Compiling Stream Applications for Heterogeneous Architectures

by

Amir H. Hormati

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2011

To my family

ACKNOWLEDGEMENTS

First, I would like to express my sincerest gratitude to my adviser Prof. Scott Mahlke. I consider myself truly lucky to have worked with him these past years. He has shown incredible patience, served as an excellent mentor, and provided me every opportunity to succeed in this field.

I also owe thanks to the remaining members of my dissertation committee, Prof. Austin, Prof. Mudge, Prof. Sylvester and Dr. Rodric Rabbah. They all donated their time to help shape this research into what it has become today. I would particularly like to thank Rodric for his insightful comments and invaluable advice during my internships at IBM T.J. Watson that helped me in finding an interesting research path.

I was lucky to be part of a research group whose members not only assisted me intellectually in my research but were also a comfort during those long nights before each deadline. Nathan Clark helped me in the first two of years of my PhD. His patience and technical help was the reason I survived those years. Mark Woh spent a countless number of hours discussing new ideas with me and helping me write my papers. Shuguang Feng, Shantanu Gupta, Ganesh Dasika, and Mojtaba Mehrara also helped in proof reading the papers and refining my ideas. Mehrzad Samadi did a great deal of work on the part of this thesis presented in Chapter V.

More importantly than the technical assistance, I would like to thank all the members of the CCCP research group who I've ever shared an office with over the years for their social support: Kevin Fan, Mike Chu, Manjunath Kudlur, Amin Ansari, Hyoun Kyu Cho, Ganesh Dasika, Shuguang Feng, Shantanu Gupta, Po-Chun Hsu, Mojtaba Mehrara, Yongjun Park, Hyunchul Park and Jeff Hao. You folks made coming to work a lot more fun, and I would never have made it through without you. I also want to thank Ganesh Dasika for letting me beat him in Quake. Finishing a PhD without playing Quake would have not been possible.

Finally and most importantly, my family deserves major gratitude. My parents and my sister provided their unconditional love and support through this whole process. I also want to thank my brother who helped me all the way through middle school and high school. My cousin, Abbas, was the reason I chose computer engineering as my major in the first place and I want to thank him for showing me this path. And above all, I really appreciate the love and support of my wife, Mona. Getting through the PhD program, especially in a foreign country, would have not been possible without her.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Compiling Stream Applications for Heterogeneous Architectures

by

Amir H. Hormati

Chair: Scott Mahlke

Heterogeneous processing systems have become the industry standard in almost every segment of the computing market from servers to mobile systems. In addition to employing shared/distributed memory processors, the current trend is to use hardware components such as field programmable gate arrays (FPGAs), single instruction multiple data (SIMD) engines and graphics processing units (GPUs) in heterogeneous systems.

As a result of this trend, extracting maximum performance requires compilation to highly heterogeneous architectures that include parts with different memory and computation models. Although there has been significant amount of research on programing each of these architectures individually, targeting a heterogeneous system without specializing an application to each component separately is still an open problem. Besides performance, the portability of an application between different pieces of a system and retargetability to various heterogeneous architectures is a significant challenge for programmers. To efficiently exploit the heterogeneity, it is necessary to have a programming model that provides a higher-level of abstraction to the programmer and the related compilation framework.

In this thesis, we first focus on enabling a write-once programming paradigm in the context of the stream programming model for various components of heterogeneous systems. We mainly focus on FPGAs, SIMD engines and GPUs as these architectures will play an important role in accelerating various parts of applications on heterogeneous systems. We introduce several compiler optimizations that facilitate portability and retargetability while achieving high performance. As a result of our compilation system, programmers can write a program once and efficiently run it on different components of a system.

Second, we focus on an important challenge that arises in heterogeneous systems when there are dynamic resource changes. The ability to dynamically adapt a running application to a target architecture in the face of changes in resource availability (e.g., number of cores, available memory or bandwidth) is crucial to a wider adoption of heterogeneous architectures. In this work, we introduce a hybrid flexible compilation framework that facilitates dynamic adaption of applications to the changing characteristics of the underlying architecture.

CHAPTER I

Introduction

Support for parallelism in hardware has greatly evolved in the past decade as a response to the ever-increasing demand for higher performance and better power efficiency in different application domains. Various companies have introduced vastly different solutions to bridge the performance and power gap that many applications are facing. These solutions include shared-memory multicore systems (Intel Core i7 [43]), distributed-memory multicore processors (IBM Cell [40]), tiled architectures (Tilera [80]) and in some cases a combination of these (Intel Larrabee [72] and Intel Stellarton [45]). Among these solutions, heterogeneous architectures, as shown in Figure 1.1, not only achieve higher performance and efficiency by combining multiple cores into one die, but they are also equipped with acceleration engines to enable more efficient parallelism support for certain application domains. For example, SIMD engines (e.g., Altivec [73], Neon [6], SSE4 [42]) integrated into multi-core systems enable more efficient data-level parallelism support for several important application domains such as multimedia, graphics, and encryption. Although acceleration engines, such as SIMD units or FPGAs, are not suitable for all applications, if an application can be tailored to efficiently exploit them, the performance and power benefits

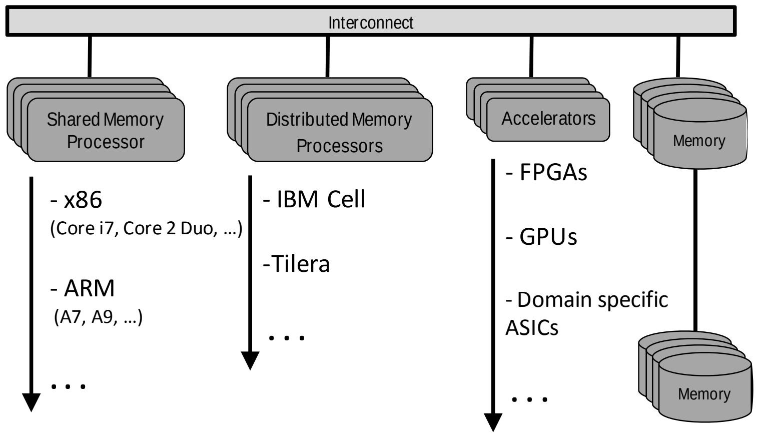


Figure 1.1: *This figure shows a template for future heterogeneous systems.*

can often be superior to the gains from other general purpose architecture solutions.

Programming heterogeneous architectures is an important problem that is impeding the wider adoption of such systems. Traditional sequential programming languages are ill-suited for heterogeneous architectures because they have a single instruction stream and a monolithic memory. Extracting task/pipeline/data-level parallelism from these languages needs extensive and often intractable compiler analysis. Using a different programming model and compilation framework for each component of the system is also undesirable because it limits the portability and retargetability of the program requiring *each program to be rewritten and optimized for a specific architecture*. Architecture-specific programming models and languages, such as Verilog and CUDA [65], that target specific components, such as FPGAs and GPUs, expose parallelism to the compiler, but in their current form fail to provide portable code and do not present a unified model to the programmer. The main problem with these languages is that explicitly-programmed parallelism in each application has to be tuned for different targets based on the parameters of each hardware component and interfacing between different parts of an application written in different architecture-specific languages is non-trivial.

A higher level of programming abstraction along with intelligent static and dynamic compiler optimizations can solve the issues of programming heterogeneous systems while maintaining portability and retargetability. One such abstraction is offered by the streaming paradigm. This programming paradigm provides an extensive set of compiler optimizations for mapping and scheduling applications to various parallel architectures ([25, 26]). The retargetability of streaming languages, such as StreamIt [79], has made them a good choice for parallel system programmers. Streaming language retargetability and the resulting performance benefits on multi-core systems are mainly due to having well-encapsulated constructs that expose the parallelism and communication without depending on the topology or granularity of the underlying architecture. Compilers for these languages take advantage of the high-level information available at the program level to efficiently map the exposed parallelism to the target architecture.

Most of the work on stream compilation has so far focused on how to compile streaming applications to homogeneous multi-core systems. However, compiling stream programs to other important components of heterogeneous architectures, such as FPGAs, SIMD engines and GPUs, is still an open question. In this thesis, we propose new techniques and compilation frameworks for static and dynamic compilation of programs in the streaming domain, specifically those implemented in synchronous data flow (SDF, see Chapter II) model, to various components of heterogeneous systems. Our techniques further extend the retargetability and portability of streaming applications by enabling programmers to write a streaming application once and efficiently run it on various parts of a system. An overview of our system is shown in Figure 1.2. The following sections briefly explain the

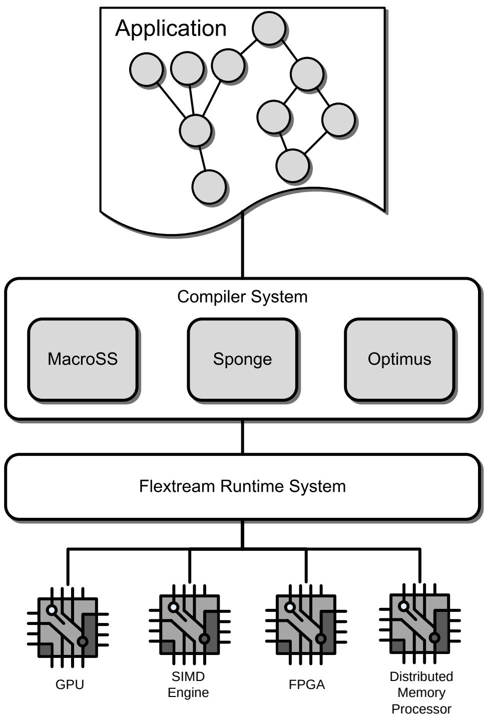


Figure 1.2: *Overview of our heterogeneous compilation system.*

four parts of our compiler and runtime system: Optimus, Macross, Sponge and Flextream.

1.1 Streaming to FPGAs

In the world of heterogeneous systems, especially embedded architectures, there are many devices that offer increasingly powerful computing capabilities. It is predicted that mobile computing devices with embedded processors will ultimately change the industry much as laptops supplanted desktops as the primary commodity processing platform. However, the power and frequency concerns that plague the microprocessor industry effectively mean architects have to find new ways to provide increasing performance since conventional frequency scaling methodologies no longer apply. As a result, there is a significant opportunity to explore alternate architectures that can enable the next evolutionary step in computing.

One significantly promising approach is to provide automatic customization of hardware according to the applications they run. An application-customized architecture can offer extremely high performance with very low power compared to a more general-purpose design. Furthermore, the increasing availability of reconfigurable field-programmable gate arrays (FPGAs) as co-processors and processing ingredients in heterogeneous systems-on-a-chip [1, 2, 45] means emerging architectures can offer enormous flexibility and adaptability in the face of rapidly changing software standards and customer needs.

In the first part of this thesis, we introduce *Optimus* [37], an optimizing synthesis framework for streaming applications. The main contribution of Optimus is generating hardware by shifting the focus from micro-functional details to macro-functional ones. Specifically, our work does not focus so much on how individual modules are synthesized (i.e., micro-functional), but rather on how modules are composed to assemble an overall design (i.e., macro-functional). As a result, we can synthesize entire applications onto a hardware substrate, and not just individual loops and kernels as is the case with a lot of existing work. Thus, our work is complementary to existing work on high level synthesis while offering new opportunities for efficient assembly of streaming applications in hardware.

1.2 Streaming to SIMD Engines

In recent years, almost every single-core or multi-core system has been equipped with one or more single-instruction-multiple-data (SIMD) engines to enable more efficient data-level parallelism support for several important application domains such as multimedia, graphics, and encryption. SIMD engines are not the right choice for all applications, but in cases where an application can efficiently exploit them, the performance and power gains can be significant. Therefore, SIMD engines like Altivec [73], Neon [6], SSE4 [42] are now an essential part of most architectures on the market. With SIMD width expanding in future architectures, such as Intel's Larrabee, under-utilization of the SIMD units will translate into a significant loss in performance and increase in power consumption.

In general, utilizing SIMD engines is preferred, even for applications where multi-core speedup is close to the theoretical maximum, because SIMD engines can improve performance without increasing communication overhead and memory/cache traffic. Exploiting SIMD engines, in some cases, can achieve greater performance than multi-core while using less area and power.

To exploit SIMD engines in streaming applications, current streaming compilers translate the streaming languages down to an intermediate language, such C++ or Java, and then apply vectorization[1] techniques to generate SIMD-enabled code. The most common techniques are hand-optimizing the code and traditional auto-SIMDization [3, 4, 64, 5, 51]. Both of these solutions have proven difficult to apply in real world scenarios. Hand-optimizing the binary or sequential code using architecture-specific instructions or intrinsic functions is a time-consuming and error-prone task which results in an inflexible and unportable binary. Auto-vectorization is, at this stage, still impractical and far from being able to universally utilize the various kinds of available SIMD facilities. Also, performing SIMDization on streaming applications after intermediate-level code generation may result in an inefficient schedule and mapping of the stream graph since the schedule is already

[1]In this work, we use SIMD(ize) and Vector(ize) interchangeably.

fixed and information that is available in the high-level stream graph is lost. Extracting this information from the generated code is predicated on performing complex compiler analysis and transformations which are impossible in some cases. In summary, *the lack of global knowledge about the program, the inability to adjust the schedule,* and also *the loss of data flow information* are the main reasons behind inefficiency of traditional auto-vectorization techniques in dealing with streaming applications.

To address these issues, we introduce *MacroSS* [36]; a streaming compiler for streaming applications that is capable of performing macro-SIMDization on stream graphs. Macro-SIMDization uses high-level information such as the valid set of schedules and communication patterns between actors to transform the graph structure, vectorize actors of a streaming program, and generate intermediate code (C++ in this work). Then, it uses the host compiler to compile the generated intermediate code to binary for a specific target processor. The information that is used by MacroSS is deduced from the high-level program structure and is not available to low-level traditional compilers that are used to compile the intermediate code. As a result, MacroSS has a broader understanding of the program structure and macro-level characteristics of the streaming application that allows the compiler to utilize SIMD engines more efficiently.

1.3 Streaming to GPUs

Recently, heterogeneous systems that combine traditional processors with powerful GPUs have become standard in all systems ranging from servers to cell phones. GPUs achieve their high performance and efficiency by providing a massively parallel architecture with hundreds of in-order cores while exposing parallelism mechanisms and the memory hierarchy to the programmer. Recent works have shown that in the optimistic case, speedups of 100-300x [67] and in the pessimistic case, speedups of 2.5x [54] have been achieved between the most recent versions of GPUs compared to the latest processors. Maximizing the utilization of the GPU in heterogeneous systems will be key to achieving high performance and efficiency.

While GPUs provide an inexpensive, highly parallel system for accelerating parallel workloads, the programming complexity posed to application developers is a significant challenge. Graphics chip manufacturers, such as NVIDIA, have tried to alleviate the complexity problem by introducing user-friendly programming models, such as CUDA [65] and OpenCL [48]. Although such programming models abstract the underlying GPU architecture by providing a unified processor model, managing the amount of on-chip memory used per thread, the total number of threads per multiprocessor, and the pattern of off-chip memory accesses are examples of problems that developers still need to manage in order to maximize GPU utilization [70]. Often the programmer must perform a tedious cycle of performance tuning to extract the desired performance.

Another problem of developing applications in CUDA is the lack of efficient portability between different generations of GPUs and also between the host processors and GPUs in the system. Different NVIDIA GPUs vary in several key micro-architectural parameters such as number of registers, maximum number of active threads, and the size of global memory. These parameters will vary even more when newer high performance cards, such as NVIDIA's Fermi [66], and future resource-constrained mobile GPUs with less resources are released. These differences in hardware lead to a different set of optimization choices for each GPU. As a result, optimization decisions for one generation of GPUs are likely to be poor choices for another generation.

One solution to the complexity of GPU programming is to adopt a higher level programming abstraction such as the stream programming model. The higher level domain information exposed as a result of employing the streaming programing model can guide the compiler optimizations in generating high-quality code for GPUs. Therefore, using stream programming model, programmers can implement their application without worrying about the parameters of the underlying hardware and the compiler can perform intelligent optimizations to tune the available parallelism in a streaming application to a specific GPU.

In this thesis, we introduce *Sponge* [38], a compiler for the StreamIt language that is capable of producing customized CUDA code for a wide range of GPUs. Sponge consists of stream graph optimizations to optimize the organization of the computation graph and an efficient CUDA code generator to express the parallelism for the target GPU. Producing efficient CUDA code is a multi-variable optimization problem and can be difficult for software programmers due to the unconventional organization and the interaction of computing resources of GPUs. Sponge is equipped with a set of optimizations to handle the memory hierarchy of GPUs and also to efficiently utilize the processing units.

1.4 Flexible Compilation for Streaming

As the number of applications that can effectively use multiple cores increases, it will become necessary to develop strategies that can adequately manage the allocation of resources between applications. Resource allocation is a challenging problem because application behavior (and hence resource requirements) can often vary in unpredictable ways, depending on factors that include dynamic workloads and variability in end-user scenarios. The issue is made more challenging by the numerous heterogeneous architectural resources that are already exposed to software (e.g., the compiler). We believe that managing the allocation of resources effectively requires many non-trivial tradeoffs, and we introduce Flextream [35] as a means to address this issue.

Flextream provides a compilation and runtime adaptation system for distributed memory heterogeneous systems. It is aimed at addressing the challenges described above in the context of streaming applications. The main innovation in Flextream is an *adaptive stream graph modulo scheduling* algorithm that combines the benefits of static scheduling with the advantages of dynamic adaptation. The strategy of using an adaptive hybrid (static-dynamic) compilation approach can lead to significantly better resource utilization, and can help deliver the promise of many-cores to end-users.

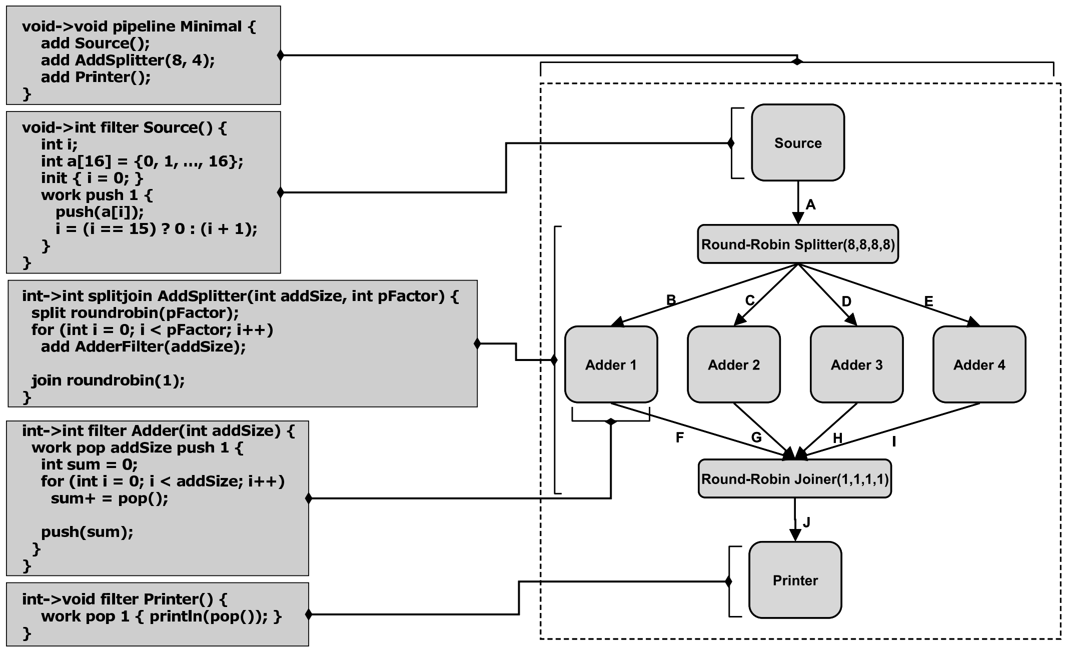
The rest of this thesis is organized as follows. The streaming model used throughout this thesis is explained in Chapter II. Then, Optimus, our synthesizing compiler is introduced in Chapter III. In Chapter IV, we demonstrate how streaming applications can be mapped to SIMD engines using MacroSS compiler. Then, Chapter V explains our compilation system, Sponge, for mapping streaming applications to GPUs. In Chapter VI, details of static compilation and online adaptation in Flextream for adjusting the schedule of stream programs in the presence of runtime resource changes is discussed. Finally, we conclude in Chapter VII and talk about future steps to extend the applicability of this work for future heterogeneous systems.

CHAPTER II

Input Language

We use the StreamIt [79] language as the input language to the compiler. The emphasis on stream programs is self-evident as recent years have witnessed the proliferation of streaming applications in many areas including digital signal processing, graphics, multimedia, network processing, and encryption. There are several new streaming languages and the area currently commands considerable attention from academia and industry. The stream programming paradigm offers a promising approach for programming multicore architectures. Examples of relatively new streaming languages include StreamIt, Brook [13], CUDA [61], SPUR [89], Cg [55], Baker [16], and Spidle [19].

StreamIt is an architecture-independent programming language for high-performance streaming applications [79]. Programs in StreamIt are represented as graphs where nodes, called *filters* or *actors* encapsulate computation, and edges represent FIFO communication. StreamIt is based on the synchronous dataflow (SDF) [52] model of computation. Each filter consists of a *work* function that repeatedly executes when sufficient data is available on its input FIFO (queue). The work function reads data from its input queue using *pop* operations, and writes data to its output queue using *push* operations. The work function



```
void->void pipeline Minimal {
   add Source();
   add AddSplitter(8, 4);
   add Printer();
}
```

```
void->int filter Source() {
   int i;
   int a[16] = {0, 1, ..., 16};
   init { i = 0; }
   work push 1 {
      push(a[i]);
      i = (i == 15) ? 0 : (i + 1);
   }
}
```

```
int->int splitjoin AddSplitter(int addSize, int pFactor) {
  split roundrobin(pFactor);
  for (int i = 0; i < pFactor; i++)
    add AdderFilter(addSize);

  join roundrobin(1);
}
```

```
int->int filter Adder(int addSize) {
  work pop addSize push 1 {
    int sum = 0;
    for (int i = 0; i < addSize; i++)
      sum+ = pop();

    push(sum);
  }
}
```

```
int->void filter Printer() {
   work pop 1 { println(pop()); }
}
```

Figure 2.1: *A sample StreamIt program is shown on the left. The corresponding stream graph with all the filters instantiated is shown on the right.*

can also inspect input without removing them from the FIFO using a *peek* operation. Peek operations are critical for exposing data parallelism in sliding-window filters (e.g., FIR filters), as they elide the need for internal filter state. StreamIt provides three hierarchical stream primitives for composing filters into larger stream graphs: *pipeline*, *splitjoin*, and *feedback loop*. A pipeline connects streams sequentially. A splitjoin specifies task or data parallel streams that diverge from a common splitter and merge into a common joiner. A feedback loop creates a cycle in the dataflow graph.

A simple StreamIt program and its corresponding stream graph are illustrated in Figure 2.1. This example consists of five streams: `Minimal`, `Source`, `AddSplitter`, `Adder`, and `Printer`. `Minimal` is a top level pipeline with three-stages. The middle stage, `AddSplitter`, consists of a splitter, 4 parallel `Adder` filters, and a joiner. The splitter distributes data to each of its connected filters in a roundrobin fashion. Each `Adder`

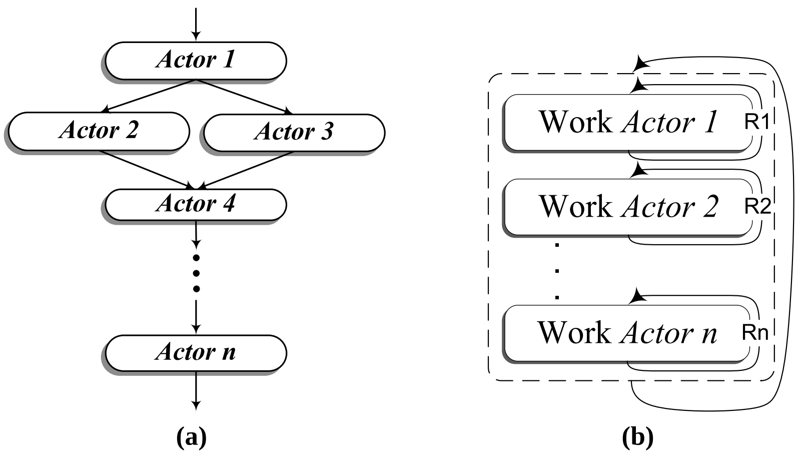


Figure 2.2: *This figure shows an example stream graph and also the intermediate code template for executing steady state schedule.* R_i *is the repetition number for actor* i*.*

receives eight data elements at a time. StreamIt allows stream graphs to be described programmatically, and affords the compiler the ability to fully elaborate the graph at compile time by instantiating and connecting instances of the filters.

Filters in StreamIt are self-contained, and can only access their locally declared variables and fields. Hence, data exchange between filters is accomplished using explicit transfers across inter-filter FIFOs (queues) using the push and pop operations. StreamIt filters may be either stateful or stateless. In Figure 2.1, the `Source` filter is stateful; all the other filters are stateless. `Source` is stateful because the `i` field carries a dependence from one execution of the work function to the next. In addition to the work function, filters may also define an *init* function to initialize local fields.

A crucial consideration in StreamIt programs is to create a steady state schedule which involves rate-matching of the stream graph. Rate-matching guarantees that, in the steady state, the number of data elements that is produced by an actor is equal to the number of data elements its successors will consume. Rate-matching assigns a static repetition number to each actor. In the implementation of a StreamIt schedule, an actor is enclosed by a *for-loop* that iterates as many times as its repetition number. The steady state schedule is a sequence of appearances of these *for-loops* enclosed in an outer-loop whose main job is to repeat the steady schedule. The template code in Figure 2.2b shows the intermediate code for the steady state schedule of the streaming graph shown in Figure 2.2a.

CHAPTER III

Mapping Streams to FPGAs

3.1 Introduction

In the world of embedded systems, there are many devices that offer increasingly powerful computing capabilities. It is predicted that mobile computing devices with embedded processors will ultimately change the industry much as laptops supplanted desktops as the primary commodity processing platform. However, the power and frequency concerns that plague the microprocessor industry effectively mean architects have to find new ways to provide increasing performance since conventional frequency scaling methodologies no longer apply. As a result, there is a significant opportunity to explore alternate architectures that can enable the next evolutionary step in computing.

One significantly promising approach is to provide automatic customization of hardware according to the applications they run. An application-customized architecture can offer extremely high performance with very low power compared to a more general-purpose design. Furthermore, the increasing availability of reconfigurable field-programmable gate arrays (FPGAs) as co-processors and processing ingredients in heterogeneous systems-on-a-chip [1, 2] means emerging architectures can offer enormous flexibility and adaptability in the face of rapidly changing software standards and customer needs.

This part of the thesis describes a methodology and a set of complementary optimizations to efficiently realize stream graphs directly in hardware. Our ultimate goal is to automatically refine a high level stream program into either software or hardware. In the case of the former, a program can run on a conventional processor or a multicore architecture. In the case of the latter, the application is realized as an efficient customized circuit design mapped onto FPGAs.

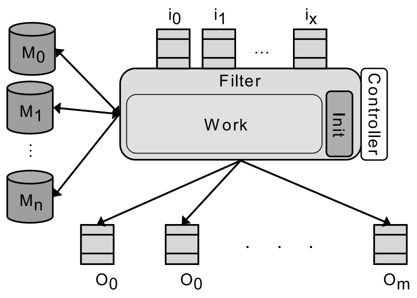
As previously discussed in Chapter II, We adopt a stream programming model where applications can be naturally described as dataflow graphs where nodes embody computation and edges imply communication. Such a streaming model is attractive from a multicore perspective because it makes the abundant parallelism inherent to streaming applications quite explicit. As a result, compilers can more readily derive concurrent implementations from high level applications, with relatively less effort compared to automatic parallelization starting from imperative sequential languages such as C [77, 25, 49]. In the same way, mapping a high-level stream program to hardware (e.g., FPGAs) becomes more practical and productive—compared to using a hardware description language such as Verilog or VHDL, or HDL derivatives of C such as SystemC or Handel-C—if a compiler can readily generate efficient hardware implementations from the programs described in a streaming language.

The idea of mapping high level programs directly into hardware is not a new one. Indeed, there is a lot of work on automatic synthesis of hardware starting from C and its many HDL-oriented derivatives. This work differs from most existing work on the topic of high level synthesis (Section 3.5) by shifting the focus from micro-functional details to macro-functional ones. Specifically, our work does not focus so much on how individual modules are synthesized (i.e., micro-functional), but rather on how modules are composed to assemble an overall design (i.e., macro-functional). As a result, we can synthesize entire applications into a hardware substrate, and not just individual loops and kernels as is the case with a lot of existing work. Thus, our work is complementary to existing work on high level synthesis while offering new opportunities for efficient assembly of streaming applications in hardware.

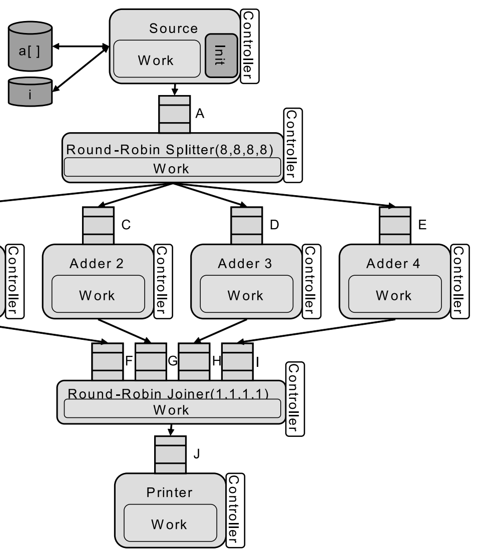
This chapter describes Optimus, our optimizing synthesis framework for streaming applications. Optimus uses a canonical intermediate representation to describe streaming programs. A program is comprised of interconnected *filters*, derived from the dataflow graph representation of the program. Each filter is comprised of *blocks* that contain statements. The blocks are themselves interconnected based on control and dataflow dependences. Our set of optimizations that deal with inter-filter details address macro-functional concerns. Similarly, our micro-functional optimizations address synthesis issues that arise from dataflow dependences between blocks. The Optimus model allows us to leverage decades of classic compiler research studied by others in their work to generate high-quality circuits, while also offering the ability to apply macro-functional optimizations that are specifically targeted for streaming applications. Macro-functional optimizations, which address how filters (modules) are assembled to implement an application tend to be tedious and time-consuming to perform manually, and require expertise in hardware design. An important example of a macro-functional optimization is deciding on how much buffering to allow between a pair of communicating modules: if too little buffering is provided, then throughput decreases as modules stall to send or receive data; whereas too much buffering incurs substantial space overheads. Macro-functional optimizations require careful consideration of area and performance tradeoffs to judiciously maximize application throughput at the lowest costs.

Our results (Section 3.4) using eight streaming benchmarks, including FFT, DCT, DES, sorting, and matrix multiplication, show that we can achieve significant performance advantages compared to an embedded processor for a fraction of the energy. It is not surprising that a custom hardware design is better than a general-purpose processor. We also found that Optimus-generated designs are performance-competitive and incur small area overhead in comparison to some of the benchmarks that we also implemented in Handel-C.

The primary emphasis of Optimus is on the salient macro- and micro-functional optimizations for streaming programs. We use the StreamIt programming language as our input language although other languages that embody the same streaming model are equally applicable. Optimus compiles StreamIt programs to Verilog. We then use standard synthesis tools to generate FPGA designs. Optimus uses its own hardware models to characterize space-time tradeoffs, and performs many optimizations including critical path balancing and memory allocation. It is built on top of the Trimaran compiler [81], and hence it inherits a rich suite of ILP optimizations (for micro-functional efficiency). The compiler also admits profile-guided optimizations to simplify circuit models for streaming applications. Profiling data provides a cheap and practical alternative to otherwise difficult and intractable optimization problems. The core optimizations are described in Section 3.3, and Section 3.2 describes our overall stream-oriented synthesis framework with both macro- and micro-functional emphasis.



(a) Specialized filter template



(b) Hardware structure for the example in Figure 2.1

Figure 3.1: *(a) The specialized template used for synthesizing filters. (b) The complete hardware for the stream graph shown in Figure 2.1.*

3.2 From StreamIt to Hardware

Optimus is a compiler and synthesizer that takes as input a streaming application and generates an efficient FPGA (hardware) implementation. We designed a hardware template capable of representing fairly optimized circuits for streaming applications. The template captures the salient properties of streaming codes, and is malleable enough that it can be used in many different circuit designs we generate. This section details our approach using a simple example illustrated in Figure 2.1.

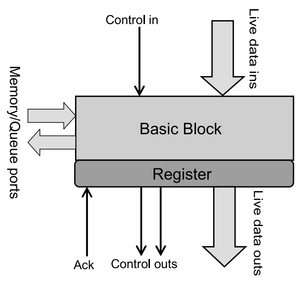
3.2.1 Synthesizing a Stream Graph

Optimus uses a specialized filter template to synthesize the filters that appear in the input stream graph. The template is shown in Figure 3.1a. The template consists of five main components: input queues, output queues, memories, the filter itself, and the controller. Input and output queues are used to send and receive data. The template supports an arbitrary number of input and output queues to implement splitters and joiners. Memory modules are used to store the state for stateful filters. Each filter can be connected to several memory components. All the memory modules are local to each filter. For each memory module, there are dedicated read and write buses between the module and the corresponding filter. The buses are shared between the accessors of the memory in the filter. The hardware block implementing the filter consists of the work module and an optional init module. Both init and work modules will be connected to a memory module in case that module needs initialization. The controller makes sure that the init function gets executed only once before the first invocation of the work function. Depending on the way that the circuit is scheduled, the controller may have other responsibilities to orchestrate the execution.

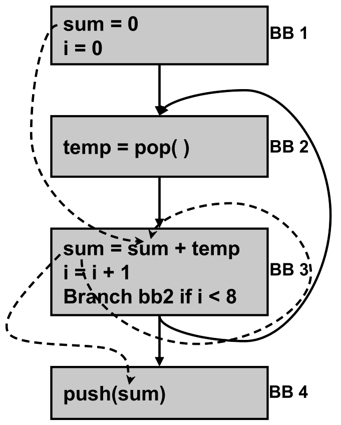
After instantiating the template for all filters in a stream graph, the next step is to connect them. This step is straightforward based on the stream graph and the way data flows through the graph. Whenever Optimus connects the template for two filters together, it merges their input and output queues together. In other words, those two filters will share one FIFO queue for transferring data between them. Figure 3.1b shows the top-level hardware for the stream graph in Figure 2.1. As it is illustrated, the only stateful filter with memory components is the `Source` filter. The `Source` filter also is the only filter with an init component.

3.2.2 Synthesizing Filters

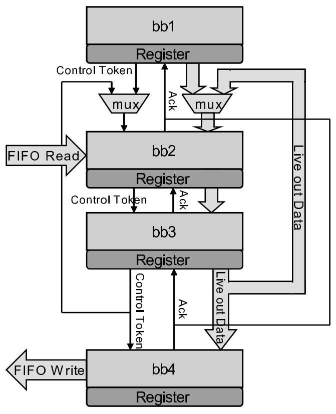
Each filter is organized as a control flow graph (CFG) with an overlayed data flow graph (DFG). Basic blocks (BBs) of instructions are used as the core building units for each filter. The template for the BBs is shown in Figure 3.2a. Each BB module has four sets of input/output signals. The first set includes the control signals. All BBs have one *control input* signal and one or more *control output* signals. A control input signal will activate a BB as long as the signal is active. A control output signal will be connected to the inputs of the other BBs in order to activate them in the right order. Connecting these control signals is done based on the edges in the corresponding CFG. The second set of input/outputs consists of data signals which carry operand values. Optimus uses a DFG for connecting these signals. The third set of input/output signals helps each module to communicate with external resources such as queues, memories, and other types of IP (intellectual property) cores. These signals provide a unified interface in which any IP core can be connected to the hardware. The last set of signals, marked as *Ack* in Figure 3.2a, is meant for flow control. The Ack signals are useful when a BB cannot perform its operations in the associated clock cycle and needs to wait one or more cycles. This mainly happens when a BB accesses an external resource (e.g., memory) and the resource is not ready to respond within the same cycle.



(a) Specialized BB template



(b) Control and (partial) data flow graphs for the `Adder` filter



(c) Hardware structure for the `Adder` filter

Figure 3.2: *(a) The template used for synthesizing basic blocks. (b) Control flow graph and partial data flow graph for the* `Adder` *filter. (c) The complete hardware generated for the* `Adder` *filter.*

Generally, each BB ends with one or more registers to store live-out data and control signals. In the baseline design, it is assumed that all the live-out values are registered to control the wire latency in the final design. Since all the live values are latched at the basic block outputs, one clock cycle is needed to transfer data from one BB to its successors. In other words, the execution of each BB takes at least one cycle.

After the hardware module for each BB is generated, Optimus will connect the modules based on the CFG and DFG for each work or init function. Connecting the control signals is based on the CFG. The control outputs of all BBs are connected to the control inputs of the immediate successor BBs. In case a BB has more than one control input signal, MUXes are used to select the right control input signal. The DFG is used for connecting the data signals, such that the live-out signals of each BB are connected to the live-ins of the immediate successor. MUXes are again used in case a value can reach a BB from two different paths.

We will use the `Adder` filter as an example to clarify the main points. Figure 3.2b shows the CFG and DFG for `Adder`. The solid lines show the control flow and the dashed lines show the data flow for *sum*. This graph has four BBs and there is a backedge from BB 3 to BB 2. Based on the DFG, a data signal is needed for transferring the value for the variable *sum* from BB 1 to BB 3 through BB 2. All the control flow signals in the figure are connected based on the CFG for the `Adder` filter. Since BB 2 is the target of two branches (the fall through from BB 1 and the loop target from BB 3), a MUX is added to its inputs for selecting the appropriate control signal. The execution of the `Adder` filter will take 18 cycles (2 cycles for each of the 8 iterations, and 2 cycles for the rest of BBs).

The only remaining task is to generate hardware to fill each BB module based on the

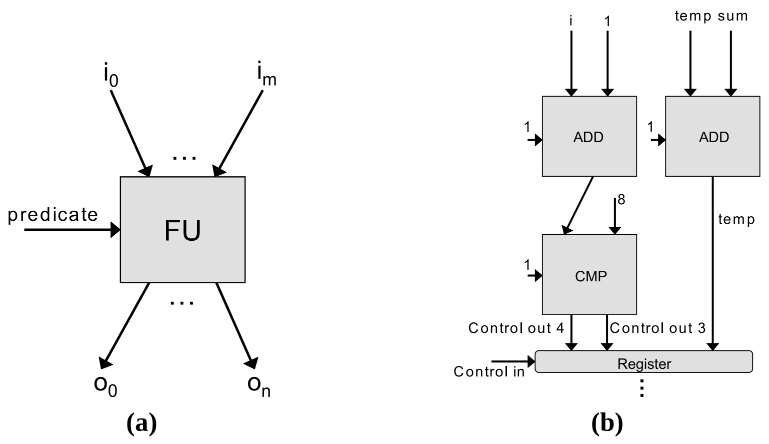


Figure 3.3: *(a) Template for synthesizing operations. (b) Simplified hardware structure for BB 3 in Figure 3.2b.*

operations of that BB. Optimus generates a function unit (FU), similar to Figure 3.3a, for each operation. Each FU can have multiple inputs and outputs and one predicate input. If a BB has a conditional branch operation, Optimus will generate a comparator FU to compute the *control output* signals. The data flow graph in each BB determines how the FUs should be connected to each other. At the end of each BB, its *control input* signal is used to enable the register module. Figure 3.3b illustrates the FU and the necessary connectivities for BB 3 of the `Adder` filter. This figure does not show all the details of computing control signals and setting the Ack signal.

3.2.3 Hardware Orchestration

The final issue is the orchestration of execution for the entire streaming circuit. We focus on two ways of scheduling the filter executions: *Static* and *Greedy*. In a static schedule, the compiler dictates the number of executions of each filter, such that it consumes all of its input data and produces sufficient data for its consumers. In this model, the compiler guarantees that a filter will have a sufficient number of input data available. Hence the execution of the filter work function will not block on reads (i.e., pops). Similarly, the compiler also asserts that the output queue from a filter is sufficiently empty so that all the writes (i.e., pushes) also succeed without blocking the filter. In this type of scheduling, double buffering is used between pairs of filters to provide communication-computation concurrency. This allows the producer and consumer to run independent of each other. The size of each individual queue is typically set to the least common multiple of the pop rate and push rate of the consumer and procedure filters. We refer to this form of scheduling and FIFO sizing as “rate-matched”.

A greedy schedule takes a different approach and does not try to statically rate-match filters. In this approach, filters execute eagerly, and block when they attempt to read from an empty queue, or write to a full queue. Since all queue accesses are blocking in this approach, the size of the queue throttles the execution of the stream graph. This allows for a tradeoff between the size of the queue and the overall circuit throughput. Smaller queues take up less area, but may not be optimal. In our benchmarks, we observed that it is common that a queue size of one element is sufficient for correct execution that is also as efficient as a rate-matched static scheduled. The queue sizing is further discussed in the following section.

Optimus is capable of generating the necessary hardware for both schedulers. This choice has implications on the rest of the circuit in terms of queue sizes, power consumption, execution time, and allowed hardware sharing. In this work, the greedy scheduler is used for all designs. The comparison between the two schedulers is left for future work.

3.3 Stream Optimizations

Streaming languages allow programmers to focus on designing their applications. Specifically, programmers describe their computation programmatically and algorithmically, and do not need to commit to specific implementation details related to scheduling, buffering, synchronization, or the underlying data transport mechanisms in their target platforms. This programming practice leads to code that is easy to maintain and port, but places a burden on the compiler to derive high-performing implementations.

Optimus applies many of the classical optimizations used in previous works, and introduces a set of new macro-functional optimizations that specifically target streaming programs. Our compiler focus is on improving communication latency and reducing memory storage requirements (i.e., area). Communication latency can be optimized by sending larger chunks of data between filters. Storage can be optimized by intelligently sizing queues between filters, and allocating output registers to increase spatial reuse. It is not uncommon in today's synthesis frameworks to apply many of these optimizations manually, either directly in the source code or after the circuit is generated. This process can be time-consuming, error-prone, and complex for large benchmarks. It also defeats the purpose of using elegant and practical streaming languages that are attractive because they promote productive and portable programming.

3.3.1 Queue Allocation

The queues that connect hardware filters are implemented using the SRAM structures on the FPGA. FPGAs have limited SRAM capacity, ranging from 4 KB on the low-end

FPGAs to 128 KB on the high-end ones. The SRAM is also used to implement the local arrays and other data structures used by the filters. Thus, for large stream graphs, the SRAM quickly becomes the bottleneck resource. The scheduling strategy used to orchestrate the execution of the filters can significantly impact the storage requirements. Optimus judiciously calculates the size of each queue to allocate between filters in order to better utilize the SRAM and maintain the high throughput achieved by a rate-matched static schedule.

The idea behind our approach is to recognize that a slot in the queue may be reused if the value that previously occupied the slot is already consumed. Thus, we can reduce the total storage requirement for the inter-filter FIFOs if we can determine the maximum number of overlapping lifetimes for the values exchanged between filters.

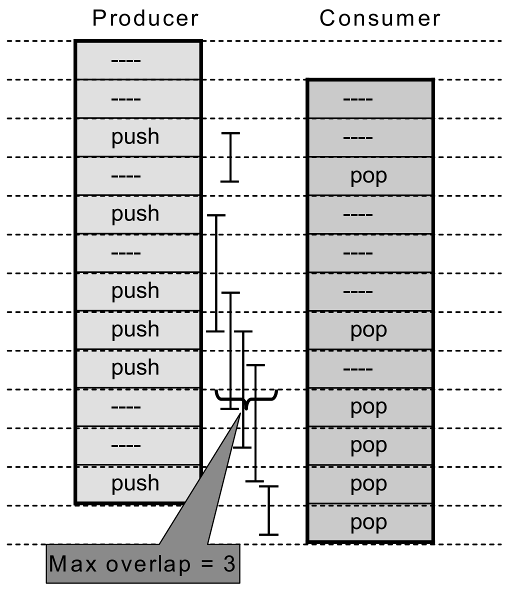
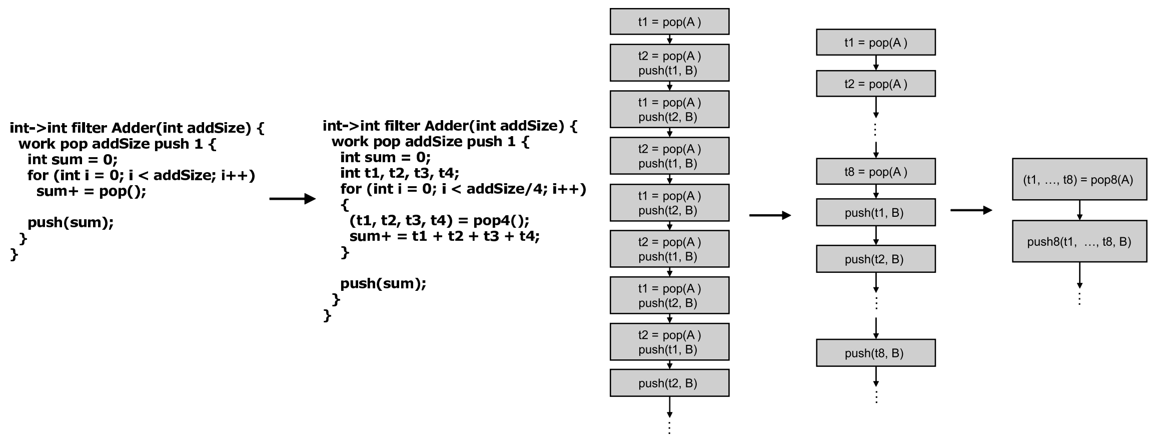


Figure 3.4: *Overlapped producer-consumer schedules showing maximum number of overlapping lifetimes.*

Figure 3.4 shows the cycle-by-cycle schedule of a pair of communicating filters. Only the push and pop operations are shown. The schedule shows all the cycles in the steady state executions of the producer-consumer pair. Suppose the producer pushed N items per execution of its work function, and the consumer popped M items from the queue every time its work function executes. For the filters to be rate-matched, the producer must run its work function $\frac{LCM(M,N)}{M}$ times, and the consumer $\frac{LCM(M,N)}{N}$ times. We determine the maximum number of overlapping lifetimes by simulating the rate-matched schedule. We use double-buffering during simulation to provide communication-computation concurrency. The simulation needs to only cover one steady-state execution of the filters. In the case a filter peeks at more data than it pops, an initialization schedule is run to prime all the FIFOs.

A causally correct schedule is obtained by shifting the producer schedule to occur at time 0, and shifting the consumer schedule down such that all pops appear at least one cycle after their corresponding pushes. Figure 3.4 shows an example schedule. Such a schedule reveals the lifetime of every entry in the queue between the producer and the consumer. The lifetime extends from the cycle at which an entry is pushed and the cycle at which the entry is popped. The maximum number of queue entries whose lifetimes overlap can be easily calculated from the schedule. In Figure 3.4, the maximum number of overlapping lifetimes is 3. Setting the queue size to a value less than this maximum will stall the filters because one of the pushes at the producer cannot succeed as it would appear before the pop of the previous queue entry. Conversely, setting the queue size to a value more than this maximum would not improve the schedule. Thus, the minimum queue size for a producer-consumer pair that retains the throughput of the static schedule is obtained by calculating the maximum number of overlapping lifetimes of a rate-matched schedule.



```
int->int filter Adder(int addSize) {
  work pop addSize push 1 {
    int sum = 0;
    for (int i = 0; i < addSize; i++)
      sum+ = pop();

    push(sum);
  }
}
```

```
int->int filter Adder(int addSize) {
  work pop addSize push 1 {
    int sum = 0;
    int t1, t2, t3, t4;
    for (int i = 0; i < addSize/4; i++)
    {
      (t1, t2, t3, t4) = pop4();
      sum+ = t1 + t2 + t3 + t4;
    }

    push(sum);
  }
}
```

(a) `Adder` **(b)** `Splitter`

Figure 3.5: *An example of access fusion using the stream program in Figure 2.1.*

3.3.2 Queue Access Fusion

A critical factor in streaming applications is sustained throughput. One of the key issues that can have negative effect on the throughput of a streaming circuit is communication latency between different filters. This issue arises from the fact that each queue or memory access, regardless of data width, takes at least one cycle. The one cycle access time would have a direct affect on the latency of the longest path in filters. It can also limit the filter-level parallelism in splitters and joiners. To overcome these bottlenecks, we consider bundling similar queue accesses together to create a single wide access using *queue access fusion*. This is conceptually similar to creating SIMD loads and stores. Of course, to support fused queue accesses, the basic queue structure requires modifications.

Code motion and loop unrolling are applied to find opportunities for fusing queue accesses and shortening the longest paths in a filter. Automatic SIMDization techniques use a similar approach with one difference: the vector length is known a priori, whereas we can realize variable vector lengths between producer-consumer filter pairs. Loop unrolling is applied to loops with queue or memory operations to expose the operations to the code motion phase. Optimus needs to consider area constraints while it is performing the unrolling because unrolling may result in area expansion and cause the design to overflow the target FPGA. The next step, is to cluster memory and queue operations via aggressive code motion. The end result is a several clusters of memory and queue operations with no other intervening operations. Each cluster of operations is assigned a vector length according to the number of operations in the cluster. Subsequently, the compiler determines a single vector length for the filter by calculating the greatest common divisor of each cluster's vector length. For example, if the vector lengths of the clusters are 8, 12, and 16, then the filter's vector length is 4.

Figure 3.5 shows this optimization applied to a filter and a splitter from Figure 2.1. For the `Adder` filter in Figure 3.5a, the loop is unrolled 4 times and a vector length of 4 is chosen for fusion. The loop is not fully unrolled because of area constraints. The unoptimized `Adder` filter will take 18 cycles to finish (assuming the value of `AddSize` is 8), but the optimized one will take only 6 cycles. Figure 3.5b illustrates the effectiveness of the fusion optimization for the splitter filter. The unoptimized splitter needs 9 cycles to read 8 data values from its input queue and push them to the input queue of `Adder1`. During these 9 cycles, the next filter in the splitjoin (`Adder2`) would be idle while it awaits its input data to arrive. In this case, the access fusion optimization will reduce the filter's idle time to 2 cycles. The optimization in general reduces the critical path of computation and can reduce execution time. If the optimization is successful in finding large clusters of accesses and fusing them, it will also significantly reduce the total area of the design. If the optimization is not successful, the loop unrolling would result in area expansion. However, an intelligent compiler would reverse the unrolling when it is not profitable.

One of the restrictions imposed by the our generated hardware is that the vector length for all accesses from a filter to a specific queue has to be the same, although vector lengths to the same FIFO from different filters may differ. This is realized by incrementing and decrementing read and write pointers using different constant offsets. For example, if the read vector length is 1 and the write vector length is 8, the queue can be viewed as an 8x8 matrix with the write pointer pointing to the rows and the read pointer pointing to individual elements of the matrix. Figure 3.6 illustrates the possible configurations.

3.3.3 Flip-flop Elimination

As it was discussed in Section 3.2, all live-out data signals, including pass-through live signals, are registered at the end of each basic block to bound wire delays. The output of memory and queue operations cannot be registered in the block that issues those operations because memory (and queues) needs one cycle to respond. Therefore, the results of those operations are registered in the immediate successors of the issuing basic block, as well as along all blocks that transmit the values along to their destinations. The CFG in Figure 3.7a illustrates the registers added for various operands as rounded-edge rectangles attached to the basic blocks. Note that live operands are saved at the end of each basic block regardless of whether they are passing through or generated in that block. This register assignment ensures that the critical path in a CFG is not greater than the maximum of delays through the basic blocks.

Many of these flip-flops are unnecessary and can be removed without affecting the clock speed. In order to keep the circuit functional, a subset of registers must be main-

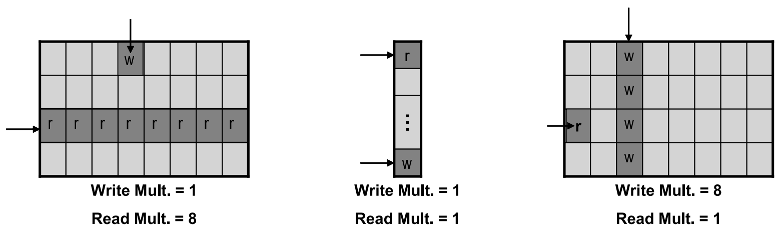
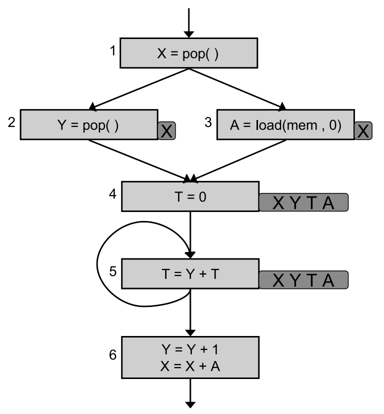


Figure 3.6: *Various configuration of queues used by queue access fusion optimization.*

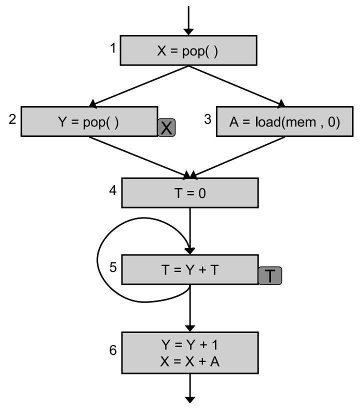
tained. There are two main situations where flip-flops cannot be removed. First, if an operand is both live-in and live-out along a backedge, it has to be registered before or after the backedge to prevent formation of a combinational loop. The second case is more complex. If an operand is the result of a queue or memory read, it does not have to be registered because the hardware for the queue and memory hold its output as long as no other operation has changed its read status. For a read operation from a queue, a status change occurs when another pop is issued. In a memory structure, status changes when a store writes to the same address as a read. When the compiler can determine that no intervening pops or read/write conflicts occurs, then it can elide the corresponding registers.

Figure 3.7 shows a sample CFG and all the data registers before and after the flip-flop optimization. Based on the rules for flip-flop elimination and ignoring clock cycle constraints, all the registers can be removed except X-register in BB 2 and the T-register in BB 5. The X-Register cannot be removed because there is an interleaving pop operation in BB 2 that can change the status of the input queue. Note that if the control flows to the right instead of left after BB 1, then no register is needed because there are no pop operation along that path. The register for T also cannot be removed because T is both live-in and live-out along the backedge going from BB 5 to itself.

An issue with flip-flop elimination is the possibility of increasing the critical path



(a) Sample control flow graph before flip-flop elimination



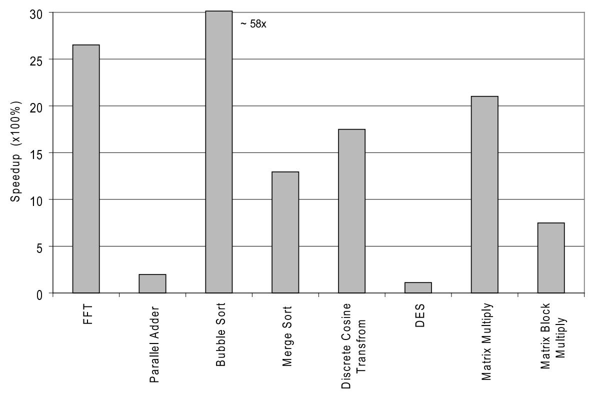
(b) After flip-flop elimination

Figure 3.7: *An example of flip-flop elimination.*

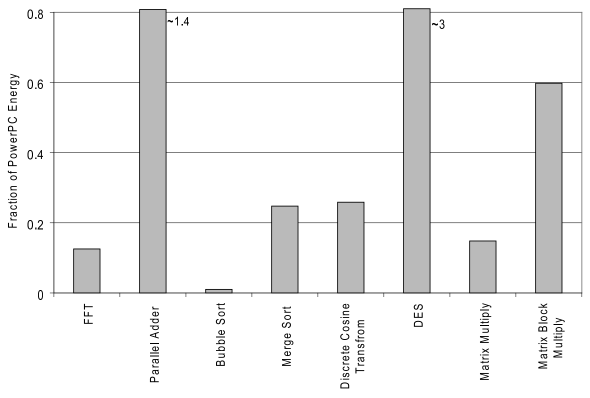
length. In general, Optimus tries to balance the length of the combinational paths by splitting the large basic blocks and adding registers to the end of each BB. Optimus has an internal model of the target FPGAs to assess the latency of different combinational operations. If removing any of the registers in the flip-flop elimination optimization lengthens the critical path, then that register is left in place.

3.4 Experiments

We compiled and simulated various applications from different domains. Our target platform is a Xilinx Virtex-4 (XC4VLX200) FPGA [85]. ISE Foundation was used for synthesizing the HDL generated by Optimus. Xilinx Xpower is used to measure the energy and power consumption of our circuits. For comparison, we used a 300 mW 300 MHZ embedded PowerPC 405 processor. We compare our FPGA results to the benchmarks compiled and executed on the PowerPC. We use the StreamIt compiler, and the same StreamIt source code for the benchmarks, to generate binaries that run on the PowerPC processor. Our benchmarks are FFT (fast Fourier transform), parallel adder, bubble sort and merge sort,



(a) Performance Comparison

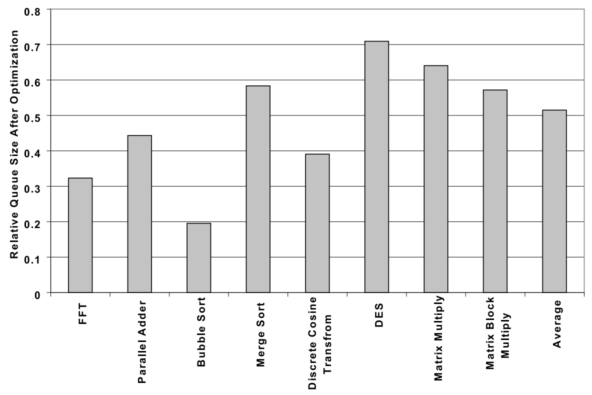


(b) Energy Consumption Comparison

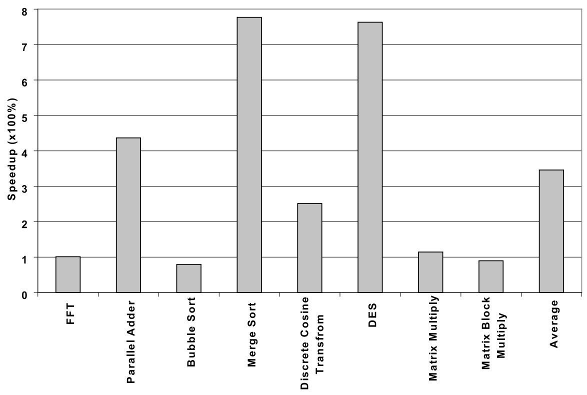
Figure 3.8: *Figure 3.8a illustrates the speedup comparison between the hardware designs and a 300 mW PowerPC 405 running at 300 MHZ. Figure 3.8b shows the energy consumption of the FPGA as a fraction of PowerPC energy use for various benchmarks.*

integer inverse DCT (discrete cosine transform), DES (data encryption standard), matrix multiply and its blocked variant. In the case of DES, we used a reference C implementation of the benchmark instead of the StreamIt version for the PowerPC measurements. This is because DES performs a lot of bit-level operations, and tuned implementation can cleverly carry out the operations in parallel using word-wide masks. In the case of the FPGA, we compile the StreamIt version of DES down to HDL.

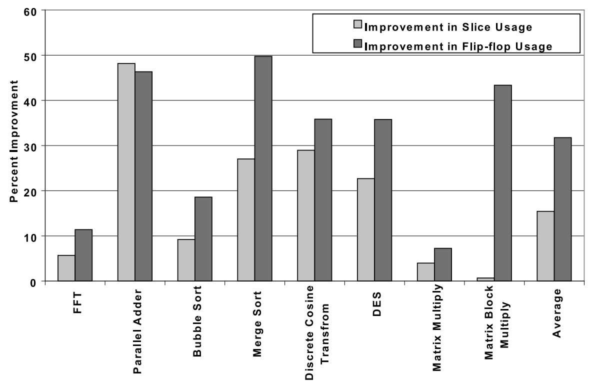
Performance and Energy Consumption: Figure 3.8a shows the performance of streaming hardware compared with PowerPC for various benchmarks. In this experiment, none of the streaming-specific optimizations are used. Speedup varies from 1.1x to 58x for different benchmarks. Bubble sort achieves the highest speedup because it heavily exploits pipeline-level and instruction-level parallelism. Parallel adder has the lowest speedup over the baseline because the communication to computation ratio is high in this benchmark. Figure 3.8b illustrates the energy consumption of the circuits generated by Optimus as a fraction of the PowerPC energy usage. On average, these benchmarks consume 0.7x of the



(a) Queue Allocation Optimization



(b) Queue Access Fusion Optimization



(c) Flip-flop Elimination

Figure 3.9: *Performance improvements and area savings due to different optimizations performed by Optimus.*

PowerPC energy. The only benchmarks which use more energy on the FPGA are parallel adder and DES. This again happens due to large communication to computation ratio in case of the parallel adder. In DES, the higher energy consumption is due to the inability of Optimus to efficiently take advantage of the bit-level parallelism in the stream graph. Considering the fact that the baseline processor is a 300mW core, these results show that the hardware generated by the Optimus system is suitable for low-power embedded systems in terms of both performance and energy consumption.

Queue Allocation: In the designs generated by Optimus, one of the main components that uses the on-chip memory is the queue structure. The queue allocation optimization tries to efficiently reduce the sizes of the queues without affecting the performance. The StreamIt compiler generally uses rate matching between the filters to calculate queue sizes. We used the rate matched queue sizes as the baseline and show the savings due to the queue allocation algorithm in Figure 3.9a. As shown in the figure, this optimization reduces the queue sizes by an average of nearly 50%. Additionally, after reducing the queue sizes to the new values, no performance loss was observed in any of these benchmarks. These results demonstrate that the queue allocation optimization used by Optimus is quiet effective in saving the on-chip memory resources.

Queue Access Fusion: As discussed in Section 3.3.2, the goal of queue access fusion is to increase the throughput of streaming circuits by fusing multiple queue operations into a single (wider) operation. Figure 3.9b illustrates the effect of this optimization on various benchmarks in terms of performance. We limit the maximum vector length to 8. This means that the maximum speedup achievable is 8x. As shown in the figure, the average speedup is 3.2x, and 7.2x in the best case. In some benchmarks, no speedup is achieved because there was not any opportunity to fuse accesses in the slowest filters. The slowdowns are typically due to the fact that the wider queues are marginally slower than normal queues. In order to understand the area and performance tradeoff between different queue configurations, we synthesized three queues with the same size but different read/write widths. As the results in Table 3.1 show, the wider queues are slightly larger than their narrower counterparts.

Flip-flop Elimination: The goal of this optimization is to identify and eliminate redundant registers such that the circuit still functions properly and the critical path length does not change. The results of this optimization are shown in Figure 3.9c. Flip-flop elimination

Queue Configuration	Total number of bits	Number of Slices	Clock (MHZ)
(read width = 128, write width = 16)	4096	70	$>$300
(read width = 16, write width = 16)	4096	56	$>$300
(read width = 16, write width = 128)	4096	95	$>$300

Table 3.1: *Area and delay for different queue configurations*

reduces flip-flop utilization by 30% and slice utilization by 16%. As shown in the figure, the improvement in flip-flop use is always greater than slice utilization. This means that there are many slices used only for latching purposes and not for logic computation. The area savings due to this optimization vary based on the number of pops and loads and their arrangement in each benchmark.

Comparison to Handel-C: We compared our generated circuits to those generated using Handel-C and its compilation tool chain. Handel-C is a variant of the C programming language. It is aimed toward synthesizing hardware from C code. We implemented DES and DCT in Handel-C and generated their hardware designs. The Handel-C implementations preserved the overall streaming structure of the benchmarks. Our area and performance comparisons show that the Optimus-generated circuits are an average of 5% faster and 66% larger. Using our stream-specific optimization, we can further improve the performance of the Optimus-generated circuits so that they are 12x faster, although the designs are also 90% larger than the Handel-C designs.

There are several important factors that make the Handel-C designs inefficient in terms of performance. First, Handel-C is not able to automatically perform the same kind of macro-level optimizations that Optimus carries out. Second, Handel-C does not try to balance the critical paths between flip-flops to achieve higher frequency designs. The lack of these optimizations and transformations is the main reason the Handel-C designs lag in performance compared to the Optimus-generated ones. The optimizations can be done manually in the Handel-C code, but that requires more work for the programmer, and it would obfuscate the streaming nature of the code.

In terms of area comparisons, the designs in Handel-C are more area-efficient for two main reasons. First, Handel-C tries to utilize resources (IPs) that are unique to various families of FPGAs. Designs generated in this way are usually more area-efficient. Second, Handel-C performs some low-level netlist optimizations that improve the area by a large factor. We believe netlist-level optimizations should be implemented in the low-level hardware synthesis tool and not in a high-level compiler. Therefore, Optimus does not implement any of the low-level optimizations that Handel-C performs to improve the area efficiency.

3.5 Related Work

C is closely linked to the Von Neumann processor model, in which variables correspond to memory locations and function invocations reside on stacks. C lets users manipulate pointers to memory and to functions, which does not make sense in an FPGA circuit model. Thus, any attempt to compile C to FPGA configurations would encounter problems that derive purely from the C language, not from the the application itself. Several projects have tried to address the inadequacies of C with different techniques.

As a result of an extensive amount of research in the area of high-level synthesis, researchers have introduced several compiler systems and abstraction languages [41, 57, 74, 56, 32, 24, 12, 29, 28] each of which has some unique capabilities. ROCCC [28] is a C to hardware compilation project whose objective is the FPGA-based acceleration of frequently executed loop nests. This compiler performs extensive compile-time transformations to maximize various forms of parallelism and minimize the number of off-FPGA memory accesses. Circuits generated by ROCCC can be used by Optimus as IP blocks to accelerate the execution of loop nests. Another C to hardware compiler is SPARK [29], which takes a subset of C as input and outputs synthesizable VHDL. Its optimizations include code motion, variable renaming, FSM state minimization, etc. Streams-C [24] relies on a CSP model for communication between processes and can meet relatively high-density control requirements. Researchers in academia and industry have also designed various high-level abstraction languages such as SA-C [59], Handel-C [14], SystemC [76], etc., to make designing hardware systems easier for average software developers. SA-C helps compilers exploit data reuse because of its special constructs (e.g., windows) and it functional nature. Handel-C is a low level hardware/software construction language with C syntax that supports behavioral descriptions and uses a CSP-style communication model.

Although all these systems and abstraction languages have proved useful in various domains, they have different shortcomings. GARP, Streams-C, and SPARK do not support accesses to two-dimensional arrays, so image processing applications must be mapped manually. ROCCC accepts only perfectly nested and constant bound loops operating on arrays with affine index expressions. Moreover, all arrays are assumed to be located in the memory and no local data is allowed. The previous systems and languages do not support the stream-oriented optimizations that we discuss in this work. They also do not provide some of the constructs that are essential for stream programming such as peeking.

3.6 Summary

Streaming applications are important to embedded systems developers. Improving the performance of these applications in an embedded setting is typically accomplished via special purpose processors and ASICs that are inflexible and invariably expensive to design. An alternate approach is to use configurable hardware fabrics such as FPGAs that provide a performance- and power-competitive platform for their cost. In addition, FPGAs are increasing available as components in heterogeneous systems, and their versatility makes them attractive platforms in a domain where software and consumer requirements change rapidly. Unfortunately, the complexity of programming FPGAs has limited their benefits as only system engineers with hardware design expertise are able to effectively map software down to hardware circuits.

The goal of our work is to enable the efficient realization of streaming programs directly in hardware, when appropriate. Our Optimus compilation methodology allows for streaming programs expressed in a high-level streaming language such as StreamIt to be automatically refined to hardware and realized as circuits in FPGAs. The Optimus compiler uses a hierarchical compilation strategy that separates concerns between macro- and micro-functional requirements. Macro-functional optimization are geared to efficiently assembly filter module into larger applications. These optimizations affect space (area) and time (throughput) characteristics of the application circuits. Our goal in this regard is to provide the highest performance for the lowest area cost. Comparing our generated designs to an industry-strength compiler shows that we are performance and area competitive although we believe there is much more to be gained in our framework. Our results are largely enabled by stream-specific considerations and optimizations. Micro-functional optimizations are designed to improve the efficiency of the filter modules themselves. Our stream-aware optimization improve performance an average of 255% and reduce the area requirements by 16% compared to our baseline results.

CHAPTER IV

SIMDization of Stream Graphs

4.1 Introduction

Support for parallelism in hardware has greatly evolved in the past decade as a response to the ever-increasing demand for higher performance and better power efficiency in different application domains. Various companies have introduced vastly different solutions to bridge the performance and power gap that many applications are facing. These solutions include shared-memory multicore systems (Intel Core i7 [43]), distributed-memory multicore processors (IBM Cell [40]), tiled architectures (Tilera [80]) and in some cases a combination of these (Intel Larrabee [72]). These architectures not only achieve higher performance and efficiency by combining multiple cores into one die, but they are also equipped with one or more single-instruction-multiple-data (SIMD) engines to enable more efficient data-level parallelism support for several important application domains such as multimedia, graphics, and encryption. SIMD engines are not suitable for all applications, but if an application can be tailored to efficiently exploit them, the performance and power benefits can often be superior to the gains from other architecture solutions. Therefore, SIMD engines like Altivec [73], Neon [6], SSE4 [42] are now an essential part of most architectures on the market. With SIMD width expanding in future architectures, such as Intel's Larrabee, under-utilization of the SIMD units would translate into a loss in performance and also power consumption.

Mapping abundant parallelism of streaming applications onto multi-core provides reasonable speedup for but can also experience slowdown due to inter-core communication overhead and high memory/cache traffic. Utilizing SIMD engines is preferred, even for applications where multi-core speedup is close to the theoretical maximum, because SIMD engines can improve performance without increasing communication overhead and memory/cache traffic. Exploiting SIMD engines, in some cases, can achieve greater performance than multi-core while using less area and power.

Extending the retargetability of streaming languages for multi-core systems by adding effective SIMD support to their compilers is desirable because of the variation in characteristics of SIMD accelerators between different standards, such as number of lanes, memory interface, and scalar/vector transfers. Implementing and porting applications between different architectures can be difficult and error-prone. Therefore, efficiently vectorizing stream programs is essential to expand their applicability as a universal programming paradigm for current and future single/multicore architectures with various wide or narrow SIMD units.

To exploit SIMD engines, current streaming compilers translate the streaming languages down to an intermediate language, such C++ or Java, and then apply vectorization techniques to generate SIMD-enabled code. The most popular techniques are hand-optimizing the code and traditional auto-SIMDization [3, 4, 64, 5, 51]. Both of these solutions have proven difficult to apply in real world scenarios. Hand-optimizing the binary or sequential code using architecture-specific instructions or intrinsic functions is a time-consuming and error-prone task which results in an inflexible and unportable binary. Auto-vectorization is, at this stage, still impractical and far from being able to universally utilize the various kinds of available SIMD facilities. Also, performing SIMDization on streaming applications after intermediate-level code generation may result in an inefficient schedule and mapping of the stream graph since the schedule is already fixed and information that is available in the high-level stream graph is lost. Extracting this information from the generated code is predicated on performing complex compiler analysis and transformations which are impossible in some cases. In summary, *lack of global knowledge about the program*, *inability to adjust the schedule*, and also *loss of data flow information* are the main reasons behind inefficiency of traditional auto-vectorization techniques in dealing with streaming applications.

In this work, we introduce *MacroSS*; a streaming compiler for the StreamIt language that is capable of performing macro-SIMDization on stream graphs. Macro-SIMDization uses high-level information such as the valid set of schedules and communication patterns between actors to transform the graph structure, vectorize actors of a streaming program, and generate intermediate code (C++ in this work). Then, it uses the host compiler to compile the generated intermediate code to binary for a specific target processor. The information that is used by MacroSS is deduced from the high-level program structure and is not available to low-level traditional compilers that are used to compile the intermediate code. As a result, MacroSS has a broader understanding of the program structure and macro-level characteristics of the streaming application that allows the compiler to utilize

SIMD engines more efficiently.

MacroSS is capable of performing single-actor, vertical, and horizontal SIMDization of actors. Single-actor SIMDization targets each SIMDizable actor separately and transforms consecutive sequential executions of a SIMDizable actor to data-parallel executions on the SIMD engine. Vertical SIMDization fuses a pipeline of vectorizable actors to build a larger vectorizable actor and reduces the scalar-to-vector (packing)/vector-to-scalar (unpacking) overhead that exists between actors. Our experiments show that vertical SIMDization is applicable in many cases and can significantly improve performance by eliminating the need for translating back and forth between scalar and vector. Finally, horizontal SIMDization takes a set of isomorphic task parallel actors and replaces them with one or more data parallel actors. The choice of which vectorization technique to apply to a stream graph is based on the internal target-specific cost model and the structure of the graph. After SIMDization, MacroSS is able to generate architecture-specific intermediate code with SIMD intrinsics. This intermediate code uses vector types and intrinsics specific to the target architecture and can be compiled using the host compiler.

Packing of scalar values to a vector or unpacking a vector to scalar values typically takes between a couple of cycles to tens of cycles depending on the architecture. Since communicating data between vectorized and scalar actors or vice versa needs several packing/unpacking operations, MacroSS is equipped with two techniques to optimize this costly communication overhead. The first technique tries to replace the packing/unpacking operations with permutation instructions in actors that, during each execution, read or write 2^n elements. In the second technique, we introduce a low-overhead dynamic shuffler called the streaming address generation unit (SAGU). This unit eliminates the need to perform complicated address translations, data alignment, and packing/unpacking of data as data crosses vector-scalar boundaries of the graph.

To summarize, in this part, we make the following contributions:

- Introduction of macro-level SIMDization techniques for streaming languages: single actor, vertical and horizontal SIMDization. Based on these techniques, MacroSS compiler for the StreamIt language is implemented.

- Hardware and permutation-based tape optimizations for reducing the overhead of scalar-to-vector and vector-to-scalar data conversions.

- Evaluation of MacroSS on various streaming workloads from the StreamIt benchmark suite [79] on the Intel Core i7.

Macro-SIMDization and the related optimizations in MacroSS are explained in Section 4.2. Section 4.3 includes a brief discussion about the differences between traditional auto-vectorization and macro-SIMDization. Experiments are shown in Section 4.4. Finally, in Section 4.5, we discuss related works.

4.2 Macro-SIMDization

The SIMDization path in MacroSS consists of several steps to make the streaming graph more amenable to vectorization, tune the steady state schedule, vectorize actors, and perform target-specific code generation.

MacroSS is equipped with three main techniques: *Single-Actor, Vertical,* and *Horizontal SIMDization*. Single actor SIMDization targets each stateless actor separately. The

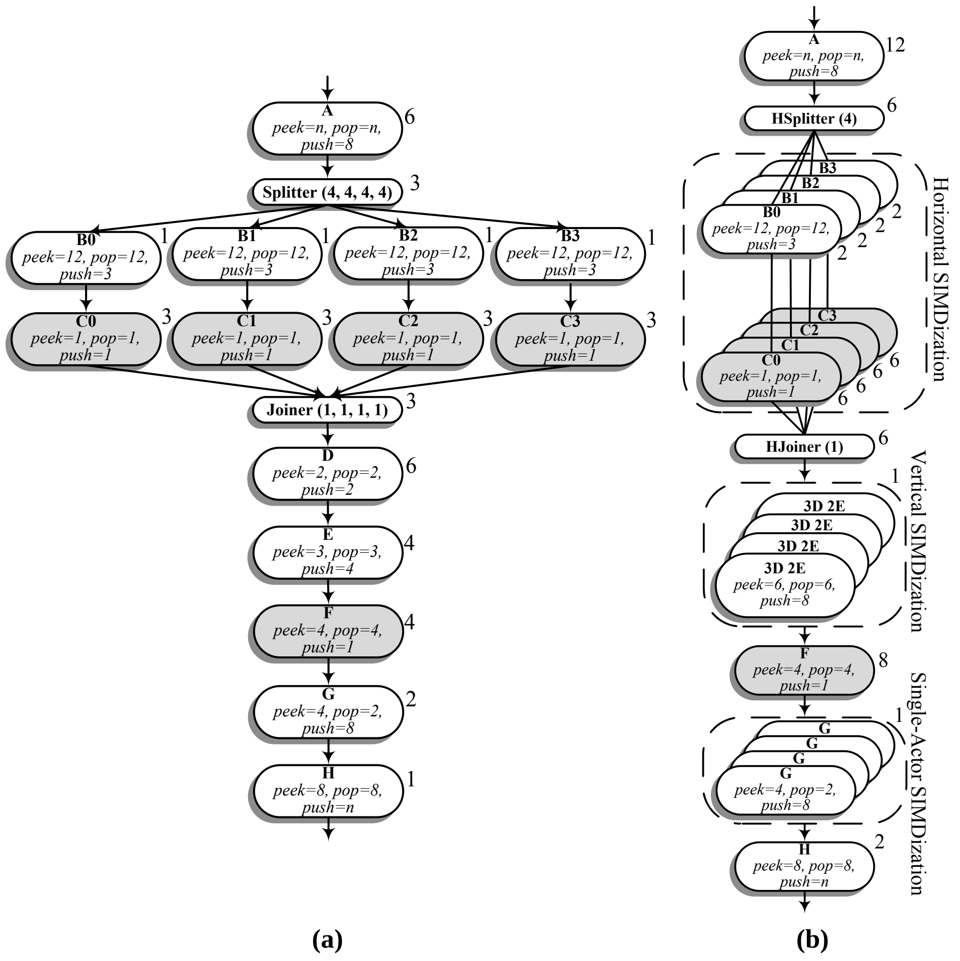


Figure 4.1: *Part (a) of this figure shows the stream graph used as a running example. Part (b) shows the same stream graph after MacroSS has SIMDized it.*

goal is to convert multiple (equal to SIMD-Width) consecutive executions of a SIMDizable actor into one data-parallel execution on the target SIMD engine. Single-actor SIMDization leaves the input and output tapes of a vectorized actor as scalar and does not convert the tape accesses to vector since complicated shuffle operations must be introduced in the code in case vector tape accesses are used. The scalar tapes introduce packing/unpacking overheads in each SIMDized actor. Vertical SIMDization, which is a more optimized way of performing single-actor SIMDization on a pipeline of vectorizable actors, reduces this overhead. It enables MacroSS to implement vector communication between the actors of a

SIMDizable pipeline. Both single-actor and vertical SIMDization try to convert sequential execution of a single actor or a pipeline of actors to data-parallel execution. The third technique, horizontal SIMDization, converts task parallelism into data parallelism for a group of isomorphic actors (stateful or stateless) in a stream graph. Horizontal SIMDization is mainly beneficial in cases where a group of several isomorphic actors are placed between a splitter and joiner and it is not possible to fuse these actors into one coarse actor to perform vertical SIMDization. MacroSS finds the parts of a graph that are suitable for this kind of SIMDization and converts the eligible task-parallel actors into one or more SIMD actors.

The stream graph illustrated in Figure 4.1a is used as a running example to explain different actions that the compiler takes to perform macro-SIMDization. This graph shows the structure of a streaming application with 10 unique actors. Each box shows one actor in the program. Each edge in this graph indicates a tape implemented using FIFO queues. The text written inside each box shows how each actor interacts with its input and output tapes. Each shaded box represents a stateful actor. On the right side of each node, the repetition number of that node in the steady state is shown. Even though MacroSS is able to target processors equipped with SIMD engines with any SIMD width, for the sake of presentation, the target hardware platform to which MacroSS compiles is set to a core with SIMD width of four 32-bit data types, and main memory line width of 128-bit. Figure 4.1b shows how MacroSS vectorizes the streaming graph. The *split-join* structure is horizontally vectorized. The task-parallel actors between the splitter and joiner are converted to SIMD actors and the splitter and joiner are replaced with horizontal versions. Actors D and E are vertically fused and SIMDized. Single-actor SIMDization is applied to actor G.

The details of how MacroSS performs SIMDization on a streaming graph are explained in the following three subsections. Next, in Section 4.2.4, the way MacroSS deals with SIMDization of tapes in the presence of architectural support is explained. Finally, Section 4.2.5 explains the overall structure of the macro-SIMDization technique in MacroSS.

4.2.1 Single-Actor SIMDization

Let SW denote the SIMD width of the target machine. The goal of single-actor SIMDization is to run SW consecutive executions of an actor in data-parallel fashion using the target SIMD engine. As mentioned before, actors in a StreamIt program execute based on a steady state schedule in which each actor is enclosed by a *for-loop* that iterates as many times as its repetition number (see Figure 2.2b). Conceptually, single-actor SIMDization is similar to vectorizing the actor's enclosing *for-loop* whose trip count is the repetition number of the actor. Therefore, MacroSS adjusts the repetition numbers of all actors to make them multiples of SW before single-actor SIMDization.

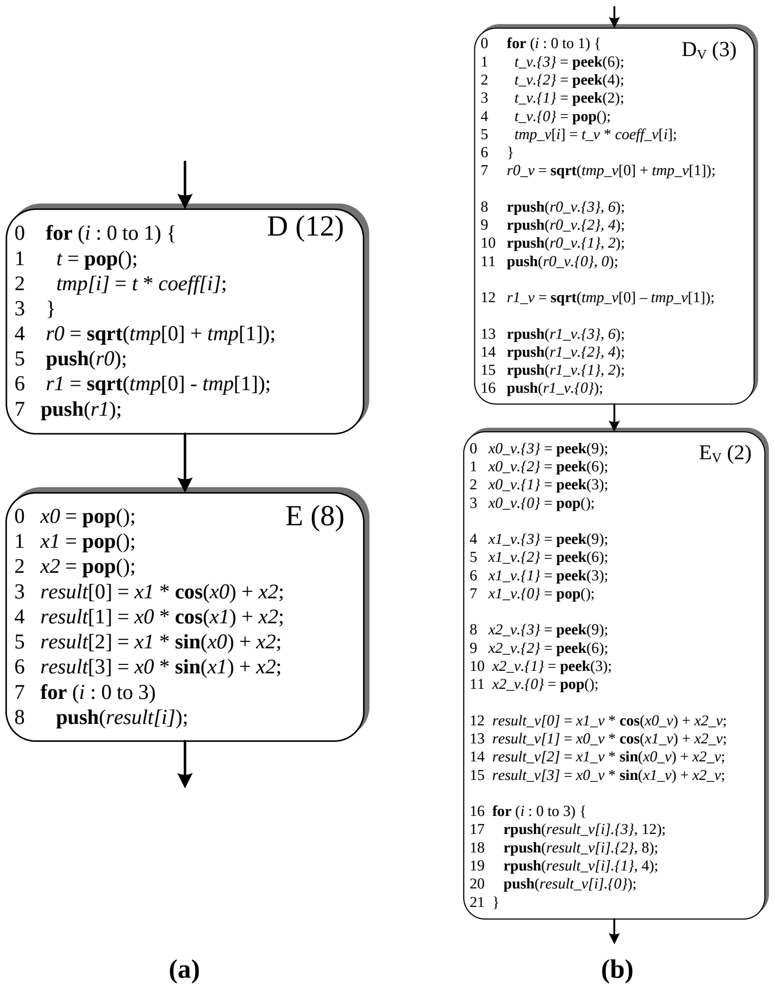
MacroSS finds the smallest factor that the repetition number of each vectorizable actor should be multiplied by based on the following equation:

$$M = Max\{\frac{LCM(SW, R_i)}{R_i}, \quad \forall \text{ SIMDizable actor } A_i\} \tag{4.1}$$

Each term of the Max function finds the smallest factor that each repetition number (R_i) should be multiplied by to make it a multiple of SW. After finding the minimum for each SIMDizable actor, the largest factor is chosen and all of the repetition numbers are scaled based on that. According to Equation (4.1), the repetition numbers of the graph in Figure 4.1a must be scaled by 2 (= M) before SIMDization.

Suppose that, after this adjustment of the repetition numbers, the resulting repetition number of an actor A is $m \times SW$. Then, MacroSS transforms the $m \times SW$ sequential executions of A into m sequential executions of SW data-parallel A's. Since several executions of the SIMDized actor will be running at the same time, only stateless actors are eligible for single-actor SIMDization. This kind of SIMDization can be applied to actors D, E, and G in the example shown in Figure 4.1a. The code in Figure 4.2 illustrates how single-actor SIMDization is performed for actors D and E. Ignoring the tape accesses, it can be seen that the variables in the original actors are packed into vector variables and computation functions are calculated on vector variables instead of scalar. Vector variables are depicted by `_v` suffix as in `tmp_v[]`, `t_v` and `coeff_v[]`. Actors D and E originally had repetition number of 12 and 8 and after SIMDization are executed 3 times and 2 times since each execution of the vectorized actors is in fact 4 data-parallel executions of the original actors.

In the single-actor vectorization, the input and output tapes of a vectorized actor are left as scalar in two cases. First, the producer actor that fills the input tape of the vectorized actor is not SIMDizable. Second, the producer actor is vectorizable but its push rate is different from the pop rate of the consumer actor. For similar reasons, applying vectorization to the tape between the vectorized actors and its consumer is not possible in some cases. Therefore, the input and output tapes of a vectorized actor using single-actor SIMDization are not vectorized and remain as scalar. In order to read or write data elements in the correct order from the scalar input or output tapes in the vectorized actor, the pops/peeks for reading from the input tape and pushes for writing to the output tape must be done in a scalar fashion.



```
0   for (i : 0 to 1) {
1     t = pop();
2     tmp[i] = t * coeff[i];
3   }
4   r0 = sqrt(tmp[0] + tmp[1]);
5   push(r0);
6   r1 = sqrt(tmp[0] - tmp[1]);
7   push(r1);
```

```
0  x0 = pop();
1  x1 = pop();
2  x2 = pop();
3  result[0] = x1 * cos(x0) + x2;
4  result[1] = x0 * cos(x1) + x2;
5  result[2] = x1 * sin(x0) + x2;
6  result[3] = x0 * sin(x1) + x2;
7  for (i : 0 to 3)
8    push(result[i]);
```

```
0    for (i : 0 to 1) {
1      t_v.{3} = peek(6);
2      t_v.{2} = peek(4);
3      t_v.{1} = peek(2);
4      t_v.{0} = pop();
5      tmp_v[i] = t_v * coeff_v[i];
6    }
7    r0_v = sqrt(tmp_v[0] + tmp_v[1]);

8    rpush(r0_v.{3}, 6);
9    rpush(r0_v.{2}, 4);
10   rpush(r0_v.{1}, 2);
11   push(r0_v.{0}, 0);

12   r1_v = sqrt(tmp_v[0] – tmp_v[1]);

13   rpush(r1_v.{3}, 6);
14   rpush(r1_v.{2}, 4);
15   rpush(r1_v.{1}, 2);
16   push(r1_v.{0});
```

```
0   x0_v.{3} = peek(9);
1   x0_v.{2} = peek(6);
2   x0_v.{1} = peek(3);
3   x0_v.{0} = pop();

4   x1_v.{3} = peek(9);
5   x1_v.{2} = peek(6);
6   x1_v.{1} = peek(3);
7   x1_v.{0} = pop();

8   x2_v.{3} = peek(9);
9   x2_v.{2} = peek(6);
10  x2_v.{1} = peek(3);
11  x2_v.{0} = pop();

12  result_v[0] = x1_v * cos(x0_v) + x2_v;
13  result_v[1] = x0_v * cos(x1_v) + x2_v;
14  result_v[2] = x1_v * sin(x0_v) + x2_v;
15  result_v[3] = x0_v * sin(x1_v) + x2_v;

16  for (i : 0 to 3) {
17    rpush(result_v[i].{3}, 12);
18    rpush(result_v[i].{2}, 8);
19    rpush(result_v[i].{1}, 4);
20    push(result_v[i].{0});
21  }
```

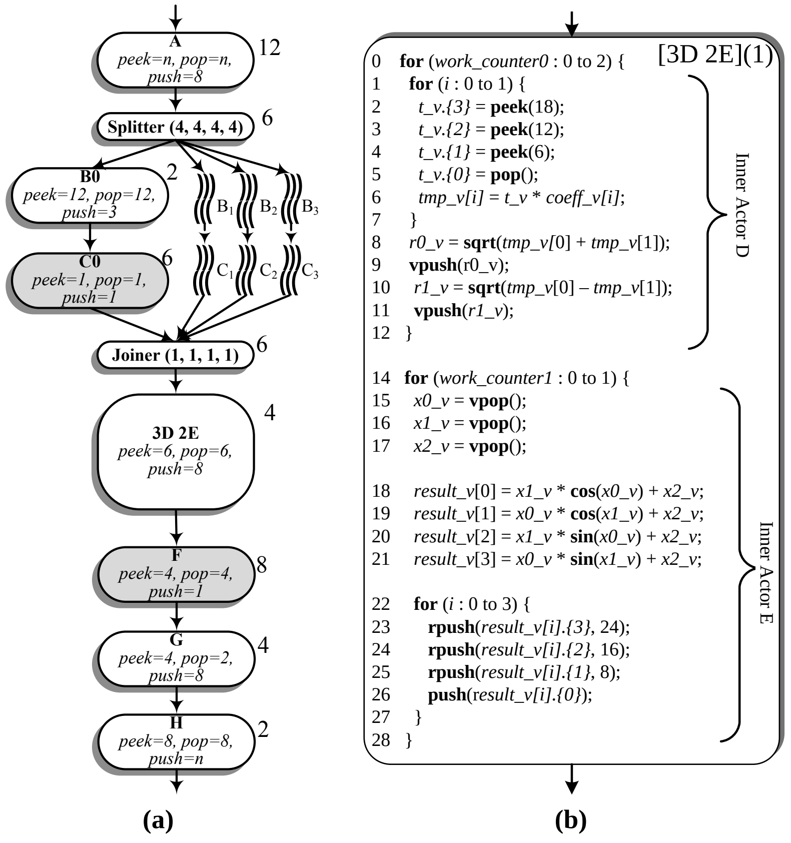
Figure 4.2: *This figure shows how single-actor SIMDization transforms actors D and E into D_V and D_E. All the vector variables are concatenated with $\_v$ at the end. Part (a) of this figure shows the code for actors D and E in scalar mode. Part (b) illustrates the vectorized version of actors D and E.*

Lines 1-4 of D_V in Figure 4.2b show the scalar tape read accesses. After single actor vectorization, the three `peek()`s and one `pop()` in lines 1-4 are induced from one `pop()` in the original code, line 1 of D in Figure 4.2a. The `peek()`s and `pop()`s are reading the scalar input tape for 4 (=SW) consecutive executions of the original actor and packing those four read elements into a vector by writing each element to a lane of a vector variable. The accesses to the i^{th} lane of a vector variable are indicated by `_v.{i}`. After a vector is formed from the scalar input tape in this way, the vector will be used for the computation in the rest of the actor's code. When the actor wants to write data to the output tape, it unpacks the data to scalar variables and pushes them to the scalar output tape (lines 8-11 of D_V in Figure 4.2b). In other words, after each read and before each write to tapes, a SIMDized actor should perform packing and unpacking operations.

Since the tapes are left as scalar and each tape read is replaced by SW tape reads after single-actor SIMDization, it is necessary to perform strided reads to receive the right data element for each of the SW pops. The stride for each set of SW reads in a SIMDized actor is equal to the pop rate in the original actor. For example in Figure 4.2a, since the pop rate of actor D is 2, the `pop()` in line 1 is converted into 4 stride-two input tape reads as shown in lines 1-4 of Figure 4.2b. To read the scalar input tape in a non-destructive way, `peek()` is used instead of `pop()` for the first 3 reads, and the `pop()` is used only for the last read which also adjusts the read pointer of the input tape. For the same set of reasons, the scalar output tape is written with a stride equal to the push rate of the original actor. In Figure 4.2b, lines 8-11 unpack vector variable `r0_v` and write each element to the scalar output tape with a stride of 2, since the push rate of the original actor, D, is 2. The first 3 writes are done using *random access push* operations that do not move the write pointer of the tape (lines 8-10 and 13-15). Random access push operation are indicated by `rpush(data, offset)` in the code. The first argument of `rpush()` is the data to write and the second argument is the offset from the write pointer of the output tape to which the data will be written. The last write of each set of writes is performed using a normal push operation which updates the write pointer of the tape.

In Figure 4.2, only the code for the *work* functions of D and E is shown and the *init* functions are omitted. Actual vectorization of an actor's *work* and *init* method comprises of two parts: identifying variables and constants to be vectorized in an actor and rewriting the actor by replacing the vectorized variables with vector accesses and fixing the tape accesses. Identifying variables and constants to be vectorized can exploit the fact that the tape reads are the source of data for the variables used in the computation assignments inside an actor. A variable definition (i.e. *def*) originating for a pop/peek is marked to be vectorized. For other assignment statements, the *def* is identified as vector if its right hand side contains all variable *use*s marked as a vector. Also, a variable *use* that is used with other vector variable *use*s on the right hand side of a statement is marked as a vector. Similarly, constants used with other vector variable *use*s are marked to be vectorized as well. For example, in line 2 of actor D in Figure 4.2a, `tmp[]` is identified as a vector because the right hand side variable, `t`, is written to by `pop` in line 1. After that, `coeff[]` is also detected as vector because of `t` on the right hand side. After identifying the variables, the statements are rewritten using the vector constructs. Also, the tape accesses are replaced with strided accesses at this point.

Single-actor SIMDization is not applicable to all the actors in a stream graph. Actors with mutable state (i.e. stateful) are excluded from single-actor SIMDization because it is not possible to run multiple executions of them in parallel. Splitters and joiners at this point are also excluded since they consist of only tape access operations without any substantial computation. Actors with function calls that are not supported by the SIMD engine are not SIMDized either. Input-tape-dependent control flow (i.e. *if* statements with pop-dependent conditions) or memory accesses (i.e. pop-dependent array subscripts) can also prevent MacroSS from performing single-actor SIMDization. The way MacroSS handles the input-tape-dependent control-flow structures or memory accesses is by switching to scalar mode (unpacking) before the input-tape-dependent structure and switching back to vector mode after the pop-dependent structure is finished (packing). MacroSS uses an internal cost model to decide if SIMDizing an actor with input-tape-dependent *if* or *for-loop* structures is beneficial or not.



```
for (work_counter0 : 0 to 2) {
  for (i : 0 to 1) {
    t_v.{3} = peek(18);
    t_v.{2} = peek(12);
    t_v.{1} = peek(6);
    t_v.{0} = pop();
    tmp_v[i] = t_v * coeff_v[i];
  }
  r0_v = sqrt(tmp_v[0] + tmp_v[1]);
  vpush(r0_v);
   r1_v = sqrt(tmp_v[0] – tmp_v[1]);
   vpush(r1_v);
}

for (work_counter1 : 0 to 1) {
  x0_v = vpop();
  x1_v = vpop();
  x2_v = vpop();

  result_v[0] = x1_v * cos(x0_v) + x2_v;
  result_v[1] = x0_v * cos(x1_v) + x2_v;
  result_v[2] = x1_v * sin(x0_v) + x2_v;
  result_v[3] = x0_v * sin(x1_v) + x2_v;

  for (i : 0 to 3) {
     rpush(result_v[i].{3}, 24);
     rpush(result_v[i].{2}, 16);
     rpush(result_v[i].{1}, 8);
     push(result_v[i].{0});
  }
}
```

Figure 4.3: *Part (a) of this figure shows the stream graph in Figure 4.1a after vertical fusion of D and E. Part (b) illustrates the vectorized code for the fused actor, $3D\_2E$.*

4.2.2 Vertical SIMDization

Each actor vectorized by single-actor SIMDization performs packing and unpacking at points where tape reads or writes are performed for communicating with producer and consumer actors. The overhead introduced by the packing and unpacking operations can negatively affect the performance gains, even resulting in slowdowns in some cases. Verti-

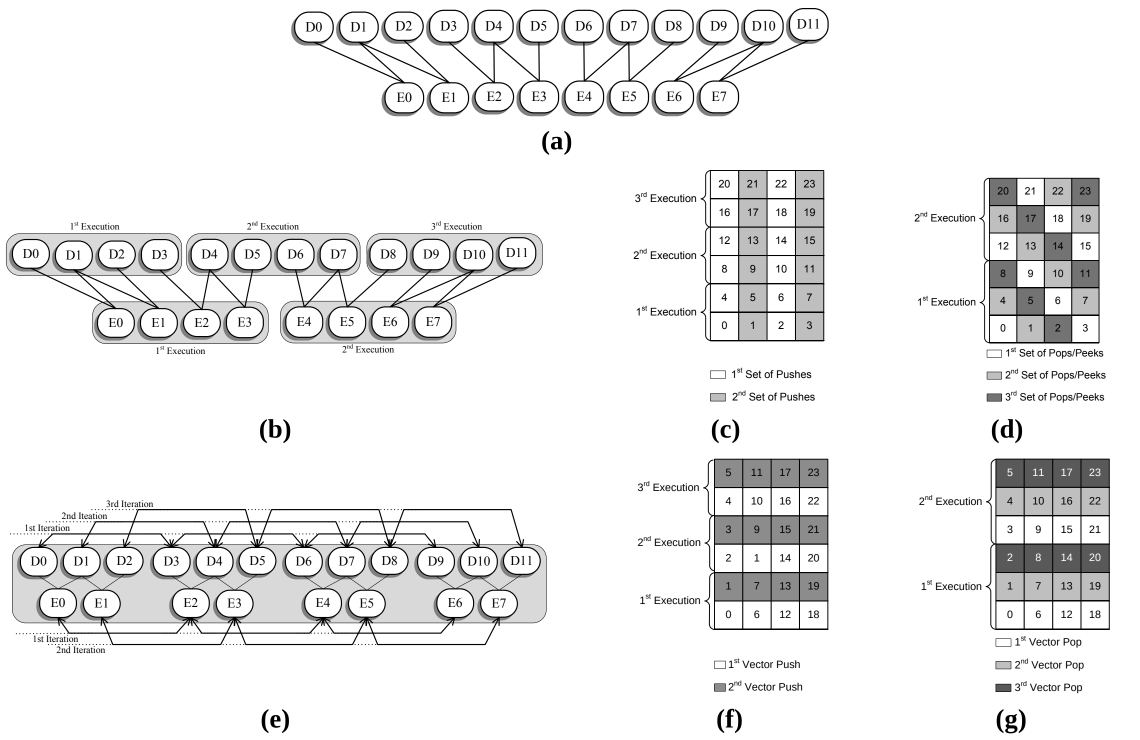


Figure 4.4: *Part (a) shows scalar execution of actors D and E. Part (b) shows the execution of D and E after performing single-actor SIMDization. Part (c) illustrates the order that data elements are written to the tape in the main memory from D. The elements with the same colors are written in one set of push operations. Part (d) is similar to (c) but for the reads in actor E. Part (e) shows how vertical SIMDization changes the execution order of actors D and E. Parts (f) and (g) illustrate the order that the elements are written to and read from the internal buffer between the inner actors D and E.*

cal SIMDization is introduced in MacroSS to overcome this problem by merging vertically aligned vectorizable actors and reducing the number of packing and unpacking operations. In vertical SIMDization, pipelines of vectorizable actors are detected and transformed into a single actor. As long as the original actors in a pipeline are vectorizable, and no actor performs peek operations except the first and the last actor in the pipeline, the resulting coarse actor is guaranteed to be SIMDizable since the transformation does not introduce state or any other construct that may prevent SIMDization. The original actors, which are encapsulated in the new coarse node, are called *inner actors*. Figure 4.3 shows the stream graph after applying vertical fusion to nodes D and E and the resulting coarse actor $3D\_2E$.

After vertical fusion, MacroSS adjusts the repetition numbers of all actors to guarantee that they are all the smallest possible multiples of SIMD width, SW. This adjustment is done in two steps. First, the repetition numbers of inner actors and the coarse actor are changed. The repetition number of each inner actor will be its original repetition number multiplied by $\frac{M'}{SW}$. M' is found by plugging the repetition numbers of the inner actors into Equation (4.1). The repetition number of the coarse actors is set to $\frac{SW}{M'}$. This guarantees that the repetition number of the coarse actor is set to the largest possible multiple or divisor of SW. After doing this step for each vertically fused SIMDizable actor, MacroSS applies Equation (4.1) to the entire graph to ensure that repetition number of all SIMDizable actors, including the coarse actor, is multiple of SW. In general, applying this method guarantees that the repetition vector of the graph is scaled by the smallest possible number. Using this method, the inner actors for D and E in $3D\_2E$ have repetition numbers of 3 and 2, while the new node $3D\_2E$ has a repetition number of 4. The pop rate of $3D\_2E$ is set to 6, which equals the original pop rate of the first inner actor (D) multiplied by the repetition number of that inner actor. Similarly, the push rate of $3D\_2E$ is set to 8. Note that the total number of times that D and E run after the fusion is exactly equal to the number of times before applying fusion.

The graph resulting after vertical fusion will have coarser nodes. The communication between the inner actors of a coarse actor is done through internal buffers (i.e. arrays) instead of global tapes. Transferring data between the inner nodes can be completely done using vectors since packing and unpacking are needed only during tape reads (pops) and writes (pushes) of the new coarse node at the boundaries. The main reason behind this is due to the change in the relative execution order of D and E. This will be illustrated shortly using an example. At this point, single-actor SIMDization can be applied to the vertically fused actor. The code in Figure 4.3b shows how the SIMDization is applied to the new actor. Since actor $3D\_2E$ has 6 pops and 8 pushes, the strides for accessing input and output tapes of $3D\_2E$ are set to 6 and 8. These reads and writes from input and to output tapes are performed, as described in Section 4.2.1, using `peek`, `pop` and `rpush` operations at the beginning and end of $3D\_2E$ (lines 2-5 and 23-26).

The reads and writes between inner actors are handled differently. The previous scalar tape writes of D in lines 8-11 and 13-16 of D_V in Figure 4.2b are now written using vector writes as shown in line 9 and 11 of Figure 4.3b. Vector variable `r0_v` is written to the internal vector buffer between *inner* D and *inner* E using `vpush(r0_v)`. Also, the scalar tape reads of E in lines 0-11 of E_v of Figure 4.2b are replaced with reads from the internal vector buffer as in lines 15-17 in $3D\_2E$. Compared to the code generated after SIMDizing D and E separately, the vertical SIMDization technique in MacroSS eliminates 24 unpacking ([D's repetition number] * [D's push rate] * [SIMD width]) and 24 packing ([E's repetition number] * [E's pop rate] * [SIMD width]) operations.

Figure 4.4 shows the details of how vertical SIMDization changes the execution of a stream graph and eliminates the packing/unpacking operations between the fused inner nodes. Part (a) of this figure shows how actors D and E interact with each other in scalar mode. Since D has a push rate of 2 and E has a pop rate of 3, 12 invocations of actor D feeds 8 invocations of actor E (D_i and E_i denote i^{th} executions of D and E, respectively). In other words, every 3 consecutive executions of D produce enough data for E to consecutively execute 2 times. The 24 elements produced by D are written to the tape in order and read by E in the same order. After performing single-actor SIMDization, every 4 consecutive invocations of D is merged in actor D_V. The first execution of this new actor is similar to executing $D0$, $D1$, $D2$, and $D3$ in parallel as shown in Figure 4.4b. Since every 3 consecutive Ds feeds 2 Es, MacroSS needs to convert the vectors to scalars before each set of scalar strided writes to the output tape of D and then form vectors after each set of scalar strided reads in E to guarantee that E is receiving its data elements in the correct order. Parts (c) and (d) of Figure 4.4 show the order that the pushes in D_V write and pops in E_V read the data elements. If the pushes in D were replaced by a vector push, then elements 0, 2, 4, and 6 would be written to the first row in memory. In that case E will receive its input in the wrong order.

Vertical SIMDization applied to D and E replaces these 2 actors with actor $3D\_2E$. After vectorizing this new actor, every 4 consecutive executions of $3D\_2E$ will be merged together as shown in Figure 4.4e. Since each invocation of this actor executes three Ds first (for-loop in line 0 of Figure 4.3b) and then two Es (for-loop in line 14 of Figure 4.3b), running 4 of them in parallel will result in first running $\{D0, D3, D6, D9\}$, $\{D1, D4, D7,$-$D10\}$, $\{D2, D5, D8, D11\}$ and then $\{E0, E2, E4, E6\}$ and $\{E1, E3, E5, E7\}$. Therefore, because the Ds are generating their outputs in the same order as the Es need them, the scalar tape between original D and E can be changed to vector buffers and extra packing/unpacking operations can be deleted. Figures 4.4f and 4.4g show how the reads and writes are done between internal Ds and Es. As shown, the vertical SIMDization has eliminated the need to perform packing and unpacking between D and E. In summary, vertical fusion of vectorizable actors into a new coarse actor always results in less packing and unpacking operations because of the execution reordering of the inner actors.

4.2.3 Horizontal SIMDization

As mentioned earlier, only actors without mutable state can be SIMDized over using single-actor and vertical SIMDization. Since an invocation of a stateful actor depends on the previous invocation of the actor, different invocations cannot be parallelized. Due to the same reason, the existence of a stateful actor within a pipeline of actors or an actor whose peek rate is greater than pop rate prevents MacroSS from performing vertical SIMDization because the actor resulting after vertical fusion will be a stateful actor.

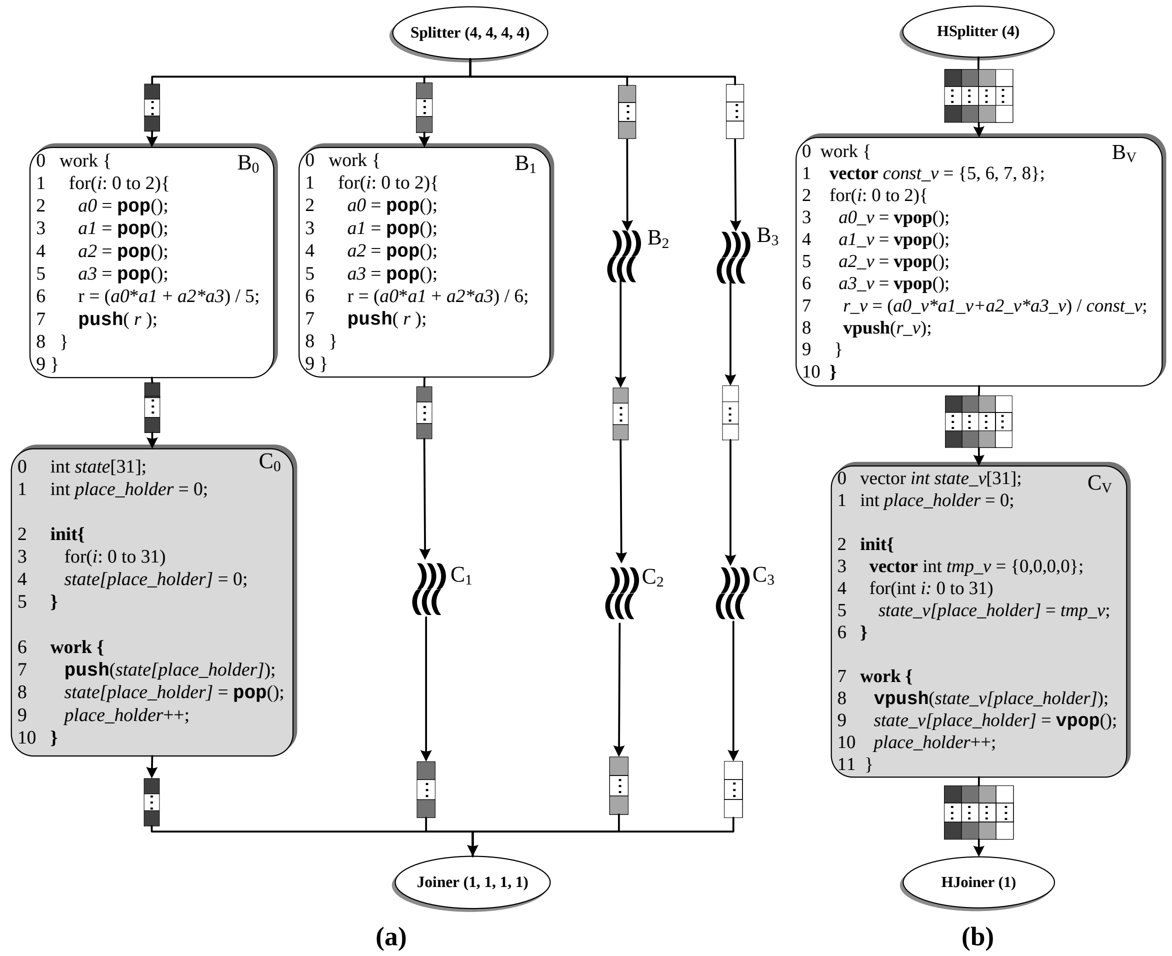
Horizontal SIMDization is an alternative approach taken by MacroSS to vectorize a set of task-parallel isomorphic actors when vertical and single-actor SIMDization are not applicable or result in inefficient SIMD code. First, Horizontal SIMDization finds task-parallel isomorphic actors by investigating each *split-join* (i.e. a subgraph containing a splitter and a joiner and all task-parallel actors between them). After finding the candidates, MacroSS horizontally SIMDizes SW (SIMD Width) isomorphic actors by, conceptually, executing them together side by side. Input (output) tapes of SW actors in a SIMDized set are also SIMDized, making each scalar tape a lane of a SW-wide SIMDized tape. Each actor in a SIMDized set still works on its own tape by accessing each lane of the SIMDized tape. Horizontal SIMDization is able to vectorize stateful actors as well as stateless actors because the state variables are kept in different vector lanes and updated separately similar to the non-vectorized case. The repetition number of the actors involved in this kind of SIMDization, unlike vertical and single-actor SIMDization, is not changed and can be numbers that are not multiples of SW.

Horizontal SIMDization mainly targets task-parallel *isomorphic* actors in *split-joins*.

Two actors are called isomorphic if they have identical *work* and *init* functions with similar or different constant literals. A set of SW isomorphic actors can be horizontally SIMDized as long as the following conditions are true: (1) all of them have the same repetition numbers, (2) all of them have the same *push* and *pop* rates, and (3) all of them are at the same level in a set of pipelines that are children of a *split-join*. Actors B_0 to B_3 and also C_0 to C_3 are considered isomorphic in Figure 4.1a.

Figure 4.5a shows a *split-join* subgraph of the stream graph in Figure 4.1a in more detail. Waves are used for depicting isomorphic actors due to the lack of space. Shaded actors C_0 to C_3 are stateful and can not be vectorized using any of the previously mentioned techniques. Although actors B_0 to B_3 are stateless, fusing each of them with the C_i right after them prevents MacroSS from performing vertical SIMDization on the fused actor. Horizontal SIMDization can overcome this problem by forming one SIMDized actor out of actors B_0 to B_3 and another SIMDized actor out of actors C_0 to C_3 as shown in Figure 4.5b. Note that although the constants in line 6 of B_is are different in each actor, the B_is are still considered isomorphic because the constants can be vectorized together as shown in line 1 of actor B_V in Figure 4.5b.

Before horizontal vectorization, each pipeline of B_i and C_i actors works on a separate set of scalar tapes highlighted by different shades in Figure 4.5a. Horizontal vectorization SIMDizes this set of four scalar tapes into one vector tape (See Figure 4.5b). `vpop()` in line 3 of B_V reads 4 data items at once from the vectorized input tape. The lanes of this vector tape correspond to B_0, B_1, B_2 and B_3 respectively. Similarly, `vpush()` in line 8 pushes 4 data items at once to the vectorized output tape. Since tapes are also vectorized, no non-unit strided access using `peek()` or `rpush()` is needed. Horizontally vectorizing



```
work {
  for(i: 0 to 2){
    a0 = pop();
    a1 = pop();
    a2 = pop();
    a3 = pop();
    r = (a0*a1 + a2*a3) / 5;
    push( r );
  }
}
```

```
work {
  for(i: 0 to 2){
    a0 = pop();
    a1 = pop();
    a2 = pop();
    a3 = pop();
    r = (a0*a1 + a2*a3) / 6;
    push( r );
  }
}
```

```
int state[31];
int place_holder = 0;

init{
  for(i: 0 to 31)
  state[place_holder] = 0;
}

work {
  push(state[place_holder]);
  state[place_holder] = pop();
  place_holder++;
}
```

```
work {
 vector const_v = {5, 6, 7, 8};
 for(i: 0 to 2){
   a0_v = vpop();
   a1_v = vpop();
   a2_v = vpop();
   a3_v = vpop();
   r_v = (a0_v*a1_v+a2_v*a3_v) / const_v;
   vpush(r_v);
  }
}
```

```
vector int state_v[31];
int place_holder = 0;

init{
  vector int tmp_v = {0,0,0,0};
  for(int i: 0 to 31)
    state_v[place_holder] = tmp_v;
}

work {
  vpush(state_v[place_holder]);
  state_v[place_holder] = vpop();
  place_holder++;
}
```

Figure 4.5: *Part (a) and (b) show the graph before and after horizontal SIMDization, respectively.*

tapes can greatly improve the final performance by replacing the scalar tape accesses with vector accesses and, therefore, better utilizing the memory bandwidth. Actors B_0 to B_3, originally had 96 pops (= [pop rates: 12] $\times$ [repetition numbers: 2] $\times$ [SIMD with: 4]) which is reduced to 24 vector pops (= [vector pop rates: 12] $\times$ [repetition number: 2]) after SIMDization. Similarly, the number of pushes in B_is decreases to 6 vector pushes from 24 pushes, and C_i's 24 pops (pushes) drops to 6 vector pops (pushes). In general, the number of tape accesses in the actors between a horizontally vectorized *split-join* structure is always reduced by factor of SW.

During horizontal SIMDization, MacroSS replaces the original splitter and joiner with *horizontal splitter (HSplitter)* and *horizontal joiner (HJoiner)*. In a horizontally vectorized structure, transitions between a scalar tape and vector tape occurs within the HSplitter and HJoiner. The HSplitter reads from a scalar tape and performs packing operations and writes them to its vectorized output tape. The HJoiner reads vector data types from its input and converts them to scalar before writing them to its scalar output tape. For example, in Figure 4.5, before SIMDization, the splitter executes 6 times and, during each execution, it conducts 16 pops from its scalar input tape and distributes the popped values between its scalar output tapes in a round-robin fashion using scalar push operations. After horizontal vectorization, the new HSplitter still executes 6 times and it performs 16 pops from its scalar input tape each time it executes. It forms 4 vectors out of the 16 data elements using packing operations and finally does a vector push to its vector output tape. The HJoiner is formed in a similar way, but instead of packing, it performs unpacking on the vector data it reads from its input tape.

Horizontal vectorization of an actor's *work* and *init* method comprises of two parts similar to single-actor SIMDization: identifying the vectors and rewriting the code for the actor. First, MacroSS needs to identify variables and constants for vectorization. The destination of pop and peek operations are marked as vector variables. Also, if the value of a constant in an actor is different from that of a matching constant in another isomorphic actor, the constant should be raised to a vector constant that contains the values of a matching constant of SW actors. The vector variable `const_v` in line 1 of B_V in Figure 4.5b is created from 4 different constants in B_0 to B_3. The identified vector variables and constants are used as the seeds for marking the other vector variables similar to single-actor SIMDization. After marking is done, MacroSS rewrites the horizontally SIMDizable actors using the marked vectors and changes their input and output tapes to vector tapes. Finally, the splitter and joiner in the horizontally SIMDizable *split-join* are replaced with horizontal splitter and joiner actors.

In summary, horizontal SIMDization is different from vertical and single-actor SIMDization in several ways. First, horizontal SIMDization can be applied only to isomorphic actors. Second, unlike other techniques used by MacroSS, it can handle stateful actors. Third, horizontal SIMDization does not affect the latency of the graph because there is no need to scale the repetition numbers of the actors. Finally, using horizontal vectorization, MacroSS can transform the existing task-level parallelism among the isomorphic actors to data-level parallelism.

4.2.4 Architecture Support for Tape SIMDization

In both single-actor and vertical SIMDization techniques, tape accesses are left as scalar. Converting these accesses to SIMD accesses results in reading or writing the data elements in an order which is different from the scalar execution. Vertical SIMDization reduces this overhead by replacing the scalar tape accesses between a pipeline of SIMDizable actors that are fuse-able with vector accesses to an internal buffer. In this section, two techniques that MacroSS uses to optimize the scalar tape accesses are discussed. The first technique uses a permutation based approach to target the overhead of performing packing/unpacking whenever data is communicated between scalar and vector parts of the stream graph. The second technique shows how MacroSS can simplify the read and write accesses of data that moves between scalar and vector actors in the presence of a unit called the streaming address generation unit (SAGU).

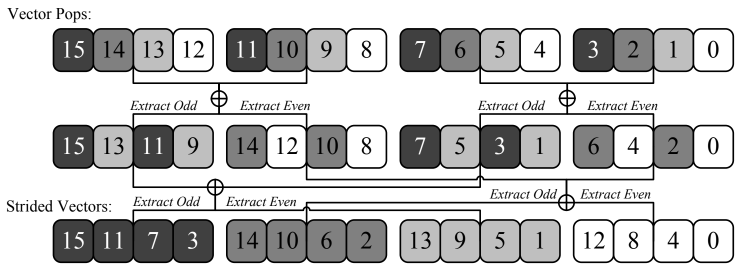


Figure 4.6: *This graph shows how 16 stride-4 tape reads in an actor are replaced with 4 vector pops and 8 permutation instructions*

Permutation-based Tape Accesses: The packing/unpacking overhead exists between scalar and vector actors, such as F and G, in the SIMDized graph in Figure 4.1b. MacroSS optimizes these data conversions for actors whose push or pop counts are powers of 2 using two general architecture independent permutation operation:*extract_even(V1, V2, R)*, *extract_odd(V1, V2, R)*. The *extract_even* (*extract_odd*) operation takes two input vectors, $V1$ and $V2$, and constructs a third vector, R, using even (odd) positions of the inputs. This kind of permutation is supported by almost all SIMD standards (SSE, Altivec, Cell SPU, Neon).

Assume an actor(A) has X_r pop accesses without any peeks. Each pop access is a load operation followed by an add to adjust the position of the read pointer. After single-actor vectorization on A, the stride for scalar pop accesses will be X_r. For example, actor D in Figure 4.1a originally had $X_r = 2$ pops and after SIMDization the stride is 2 as well. This stride guarantees that each set of scalar pops reads the right elements from the input tape. If a load instruction takes C_r cycles, ignoring the add operations, popping the elements from the input tape in actor A_v, actor A after SIMDization, takes $C_r \times X_r \times SW$ cycles. The other way that MacroSS can perform the same pop operations is to do X_r vector loads, and then perform a set of permutations to form vectors identical to the case that the pops were in strided scalar format. MacroSS finds the minimum number of *extract_odd* and *extract_-even* operations to shuffle the elements in the vectors after the vector pops. An example of this is shown in Figure 4.6. Assume that MacroSS is trying to SIMDize an actor with 4 pop operations. Instead of performing 16 strided pop/peek operations, MacroSS can generate 4 vector pops and then use 8 permutation operations (4 *extract_even* and 4 *extract_odd*) to form the strided pattern. This reduces the 16 scalar load operations to 4 vector load operations and 8 permutations. We ignore the savings due to removal of address generation operations.

In general, shuffling the elements of X_r vectors to get to the same number of vectors each with elements strided at distance of X_r from the original vector needs $X_r lg_2 X_r$ *extract_odd* and *extract_even* operations [63]. The same formula can be used to find the number of permutations that are needed to replace scalar push or peek operations with their vector equivalent. MacroSS compares the overhead of performing scalar tape accesses and vector tape accesses to identify the cheaper solution. After finding the cheaper solution, MacroSS transforms the tape accesses. The best solution can be different based on the SIMD width, tape access strides, permutation cost, and also read/write access latencies.

Streaming Address Generation Unit: Exploiting permutation-based tape accesses becomes harder when the push and pop rates are not powers of two or the underlying architecture does not support the needed permutation instructions. In these scenarios, replacing the strided scalar push or pop operations with vector versions in a vectorized actor forces subsequent scalar consumer or producer actors to perform complex address calculations to access the tape in the correct order. Although replacing the scalar accesses with vector accesses reduces the number of memory accesses and address generation operations in the vector actor, the overhead introduced due to additional address calculation operation in the direct consumer or producer is non-trivial. The code in Figure 4.7 shows how the address calculation should be performed in scalar actors that are connected to vectorized actors in which all the pushes are replaced with vector pushes. The *PushCnt* is set to the push rate of the vectorized actor. The overhead introduced by this code on the Intel Core i7 is at best 6 cycles on top of the memory access overhead assuming multiple back-to-back pop operations.

```
if (PushCnt - (BaseCntr-1) == 0 ) {
   BaseCntr = 0;

   if (StrideCntr - (SIMD_SIZE-1) == 0) {
      StrideCntr = 0;
      OffsetAddr = OffsetAddr + (PushCnt << LOG2_SIMD));
   } else { StrideCntr++; }

} else { BaseCntr++; }

OffsetValue = BaseCntr << LOG2_SIMD;
OffsetValue += StrideCntr;
OffsetValue += OffsetAddr;
ResultAddr = OffsetValue + BaseAddr;
```

Figure 4.7: *This code shows the address calculation in a scalar actor which is the consumer of a vectorized actor with vector pushes.*

To deal with this problem, we developed the Streaming Address Generation Unit (SAGU). The SAGU is able to reduce the overhead cost of address calculation in a scalar actor that is connected to a vectorized actor, in which all the scalar strided tape accesses are replaced with vector version, through a special functional unit that loads configuration data (push or pop count) and holds internal state allowing for quick generation of the required addresses. Figure 4.8 shows the hardware of the SAGU. Conceptually, when vector pushes (pops) occur the writes (reads) are row based but the reads (writes) have to access tape in a column-wise order to access the data elements in correct order. The *Stride_Counter* points to the column that needs to be accessed. The *Base_Counter* register points to the row location in the current column that contains the data element needed by the actor. The *Offset_Address* register offsets the *Base_Address* to the next set of vector data elements. Each scalar pop increments the *Base_Counter*. After the number of pops equals to the *Push_Count*, the *Stride_Counter* increments in order to access the next column and the *Base_Counter* is reset. When the *Stride_Counter* equals the *SW*, the *Stride_Counter* resets and the *Offset_Address* increments. The same operation occurs when scalar pushes are used. When designing the SAGU, we found that the largest push/pop count for SIMD to scalar conversion across all the kernels was 16K. With a SIMD width of 4, this allows us to use only 16-bit calculations throughout the unit except when we add the results to the base address register to generate the effective address. Most of the operations occur in parallel making the critical path two 16-bit operations and the 64-bit base address calculation. When optimized, we find that this unit will not be on the critical path allowing the address calculation to take the same amount of time as other address calculation instructions.

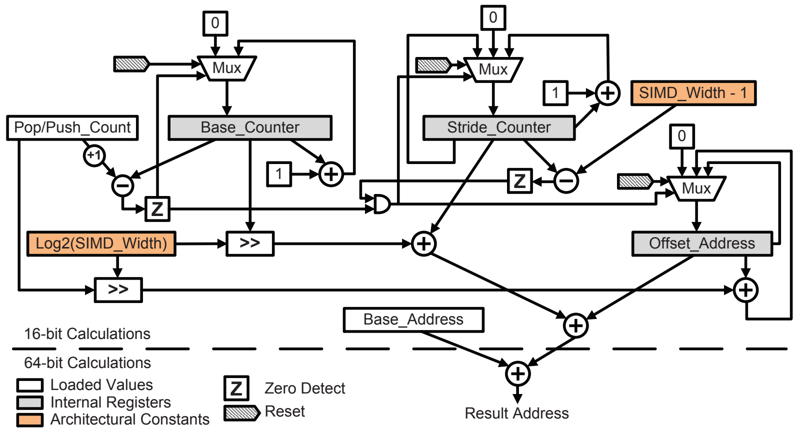


Figure 4.8: *This figure shows the hardware for the SAGU.*

To use the SAGU, only minor modification to the ISA or hardware needs to be done. Many ISAs like Intel x86 [43] and ARM [71] support multiple addressing modes which can perform operations on multiple address registers. There are available addressing mode configurations in these ISAs that we can modify to support the SAGU addressing mode. Effectively, this would be like performing a post-increment on an address register which would be transparent to the programmer and architecture. The alternative to this technique, if the ISA cannot support the addressing mode, would be to add another opcode to setup the SAGU and to increment it. Before starting each scalar actor, we would perform a SAGU setup and write the pop or push count. This would reset the internal counters to 0. After performing a pop or push operation, on the address register we would execute a SAGU increment to update the value to the next memory location. This would only require 2 additional instructions to the ISA and introduce 1 extra instruction for each memory operation in the program which would be far less than directly calculating the address. Because of the low cost[1] of the SAGU and the speed of the calculation, multiple units can be implemented if needed with little to no overhead.

4.2.5 Implementation

MacroSS's SIMDization algorithm can be divided into several distinct phases. In this section, a high-level overview of these steps are given. Algorithms 1 illustrates the overall ordering of the macro-SIMDization phases in MacroSS for vertical, horizontal SIMDization, and tape SIMDization. The remainder of this section explains each of the phases and their relationship to one another.

Prepass Optimizations and Scheduling: MacroSS applies a set of classic and streaming optimizations and also performs scheduling before starting the macro-SIMDization. The classic and streaming optimizations mainly improve the overall performance of the

[1]Area overhead is less than 1% of the area of the Core i7. This was measured by synthesizing the hardware model.

Algorithm 1 Macro SIMDization Steps

```
Input: Stream Graph $G$, Architecture Description $A$
    {Apply prepass classic and streaming optimizations and also perform scheduling on the graph.}
1   Prepass-Optimizations ($G$);
2   Prepass-Scheduling ($G$);

    {Find the segments suitable for vertical/horizontal SIMDization.}
3   ($G_V$, $G_H$) := Find-Vectorizable-Segments($G$, $A.CostModel$);

    {Adjust the repetition numbers and perform vertical SIMDization on the specified segments.}
4   Adjust-Repetition-Numbers ($G$);
5   Vertically-SIMDize ($G_V$, $A.CostModel$);

    {Perform horizontal SIMDization after vertical is finished.}
6   Horizontally-SIMDize ($G_H$, $A.CostModel$);

    {Apply Permutation-based optimizations and exploit SAGU.}
7   Optimize-Tapes ($G$, $A.CostModel$);

    {Generate intermediate code for the specified target.}
8   Emit-Intermediate-Code ($G$, $A$);
```

graph. The streaming optimization in some cases result in more efficient macro-SIMDization. For example, static parameter propagation, which propagates the values of the static read-only variables of an actor to all of its instances, helps detection of isomorphic actors. The steady state scheduling of the stream graph is also performed as a prepass.

Identify Vectorizable Segments: In this phase, MacroSS examines the stream graph and finds the segments of the graph that are suitable for vertical and horizontal SIMDization. For vertical SIMDization, MacroSS starts from a single vectorizable actor. This actor is added to an empty pipeline of vectorizable actors. Then MacroSS examines the consumer of that actor. If the consumer is also vectorizable and can be fused with the original actor without introducing state, it is added to the pipeline. This is repeated until the pipeline can not be extended anymore. At this point, all the actors in the pipeline are marked for vertical vectorization and added to G_V. Identifying horizontally vectorizable *split-join*s starts by testing the eligibility of a given *split-join* based on the definition given in Section 3.3. If a *split-join* passes the eligibility test it will be added o G_H.

One actor may be a member of both G_V and G_H. Since MacroSS applies one form of SIMDization to any actor, it uses its cost model to choose what type of SIMDization (vertical or horizontal) is more effective for the actors that are in both G_V and G_H. At the end, MacroSS guarantees that the intersection of the sets G_V and G_H is empty.

Vertical SIMDization and Repetition Number Adjustment: After finding the segments suitable for horizontal and vertical SIMDization, MacroSS adjusts the repetition numbers of the actors as described in Section 4.2.2. Then, the actual vertical vectorization is performed. This parts fuses the pipelines of vectorizable actors (G_V) found in the previous steps and changes them to vectorizable actors. Single-actor SIMDization is done as a special case of vertical SIMDization when a pipeline of vectorizable actor contains only one actor.

Horizontal SIMDization: After vertical SIMDization, the steady state repetition numbers are finalized. *Split-join*s eligible for horizontal SIMDization are passed to this phase and MacroSS changes the splitter and joiner actors to their horizontal versions. The statements in the task-parallel actors between the splitter and joiner are also merged to form vector instructions.

Tape Optimization: After vertical and horizontal vectorization, MacroSS searches for opportunities to perform tape optimization that are discussed in Section 4.2.4. This phase basically finds eligible set of reads or writes. Then, if it is cheaper, MacroSS replaces them with vector read or writes plus permutation instructions. If the target architecture is equipped with SAGU, MacroSS looks for cases where it can be exploited.

Code Generation: The final phase of macro-SIMDization deals with intermediate code generation. In this phase, MacroSS maps the internal stream representation to the target specific code (C++ in this case) and uses available architecture-dependent intrinsics to better utilize the target SIMD engines.

4.3 Comparison To Traditional SIMDization

Since MacroSS generates the intermediate code in a conventional imperative language, such as C or C++, traditional vectorization techniques can also be a viable approach to perform SIMDization on streaming applications. Traditional vectorization techniques mainly consist of inner-most loop, outer loop, and superword level parallelism extraction [3, 4, 64, 5, 51]. In this section, we try to compare MacroSS's graph-level SIMDization to traditional techniques and highlight the differences.

As streaming code gets converted to imperative intermediate code, it gets harder to extract the high-level information that is available at the graph-level. As a result, performing effective SIMDization becomes very difficult for some actors. Second, in some cases, traditional SIMDization is predicated on having complicated, carefully phase-ordered compiler analysis that needs the code in a certain templated form.

One of the points that makes MacroSS's SIMDization more powerful than any other vectorization technique on intermediate codes is the ability to identify isomorphic actors and perform horizontal SIMDization. At the graph level, MacroSS knows the relation between the actors and can detect the task-parallel isomorphic actors by doing a graph

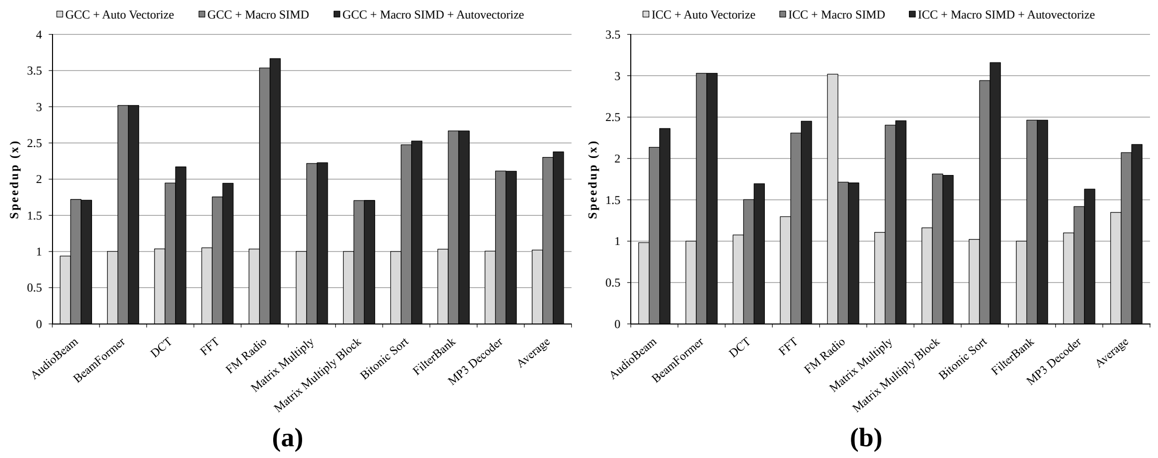


Figure 4.9: *In this graph the performance benefits of applying traditional auto-vectorization, macro-SIMDization, and both of them together are compared. Part (a) shows the speedups when GCC is used as the intermediate compiler. Applications in part (b) are compiled with Intel Compiler (ICC).*

traversal. Performing the same task on the intermediate code is complicated. To find the isomorphic actors, the auto-vectorizer needs to extract the task graph and then compare the source code for the actors. Both of extracting the task graph and matching source code can be obfuscated by other optimizations.

The other issue that may disable auto-vectorization of the intermediate code is inability to adjust the schedule of the task graph. One of the main parts of the schedule is the repetition numbers. MacroSS can intelligently scale the repetition numbers as needed by the SIMDization. Since the repetition numbers affect many parts of the generated code such as buffer (i.e. tape) allocation, and for-loop boundaries, they are not easily possible to adjust after generation of intermediate code.

Vertical SIMDization is another technique that MacroSS uses to perform vectorization. Even though performing vertical fusion on selected actors is in theory possible on intermediate code, it needs complex transformations and compiler analysis such as memory aliasing analysis, loop distribution, and loop relation analysis. MacroSS does not need these complex transformations and analyses since, at the graph-level, aliasing information and the relation between across is already embedded.

Although we are not proposing any universal partitioning approach that can handle both SIMDization and multi-core partitioning, performing vectorization on the high-level graph makes it possible for the partitioner and mapper parts of the streaming compiler to be able to make SIMD-aware decisions. This can lead to finding more efficient graph partitioning and mapping decisions. Since the intermediate code is already partitioned without considering possibility of SIMDization, it under-performs the macro-SIMDized code even after auto-vectorization.

In summary, MacroSS's SIMDization techniques are more efficient than auto-vectorization approaches because MacroSS has the ability to decide which actors are suitable for what kind of vectorization at the graph-level, transform the graph, adjust the schedule accordingly and generate permutation instructions based on actors read and write characteristics. Performing the same tasks during auto-vectorization after generation of intermediate code is difficult.

4.4 Experiments

In this section, macro-SIMDization techniques in MacroSS are evaluated and compared against traditional techniques to perform auto-vectorization on languages. Also, the effectiveness of vertical and horizontal SIMDization is shown. The performance benefits of the streaming address generation unit is measured and presented in this section. Finally, the interaction between macro-SIMDization and multi-core scheduling is discussed.

Methodology: A set of benchmarks from the StreamIt benchmark suite [79] is used to evaluate MacroSS. The benchmarks are compiled and evaluated on a 3.26 GHz Intel Core i7 processor. The Intel Core i7 is used because it is equipped with the latest version of the SIMD engine from Intel, SSE 4.2.

MacroSS implementation is based on the StreamIt compiler. The macro-SIMDization steps are implemented as a separate compiler backend. The output of MacroSS is C++ code. To convert the generated C++ to x86 binary, GCC 4.3 [23] and Intel Compiler (ICC) 11.1 [44] are used. Both of these compilers are capable of performing aggressive optimizations and also auto-vectorization on C++ code. ICC is considered one of the best for its capabilities in performing inner-most, outer-most loop and superword-level parallelism vectorization. GCC also supports auto-vectorization for x86 processors and is widely used to compile C/C++ for Intel processors. In order to isolate the benefits of macro-SIMDization, all the experiments are performed using only one core of the processor except in the last experiment where we show performance benefits compared to multiple cores.

The original StreamIt backend in MacroSS is used to generate the baseline scalar intermediate C++ code. The baseline intermediate code is compiled to x86 binary using GCC or ICC with aggressive optimization flags enabled. The auto-vectorization pass in these compilers is used to perform traditional auto-vectorization on the generated C++ code. To macro-SIMDize streaming applications, the new backend in MacroSS is used to generate macro-SIMDized intermediate C++ code using target specific vector types and intrinsics. For measuring the performance of the generated binary the performance counters on the Intel Core i7 are exploited.

Overall Performance: The set of StreamIt benchmarks is compiled using macro-

SIMDization and compared against ICC's and GCC's auto-vectorization. ICC and GCC are the leading auto-vectorizer compilers for Intel architectures capable of applying complex vectorization techniques proposed in the literature. Figure 4.9 illustrates how MacroSS's techniques perform compared to traditional auto-vectorization techniques. Figure 4.9a shows performance comparison between GCC's auto-vectorized, macro-SIMDized and auto-vectorized macro-SIMDized code. Figure 4.9b contains the same comparison for ICC. In both cases, macro-SIMDization achieves higher performance gains compared to auto-vectorization. On average, macro-SIMDization improves the final performance by an additional 54% and 26% compared to GCC and ICC auto-vectorizations. Applying both macro-SIMDization and auto-SIMDization can improve the performance by another 1.5% and 2.2% in benchmarks compiled using GCC and ICC. The only case that traditional auto-vectorization outperforms macro-SIMDization is *FMRadio* on ICC. In this special case, ICC performs inner-loop vectorization on the main for-loop in the code which results to aligned memory accesses but MacroSS's macro SIMDization results in unaligned memory accesses. It is possible to make MacroSS leave this for-loop for inner-loop vectorizer since, during macro-SIMDization, it knows inner loop vectorization will be more efficient in this special case. *BeamFormer* and *FilterBank* mainly consists of several pipelines of *split-join* structures with isomorphic task-parallel actors. It is not possible to collapse these pipelines into one pipeline because they have stateful actors. Therefore, the speedups in these two benchmarks are mainly due to horizontal vectorization. In summary, GCC shows unimpressive gains using auto-vectorization. Although, ICC shows fairly large gains (1.34x on average), MacroSS's techniques result in even larger gains (2.07x on average). Having access to global information enables MacroSS to achieve significant speedup.

Effect of Vertical SIMDization: Vertical SIMDization is one of the main techniques that MacroSS uses to perform vectorization on streaming graphs. Figure 4.10 illustrates, the effectiveness of this type of SIMDization. In this experiment, the baseline is a streaming graph macro-SIMDized with only single-actor SIMDization and compiled with GCC. As shown in the figure, vertical SIMDization, on average, improves the performance of the baseline by 40%. *Matrix Multiply Block* benefits the most because the vertical fusion of SIMDizable actors eliminates a large number of packing/unpacking operations. Without vertical fusion, macro-SIMDization in this benchmark would result in significantly less speedup then that shown in Figure 4.9a. The benefits in *FilterBank* and *BeamFormer* are very negligible because these benchmarks are vectorized mostly using horizontal vectorization. In *FMRadio* and *AudioBeam* the opportunity for performing vertical SIMDization is very small because most of the vectorizable actors in these benchmarks are isolated from each other and do not form a pipeline.

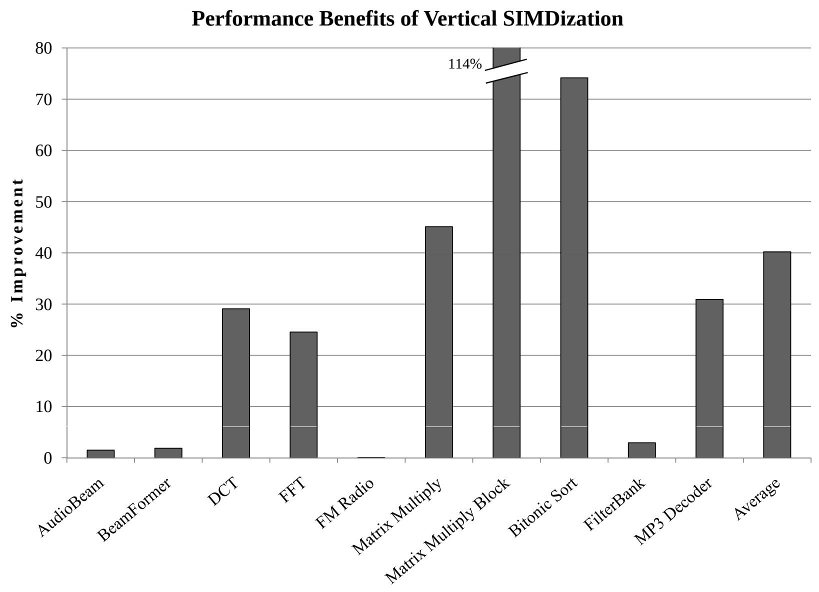


Figure 4.10: *This graph shows percent speedup due to vertical SIMDization compared to single-actor SIMDization.*

Streaming Address Generation Unit: MacroSS utilizes the SAGU to eliminate the packing/unpacking overhead and also improve memory bandwidth utilization when data is crossing scalar and vector boundaries in a stream graph. To evaluate the benefits of the SAGU, we use the performance counters on the Intel Core i7 to find the overheads introduced by packing and unpacking operations and also scalar memory accesses. Figure 4.11 illustrates the effect of utilizing SAGU. The baseline in this graph is macro-SIMDized code. On average, this unit can improve the final performance of the macro-SIMDized benchmarks by 8.1%. The performance of *Matrix Multiply* and *DCT* are improved 22% and 17% respectively because they perform a large number of packing/unpacking operations and scalar memory reads and writes. *BeamFormer* shows the least improvement because almost all the speedup in this benchmark is due to horizontal SIMDization. *MP3 Decoder* is also not affected by the SAGU because its computation to communication ratio is very high and the packing/unpacking operations do not cause a substantial performance overhead.

Multicore and Macro-SIMDization: Implementing a scheduler to decide how to partition a stream graph between multiple cores and also use the SIMD engines is a non-trivial task. Partitioning and mapping decisions taken by a naive multi-core scheduler may reduce the SIMD opportunities. In this section, we show conservatively estimated numbers on how a simple SIMD-aware multi-core scheduler/partitioner performs. The scheduler we use in this experiment first performs multi-core partitioning and then performs macro-SIMDization. This approach reduces the opportunities for performing vertical fusion and also horizontal SIMDization. If multi-core partitioning removes most of the benefits of the SIMDization and the scheduler has to choose between SIMDization and multi-core execu-

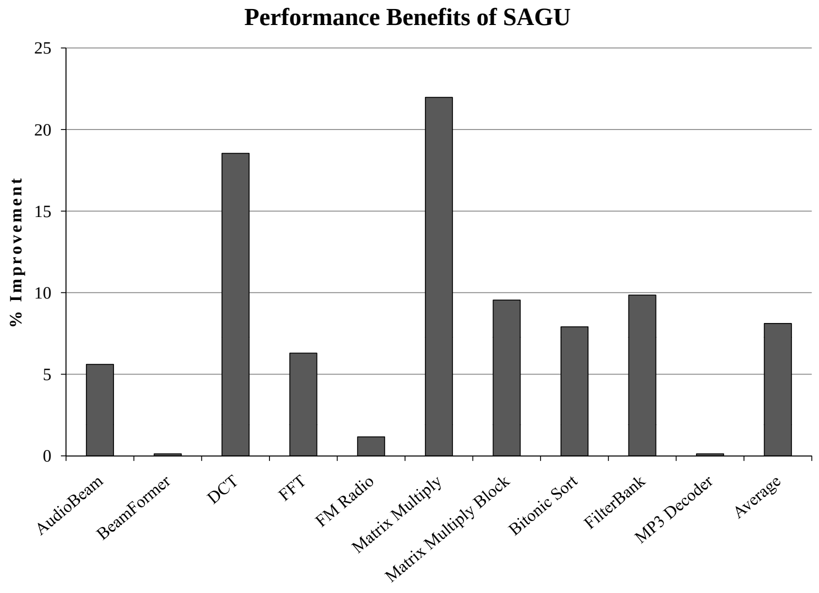


Figure 4.11: *This graph shows how SAGU can improve the performance of a macro-SIMDized graph.*

tion, it always chooses SIMDization because it reduces memory/cache traffic and communication overhead between the cores. Since the multi-core scheduler does not consider the possible benefits of vertical fusion and horizontal SIMDization in several benchmarks, the performance benefits of SIMDization is reduced compared to Figure 4.9. Therefore, these numbers are conservative estimates of the performance of a SIMD-aware multi-core scheduler. As shown in Figure 4.12, the performance benefits of 4-core execution is within 5% of macro-SIMDized 2-core execution. Exploiting the SIMD engines increases the speedup from 1.28x to 2.03x in 2-core schedule and from 1.85x to 3.17x in 4-core schedule. For *Matrix Multiply* and *Matrix Multiply Block*, the scheduler prefers to only use the SIMD engines because multi-core partitioning, in this case, leads to high inter-core communication overhead.

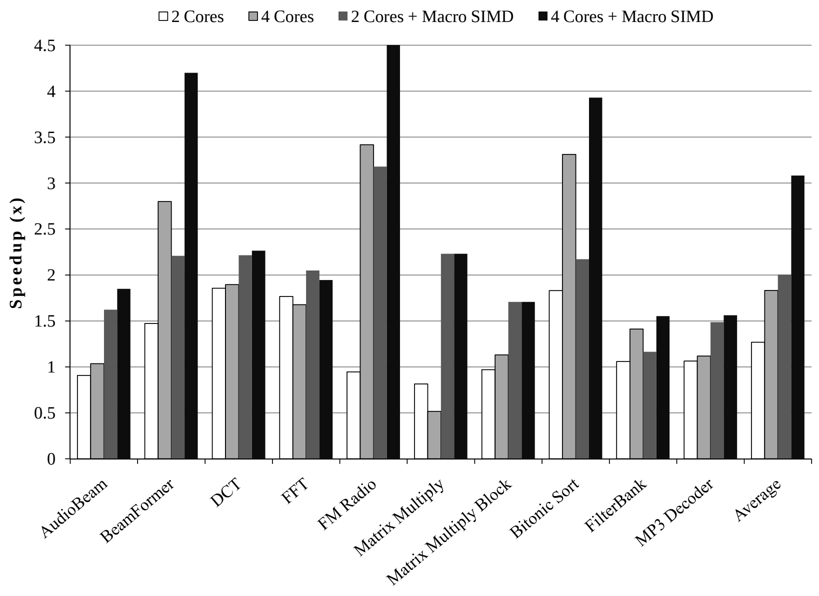


Figure 4.12: *The performance benefit of SIMDization in case a graph is scheduled for multi-core is shown in this graph.*

4.5 Related Work

There is a large body of literature that deals with exploiting parallelism in streaming languages for better performance [79, 13, 16]. The most relevant works include stream graph refinements to extract coarse-grain task-level, data-level and pipeline parallelism and map them onto multi-core architectures [26, 25]. Authors in [49] applied modulo scheduling to task graphs for maximizing pipeline parallelism also on multi-core architectures. Our work is distinctively different from and complementary to these previous works in its ability to exploit SIMD parallelism and generate SIMD enabled codes for various architectures. Vertical SIMDization focuses on fine-grain SIMD parallelism, while horizontal SIMDization transforms task-level parallelism to SIMD parallelism.

Auto-vectorization and SIMD code generation were studied extensively in the literature. The seminal work of Allen and Kennedy on the Parallel Fortran Converter [3, 4] set the grounds for most of the work on auto-vectorization that followed. For targeting a variety of SIMD architectures and solving severe problems that arise, specifically data alignments and permutations, a large number of studies has been conducted [84, 69, 63, 62, 51, 20]. All these techniques can be applied to the generated intermediate code of streaming applications. However, our work is unique in that vectorization is applied on a higher level of representation of the program, which enables us to utilize global information such as execution rates of actors and exposed data communications for generating better vectorized codes. In contrast to focusing on local structures like loop nests and basic blocks, our macro-SIMDization leverages the streaming applications' static characteristics, such as static schedules and pre-defined data access patterns.

There has been recent work [60] on generating efficient permutation instructions based on StreamIt, but for only one specific SIMD device (VIRAM). MacroSS provides efficient SIMDization for streaming applications which is flexible and portable enough to be applied to a variety of SIMD architectures.

Vectorizing computations that access non-unit stride data motivated the development of the SIMdD (Single Instructions on Multiple disjoint Data) model and SIMdD architectures, such as the IBM eLite DSP[58]. Such architectures better support non-consecutive data accesses via vector pointer hardware. Tuned for streaming applications in which non-unit strides are statically known and fixed for the entire execution of an actor, our architectural support, SAGU, is simpler and entails smaller overheads than what is available in general SIMdD architectures.

4.6 Summary

As SIMD-enabled multi-core systems become ubiquitous, it is critical for programming languages and compilers to be able to flexibly target both the SIMD and multi-core aspects of these architectures. Several retargetable streaming languages, such as StreamIt, have been proposed to exploit parallelism across the cores. These languages apply traditional auto-vectorization to the imperative intermediate code (e.g. C/C++) to target SIMD engines. In many cases, applying auto-vectorization to the generated intermediate code results in under-utilization of SIMD engines because much of the high-level information available in the streaming application, such as data-flow information and the set of valid schedules, is not used by the auto-vectorizer.

In this work, we introduce macro-SIMDization: a technique for vectorizing stream graphs using the high-level information available in streaming programs. A new compilation system, MacroSS, is developed to show the benefits of macro-SIMDization compared to traditional SIMDization techniques. MacroSS utilizes three new techniques to achieve high utilization of the SIMD engines: single-actor, vertical, and horizontal SIMDization. Architectural support for tape optimizations, using general permutation operations and a streaming address generation unit (SAGU) is also discussed as a part of this work.

Our results show that MacroSS is capable of improving the performance of streaming applications by an average of 54% and 26% compared to auto-vectorizers in GCC and Intel compiler, respectively. In the experiments, we also evaluated how the SAGU can improve the performance on average by an additional 8.1% by eliminating packing/unpacking operations between scalar and vector actors. Finally, we show the performance benefits of macro-SIMDization in the presence of a naive multi-core scheduler for streaming applications. Even with a naive multi-core scheduler, we estimate that we can achieve better performance than a 4-core Intel Core i7 on only 2-cores using SIMD. The results indicate that performing macro-SIMDization can significantly improve the performance of streaming applications and extend their retargetability by making them more suitable for SIMD programming.

CHAPTER V

Portable Stream Compilation for GPUs

5.1 Introduction

Among the multitude of vastly different solutions offered by hardware companies, graphics processing units (GPUs) have been shown to provide significant performance, power efficiency and cost benefits for general purpose computing in highly parallel computing domains. Recently, heterogeneous systems that combine traditional processors with powerful GPUs have become standard in all systems ranging from servers to cell phones. GPUs achieve their high performance and efficiency by providing a massively parallel architecture with hundreds of in-order cores while exposing parallelism mechanisms and the memory hierarchy to the programmer. Recent works have shown that in the optimistic case, speedups of 100-300x [67] and in the pessimistic case, speedups of 2.5x [54] have been achieved between the most recent versions of GPUs compared to the latest processors. Maximizing the utilization of the GPU in heterogeneous systems will be key to achieving high performance and efficiency.

While GPUs provide an inexpensive, highly parallel system for accelerating parallel workloads, the programming complexity posed to application developers is a significant challenge. Developing applications to utilize the massive compute power and memory bandwidth requires a thorough understanding of the algorithm and details of the underlying architecture. Graphics chip manufacturers, such as NVIDIA, have tried to alleviate the complexity problem by introducing user-friendly programming models, such as CUDA [65]. Although CUDA and other similar programming models abstract the underlying GPU architecture by providing a unified processor model, managing the amount of on-chip memory used per thread, the total number of threads per multiprocessor, and the pattern of off-chip memory accesses are examples of problems that developers still need to manage in order to maximize GPU utilization [70]. Often the programmer must perform a tedious cycle of performance tuning to extract the desired performance.

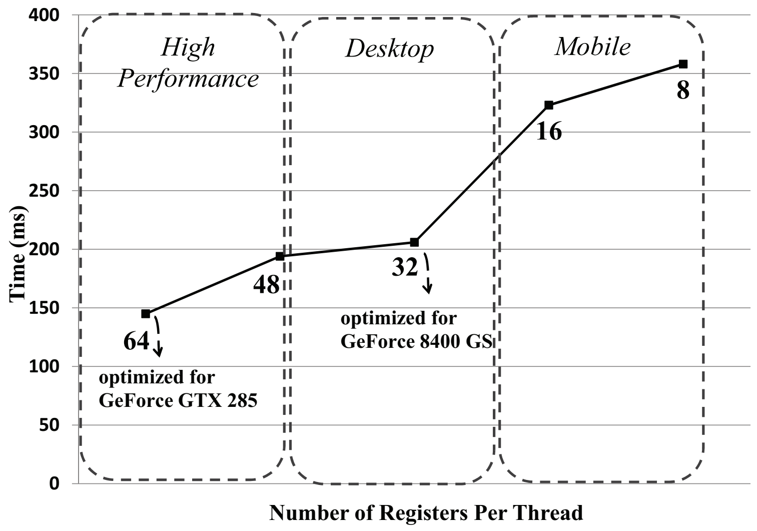


Figure 5.1: *This graph shows the runtime of a kernel optimized for architectures with different number of registers on a GeForce GTX 285 which has the most number of registers. The kernel used in this graph is organized in 128 blocks each with 256 threads.*

Another problem of developing applications in CUDA is the lack of efficient portability between different generations of GPUs and also between the host processors and GPUs in the system. Different NVIDIA GPUs vary in several key micro-architectural parameters such as number of registers, maximum number of active threads, and the size of global memory. These parameters will vary even more when newer high performance cards, such as NVIDIA's Fermi [66], and future resource-constrained mobile GPUs with less resources are released. These differences in hardware lead to a different set of optimization choices for each GPU. As a result, optimization decisions for one generation of GPUs are likely to be poor choices for another generation. An example of this is shown in Figure 5.1. This figure shows a CUDA kernel that requires 78 registers per thread, running with 128 blocks of 256 threads per block on an NVIDIA GeForce GTX 285. This graph shows how the runtime (lower is better) would change if the benchmark was optimized for GPU architectures with less than 16K registers available on each streaming multiprocessor of the GTX 285. For example, if this kernel is compiled for GeForce 8400 GS, it will use 32 registers per thread since there are 8K registers available for the 256 threads in each block on that architecture. Data elements that do not fit in the smaller register file will be spilled to the slower parts of the memory hierarchy causing performance degradation. In short, CUDA code must be separately customized for each target GPU as the choice of optimizations for peak performance is typically sensitive to the hardware configuration.

One solution to the GPU programming complexity is to adopt a higher level programming abstraction similar to the stream programming model. The streaming model provides an extensive set of compiler optimizations for mapping and scheduling applications to various homogeneous and heterogeneous architectures ([25, 26, 49]). The retargetability of streaming languages, such as StreamIt [79], has made them an excellent choice for parallel system programmers in shared/distributed memory and tiled architectures. Streaming language retargetability and performance benefits on heterogeneous systems are mainly a result of having well-encapsulated constructs that expose parallelism and communication without depending on the topology or granularity of the underlying architecture.

GPUs are important drivers for current and future heterogeneous systems, therefore extending the applicability of streaming languages to GPUs is advantageous for several reasons. First, streaming, which expresses programs at a higher level than CUDA, enables optimizing and porting to different generations of GPUs and between different topologies of CPUs and GPUs. Second, exposed communication in streaming programs help the compiler to efficiently map data transfers onto different memory hierarchies. Finally, streaming applications can be tailored for any number of cores and devices by performing graph restructurings such as horizontal or vertical fusion or fission of actors.

In this work, we introduce *Sponge*, a streaming compiler for the StreamIt language that is capable of automatically producing customized CUDA code for a wide range of GPUs. Sponge consists of stream graph optimizations to optimize the organization of the computation graph and an efficient CUDA code generator to express the parallelism for the target GPU. Producing efficient CUDA code is a multi-variable optimization problem and can be difficult for software programmers due to the unconventional organization and the interaction of computing resources of GPUs. Sponge is equipped with a set of optimizations to handle the memory hierarchy and also to efficiently utilize the processing units.

The Stream-to-CUDA compilation in Sponge consists of four steps. First, Sponge performs graph reorganization and modification on the stream graph and also classifies actors based on their memory traffic. The classification information is used throughout all the phases of the compilation. Second, memory layout optimizations are performed. These optimizations are designed to enable efficient utilization of the memory bandwidth. In this phase, Sponge decides if actors should use the faster but smaller on-chip memories or the slower but larger off-chip memory on the GPU. Also, techniques such as *helper threads* and *bank conflict resolution* in the context of StreamIt are introduced to increase the efficiency of memory accesses. The third compilation phase deals with actor size granularity of each thread. In this step, based on the classification information from step one, Sponge tries to create larger threads by fusing producer/consumer actors in order to reduce communication and kernel call overheads. Finally, software prefetching and loop unrolling are used to exploit unused registers to decrease loop control code overhead and increase memory bandwidth utilization.

In summary, this part makes the following contributions:

- Extending applicability and portability of synchronous data-flow languages, specifically StreamIt, to GPUs.
- Streaming-specific optimizations for CUDA and generic CUDA optimizations for streaming applications.
- Discussion of the limitations of StreamIt as a GPU programming language.

The rest of this chapter is organized as follows. In Section 5.1, the stream programming model, the input language (StreamIt), and the CUDA programming model are discussed. Portable stream compilation in Sponge and its optimizations are explained in Section 5.3. Experiments are shown in Section 5.4. A comparison between two hand-optimized CUDA benchmarks and their StreamIt implementation is done in Section 5.5. Related works are discussed in Section 5.6. Finally, Section 5.7 contains the summary.

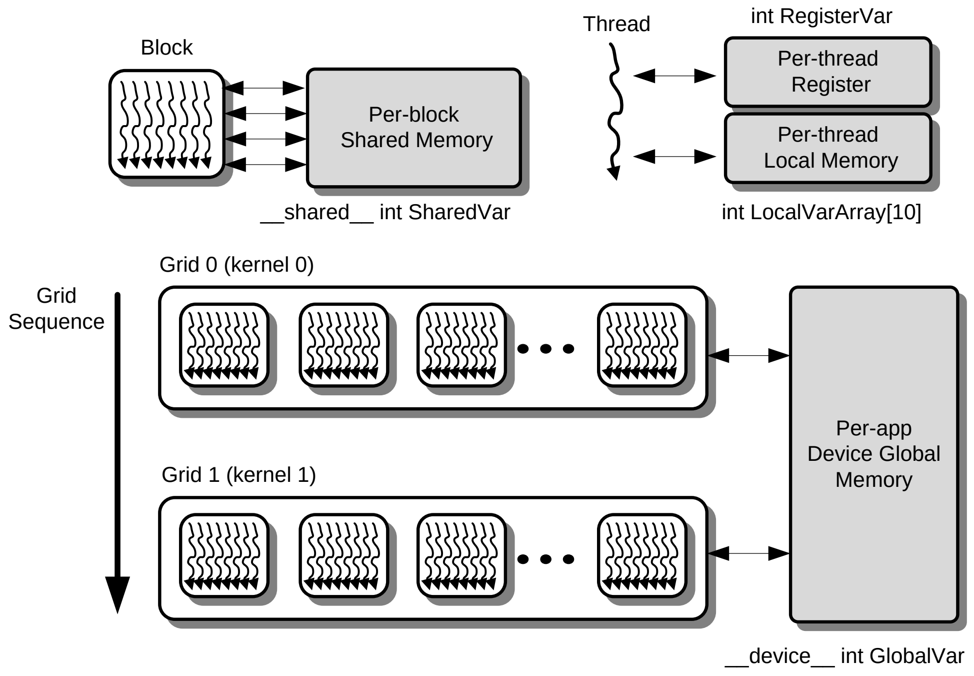


Figure 5.2: *CUDA/GPU Execution Model*

5.2 CUDA and GPUs

The CUDA programming model is a multi-threaded SIMD model that enables implementation of general purpose programs on heterogeneous GPU/CPU systems. There are two different device types in CUDA: the Host processor and the GPU. A CUDA program consists of a host code segment that contains the sequential sections of the program, which is run on the CPU, and a parallel code segment which is launched from the host onto one or more GPU devices. Data-level parallelism (DLP) and thread-level parallelism (TLP) are handled differently in these systems. DLP is converted into TLP and executed onto the GPU devices, while TLP is handled by executing multiple kernels on different GPU devices launched by the host processor. The threading and memory abstraction of the CUDA model is shown in Figure 5.2.

The threading abstraction in CUDA consists of three levels of hierarchy. The basic block of work is a single *thread*. A group of threads executing the same code are combined together to form a *thread block* or simply a *block*. Together, these thread blocks combine to form the parallel segments called *grids* where each grid is scheduled onto a GPU at a time. Threads within a thread block are synchronized together through a barrier operation ($\_\_syncthreads()$). However, there is no explicit software or hardware support for synchronization across thread blocks. Synchronization between thread blocks is performed through the global memory of the GPU, and the barriers needed for synchronization are handled by the host processor. Thread blocks communicate by executing separate kernels on the GPU.

The memory abstraction in CUDA consists of multiple levels of hierarchy. The lowest level of memory is *registers*, which are on-chip memories private to a single thread. The next level of memory is *shared memory*, which is an on-chip memory shared only by threads within the same thread block. Access latency to both the registers and shared memory is extremely low. The next level of memory is *local memory*, which is an off-chip memory private to a single thread. Local memory is mainly used as spill memory for local arrays. Mapping arrays to shared memory instead of spilling to local memory can provide much better performance. Finally, the last level of memory is *global memory*, which is an off-chip memory that is accessible to all threads in the grid. This memory is used primarily to stream data in and out of the GPU from the host processor. The latency for off-chip memory is 100-150x more than that for on-chip memories. Two other memory levels exist on-chip called the *texture memory* and *constant memory*. Texture memory is accessible through special built-in texture functions and constant memory is accessible to all threads in the grid.

The CUDA programming model is an abstraction layer to access GPUs. NVIDIA GPUs use a single instruction multiple thread (SIMT) model of execution where multiple thread blocks are mapped to streaming multiprocessors (SM). Each SM contains a number of processing elements called Streaming Processors (SP). A thread executes on a single SP. Threads in a block are executed in smaller execution groups of threads called *warps*. All threads in a warp share one program counter and execute the same instructions. If conditional branches within a warp take different paths, called *control path divergence*, the warp will execute each branch path serially, stalling the other paths until all the paths are complete. Such control path divergences severely degrade the performance.

Because off-chip global memory access is very slow, GPUs support *coalesced memory accesses*. Coalescing memory accesses allows one bulk memory request from multiple threads in a half-warp to be sent to global memory instead of multiple separate requests. In order to coalesce memory accesses, three general restrictions apply: each thread in a half-warp must access successive addresses in order of the thread number, the memory accesses can only be 32, 64, or 128-bit, and all the addresses must be aligned to either 64, 128 or 256-byte boundaries. Effective memory bandwidth is an order of magnitude lower using non-coalesced memory accesses which further signifies the importance of memory coalescing for achieving high performance.

In modern GPUs, such as NVIDIA GTX 285, there are 30 SMs each with 8 SPs. Each SM processes warp sizes of 32 threads. The memory sizes for this GPU are: 16K of registers per SM, 16KB divided into 16 banks of shared memory per SM, and 2GB of global memory shared across all threads in the GPU.

We use the StreamIt programming language to implement streaming programs. StreamIt is an architecture-independent streaming language based on SDF. The language allows a programmer to algorithmically describe the computational graph. In StreamIt, actors are known as filters. Filters can be organized hierarchically into *pipelines* (i.e., sequential composition), *split-joins* (i.e., parallel composition), and *feedback loops* (i.e., cyclic composition). StreamIt is a convenient language for describing streaming algorithms, and its accompanying static compilation technology makes it suitable for our work.

5.3 Portable Stream Compilation

Sponge takes StreamIt programs as its input and generates GPU-specific CUDA code. Each actor in the StreamIt graph is converted to a CUDA kernel running with some number of threads and blocks. By performing portable stream compilation, Sponge decides how many threads and blocks to assign to the CUDA kernel generated for each actor. The input buffer size of the first actor, A_i, in the graph determines how many times that actor has to run (R_i). As a result, R_i is changed to R_i divided by the multiplication of number of threads and blocks assigned to the actor. We call the result *number of iterations* for actor

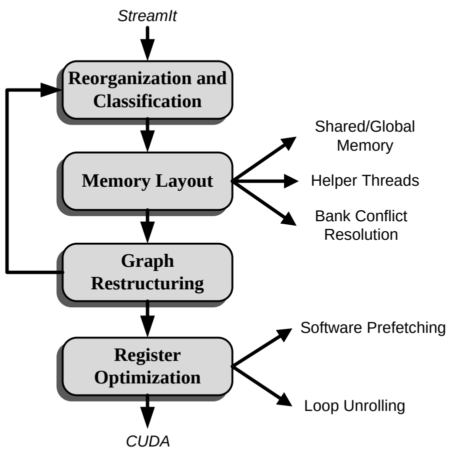


Figure 5.3: *Compilation flow in Sponge.*

$$\begin{aligned}
ActiveWarpsPerSM &= \frac{ThreadsPerBlock \times ActiveBlocksPerSM}{THREADS\_PER\_WARP} && (5.1)\\
Iterations &= \frac{InputBufferSize}{Pop \times ThreadsPerBlock \times Blocks} && (5.2)\\
ThreadsPerBlock_{LoT} &= \frac{SHARED\_MEMORY\_SIZE}{(Pop + Push)} && (5.3)\\
ExecCycles_{LoT} &= \frac{CompInsts \times COMP\_INST\_ISSUE\_DELAY}{NUMBER\_SM} \times \frac{ThreadsPerBlock_{LoT} \times Iterations}{ActiveWarpsPerSM} && (5.4)
\end{aligned}$$

$$\begin{aligned}
ThreadsPerBlock_{HiT} &= MAX\_THREAD\_PER\_BLOCK && (5.5)\\
MemCycles &= (UncoalMemInsts + CoalMemInsts/COAL\_FACTOR) && \\
&\quad \times MEMORY\_DELAY + MEM\_INST\_ISSUE\_DELAY && (5.6)\\
ExecCycles_{HiT} &= \frac{MemCycles}{NUMBER\_SM} \times \frac{ThreadsPerBlock_{HiT} \times Iterations}{ActiveWarpsPerSM} && (5.7)
\end{aligned}$$

Name	Description
$SHARED\_MEMORY\_SIZE$	Size of shared memory on GPU
$THREADS\_PER\_WARP$	Number of threads in each warp
$NUMBER\_SMs$	Number of streaming processor on GPU
$MAX\_THREAD\_PER\_BLOCK$	Max number of threads allowed per block
$MEMORY\_DELAY$	Number of cycles to access global memory
$COAL\_FACTOR$	Max number of memory accesses that can be coalesced
$MEM\_INST\_ISSUE\_DELAY$	Number of cycles to issue a memory instruction
$COMP\_INST\_ISSUE\_DELAY$	Number of cycles to issue a compute instruction
$pop, push$	push and pop rate of an actor
$InputBufferSize$	Size of input buffer for an actor
$ThreadsPerBlock$	Number of threads in one block
$Blocks$	Number of blocks on the GPU
$Iterations$	Number of iterations to run an actor on the GPU
$ActiveBlocksPerSM$	Blocks active on one SM
$(Un)CoalMemInsts$	(Un)Coalesced instructions in one actor

Figure 5.4: *In this Figure, equations for calculating execution cycles of both HiT and LoT actors are shown. Equations 5.1 and 5.2 can be used for both HiT and LoT actors. The table summarizes what each variable means.*

A_i.

Portable stream compilation in *Sponge* consists of four main steps as shown in Figure 5.3. In the first phase, Sponge reads a StreamIt program and performs *Actor Reorganization and Classification* in which simple graph reorganization is done and actors are classified into two categories: *High-Traffic (HiT) and Low-Traffic (LoT)*. The classification information is used throughout all the phases of the compilation flow. The second phase deals with the *Memory Layout and Optimization* of each actor. This step decides if an actor uses shared or global memory, eliminates shared memory bank conflicts and also improves memory performance by introducing *Helper Threads* to better utilize the unused processors and bring the data needed by an actor into shared memory faster. This compilation step is crucial to achieving better performance since memory bandwidth can be a limiting factor on GPUs. The third phase performs *Graph Restructuring* by changing the granularity of the kernels and vertically fusing actors based on classification results. After graph restructuring, the compiler reiterates from the beginning of the compilation flow, treating the post-fused stream graph as the input until no more graph restructuring is possible. Finally, *Register Optimization* tries to utilize unused registers on each SM by employing software prefetching and also by unrolling *for loop*s in each kernel.

5.3.1 Actor Reorganization and Classification

As mentioned in Section 5.2, GPUs are built for data-level parallelism and are not suitable for task-level parallelism and global synchronization. Therefore, *splitter-joiner* structures will not perform well on the GPU since each joiner introduces a synchronization point. First, Sponge collapses *splitter-joiner*s to one actor in cases that the actors between the *splitter* and *joiner* are stateless and equivalent. This will remove the *splitter* and *joiner* actors and replace the structure with a single actor. In cases where it is not possible to collapse a *splitter-joiner* structure to one actor, Sponge treats the *splitter* and *joiner* as special actors with more than one input and output. Based on the type and weights of the *splitter* and *joiner* actors, Sponge decides to allocate their input and output buffers in shared memory or global memory.

Sponge excludes stateful actors from being executed on the GPU and runs them on the host CPU. This is because only one instance of a stateful actor can be active and data-parallelism is not applicable to these actors. Host to GPU and GPU to host transfers are inserted before and after stateful actors, if necessary.

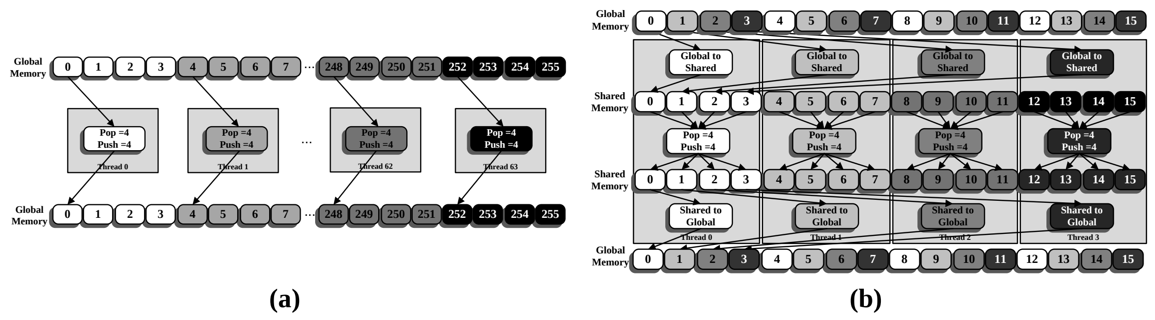


Figure 5.5: *This figure shows how HiT and LoT threads access their buffers. Part (a) illustrates the memory access pattern for a sample HiT actor with four pops and four pushes. Part (b) shows the access pattern for a LoT actor.*

Next, Sponge classifies actors assigned to the GPU as either High-Traffic (HiT) or Low-Traffic (LoT). HiT actors have a large number of memory accesses. These actors perform better on a GPU if their buffers are mapped to global memory rather than shared memory because mapping the buffers to shared memory will result in having too few threads and under-utilizing the processors and the available memory bandwidth. LoT actors, on the other hand, are mostly computation dominated and if mapped to shared memory will have a reasonable number of threads to utilize the GPU.

In order to determine if an actor is a LoT or HiT, Sponge estimates execution cycles of an actor for both global memory (HiT) and shared memory (LoT) mappings, based on Equations 5.1-5.7 in Figure 5.4. For each actor, Sponge treats that actor as both HiT and LoT and calculates the corresponding execution cycles ($ExecCycles_{LoT}$, $ExecCycle$-s_{HiT}). The two numbers show if that actor is suitable to be treated as a LoT or HiT actor. If $ExecCycles_{HiT}$ is smaller than $ExecCycles_{LoT}$ for an actor, that actor will perform better if its buffer is mapped to global memory. Otherwise, it will be classified as a LoT actor for which both shared and global memory will be used to help with coalescing of data accesses.

In the equations for LoT actors, number of threads per block ($ThreadsPerBlock$) is determined by the size of shared memory($SHARED\_MEMORY\_SIZE$ [1]) and the number of pushes and pops. Threads per block defines the number of active warps per SM ($ActiveWarpsPerSM$) and the number of iterations based on Equations 5.1 and 5.2. Finally, the execution cycle of a LoT actor is estimated depending on the number of compute instructions and the distribution of threads in the GPU (Equation 5.4).

Execution time of HiT actors is calculated based on their memory access time because these actors are mapped to global memory and have a large number of global memory reads and writes. Equations 5.5-5.7 show how execution time estimation is done based on the number of coalesced and uncoalesced memory accesses. Unlike shared memory, the size of global memory does not limit the number of threads. Therefore, the number of threads per block for HiT actors can be equal to the maximum number of threads allowed in each block ($MAX\_THREAD\_PER\_BLOCK$).

In this section, memory layout and optimization techniques used in Sponge are discussed. First, the way shared memory is utilized for LoT actors is explained. Second, helper threads, a technique that Sponge uses to reduce global memory access time of actors, is discussed. Finally, shared bank conflict resolution in Sponge is explained.

5.3.1.1 Shared/Global Memory

To deal with high-latency memory access issues, Sponge uses the classification information calculated in the previous phase and tries to alleviate the problem by coalescing the buffer accesses or overlapping a large number of uncoalesced buffer accesses to amortize the cost. As discussed earlier, HiT actors will be mapped to global memory and LoT actors

[1]Variables with all capital characters show GPU-specific parameters

will use the shared memory. The kernel generated for these actors will have a large number of threads, each accessing its own buffer sequentially in global memory. The memory accesses will not be coalesced because the accesses of consecutive threads are not consecutive in the memory. Since the number of threads is large, the overhead of memory accesses will be hidden by the execution of many threads. Figure 5.5a illustrates how a HiT actor with four pops and four pushes accesses global memory. Since the addresses generated by the first pop operations of the threads are not consecutive in the memory, they are not coalesced.

LoT actors, unlike HiT actors, have a higher compute to memory ratio. Therefore, a LoT actor can use shared memory and have a large number of threads. As shown in Figure 5.5b, threads of a LoT kernel in a block can use coalesced memory accesses to copy their input (output) buffer to (from) shared memory from (to) global memory. To do so, the threads of a block work as a group and bring parts of data that belong to other threads as well as part of their own data. In this way, consecutive threads' accesses to shared memory will be to consecutive locations and will get coalesced. Since all of the data is in shared memory, all threads in a block will have access to it. Figure 5.6 shows how the CUDA code needs to be changed to utilize shared memory in LoT actors. In the baseline form (Figure 5.6a) the input and output buffers are allocated in global memory and the work function directly accesses global memory. If shared memory is used, then two *for loop*s are added before and after the work function to copy the data in and out of shared memory, as shown in Figure 5.6b. The addresses for the memory reads and writes in these *for loop*s are set based on the $ThreadID$, and the number of pushes and pops. Before and after the two new *for loop*s, L_1, L_2, barriers (*synchthreads*) are necessary because, as mentioned earlier,

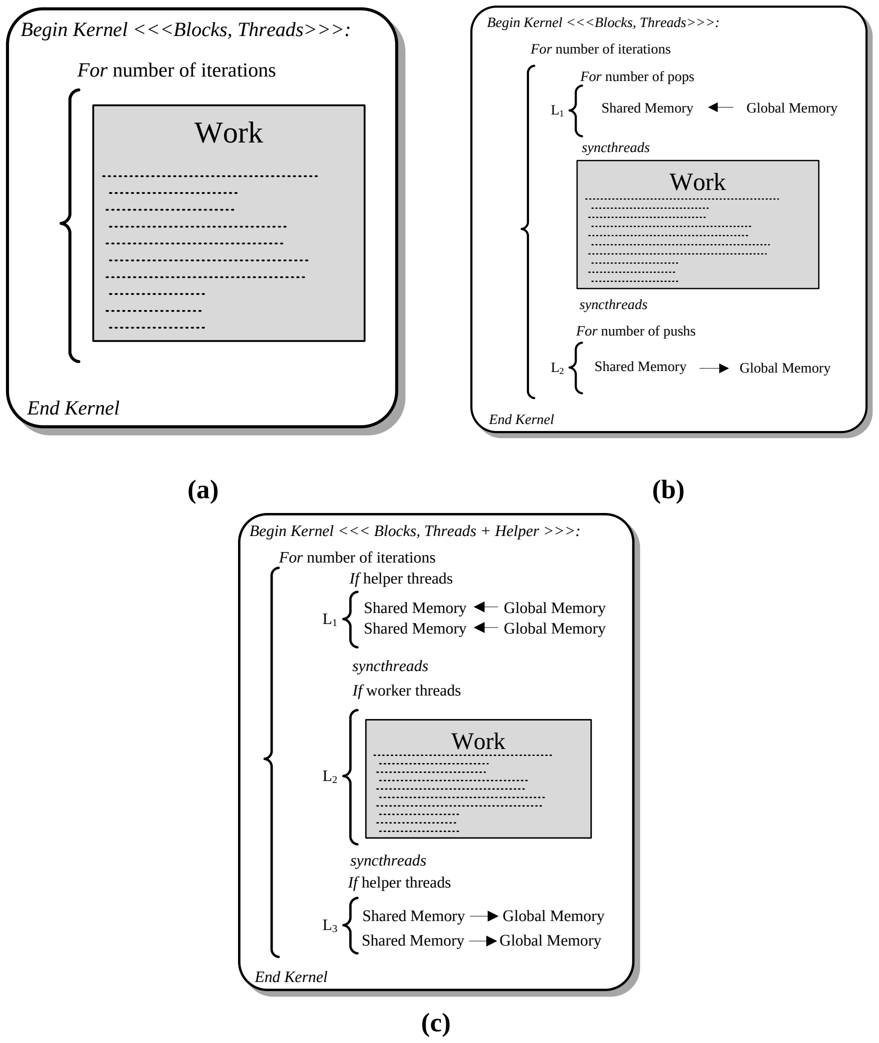


Figure 5.6: *Part (a) shows the baseline translation for a HiT actor. How shared memory is used in a LoT actor is illustrated in part (b). In part (c) the way Sponge generates CUDA code to divide threads as helpers and workers is shown.*

each thread does not fetch all of its own data and has to wait for other threads in the block to finish their data-fetch phase.

5.3.2 Memory Layout and Optimization

Memory hierarchy in GPUs is significantly different from both conventional shared memory and distributed memory systems. As mentioned in Section 5.2, efficient use of global memory, shared memory and registers on GPUs is crucial to obtain high performance. Coalescing accesses to global memory can greatly reduce memory access overheads, but it will not be possible without careful memory layout. Utilizing shared memory, which is significantly faster than global memory, is also very beneficial. Due to its limited size, shared memory can restrict the number of threads and degrade the performance. In this section, techniques used for memory layout and optimization in Sponge are discussed.

5.3.2.1 Helper Threads

The first optimization of this phase is to use $helper$ threads to fetch data for the $worker$ threads. In cases where there are not enough threads to efficiently utilize all the SMs for LoT kernels or a HiT actor has a fair number of threads when it is treated as a LoT actor (mapped to shared memory), Sponge uses helper threads to reduce the buffer (i.e. memory) accesses of each thread (push and pop rate). Each helper thread aids some worker threads to bring their data to shared memory in a coalesced way.

Figure 5.6c shows how the CUDA code is modified. Based on the thread IDs, Sponge generates the helper and worker threads. Helper threads are in charge of handling the data accesses and worker threads are in charge of the computation. In order to avoid control flow divergence, the thread assignment is performed such that the helper and worker threads form complete warps. If the number of worker threads are less than the warp size, then the helper threads are placed in the first set of warps and the worker threads form the last warp. This is done by predicating out the work function for the helper threads and the memory access *for loop*s for the worker functions. The *if* statement in Figure 5.6c does this based on $threadID$. This technique works because control flow divergence negatively affects the performance within one warp but not across warps.

Sponge estimates number of instructions that helper threads will add to each thread and also takes into account the parallelism between the helper and worker threads to calculate how beneficial helper thread optimization will be for both LoT and HiT threads. As illustrated in Figure 5.6c, Sponge counts the time it takes to run L_1, L_2, and L_3 sections and estimates the total execution time based on the equations in Figure 5.4. If the total execution time using helper threads is reduced, Sponge generates CUDA code using them.

5.3.2.2 Bank Conflict Resolution

Shared memory bank conflict is another source of bottleneck in GPU systems. For example, whenever threads of a kernel access their input buffer in shared memory with:

$$data \quad = \quad buffer[baseAddress + s * threadId];$$

threads $threadId$ and $threadId + n$ access the same bank whenever n is a multiple of m/d (m is the number of memory banks) where d is the greatest common divisor of m and s. As a consequence, there will be no bank conflicts only if half the warp size is less than or equal to m/d. For current NVIDIA devices, this translates to no bank conflict only if d is equal to 1, or in other words, only if s is odd since m is a power of two (16 for GTX 285 [65]). In the StreamIt code, s is the number of pops. To make s odd, if the number of pops is even, Sponge artificially changes the pop rate of an actor by incrementing the pops by one. In this way, an actor with $2k$ pops will use $2k + 1$ entries in the memory and the buffers get shifted in the memory. Removing bank conflicts greatly improves the performance of some of the benchmarks. The same technique can be applied for pushes.

5.3.3 Graph Restructuring

In this part, Sponge vertically fuses some actors to improve performance by increasing coalesced memory accesses, removing kernel call overhead, and also increasing instruction overlap. Fusion is not beneficial in all cases because it can increase the memory traffic (push + pop) of a pair of LoT actors and reduce the number of threads (Equation 5.3). For HiT actors, fusion may increase the memory traffic as a result of register spilling.

The main benefit of fusing HiT actors is replacing uncoalesced memory accesses at the end of the first actor and at the beginning of the second actor with coalesced accesses. The memory accesses become coalesced because the two actors within the fused actor are rate matched. Therefore, the first actor can write to the internal buffer using coalesced memory writes and the second actor can read the same data with coalesced memory reads. Figure 5.7 illustrates how fusion can lead to coalescing of memory accesses in a simple GPU that has warp size of four and can coalesce two memory accesses into one. In this figure, the memory accesses between actors A (producer with 2 pushes running with 8 threads) and B (consumer with 8 pops running with 2 threads) are shown. $W_{i,j}$ is jth push by the ith thread of A, and $R_{k,m}$ is the mth pop of the kth thread of B. Figure 5.7a shows how writes and reads are performed between these actors in the case of no fusion. In this case each thread serially writes and reads from global memory which results in all uncoalesced accesses (marked by U). If the buffer allocation for A is changed such that its memory accesses can be coalesced (marked by C), as shown in Figure 5.7b, the accesses of threads running B will still be uncoalesced. Figure 5.7c shows the accesses to the internal buffer between A and B after fusion is performed. The new actor, $(4A)B$, runs with two threads. Since there are 8 pushes and pops between $4A$ and B all the accesses will be coalesced, as shown in Figure 5.7c.

For the LoT actors, global memory accesses are already coalesced with the help of shared memory. These accesses happen in two *for loop*s before and after the work function. Similar to the HiT case, the accesses between the two LoT actors become coalesced. Therefore, the resulting LoT actor does not need to use shared memory anymore. This will result in elimination of a large number of complex address calculations and *for loop* control instructions.

Sponge uses its cost estimation equations to decide if fusing a pair of actors is beneficial or not. For a candidate pair, Sponge calculates the number of cycles for both cases where the resulting actor is HiT or LoT. If in either case the execution time is less than the sum of the original actors' execution times, fusion is performed.

5.3.4 Register Optimization

Registers on GPUs are a precious resource. Efficiently using the registers can greatly improve performance. In this section, two optimizations that Sponge performs to increase register utilization are discussed.

5.3.4.1 Software Prefetch

To better tolerate long memory access latency, the CUDA threading model allows some warps to make progress while others wait for their memory access results. This mechanism

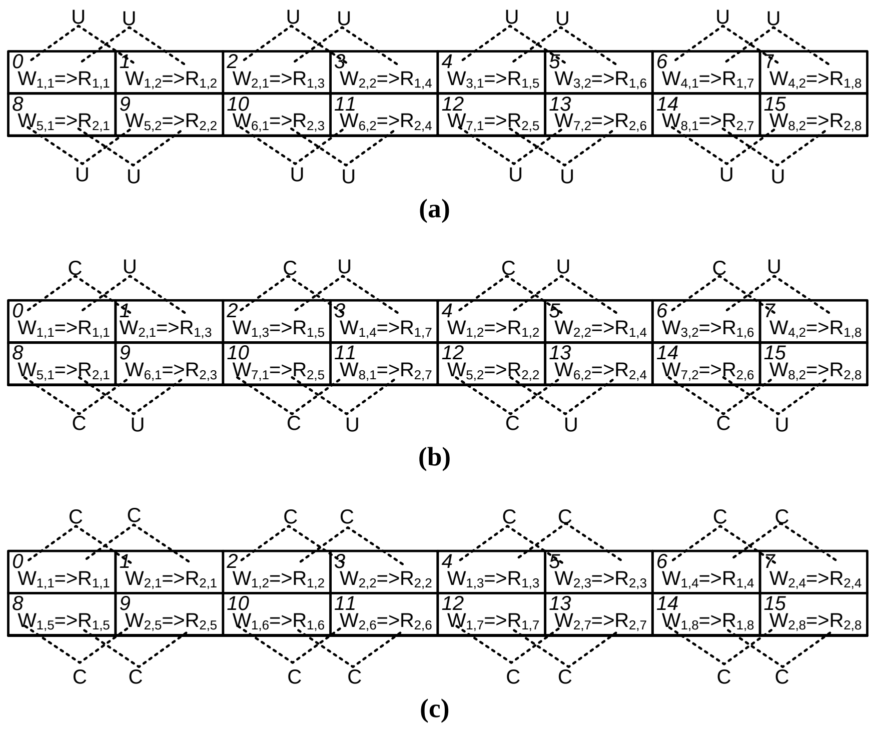


Figure 5.7: *This figure shows the memory accesses between actors A with 2 pushes and 8 threads and B with 8 pops and 2 threads. $W_{i,j}(R_{i,j})$ shows jth memory write (read) performed by ith thread running actor A (B). U and C denote uncoalesced and coalesced. Part (a) shows the accesses in the base case. Part (b) illustrates the same accesses when the buffer for A is allocated such that its writes are coalesced. Part (c) shows coalesced accesses between these two actors when they are fused as $(4A)B$ and executed with two threads. The number on the top left corner of each box shows the memory address of that location.*

is not effective in some cases where all threads are waiting for their memory access results. This case happens if all threads have very few independent instructions between memory access instructions and the use of the accessed data. Prefetching is a technique that some CUDA programs use to overlap fetching data from global memory for iteration $i + 1$ of an actor with compute instructions in iteration i by utilizing the available registers.

Figure 5.8a shows how software prefetching can be done for LoT actors. Before the main *for loop*, the first batch of data (for iteration 1) is loaded into registers (L_1). Once L_2 has started, the data is moved from registers into shared memory. At this point, threads have to wait for the shared memory transfers to finish before they can progress because that data is needed for computation after this point. After all threads are done moving the data to shared memory, they pass the barrier synchronization point and begin to load the next batch of data into registers. The key is that the work function does not need the data from these memory accesses and overlapping of compute and memory accesses can happen. Loop L_4 has to be wrapped in an *if* statement because the last iteration of the kernel does not need to prefetch any data. This *if* statement does not introduce branch divergence since all the threads take the same path at this point.

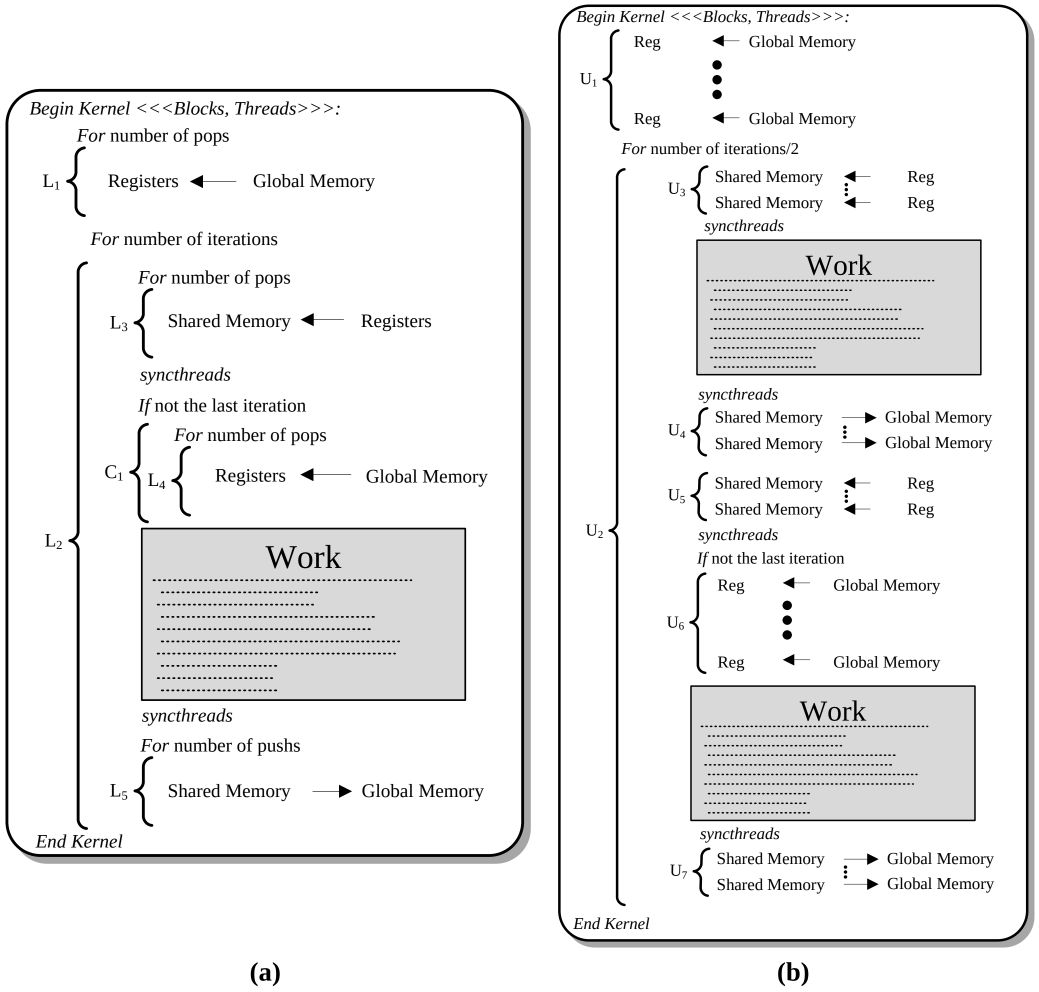


Figure 5.8: *Part (a) shows how prefetching is performed to improve the performance of a kernel. Part (b) depicts the result of unrolling on the kernel in part (a).*

One possible downside of this technique is that using additional registers for prefetching can reduce the number of blocks that can run on an SM. However, prefetching is beneficial if it significantly reduces the amount of time each thread waits for global memory accesses. Since different classes of NVIDIA GPUs are equipped with different number of registers, Sponge tunes this optimization for each GPU target. If performing prefetching for the whole buffer introduces register spill or reduce the number of concurrent blocks, Sponge tunes the prefetching optimization by applying it to only a fraction of the input buffer.

5.3.4.2 Loop Unrolling

Instruction processing bandwidth on the processing cores of current CUDA graphics engines can negatively affect the performance of an actor. Address calculation and loop control instructions can become important if an actor has small number of computation instructions. In other words, these type of instructions introduce overhead and prevent a kernel from utilizing the peak performance of a GPU. Loop unrolling is one way to reduce the overhead. This optimization can also increase the register utilization by unrolling loops that use registers. The degree of unrolling depends on the number of registers the kernel uses and also the number of registers that are available on the GPU. Since different classes of GPUs are equipped with different number of registers per SM, blindly applying unrolling to the *for-loop*s in a kernel may worsen the performance.

An example of the unrolling is shown in Figure 5.8b. There are five potential *for loop*s in a typical LoT actor generated by Sponge, as shown in Figure 5.8a, 2 for transferring data to and from global memory, 2 for prefetching and 1 for the work function. Depending on the number of registers available on the target GPU and the instruction mix of the kernel, Sponge decides to perform loop unrolling on the *for loop*s.

Figure 5.8b shows how the unrolling is applied to all five *for loop*s to both remove the *for loop* overheads and also increase the register utilization. In this example, unrolling factor of two is applied to the *work* function. As shown in Figure 5.8a, loop L_1 is unrolled to U_1. Because the work function is unrolled two times, all the corresponding *for loop*s now appear twice except L_4. Replicating L_4 two times will result in having two *if* statements. To remove the conditional branch instruction overhead, these two replicated *for loop*s are merged into U_6.

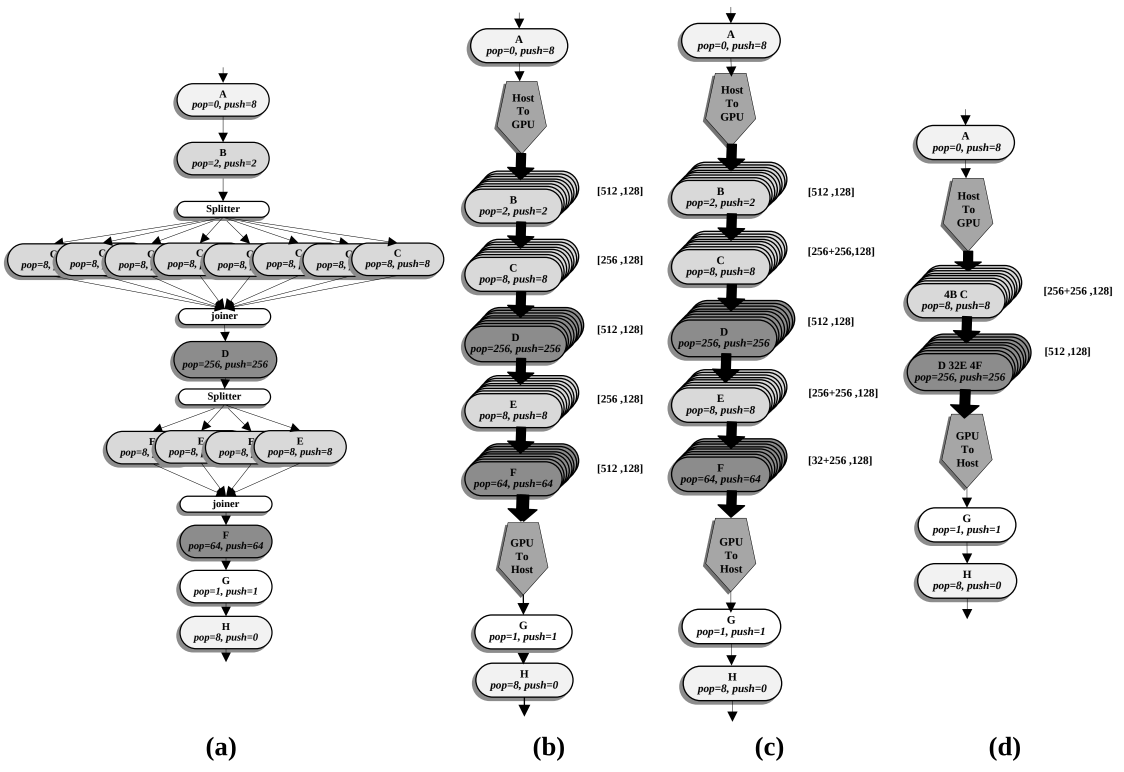


Figure 5.9: *Part (a) shows a stream graph with 12 unique actors. Part (b) is about how actor classification and graph reorganization affects this graph. In this part, shaded actor are HiT actors. Part (c) illustrates the result of the helper thread optimization. Part (d) depicts the same graph after applying graph restructuring. [i, j] next to each GPU actor shows number of threads (i) and number of blocks (j) that will run that actor. If i is written as $w + h$, w is number of worker threads and h is the number of helper threads.*

5.3.5 A Stream Compilation Example

In this section, a running example, as shown in Figure 5.9, is used to better illustrate how the optimizations affect the streaming graph. The base graph in Figure 5.9a has 12 unique actors two of which are in a *splitter-joiner* structure. Each box shows one actor in the program. Each edge in this graph indicates a tape implemented using FIFO queues. The text written inside each box shows how each actor interacts with its input and output tapes. All the actors are stateless except G. This actor as well as the source (A) and the sink (H) actors are mapped to the host processor.

In the classification phase, Sponge will remove the two *splitter*s and *joiner*s and replace all the copies of C and E with one of each. This is done because GPUs do not support task level parallelism and the *joiner* will introduce synchronization overhead. After this, actors are classified as HiT and LoT based on their memory traffic and computation instructions. LoT actors use shared memory but HiT actors operate on global memory. In the example, actors D and F are identified as HiT actors (shown with a darker color) and B, C and E as LoT actors. $[i,\ j]$ next to each GPU actor shows number of threads (i) and number of blocks (j) that will run that actor. For the LoT actors, the number of threads depends on the size of shared memory and the memory usage of the actor. For HiT actors, the number of threads is always equal to the maximum number of threads allowed per block because global memory is significantly larger than the actors' memory footprint.

Next, helper threads are used to fetch data from global memory to shared memory more efficiently. After applying this, the number of threads for LoT actors will increase but HiT actors will be running with less threads because they have to use the shared memory. In the example, as shown in Figure 5.9c, except actor D, every actor benefits from using helper threads. If the number of threads for an actor is written as $w + h$, then w shows the number of worker threads and h shows number of helper threads assigned to that actor.

Finally, graph restructuring is performed on the graph and as a result several actors get fused together. Figure 5.9d shows the result of fusion and then re-applying classification and helper thread optimization. Actor B and C are fused together in a LoT actor and actors D, E, and F are classified as a HiT actor.

5.4 Experiments

In this section Sponge's optimization techniques are evaluated and compared with two alternative approaches:

1. GPU baseline: All stateless actors of the benchmarks are mapped to the GPU utilizing the maximum number of threads supported ($MAX\_THREAD\_PER\_BLOCK$). In this technique, all of the actors are compiled as HiT actors. Stateful actors as well as source and sink actors are mapped to the host processor.

2. CPU baseline: All the actors are executed sequentially on the CPU.

5.4.1 Methodology

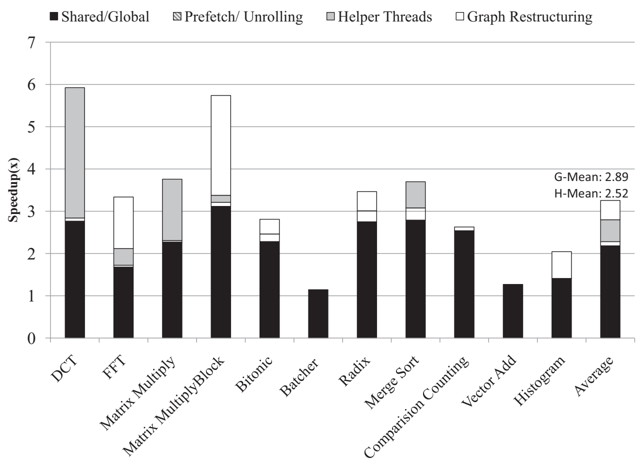
A set of benchmarks from the StreamIt suite [79] are used to evaluate Sponge. The benchmarks are compiled and evaluated on a system containing a 3GHz Intel Core 2 Extreme CPU with 6GB of RAM and a GeForce GTX 285 GPU with 2GB DDR3 global memory. Sponge compilation phases are implemented as a compiler backend to the StreamIt compiler. Sponge generates customized CUDA code which is compiled using NVIDIA nvcc 3.1 for execution on the GPU. GCC 4.1 is used to generate the x86 binary for execution on the host processor.

5.4.2 Techniques Performance

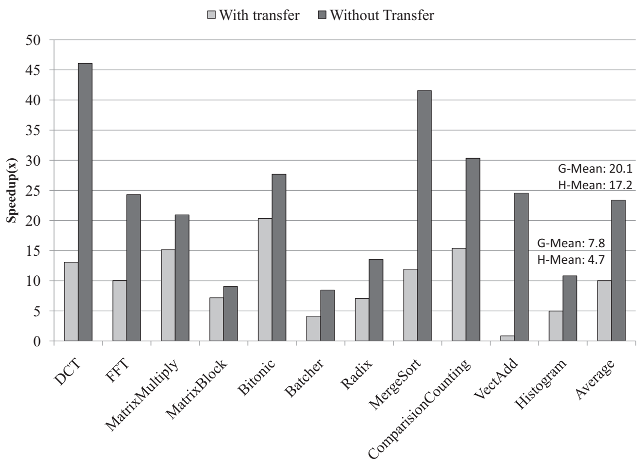
In this section, we try to compare the Sponge optimization techniques to the GPU baseline and highlight the effectiveness of each optimization. Figure 5.10a shows how Sponge-generated CUDA code performs and shows the performance gain of each optimization technique. On average, Sponge improves the performance by 3.2x compared to the GPU baseline.

The first optimization, shared/global memory, which divides actors into two categories LoT and HiT, is one of the most beneficial Sponge techniques. By using shared memory, Sponge is able to coalesce all the memory accesses in LoT actors, therefore performance of benchmarks containing LoT actors will significantly increase. As shown in the Figure 5.10a, *Matrix Multiply Block* benefits the most because this benchmark has several LoT actors. As a result, most of the actors in *Matrix Multiply Block* have coalesced memory accesses. In some benchmarks, such as *Histogram*, little benefit is seen using this optimization because most actors are HiT actors.

Prefetching and unrolling are two other optimizations illustrated in Figure 5.10a. These optimizations, collectively, contribute to 3.1% of the total average speedup. Prefetching technique is used only for LoT actors and is useful mostly in applications with many LoT actors such as *Merge sort* and *Bitonic*. Unrolling allows Sponge to utilize unused registers and reduce the number of instructions. This technique can increase the performance of LoT actors that use few registers. *DCT*, *Merge Sort*, *Radix*, and *Bitonic* have such actors and unrolling can increase their performance.



(a) Performance breakdown of Sponge optimizations in comparison to the baseline CUDA code, both running on the GPU.



(b) Speedup of Sponge optimized code in comparison to the host CPU with and without data transfer overhead.

Figure 5.10: *Effectiveness of Sponge optimization techniques on StreamIt benchmarks.*

Another effective optimization in Sponge is employing helper threads. As described in the previous sections, helper threads can reduce the execution time of both LoT and HiT actors with two exceptions:

- LoT actors with many threads: In this case, it is not possible to run more threads

GTX 285											
	DCT	FFT	MM	MM Block	Bitonic	Batcher	Radix	Merge Sort	Comp Count	Vect Add	Histogram
Shared	66.7	66.7	62.5	80	100	100	100	100	100	100	33.3
Prefetch	0	4.2	12.5	0	40.7	0	0	34.8	13	0	0
Unrolling	50	70.8	37.5	80	50.7	100	100	30.4	63.2	0	33.3
Helping Threads	50	16.7	25	20	0.7	0	0	52.1	0	0	0
Tesla C2050											
Shared	100	100	87.5	93.3	100	100	100	100	100	100	33.3
Prefetch	0	41.7	12.5	6.7	1.5	0	100	0	100	0	66.7
Unrolling	50	33.3	50	60	34.4	100	0	0	0	0	0
Helping Threads	50	0	25	6.7	0.7	0	0	0	0	0	33.3

Table 5.1: *This table shows how Sponge optimizes each benchmark differently for two GPU targets. For each benchmark and target, the percentage of actors that are optimized by each optimization is shown.*

to help the worker threads. Reducing the number of worker threads would decrease performance.

- HiT actors with few threads: To utilize helper threads, HiT actors would be converted into LoT actors, which have less threads because of the limited shared memory size. Though transferring data to shared memory improves memory performance, too few worker threads can become a bottleneck, under-utilizing the SMs and decreasing the overall performance.

As shown in Figure 5.10a, helper thread optimization effectively increases the performance of *DCT*, *FFT*, *Matrix multiply* and *Merge sort*. For example *DCT* has multiple HiT actors with a large number of worker threads. In this case, coalescing data accesses using shared memory provides enough performance gain that running the actors with less threads will not result in slowdown. On average, helper threads contributes to 16% of the total average speedup compared to the GPU baseline.

Graph restructuring decreases the overhead of kernel launching and uncoalesced memory accesses. As discussed in Section 5.3.3, there are some cases where fusing two actors may result in degraded performance. Since two actors that are fused must execute together, the number of threads that the resulting actor can run will be less than the number of threads created by running each actor separately. Because the reduction of threads can decrease the performance, Sponge intelligently decides whether or not to use this optimization. Several benchmarks, such as *FFT*, have large pipelines of actors that are all fused together by Sponge. Graph restructuring provides a large portion of the speedup for these types of benchmarks. Since *Batcher* and *Vector Add* have only one actor, fusion cannot increase their performance. In *Merge Sort*, the opportunity for performing fusion is minimal because most of the actors in this benchmark are isolated from each other and do not form a pipeline.

5.4.3 Overall performance

Figure 5.10b presents the speedup of Sponge's generated CUDA applications against the CPU baseline, both with and without the data transfer time between the GPU and CPU. On average, Sponge achieves about 20x speedup compared to running each benchmark completely on the CPU. The only case that the CPU baseline outperforms Sponge is *Vector Add* including the data transfer overhead. In this special case, the memory to compute ratio in *Vector Add* is very high. Although the GPU can execute the *Vector Add* actor 10x faster than CPU, the overhead of transferring the data between the host and GPU global memory decreases the overall performance.

5.4.4 Portability

Quantifying portability is inherently a hard problem. To show how Sponge solves the portability issue, we show how it optimizes each benchmark differently for two GPUs, Tesla C2050 and GeForce GTX 285. The C2050 is based on the newer NVIDIA architecture (Fermi) which has 48KB of shared memory, 32K registers and 420 streaming processors, providing Sponge with more resources to exploit. Table 5.1 shows how Sponge makes different decisions based on the target architecture. This table illustrates the percentage of actors in each benchmark optimized using various optimizations in Sponge. As shown in Table 5.1, Sponge is able to classify more actors as LoT actors and utilize the larger shared memory in C2050. The number of registers also affects how Sponge performs unrolling and prefetching for each target. In general, Sponge adopts its compilation strategy based on the characteristics of the GPU target without any source code modification or programmer involvement.

5.5 Case Study and Future Work

Sponge is designed to reduce the performance gap between automatically generated CUDA programs and hand-optimized ones. In this section two hand-optimized CUDA programs from the NVIDIA SDK are analyzed to highlight the reasons for performance differences between Sponge-generated and hand-optimized CUDA code.

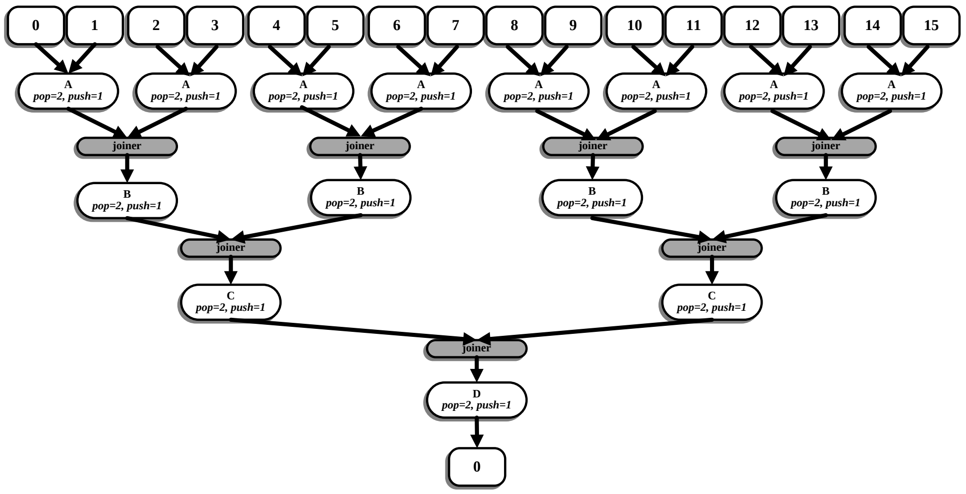


Figure 5.11: *This graph shows the stream graph of a generic stream reduction kernel.*

5.5.1 Black-Scholes

The *Black-Scholes* algorithm is a differential equation that can predict how the value of an option changes. This equation reads five parameters from the input data and computes the price for an option call and an option put and writes these two values to the output. In the code generated by Sponge for GTX 285 GPU, there is only one kernel that pops five memory element from the input and calculates the output and pushes two results to the output buffer. This actor is classified as an LoT actor. Therefore, Sponge uses shared memory to coalesce all the buffer accesses in that actor. In the hand-optimized code, only one kernel is launched as well, but each parameter is placed in a different array. The kernel has five input arrays and two output arrays. By using this technique, all threads are able to read data from each input array and write data to each output array consecutively allowing all memory accesses to be coalesced. Coalescing all accesses without using shared memory reduces the number of instructions in the hand-optimized version. As a result, the performance of the hand-written program is 1.3x better than Sponge's generated code.

This input/output buffer re-mapping is not currently done in Sponge because StreamIt does not support actors with multi-inputs and multi-outputs streams. All input and output streams between StreamIt actors are through a single shared buffer between the actors. Future work will try to represent these multiple input/output streams in StreamIt so the compiler can detect such cases and improve memory layout for GPUs.

5.5.2 Histogram

The *histogram* benchmark computes the distribution of pixel intensities within an image. *Histogram* is implemented using a technique called stream reduction, which is common in many GPU applications. Each phase of stream reduction removes some elements of input data, performs computation on them, and sends the results as a new input to the next phase. The *histogram* benchmark has several phases. In the first phase, the input data array is divided into fixed size blocks. In the second phase, a sub-histogram for each block is computed. In the final phase, all the sub-histograms are collated into a single histogram.

A StreamIt graph of stream reduction is shown in Figure 5.11. The number of actors in these type of benchmarks is data-size dependent, therefore, as the size of the input data grows, the number of phases increases and the overhead of launching the kernels becomes dominant. Sponge can fuse all of these phases together but the final actor would have a large *pop* rate. Since the *pop* rate of this actor is very large, it is not possible to use the limited shared memory to coalesce its memory accesses. In both cases, the large number of kernels and the uncoalesced memory accesses result in degraded performance.

In the hand-optimized CUDA implementation, there is only one kernel for all of the phases of the reduction but the number of threads that do the actual work in each phase is different. As a result, the hand-written CUDA histogram benchmark outperforms Sponge's generate CUDA code by 5x.

We would like to enhance the performance of Sponge in this type of benchmarks by detecting the stream reduction subgraph in the compiler and replacing them with one specialized stream reduction kernel that mimics the behavior of the hand-optimized CUDA.

5.6 Related Work

The most common language GPU programmers use to write CUDA code is ”C for CUDA” (C with NVIDIA extensions and certain restrictions). Tuning these C like programs is highly challenging because managing the amount of on-chip memory used per thread, the total number of threads per multiprocessor, and the pattern of off-chip memory accesses are some of the problems that developer need to solve manually to achieve good performance [70]. To alleviate this burden, recent studies has been done to automatically manage these parameters in CUDA programs.

One study, closely related to Sponge, is the optimizing compiler introduced by Udupa et al. [82]. They compile stream programs for GPUs using software pipelining techniques. In the software pipelining approach, different actors from different iterations are simultaneously processed. Their technique, though promising, does not perform well on GPUs because it exploits task-level parallelism and is not able to exploit the massive amount of data-level parallelism power of GPUs. There has been recent work [86] on GPU compilation for memory optimization and parallelism management. The input to this compiler is a naive GPU kernel function and their compiler analyzes the code and generates optimized CUDA code. This work is distinctively different from this work because Sponge is able to exploit the information in the high level stream graph and perform kernel-level optimizations specific to StreamIt, such as graph restructuring, and then apply lower optimizations , such as memory and thread hierarchy management.

CUDA-Lite [87] is another compilation framework that takes naive GPU kernel functions as input and tries to coalesce all memory accesses by using shared memory. Programmers need to provide annotations describing certain properties of data structures and code regions designated for GPU execution. Our work is different because Sponge does not need any annotations. Sponge also uses shared memory to coalesce memory accesses and can maximize the utilization of various resources on GPUs, such as registers. Another difference is that when the size of shared memory limits the number of worker threads, Sponge is able to insert helper threads to accelerate the transferring of data between global and shared memory. hiCUDA [31] is a high level directive based compiler framework for CUDA programming where programmers need to insert directives to define the boundaries of the kernel function into sequential C code. Another work in the area of automatic CUDA generation is [53]. The Authors in this work generate optimized CUDA programs from OpenMP programs. They do not use shared memory in their compiler for coalescing memory accesses. Hong et al. [34] propose an analytical performance model for GPUs that compilers can use to predict the behavior of their generated code. Fung et al. [22] regroup threads into new warps to minimize the number of divergent warps. Chen et al. [15] use communication and computation threads to overlap the data exchange of the boundary nodes between adjacent thread blocks. This is fundamentally different from what Sponge achieves using helper threads by performing parallel prefetching of data.

MCUDA [75] tries to compile CUDA programs for a conventional shared memory architecture. MCUDA can be used to increase the performance of traditional shared memory parallel systems using CUDA optimization techniques. With the stream programming model, it is possible to use architecture specific optimizations for a wide range of architectures. Researchers have already proposed ways to map and optimize synchronous data-flow languages to SIMD engines [36], distributed shared memory systems [49], and also field programmable gate arrays [37].

Performing runtime re-compilation of GPU binaries for adapting code to different targets is another approach that can provide portability across GPUs. OpenCL [48] is one the approaches taken by industry to achieve portability. We believe OpenCL in its current form suffers from the same inefficiencies as CUDA and does not provide an architecture independent solution.

There is a large body of literature that deals with exploiting parallelism in streaming codes for better performance. The most recent and relevant works include compilation of new streaming languages such as StreamIt, Brook [13], Sequoia [21], and Cg [55] to multi-cores or data-parallel architectures. For example, Gordon et al. [26] and [25] perform stream graph refinements to statically determine the best mapping of a StreamIt program to a multi-core CPU. Liao et al. applies classic affine partitioning techniques to exploit the properties of stream operators [83]. There is also a rich history of scheduling and resource allocation techniques developed in Ptolemy that make fundamental contributions to stream-scheduling (e.g., [68, 30]). In a recent work [78], the authors talk about the usefulness of different features of StreamIt to a wide range of streaming applications. Several works, such as [35], propose techniques to dynamically recompile streaming application based on availability of resources in heterogeneous system. Sponge can be a complementary addition to these works as GPUs are becoming a commodity in heterogeneous systems.

5.7 Summary

Heterogeneous systems, where sequential work is done on traditional processors and parallelizable work is offloaded to a specialized computing engine, will be ubiquitous in the future. Among the different solutions that can take advantage of this parallelism, GPUs are the most popular solution and have been shown to provide significant performance, power efficiency and cost benefits for general purpose computing in highly-parallel computing domains. GPUs achieve their high performance and efficiency by providing a massively parallel architecture with hundreds of in-order cores and exposing parallelism mechanism and also the memory hierarchy to the programmer. One key to maximizing the performance in these future heterogeneous systems will be to efficiently utilize not only the host processor, but also the GPU.

While GPUs provide a very desirable target platform for accelerating parallel workloads, their programming complexity poses a significant challenge to application developers. Languages, such as CUDA, alleviate the complexity problem to some extent but fail at abstracting the underlying GPU architecture. Therefore, managing the amount of on-chip memory used per thread, the total number of threads per multiprocessor, and the pattern of off-chip memory accesses are problems that developers still need to manage in order to maximize GPU utilization.

In this work, we propose Sponge; a streaming compiler for the StreamIt language that is capable of performing an array of optimizations on stream graphs and generate efficient CUDA code for GPUs. Optimizations in Sponge facilitate a write-once software paradigm where programmers can rely on the compiler to automatically create customized CUDA for a wide variety of GPU targets. The optimizations in Sponge improve the performance compared to naive CUDA implementations by an average of 3.2x. Finally, as a case study, we compare the performance and implementation of two hand-optimized CUDA benchmarks, Black-Scholes and Histogram. For Black-Scholes, Sponge is able to achieve within 30% of the performance of the hand-optimized CUDA code. Future work on Sponge will improve automatic detection of certain memory layout characteristics and stream graph representations that are currently not supported.

CHAPTER VI

Flexible Compilation for Dynamic Resource Changes

6.1 Introduction

Many-core processors provide a lot of flexibility in that they can potentially speed up the execution of individual applications (because of increased parallelism), while also having the ability to run many applications at the same time. As the number of applications that can effectively use multiple cores increases, it will become necessary to develop strategies that can adequately manage the allocation of resources between applications. Resource allocation is a challenging problem because application behavior (and hence resource requirements) can often vary in unpredictable ways, depending on factors that include dynamic workloads and variability in end-user scenarios. The issue is made more challenging by the numerous heterogeneous architectural resources that are already exposed to software (e.g., the compiler). We believe that managing the allocation of resources effectively requires many non-trivial tradeoffs, and we introduce Flextream as a means to address this issue.

Specifically, we address the issue of provisioning an individual application to run on a heterogeneous architecture under varying configurations of resource allotments. In doing so, applications are able to efficiently and effectively adapt, at runtime, to changes in the number and kind of resources at their disposal. For example, consider a mobile device that serves as a multimedia player and an internet browser. If the user is running only one of the two applications, then that application can potentially exploit all of the available resources in the device. However, as soon as the user also starts browsing the web, the resources available for the media player must change to accommodate the new application. If either of the applications is not properly provisioned to run on a varying number of resources, the end-user experience will almost surely be a poor one.

Static compilation approaches, in general, can generate high-quality resource allocations offline. However, such solutions are often sensitive to runtime variations in resource availability. In other words, any change in the underlying architecture's parameters, such as available on-chip memory or the number of cores, will result in an inefficient execution of a statically scheduled application in the best case, or code that can not execute in the worst case.

One potential solution to this problem is to compile alternative versions of an application, and to dynamically switch between versions according to the resources that are available in the architecture. For example, the media application running on an 8-core device can be provisioned to run on either 1, 2, 4, or 8 cores. The obvious deficiencies of this approach are three fold. First, this strategy can lead to large amounts of code bloat. Second, it may be impractical to statically consider a high number of architectural configurations. Lastly, the application may have to execute an inefficient fail-safe implementation (e.g., sequential) if the runtime scenario yields a set of resources that was not statically considered.

An alternative solution is dynamic compilation, where the application is repeatedly compiled at runtime when resources change—this can arise if the number of available cores, or the amount of memory that is available, or the available bandwidth varies. This is a promising approach because it can continuously adapt to changes in resource availability, if only the costs of compilation and adaptation can be made low enough to be practical. In order to keep the costs of compilation down, the runtime compiler is likely to be limited to a small set of optimizations. Furthermore, if we consider all of the resource ingredients that can vary at runtime, it will be quite challenging to engineer an efficient solution that addresses all of them well.

In this work, we propose a compilation and runtime adaptation system called Flextream. It is aimed at addressing the challenges described above in the context of streaming applications. In Flextream, a streaming application is represented as a graph, where the nodes encapsulate computation, and the edges between nodes describe dataflow. A stream program (graph) is mapped to a many-core heterogeneous architecture by assigning nodes to cores, and dataflow to communication channels between cores (e.g., DMA transfers between cores, or between main and local memories). The main innovation in Flextream is an *adaptive stream graph modulo scheduling* algorithm that combines the benefits of static scheduling with the advantages of dynamic adaptation. This strategy, of using an adaptive hybrid (static-dynamic) compilation approach, can lead to significantly better resource utilization, and can help deliver the promise of many-cores to end-users.

Flextream consists of two main components. The first part performs static compilation of an application to a virtualized multicore system using heuristics for controlling the amount of parallelism in the graph, and an integer linear programming (ILP) formulation to find the optimal mapping of nodes to resources (i.e., work partitioning). The second part consists of a light-weight online (dynamic) adaptation system that modifies the active schedule based on the available resources in the architecture. Dynamic adaptation consists of several phases including finding a new processor assignment, stage assignment, and buffer allocation. The online phases are designed to be light-weight and yet produce efficient results.

In this chapter, we mainly focus on heterogeneous systems with distributed memory similar to the IBM Cell [33] processor. Using the proposed framework, an application is statically compiled for a configuration of the architecture with the greatest number of resources which may include processing elements, on-chip storage and bandwidth. This results in high-quality solutions for a specific configuration. The dynamic light-weight layer uses the result of the static compilation as a hint to quickly discover an efficient solution for the new system configuration. Our experiments show that assisting the online adaptation phase with a static solution reduces runtime overhead and greatly improves the quality of the solutions that the online phase discovers. Our approach eschews the need for recompilation when resources change, and thus enables software developers to produce adaptive and high-quality streaming applications. The online adaptation phase uses a technique similar to [88] (called Multicore Streaming Layer or MSL) to stop the current schedule and distribute the new schedule between the processors. More details about this technique are mentioned in Section 6.2.2.

This work makes the following contributions:

- An efficient framework for adaptive compilation of streaming applications to heterogeneous multicore systems is proposed.

- A parallelism-tuning heuristic coupled with a scalable work partitioning based on ILP formulation is proposed to find a static software pipelined scheduling for streaming applications.

- Highly efficient dynamic work redistribution and buffer allocation algorithms are introduced to adapt the software pipelined schedule dynamically to efficiently exploit the capabilities of the target platform.

The rest of the chapter is organized as follows. In Section 6.2, the target architecture, input language, and multicore streaming layer are discussed. Then, the static compilation and online adaptation layer of Flextream are discussed in Section 6.3. Finally, in Section 6.4, the framework is evaluated. Section 6.5 discusses some of the related works that motivated this system.

6.2 Background

6.2.1 Architecture

The compilation target in this chapter is a streaming memory multicore architecture where on-chip memory structures are addressed as local memory and are explicitly managed. Such architecture provides the compiler with a great deal of flexibility in terms of orchestrating code and data locality, and managing communication granularity, frequency, and latency.

The target system is similar to the Cell processor in terms of the high-level architecture. It consists of a more powerful master processor and several slave processing elements. The

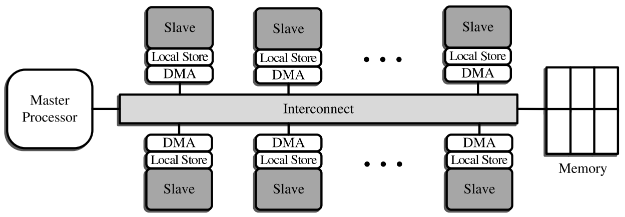


Figure 6.1: *General architecture template*

master processor is similar to the PowerPC core in the Cell processor running at 2GHZ with 32KB L1 and 1MB L2 cache. Each slave core contains a local memory for instruction and data, called *local store*, and a memory flow control (MFC) unit which can perform DMA operations to and from the local stores independent of the cores. The slave cores can only access the local store, so any sharing of data has to be performed through explicit DMA operations. The ability to perform asynchronous DMA operations allows overlap of computation and communication, and is leveraged for efficient software pipelining of stream graphs. The multicore system used during static compilation (Section 6.3.1) is similar to the processor in Figure 6.1 and has 32 slave cores. The actual physical processor used during online adaptation (Section 6.3.2) also has the same architectural template but the number of slave cores varies in each experiment from 2 to 32.

6.2.2 Multicore Streaming Layer

We use the runtime system introduced in [88] to dynamically manage resource allocations. The runtime system, called the *multicore streaming layer (MSL)*, supports loading and unloading of computation (e.g., streaming actors) on different cores, allocating local and global buffers, and managing DMA transfers for orchestrating communication. The MSL also consists of a set of commands that the online adaptation system can use to migrate from one schedule to another by moving computation between cores, allocating new buffers in different regions of local or global memory, and so on.

In our implementation of the MSL, the master processor generates the commands that are necessary for adapting an extant schedule. These commands are sent to the slave processors through memory mapped registers called mailboxes. Each slave processor runs a very light-weight manager that is able to receive the commands from its input mailbox, decode the instructions, and act on them. Based on the commands, the slave processors can allocate buffers in their local stores, setup DMA transfers and run code for a desired duration. The overhead of delivering the commands varies according to the size of the command and the latency of mailbox transfers. The results that are presented in this chapter show that we achieve a very low overhead when adapting to resource changes. This work does not detail the design of the command system. The interested reader is referred to [88] and [46].

6.3 Compiler Framework

This section describes our method for scheduling a stream graph onto a heterogeneous streaming multicore system. The objective is to obtain a maximal throughput adaptive modulo schedule of the stream graph, taking computation/communication overheads and memory requirements into account. The structure of the Flextream compilation framework is shown in Figure 6.2. The compilation is divided into two separate phases, static compilation and online(dynamic) adaptation. In the next two sections, the details of the static and online phases are discussed.

Before talking about details of the compilation steps, it is important to understand how an application compiled by Flextream behaves at runtime in the face of dynamic resource

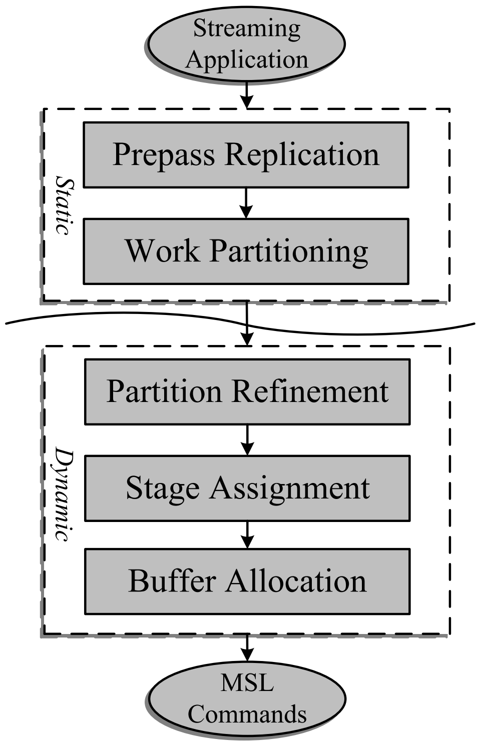


Figure 6.2: *General flow of the Flextream framework*

changes. Figure 6.3 shows an example runtime scenario. At each point during the execution, only one schedule is active. Execution starts with $schedule1$. If some of the currently-used resources become unavailable or new resources become free, an online reschedule becomes necessary. The new schedule is marked by $schedule2$ in the figure. The process of migrating from $schedule1$ to $schedule2$ consists of three main parts. First, the online adaptation phase has to generate the new schedule and the necessary MSL commands using the solution found by static phase. Second, the current schedule has to be stopped(drained). The latency of this case is directly related to the number of stages in the module schedule and the work of the maximally loaded processor. Third, the generated commands have to be sent to the active processors. In the experiments section, the overhead of each of these phases and also the performance of a full runtime scenario are discussed.

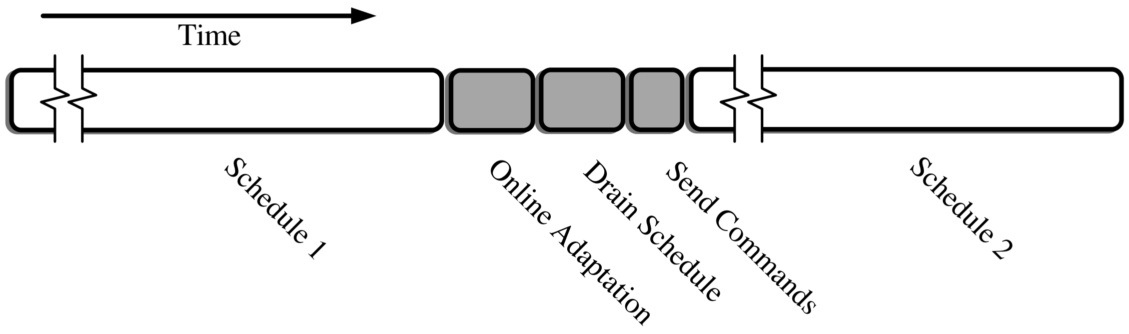


Figure 6.3: *Overall execution flow at runtime in the case of resource changes.*

6.3.1 Static Compilation

The static phase's goal is to find an optimal schedule for a virtualized member of a family of streaming multicore processors while considering bandwidth, storage and the processing capabilities of the system. This phase consists of two major sub-phases shown in Figure 6.2. First, a prepass replication is performed on the stream graph to adjust the amount of parallelism for the target system by replicating actors. Second, an ILP formulation is used to optimally partition the work between the slave cores of the target system. The virtualized system used in this phase is generally the most powerful processor of a streaming multicore family. For example, if a streaming application should be compiled for the IBM cell processor family with 4, 8, or 16 processors with local store size of 128KB or 256KB, the 16 processor version with 256KB is chosen as the virtualized system. Selecting the virtualized system in this manner, increases the freedom of the next phases to find a high quality schedule in case the program is ported to another configuration with a more limited set of resources or the availability of the resources changes at runtime.

Compared to [49], Flextream's static phase takes a different approach toward static modulo scheduling. The static phase consists of a separate step to perform replication instead of integrating it with the ILP formulation. This greatly improves the scalability of the

ILP formulation and enables the inclusion of other crucial constraints about memory allocation and data transfer overheads. Ignoring these factors can have a significant negative impact on the runtime performance in systems with low-bandwidth interconnects.

6.3.1.1 Prepass Replication

Figure 6.4 shows the theoretical speedup possible for a set of unmodified stream programs for 2 to 64 processors. The actors present in the programmer-conceived stream graph are assigned to processors in an optimal fashion such that the maximal load (work) on any processor is minimized. Speedup is calculated by dividing the single processor runtime by the load on the maximally loaded processor. The programmer-conceived stream graph has ample parallelism that can be exploited on up to 8 processors. Beyond 8 processors, the speedup begins to level off.

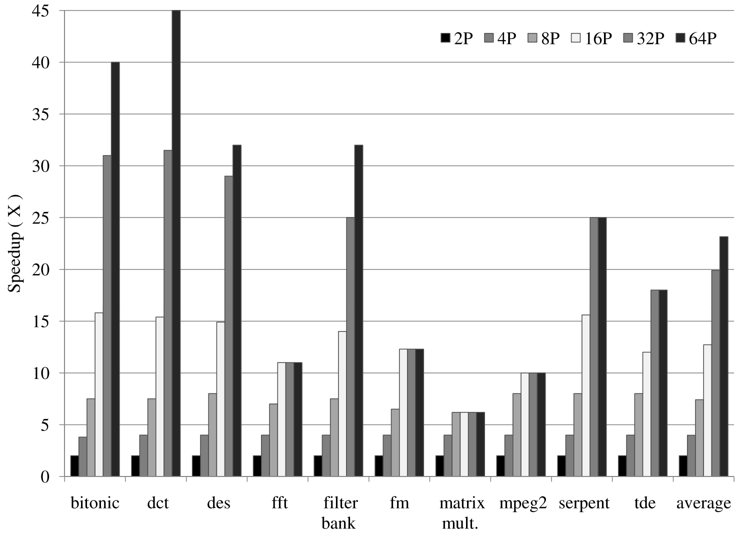
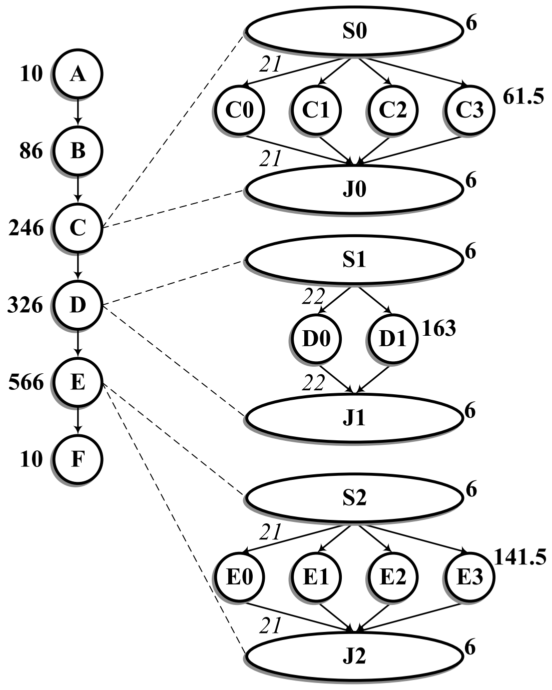


Figure 6.4: *Theoretical speedup in the absence of replication.*

Most benchmarks just do not have enough actors to span all processors. For example, `fft` has only 17 actors in its stream graph, therefore no speedup is possible beyond 17 processors. Another reason for the speedup limitation is that work is not evenly distributed across the actors. Even though the computation has been split into multiple actors, the programmer has no accurate idea how long an actor's work function will take to execute on a processor when coding the function. This leads to less scaling on 16 or more processors.



(a)

Proc.	*Before*	*After*
P0	A	A, E0
P1	B	B, C0
P2	C	C1,C2, C3
P3	D	D0
P4	E	E1
P5	F	F, E2
P6		E3
P7		D1

(b)

Figure 6.5: *This figure shows an example stream graph and how replication is performed. Part (a) shows the original graph and the version after replication. In part (b), the partitions before and after replication are shown.*

Most of the stream benchmarks are completely stateless, i.e., all actors are data parallel [25]. In fact, only `mpeg2` has actors that are stateful. Data parallel actors can be replicated any number of times without changing the meaning of the program. Replicating data parallel actors not only allows work to span more processors, it also allows work to be evenly distributed across processors by making the largest indivisible unit of work smaller.

To provide the next phases of the compilation flow with ample opportunity to efficiently utilize the target system's capabilities, a prepass replication is performed on the stream graph. Algorithm 2 shows the general steps of this phase. The main task is to heuristically replicate larger actors based on an estimate of the optimal work partitioning of the current graph. Maximally replicating the larger actors may not always result in the best solution for the next phase. Excessive replication of actors is always discouraged, because that increases split/join overhead and overall code size. Therefore, graph partitioning on the original stream graph is used to estimate the solution of the work partitioning phase. The number of requested partitions is set to the number of processors in the virtualized target processor.

Graph partitioning is fairly fast and produces a reasonable estimate of the optimal work distribution of the stream graph for the virtualized target system without considering low-level constraints such as memory size, interconnect bandwidth, etc.. Each resulting partition corresponds to one of the cores in the multicore system. This solution approximately reflects the quality of the optimal solution if the current stream graph is used. Next, the replication algorithm tries to balance the partitions by replicating the largest actor in the partition with the maximum amount of work and moving the new replicas to the partition with minimum work. This process is repeated until the ratio between the maximum workload and minimum workload is less than the balance factor specified as an input to the algorithm or no more replication is possible. The while loop in Algorithm 2 performs the partition balancing task. Lines 8-10 check the degree of imbalance between partitions. Lines 14-16 determine how many replicas of the actor selected from the largest partition should be created.

An example of the prepass replication algorithm is shown in Figure 6.5. In this example, the virtualized target system has 8 cores. The original graph, shown in the left part of Figure 6.5a, has only 6 nodes and clearly will not efficiently use all 8 cores. The replica-

Algorithm 2 Prepass Replication Algorithm

```
Input: G:(V, E), #virtualProcessors, balanceFactor
 1 partitions ← PartitionGraph(G, #virtualProcessors);
 2 while true do
 3    SortPartitionsByWeight(partitions);

      { Find partitions with max and min weights. }
 4    repeat
 5       maxPartition ← NextMaxWeightPartition(partitions);
 6    until maxPartition has a dividable node
 7    minPartition ← MinPartitionWeight(partitions);

      { Check the overall balance of the partitions.}
 8    if (Weight(maxPartition) < Weight( minPartition) ∗ balanceFactor) then
 9       Finish;
10    end if

      {Find an actor in the max partition that can be replicated.}
11    repeat
12       actor ← NextLargestFilter(maxPartition);
13    until (actor can be replicated)

      { Find out how many times the actor should be replicated.}
14    replicationFactor ← Work(actor) / (Weight(maxPartition) - Weight(minPartition));
15    replicationFactor ← Max(replicationFactor, 2);
16    newFilters[ ] ← Split(actor, replicationFactor);

      {Modify the min and max partitions.}
17    AddTo(minPartition, newFilters[1]);
18    RemoveFrom(maxPartition, actor);
19    AddTo(maxPartition, newFilters[2..replicationFactor]);
20 end while
```

tion algorithm performs an initial graph partitioning on this stream graph and then tries to replicate nodes and balance the partitions. The *balance factor* for this example is set to 1.5. Figure 6.5b shows the partitions before and after replication. At the end, the ratio between maximum weight (P1) and minimum wight (P2) is 1.3. The modified graph is illustrated in the right part of Figure 6.5a.

6.3.1.2 Work Partitioning

Consider a dataflow graph $G = (V, E)$ corresponding to a stream program. Let $|V| = N$ be the number of actors. Let the basic repetition vector be r, where r_i specifies the number of times v_i is executed in the steady state. The rest of the section assumes r_i executions of v_i as the basic schedulable unit. Given P processors, a software pipeline needs some assignment of the actors and data transfer operations to the processors. The throughput of the software pipeline is determined by the load on the maximally loaded processor. For each actor and DMA transfer in the stream graph, the following ILP formulation finds a valid assignment based on the computational power of processors, bandwidth of the interconnect, and amount of on-chip memory.

In the formulation, maximization of throughput is the main objective. We borrow the terminology from operation centric modulo scheduling used in compiler backends, and use the term Initiation Interval (II) to denote the inverse of the throughput. A set of 0-1 integer variables is introduced to find the processor assignment for actors and data transfer operations. These variable are explained below:

- $a_{ij} = \{0, 1\}$: Indicates if actor i is running on processor j

- $b_{i_1 i_2 j} = \{0,1\}$: This variable will be 1 if connected actors (producer-consumer) i_1 and i_2 are both assigned to processor j

Assuming that there are n actors in the stream graph and m processors in the target system, i is between 0 and $(n-1)$ and j is between 0 and $(m-1)$. A set of constraints are designed to find a valid actor and DMA assignment under memory, bandwidth and performance characteristics of the target system. The following constraint ensures that each actor is assigned to exactly one processor.

$$\sum_{j=0}^{P} a_{ij} = 1, \quad \forall i \tag{6.1}$$

The $b_{i_1 i_2 j}$ indicator variables serve two purposes. First, they are necessary to ensure that a DMA transfer is not introduced between two connected actors if they are on the same processor. Second, the b variables help in buffer allocation constraints because the size of the buffers between a pair of connected actors varies based on when they start execution and whether they are on the same processor. The following inequalities are used for setting the b variables.

$$\begin{aligned} b_{i_1 i_2 j} &\leq a_{i_1 j} \quad \forall \text{ connected actor pairs } i_1, i_2 \\ b_{i_1 i_2 j} &\leq a_{i_2 j} \quad \forall \text{ connected actor pairs } i_1, i_2 \\ b_{i_1 i_2 j} &\geq a_{i_2 j} + a_{i_1 j} - 1 \quad \forall \text{ connected actor pairs } i_1, i_2 \end{aligned} \tag{6.2}$$

The throughput is decided based on the workload of the maximally loaded processor which is the maximum of the computation workload and the data transfer workload across all processors. In the schedule, it is always assumed that the DMA transfer between a pair of connected actors is located on the processor on which the destination actor is running. The following two inequalities denote the relation between II and the workload of each processor.

$$\sum_{i=0}^{N}(a_{ij} \times W_i) \quad \leq \quad II \quad \forall j \tag{6.3}$$

$$\sum_{(i_1\ i_2)}^{|E|}((a_{i_2 j} - b_{i_1 i_2 j}) \times D_{i_1 i_2}) \quad \leq \quad II \quad \forall j \tag{6.4}$$

W_i in Equation 6.3 indicates the work estimate of actor i on processor j. $D_{i_1 i_2}$ show the data transfer cost between a pair of connected actors i_1 and i_2. Equation 6.4 uses $b_{i_1 i_2}$ to ensure that a DMA transfer between actors is only added if they are assigned to different processors.

As it will be discussed later, the amount buffering between two connected actors depends on both where they are running and what stage they are in. Since stage assignment is a phase of the online adaptation layer, the ILP formulation can only have an estimate of the actual memory consumption of the current mapping. To obtain this estimate, it is assumed that two connected actors will be in consecutive stages if they are not on the same processor; otherwise, they are in the same stage. Based on the results of the stage assignment phase, this is a practical overestimate of the actual buffer usage. The following set of inequalities is added to the formation for the purpose of buffer allocation.

$$\sum_{(i_1,i_2)}^{|E|}[(2a_{i_1 j} + 2a_{i_2 j} - 3b_{i_1 i_2 j}) \times Buff(i_1, i_2)] \leq Mem_j, \quad \forall j \tag{6.5}$$

For each pair of connected actors i_1 and i_2 and a processor j, there are four possible values for a_{i_1j} and a_{i_2j}. In each of these cases, the amount of necessary buffering differs. Equation 6.5 is an estimate of the actual memory requirement. Sections 6.3.2.2 and 6.3.2.3 talks about the mechanics of buffer allocation at runtime in more detail.

Figure 6.6a illustrates the result of the ILP-based work partitioning on the graph shown in Figure 6.5a. Since the cores in our system are able to perform DMAs and run computation at the same time, the workload of each processor would be the maximum of the computation and data transfer workloads. The II of this system is determined by the maximally loaded processor, P0. Comparing the achieved II of 184 with the single core performance of the graph (sum of all the weights in the original graph) reveals that a 6.8x speedup is achieved on 8 cores.

6.3.2 Online Adaptation

After static compilation is performed, the generated code can be efficiently executed on a system that matches the virtual specification used during the static compilation. As mentioned before, due to the desire for porting software within members of a streaming multicore family and also for efficiently tolerating resource availability changes at runtime, online adaptation is crucial for software developers. In this section, we talk about various phases of the light-weight online adaptation layer in the Flextream framework.

Online adaptation, is mainly designed to perform light-weight adaptation of modulo scheduling solutions at runtime for the current active configuration. As shown in Figure 6.2, this part consists of three main steps, *Partition Refinement*, *Stage Assignment*, and *Buffer Allocation*. The first step tries to change the mapping of actors to processors based on the number of available processors to rebalance work assignment and memory consumption

Algorithm 3 Algorithm for Partition Refinement

Input: *processorMap(processor:actor[]), #physicalProcessors*
Output: *newProcessorMap*

```
   {Assign one workload from the current processor map to each physical processor}
 1 SortByNumberOfFiltersAscending(processorMap)
 2 for i ← 1 to #physicalProcessors do
 3     (processor:actor[]) ← RemoveNextPair(processorMap);
 4     AddTo(newProcessorMap, (processor:actor[]));
 5 end for

   {Prioritize the remaining actors and the chosen processor workloads}
 6 remainingFilters ← AllFiltersIn(processorMap);
 7 SortFiltersByWeightAscending(remainingFilters);
 8 SortByWorkAssignmentDescending(newProcessorMap);

   {Distribute the remaining actors between the chosen processor workloads}
 9 weightThreshold ← TotalRemainingWeight(remainingFilters ) / #physicalProcessors;
10 repeat
11     actor ← RemoveNextFilter(remainingFilters);
12     currentWeight ← Weight(actor);
13     AddTo(currentList, actor);
14     if (currentWeight > weightThreshold) then
15         processor ← NextPhysicalProcessor(newProcessorMap);
16         AddTo(newProcessorMap, processor:currentList);
17         Clear(currentList);
18         currentWeight ← 0;
19     end if
20 until remainingFilters is not empty
```

on each core. The solution specifies how actor executions are overlapped across processors (in space). The stage assignment step determines how the executions are overlapped in time by specifying the starting order of the actors and DMAs. The last step of the online adaptation, buffer allocation, tries to efficiently fit the buffers in the available storage units.

6.3.2.1 Partition Refinement

The virtual multicore system used in static compilation is always a superset of the actual physical system meaning that it has more cores, more memory, etc.. Therefore, the runtime configuration, which Flextream has to target, will always have more limited resources. Partition refinement is a general step that, at runtime, tunes the actor-processor mapping to the real configuration of the system. The algorithm discussed here for performing the refinement concentrates only on the computation workload of each core in a streaming multicore system, but the heuristics can be extended to account for memory and bandwidth.

Assume that the virtualized system had n slave cores (number of virtual cores) and the real system has m cores (number of physical cores). m is less than n because the real system is a less powerful member of the multicore family or some of the cores in system with n cores have to be used to perform more critical tasks. The main objective here is to reassign the actors to m cores with low overhead at runtime.

As shown in Algorithm 3, the general idea is to choose m processor assignments from the original n assignments created by the static phase. Then, take all the actors in the $(n - m)$ remaining partitions and try to evenly distribute them between the chosen m partitions. Since solving this problem based on another ILP formulation or graph partitioning will have significant overhead at runtime, a heuristic-based approach is taken.

In the algorithm, lines 1-5 choose the m work assignments with the least number of actors from the original n. The reason the assignments with least number of actors are chosen first is to increase the freedom of the second phase of the algorithm to evenly distribute the remaining actors. Then, in lines 6-8, the remaining actors and the m chosen assignments are prioritized. The remaining actors are all put in one list and sorted by work estimate (weight) in ascending order. The chosen assignments are sorted based on the total weight of each assignment in descending order. Line 9 calculates, in the ideal situation, what fraction of the remaining actors will go to each of the chosen assignments. Lines 10-20 walk through the remaining actors(sorted by ascending weight) and assigns them to the currently chosen processors(sorted by descending weight) based on the weight threshold calculated in line 9.

Figure 6.6b shows the refinement solution for the example in Figure 6.5a when the number of cores is reduced from eight to five. In this figure, the five processors are the processors that are chosen from the original work assignment shown in Figure 6.6a. The highlighted nodes denote the nodes that were originally assigned to these processors. The rest of the nodes are mapped to these processors as a result of the refinement pass. The text above each processor shows the name of the processor in the original work assignment, and the computation workload followed by the data transfer workload. In the new work assignment, the II is 289 determined by P3. The optimal static solution for the 5-core problem will have II of 283 which is about 3% faster than the solution shown here.

Although the algorithm in this section ignores memory requirements, it is sufficient to modify the heuristics used here to consider memory requirements of the assignments. Prioritization of the remaining nodes after the initial selection can be done based on an affinity

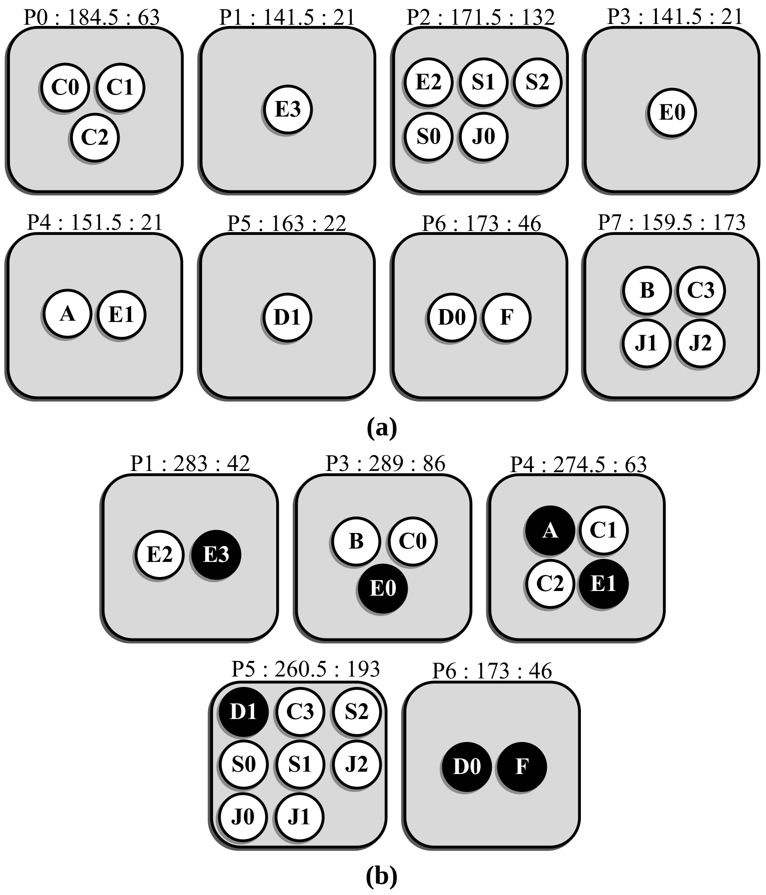


Figure 6.6: *Part (a) shows the solution of the work partitioning onto 8 cores for the example shown in Figure 6.5a. Part (b) illustrates the solution of the partition refinement if number of cores changes to 5. The actors shaded in black exist in the related processors in the original solution(a) as well as final solution(b).*

function that estimates the extra necessary memory if a node is added to a chosen processor. This type of priority function helps to keep the memory usage of each assignment under control.

6.3.2.2 Stage Assignment

The processor assignment obtained by the method described in the previous section provides only partial information for a software pipeline. Namely, it specifies how actor executions are overlapped across processors, but it does not specify how they are overlapped in time. The only goal of the processor assignment step is to load balance, therefore it assigns actors to different processors without taking any data precedence constraints into consideration. An actor assigned to a processor could have its producer assigned to a different processor, and have its consumer assigned to yet another processor. To honor data dependence constraints and still realize the throughput obtained from processor assignment, the actor executions corresponding to a single iteration of the entire stream graph are grouped into *stages*. Within a single processor, no stages are *active* at the beginning of execution. During the initial few iterations, stages are activated sequentially, thus filling up the pipeline and enabling executions of data dependent actors belonging to earlier iterations concurrently with actors from later iterations. In steady state, all stages are active on a processor, thus realizing the throughput obtained from processor assignment. The pipeline is drained by deactivating stages during the final few iterations.

Algorithm 4 Actor Stage Assignment Algorithm

```
Input: G:(V, E), processorMap(processor:actor[])
Output: actorStageMap(actor:int)
for all (actor f1 in G in topological order) do
    maxStage ← 0; flag ← false;
    for all actor f2 in parents f1 do
        if (Stage(actorStageMap, f2) ≥ maxStage) then
            maxStage ← Stage(actorStageMap, f2);
            if (Processor(processorMap, f1) != Processor(processorMap, f2)) then
                flag ← true;
            end if
        end if
    end for
    if (flag) then
        stage ← maxStage + 2;
    else
        stage ← maxStage;
    end if
    AddTo(actorStageMap, f1:stage)
end for
```

The main goal of the stage assignment step is to overlap all data communication (DMAs)

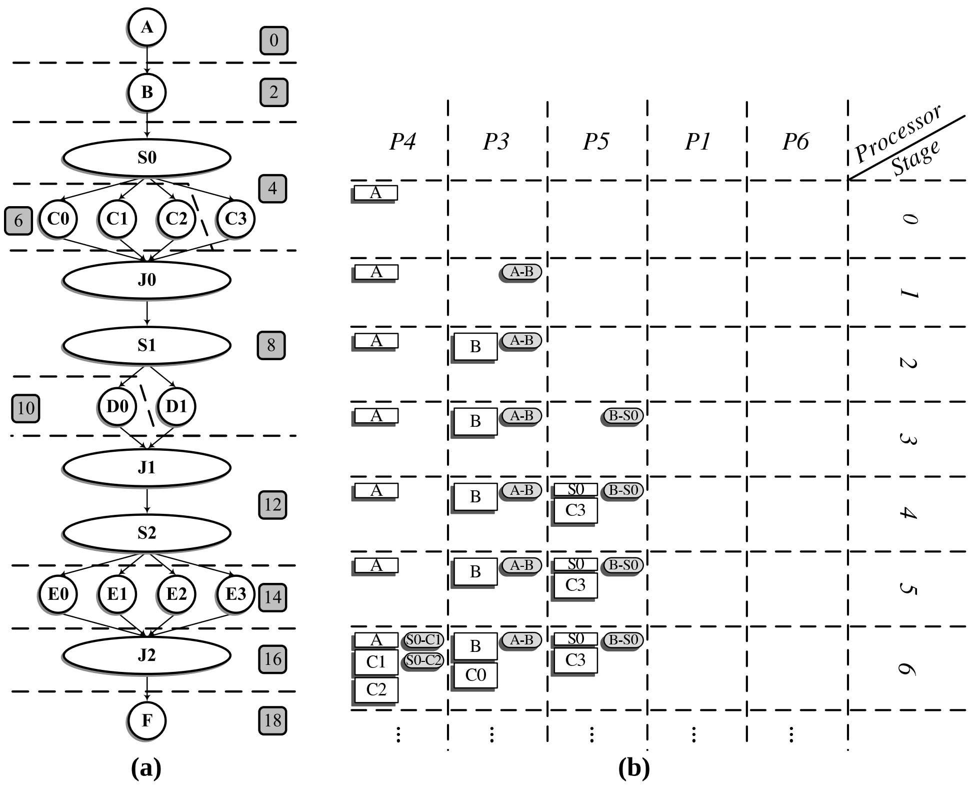


Figure 6.7: *The example shown in 6.5a after stage assignment is illustrated in part (a). The number in the gray boxes show the stage number of the actors marked by the dashed lines. Part (b) demonstrates the execution of the first 6 stages of the schedule found by Flextream.*

between actors. To achieve this, the stage assignment step considers the DMAs as schedulable units. To honor data dependences and ensure DMAs can be overlapped with actor executions, certain properties are enforced on the stage numbers of actors. Consider a stream graph $G = (V, E)$. The stage to which an actor i is assigned is denoted by S_i. In addition, the processor to which i is assigned is denoted by p_i. The following rules enforce data dependence and ensure DMA overlap.

- $(i_1, i_2) \in E \Rightarrow S_{i_2} \geq S_{i_1}$, i.e., the stage number of a consuming actor should come after the producing actor. This is to preserve data dependence.

- If $(i_1, i_2) \in E$ and $p_{i_1} \neq p_{i_2}$, then a DMA operation has to be performed to transfer the data from p_{i_1} to p_{i_2}. The DMA operation is given a separate stage number S_{DMA}.

The inequality $S_{i_1} < S_{DMA} < S_{i_2}$ is enforced between the stages of the different actors and the DMA operation. The DMA operation is separated from the producer by at least one stage, and similarly, the consumer is separated from the DMA operation by one stage. This ensures decoupling, and allows the overlap of the producer and the DMA, as well as the DMA and the consumer.

As shown in Algorithm 4, a topological traversal of the stream graph is necessary to assign stages to actors. For each actor, the maximum stage of its parents is found and a flag is set if the parent with maximum stage is not on the same processor as the actor. This part of the algorithm is done in lines 3-10. For each actor, if the parent with maximum stage number is on a different processor, there will be a two stage gap between the parent and the child. Otherwise the child actor can be placed in the same stage as the parent with maximum stage (lines 11-16). The result of the stage assignment is illustrated in Figure 6.7a. There are total of 18 stages in this schedule. One interesting point in this figure is that actors $D0$ and $D1$ are not in the same stage. This is because $D1$ is located on the same processor as $S1$. This figure does not show the stages for DMA operations for the sake of figure readability. Figure 6.7b shows how the schedule runs based on the work assignment and stage assignment. In this figure, DMAs are shown as shaded boxes. This figure demonstrates how stage assignment specifies the ordering between actors in time and work partitioning (and partition refinement) determines actor-to-processor (space) assignment.

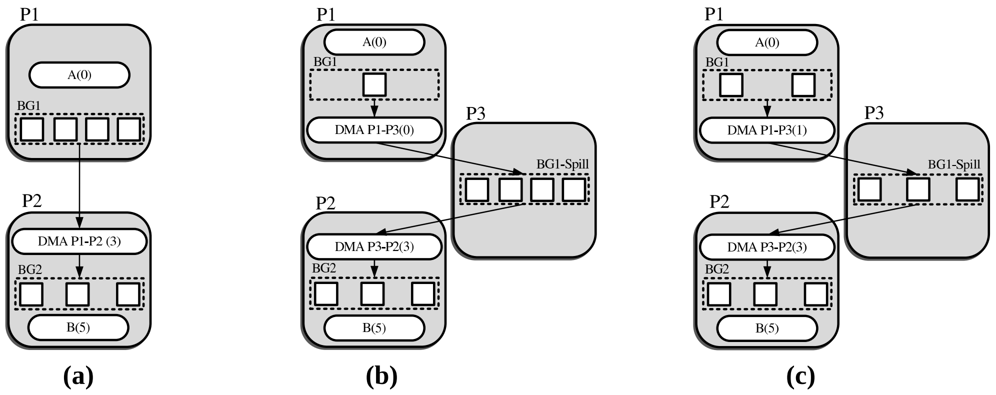


Figure 6.8: *Different approaches to buffer allocation for a producer-consumer pair is demonstrated in this figure. In (a), the original arrangement of buffers before performing buffer allocation is shown. In parts (b) and (c), two approaches that Flextream could take and their effects on the overall schedule and memory consumption is illustrated.*

6.3.2.3 Buffer Allocation

Buffer allocation tries to efficiently fit the storage requirements of the schedule, found by the previous phases, into the available memory units. In the software pipelined schedule, connected actors communicate through a set of buffers. The number of necessary buffers for a producer-consumer pair varies depending on the time they start (stage mapping). In this section, the set of buffers between a producer-consumer pair is called a *buffer group*. Based on its stage number, a producer actor could be executed multiple times before one of its consumers is ever executed. The number of buffers in a buffer group needed to store the output of a producer (actor or DMA operation) assigned to stage S_p feeding a consumer(actor to DMA operation) on stage S_c can be calculated as $S_c - S_p + 1$. For example, in Figure 6.7b, the number of buffers necessary between actor $S0$ and DMA operation $S0$-$C1$ is 2 because they are in stages 4 and 5, respectively. All the phases before buffer allocation assume that the buffers between a producer actor and a DMA operation are stored in the *local memory* of the processor on which the producer is running. Symmetrically, the

Algorithm 5 Buffer Allocation Algorithm

```
Input: procMap(processor:actor[]), stageMap(actor:int)
    {Compute memory usage per local store based on work and stage assignment}
 1  memoryUsage[processor:long] ← Update(procMap, stageMap);

    {Find the processors that their local store needs spilling}
 2  (victimProcs[], nonVictimProcs[]) ← FindVictims(memoryUsage);
 3  SortByWorkLoadDescending(victimProcs);

    {Find victim buffers}
 4  for all Processor p in victimProcs do
 5      savings = 0;
 6      BufferGroup buffs[] = BufferGroups(p);
 7      SortBySpillSizeDescending(buffs);
 8      for all BufferGroup bg in buffs do
 9          savings ← savings + SpillSize(bg);
10          if (memoryUsage[p] - savings ¡ LocalStoreSize(p)) then
11              break;
12          end if
13          add(victimBuffers, bg);
14      end for
15  end for

    {Find target location for the victim buffers and fix the schedule}
16  for all BufferGroup bg in victimBuffers do
17      target = findTarget(bg, memoryUsage, nonVictimProcs);
18      MoveBufferTo(bg, target);
19      newDMA[] = CreatNewDMA(bg);
20      UpdateStageMap(newDMA);
21      Update(memoryUsage);
22  end for
```

buffers between a DMA operation and a consuming actor are stored on the local store of the consuming processor.

In the work partitioning, partition refinement and stage assignment, it is assumed that all the buffer groups will fit in the local stores of the cores on which the corresponding actors are running. Therefore all the DMAs are from local store to local store. In some situations, based on the stage map and amount of buffering that is needed between a pair of actors, the local store may not be large enough to fit all the buffers. In those cases, in order to have a schedule that can actually run on the target system, some of the buffer groups have to be spilled to other local stores that have empty space or main memory. Spilling buffer groups will result in changes in the schedule. Basically, after moving a buffer group to another storage unit, new DMAs have to be added to the schedule. These DMAs are needed to ship the data between the local store of the processors on which the related actor is running and the new memory unit. The addition of the DMAs can increase the workload of the processors resulting in an increase of II. Since the cost of a DMA to and from main memory is significantly higher than the cost of a transfer between local stores, it is desirable to first exploit the free space in the local stores before utilizing the main memory.

The buffer allocation algorithm is shown in Algorithm 5. First, the memory usage of the current schedule is calculated based on the processor and stage assignments (Line 1). Then, the list of victim (overcommitted) processors is formed. This list contains all processors that exceed the size of their local stores (Line 2) and is sorted in descending order by the amount of work that is assigned to each processor (Line 3). The victim processors are given the chance to make use of other local stores with priority given to processors with more work. It is more beneficial to spill the buffers into the processors with more work first, because these spilled buffers are more likely to fit in other processors' local stores, resulting in less DMA overhead. Then, in lines 4 to 15, the list of buffer groups that do not fit in the corresponding local stores is extracted. This part tries to spill as few buffer groups as possible (by spilling the largest ones first) to reduce the overhead of DMA transfers. At the end(lines 16-22), the algorithm goes through the selected buffer groups and tries to move them to other local stores first and then main memory. After finding the target (local store, main memory), for each spilled buffer group, new DMAs are added to the schedule and the current stage assignment is updated.

The function, *UpdateStageMap*, in line 20 of Algorithm 5 can take two different approaches for updating the stage assignment and adding the new DMAs. These approaches are illustrated in Figure 6.8. The first part of the figure shows the stage and processor assignment for a pair of actors. Actors A and B are mapped $P1$ and $P2$ and start at stages 0 and 5. A DMA located on $P2$ in stage 3 transfers data between A and B. The buffer groups and their placement before running the buffer allocation are shown in Figure 6.8a. Assume that out of the 4 buffers in buffer group 1 ($BG1$), 2 will not fit in $P1$'s local store. $P3$ is a candidate for spilling in this buffer group. In Figure 6.8b, the first possible solution to buffer allocation is shown. In this case, the buffer group is moved to $P3$'s local store and a new DMA is added to $P1$ in stage 0. The original DMA between $P1$ and $P2$ is modified to read from $P3$'s local store. The number of buffers on $P1$'s local store is reduced to 1. Since the new DMA (between $P1$ and $P3$) is in stage 0 and there is only 1 buffer between this DMA and A, the DMA has to run sequentially after A is done, increasing the workload of $P1$. The second approach, shown in Figure 6.8c, tries to place the new DMA (between $P1$ and $P3$) 2 stage after A's stage (1 in this example). In this case, the number of buffers needed in $P3$ decreases to 3, but 1 more buffer from buffer group 1 remains in $P1$'s local store. The benefit of this approach is that the new DMA can be executed in parallel with A, eliminating the possibility of increasing the workload of $P1$. Each of these approaches has its own benefit(memory usage vs. performance) and the buffer allocation algorithm chooses between them based on the size of the local stores and workload of each victim processor.

6.4 Experiments

We evaluated Flextream using a heterogeneous multicore simulator that we have built. We also leveraged the StreamIt compiler as a starting point for implementing our heuristics and used Metis [47] for graph partitioning. For the evaluation and results, we simulated a multicore system with 32 slave cores and one master core. The master core is similar to a PowerPC processor running at 2GHZ with a 32KB L1 and a 1MB L2 cache. Each slave core includes a local store for instructions and data, and a memory flow control (MFC) unit that performs DMA operations to and from the local stores independent of the slave cores.

Performance Comparison: We first compare the performance achieved using Flextream to that achieved using online whole-program graph partitioning. The graph partitioner uses the work estimate of the actors as the node weights, and the communication costs as the edge weights. We perform prepass replication for both approaches. In this experiment, we measure the performance degradation caused by either strategy, compared to the optimal schedule. We use benchmarks drawn from the StreamIt benchmark suite. Each benchmark is run 31 times, and in each run $1 < i \leq 32$, the total number of processors starts at 32 cores, and is subsequently reduced to a smaller number of cores equal to i. The average slowdown per benchmark is shown in Figure 6.9. Flextream is 9% worse than then the performance achieved using an optimal schedule, but 8% better than applying graph partitioning at runtime. The main reason for Flextream's performance edge is that Flextream leverages the optimal scheduling solution found by the static compilation phase.

Figure 6.10 compares the average time that Flextream's partition refinement step needs to generate a new processor mapping to the time taken by the graph partitioning approach.

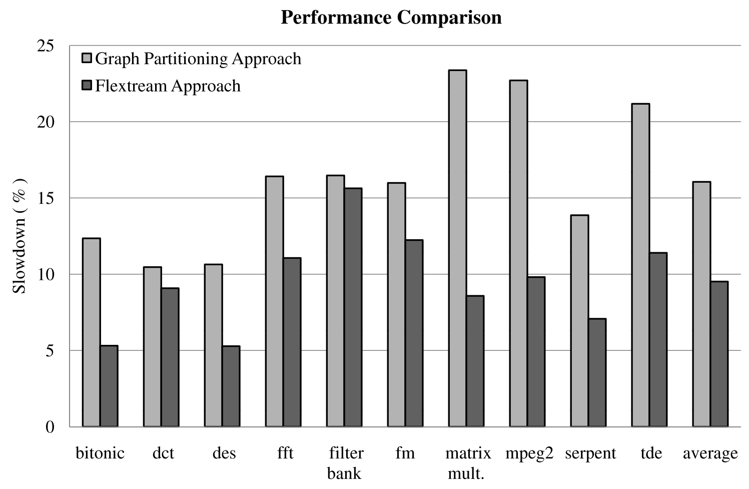


Figure 6.9: *This graph shows performance degradation when online adaptation is carried out using two different strategies.*

On average, Flextream's approach is 50%(3ms) faster than the graph partitioning approach. The results suggest that Flextream is a superior strategy to repartitioning, considering that the scheduling solutions are derived faster and yield better performance. It is also worthy to note that the runtime overheads are likely to be very high in the absence of good starting solutions. The combination of static compilation (ILP and prepass fission) and dynamic adaptation is an attractive combination that combines the benefits of static and dynamic paradigms.

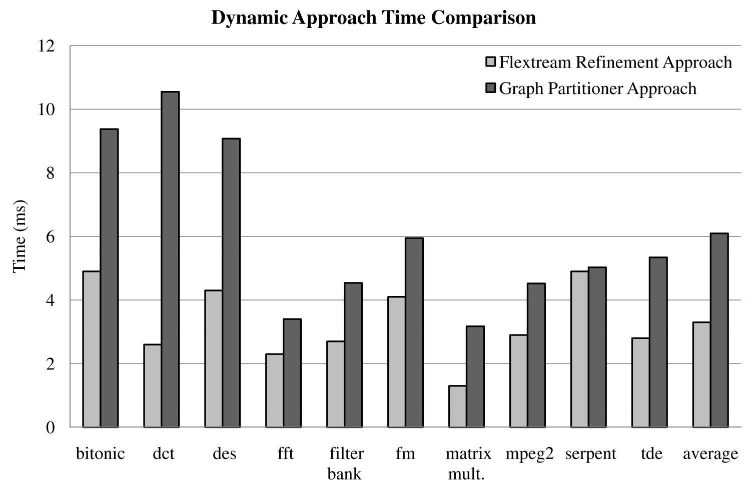


Figure 6.10: *This graph illustrates the amount of time Flextream's partition refinement takes and compares it with the graph partitioning approach.*

Overhead: We measured the overhead associated with each Flextream phase. Figure 6.11 illustrates the relative and absolute values of the times taken by each phase. We exclude from this graph the time taken to perform work partitioning since it can take several minutes for the work partitioning to find a valid ILP solution. Each of the bars in the figure include a label that represents the absolute time (in milliseconds) taken by that phase. The prepass replication requires 1283ms and is significantly longer than the time taken by the other 3 *online* phases (notice that the Y-axis starts at 90%). Among the online phases, stage assignment is the longest, followed by buffer allocation and work refinement. Most of the overhead for stage assignment is due to the topological traversal of the graph. The results indicate that the time spent in prepass replication is proportional to the size of the application (graph). Overall, this experiment supports the hypothesis that performing online adaptation using Flextream is an efficient endeavor.

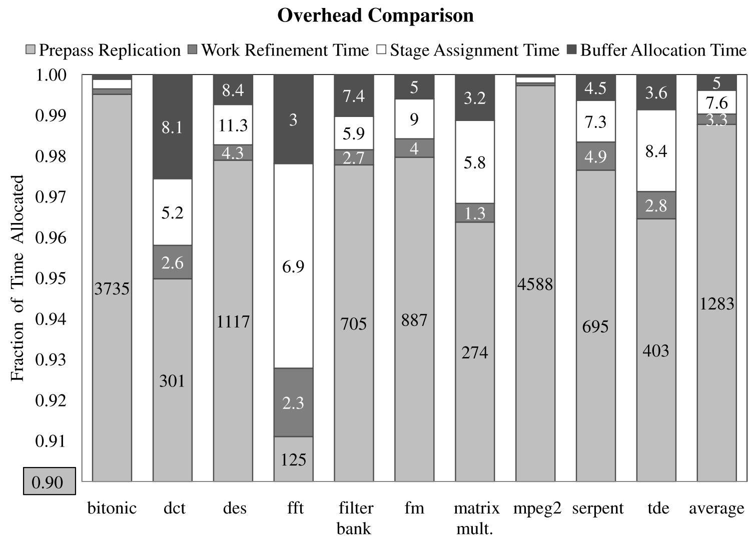


Figure 6.11: *Flextream overhead categorized by phases. Each bar has 4 parts, showing the relative (Y-axis) and absolute (labels within the bars) times spent in each of the static and online phases. Note that the Y-axis starts at 90%.*

Buffer Allocation: Buffer allocation is the last Flextream phase. This step can lead to new DMA requests and can increase the processor workloads. Buffer allocation attempts to maximize the use of the local store in order to avoid the long latencies associated with accessing main memory. The graph in Figure 6.12 shows how this optimization impacts overall performance. For this experiment the number of processors is changed at runtime from 32 cores to 8. We gradually decrease the size of the local store, starting at `Max Mem` which is large enough to ensure that no spilling occurs. This experiment shows the effectiveness of the buffer allocation algorithm in using local stores. As expected, the performance degrades when the size of the local store is reduced. The buffer allocator uses the local stores until it exhausts their capacity, at which point it has only one recourse, and that is to use main memory. For some benchmarks, reducing the local store capacity has negligible impact (e.g., `mpeg2`) because new DMA requests are added to the processors that have less work according to the original schedule (before buffer allocation). In other words, the overhead of the new DMA operations do not increase the size of the maximum workload.

A Full Runtime Scenario: We also carried out an experiment to demonstrate how Flextream might perform in a real scenario where resource availability changes multiple times at runtime. Each time the the number of available cores changes, a new schedule is generated using the online adaptation mechanism. The extant schedule is drained and the new schedule is communicated to the slave processors using the multicore stream layer (see section 6.2.2). The adaptation overhead therefore is the sum of the time taken by each of these steps. Among all of the benchmarks, the maximum overhead for sending commands is 11 micro seconds. This assumes the overhead for sending each command is 20 cycles.

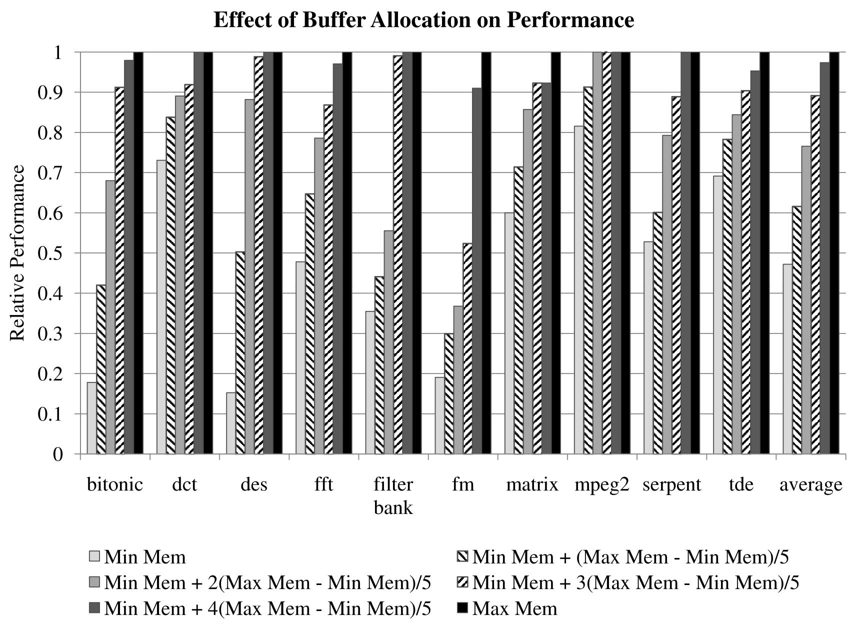


Figure 6.12: *Effect of buffer allocation on benchmark throughput. For each benchmark, the amount of memory is increased from a minimum to a maximum capacity. Throughput is recorded for 6 uniformly distributed memory sizes per benchmark.*

	Drain(ms)	Adaptation(ms)	1K sec-Flextream	1K sec-Static
bitonic sort	6.14	89.42	350M	356M
dct	0.79	42.80	380 M	452 M
des	32.39	113.80	148 M	150 M
fft	2.37	142.95	222 M	230 M
filter bank	0.44	142.95	448 M	490 M
fm	2.16	65.71	133 M	143 M
matrix mult.	3.07	37.19	62 M	71 M
mpeg2	4619	43	156 K	177 K
serpent	81.11	79.09	52 M	54 M
tde	780	66.08	1.2 M	1.3 M

Table 6.1: *Performance comparison between Flextream and optimal for a runtime scenario in which number of cores varies in this order: 32, 16, 10, 6. Each configuration runs for 250 seconds.*

Table 6.1 compares the performance of our approach with the theoretical optimal in a scenario where the number of available cores at runtime changes from 32 to 16, then to 10, and finally to 6. We assume each configuration runs for 250 seconds, for a total processing time of 1000 seconds. The theoretical optimal solution, for each runtime configuration, uses a schedule found by the static phase. The first column shows the total time needed to drain the schedules. The overhead of the online adaptation is shown in the second column. The last two columns show how many iterations of each stream graph can be executed using Flextream versus the optimal approaches. The largest difference between the last two columns occurs in `dct` which loses 16% of its throughput when using Flextream. The best performing benchmarks are `bitonic sort` and `serpent`, losing only 3% of their throughput compared to optimal. Overall, these results imply that solutions found by Flextream, in real execution scenarios, can perform close to theoretical optimal solutions.

6.5 Related Work

There is a large body of literature that deals with exploiting parallelism in streaming codes for better performance. The most recent and relevant works include compilation of new streaming languages such as StreamIt, Brook, StreamC/KernelC, and Cg to multicores or data-parallel architectures. For example, Gordon et al. [26] and [25] perform stream graph refinements to statically determine the best mapping of a StreamIt program to a multicore similar to the one we consider in this chapter. Kudlur and Mahlke apply modulo scheduling to an unrefined stream graph to maximize throughput [49]. Liao et al. apply classic affine partitioning techniques to exploit properties of stream operators [83]. There is also a rich history of scheduling and resource allocation techniques developed in Ptolemy that make fundamental contributions to stream-scheduling (e.g., [68, 30]). Flextream is unique relative to these past contributions in its ability to dynamically adapt a static schedule and resource allocation to changes in available resource at runtime. Viewed in this way, Flextream is complimentary to some static scheduling techniques, and can be applied more generally as long as we can extract a graph-representation of the streaming computation.

In contrast to static compilation techniques, there are also many existing ideas related to compilation for multicores. In [27], the authors dynamically map an abstract representation of a stream program [50] to threads that can execute in parallel on a general purpose multiprocessor. In CellSs [8], computation is expressed as functions that may be composed to form a dataflow graph. A runtime scheduler treats this graph in the same way a superscalar processor treats operations, and schedules these functions onto the Cell cores as soon as their inputs are ready. In [11], the authors describe an application-specific parallelization strategy that they applied manually. They were able to target for various configurations of the Cell architecture, which varied the number of cores in each configuration. Our work is distinctly different from these works in that we use a static compile-time schedule to automatically perform dynamic optimizations that lead to new and efficient resource allocations.

Adaptive compilation to a virtualized system is not an entirely new idea. Recent examples include Veal [18] and Liquid SIMD [17]. The authors in these works take similar approaches to the one in this work but in very different domains than the one we address in this work. In [18], adaptive loop modulo scheduling is performed for a virtualized loop accelerator system. The authors in [17] propose hybrid compilation techniques for mapping a vectorizable program to SIMD engines that have different vector lengths.

6.6 Summary

In this work, we focus on the increasingly important area of streaming computing and introduce Flextream as a flexible compilation framework that can dynamically adapt applications to the changing characteristics of the underlying architecture. This is an important contribution as software developers grapple with the details of parallelism in a rapidly changing architecture landscape. The main innovation in Flextream is an adaptive stream graph modulo scheduling algorithm that combines the benefits of static scheduling with the advantages of dynamic adaptation. Our results indicate that Flextream's approach can achieve high-performance resource allocations that are within an average of 9% compared the optimal solutions with low overhead.

CHAPTER VII

Conclusion and Future Directions

With the end of frequency scaling, industry has moved its focus from sequential single-cores to parallel architectures. In the face of a limited power budget, homogeneous parallel systems are not sufficient to meet the performance demands of various future application domains. As a result, there has been significant emphasis in both academia and industry on heterogeneous parallel architectures. These systems offer a considerable degree of specialization of the hardware resources based on the mix of applications.

Programming heterogeneous systems is more challenging than both single-core and homogeneous multi-core systems because of the heterogeneity of the available parallelism. In these systems, a programmer not only needs to worry about extracting parallelism but also has to worry about the heterogeneity of the components of the system and their respective memory and computation models. Therefore, traditional general purpose sequential programming models, such as C++ and Java, do not provide the right set of abstractions necessary to program future heterogeneous systems.

In this work, we focused on utilizing several key components of future heterogeneous systems, such as FPGAs, SIMD engines and GPUs, using the streaming program model.

Each of these components provide an efficient way to execute certain classes of applications. For example, applications with bit-level parallelism, such as encryption, are suitable for FPGAs. Since each of these platforms have their unique way of programming, we investigated how streaming paradigm can be used to implement an application once and target it to all these systems using intelligent compiler optimizations.

We also investigated how to dynamically adapt streaming applications to the availability of runtime resources. This is an important issue when multiple applications are running on a heterogeneous systems and they have different resource requirements. It is impossible to statically prepare for all the possible scenarios that may arise at runtime. Therefore, we propose a light-weight system to perform dynamic adaptation based on static compilation results.

As a whole, this dissertation demonstrates how a domain-specific approach, such as streaming, provides enough high-level information to the static compiler and runtime system such that they can tailor each domain-specific application, in this case streaming applications, to various targets of a heterogeneous system.

Finally, we propose to extend this work in the following directions:

Extending The Streaming Framework: Previous research has shown the benefits of synchronous dataflow languages as a tool to program multi-core systems. Despite its benefits, SDFs are not widely adopted by major industry players for several reasons:

- Integration of streaming with other object oriented languages is a an issue that makes application development using streaming models a complicated task. There will always be parts of any real-life application that do not fit in the streaming model.

Implementing such components in such a model is not only cumbersome but also results in poor performance and efficiency. Therefore, in order to facilitate the use of streaming models for real applications, it is necessary to semantically integrate streaming and imperative programming languages. In a unified language, programmers can decide if a part of an application fits in the streaming model supported by the language or the general purpose features of the object oriented language are more suitable. Several new projects have already started in this direction [39, 7]. We believe this is a promising approach to enable wider adoption of streaming.

- Another obstacle to wider use of SDFs is their complete static form. SDFs have static communication rates and also a static graph structure. Although these features enable more aggressive static compiler optimizations, they limit the scope of SDFs. There are already dataflow models that relax some of the constraints that SDFs impose on the applications such as Cyclo-Static [10] and Parameterizable data flow [9] models. Building static and dynamic compilation systems around the more relaxed forms of streaming is another way to extend the applicability of streaming.

Extending The Domain-Specific Approach: This work shows that if an application can be efficiently implemented using a synchronous data flow model, then the compiler is able to perform aggressive optimizations to tailor the application for various components of a heterogeneous systems. The main reason behind this capability is that domain-specific information, such as communication patterns, is exposed to the compiler. The domain knowledge helps the compiler to carry out several domain-specific optimizations that are impossible for a general purpose compiler to perform. We believe, in order to utilize future parallel systems, this domain specific approach should be extended to domains other than streaming. In this way, compilers can perform more aggressive optimizations using the domain information that is exposed by the programmer. Also, in such a programming framework, programmers are isolated from the low-level details of the underlying architecture and therefore can write portable applications.

There are two main challenges in extending our approach to other domains:

- The first challenge is to identify what other domains will be widely used in future applications. Clearly, streaming is not suitable for covering the entire application domain. Other domain-specific extensions and models are necessary to gain benefits for parts of the application domain that do not fit in the streaming model. Solving this challenge requires understanding and collaborating with domain experts about the future performance requirement in each domain and specifying the domains at the right granularity.

- The second challenge is to formally formulate each domain as a part of a unified programming language. It is impractical to present each domain in a separate language since an application may consist of parts in different domains and implementing one application using more than one language is a cumbersome task. Semantically integrating all the domains in one language is a non-trivial task which requires more research.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] AMD torrenza architecture, 2008. http://enterprise.amd.com/us-en/AMD-Business/Technology-Home/Torrenza.aspx.

[2] Intel quickassist technology, 2008. http://www.intel.com/technology/platforms/quickassist/index.htm.

[3] R. Allen and K. Kennedy. Pfc: A program to convert fortran to parallel form. Technical Report 82-6, Dept. of Math. Sciences., Rice University, Mar. 1982.

[4] R. Allen and K. Kennedy. Automatic translation of fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, 1987.

[5] R. Allen and K. Kennedy. *Optimizing compilers for modern architectures: A dependence-based approach*. Morgan Kaufmann Publishers Inc., 2002.

[6] ARM Ltd. *ARM Neon*, 2009. http://www.arm.com/miscPDFs/6629.pdf.

[7] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah. Lime: a java-compatible and synthesizable language for heterogeneous architectures. In *Proceedings of the OOPSLA'10*, pages 89–108, 2010.

[8] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta. CellSs: a programming model for the cell be architecture. *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 00(1):5, 2006.

[9] B. Bhattacharya, S. S. Bhattacharyya, and S. Member. Parameterized dataflow modeling for dsp systems. *IEEE Transactions on Signal Processing*, 49:2408–2421, 2001.

[10] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-static data flow. *IEEE International Conference on Acoustics Speech and Signal Processing*, 5:3255–3258, 1995.

[11] F. Blagojevic, D. S. Nikolopoulos, A. Stamatakis, and C. D. Antonopoulos. Dynamic multigrain parallelization on the cell broadband engine. In *Proc. of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 90–100, New York, NY, USA, 2007. ACM Press.

[12] K. Bondalapati et al. DEFACTO: A design environment for adaptive computing technology. In *Proc. of the Reconfigurable Architectures Workshop*, pages 570–578, Apr. 1999.

[13] I. Buck et al. Brook for GPUs: Stream computing on graphics hardware. *ACM Transactions on Graphics*, 23(3):777–786, Aug. 2004.

[14] Celoxica. Handel-C language overview, 1996. http://www.celoxica.com.

[15] J. Chen, Z. Huang, F. Su, J.-K. Peir, J. Ho, and L. Peng. Weak execution ordering - exploiting iterative methods on many-core gpus. In *Proc. of the 2010 IEEE Symposium on Performance Analysis of Systems and Software*, pages 154–163, 2010.

[16] M. Chen, X. Li, R. Lian, J. Lin, L. Liu, T. Liu, and R. Ju. Shangri-la: Achieving high performance from compiled network applications while enabling ease of programming. In *Proc. of the '05 Conference on Programming Language Design and Implementation*, pages 224–236, June 2005.

[17] N. Clark et al. Liquid SIMD: Abstracting SIMD hardware using lightweight dynamic mapping. In *Proc. of the 13th International Symposium on High-Performance Computer Architecture*, pages 216–227, 2007.

[18] N. Clark, A. Hormati, and S. Mahlke. VEAL: Virtualized execution accelerator for loops. In *Proc. of the 35th Annual International Symposium on Computer Architecture*, pages 389–400, June 2008.

[19] C. Consel et al. Spidle: A DSL approach to specifying streaming applications. In *Proc. of the* 2^{nd}*Intl. Conference on Generative Programming and Component Engineering*, pages 1–17, 2003.

[20] A. E. Eichenberger, P. Wu, and K. O'Brien. Vectorization for simd architectures with alignment constraints. In *Proc. of the '04 Conference on Programming Language Design and Implementation*, pages 82–93, 2004.

[21] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 83, 2006.

[22] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic warp formation and

scheduling for efficient GPU control flow. In *Proc. of the 40th Annual International Symposium on Microarchitecture*, pages 407–420, 2007.

[23] GNU Compiler Collection. Gcc 4.3.2, 2008. http://gcc.gnu.org/gcc-4.3/.

[24] M. Gokhale, J. Stone, J. Arnold, and M. Kalinowski. Stream-oriented FPGA computing in the Streams-C high level language. In *Proc. of the 8th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 49–56, Apr. 2000.

[25] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 151–162, 2006.

[26] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 291–303, Oct. 2002.

[27] J. Gummaraju and M. Rosenblum. Stream programming on general-purpose processors. In *Proc. of the 38th Annual International Symposium on Microarchitecture*, pages 343–354, Washington, DC, USA, 2005. IEEE Computer Society.

[28] Z. Guo, B. Buyukkurt, W. Najjar, and K. Vissers. Optimized generation of data-path from c codes for fpgas. In *Proc. of the 2005 Design, Automation and Test in Europe*, pages 112–117, Washington, DC, USA, 2005. IEEE Computer Society.

[29] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. SPARK: A high-level synthesis framework for applying parallelizing compiler transformations. In *Proc. of the 2003 International Conference on VLSI Design*, pages 461–466, Jan. 2003.

[30] S. Ha and E. A. Lee. Compile-time scheduling and assignment of data-flow program graphs with data-dependent iteration. *IEEE Transactions on Computers*, 40(11):1225–1238, 1991.

[31] T. Han and T. Abdelrahman. hicuda: High-level gpgpu programming. *IEEE Transactions on Parallel and Distributed Systems*, (99):1–1, 2010.

[32] J. R. Hauser and J. Wawrzynek. GARP: A MIPS processor with a reconfigurable coprocessor. In *Proc. of the 5th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 12–21, Apr. 1997.

[33] H. P. Hofstee. Power efficient processor design and the Cell processor. In *Proc. of the 11th International Symposium on High-Performance Computer Architecture*, pages 258–262, Feb. 2005.

[34] S. Hong and H. Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *Proc. of the 36th Annual International Symposium on Computer Architecture*, pages 152–163, 2009.

[35] A. Hormati, Y. Choi, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke. Flextream: Adaptive compilation of streaming applications for heterogeneous architectures. In *Proc. of the 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 214–223, 2009.

[36] A. Hormati, Y. Choi, M. Woh, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke. Macross: Macro-simdization of streaming applications. In *18th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 285–296, 2010.

[37] A. Hormati, M. Kudlur, D. Bacon, S. Mahlke, and R. Rabbah. Optimus: Efficient realization of streaming applications on FPGAs. In *Proc. of the 2008 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 41–50, Oct. 2008.

[38] A. Hormati, M. Samadi, M. Woh, T. Mudge, and S. Mahlke. Sponge: Portable stream programming on graphics engines. In *19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[39] S. S. Huang, A. Hormati, D. F. Bacon, and R. Rabbah. Liquid metal: Object-oriented programming across the hardware/software boundary. In *ecoop08*, pages 76–103, 2008.

[40] IBM. *Cell Broadband Engine Architecture*, Mar. 2006.

[41] Impulse-CoDeveloper. http://www.impulsec.com/.

[42] Intel. Intel sse4, 2006. http://download.intel.com/technology/architecture/new-instructions-paper.pdf.

[43] Intel. Intel Core i7, 2008. http://www.intel.com/products/processor/corei7/index.htm.

[44] Intel. Intel compiler, 2009. software.intel.com/en-us/intel-compilers/.

[45] Intel. Intel stellarton, configurable intel atom-based processor, 2010. http://newsroom.intel.com/docs/DOC-1512.

[46] F. Karim, A. Mellan, A. Nguyen, U. Aydonat, and T. Abdelrahman. A multilevel computing architecture for embedded multimedia applications. *IEEE Micro*, 24(3):56–66, 2004.

[47] G. Karypis and V. Kumar. *Metis: A Software Package for Paritioning Unstructured Graphs, Partitioning Meshes and Computing Fill-Reducing Orderings of Sparce Matrices*. University of Minnesota, Sept. 1998.

[48] KHRONOS Group. OpenCL - the open standard for parallel programming of heterogeneous systems, 2010.

[49] M. Kudlur and S. Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *Proc. of the '08 Conference on Programming Language Design and Implementation*, pages 114–124, June 2008.

[50] F. Labonte, P. Mattson, W. Thies, I. Buck, C. Kozyrakis, and M. Horowitz. The stream virtual machine. In *Proc. of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 267–277, 2004.

[51] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proc. of the '00 Conference on Programming Language Design and Implementation*, pages 145–156, June 2000.

[52] E. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[53] S. Lee, S.-J. Min, and R. Eigenmann. Openmp to gpgpu: a compiler framework for automatic translation and optimization. In *Proc. of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 101–110, 2009.

[54] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *Proc. of the 37th Annual International Symposium on Computer Architecture*, pages 451–460, 2010.

[55] W. Mark, R. Glanville, K. Akeley, and J. Kilgard. Cg: A system for programming graphics hardware in a C-like language. In *Proc. of the* 30^{th}*International Conference on Computer Graphics and Interactive Techniques*, pages 893–907, July 2003.

[56] O. Mencer, H. Hubert, M. Morf, and M. J. Flynn. Stream: Object-oriented programming of stream architectures using pam-blox. pages 595–604, London, UK, 2000. Springer-Verlag.

[57] Mentor. Catapult C. http://www.mentor.com/products/esl/high_level_synthesis/catapult_synthesis/.

[58] J. H. Moreno, V. Zyuban, U. Shvadron, F. D. Neeser, J. H. Derby, M. S. Ware, K. Kailas, A. Zaks, A. Geva, S. Ben-David, S. W. Asaad, T. W. Fox, D. Littrell, M. Biberstein, D. Naishlos, and H. Hunter. An innovative low-power high-performance programmable signal processor for digital communications. *IBM Journal of Research and Development*, 47(2-3):299–326, 2003.

[59] W. A. Najjar, W. Bohm, B. A. Draper, J. Hammes, R. Rinker, J. R. Beveridge, M. Chawathe, and C. Ross. High-level language abstraction for reconfigurable computing. *IEEE Computer*, 36(8):63–69, 2003.

[60] M. Narayanan and K. A. Yelick. Generating permutation instructions from a high-level description. Technical Report UCB/CSD-03-1287, EECS Department, University of California, Berkeley, Jan. 2003.

[61] J. Nickolls and I. Buck. NVIDIA CUDA software and GPU parallel computing architecture. In *Microprocessor Forum*, May 2007.

[62] D. Nuzman and R. Henderson. Multi-platform auto-vectorization. In *Proc. of the 2006 International Symposium on Code Generation and Optimization*, pages 281–294, 2006.

[63] D. Nuzman, I. Rosen, and A. Zaks. Auto-vectorization of interleaved data for simd. In *Proc. of the '06 Conference on Programming Language Design and Implementation*, pages 132–142, 2006.

[64] D. Nuzman and A. Zaks. Outer-loop vectorization - revisited for short simd architectures. In *Proc. of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 2–11, 2008.

[65] NVIDIA. *CUDA Programming Guide*, June 2007. http://developer.download.nvidia.com/compute/cuda.

[66] NVIDIA. Fermi: Nvidias next generation cuda compute architecture,

2009. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_-Compute_Architecture_Whitepaper.pdf.

[67] NVIDIA. Gpus are only up to 14 times faster than cpus says intel, 2010. http://blogs.nvidia.com/ntersect/2010/06/gpus-are-only-up-to-14-times-faster-than-cpus-says-intel.html.

[68] J. L. Pino, S. S. Bhattacharyya, and E. A. Lee. A hierarchical multiprocessor scheduling framework for synchronous dataflow graphs. Technical Report UCB/ERL M95/36, University of California, Berkeley, May 1995.

[69] G. Ren, P. Wu, and D. Padua. Optimizing data permutations for simd devices. In *Proc. of the '06 Conference on Programming Language Design and Implementation*, pages 118–131, 2006.

[70] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. mei W. Hwu. Optimization principles and application performance evaluation of a multi-threaded gpu using cuda. In *Proc. of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 73–82, 2008.

[71] D. Seal. *ARM Architecture Reference Manual*. Addison-Wesley, London, UK, 2000.

[72] L. Seiler et al. Larrabee: a many-core x86 architecture for visual computing. *ACM Transactions on Graphics*, 27(3):1–15, 2008.

[73] F. Semiconductor. Altivec, 2009. www.freescale.com/altivec.

[74] S. Sirowy, G. Stitt, and F. Vahid. C is for circuits: capturing fpga circuits as sequential code for portability. pages 117–126, New York, NY, USA, 2008. ACM.

[75] J. A. Stratton, S. S. Stone, and W.-M. W. Hwu. Mcuda: An efficient implementation of cuda kernels for multi-core cpus. In *Proc. of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 16–30, 2008.

[76] SystemC-Consortuim. SystemC language overview, 2000. http://www.systemc.org.

[77] M. Taylor et al. Evaluation of the Raw microprocessor: An exposed-wire-delay architecture for ILP and streams. In *Proc. of the 31st Annual International Symposium on Computer Architecture*, pages 2–13, June 2004.

[78] W. Thies and S. Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *Proc. of the 19th International Conference on Parallel Architectures and Compilation Techniques*, page To Appear, 2010.

[79] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A language for streaming applications. In *Proc. of the 2002 International Conference on Compiler Construction*, pages 179–196, 2002.

[80] Tilera. Tile64 processor - product brief, 2008. http://www.tilera.com/pdf/.

[81] Trimaran. An infrastructure for research in ILP, 2000. http://www.trimaran.org/.

[82] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil. Software pipelined execution

of stream programs on gpus. In *Proc. of the 2009 International Symposium on Code Generation and Optimization*, pages 200–209, 2009.

[83] S. wei Liao, Z. Du, G. Wu, and G.-Y. Lueh. Data and computation transformations for brook streaming applications on multiprocessors. *Proc. of the 2006 International Symposium on Code Generation and Optimization*, 0(1):196–207, 2006.

[84] P. Wu, A. E. Eichenberger, and A. Wang. Efficient simd code generation for runtime alignment and length conversion. In *Proc. of the 2005 International Symposium on Code Generation and Optimization*, pages 153–164, 2005.

[85] Xilinx. Virtex-4 data sheets, 2004. http://www.xilinx.com/support/documentation/v-irtex-4.htm.

[86] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A gpgpu compiler for memory optimization and parallelism management. In *Proc. of the '10 Conference on Programming Language Design and Implementation*, pages 86–97, 2010.

[87] S. zee Ueng, M. Lathara, S. S. Baghsorkhi, and W. mei W. Hwu. Cuda-lite: Reducing gpu programming complexity. In *Proc. of the 21st Workshop on Languages and Compilers for Parallel Computing*, pages 1–15, 2008.

[88] D. Zhang, Q. J. Li, R. Rabbah, and S. Amarasinghe. A lightweight streaming layer for multicore execution. *ACM SIGARCH Computer Architecture News*, 36(2):18–27, 2008.

[89] D. Zhang, Z. Li, H. Song, and L. Liu. A programming model for an embedded media processing architecture. In *Proc. of the 5^{th}International Symposium on Systems,*

Architectures, Modeling, and Simulation, volume 3553 of *Lecture Notes in Computer Science*, pages 251–261, July 2005.