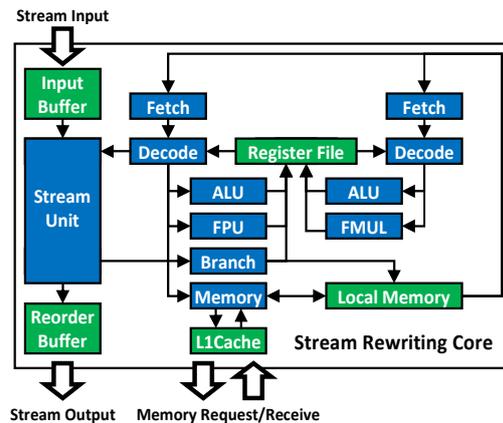


Figure 93. Schematic of the stream rewriting core.

handle frequently varying and incoherent streams. Similar to the echelon research GPU from NVidia [181], it runs only four unrelated threads to execute different code paths without performance loss. Since the results of a rewriting operation must be reassembled into a single stream, a moderate number of threads also helps to minimize the capacity of the required input and reorder buffers (255 tokens per thread). In addition, the SRC also contains a dual-issue pipeline as well as an 8-way associative L1 cache (Figure 93).

The instruction-set of the SRC is mostly binary compatible to the *MicroBlaze* architecture from Xilinx, so that the existing GCC tool-chain can be employed for the compilation of shader programs.

Table 10. Instructions for Stream Rewriting

OpCode	Description
<b>get(x)</b>	Read a literal value from the stream and store it in variable $x$ .
<b>put(x)</b>	Write $x$ as a literal on the stream.
<b>call(func, argc)</b>	Generate a CALL token for function <code>func</code> with <code>argc</code> arguments.

However, the internal design of the SRC is fundamentally different and optimized for moderate instruction-level parallelism. Four hardware threads are cycling through the processor in order to compensate the latency of floating-point operations without the costs of dynamic scheduling. Also, the FETCH stages load two subsequent opcodes from the 32KB local memory, which is shared among all threads and also contains a separate stack for each thread as well as local data. In the DECODE stages, up to five operands are read from the register file with  $4 \times 32 = 128$  entries.

The two ALU stages perform integer arithmetic, the full FPU is responsible for floating-point addition, multiplication, division, and square-root but there is also a second FMUL unit, so that two multiplications can be issued per cycle. In addition, the branch unit handles control flow and computes the address of the next instructions to be fetched. Besides floating-point division and square-root, most of the functional units are pipelined, so that the SRC can reach a maximum throughput of two instructions per cycle.

Each of the four threads owns a small cache with a size of 512 bytes, which are divided into eight fully associative blocks. Since the rendering of a frame usually requires random access of multiple source and destination buffers, associativity has been preferred in favor of the total cache size. Due to the small number of threads, a cache becomes necessary in this design to reduce the costs of an external memory access ( $> 40$  cycles latency).

For stream rewriting, the SRC implements the three additional opcodes *get*, *put* and *call*, described in Table 10. In particular, *get* reads a literal token from the stream, *put* emits a literal and *call* generates a new function call. Initially, the STREAM UNIT (Figure 93) scans the stream for executable tasks and invokes the corresponding function, whose entry point is an address in local memory and given by the stage identifier  $i$ . The function itself is then responsible for fetching arguments and producing output tokens. For instance, the following shader with  $f: \mathbb{Z}^2 \rightarrow \mathbb{Z}$  returns the sum of its two arguments:

```
void f()
{
    int x, y;
    get(x);    // read argument x
    get(y);    // read argument y
    put(x+y);  // write sum of x and y
}
```

More complex examples are presented in Section 5.6.6. Instead of specifying the stream opcodes manually, it would be also possible to let a high-level compiler map the basic blocks into individual shaders (Section 3.3).

Table 11. Runtime API of the SRP

Function	Description
<b>Stream</b>	
put(x)	Append literal to the stream.
call(func)	Append CALL token to the stream.
receive(size)	Read results from the SRP.
<b>Synchronization</b>	
sync()	Ensures that all preceding calls are finished.
flush()	Start rewriting of the token stream.
<b>Memory</b>	
mem_alloc(size)	Allocates <i>size</i> bytes of global memory.
mem_free(ptr)	Frees a global memory region starting at <i>ptr</i> .
mem_write(addr,size, data)	Write into global memory.
mem_read(addr,size)	Read <i>size</i> bytes from global memory starting at <i>addr</i> .
mem_set(addr,size, data)	Clear global memory region starting at <i>addr</i> with <i>data</i> .
<b>Utility</b>	
display(addr)	Display video output at the address <i>addr</i> .
config(addr, size, data)	Write into local memory of all SRCs.

#### 5.6.4 Runtime API

The runtime library abstracts the software interface to the SRP, which consists of the functions listed in Table 11. In particular, the main purpose of this library is to build the command stream for the SRP, to provide basic memory management, and to invoke certain system calls and utility functions. Therefore, more high-level libraries like OpenGL can omit hardware details and token encoding. In addition, the runtime API works mostly transparent with all three device types, so that test cases can be quickly brought from the emulator to the real hardware with *minimal* effort.

The stream instructions are similar to the opcodes of the SRP itself (Table 10). Here, *put* writes a literal and *call* inserts an invocation into the command stream. Likewise, results of a computation can be retrieved by *receive*. As an important implementation detail, the stream is buffered locally before it is written to the kernel mode driver. In addition, the runtime library is usually linked statically, so that the corresponding functions can be inlined to reduce the overhead of this immediate style API.

The runtime API supports two different instructions for synchronization. A call to *flush* transfers the current stream to the SRP and starts the rewriting process. Hence, it is normally called at the end of a command sequence and also before receiving data from the SRP. Since there might be still tokens from a previous call circulating in the ring (Figure 91), there is a second function *sync*, which ensures that all preceding tasks are rewritten completely. For this purpose, it blocks the entry point until the ring becomes empty and is usually called before modifying global configuration data. Both functions are inserted into the stream and execute in-order but asynchronously.

In addition, the runtime library provides the functions *mem\_alloc* and *mem\_free* for allocating and deallocating global memory. There are also three system calls for writing, read and clearing a continuous region of memory, which are executed on the SRC. In contrast to the global memory, the local RAM must be managed manually and can be written for all SRCs in parallel by using the *config* system call. At the end of a frame, *display* sets the start address of the front buffer for HDMI output.

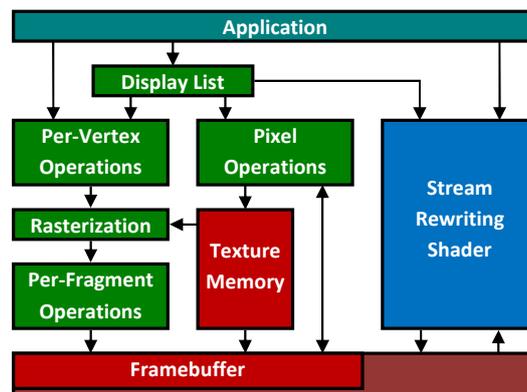


Figure 94. OpenGL pipeline with stream rewriting module.

### 5.6.5 OpenGL Integration

On top of the runtime library (Section 5.6.4), a proof-of-concept OpenGL front-end has been implemented without support of the fixed function pipeline. A special feature of OpenGL is the extension mechanism, which allows to introduce experimental functionality into the API before it is standardized. The standard OpenGL pipeline and its relation to stream rewriting are illustrated in Figure 94. Initially, rendering commands are processed by the per-vertex, rasterization, and per-fragment stages, which also contain programmable shaders in more recent OpenGL versions. This thesis propose an alternative type of programmable stage called the *stream rewriting shader*, which bypasses the existing per-vertex, rasterization, and per-fragment processing but stream rewriting commands are still compiled into display lists.

In order to minimize the number of modifications, the extension *EXT\_stream\_rewriting* is built on top of the existing OpenGL shader API (*glCreateProgram*, *glDeleteProgram*). A stream program is actually written in C, compiled using the existing GCC tool chain for the MicroBlaze architecture and the resulting ELF executable can be loaded via *glProgramBinary*. The stream program exports a list of global variables and functions, whose address can be retrieved by calling *glGetUniformLocation*. Variables are set using the class of *glUniform\** functions and for the invocation of a stream program, the following commands are added to emit literal tokens of various types (\*) and function calls:

```
void glPut*(T x, T ...);  
void glCall(GLuint addr);
```

Similar to the *glBegin/glEnd* immediate mode, stream commands are exclusively placed inside a block of *glBeginStream* and *glEndStream*. As a result, the same GL state can be used for multiple calls to reduce the validation overhead in the driver.

```
void glBeginStream();  
void glEndStream();
```

The minimalistic example on the next page shows the recursive shader *subdiv*, which tessellates the triangle (*a*; *b*; *c*) until the specified detail level has been reached. In this case, the coordinates are passed to the next stage (*draw\_tris*) for further processing.

```

#define get3(v) get(v[0]); get(v[1]); get(v[2]);
#define put3(v) put(v[0]); put(v[1]); put(v[2]);

void subdiv()
{
    float a[3], b[3], c[3], ab[3], bc[3], ca[3];
    get3(a); get3(b); get3(c); // read 3x3 floats

    int level;
    get(level); // read detail level

    if (level == 0) // end of recursion reached {
        put3(a); put3(b); put3(c); // draw current triangle
        call(draw_tris, 12);
        return;
    }

    int d = level-1; // proceed with subdivision

    for (int i = 0; i < 3; i++) {
        ab[i] = (a[i]+b[i])*0.5f; // midpoint of a and b
        bc[i] = (b[i]+c[i])*0.5f; // midpoint of b and c
        ca[i] = (c[i]+a[i])*0.5f; // midpoint of c and a
    }

    // create four smaller triangles
    put3(ca); put3(a); put3(ab); put(d); call(subdiv,13);
    put3(ab); put3(b); put3(bc); put(d); call(subdiv,13);
    put3(bc); put3(c); put3(ca); put(d); call(subdiv,13);
    put3(ab); put3(bc); put3(ca); put(d); call(subdiv,13);
}

```

The compiled ELF image is loaded into a program object *p*:

```

const char elf_image[] = { ... };
GLuint p = glCreateProgram();
glProgramBinary(p, GL_FORMAT_ELF, elf, sizeof(elf));
GLuint subdiv = glGetUniformLocation(p, "subdiv");

```

The following code selects the program *p* and utilizes the new functions to invoke the shader for the coordinates (*a*; *b*; *c*):

```

glUseProgram(p); // select shader program p
glBeginStream(); // begin initial stream
glPut3fv(a); // write coordinates a, b, c
glPut3fv(b);
glPut3fv(c);
glCall(subdiv); // append call token
glEndStream(); // submit initial stream

```

In the next section, the proposed extension *EXT\_stream\_rewriting* and the OpenGL prototype driver are utilized to implement more complex examples.

### 5.6.6 Results

In this section, experimental results and performance measurements of the FPGA prototype are presented. The design is implemented using the VC707 board and connected to a PC with a dual core 3.20GHz Pentium 4 CPU that runs Ubuntu (kernel 3.2).

Hardware configurations for up to 32 cores have been tested. In addition, the software simulator (SW) is evaluated using an i5-4670K processor with 3.4HGz and Windows 8.1. For the hardware version, the execution time of a shader is measured using the *ARB\_timer\_query* extensions with a resolution of 5ns. The images have been rendered at a resolution of 640x480 pixels. This section consists of three parts. As a reference for subsequent test cases, arithmetic and memory throughput of the SRP is evaluated first. The second part utilizes several basic rendering tests to demonstrate the ability of the SRP to execute various graphics pipelines in software. Finally, the third part contains three advanced examples, which are difficult to implement on current GPUs and therefore especially benefit from the more flexible architecture of the stream rewriting processor.

Table 12. Arithmetic Performance in MIPS

#	ADD	MULT	FADD	FMULT	FDIV
<b>Max./SRC</b>	400.00	400.00	200.00	400.00	27.59
<b>1</b>	375.98	376.39	196.90	376.38	23.53
<b>2</b>	751.11	752.75	393.79	752.73	47.06
<b>4</b>	1,498.80	1,505.29	787.53	1,505.05	94.12
<b>8</b>	2,983.74	3,008.52	1,574.73	3,008.66	188.22
<b>16</b>	5,911.50	6,009.33	3,146.95	6,010.53	376.41
<b>32</b>	11,598.97	11,957.08	6,277.81	11,960.83	752.60
<b>SW</b>	2,083.41	1,666.92	1,655.36	3,428.56	524.09

Table 13. Memory Throughput in MB/s

#	Read	Write	Increment	Clear
<b>1</b>	254.37	325.80	149.15	1,517.96
<b>2</b>	498.92	609.05	293.77	3,017.31
<b>4</b>	971.44	1,047.80	561.56	5,955.01
<b>8</b>	1,432.23	1,564.73	935.67	5,946.51
<b>16</b>	1,291.72	1,133.41	1,006.56	5,938.92
<b>32</b>	1,262.05	916.24	852.54	5,929.74
<b>SW</b>	8,870.77	7,735.24	9,226.41	not supported

#### 5.6.6.1 Performance Tests

The ALU performance is measured by distributing  $2^{32}$  operations of a specific type equally on all SRCs and the results are shown in Table 12. In addition, the theoretical maximum of each instruction type per SRC is listed in the first row. As a result, the largest configuration reaches  $\approx 90\%$  of the theoretical performance. Beside the division, the raw ALU performance of the hardware SRP exceeds the results of the software emulation, so that it most likely does not represent a bottleneck in further test cases.

The VC707 prototyping platform contains a DDR3 SODIMM 800MHz / 1600Mbps with 64-bit data width, which leads to a maximum throughput of 12.8 GB/s for the external memory interface. In order to match the peak performance, the L2 cache and the memory interconnect are running at 200 MHz with 512-bit data width. However, the actual efficiency of DRAM access heavily depends on the address pattern. In this test, the capability of the memory sub-system to handle concurrent requests from up to 32 SRCs is measured. In particular, 256x1MB are read, written, incremented and cleared by separate threads (Table 13). In comparison to existing GPUs and the software simulator, memory bandwidth is several magnitudes smaller, so that a performance difference in further tests is certainly caused by memory access.

### 5.6.6.2 Generic Tests

After measuring the basic performance of the SRP, several generic applications are run to evaluate distinct use-cases of the system.

The *Mandelbrot* example demonstrates the evaluation of a data parallel function using stream rewriting, where each pixel is computed individually. Similarly, *Rotozoom* displays a rotating texture, but instead of expensive computations, there is one read and one write operation per pixel. Further tests include the rendering of a texture-mapped cube, a torus with per-pixel lighting, and the bunny with 69,451 triangles from the *Stanford Computer Graphics Laboratory*.

Bezier3D contains a complex pipeline, which sub-divides and draws a bi-cubic Bezier patch, and a low-poly object is tessellated using *N-patches* [182]. Ray-casting reimplements a classic 2.5D ray casting engine in the shader and displays an old cellar environment, while the ray-tracing tests draws a reflecting sphere on a plane with a chessboard pattern to measure the efficiency of dynamic scheduling with non-uniform workload. For each of test, the average rendering times per frame are shown in Figure 96 and selected images are displayed in Figure 95.

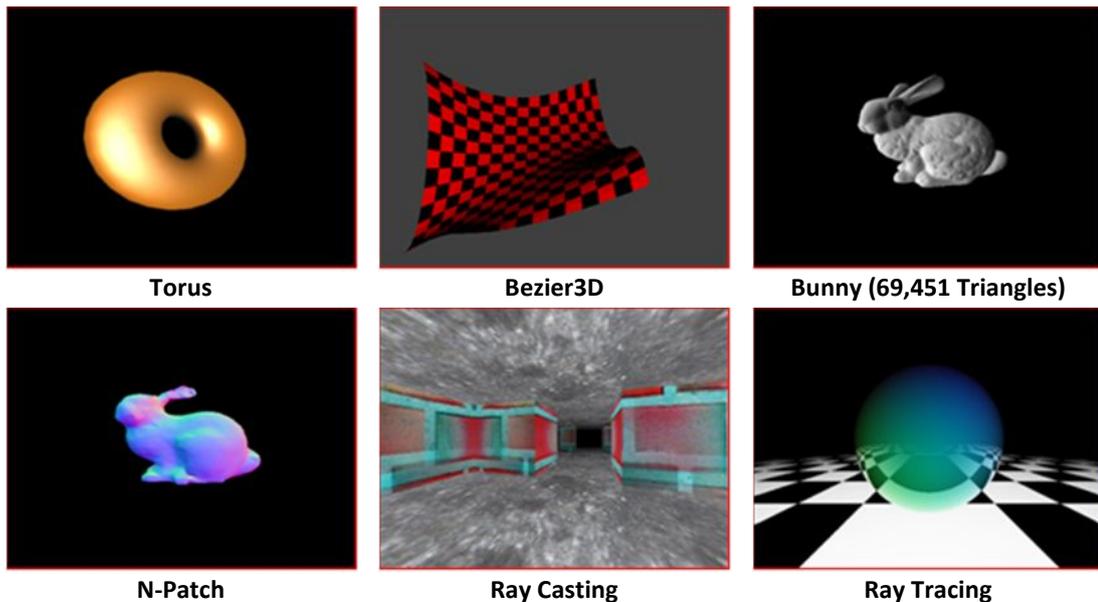


Figure 95. Screenshots (640x480) of the rendering tests

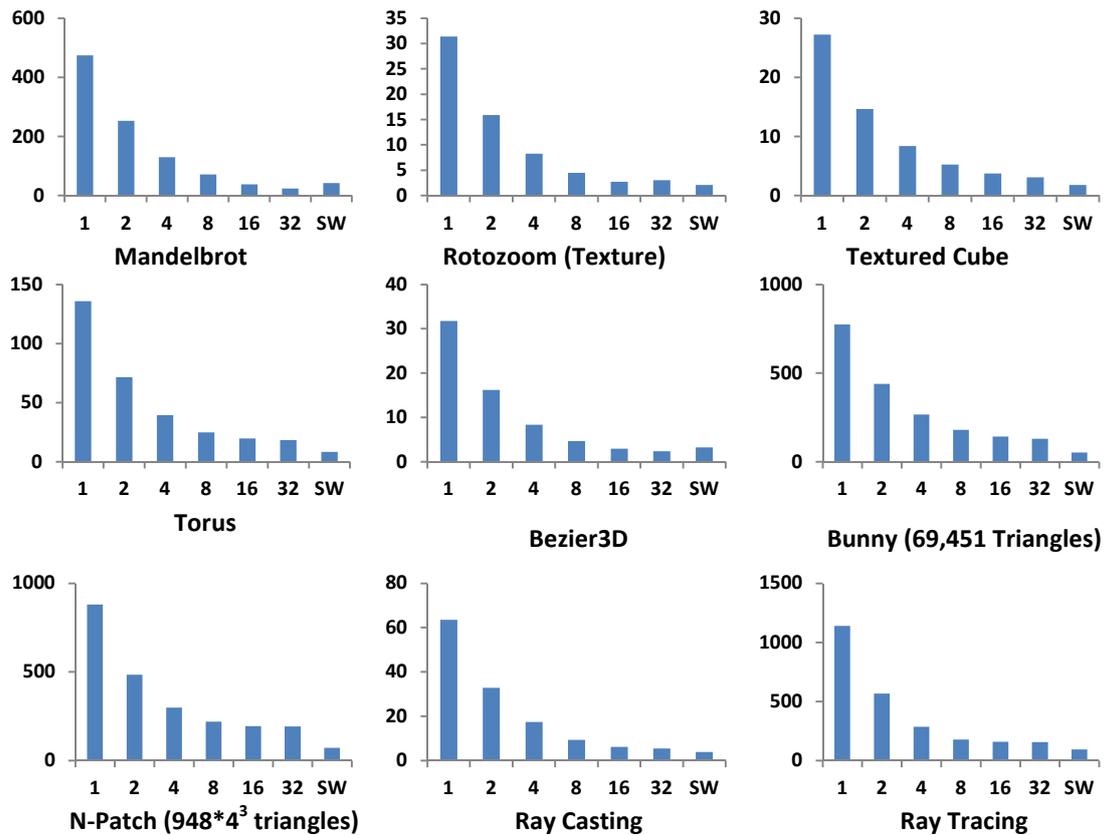


Figure 96. Average rendering time per frame in ms for different number of cores.

The results expose the general scalability of the system but also limitations of the current implementation. For instance, the software emulator is faster in all cases except *Bezier3D* and *Mandelbrot*, which suggests stream contention or cache trashing. Also, an increased number of small triangles cause stagnation at eight cores for the *Bunny* and *N-Patch* tests.

### 5.6.6.3 Advanced Tests

In this section, three examples, which benefit from the improved flexibility of the stream rewriting processor, are presented. The corresponding images are shown in Figure 97 and measurements are presented in Figure 99.

The *k-Buffer* has been proposed to solve the problem of order-independent transparency with a single geometry pass [150]. Regardless of the draw order, it collects the  $k$  nearest color and depth samples during the rendering of the scene. At the end of a frame, the stored samples are combined using alpha-blending. Hence, the rendering of a pixel requires the insertion of the current color and depth values into a sorted list of  $k$  elements. A significant technical challenge is to guarantee the correctness of this read-modify-write operation in case of multiple concurrent threads accessing the same location. However, recent GPUs often support only a fixed set of atomic operations for basic arithmetic, which do not cover this more complex and application-dependent case. Thus, the original *k-buffer* suffers from hazards and rendering artefact, which can be resolved by introducing per-pixel spin-locks [140] or applying software error correction [151].

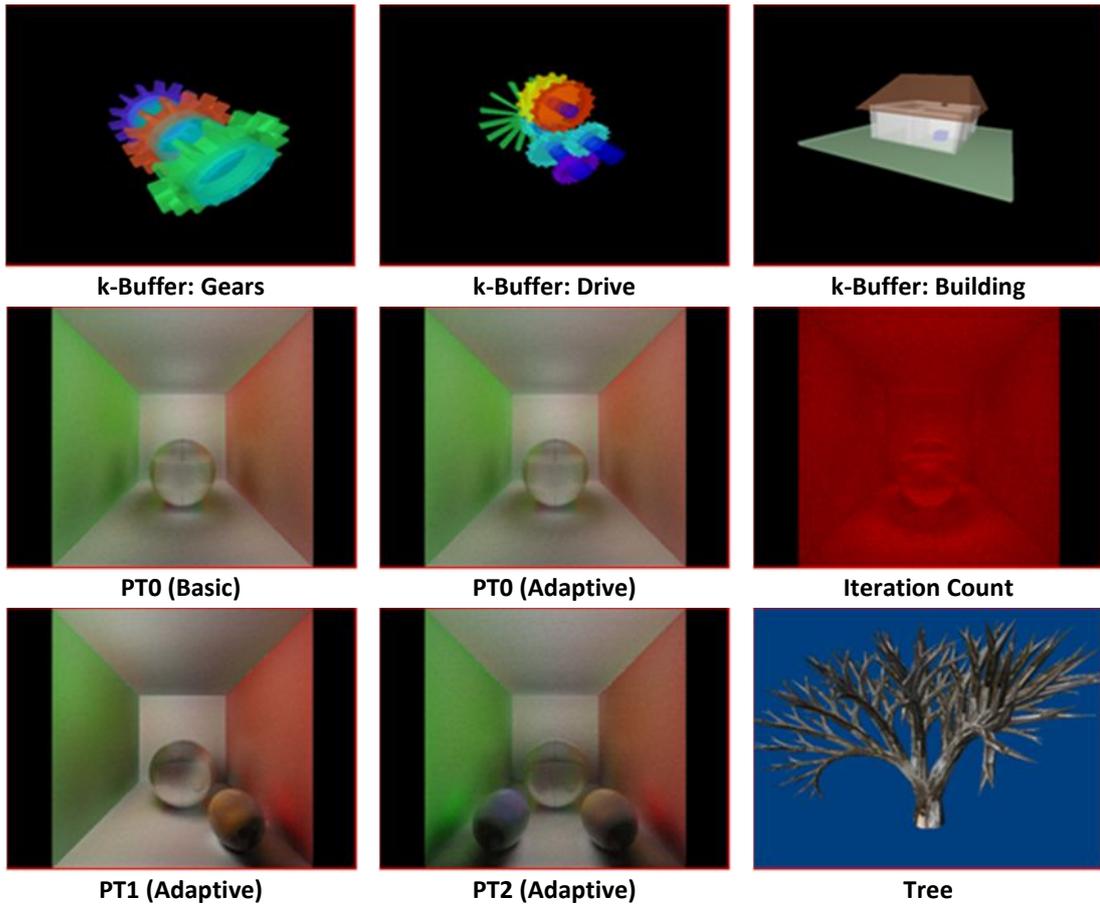


Figure 97. Screenshots (640x480) of the rendering tests.

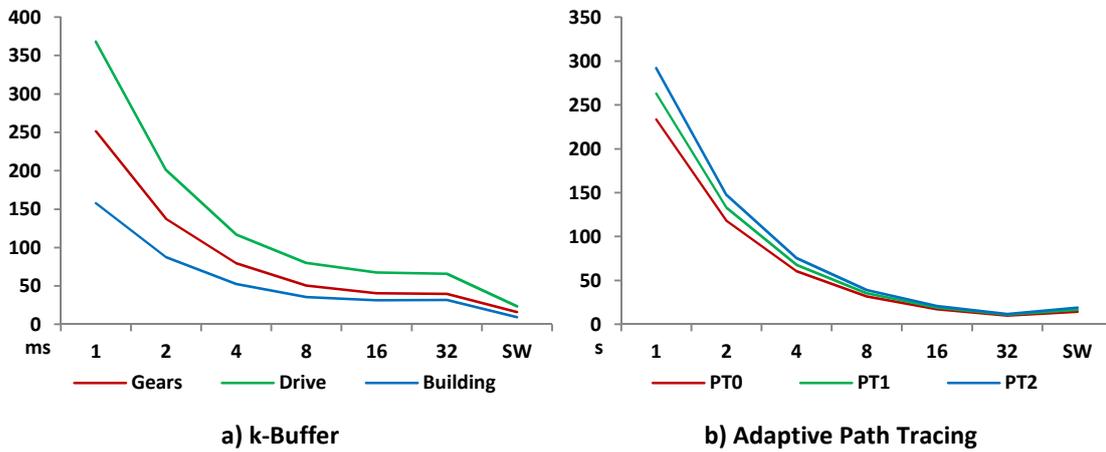


Figure 98. Performance of advanced rendering tests.

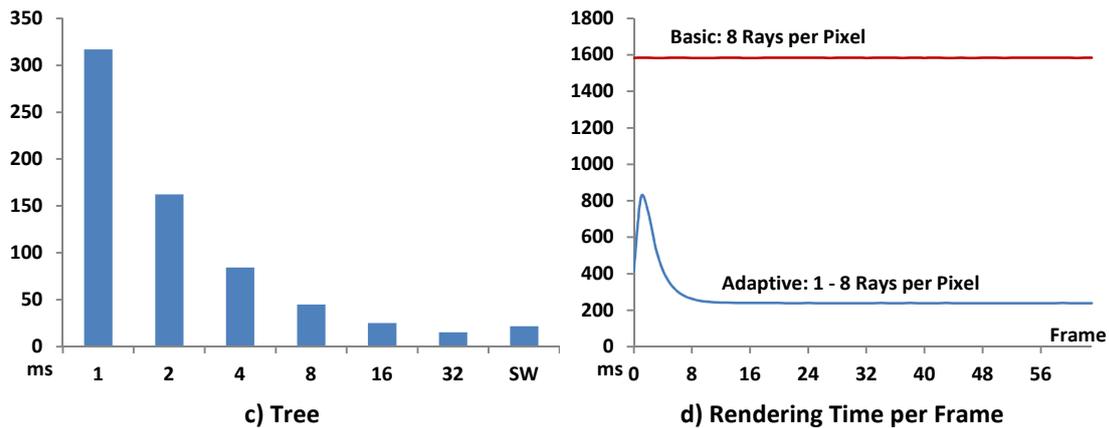


Figure 99. Performance of advanced rendering tests.

The k-Buffer implementation for stream rewriting follows the original proposal, which describes a possible hardware extension to assign pixels to a specific shader core based on their coordinates (Fig. 2 in [150]). The suggested programmable scheduler can be implemented on the SRP in software using the atomic calls described in Section 2.5. Since this SRP is optimized towards coarse grained threads, the dynamic binding is not performed at the level of pixels but instead assigns scanlines to a particular core.

The k-Buffer implementation of this tests stores  $k = 4$  elements per pixel and can therefore display up to four levels of transparency. It is evaluated using the gears, drive and building models (Figure 97). The average rendering times per frame are shown in Figure 98a) and indicate acceptable scalability for up to eight cores as well as a moderate speed up for 16 cores. Since each pixel stores  $2 \cdot k = 8$  32-bit values for color and depth information, memory bandwidth can become a bottleneck of this test and therefore, the software emulator outperforms the hardware implementation.

The second test employs *path tracing* to approximate the problem of global illumination. For this purpose, *path tracing* generates random rays, which are iteratively reflected from surfaces to collect both direct and indirect light contributions [183]. This technique effectively utilizes a Monte Carlo algorithm to approximate the integral of the rendering equation and is able to produce realistic images at the expense of computation time. However, since the calculations of pixels and rays are independent, it can benefit from many-core architectures. Unfortunately, stochastic algorithms generally do not map well to the data parallel SIMT architectures of current GPUs and in this case, the randomly generated rays lead to divergent control flow in the shader.

As a possible solution, the author of [184] suggests to remove inactive rays after each iteration and compact the stream to improve utilization. Since the SRP already reassembles rewritten fragments into a single continuous stream, the compaction is actually performed in hardware by the combiners of the stream rewriting network. As a result, the presented SRP architecture can handle non-data-parallel threads with different code paths at full speed. In addition, the conditional creation of additional rays corresponds to the dynamic instantiation of threads and fits well into the execution model of stream rewriting.

Three test cases (PT0, PT1, PT2) are presented in Figure 97 on page 125 and show (1, 2, 3) partly reflective spheres in a colored box. The basic example always computes 64 frames with 8 rays per pixel but the adaptive version skips the remaining rays of a frame as soon as a pixel converges. Hence, each pixel in the image is computed using a different number of iterations (Figure 97, middle right, colors normalized), so that the rendering time of the adaptive approach drops after a few frames (Figure 99d). Contrary, the basic approach with a fixed number of rays wastes computational time by processing pixels, which are not likely to change. Due to the large amount of floating-point arithmetic, the adaptive path tracing achieves good scalability (Figure 98b). However, for the 32 core version, a delay of 1s had to be inserted after each frame, which is not shown here, to stabilize the power supply of the FPGA board.

Similar to shape grammars [153], the recursive refinement of the token stream naturally corresponds to the creation of procedural geometry on the GPU and exceeds the capabilities of the Direct3D 11 tessellator stage. In order to demonstrate this functionality, a growing tree is generated in the shader based on an age parameter. For this purpose, the tree is constructed from cylindrical shapes [185], which are then rasterized as triangles. The final image of the tree is shown in Figure 97 and the performance results (Figure 99c) indicate scalability.

#### **5.6.7 Conclusion**

Based on the concept of stream rewriting, a novel graphics processor has been designed, which supports custom rendering pipelines, recursion, complex atomic operations and incoherent workloads. It combines the performance advantages of hardware-based scheduling with the flexibility of general purpose processors. Beside its limitations, the FPGA prototype provides worthwhile insights for future GPU architectures and runs a large variety of applications.

## 6 High-Level Synthesis

This section describes the usage of stream rewriting for high-level synthesis of hardware components. Current high-level synthesis tools based on C/C++ offer only limited support for recursion and functions pointers. Stream rewriting enables hardware support for the dynamic creation of threads, parallel recursive tasks, and data-dependent branching in hardware. Complex examples are used to show the effectiveness of the proposed method. This section is mostly based on the paper [83].

### 6.1 Introduction

The synthesis of hardware components from a high-level software specification makes it possible to increase the complexity of the system while shortening the development time. In addition, resource consumption and performance can be optimized by automated design space exploration. Further, when compared to a manual implementation, the automated process is less error prone and can be verified, so that the quality of the resulting design is greatly improved [186]. However, most tools only support a subset of the C/C++ language and focus on either data-flow or control-flow based applications [187]. While data-flow graphs can be mapped very efficiently into hardware, the synthesis of dynamic control-flow and especially recursive function calls is often not possible. Usually, pointers are also not supported in general, because the tools rely on static analysis to constrain the set of possible locations [188] [189]. If a program does not match the expectations of the synthesis software, the compilation either fails or produces inappropriate large structures. Hence, a program must be adapted according to the restrictions of the synthesis tools before it can be implemented in hardware [190]. Although high-level synthesis can be considered as a great success, the following open problems can be identified:

1. Often a fixed mapping between high-level functionality and hardware components is constructed, so that the design cannot react to varying workloads at runtime.
2. Arbitrary communication between two modules requires at least one logical channel implemented via a network or direct connections.
3. If recursion is supported, the local state is often stored on a stack which enforces sequential access and hinders concurrent execution of multiple branches.

In this context, the computational model of stream rewriting offers significant advantages for the high-level synthesis of RTL code from C sources. First, it has been already shown in Section 3.3 that a restricted form of C can be translated into a SRM. In this section, the SRM will be further used to generate RTL Verilog code. Finally, both steps are combined into a complete tool-chain for the high-level synthesis of recursive C functions (Figure 100).

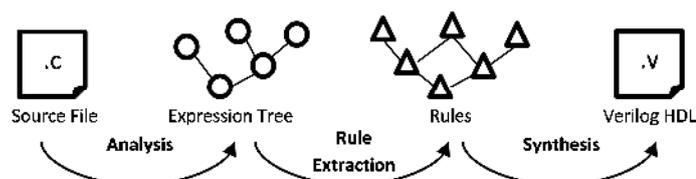


Figure 100. The synthesis tool-chain consists of several steps

First, the compiler parses the source file and converts the program into an expression tree similar to a functional language (Section 3.3). This tree is then rewritten and mapped into an acyclic graph of rules describing functional units of the hardware. In the last step, the hardware implementation of these rules is generated and written to a Verilog source file, which serves as an input for RTL synthesis. In contrast to the fundamental different representation, all C languages features except indirect access of local variables and arguments can be supported. Functions with and without return value can be marked as asynchronous to execute on a new thread. The synchronization occurs when the result is accessed the first time.

As a result, recursive functions, indirect calls via pointers and the dynamic creation of new threads are supported natively. However, unlike a software-based solution, which uses a stack for recursion, the synthesized hardware is able to execute multiple branches in parallel. Based on the distributed light-weight scheduling mechanism of stream rewriting, it becomes possible to spawn and synchronize multiple threads per clock cycle. As shown in Section 3.3, all branch operations are implemented as asynchronous function calls by the synthesis front end when it splits the source program, so that even normal control flow can take advantage of the hardware-based scheduling to hide the latency of long pipelines.

## 6.2 Related Work

In the context of hardware synthesis, there exist two different types of recursion:

- *Structural recursion* allows creating hierarchical designs and, similar to macros or inline functions, it is always expanded during synthesis. As a result, most tools including VHDL or Verilog provide support for structural recursion.
- *Generative recursion* is based on dynamically computed values and requires the expansion of a function at runtime. Stream rewriting provides a method for generating hardware from a recursive behavior specification.

For comparison, there have been several attempts of recursive hardware synthesis [191]: The stack-based approach presented in [192] offers a general solution to this problem. Two stacks for storing local variables and execution position are used to remember the previous state. The control flow is implemented as a hierarchical state machine controlling these stacks, which are implemented in fast on-chip memory. Although the computations within each function can be parallelized, it is not possible to execute several recursive branches in concurrently. Similarly, the recent work on synthesis of affine recursion [193] uses a single stack but does not permit multiple invocations.

A stack-less solution has been proposed in [194] to enable parallel processing of recursive calls. Similar to the SRM, a chain of processing nodes is connected in a pipeline. However, if the recursion level exceed the maximum supported depth, an expensive dynamic reconfiguration of the chip is required. Contrary, the SRM is fully functional with only one node and the maximum recursion depth is only limited by the amount of available memory.

Term rewriting systems have been already used for both verification [195] and synthesis [196] of hardware components. Also, the *Bluespec* language [197], which is based on *Haskell*, transforms the program into a set of rules for hardware implementation. Though, their concept is fundamental different from the stream rewriting approach. The rules in *Bluespec* are guards monitoring the state and signals of each module. A rule executes by performing an atomic modification of the module's state. Concurrency is employed by allowing different rules to fire simultaneously and partitioning the functionality into separate modules. Contrary, the stream rewriting rules work on a stream of tokens and allow substituting several non-overlapping parts of the stream in parallel.

A very similar approach is the *queue machine* [67], which also performs iterative replacement operations on a stream. It is optimized for pipelined operations, but does not support flow control nor recursion or indirect calls. In addition, it always operates on neighboring tokens and does not provide random access to arguments.

The concept of program evaluation by term rewriting has been presented for the software development language *Joy* [198], which is similar to *FORTH* [99]. The execution model of stream rewriting is related to the approach of *Joy*, but supports random access of arguments to avoid the costly reordering like the *queue machine*.

### 6.3 Implementation

In this section, the generated hardware architecture is described and resembles a more specialized version of the general purpose (Section 4) and graphics processors (Section 5). Though, it is based on the pre-order format of the stream grammar.

#### 6.3.1 Architecture

Figure 101 shows the layout of the component generated by the SRM tool-chain. In contrast to the abstract model, there are multiple processing elements named  $core_1$  to  $core_n$  and a shared memory block for data-exchange. However, one core would be also sufficient, because each of them implements the complete functionality necessary for substituting the *CALL* tokens. The result is passed to the next stage, so that  $n$  subsequent iterations can be overlapped and executed in parallel. The shared memory can be accessed from any core as part of term rewriting processes. In addition to the formal specification, a replacement function may also modify or read the shared memory. Though, the execution order of rules is based purely on data-dependencies and does not account for side-effects. Hence, memory accesses must be serialized similar to synchronous calls if necessary.

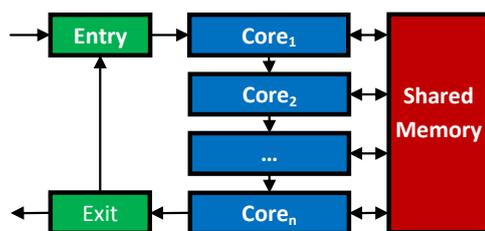


Figure 101. Synthesized hardware architecture

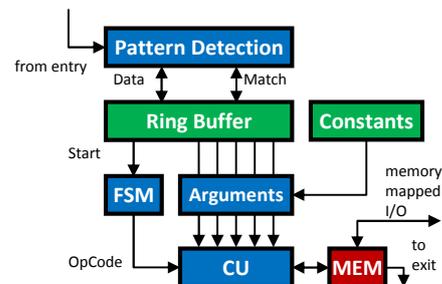


Figure 102. Structure of the rewriting core

The actual stream processing takes place inside the rewriting core which is illustrated in Figure 102. The token stream, coming from the entry point, is arriving at the top-left corner of this scheme. First, it goes through the pattern detector which marks all CALL tokens followed by a sufficient number of arguments. Each of these tokens can be executed immediately and is tagged with an extra bit indicating the positive match. The ring buffer is used as a temporary storage for the arguments, allowing them to be later accessed in any order. The buffer can hold a pending rule while another one is evaluated, so that argument fetch and execution can be overlapped. As a result, even the last argument can be read in the first cycle of the term rewriting process. Depending on the state of the match bit, the FSM either issues the corresponding sequence of tokens or initiates a pass-through operation. The pass-through operation is effectively a NOP, so that unmatched parts of the stream are preserved for further iterations. The computational unit (CU) is responsible for calculating types and values of the new tokens by arithmetic or logic operations. Arguments are received either from the ring buffer or from a special constant RAM. The CU itself consists of several functional units (FU), each one implementing a special function in hardware. During synthesis, the operations required by all rules determine the concrete set of FUs built into the computational unit. In addition to the formal specification, the last stage provides a general I/O interface, allowing rules to access external memory or periphery. Since the execution order of rules is based purely on data-dependencies, memory accesses must be explicitly serialized.

### 6.3.2 Rule Extraction

The functional program generated by the analysis phase Figure 100 does not yet represent hardware operations. First, the abstract operations like addition or multiplication are mapped into functional units (FU) from a library. In addition, multiple simple operations can be combined, but this merge may hinder the reusability of a FU. Since every core is able to execute all functions hardware resources can be shared between rewriting rules. Thus, the hardware costs of the CU (Figure 102) do not increase if an operation is used multiple times. Instead, it will be composed of all functional units selected in this phase.

Almost the complete execution is pipelined to support complex arithmetic like floating-point at high clock rates. Hence, there is no possibility to reuse temporary results. All inputs must be either arguments of the rule or can be fetched from the constant memory. As a consequence, rules containing multiple FUs in a chain must be split and are executed in multiple steps (Figure 103).

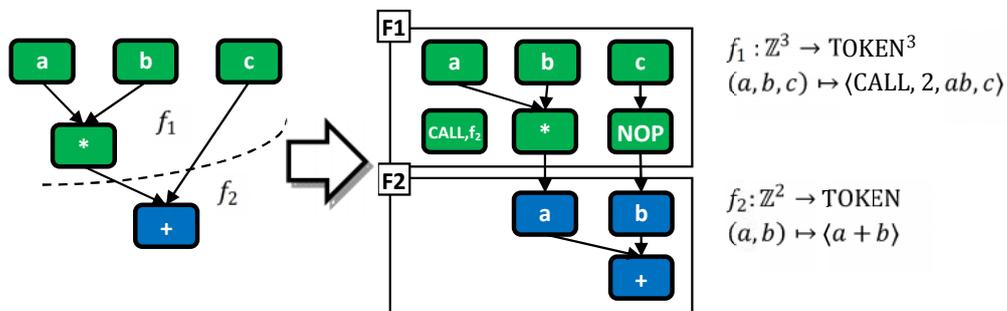


Figure 103. A complex expression tree is split into two simpler rewriting rules  $f_1$  and  $f_2$ .

However, the proposed architecture allows pipelining these calculations, because the next step is then evaluated on the next core. In the example illustrated in Figure 103, the argument  $c$  is used in the sum and is therefore copied using a NOP operation. The expression tree shown on the left side of Figure 103 implements a multiply-add operation. For this example, it is assumed that there exist only separate functional units for addition and multiplication. Hence, a break is inserted after the multiplication to split the tree into the two rewriting rules  $f_1$  and  $f_2$ . Both rules are linked by inserting a  $\langle CALL, 2 \rangle$  token from the first rule  $f_1$  to pass the temporary results to the second rule  $f_2$ .

### 6.3.3 Analysis

The definition of the rewriting system allows for a sufficient hardware implementation due to the following properties:

- **Prefix rule facilitates hardware decoding**  
All rules using the *CALL* token belong to a single class of prefix rules, so that a single parametrized pattern detector component can detect all of them. As a result, the complexity of this module does not depend on the number of rules in the program. Further, all rules are prefix rules, so that no backtracking is required either.
- **Pipelining**  
Each rule corresponds to an acyclic data flow graph containing a fixed number of inputs and outputs. Therefore, the substitution of tokens can be pipelined to increase the maximum clock rate of the synthesized circuit.
- **Light-weight dynamic scheduling**  
Deep pipelines also increase the latency of the module, but it can be partly compensated if the ring contains an appropriate number of matching function calls. As a result, complex functions containing both compute-intensive arithmetic and irregular control flow can be implemented without branch prediction.
- **Locality of stream modifications**  
A replacement operation is a local modification of the stream removing  $n$  tokens and inserting  $m$  results. Hence, the size of the ring buffer (Figure 102), which is required to hold the inputs of at most two executable *CALL* expression, is proportional to the maximum number of arguments. Due to this locality and the absence of an explicit execution order, the stream can be modified at multiple positions simultaneously without affecting the final result.

The design space can be explored synthesizing multiple functionally equivalent versions of the hardware with different resource and performance characteristics.

### 6.3.4 Scalability

All rules perform local rewrite operations by removing  $n$  tokens and inserting  $m$  results. Further, the abstract model of the SRM does not define an explicit execution order, so that multiple matching rules can be replaced simultaneously without affecting the final result. Hence, the computation can be accelerated by instantiating several identical *Rewriting Cores* to modify the stream of tokens at multiple positions.

According to Figure 101, the architecture of the *Stream Rewriting Machine* consists of  $n$  *Rewriting Cores*  $core_1$  to  $core_n$  and a shared memory block that may be replaced by a more sophisticated approach.

Since, the results of one core are passed to the next stage,  $n$  subsequent iterations can be overlapped and executed in parallel. Hence, the function *rewrite* is composed  $n$  times to create a pipeline of  $n$  stages:

$$pipe_n := rewrite^n \quad (\$ \$)$$

Thus, the number of iterations required to calculate the result is reduced by the factor  $n$ .

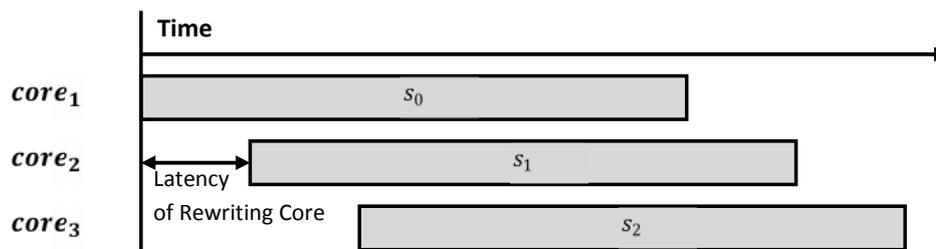


Figure 104. Pipelined rewriting of consecutive iterations.

## 6.4 Results

The proposed tool-chain has been implemented and employs a prototype compiler that generates RTL Verilog from restricted C source code.

### 6.4.1 Recursive Functions

On the next page, Table 14 and Figure 105 show the execution time in cycles for several recursive functions and multiple cores. All examples have been generated by the presented tool chain from and the resulting Verilog modules have been evaluated using a cycle accurate RTL simulator. The number of cores is modified using a generic parameter and does not require a new run of the high-level synthesis tools. The *sum* function is a simple recursive sum using integer addition with a latency of one cycle. It runs slowly because every instance creates only one additional thread, but due to pipelining it achieves a moderate speedup. The *rsum* function is almost 10 times faster, because it uses a balanced binary call-tree, which better utilizes the architecture of the ring. The Fibonacci (*fib*) examples show a similar behavior and also compute multiple branches in parallel. The Ackermann function (*ack*) also contains two recursive invocations, but has dependencies, which hinder a concurrent evaluation.

Table 14. Execution Time in Cycles

Test\Cores	1	2	4	8	16
sum(100)	32148	16589	9229	6479	6255
rsum(100)	3460	1989	1262	1223	1215
fib(9)	1530	959	709	734	735
fib(12)	5827	3226	1894	1335	1463
fib(16)	38698	20443	10502	5871	4930
ack(2,5)	6821	6253	5981	5845	5777
ack(3,2)	53789	37945	35729	34917	34649

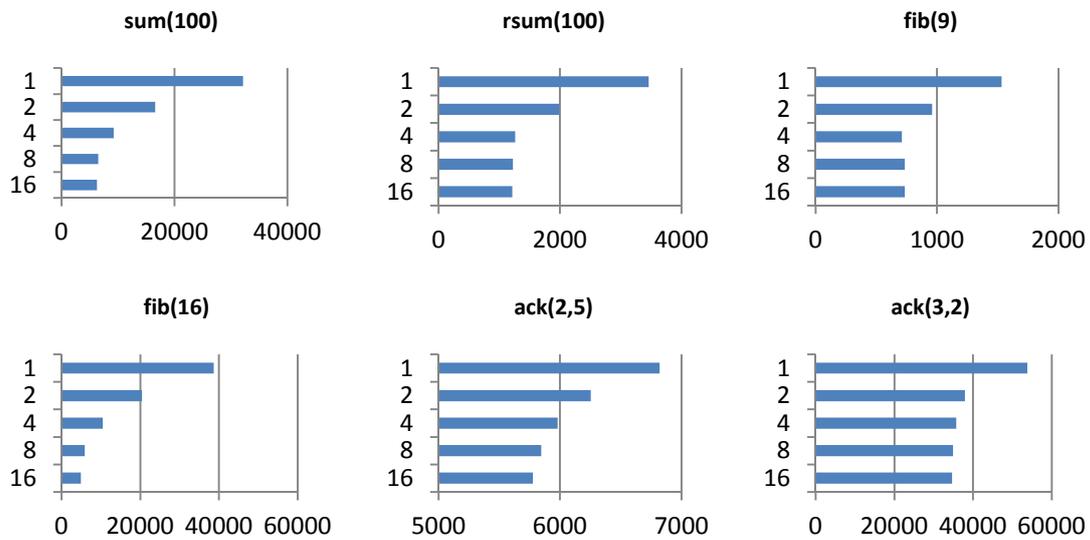
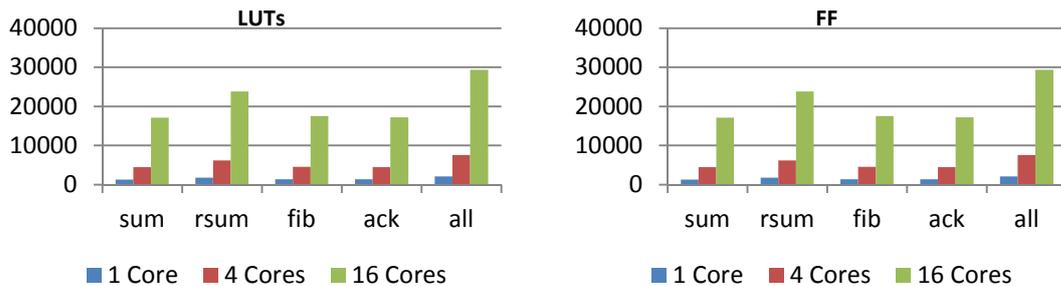


Figure 105. Execution Time in Cycles



Test	1 Core			4 Cores			16 Cores		
	LUT	FF	BRAM	LUT	FF	BRAM	LUT	FF	BRAM
sum	1311	1085	7	4443	3809	27	17109	14378	105
rsum	1727	1402	8	6170	5074	29	23829	19770	113
fib	1335	1089	7	4539	3825	27	17493	14442	105
ack	1318	1096	7	4471	3853	27	17221	14554	105
all	2060	1669	10	7580	6142	34	29344	24046	130

Figure 106. Resource usage of look-up tables (LUT), registers (FF) and BRAMs (LUT/FF/BRAM).

All functions have been synthesized for the XC6VLX240T-FF1156-1 FPGA from Xilinx by using the provided Verilog RTL compiler (XST). The source code of the Fibonacci function is shown below and the other test cases are implemented in a similar way:

```
// writes x to the exit point
// generates OUT token
void print(int x);

// Calculates the fibonacci number x
int fib(int x)
{
    if (x < 2)
        return x;
    return fib(x-1) + fib(x-2);
}

void main(int x)
{
    print(fib(x));
}
```

The resource usage of 6-input look-up tables (LUT), flip-flops (FF) and embedded RAMs (BRAM) consumed by a particular implementation is shown in Figure 106. The high-level synthesis time without RTL synthesis always remained below one second using an Intel i7-2720 CPU and 8GB RAM. In addition, a *Stream Rewriting Machine* implementing all four functions has been synthesized for comparison. Since every functional unit is shared globally across all rules, the combination requires only 30% more resources. It does not achieve a true linear speedup, because the function of multiple iterations *pipe* also consumes more cycles than the single-core variant (*rewrite*). However, according to the definition of the term rewriting system, a full parallel evaluation is possible, so that these results are unrelated to the computational model and only reflect a drawback of the current implementation.

#### 6.4.2 Ray-tracing

In order to provide a more complete example, a ray tracing application has been implemented. It shows the usage of function pointers, recursive functions and the parallel evaluation of branches. Hence, all elements of the computational model are demonstrated in this example. Ray tracing generates images by sending rays from the camera through every pixel into a three-dimensional scene. The color of the pixel is then determined by the object at the nearest intersection point. In this example, the objects are stored as a set of parameters and a pointer to an intersection function. It takes origin and direction of a ray and returns the color of the corresponding object at the intersection point.

Similar to virtual functions, the usage of the function pointer allows to handle the three types of objects (Plane, Sphere, Node) equally and without switch statements. Also the scene itself is stored as an object and represents a tree that is visited recursively. For instance, the basic data structures of this test case are declared as:

```

struct Node;
struct Intersection;

// Generic intersection function
typedef Intersection (*IntersectFunc)(Node *node, float3 start, float3 dir);

// Stores one object of the scene
struct Node
{
    float4 pos;           // position
    IntersectFunc intersect; // intersection function
    int left;            // left child node
    int right;           // right child node
    float3 color;        // color rgb
};

// Represents an intersection between a ray and the scene
struct Intersection
{
    float dist; // distance from the start
    float3 color; // color at this position
};

// intersection functions for all three types
Intersection plane_intersect (Node *node, float3 start, float3 dir);
Intersection sphere_intersect (Node *node, float3 start, float3 dir);
Intersection node_intersect (Node *node, float3 start, float3 dir);

```

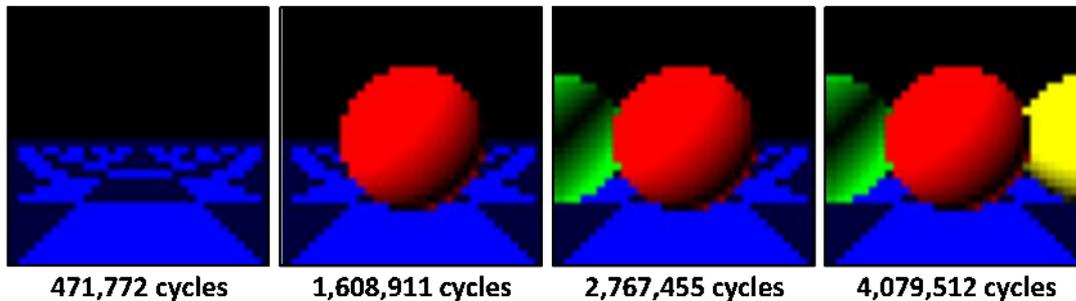


Figure 107. Results of the ray tracing simulation

The ray tracing simulation has been run to produce 32x32 pixel images of different scenes and assumes 12 cycles latency for floating-point addition, 8 cycles for multiplication and 28 cycles for division and square root. These values nearly correspond to the parameters of the floating-point cores from Xilinx when optimized for maximum frequency. The resulting images and the number of cycles are shown in Figure 107. As a result, it can be seen that the computational model is able to evaluate complex C programs containing recursion and function pointers. As a further improvement, it would be possible to place the state machine into a RAM, so that already synthesized hardware could be patched by updating the microcode.

## 7 Conclusion

This section contains an integral discussion of stream rewriting, a comparison of the presented approaches and ideas for further improvements.

### 7.1 Results

This thesis on stream rewriting has led to several new insights for dynamic scheduling in multi- and many-core environments. The achievements can be summarized as follows:

- **Stream rewriting as a new model of computation**

Although term or string rewriting systems are well known, the concept of stream rewriting has been first described in this thesis and the related papers. The most significant difference is the aspect of local rewriting rules, which enable an efficient many-core implementation. In contrast to other models of computation, stream rewriting can encapsulate data and pipeline parallelism, supports dynamic topologies, and schedules a large number of threads based on their dependencies.

- **Concept for concurrent stream processing**

The basic approach of replacing pattern in the stream already employs a large degree of concurrency by construction. Several parallel stream rewriting machines have been proposed (Section 2.2), implemented and, evaluated in the section 4,5, and 6.

- **Synchronization via the stream and without shared memory**

Stream rewriting supports the synchronization of concurrent threads without atomic operations in shared memory. Parallel data paths are joined locally using rewriting rules and the access to global resources is synchronized by dynamically binding interfering tasks to the same rewriting unit.

- **Connection of task graphs and stream rewriting**

Tasks graphs are commonly used to specify concurrent and dependent parts of an application. This thesis shows an approach for translating these graphs into stream rewriting programs (Section 3.1).

- **Compilation of a high-level language for the SRM**

In Section 3.1, a compiler for the hardware synthesis of recursive C functions is presented. It generates HDL code for a specialized SRM. In particular, the compiler converts the control flow between basic blocks into rewriting rules and passes local variables as arguments.

- **Extensive tests using several FPGA prototypes**

Beside Section 6, which is based on an RTL simulation, all stream rewriting architectures have been actually prototyped using an FPGA implementation. Since hardware never works in isolation, the experiments have been performed in a realistic test environment including external memory, a PCIe connection to the host processor, drivers, OS and application software, so that the measurements cover all aspects of the system. A large number of complex examples have been run to prove the usefulness of stream rewriting for a wide variety of problems and platforms.

## 7.2 Future Work

There are still some unsolved questions regarding the concept and the implementation of stream rewriting, which might be the topic of further research.

### 7.2.1 Stream Rewriting

The following possible improvements have been observed for the basic rewriting model:

- **Modeling of arbitrary graph topologies**

Although hierarchically expandable task graphs can be mapped into rewriting rules, their topology must conform to the sub-class of series-parallel graphs (Section 3.2.3). It might be possible to extend the basic definition of the SRM with additional rules that would allow for more complex elements like feed-back loops or even arbitrary topologies. Currently, the restriction to series-parallel graphs ensures the locality of operations since sub-graphs correspond to compact sub-streams. Hence, the most important challenge is the efficient processing of an arbitrary graph as a token stream.

- **Classification of data flow graphs**

The capabilities of stream rewriting in relation to data flow graphs are not completely clear. Though, the SRM supports the flexibility to conditionally emit tokens to arbitrary actors and to instantiate sub-graphs at runtime. On the other hand, even very simple homogeneous data flow graphs cannot be mapped into rewriting rules if their topology contains a loop or is not series-parallel. In addition to arbitrary topologies, it should be researched what type of data flow graphs might be translated into rewriting rules.

- **Reassembly of stream fragments causes implicit synchronization**

The synchronization of threads is a local operation since it involves only a limited range of tokens on the stream. Similarly, the rewriting process can be parallelized since matching rewriting rules never overlap. However, the reassembly of the rewritten stream fragments is the most expensive part because at this point, the system must guarantee that the resulting sub-streams are concatenated in their original order. In case of significantly varying task granularities, some processors might have finished the rewriting of their sub-stream earlier than others or might have generated a larger amount of tokens. Although reorder buffers usually compensate for different throughputs, a stall might occur at the synchronization point of the streams.

### 7.2.2 Embedded Systems

Stream rewriting is suitable for dynamic scheduling in embedded and mobile systems and might be especially valuable for the connection of heterogeneous components and different computational models at the system level. For this thesis, the following open problems have been identified:

- **Real-time scheduling**

The current research of stream rewriting has put a focus on handling a large number of dynamically created threads. Especially for graphics processing, throughput are more important than latency and the SRM is untimed by construction. The presented model does not impose any constraints concerning the execution times of threads because most of these decisions are made dynamically at runtime to facilitate concurrency. Hence, stream rewriting is not suitable for real-time systems with hard deadlines.

- **Static and dynamic scheduling**

Stream rewriting offers a novel mechanism for scheduling a large amount of dynamically created threads and also enables the recursive expansion of task graphs at runtime. Although this flexibility induces fixed costs, it might not be required entirely for all problems. By identifying static parts of an application, it might be possible to determine a tradeoff between static and dynamic scheduling.

- **Power estimation**

The topic of power consumption has been completely omitted in this work but might be important for the usage of stream rewriting in embedded or mobile systems.

### 7.2.3 Graphics Processing

The initial concept of stream rewriting has been developed to improve the flexibility of graphics processing by introducing a dynamic rendering pipeline with programmable communication. However, for competitive implementation, the following issues should be resolved:

- **Efficient GPU Implementation**

The performance of the FPGA prototype is limited by its size and several technological factors. As a result, the presented implementation is scalable to some extent but still several magnitudes slower than modern graphics processors. By porting stream rewriting to CUDA or OpenCL, the improved flexibility of the SRM could be possible combined with the massive computational power of the GPUs. However, the main challenge is the adaption of the rewriting algorithm to data parallel SIMT architectures since the execution of different threads might not be feasible with current hardware.

- **OpenGL and Direct3D drivers**

This thesis describes an incomplete OpenGL implementation that offers the stream rewriting functionality via an extension. By adding the remaining standard functions, it would become feasible to evaluate the performance of existing OpenGL applications. Similarly, a Direct3D driver for the SRM graphics processor would enable a much larger amount of test cases.

- **Novel and minimalistic graphics API**

Due to legacy interfaces and inappropriate abstractions, the communication between CPU and graphics processor involves a significant overhead. Especially in Direct3D, each individual draw call causes an expensive transition between user and kernel mode, so that an application can be quickly become CPU bound. Also for OpenGL, the classic rendering pipeline does no longer represent the structure and the capabilities of the underlying hardware. As a consequence, the driver has to perform dynamic code generation and validation at runtime to translate the current OpenGL state into hardware instructions.

The runtime API (Section 5) is a statically linked library, which provides a minimal interface to the stream rewriting hardware and consists of a few functions for encoding the initial stream. Likewise, shaders are written in plain C/C++, compiled via the standard GCC tool chain and submitted as pre-compiled binaries. The functionality of fixed hardware components like texture mapping units or the rasterizer is provided via software libraries and included into the shader binary. Hence, an application has full control over the device but can still apply domain-specific optimizations.

## 8 References

- [1] David Geer, "Industry Trends: Chip Makers Turn to Multicore Processors," *Computer*, vol. 38, no. 5, pp. 11-13, May 2005.
- [2] G. Blake, R.G. Dreslinski, and T Mudge, "A survey of multicore processors," *Signal Processing Magazine, IEEE*, vol. 26, no. 6, pp. 26-37, November 2009.
- [3] Per Hammarlund et al., "Haswell: The Fourth-Generation Intel Core Processor," *IEEE Micro*, vol. 34, no. 2, pp. 6-20, March 2014.
- [4] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU microarchitecture through microbenchmarking," in *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, White Plains, NY , march 2010, pp. 235-246.
- [5] K. Hirata and J. Goodacre, "ARM MPCore; The streamlined and scalable ARM11 processor core," in *Design Automation Conference, 2007. ASP-DAC '07. Asia and South Pacific*, Yokohama, 2007, pp. 747-748.
- [6] Shekha Borkar and Andrew A. Chien, "The future of microprocessors," *Communications of the ACM*, vol. 54, no. 5, pp. 67-77, May 2011.
- [7] P. Choudhury, P.P. Chakrabarti, and R. Kumar, "Online Scheduling of Dynamic Task Graphs with Communication and Contention for Multiprocessors," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 23, no. 1, pp. 126-133, jan. 2012.
- [8] NVIDIA, Nvidia's next generation cuda compute architecture: Kepler gk110. [Online]. <http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf>
- [9] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *Micro, IEEE*, vol. 28, no. 2, pp. 39-55, March 2008.
- [10] T.G. Mattson et al., "The 48-core SCC Processor: the Programmer's View," in *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, New Orleans, LA, 2010, pp. 1-11.
- [11] Anant Agarwal, "The tile processor: A 64-core multicore for embedded processing," in *Proceedings of HPEC Workshop*, 2007.

- [12] J. Henkel et al., "Invasive manycore architectures," in *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, Sydney, NSW, 2012, pp. 193-200.
- [13] L. Gauthier, Sungjoo Yoo, and A.A. Jerraya, "Automatic generation and targeting of application-specific operating systems and embedded systems software," vol. 20, no. 11, pp. 1293-1301, Nov 2001.
- [14] John Nickolls and William J Dally, "The GPU computing era," *Micro, IEEE*, vol. 30, no. 2, pp. 56-69, March 2010.
- [15] Victor W. Lee et al., "Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 451-460, June 2010.
- [16] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal of Research and Development*, vol. 11, no. 1, pp. 25-33, January 1967.
- [17] David Blythe, "The Direct3D 10 system," *ACM Transactions on Graphics (TOG)*, vol. 3, pp. 724-734, July 2006.
- [18] Chas Boyd, "The DirectX 11 compute shader," *ACM SIGGRAPH 2008 classes*, 2008.
- [19] Rob Farber, *CUDA Application Design and Development.*: Morgan Kaufmann, 2011.
- [20] Timo Aila, Samuli Laine, and Tero Karras, "Understanding the efficiency of ray traversal on GPUs--Kepler and Fermi addendum," *Proceedings of ACM High Performance Graphics 2012, Posters*, pp. 9-16, 2012.
- [21] H. Tomiyama, T. Hieda, N. Nishiyama, N. Etani, and I. Taniguchi, "SMYLE OpenCL: A programming framework for embedded many-core SoCs," in *Design Automation Conference (ASP-DAC), 2013 18th Asia and South Pacific*, Yokohama, Japan, 2013, pp. 565-567.
- [22] Tianyun Ni, "Direct Compute - Bring GPU Computing to the Mainstream," in *GPU Technology Conference*, 2009.
- [23] Kate Gregory and Ade Miller, *C++ AMP: Accelerated Massive Parallelism with Microsoft Visual C++.*: Microsoft Press, 2012.
- [24] Leonardo Dagum and Ramesh Menon, "OpenMP: an industry standard API for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46-55, January 1998.

- [25] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann, "OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization," *SIGPLAN Not.*, vol. 44, no. 4, pp. 101-110, April 2009.
- [26] M.E. Wolf and M.S. Lam, "A Loop Transformation Theory and an Algorithm to Maximize Parallelism," *IEEE Transactions on Parallel and Distributed Systems*, pp. 452-471, October 1991.
- [27] Hritam Dutta, Frank Hannig, Holger Ruckdeschel, and Jürgen Teich, "Efficient control generation for mapping nested loop programs onto processor arrays," *Journal of Systems Architecture*, vol. 53, no. 5-6, pp. 300-309, May 2007.
- [28] Joachim Keinert et al., "SystemCoDesigner- an Automatic ESL Synthesis Approach by Design Space Exploration and Behavioral Synthesis for Streaming Applications," *ACM Transactions on Design Automation of Electronic Systems*, vol. 14, no. 1, pp. 1:1--1:23, January 2009.
- [29] E.A. Lee and T.M. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773-801, May 1995.
- [30] Jörn W. Janneck et al., "Synthesizing hardware from dataflow programs: An MPEG-4 simple profile decoder case study," in *Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on*, 2008, pp. 287-292.
- [31] Gustav Cedersjö and Jörn W. Janneck, "Software Code Generation for Dynamic Dataflow Programs," in *Proceedings of the 17th International Workshop on Software and Compilers for Embedded Systems*, vol. SCOPES '14, Sankt Goar, Germany, 2014, pp. 31-39.
- [32] Joachim Falk, Joachim Keinert, Christian Haubelt, Jürgen Teich, and Shuvra S. Bhattacharyya, "A Generalized Static Data Flow Clustering Algorithm for Mpsoc Scheduling of Multimedia Applications," in *Proceedings of the 8th ACM International Conference on Embedded Software*, Atlanta, GA, USA, 2008, pp. 189-198.
- [33] E.A. Lee and D.G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235-1245, September 1987.
- [34] Edward Ashford Lee and David G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing," *IEEE Transactions on Computers*, vol. C-36, no. 1, pp. 24-35, Jan. 1987.

- [35] M. Damavandpeyma, S. Stuijk, T. Basten, M. Geilen, and H. Corporaal, "Modeling static-order schedules in synchronous dataflow graphs," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, Dresden, march 2012, pp. 775-780.
- [36] Joachim Falk et al., "Analysis of SystemC actor networks for efficient synthesis," *ACM Transaction on Embedded Computing*, vol. 10, no. 2, pp. 18:1-18:34, jan 2011.
- [37] Peng Yang et al., "Managing dynamic concurrent tasks in embedded real-time multimedia systems," in *15th International Symposium on System Synthesis, 2002.* , Kyoto, Japan, 2002, pp. 112-119.
- [38] S. Stuijk, M. Geilen, B. Theelen, and T. Basten, "Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications," in *Embedded Computer Systems (SAMOS), 2011 International Conference on*, Samos, july 2011, pp. 404-411.
- [39] Qinghua Li, Youlin Ruan, ShidaYang, and Tingyao Jiang, "An optimal scheduling algorithm for fork-join task graphs," in *Parallel and Distributed Computing, Applications and Technologies, 2003. PDCAT'2003. Proceedings of the Fourth International Conference on*, aug. 2003, pp. 587-589.
- [40] Weijia Che and Karam S. Chatha, "Unrolling and retiming of stream applications onto embedded multicore processors," in *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, San Francisco, CA , 2012, pp. 1268-1273.
- [41] Michael I. Gordon, William Thies, and Saman Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, San Jose, California, USA, 2006, pp. 151-162.
- [42] Manjunath Kudlur and Scott Mahlke, "Orchestrating the execution of stream programs on multicore platforms," *ACM SIGPLAN Notices*, vol. 43, no. 6, pp. 114-124, jun 2008. [Online]. <http://doi.acm.org/10.1145/1379022.1375596>
- [43] A. Hagiescu, Weng-Fai Wong, D.F. Bacon, and R. Rabbah, "A computing origami: Folding streams in FPGAs," in *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, San Francisco, CA , july 2009, pp. 282-287.
- [44] Amir Hormati, Manjunath Kudlur, Scott Mahlke, David Bacon, and Rodric Rabbah, "Optimus: efficient realization of streaming applications on FPGAs," in *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, New York, NY, USA, 2008, pp. 41-50.

- [45] Amir H. Hormati, Mehrzad Samadi, Mark Woh, Trevor Mudge, and Scott Mahlke, "Sponge: portable stream programming on graphics engines," in *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, Newport Beach, California, USA, 2011, pp. 381-392.
- [46] J. Kaida et al., "Task mapping techniques for embedded many-core SoCs," in *SoC Design Conference (ISOCC), 2012 International*, Jeju Island, 2012, pp. 204-207.
- [47] I. Ahmad and Yu-Kwong Kwok, "On parallelizing the multiprocessor scheduling problem," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 10, no. 4, pp. 414-431, apr 1999.
- [48] M. Abid, K. Jerbi, M. Raulet, O. Deforges, and M. Abid, "System level synthesis of dataflow programs: HEVC decoder case study," in *Electronic System Level Synthesis Conference (ESLsyn), 2013*, 2013, pp. 1-6.
- [49] H. El-Rewini, H.H. Ali, and T. Lewis, "Task scheduling in multiprocessing systems," vol. 28, no. 12, pp. 27-37, Dec 1995.
- [50] Yu-Kwong Kwok and Ishfaq Ahmad, "Benchmarking and Comparison of the Task Graph Scheduling Algorithms," *Journal of Parallel and Distributed Computing*, vol. 59, no. 3, pp. 381-422, December 1999.
- [51] Ying Yi, Wei Han, Xin Zhao, A.T. Erdogan, and T. Arslan, "An ILP formulation for task mapping and scheduling on multi-core architectures," in *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, Nice, april 2009, pp. 33-38.
- [52] H. Topcuoglu, S. Hariri, and Min-You Wu, "Task scheduling algorithms for heterogeneous processors," in *Heterogeneous Computing Workshop, 1999. (HCW '99) Proceedings. Eighth*, San Juan , 1999, pp. 3-14.
- [53] Hoeseok Yang and Soonhoi Ha, "Pipelined data parallel task mapping/scheduling technique for MPSoC," in *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, Nice, 2009, pp. 69-74.
- [54] Chen-Ling Chou and R. Marculescu, "User-Aware Dynamic Task Allocation in Networks-on-Chip," in *Design, Automation and Test in Europe, 2008. DATE '08*, Munich, 2008, pp. 1232-1237.

- [55] Ping Xiang, Yi Yang, Mike Mantor, Norm Rubin, and Huiyang Zhou, "Many-thread aware instruction-level parallelism: architecting shader cores for GPU computing," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, Minneapolis, Minnesota, USA, 2012, pp. 449-450.
- [56] D. Baudisch, J. Brandt, and K. Schneider, "Out-Of-Order Execution of Synchronous Data-Flow Networks," in *Embedded Computer Systems (SAMOS), 2012 International Conference on*, 2012, pp. 168-175.
- [57] Jürgen Teich et al., "Invasive computing: An overview," in *Multiprocessor System-on-Chip*.: Springer, 2011, pp. 241-268.
- [58] J. Becker, S. Friederich, J. Heisswolf, R. Koenig, and D. May, "Hardware prototyping of novel invasive multicore architectures," in *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, 2012, pp. 201-206.
- [59] K. Agrawal, C.E. Leiserson, and J. Sukha, "Executing task graphs using work-stealing," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, 2010, pp. 1-12.
- [60] Yizhuo Wang, Weixing Ji, Feng Shi, and Qi Zuo, "A work-stealing scheduling framework supporting fault tolerance," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013, Grenoble, France*, 2013, pp. 695-700.
- [61] Robert D. Blumofe and Charles E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM (JACM)*, vol. 46, no. 5, pp. 720-748, 1999.
- [62] Robert D. Blumofe et al., "Cilk: an efficient multithreaded runtime system," in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, California, USA, 1995, pp. 207-216.
- [63] M.D. McCool, "Scalable Programming Models for Massively Multicore Processors," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 816-831, May 2008.
- [64] B.A. Nayfeh and K. Olukotun, "A single-chip multiprocessor," *Computer*, vol. 30, no. 9, pp. 79-85, September 1997.
- [65] R. Hartenstein, "A decade of reconfigurable computing: a visionary retrospective," in *Design, Automation and Test in Europe, 2001. Conference and Exhibition 2001. Proceedings*, Munich, 2001, pp. 642-649.
- [66] S.C. Goldstein et al., "PipeRench: a reconfigurable architecture and compiler," *Computer*, vol. 33, no. 4, pp. 70-77, April 2000.

- [67] Herman Schmit, Benjamin Levine, and Benjamin Ylvisaker, "Queue Machines: Hardware Compilation in Hardware," in *Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2002, pp. 152-160.
- [68] M. Lam, "Software pipelining: an effective scheduling technique for VLIW machines," *ACM SIGPLAN Notices*, vol. 23, no. 7, pp. 318-328, July 1988.
- [69] S. Pillai and M.F. Jacome, "Compiler-directed ILP extraction for clustered VLIW/EPIC machines: predication, speculation and modulo scheduling," in *Design, Automation and Test in Europe Conference and Exhibition, 2003*, Munich, Germany, 2003, pp. 422-427.
- [70] V.A. Zivkovic, R. J W T Tangelder, and H.G. Kerkhoff, "Design and test space exploration of transport-triggered architectures," in *Design, Automation and Test in Europe Conference and Exhibition 2000. Proceedings*, Paris, 2000, pp. 146-151.
- [71] Alex K. Jones, Raymond Hoare, Dara Kusic, Joshua Fazekas, and John Foster, "An FPGA-based VLIW processor with custom hardware execution," in *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, 2005, pp. 107-117.
- [72] John R Ellis, "Bulldog: A compiler for VLIW architectures," New Haven, CT, USA, Tech. rep. 1985.
- [73] Bingfeng Mei, S. Vernalde, D. Verkest, and R. Lauwereins, "Design methodology for a tightly coupled VLIW/reconfigurable matrix architecture: a case study," in *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, vol. 2, 2004, pp. 1224-1229.
- [74] S. Naffziger et al., "The implementation of a 2-core, multi-threaded itanium family processor," *IEEE Journal of Solid-State Circuits*, vol. 41, no. 1, pp. 197-209, Jan 2006.
- [75] Harsh Sharangpani and Ken Arora, "Itanium processor microarchitecture," *Micro, IEEE*, vol. 20, no. 5, pp. 24-43, September 2000.
- [76] S.K. Raman, V. Pentkovski, and J. Keshava, "Implementing streaming SIMD extensions on the Pentium III processor," *Micro, IEEE*, vol. 20, no. 4, pp. 47-57, July 2000.
- [77] Nadeem Firasta, Mark Buxton, Paula Jinbo, Kaveh Nasri, and Shihjong Kuo, "Intel avx: New frontiers in performance improvements and energy efficiency," 2008.

- [78] P. Bonnot et al., "Definition and SIMD Implementation of a Multi-Processing Architecture Approach on FPGA," in *Design, Automation and Test in Europe, 2008. DATE '08, 2008*, pp. 610-615.
- [79] N. Togawa, K. Tachikake, Y. Miyaoka, M. Yanagisawa, and T. Ohtsuki, "Instruction set and functional unit synthesis for SIMD processor cores," in *Design Automation Conference, 2004. Proceedings of the ASP-DAC 2004. Asia and South Pacific*, Yohohama, Japan, 2004, pp. 743-750.
- [80] J. Davila et al., "Design and implementation of a rendering algorithm in a SIMD reconfigurable architecture (MorphoSys)," in *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, vol. 2, Munich, 2006, pp. 52-57.
- [81] C. Kozyrakis and D. Patterson, "Overcoming the limitations of conventional vector processors," in *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, 2003, pp. 399-409.
- [82] David W Wall, "Limits of instruction-level parallelism," *SIGOPS - Operating Systems Review*, vol. 25, no. Special Issue, pp. 176-188, April 1991.
- [83] Lars Middendorf, Christophe Bobda, and Christian Haubelt, "Hardware synthesis of recursive functions through partial stream rewriting," in *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, San Francisco, CA , june 2012, pp. 1203-1211.
- [84] Lars Middendorf, Christian Zebelein, and Christian Haubelt, "Dynamic Task Mapping onto Multi-Core Architectures through Stream Rewriting," in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), 2013 International Conference on*, Agios Konstantinos, 2013, pp. 196-204.
- [85] L. Middendorf and C. Haubelt, "A novel graphics processor architecture based on partial stream rewriting," in *Design and Architectures for Signal and Image Processing (DASIP), 2013 Conference on*, Cagliari, 2013, pp. 38-45.
- [86] Lars Middendorf and Christian Haubelt, "A Programmable Graphics Processor based on Partial Stream Rewriting," *Computer Graphics Forum*, vol. 32, no. 7, pp. 325-334, 2013.
- [87] Lars Middendorf and Christian Haubelt, "System Level Synthesis of Many-Core Architectures using Parallel Stream Rewriting," in *Electronic System Level Synthesis Conference (ESLsyn), Proceedings of the 2014*, San Francisco, CA , 2014, pp. 1-6.

- [88] Lars Middendorf and Christian Haubelt, "Scheduling of Recursive and Dynamic Data-Flow Graphs using Stream Rewriting," in *Proceedings of Special Edition on Data-flow Programming Models and Machines (MPP'14)*, Paris, France, 2014.
- [89] Christian Haubelt, Florian Ludwig, Lars Middendorf, and Christian Zebelein, "Using stream rewriting for mapping and scheduling data flow graphs onto many-core architectures," in *Signals, Systems and Computers, 2013 Asilomar Conference on*, 2013, pp. 1431-1435.
- [90] A. W. Appel and T. Jim, "Continuation-passing, closure-passing style," in *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, Austin, Texas, USA, 1989, pp. 293-302.
- [91] Haik Lorenz and Jürgen Döllner, "Dynamic mesh refinement on GPU using geometry shaders," in *16-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG) - Full Papers*, 2008, pp. 97-104.
- [92] Christopher Dyken, Martin Reimers, and Johan Seland, "Semi-Uniform Adaptive Patch Tessellation," *Computer Graphics Forum*, vol. 28, no. 8, pp. 2255-2263, 2009.
- [93] Craig M. Wittenbrink, "R-buffer: A Pointerless A-buffer Hardware Architecture," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, Los Angeles, California, USA, 2001, pp. 73-80.
- [94] Matthew Eldridge, Homan Igehy, and Pat Hanrahan, "Pomegranate: a fully scalable graphics architecture," in *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, 2000, pp. 443-454.
- [95] K.C. Yeager, "The Mips R10000 superscalar microprocessor," *Micro, IEEE*, vol. 16, no. 2, pp. 28-41, April 1996.
- [96] Craig M Wittenbrink, Emmett Kilgariff, and Arjun Prabhu, "Fermi GF100 GPU architecture," *Micro, IEEE*, vol. 31, no. 2, pp. 50-59, March 2011.
- [97] Robin Milner, *The definition of standard ML: revised.*: MIT press, 1997.
- [98] Simon L Peyton Jones, *Haskell 98 language and libraries: the revised report.*: Cambridge University Press, 2003.
- [99] Paul Frenger, "The JOY of forth," *SIGPLAN Not.*, vol. 38, no. 8, pp. 15-17, aug 2003.
- [100] José Meseguer, "Conditional rewriting logic as a unified model of concurrency," *Theoretical Computer Science*, vol. 96, no. 1, pp. 73-155, April 1992.

- [101] Jochem H. Rutgers, Marco J. G. Bekooij, and Gerard J. M. Smit, "Programming a Multicore Architecture Without Coherency and Atomic Operations," in *Proceedings of Programming Models and Applications on Multicores and Manycores*, Orlando, FL, USA, 2014, pp. 29:29-29:38.
- [102] Goguen Joseph, Kirchner Claude, and Meseguer Josè , "Concurrent Term Rewriting As a Model of Computation," in *Proc. Of a Workshop on Graph Reduction*, Santa Fe, New Mexico, USA, 1987, pp. 53-93.
- [103] Matthew Naylor and Colin Runciman, "The Reduceron Reconfigured and Re-evaluated," *Journal of Functional Programming*, vol. 22, no. 4-5, pp. 574-613, August 2012.
- [104] Arjan Boeijink, Philip K. F. Hölzenspies, and Jan Kuper, "Introducing the PilGRIM: A Processor for Executing Lazy Functional Languages," in *Implementation and Application of Functional Languages.*: Springer Berlin Heidelberg, 2011, pp. 54-71.
- [105] Tao Yang and A. Gerasoulis, "DSC: scheduling parallel tasks on an unbounded number of processors," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 5, no. 9, pp. 951-967, September 1994.
- [106] Tang Lei and S. Kumar, "A two-step genetic algorithm for mapping task graphs to a network on chip architecture," in *Digital System Design, 2003. Proceedings. Euromicro Symposium on*, Belek-Antalya, Turkey, sept. 2003, pp. 180-187.
- [107] Yu-Kwong Kwok and Ishfaq Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Comput. Surv.*, vol. 31, no. 4, pp. 406-471, dec 1999.
- [108] Yu-Kwong Kwok and I. Ahmad, "Dynamic critical-path scheduling: an effective technique for allocating task graphs to multiprocessors," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 7, no. 5, pp. 506-521, May 1996.
- [109] Jacobo Valdes, Robert E. Tarjan, and Eugene L. Lawler, "The recognition of Series Parallel digraphs," in *Proceedings of the eleventh annual ACM symposium on Theory of computing*, Atlanta, Georgia, USA, 1979, pp. 1-12.
- [110] Lucian Finta, Zhen Liu, Ioannis Milis, and Evripidis Bampis, "Scheduling UET-UCT Series-Parallel Graphs on Two Processors," *Theoretical Computer Science*, vol. 162, no. 2, pp. 323-340, Aug. 1996.

- [111] Yuan Xie and W. Wolf, "Allocation and scheduling of conditional task graph in hardware/software co-synthesis," in *Design, Automation and Test in Europe, 2001. Conference and Exhibition 2001. Proceedings*, Munich, 2001, pp. 620-625.
- [112] P. Eles, K. Kuchcinski, Z. Peng, A. Doboli, and P. Pop, "Scheduling of conditional process graphs for the synthesis of embedded systems," in *Design, Automation and Test in Europe, 1998., Proceedings*, Paris, 1998, pp. 132-139.
- [113] M. Cosnard and M. Loi, "Automatic task graph generation techniques," in *System Sciences, 1995. Vol. II. Proceedings of the Twenty-Eighth Hawaii International Conference*, vol. 2, Wailea, HI, jan 1995, pp. 113-122.
- [114] M. Cosnard, E. Jeannot, and L. Rougeot, "Low memory cost dynamic scheduling of large coarse grained task graphs," in *Parallel Processing Symposium, 1998. IPPS/SPDP 1998.*, Orlando, FL , mar-3 apr 1998, pp. 524-530.
- [115] A.N. Choudhary, B. Narahari, D.M. Nicol, and R. Simha, "Optimal processor assignment for a class of pipelined computations," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 5, no. 4, pp. 439-445, apr 1994.
- [116] J. C. M. Baeten, "A brief history of process algebra," *Theoretical Computer Science*, vol. 335, no. 2-3, pp. 131-146, may 2005.
- [117] Christian Zebelein, Joachim Falk, Christian Haubelt, and Jürgen Teich, "Classification of General Data Flow Actors into Known Models of Computation," in *6th ACM/IEEE International Conference on Formal Methods and Models for Co-Design, 2008. MEMOCODE 2008.*, Anaheim, CA, 2008, pp. 119-128.
- [118] W. Plishker, N. Sane, and S.S. Bhattacharyya, "A generalized scheduling approach for dynamic dataflow applications," in *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, Nice, april 2009, pp. 111-116.
- [119] Shin-ichi Minato and Shinya Ishihara, "Streaming BDD manipulation for large-scale combinatorial problems," in *Design, Automation and Test in Europe, 2001. Conference and Exhibition 2001. Proceedings*, Munich, 2001, pp. 702-707.
- [120] Richard Membarth et al., "Dynamic Task-Scheduling and Resource Management for GPU Accelerators in Medical Imaging," in *Proceedings of the 25th International Conference on Architecture of Computing Systems*, Munich, Germany, 2012, pp. 147-159.
- [121] Dave Shreiner, *OpenGL programming guide: the official guide to learning OpenGL, versions 3.0 and 3.1*. Amsterdam: Addison-Wesley, 2009.

- [122] Erik Lindholm, Mark J. Kilgard, and Henry Moreton, "A User-programmable Vertex Engine," in *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, 2001, pp. 149-158.
- [123] Shuai Mu et al., "Evaluating the potential of graphics processors for high performance embedded computing," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, Grenoble, march 2011, pp. 1-6.
- [124] M. Garland et al., "Parallel Computing Experiences with CUDA," *Micro, IEEE*, vol. 28, no. 4, pp. 13-27, july 2008.
- [125] Benedict Gaster, Timothy G. Mattson Aaftab Munshi, *OpenCL Programming Guide*. Amsterdam: Addison-Wesley Longman, 2011.
- [126] C. Nugteren, H. Corporaal, and B. Mesman, "Skeleton-based automatic parallelization of image processing algorithms for GPUs," in *Embedded Computer Systems (SAMOS), 2011 International Conference on*, Samos, july 2011, pp. 25-32.
- [127] G.-J. van den Braak, B. Mesman, and H. Corporaal, "Compile-time GPU memory access optimizations," in *Embedded Computer Systems (SAMOS), 2010 International Conference on*, Samos, july 2010, pp. 200-207.
- [128] Anthony A Apodaca and Larry Gritz, *Advanced RenderMan: Creating CGI for motion pictures.*: Morgan Kaufmann, 2000.
- [129] Fang Liu, Meng-Cheng Huang, Xue-Hui Liu, and En-Hua Wu, "CUDA renderer: a programmable graphics pipeline," in *ACM SIGGRAPH ASIA 2009 Sketches*, 2009, pp. 34:1-34:1.
- [130] Jason Zink, Matt Pettineo, and Jack Hoxley, *Practical rendering and computation with Direct3D 11.*: CRC Press, 2011.
- [131] Kun Zhou et al., "RenderAnts: interactive Reyes rendering on GPUs," in *ACM SIGGRAPH Asia 2009 papers*, Yokohama, Japan, 2009, pp. 155:1--155:11.
- [132] Zhang Ying, Peng Lu, Li Bin, Peir Jih-Kwon, and Chen Jianmin, "Architecture comparisons between Nvidia and ATI GPUs: Computation parallelism and data communications," in *2011 IEEE International Symposium on Workload Characterization (IISWC)*, Austin, TX , 2011, pp. 205-215.
- [133] L. Seiler et al., "Larrabee: A Many-Core x86 Architecture for Visual Computing," *ACM Transactions on Graphics*, vol. 27, no. 3, pp. 18:1-18:15, August 2008.

- [134] Sven Woop, Gerd Marmitt, and Philipp Slusallek, "B-KD trees for hardware accelerated ray tracing of dynamic scenes," in *Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, Vienna, Austria, 2006, pp. 67-77.
- [135] Sven Woop, Jörg Schmittler, and Philipp Slusallek, "RPU: a programmable ray processing unit for realtime ray tracing," in *ACM SIGGRAPH 2005 Papers*, Los Angeles, California, 2005, pp. 434-444.
- [136] Won-Jong Lee et al., "SGRT: a mobile GPU architecture for real-time ray tracing," in *Proceedings of the 5th High-Performance Graphics Conference*, Anaheim, California, 2013, pp. 109-119.
- [137] Jon Hasselgren and Thomas Akenine-Möller, "PCU: the programmable culling unit," *ACM Transactions On Graphics*, vol. 26, no. 3, pp. 92:1-92:10, 2007.
- [138] William R. Mark and Keko Proudfoot, "The F-buffer: A Rasterization-order FIFO Buffer for Multi-pass Rendering," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, Los Angeles, California, USA, 2001, pp. 57-64.
- [139] Fang Liu, Meng-Cheng Huang, Xue-Hui Liu, and En-Hua Wu, "Efficient Depth Peeling via Bucket Sort," in *Proceedings of the Conference on High Performance Graphics*, New Orleans, Louisiana, 2009, pp. 51-57.
- [140] Andreas A. Vasilakis and Ioannis Fudos, "K+-buffer: Fragment Synchronized K-buffer," in *Proceedings of the 18th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, San Francisco, California, 2014, pp. 143-150.
- [141] Jason C Yang, Justin Hensley, Holger Grün, and Nicolas Thibieroz, "Real-Time Concurrent Linked List Construction on the GPU," *Computer Graphics Forum*, vol. 29, no. 4, pp. 1297-1304, 2010.
- [142] Timo Aila, Ville Miettinen, and Petri Nordlund, "Delay Streams for Graphics Hardware," *ACM Transactions on Graphics (TOG)*, vol. 3, pp. 792-800, July 2003.
- [143] Fang Liu, Meng-Cheng Huang, Xue-Hui Liu, and En-Hua Wu, "FreePipe: a programmable parallel rendering architecture for efficient multi-fragment effects," in *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, Washington, D.C., 2010, pp. 75-82.
- [144] Markus Steinberger et al., "Softshell: Dynamic Scheduling on GPUs," *ACM Transactions on Graphics*, vol. 31, no. 6, pp. 161:1--161:11, November 2012.

- [145] Stanley Tzeng, Anjul Patney, and John D. Owens, "Task management for irregular-parallel workloads on the GPU," in *Proceedings of the Conference on High Performance Graphics*, Saarbrücken, Germany, 2010, pp. 29-37.
- [146] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens, "Scan Primitives for GPU Computing," in *Proceedings of the 22Nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, San Diego, California, 2007, pp. 97-106.
- [147] Samuli Laine, Tero Karras, and Timo Aila, "Megakernels Considered Harmful: Wavefront Path Tracing on GPUs," in *Proceedings of the 5th High-Performance Graphics Conference*, Anaheim, California, 2013, pp. 137-143.
- [148] Stephen Jones. (2012) Introduction to dynamic parallelism. [Online]. <http://on-demand.gputechconf.com/gtc/2012/presentations/S0338-GTC2012-CUDA-Programming-Model.pdf>
- [149] D. Sanchez, D. Lo, R.M. Yoo, J. Sugerman, and C. Kozyrakis, "Dynamic Fine-Grain Scheduling of Pipeline Parallelism," in *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, Washington, DC, USA, oct. 2011, pp. 22-32.
- [150] Louis Bavoil, Steven P. Callahan, Aaron Lefohn, Joao L. D. Comba, and Claudio T. Silva, "Multi-fragment Effects on the GPU Using the K-buffer," in *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, Seattle, Washington, 2007, pp. 97-104.
- [151] Nan Zhang, "Memory-Hazard-Aware K-Buffer Algorithm for Order-Independent Transparency Rendering," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 2, pp. 238-248, Feb 2014.
- [152] Tim Foley and Pat Hanrahan, "Spark: modular, composable shaders for graphics hardware," *ACM Transactions on Graphics*, vol. 30, no. 4, pp. 107:1-107:12, July 2011.
- [153] Jean-Eudes Marvie, Cyprien Buron, Pascal Gautron, Patrice Hirtzlin, and Ga, "Gpu shape grammars," *Computer Graphics Forum*, vol. 31, no. 7pt1, pp. 2087-2095, September 2012.
- [154] Cyprien Buron, Jean-Eudes Marvie, and Pascal Gautron, "GPU Roof Grammars," in *Eurographics 2013-Short Papers*, 2013, pp. 85-88.
- [155] Qiming Hou, Kun Zhou, and Baining Guo, "BSGP: bulk-synchronous GPU programming," *ACM Transactions on Graphics*, vol. 27, no. 3, pp. 19:1-19:12, August 2008.

- [156] Ralf Karrenberg, Dmitri Rubinstein, Philipp Slusallek, and Sebastian Hack, "AnySL: efficient and portable shading for ray tracing," in *Proceedings of the Conference on High Performance Graphics*, Saarbrücken, Germany, 2010, pp. 97-105.
- [157] Conal Elliott, "Programming Graphics Processors Functionally," in *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, Snowbird, Utah, USA, 2004, pp. 45-56.
- [158] Chad Austin and Dirk Reiners, "Renaissance: A functional shading language," in *Proceedings of Graphics Hardware*, New York, 2005, pp. 1-8.
- [159] Joel Svensson, Koen Claessen, and Mary Sheeran, "GPGPU kernel implementation and refinement using Obsidian," in *Procedia Computer Science, ICCS 2010*, 2010, pp. 2059-2068.
- [160] Nathan Bell and Jared Hoberock, "Thrust: A productivity-oriented library for CUDA," *GPU Computing Gems*, vol. 7, pp. 359-373, 2011.
- [161] Yi Yang and Huiyang Zhou, "CUDA-NP: Realizing Nested Thread-level Parallelism in GPGPU Applications," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '14*, Orlando, Florida, USA, 2014, pp. 93-106.
- [162] Lars Bergstrom and John Reppy, "Nested data-parallelism on the gpu," in *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, Copenhagen, Denmark, 2012, pp. 247-258. [Online].  
<http://doi.acm.org/10.1145/2364527.2364563>
- [163] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron, "Scalable Parallel Programming with CUDA," *Queue*, vol. 6, no. 2, pp. 40-53, March 2008.
- [164] Xin Huo, Sriram Krishnamoorthy, and Gagan Agrawal, "Efficient Scheduling of Recursive Control Flow on GPUs," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, Eugene, Oregon, USA, 2013, pp. 409-420.
- [165] Hanan Samet, *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1990.
- [166] Mathias Holst and Heidrun Schumann, "Normal Mapping for Surfel-Based Rendering," *Journal of WSCG*, pp. 9-16, 2007.

- [167] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek, "Stackless KD-Tree Traversal for High Performance GPU Ray Tracing," *Computer Graphics Forum*, vol. 26, no. 3, pp. 415-424, 2007.
- [168] Henry Schäfer, Matthias Nießner, Benjamin Keinert, Marc Stamminger, and Charles Loop, "State of the Art Report on Real-time Rendering with Hardware Tessellation," *Eurographics 2014 - State of the Art Reports*, pp. 93-117, 2014.
- [169] Falko Löffler, Andreas Müller, and Heidrun Schumann, "Real-time rendering of stack-based terrains," in *Proceedings of the 16th Annual Workshop on Vision, Modeling and Visualization*, Berlin, Germany, 2011, pp. 161-168.
- [170] Hyunjin Lee, Yuna Jeong, and Sungkil Lee, "Recursive Tessellation," in *SIGGRAPH Asia 2013 Posters*, Hong Kong, Hong Kong, 2013, pp. 16:1--16:1.
- [171] Michael Schwarz and Marc Stamminger, "Fast GPU-based Adaptive Tessellation with CUDA," *Computer Graphics Forum*, vol. 28, no. 2, pp. 365-374, 2009.
- [172] Kirill Garanzha, Jacopo Pantaleoni, and David McAllister, "Simpler and Faster HLBVH with Work Queues," in *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, Vancouver, British Columbia, Canada, 2011, pp. 59-64.
- [173] Qiming Hou, Kun Zhou, and Baining Guo, "Debugging GPU Stream Programs Through Automatic Dataflow Recording and Visualization," *ACM Transactions on Graphics*, vol. 28, no. 5, pp. 153:1--153:11, December 2009.
- [174] Magnus Strengert, Thomas Klein, and Thomas Ertl, "A hardware-aware debugger for the OpenGL shading language," in *Proceedings of the 22Nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, San Diego, California, 2007, pp. 81-88.
- [175] Juan Pineda, "A parallel algorithm for polygon rasterization," in *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, 1988, pp. 17-20. [Online]. <http://doi.acm.org/10.1145/54852.378457>
- [176] Samuli Laine and Tero Karras, "High-performance software rasterization on GPUs," in *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, Vancouver, British Columbia, Canada, 2011, pp. 79-88.
- [177] Chris Lomont. (2003) Fast inverse square root. [Online]. [www.lomont.org/Math/Papers/2003/InvSqrt.pdf](http://www.lomont.org/Math/Papers/2003/InvSqrt.pdf)
- [178] Michael Abrash, "Rasterization on larrabee," *Dr. Dobbs Journal*, 2009.

- [179] Marc Olano and Trey Greer, "Triangle scan conversion using 2D homogeneous coordinates," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, Los Angeles, California, USA, 1997, pp. 89-95.
- [180] Tamy Boubekeur and Marc Alexa, "Phong tessellation," *ACM Transactions on Graphics (TOG)*, vol. 27, no. 5, pp. 141:1-141:5, 2008.
- [181] Stephen W Keckler, William J Dally, Brucec Khailany, Michael Garland, and David Glasco, "GPUs and the future of parallel computing," *IEEE Micro*, vol. 31, no. 5, pp. 7-17, September 2011.
- [182] Alex Vlachos, Jörg Peters, Chas Boyd, and Jason L. Mitchell, "Curved PN Triangles," in *Proceedings of the 2001 Symposium on Interactive 3D Graphics*, vol. I3D '01, New York, NY, USA, 2001, pp. 159-166.
- [183] James T Kajiya, "The rendering equation," in *ACM Siggraph Computer Graphics*, vol. 20, 1986, pp. 143-150.
- [184] Dietger van Antwerpen, "Improving SIMD Efficiency for Parallel Monte Carlo Light Transport on the GPU," in *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, Vancouver, British Columbia, Canada, 2011, pp. 41-50.
- [185] Yoichiro Kawaguchi, "A Morphological Study of the Form of Nature," *SIGGRAPH Computer Graphics*, vol. 16, no. 3, pp. 223-232, July 1982.
- [186] M.C. McFarland, A.C. Parker, and R. Camposano, "The high-level synthesis of digital systems," *Proceedings of the IEEE*, vol. 78, no. 2, pp. 301-318, feb 1990.
- [187] G. Martin and G. Smith, "High-Level Synthesis: Past, Present, and Future," *Design Test of Computers, IEEE*, vol. 26, no. 4, pp. 18-25, july-aug. 2009.
- [188] Luc and Micheli, Giovanni De Semeria, "Resolution, optimization, and encoding of pointer variables," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 20, no. 2, pp. 213-233, 2001.
- [189] Luc and Sato, Koichi and Micheli, Giovanni De Semeria, "Synthesis of hardware models in C with pointers and complex data structures," *IEEE Trans. VLSI Syst.*, vol. 9, no. 6, pp. 743-756, 2001.
- [190] S.A. Edwards, "The challenges of hardware synthesis from C-like languages," in *Design, Automation and Test in Europe, 2005. Proceedings*, vol. 1, Munich, march 2005, pp. 66-67.

- [191] I. Skliarova and V. Sklyarov, "Recursion in reconfigurable computing: A survey of implementation approaches," in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, 31 2009-sept. 2 2009, pp. 224-229.
- [192] V. Sklyarov, I. Skliarova, and B. Pimentel, "FPGA-based implementation and comparison of recursive and iterative algorithms," in *Field Programmable Logic and Applications, 2005. International Conference on*, aug. 2005, pp. 235-240.
- [193] Dan R. and Smith, Alex and Singh, Satnam Ghica, "Geometry of synthesis iv: compiling affine recursion into Static Hardware," in *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, Tokyo, Japan, 2011, pp. 221-233.
- [194] George and ElGindy, Hossam A. Ferizis, "Mapping Recursive Functions to Reconfigurable Hardware," in *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, Madrid, 2006, pp. 1-6.
- [195] M. S. Chandrasekhar, J. P. Privitera, and K. W. Conradt, "Application of term rewriting techniques to hardware design verification," in *Proceedings of the 24th ACM/IEEE Design Automation Conference*, New York, NY, USA, 1987, pp. 277-282. [Online]. <http://doi.acm.org/10.1145/37888.37930>
- [196] James C. and Arvind Hoe, "Hardware Synthesis from Term Rewriting Systems," in *VLSI: Systems on a Chip.*: Springer US, 2000, vol. 34, pp. 595-619.
- [197] N. Dave, A. Pellauer, and M. Pellauer, "Scheduling as Rule Composition," in *Formal Methods and Models for Codesign, 2007. MEMOCODE 2007. 5th IEEE/ACM International Conference on*, Nice, 30 2007-june 2 2007, pp. 51-60.
- [198] M. von Thun. (1996) A rewriting system for Joy. [Online]. <http://www.latrobe.edu.au/humanities/research/research-projects/past-projects/joy-programming-language>

## Abstract

Multi and many-core systems offer numerous benefits like reduced energy consumption and latency, as well as improved throughput for both high-performance and low-power applications. Scalability is achieved by parallelization instead of high clock rates and can therefore overcome technological limitations like thermal heat or power issues. While functional units and also processor cores can be replicated at the expense of increased area, the utilization of the additional resources remains a more fundamental problem. Hence, beside the design of the actual hardware architecture, also the programming of a many-core system raises several challenges. Especially in case of irregular tasks and dynamic parallelism, the binding and scheduling of tasks to processor cores as well as the efficient communication and synchronization at the system level are not yet solved in general. As a consequence, the majority of static optimizations, which are based on an extensive knowledge of the behavior at design time, cannot be applied in this case.

This thesis proposes a novel model of computation, called *stream rewriting*, for the specification and implementation of highly current applications. Basically, the active tasks of an application and their dependencies are encoded as a token stream, which is iteratively modified by a set of rewriting rules at runtime. As a result, the creation of new tasks, the synchronization of cooperating tasks, and the scheduling of dependent tasks are implemented as local pattern matching, which can be performed in parallel on several regions of the stream and does not require a central scheduler. Hence, stream rewriting is most useful for compute-intensive applications with frequently varying and unpredictable data rates and further enables global resource sharing as well as lightweight lock-free synchronization. In addition, this thesis presents a balanced scheduling algorithm, which restricts the memory usage of dynamic and recursive computations, by switching automatically between deep-first and breadth-first traversal.

The concept of stream rewriting has been extensively evaluated for several different use cases and all results presented in this thesis have been either retrieved from a cycle accurate HDL simulation or an FPGA implementation. For example, stream rewriting has been utilized in a high-level synthesis tool to compile recursive functions into hardware. Several many-core systems with up to 128 general purpose processors have been implemented and show the scalability of stream rewriting for complex examples and recursive algorithms.

In particular, the concept of stream rewriting fits perfectly to the requirements of graphics processing. Since both data and pipeline parallelism are available and match the requirements of specialized rendering algorithms, previously fixed blocks like the rasterizer or tessellator stages can be stored in a software library. Recursive and indirect function calls allow generating workloads dynamically at runtime, so that both rasterization and ray-tracing can be implemented as a shader program. A novel graphics processor has been designed, which supports custom rendering pipelines, recursion, complex atomic operations and incoherent workloads. It combines the performance advantages of hardware-based scheduling through stream rewriting with the flexibility of general purpose processors and thus provides worthwhile insights for future GPU architectures.

## Kurzfassung

Mehrkernprozessoren ermöglichen im Gegensatz zu Einkernprozessoren einen verringerten Energieverbrauch, eine geringere Latenz sowie ein höherer Durchsatz für rechenintensive Anwendungen. Anstelle von einer immer höheren Taktrate, basiert die Leistungssteigerung dieser System auf der Parallelisierung von Berechnungen, so dass die üblichen Probleme bei hohen Frequenzen wie Wärmeentwicklung oder Stromverbrauch umgangen werden können. Dabei stellt die Duplizierung von einzelne Funktionseinheiten und Prozessoren auf Hardwareebene oft kein unlösbares Problem mehr dar und macht sich lediglich in einem höheren Platzverbrauch bemerkbar. Die effiziente Nutzung der zusätzlichen Funktionalität ist dagegen die größere Herausforderung und hängt von einer Vielzahl ungelöster Fragen ab. Diese Arbeit beschäftigt sich insbesondere mit dem Problem, wie man eine große Anzahl an dynamischen und irregulären Tasks in einem Mehrkernsystem auf die Prozessoren verteilen und einplanen kann. Besonders zu beachten ist hierbei, dass ohne Vorwissen über die Art Anwendungen und die Aufgabenverteilung, viele der bekannten Optimierungen für statische Systeme hier nicht angewandt werden können.

In dieser Dissertation wurde daher *Stream Rewriting* als eine neue Methode entwickelt um Anwendungen mit einer großen Anzahl von dynamischen Tasks zu beschreiben und effizient zur Laufzeit verwalten zu können. Dabei werden die aktiven Tasks in einem Datenstrom verpackt, der zur Laufzeit durch wiederholtes Suchen und Ersetzen fortlaufen umgeschrieben wird. Sowohl das Erstellen von neuen Tasks, als auch die Synchronisierung von abhängigen Tasks können dabei durch Ersetzungsregeln dargestellt werden, die jeweils nur auf einem begrenzten Abschnitt des Streams arbeiten und daher auf mehreren Prozessoren parallel ausgeführt werden können. Diese Technik ist daher besonders für rechenintensive Anwendungen mit vielen und nicht vorhersehbaren Tasks geeignet.

Um die Performance und Skalierbarkeit von Stream Rewriting zu bestimmen, wurde eine Vielzahl von Experimenten mit Multi-Core Systemen durchgeführt. Zu einem wurden Mehrkernprozessoren mit bis zu 128 Kernen auf einem FPGA aufgebaut und die Verwaltung von Tasks über Stream Rewriting in Software und Hardware implementiert. Tests mit verschiedenen Anwendungen zeigen die Skalierbarkeit dieser Technik für eine große Anzahl an dynamischen und rekursiven Tasks. Außerdem eignet sich Stream Rewriting besonders gut für Grafikverarbeitung, da es sowohl Pipeline- als auch Datenparallelität unterstützt und so die Implementierung von neuen Rendering-Algorithmen erleichtert. Der vorgestellte Grafikprozessor basiert auf Stream Rewriting und ermöglicht es benutzerdefinierte Rendering Pipelines, Rekursion, sowie komplexe atomare Operationen und irreguläre Tasks zu definieren. Die neue Architektur kombiniert eine hardwarebasierte Taskverwaltung über Stream Rewriting mit der Flexibilität von Softwarerendering und zeigt so Möglichkeiten für die Entwicklung von zukünftigen Grafikprozessoren auf.

## Theses

### Stream Rewriting

- Stream rewriting is a novel model of computation for the specification, analysis and implementation of parallel applications. It encapsulates both data and pipeline parallelism to manage a large number of dynamic, irregular and also recursive expandable tasks without a central scheduler.
- Rewriting operations modify locally constrained and non-overlapping intervals of the stream and can be therefore performed on different regions in parallel. The scalability of this approach is demonstrated by several hardware and software architectures for parallel stream rewriting.
- The execution order of tasks is defined implicitly by data dependencies, so that an implementation can choose an optimal schedule at runtime in order to maximize utilization of available processing units.
- The results of concurrent threads are synchronized via pattern matching on the stream, so that the usage of shared memory or atomic operations can be omitted.
- Although data parallelism is supported, stream rewriting has been especially designed for handling irregular and dynamic work load. For this purpose, the basic algorithm requires a single rewriting rule and reduces all scheduling decisions into find-and-replace operations.
- Access to shared resources is serialized by dynamically binding interfering tasks to the same processor. This approach enables complex atomic read-modify-write operations in many-core systems without global locks or bus snooping and is also suitable for non-uniform memory.
- The stack-based scheduling automatically balances between depth-first and breadth-first traversal of the call tree to avoid an exponential growth of the token stream while still maintaining a large degree of concurrency. Most important, only a small part of the potentially large stream must be accessible, so that the remainder can be paged out into external memory.
- Local environments redefine the value of a global variable for restricted sub-stream and therefore offer an efficient mechanism for the distribution of shared data in a highly concurrent environment. At the end of computation, environments are automatically garbage collected and removed from the stream.

- Instanting reduces the size of the stream by specifying data parallel rewriting rules only once, so that multiple iterations can be expanded on demand.
- The token stream serves as a platform independent representation of a task graph or a functional program. Since each processor is only responsible for replacing its own patterns in the stream, different software or hardware components can cooperate without explicit knowledge of each other.

### **Source Models**

- The basic execution model of stream rewriting corresponds to a functional language without side effects, which is well suited for modeling highly concurrent tasks. Hence, the implementations of the stream rewriting machine are in fact hardware and software interpreters for this functional language.
- The sub-class of series-parallel (SP) task graphs can be translated into stream rewriting programs. In addition, stream rewriting allows adapting the topology of the graph to varying workloads at runtime by the recursive expansion of nodes.
- High-Level C programs can be converted into rewriting rules and therefore benefit from the parallel scheduling of recursive invocations. This thesis presents a compiler for the hardware synthesis of a C-like language, which generates HDL code for a specialized stream rewriting machine by mapping the control flow between basic blocks into rewriting rules.

### **Multi-Core Architectures**

- Stream rewriting can be implemented as a thin software library for general purpose many-core architectures and embedded systems with strict resource constraints. Hence, the concept of stream rewriting is not tied to a specific execution platform.
- Due to the simplicity of the pattern matching, both the binding and the scheduling of rewriting rules can be also implemented in hardware, while the functionality of the tasks remains in software for maximum flexibility. The resulting stream rewriting network connects up to 128 general purpose cores and the FPGA implementation indicates scalability for a large number of complex test cases.

## Graphics Processing

- Despite the computational power and memory bandwidth of modern graphics processing units (GPU), a main limitation of these architectures is often the lack of efficient on-chip communication between different shader cores. The dynamic binding of stream rewriting helps to overcome these issues and leads to a more flexible architecture for graphics processing units.
- Although individual stages of the Direct3D and OpenGL rendering pipeline are programmable, its topology and data flow are fixed. This thesis describes a graphics processor based on stream rewriting, which supports an arbitrary amount of dynamic and recursive shader stages as well as complex data flow and light-weight synchronization within the rendering pipeline.
- The semantic gap between the sequential execution model of the CPU and the highly parallel but more restricted GPUs require an application to be split into two distinct parts, which are developed using separate languages and optimized according to different rules. In this context, stream rewriting offers a unified execution model for task, pipeline and data parallelism as well as support for individual threads.
- Most important, the concept of stream rewriting for graphics processing has been evaluated as part of a complete test system consisting of several applications, an OpenGL driver, a kernel mode driver, and the FPGA prototype connected via PCIe. Despite the lower clock rate of the FPGA, the measurements include the delays and latencies of the whole system, which would also occur for an ASIC design.
- This thesis proposes a novel OpenGL extension, which integrates stream rewriting into the existing programming model and adds the stream rewriting shader as a new general purpose shader stage.
- Stream rewriting greatly reduce the complexity of several advanced rendering techniques, like order-independent transparency via K-buffering, path-tracing and the recursive generation of procedural geometry, which require significant effort to run efficiently on current GPUs.

# Lebenslauf

## 1. Allgemeine Informationen

**Persönliche Angaben:** Geburtsort: Iserlohn  
Geburtstag: 21.09.1982

**Adresse:** Lars Middendorf  
Flensburger Str. 26  
18109 Rostock

**Kontakt:** Telefon: 0151 1447808  
E-Mail: [lmid@gmx.de](mailto:lmid@gmx.de)

## 2. Ausbildung

**Schulbildung:**  
Aug. 1989 – Juli 1993 Brabeckschule Hemer  
Aug. 1993 – Juni 2002 Friedrich-Leopold-Woeste-Gymnasium, Hemer, Abitur

**Wehrdienst:**  
Juli 2002 – März 2003 Grundwehrdienst in Goslar, Burbach und Hemer

**Studium:**  
April 2003 – März 2007 Studium Diplom Informatik, TU Kaiserslautern

## 3. Arbeitsstellen

**RG Technologies GmbH**  
Juni 2007 – April 2009 Softwareentwickler für ein CAD System bei der RG Technologies GmbH in Unterhaching

**Universität Potsdam**  
Mai 2009 – Juli 2011 Wissenschaftlicher Mitarbeiter am Institut für Informatik der Universität Potsdam

**Universität Rostock**  
Seit August 2011 Wissenschaftlicher Mitarbeiter am Institut für Angewandte Mikroelektronik und Datentechnik der Universität Rostock

## Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Dissertation mit dem Titel:

**„Dynamic Task Scheduling and Binding for Many-Core Systems  
through Stream Rewriting “**

selbstständig verfasst und ausschließlich die angegebenen Quellen und Hilfsmittel in Anspruch genommen habe, sowie alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht habe. Die vorliegende Arbeit wurde in gleicher oder ähnlicher Fassung bisher nicht als Prüfungsarbeit zur Begutachtung vorgelegt.

Rostock, 28.03.2015

---

Lars Middendorf