A COMPUTING ORIGAMI:

OPTIMIZED CODE GENERATION FOR EMERGING PARALLEL PLATFORMS

ANDREI MIHAI HAGIESCU MIRISTE

(Dipl.-Eng., Politehnica University of Bucharest, Romania)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2011

ACKNOWLEDGEMENTS

I am grateful to all the people who have helped me through my PhD candidature. First of all, I would like to extend my deep appreciation to Assoc. Prof. Weng-Fai Wong, who has guided me with enthusiasm in the world of research. Numerous hours of late work, discussions and brainstorming sessions had always been offered when I needed them more.

I have had much to learn from several other professors at the National University of Singapore, including Assoc. Prof. Tulika Mitra, Prof. P. S. Thiagarajan and Prof. Samarjit Chakraborty. Prof. Saman Amarasinghe graciously agreed to be my external examiner, and his feedback was much appreciated. I am also grateful to Prof. Nicolae Tapus from the Politehnica University of Bucharest, who initiated me to academic research.

I would like to mention my closest collaborators from whom I learnt a great amount during these last years. In no specific order, I would like to thank Rodric Rabbah, Huynh Phung Huynh and Unmesh Bordoloi.

Several friends participating in the research program of the university have provided their support, and it will be only fair to mention them here: Cristian, Narcisa, Dorin, Hossein, Ioana, Bogdan, Cristina, Mihai and Chi-Tsai.

On the personal side, I am grateful to my parents Anca and Bogdan, my sister Ioana and my uncle Cristian Lupu for their constant support in pursuing this academic quest. Before I conclude, I would like to thank and *wai* my wife, who has never let me down, no matter the distance, Hathairat Chanphao.

TABLE OF CONTENTS

PUBLICATIONS

Publications related to this thesis:

- *A Computing Origami: Folding Streams in FPGAs.* Andrei Hagiescu, Weng-Fai Wong, David F. Bacon and Rodric Rabbah. Design Automation Conference (**DAC**), 2009
- *Co-synthesis of FPGA-Based Application-Specific Floating Point SIMD Accelerators.* Andrei Hagiescu and Weng-Fai Wong. International Symposium on Field Programmable Gate Arrays (**FPGA**), 2011
- *Automated architecture-aware mapping of streaming applications onto GPUs.* Andrei Hagiescu, Huynh Phung Huynh, Weng-Fai Wong and Rick Siow Mong Goh. International Parallel and Distributed Processing Symposium (**IPDPS**), 2011
- *Scalable Framework for Mapping Streaming Applications onto Multi-GPU Systems.* Huynh Phung Huynh, Andrei Hagiescu, Weng-Fai Wong and Rick Siow Mong Goh. Symposium on Principles and Practice of Parallel Programming (**PPoPP**), 2012

Other publications:

- *Performance analysis of FlexRay-based ECU networks.* Andrei Hagiescu, Unmesh D. Bordoloi, Samarjit Chakraborty et al. Design Automation Conference (**DAC**), 2007
- *Performance Debugging of Heterogeneous Real-Time Systems.* Unmesh D. Bordoloi, Samarjit Chakraborty and Andrei Hagiescu. Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems, 2007

SUMMARY

This thesis deals with code generation for parallel applications on emerging platforms, in particular FPGA and GPU-based platforms. These platforms expose a large design space, throughout which performance is affected by significant architectural idiosyncrasies. In this context, generating efficient code is a global optimization problem. The code generation methods described in this thesis apply to applications which expose a flexible parallel structure that is not bound to the target platform. The application is restructured in a way which can be intuitively visualized as *Origami* (the Japanese art of paper folding).

The thesis makes three significant contributions:

- It provides code generation methods starting from a general stream processing language (StreamIt) for both FPGA and GPU platforms.
- It describes how the code generation methods can be extended beyond streaming applications to finer-grained parallel computation. On FPGAs, this is illustrated by a method that generates configurable floating-point SIMD coprocessors for vectorizable code. On GPUs, the method is extended to applications which expose fine-grained parallel code accompanied by a significant amount of read sharing.
- It shows how these methods can be used on a platform which consists of multiple GPU devices connected to a host CPU.

The methods can be applied to a broad range of applications. They go beyond mapping and provide tightly integrated *code generation* tools that handle together high-level mapping, code rewriting, optimizations and modular compilation. These methods target FPGA and GPU platforms without requiring user-added annotations. The results indicate the efficiency of the methods described.

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

This thesis describes high-level code generation methods which connect mapping, code rewriting, optimizations and modular compilation in an integrated approach. In particular, it describes code generation methods for two promising parallel platforms that have emerged in mainstream computing: Field Programmable Gate Arrays (FPGAs) and Graphic Processing Units (GPUs).

Both FPGA and GPU platforms tightly integrate a large number of parallel processing units. This results in lower communication overhead [2, 39], which favors the execution of a broader spectrum of parallel applications [15, 71]. However, complex architectural constraints, inherent in these platforms, prevent the mapping of the parallel computation expressed through the application code, in a straight-forward manner, to the processing units.

This thesis shows that it is beneficial to combine the mapping step with the subsequent compilation step in an integrated approach. The thesis describes code generation methods for applications that expose a flexible program structure. The methods use either the coarse-grained parallel structure exposed by the StreamIt language, or the fine-grained parallel structure derived from the application code. In both cases, the experiments show the suitability of the proposed methods.

1.1 Code generation

In general, *code generation* consists of a series of sequential transformation steps. The first step is to *map* the application structure to the platform. Then, the application undergoes an intermediate *code rewriting* step which commits the mapping results and converts the application code to a program representation supported by the platform compiler. Eventually, the rewritten application

undergoes the final *compilation*. During each step, additional *optimizing* transformations are applied, based on the projected effect of these transformations. Some of the high-level application structure is likely to be discarded during the optimization process. Mapping and optimization decisions can not be unrolled thereafter, even if it becomes obvious, after compilation, that the application would benefit from them.

This problem becomes increasingly relevant, as the parallel platforms evolve, because the level of application abstraction is rising steadily. In order to cover a larger number of alternative platforms, the code representation tends to abstract more platform details and eventually to become platform independent [64]. Therefore, good execution performance relies heavily on the decisions taken during the mapping step, and how this step closes the gap between the abstract code structure exposed by the programmer and the target platform architecture.

FPGAs and GPUs have emerged as lead competitors in the parallel application domain. Both are characterized by shortened development cycles and increased platform variability [7]. Therefore, mapping on these platforms can not benefit from comprehensive performance projection models, similar to more matured architectures [86, 87]. This impedes application portability and clutters the accuracy of the mapping decisions.

As a workaround, current mapping tools often rely on a significant amount of user-added annotations [48, 66, 67, 72, 102] that drive the solution selection for each target platform. Using annotations reduces the inherent complexity of the mapping step for these platforms. As the platform architecture may handle hundreds of parallel threads with complex resource constraints, the annotations complement the mapping algorithms and provide guidelines for global decisions spanning the entire design space. However, the annotations are platform specific and nontrivial to assert.

Also, because mapping precedes compilation, the mapping decisions can not always capture the side effects of compilation on the performance and resource utilization of the mapped application. For example, resource sharing during compilation can decrease total resource usage, while it may introduce inter-task

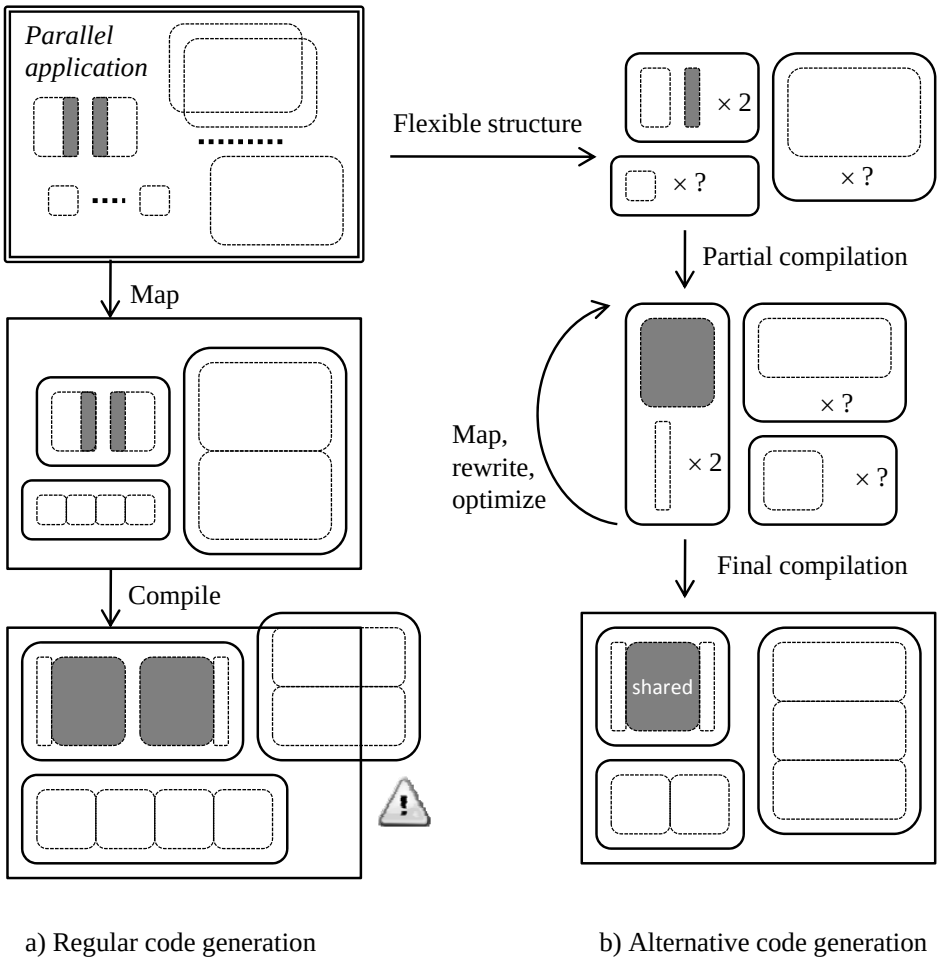


Figure 1.1: Improving code generation under resource constraints. The resource utilization is suggested by the area of the corresponding boxes.

dependencies, which lead to serial execution. Significant effort has been invested in developing new programming models and compilation methods, which can expose the platform structure and steer developers to write their code in a way that improves mapping [26, 48, 79, 90]. Usually, the developer is encouraged to write modular code that corresponds to parallel tasks which can be compiled independently. In addition, the programming models may structure data placement, often separating the computation from communication. Using these dedicated models eases the mapping to particular platforms and hides many of the platform idiosyncrasies from the user.

Consequently, current mapping tools seldom modify the structure of the program parallelism expressed by the user through the programming model. This is based on the assumptions that: (1) the programmer has gone the extra step to

ensure that all the available parallel computation is exposed, and (2) exactly that parallel structure was determined by the user to be beneficial. Unfortunately, applications are often ported to different platforms, and a certain amount of design restructuring and application tuning [67] is usually apparent after compilation, once the resource usage becomes evident, either to match the platform resources, or to match the actual degree of parallelism that maximizes the performance of the compiled application on the target parallel platform. However, after compilation, the application representation is usually flattened, and it is beyond the ability of the current code generation methods to modify the parallel structure of the application without user intervention.

While multiple design points can be manually or semi-automatically explored, large performance variability prevents proper pruning of the design space. Therefore, the adequate set of mapping and optimization decisions taken during the high-level stages of the compilation leads to a challenging problem, which affects the outcome of the entire compilation process.

1.2 Problem Description

The mapping of applications to FPGAs and GPUs is dictated by the availability of certain key resources. However, the resource usage is commonly available only as a result of the compilation step. Attempts to model resource consumption have only limited success due to the complexity of the platforms involved. In this context, disjoint mapping and compilation may lead to sub-optimal performance of the automatically generated code. Specific architectural and resource constraints on these platforms exacerbate this problem. Hence, it is important to identify methods to generate optimized code while considering these constraints.

Figure 1.1a illustrates an intuitive perspective of the problem described above, as it appears in regular code generation methods. The resources utilized by the code blocks, as well as the resources made available by the platform, are indicated by the area of the corresponding boxes. The parallel application is first mapped using a model of the target platform. Because the mapping is

done as a separate step, at the beginning of code generation, further optimization, code rewriting and compilation can lead to an entirely different outcome, in terms of resource usage, than the one predicted by the mapping decisions. The model may lack accuracy or may not capture the complete interactions between parallel compute blocks (i.e. resources shared between FPGA blocks, or serialization of parallel threads on GPU). After compilation, any inaccuracy of the original mapping model leads to a mismatch in terms of resource usage, which translates to infeasible or poor performance designs.

On FPGAs [1, 61], a common instance of this problem is related to the resource usage of each code block when implemented in reconfigurable hardware. To achieve the greatest performance, it is desirable to use most of the reconfigurable resources. However, mapping is usually overly conservative in terms of resources, because the compilation outcome can not be easily predicted, and exceeding the number of available resources leads to infeasible solutions.

On GPUs [3, 67], this problem is related to the size of fast on-chip memories. Because these memories are small, the size of the working set of each thread determines the feasibility of its placement in these memories. As this size is determined only during compilation, mapping may conservatively confine it to a large long-latency memory. The mapping then determines that an increased number of threads is necessary to offset the memory access delay, but this often leads to memory bandwidth saturation. Faster but small memories could be used if the total memory requirement of all threads is known.

1.3 Thesis Overview

This thesis describes code generation methods leading to design points that maximize performance subject to platform constraints. As previously explained, it is often too late to restructure the application after the compilation step. Hence, it is beneficial to preserve the flexibility exposed initially through the program structure, especially the information that captures the parallel structure of the computation. During code generation, the original computation blocks can be compiled separately. The resource information from each block can be further

utilized to adequately map the pre-compiled blocks of the application, in order to match the constraints of the underlying architecture.

Throughout this thesis, the code generation steps are reorganized as shown in Figure 1.1b. The flexibility in the application structure is preserved beyond an initial partial compilation. Consequently, the application structure can be modified during the iterative mapping and optimization steps. Finalizing the mapping decisions and committing the application structure are deferred until the final compilation. These additional restructuring opportunities can enhance the accuracy of resource utilisation. Including the mapping step into an integrated code generation method is a major departure from the traditional code generation, where mapping precedes compilation.

Data flow computing or streaming programming models are suitable to express applications in a platform independent manner [12, 84]. These models also expose a tremendous amount of parallel code structure. For both GPUs and FPGAs, there are significant opportunities for performance improvement if the code generation starts from a streaming programming model which exposes a flexible application structure. StreamIt [84], a recent hierarchical streaming language, has been selected as an input programming model, without loss of generality. Among the major advantages of using this language, the most relevant are its high level of abstraction, its finer granularity, expressiveness and possibility to use complex structured communication primitives. Its hierarchical structure naturally augments the flexibility in reorganizing the application.

Alternative stream programming models capture an increasing range of applications [73]. A relevant, recent example is the OpenCL programming model [48], which was originally designed for CPU-GPU platforms, and which is now extended to target FPGAs. If this succeeds, it will provide an alternative streaming model which supports the same target platforms as the methods described in this thesis. However, OpenCL provides a weaker semantics for communication between computation blocks, and this penalizes global transformations of the application structure.

This thesis describes code generation methods that start from the StreamIt

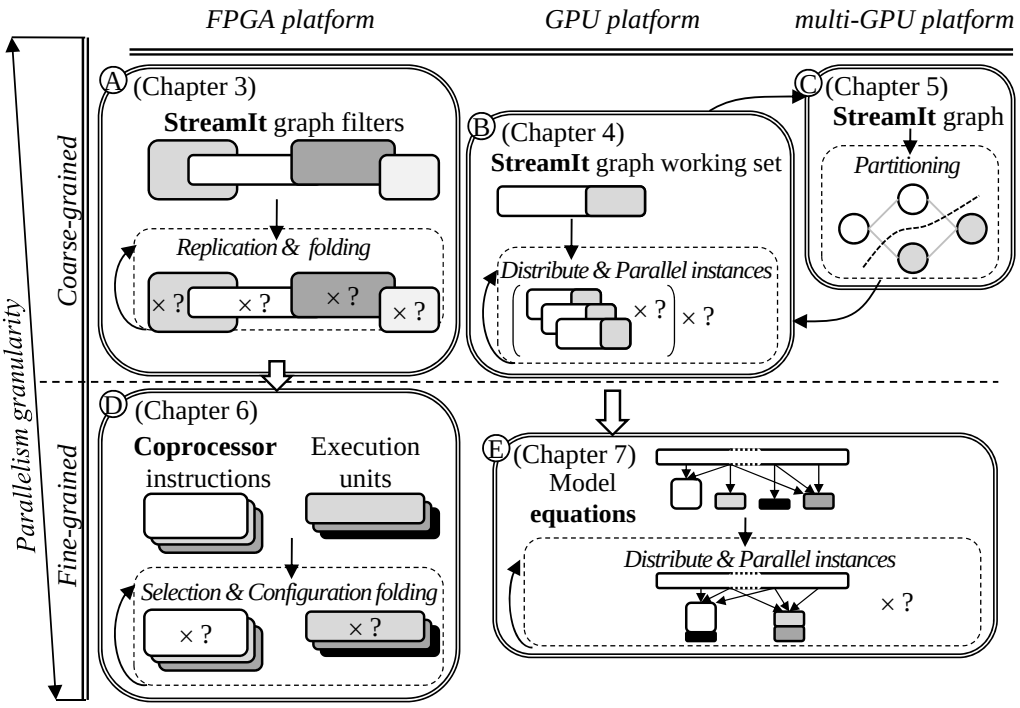


Figure 1.2: Thesis road map.

parallel application representation and target FPGA and GPU devices. The GPU code generation method also supports multiple GPU devices connected to a host CPU. Large amounts of *coarse-grained* parallelism is extracted from the StreamIt programming language. This parallelism is exposed through parallel and pipelined filters in the stream graph representation, and also extracted from the execution model.

The methods described in this thesis are extended to finer-grained parallelism usually exposed by specialized models and libraries. *Fine-grained* parallelism can be identified by the processor at run-time, or it may be exposed by the compiler, through SIMD or VLIW instructions. Significant hardware resources are required to identify parallel instructions in the former case, and yet the amount of parallel operations identified at run-time is affected by how the compiler schedules the code instructions. Usually a mix of platform and compiler support is required to fully utilize this type of parallelism. Based on this observation, this thesis employs an algebra library to expose fine-grained parallelism in vectorizable code, and describes an FPGA-based code generation method that generates custom floating-point SIMD coprocessors which utilise

the exposed parallelism. Complementary, on GPU, the thesis shows how to utilise the fine-grained parallelism exposed by a set of equations backed by a shared working set, and describes a method that generates code to support the parallel execution of these equations.

Although seemingly unrelated, FPGAs and GPUs share a number of similar characteristics from the point of view of this thesis. The most noteworthy of these is their ability to support broad parallelism with tightly coupled threads. The granularity of these threads also covers a large spectrum of applications. For both platforms, these advantages are throttled by tight resource constraints which have to be accounted during code generation.

1.4 Contributions

This thesis proposes a novel approach to integrate mapping and platform-specific compilation to maximize performance for FPGAs and GPUs. Figure 1.2 indicates how the code generation strategy described in Figure 1.1b is projected to the target platforms. It also indicates the parallel granularity of each contribution. The following is a list of contributions included in this thesis:

(A) a novel code generation method for FPGA platforms [38], which starts from a StreamIt graph, and determines the amount of *replication* and *folding* for the graph filters, such that it maximizes the throughput of the application under global resource and latency constraints; this approach utilises coarse-grained parallelism exposed by the StreamIt graph.

(B) the first code generation method for GPU platforms [36] which introduces heterogeneous threads in order to cope with resource limitations. This method takes into account the tight memory constraints of the platform and determines how many *parallel instances* of the StreamIt graph can store their working set in memory, and how to *distribute* the execution of these instances, as well as their working set, in order to increase the throughput.

(C) a scalable extension of the above method, which targets a platform containing multiple GPUs connected to a host CPU [43]; this extension relies

on the single GPU method to determine a feasible set of *partitions*; this method lifts most of the limitations that appear in the single GPU version.

(D) a novel co-design method which analyses the fine-grained parallelism available in vectorizable code, and generates a configurable SIMD floating-point coprocessor that boosts application performance. The customization is the first to allow coexisting vectors with different lengths. The proposed method *selects* which vector instructions are supported, and how their operations are *folded* onto a custom configuration of execution units [37].

(E) an improved code generation method that analyses both the coarse-grained as well as the fine-grained parallelism exposed by a systems biology application, maps *parallel instances* of this application, and *distributes* fine-grained code blocks to a set of threads which share a common working set.

1.5 Outline

Chapter 2 provides a detailed background of existing code generation solutions for the platforms of interest. This chapter also includes details regarding the StreamIt language. Chapter 3 presents the first method that applies to StreamIt code generation for FPGA platforms. The next chapter presents a method that generates GPU code for StreamIt. This method can be extended to a multi-GPU platform as described in Chapter 5, with emphasis on scalability. This is followed in Chapter 6 by a FPGA contribution, complementary to that in Chapter 3, for finer-grained parallelism, that generates SIMD coprocessors for the FPGA platform. To justify the generality of the method introduced in Chapter 4, Chapter 7 presents code generation for a model exposing finer-grained parallelism. Chapter 8 concludes this thesis.

CHAPTER 2

BACKGROUND AND RELATED WORK

Chapter 1 indicated that the streaming programming model exposes a significant amount of parallelism that can be used for efficient code generation. Indeed, previous research shows that streaming programming languages [9, 12, 27] have been successfully utilized to describe applications for parallel platforms. This chapter presents relevant work related to code generation for StreamIt applications. Background regarding the StreamIt language and previous code generation attempts are described in Section 2.1.

This thesis describes code generation methods for the FPGA and GPU platforms. Therefore, this background chapter provides a description of the architecture of each of these platforms. Exposing a reconfigurable structure, the FPGA architecture has been actively used by the research community in application acceleration, by implementing either custom processors or dedicated computation blocks. FPGA circuits are prone to implement applications with a high degree of parallelism, but are subject to tight capacity (resource utilisation) constraints. Relevant work on automatically generated code for FPGAs is presented in Section 2.2.

The GPU architecture follows a different paradigm. It can also handle applications with a high degree of parallelism, but it imposes tight constraints on the resources shared by the parallel threads. Due to the complexity involved, modeling and experimentation have been the norm in writing efficient applications. Because actual GPU code performance is difficult to estimate, automatic generation of efficient code has raised increased interest in the research community, as shown in Section 2.3.

2.1 StreamIt: A Parallel Programming Environment

Stream processing is a data-centric execution model which represents an important class of applications that spans telecommunications, multimedia and the Internet. The compilation of the streaming programs has attracted significant attention because of the parallelism they expose. Languages, tools, and even custom hardware for streaming have been proposed, some of which are commercially available.

The StreamIt language [84] is a hierarchical streaming programming language and infrastructure built upon the experience of a large spectrum of previous streaming languages such as Lustre [14], Esterel [9], Brook [12], Streams-C [27], etc. StreamIt is built on top of the synchronous data flow model [52].

2.1.1 Language Background

StreamIt was designed to expose the parallel and pipelined nature of the streaming applications. The high-level structure of a StreamIt program is a hierarchical graph whose leaf nodes are *filters* which communicate through data channels. Filters can be combined to execute in *pipelines*. The flow of data can be distributed using *splitters* and *joiners* that describe parallel execution paths in the application. These constructs expose coarse-grained parallelism in the application.

Filters are written in C-like code with special constructs to access their input and output channels. A filter consumes data from an input channel using *pop* constructs and produces data on the output channel using *push* constructs. An example filter declaration, with different input and output data rates, is filter $F1$ in the example below.

This example includes a pipeline $P1$ which connects the output of filter $F1$ to a subsequent splitter. This splitter and a joiner are encapsulated in a *splitjoin* construct. The splitter is instructed to route alternative elements, using a roundrobin scheme, to the pipelines $P2$ and $P3$ which it encapsulates. The results of the pipelines are combined, in the same order, to form the output of the splitjoin construct. An alternative splitter policy exists, where all the paths duplicate the same data.

```
int->int filter F1(int N) {
  work pop N push N/2 {
    for (int i = 0; i < N/2; i++) {
      int x = pop(); // read/dequeue from input FIFO
      int y = pop();
      push(x-y);     // write/enqueue to output FIFO
    }
  }
}

int->float pipeline P1() {
  add F1(2);
  add splitjoin {
    split roundrobin(1,1);
      add P2();
      add P3();
    join roundrobin(1,1);
  }
}
```

The StreamIt compiler flattens the hierarchical stream program to a set of base *operators* (filters, splitters and joiners). It produces a schedule that consists of a sequence of operators, and the number of times they are executed (*fired*). Note that multiple firings may be necessary, because filters are allowed to have non-matching input and output rates and hence the elements produced by one filter's firing may require multiple firings of the consumer filter. These multiple firings describe data parallelism in the streaming application. In the above example, as $F1$ produces a single element, the derived execution schedule must include pairs of executions of $F1$, in order to produce one element for each of the two subsequent pipelines $P2$ and $P3$.

The schedule may require an initialization part which is executed once when

the program is launched. Apart from the initialization part, the resulting schedule consists of a steady-state component that can be executed as many times as required to process all the given input.

Dependencies between filters are made explicit by the communication channels. Each filter has its own control logic and an independent address space, and it executes repeatedly as long as a sufficient number of tokens are available on its input channels. However, the filters have the capability to *peek* data from the input channel beyond what they are going to consume. This feature allows structured data dependencies between consecutive filter firings. Peeking is useful for sliding-window computations, and provides an opportunity to rewrite filters that otherwise require internal state to preserve previous values.

A few features in StreamIt may introduce unstructured data dependencies which would prevent parallel code generation. Their usage is not supported by the code generation methods described in this thesis. These features include feedback loops and portals, which create cycles in the stream graph. If these constructs are eliminated, the stream graph can always be flattened to an acyclic directed graph. The code generation methods also require filters with statically defined rates in order to derive the schedules statically.

2.1.2 Related Work on StreamIt

Since its introduction [84], StreamIt has been ported to several distinct platforms. The parallelism it exposes makes it a natural candidate for programming parallel platforms. Each filter in StreamIt declares its data input and output rates. This explicit information enables many optimizations that can yield efficient implementations of the stream computation onto platforms with a high degree of parallelism.

The Raw platform back-end [31] introduces several load balancing optimizations. *Fission* is utilized to split a filter's contents into a pipeline of finer-grained filters. Such a pipeline may achieve better load distribution between parallel threads. In the opposite direction, too fine-grained filters are *fused* together. This optimization also assists with load balancing, as it removes some of the

synchronization overhead, if several filters are to be grouped on the same processing unit.

As a special type of fission, a filter can be *replicated* [31, 63] in order to expose more parallel instances to the compiler. If the number of filters is smaller than the number of processing cores, the compiler replicates the filters with the highest computation requirements. While this strategy works well when generating code for a platform with a finite number of compute engines, it is not clear how to adapt it for platforms where the number of independent processing cores can not be modelled independently from the application. This thesis relies on an extended version of this optimization. Replication appears in different methods throughout this thesis, and is backed up by special orchestration, which allows structured usage of arbitrary replication factors. The reverse, where replication has to be rolled back is called *folding*.

Later, StreamIt has been ported to multi-core processors [30]. This back-end emphasizes on additional challenges of the code generation problem. Despite exposing a significant amount of task and data parallelism, optimizations are often hindered by communication costs. In this context, careful consideration has been given to match the cache size of the underlying processors to prevent performance degradation of operators executed on the same processor.

This back-end has described other trade-offs involved in the execution of the derived stream schedule among multiple cores. It differentiates between *software pipelining*, which pre-encodes a static schedule on each execution core and *hardware pipelining* which relies on computation driven by dynamic data arrival. Software pipelining is found to be suitable on the shared memory architectures utilized. In contrast, Chapter 3 shows that hardware pipelining can significantly reduce latency for the FPGA architecture, where communication is implemented with dedicated channels.

With the emergence of new parallel platforms, a StreamIt back-end has been proposed for the Cell platform [51]. The integer linear programming solution employed to map StreamIt to this platform targets maximum throughput based on the modelled computation and communication overhead. It generates a software

pipelined schedule which attempts to overlap communication with computation.

The architectures of all the platforms presented above share a common characteristic. They utilize a fixed number of cores capable of executing threads independently. However, both the FPGAs and GPUs diverge from this characterisation and introduce global interrelations between the implemented threads. An FPGA mapping can vary the number of parallel computation blocks based on the size of the reconfigurable resources they utilise, while a GPU mapping has to consider the complex relations between parallel threads that impact their performance. Prior implementations of StreamIt to FPGAs and GPUs are discussed in Section 2.2.1 and 2.3.1.

2.1.3 Benchmark Suite

The StreamIt compiler provides a suite of standard benchmarks [80]. These benchmarks describe realistic stream graphs and have been utilized throughout this thesis. To adjust the workload included in the benchmarks, the benchmarks allow parameterization. Table 2.1 describes the benchmarks and how they were parameterized.

2.2 FPGA Architecture

FPGA platforms expose a parallel architecture that consists of a large number of reconfigurable gates that can be reprogrammed to accelerate application-specific code. A broad class of applications, including multimedia, networking, graphics, and security codes, provide ample opportunities to exploit FPGA-based acceleration.

FPGA performance is drawn from the flexibility of its reconfigurable gates, called Look-Up Tables (LUTs)[1]. The LUTs are generic multiple input logic functions with 5 or recently 6 inputs, and 1 or 2 outputs. The configuration of the LUTs can be changed at run-time through FPGA reconfiguration. These gates are connected to each other through a reconfigurable interconnect. Together, the LUTs and the interconnect form a fully reconfigurable architecture which can provide operating frequencies up to 400 MHz.

[1] Xilinx terminology is used throughout this thesis.

Table 2.1: Benchmark characterization.

Benchmark	Description
Bitonic(N)	Sorting algorithm for N float elements applying the bitonic algorithm
BitonicRec(N)	Same as above, recursive method
DCT(N)	Discrete cosine transform followed by the inverse transform for a matrix of $N \times N$ floats
iDCT(N)	Inverse discrete cosine transform for a matrix of $N \times N$ floats
DES	DES encryption algorithm with N rounds, input 8 bytes, output as 16 hex digits
Serpent(N)	Serpent encryption algorithm with N rounds, it includes a bit level linear transform
FFT(N)	Fine grained FFT transform on N float elements
FFT'(N)	Very fine grained FFT transform on N float elements described in Appendix A
FilterBank(N)	Instantiates N filter banks to process multirate signals
FMRadio(N)	$(N + 3)$-band equalizer radio
MatrixMult2(N)	Blocked matrix multiplication algorithm for $2N \times 2N$ matrices, split into blocks of 2×2
MatrixMult2(N, M)	Blocked matrix multiplication algorithm for $(2N \times 2N) \times (2M \times 2N)$ matrices, split into blocks of 2×2
MatrixMult3(N)	Same as above for $((3N+3)\times(3N+3))\times(3N\times(3N+3))$ matrices, with blocks of 3×3

The configuration of an FPGA is usually determined through hardware synthesis. The circuit is described in a high-level hardware description language (HDL) such as Verilog or VHDL [79] and further processed by vendor-specific tools. It is first synthesized into a netlist, which matches the characteristics of the LUTs and other reconfigurable resources in the target FPGA, and further fitted to the actual circuit layout, which fixes the placement and routing of each resource. Both steps take a large amount of time, in the range of hours, and they are often seen as the most significant factor limiting the popularity of FPGA technology.

Besides LUTs, the FPGA architecture now contains other reconfigurable resources, such as memories, DSP blocks, clock generators and even hard-wired processor cores, all of which can be included in user designs. Utilising these pre-defined hard-wired resources increases the performance of the synthesized application.

Various strategies are employed to reduce the design synthesis time. Among

these strategies, manually added annotations are the most frequently utilised, as they allow the circuit designer to fine tune the implementation. However, in the context of automatic HDL generation, such an option is infeasible. Instead, HDL generation tools often utilise libraries of pre-synthesized components which can be combined in larger designs, and which rely exclusively on the capabilities of the vendor synthesis tools in order to improve the performance of the resulting design.

In this context, applying automatic replication or folding strategies, as described in Chapters 3 and 6, exploits the possibility of duplicating or sharing not only the HDL code, but also the synthesized version of the code. As the parallel granularity of the FPGA resources is fully customizable, the applicability of the folding strategy is possible for several levels of parallelism. Replicating synthesized modules ensures balanced circuits capable of higher performance. The size and performance of the application which can run on the FPGA are only limited by the total available resources.

2.2.1 Related Work on FPGA code generation

There are several platforms that integrate FPGAs with hard-wired processor cores [1, 29, 61, 81], and recent announcements [21] from leading vendors suggest that FPGAs are likely to become widely available as programmable coprocessors. Sequential parts of the applications can be assigned to run on the host processor, while those parts with abundant parallelism can pass through code generation methods that lead to FPGA implementations. These application parts can expose parallel computation, which is fine-grained (i.e. data parallel paths), or coarse-grained (i.e. parallel tasks).

2.2.1.1 Fine-grained Parallel Computation on FPGAs

Fine-grained parallel computation is usually implemented as *custom instructions* that extend a given processor core. Previous research has shown how custom instructions can be added to an existing processor in a systematic approach. A number of commercial products are available, such as those developed by Tensilica [29] and Stretch [81].

The typical approach, used to automatically generate custom instructions, involves the analysis of the data flow graph obtained as a result of compilation, followed by the enumeration and selection of sub-graphs as candidates for custom instruction implementation [20, 100]. While the selected sub-graphs are identified during application compilation, resource usage is only estimated [10], and it may be affected by optimizations during HDL synthesis. This phase dependency prevents code generation tools from controlling accurately the reconfigurable resource count of the sub-graphs that would be synthesised. This is particularly important if the size of the custom instructions is large.

Previous research has usually focused on integer custom instructions [29, 100], which are lightweight and must be tightly integrated in the processor pipeline to achieve high performance. However, the overhead of such an approach is small only if the processor core resides in the FPGA reconfigurable resources, as well. If this is the case, the performance of the entire processor is offset by the implementation of the processor core in reconfigurable resources.

An alternative option becomes viable for floating point instructions. Because floating-point operations are usually supported by a bulkier implementation, their integration in the main processor pipeline can be less tightly coupled [97]. In this case, the processor core may be hard-wired, and only the floating-point instructions are implemented in FPGA. However, the number of floating-point pipelines that can be implemented in hardware is small, and resource sharing can certainly improve the designs. Therefore, the code generation for custom coprocessors, described in Chapter 6, combines custom instructions with resource sharing into folding methods. These methods exploit the regular structure of the vector instructions in order to generate automatically resource-constrained implementations.

While sharing methods have been previously applied to custom instructions [10, 83], the regular structure exposed by vector integer instructions is not suitable for sharing, due to the considerable cost of the multiplexers required for sharing purposes, compared to the size of the fine-grained integer operations. Indeed, the custom integer vector instructions offered by Tensilica [29] do

not share the operations, hence they exhibit only limited resemblance with the method described in this thesis.

However, there are also a number of customized floating-point SIMD processor architectures [29, 85, 92, 99]. They provide a rich set of reconfigurable parameters. However, the final result is a monolithic processor instance with all instructions tightly integrated into the base pipeline. As such, these processors can not take advantage from fine-grained application-specific parallelism, and they are suitable to be implemented either in silicon or entirely as soft-cores [58].

Lastly, some vendors already offered hard-wired SIMD floating point coprocessors for their embedded processors [57]. The iPhone, for example, includes such a core [45]. This is additional evidence for the growing importance of floating point computation in a design domain characterized by tight resource and performance constraints. However, these coprocessors are silicon-based, and hence do not possess the flexibility of the solution presented in Chapter 6.

2.2.1.2 Coarse-grained Parallel Computation on FPGAs

As discussed in the previous section, there is some overhead associated with the attachment of fine-grained FPGA computation to a hard-wired processor core. Coarser blocks, such as hardware *loop accelerators* [77, 104] have been proposed, relieving the processor of the steady issue of instructions and operands. Using this method, loop specific optimizations such as unrolling and pipelining can be used to improve the efficiency and utilization of the hardware execution units. Several tools [19, 34] are capable of deriving dedicated loop accelerators from the application code by applying static transformations to extract the necessary data parallelism. These methods, however, do not support irregular loop structures or complex control flow. In addition, dedicated memory connections are required to provide data for the loops. The method presented in Chapter 6, on the other hand, relies on the core processor to resolve all dynamic control flow and the data transfers, issuing scheduled vector instructions and operands in the proper order to the hardware.

Exploiting data parallelism through replication can increase the throughput

of the computation blocks [13, 18]. The replication applied to StreamIt operators in Chapters 3 addresses this issue in the context of the FPGA platforms. A method is described that performs maximal replication of the operators bound only by the size of the FPGA, then folds back those that do not improve throughput.

This replication method does not address the synthesis of the actual computation from stream operators to HDL. The emphasis is on the composition of the synthesized operators into an overall space-time efficient design. Recent work [41] specifically addressed the issue of hardware generation from StreamIt, and this method is orthogonal to it. Similarly, many of the existing state of the art C-to-hardware compiler technologies can be used to complement this method. Hence the method described is complementary to most of the ongoing research in the community that address sequential code high-level synthesis.

The replication strategy improves the accuracy in modelling the communication overhead. Because the replicas are identical, the data routing is simplified and the associated overhead is more accurately accounted for. This improves the global performance of the generated code. The modularity and composability of this method distinguishes it from global optimization of loop nests [103].

2.3 The GPU Architecture

The GPU platforms have a massively parallel architecture that allows the concurrent execution of thousands of threads. The architecture consists of a number of streaming multiprocessors (SM), which in turn contain a number of processing cores. The number of processing cores in each of the streaming multiprocessors continues to increase with each new generation of GPUs(up to 48 cores per SM in the most recent nVidia GPU, compared to S2050's 32 cores and S1070's 16 cores).

The processing cores are running in lockstep, similar to SIMD execution. Blocks of parallel software threads run on each of the available SM. Typically, there are much more software threads than there are processing cores. In order to schedule the many threads on SM, they are statically grouped into scheduling

units called *warps*[2]. For the current generation of nVidia GPU, a warp consists of 32 threads. Because threads in a warp execute in lockstep on the processing cores, any intra-warp control flow discrepancies will lead to serialized execution. However, threads that belong to different warps are independent of any divergent control flow penalty.

A hardware scheduler typically selects one warp and issues the current instruction from all its threads onto the pipeline of the processing cores in 2 - 4 consecutive cycles [94]. Afterwards, this warp becomes unavailable for a number of cycles until its instructions clear the pipeline. The scheduler switches to execute a different warp with zero overhead. As a result, though a large number of parallel threads can be spawned, their executions are actually *interleaved* on the processing cores. As opposed to CPU, where advanced compiler and run-time support is necessary to extract the fine-grained parallel operations, the GPU scheduler can simply issue, in parallel, independent instructions from inherently parallel threads.

The GPU architecture benefits from an exposed memory hierarchy where threads explicitly specify which memory they access. All threads can access off-chip global memory. However, the latency of accessing this memory is high. In addition, each SM in a GPU contains a small but very fast on-chip memory that is shared among all the threads in the SM. This *SM memory*[3] has close to register latency.

The register file is distributed among all the threads of the GPU. Hence, instantiating more threads leads to fewer registers allocated to each thread. This may lead to spills, which are directed to a *local memory*. Unfortunately, local memory is backed by private areas in the long-latency global memory, and performance is again significantly affected.

The long stalls affecting a warp that accesses global and local memory can be partially hidden if the scheduler can launch enough alternative warps. However, the architecture is not able to sustain execution without stalls when all warps

[2]nVidia terminology is used throughout this thesis.
[3]The nVidia way of referring to this as *shared memory* is potentially confusing.

access the memory simultaneously. This observation suggests that the GPU scheduler would benefit from a mix of threads with different execution patterns and from reduced memory access rate.

In this context, the methods described in Chapters 4, 5 and 7 show how the parallelism extracted from the application can be utilized to provide a steady number of stall-free warps to the scheduler, hence hiding most of the global memory latency. It also shows how the finer-grained parallelism in the code can be utilized to reduce the ratio of computation to memory access.

2.3.1 Related Work on GPU code generation

Computing on GPU platforms involves kernels that usually communicate to each other through global memory. Therefore, the overall performance is limited by the high latency of memory access. Hence, memory latency hiding is one of the most significant concerns in GPU programming. Basic strategies that enhance the memory access for a variety of GPU applications are detailed in [69].

Selecting the right number of parallel threads and the location of frequently used data is not trivial [75]. One well-known approach that boosts performance is to prefetch data from global memory to SM memory [98]. This is the approach taken by other high-level language translations [8, 74, 93] to CUDA and OpenCL.

The method presented in Chapter 4 uses two classes of dedicated threads for: (1) loading / storing data from global memory to SM memory and (2) computing using data preloaded in SM memory. A recently proposed method [42] exploited efficiently only the coarse-grained task parallelism exposed by StreamIt, while the method presented in this thesis also takes advantage of finer-grained data parallelism when generating code for the stream graph. Therefore, a single instance of the stream graph spans several computing threads.

Because the amount of SM memory is limited, it is necessary to reduce the working set footprint. When generating GPU code for StreamIt, two complementary methods are possible. One relies on caching transformations for StreamIt that have included narrowing the memory requirement through modulation or copy-shift [78]. The other is to use a scratchpad memory, as optimal algorithms

have been proposed for its management [53]. The method in Chapter 4 is based on the copy-shift method, adapted to the way the stream graph executions share a common memory.

StreamIt applications have been previously executed on GPU platforms [42, 89]. The stream graph is usually mapped directly to kernels encapsulating operators that communicate via global memory. Mapping communication to global memory penalizes performance and eventually saturates the memory bandwidth. In order to reduce run-time overhead, communication between SM executing different kernels has to be deferred until a large amount of data is processed locally. As a result, the latency of executing the stream graph is large, while the throughput is limited by the memory bandwidth, despite the use of pipelining.

The methods described in this thesis generate code encapsulating stream graph partitions. Instead of mapping each operator separately, they execute multiple instances of larger stream graph partitions in parallel on each SM, taking care to adjust the number of parallel instances to match the resource constraints. The aim is to achieve a balance between the number of GPU threads, the layout of the SM memory, and the memory bandwidth consumption, such that performance is maximized.

A promising solution to deal with scalability issues is the utilization of multi-GPU platforms. Such systems are well-suited to process large data set applications [82]. Performance modeling for GPU architectures was comprehensively investigated by analytic and quantitative approaches [4, 101, 40] which highlighted the important balance between computation and memory access, as well as the utilization of SM memory. It is possible to estimate statically the performance of an application running on multiple GPUs based on characterizing computation and different communication costs [76].

On the other hand, efficient run-time systems for multiple GPUs have been proposed to explore speculative execution [22] and to investigate load balancing [17]. None of these works has attempted to generate code automatically, nor to provide an execution model for streaming languages onto multiple GPUs.

CHAPTER 3

STREAMIT CODE GENERATION FOR FPGAS

The first contribution described in this thesis tackles the optimized code generation of StreamIt applications to FPGA platforms. The architecture of these platforms does not directly constrain the degree of parallelism that can be derived from the application code. However, there are several other significant challenges to FPGA code generation for streaming applications. Since FPGA platforms have finite reconfigurable resources, there are many non-trivial trade-offs between the performance achieved and the number of reconfigurable resources utilized.

In addition, the performance of such an application is not reflected only by its throughput. Different design domains may trade throughput for the overall latency of the computation. The latency, defined as the time lapsed between the moment when an input appears at the input of the FPGA, until the moment when a corresponding output is produced, is an important constraint in application domains such as real-time control [88], network and media applications [101] as well as in the financial domain, for high frequency algorithmic trading [96].

This chapter describes a code generation method that takes StreamIt programs and generates HDL code suitable for FPGA implementation, with focus on the improvement of the high-level mapping steps. The optimized code generation method includes an algorithm that assists with the refinement of the stream graph applications. The design points processed are further refined for the highest achievable throughput subject to user-specified latency constraints and target FPGA resource bounds.

Starting with an application represented in StreamIt, ample parallelism is available due to the stream-oriented programming model. This chapter addresses the following question: is there a refinement of the flexible input stream graph

that can maximize the processing throughput of the overall graph? Furthermore, because FPGA reconfigurable resources are finite, and latency is typically an important consideration in this application domain, the optimization goal is extended to consider both resources *and* latency constraints. The throughput improvement algorithm described is the first tackling the combined constraint.

The intuition behind the algorithm is the following. The filters that may cause bottlenecks in the stream graph are identified and inspected. If the filters do not maintain a history of their past execution, then their throughput can be boosted by exploiting automatically the data parallel properties they expose. This is achieved by judiciously *replicating* the bottleneck filters.

Replicating the filters has several advantages. The replicated filter instances do not require to be synthesised again as they are all instances of the same filter, and the synthesis results are reusable. This is in contrast to prior work on global optimization of loop nests on FPGAs [103] which requires recompilation and evaluation of the recompiled designs based on heuristics. Such an approach will not scale for large designs.

The algorithm operates on a stream graph and relies on a previously synthesized set of filters. It determines how to assemble the pre-synthesized filters in order to achieve the best possible throughput. If a filter is replicated, additional code is automatically generated for specific hardware circuitry required to route the data flow to and from the replicated filters. This method makes the issue of filter synthesis orthogonal to design assembly and generation. Hence, this method is complementary to a lot of the ongoing research in the community that addresses high-level synthesis of the filter code itself.

The algorithm can be briefly described as first aggressively replicating candidate filters, then folding back the graph to reduce the number of replicas if they are not profitable given the constraints. The next section provides a motivating example that shows some of the trade offs considered by the code generation method. Subsequently, the details of the replication and folding algorithm are discussed, together with the evaluation results.

3.1 Rationale

A stream graph example is shown in Figure 3.1. In this figure, F_i are filters, and the hardware footprint ($R(i)$, reconfigurable resources utilized) of each filter is correlated to the area of its corresponding rectangle, while the execution time $T(i)$ of a filter firing is correlated to the length of the rectangle. The figure represents the number of input tokens popped and pushed by each filter. The edges in the graph describe channels routing the data flow between operators. The splitter and joiner are illustrated using arched double-headed arrows. Section 2.1.1 provides a complete description of the StreamIt language.

The data flow is split between F_1 and F_2 in a periodic and round-robin manner, with 3 tokens dispatched to F_1 and 6 to F_2. This information is annotated on the edges that fan-out from the splitter. Similarly, the joiner collects data from the input streams in a round-robin manner. The weight annotations on each edge describe how the data is aggregated from the streams: 3 tokens from F_1 and 3 from F_2.

This example contains a pipeline stream container, connecting a splitjoin to F_3. The splitjoin is a stream container, with a splitter at the source, a joiner at the sink, and filters F_1 and F_2 between them. The stream graph in the figure can be described as follows in StreamIt:

```
int->int pipeline Example() {
    add pipeline {
        add splitjoin {
            split roundrobin(3, 6);
            add F1();
            add F2(2); // instantiating parameterized filter
            join roundrobin(3, 3);
        }
        add F3();
    }
}
```

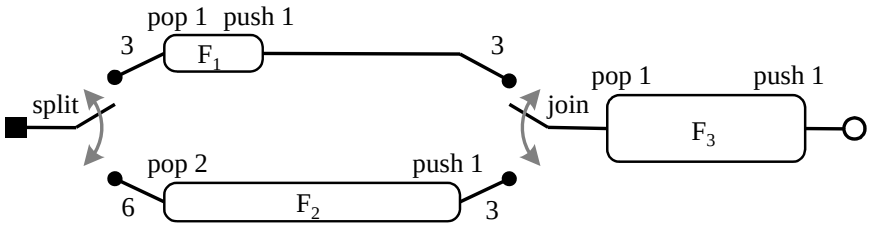


Figure 3.1: An example stream graph.

Conceptually, operators fire autonomously and concurrently. Since there is no data dependency between F_1 and F_2, they can fire in parallel. The firings of filter F_3 can be pipelined relative to the other filters. It is obvious from the graph that F_1 and F_2 execute the same number of times in order for F_3 to fire (that is, both F_1 and F_2 fire 3 times to produce the requisite amount of data for the joiner, which ultimately provides the data to filter F_3).

The code generated ensures that each of the operators in the stream graph executes only when there is a sufficient number of data tokens on its input edge. However, the implemented execution model does not enforce a synchronous schedule, but allows filters to execute ahead of time, if enough elements are available at a filter input. Because the performance of the asynchronous approach is not worse than the synchronous execution, analysing the latter is sufficient to derive performance guarantees.

A StreamIt program exposes the flexible program structure and communication topology to the code generation tool, which can decide on the best implementation choices based on the target hardware architecture. A naïve code generation, instantiating the operator structure as it is described in the stream graph does not produce an efficient implementation: a single instance of filters F_1 and F_2 is not load-balanced. However, if a filter has no internal state – that is, it does not maintain any history of its previous executions – it can be replicated in order to achieve a more load-balanced implementation. Replicating a filter creates several instances of the filter and adds a splitter to distribute data between the filter and its replicas, and a joiner to collect the results. The replicas effectively increase the firing rate capability of the filter, but also increase the reconfigurable resource usage: each of the replicas incurs a resource overhead

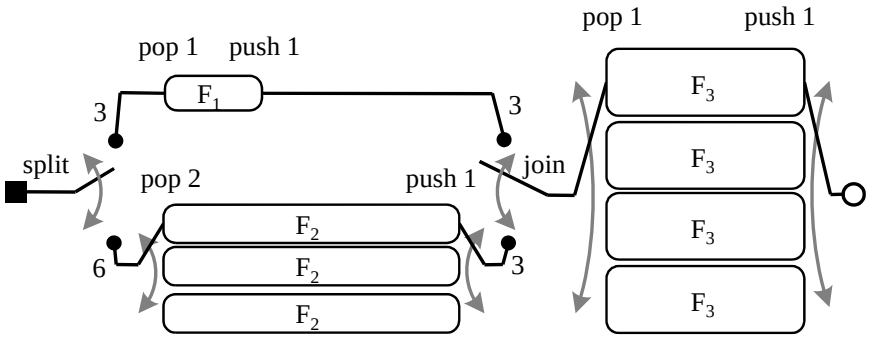


Figure 3.2: A stream graph with replicated filters that achieves maximum throughput, subject to resource constraints.

that is equal to the original instance, and in addition, there is an added overhead incurred by the splitter-joiner pair that routes the new data flow. Since the target FPGA architecture has finite reconfigurable resources, careful judgement is necessary to decide which filters should be replicated and to what extent. The algorithm replicates and folds a given stream graph to determine where and to what extent replication will be most profitable.

The primary contribution of this method compared to related work described in Section 2.2.1 is the co-optimization of space and time (throughput and latency). The stream graph in Figure 3.2 illustrates the replication of filters F_2 and F_3. The graph achieves the best throughput to reconfigurable resource usage ratio: filters fire continuously, making efficient use of the hardware. At steady state, the throughput of the joiner aggregating the outputs of F_1 and F_2 is three times higher than the corresponding joiner in the original graph shown in Figure 3.1. A hardware design and implementation of a stream graph that uses replication increases throughput, but may also affect the latency of the computation.

Higher throughput does not guarantee a minimum latency design. The latency is determined by assuming that sets of data items are available when needed at the stream input and then determining the maximum time required to generate all the corresponding outputs. While a unique replica of F_1 can handle its three input data sets in consecutive runs, because its firing time is short, the result of the second and third firing will still occupy two replicas of

F_3 when the results of F_2 become available, creating backlog. The joiner will pass onwards this burst of data to the replicas of F_3, but they will not be able to process all of it immediately, and at least one element will be delayed in the channel at the input of F_3, hence increasing the latency.

Figure 3.3 presents an alternate stream folding strategy that trades some throughput in order to achieve a lower overall latency. Due to the additional replication of F_1, all the results processed on that data path are available earlier and can be processed immediately, in parallel, by the slightly lower number of copies of F_3. In this design, when the results generated by F_2 become ready, the same copies of F_3 are all free and can process them without delay. While the latency has decreased, the sustained throughput of this design is also reduced because of the reduced replication of F_3 which now represents the bottleneck.

Peeking filters may be replicated as following: the input stream is duplicated to the filter and its replicas, and then each replica discards locally the parts of the stream that are not relevant to it. This approach may create some redundant communication, depending on how much input data is not actually consumed by each replica, but others have shown that it is possible to design efficient hardware mechanisms to exploit the structured data reuse [33].

3.2 Code Generation Method

This method determines which filters to replicate (and by what factor), in order to maximize the processing throughput, subject to reconfigurable resources and latency constraints. The philosophy is to describe the desired design topology, and generate code that stitches together the filters and streams as directed by the algorithm.

The algorithm described assumes that individual filters are already synthesized as part of the code generation method, and both reconfigurable resource usage and profiling information (worst-case execution time estimates) are retrieved from the compilation and profiling of individual filters. If the filters take less time to execute than the worst-case estimate, the correctness of the solution is not affected.

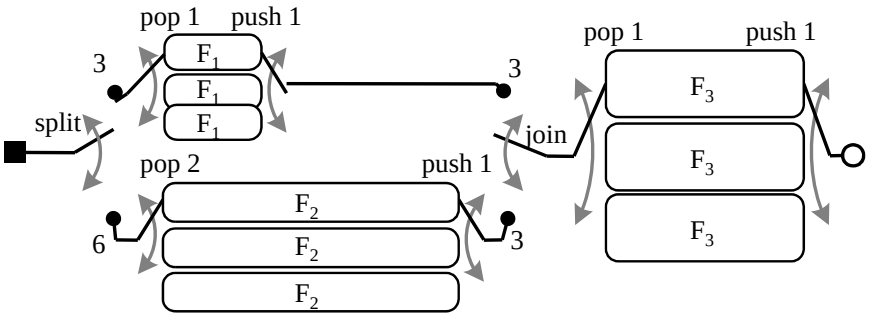


Figure 3.3: Reducing the latency for the graph in Figure 3.2 under the same resource constraints.

The input to the algorithm is a stream graph derived from StreamIt code. In StreamIt, it is the programmer's responsibility to describe the streaming application in such a way that exposes sufficient parallelism. The compiler is commissioned with the task of refining the application graph into a form that is amenable for synthesis. The stream folding strategy manages the complexity of the design search space by exploiting the flexible nature of the stream graph.

Replicating filters with internal state is beyond the scope of this method (although past work has shown it may be profitable to do so [51]), and they impose a performance limit. If such a filter dominates the execution throughput, then replication of other filters is not likely to be profitable. However, in many cases, good code writing practises prevent filters with internal state from growing too large, or they can be easily rewritten as peeking filters which are handled by this algorithm.

The highest throughput design is derived using the steps shown in Algorithm 3.1. First, each filter is analysed and a corresponding work factor is computed by multiplying its firing execution time $T(i)$ and its firing rate (line 2). The firing rate $S(i)$ is derived by the algorithm with support from the StreamIt compiler as described below; it equals the number of firings of a filter so that it is rate-matched to its producer and consumer. The computed *workFactor* combines the two and indicates the proportional amount of replication required by each filter. An initial count *initRes* on the resources utilized by a throughput balanced design point is derived (line 4) utilizing the work factor to scale the individual filter resource requirements $R(i)$.

Algorithm 3.1 Folding a design for throughput

Input StreamIt program graph G, global resource constraint R_{FPGA}, filter firing execution time T, filter resources R

Output Replication coefficients C

1: **foreach** Filter i in G **do**
2: $workFactor(i) = T(i) \cdot S(i)$
3: **end foreach**
4: $initRes = \sum_{i \in G} R(i) \cdot workFactor(i)$
5: $maxScale = \max_{i \text{ has state}} (\frac{1}{workFactor(i)})$
6: $scale = \min(R_{FPGA}/initRes, maxScale)$
7: **foreach** Filter i in G **do**
8: $C(i) = \lceil workFactor(i) \cdot scale \rceil$
9: **end foreach**
10: **while** $\sum_{i \in G} R(i) \cdot C(i) > R_{FPGA}$ **do**
11: $C = reduceThroughput(C, workFactor)$
12: **end while**

The next lines determine the maximum replication factor that matches the global resource constraint R_{FPGA}. Line 5 determines which of the filters with internal state, if any, constrains the replication to *maxScale*; it is the one with the greatest work factor. The first term in the *min* equation on line 6 determines how many of the FPGA resources are available for replication. The resulting *scale* factor determines the replication counts for the initial design point(line 8).

Due to rounding, a design that instantiates the calculated replicas counts may lead to a resource requirement slightly larger than the input resource constraint, although the design will be on the pareto-optimal frontier with respect to throughput and resource requirement. This initial replicated design is refined further, reducing its throughput while maintaining it on the pareto-optimal front of the design space. Each iteration in lines 10-12 reduces the resource requirement of the design by eliminating one filter replica instance at a time, starting with filter replica instances that only marginally improve throughput. This is accomplished in the *reduceThroughput* procedure presented in Algorithm 3.2. Finally, a maximum throughput design that fits the available resources is obtained, and the only step to be determined is if the latency constraint is satisfied.

Algorithm 3.2 selects a pareto-optimal design with the smallest reduction in throughput. As the input design is on the pareto-optimal curve, any reduction

Algorithm 3.2 Throughput reduction procedure *reduceThroughput*

Input Current design point C, work factors *workFactor*
Output New design point C'
1: $p = \min_{i \in G} \frac{workFactor(i)}{C(i)-1}$
2: **foreach** Filter i in G **do**
3: $\quad C'(i) = x | \forall x' < x, \frac{workFactor(i)}{x'} > p, \frac{workFactor(i)}{x} < p$
4: **end foreach**
5: **return** C'

in the replication coefficient $C(i)$ of a filter leads to a reduction in throughput, and a corresponding increase in the minimum initiation interval (or period) p. The smallest such increase is recorded in line 1. The newly derived execution time threshold determines the possible changes in the other coefficients. These coefficients are recalculated for all filters in line 3. This ensures that the new design remains on the pareto-optimal curve.

Algorithm 3.3 generates design points sorted by throughput, starting with the one achieving the highest throughput as constructed in the previous step. Reducing the throughput iteratively (line 10) allows the exploration of other design points that can utilize the additional freed resources to improve the latency of the implementation. The latency can be improved by increasing the replication of a subset of filters.

An approach based on simulated annealing is used to search through the potential candidates while pruning away as much of the infeasible design space as possible. This is achieved using a custom neighbor visit function that avoids illegal configurations defined by the resource constraint, throughput lower bound and the latency constraint (line 4).

A reduced latency design may be found if throughput is slightly reduced. This scenario occurs, for example, if a joiner path delays the data more than p, hence blocking for a while data propagated through the other paths. In other words, the latency constraint ΔT may be satisfied only for initiation intervals larger than that defined by the maximum sustainable throughput of the design explored. Therefore, for offering latency guarantees, a design may be constrained to a throughput below the maximum achievable and this is reflected by the *sustainablePeriod* procedure. Recording the minimum initiation interval

Algorithm 3.3 Latency constrained design folding

Input Best throughput configuration (C), latency constraint ΔT, resource constraint R_{FPGA}
Output Latency constrained configuration (L)

1: $p = \infty$
2: **while** $\max_{i \in G} \frac{workFactor(i)}{C(i)} \leq p$ **do**
3: **if** *feasibleImprovement*$(C, \Delta T)$ **then**
4: **foreach** Configuration C' in *simAnnealing*$(C, R_{FPGA}, \Delta T)$ **do**
5: **if** *sustainablePeriod*$(C', \Delta T) < p$ **then**
6: $L = C'$; $p =$ *sustainablePeriod*$(C', \Delta T)$
7: **end if**
8: **end foreach**
9: **end if**
10: $C =$ *reduceThroughput*(C)
11: **end while**
12: return L

for which a design under analysis matches the latency constraint tightens the lower bound of throughput exploration for subsequent design points (line 5-7). Only designs that can sustain initiation intervals below this maximum initiation interval are tried (line 2) as only these designs can offer both better throughput and additional replication possibilities (more spare resources are available to selectively increase replication) than those previously explored.

As long as a candidate is found in one of the steps of the exploration, the search converges easily, being limited to a few tightly constrained simulated annealing steps. However, if no candidate is found, a larger number of possible designs points may be explored. The design space is pruned using the *feasibleImprovement* procedure, by checking if a resource unconstrained design having the same throughput as the design point analysed can offer the required latency (line 3). This is accomplished by analyzing the latency of a design replicating all filters except the bottleneck by as much as possible.

3.2.1 Calculating Throughput

The hierarchical nature of the stream graphs derived from StreamIt is used to compute efficiently the overall throughput of a streaming program. The maximum input throughput t_{in} and output throughput t_{out} of a filter F_i is defined as follows:

$$t_{in}(i) = \frac{pop(i)}{T(i)},\ t_{out}(i) = \frac{push(i)}{T(i)}$$

where $pop(i)$ and $push(i)$ correspond to the number of data elements dequeued from and enqueued to the input and output channels of the filter F_i, and $T(i)$ equals the number of cycles spanned by a single firing of F_i. Utilizing these results, the throughput of stream containers can be computed hierarchically.

Pipeline and SplitJoin throughput The throughput of a pipeline is equal to the lowest throughout of its filters. Furthermore, since individual filters may push and pop at different rates, the rates observed at different points in the pipeline will vary, although filters have to sustain correlated rates. The throughput limitation imposed by a filter F_i on the output of a pipeline consisting of the filters $\pi = \{F_1, \ldots, F_n\}$ is

$$t^{\pi}_{out}(i) = t_{out}(i) \cdot \prod_{i<j\leq n} \frac{push(j)}{pop(j)}$$

and therefore, the actual output throughput of the pipeline is

$$t^{\pi}_{out} = \min_{1\leq i\leq n} t^{\pi}_{out}(i)$$

.

For a splitjoin $\sigma = \{F_1, \ldots, F_n\}$ where the joiner weights are $(w_1, \ldots, w_n)$, the output throughput is

$$t^{\sigma}_{out} = \min_{1\leq n} \left(t_{out}(i) \cdot \frac{\sum_{1\leq j\leq n} w_j}{w_i} \right)$$

Overall throughput It is possible to apply these relations to the whole stream graph in a composable manner,

$$t^{G}_{out} = \min_{i\in G} t^{G}_{out}(i) = \min_{i\in G} \left(\frac{1}{S(i)\cdot T(i)} \right) = \min_{i\in G} \frac{1}{workFactor(i)}$$

where $S(i)$ is a constant that can be determined hierarchically as above, and which is equivalent to the number of firings in the single appearance schedule [6] generated by the StreamIt front-end. To prove this relation, assume a stream can

sustain a throughput $t' > t^G_{out}$. Propagating this downwards through the stream hierarchy, for all stream containers $\hat{G}$, $t^G_{out}(\hat{G}) \geq t'$. Continuing the stream decomposition and applying this relation down to individual filters, results in $\forall i, t^G_{out}(i) \geq t'$, which is a contradiction.

The replication of a filter has the effect of multiplying its throughput by its replication factor. If $C(i)$ is the replication factor, the modified formula becomes

$$t^G_{out} = \min_{i \in G} \left(\frac{C(i)}{workFactor(i)} \right)$$

and its inverse which characterizes the minimum initiation interval is

$$p = \max_{i \in G} \left(\frac{workFactor(i)}{C(i)} \right)$$

.

3.2.2 Calculating Latency

In general, the stream graphs run on FPGA circuits coupled to host processors. Data transport between the host and the FPGA is achieved through a bulk transfer mechanism (i.e., DMA). In this context, the number p of clock cycles between such transfers is called the *initiation interval*. The minimum initiation interval can be computed based on the reciprocal of the highest throughput sustained by the stream graph. Results are expected to be ready after a time interval ΔT, called the *latency*.

Data-token reordering and local congestion at a filter's input due to non-periodic data arrival are the major factors for latency variation. While replication improves throughput, it often increases the latency. An important problem is to obtain exact latency bounds that can offer guarantees especially for real-time stream performance.

Given the stream graph, this analysis determines a valid set of initiation intervals for which the delays are evolving linearly. There is a finite set of such intervals and they can be computed starting from the minimum sustainable p [35]. The data arrival time at each filter input is a linear expression $\alpha p + \beta$ which can be used to derive the time when a result is generated (also a linear expression). During the analysis, an additional constraint may be generated

Table 3.1: Example latency calculation.

Input	replica 1	replica 2	Constraint	replica 1	replica 2	Constraint
0	$[0,T)$		$p \geq 3T$	$[0,T)$		$p \geq 4T$
0		$[0,T)$			$[0,T)$	
0	$[T,2T)$			$[T,2T)$		
$p/2$		$[T,2T)$	$p/2 < 2T$		$[p/2, p/2+T)$	
$p/2$	$[2T,3T)$		$(p < 4T)$	$[p/2, p/2+T)$		
$p/2$		$[2T,3T)$			$[p/2+T, p/2+2T)$	
	Interval: $[3T,4T)$			Interval: $[4T,\infty)$		

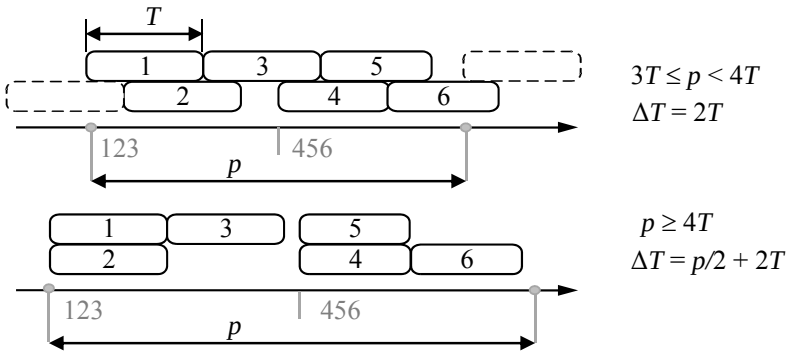


Figure 3.4: Schedule used to determine latency. Six data tokens arrive every interval p. With two replicas, computation occurs in parallel.

on the upper bound of the input initiation interval where this expression is valid. All filters are processed, until obtaining a linear expression of the overall latency of the stream and a constrained initiation interval range where the linear expression of the latency holds. Subsequently, the adjacent initiation interval range is analysed, generating a new constraint on the interval that will lead recursively to new intervals to be analysed.

Table 3.1 shows the computations necessary in case of a filter, replicated two times, receiving two equally spaced groups of 3 input tokens each initiation interval p. The corresponding schedule is presented in Figure 3.4.

The implementation hierarchically iterates over the stream structure, deriving output times based on the input times. In case of replicated filters, it maintains a set of ready times for each replica as linear dependencies on p. The input tokens are already in order and other ordering constraints are generated to ensure that the current replica is ready to fire when its data arrive.

Joiners may add additional reordering constraints, increasing the number of analysed intervals. The automatically generated joiner code can process one

element every clock cycle. Every time the joiners switch to process data from a different input branch, the ordering of the data elements has to be ensured through a constraint. This is one of the main causes of latency increase, because elements usually accumulate in the channel of the other path, hence the joiners usually generates data bursts.

3.2.3 HDL Generation

The distribution and gathering of data to and from the replicas require hardware resources akin to programmed multiplexers. The design which is automatically generated to implement this distribution mechanism is illustrated in Figure 3.5. The design allows a data token to be routed each clock cycle.

This replication logic is included in the resource estimation algorithm to reflect the resources utilized by the automatically generated application code. Because the multiplexers involved are synthesized later with platform-specific HDL synthesis tools, the utilized resources can not be estimated directly based on the requested replication factor and bus width. A library of such multiplexers, with a wide range of replication factors, is generated to cover exhaustively the replication space. Based on this library, a linear model is determined, which estimates the number of utilized resources.

The distribution logic in Figure 3.5 also includes a state machine which maintains information regarding which replica is waiting for input data. Connections

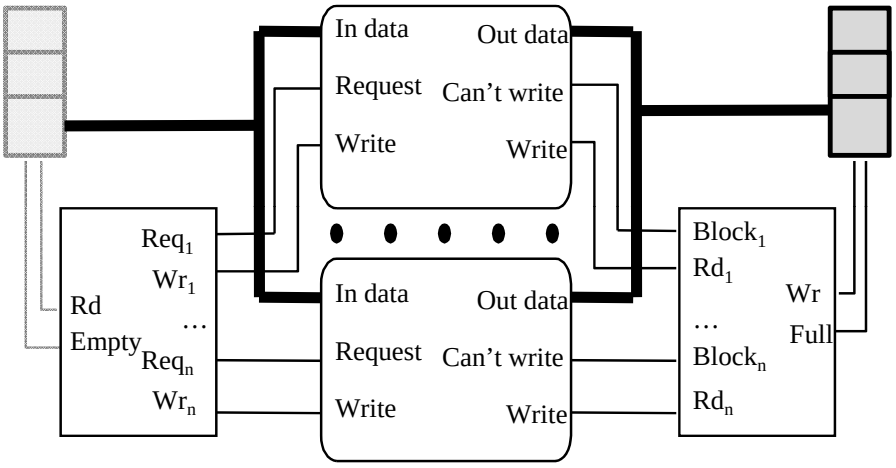


Figure 3.5: Hardware structure of the replication mechanism.

are created to notify the replica when new data can be received. In a similar manner, the result collection is handled by this mechanism, such that it can stall subsequent filters in a pipeline that attempt to read when no data is available from the current replica. The data flow is thus multiplexed, and combines handshaking signals similar to FIFOs empty / full signals.

In addition to the distribution logic generated for replicas, additional logic is generated to implement the original splitjoin constructs from the stream graph. These splitjoin constructs differ from the replication distribution logic as they have channels at both ends. These channels, as well as the channels between ordinary filters are implemented with hard-wired memory contained inside the FPGA. The number of channels is not affected by the replication mechanism, hence the total amount of memory required to implement all the channels is accounted before replicating the design.

Because the streaming model is data-centric, the control flow between filters is far less complex than the data flow. This fact enables the use of this modular method and avoids the expensive alternative of resynthesizing all the replicated copies in a SIMD-like fashion, followed by generating a unified control flow.

For filter synthesis, a filter is required to read all of its inputs in consecutive clock cycles and write its output in consecutive cycles. This requirement can be easily accommodated by existing C to HDL compilers. This also ensures that data applied to a replica does not block the distribution mechanism unless all the replicas are currently busy. Similarly, unless a filter finishes the computation earlier than its execution time considered in the synchronous model, the joiner is able to gather the generated results without stalls.

3.3 Results

The method was evaluated using some of the benchmarks described in Section 2.1.3. These benchmarks cover filters with a wide range of resources and latency requirements. The example benchmark used in Section 3.1 has been included to explore the performance of the implementation beyond the limit of the available benchmarks. The code generation algorithm was implemented in Java,

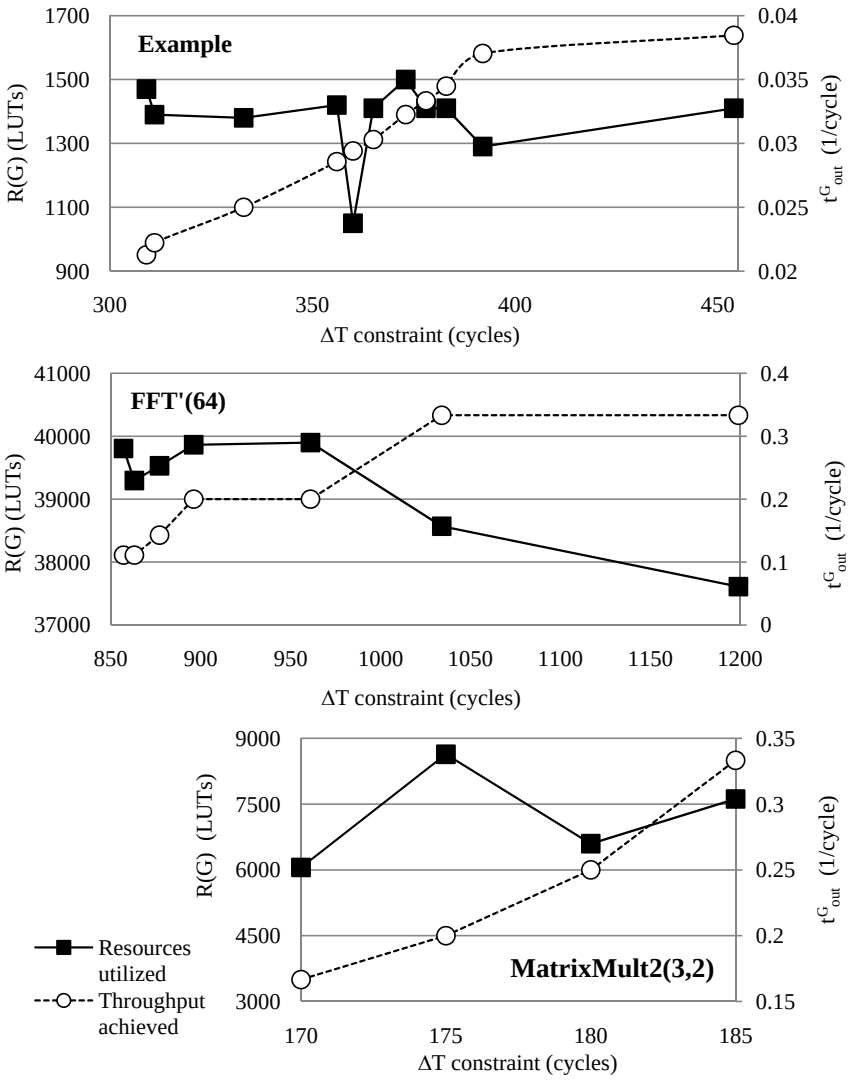


Figure 3.6: Design space exploration with a maximum resource constraint. The latency constraint is relaxed, hence the throughput can increase. The actual resource usage is influenced by both throughput and latency.

as a back-end for the StreamIt compiler, using the Opt4J library [68] to perform simulated annealing. To verify the latency reduction feature of the algorithm, a set of latency constraints, feasible in real implementations, is specified.

Generating HDL code for individual filters was an orthogonal problem [41]. Consequently, a library of filter implementations was readily available during code generation. Each filter was synthesized separately to determine its reconfigurable resource usage for the target FPGA architecture, considering LUTs, DSP blocks and Block RAM as part of the resource metric. In addition, the number of clock cycles taken by the execution of each filter was measured. Also, the FPGA implementation obtains its data through a HyperTransport interface

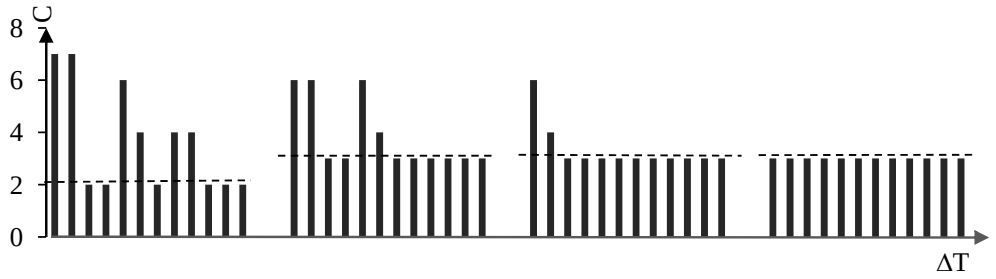


Figure 3.7: FFT design points with increasing latency. Sets of bars represent replication factors for instances of filter *CombineDFT* belonging to each design point. The dotted line separates the replication that ensures a specific throughput (below) from that necessary to decrease latency (above).

which can sustain transfer speeds of one word each clock cycle. This ensures that the off-chip data transfer does not throttle the performance.

The algorithm builds design points with the maximum achievable throughput under arbitrary latency constraints. The design-space exploration identifies lower latency implementations by slightly degrading the throughput and utilising more resources in certain areas of the design.

Figure 3.6 shows the results produced by the code generation algorithm for three benchmarks. A fixed upper bound is applied to the total resource constraint, and the latency constraint ΔT is modified over a range of possible values. As the latency constraint is relaxed, less resources are utilised for additional filter replication, and the throughput can increase monotonically taking advantage of the remaining resources. In some cases, the maximum throughput can be reached, such as in Figure 3.6b. As the latency constraint is relaxed, the number of utilized resources may decrease. However, the throughput boost may require additional replicas. These two factors affect in opposite directions the amount of resources taken by a design, and it is not possible to correlate it trivially with neither latency nor throughput.

A benchmark of interest was FFT' (a fine grained implementation of FFT described in Appendix A). This benchmark was modified so that each floating-point operation is encapsulated in a filter. This exposes the bulky floating-point arithmetic operations contained in this benchmark to the replication algorithm, such that the most suitable number of floating-point units is generated. The

Table 3.2: Design points generated for maximum throughput and under resource and latency constraints.

	Min. resources			Best throughput			
Benchmark	LUTs	ΔT	p	LUTs	ΔT	p	Speedup
MatrixMult2(3,2)	1498	480	19	7618	185	3	6.3x
Serpent(1)	3028	1027	4	3878	773	2	2x
FFT'(64)	37610	1199	3	43370	764	2	1.5x
FMRadio(8)	37458	371	39	87564	371	13	3x
iDCT(8)	45752	349	3	137256	349	1	3x
BitonicSort(32)	43920	1042	3	131760	1042	1	3x
Example	350	309	135	15990	504	2	67x

	Constrained design				
Benchmark	LUTs	ΔT	p	Constraint	Run time
MatrixMult2(3,2)	4558	175 (-6%)	7	$\Delta T \leq 175$	1.14s
Serpent(1)	3053 (-21%)	901 (-12%)	4	$\Delta T \leq 910$ $R_{FPGA} \leq 3500$	0.73s
FFT'(64)	39530 (-9%)	868 (-27%)	7	$\Delta T \leq 880$ $R_{FPGA} \leq 40000$	34.7s
FMRadio(8)	62511 (-29%)	371	20	$R_{FPGA} \leq 65000$	1.01s
iDCT(8)	91504 (-33%)	349	2	$R_{FPGA} \leq 120000$	0.73s
BitonicSort(32)	47400 (-64%)	1282	2	$R_{FPGA} \leq 50000$	18.3s
Example	1490 (-90%)	309 (-38%)	47	$\Delta T \leq 309$ $R_{FPGA} \leq 1500$	0.43s

total number of filters in this implementation is 118 filters (compared to 22 in the original FFT). The results are shown in Figure 3.6b. The graph shows that there is a significant opportunity for stream folding. A benchmark that has a tighter range of latency variation is matrix multiply (Figure 3.6c). No solutions would be possible if the latency was constrained any tighter than shown.

Figure 3.7 represents the replication factors of different design solutions of the original FFT. Each group of bars shows the replication factors for unique instances of the CombineDFT filter (which appears in multiple places throughout FFT) for a particular design point. The dotted line for each group of bars represents the replication factor that yields the maximum throughput for that design point. Designs that are subject to lower latency constraints require greater replication for several filters.

Table 3.2 shows several design points obtained using the described algorithm.

For each benchmark, the design points of interest where those having (a) the minimum resource usage, (b) the best throughput (resources limited only by the device considered, a Xilinx XC4VFX140), and (c) a constrained design between the two. The results are meant to demonstrate the versatility of the algorithm.

Most of the benchmarks are explored in a few seconds on a Core2 Duo 2.33GHz, as long as individual filter replicas monotonically contribute toward lower latencies. In cases with tight latency constraints, joiners might introduce notable adverse latency increases that degrade the convergence of the algorithm. During the extensive testing the longest running times were on the order of minutes.

3.4 Summary

This chapter described a solution to the problem of generating optimized FPGA code for streaming applications represented as stream graphs. A filter replication method increases processing throughput up to 6.3× for realistic StreamIt benchmarks, compared to the base design. A secondary goal was constraining the latency. This is achieved by throttling the maximum throughput and utilizing the spare resources to replicate the filters where the most backlog is accumulating. Hence, the algorithm yields solutions that satisfy resources *and* latency constraints. This solution provides an automatic method to realize efficiently stream applications with a flexible parallel structure on FPGA platforms.

CHAPTER 4

STREAMIT CODE GENERATION FOR GPUS

The code generation method presented in Chapter 3 has shown how the coarse-grained parallelism in the StreamIt application structure can be exploited to improve the performance of an FPGA design. However, this is associated with a throughput penalty, because the reconfigurable logic introduces run-time overhead.

The following two chapters explore code generation methods for GPU platforms, as they also expose massive parallelism. General purpose streaming applications are suitable for GPU processing as they expose significant coarse-grained parallelism. StreamIt applications consist of operators that communicate through channels, and the flexible computation structure exposed is a suitable match for the integrated code generation method proposed.

GPU platforms have gained good traction in mainstream computing and, in particular, high performance computing [65, 70]. The GPU consists of multiple streaming multiprocessors (SM), which can handle the execution of a large number of parallel threads. Groups of threads (*warps*) are selected by a hardware scheduler and executed in lockstep on a set of processing cores. Thread execution performance is influenced by whether the code can satisfy the warp's lockstep requirement and by how the data layout in memory is optimized.

This chapter describes a method that takes StreamIt applications and generates GPU optimized code, while the next one extends the method to multi-GPU platforms. The method is exemplified for the nVidia GPU architecture. The optimized code generation considers the GPU execution model and the memory hierarchy. In particular, the method described shows that high GPU utilization can be achieved using a smaller number of processing threads. This scheme goes against the conventional wisdom of GPU programming, which is to use a large

number of homogeneous threads. Instead, it uses a mix of *compute* and *memory access* threads, together with a carefully crafted schedule, that exploits the flexible parallelism in the application, while maximizing the effectiveness of the memory hierarchy.

Various programming frameworks and run-time environments have been proposed, such as CUDA [67] and OpenCL [48]. In both environments, computation is clustered in intensive data parallel *kernels*. Evolving from specialized graphic processors to general purpose computing platforms, the GPU processors are built around a special case of streaming execution model. Hence, some streaming programming languages [12] have been used to describe GPU computation, and to capture coarse interactions between these kernels.

The GPU programmers are generally encouraged to expose a larger number of parallel threads inside each kernel, so that the hardware run-time scheduler can utilize more ready threads to hide potential stalls [67]. However, there is a cost to having too many threads – increasing the number of threads diminishes the number of registers allocated to each thread, potentially causing spills to off-chip global memory. Besides this trade-off, a second hidden penalty is often overlooked. More threads reading their input, output and local data stored in the off-chip global memory lead to more memory traffic, potentially exceeding the available memory bandwidth. Jittery, application-specific memory access patterns (such as intensive memory access at the beginning of a computation block to read the input data) can further exacerbate these problems. Also, many stream processing applications exhibit low computation-to-communication ratio that may lead to stalls in a straight-forward GPU code generation approach.

One solution is to prefetch the data on-chip, in the SM memory, but this memory is typically not well utilized due to its limited size (i.e. 16KB for each SM in the nVidia Tesla 1.x-series and 48KB in the 2.0-series 'Fermi') and because the large number of concurrent data-parallel threads requires to large memory footprints. However, the described code generation method orchestrates StreamIt programs that execute on GPU platforms avoiding the issues mentioned above, and in particular maximizing the effectiveness of the SM memory. At the heart

of the code generation method is a mapping scheme that is based on a static GPU performance model derived from the GPU specifications. It relies on (1) manipulating the flexible fine-grained structure exposed by StreamIt applications to match the architecture and (2) moving slow *global* memory accesses from compute threads into another class of threads so that the former can run unobstructed while the latter satisfy the memory access requirements.

The method generates code for two kinds of threads from a StreamIt program: specialized *memory access* ($\mathcal{M}$) threads and *compute* ($\mathcal{C}$) threads. The $\mathcal{M}$ threads transfer data sets from global memory to the fast SM memory. The $\mathcal{C}$ threads compute instances of the stream graph to obtain results locally inside each SM using the data sets loaded earlier by the $\mathcal{M}$ threads. The number of $\mathcal{C}$ threads is constrained such that they work exclusively with the SM memory. These are major departures from the norm of using a large number of homogeneous parallel threads in GPU programming. The results show that these counter intuitive measures can yield significant speedups compared to more traditional approaches of generating GPU code for applications. In particular, this method is compared with a previous method that maps StreamIt using a coarse-grained approach [89].

4.1 Rationale

The GPU platform is massively parallel, and the current trend indicates a further increase in the number of threads supported in each SM. As described in Section 2.3, the GPU hardware scheduler divides the thread pool of each SM into warps which are executed in parallel lockstep. At each instruction issue interval, the scheduler can select a different warp and dispatch it to the processing cores even before the previous warp finishes processing. Thus, while there is a large number of parallel threads, they are actually interleaved onto a limited number of processing cores at the granularity of a warp.

Since StreamIt exposes a large amount of data level parallelism within applications, the method described in this chapter replicates the operator code and distributes each instance onto multiple threads. This increases the number

of instantiated warps while keeping constant the amount of off-chip communication. Replicating the code on multiple threads is supported by the implicit synchronization of the threads in each warp.

Having additional warps is important when the hardware scheduler attempts to select one, as a large number of them may not be ready to execute. Two factors can prevent the execution of a warp: the first is due to unsatisfied dependencies resulting from the latency of the processing cores (their pipeline depth is typically 22 cycles), and the other is due to the latency of the global memory access (around 400 cycles). For example, to hide the latency of the processing cores, nVidia suggests 6 ready warps on older devices of capability 1.x and 11 warps on devices of capability 2.0 [67]. As the global memory access is an order of magnitude slower, the number of warps required to completely hide this latency will exceed the maximum number that can be instantiated if all the warps require concurrent access to global memory.

Unfortunately, the replication method alone is not sufficient to increase enough the computation ratio to balance the large number of memory accesses that appear in many StreamIt applications. Typically, StreamIt operator execution is phased: (1) reading the data set from the input channel, (2) performing the computation, and (3) writing it to the output channel. Therefore, if the operator's input and output channels are stored in global memory, operator instances will spend most of their time stalled on memory access. It is therefore advantageous to bring the data into the SM memory. This way operators can process the prefetched data set at a much faster rate. But again, due to the lack of reuse of data from the channels, simply prefetching it in each thread before computation merely rearranges the memory accesses, hence this method is unable to change the ratio of computation to communication even if additional threads are instantiated.

To adjust this ratio, this method generates two classes of specialized threads: memory access ($\mathcal{M}$) threads and compute ($\mathcal{C}$) threads. The $\mathcal{M}$ threads perform prefetching while $\mathcal{C}$ threads execute on data fetched by the $\mathcal{M}$ threads into the SM memory. Intuitively, because the $\mathcal{C}$ threads will always access SM memory,

they will always be ready for execution, while the $\mathcal{M}$ threads will be scheduled from time to time to initiate more parallel memory transfers. Due to the architectural constraint that only threads in the same SM can communicate through the fast SM memory, the entire stream graph must reside in the same SM. It is replicated on all the other SM to fully utilize the GPU.

4.2 Code Generation Method

This code generation method applies a sequence of code transformations in order to:

- match the large number of parallel threads supported by the architecture,
- cluster the large latency memory transfer operations into dedicated threads,
- transform the data flow based on the parallelism exposed by StreamIt, and
- apply a novel channel manipulation scheme that replaces the one used by StreamIt compiler for inter-filter communication.

The components of the code generation method are shown as grey boxes in Figure 4.1. The method is implemented as a back-end to the StreamIt compiler. It intercepts the single appearance schedule generated by StreamIt. From this point on, this method takes over.

The memory requirements of each operator in the schedule are compiled in a compact working set (detailed in Section 4.2.3), which can be allocated in the fast SM memory. Once this working set size for a single stream schedule execution is known, additional parameters can be determined, such as the number of stream schedules that are to execute in parallel, the number of $\mathcal{C}$ threads supporting the execution of each stream schedule, and the number of dedicated $\mathcal{M}$ threads accessing global memory. These parameters are determined by analysing the stream schedule structure and the specification of the target GPU. The number of threads supporting the execution of each stream schedule modifies the schedule structure.

The C code generated for operators is enhanced with special code for the push, pop and peek primitives. This code performs the access to the working set

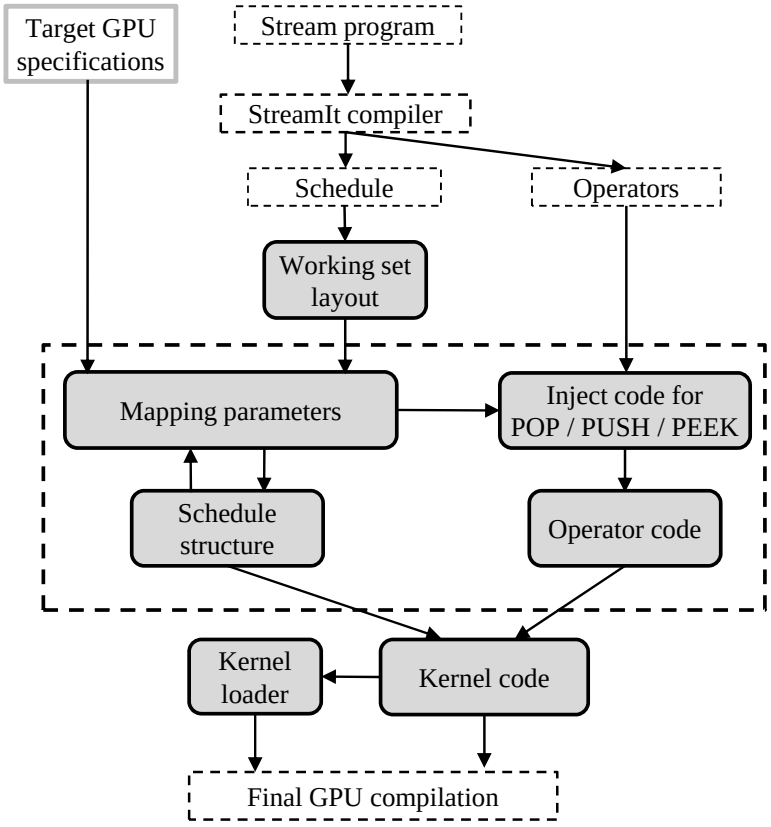
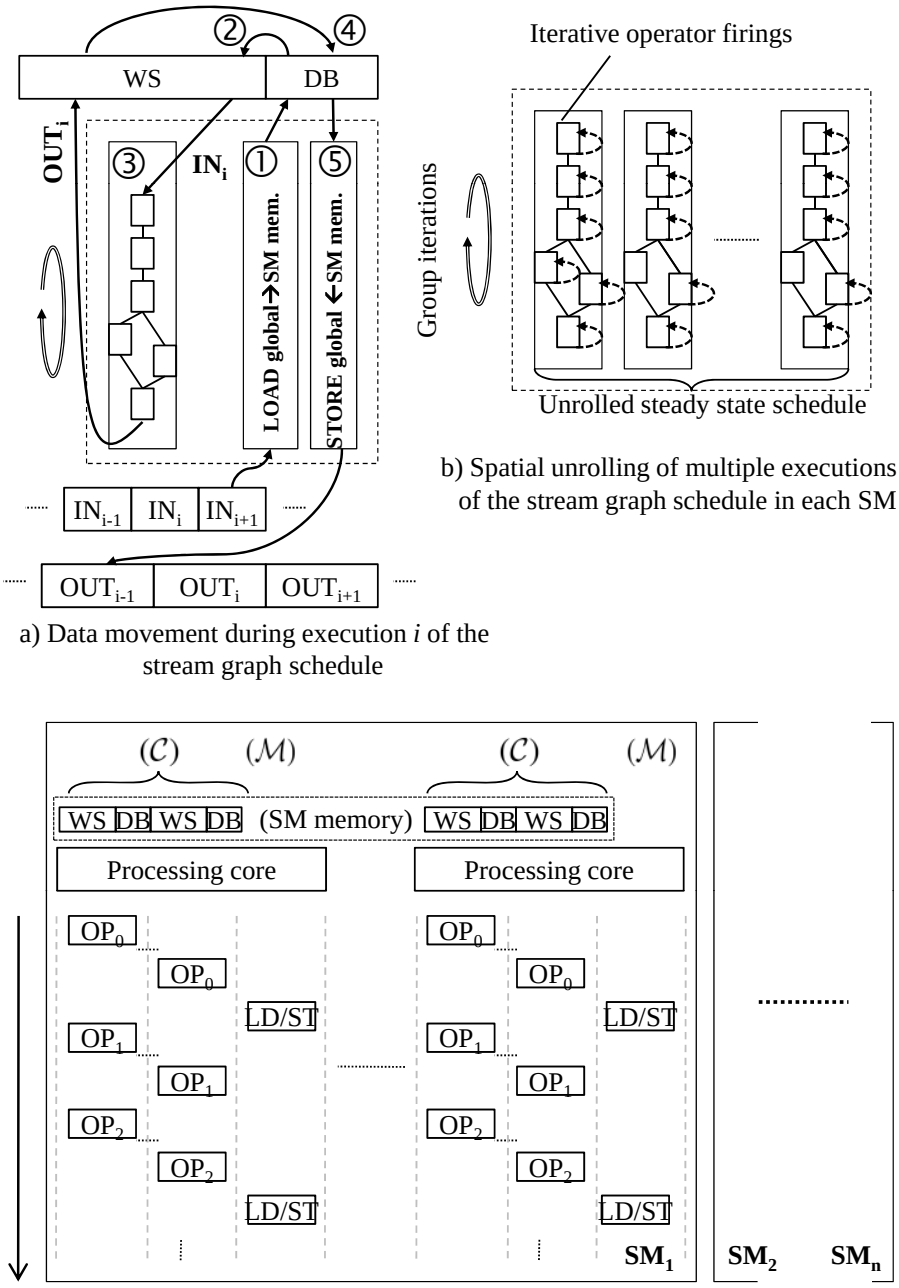


Figure 4.1: The code generation method.

in SM memory. The code for each operator is modified accordingly, and together with the restructured schedule the following two components are generated: (1) a GPU code *kernel* that implements the strategy described in Section 4.2.1, and (2) a loader which will run on the CPU and which will coordinate the memory allocation and configuration for the kernel.

The CPU host allocates input and output channels for the entire stream graph in the off-chip global memory of the GPU. Current GPU run-time environments are capable of concurrent code execution and host memory transfer, and hence the assumption is that data transfer from the host CPU to the GPU incurs no penalty. If the stream graph is partitioned and distributed over several GPU kernels, these kernels have to communicate through the global memory. This chapter focuses on executing the stream graph as a single partition, hence complete instances of the entire steady state schedule of the stream graph are executed in a single kernel. Chapter 5 shows how to handle multiple partitions and the global memory communication between them.



a) Data movement during execution *i* of the stream graph schedule

b) Spatial unrolling of multiple executions of the stream graph schedule in each SM

c) Apparently concurrent execution of multiple compute threads $(\mathcal{C})$ and transfer threads $(\mathcal{M})$ on each of the available SMs

Figure 4.2: Parallel memory access and orchestration of the stream graph.

4.2.1 Mapping Stream Graph Executions

Figure 4.2a shows how the stream graph is executed on the GPU. Let i be the current execution of the steady state schedule. A working set (WS) is allocated in SM memory to hold the inputs, outputs, as well as the channels between operators for one execution of the schedule. In addition, a second, smaller, buffer, DB, is required in SM memory. It is an intermediary buffer that is large

enough to hold all the stream graph inputs, or outputs, whichever is larger. The *double buffering* prefetch scheme works as follows. Let IN_i and OUT_i denote the input and output of execution i, respectively. During execution $i-1$, IN_i is brought into the buffer *DB* (step ①). Before the start of execution i, IN_i is copied from *DB* into the input channel allocated in *WS* (step ②). After the completion of this copying, execution i may begin (step ③). OUT_i would reside in *WS* at the end of execution i. Concurrent with execution i in step ③, IN_{i+1} is brought into the buffer *DB* for the next execution. At the end of execution i, IN_{i+1} is copied from *DB* into *WS* replacing IN_i, after which, OUT_i is copied from *WS* into *DB* (step ④). Execution $i+1$ then begins. Concurrent with execution $i+1$, OUT_i is written back to global memory (step ⑤). This last step is interleaved with the prefetching of IN_{i+1} so that *DB* can be reused.

The above describes what happens in one instance of the steady state schedule. A group of W instances of the steady state schedule is further unrolled and is executed in parallel. Each of these executions stores its local data into a separate working set allocated in the fast SM memory. These executions are mostly independent except for peeking (which will be discussed later), and suitable for a parallel orchestration as described in Figure 4.2b. Each steady state schedule includes a sequence of operator firings that may be iterative. One complete processing of a stream graph is called an *execution* of the steady state schedule. Each schedule execution is distributed to one or more $\mathcal{C}$ threads. The processing over the group of W parallel executions is iterated as many times as necessary to process all the application's inputs. Such a pass over the group of W executions is called a *group iteration*. Each SM is assigned a different part of the input and output stream in sequence. In particular, for SM_1, this sequence number starts from the beginning of the stream in global memory. All the SM will compute the results for distinct portions of the input stream, and the access offsets in these streams are known and computed by the loader before the kernel launches.

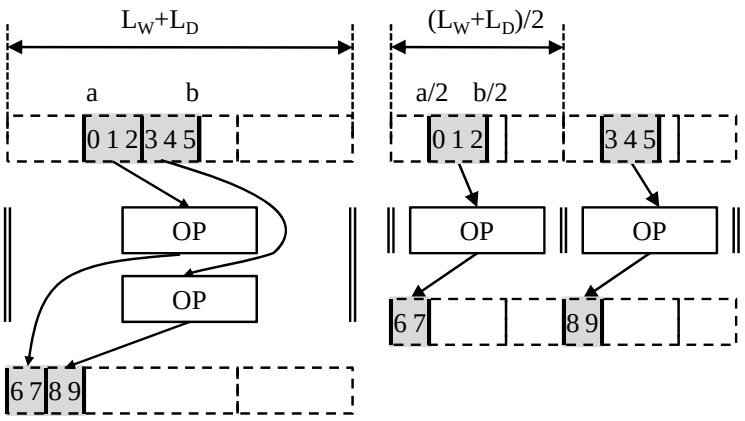
Furthermore, the double buffering mechanism described in Figure 4.2a can be refined for a group of parallel executions. Loading and storing to global memory are performed by a set of parallel $\mathcal{M}$ threads that combine the load and store

operations corresponding to all executions. Let F be the number of $\mathcal{M}$ threads. The proper steps are taken to ensure that $\mathcal{C}$ threads and $\mathcal{M}$ threads are allocated to distinct warps. Therefore, they execute in an interleaved manner as shown in Figure 4.2c. In general, $\mathcal{C}$ threads will always be available for execution, as their data dependencies are satisfied from registers or SM memory. $\mathcal{M}$ threads, however, issue long latency global memory operations, and are scheduled only sporadically. The intuition is that by adjusting the number of $\mathcal{M}$ threads (F), and $\mathcal{C}$ threads (W), the latency of the global memory accesses can be completely hidden.

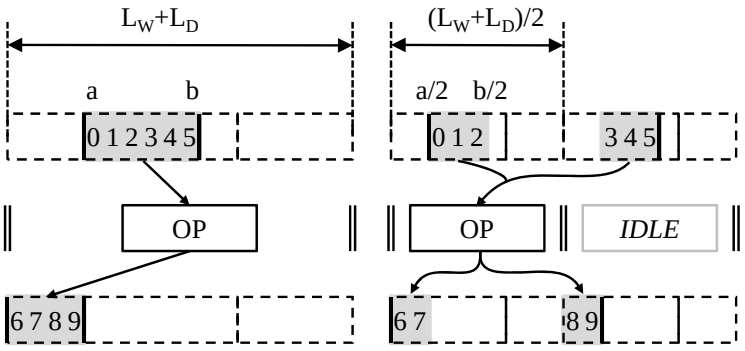
The *steady state schedule*, **E**, is an ordered sequence of stream operator firings that consumes a set of inputs, and eventually generates a set of results. The amount of intermediate data obtained during these executions may require more memory than the IN and OUT channel region. Many other channels are also found in *WS*, for operators in the graph which communicate with one another. Let the total memory requirement for *WS* be L_W. The size of the secondary buffer *DB*, on the other hand, is $L_D = \max(\text{size(IN)}, \text{size(OUT)})$.

Each core in a CPU multi-core approach executes a single instance of the schedule, and has a large amount of memory available. The StreamIt compiler offers a feature that may fuse filters only to tune their channel size to the cache size. Nevertheless, as the channels are not reused, there was no effort to optimize the memory resource usage over the entire graph. The efficiency of this method depends on the working set size, as this dictates how many parallel executions of the graph can run, because the complete working set must be stored in SM memory. The algorithm used to determine a compressed *WS* layout is described in Section 4.2.3.

If the schedule fires an operator OP_i R_i times, these firings are independent and can be executed in parallel in a number of $\mathcal{C}$ threads. Therefore, each of the W steady state executions of the schedule can be distributed among S $\mathcal{C}$ threads of the GPU. This effectively multiplies the available parallelism, and is essential in improving the GPU's utilization. Otherwise, the number of $\mathcal{C}$ threads utilized would be limited by the size of the SM memory. Accordingly,



a) Scattering filter firings in GPU threads: S = 1 (left) and S = 2 (right)



b) Single filter firing with scattered data: S = 1 (left) and S = 2 (right)

Figure 4.3: Memory layout transformation examples.

the *WS* of a steady state execution is split into equal sections associated to each $\mathcal{C}$ thread. If an operator fires for less than S times, then it will be assigned to some of the threads, while the remaining threads will be idle, without any additional performance penalty. Operators firing more than S times will be executed several times by each $\mathcal{C}$ thread. Such a distribution is valid $\forall S$ such that $\forall i, \gcd(R_i, S) = \min(R_i, S)$.

SM memory is banked and, therefore, it supports parallel access, provided the same bank is not accessed twice. Because warps execute in lockstep, all the $\mathcal{C}$ threads in a warp are accessing the SM memory simultaneously. If the accesses are to distinct banks, then the hardware will coalesce the accesses into a parallel access [67]. For automatically generated code, the accesses to the SM memory can be arranged to be coalesced as follows. The *WS* and *DB* are stored in a contiguous region of SM memory. Since the number of banks is a power of 2 (typically 16), coalescing can be enforced if $L_W + L_D$ is adjusted up to the next

odd number. If the gap between consecutive regions is an odd number p, then any offset in thread i and thread $i + j$, $\forall i, \forall j < 2^b$ is separated by a distance $j \cdot p$ which is coprime with 2^b, thereby ensuring that all banks are used. Utilizing this result, the total number of parallel executions, W that can fit a memory of size L_{SM} is

$$W \leq L_{SM}/\Lambda$$

where Λ is the adjusted memory requirement for a single stream schedule execution, $\Lambda = 2 \cdot \lfloor \frac{L_W + L_D}{2} \rfloor + 1$.

Figure 4.3a compares the execution of an operator OP, scheduled to fire two times in a single thread with that of distributing it among two parallel threads. A memory region $[a, b]$ of length $L_W + L_D$ will be divided into a set of smaller regions, each of length $\frac{L_W + L_D}{S}$. The elements in *WS* and *DB* are redistributed in sequence among the smaller regions, filling the *WS* of one thread before continuing to the next. By doing so, if the stream operator OP fires $f \cdot S$ times in the schedule, its firings can be distributed among S threads, in parallel, each thread handling f firings using data from its properly aligned section of the *WS*. Most of the additional synchronization overhead for this scheme is avoided by taking advantage of the lockstep nature of the threads in the same warp. The channels in the original *WS* also need to be aligned to a multiple of S elements to allow this transformation.

Complementary, Figure 4.3b shows the execution of a single firing of operator OP, when two threads are implemented. By means of a conditional, the execution in the second thread is simply disabled. The operator running in the first thread can access elements from both *WS* regions, and the same coalescing properties are maintained among the active threads in a warp.

4.2.2 Parallel Execution Orchestration

A complete example of how this method orchestrates parallel executions of the steady state schedule, each onto multiple $\mathcal{C}$ threads ($S = 2$ in this example), is shown in Figure 4.4. The stream graph in the shaded box on the left is automatically translated to the execution scheme to its right. Whenever possible,

operator firings are handled by parallel $\mathcal{C}$ threads. Each thread is allocated a *WS* size of half the total *WS* size, precomputed for the entire steady state schedule. The *WS* stores the intermediate results for future operator firings. For clarity, those parts of the *WS* used as input by the current operator firing are shadowed. The *DB* buffer is not included in this illustration.

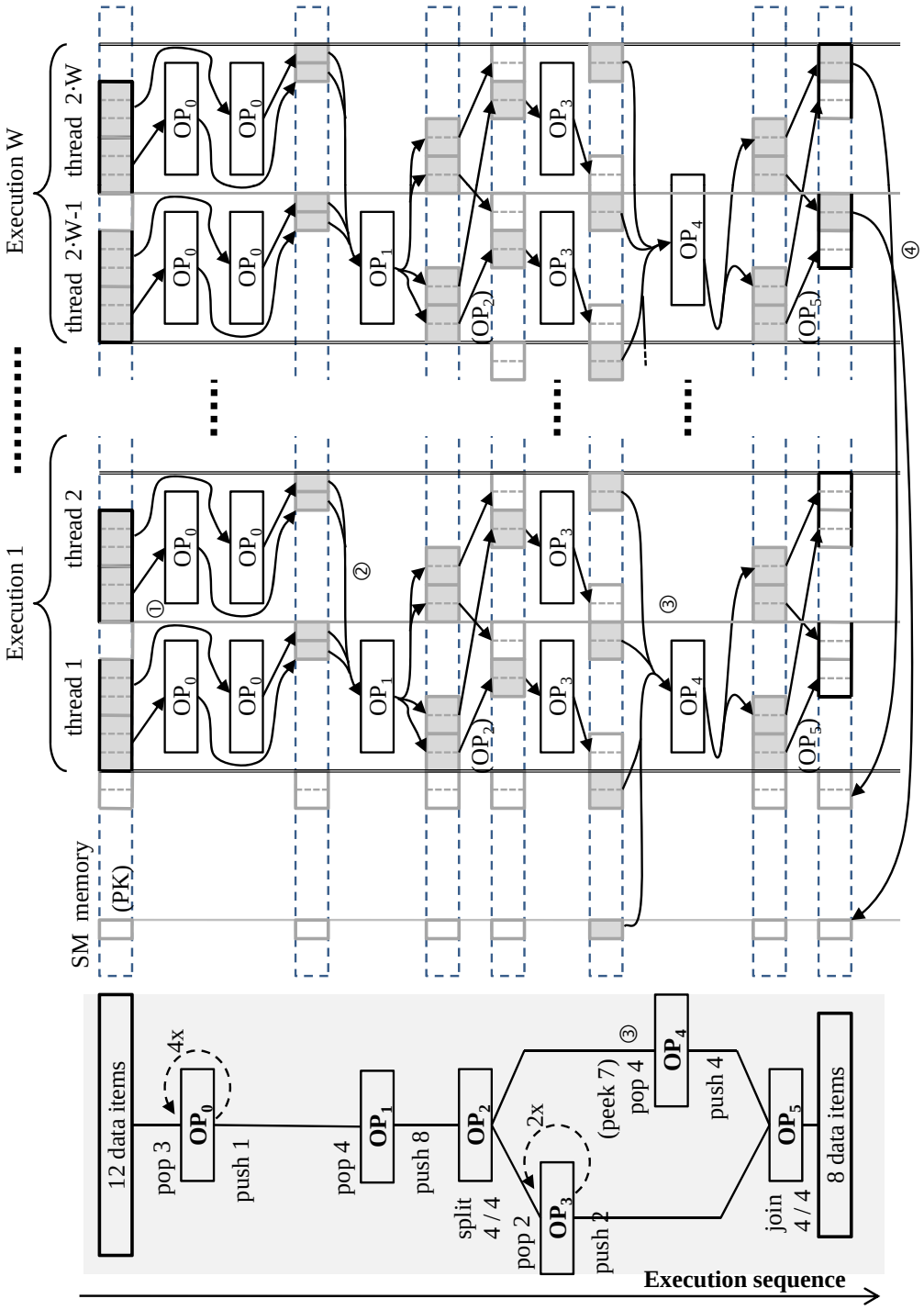


Figure 4.4: Example of the orchestration for a single group iteration. Two $\mathcal{C}$ threads are assigned to each of the W parallel executions of the stream graph.

In this example, the 12 input items consumed by the stream graph during each execution of the steady state schedule are distributed among the SM working sets of the two $\mathcal{C}$ threads corresponding to each execution. Because OP_0 pushes only one element but OP_1 pops four elements, the schedule will consist of four firings of OP_0 for each firing of OP_1. The firings of OP_0 are distributed among the two threads, two in each thread ①. The outputs of OP_0 are written back to the *WS* of both threads using a similar layout.

OP_1 needs four elements in a single firing, executed in the first thread, so it requires access to both its own *WS* and the adjacent thread's *WS*, both available in the SM memory ②. To avoid the run-time overheads, the solution is to generate precomputed tables that translate the 0-based consecutive indices of pop and push operations into relative offsets to the beginning of the allocated *WS*. These relative offsets specify access ranges beyond the limits of the *WS* of the current thread, and thus support the fetching of data produced by adjacent threads that cooperate for the same schedule execution.

The output of OP_1 is the input of the splitter OP_2. The splitter divides the eight data items into two distinct regions of four items. As required by the code generation method, each of these output regions needs to be distributed between the *WS* of both threads. Therefore, the automatically generated splitter operator distributes consecutive groups of two elements between the two *WS*. The execution of OP_3 and OP_4 is serialized in the steady state schedule. Each firing of OP_3 utilizes the set composed of the first two elements from each *WS*, and runs in one of the two GPU threads. OP_3 does not utilize the second set of elements generated by OP_2.

Support for peeking: OP_4 is a *peeking* operator. In this example, OP_2 is required to push seven elements to the input of OP_4, before the latter can be fired ③. However, only the first four elements produced will be consumed. Therefore, the semantics of peeking requires preceding operators in the schedule to generate more data, which will be only inspected, but not consumed. In the current execution, OP_2 generates only four elements for OP_4. OP_4 must obtain the other three from another firing of OP_2, either in the current or the previous group

iteration. The peeking scheme shifts the reference of the peeking filter's input channel into the previous execution's *WS*. Intuitively, the first accessed elements in the sequence, which are those popped, were generated during a previous execution of the steady state graph, while the most recent ones, generated by the current steady state execution, are only peeked. The precomputed tables take into account the popping/peeking requirements, and may contain negative relative offsets at the beginning of the sequence so that peeking filters can access elements in of the previous execution's *WS*.

All the necessary offset tables are precomputed on the host CPU and preloaded in the constant memory. Such tables are required for each channel input / output rate. For example, for a filter firing once and having a pop rate p and a peek rate e, the input table T has e elements computed as follows: $\forall i \in [0, e], T_i = \frac{(p-e+i)\cdot S}{p} \cdot \text{size}(\Lambda) + \frac{(p-e+i)\cdot S \bmod p}{S}$. The first term determines the *WS* to access and adjusts the offset by the relative offset of that *WS* with respect to the current *WS*. The second term specifies the relative position inside the *WS*. Integer division returns the lower integer as the result, while the mod returns only positive values. As the constant memory is cached and the practical number of tables is small, this indirection has lower overhead than computing the values at run-time.

To support this peeking scheme in all parallel executions, an additional 'PK' section is reserved (Figure 4.4) at the beginning of the SM memory. It is required to hold the content of the input channel data corresponding to the last executions of the previous group iteration. This is necessary to expose the additional elements required by the first parallel $\mathcal{C}$ threads of the current group iteration. Suppose the current group iteration is j. OP_4 of execution $k, k > 1$ of group iteration j will obtain the three additional elements from execution $k-1$ of group iteration j, as they were written by OP_2. The situation for the first execution is special. OP_4 of first execution of group iteration j will have to get the elements from execution W of group iteration $(j-1)$ via the PK area. In this example, these last three elements are copied as the last step of the schedule execution in group iteration $(j-1)$, because OP_4's input channel is not reused ④.

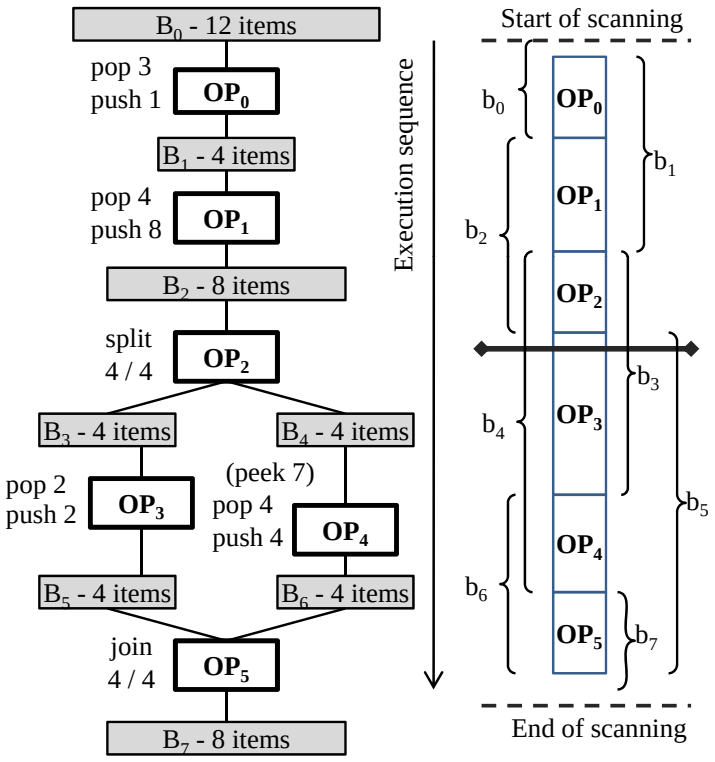


Figure 4.5: Liveness and lower bound analysis on working set size.

To ensure access consistency to elements from adjacent executions, additional synchronization was introduced among the $\mathcal{C}$ threads before firing each peeking filter. This guarantees that the $\mathcal{C}$ threads belonging to different warps have completed execution of predecessor operators, and have produced all the necessary input data. Because only $\mathcal{C}$ threads require synchronization and this does not have to interfere with the $\mathcal{M}$ threads, the SM thread synchronization primitives are not suitable. Instead, a simple workaround barrier is implemented, which takes advantage of the lockstep execution within a warp. A thread representative is appointed for each $\mathcal{C}$ warp. This owns and increments a counter residing in SM memory once it reaches a synchronization point. Afterwards, it repeatedly checks if its counter has a value smaller or equal to the other appointed threads' counters. If not, it waits. To avoid busy waiting, the hardware scheduler is forced to run other warps by accessing a global memory location marked as volatile. Because all the threads in a $\mathcal{C}$ warp are in lockstep, synchronizing a single thread from each warp reduces the workload required, while holding all the warp's threads synchronized.

In addition, stream graphs containing peeking operators, need a special initialization (warmup) schedule before the steady state groups can begin. This is necessary to initialize the regions accessed during peeking. Otherwise, for example, OP_4 of execution 1 of very first group iteration would never have the additional three elements needed to be fired up. The number of initialization iterations can be determined statically, and this method coordinates the GPU to execute an additional number of group iterations of the steady state for which it ignores the final outputs, but it updates all the intermediate values in the *WS*, hence initializing them. The correct offset in the input stream is determined statically, such that the first group iteration after warmup utilizes all the $\mathcal{C}$ threads.

4.2.3 Working Set Layout

The size of the *WS* stored in SM memory has a direct impact on the performance of this method. The amount of SM memory is small, and a compact *WS* will enable a larger number of parallel stream executions. The algorithm described below provides a near-optimal *WS* layout. It first identifies a lower bound on the *WS* size. Next, using a simple yet efficient heuristic, it performs working set allocation, slightly increasing the *WS* size, if necessary, to accommodate this layout.

Figure 4.5 revisits the stream graph example in Section 4.2.1, showing the working set size required for each operator. Filters have a single input and output channel each, while splitters and joiners transfer data from and to multiple channels. An operator can be fired, if, and only if, its input/output channels are in memory before and after its firing. Each channel is written and read only once. Therefore, the identified channel layout should ensure that no channels are overwritten before the data they contained is used.

Let B_k be the channel between the output of operator OP_i and the input of operator OP_j. The *liveness* interval of B_k is defined as the interval $b_k = [E(i), E(j)]$, where $E(n)$ is the position of operator OP_n in the execution schedule **E**. Figure 4.5 shows the liveness interval of each channel in the stream graph.

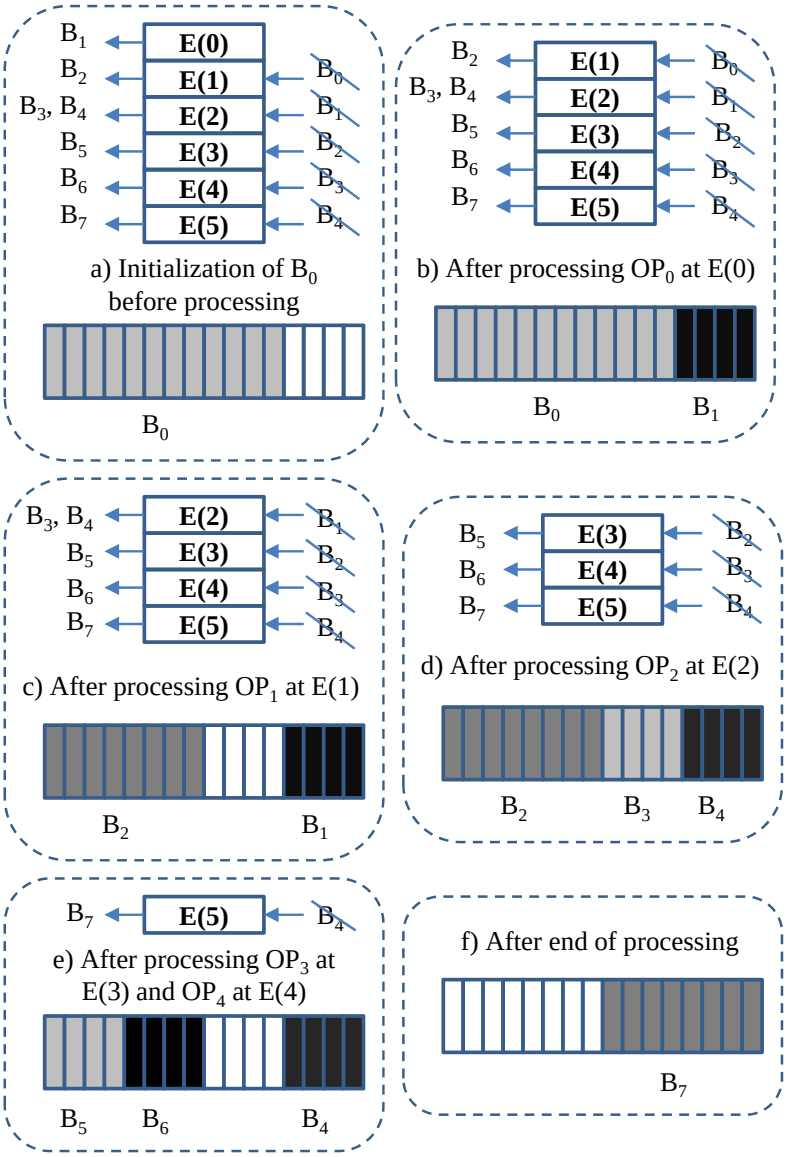


Figure 4.6: Working set allocation example.

For example, the liveness interval b_4 begins before the firing of OP_2 and ends after the firing of OP_4.

Based on the liveness intervals, the lower bound of the *WS* size can be computed for the entire stream graph as follows. A linear scan of the execution of the N operators in the steady state (as shown in Figure 4.5) determines the minimum *WS* size as $L_B = \max_{k \in [E(0),E(N)]} (\sum_{\forall n, k \in b_n} \mathrm{size}(B_n))$.

This lower bound is the minimum *WS* size that can store all the necessary channels during the entire execution of the steady state of the stream graph. The computation of this lower bound does not take into account the memory

fragmentation caused by the fact that channels should be allocated in contiguous memory ranges. Channel relocation is not allowed. Instead, the *WS* size may slightly be increased to accommodate channels in a fragmented *WS*.

Given the lower bound on *WS* determined above, this is used as a starting point in a heuristic approach that allocates channels for each operator. Figure 4.6 walks through the allocation algorithm for the above mentioned stream graph and uses the lower bound of *WS* size identified as $L_B = \text{size}(B_5 \cup B_6 \cup B_7) = 16$). Initially, the input B_0 is allocated in the *WS* (a). After $E(0)$, B_1 is placed into the SM memory (b). When processing $E(1)$, according to the liveness analysis, the space utilized by B_0 can be reused for B_2 (c). Next, splitter OP_2 will have its output allocated (d). After the analysis of $E(3)$ and $E(4)$ (e), the joiner OP_5 has all its input allocated, and its output is allocated at $E(5)$, completing the steady state schedule analysis (f).

Algorithm 4.1 summarizes the working set allocation strategy. To allocate channels for each ready operator in the execution schedule, the availability of the locations ins *WS* is updated, deallocating all the channels for which liveness has ended (lines 3-5). The memory for the deallocated channels becomes available, and is combined to form large contiguous blocks of available memory. For each channel that becomes live at this step, an available memory slot is searched (line 8, though not shown in detail) using a simple heuristic: starting from the last successful allocation, try to find the nearest slot that will fit the current allocation request. The intuition is that neighboring channels tend to expire together or close to one another, thereby increasing the likelihood of large chunks of contiguous free slots. If a suitable memory slot can not be found, the current *WS* is extended to fit the current channel (line 12). Note that if there is some available memory at the rear of the *WS*, its size is extended only with the difference required to accommodate the new channel. Finally, the allocated configuration and the final *WS* size L_W are returned (line 17).

Several constraints apply to the algorithm described above. The working set alignment must be equal to the split factor S, to enable the splitting mechanism described in Section 4.2.1. Furthermore, peeking operators can not overlap their

input channel ranges as the data is saved at the same offset in the PK section of the SM memory. Therefore, if two peeking filters overlap in their input channel ranges, their PK buffers will also overlap, because the allocation in the PK section for peeking operators never expires, and will result in data corruption. Therefore, while analyzing the stream graph, the set of peeking operators and their channel requirements are recorded to avoid allocating another peeking operator in the same memory range. However, this issue does not occur between a peeking operator and a non-peeking one as the latter does not have a persistent presence in memory.

A special optimization is introduced for duplicate splitters. These splitters are a special type of splitters that generate multiple identical output channels from a single input channel. To prevent expensive data movement, the liveness of its input channel is extended until the last use of the splitter's original outputs.

4.3 Design Space Characterization for Different GPUs

This section provides a characterization of the design space of the code generation method. The benchmarks utilized are those described in Section 2.1.3. The three parameters that determine the execution time were defined in Section 4.2.1, namely:

- W, the number of parallel stream schedule executions;
- S, the number of $\mathcal{C}$ threads per execution;
- F, the number of $\mathcal{M}$ threads that transfer data between global and SM memory.

These parameters are varied in order to observe their impact on performance. The number of $\mathcal{C}$ threads is increased until the maximum possible. The $\mathcal{C}$ threads will generally be available for execution, and their workload is matched by a set of $\mathcal{M}$ threads. Scheduling is done at the granularity of a warp, so if $\mathcal{M}$ and $\mathcal{C}$ threads are in distinct warps, they will execute concurrently.

According to nVidia [67], hiding the latencies of the processing cores requires, for example, 192 and 352 threads (6 / 11 warps) for integer operations on devices

Algorithm 4.1 Working set allocation algorithm

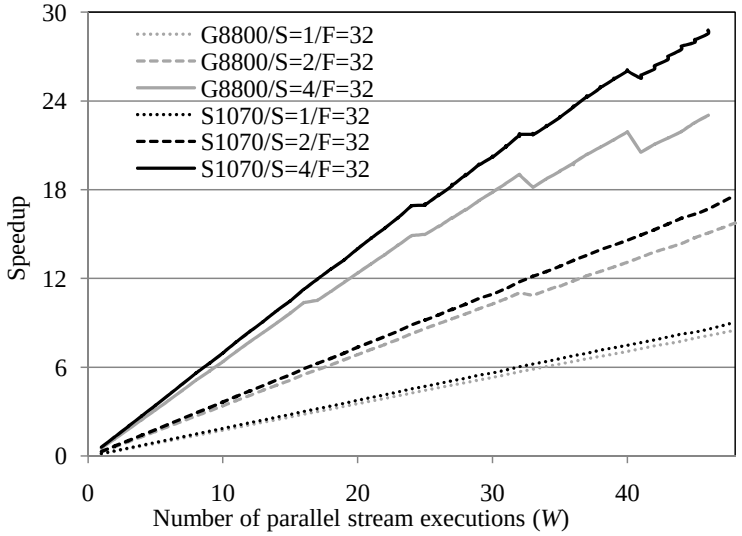
Input steady state schedule E; number of operators N; lower bound L_B of the working set size

Output *allocation* for each channel in the *WS*, feasible *WS* size L_W;

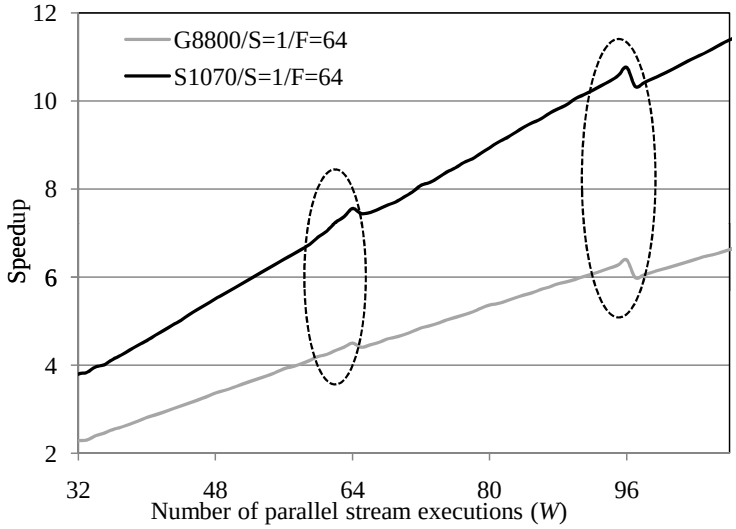
```
1: L_W ← L_B;
2: for i = 0 to N − 1 do
3:    for each b_j = [..., E(i − 1)] do
4:       deallocate(b_j);
5:    end for
6:    update_availability(L_W);
7:    for each b_j = [E(i), ...] do
8:       if (find_next_slot(b_j)) then
9:          allocate(b_j);
10:         update_availability(L_W);
11:      else
12:         extend(L_W);
13:      end if
14:      record_allocation(b_j, allocation);
15:   end for
16: end for
17: return allocation, L_W;
```

of capability 1.x and 2.x, respectively. This number assumes no global memory stalls, and it will be referred to as N_G. Execution time improvement is expected as long as the number of $\mathcal{C}$ threads that run the stream graph schedule in parallel is lower than N_G. As W is limited by the total size of the SM memory, the split factor S multiplies the number of $\mathcal{C}$ threads.

Figure 4.7a characterizes the speedup as a function of the number of parallel stream executions for the FilterBank benchmark. The number of $\mathcal{M}$ threads ($F = 32$) was chosen high enough to sustain the transfer demands for the given design space. All the possible range of values for W and S is enumerated. The speedup is measured for the same benchmark configuration for two nVidia GPUs of capability 1.x, namely the G8800 and the Tesla S1070. The X- and Y-axis show the number of stream executions W, in each SM, and speedup, respectively. For each GPU type, different lines represent the speedup for different S factors (number of $\mathcal{C}$ threads per steady state schedule execution). As expected, if the number of $\mathcal{C}$ threads increases, the speedup of the application increases accordingly. The speedup is defined as the ratio of execution time of the application code generated for GPU, compared to the execution time of the CPU



a) Design space characterization for FilterBank(4) benchmark



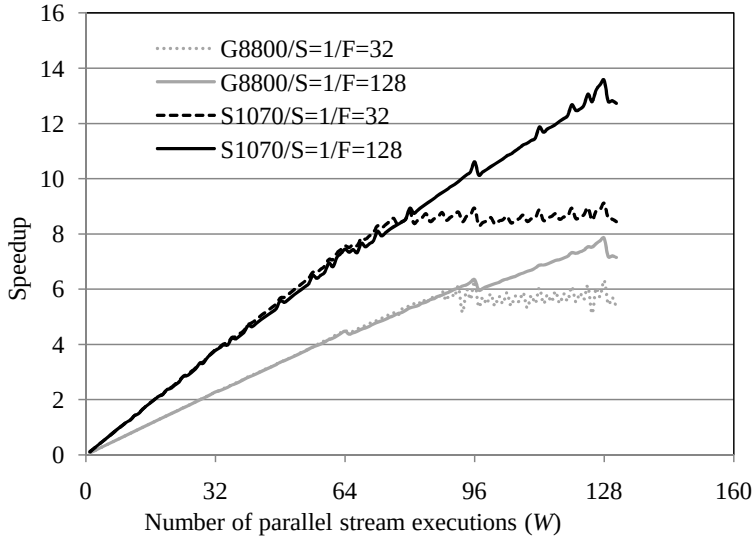
b) Speedup non-monotonicity for BitonicSort(8) benchmark

Figure 4.7: Characterizing the design space.

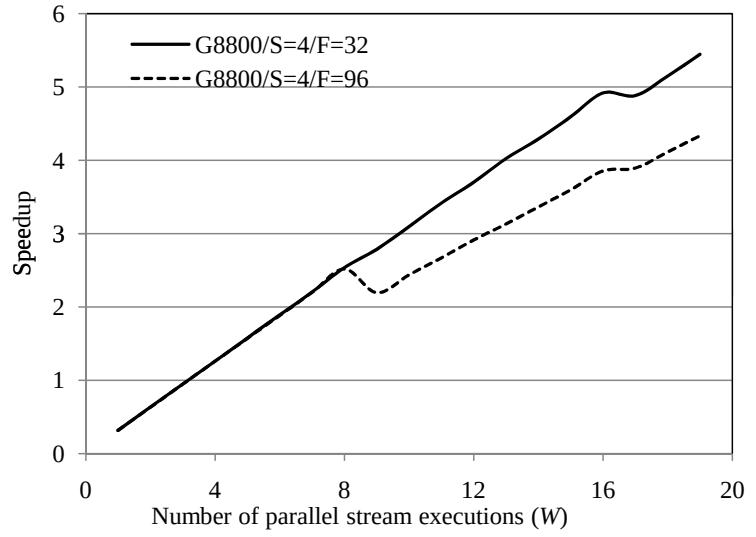
code (2.83 GHz Intel Xeon E5440) compilation. For the same number of iterations W, increasing S leads to better performance. The result also shows that the speedups on the S1070 are higher than those obtained on the G8800.

Does a higher number of $\mathcal{C}$ threads always guarantee higher speedup? Figure 4.7b shows an interesting scenario where a higher number of $\mathcal{C}$ threads may hurt speedup. These anomalies can be explained by the correspondence of $\mathcal{C}$ threads to warps. If the number of $\mathcal{C}$ threads is a multiple of 32, warp occupancy will be at its highest, and only full warps are scheduled. On the other hand, if additional $\mathcal{C}$ threads are scheduled, the last warp is under-utilized but it

shares registers and a scheduling slot with the other warps. In Figure 4.7b, the speedup falls exactly at the above-mentioned points (because $S = 1$, the actual number of $\mathcal{C}$ threads is equal to the number of parallel stream executions). After a point, if the number of $\mathcal{C}$ threads increases the warp occupancy, the speedup gradually recovers.



a) Different designs varying F for BitonicSort(8) benchmark



b) Speedup penalty if F is too high for MatrixMult2(2) benchmark

Figure 4.8: The trade-offs for F, the number of $\mathcal{M}$ threads.

As mentioned above, the number of $\mathcal{M}$ threads plays an important role if the $\mathcal{C}$ threads execute fast relative to the latency of global memory. Figure 4.8a shows the performance penalty if not enough $\mathcal{M}$ threads are scheduled for both the G8800 and S1070. Experimental data is presented for two different scenarios: one

in which the data demand of the $\mathcal{C}$ threads ($F = 32$) is not satisfied, and another in which it is ($F = 128$). After linearly increasing, the speedup corresponding to the smaller number of $\mathcal{M}$ threads reaches an upper bound, while the speedup corresponding to the higher number of $\mathcal{M}$ threads increases steadily on both GPU. If the number of $\mathcal{C}$ threads is high enough, the data transferred by a small number of $\mathcal{M}$ threads is unable to keep up with the demand for data from the global memory. If the number of $\mathcal{M}$ threads increases correspondingly with demand, speedup increases nearly linear in terms of the number of $\mathcal{C}$ threads.

If the number of $\mathcal{M}$ threads is too high, performance (speedup) also degrades. Note that $\mathcal{M}$ threads compete for SM occupancy with $\mathcal{C}$ threads. All threads, irrespective of their type, are allocated an equal number of registers, and a higher SM occupancy leads to less registers available to each of the $\mathcal{C}$ threads. Figure 4.8b shows that performance may degrade as less registers are allocated to each warp. The experiments show that the number of $\mathcal{M}$ threads typically required is 32 or 64. This result matches with the intuition that a small number of $\mathcal{M}$ threads is sufficient to match the demands of the W stream executions on each SM.

Heuristic equations for parameter selection: Based on these insights, a set of equations can be used to compute the correct number of $\mathcal{C}$ and $\mathcal{M}$ threads for any streaming application.

The first architectural constraint introduced is to limit the number of $\mathcal{C}$ threads to N_G because this number of threads fully utilizes the GPU in the absence of global memory stalls. $\mathcal{M}$ threads do not execute often, and it can be assumed that they do not contribute to the total utilization. Thus, $W \cdot S \leq N_G$. The next constraint considered is that presented in Section 4.2.1 and it is used to derive the maximum number of parallel executions as a function of S:

$$W(S) = \min(\frac{N_G}{S}, \frac{L_{SM}}{\Lambda})$$

The execution time $T(\mathbf{E}, S)$ of a group iteration depends on how the steady state schedule $\mathbf{E}$ is distributed over the S $\mathcal{C}$ threads. Only operators fired iteratively in the schedule of the stream graph can be distributed to decrease the

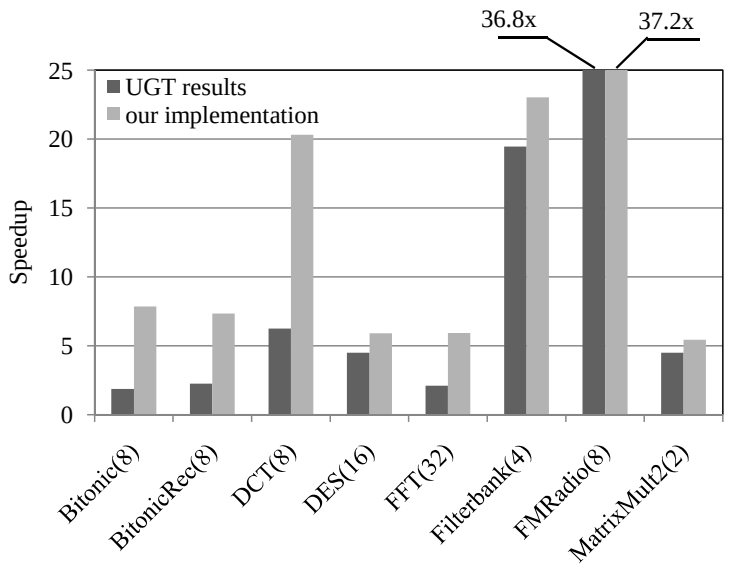


Figure 4.9: The comparison between *UGT* and this method.

execution time. Therefore, the execution schedule $\mathbf{E}_t^S$ of each thread t from the set of S threads associated with a stream graph execution is analysed. The estimated workload $\mathrm{WL}(p)$ for each operator OP_p is derived by the StreamIt compiler. Putting these together, it results:

$$T(\mathbf{E}, S) = \max_{t<S}\Big(\sum_{p\in\mathbf{E}_t^S} \mathrm{WL}(p)\Big)$$

To maximize the speedup, i.e. $W(S)/T(\mathbf{E}, S)$, the value of S_m is determined such that $\forall S_i \neq S_m, W(S_i)/T(\mathbf{E}, S_i) \leq W(S_m)/T(\mathbf{E}, S_m)$ which corresponds to $W' = W(S_m)$ executions. However, Figure 4.7b suggests that packing the $W' \cdot S_m$ threads into warps must not leave the number of active threads in the last warp to be less than $(W' \cdot S_m)/16$. If the last warp is underutilized, then W is reduced to $\lfloor \frac{W' \cdot S_m}{32} \rfloor \cdot 32$, otherwise use $W = W'$.

In addition, the ratio of parallel executions to $\mathcal{M}$ threads is analysed. This has to match the ratio between the run time of the parallel stream executions and their *DB* size L_D.

$$\frac{W}{F} = k \cdot \frac{T(\mathbf{E}, S)}{L_D}$$

where k is a GPU-dependent constant derived experimentally. The value of F is rounded to the next full warp value.

4.4 Results

The heuristic presented above can select efficiently the number of $\mathcal{C}$ and $\mathcal{M}$ threads. The following experimental results compare the speedups between:

- the previous state of the art implementation [89] and the results obtained using the described method;
- different nVidia platforms.

The first comparison is between the results obtained utilizing the described code generation method and the results presented by a recent method [89], mentioned in Section 2.3.1, and referred here by the acronym 'UGT'. This method partitions the stream graph between SM, and launches a large set of homogeneous parallel threads in each SM. Data transfers between SM are done via the global memory.

This code generation method is implemented as a back-end for the StreamIt 2.1.1 compiler. As presented in Section 4.2, the code generated can be compiled and run on different GPU platforms using the parameters selected by the heuristic equation. In order to match the experimental setup of UGT, a set of experiments was run on the nVidia G8800 with an old driver of release number 177.73. As a baseline, the baseline platform used by UGT was used, namely, an Intel Xeon E5440 running at 2.83 GHz, with the executable obtained through the uniprocessor back-end of StreamIt, and compiled using the '-O3' option of GCC 4.1.2. Based on the description found in the UGT paper, the benchmark parameters were adjusted for a similar configuration[1].

Figure 4.9 shows that this code generation method outperforms UGT. The speedup in the graph is the ratio of execution time on GPU to that on the CPU. For all 8 benchmarks, this method executes faster than the UGT method, by as much as 4.2×. On average, it is 2.8× better than UGT. The smallest improvement is for FMRadio, but this is due to an opportunistic optimization that was introduced in the UGT implementation. Because the *WS* of each

[1]The configuration used by UGT for matrix multiply could not be determined based on the provided description. Instead, the one reported here was used.

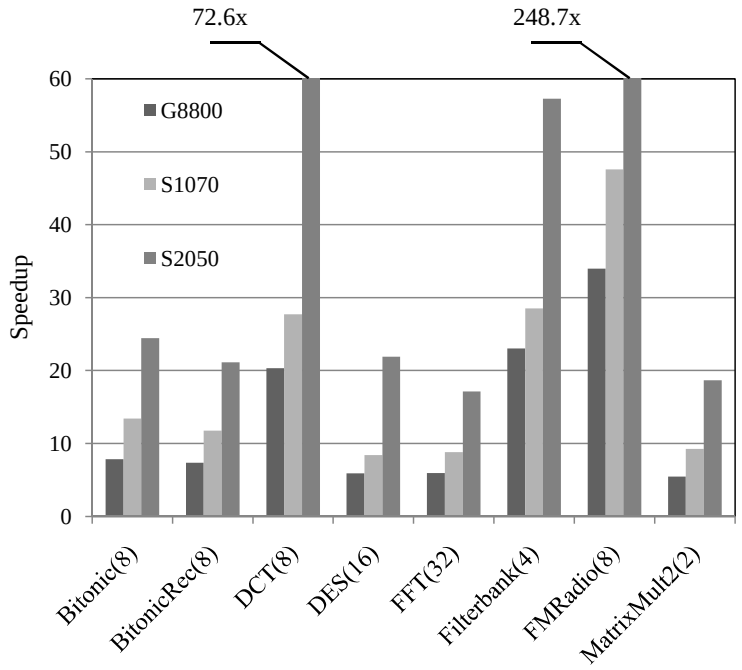


Figure 4.10: The versatility of the code generation method.

iteration in this stream graph is relatively small, the entire WS for a large number of iterations was allocated in SM memory. This result actually confirms the direction taken by this code generation strategy.

In order to demonstrate the versatility of this method on newer GPU devices, the performance of the code generated for nVidia G8800 GPUs was compared with that achieved by code generated for a Tesla S1070 platform (capability 1.3) and for a Tesla S2050 platform (capability 2.0). The CUDA toolkit and driver version 3.1 were utilized for these experiments. For S2050, the extended 48 KB SM memory was enabled. This extended SM memory is mutually exclusive with a larger cache. In this case, caching on demand would not have performed better as data is prefetched. The results are shown in Figure 4.10 and Table 4.1.

The results in Figure 4.10 show that the code generation method takes advantage of the features available on the newer GPU platforms. On average, code generation for the S1070 GPU leads to 1.44× speedup compared to G8800, while, on the S2050, the performance is 2.62× better than the S1070. This significant improvement is due to the additional processing cores and to the larger SM memory. Combined, these allowed a larger total number W of parallel stream executions.

Table 4.1: The versatility of the code generation method.

Benchmark	**Λ** in words	**G8800, S1070** configuration ①			**S2050** configuration ②			Method benefit	
		W	S	F	W	S	F	Speedup ② / ①	SM activity
Bitonic(8)	31	128	1	64	352	1	160	2.32×	98.8%
BitonicRec(8)	31	128	1	64	352	1	192	2.31×	99%
DCT(8)	193	20	4	32	62	4	96	2.95×	99.6%
DES(16)	95	40	2	32	128	2	32	2.78×	99.6%
FFT(32)	129	30	4	32	88	4	96	2.46×	99.5%
FilterBank(4)	49	46	4	32	88	4	32	1.95×	99.5%
FMRadio(8)	27	88	1	32	352	1	32	3.15×	99.2%
MatrixMult2(2)	209	19	4	32	56	4	96	2.84×	99.1%

Table 4.1 shows the benchmark configurations generated for these platforms, as well as the additional speedup achieved by using a platform-specific configuration on S2050 (speedup ②/①). The suitability of this approach is also reflected by the SM activity metric which can be derived through profiling. The SM activity reflects the number of active cycles (cycles when the scheduler can identify a warp which is ready to issue an instruction) as a fraction of the total number of issue cycles. As the code generation method described above ensures that $\mathcal{C}$ warps are never stalled by global memory access, a very small number of inactive cycles is observed.

4.5 Summary

This chapter described a novel and efficient GPU code generation method that exploits coarse-grained parallelism. This method involves pipelining the activity of dedicated memory access threads that prefetch data from the off-chip memory to the on-chip memory, and that of compute threads, which are disconnected from the off-chip memory. This method supports all the described features of the StreamIt language, except filters with internal state. Compared with previous results of compiling StreamIt to GPU, this code generation method performs by as much as 4.2× better, on the same experimental setup.

The performance characterization shows the non-trivial trade-off between memory access and compute threads. A heuristic assists in automatically selecting the best code generation parameters. All the benchmarks were implemented within a single partition. However, in some cases the working set grows too

large, and this method requires the partitioning of the stream graph into multiple sub-graphs for optimized execution, using the off-chip memory as intermediate storage. An extension method that supports this idea is described in the next chapter.

CHAPTER 5

STREAMIT CODE GENERATION FOR MULTIPLE GPUS

The method in Chapter 4 shows how StreamIt applications can target single GPU platforms. The code generation method leads to a monolithic implementation that has to match the memory requirement of the entire application with the small amount of memory available on the GPU platform. Therefore, effective solutions are feasible only if the memory requirement for the entire stream graph is less than or equal to the capacity of the on-chip SM memory.

In order to deal with the increased memory requirement from large streaming applications, the stream graphs of these applications can be divided into partitions (sub-graphs) whose memory requirements match the SM memory constraint. To improve scalability, the code generation method described in this chapter targets a platform which consists of a multi-core CPU and multiple GPUs attached to it. Altogether, this chapter describes a scalable extension that accommodates the execution of streaming applications onto multi-GPU systems. It takes advantage of the entire processing and memory hierarchy exposed by the combined GPU and CPU platform.

Several key features are included to ensure the scalability required by complex streaming applications. First, the partitioning algorithm is driven directly by memory constraints. Subsequently, the partitions benefit from the efficient architecture-driven code generation described in Chapter 4. The partitions are balanced among the GPU devices and their boundaries take into account the communication overhead. Finally, a highly effective pipeline orchestration is employed for the execution of the partitions on the multi-GPU system.

The code generation method described in Chapter 4 exploits the parallelism exposed by StreamIt and instantiates a mixed pool of compute and memory access threads that maximizes the utilization of the SM memory available on each

streaming multiprocessor. The resulting computation structure also maximizes the utilization of the GPU. However, the previous method is limited to a single partition. Hence, the application performance is severely limited if its working set size grows large, and if not enough parallel threads can be instantiated in the size-limited SM memory.

The method in this chapter can handle complex streaming applications with overall memory requirements larger than the available SM memory. This is achieved through an efficient graph partitioning algorithm that splits the application into several smaller partitions which are valid under SM memory constraint and hence achieve good performance individually.

Scalability is augmented by the ability to distribute the resulting partitions among a set of GPU devices. In addition, one or more partitions can be mapped to the same GPU, resulting in a combined spatial and temporal distribution. Complementary, streaming operators that maintain internal state are mapped to CPU cores. Also, the communication overhead between partitions is considered during the mapping step. This is necessary because the high volume of data streamed between partitions has to pass through one or more of the slower levels of the memory hierarchy as determined by the locality of the partitions.

The extended method is implemented as a back-end of the StreamIt programming language compiler. The comprehensive set of experiments show its scalability and significant performance speedup compared with the solution presented in Chapter 4. The contributions of this chapter include:

- a scalable code generation method that handles complex StreamIt applications (Section 5.1)
- the division of the applications onto several optimized partitions valid under SM memory constraints (Section 5.2).
- the optimized mapping of the partitions to multiple GPUs and the orchestration of their execution (Section 5.3).

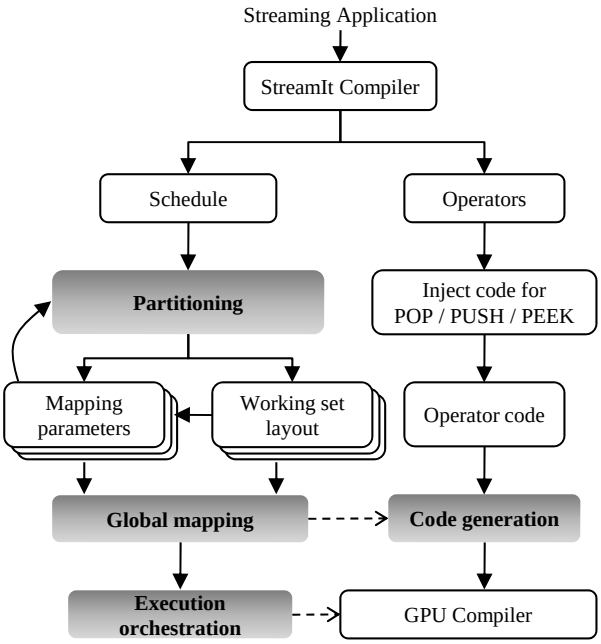


Figure 5.1: Scalable code generation method.

5.1 Code Generation Method

The scalable code generation method builds upon the infrastructure described in Section 4.2. The input to the framework are the applications written in the StreamIt language. The front-end of the StreamIt compiler [31] analyzes the input program to generate a schedule of operators, as well as perform some common optimizations. This schedule is utilized to determine the stream graph dependencies for the graph partitioning component. Also, the push, pop and peek primitives of each operator are enhanced with code that performs the correct accesses to the channels stored in SM memory. The enhanced code of the operators is injected in during code generation.

The stream graph partitioning component prunes the design space by analysing the validity and estimated performance of the possible partitions (details in Section 5.2). The performance estimation considers the specification of the target GPU. The result of this partitioning is a set of convex and disjoint sub-graphs which are ready for mapping to the multiple GPUs. Operators that maintain internal state are included in separate partitions that, as an exception, are executed on the CPU cores. For each GPU partition, a compact memory layout that can be realized in the fast SM memory is computed. Given the memory layout

for each partition, the other parallel code mapping parameters are determined by the heuristics in Section 4.3: (1) the number of stream schedules that are executed in parallel in each SM, (2) the number of $\mathcal{C}$ threads that accelerate the execution of each individual stream execution, and (3) the number of dedicated $\mathcal{M}$ threads that prefetch data from the GPU global memory.

Finally, the resulting set of partitions is passed to a global mapping step which assigns each partition to a specific GPU or CPU core. At this stage, communication channels between partitions are instantiated (details in Section 5.3). The generated code is orchestrated by an execution environment which contains: (1) a multi-threaded controller which will run on the host CPU, and (2) the inter-partition memory communication scheme for pipelined execution. The controller consists of threads that coordinate the kernels loaded on each GPU, as well as threads that execute the CPU partitions.

5.2 Partitioning of the Stream Graph

Given a stream graph, the objective of partitioning is to maximize the overall performance of the stream graph, while ensuring that the partitions satisfy resource constraints, and yet effectively utilize the GPU.

A stream graph $G(V, C)$ represents the data flow within the stream application. The nodes V represent the operators and the edges C represent the channels (dependencies) between the operators. A channel connects the output of a producer operator to the input of a consumer operator. As advanced features of StreamIt such as feedback looks and portals are not supported, $G(V, C)$ is always a directed acyclic graph.

A partition P must be a convex subgraph, as non-convex subgraphs cause heavy communication to the adjacent subgraphs. Even worse, they may lead to deadlocks. P is convex if there does not exist a path in $G(V, C)$ from an operator $V_m \in P$ to another operator $V_n \in P$, which contains an operator $V_p \notin P$.

The method employed to identify suitable partitions relies on the well-known k-way graph partitioning, a well studied algorithm in the research community [46]. In this algorithm, the nodes of a graph are partitioned into k roughly

equal partitions so that the weight of the edges between nodes in different partitions (edge-cut) is minimized. Intuitively, this results in load balanced partitions that have minimum communication. However, the partitioning described in this chapter differs from the standard algorithm in several aspects: (1) the number of partitions k is not an input to the problem – the value of k will be determined during the run-time of the algorithm such that it maximizes the overall performance, (2) there are convexity and memory constraints on each partition – these constraints affect the performance of combined partitions, and (3) the objective of the algorithm is to maximize the performance of the application. The performance objective is estimated as $\sum_{i=1...k} T(P_i)$, where T is an estimation of the execution time of P_i. Even if multiple GPUs are utilized, a balanced distribution of the partitions to the multiple GPUs ensures that this objective continues to reflect the overall stream graph execution performance.

Nevertheless, the *multilevel graph partitioning* (MLGP) algorithm used to solve the k-way problem can be effectively employed to solve this problem. In MLGP, the nodes in the original graph are grouped to create coarser nodes (the *coarsening phase*). The original graph is iteratively coarsened down to k partitions, over a number of levels, in order to create the initial partitioning solution (the *partitioning phase*). Then, the initial solution is *uncoarsened* back to the original graph by using the same number of levels as in the coarsening phase. While uncoarsening, the partitioning solution is refined by the movement of nodes to adjacent partitions so as to improve the overall performance.

A recently proposed multi-level algorithm [44] is adapted to this graph partitioning problem. The strategy is to continue to decrease the number of partitions of the stream graph as long as the overall performance of the entire stream graph is still increasing. As no particular value k is provided as input to the graph partitioning, this limit is removed from the MLGP algorithm. Alternatively, the number of partitions of the solution is the number of nodes in the coarsest graph obtained. The details of the coarsening and uncoarsening phases are described below.

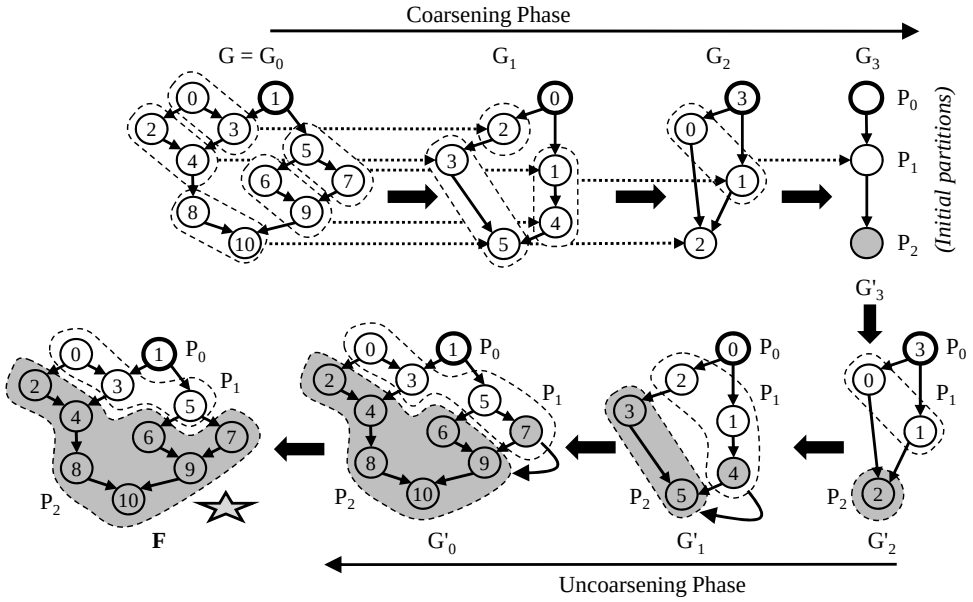


Figure 5.2: Illustration of Multi-Level Graph Partitioning. The dashed lines show the projection of a vertex from a coarser graph to a finer graph.

5.2.1 Coarsening Phase

A sequence of coarser graphs $G_i = (V_i, C_i)$ are created from the original directed stream graph $G = (V, C)$, by clustering together pairs of nodes. A node $u \in V_{i+1}$ in a coarsened graph G_{i+1} at level $i + 1$ is the result of merging two *matching* nodes $v, w \in V_i$ of the finer graph G_i at level i such that u is convex and can be implemented on the GPU. Otherwise, if no convex combination can be identified, node u is simply set to vertex $v \in V_i$ of G_i. Note that each node u in a coarse graph is a sub-graph of G_0 when projected from the constituent nodes of u in the finer graph. In G_2 of Figure 5.2, the sub-graph corresponding to coarse vertices {0,1} consists of vertices {0,3,5,6,7,9} of G_0. After constructing coarser nodes, the edges C_{i+1} of the coarser graph are also derived. A directed edge between two nodes in coarser graph G_{i+1} is built if there exists a directed edge between their constituent nodes in the finer graph G_i.

The proposed matching heuristics visits the nodes of G_i in random order. An unmatched node $v \in V_i$ is selected for matching to create a node u in the coarser graph G_{i+1}. The adjacent unmatched nodes of v are iterated to find a possible match w under the convexity constraint. Only the adjacent nodes are considered because significant communication overhead among coarse nodes

will occur if non-adjacent nodes are merged. Nodes v and w are matched if the matching returns the best performance gain. The performance gain is defined as: $\Delta T = T(w) + T(v) - T(u)$.

The estimation of T is based on the amount of SM memory required by the sub-graph executions, because this determines the number of sub-graph executions that can run in parallel. The SM memory requirement is derived through channel layout analysis. While Section 4.2.3 describes an algorithm which considers the influence of fragmentation on the channel layout, the complexity of the partitioning algorithm can be reduced by using an estimation which does not consider the effect of fragmentation.

In case a feasible matching for v can not be found, u inherits only a single node v. In Figure 5.2, nodes 1 and 4 of G_1 are matched to form node 1 of G_2 while node 2 of G_1 is assigned to node 0 of G_2. Note that filters that maintain state are more suited for CPU execution (i.e. node 1 in G_0), because they can not be parallelized. Therefore, these operators will not be matched as they will be included in special partitions to be mapped to CPU cores.

If the graph cannot be coarsened any further, i.e. $G_{i+1} = G_i$, the coarsening phase ends. Let $G_m = (V_m, C_m)$ be the coarsest graph achieved. The initial partitioning solution utilizes this configuration, and each node $v \in V_m$ is selected as a partition. The number of partitions, k, is just $|V_m|$. This value is not an input as is the case in the standard k-way problem, but it is only determined when the coarsest graph is reached. These initial partitions will be refined during the uncoarsening phase to project back to G_0. In Figure 5.2, the coarsening phase goes through a sequence of coarse graphs $\{G_0, G_1, G_2, G_3\}$ and the initial coarsening leads to three partitions P_0, P_1 and P_2.

5.2.2 Uncoarsening Phase

From the coarsest graph G_m, the initial partitions are projected back to the original graph by traversing a sequence of finer graphs $G'_{m-1}, \ldots, G'_0$, where G'_i is a refinement of G_i. During this uncoarsening process, it is necessary to trace the partition to which the finer nodes belong. Let $P(v)$ be the partition

assignment for a node v. Each node of the coarsest graph G_m represents a partition, so, utilizing this notation, $P(v_i) = P_i$ ($v_i \in V_m$). Because nodes in a level $i+1$ graph include one or two nodes from the level i graph, the partitioning information can easily be propagated through all the levels.

Moving nodes from one partition to another may yield improvements. Because G_j is less coarse than G_{j+1}, there is more freedom to move the nodes in G_j. The movement may reduce the communication or increase the combined performance of the partitions after the move. There are local movement heuristics [47, 24] which can yield good results for bi-partitioned graphs. However, using these heuristics in a k-way problem leads to significant complexity, because a node from a partition can move to several other partitions. Instead, the method described here relies on an efficient greedy refinement algorithm [46].

This algorithm tries to move the boundary nodes of a partition to the adjacent partitions. A boundary node of partition P_i in coarse graph $G_j = (V_j, C_j)$ is a node $v \in V_j$ that has at least one adjacent node $u \in V_j$ that belongs to a different partitions ($P(v) \neq P(u)$). In Figure 5.2, nodes {2,4,6,9} in G'_0 are the boundary nodes of partition P_2, while {8,10} are the internal nodes. A boundary node v is randomly selected and moved from partition $P(v)$ (the *source partition*) to the neighborhood partitions $P(u)$ (the *destination partition*). For G'_0 in Figure 5.2, a neighborhood partition of node 7 is P_2.

After moving node v from source partition $P(v)$ to destination partition $P(u)$, if the source and destination partitions still satisfy the convexity and SM memory constraints, the movement is deemed valid and would transform the two original partitions $P(v)$ and $P(u)$ into the new partitions $P(v)'$ and $P(u)'$. Among the valid movements of the boundary node v to the neighborhood partitions, the movement which has the highest $\Delta T = T(P(v)) + T(P(u)) - T(P(v)') - T(P(u)')$ is selected and node v is moved to the particular destination partition. The source and destination partitions are updated respectively. The moved nodes will not be considered again for analysis during the current coarsening level. The movement algorithm for the current level stops if there are no more boundary nodes to move. Once the movement algorithm for G_i finishes, the uncoarsening

phase continues by projecting back to G_{i-1}. Finally, after G'_0 is analysed, the final assignment of the nodes to partitions is produced. In Figure 5.2, node 4 is moved from P_1 to P_2 when G'_1 is analysed, and node 7 is moved when G'_0 is analysed. The result is the partitioned graph F, which captures the refinements to the partitioning solution.

5.3 Execution on Multiple GPUs

After the original stream graph is partitioned, as described in the previous section, these partitions are mapped onto the multiple GPU and CPU cores such that the workload is balanced. This section describes how the mapping is achieved. Then, it describes the execution model that ensures the efficient utilization of the mapped partitions.

Each partition that belongs to the solution F obtained in the previous section forms a single node in the coarsest graph G_m. An edge between any two level $i+1$ nodes exists if there is an edge connecting constituent nodes of those two coarse nodes in G_i. The edge is assigned a *weight* which is the sum of the communication overhead of the level i edges. The communication overhead is the ratio between the data amount exchanged by two level $i+1$ nodes, and the memory transfer bandwidth between CPU and GPU. In order to map G_m onto a system with x GPU devices, the partitions in G_m are distributed onto the x processing elements with the objectives: (1) the load should be balanced, and (2) the overhead of the communication edges should be minimized. The k-way partitioning algorithm is a good match for this mapping problem because it splits the nodes of a graph into x roughly equal partitions, such that the communication between the different partitions is minimized.

An exception are the partitions that maintain internal state. They correspond to individual filters due to the coarsening restrictions from Section 5.2. These partitions are pre-mapped to CPU threads and are not included in this second k-way partitioning pass. The remaining partitions are analysed, and divided among the x GPU devices.

5.3.1 Communication Channels

After mapping, each GPU has to multiplex the execution of the several partitions assigned to it. A GPU kernel is generated for each partition and additional code is inserted to assist with the pipelined execution of the partitions. The execution schedule on each GPU is coordinated by a dedicated CPU thread. Additional CPU threads are launched to support the CPU partitions.

The entire execution schedule utilises FIFO (First In First Out) channels for data transfer. The FIFO length ensures that there will be no stall during the execution. Each FIFO element contains data corresponding to a large number of stream executions. This coarse data granularity takes advantage of the exposed data parallelism, and hides the unnecessary overhead of handling data separately for independent iterations within a partition. Overall, three levels of data transfer are employed between: (1) the different partitions, (2) the asynchronous launches of the partition kernel using GPU streams, and (3) the compute $\mathcal{C}$ and memory access $\mathcal{M}$ threads inside each GPU partition.

Level 1 data transfer A number of $x + y$ CPU threads is spawn to manage the parallel execution on the x GPU devices and the y additional threads supporting CPU partitions. CPU synchronization primitives ensure uncorrupted access to the channels between the partitions. The FIFO channels between two CPU partitions or two partitions executed on the same GPU employ a standard circular buffer where memory pointers are passed directly between the threads.

However, when data needs to be transferred between CPU and GPU partitions, an additional buffering scheme that copies the channel data from CPU/GPU memory to GPU/CPU memory is used. For example, in Figure 5.3, the data in channel C_3 between partition P_3 (on GPU_1) and partition P_4 (on CPU) requires this buffering scheme.

Finally, in order to transfer data between partitions on different GPU devices, the data is always copied first to the CPU, where the FIFO channel is implemented. Afterwards, data is copied from CPU memory to the other GPU using the buffering scheme described above. In Figure 5.3, the output buffer of

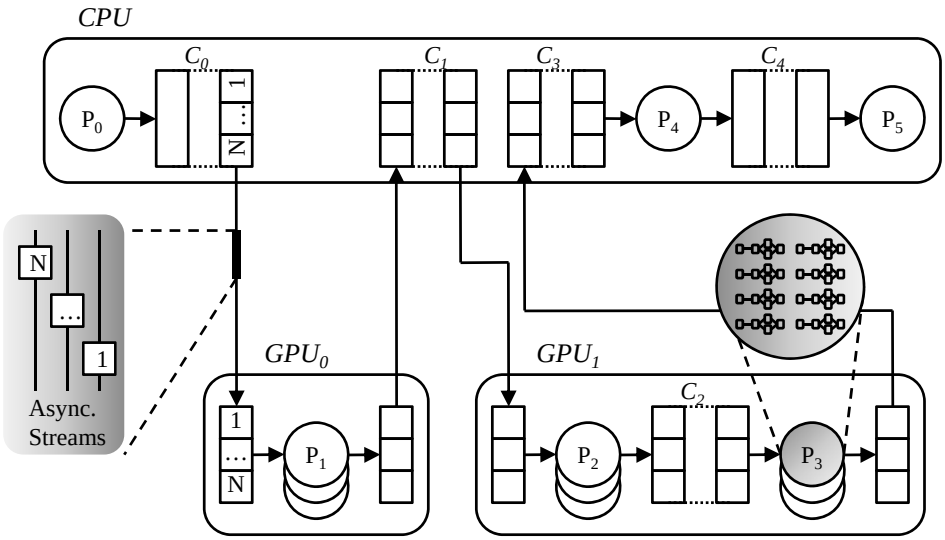


Figure 5.3: Execution and data transfer among partitions on a multi-GPU system.

partition P_1 (in GPU_0) is copied to the channel C_1 in CPU memory and data from this channel is copied to the input buffer of partition P_2 (in GPU_1).

This communication scheme between partitions on different GPUs can be easily adapted to the recent peer-to-peer memory access in CUDA 4.0 [67]. Note that peer-to-peer memory access is specific to nVidia GPUs. Moreover, in order to use peer-to-peer memory access for pipelined execution, synchronization among different CPU threads is still required to ensure that memory accesses are uncorrupted. More importantly, peer-to-peer memory copy between two GPUs can not be initiated until all commands previously issued to either GPU have completed, and has to complete before any asynchronous commands issued after the copy to either GPU can start. This may downgrade the benefit of peer-to-peer communication in comparison with communication through CPU, which can benefit from the support of asynchronous GPU streams.

Level 2 data transfer The asynchronous streaming support for the GPU devices is utilized to hide the CPU/GPU memory copy overhead. The coarse data elements from the FIFO channels are divided into smaller fragments. A stream of asynchronous memory copy and partition kernel launch requests is generated to process the GPU copy and execution. As the operations on these

fragments are independent, memory transfers and kernel executions of different fragments can overlap. N fragments are created, as shown in Figure 5.3. Each stream will correlate data transfer and execution for its corresponding fragment. While Stream 1 is performing computation of fragment 1 in GPU_0, Stream 2 can transfer fragment 2 from CPU to GPU_0.

If several partitions are mapped to a single GPU, these partitions are time multiplexed. In Figure 5.3, the execution of partition P_3 and P_2 is interleaved.

Level 3 data transfer Each GPU kernel executes multiple iterations over the group of parallel executions of the stream graph partition it includes. Using a mix of $\mathcal{C}$ and $\mathcal{M}$ threads, data can be prefetched and computed without stalls inside each SM. This heterogeneous scheme can be executed efficiently on the GPU architecture as long as $\mathcal{C}$ and $\mathcal{M}$ threads are allocated into different *warps*. In such an implementation, $\mathcal{C}$ threads never access slow GPU memory and can compute a larger number of stream graph executions using exclusively a small working set *WS* stored in SM memory. Concurrently, $\mathcal{M}$ threads fetch the next input data from GPU memory to a double buffer *DB* in SM memory and store back the previous output data. A single synchronization point is required, when prefetched data from *DB* is swapped in *WS* and the previously computed results are swapped out from *WS* to *DB*.

The stream graph partitions supported by this implementation are connected through multiple input and output channels. Their corresponding data should be swapped between *WS* and *DB*. This is trivial only when a single input and output channel is involved, a scheme described in Section 4.2.3. In this case, the input channel corresponds to a contiguous range of memory locations, which overlaps with the output channel in *DB* and may also overlap in *WS*. Simply iterating through the data stored in *DB* in the correct direction ensures that no data is corrupted, and it is possible to swap data in parallel using multiple GPU threads. However, special care is required to support multiple channels.

The *WS* memory range corresponding to the channels of each graph operator is determined by a static memory allocator, based on liveness analysis. This

allocator ensures that the channels receive a contiguous memory range (as a result gaps may occur between channels). The input and output channels of a stream graph partition are also stored in *WS*, and the allocator may place them arbitrarily. If multiple channels need to be swapped from *WS* to *DB* and no additional constraints are in place, the actual location of the channels can lead to long dependency chains which may prevent swapping pairs of elements.

A possible scenario is shown in Figure 5.4a. The shaded boxes are the current channels in *WS* and *DB*. In this example the elements from input I_0 can not be swapped into their designated location in *WS* as long as the contents of the output channel O_1 has not been swapped out. However, this output channel can not be moved as it will corrupt I_1 which has not been processed yet. Also, I_1 can not be processed as it will corrupt O_0, etc. Utilizing temporary memory storage is not feasible because the SM memory is limited and any extension degrades performance. Therefore, the proposed extension of the single channel swapping scheme ensures that single element swaps can proceed without data corruption.

The static allocator is directed to layout the input channels without fragmentation from the first location in *WS*. This is possible as there are no previous data in *WS*. However, the range of the output channels may not be contiguous. Nevertheless, the order in which they are allocated can be recorded. The same order is replicated in the *DB*, where both input and output channels can be allocated contiguously. Such a layout is illustrated in Figure 5.4b.

Using this layout guarantees corruption free swapping and this can be proved through induction on the index in *DB*. The basis case is for the first location in *DB*. The input stored at location 0 in *DB* can be moved to *WS*, and any output value it overwrites in *WS* can be moved to *DB* at the same location, because the outputs are compacted, in order, in *DB*. This can be implemented by storing the output first in a temporary register, and saving it afterwards to *DB*. If no output element exists in *WS* at location 0, there still obviously is no data corruption.

Assuming there is no data corruption until index $p-1$ in *DB*, when the input element at index p in *DB* is moved to location p in *WS* it may overwrite an unmoved output. In this case, the overwritten output element has to be moved

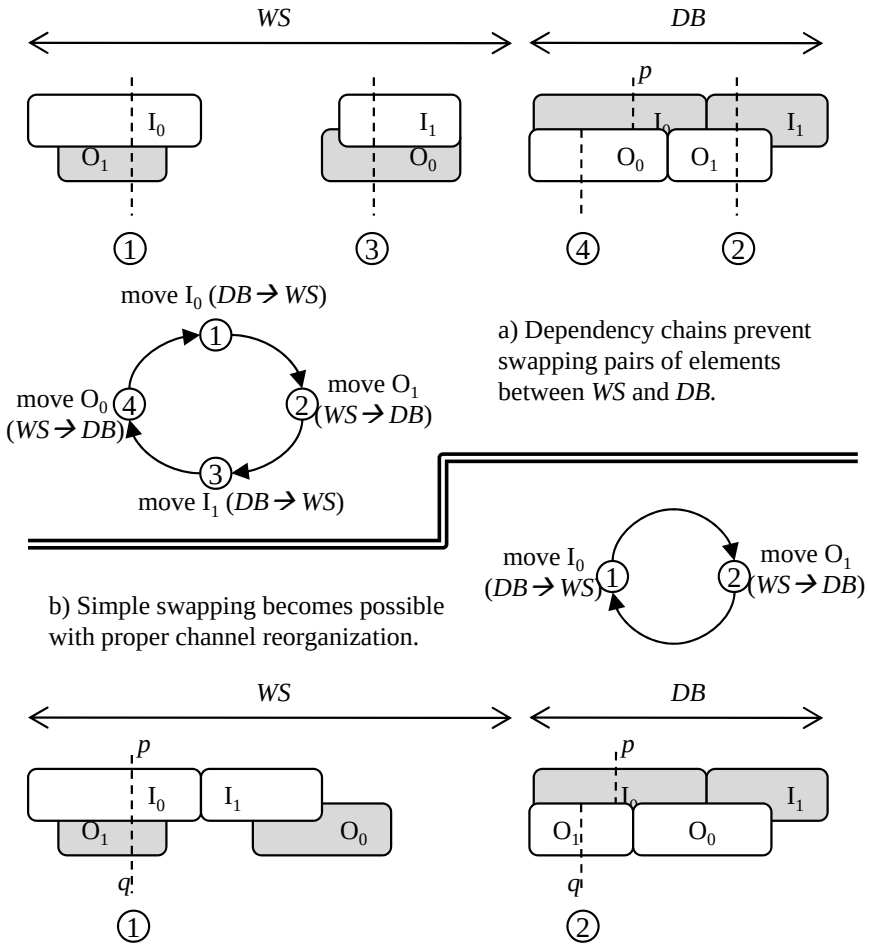


Figure 5.4: Execution snapshot showing the challenges of partition I/O handling. The inputs for the next iteration have to swap with the outputs of the previous iteration.

to index $q \leq p$ in the contiguous sequence of outputs in *DB*. This inequality is ensured by the contiguous allocation of input buffers in *WS* and *DB*, and the possible fragmentation of the outputs in *WS*. Therefore, the movement of the output to the index q in *DB* does not corrupt any input not yet transferred. This concludes the induction case if the number of inputs is larger than the number of outputs. Otherwise, the remaining outputs can be transferred to *DB* safely, as there is no remaining input in *DB*.

The automatically generated code relies on the above channel allocation. A set of intervals is determined, such that the swap indices for both input and output increase linearly. For each such interval, swaps can be applied to pairs of elements at consecutive locations.

5.3.2 Mapping Parameters Selection

Code generation for each GPU partition requires a few parameters, such as the number of $\mathcal{C}$ and $\mathcal{M}$ threads, and the number S of parallel $\mathcal{C}$ threads supporting each stream graph partition execution. These parameters are computed using the method described in Section 4.3. In summary, the number of concurrent executions is determined based on the SM memory size and the memory requirement of each stream graph partition execution. Then, S threads are allocated for each partition execution, exploiting the data parallelism of the partition extracted through the stream graph structure. Finally, the data transfer requirements of the $\mathcal{C}$ threads are matched with a corresponding number of $\mathcal{M}$ threads to minimize the stalls. The same parameters are replicated for all SM, as each SM process parallel fragments of the input data. These parameters were estimated once during the performance evaluation of each partition in Section 5.2. However, during the final code generation step, the exact SM memory layout is derived, and the resulting footprint may increase due to fragmentation.

The number of N concurrent GPU streams utilized for the level 2 data transfer can influence how much of the CPU to GPU data transfer overhead is hidden. However, there is some penalty associated with each GPU partition kernel launch, and this surfaces if too many concurrent streams are utilized. The current implementation utilizes 4 parallel streams to provide a good coverage of the memory transfer delays.

5.4 Results

In order to show the scalability and efficiency of this extended method, its performance is compared with the benchmarks included in Section 4.4. These benchmarks were processed automatically, and code was generated for multiple partitions. The benchmarks were altered to create larger stream graphs by utilising a parameter N (i.e. the graph of DES for $N = 40$ reached 1047 filters). The stream graphs are mapped onto one to four GPU devices connected to the same CPU host. The benchmarks were augmented with source and sink filters that include code to verify the results of the computations. Because these filters

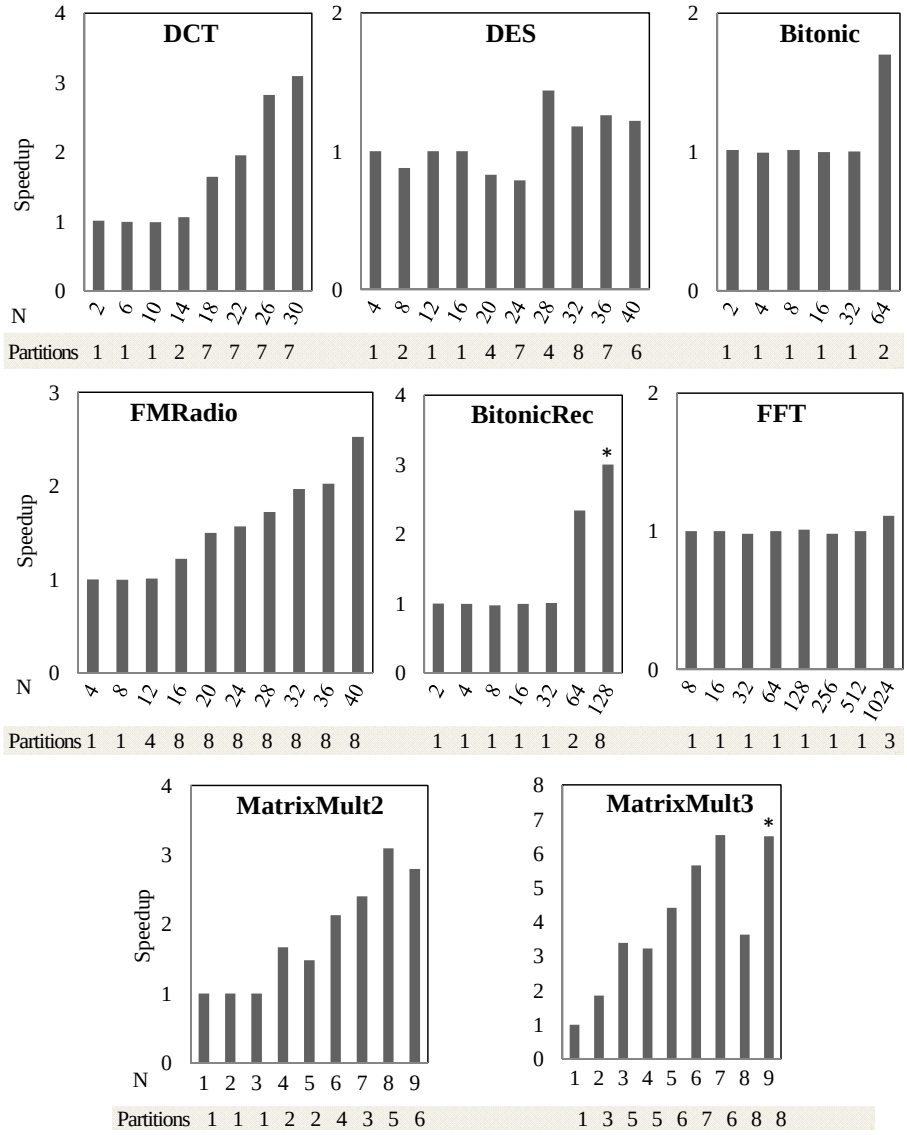


Figure 5.5: Mapping to a single partition and to multiple partitions (the number of partitions is listed under the graphs) on a single GPU. The speedup is the execution time ratio between the two. Design points marked with (*) were not supported by the single partition implementation in Chapter 4

maintain internal state, this also validated the support provided for such filters. The extended method is built as a back-end for the StreamIt 2.1.1 compiler. The baseline CPU timing was obtained on an Intel Xeon E5405 running at 2 GHz, with the executable generated through the uniprocessor back-end of StreamIt, and compiled using the '-O3' option on GCC 4.1.2. The experiments target the newer C2070 "Fermi" GPU platforms.

Comparison with the single partition mapping Figure 5.5 shows the speedup achieved on a single GPU compared to the single partition approach

described in the previous chapter. It also shows the number of partitions generated for each benchmark instance. Most benchmarks benefit from multiple partitions when N increases. Using the proposed algorithm, multiple partitions yielded better performance than a single partition, because each partition requires a much smaller memory footprint. If only a single partition is used, the large working set resulted in poorer performance. To capture the CPU to GPU transfer overhead, the benchmarks maintain stateful source / sink filters. The additional speedup can be as high as 6.53×. In addition, there are a few cases where a single partition mapping could not return a solution (such as MatrixMult3 for size 9). However, for some benchmarks, such as Bitonic, DES or FFT, the working set size does not change significantly, and both single and multiple partitions mappings had similar performance.

A comparison with a vendor-provided hand-tuned implementation of matrix multiply is necessary to increase the relevance of these results. The blocked MatrixMult ($N = 8 \rightarrow 16 \times 16$ matrices) was compared to a similar matrix product handled by CUBLAS. The latter runs 1.6× faster than the automatically generated code on the Tesla C2070.

Multiple partitions on a single GPU Figure 5.6 shows the speedup of the proposed partitioning approach relative to the CPU baseline. While the speedup may diminish for large values of N, this approach proves capable of sustaining good throughput for most benchmarks. If the size of the benchmark is too large to fit the SM memory, the benchmark is split into multiple partitions. In some cases, the overhead of data communication among the partitions severely impacted performance.

Multi-GPU mapping Results beyond the single GPU speedups shown above can be obtained when applying the code generation method to large benchmarks using the orchestration described in Section 5.3. The speedup obtained by running the benchmarks on 2 to 4 GPUs compared to a single GPU mapping is shown in Figure 5.7. In general, when the size of the benchmark is not large enough, multiple GPUs do not provide any benefit. In these cases, a single GPU

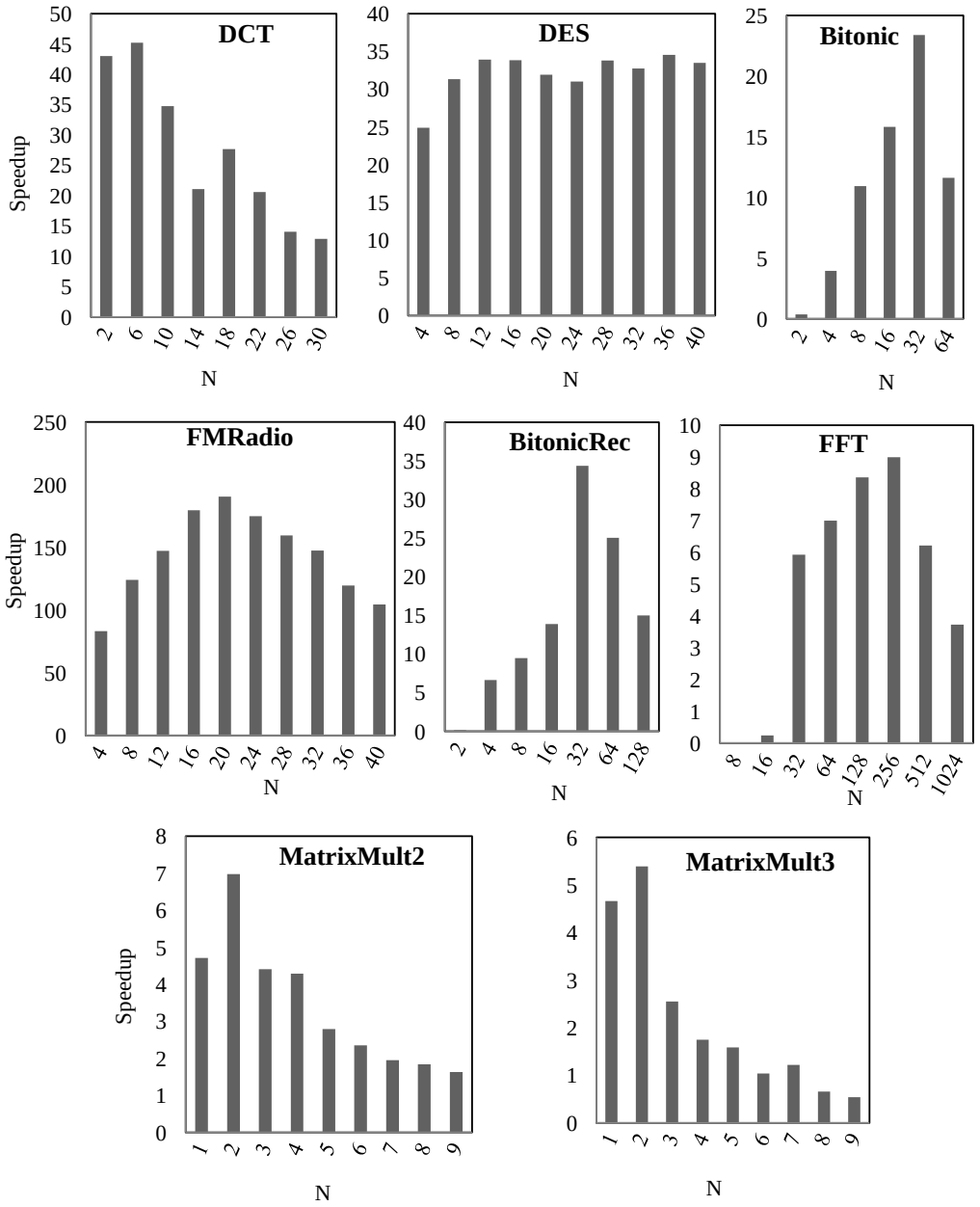


Figure 5.6: Mapping to a single GPU. The speedup is reported relative to a CPU implementation.

mapping is the best solution. This is mainly due to the communication overhead of transferring the data between the GPU and the CPU that could not be completely masked by computation. The single GPU implementation corresponds to the white bars in the figure.

However, the multi-GPU implementation proves profitable if N increases. As shown in Figure 5.6, the speedup of the single GPU mapping diminishes for large benchmarks. However, in this case, mapping to multiple GPUs starts to

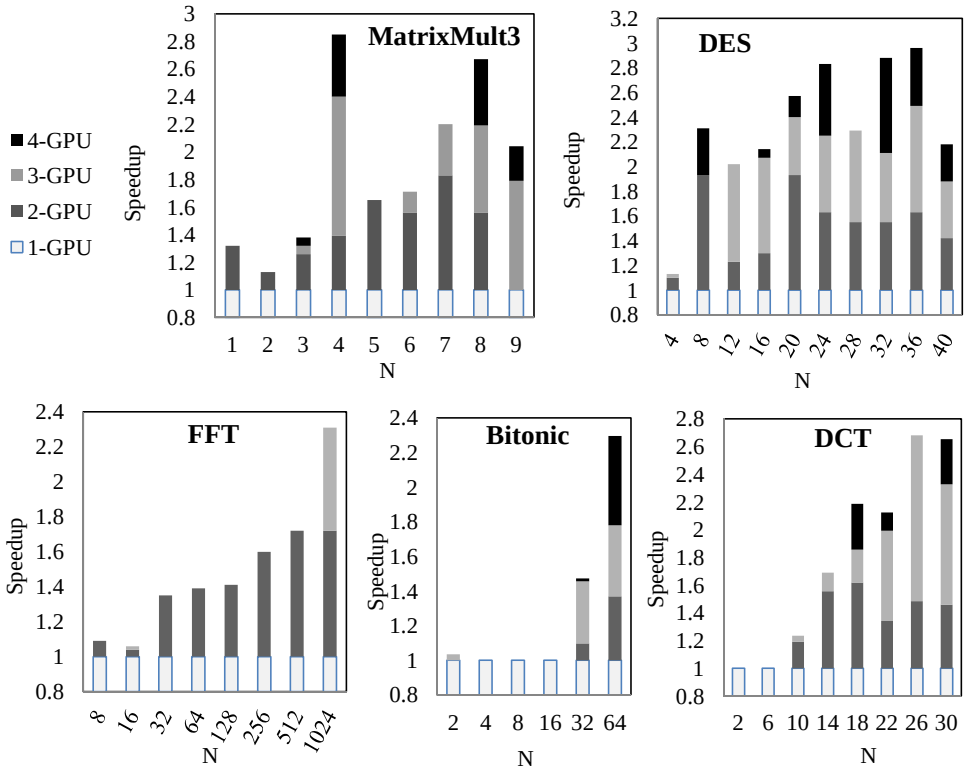


Figure 5.7: Additional speedup resulted from the mapping to multiple GPUs compared to a single GPU.

show its advantages. The speedup reaches 2.97× compared to a single GPU mapping. This is evidence that for applications that have large working sets, this multi-GPU solution can effectively speed up their execution.

Mapping to multiple GPUs (Figure 5.7) shows some divergent performance results for different values of N. The divergence can be explained because the nature of the stream graph itself may lead to solutions that are easily balanced on a specific number of GPUs, and adding additional GPUs may affect the balancing. Moreover, if significant communication exists between fine-grained filters, the performance will hardly increase if we put those filters across multiple GPUs.

Moreover, Figure 5.7 offers an indirect insight that the communication overhead can be effectively masked. The performance boost of a 2 GPU solution, compared to that achieved on a single GPU, is affected by several factors (such as how the workload is balanced between the 2 GPUs) in addition to the overhead of the complex communication mechanism. However, some design points

of DES and MatrixMult3 mapped to 2 GPUs reach 1.93x and 1.83x speedup respectively, compared to a single GPU solution that does not have inter-GPU communication.

5.5 Summary

The method proposed in this chapter is capable of automatically generating code for large stream processing applications onto multi-GPU systems. It relies on an efficient graph partitioning algorithm to split the complex application into several partitions that can utilize the small SM memory effectively, and hence achieve good performance. The results indicate that this method augments the initial method described in Chapter 4. In addition, this method is able to scale the performance to up to four GPUs. The method also supports stateful filters by running them on the CPU cores. The code generation scheme proposed is able to orchestrate the exchange of data withing the individual GPUs, between the multiple GPUs, as well as the GPUs and the CPU cores. The results indicate the scalability and improvement when several GPUs are targeted.

It is conceivable that certain embarrassingly parallel applications can be mapped successfully to large scale multi-node, multi-GPU systems. However, on a single node, it is unlikely that the number of GPUs per node will increase significantly beyond the current four due to power, interconnect, and form-factor issues. This work is the first to show that complex and often tightly coupled streaming applications can be successfully partitioned and mapped automatically onto multi-GPU systems.

CHAPTER 6

FLOATING-POINT SIMD COPROCESSORS ON FPGAS

The previous chapters showed how code generation methods can take advantage of the extensive coarse-grained parallelism exposed by StreamIt. Nevertheless, significant opportunities for design improvement can be found in the fine-grained data parallelism, which abounds in some application domains. This is particularly important for low-power embedded devices, which seldom have the hardware capability to use this fine-grained parallelism.

Embedding hard-wired processor cores in FPGAs offers a design path that can meet both the performance and the energy demands. These processors can be customized using FPGA resources, and this process reduces the design effort, and indirectly the time to market. *Floating-point* execution units were not considered by existing customization tools, as they were deemed to consume too many resources. However, floating-point execution units offer the right granularity to justify sharing and parallelization opportunities in the recent and larger FPGA circuits. Moreover, floating-point support is seldom included in the hard-wired processor itself.

This chapter describes a method for accelerating floating-point computation for embedded platforms via application-specific SIMD coprocessors implemented in FPGAs. It consists of a co-design method that generates code *and* application-specific coprocessors that implement vector instructions with a parameterizable number of vector elements. The flexible parallelism captured by encapsulating computation in vector instructions is matched to an adjustable pool of execution units implemented in FPGA hardware.

Embedded applications vary widely in their code structure and profile. As they are often subjected to serious power and resource constraints, specialized hardware coprocessors are added conservatively. Designers of such applications

often use an FPGA to supplement their silicon processor [29, 95]. The FPGAs are particularly effective when there is a large amount of instruction-level parallelism. Although floating point applications often come with a high amount of parallelism, they were deemed to consume too many resources, and the desired performance could only be obtained by hand-tuning the application, including conversion to fixed point and then tackling the precision issue. Yet, floating point computation is often the most straight-forward means of expressing an algorithm, especially where fractions and accuracy are involved, and recent advances in FPGA architecture have made such implementations feasible. This chapter describes a method that exploits *automatically* the parallelism available in floating point code, using a co-design method that also generates the HDL code for a SIMD coprocessor able to support the custom vector operations derived.

A standard architecture consists of a hard-wired processor core coupled with FPGA reconfigurable resources [1, 81]. The reconfigurable resources offer flexibility, but one cannot possibly hope to match the speed and efficiency of a silicon processor core. On the other hand, a silicon processor core with full vector capabilities like those found in desktop- and server-class processors would mean committing silicon without consideration for the applications' requirements. A mixed approach requires a dedicated interconnect, and the combined performance is affected by the partitioning strategy and the data transfer overhead. Fine-grained partitioning proves beneficial only where a fast interconnect is available, for example, when both the processor core and the reconfigurable fabric are placed on the same silicon die.

The alternative method described uses the FPGA to implement floating point units when the need arises [25]. The method takes advantage of existing auto-vectorization capabilities in compilers, and co-synthesizes code and customized floating point SIMD coprocessors in reconfigurable hardware. It relies on an algorithm that determines the maximum performance configurations for the SIMD coprocessor architecture under the given resource constraints.

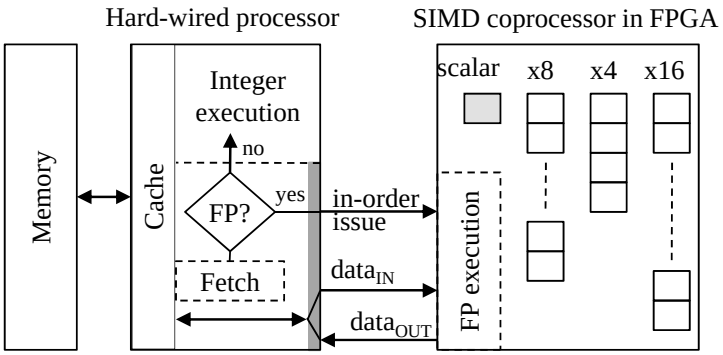


Figure 6.1: The target architecture configuration.

Vector instructions are a natural candidate for FPGA coprocessors [16] because they yield an efficient encoding of short instructions that capture a large number of operations and data transfers. Their regular structure also expresses a significant amount of data parallelism. This parallelism allows flexibility and customization of the number of elements in the vector. Custom vector lengths can be used for the register set, operands and operators. This novel architecture supports *concurrent* execution of vector instructions with different vector lengths. In particular, it supports the concurrent execution of a mix of single precision 4-, 8-, and 16-float long vector instructions[1]. The exact mix used is determined by how the required processing throughput can be matched to the available reconfigurable resources. The best matching is achieved by *folding* the execution of larger vectors when resource is scarce.

In essence, the architecture exposes a *set* of virtual instruction set architectures (determined by instruction level parallelism and other program characteristics) that is implemented by a shared pool of floating point execution units (determined by the reconfigurable resources available). The method inherits from the advantages of both custom instructions and loop accelerators. It offers an alternative at an abstraction level where it is easier to find acceleration, as well as resource sharing opportunities. On top of that, this method offers a tighter integration in the design compilation flow. The novel features of this method can be summarized as follows:

[1]For brevity, in the rest of the chapter, these shall be called 'x4', 'x8', and 'x16', respectively.

- It presents the design of a customizable SIMD floating point coprocessor on a hybrid architecture that includes both a hard-wired processor core and FPGA reconfigurable resources.

- It describes the implementation of a co-design method for this coprocessor, in which both the executable and the HDL code for the optimized coprocessor configuration are automatically generated in an integrated approach.

- It describes a method for further improving resource usage and energy efficiency by the independent folding of each kind of execution units in the final design.

Experiments on actual hardware show increased performance and scalability. This method provides an important extension to the capabilities of FPGAs with hard-wired processors, which traditionally dealt with bit, integer, and low intensity floating point code, to now being able to handle vectorizable floating point computation.

6.1 Rationale

In silicon-based processors, the vector instruction set and the vector length of the SIMD coprocessor are chosen to suit a broad spectrum of computation patterns and instruction level parallelism exposed across all the application domains. However, if the SIMD coprocessor is reconfigurable, then one may choose to implement only a particular set of vector instructions that best benefit the application at hand, and have the system reconfigured to something altogether different when the demand changes. The envisioned system architecture is abstracted in Figure 6.1. Scalar and vector floating-point (FP) instructions are executed outside the hard-wired *processor core*, in the attached FPGA *coprocessor*. These instructions are issued in program order on a dedicated interface. One of the key insights behind this method is that, unlike general integer computation, many floating point applications have the proper granularity to overcome the inherent penalty of issuing instructions outside the processor cores.

In this architecture, load and store instructions have to transfer data through the processor core to the memory. Most instructions are autonomously executed by the FPGA, with the exception of vector stores, which require data computed in the coprocessor to be written back to memory. The latter entails blocking the subsequent instruction issue until the data transfer is complete. Otherwise, for most other types of instructions, new instructions can be issued in consecutive clock cycles.

The following are some of the considerations that affect the selection of the SIMD coprocessor configuration:

- The use of longer vectors will decrease the number of instructions issued to the coprocessor, each instruction encoding coarser-grained computation.
- As the overall number of issued instructions decreases, the performance bottleneck will shift from the instruction issue to the execution stage. There is an opportunity here to reorganize the individual operations encapsulated by each instruction, and determine a compact hardware implementation according to the exposed data dependencies.
- Larger vectors require more data to be transferred before computation can begin. This may cause delays, especially in systems where the memory latency is large. In other words, the use of longer vectors may prevent the effective overlapping of memory transfers with computation.
- The kind of data movement is often limited by what the instructions can do. This can degrade the performance of certain operations, such as data transpositions, or the epilogue of vector reductions.

The following example shows the impact of the selected vector length on the execution time of a loop. Figure 6.2a shows a simple loop expressed in a C-like language compiled for the Altivec instruction set. The vectorization process identifies computation patterns across multiple loop iterations and coalesces them into vector operations. SIMD architectures available today mostly use a vector length of four [23]. This pseudo-compiled code example uses vector registers `vr`

that can store four single precision floating point numbers. The `LV` instructions bring the operands from memory into the registers. `VADD` instructions perform the parallel additions of the corresponding four vector elements, while the `STV` instructions store data back to memory. Because the loop is vectorized, each iteration corresponds to four scalar loop iterations.

Figure 6.2b contains the issue and execution schedule of this vectorized loop as it would be handled by a target architecture with a vector length of four (i.e., x4). It traces the issue and execution of instructions corresponding to two consecutive vector iterations processing elements with base index i and $i + 4$ respectively. The compiler unrolled the loop twice and software pipelined the `LV` and `STV` instructions, placing them in adjacent cycles. Loop unrolling increases register pressure, but is beneficial as it can partially hide the latency of the memory transfer operations and subsequently run the pipelines of the execution units more efficient. Each instruction requires a distinct issue cycle which corresponds to the data transfer between the core processor and the SIMD coprocessor. Once issued, most instructions execute autonomously, spending one or more clock cycles in the pipeline of an execution unit. The exception is the vector store (`STV`) instruction which occupies the issue bus for several cycles until it returns the `vr` to the processor core for subsequent stores to memory. In this example, the critical path consists of the two dependent additions, $\texttt{VADD}_1$ and $\texttt{VADD}_2$. `LV` instructions are software pipelined in the available issue cycles before the start of the current iteration, while `STV` instructions are issued after the end of the current iteration.

However, due to the sequential data exchange between the core processor and the SIMD coprocessor, repeated issuing of vector operations to the coprocessor is costly. This can be detrimental to the overall performance because it limits the issue rate of compute instructions. Alternatively, the same vector loop can be compiled for a vector length of eight (x8), as shown in Figure 6.2c. However, in this architecture, the processor core remains unchanged, and thus all memory transfers, which are routed through the core processor, are split into chunks of four elements. Accordingly, `LV` instructions now require two issue cycles, while

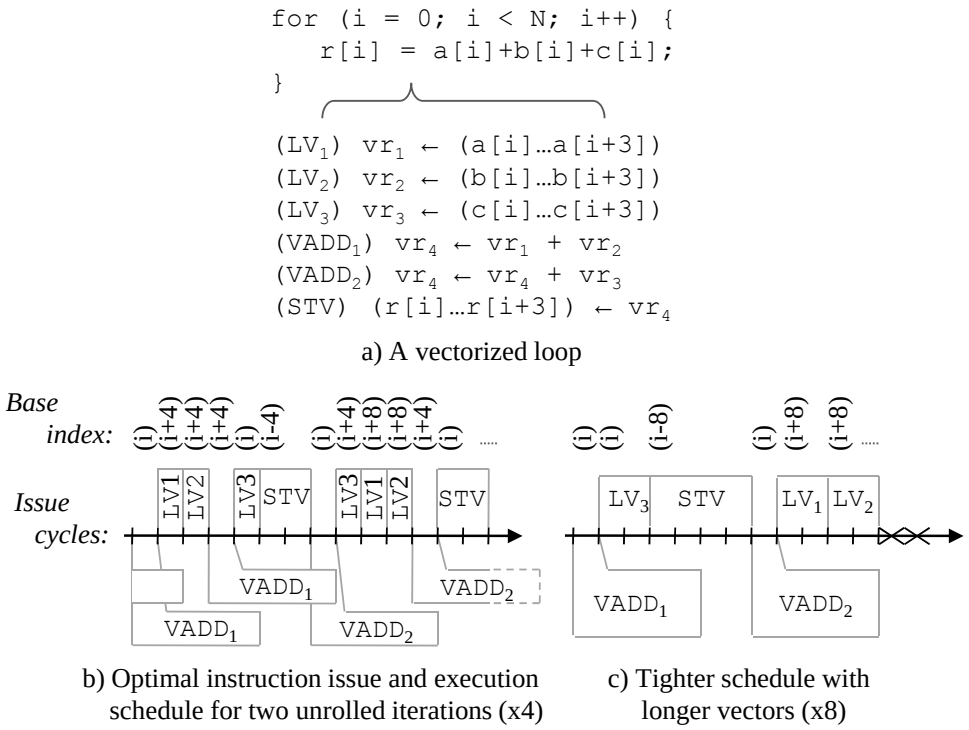


Figure 6.2: Executing a loop using x4 and x8 vector instructions.

`STV` instructions require four cycles to transfer the operands. In this configuration, a single x8 loop iteration can handle the computation previously handled by both x4 loop iterations. The schedule becomes shorter, because less `VADD` instructions are issued.

Of interest is the comparison between the execution time of N iterations of an unrolled vector loop to that of a single equivalent iteration of a vector loop where vectors were lengthened N times. It is assumed that once the instructions are issued, they will execute autonomously. Ideally, performance is maximized if there is an execution schedule where, once instructions are issued, they can execute without delay caused by operand dependencies. Let t_M and t_I be the number of issue cycles used by memory transfer and non-memory transfer instructions in one iteration of the vectorized loop body respectively. In this model, the unrolled version takes $N \cdot (t_M + t_I)$ cycles to complete, while a single iteration of the loop with longer vectors takes $N \cdot t_M + t_I$ cycles. Thus, the alternative approach will improve performance by $\frac{(N-1)t_I}{N(t_M+t_I)}$. For example, if $t_I = t_M$ and $N = 4$, this translates to 37.5% improvement. In practice, this speedup is generally higher for longer vectors, because the compiler generated schedule may not

be able to hide completely the instruction dependencies. However, irregular data movement patterns can make it harder to use longer vectors as additional data movement instructions will be required to correctly marshal data into the vector registers. In the final analysis, which vector length yields better performance depends on the computation pattern and available instructions.

The parallel operations captured by a vector instruction may be associated with a shared set of execution units, thereby trading off performance for lower resource demands. By doing so, the overall execution time may increase if the affected instructions are on the critical path. The profitability of each vector length configuration is evaluated during compilation, and the best is selected based on a static model. In the resulting platform, multiple versions of the same instruction corresponding to different vector lengths may coexist. The architecture allows these versions to share the same execution units. Due to the sizable resources involved, the alternative of switching configurations via run-time FPGA reconfiguration introduces significant overhead, potentially eliminating most of the performance benefits of SIMDization.

6.2 Co-design Method

An established approach for getting good performance in compute-intensive applications is to take advantage of the inherent parallelism. There is an opportunity to do so by using mathematical libraries such as ATLAS [91]. The granularity and flexibility of the computation are key factors in achieving optimized results over large portions of the application. Hence, the trend is to capture computation at a higher level of abstraction such as vectors or matrices which expose flexible parallelism. Libraries rely internally on parallelization techniques such as compiler auto-vectorization to deliver the best performance. Support is required in hardware to support the utilised vector instructions.

Eigen [32] is a C++ template library for linear algebra vectorization, achieving comparable performance to ATLAS and it was suitable for the method described in this chapter, without loss of generality. It includes abstract data structures for vectors and matrices, as well as their related algorithms. Eigen

uses code inlining extensively to automatically rewrite and lower the code representation, applying vectorization where possible. Its recursive template inlining and instantiation removes unnecessary temporaries, thereby building loops with large bodies. Multiple instances of the same function template are generated and inlined in different code fragments, so they can benefit from local optimizations. The term *kernel* would be used to refer to an optimized section of the instantiated code, which has dependencies that are satisfied using (virtual) vector registers. Each kernel may consist of one or more loop nests and the code in between. What is important is that kernels are independent of each other and can be implemented using different vector lengths.

The original Eigen compilation flow is presented in Figure 6.3a. For each computation kernel, the C++ preprocessor uses Eigen library templates and proceeds to recursive inlining, which lowers the computation down to built-in functions matching directly assembler instructions. Internally, Eigen uses a layered instantiation and its lowest level relies on a set of primitive function templates that correspond to the actual vector instructions supported by the target architecture. The resulting executable code contains all the properly vectorized code inlined into the original functions.

The recursive template instantiation mechanism is shown in Figure 6.3b. The C++ computation is expressed in terms of vectors v and matrices m. The C++ code in this example sums each of the columns of m, adding the resulting vector to v_1 and v_2. The template library breaks up this sequence of operations hierarchically into a vector addition, which expects to add the result of $v_1 + v_2$ with the result of *m.colwise().sum()*. At this point, the addition is expressed in terms of flexible packets which will later be transformed to fit the length of the hardware vectors. The inlining process continues by transforming the addition of packets from v_1 and v_2 to a `VADD` instruction, while the matrix column summation is transformed to a loop that sums packets of elements from different rows. This loop and the previous `VADD` are combined in a loop nest in the final assembly code. It is only during the final transformations that the hardware vector length information is utilized. The packet length is propagated as a constant throughout

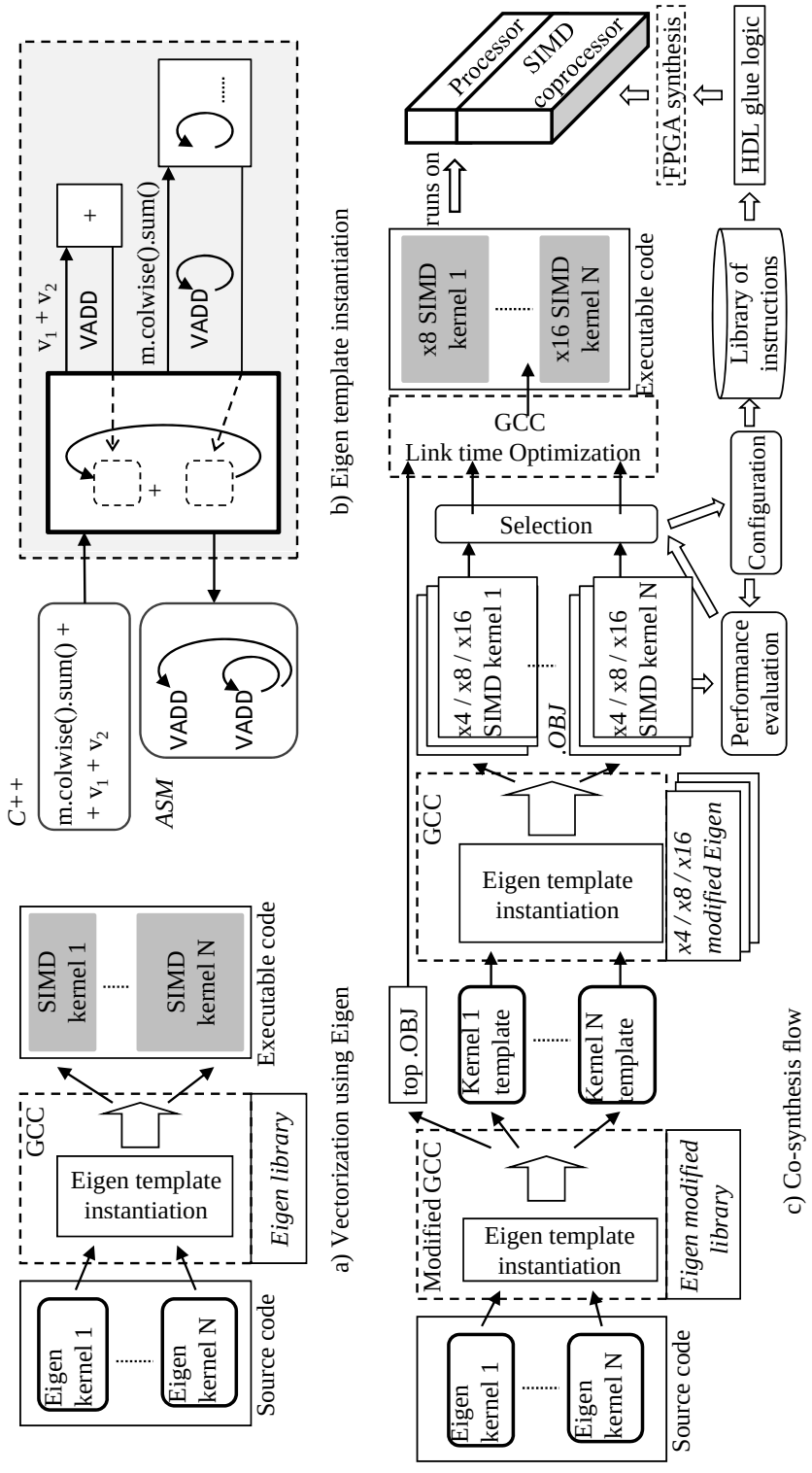


Figure 6.3: The code and coprocessor generation method.

the kernel. The compiler backend then optimises the resulting code. Recursive template instantiation and inlining are merely techniques for code rewriting. Therefore, despite the heavy use of template instantiation and inlining, the final executable does not suffer from code explosion.

Figure 6.3c shows how the library was modified to cease automatic instantiation at the level of kernels, such that the compilation of each template can be fine-tuned. Because kernels do not share code, each kernel can be compiled independently. Each of the kernels is passed to the compiler for explicit instantiation. The explicit instantiation steps are repeated to obtain all the different vector length versions of all the kernels. The performance of each kernel version is projected as a function of the parameters of the SIMD coprocessor configurations (Section 6.4). The list of instructions used by all versions of all the kernels is recorded. This information is used in a global selection step (Section 6.5) that determines the versions of each kernel to be used in the final executable, as well as the coprocessor configuration to be synthesized in order to execute the selected kernels. This is done in an integrated approach and does not require repeated hardware synthesis. The *link-time optimization* feature of GCC [55] is used to derive all versions of the kernels as well as inline the selected kernel versions for the final executable.

Eigen includes an ISA specific set of primitive template functions that corresponds to vector instructions and their GCC built-ins. Additional Eigen templates have been added for other vector lengths, and GCC was modified by adding new built-ins and machine descriptions. It is important to note that while choosing Eigen minimized the engineering effort, the same method can be adapted to any vectorizer that is capable of handling multiple vector lengths such as the GCC vectorizer.

6.3 Customizable SIMD Coprocessor Architecture

This section describes the details of the novel SIMD coprocessor architecture that can support the vector instructions generated through this method. The implementation targets the Xilinx Virtex-5 class of FPGAs and its hard-wired PPC440 processor core. This drove the compatibility requirement with the IBM AltiVec instruction set architecture [23]. However, with some amount of re-engineering, it should be possible to port this method to other similar architectures of which several alternatives are commercially available [1, 29].

The instructions for longer vector lengths are semantically simple extensions of the standard AltiVec instructions. Parallel vector operators are extended by increasing the range of their vector indices. Several of the data movement instructions that do not use operands with absolute element indexing can also be extended in the same manner. For the instructions where an index is given as an immediate operand (i.e. `VSLDOI`, a left shift instruction), or where such indices are included in one of the vector registers (i.e. `VPERM`, a generic permutation instruction that receives an index-based permutation pattern in a vector register) the bit-width limitations in the index encoding require special attention. When possible, previously unused bits in the operand field encode the larger index. Otherwise, the granularity of the indexed data was increased beyond the default, which is a byte. Supporting only vector FP operations, the granularity can increase four-fold without affecting the instruction semantics.

The custom SIMD coprocessor is implemented in the FPGA reconfigurable resources and is attached to the Auxiliary Processor Unit (APU) interface of the hard-wired PowerPC processor. The overall architecture of the coprocessor and its connections to the APU interface are presented in Figure 6.4. Scalar FP instructions use a legacy IP library module provided by Xilinx [97], attached in parallel to the same APU interface. The PowerPC core *issues* floating point instructions to the coprocessor via the APU interface. The SIMD coprocessor includes a collection of vector FP instruction *control blocks*, a unified vector register file and a set of scalar FP pipelined execution units. An instruction control block implements one or more related vector instructions. For example, a single control block implements both the vector add and subtract instructions. The execution units are shared by the vector instructions. Instructions and execution units are connected together with minimal glue logic. Clear design boundaries have been maintained between the execution units, instruction control blocks and the SIMD coprocessor interface, which allows modular synthesis to be employed.

Assuming that there are M distinct types of execution units (i.e., adders, multipliers, fused multiply-add units), a *configuration* E is defined as the tuple $(e_1, \ldots, e_M)$ where each e_i is the number of execution units of type EU_i to be

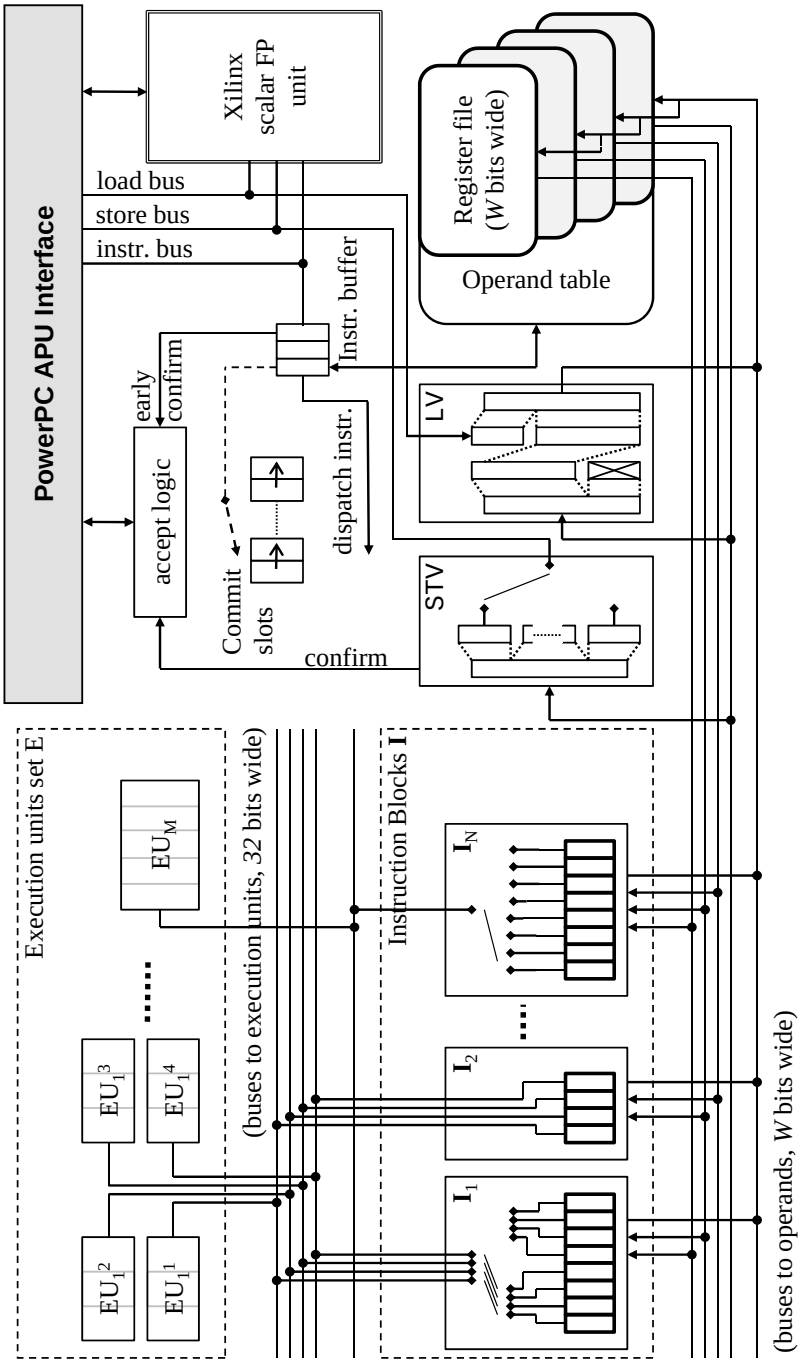


Figure 6.4: The architecture of the SIMD coprocessor.

instantiated. The coprocessor implements the instructions using N instruction control blocks $\mathbf{I} = \{\mathbf{I}_1, \ldots, \mathbf{I}_N\}$. Note that an instruction and its control block have the same vector length denoted by $\|\mathbf{I}_k\|$, that is implicitly encoded. The pair $(E, \mathbf{I})$ fully characterizes the SIMD coprocessor.

6.3.1 Instruction Handling

A single register file is used by all the vector instructions, irrespective of their vector lengths. It is configured such that it can store the longest supported vectors. Instructions handling shorter vectors will use only the lower bits of each register. The register file is implemented using the Virtex-5's block RAMs (BRAMs). To handle the longest vectors, it has to be $W = 32 \times \max_{\mathbf{I}_k \in \mathbf{I}}(\|\mathbf{I}_k\|)$ bits wide since each single precision floating point number occupies 32 bits. Altivec instructions can have up to three input and one output operands. However, the exact position of each in the instruction encoding varies. To avoid the overhead of multiplexing the possible positions to the register file, the register file has been implemented with four read ports and one write port. The amount of BRAM available in the Virtex-5 is large enough to implement four identical copies of the register file, each allowing one synchronous read and write. Read requests from different instruction operands will be serviced concurrently by the different copies of the register file. However, all the register file copies are written concurrently on any update. This ensures that all copies of the register file contain identical data, hence consistency is enforced.

A variant of scoreboarding is employed throughout the SIMD coprocessor. It manages instruction issue and retirement, enabling out-of-order completion. As soon as instructions appear on the APU interface, they are copied into an *instruction buffer*. A confirmation is immediately returned to the PowerPC core. The only exception is the vector store (`STV`) instruction, which needs to return data from a vector register to the PowerPC core. This instruction sends out a confirmation signal once it completes.

In the instruction buffer, a vector instruction will wait for its *dispatch* to the corresponding vector instruction control block $\mathbf{I}_k$. The instruction is dispatched only when all its operands are available in the register file, and when the hardware determines, based on the known execution time of $\mathbf{I}_k$, that the write port of the register file is available to commit the result during the clock cycle when the execution completes. These conditions together ensure that, once dispatched, instructions execute and commit their results without blocking. The moment

when instructions commit their results to the register file is tracked using a set of *commit slots* that correspond, in order, to reservations during future cycles to the write port of the register file. These commit slots are maintained consistent by shifting their contents to the previous slot at every clock cycle.

When an instruction is dispatched to a vector instruction control block, its operands are read from the register file. The control block will latch them and execute an internal schedule that accomplishes the desired functionality using the available execution units. If there are enough execution units, all operations can be launched in the same clock cycle (such as $\mathbf{I}_2$ in Figure 6.4). Otherwise, a *folding* mechanism, described below, schedules the operations on the available execution units over several clock cycles. Once the execution of the instruction completes, the result is placed on the write bus, and the destination register is marked as available in the operand availability table.

6.3.2 Folding of SIMD Operations

Folding is the mechanism used by the vector instruction control blocks to schedule the execution of vector operations on a smaller set of execution units. The implementation currently supports folding only if the vector lengths and the number of execution units are powers of two. The folding mechanism sequences the inputs for all operations to the execution units over several clock cycles. Because the execution units are pipelined, the coprocessor can launch internally a new set of operations each clock cycle, i.e., the *initiation interval* is one. In particular, for a vector instruction I executed by a control block $\mathbf{I}_j$, the number of consecutive cycles required to place all the operations in the e_k execution pipelines of type E_k is: $\text{fold}(I, E_k) = \|\mathbf{I}_j\|/e_k$ if the control block $\mathbf{I}_j$ requires the use of the execution units of type E_k. Otherwise, $\text{fold}(I, E_k) = 0$.

Folding requires hardware multiplexers to redirect data from several vector locations to the smaller number of execution units. These multiplexers are embedded in the instruction control block and driven by state machines. The instruction control block is aware of the number of execution units available in hardware. During instruction execution, after data is fetched from the registers,

a sequence of data insertions into the pipeline of the execution units is initiated.

Folding affects the rate at which the instruction control blocks can handle incoming instructions. If the operators are not folded, the entire instruction execution is fully pipelined, and a new instruction can be initiated every cycle. Otherwise, the instruction control block flags the execution units as busy, and this is an additional factor that may block the dispatch of the next instruction from the instruction buffer. The total execution time of an instruction $L(I)$ is also affected by folding. This latency consists of a fixed number of clock cycles spent in the instruction control block and in the execution pipeline, and a variable number of clock cycles required to fold the instruction. The latter depends on the vector length of the instruction and the number of execution units available for its execution.

The custom coprocessor design is *feasible* if the total resources (i.e., LUTs) occupied by all the instruction control blocks, execution units and other logic, including the scalar FP unit, fit the resources of the FPGA. Because the register file resides in BRAM, a significant number of additional LUTs was freed. Post-synthesis resource information is obtained for individual modules of the design, and together with the additional LUTs used by the folding multiplexers, it is used to derive the total resource requirement of a folded design. Besides resource constraints, the scalability of the design is limited by the critical path of multiplexing the results back to the register file's write port via a single result bus. Nonetheless, the place and route tools are successful with as many as 32 multiplexed write sources.

The execution units were designed from scratch, including the single-precision floating point adder, multiplier and fused multiply-adder. Their post-synthesis resource usage and performance are shown in Table 6.1. The multiply-add unit fuses the two operations without the intermediary result normalization, and hence is equivalent to its standard AltiVec counterpart. Note that this does not preclude the use of other arithmetic unit designs.

32-bit FP execution unit	Resources		Frequency (MHz)	Stages
	LUTs	DSP48E		
add	621	0	188	5
multiply	132	2	188	3
multiply-add	1101	2	152	7

Table 6.1: Characteristics of execution units.

6.3.3 Memory Access

Some of the design decisions were driven by the idiosyncrasies of the Virtex-5 development board [61]. Vector load (`LV`) and store (`STV`) instructions are handled in a special way so as to account for the fact that the memory APU bus width is a fixed 128 bits, regardless of the vector length. The processor core handles all memory accesses including those made by the coprocessor. The solution for dealing with this constraint is a special *load-shift semantics* for vector load operations: consecutive loads targeting the same vector register will shift the content of the register before incorporating the incoming data in the lower bit positions. If the instructions require a vector length of 128 bits, a single load is issued, and the corresponding data is placed in the lower 128 bits of the longer vector register, while the rest of the vector register, containing shifted data, becomes irrelevant. If instructions require longer vectors, multiple 128 bit loads are issued. Each of these loads will shift the previously loaded data to more significant positions. If loads are issued in the correct order, the long vector load can be replaced by a sequence of regular loads.

Unfortunately, vector stores cannot be handled transparently. Stores may be canceled and reissued by the PowerPC due to branch mispredictions or page faults. Consequently, there is no easy way to check if a previous store has succeeded or not that is compatible with the described coprocessor. The solution was to implement an explicit bank selection instruction that specifies which part of the longer vector needs to be stored via the 128 bit APU bus. The bank index is initially reset, thereby making this mechanism transparent to the x4 instructions.

6.4 Performance Projection Model

A key component of the method is the static evaluation of design points. This section describes the model by which the performance of each kernel is projected. Without such a model, it would be impossible to offer a method that selects the best possible hardware configuration without trial synthesis of many candidate configurations.

The SIMD coprocessor described in the previous section has two degrees of flexibility. It can adjust the number of execution units of each type. It can also choose whether or not to implement instructions of various vector lengths. Recall that while a kernel can only be of one vector length, different kernels in a single application are allowed to have different vector lengths. Estimating the performance of each kernel on a configuration E enables the usage of this metric to drive the instruction selection and implementation in Section 6.5.

The first step is to identify the sequence of vector instructions $I = \{I_1, \ldots, I_n\}$ for each loop body in the kernel k. For most practical situations, there is usually only one loop body in the kernel. These instructions will be issued by the PowerPC processor in program order to the SIMD coprocessor. The remaining scalar instructions execute out-of-order but have no impact on the execution time. Furthermore, the PowerPC processor is able to start prefetching the data for all the vector loads as soon as they are encountered. The PowerPC core maintains a look ahead window of δ instructions, prefetching additional instructions while it attempts to issue an instruction to the SIMD coprocessor. Memory accesses are modelled assuming that they will hit the cache and that a 128-bit load or store transaction takes d cycles, based on the processor memory bandwidth. The execution time $L(I_p)$ of instruction I_p is computed. Any folding is accounted for in $L(I_p)$ as previously described.

Based on the above assumptions, an estimate of the number of clock cycles $T(E, k)$ required to execute one iteration of a loop in kernel k on a configuration E is computed using Algorithm 6.1. For each instruction I_p, the following timings are derived relative to the beginning of the iteration: (a) the time when the instruction reaches the look-ahead window (α_p), (b) the time when it is issued

Algorithm 6.1 $T(E, k)$ for a single loop in a kernel

Input A configuration $E = (e_1, \ldots, e_M)$ and the sequence of vector instructions $I = \{I_1, \ldots, I_n\}$ that forms the loop body inside kernel k

$\Xi = F_1 = \ldots F_n = -\infty$
$\beta_{-\delta} = \ldots = \beta_{-1} = -\infty$ // hold the issue times from the previous iteration
repeat
  for $I_p \in I$ **do**
    $\alpha_p = \beta_{p-\delta} + 1$ // models PowerPC lookahead
    $t = \max(0, \beta_{p-1} + 1, \alpha_p, \max_{m \in \text{in}(I_p)} \gamma_m)$
    $\beta_p = \max_{E_i \text{ used by } I_p} (t, (F_i))$; // models blocking
    for functional unit type i used by I_p **do**
      $F_i = \beta_p + \text{fold}(I_p, E_i)$
    end for
    if I_p is memory transfer **then**
      $\Xi = \max(\Xi, \alpha_p) + d$
      $\beta_p = \max(\beta_p, \Xi)$
    end if
    $\gamma_p = \beta_p + L(I_p)$ // instruction ready time
  end for
  $\tau = \gamma_n$ // ready time of last instruction
  for $j < \delta$ **do**
    $\beta_{-j} = \beta_{n-1-j} - \tau$
  end for
  for E_i **do**
    $F_i = F_i - \tau$
  end for
  $\Xi = \Xi - \tau$
until τ does not increase
return τ

to the SIMD coprocessor (β_p), and (c) the time when it finishes execution (γ_p). Other elements tracked are the time when the memory bus becomes available (Ξ), and the time when execution units of each type i are available (F_i). The timing obtained at the end of an iteration is used to seed the computation of the next iteration. This is repeated until reaching a fixed point, then the result is returned. In the description, 'in(I_p)' are the predecessor instructions of I_p in the data dependency graph, and 'fold(I_p, E)' was defined in the previous section.

Let V be a set of vector lengths. In the current context, $V = \{4, 8, 16\}$. The different versions of the kernel k_x are denoted by the set $\{k_x^{v_i}\}$, $v_i \in V$. The Eigen library is modified to obtain the relative iteration counts of the kernel loops compiled for each of the vector lengths. If a kernel has more than one loop, then Algorithm 6.1 can be applied to each loop inside the kernel. A combined

per-iteration execution time for each kernel is derived by summing the per-iteration execution times of each loop weighed by their relative counts. The relative counts of different kernels are combined with actual profiling data from a scalar execution of the application to project the normalized weight $\omega_x^{v_i}$ of each vectorized kernel $k_x^{v_i}$. Profiling needs to be done only once for the non-vectorized application and can be accomplished with a regular GCC compiler and `gprof`.

The performance estimate $T(E, k)$ has to be recomputed for all kernels over all the configurations, as the latency of the instructions is influenced by the folding factor. Even though some of the estimations can be reused, the number of design points and combination of kernels is large. In this experimental setup, for example, there were 25 configurations, and 36 instruction candidates. The exploration space can be pruned based on the timing relationships between the configurations. Suppose there are two configurations, $E_1 = \{e_1^1, \ldots, e_M^1\}$ and $E_2 = \{e_1^2, \ldots, e_M^2\}$. For a configuration E_i, the minimum overall execution time $\mathcal{T}(E_i)$ is reached if the version selected for each kernel k_x has the lowest execution time. In other words, $\mathcal{T}(E_i) = \sum_x \min_{v_i \in V}(\omega_x^{v_i} \times T(E_i, k_x^{v_i}))$. However, this lower bound on execution time may not be achieved if some of the required instructions are not implemented due to resource constraints.

6.5 Configuration Selection and Code Generation

The selection of the best coprocessor configuration (shown in Algorithm 6.2) is based on statically projecting the performance achievable by the entire application on the feasible coprocessor configurations. As its input, it takes the independently vectorized kernel versions of the application. It also requires the set of possible vector lengths, the relative weights of the kernel calls in the call graph, and a resource constraint.

The output of the algorithm consists of a recommended configuration and the subset of the extended AltiVec instruction set to instantiate in the custom coprocessor. In particular, for the latter, suppose there are N distinct instructions control blocks in the set $\mathbf{I}$, and that the set of possible vector lengths is V. The output is a Boolean decision matrix $\Phi = \{\{\phi_1^{v_1}, \ldots, \phi_N^{v_1}\}, \ldots, \{\phi_1^{v_{|V|}}, \ldots, \phi_N^{v_{|V|}}\}\}$.

$\phi_j^{v_i} = 1$ if the control block of instruction j for vector length v_i is to be supported in hardware. The resource usage of this instruction is denoted by $a_j^{v_i}$.

The design space is explored according to a *dominance relationship*. Let $E_1 = (e_1^1, \ldots, e_M^1)$ and $E_2 = (e_1^2, \ldots, e_M^2)$ be two configurations. E_1 is dominated by E_2 (denoted as $E1 \prec E_2$) if $\exists i, e_i^1 < e_i^2$ and $\forall j, j \neq i, e_j^1 \leq e_j^2$. In other words, configuration E_1 has strictly less number of units of type i than E_2, and at most the same number of units as E_2 for all other types. In a dominated configuration, i.e. E_1 here, one may implement even more vector instruction control blocks using the difference in the resource of E_2 and E_1. Even so, $E_1 \prec E_2 \Rightarrow \mathcal{T}(E_1) \geq \mathcal{T}(E_2)$ regardless of what instructions are added to E_1. This means that if the current configuration is E, and $\mathcal{T}(E)$ is larger than the best found so far ($\hat{X}$ in Algorithm 2), then none of the configurations dominated by E can do better, and so can be discarded. A variant of this strategy has been described in Chapter 3.2.

Algorithm 6.2 starts analysing the largest configurations that use less resources than the given resource constraint, and are not dominated by any other configuration (line 1), one at a time. For each configuration E being explored, $\mathcal{T}(E)$ is computed by selecting the best version of each kernel without any resource constraints (line 5). If this unconstrained lower bound $\mathcal{T}(E)$ is no better than what has been already found, it is discarded together with all the configurations dominated by E, and the next candidate is analysed. Otherwise, E is a possible solution. A set of resource-based constraints (lines 9-11) is built, and a SAT solver is invoked to generate feasible solutions with the help of an evolutionary optimizer [68] with the goal of deriving a solution with the minimum execution time. The binary decision variable $s_j^{v_i}$ indicates whether kernel j vectorized with length v_i is part of the solution. $\phi_j^{v_i}$ indicates if instruction j with vector length v_i is to be part of the final coprocessor ISA. The constraint in line 10 is to ensure that if a particular vectorized kernel is chosen, then all the instructions used by that kernel are also chosen. The auxiliary function $R(E)$ estimates the resources used by configuration E. If the SAT solver is able to arrive at a solution X that is faster than the existing best solution ($\hat{X}$) (line 12), then

Algorithm 6.2 SIMD coprocessor configuration selection

Input A set of vector lengths (V), all kernels vectorized by the various vector lengths ($K = \{k_j^{v_i}\}, v_i \in V$), the relative weight of all kernels ($\{\omega_j^{v_i}\}, k_j^{v_i} \in K$), and a resource constraint (R_{FPGA})

Output A configuration (Π), and the subset of instructions to be implemented (Φ)

1: $SFU = \{E | \nexists E', E \prec E' \wedge R(E) \leq R_{FPGA}\}$
2: $\hat{X} = \infty; \Pi = \Phi = \emptyset$
3: **while** $SFU \neq \emptyset$ **do**
4: $E = \text{pop}(SFU)$
5: $\mathcal{T}(E) = \sum_j \min_{v_i \in V}(\omega_j^v \times T(E, k_j^{v_i}))$
6: **if** $\mathcal{T}(E) < \hat{X}$ **then**
7: $\Phi' = \{\{\phi_1^{v_1}, \ldots, \phi_N^{v_1}\}, \ldots, \{\phi_1^{v_{|V|}}, \ldots, \phi_N^{v_{|V|}}\}\}$
8: **SATslv minimize** $X = \sum_{j, v_i \in V} (T(E, k_j^{v_i}) \cdot \omega_j^{v_i} \cdot s_j^{v_i})$
subject to
9: $\sum_{j, v_i \in V} \phi_j^{v_i} \cdot a_j^{v_i} \leq R_{FPGA} - R(E)$
10: $\forall j \forall v_i \in V, s_j^{v_i} \leq \phi_x^{v_i}$ if instruction x of length v_i is needed in the implementation of kernel $k_j^{v_i}$
11: $\forall j, \sum_{v_i \in V} s_j^{v_i} = 1$
12: **end SATslv minimize**
13: **if** $X < \hat{X}$ **then**
14: $\Pi = E; \Phi = \Phi'; \hat{X} = X$
15: **end if**
16: **if** $X > \mathcal{T}(E)$ **then**
17: $SFU = SFU \cup \{E' | E' \prec E \wedge \nexists E'' \text{s.t.} E'' \prec E' \wedge E'' \prec E\}$
18: **end if**
19: **end if**
20: **end while**
return Π and Φ

the newly found solution replaces it (line 13). Furthermore, if X is worse than the unconstrained bound of $\mathcal{T}(E)$ (line 14), it would imply that there is room for improvement. All configurations immediately dominated by E are added to the list of configurations to be considered (line 15). The idea here is that in one of these (say E'), it may be possible to obtain an improved execution time by implementing additional instructions using the resource difference between E and E'.

The algorithm is guaranteed to terminate because (a) there is a limit on the number of iterations of the optimizer, and (b) only smaller configurations are added for future consideration. In practice, for the benchmarks reported in Section 6.6, it took no more than a minute on a Intel Core 2.

The solution returned by the algorithm drives the code generation step, which puts together a mix of Verilog and VHDL modules that is then pushed through the Xilinx synthesis flow to obtain the bitstream of the SIMD coprocessor. The solution is also used in the modified version of the GNU assembler which is utilized in conjunction with the modified GCC-LTO compiler to generate the executable.

6.6 Results

The resulting coprocessor configurations were implemented on a Xilinx ML510 system [61]. The Virtex-5 VFX130 FPGA on board includes a PowerPC 440 core, and 81,920 look-up tables (LUTs). All the experiments reported here are based on the HDL generated automatically with this method, which is implemented around the modified versions of Eigen and GCC. 18.5% of the LUTs were used for a system wrapper, the scalar FP unit from the Xilinx library and the SIMD coprocessor interface. A set of linear algebra benchmarks were selected, such that they would be ideal candidates for vectorization in embedded applications such as media processing [5], sensor array data processing, global positioning systems and beamforming solutions [57]. Several vector and matrix benchmarks are provided in Eigen. These benchmarks are compute intensive functions and reach performance comparable to BLAS on mainstream architectures. Furthermore, Eigen was used to vectorize benchmarks from the Iterative Template Library [56], a library which provides iterative methods for solving

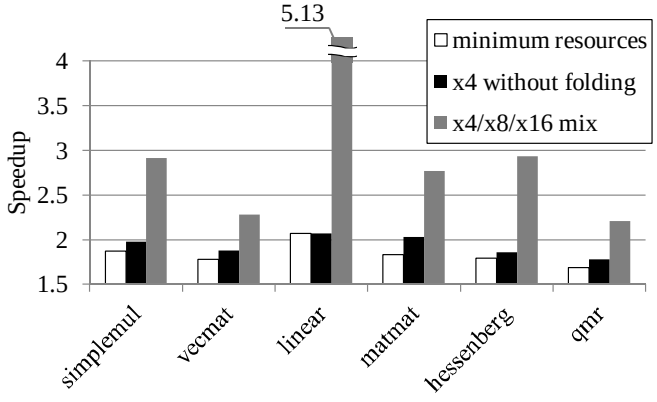


Figure 6.5: Speedup of different design points compared to scalar FP execution.

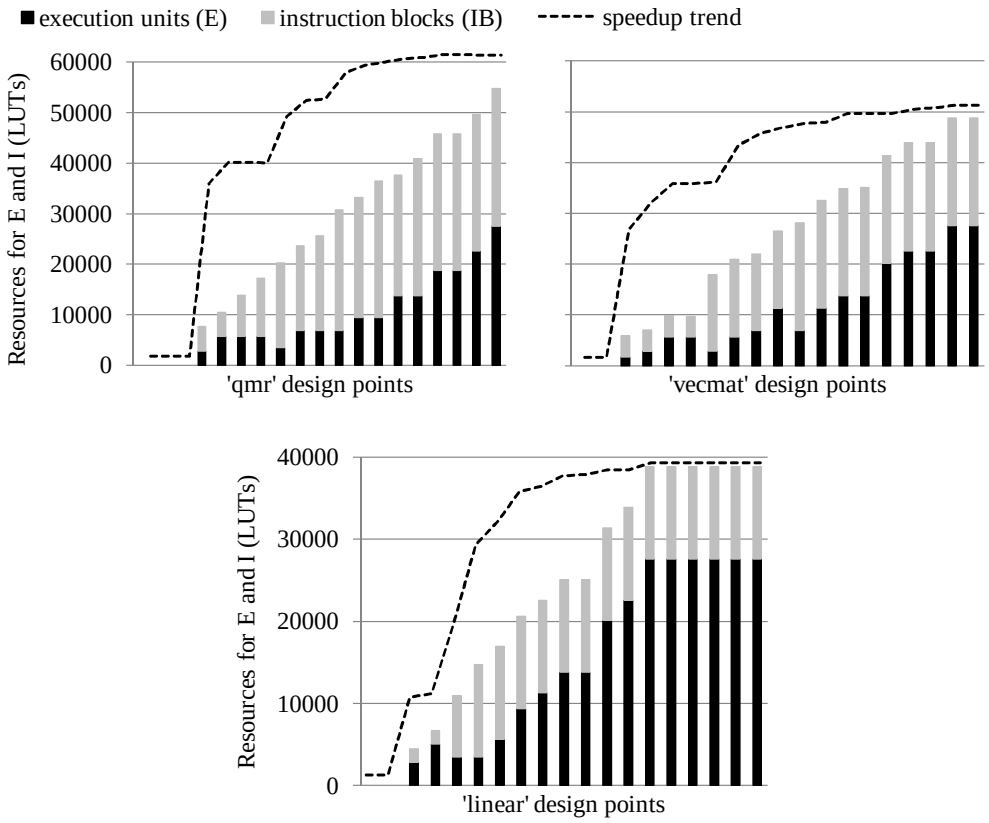


Figure 6.6: Resources used by execution units vs. instructions throughout the design space.

linear systems. An extensive range of design points was explored for the SIMD coprocessor, allocating up to 67% additional LUTs (leading to a total 85% utilization of the FPGA resources). For all these points, coprocessor configurations were automatically synthesized with the same frequency constraints. The core processor runs at 400 MHz while the coprocessor runs at 133 MHz.

Figure 6.5 shows the speedup achieved by the co-synthesized design compared to scalar FP execution using the Xilinx IP core. The performance of three design points is presented for each benchmark: (1) a design that utilizes a very low amount of resources, hence the execution units and instructions are constrained to 10% of the total LUTs, (2) a x4 design where the number of execution units matches the vector length (no folding, would correspond to a naïve implementation), and (3) a design using the optimized mix of x4/x8/x16 instructions, constrained only by the maximum available resources. The minor performance improvement observed between the former two designs, which use x4 instructions, supports the observation that, for short vectors, the bottleneck

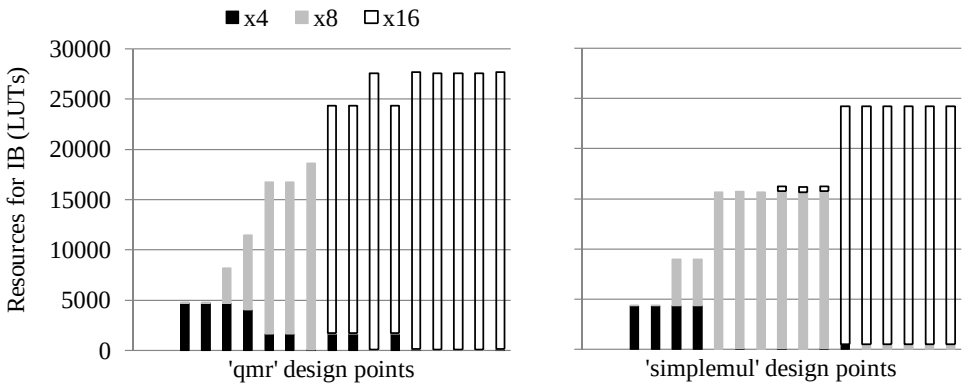


Figure 6.7: Distribution of resources among x4, x8 and x16 instructions for 'qmr'.

is at instruction issue, and merely adding execution units has a limited impact on performance. Instead, using longer vectors led to up to 5.13× improvement.

There is a non-trivial balance between the execution units configuration E, and the instructions control blocks implemented $\mathbf{I}$. The x16 instructions encapsulate larger multiplexers and use significantly more resources than the shorter vector versions, but deliver better performance. However, there are numerous cases where, due to resource constraints, the optimized solution involves using a smaller set of possibly shorter vector instructions complemented by a higher number of execution units. This validates the initial assumption that deferring the flexibility extracted from the input application is a method to select high performance solutions. Figure 6.6 presents the performance and resource utilization for three benchmarks, for a set of design points where the resource constraints are incrementally relaxed. This figure presents only the total amount of resources used for each of the two configurable portions of the design (E, $\mathbf{I}$). While performance increases monotonically, the non-trivial distribution of resources between the instructions and the execution units shows the necessity of the search algorithm described. In addition, Figure 6.7 gives the composition of the instructions implemented for 'qmr' at the given design points. It shows the resource usage associated with instructions of different vector lengths. The x4 instructions use significantly less resources than x16.

Besides speedup, using vector instructions leads to significant energy savings. The total energy of the fastest design is compared to that of a design using solely

	Benchmarks					
	simplemul	vecmat	linear	matmat	hessenberg	qmr
Data size	256	128	1024	128	128	1024
Best time(s)	2.45	3.11	2.42	1.45	2.22	14.98
Best energy(J)	7.5	9.1	7.6	5.7	8.2	42.2
Scalar energy(J)	18.4	18.3	34.1	13.9	14.3	83.4
Energy gain	41%	50%	22%	41%	57%	51%

Table 6.2: Execution time and energy.

the scalar FP Xilinx IP core. The result reflects the total energy consumption of the FPGA core, which includes the energy of the PowerPC hard-wired processor. The current sensor provided on the ML510 board is used to measure its average value during program execution using a multimeter. Table 6.2 reports the energy consumption derived from the measurements using the best mix of instructions. The energy consumption of the best mix can be as low as 22% of the original scalar version.

6.7 Summary

This chapter described the issues involved in the acceleration of floating-point computation on a reconfigurable platform attached to a hard-wired processor. Obtaining good performance for compute intensive applications on such coprocessors depends on a number of issues including the amount of instruction level parallelism available in the application, the structure of the loops, the processor cores' issue rate, memory bandwidth, and the reconfigurable resources available. Due to the intricate balances involved, a 'one size fits all' approach to vectorization is not always suitable. In fact, a mix of vector lengths may be required. Furthermore, this mix differs from application to application.

The described method represents the first fully automatic method that co-optimizes and co-synthesizes an application and its custom floating point SIMD coprocessor. The method determines the best vector length mix for each individual kernel in an integrated approach, requiring only hotspot profiling. The architecture described is able to take advantage of the flexible application parallelism and to share floating-point execution units, so as to meet a given resource constraint. The experiments showed that this method yielded up to 5.13× speedup

compared to the use of the standard Xilinx floating point IP cores and also showed significant energy reductions. This automatic code generation method provides efficient implementations of fine-grained applications on FPGA platforms.

CHAPTER 7

FINE-GRAINED CODE GENERATION FOR GPUS

Chapters 4 and 5 have shown how the coarse-grained task and data parallelism captured by StreamIt applications can lead to efficient code generation for GPU platforms. The GPU architecture also handles efficiently fine-grained parallel operations. However, the GPUs do not benefit from any hardware mechanism that can identify inter-thread data parallelism at run-time, hence all the available parallel code has to be identified during code generation and distributed to parallel threads.

The code generation method presented in this chapter is exemplified on an application arising in computational systems biology. This application includes a large system of differential equations that are solved repeatedly through numerical approximation. As expected, the intermediate results are too large to be stored in SM memory if each GPU thread contains one instance of the equation system. Therefore, following a direct implementation strategy will incur a severe performance penalty due to the significant number of global memory accesses. This is avoided through an optimized code generation method. This method manages the large amount of fine-grained parallelism exposed by the equations and reorganizes the computation in a platform-aware manner.

Specifically, a heterogeneous pool of threads is instantiated to distribute the computation corresponding to each instance of the system of equations over several threads. This increases the utilization of the GPU, while fitting the equation data in the SM memory. This method is an extension of the mix of $\mathcal{C}$ and $\mathcal{M}$ threads presented in Chapter 4. It achieves significant performance improvements on realistic equation systems and it scales well with problem size. While the code generation method is illustrated on a particular application, the implementation strategy is generic and is applicable to similar problems, in which fine-grained parallelism is accompanied by a significant amount of read sharing.

7.1 Rationale

This chapter describes how the code generation method applies to a model approximating a biochemical network. A network of biochemical reactions [49] can be modeled as a system of ordinary differential equations (ODE). The equations describe the biochemical reactions, while the variables represent the concentration levels of the molecular species. Typical networks will involve a large number of species and reactions, and the corresponding systems of ODE will not admit closed-form solutions. The initial states will often be available only as intervals of values. Consequently, the ODE-based model is simulated numerically, and Monte Carlo methods are used to ensure that sufficiently many point values are sampled. As a result, basic tasks such as parameter estimation, model validation and sensitivity analysis will require the repeated generation of a large number of numerical simulations.

By sampling the prior distribution of initial states, a sufficiently large representative set of the trajectories induced by the ODE dynamics can be computed numerically on the GPU. The dependencies in the pathway structure are exploited to encode compactly these trajectories as a time-variant dynamic Bayesian network [62]. This dynamic Bayesian network (DBN) is viewed as an approximation of the ODE dynamics, and analysis tasks are performed on this simpler model using standard Bayesian inference techniques [50, 54].

The DBN approximation of the ODE model requires the one-time generation of a large number of trajectories. Clearly, sampling initial states and generating trajectories through numerical integration is a potential source of massive parallelism. However, for each trajectory, equations share variables, hence the exposed parallelism inside each trajectory is fine-grained. In addition, the threads generating the trajectories will have to record intermediate data. For large problems, the total amount of variables and intermediate data required by all the trajectories, when each trajectory computation is associated with a single GPU thread, may be too large to be stored in the registers or the SM memory. At the same time, relying solely on the global memory for this purpose will incur a severe performance penalty as it will lead to a vicious cycle in which more and more

parallel threads will have to be launched to hide the memory latency, which in turn will generate increased access to the global memory.

The solution is to create a *heterogeneous* pool of threads that uses the SM memory and can support the fine-grained parallelism in the application. The number of parallel trajectories is reduced to fit the SM memory. Then, the computation for each parallel trajectory is distributed among several threads to increase GPU occupancy. A fine-grained distribution encompassing single equations is feasible, due to the low SM synchronization cost. The corresponding threads from parallel trajectories are grouped in the same warps so as to maximize the utilization of the processing cores. All the threads corresponding to a trajectory share the same data. Of particular significance is that this solution can be applied to any problem that exposes fine-grained parallelism.

7.2 Application Description

A recent DBN construction approach [54] is utilized. The value domains of the variables and unknown parameters are discretized into finite sets of intervals. The states of the system are observed only at a finite number of time points, $\{0, 1, \ldots, T\}$. Once the DBN approximation has been constructed by computing trajectories from all the initial states, analysis can be performed using Bayesian inference techniques [62].

Biological pathways are often described as a system of ODE with one equation of the form $\frac{dx}{dt} = f(\mathbf{x}, \mathbf{p})$ for each molecular species $x \in \mathbf{x}$, with f describing the kinetics of the reactions that produce and consume x, while $\mathbf{p}$ denoting the parameters associated with these reactions. For large pathways, this system of ODE will not admit a closed-form solution. The range of each variable x_i (parameter p_j) is partitioned into a set of intervals. The initial values of the variables, as well as the parameters, are assumed to be distributions (usually uniform) over certain intervals. The initial states are sampled sufficiently many times [54], and a trajectory is obtained by numerical integration from each sampled initial state. The resulting set of trajectories is then treated as an approximation of the dynamics of the ODE system. A uniform distribution is fixed over *all* the intervals.

The time domain is discretized into a finite set of time points as $\{0, 1, \ldots, T\}$. For convenience, this discretization is assumed to be uniform. Consequently, a time step $\Delta t > 0$ is fixed, and the time points of interest are assumed to be the set $\{0, \Delta t, 2 \cdot \Delta t, \ldots, D \cdot \Delta t\}$ so that $T = D \cdot \Delta t$.

A Mersenne twister random number generator [60] is used to sample an initial state viewed as a vector of values; one value for each variable and unknown parameter. Starting from the initial vector, the generation of a single trajectory iteratively advances time by Δt and computes, through numerical integration, updated values for all the variables. To ensure numerical accuracy, the interval $[0, \Delta t]$ is uniformly subdivided into k intervals for a suitable choice of k. After every k time steps (each of length $\frac{\Delta t}{k}$), the current values of the variables are recorded. A fourth order Runge Kutta integration algorithm is used to compute the next value of a variable for each time step. Thus, each trajectory is numerically simulated for $k \cdot D$ steps.

Finally, the current values of the variables at each of the time points $\{0, \Delta t, 2 \cdot \Delta t, \ldots, D \cdot \Delta t\}$ are used to count how many of the trajectories hit a particular interval of values for each variable at that time point. These counts are then used to derive the entries in a Conditional Probability Table (CPT) [54] from which the DBN is derived.

7.3 Code Generation Method

This section describes the method developed to generate efficient code for the application described above onto GPU platforms. The computationally intensive task of generating a large number of trajectories can, in principle, utilize a single thread for each trajectory, since the control flow of the underlying computations is similar among different trajectories. This matches the GPU's affinity for lockstep execution in warps. However, the memory requirements of the computations generating the trajectories pose a severe barrier to obtaining an efficient memory layout. This is due to the fact that the dependencies between the variables in the system of ODE require the entire front of variables belonging to each trajectory to be computed together. Specifically, each variable x_i has an

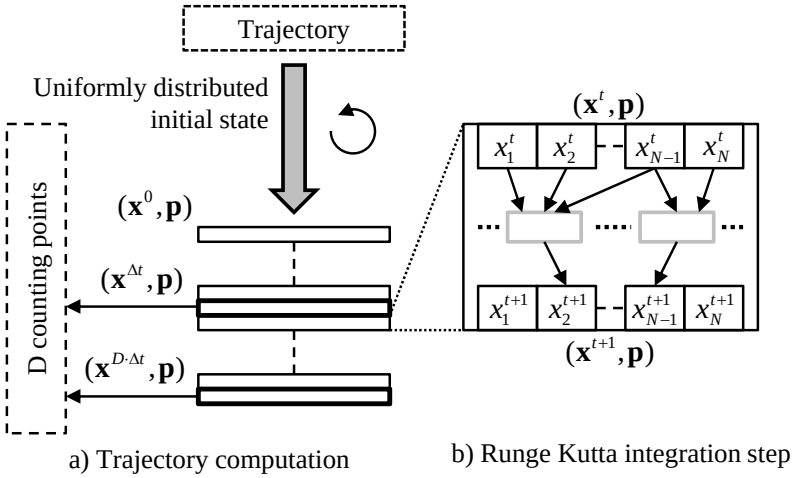


Figure 7.1: Computation flow.

associated equation in the ODE system. For each time interval, the value of x_i^{t+1} is determined by applying a Runge Kutta numerical integration step using the current value of x_i^t and the current values of other variables (and parameters) appearing in the ODE for $x_{j_i}^t$ (Figure 7.1).

By exploiting the fine-grained parallelism available in the ODE, a group of threads can compute together each trajectory. Each thread in the group will perform numerical integration on some of the variables. This entails sharing of the variables and parameters within a group. Frequent synchronization is required in fine-grained code sections involving only a single Runge Kutta time step from as little as a single equation. Variables and parameters must be stored in SM memory to allow consistent low-overhead multi-threaded access. More important, the parallel collaborating threads will need to execute with divergent control flow because each thread will handle a different subset of the model equations.

The key to an efficient implementation, under these conditions, lies in clustering threads with similar computation into the same warp so that they can be executed under the lockstep constraint. Therefore, this method goes one step beyond the concept of $\mathcal{C}$ threads introduced in Section 4.2, which defines a large number of similar compute threads. Instead, a heterogeneous pool of compute threads is instantiated here to enable the sharing of data between many concurrent threads. The similar threads computing the same equations from parallel

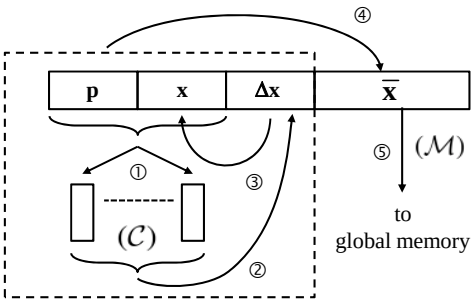


Figure 7.2: Data movement during trajectory generation and counting steps.

trajectories will be clustered in the same warp.

The data movement during one computation step is illustrated in Figure 7.2. The equations are distributed into several $\mathcal{C}$ threads (C_θ) that will collaborate to generate a single trajectory. These threads read the current values of the variables ($\mathbf{x}$) and parameters ($\mathbf{p}$) from memory (step ①). The $\Delta\mathbf{x}$ changes during a time step are computed in parallel and are stored back to memory (step ②). The vector of variables $\mathbf{x}$ is then updated (step ③). This process is applied iteratively for each time step (of duration $\frac{\Delta t}{k}$).

Further, to enable the counting steps which record the number of times the threads hit the various intervals of values of the variables (and unknown parameters), the vector $\mathbf{x}$ is replicated as $\bar{\mathbf{x}}$ (step ④). The counting process executes in parallel, during the next Δt iteration, using memory access threads ($\mathcal{M}$), which will store the results in a large table located in global memory (step ⑤). This ensures that the numerical integration can continue, while the counting process accesses the long latency memory.

The vectors $\mathbf{x}$, $\mathbf{p}$, $\Delta\mathbf{x}$, and $\bar{\mathbf{x}}$ together form the working set of a trajectory because they are accessed frequently. During the generation of a trajectory, variables will be updated once every integration step. In between these updates, the variables may be read by several equations. Because the computation of a trajectory has been divided onto multiple threads and because the number of concurrent trajectories has been restricted, the total memory footprint, consisting of the working sets of all the trajectories being computed in a SM, can be kept within the limit of the available SM memory. In addition, the working sets are carefully placed in the SM memory so as to prevent bank conflicts. Their

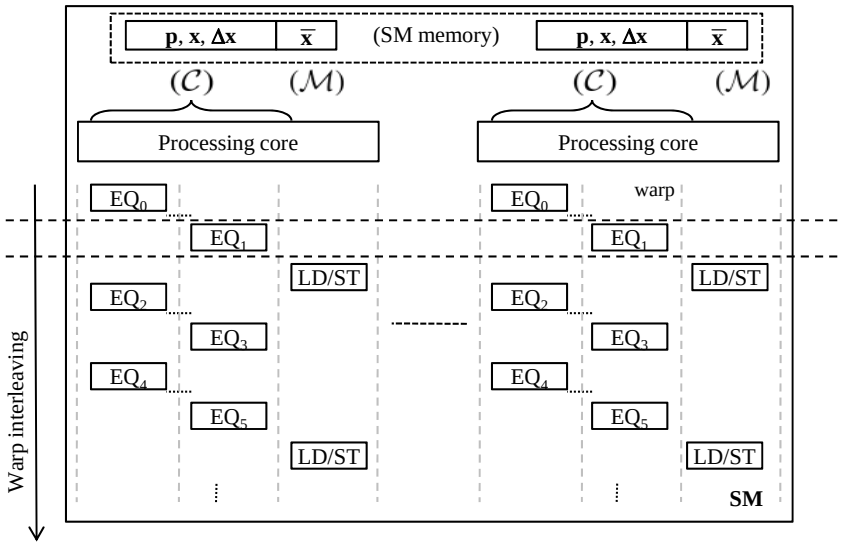


Figure 7.3: Concurrent execution of trajectories inside an SM.

relative offsets are adjusted so that the hardware will coalesce the accesses to the SM memory.

The number of trajectories resident in each SM is chosen such that it is congruent with the warp size. The threads are organised so that threads executed together in the same warp process the *same* subset of model equations from *different* trajectories. This will eliminate control flow divergence in each warp. In other words, threads computing different equation groups are assigned to different warps as shown in Figure 7.3. It is important to underline that this method can be applied for fine-grained parallel computation in other applications.

The pursued global memory access strategy requires one additional type of threads for global memory access. In this context, the $\mathcal{M}$ threads introduced in Section 4.2 become one type of threads, besides the several types of $\mathcal{C}$ threads. Threads $\mathcal{C}$ perform the numerical integration using the variables and parameters stored in the SM memory, while global memory access threads $\mathcal{M}$ perform the counting. In addition, $\mathcal{M}$ threads are coalesced into exclusive warps which depend on, but not interfere with, the $\mathcal{C}$ threads' executions.

The Mersenne twister algorithm also requires access to a large table stored in global memory. As this is invoked only once for each trajectory, during the

initialization step, the overhead of storing this table in global memory is minimal. To ensure randomness across the threads, each Mersenne instance table is initialized using a different seed.

The Runge Kutta numerical integration process is at the heart of the trajectory computation. A fourth order Runge Kutta algorithm was sufficient, and it requires each equation to be applied 4 times. The code generation method employed converts each equation into a set of memory loads and stores and a set of arithmetic operations.

The number of issue cycles for each arithmetic operation is statically known, and this information can be utilised to balance the workload of the computing threads. To achieve a good balance, a timing model was created to compute the weight of each equation. This model assigns a weight to each operator that appears in the code corresponding to each equation. Then, a distribution algorithm places the equations such that the corresponding weight of the operators appearing among the $\mathcal{C}$ threads is balanced. The granularity of this distribution is fine-grained, each equation including only a few arithmetic instructions. Because the hardware interleaves the warps on to the processing cores, in order to hide the operations latency, as long as all warps are balanced, they can issue new operations into the pipeline and the processing units are fully utilized.

The parallel nature of the trajectories also exposes a coarse level of parallelism which makes this problem suitable for multi-GPU code generation. While this aspect is beyond the problem investigated in this thesis, exploiting it extends the performance of this biopathway modelling implementation. Figure 7.4 shows how this method generates code for an array of GPUs. The GPU platforms may be connected to several hosts. Those connected to the same host communicate through dedicated high-bandwidth PCI Express connectors, while the hosts are connected to each other through standard Ethernet.

The computation is launched by a master host, and each GPU processes a fraction of the trajectories. Different random number seeds are broadcast to each of the GPU (step ①), such that no two seeds overlap. Each GPU simulates independently its portion of the trajectories and records the results in a table

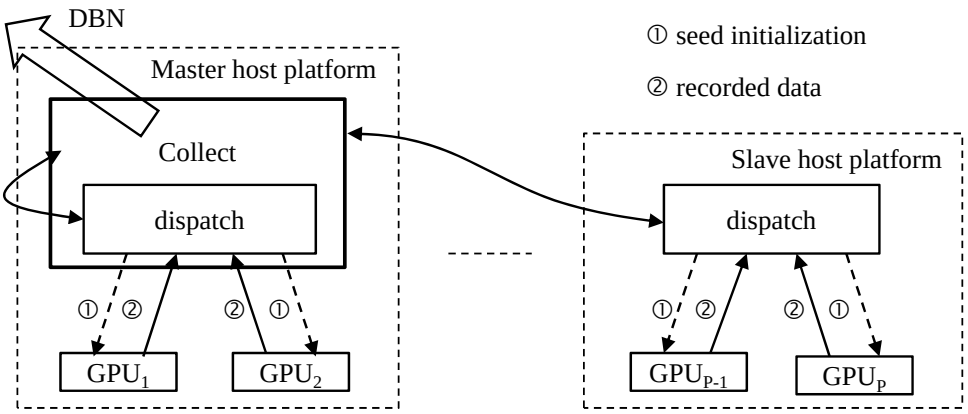


Figure 7.4: Distributed execution among multiple GPUs.

in its own global memory. Once the computation has finished, these tables are transferred over the network to the master host (step ②) that combines them to form the CPTs.

For large models, the memory required to store all the counting data might exceed the size of the GPU's global memory. To get around this, each GPU device records only the memory offset of each recording point ($\mathcal{M}$ threads process only the pointer to the location in the CPT). Later, at the master host, all the data based on the offset information collected from the GPU devices is updated. In essence, the results are encoded as sparse matrices and utilise the master host to combine the results into a regular matrix.

7.4 Results

This code generation method has been tested on the S2050 GPU platforms. The performance obtained on a cluster of 10 CPU is compared to that of 4 GPU platforms (attached to two CPU hosts; 2 GPUs per host). The 10 CPU cluster consists of Xeon E5430 @ 2.66 GHz with 40 GB of memory each. Each GPU is a S2050 Tesla @ 1.15 GHz with 2 GB of memory. Each two of them are attached to a Xeon E5405 @ 2GHz host with 16 GB of memory through PCI-E. nVidia GPUs support both single precision and double precision floating point types. However, the computational throughput for double precision is known to be less than half of single precision, with significant overhead for divisions. For this application, single precision computation suffices.

Model	Description	$\|\mathbf{x}\|$	$\|\mathbf{p}\|$
EGF-NGF	PC12 cells proliferation model [11]. 20 parameters assumed unknown and discretized into 5 intervals [54]	32	48
Segmentation Clock	Signaling network for the formation of somites in embryos [28]. 40 parameters assumed unknown and discretized into 5 intervals [54]	22	75
Thrombin-MLC	Models how thrombin can induce MLC phosphrylation in endothelia cells through two different signaling cascades [59]. 164 parameters assumed unknown and discretized in 5 intervals	105	197

Table 7.1: Biopathway models.

The three pathway models in Table 7.4 were used to evaluate the performance gains of the GPU-based implementation. Although the parameter values for these models are known, a subset of the parameters was set as 'unknown' to mimic realistic biopathways models, and the DBN approximation was constructed accordingly. This considerably increases the computational demands placed on the approximation algorithm.

The performance of the generated code varies based on the:

- number of $\mathcal{C}$ threads generated in each SM;
- C_θ, the number of $\mathcal{C}$ threads collaborating to generate a trajectory;
- M_θ, the number of $\mathcal{M}$ threads that are used to increment the global memory counts corresponding to a hit of an interval by a trajectory.

The maximum number of trajectories is constrained by the amount of SM memory available. The working set size $3|\mathbf{x}| + |\mathbf{p}|$ can be calculated irrespective of C_θ, and its size is rounded up to the next odd number only to ensure memory access coalescing (see Section 4.2.1) [67].

The number of threads C_θ determines the number of warps. If too few warps are instantiated, the processing cores cannot be fully utilized. On the other hand, if there are too many warps, there is additional overhead due to the limited number of registers that can be assigned to each thread. In addition, the equations may not be perfectly balanced among a large number of warps.

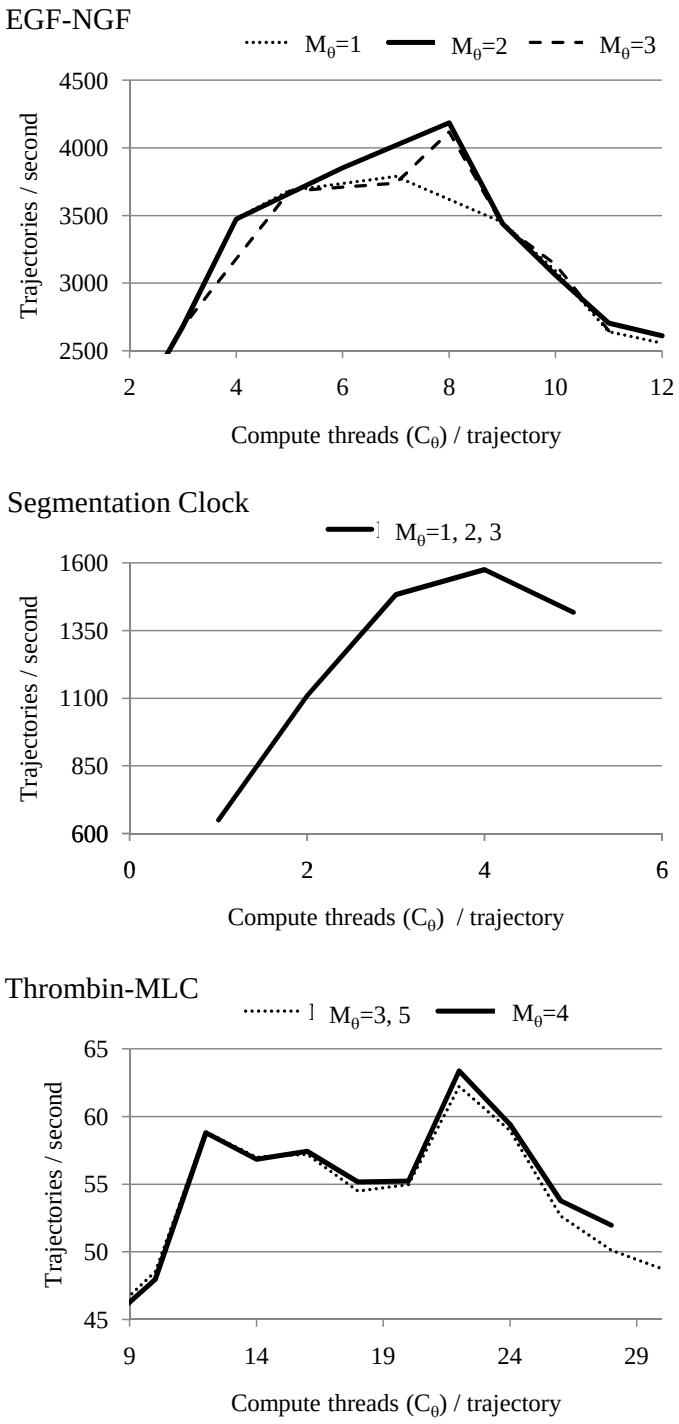


Figure 7.5: Design space exploration on the S2050 GPU.

Therefore, the value of C_θ has a significant impact on performance. Its influence is characterized in Figure 7.5. The performance of the three pathway models varies as the number of $\mathcal{C}$ threads per trajectory is modified. As expected, over-estimating M_θ does not have a significant impact on the overall performance since the associated memory accesses are seldom scheduled in the execution pipeline.

Pathway model	Model scale			Cluster run-time(s)		
	Trajectories	Steps	$\|\mathbf{x}\|$	CPU	GPU	Speedup
EGF-NGF	3×10^6	10^4	32	4985	191.4	26×
Segm. clock	3×10^6	5×10^4	16	17881	543.6	32.9×
Thrombin-MLC	3×10^4	2×10^6	105	132926	2044	65×

Table 7.2: Comparative performance of a cluster of CPU to multiple GPUs.

Pathway model	Trajectories	Run-time (s)		
		Naïve	Fine-grained	Speedup
EGF-NGF	10^5	23.82	23.33	2.0%
Segmentation clock	10^5	90.49	66.06	36.9%
Thrombin-MLC	10^5	22,512.00	15,749.00	42.9%

Table 7.3: Performance of the fine-grained method, compared to a naïve GPU implementation, for trajectories generated on a single GPU.

The number of trajectories was chosen so that the resulting DBN approximation was of sufficiently good quality. The performance of Thrombin-MLC on the CPU cluster was so slow that a smaller number of trajectories was utilised. The overall results are shown in Table 7.2.

The benefit of this code generation method, which uses the fine-grained parallel distribution of the computation of each trajectory, is highlighted by comparing it with an alternative implementation which naïvely exploits the parallel nature of the trajectories, but does not attempt to minimize the working set by sharing it over several threads. Th naïve version follows the conventional wisdom that a large number of threads are to be made available to the GPU, and their scheduling is to be handled automatically by the hardware. To achieve the required large number of threads, the working set for all trajectories had to be placed in the global memory. The performance of the naïve version is compared with the results of this method in Table 7.3.

The S2050 GPUs allows the execution of up to 2 interleaved warps without stalls, as long as all the operations are issued in two cycles. However, some of the operations can require more issue cycles (i.e. division). For these reasons, the optimal choice of C_θ and M_θ is not straight forward and is obtained through profiling. The configurations leading to the design points with the best performance in the experiments are presented in Table 7.4.

Pathway model	Parallel trajectories	C_θ	M_θ
EGF-NGF	64	8	2
Segmentation clock	128	4	1
Thrombin-MLC	16	22	4

Table 7.4: Optimized SM configuration for the presented models.

7.5 Summary

This chapter suggests an automatic method to realize fine-grained parallel applications on GPU platforms. In particular, this chapter has described how the instruction level parallelism available in a high dimensional system of differential equations can be exploited on the GPU.

The main challenge was to reorganise the computation in order to utilize the significant amount of fine-grained parallelism that exists between the model equations, in addition to the coarse-grained parallelism which exists between computed trajectories. A naïve GPU implementation, where parallel trajectories are assigned to parallel threads, does not scale well for large biopathways models. The chief contribution of this chapter is a novel code generation method, in which load balancing of heterogeneous threads via a static timing model delivered significant performance. This method works well for models of large biochemical networks.

This novel GPU implementation is compared both against a conventional GPU implementation as well as a CPU implementation. As indicated by the results, even cluster based implementations begin to consume unreasonable amounts of resources as the models get larger and more complex. The code generation method takes advantage of the fine-grained parallelism available in the model equations and produces CUDA code that is optimized for the architecture at hand. Using 4 GPUs, up to 65× speedup is achieved compared to a 10 CPU core implementation, with a roughly 43% improvement over a conventional, naïve GPU implementation.

While this chapter uses systems biology as the driving application, the key contribution is the principle by which compute tasks should be distributed among a set of heterogeneous GPU threads given the resource and execution constraints

of the device. By segregating computation and global memory accesses, a lower number of threads is sufficient to fill the GPU pipelines. This also leads to a smaller local memory footprint. This method is effective for other fine-grained parallel applications, such as many body simulations, that have significant read sharing.

CHAPTER 8

CONCLUSIONS

This thesis described how to generate efficient code through methods that integrate mapping, code rewriting, optimizations and compilation. Information from the compilation step augments the mapping decisions, such that the code structure can be reorganized to match the platform constraints. Therefore, a mapping solution is not committed before the platform-specific performance of its components is determined. These code generation methods ensured the feasibility and efficiency of the generated solutions.

Efficient code can be generated for both FPGA and GPU platforms, despite the complex interactions between their various resources. The methods described cover a broad spectrum of parallelism exposed by the applications. First, the methods showed how to use coarse-grained parallelism exposed by StreamIt applications, and how to use reorganization opportunities exposed by the application representation. On FPGAs (Chapter 3), replication and folding of pre-synthesized filters was at the base of the code generation algorithm. On GPUs (Chapters 4 and 5), multiple instances of several graph partitions can be executed in parallel. In addition, parallelism between filter iterations allowed splitting each partition's working set among multiple GPU threads.

Second, the methods were applied to applications that expose fine-grained parallelism. Chapter 6 showed how to select a feasible subset of vector instructions that match the available parallelism, and how to fold the floating-point operations included in the instructions onto a smaller set of execution units, using the regular structure of the vector operations. Chapter 7 showed that parallel computation, comprising of floating-point operations, could be split with little overhead among a set of threads.

A distinguishing feature of the methods described is the wide range (in terms of application size) of accepted input applications. The code generation methods

take into account how the application was transformed by the platform-specific compilation tools, and adjust accordingly the mapping and optimization decisions (i.e. adjust the amount of folding on FPGAs, or the split factor of the working set on the GPUs). Hence, the integrated code generation was successful, irrespective of the input application size, up to platform specific limits. Obviously, performance of larger applications could benefit to a smaller degree from parallel optimizations because they increased resource usage.

Consequently, for implementations which could produce a feasible solution, one was generated, considering the resource constraints. This is in contrast with regular methods, which generate solutions based on annotations. The described code generation methods did not involve user annotations as they extracted all the necessary information from the application structure. The methods validated the hypothesis that providing flexibility in the application structure, and postponing the mapping of this structure until it can be matched with the platform, in an integrated code generation step, can improve accuracy, without resorting to annotations. The code generation methods were built around existing platform compilers and could utilise readily available resource estimations. The flexible structure exposed by the application code was essential, as it described valuable information that the user was willing to provide, but which could not be extracted automatically.

The GPU code generation methods provided optimized solutions with polynomial complexity. On FPGAs, however, only throughput-optimized designs could be obtained with equivalent complexity. The latency improvement component of the FPGA methods finely tuned the initial solution through a meta-heuristic step, which is heavily pruned and converged fast to a solution for most practical cases. This is a small price to pay, given the FPGA synthesis time.

The code generation for the two platforms augmented complementary goals. The FPGA platform favors the realisation of low-latency designs, at a penalty for throughput, which is offset by the FPGA operating frequency. The ability to reduce latency justified the introduction of latency constraints for the code generation from StreamIt, and it decreased the overhead introduced by the custom

SIMD coprocessor.

The GPU platform favors the realisation of high-throughput designs, but performs poorly in terms of latency. It requires coarse packets of data to be accumulated before computation can start. The host CPU to GPU memory transfer also adds to the total latency. All these costs could only be offset by a large number of parallel stream executions, hence data had to be delayed and accumulated in channels before being processed.

StreamIt benchmarks were accelerated up to 6.3× on FPGAs and 3.8× on a GPU, compared to mapping methods that used directly the application structure. Fine-grained methods also yielded up to 5.1× speedup for floating-point instructions on FPGAs and 1.4× improvement on GPUs, compared to regular implementations.

8.1 Future Work

Each of the directions investigated described a complementary code generation path. Besides incremental improvements, it is possible to generate code targeting a heterogeneous platform which combines FPGAs and GPUs. This direction is open to exploration, as applications that would benefit from the combined performance of both platforms, as well as justify the communication overhead, have yet to be identified.

8.1.1 FPGA

The replication / folding mechanism allows pre-synthesized blocks of computation to be easily attached / disconnected. The vector instructions in Chapter 6 have a feature that allows them to drive a run-time defined number of folded execution units. This leads to slightly larger (generic) multiplexers inside the vector instruction control block. However, this paves the path for the utilization of partial reconfiguration techniques. The algorithm which determines the number of floating-point execution units of each type can be replaced with an FPGPA-based run-time counterpart that swaps in and out execution units based on application needs. Only the execution units are reconfigured, while the rest of the circuit is not modified. This would enable applications with time dependent

workload to benefit from localized, or function specific, throughput improvements, as the vector instructions can change their latency based on the modified number of available execution units.

A different FPGA-specific improvement may address the FIFO channels between replicated filters. Enhanced communication channels between filters would allow direct connection of filters running across different frequency domains. Such an improvement would allow automatically generated filters to be combined without penalty due to any mismatch in their running frequencies. The introduction of several clock domains would not alter the space unrolling algorithms presented.

8.1.2 GPU

The current GPU code generation methods address nVidia platforms. A discrepancy observed while porting for AMD GPUs suggests that some changes are required for optimized mapping on these GPUs. The $\mathcal{C}$ and $\mathcal{M}$ threads were associated with different wave-fronts (the AMD equivalent of warps) on an ATI HD5870 GPU board. However, while the speedups achieved were linear, the experiments showed that using $\mathcal{M}$ threads incurred an unexpected 30% of overhead. While AMD documentation does not fully disclose the wavefront scheduling algorithm, it is mentioned that a pair of wave-fronts hides all the ALU execution latency [3]. Therefore, one possible explanation is that if this pair of wavefronts contains only $\mathcal{C}$ threads, the scheduler disadvantages $\mathcal{M}$ threads, and they are unable to load the data in time. The $\mathcal{M}$ threads became a liability instead.

In addition, the performance of the code generation method in Chapter 4 is tightly dependent of the working set layout. The number of stream iterations that can be run in parallel depends on their individual working set size. Therefore, performance may also be improved by the introduction of a working set layout algorithm that performs better than the current heuristic.

BIBLIOGRAPHY

[1] "Nios 2 Processor," 2011. http://www.altera.com/literature/lit-nio2.jsp.

[2] Amdahl, G. M., "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), (New York, NY, USA), pp. 483–485, ACM, 1967.

[3] "ATI stream computing programming guide," 2010. http://developer.amd.com/gpu_assets/ATI_Stream_SDK_OpenCL_Programming_Guide.pdf.

[4] Baghsorkhi, S. S., Delahaye, M., Patel, S. J., Gropp, W. D., and Hwu, W.-m. W., "An adaptive performance modeling tool for GPU architectures," in *The 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '10)*, 2010.

[5] Barrio, V. M. D., Gonzlez, C., Roc, J., Fernndez, A., and Espasa, R., "A Single (Unified) Shader GPU Microarchitecture for Embedded Systems," in *High Performance Embedded Architectures and Compilers*, pp. 286–301, 2005.

[6] Battacharyya, S. S., Lee, E. A., and Murthy, P. K., *Software synthesis from dataflow graphs*. Norwell, MA, USA: Kluwer Academic Publishers, 1996.

[7] Bertels, K., Sima, V.-M., Yankova, Y., Kuzmanov, G., Luk, W., Coutinho, G., Ferrandi, F., Pilato, C., Lattuada, M., Sciuto, D., and Michelotti, A., "HArtes: Hardware-Software Codesign for Heterogeneous Multicore Platforms," *IEEE Micro*, vol. 30, pp. 88–97, September 2010.

[8] Bodin, F. and Bihan, S., "Heterogeneous multicore parallel programming for graphics processing units," *Sci. Program.*, vol. 17, no. 4, pp. 325–336, 2009.

[9] Boussinot, F. and de Simone, R., "The ESTEREL language," *Proceedings of the IEEE*, vol. 79, pp. 1293 –1304, sep 1991.

[10] Brisk, P., Kaplan, A., and Sarrafzadeh, M., "Area-efficient instruction set synthesis for reconfigurable system-on-chip designs," in *Proceedings of the 41st annual Design Automation Conference*, DAC '04, (New York, NY, USA), pp. 395–400, ACM, 2004.

[11] Brown, K. S., Hill, C. C., Calero, G. A., Lee, K. H., Sethna, J. P., and Cerione, R. A., "The statistical mechanics of complex signaling networks: nerve growth factor signaling," *Physical Biology*, vol. 1, pp. 184–195, 2004.

[12] Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., and Hanrahan, P., "Brook for GPUs: stream computing on graphics hardware," in *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, (New York, NY, USA), pp. 777–786, ACM, 2004.

[13] Buyukkurt, B. and Najj, W., "Compiler generated systolic arrays for wavefront algorithm acceleration on FPGAs," *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pp. 655–658, Sept. 2008.

[14] Caspi, P., Pilaud, D., Halbwachs, N., and Plaice, J. A., "LUSTRE: a declarative language for real-time programming," in *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '87, (New York, NY, USA), pp. 178–188, ACM, 1987.

[15] Chapman, B., Jost, G., and Pas, R. v. d., *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[16] Cheema, M. O. and Hammami, O., "Application-specific SIMD synthesis for reconfigurable architectures," *Microproc. and Microsys.*, vol. 30, no. 6, pp. 398 – 412, 2006.

[17] Chen, L., Villa, O., Krishnamoorthy, S., and Gao, G., "Dynamic load balancing on single- and multi-GPU systems," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1 –12, april 2010.

[18] Cho, M. H., Cheng, C.-C., Kinsy, M., Suh, G. E., and Devadas, S., "Diastolic arrays: throughput-driven reconfigurable computing," in *Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '08, (Piscataway, NJ, USA), pp. 457–464, IEEE Press, 2008.

[19] Clark, N., Hormati, A., and Mahlke, S., "VEAL: virtualized execution accelerator for loops," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, (Washington, DC, USA), pp. 389–400, IEEE Computer Society, 2008.

[20] Clark, N. T., Zhong, H., and Mahlke, S. A., "Automated custom instruction generation for domain-specific processor acceleration," *IEEE Trans. Comput.*, vol. 54, pp. 1258–1270, October 2005.

[21] DeHaven, K., "Extensible processing platform ideal solution for a wide range of embedded systems," 2010. `http://www.xilinx.com/support/documentation/white_papers/wp369_Extensible_Processing_Platform_Overview.pdf`.

[22] Diamos, G. and Yalamanchili, S., "Speculative execution on multi-GPU systems," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1 –12, april 2010.

[23] Diefendorff, K., Dubey, P. K., Hochsprung, R., and Scales, H., "AltiVec extension to PowerPC accelerates media processing," *IEEE Micro*, vol. 20, pp. 85–95, March 2000.

[24] Fiduccia, C. M. and Mattheyses, R. M., "A linear-time heuristic for improving network partitions," in *Proceedings of the 19th Design Automation Conference*, DAC '82, (Piscataway, NJ, USA), pp. 175–181, IEEE Press, 1982.

[25] Fisher, J. A., Faraboschi, P., and Desoli, G., "Custom-fit processors: letting applications define architectures," in *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 29, (Washington, DC, USA), pp. 324–335, IEEE Computer Society, 1996.

[26] Gelernter, D. and Carriero, N., "Coordination languages and their significance," *Commun. ACM*, vol. 35, pp. 97–107, February 1992.

[27] Gokhale, M. B., Stone, J. M., Arnold, J., and Kalinowski, M., "Stream-Oriented FPGA Computing in the Streams-C High Level Language," *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, vol. 0, p. 49, 2000.

[28] Goldbeter, A. and Pourquie, O., "Modeling the segmentation clock as a network of coupled oscillations in the Notch, Wnt and FGF signaling pathways," *Journal of Theoretical Biology*, vol. 252, pp. 574–585, 2008.

[29] Gonzalez, R. E., "Xtensa: a configurable and extensible processor," *IEEE Micro*, vol. 20, pp. 60–70, March 2000.

[30] Gordon, M. I., Thies, W., and Amarasinghe, S., "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, (New York, NY, USA), pp. 151–162, ACM, 2006.

[31] Gordon, M. I., Thies, W., Karczmarek, M., Lin, J., Meli, A. S., Lamb, A. A., Leger, C., Wong, J., Hoffmann, H., Maze, D., and

Amarasinghe, S., "A stream compiler for communication-exposed architectures," in *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, ASPLOS-X, (New York, NY, USA), pp. 291–303, ACM, 2002.

[32] Guennebaud, G. and Jacob, B., "Eigen library," 2010. `http://eigen.tuxfamily.org`.

[33] Guo, Z., Buyukkurt, B., and Najjar, W., "Input data reuse in compiling window operations onto reconfigurable hardware," *SIGPLAN Not.*, vol. 39, no. 7, pp. 249–256, 2004.

[34] Guo, Z., Najjar, W., and Buyukkurt, B., "Efficient hardware code generation for FPGAs," *ACM Trans. Archit. Code Optim.*, vol. 5, pp. 6:1–6:26, May 2008.

[35] Hagiescu, A., Bordoloi, U. D., Chakraborty, S., Sampath, P., Ganesan, P. V. V., and Ramesh, S., "Performance analysis of FlexRay-based ECU networks," in *Proceedings of the 44th annual Design Automation Conference*, DAC '07, (New York, NY, USA), pp. 284–289, ACM, 2007.

[36] Hagiescu, A., Huynh, H. P., Wong, W.-F., and Goh, R. S. M., "Automated architecture-aware mapping of streaming applications onto GPUs," in *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium*, IPDPS '11, pp. 467–478, IEEE, 2011.

[37] Hagiescu, A. and Wong, W.-F., "Co-synthesis of FPGA-based application-specific floating point SIMD accelerators," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '11, (New York, NY, USA), pp. 247–256, ACM, 2011.

[38] Hagiescu, A., Wong, W.-F., Bacon, D. F., and Rabbah, R., "A computing origami: folding streams in FPGAs," in *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, (New York, NY, USA), pp. 282–287, ACM, 2009.

[39] Hill, M. D. and Marty, M. R., "Amdahl's Law in the Multicore Era," *Computer*, vol. 41, pp. 33–38, July 2008.

[40] Hong, S. and Kim, H., "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," in *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, (New York, NY, USA), pp. 152–163, ACM, 2009.

[41] Hormati, A., Kudlur, M., Mahlke, S., Bacon, D., and Rabbah, R., "Optimus: efficient realization of streaming applications on FPGAs," in *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, CASES '08, (New York, NY, USA), pp. 41–50, ACM, 2008.

[42] Hormati, A. H., Samadi, M., Woh, M., Mudge, T., and Mahlke, S., "Sponge: portable stream programming on graphics engines," in *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, (New York, NY, USA), pp. 381–392, ACM, 2011.

[43] Huynh, H. P., Hagiescu, A., Wong, W.-F., and Goh, R. S. M., "Scalable Framework for Mapping Streaming Applications onto Multi-GPU Systems," to appear in *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP'12 (New Orleans, LA, USA), 2012.

[44] Huynh, H. P., Liang, Y., and Mitra, T., "Efficient custom instructions generation for system-level design," in *Field-Programmable Technology (FPT), 2010 International Conference on*, pp. 445 –448, dec. 2010.

[45] "VFP math library." `http://code.google.com/p/vfpmathlibrary`.

[46] Karypis, G. and Kumar, V., "Multilevel k-way partitioning scheme for irregular graphs," *J. Parallel Distrib. Comput.*, vol. 48, pp. 96–129, January 1998.

[47] Kernighan, B. W. and Lin, S., "An efficient heuristic procedure for partitioning graphs," *The Bell System Technical Journal*, vol. 49, no. 2, pp. 76–80, 1970.

[48] Khronos OpenCL Working Group, *The OpenCL specification, version 1.0.29*, 2008. `http://www.khronos.org/registry/cl/specs/opencl-1.0.29.pdf`.

[49] Kitano, H., "Computational systems biology," *Nature*, vol. 420, pp. 206–210, 2002.

[50] Koller, D. and Friedman, N., *Probabilistic graphical models: principles and techniques (adaptive computation and machine learning)*. The MIT Press, 2009.

[51] Kudlur, M. and Mahlke, S., "Orchestrating the execution of stream programs on multicore platforms," in *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, (New York, NY, USA), pp. 114–124, ACM, 2008.

[52] Lee, E. A. and Messerschmitt, D. G., "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comput.*, vol. 36, no. 1, pp. 24–35, 1987.

[53] Li, L., Feng, H., and Xue, J., "Compiler-directed scratchpad memory management via graph coloring," *ACM Trans. Archit. Code Optim.*, vol. 6, no. 3, pp. 1–17, 2009.

[54] Liu, B., Hsu, D., and Thiagarajan, P. S., "Probabilistic approximations of ODEs based bio-pathway dynamics," vol. 412, no. 21, pp. 2188–2206, 2011. Theoretical Computer Science.

[55] "Link time optimization," 2009. `http://gcc.gnu.org/wiki/LinkTimeOptimization`.

[56] Lumsdaine, A., Lee, L.-Q., and Siek, J., "The iterative template library," 2006. `http://osl.iu.edu/research/itl`.

[57] Lutz, D. and Hinds, C., "Accelerating floating-point 3D graphics for vector microprocessors," in *Signals, systems and computers*, vol. 1, pp. 355 – 359, 2003.

[58] Lysecky, R. and Vahid, F., "A study of the speedups and competitiveness of FPGA soft processor cores using dynamic hardware/software partitioning," in *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, (Washington, DC, USA), pp. 18–23, IEEE Computer Society, 2005.

[59] Maedo, A., Ozaki, Y., Sivakumaran, S., Akiyama, T., Urakubo, H., Usami, A., Sato, M., Kaibuchi, K., and Kuroda, S., "Ca^{2+}-independent phospholipase A2-dependent sustained Rho-kinase activation exhibits all-or-none response," *Genes to Cells*, vol. 11, pp. 1071–1083, 2006.

[60] Matsumoto, M. and Nishimura, T., "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Trans. Model. Comput. Simul.*, vol. 8, pp. 3–30, January 1998.

[61] "Virtex-5 ML510 development platform," 2009. `http://xilinx.com/support/documentation/ml510.htm`.

[62] Murphy, K. P., *Dynamic bayesian networks: representation, inference and learning*. PhD thesis, University of California, Berkeley, 2002.

[63] Neuendorffer, S. and Vissers, K., "Streaming systems in FPGAs," in *SAMOS* (Berekovic, M., Dimopoulos, N. J., and Wong, S., eds.), vol. 5114 of *Lecture Notes in Computer Science*, pp. 147–156, Springer, 2008.

[64] Newburn, C. J., So, B., Liu, Z., McCool, M. D., Ghuloum, A. M., Toit, S. D., Wang, Z.-G., Du, Z., Chen, Y., Wu, G., Guo, P., Liu, Z., and Zhang, D., "Intel's Array Building Blocks: A retargetable, dynamic compiler and embedded language.," in *CGO*, pp. 224–235, IEEE, 2011.

[65] Nickolls, J. and Dally, W. J., "The GPU computing era," *IEEE Micro*, vol. 30, pp. 56–69, 2010.

[66] Nikhil, R. S., "Using GPCE principles for hardware systems and accelerators: (bridging the gap to HW design)," in *Proceedings of the eighth international conference on Generative programming and component engineering*, GPCE '09, (New York, NY, USA), pp. 1–2, ACM, 2009.

[67] "NVIDIA CUDA." `http://www.nvidia.com/object/cuda_home_new.html`.

[68] "Optimization framework for java," 2010. `http://opt4j.sourceforge.net/`.

[69] Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., and Phillips, J. C., "GPU Computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[70] Owens, J. D., Luebke, David, Govindaraju, Naga, Harris, Mark, Kruger, Jens, Lefohn, Aaron, E., Purcell, and Timothy, J., "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007.

[71] Pacheco, P. S., *Parallel programming with MPI*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996.

[72] Papakonstantinou, A., Gururaj, K., Stratton, J., Chen, D., Cong, J., and Hwu, W.-M., "FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs," in *Application Specific Processors, 2009. SASP '09. IEEE 7th Symposium on*, pp. 35 –42, july 2009.

[73] Papakonstantinou, A., Liang, Y., Stratton, J. A., Gururaj, K., Chen, D., Hwu, W.-M. W., and Cong, J., "Multilevel Granularity Parallelism Synthesis on FPGAs," in *Proceedings of the 2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '11, (Washington, DC, USA), pp. 178–185, IEEE Computer Society, 2011.

[74] "HPC project, Par4All," 2011. http://www.par4all.org.

[75] Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B., and Hwu, W.-m. W., "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, (New York, NY, USA), pp. 73–82, ACM, 2008.

[76] Schaa, D. and Kaeli, D., "Exploring the multiple-GPU design space," in *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, (Washington, DC, USA), pp. 1–12, IEEE Computer Society, 2009.

[77] Schreiber, R., Aditya, S., Mahlke, S., Kathail, V., Rau, B. R., Cronquist, D., and Sivaraman, M., "PICO-NPA: high-level synthesis of nonprogrammable hardware accelerators," *J. VLSI Signal Process. Syst.*, vol. 31, pp. 127–142, June 2002.

[78] Sermulins, J., Thies, W., Rabbah, R., and Amarasinghe, S., "Cache aware optimization of stream programs," *SIGPLAN Not.*, vol. 40, no. 7, pp. 115–126, 2005.

[79] Skahill, K., *VHDL for Programmable Logic*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1996.

[80] "StreamIt benchmarks," 2006. http://groups.csail.mit.edu/cag/streamit/shtml/benchmarks.shtml.

[81] "Stretch: software reconfigurable processors," 2010. http://www.stretchinc.com.

[82] Stuart, J. A. and Owens, J. D., "Multi-GPU MapReduce on GPU Clusters," in *2011 IEEE International Parallel and Distributed Processing Symposium (IPDPS '11)*, 2011.

[83] Sun, W., Wirthlin, M. J., and Neuendorffer, S., "Combining module selection and resource sharing for efficient FPGA pipeline synthesis," in *Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, FPGA '06, (New York, NY, USA), pp. 179–188, ACM, 2006.

[84] Thies, W., Karczmarek, M., and Amarasinghe, S. P., "StreamIt: a language for streaming applications," in *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, (London, UK), pp. 179–196, Springer-Verlag, 2002.

[85] Togawa, N., Tachikake, K., Miyaoka, Y., Yanagisawa, M., and Ohtsuki, T., "Instruction set and functional unit synthesis for SIMD processor cores," in *Proceedings of the 2004 Asia and South Pacific Design Automation Conference*, ASP-DAC '04, (Piscataway, NJ, USA), pp. 743–750, IEEE Press, 2004.

[86] Tournavitis, G., Wang, Z., Franke, B., and O'Boyle, M. F., "Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping," *SIGPLAN Not.*, vol. 44, pp. 177–187, June 2009.

[87] Tudor, B. M. and Teo, Y. M., "A Practical Approach for Performance Analysis of Shared Memory Programs," in *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium*, IPDPS '11, pp. 649–660, IEEE, 2011.

[88] Tumeo, A., Branca, M., Camerini, L., Ceriani, M., Palermo, G., Ferrandi, F., Sciuto, D., and Monchiero, M., "A dual-priority real-time multiprocessor system on FPGA for automotive applications," in *Proceedings of the conference on Design, automation and test in Europe*, DATE '08, (New York, NY, USA), pp. 1039–1044, ACM, 2008.

[89] Udupa, A., Govindarajan, R., and Thazhuthaveetil, M. J., "Software pipelined execution of stream programs on GPUs," in *Proceedings of*

the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '09, (Washington, DC, USA), pp. 200–209, IEEE Computer Society, 2009.

[90] Virding, R., Wikström, C., and Williams, M., *Concurrent programming in ERLANG (2nd ed.)*. Hertfordshire, UK, UK: Prentice Hall International (UK) Ltd., 1996.

[91] Whaley, R. C. and Petitet, A., "Minimizing development and maintenance costs in supporting persistently optimized BLAS," *Softw. Pract. Exper.*, vol. 35, pp. 101–121, February 2005.

[92] Woh, M., Lin, Y., Seo, S., Mahlke, S., Mudge, T., Chakrabarti, C., Bruce, R., Kershaw, D., Reid, A., Wilder, M., and Flautner, K., "From SODA to scotch: The evolution of a wireless baseband processor," in *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, (Washington, DC, USA), pp. 152–163, IEEE Computer Society, 2008.

[93] Wolfe, M., "Implementing the PGI accelerator model," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, (New York, NY, USA), pp. 43–50, ACM, 2010.

[94] Wong, H., Papadopoulou, M.-M., Sadooghi-Alvandi, M., and Moshovos, A., "Demystifying GPU microarchitecture through microbenchmarking," in *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pp. 235 –246, march 2010.

[95] Woods, N., "Integrating FPGAs in high-performance computing: the architecture and implementation perspective," in *Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*, FPGA '07, (New York, NY, USA), pp. 132–132, ACM, 2007.

[96] Wray, S., Luk, W., and Pietzuch, P., "Exploring algorithmic trading in reconfigurable hardware," in *Application-specific Systems Architectures and Processors (ASAP), 2010 21st IEEE International Conference on*, pp. 325 –328, july 2010.

[97] "LogiCORE IP Virtex-5 APU Floating-Point Unit (v1.01a)," 2011. `http://www.xilinx.com/support/documentation/ip_documentation/apu_fpu_virtex5.pdf`.

[98] Ye, X., Fan, D., Lin, W., Yuan, N., and Ienne, P., "High performance comparison-based sorting algorithm on many-core GPUs," in *IPDPS 2010*, pp. 1–10, Apr. 2010.

[99] Yiannacouras, P., Steffan, J. G., and Rose, J., "VESPA: portable, scalable, and flexible FPGA-based vector processors," in *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, CASES '08, (New York, NY, USA), pp. 61–70, ACM, 2008.

[100] Yu, P. and Mitra, T., "Scalable custom instructions identification for instruction-set extensible processors," in *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '04, (New York, NY, USA), pp. 69–78, ACM, 2004.

[101] Zhang, L., Zhang, K., Chang, T. S., Lafruit, G., Kuzmanov, G. K., and Verkest, D., "Real-time high-definition stereo matching on FPGA," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '11, (New York, NY, USA), pp. 55–64, ACM, 2011.

[102] Zhang, Z., Fan, Y., Jiang, W., Han, G., Yang, C., and Cong, J., *AutoPilot: A platform-based ESL synthesis system).* Springer Science, 2008.

[103] Ziegler, H. and Hall, M., "Evaluating heuristics in automatically mapping multi-loop applications to FPGAs," in *Proceedings of the 2005*

ACM/SIGDA 13th international symposium on Field-programmable gate arrays, FPGA '05, (New York, NY, USA), pp. 184–195, ACM, 2005.

[104] Zuluaga, M., Kluter, T., Brisk, P., Topham, N., and Ienne, P., "Introducing control-flow inclusion to support pipelining in custom instruction set extensions," *Application Specific Processors, Symposium on*, vol. 0, pp. 114–121, 2009.

APPENDIX A

ADDITIONAL BENCHMARKS

A custom version of FFT, named FFT' is described below. It suits code generation for FPGA platforms. This implementation revolves around the expensive floating-point operations. Each floating-point operator is encapsulated in a distinct filter, and it is implemented using one of the pipelines described in Table 6.1. Because the filter serializes the inputs, the initiation interval of the pipeline is assumed to be 2 cycles. Using this implementation, the replication algorithm in Chapter 3 can better tune the resources to obtain the fastest design. The implementation receives as an input parameter the vector size N.

```
/////////////////
// Entry point //
/////////////////
float->float pipeline FFT'(N) {
  add splitjoin {
    split roundrobin(2*n);
    for(int i=0; i<2; i++) {
      add pipeline {
        add FFTReorder(n);
        for(int j=2; j<=n; j*=2)
          add CombineDFT(j);
      }
    }
    join roundrobin(2*n);
  }
}

////////////////////////////////////////////////////////////////
```

```
// Floating point operators encapsulated in StreamIt filters  //
////////////////////////////////////////////////////////////////
float->float filter Multiply() {
  work push 1 pop 2 {
    float a = pop();
    float b = pop();
    float c = a*b;
    push (c);
  }
}
float->float filter Add() {
  work push 1 pop 2 {
    float a = pop();
    float b = pop();
    float c = a+b;
    push (c);
  }
}
float->float filter Subtract() {
  work push 1 pop 2 {
    float a = pop();
    float b = pop();
    float c = a-b;
    push (c);
  }
}

////////////////////////////////////////////
// Tables for constant sin and cos functions  //
////////////////////////////////////////////
void->float filter GenerateWi(int n) {
```

```
  float[n] w;
  init {
    float wn_r = (float)cos(2 * 3.141592654 / n);
    float wn_i = (float)sin(-2 * 3.141592654 / n);
    float real = 1;
    float imag = 0;
    float next_real, next_imag;
    for (int i=0; i<n; i+=2) {
      w[i] = real;
      w[i+1] = imag;
      next_real = real * wn_r - imag * wn_i;
      next_imag = real * wn_i + imag * wn_r;
      real = next_real;
      imag = next_imag;
    }
  }
  work push 2*n pop 0 {
    int i;
    for (i = 0; i < n; i += 2) {
      float weight_real = w[i];
      float weight_imag = w[i+1];
      push (weight_real);
      push (weight_imag);
      push (weight_imag);
      push (weight_real);
    }
  }
}

////////////////////////////////////////////////
// Cross combine the tables and the input data  //
```

```
////////////////////////////////////////////////
float->float pipeline CrossComp(int n) {
  add splitjoin {
    split roundrobin(1,0);
      add splitjoin {
        split duplicate;
          add Identity<float>;
          add Identity<float>;
        join roundrobin(1);
      }
      add GenerateWi(n);
    join roundrobin(1);
  }
  add Multiply();
  add splitjoin {
    split roundrobin(1);
    add Subtract();
    add Add();
    join roundrobin(1);
  }
}



////////////////////
// One DFT round  //
////////////////////
float->float pipeline CombineDFT(int n) {
  add splitjoin {
    split roundrobin(n);
    add Identity<float>;
    add CrossComp(n);
```

```
    join roundrobin(1);
  }
  add splitjoin {
    split duplicate;
    add Add();
    add Subtract();
    join roundrobin(n);
  }
}

//////////////////////
// Data reordering  //
//////////////////////
float->float filter FFTReorderSimple(int n) {
  int totalData;
  init {
    totalData = 2*n;
  }
  work push 2*n pop 2*n {
    int i;
    for (i = 0; i < totalData; i+=4) {
      push(peek(i));
      push(peek(i+1));
    }
    for (i = 2; i < totalData; i+=4) {
      push(peek(i));
      push(peek(i+1));
    }
    for (i=0;i<n;i++) {
      pop();
      pop();
```

```
    }
  }
}
float->float pipeline FFTReorder(int n) {
  for(int i=1; i<(n/2); i*= 2)
add FFTReorderSimple(n/i);
}
```