TO

My Parents

Who have stood by me unconditionally and are a constant source of support and encouragement through everything
Acknowledgements

This thesis would never have been possible — and that’s not mere hyperbole — without the support and mentoring of my research supervisors Prof. R. Govindarajan and Prof. Matthew J. Thazhuthaveetil. From the long discussions to the midnight reviews of papers over the phone and in person, I sincerely thank both of them for their patience, guidance and optimism through the last two years, which has had an overall positive impact on me over and above assisting me to no end towards this thesis. I would also like to thank Prof. Y. Narahari for generously allowing me to use the CPLEX solver, taking hours of CPU time, in his lab.

I would also like to thank the chairman of the Dept. of Computer Science and Automation, Prof. M. Narasimha Murthy for being a source of constant support and easing a lot of administrative procedures I’ve had to face over the course of my stay here. Special thanks go to the staff at the Dept. of CSA, Ms. Lalitha Mohan and Ms. Suguna for taking prompt action, even during all the times I’ve approached them at the last minute for help with some paperwork.

The folks at the Laboratory for High Performance Computing (a.k.a HPC Lab), where the bulk of this work was carried out, have my sincere gratitude for maintaining a wonderful working atmosphere in the lab: Mani and Kaushik for the awesome company and conversation (technical and non-technical) over the innumerable trips for coffee and tea and for reviewing drafts of my papers. Mani, also for providing “cash advances” and “zero interest short-term loans” to fund travel for attending the two conferences and Kaushik for all the evenings of “pagan traditions” and his casual easy sense of humor, as well as the use of his credit card for some of my transactions. Thanks to Girish for his entertaining the lab with his collection of Kannada songs and
his cheerful good humor and also for the trips to the coffee board and the occasional trips to S.P. Road. Thanks are also due to Santhosh Nagarakatte and Aditya Thakur for many technical discussions. Sreepathi for maintaining the infrastructure in the lab in excellent shape as the lab admin and Mrugesh for the debates on topics ranging from politics to socio-politico-cultural issues.

I would also like to thank my friends from the Visualization and Graphics lab, Suthambhara, Nithin and Sumit, for the relaxing and often humorous conversations over coffee and over dinner over the last two and a half years.

Special thanks to my parents who have put up with and accepted all my crazy ideas on what life is about as well as being a constant source of support and encouragement to whatever endeavors I chose to undertake. I couldn’t have done this without them.

Thanks are also due to various musicians like AC/DC, Aerosmith, Iron Maiden, Creedence Clearwater Revival, Lynyrd Skynyrd, Metallica, Finntroll, Rammstein, Pink Floyd, Jefferson Airplane, The Doors, Judas Priest, Eagles, Iggy Pop, The Ramones, Led Zeppelin, The Who, Chris Cornell, Eric Clapton, The Beatles, Tool, Hariprasad Chaurasia and Pt. Ravi Shankar to name a few, and many others who’ve provided entertainment kept me in “the right state of mind” through and through.

Lastly, but not the least, I would like to express my gratitude to God for constantly keeping me on the right track and helping me do well in all my endeavors and bringing hope through all my little trials and tribulations.
Publications based on this Thesis

1. Abhishek Udupa, R. Govindarajan and Matthew J. Thazhuthaveetil; “Software Pipelined Execution of Stream Programs on GPUs”. In the Proceedings of the International Symposium on Code Generation and Optimization 2009 (CGO 2009), Seattle, WA, USA, March 2009

Abstract

Over the past two decades, microprocessor manufacturers have typically relied on wider issue widths and deeper pipelines to obtain performance improvements for single threaded applications. However, in the recent years, with power dissipation and wire delays becoming primary design constraints, this approach can no longer be effectively used to yield performance improvements. Thus processor designers and vendors are universally moving towards multi-core designs. Examples for these are the commodity general purpose multi-core processors, the CellBE accelerator from IBM and the Graphics Processing Units from NVIDIA and ATI. Although these many and multi-core architectures can provide enormous performance benefits, it is difficult to program for them, due to the complexity of writing explicitly parallel code. The ubiquity of computationally intensive media processing applications, makes it imperative to consider new programming frameworks and languages that can express parallelism in an easy, portable manner. The StreamIt programming language has been proposed to efficiently exploit parallelism at various levels on general purpose multi-core architectures and stream processors and allow media processing and DSP application to be developed in an easy and portable fashion. The StreamIt model allows programmers to specify programs as a set of filters connected by FIFO communication channels. The graphs thus specified by the StreamIt programs describe task, data and pipeline parallelism which can be potentially exploited on modern Graphics Processing Units (GPUs), which have emerged as powerful, commodity stream processors, which support abundant parallelism in hardware.

The first part of this thesis deals with the challenges in mapping StreamIt programs to GPUs and proposes an efficient technique to software pipeline the execution
of stream programs on GPUs. We formulate this problem — both scheduling and assignment of filters to processors — as an efficient Integer Linear Program (ILP), which is then solved using ILP solvers. We also describe a novel buffer layout technique for GPUs which facilitates exploiting the high memory bandwidth available in GPUs. The proposed scheduling utilizes both the scalar units in GPU, to exploit data parallelism, and multiprocessors, to exploit task and pipeline parallelism. Our approach yields a (geometric) mean speedup of 5.02X, with a maximum speedup of 36.83X across a set of StreamIt benchmarks, with the speedup measured relative to an optimized single threaded CPU execution.

While the approach of software pipelining the execution of stream programs on GPUs is efficient and performs well, it does not utilize the CPU cores to perform useful computation. Further, it does not support programs with stateful filters, which are essentially filters that are not data parallel owing to a dependence between each successive firing that is carried through the implicit state of the filter. The second part of the thesis aims at addressing these issues and describes a novel method to orchestrate the execution of a StreamIt program on the multiple cores of a system and GPUs in a synergistic manner. The proposed approach identifies, using profiling, the relative benefits of executing a task on the superscalar CPU cores and the accelerator. We formulate the problem of partitioning the work between the CPU cores and the GPU, taking into account the latencies for data transfers and the required buffer layout transformations associated with the partitioning, as an integrated Integer Linear Program (ILP) which can then be solved by an ILP solver. We also propose an efficient heuristic algorithm for the work partitioning between the CPU and the GPU, which provides solutions which are within 9.05% of the optimal solution on an average across the benchmark suite. The partitioned tasks are then software pipelined to execute on the multiple CPU cores and the Streaming Multiprocessors (SMs) of the GPU. The software pipelining algorithm orchestrates the execution between CPU cores and the GPU by emitting the code for the CPU and the GPU, and the code for the required data transfers. Our experiments on a platform with 8 CPU cores and a GeForce 8800 GTS 512 GPU show a (geometric) mean speedup of 6.84X with a maximum of 51.96X over a single threaded CPU execution across a set of StreamIt benchmarks.
Although the techniques proposed in this thesis have been implemented using the StreamIt compiler framework and target the NVIDIA GPUs, we believe that the ideas can easily be applied to other stream programming languages and other accelerator architectures which are similar to GPUs. Thus, this thesis proposes and evaluates some effective techniques for the compilation of stream programs for execution on multi-core platforms, which are equipped with an accelerator such as the GPU.
# Contents

<table>
<thead>
<tr>
<th>Acknowledgements</th>
<th>iii</th>
</tr>
</thead>
<tbody>
<tr>
<td>Publications based on this Thesis</td>
<td>v</td>
</tr>
<tr>
<td>Abstract</td>
<td>vii</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Software Pipelined Execution of Stream Programs on GPUs</td>
<td>6</td>
</tr>
<tr>
<td>1.2 Synergistic Execution of Stream Programs</td>
<td>8</td>
</tr>
<tr>
<td>2 Background</td>
<td>13</td>
</tr>
<tr>
<td>2.1 NVIDIA GPU Architecture</td>
<td>13</td>
</tr>
<tr>
<td>2.2 Background on CUDA</td>
<td>16</td>
</tr>
<tr>
<td>2.2.1 Definitions</td>
<td>17</td>
</tr>
<tr>
<td>2.2.2 The CUDA API</td>
<td>19</td>
</tr>
<tr>
<td>2.2.3 Execution Configuration</td>
<td>20</td>
</tr>
<tr>
<td>2.2.4 An Illustrative Example</td>
<td>23</td>
</tr>
<tr>
<td>2.3 The StreamIt Language</td>
<td>27</td>
</tr>
<tr>
<td>2.3.1 The StreamIt Constructs</td>
<td>27</td>
</tr>
<tr>
<td>2.3.2 Scheduling of StreamIt Programs</td>
<td>30</td>
</tr>
<tr>
<td>3 Software Pipelining of Stream Programs onto GPUs</td>
<td>35</td>
</tr>
<tr>
<td>3.1 Introduction</td>
<td>35</td>
</tr>
<tr>
<td>3.2 A Motivating Example</td>
<td>37</td>
</tr>
<tr>
<td>3.3 Overview of the Compilation Process</td>
<td>43</td>
</tr>
<tr>
<td>3.4 Profiling and Execution Configuration Selection</td>
<td>44</td>
</tr>
<tr>
<td>3.4.1 Profiling</td>
<td>44</td>
</tr>
<tr>
<td>3.4.2 Execution Configuration Selection</td>
<td>46</td>
</tr>
<tr>
<td>3.4.3 Execution Configuration Selection: Putting it all Together</td>
<td>49</td>
</tr>
<tr>
<td>3.5 ILP Formulation</td>
<td>51</td>
</tr>
<tr>
<td>3.5.1 Definitions</td>
<td>51</td>
</tr>
<tr>
<td>3.5.2 Resource Constraints</td>
<td>52</td>
</tr>
<tr>
<td>3.5.3 Dependence Constraints</td>
<td>54</td>
</tr>
<tr>
<td>3.5.4 Buffer Constraints</td>
<td>57</td>
</tr>
<tr>
<td>3.6 Code Generation</td>
<td>59</td>
</tr>
</tbody>
</table>
### 3.7 Buffer Layout Optimizations

- 3.7.1 Experimental Methodology
- 3.7.2 Experimental Results

### 3.8 Experimental Evaluation

- 3.8.1 Experimental Methodology
- 3.8.2 Experimental Results
  - 3.8.2.1 Effect of Coarsening the Granularity
  - 3.8.2.2 Comparison with Other Techniques
  - 3.8.2.3 Efficacy of the ILP formulation

### 3.9 Summary

---

### 4 Synergistic Execution of Stream Programs

- 4.1 Introduction
- 4.2 Buffer Layout Considerations
- 4.3 A Motivating Example
- 4.4 Overview of the Compilation Methodology
- 4.5 ILP Formulation of the Partitioning Problem
  - 4.5.1 Task Partitioning
    - 4.5.1.1 Definitions
    - 4.5.1.2 Formulation of the Task Partitioning Problem
  - 4.5.2 Instance Partitioning
    - 4.5.2.1 Definitions
    - 4.5.2.2 Formulation of the Instance Partitioning Problem
- 4.6 An Efficient Heuristic Partitioning Algorithm
- 4.7 The Shuffle and Deshuffle Operations
- 4.8 Modulo Scheduling
- 4.9 Code Generation
- 4.10 Experimental Evaluation
  - 4.10.1 Experimental Methodology
  - 4.10.2 Experimental Results
    - 4.10.2.1 Efficiency of the Heuristic Partitioner
    - 4.10.2.2 Execution Using the Heuristic and ILP Approaches
    - 4.10.2.3 Comparison with Simpler Heuristics
    - 4.10.2.4 Scalability Study: Tesla 1060 GPU
- 4.11 Summary

---

### 5 Related Work

- 5.1 Stream Programming
- 5.2 Stream Architectures
- 5.3 General Purpose and Stream Programming on GPUs

---

### 6 Conclusions

- 6.1 Summary
- 6.2 Future Work

---

### References
List of Tables

1.1 Power and Frequency trends of Intel Microprocessors .................. 2

2.1 Impact of Execution Configuration on Performance ...................... 21
   (a) A Program which requires 9 registers per thread .................. 21
   (b) A Program which requires 12 registers per thread ................. 21
   (c) A Program which requires 20 registers per thread ................. 21
   (d) A Program which requires 26 registers per thread ................. 21

3.1 Benchmarks Evaluated .................................................. 68
3.2 Buffer Requirements of the Benchmarks .................................. 71
3.3 Solution Times and Relaxation for the Benchmarks ....................... 72

4.1 Impact of Data Layout on Performance .................................... 78
4.2 Details of the Benchmarks Evaluated ...................................... 100
4.3 Evaluation of the Heuristic Partitioner ................................... 102
4.4 Evaluation of the Heuristic Partitioner Applied to the Tesla C1060 .... 107
## List of Figures

2.1 The Organization of the GeForce 8800 GPU ........................................... 14
2.2 The \textit{saxpy} kernel on a uni-threaded CPU ........................................... 24
2.3 The \textit{saxpy} kernel on the SSE SIMD unit using gcc intrinsics ................. 24
2.4 The \textit{saxpy} kernel on the GPU using CUDA ............................................ 25
2.5 Execution of the \textit{saxpy} kernel on different architectures ....................... 26
2.6 The StreamIt Constructs ........................................................................ 27
2.7 An example stream graph ....................................................................... 31
3.1 Motivating Example .............................................................................. 40
   (a) An Example StreamIt Graph ................................................................. 40
   (b) Simplified Stream Graph .................................................................. 40
   (c) SIMD Execution of a Stream Program .............................................. 40
   (d) Software Pipelined Execution of the Stream Program ...................... 40
3.2 The Software Pipelined Kernel Unrolled .................................................. 42
3.3 Overview of the Compilation Methodology ............................................. 45
3.4 Execution Configuration A: 128 and B: 128 threads ................................ 49
3.5 Execution Configuration A: 256 and B: 128 threads ................................ 50
3.6 A Multi-rate Stream Graph and its Dependency Graph ............................. 55
3.7 Code Generated for the Software Pipelined Kernel ................................. 61
3.8 A Naïve Buffer Layout that causes Bank Conflicts ................................. 62
   (a) A Naïve Buffer Layout: Bank Conflicts at time $t_0$ ......................... 62
   (b) A Naïve Buffer Layout: Bank Conflicts at time $t_1$ ......................... 62
3.9 A Conflict-free Data Layout .................................................................... 63
   (a) A Layout that causes no Bank Conflicts at time $t_0$ ......................... 63
   (b) A Layout that causes no Bank Conflicts at time $t_1$ ......................... 63
3.10 A Conflict-free Buffer Layout ................................................................. 65
3.11 Effect of Coarsening on Performance .................................................... 69
3.12 Comparison of the SWP scheme with other schemes ......................... 70
4.1 Motivating Example for Synergistic Partitioning ....................................... 80
4.2 The Synergistically Software Pipelined Kernel ....................................... 81
4.3 Overview of the Synergistic Compilation Methodology ............................. 83
4.4 The Shuffle Process on the GPU ............................................................. 94
4.5 The Deshuffle Process on the GPU .......................................................... 95
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.6</td>
<td>Performance of the ILP vs. Heuristic Partitioner</td>
<td>103</td>
</tr>
<tr>
<td>4.7</td>
<td>Comparison of Synergistic Execution with other schemes</td>
<td>105</td>
</tr>
<tr>
<td>4.8</td>
<td>Performance of the partitioning schemes on the Tesla C1060</td>
<td>108</td>
</tr>
<tr>
<td>4.9</td>
<td>Performance on a Tesla C1060 and a GeForce 8800 GTS 512</td>
<td>109</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

When tomorrow is today the bells may toll for some
But nothing can change the shape of things to come

The Shape of Things to Come
THE RAMONES

Controlling complexity is the essence of computer programming
BRIAN W. KERNIGHAN

Over the last two decades, general purpose microprocessor manufacturers have continually relied on wider issue widths, larger caches with higher associativity and deeper pipelines to provide performance improvements for single threaded applications. Unfortunately, despite improvements in feature sizes and reduced operating voltages, the power dissipation of successive generations of microprocessors has been increasing at an alarming rate as shown in Table 1.1 [6]. Power dissipation and wire delays have become dominant processor design constraints to the extent that it is no longer possible to rely solely on micro-architectural enhancements for performance improvements. Due to these reasons, microprocessor manufacturers in the recent years have almost universally moved towards multi-core architectures, thus effectively ending the era of the “free lunch”, where programmers and users could rely solely on micro-architectural enhancements to improve the performance of their uni-threaded or sequential applications.

Programmers now need to restructure or rewrite their sequential applications to expose parallelism explicitly, rather than depend on the micro-architecture or the compiler
Chapter 1. Introduction

<table>
<thead>
<tr>
<th>Model</th>
<th>Clock Speed (MHz)</th>
<th>Year</th>
<th>Power Dissipation (W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium</td>
<td>75</td>
<td>1993</td>
<td>8.0</td>
</tr>
<tr>
<td>Pentium MMX</td>
<td>233</td>
<td>1996</td>
<td>17.0</td>
</tr>
<tr>
<td>Pentium II</td>
<td>450</td>
<td>1997</td>
<td>27.1</td>
</tr>
<tr>
<td>Pentium III</td>
<td>1400</td>
<td>2001</td>
<td>31.2</td>
</tr>
<tr>
<td>Pentium IV HT</td>
<td>3066</td>
<td>2004</td>
<td>81.8</td>
</tr>
<tr>
<td>Core 2 Extreme QX9770</td>
<td>3200</td>
<td>2008</td>
<td>150</td>
</tr>
</tbody>
</table>

Table 1.1: Power and Frequency trends of Intel Microprocessors

to exploit implicit parallelism in the form of Instruction Level Parallelism. This further complicates the development of streaming applications on general purpose processors, since, in addition to hand tuning their applications for specific platforms, programmers would now need to develop concurrent or multi-threaded applications, which are much harder to develop and reason about than sequential applications. Lastly, general purpose processors have traditionally added larger and multiple levels of caches in order to tolerate the growing gap between processor speeds and memory access latency. While this approach works well for applications with good locality characteristics, they are wasteful for streaming applications, since these applications have little or no data reuse. Streaming applications would benefit more from increased processing resources and small software managed scratchpad memories.

Media Processing and DSP applications have seen widespread use in the recent years owing to audio and video increasingly being archived and distributed digitally. Functions which have traditionally been implemented by dedicated analog circuitry are often being implemented as software, as is the case with Software Defined Radio (SDR). These media processing and DSP applications are extremely compute intensive and usually have real-time performance requirements. These applications have traditionally been implemented in C or C++, with hand tuned optimizations for each target platform, primarily due to performance concerns. Needless to say, development and maintenance of such applications across multiple platforms is a challenge. However, these media processing applications fit well into the stream model of computation. The
stream model of computation is characterized by the following aspects:

- Organization of the application as a set of filters, which operate on data and produce output, which may in turn be inputs to other filters for further processing
- Data parallelism, where a filter may process multiple inputs in parallel, rather than sequentially
- Regular and predictable data flow between filters
- Producer-consumer locality between filters

Development of such programs in C or C++ tends to be tedious and leads to obscure code, owing to the fact that the programmer needs to explicitly manage the communication between program fragments or filters. A language which exposes the semantics of the program, including the stream structure, would be better suited for describing such applications.

This confluence of increased performance demands by media processing applications, the difficulty of developing and maintaining these applications written in conventional programming languages like C and C++ and the inability of general purpose processors to provide the performance demanded by these applications has led to:

1. Exploration of processor architectures that exploit the characteristics of streaming applications in order to provide improved performance
2. Development of programming models and frameworks that are better suited to describing streaming applications than C or C++

We now briefly discuss the progress in each of these directions.

**Stream Processors:** These are processors which have been designed from the ground up with streaming applications in mind. Notable examples are the Imagine Stream Processor [33][39] and the RAW processor [64].

The Imagine processor is a clustered VLIW architecture with local scratchpad memories and a high speed interconnect between clusters. The clusters execute VLIW instructions broadcast to them by the front-end micro-controller in a SIMD fashion. Each
cluster consists of a number of Arithmetic and Logical Units (ALUs), with varying capabilities, and also local register files for each of the ALUs and a global scratchpad memory, shared across the ALUs. The routing of operands between the local register files of each ALU must be specified by the programmer as discussed in [49]. Each cluster also has a high speed dedicated inter cluster communication controller and a high-speed interface to the stream register file, which is a large fast register file into which data can be streamed from or to the main memory. The programming model comprises of a set of extensions to the C programming language, KernelC and StreamC for orchestrating computation on the clusters and orchestrating the execution between them and for data transfers to and from the stream register file respectively. Conceptually, the StreamC program orchestrates the execution and data transfers required for the execution of the individual kernels, which actually perform the computation on the clusters in the Imagine stream processors. Needless to say, the programmer needs to orchestrate all the operations manually using the StreamC model. This often tends to obscure the streaming nature of the computation and places unnecessary burden on the programmer for regular and structured stream programs.

The RAW processor on the other hand is organized as a tiled architecture, where each tile consists of a MIPS style pipeline and routers for transferring operands to and from tiles. The routers are programmable and are a part of the high speed interconnect between tiles, which is mapped to the register file on each tile. The idea behind the architecture is to have a large number of simple processor cores with the communication network (and the latency of this network) between these processor cores exposed to the programmer. Imperative programming languages like C have been compiled for the RAW processor, as described in [46] and [47], which attempt to exploit Instruction Level Parallelism (ILP) present in the program and map them to the multiple processing cores available. Stream languages like StreamIt [65] have also been successfully translated for the RAW architecture [25] [26].

While the two approaches have shown a lot of promise, they have not yet become “off-the-shelf” commodities yet and as such have not seen widespread adoption. In the meantime, Graphics Processing Units (GPUs), have emerged from being fixed function processing units to powerful programmable stream processors capable of delivering
upto 400 GFLOPs of processing power \[56\] \[57\]. GPUs are commodity processors and present a very affordable alternative to specialized stream processors. Programming frameworks like CUDA \[3\], CTM \[2\] and OpenCL \[8\] have been proposed for programming GPUs. These frameworks provide primitives to expose data parallelism. However, these frameworks have a steep learning curve associated with them. Also, the stream structure of the program is not exposed and the programmer would still need to orchestrate the communication between filters. Also, the use of these programming frameworks (with the exception of OpenCL) make the program platform specific. For instance, migrating a program from an ATI GPU to an NVIDIA GPU or vice versa would require a complete rewrite of the program in the appropriate framework for the GPU. The tremendous computational power offered by GPUs and their ubiquity makes it worthwhile to explore the compilation of stream programs for GPUs.

**Stream Programming Languages and Frameworks:** On the language front, several frameworks and languages have been proposed to ease the task of developing stream programs, targeting a wide variety of platforms. The Imagine project \[33\] \[39\] has proposed the *StreamC* and *KernelC* extensions to the \(C\) programming language. As mentioned earlier, in this model, the high level stream structure is captured by the *StreamC* program, which runs on the control processor and data parallelism is captured by a set of *KernelC* programs, which run on the Imagine stream processor. The programmer is still responsible for orchestrating the execution of the filters on the stream processor. Languages like *Brook* \[16\] have similar issues. The Accelerator framework \[62\] takes a different approach and proposes the *data parallel array* for performing data parallel computations. This approach too does not expose the stream structure of the program. Further the programmer is restricted to the set of operations that are defined on the data parallel array.

Considering that all these frameworks leave a lot to be desired in terms of being able to express streaming computation in a portable and easy fashion, we focus our attention in this thesis to the StreamIt language \[65\], which allows the programmer to structure programs as a hierarchical composition of the basic stream structure (described in detail in Section \[2.3.1\]), thus exposing the stream structure of the program to the compiler. Data parallelism is trivially exposed by the description of the filters,
where multiple inputs can be processed in parallel as long as the filter in consideration contains no modifiable state that is persistent across firings. The approach also allows for portability, since the programmer only defines the operations performed by each filters and the producer-consumer relationships between them, delegating the task of producing optimal code for each platform to the specific compiler back end. Optimizing back ends for the CellBE [32, 40, 69], the RAW processor [25, 26, 64] and commodity multi-core architectures have already been developed. Thus the StreamIt language is an attractive alternative to developing streaming application in C or C++. However, to the best of our knowledge, no compilation framework exists for the compilation of StreamIt programs for GPUs.

The enormous computational capabilities and the ubiquity of GPUs and the ease of expression of streaming applications with the StreamIt language, coupled with the lack of a portable and easy to use programming framework for GPUs make a compelling case for automatically generating code to be executed on GPUs, given the StreamIt specification of the DSP or media processing application. To the best of our knowledge, this work described in thesis is the first such attempt.

1.1 Software Pipelined Execution of Stream Programs on GPUs

StreamIt programs — also called StreamIt graphs — describe a wealth of parallelism in the form of task, data and pipeline parallelism, as described in [26]. We propose a methodology which maps the parallelism available at various levels onto the structures available on a GPU for efficient execution on the GPU. While Coarse Grained Software Pipelining techniques for stream programs has been studied in [26] in the context of the RAW processor [64] and the CellBE accelerator [40, 69], the approaches cannot be applied to GPUs in a straightforward way, due to the unique constraints of GPUs. The performance of a program executing on the GPU depends on the following factors:

- The number of threads the program executes with: The GPU is a data parallel compute engine and allows the programmer to specify the level of data parallelism the program is to be executed with
1.1. Software Pipelined Execution of Stream Programs on GPUs

- The data access pattern of the program: The GPU has a high bandwidth memory channel. However, effective utilization of the bandwidth requires that the accesses be organized in a specific fashion.

We make the following contributions in the area of compiling StreamIt programs for execution on the GPU:

1. We propose a profile driven methodology for determining the optimal *execution configuration* which describes the number of threads each filter executes with on the GPU.

2. We develop an integrated ILP formulation for the work assignment and scheduling of the software pipelined kernel on the GPU in the presence of resource and buffer constraints.

3. We propose a buffer layout scheme that ensures that the large memory bandwidth available on the GPU is effectively utilized.

4. We have implemented and evaluated the proposed approach on the StreamIt compiler.

Since the execution configuration plays an important role in the performance of a program executing on the GPU, our profile driven methodology consists of executing each filter in the StreamIt graph with a varying number of threads, while accounting for the register requirements of the filters and determining the execution configuration for each filter such that it is optimal for the *entire* StreamIt program.

Our ILP formulation differs from earlier formulations as suggested in [27, 29, 40] in the following aspects:

1. We take resource constraints into account, which has not been done in [27, 29].

2. We do not model data parallel filters using explicit split-join constructs as in [40], rather using the multi-rate properties of StreamIt graphs for exposing parallelism.

3. We take into account that filter instances cannot *wrap around* an Initiation Interval (II).
4. We do not assume that the multiple executions of a given filter are serialized as done in [27, 29]. Instead, our formulation allows multiple executions of filters to execute concurrently the available parallelism to be exploited efficiently. This naturally complicates the expression of the dependence constraints, which our formulation takes care of.

5. Our formulation takes into account the inability of being able to communicate data between the different SMs of the GPU reliably, except across GPU kernel invocations and modifies the dependence constraints accordingly.

6. The ILP formulation proposed in [40] performs only the work distribution across processing elements. Our ILP formulation handles work distribution and scheduling while simultaneously taking buffer constraints into account.

The buffer layout optimizations we propose are essential for effectively utilizing the large memory bandwidth available on the GPU. Our evaluation reaffirms this, with a software pipelined execution scheme which does not coalesce accesses using the technique we propose, performing much worse than our approach, which uses an optimized buffer layout scheme in order to coalesce all accesses to the GPU device memory.

We have evaluated our implementation of the software pipelining framework for executing StreamIt programs on GPUs on a test-bed equipped with a GeForce 8800 GTS 512 GPU with 16 SMs and 512 MB of memory. Our results indicate a geometric mean speedup of 5.02X over an optimized single threaded CPU execution, with a maximum speedup of 36.83X. The approach also performs significantly better than simpler techniques for executing StreamIt programs on GPUs.

Chapter 3 describes the software pipelining of Stream Programs on GPUs in detail and also presents performance results.

1.2 Synergistic Execution of Stream Programs on Multicores with Accelerators

While the approach of software pipelining the execution of StreamIt programs makes efficient use of the GPU, it cannot handle programs with stateful filters, which have
1.2. Synergistic Execution of Stream Programs

persistent state that is updated each firing and is used to compute the values of the output. Such filters are not data parallel and are unsuited for execution on the GPU. Further, the CPU cores are left idle, while the GPU performs computation. This forms the motivation to explore the synergistic approach, where both the GPU and the CPU cores are used to perform computation.

The primary challenges with this approach are as follows:

- The CPU cores and GPU operate on disjoint address spaces
- Any data transfer between the GPU and the CPU address spaces needs to be accomplished by DMA transfers. The DMA bandwidth being limited
- The buffer layout for the GPU causes performance degradation on the CPU cores

Since the CPU cores and the GPU operate on distinct, disjoint address spaces any communication between the two would require DMA transfers into and/or out of the GPU. This requires that the partitioning technique be aware of the limited DMA bandwidth available, as well as be able to hide the latency of the DMA transfers behind useful computation on the CPU cores as well as the GPU SMs.

While the optimized buffer layout proposed in Chapter 3 facilitates the effective use of the large bandwidth available on the GPU, it results in inefficient use of the memory hierarchy on the CPU. Specifically, the strided memory accesses that result from the buffer layout being used “as is” on the CPU cores cause a large number of cache misses and may cause a complete loss of spatial locality depending on the size of the caches.

Our work addresses each of these problems and provides an integrated solution. This thesis makes the following contributions in this area:

1. We model the GPU SMs, the CPU cores and the DMA channel as resources and formulate the partitioning problem as an Integer Linear Program (ILP)

2. The proposed methodology takes into account the DMA latencies and the latencies for the buffer transformation operations necessary to ensure that the buffers are laid out in a manner most suitable for the CPU or the GPU, depending on where the operations on the buffer are scheduled
3. We also propose a heuristic partitioning technique that compared favorably with the optimal solutions obtained using the ILP formulation.

4. We have developed an efficient technique for performing the required buffer transformations on the GPU, exploiting the high memory bandwidth available on the GPU.

5. We have implemented the proposed compilation methodology on the StreamIt compiler framework.

The problem of partitioning the work across the CPU cores and the GPU is not amenable to traditional graph partitioning heuristics such as those proposed in [38], due to the fact that the weights on the nodes are not static. i.e., the weights change depending on the partition a node is assigned to, owing to the fact that the same filter has different execution times on the CPU and the GPU. Also, we do not require that the weights of the cut-edges be minimal. We only require that they do not dominate the weights of each of the partitions as far as possible.

Considering the complexity of the partitioning problem, we formulate the partitioning problem as an Integer Linear Program that accurately models the compute resources available on both the CPU and the GPU as well as the DMA bandwidth available on the system. We also propose a heuristic partitioning algorithm, owing to the large amount of time that the ILP formulation takes to be solved. The heuristic partitioner compares favorably with the ILP partitioner, both in terms of the Initiation Interval (II) achieved, as well as in terms of the actual execution times.

We have developed an efficient technique to perform the requisite buffer transformations that are associated with a partition. Our technique effectively utilizes the high memory bandwidth available on the GPU to perform these operations, while avoiding bank conflicts and uncoalesced accesses at the device memory. Our approach involves staging the data into the shared memory first, avoiding bank conflicts at the device memory instead tolerating the bank conflicts on the device memory, which are 1-cycle conflicts as opposed to the 600–800 cycle conflicts on the device memory.

Our evaluation of the synergistic software pipelining technique for StreamIt programs indicates a geometric mean speedup of 6.84X and 6.68X, respectively, for the ILP
1.2. Synergistic Execution of Stream Programs

partitioning and the heuristic partitioning techniques, with the speedups being measured relative to an optimized single threaded CPU execution. The evaluation was carried out on a test-bed utilizing four Intel Xeon processors clocked at 2.83 GHz and a GeForce 8800 GTS 512 GPU. We have also evaluated our approach using a Tesla C1060 GPU, which consists of 30 SMs and 4 GB of memory, where our technique achieves a geometric mean speedup of 8.73X and 9.41X respectively using the ILP and heuristic partitioning techniques.

The rest of the thesis studies the two problems in detail. Chapter 2 provides a background of StreamIt, GPUs and CUDA. Chapter 3 describes the software pipelining of StreamIt programs for execution on the GPUs, while Chapter 4 studies the synergistic work partitioning technique. Chapter 5 discusses the related work in the area of stream architectures, stream programming and GPU programming. Finally, Chapter 6 briefly describes the future work that may be carried out in this area and concludes.
Chapter 2

Background

‘Begin at the beginning,’ the King said gravely, ‘and go on till you come to the end: then stop.’

‘But I don’t want to go among mad people,’ Alice remarked. ‘Oh, you can’t help that,’ said the Cat: ‘we’re all mad here. I’m mad. You’re mad.’

LEWIS CARROLL

If I have seen further, it is by standing on the shoulders of giants.

SIR ISAAC NEWTON

This chapter reviews the necessary background for the work described in the subsequent chapters of this thesis. We begin by describing the architecture of the NVIDIA GPUs in Section 2.1. Section 2.2 describes the programming model for NVIDIA GPUs, CUDA, with an example. Section 2.3 describes the StreamIt programming language constructs and discusses the scheduling of StreamIt programs.

2.1 NVIDIA GPU Architecture

The organization of the NVIDIA GeForce 8800 GTS 512 GPU [3] is shown in Figure 2.1. While we describe the organization of the NVIDIA GPUs with the specific example of the GeForce 8800 GTS 512 GPU, the architecture of the other NVIDIA GPUs remains conceptually the same, differing only in scale. The GPU consists of 16 Streaming Multiprocessors (SMs), shown as SM0–SMn in Figure 2.1(a). Each SM may be conceptually viewed as a data parallel compute engine. The SMs are connected to the device memory.
Chapter 2. Background

Figure 2.1: The Organization of the GeForce 8800 GPU

using a high bandwidth memory bus, which is 256–512 bits wide, depending on the model. The latency of a read or write to memory is 600–800 cycles [3]. There are two read-only caches in the GPU:

1. The constant cache serves to cache the data values which have been declared as constants during compile time.

2. The texture cache helps in exploiting temporal locality in the memory access stream. When data in the device memory is accessed through specific operations called texture fetches, they are cached in the texture cache. This causes subsequent accesses to nearby data to hit in the cache, resulting in quicker access and reduced memory traffic.

While studies in [67] have indicated that other levels of smaller caches and TLBs exist on GPUs, these are not documented by the manufacturer. Since these details are not germane to the techniques described in this thesis, we do not discuss them further.

Figure 2.1(b) shows the internal organization of each SM, which consists of 8 Scalar Units (SUs). The SUs have a common instruction fetch and control mechanism and execute instructions in lock-step. Each SM has hardware multi-threading capabilities and periodically switches between threads to hide the latency of the device memory. This is transparent to the programmer. Thus, although each SM has only 8 SUs, it is capable of
executing up to 768 threads concurrently, owing to the hardware multi-threading support available in the SMs. However, the number of threads that may actually execute on each SM is constrained by the register and the shared memory requirements of each thread as described later in Section 2.2. Each thread executing on an SM is identified by the hardware using a unique threadid. The basic hardware schedulable entity is the warp, which is a group of 32 threads with consecutive threadids. Since the SUs operate in lock-step, and the hardware schedules threads at the granularity of warps, any divergence in control flow within the threads of a warp, causes the divergent portions of the threads to serialize, degrading performance. However, threads across warps or threads executing on different SMs may diverge with no performance penalty. Each SM has a large register file with 8192 32-bit registers capable of holding integer or single precision floating point data. The GeForce 8800 GTS 512 supports IEEE compliant 32-bit (single precision) floating point arithmetic, but lacks support for denormalized values. The more recent GPUs such as the Tesla C1060 support IEEE 64-bit (double precision) floating point arithmetic as well. Data in the register files can be accessed with a 1-cycle latency. Each SM also has 16KB shared memory, which is akin to a software managed cache and also has a latency of 1-cycle [3]. The programmer is responsible for allocation and management of the shared memory.

It should be noted that the GPU and CPU address spaces are disjoint. Data transfer between these two address spaces need to be carried out by explicit DMA transfers. APIs for performing the DMA operations and managing the device memory are provided by the CUDA framework as described later in Section 2.2. The program to be executed on the GPU is called a kernel. We shall henceforth refer GPU programs as a GPU kernel to distinguish it from the software pipelined kernel code described in Chapters 3 and 4. Also the GPU only serves as a slave co-processor for the CPU or host processor. Thus any kernel launch on the device or DMA transfer must be initiated by the CPU and the GPU is not capable of performing any of these operations autonomously.

**Synchronization and Memory Consistency:** At the time of this writing GPUs have very limited synchronization mechanisms, although subsequent models may support a richer set of synchronization mechanisms. Hardware synchronization primitives are available at two levels through intrinsics provided by the nvcc compiler:
• Across threads executing on the same SM: The NVIDIA GPUs provide hardware support for barrier-style synchronization across threads executing on the same GPU.

• Across SMs: Synchronization across SMs is possible through primitive atomic instructions. Higher level synchronization mechanisms such as mutexes and barriers can be built up on these using software.

Since the issue order of threads across SMs is undefined, threads cannot communicate reliably using the device memory. The shared memory on each SM is guaranteed to be sequentially consistent within a thread and relaxed ordering across threads [3]. However, writes to shared memory by all threads are guaranteed to be visible to all threads after they have synchronized using the barrier-style synchronization.

**Memory Access Patterns:** The device memory on the GPU is organized as multiple banks and is connected to the SMs through a wide (256 – 512 bits depending on the model) memory bus. Although the latency of the device memory is on the higher side (600 – 800 cycles), the device memory is capable of servicing loads at a high bandwidth thanks to the wide interface. However, achieving this theoretically high bandwidth requires that the accesses to device memory be organized in a bank-friendly fashion. The memory bandwidth is best utilized when access by the threads of a warp are coalesced. This essentially means that the threads with contiguous threadids in a warp access contiguous memory addresses with the address accessed by the first thread aligned suitably to the size of the warp. This ensures that all the loads of the threads of the warp can be serviced in a single memory transaction, effectively utilizing the high bandwidth provided by the wide memory interface and amortizing the high memory latency over a large number (upto 32 threads in a warp) of loads.

### 2.2 Background on CUDA

CUDA [3] is a data parallel extension to C++ for developing applications for execution on the GPU. CUDA programs consist of code to be executed on the host processor (CPU) as well as the data parallel code to be executed on the device (GPU). The CUDA compiler `nvcc` preprocesses the source code, separating the two parts. The host code
is compiled by the native compiler for the host (\texttt{gcc} or Visual Studio C++ compiler, \texttt{cl.exe}, depending on the platform), while the device code is compiled using the \texttt{nvcc} compiler into a binary format called \textit{fatbin}, which is then embedded into the resulting executable. When the program thus compiled is executed, it starts execution with the host processor in control. The host processor may then dispatch GPU kernels to the device as required, at which point the CUDA run-time library sets up and executes the relevant \textit{fatbin} \cite{7} object on the device. From this point on, we shall use the term \textit{host} to refer to the CPU and \textit{device} to refer to the GPU.

2.2.1 Definitions

We provide some definitions for the terms used in this chapter. These definitions have been adopted from \cite{3,7}.

- **Warp**: A \textit{warp} is a group of 32 threads with contiguous \textit{threadids} as described in Section 2.2.

- **Block**: A \textit{block} is a group of up to 512 threads. Blocks can have up to three dimensions. While the hardware schedules threads at the granularity of warps, it is oblivious to blocks. The block is an abstraction for the programmer to group threads and its size and dimensions are defined by the programmer when the GPU kernel is to be launched. Threads in a block are guaranteed to execute on the same SM. However, multiple blocks could potentially execute on the same SM depending on the register and the shared memory requirements of each thread. The threads in a block can synchronize using the barrier-style synchronization by calling the \texttt{__syncthreads()} intrinsic.

- **Grid**: A \textit{grid} is a set of blocks, whose size and dimensions are defined by the programmer at the time of GPU kernel launch. Each GPU kernel launch consists of exactly one grid. A grid can have up to three dimensions.

- **Execution Configuration**: An \textit{execution configuration} is the number of threads and blocks a GPU grid is to be launched with. It is specified by the programmer.
To execute a GPU kernel, the programmer specifies the number of threads each block is to be executed with and the number of blocks the kernel is to executed with, along with the entry point of the kernel and parameters to the kernel, if any. All the SMs begin executing the GPU kernel at the same point. The individual SMs may subsequently diverge without any performance penalty. Since the CPU and GPU address spaces are disjoint, CUDA uses the following function and variable type specifiers to specify whether a function is to be executed on the GPU and also the location of variables.

- The \texttt{__device__} qualifier may be used for both variables and functions. If used with a variable, it specifies that the variable resides in the GPU address space and when used with a function it specifies that the function may only be called from within code executing on the GPU.

- The \texttt{__global__} qualifier may only be used with functions and specifies a GPU function that may be only called by the host processor. The \texttt{__global__} functions are used to set up entry points for GPU kernels.

- Variables that are to reside in the shared memory space may be specified using the \texttt{__shared__} variable type qualifier.

- A variable declared with the \texttt{__constant__} qualifier resides in the constant address space and may not be modified by code executing on the GPU. These variables may be cached in the constant cache shown in Figure 2.1.

The CUDA run-time system makes the following information available to the GPU kernel through predefined variables:

1. The dimensions of the grid: \texttt{gridDim}, which is a 3-tuple

2. The dimensions of the block: \texttt{blockDim}, which is a 3-tuple

3. The thread identifier of each thread is available through the \texttt{threadDim} variable, which is again a 3-tuple
2.2. Background on CUDA

2.2.2 The CUDA API

The CUDA framework provides a rich programming interface which provides APIs for DMA transfers, memory management, stream management, event management and device management and time measurement. We describe the API features that are relevant to this thesis. Further details about the CUDA framework can be obtained from [3, 7].

DMA Transfers: Support for DMA transfers are provided by the cudaMemcpy and the cudaMemcpyAsync functions, which copy data from (into) the device memory into (from) the host memory. The difference between the two functions is that cudaMemcpy blocks until the DMA transfer is actually completed, while the cudaMemcpyAsync returns immediately after queuing the necessary DMA commands. For the asynchronous functions, the programmer would need to ensure that the transfer is actually completed by using a stream or event mechanism. These functions may also be used by the host processor to transfer data from one location in the device memory to another.

Memory Management: The function cudaMalloc allows the programmer to allocate memory in the GPU address space, while the function cudaMallocHost allows the programmer to allocate pinned or non-page-able memory (as opposed to non-pinned or page-able memory) in the host address space. Note that the cudaMemcpyAsync family of functions may only be called with a pointer to pinned memory as a parameter, if the DMA transfer involves the host memory as a source or destination.

Stream Management: CUDA provides an abstraction called streams to support concurrency. Each operation may be optionally associated with a stream. The CUDA run-time guarantees that the operations which have the same stream identifier will be executed in the order they were issued. Streams are represented by the type cudaStream_t and are created with the cudaStreamCreate function. Once a stream has been created, it can then be passed as an optional parameter to many other functions in the CUDA API.

Event Management: The CUDA framework provides the cudaEvent_t type to abstract events. Events are created using the cudaEventCreate API and recorded using the cudaEventRecord API. Once events have been recorded, the time elapsed
between two events may be determined using the function \texttt{cudaEventElapsedTime}.

### 2.2.3 Execution Configuration

As mentioned in Section 2.2.1, the execution configuration determines the number of threads and blocks a GPU kernel is to be executed with and must be specified by the programmer at the time the kernel is to be launched on the GPU. The execution configuration can impact performance as discussed in [56, 57]. The execution configuration also depends on the register and shared memory requirements of the GPU kernel. For example, if a GPU kernel requires 32 registers per thread, then the kernel may be launched with a maximum of \( \left\lfloor \frac{8192}{32} \right\rfloor = 256 \) threads, assuming the number of registers per SM to be 8192. The number of registers to be used per thread can be bounded during compile time by specifying the maximum number of registers to be used as a parameter to the \texttt{nvcc} compiler [7]. The compiler will then generate the necessary code to spill into the device memory and fill from the device memory into the register file. The impact this has on performance can vary between programs.

Table 2.1 shows the performance of four different programs with different execution configuration, with the best performing configurations shown in \textbf{boldface}. These programs are filters from the actual StreamIt benchmarks [9] and not toy examples. Each benchmark is compiled with register restrictions of 64, 32, 20 and 16 registers per thread. These numbers were chosen so as to guarantee that the program can be executed with 128, 256, 384 and 512 threads respectively. If the register requirements of a program are \textit{less than} the limit specified, then it has no effect on the compilation process. The programs were executed such that all versions perform the same amount of work, regardless of the number of threads they were executed with. For example, if the program was being executed with 128 threads, then it performs \( n \) iterations of work, while a the same program executing with 256 threads would perform only \( \frac{n}{2} \) iterations of work. In other words, the product of the number of iterations and the number of threads is maintained a constant over all the runs.

The program considered in Table 2.1(a) requires 9 registers per thread, and can thus execute with the full 512 threads allowed per block without restricting the number of registers used at compile time. Its performance varies only with the number of threads.
### 2.2. Background on CUDA

#### Table 2.1: Effect of Varying the Execution Configuration Across Different Programs

<table>
<thead>
<tr>
<th>Max. Registers</th>
<th>Number of Threads</th>
<th>Max. Registers</th>
<th>Number of Threads</th>
<th>Max. Registers</th>
<th>Number of Threads</th>
<th>Max. Registers</th>
<th>Number of Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>128</td>
<td>3.02 ms</td>
<td>256</td>
<td>3.02 ms</td>
<td>384</td>
<td>3.02 ms</td>
<td>512</td>
</tr>
<tr>
<td>32</td>
<td>4.34 ms</td>
<td>3.22 ms</td>
<td>3.02 ms</td>
<td>3.22 ms</td>
<td>3.02 ms</td>
<td>3.02 ms</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>4.34 ms</td>
<td>3.22 ms</td>
<td>3.02 ms</td>
<td>3.22 ms</td>
<td>3.02 ms</td>
<td>3.02 ms</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>4.34 ms</td>
<td>3.22 ms</td>
<td>3.02 ms</td>
<td>3.22 ms</td>
<td>3.02 ms</td>
<td>3.02 ms</td>
<td></td>
</tr>
</tbody>
</table>

(a) The `LowPassFilter` filter from the ChannelVocoder Benchmark, which requires 9 registers per thread.

<table>
<thead>
<tr>
<th>Max. Registers</th>
<th>Number of Threads</th>
<th>Max. Registers</th>
<th>Number of Threads</th>
<th>Max. Registers</th>
<th>Number of Threads</th>
<th>Max. Registers</th>
<th>Number of Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>41.83 ms</td>
<td>34.96 ms</td>
<td>33.21 ms</td>
<td>34.96 ms</td>
<td>33.21 ms</td>
<td>34.96 ms</td>
<td>34.96 ms</td>
</tr>
<tr>
<td>32</td>
<td>41.83 ms</td>
<td>34.96 ms</td>
<td>33.21 ms</td>
<td>34.96 ms</td>
<td>33.21 ms</td>
<td>34.96 ms</td>
<td>34.96 ms</td>
</tr>
<tr>
<td>20</td>
<td>41.83 ms</td>
<td>34.96 ms</td>
<td>33.21 ms</td>
<td>34.96 ms</td>
<td>33.21 ms</td>
<td>34.96 ms</td>
<td>34.96 ms</td>
</tr>
<tr>
<td>16</td>
<td>41.83 ms</td>
<td>34.96 ms</td>
<td>33.21 ms</td>
<td>34.96 ms</td>
<td>33.21 ms</td>
<td>34.96 ms</td>
<td>34.96 ms</td>
</tr>
</tbody>
</table>

(b) The `iDCT_1D_reference_fine` filter from the DCT Benchmark, which requires 12 registers per thread.

<table>
<thead>
<tr>
<th>Max. Registers</th>
<th>Number of Threads</th>
<th>Max. Registers</th>
<th>Number of Threads</th>
<th>Max. Registers</th>
<th>Number of Threads</th>
<th>Max. Registers</th>
<th>Number of Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>15.0 ms</td>
<td>15.6 ms</td>
<td>18.78 ms</td>
<td>15.6 ms</td>
<td>18.78 ms</td>
<td>15.6 ms</td>
<td>18.78 ms</td>
</tr>
<tr>
<td>32</td>
<td>15.0 ms</td>
<td>15.6 ms</td>
<td>18.78 ms</td>
<td>15.6 ms</td>
<td>18.78 ms</td>
<td>15.6 ms</td>
<td>18.78 ms</td>
</tr>
<tr>
<td>20</td>
<td>15.0 ms</td>
<td>15.6 ms</td>
<td>18.78 ms</td>
<td>15.6 ms</td>
<td>18.78 ms</td>
<td>15.6 ms</td>
<td>18.78 ms</td>
</tr>
<tr>
<td>16</td>
<td>16.32 ms</td>
<td>15.89 ms</td>
<td>15.07 ms</td>
<td>15.89 ms</td>
<td>15.07 ms</td>
<td>15.89 ms</td>
<td>15.07 ms</td>
</tr>
</tbody>
</table>

(c) The `CombineDFT` filter from the TDE benchmark, which requires 20 registers per thread.

<table>
<thead>
<tr>
<th>Max. Registers</th>
<th>Number of Threads</th>
<th>Max. Registers</th>
<th>Number of Threads</th>
<th>Max. Registers</th>
<th>Number of Threads</th>
<th>Max. Registers</th>
<th>Number of Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>19.82 ms</td>
<td>20.35 ms</td>
<td>22.22 ms</td>
<td>20.35 ms</td>
<td>22.22 ms</td>
<td>20.35 ms</td>
<td>22.22 ms</td>
</tr>
<tr>
<td>32</td>
<td>19.82 ms</td>
<td>20.35 ms</td>
<td>22.22 ms</td>
<td>20.35 ms</td>
<td>22.22 ms</td>
<td>20.35 ms</td>
<td>22.22 ms</td>
</tr>
<tr>
<td>20</td>
<td>21.31 ms</td>
<td>21.89 ms</td>
<td>22.22 ms</td>
<td>21.89 ms</td>
<td>22.22 ms</td>
<td>21.89 ms</td>
<td>22.22 ms</td>
</tr>
<tr>
<td>16</td>
<td>23.56 ms</td>
<td>24.70 ms</td>
<td>24.48 ms</td>
<td>24.70 ms</td>
<td>24.48 ms</td>
<td>24.70 ms</td>
<td>24.48 ms</td>
</tr>
</tbody>
</table>

(d) The `Sboxes` filter from the DES benchmark, which requires 26 registers per thread.
and yields the best performance with 512 threads.

On the other hand, the program considered in Table 2.1 (b) requires 12 registers per thread, and can again be executed with the full 512 threads without compile time restrictions. However, in this case, we find that the best performance is achieved with 384 threads. In fact, the performance degrades slightly when the program is executed with 512 threads. This is due to the fact that the entire memory bandwidth is effectively utilized at the level of multi-threading of 384 threads. Increasing the level of multi-threading further results in a slight performance degradation owing to the additional cost incurred by the hardware thread context switching and thread management, which is not offset by gains in throughput, since the memory bandwidth is already entirely utilized.

The program in Table 2.1 (c) requires 20 registers per thread. Thus, unless the register usage is restricted at compile time to 16 registers, this program cannot execute with 512 threads, since \(\lfloor \frac{8192}{16} \rfloor = 512\). In fact, with no register restrictions, the program can only be executed with up to 384 threads. In this case, we observe that when the register usage is restricted to 16 registers per thread the performance is worse than the other versions for 128 and 256 threads. However, the performance with 384 and 512 threads is better when the register usage is limited to 16 and yields the best performance with 512 threads. In fact, unless the register limit of 16 is applied, it is impossible to execute the program with 512 threads, as indicated by the blank columns in the first three rows of Table 2.1 (c).

Executing a program with higher levels of multi-threading by restricting the register usage at compile time need not always yield better performance. This scenario is shown in Table 2.1 (d), where the program requires 26 registers per thread. Here, the best performance is obtained for 128 threads, with register usage restricted to 64 or 32 (this effectively implies that there are no register restrictions, since the register usage of the program is only 26). The versions of the program compiled with any register restrictions perform worse than the versions with no restrictions. This is due to the overhead of the spill and fill code introduced by the register constraints, which require data to be spilled to and filled from the slower device memory. In this case, the spill-fill overhead shadow the positive effects of higher levels of multi-threading.
Thus, the execution configuration has a significant impact on the performance of a GPU kernel, especially when the register constraints are taken into consideration.

### 2.2.4 An Illustrative Example

We now provide an outline of programming with CUDA using a `saxpy` kernel in order to set the context for the rest of the thesis, as well as to highlight the differences between GPUs and other accelerator architectures like SIMD units.

The `saxpy` kernel takes as input two vectors `x` and `y` and a scalar `a`, and computes a vector \( s[i] = a \times x[i] + y[i] \). For simplicity, we consider fixed length vectors of size 8192. Figure 2.2 shows the code written for a uni-threaded CPU with no vector or SIMD extensions. The loop from lines 3 – 5 iterate over each element of the vector, computing the resultant vector. Figure 2.3 is the same program written with the Intel SSE instruction set in mind [5], using the `gcc` intrinsics [4]. We assume that all vectors are suitably aligned in memory. The loop in lines 5 – 9, processes four elements of the vector in each iteration. Thus the trip count of the loop is reduced by a factor of four in this case. Figure 2.4 shows the CUDA code, along with the driver `main()` routine for performing the same computation. The vectors `x`, `y` and `s` are declared in the device memory in lines 1 – 3. Line 4 declares the kernel with a (`__global__`) qualifier, indicating that this is the entry point to the GPU kernel. The index of the element to be processed by each thread is calculated in line 6, with the computation being performed in line 7. The driver routine copies vectors into the device memory in lines 18 – 19, and calls the GPU kernel with 16 blocks, each with 512 threads in line 22. The result is copied back into the host memory in line 23. Thus the GPU completes processing all the 8192 elements in parallel, exploiting the large levels of parallelism that the GPU can support.

Figure 2.5 graphically describes the execution of the `saxpy` kernel, which processes vectors of 8192 elements on a uni-threaded CPU, a SIMD unit, the CellBE and the GPU. Figure 2.5(a) shows a uni-threaded CPU execution where only one element of the result is computed in each iteration corresponding to the program shown in Figure 2.2. This would require 8192 iterations to process the entire vector of 8192 elements. Figure 2.5(b) depicts the execution on a 4-wide SIMD unit, which computes 4 elements of.
Chapter 2. Background

```c
void saxpy_cpu(float *s, float a, float *x, float *y) {
    for(int i = 0; i < 8192; i++) {
        s[i] = (a * x[i]) + y[i];
    }
    return;
}
```

Figure 2.2: The `saxpy` kernel on a uni-threaded CPU

```c
void saxpy_sse(float *s, float a, float *x, float *y) {
    __m128 vec_a;
    for(int i = 0; i < 8192 / 4; i++) {
        (__m128)&s[i * 4]) = _mm_mul_ps((__m128)&x[i * 4]), vec_a);
        (__m128)&s[i * 4]) = _mm_add_ps((__m128)&s[i * 4]),
    (__m128)&y[i * 4]));
}
```

Figure 2.3: The `saxpy` kernel on the SSE SIMD unit using gcc intrinsics

the result in each iteration, corresponding to the program shown in Figure 2.3. The trip count of the loop in this case is reduced to $\frac{8192}{4} = 2048$ from Figure 2.5(c) shows the execution on a single SM of a GPU, where 512 threads each calculate one element of the resultant vector. This corresponds to the execution of the program in Figure 2.4 on only one SM of the GPU rather than 16 as in Figure 2.4 In this case, the trip count of the loop is reduced to $\frac{8192}{512} = 16$. The kernel could also be launched on 16 SMs, in which case the entire vector of 8192 elements could be computed in parallel across the 16 SMs, with each SM executing just one iteration, producing 512 elements of the result vector. Note that in Figure 2.5(c) the buffers have been shown such that the vectors $x$ and $y$ are interleaved. This has been done purely for clarity; in a real implementation, such a layout would cause bank conflicts and uncoalesced accesses (apart from the fact that such an interleaved layout would be difficult to create and maintain). A real implementation would have the two vectors occupy separate arrays as shown in Figure 2.4. Thus the GPU can be thought of as an extremely wide SIMD processor. However, unlike a traditional SIMD processor, where there is no possibility for divergence, a GPU allows for divergence albeit with a performance penalty.
2.2. Background on CUDA

```
__device__ __align__(64) float s[8192];
__device__ __align__(64) float x[8192];
__device__ __align__(64) float y[8192];
__global__ void saxpy_gpu(float a)
{
    int idx = gridDim.x * blockDim.x + threadIdx.x;
    s[idx] = a * x[idx] + y[idx];
}

int main()
{
    float s_host[8192];
    float x_host[8192];
    float y_host[8192];
    float a = 3.1416;

    // Initialize x_host and y_host
    cudaMemcpyToSymbol(x, x_host, sizeof(int) * 8192);
    cudaMemcpyToSymbol(y, y_host, sizeof(int) * 8192);
    dim3 threads(512);
    dim3 blocks(16);
    saxpy_gpu <<< blocks, threads >>>(a);
    cudaMemcpyFromSymbol(s_host, s, sizeof(int) * 8192);
    // Result in s_host
    return;
}
```

Figure 2.4: The saxpy kernel on the GPU using CUDA

**Comparison with the CellBE Accelerator:** The CellBE [32] is a recently developed accelerator, which consists of a PowerPC processor (PPE), which is responsible for executing the operating system and other system programs, along with eight Synergistic Processing Elements (SPEs), each of which is a 2-way SIMD unit. Each SPE has a fast 256 KB, programmer managed local store. The SPEs can only directly access their respective local stores and cannot directly access the main memory. The code for the CellBE targeting the SPEs is very similar to the CPU or SSE code, except that the data would have to be moved into the local store of each of the SPEs by the PPE, before dispatching work to the SPEs and would need to be copied back from piecewise from each of the SPEs after the completion of the computation by the SPEs. The CellBE differs from the GPU in the number of processing elements — 8 as compared to the few hundreds on the GPU — and the fact that the SPEs can follow different control paths with no performance penalty. In other words, the CellBE is a true MIMD architecture, while the
GPU only allows MIMD execution at a large performance cost, due to serialization of threads that execute different control paths. While this may seem a disadvantage when compared to the CellBE, it is not an issue for stream programs, since stream programs have large amounts of data parallelism and can take advantage of a SIMD only model of execution.
2.3 The StreamIt Language

We now provide an overview of the StreamIt programming language and discuss the scheduling of StreamIt programs. Further details on the StreamIt language as well as the language specification can be obtained from the StreamIt home page [9].

2.3.1 The StreamIt Constructs

A StreamIt program consists of a set of nodes which perform operations on data and a set of FIFO channels through which the nodes communicate. A StreamIt graph is constructed by a hierarchical composition of filters using the basic StreamIt constructs: Pipeline, Split-Join and the Feedback Loop. These constructs are shown in Figure 2.6.

The Filter construct, shown in Figure 2.6(a), forms the basic primitive for all StreamIt programs. A filter has exactly one input and one output. The program for a filter consists of a work() function and optionally an init() function, which is called before the filter starts executing the work function to perform the initialization of any variables used within the filter. The work function may access its input FIFO using the pop() or the peek() method. The pop() method removes an element from the head of the FIFO and returns it, while the peek() method returns an element at a specified depth.
in the FIFO, without removing it. The output FIFO may be accessed using the `push()` method, which inserts a data value into the tail of the output FIFO. The work functions are not allowed to access the FIFOs through any other mechanism. Each FIFO has a type associated with it which specifies the data type that it carries. StreamIt supports the primitive types such as `int` and `float`, as well as user defined types.

The `Pipeline` construct, shown in Figure 2.6(b) allows the creation of linear pipelines. Each component of the pipeline is a `stream` in itself. A stream is recursively defined as follows:

1. A composition of other `streams` using the `pipeline`, `split-join` or `feedback loop` constructs.
2. A `StreamIt filter`.

Thus the constructs can be hierarchically composed to form a `StreamIt` program.

Figure 2.6(c) shows the `Split-Join` construct. It consists of a `splitter`, a stream for each of the `ways` of the splitter and a `joiner`. Splitters can be `duplicate splitters` or `round-robin` splitters. A duplicate splitter places a copy of each element on its input FIFO onto each of the output FIFOs, whereas a round-robin splitter places the first \( w_0 \) elements from the input FIFO onto the first output FIFO, the next \( w_1 \) elements onto the second output FIFO and so on. A `joiner` is always a round-robin and performs the inverse operation of a round-robin splitter. The number of ways \( n \) of a splitter and the weights \( w_0, w_1, \cdots, w_{n-1} \) of a splitter or a `joiner` are programmer specified.

The `Feedback Loop` construct is depicted in Figure 2.6(d). It consists of a 2-way splitter, a `body` and a `loop` stream and a 2-way `joiner`. Again, the weights of the splitter and `joiner` are programmer specified. In order to ensure deadlock-free scheduling, the feedback loop may need to have a `delay`, depending on the push and pop rates of the `loop` and `body` streams. The `delay` represents the number of tokens buffered on the feedback loop path.

**Static Rates:** The `push rate` and the `pop rate` of a filter are defined to be the number of elements (or tokens) that are pushed onto the output FIFO or popped from the input FIFO. StreamIt requires that these rates be static and known at compile time, since this is necessary for scheduling the `StreamIt` program. The `peek depth` is defined as the depth
2.3. The StreamIt Language

upto which the filter can inspect elements on the input FIFO using the \texttt{peek()} method. This is also required to be a constant for all filters.

\textbf{Firing:} A firing of a filter is exactly one execution of the filter, where it consumes \textit{push rate} elements from its input FIFO and produces \textit{pop rate} elements onto its output FIFO. In this thesis a firing of a filter is also referred to as an \textit{instance} of a filter.

\textbf{Stateful vs. Stateless Filters:} The nodes in a StreamIt graph may be \textit{stateful} or \textit{stateless}. If a node stores information that is persistent across firings in the form of state variables, which are updated on each firing, as a function of the previous values of the state variables and the input variables, and may be used to compute the outputs for subsequent firings, then it is said to be \textit{stateful}, otherwise it is said to be \textit{stateless}. Stateless nodes are data parallel since they do not have dependencies from one firing to another. In other words, several instances of a stateless node may be executed in parallel, which is not possible in the case of stateful nodes owing to the dependence between firings.

\textbf{Data Parallelism:} If multiple firings of a filter can proceed in parallel without affecting correctness, then the filter is said to be \textit{data parallel} in nature. In other words, the firings of a data parallel filter need not be serialized. By definition, only stateless filters are data parallel, since stateful filters update persistent state at each firing, creating an implicit dependence from one firing to the next. Splitters and joiners are always data parallel.

\textbf{Task Parallelism:} If two filters in a StreamIt graph are on different parallel branches of a split-join, then the output of one never reaches the other and the two filters can be executed in parallel. This is defined as \textit{task parallelism} [26].

\textbf{Pipeline Parallelism:} Pipeline parallelism refers to the parallelism seen in chains of producer-consumer dependencies in a stream graph, where different parts could be executing different iterations in a hardware pipelined or software pipelined fashion.

Every StreamIt program describes a StreamIt graph. Thus from this point on, we use the terms “StreamIt Program” and “StreamIt Graph” interchangeably. The work in this thesis deals with the scheduling of StreamIt graphs, hence we do not go into the details of the StreamIt language syntax. Further details about the StreamIt language can be obtained from the documentation and language specification from the StreamIt
Chapter 2. Background

home page [9]. We assume that the StreamIt program in textual form has been transformed by the front end of the StreamIt compiler into an intermediate representation called the Stream IR, which captures the graph structure of the program, along with the push and pop rates of each of the nodes.

2.3.2 Scheduling of StreamIt Programs

Consider a StreamIt graph \( G = \{V, E\} \), where \( V \) is the set of nodes and \( E \) is the set of edges. For each edge \((u, v) \in E\), we define \( I_{uv} \) as the number of elements popped by one firing of node \( v \), \( O_{uv} \) as the number elements pushed by node \( u \) in one firing. We also define \( m_{uv} \) to be the number of residual elements on the edge \((u, v)\). These residual elements may be required if \( v \) peeks at its inputs without consuming them.

An implicit assumption about StreamIt graphs is that there is loop enclosing the entire graph making it a looped data flow graph. In other words, the StreamIt graph is executed infinitely with an infinite stream of inputs.

Firing Rule: The firing rule for a filter states that a node \( v \) may fire only if there are at least \( I_{uv} \) elements on each incoming edge \((u, v) \in E\) and there is space for at least \( O_{vw} \) elements on each outgoing edge \((v, w) \in E\). The firing rule ensures that the data dependencies for each node are satisfied before it fires.

The problem of scheduling is to find a sequence of firings of nodes that always satisfies the firing rule, such that the sequence of firings may be repeated an infinite number of times without accumulation or depletion of on any of the edges in the stream graph. A schedule that satisfies these constraints is called a steady state schedule and one iteration of the steady state schedule is termed as a steady state iteration. The number of instances of each filter in a steady state schedule is called its steady state firing rate.

The problem of scheduling StreamIt programs is similar to the problem of scheduling Synchronous Data Flow graphs [13 43 45] with static rates. This is hardly surprising since StreamIt graphs in fact form a subset of the graphs defined by the Synchronous Data Flow model of computation. The SDF model allows a node in the SDF graph to have multiple inputs and multiple outputs whereas in StreamIt only splitters are allowed to have multiple outputs and only joiners are allowed to have multiple inputs. Owing to this property of StreamIt graphs, the results used to schedule SDF
graphs and the later Stream Flow Graphs and Regular Stream Flow Graphs (RSFGs) are directly applicable to the scheduling of StreamIt programs. Much of the terminology we define here has been adopted from.

The number of instances (or firings) $k_v$ of a filter $v$ in a steady state schedule may be determined by solving the steady state equations, which is a set of linear equations. Each steady state equation represents the condition that there ought to be no elements accumulated on an edge. In other words, each edge in the stream graph has a steady state equation associated with it, which represents the condition that the production and consumption rates on the edge over a steady state iteration are equal. Since the number of inputs (outputs) consumed (produced) by a filter at each firing is a constant, this requires that the number of firings of the filter be such that there is no accumulation of tokens or elements on any edge. The solution to the steady state equations gives precisely this firing rate. We illustrate this with an example shown in Figure 2.7. The rates on each edge are as shown in the figure. For simplicity we assume $m_{uv} = 0$ for all edges $(u, v) \in E$. In order to ensure that the buffer requirements are bounded, we need to ensure that the number of firings of each filter on each edge in the
stream graph is balanced. This forms the basis for the steady state equations which state that for each edge \((u, v) \in E\), \(k_u \times O_{uv} = k_v \times I_{uv}\). Applying this on each edge, we get the following equations:

\[
\begin{align*}
    k_0 \times 1 &= k_1 \times 2 \\
    k_0 \times 2 &= k_2 \times 1 \\
    k_1 \times 4 &= k_3 \times 2 \\
    k_2 \times 2 &= k_3 \times 4
\end{align*}
\]

These equations have an infinite number of solutions. However, we are interested only in the primitive steady state firing rate, which is such that GCD of all the \(k_i\) is one. Solving these equations yields \(k_0 = 2\), \(k_1 = 1\), \(k_2 = 4\) and \(k_3 = 2\). These are the balanced or steady state firing rates. It can be verified that firing the filters with this firing rate does not lead to the accumulation of tokens or elements on any edge. Thus, any steady state iteration must consist of exactly \(k_i\) firings of node \(v_i\). Any multiple of these firing rates is also trivially a steady state firing rate, as long as all the \(k_i\) are multiplied by the same integer. A steady state schedule with firing rates equal to that of the primitive steady state firing rates is called a primitive steady state schedule.

Although the number of firings of each filter is the same across all steady state schedules (i.e., as given by the solution to the steady state equations), two steady state schedules may differ in their buffer requirements. A Single Appearance Schedule (SAS) is a schedule where every node appears once and only once in the schedule, along with a specified multiplicity. A SAS has the maximum buffer requirement among all schedules, since each node executes all its instances at once requiring that all the elements produced and consumed be buffered at once. At the other extreme, a Push schedule is one where each node is fired just the number of time required for its successor nodes to fire. A push schedule has the lowest buffer requirements and latency among all the steady state schedules, but has the longest representation for the schedule, since each node appears multiple times at various positions in the schedule. In embedded systems, the longer representation of the schedule could lead to higher code size, since the functions are often inlined at the call sites. This trade-off between code size and buffer
requirements has been studied in detail in [13, 43, 51]. For the stream graph shown in Figure 2.7, consider the following two schedules:

\[
S_1 : \{v_0v_0v_1v_2v_2v_2v_3v_3\} = \{2v_01v_14v_22v_3\}
\]

\[
S_2 : \{v_0v_0v_1v_2v_3v_2v_2v_3\} = \{2v_01v_12v_2v_32v_2v_3\}
\]

Schedule \( S_1 \) has the following buffer requirements:

\[
\begin{align*}
(0,1) & : 2 \\
(0,2) & : 4 \\
(1,3) & : 4 \\
(2,3) & : 8
\end{align*}
\]

This yields a total buffer requirement of 18, while schedule \( S_2 \) has the buffer requirements as follows:

\[
\begin{align*}
(0,1) & : 2 \\
(0,2) & : 4 \\
(1,3) & : 4 \\
(2,3) & : 4
\end{align*}
\]

yielding a total buffer requirement of 14. Thus different steady state schedules can have different buffer requirements, with the push minimum latency schedule [13, 34, 43, 45] which corresponds to the schedule \( S_2 \) having the least buffer requirement and the Single Appearance Schedule [13, 34, 43, 45], shown in schedule \( S_2 \) above, having the highest buffer requirements. The hierarchical structure of StreamIt graphs allows for easy solutions to the problem of finding steady state schedules without actually solving the steady state equations. The algorithm works by traversing the hierarchy of streams over two passes, and propagating the execution counts of the enclosing streams to the individual child streams. More details on the scheduling algorithms used in the StreamIt compiler may be found in [34, 51].
Chapter 3

Software Pipelined Execution of Stream Programs on GPUs

3.1 Introduction

In this chapter, we describe our methodology for the compilation of StreamIt programs onto GPUs. As mentioned in Chapter 2, the GPU has emerged from being a fixed function unit to a programmable data parallel stream processor device capable of supporting thousands of threads in hardware. Compilation of StreamIt programs onto GPUs has several challenges associated with it:

- The data parallelism exposed in the form of stateless filters in a StreamIt program must be efficiently mapped to the data parallelism available on the GPU. This requires that the level of multi-threading be managed efficiently. As shown in Table 2.1 in Chapter 2, higher levels of multi-threading may not automatically
translate to better performance, owing to the register constraints and the characteristics of the program.

- The task and pipeline parallelism available in a StreamIt program needs to be exploited by orchestrating the execution of the program across the multiple Streaming Multiprocessors (SMs) available on the GPU.

We propose a *profile-driven approach* to determine the optimal level of multi-threading for a given StreamIt program and a software pipelining approach for exploiting the task and pipeline parallelism across multiple SMs of the GPU.

Software pipelining is a loop scheduling technique that schedules operations from different iterations of a loop, while ensuring that data dependencies are satisfied, to yield a higher throughput than a naïve schedule. Since a StreamIt graph can be considered to be executed over an infinite stream of inputs, there is an implicit loop surrounding the entire graph. Thus software pipelining is a natural way of scheduling such graphs. Unfortunately, software pipelining in the presence of resource constraints is known to be NP-Complete \[41\], which has resulted in the formulation of the software pipelining problem as Integer Linear Programs (ILPs) \[20, 21, 27, 28, 29\], with the objective of achieving the schedule which achieves highest throughput while minimizing the register or buffer requirements of the schedule. Prior efforts like \[27, 29\] have suggested an ILP formulation for software pipelining the execution of Regular Stream Flow Graphs (RSFGs) across multiple processors. The ILP formulation suggested in \[29\] cannot be adopted for software pipelining the execution of StreamIt programs on GPUs for the following reasons:

- Resource constraints are not considered in \[29\], which assumes that an infinite number of processors are available.

- Results produced in one GPU kernel invocation cannot be reliably accessed by a consumer within the same kernel invocation, since the GPU guarantees memory consistency at the device memory only across kernel invocations.

- The formulation of buffer constraints in \[29\] are too fine grained to be adopted for the GPU.
3.2. A Motivating Example

We now provide a brief overview of software pipelining and motivate the use of software pipelining for orchestrating the execution of StreamIt programs on GPUs with an example.

Software pipelining is a scheduling technique that has been predominantly used for instruction scheduling in loops \cite{20,28,31,41,48,53}. It has also successfully been used to orchestrate the execution of coarse grained data flow graphs described by Stream Flow Graphs \cite{27,29} as well as StreamIt graphs \cite{26,40} on parallel processor architectures. Thus, software pipelining can be considered a general technique for scheduling data flow graphs, which are allowed to contain cycles or iteration carried dependencies. Modulo scheduling is a software pipelining technique, where instances of an operation

• The formulations in \cite{27,29} assume a strict serial ordering between the execution of various instances of filters, which would prevent exploiting all the data parallelism available in the StreamIt graph. This assumption must be relaxed in order to harness all the available data parallelism, while ensuring that the data flow dependencies are still honored.

Our ILP formulation handles these additional constraints posed by the programming model of the GPUs. While the software pipelined schedule is obtained by the solution of the ILP, we relax the schedule to only yield a partial ordering among operations, which results in an efficient ILP formulation that can be solved in a reasonable amount of time on current generation workstation and server hardware.

The rest of this chapter is organized as follows: Section 3.2 motivates the use of a software pipelined approach. In Section 3.3 we provide an overview of the compilation methodology. Sections 3.4 and 3.5 discuss the profiling and execution configuration selection step of the compilation process and the Integer Linear Programming (ILP) formulation for the software pipelining problem respectively. Section 3.7 describes the buffer layout optimizations that are necessary for efficiently utilizing the memory bandwidth of the GPU. Section 3.8 presents experimental results and Section 3.9 summarizes the approach.
Chapter 3. Software Pipelining of Stream Programs onto GPUs

in successive iterations of the loop are separated by a constant time interval. This time interval is known as the *Initiation Interval* (II) of the software pipelined loop. Thus one II contains *all* the operations of the original loop, although the operations that are executed within the same II may belong to different iterations of the loop. The throughput of the software pipelined schedule is $\frac{1}{II}$, as one iteration of the original loop is completed in one II.

The II of a software pipelined loop is bounded from below by the *Minimum Initiation Interval* (MII), which is governed by the following two factors:

- The resource constraints of the architecture on which the software pipelined loop is to be executed, owing to the limited number of functional units or other computational resources. This yields a lower bound on the MII and is called the Resource Constrained Minimum Initiation Interval (ResMII) \[^{[53]}\]. If all the functional units (or processors in a multi-processor system) are homogeneous, and $\text{delay}(v)$ is the execution time of node $v$ in the data flow graph, then ResMII may be formally defined as:

$$\text{ResMII} = \sum_{v \in V} \frac{\text{delay}(v)}{N_{\text{proc}}}$$

Where $V$ is the set of all nodes in the data flow graph and $N_{\text{proc}}$ is the number of processors available. For an architecture with heterogeneous functional units, the ResMII for each type of functional unit is calculated individually in a similar manner and the overall ResMII is simply the maximum of the ResMII over all the functional unit types.

- The recurrences or inter-iteration dependencies in the original loop. This also forms a lower bound on the MII and is called the Recurrence Constrained Minimum Initiation Interval (RecMII) \[^{[53]}\]. RecMII comes into play only if the data flow graph has cycles or recurrences, which indicate inter-iteration dependencies or loop carried dependencies. If such a cycle exists, then the number of tokens on the edges of the cycle determines the distance of the recurrence. In other words it specifies the number of iterations the recurrence is separated by. Such recurrences implicitly constrain the rate at which new iterations may be initiated. RecMII is formally
3.2. A Motivating Example

defined as:

$$\text{RecMII} = \max_{c \in C} \left[ \sum_{v \in c} \text{delay}(v) \over \sum_{(u,v) \in c} \text{dist}(u,v) \right]$$

Where $C$ is the set of all cycles or recurrences in the data flow graph. $\text{delay}(v)$ is the delay of a node $v$ that is a part of cycle $c$ and $\text{dist}(u,v)$ is the dependence distance between two nodes $u$ and $v$ that are a part of cycle $c$.

Thus RecMII and ResMII provide a lower bound on the Minimum Initiation Interval. We now define the MII as:

$$\text{MII} \geq \max(\text{ResMII}, \text{RecMII})$$

While the MII is a lower bound for the II, it does not guarantee that a modulo schedule can be found at the MII. Thus, $\text{II} \geq \text{MII}$.

Consider the StreamIt graph in Figure 3.1(a), constructed using a hierarchical composition of a feedback loop construct which consists of a split-join construct, which in turn consists of two filters B and C at the lowest level of the hierarchy. Further, we assume the splitter to be a duplicate splitter. For clarity, we have simplified the example in Figure 3.1(a) to the graph shown in Figure 3.1(b), where we have collapsed $A_1$ and $A_S$ (the joiner and the splitter nodes) into a single node $A$. The nodes $D_S$ and $D_1$ have also been collapsed into node $D$. Also, assume that each node consumes exactly one element from each of its inputs on each firing and produces exactly one element onto each of its outputs and the delays of each node are as shown in Figure 3.1(b). Assume four homogeneous SMs in the GPU on which we would like to software pipeline the execution of this StreamIt graph. A naïve SIMD execution of this graph is shown in Figure 3.1(c). This execution scheme would require storage for four data values on each edge, since four instances of each filter are fired concurrently. Assuming that the buffers are not shared across edges as suggested in [50], this SIMD execution has a buffer requirement of $4 + 4 + 4 + 4 + 4 = 20$ units.

Let us now consider a software pipelined execution of this example StreamIt graph. As there are four homogeneous processors (SMs) available, computing the ResMII for
Figure 3.1: Motivation for Software Pipelining the Execution of Stream Programs on GPUs
the stream graph in Figure 3.1(b), we get:

$$\text{ResMII} = \left\lceil \frac{8}{4} \right\rceil = 2$$

However, since node A takes 3 time units to execute and since the execution model of the GPU does not support wrap around execution of filters due to the fact that a kernel launches on the GPU are serialized resulting in a barrier of sorts at the end of every kernel invocation, we need to unroll the enclosing loop twice to get a ResMII for 4 for the unrolled loop.

Now to compute the RecMII, we observe that there are two cycles in the stream graph: (i) A-C-D-A and (ii) A-B-D-A, with a dependence distance of 4 between D and A, since there are 4 tokens buffered along the edge from D to A. Computing the ratio of delays to dependence distances along the cycle A-C-D-A, we get:

$$\text{RecMII} \geq \left\lceil \frac{7}{4} \right\rceil = 2$$

and along the cycle A-B-D-A, we get:

$$\text{RecMII} \geq \left\lceil \frac{6}{4} \right\rceil = 2$$

Thus RecMII = 2 and ResMII = 2. Thus the MII = 2 and II \( \geq 2 \). Thus we get a II bounded below by 4 after unrolling the stream graph by a factor of 2, and find a solution at II = 4 as shown in Figure 3.1(d).

We now discuss the buffer requirement for the software pipelined schedule of the StreamIt graph shown in Figure 3.1(b). This execution is shown in Figure 3.1(d). The buffer requirements are: (A,B) : 2, (A,C) : 4, (B,D) : 4, (C,D) : 2 and (D,A) : 4. The buffer requirements on (A,C) and (B,D) are 4 since the 2 elements are produced by nodes A and B per iteration and need to be kept alive until the end next iteration, for nodes C and D to consume them respectively. This is depicted in Figure 3.2 where the subscript on the label of a buffer indicates the instance of the buffer. For example, a label of (A, B)\(_0\) indicates that it is instance 0 of the buffer between nodes A and B. Also, the Steady State Barrier shown in Figure 3.2 serves to separate the different iterations of
the software pipelined kernel and also represents one GPU kernel launch, leading to an implicit barrier between the different SMs, when the kernel completes. It is clear from Figure 3.2 that we require two instances of the buffer on edges (A, C) and (B, D). Thus the total buffer requirement of the software pipelined execution scheme is 16, which is lower than the buffer requirement of 20 for the naïve SIMD execution. The reduction in buffer requirements would be higher for real world StreamIt graphs which would have greater height.

The throughput of the software pipelined scheme is the same as the SIMD execution; i.e., an iteration is completed every 2 time units. Thus a software pipelined execution scheme utilizes the buffers better while providing the same throughput as the SIMD execution. This is an important consideration when the program is to be executed on a GPU, since the GPU has limited amounts of memory. While techniques such as virtualization of GPU memory have been proposed in the Direct3D 10 model [14], to the best of our knowledge, none of the GPUs today support it. Further, if the arc (D,A) had a dependence distance of 2, rather than 4 as shown in Figure 3.1(b), then
the concurrent SIMD execution as shown in Figure 3.1(c) is not possible. This would reduce the throughput of the SIMD execution to $\frac{2}{5} = 0.25$, whereas the software pipelined schedule shown in Figure 3.1(d) would still be valid, achieving a throughput of $\frac{2}{4} = 0.5$.

Thus, since the software pipelining approach has the advantage of supporting better throughput in the presence of strong recurrences and better utilization of the limited memory available on the GPU. Hence we choose this approach for orchestrating the execution of stream programs on the GPU. The rest of this chapter discusses our methodology for compiling StreamIt programs for execution on the GPU.

A recent proposal has formulated the problem of executing StreamIt programs in a software pipelined manner on the CellBE accelerator [40]. However, the approach proposed in [40] cannot be directly adapted for the problem of software pipelining the execution of StreamIt programs on GPUs, owing the GPUs being multi-threaded processing engines. Also, the approach suggested in [40] would incur overhead due to the split-join constructs that it adds in order to achieve the optimal level of data parallelism. Our approach solves this issue by considering the instances of filters as basic schedulable entities, rather than the filters themselves, thus avoiding the use of additional split-joins. Also, the formulation in [40] does not take into account the limited local store memory available on the CellBE, which is comparable to the shared memory on the GPU. The approach is thus oblivious to the buffer requirements of the schedule, while our formulation takes the buffer requirements into consideration and ensures that the schedule generated can be executed with the limited amount of device memory available on the GPU.

3.3 Overview of the Compilation Process

We now provide an overview of the compilation methodology for the compilation of StreamIt programs for execution onto the GPUs. The basic compiler work-flow is as shown in Figure 3.3. The StreamIt program is first compiled as a set of individual filters, each with a driver routine to populate its buffers and fire the filter on the GPU a fixed number of times. This forms the profiling step of the process. The filters are
Chapter 3. Software Pipelining of Stream Programs onto GPUs

profiled with a varying number of threads with appropriate register constraints on them and the timing details are collected for each configuration. Specifically, we profile the program with 128, 256, 384 and 512 threads, constraining the registers to 64, 32, 20 and 16 registers. Thus at the end of the profile runs we would have the data for the execution times of all the filters when executed with all possible combinations of register constraints and the number of threads. This data is then used to select an optimal execution configuration for the entire program using the algorithm described in Section 3.4.

Once the execution configuration has been finalized and the profile timings for filters with the optimal execution configurations are known, we would like to construct the software pipelined schedule for this execution configuration. We formulate this problem as an efficient Integer Linear Program (ILP). A constraint generator generates the set of linear constraints — i.e., the dependence, resource and buffer constraints — that a legal schedule must satisfy. These constraints are solved by the CPLEX solver, which produces a schedule, which is then used by the final code generator to produce code for the entire stream program targeting the GPU. The StreamIt compiler is a source-to-source compiler and produces code for the CUDA programming model. This code is then compiled by the CUDA compiler, nvcc, to produce the final executable. The following sections in this chapter describe each of the stages in the work-flow for compiling a StreamIt program for execution on the GPU in detail.

3.4 Profiling and Execution Configuration Selection

Profiling the individual filters of a StreamIt program forms the first stage of the compilation process. We further divide the profiling stage into two sub-stages: (i) Actual profiling and (ii) Execution configuration selection process. The timing data collected from the profile runs serve to drive the execution configuration selection process.

3.4.1 Profiling

The profile runs serve two purposes: (i) They give an estimate of the execution time of each filter, and (ii) They help in evaluating the different execution configurations...
possible for each filter. The execution time of a program varies with the number of threads as shown in Table 2.1 in Section 2.2.3. It is possible for different filters in the stream graph to have a different number of threads such that they execute optimally. However, there is an interplay between the number of threads and the register requirements. It might not be possible to execute some filters with a higher number of threads, since the register file may not be able to support such a large number of threads. For example, consider a filter where each thread would require 32 registers. Then we may run the filter with at most 256 threads, since \( \frac{8192}{32} = 256 \). Thus the execution configuration with 128 and 256 threads can be explored. Alternatively we may choose to compile the kernel with a constraint on the register usage, in which case the compiler inserts the necessary spill–fill code. For example, if we constrain the register usage to 20 registers per thread, then configurations having up to 384 threads can be considered.

Algorithm 1 describes the methodology used in our profiling runs. We generate four versions of the executable for each filter, restricting the number of registers to 16, 20, 32 and 64 per thread. These correspond to the register requirements that will allow the kernel to run with 512, 384, 256 and 128 threads respectively, rounded up to the nearest multiple of 4. Note that if the per-thread register requirement of a kernel is less than the limit specified, it has no effect on the compilation of the kernel. Next, we execute each of these four versions with 128, 256, 384 and 512 threads, such that each
Chapter 3. Software Pipelining of Stream Programs onto GPUs

Algorithm 1 run_profile()

1: for i ← 0, · · · , NumFilters − 1 do
2:     for numRegs ← 16, 20, 32, 64 do
3:         for numThreads ← 128, 256, 384, 512 do
4:             execute the version of the kernel compiled for numRegs registers with numThreads threads for \( \frac{\text{numFirings}}{\text{numThreads}} \) iterations.
5:             if kernel fails to execute due to lack of registers then
6:                 runTimes[i][numRegs][numThreads] ← ∞
7:             else
8:                 record the time taken in runTimes[i][numRegs][numThreads]
9:             end if
10:         end for
11:     end for
12: end for

 execution performs the same amount of work, irrespective of the execution configuration. We accomplish this by using a parameter called numFirings, which represents the number of single threaded firings of the filter to be executed. This parameter is set to be a multiple of 128, 256, 384 and 512, and is also set to be large enough, to amortize the cost of launching the kernel on the GPU over many iterations. For the chosen execution configuration and the version of the executable, if the number of registers required per thread is greater than the available number of registers, then the kernel execution fails and the configuration is not feasible. For example if a filter has been compiled with a register constraint of 32 registers per thread and it actually utilizes the entire set of 32 registers per thread, then it is not possible to execute the filter with more than 256 threads, since \( \frac{8192}{32} = 256 \). For all the other feasible configurations we obtain the execution time on the GPU for all the filters in the StreamIt program from these profile runs.

3.4.2 Execution Configuration Selection

Once the execution times for all the filters in the StreamIt graph over all the feasible execution configurations are estimated, the configuration selection phase described in Algorithm 2 determines the optimal execution configurations for each filter such that it minimizes the II globally for the entire program. The set feasiblePairs used in line 2 of Algorithm 2 consists of all pairs (numRegs, numThreads), such that the configuration
### 3.4. Profiling and Execution Configuration Selection

#### Algorithm 2 find_best_config()

1. minII = ∞
2. for (numRegs, numThreads) ∈ feasiblePairs do
   3. for i ← 0, ⋯, numFilters − 1 do
      4. find k < numThreads such that runTimes[i][numRegs][k] is minimized
      5. tempthr[i] ← numThreads
   6. end for
7. Solve the steady state equations for the execution configuration
8. present in tempthr and determine the number of instances of each filter,
9. storing it in the array numInstances.
10. curII ← 0
11. for i ← 0, ⋯, NumFilters − 1 do
12.   bestTime ← runTimes[i][numRegs][k], where, k < numThreads is chosen such that runTimes[i][numRegs][k] is minimized
13.   bestTime ← bestTime × numInstances[i]
14.   curII ← curII + bestTime × (k/numfirings)
15. end for
16. end if
17. Estimate total work w done in one II by the current execution configuration
18. curII ← curII/w
19. if curII < minII then
20.   minII ← curII
21.   maxThreads ← numThreads
22.   maxRegs ← numRegs
23. for i ← 0, ⋯, NumFilters − 1 do
24.   threads[i] ← k, where, k < numThreads is chosen such that runTimes[i][numRegs][k] is minimized
25.   delay[i] ← runTimes[i][numRegs][threads[i]]
26. end for
27. end if
28. end for

is a feasible configuration for all the filters. The necessity for this condition is explained as follows: The CUDA compiler currently does not support extern functions that can be called from a function executing on the device. This restriction imposes the code for all the filters is compiled as a single compilation unit, as calls to these filter work functions will be made by the software pipelined kernel, executing on the device. Thus all the filters need to be compiled with the same register usage restrictions.

Lines 3–7 take into account that a filter executing with numThreads threads will produce numThreads times as many outputs and will consume numThreads as many inputs as a single threaded version, and recomputes the number of instances of each filter that appears in the steady state schedule. Lines 9–13, calculate the candidate
initiation interval (II) that will be achieved if the current configuration is chosen. The execution time is scaled in Line 12 to account for the fact that the profile run executes \((\text{numfirings}/k)\) iterations of the filter, while we require the execution time for only one iteration of the filter. Lines 14 – 15 scale the II by the amount of work done. This is necessary, since the work done by the one steady state execution of the various configurations may not be the same. For example, a particular configuration might yield an II of 20 time units, and produce 200 output tokens\(^1\), while another might have a higher II of 40, but produce 1000 tokens. Clearly, the latter is a better alternative. The rest of the algorithm, picks the best number of threads for each filter such that the resource constrained II is minimized, and saves the optimal number of threads for each filter in an array called \(\text{threads}\).

Note that we cannot have a varying number of registers across different filters, since in the final code generated, all the filters need to be compiled as one compilation unit and the CUDA framework currently does not currently support \texttt{extern} calls for device functions. Thus the \textit{entire} StreamIt program, including the code for all the filters would need to compiled as one compilation unit. Future versions of the CUDA framework might allow \texttt{extern} calls for device functions, in which case we may easily accommodate this by modifying the execution configuration selection and the final code generation process to allow for this added flexibility.

For the optimal execution configuration selected as described above, the firing rates of the filters in a primitive steady state schedule is different from the corresponding firing rates in the primitive steady state schedule in the original StreamIt program. This is because the push and pop rates of filters are different for the optimal configuration, owing to the fact that the push and pop rate of the filter executing on the GPU is the sum of the push and pop rates of its threads, which is simply the base push and pop rates multiplied by the number of threads. For example, consider a filter that has a base push rate of 2. If the execution configuration selected for that filter by Algorithm\(^2\) consists of 256 threads, then one firing of the filter on the GPU (with 256 threads) would

\[^1\]We represent the amount of work done by an execution configuration in terms of the number of tokens produced or consumed by any filter in the StreamIt graph. Typically, we take the number of tokens popped by a sink filter in one steady state execution as a measure of the amount of work done by an execution configuration.
push $2 \times 256 = 512$ elements, for a push rate of 512. In order to avoid confusion, we refer to the input and output rates of the original filter when it is executed with one thread as the base pop rate and the base push rate respectively. This corresponds to the original push and pop rates of that filter. The push and pop rates of a filter executing on the GPU with a specified number of threads will be referred to simply as the multi-threaded push and multi-threaded pop rate of that filter from this point on. Also, unless otherwise specified the push and pop rates always refer to the base push and pop rates of a filter.

### 3.4.3 Execution Configuration Selection: Putting it all Together

We now illustrate the execution configuration selection algorithm with a simple example. Consider the StreamIt graph shown in Figure 3.4(a), which consists of a simple pipeline with 2 filters A and B. The base push rate of A is 2 and the base pop rate of B is 1 as shown in Figure 3.4(a). The original dependence graph for this StreamIt graph is shown in Figure 3.4(b), where the two elements pushed by one instance (or firing) of filter A is consumed by two instances of filter B. For simplicity, we assume that filter B
is to be executed with 128 threads always, while filter A can be executed with 128 or 256 threads. Figure 3.4(c) shows the StreamIt graph with the *multi-threaded* push and pop rates if both A and B are to be executed with 128 threads. The execution times for both the filters in this configuration are also shown in Figure 3.4(c). Thus when the steady state equations are solved in line 7 of Algorithm 2, we determine that two instances of filter B executing with 128 threads and one instance of filter A executing with 128 threads form a steady state. Figure 3.4(d) shows the dependence graph corresponding to this configuration. Note that the schedulable entities on the GPU are the *large ovals* shown in Figure 3.4(d). Applying the definition of ResMII to the execution configuration shown in Figure 3.4(d) yields a lower bound on the II as \( \lceil \frac{32+2 \times 16}{2} \rceil = 32 \).

Assuming that a schedule is possible at this lower bound, if we let the number of elements consumed by filter B to represent the work, then the work done in each II is 128.

Now, we consider the execution configuration where A is to be executed with 256 threads and B is to be executed with 128 threads. This is shown in Figure 3.5(c). Note that the execution time when filter B is executed with 256 threads has not doubled.
Solving the steady state equations yields that one instance of filter A executing with 256 threads and four instances of filter B executing with 128 threads are necessary to maintain a steady state. The corresponding dependence graph is shown in Figure 3.5(d). The lower bound on the II in this case is \( \left\lceil \frac{40+16 \times 4}{2} \right\rceil = 52 \). This is larger than the lower bound obtained for the execution configuration shown in Figure 3.4(c). However, assuming that a schedule is possible at the lower bound, the work done in this case is double the work done in the earlier execution configuration. Normalizing the II by the amount of work done, we find that executing filter A with 256 threads and filter B with 128 threads allows us to perform more work per unit time. Hence it is beneficial to choose this execution configuration.

### 3.5 ILP Formulation

We now describe the ILP formulation to derive the schedule for the optimal execution configuration identified according to the procedure described in Section 3.4. We begin by defining the terms used in the formulation and then develop the resource, dependence and the buffer constraints in the formulation. Note that in our formulation we follow the convention of denoting the names of constants with all upper case letters, while names of ILP variables are denoted in all lower case letters.

#### 3.5.1 Definitions

- \( V \) denotes the set of all nodes (filters) in the stream graph
- \( N \) is the cardinality of \( V \), i.e., the number of nodes in the stream graph
- \( E \) denotes the set of all (directed) edges in the stream graph
- \( K_v \) represents the number of instances (or firings) of a filter \( v \in V \), as dictated by the steady state rate equations. Note that these are not the firing rates for the original StreamIt graph with the base push / pop rates; instead they are the firing rates for the multi-threaded filters to be actually executed on the GPU. These are calculated as in line 7 of Algorithm 2 for the chosen execution configuration and are constant as far as the ILP formulation is concerned.
P is the set of all multiprocessors in the GPU, denoted \( \{0, 1, \ldots, P_{\text{max}} - 1\} \).

T represents the Initiation Interval (II) of the software pipelined loop. Again, this is a constant as far as the ILP formulation is concerned.

\( D(v) \) is the delay or execution time of filter \( v \in V \), which is a constant.

\( I_{uv} \) represents the number of elements consumed by filter \( v \) on each firing of \( v \), for an edge \( (u, v) \in E \) i.e., the multi-threaded pop rate of the filter.

\( O_{uv} \) denotes the number of elements produced by filter \( u \), on each firing of \( u \), for an edge \( (u, v) \in E \). Again, this is the multi-threaded push rate of the filter.

\( M_{uv} \) denotes the number of elements initially present on the edge \( (u, v) \in E \).

### 3.5.2 Resource Constraints

First we model the resource constraints for the ILP formulation of the software pipelining formulation. The instances of the filters need to be software pipelined across the SMs of the GPU while taking into account the resource constraints of the limited number of SMs, the dependence constraints of the stream graph and the buffer constraints of the limited device memory available.

We use the linear form of a software pipelined schedule given in [28] as:

\[
\sigma(j, k, v) = T \times j + a_{k,v}
\]

where \( \sigma(j, k, v) \) represents the time at which the execution of the \( k^{\text{th}} \) instance of filter \( v \) in the \( j^{\text{th}} \) steady state iteration of the stream graph is scheduled. \( a_{k,v} \quad \forall k \in [0, K_v), \quad \forall v \in V \) are integer variables and \( a_{k,v} \geq 0 \). We may write each \( a_{k,v} \) as

\[
a_{k,v} = T \times \left\lfloor \frac{a_{k,v}}{T} \right\rfloor + a_{k,v} \mod T
\]

We define \( f_{k,v} = \left\lfloor \frac{a_{k,v}}{T} \right\rfloor \) and \( o_{k,v} = a_{k,v} \mod T \). Thus, the linear form of the schedule can be written as:

\[
\sigma(j, k, v) = T \times (j + f_{k,v}) + o_{k,v}
\]
In the equation above, \( o_{k,v} \) indicates the time instant in the software pipelined kernel when the \( k^{th} \) instance of a filter \( v \) is scheduled to execute. And the \( f_{k,v} \) serve to set up the iteration numbers of the instances of the filters, in the software pipelined schedule. \((f_{k,v} - f_{k',v'})\) denotes the number of iterations that the instance \( k \) of filter \( v \) and the instance \( k' \) of filter \( v' \) are separated by. Note that \( f_{k,v} \) are integer variables greater than or equal to 0.

We define 0–1 integer variables \( w_{k,v,p} \), \( \forall k \in [0,K_v) \), \( \forall v \in [0,N) \), \( \forall p \in [0,P_{\text{max}}) \), such that, \( w_{k,v,p} = 1 \) implies that the \( k^{th} \) instance of the filter \( v \) has been assigned to multiprocessor \( p \). We now model various constraints as follows.

The following constraint ensures that each instance of each filter is assigned to exactly one SM on the GPU.

\[
\sum_{p=0}^{P_{\text{max}}-1} w_{k,v,p} = 1, \quad \forall k \in [0,K_v), \quad \forall v \in V \tag{3.2}
\]

We would need to ensure that the work assigned to each SM can be completed by it within the given II. We model this with the following constraint as:

\[
\sum_{k=0}^{K_v-1} \sum_{v \in V} w_{k,v,p} \times D(v) \leq T, \quad \forall p \in P \tag{3.3}
\]

We also need to ensure that the execution of any instance of any filter completes within the II and does not wrap around into the next II. This is because each II will be issued as a separate GPU kernel, hence each node must complete within the stipulated II. Note that if the execution time of any instance of any filter exceeds the stipulated II, which is the ResMII for the program, then this constraint would make it impossible find a schedule at an II less than the execution time of one instance of that filter. We deal with this issue by preconditioning the number of firings, i.e., the multiplicity of each filter, by multiplying it by an integer, such that the execution time of any one instance of any filter does not dominate the lower bound on the II. Conceptually, this is similar to unrolling the stream graph an integral number of times, so that the amount of work performed in each iteration is increased an integral number of times, while the delay of the heaviest filter that was dominating the II remains the same.
In order to ensure that the execution of any instance of any filter completes within the stipulated II, we once again note that in Equation (3.1) the variable $o_{k,v}$ indicates the time instant in the software pipelined kernel when the $k^{th}$ instance of a filter $v$ is scheduled to fire. We now constrain the starting time of filters $o_{k,v}$.

$$o_{k,v} + D(v) < T, \quad \forall v \in V, \quad \forall k \in [0, K_v)$$

(3.4)

to ensure that every firing of a filter is scheduled so that it can complete within the same kernel or II and thus prevents the execution of a filter from wrapping around into the next II.

### 3.5.3 Dependence Constraints

We now model the dependence constraints. In order to ensure that the firing rule is satisfied by a schedule, the admissibility of a schedule is given in [27], as below:

$$\sigma(i,v) \geq \sigma\left(\left[\frac{(i + 1) \times \text{I}_{uv} - \text{M}_{uv} - \text{O}_{uv}}{\text{O}_{uv}}\right], u\right) + D(u), \quad \forall (u,v) \in E, \quad \forall i \geq 0$$

(3.5)

where $\sigma(i,v)$ denotes the time of the $i^{th}$ execution of filter $u$. This constraint essentially ensures that there are a sufficient number of inputs buffered on each edge $(u,v)$ in the stream graph before the execution of filter $v$ is scheduled.

This admissibility condition makes an implicit assumption about the order of firing of different instances of filter $u$. i.e. It assumes that the firings the instances of filter $u$ are serialized. Thus, a condition that ensures that the instance of a filter $u$, which produces the last quantum of data to be consumed by the current instance of filter $v$ has completed execution before beginning execution of the current instance of filter $v$ is sufficient. However this assumption does not hold for the model we are building, where instances of each filter could execute out of order, or in parallel across different SMs of the GPU, as long as the firing rule is satisfied. So we must ensure that the schedule is admissible with respect to all the predecessors of the $k^{th}$ instance of the filter $v$. Figure 3.6 describes this scenario, where a filter A pushes two elements and filter B pops three elements on each firing. Figure 3.6(b) shows the dependencies that exist...
3.5. ILP Formulation

Figure 3.6: (a) A multi-rate stream graph. Filter A pushes 2 tokens on each firing and filter B pops 3 tokens on each firing. (b) The dependency graph on each instance of each filter

between the various instances of filter A and B. The label represents the filter and the subscript represents the instance number, which can range from 0 to \((K_v - 1)\) for a given filter v. As can be seen from Figure 3.6, it is not enough for \(B_0\) to wait for completion of \(A_1\). In order for \(B_0\) to fire, both \(A_0\) and \(A_1\) need to have completed execution. To model this behavior, we need to be able to analytically determine which instances of a filter u, produce data that is consumed by the \(i^{th}\) instance of a filter v, for each edge \((u, v) \in E\).

Any instance of a filter v depends on at most \(\left\lceil \frac{I_{uv}}{O_{uv}} \right\rceil + 1\) consecutive instances of filter u, \(\forall (u, v) \in E\). Intuitively, the \(i^{th}\) instance of a filter v must wait for all the \(I_{uv}\) tokens produced after the \((i - 1)^{th}\) instance has fired. We model this formally as:

\[
\sigma(i, v) \geq \sigma\left(\left\lceil \frac{i \times I_{uv} + 1 - M_{uv} - O_{uv}}{O_{uv}} \right\rceil, u\right) + D(u), \quad \forall l \in [1, I_{uv}], \quad \forall (u, v) \in E
\]

(3.6)

Note that although it appears that there are \(I_{uv}\) constraints for each edge, there are in fact at most \(\left\lceil \frac{I_{uv}}{O_{uv}} \right\rceil + 1\) constraints, since some constraints end up being repeated.

We now derive the dependency constraints that in turn determine the constraints on \(f_{k,v}\). Consider the \(k^{th}\) instance of filter v, in the \(j^{th}\) steady state iteration. Since there are \(K_v\) instances of the filter v in one steady state iteration of the stream graph, using (3.6), we get:

\[
\sigma(j, k, v) \geq \sigma\left(\left\lceil \frac{(j \times K_v + k) \times I_{uv} + 1 - M_{uv} - O_{uv}}{O_{uv}} \right\rceil, u\right) + D(u), \quad \forall l \in [1, I_{uv}] (3.7)
\]

Since \(I_{uv} \times K_v = O_{uv} \times K_u\) according to the balanced rate equations for multi-rate...
stream graphs [27], we can write:

\[ \sigma(j, k, v) \geq \sigma \left( j + K_u \left\lfloor \frac{k \times I_{uv} + 1 - M_{uv} - O_{uv}}{O_{uv}} \right\rfloor \right) + D(u), \quad \forall l \in [1, I_{uv}] \] (3.8)

Simplifying further, we get:

\[ \sigma(j, k, v) \geq \sigma(j_l', k_l', u) + D(u), \quad \forall l \in [1, I_{uv}] \] (3.9)

where:

\[ j_l' = j + j_{lag,l} = j + \left\lfloor \frac{1}{K_u} \left\lfloor \frac{k \times I_{uv} + 1 - M_{uv} - O_{uv}}{O_{uv}} \right\rfloor \right\rfloor, \quad \forall l \in [1, I_{uv}] \]

\[ k_l' = \left\lfloor \frac{k \times I_{uv} + 1 - M_{uv} - O_{uv}}{O_{uv}} \right\rfloor \mod K_u, \quad \forall l \in [1, I_{uv}] \]

Note that, \( k \in [0, K_v) \implies j_{lag,l} \leq 0, \quad \forall l \in [1, I_{uv}] \). Now, using the form in (3.9), we get:

\[ T \times (j + f_{k,v}) + o_{k,v} \geq T \times (j + j_{lag,l} + f_{k_l',u}) + o_{k_l',u} + D(u), \quad \forall l \in [1, I_{uv}] \]

Algebraic simplifications yield:

\[ T \times (f_{k,v} - f_{k_l',u}) + (o_{k,v} - o_{k_l',u}) \geq T \times j_{lag,l} + D(u), \quad \forall l \in [1, I_{uv}] \] (3.10)

This forms a system of constraints with at most \( \left\lfloor \frac{I_{uv}}{O_{uv}} \right\rfloor + 1 \) constraints for each \((u, v) \in E\).

So far, we have assumed that the result of executing a filter \( u \) is available \( D(u) \) time units after the filter \( u \) has started executing. However, the limitations of a GPU imply that this may not always be the case. More specifically, if the results are required from a producer instance of filter \( u \), which is scheduled on a different SM, than where the consumer instance is scheduled, then the data produced can be reliably accessed only in the next steady state iteration, since there is no way to determine if a particular section of code on a different SM has completed execution before executing another section of code on a given SM. Thus, to model whether the \( t^{th} \) instance of \( u \) and the \( k^{th} \) instance of filter \( v \) on an edge \((u, v) \in E\) are scheduled on the same SM, we define
0–1 integer variables  

\[ g_{l,k,u,v}, \quad \forall k \in [0, K_v), \quad \forall (u,v) \in E, \quad \forall l \in [1, I_{uv}] \]

and apply the following constraints on them:

\[ g_{l,k,u,v} \geq w_{k,v,p} - w_{k',u,p} \quad \text{and} \quad g_{l,k,u,v} \geq w_{k',u,p} - w_{k,v,p} \quad (3.11) \]

Note that \( g_{l,k,u,v} \) is equal to \( |w_{k,v,p} - w_{k',u,p}| \), \( \forall k \in [0, K_v), \quad \forall (u,v) \in E, \quad \forall l \in [1, I_{uv}] \). Equation (3.11) must be satisfied for all \( p \in [0, P_{max}) \). Also, \( g_{l,k,u,v} = 1 \) implies that for an edge \((u,v) \in E\), the instance of the filter \( u \) that produces the \( l \)th data element consumed by the \( k \)th instance of a filter \( v \) is scheduled on a different SM than the \( k \)th instance of the filter \( v \), thus requiring that we wait until the next II before scheduling the \( k \)th instance of the filter \( v \). We stress that the subscript \( l \) in the variables \( g_{l,k,u,v} \) does not correspond to the instance number of either \( u \) or \( v \), rather, it corresponds to the \( l \)th data element consumed by the \( k \)th instance of the filter \( v \). Incorporating the additional constraint mentioned above in equation 3.10 upon algebraic simplifications, we get two constraints:

\[ T \times f_{k,v} + o_{k,v} \geq T \times (j_{lag,l} + f_{k',u}) + o_{k',u} + D(u) \]

\[ T \times f_{k,v} + o_{k,v} \geq T \times (j_{lag,l} + f_{k',u} + g_{l,k,u,v}) \quad (3.12) \]

\[ \forall (u,v) \in E, \quad \forall k \in [0, K_v), \quad \forall l \in [1, I_{uvv}] \]

The two constraints are set up such that the former comes into play when \( g_{l,k,u,v} = 0 \) and the latter comes into play when \( g_{l,k,u,v} = 1 \), since \( o_{k',u} + D(u) \leq T \) at all times, from Equation (3.3). The idea is that if any predecessor of the instance \( k \) of a filter \( v \) is scheduled on a different multiprocessor that \( v \), then \( g_{l,k,u,v} = 1 \) for some \( l \).

Note that Equation (3.12) is independent of the iteration number \( j \), and forms a system of at most \( \left\lceil \frac{P_{max}}{O_{uv}} \right\rceil + 1 \) constraints for each edge \((u,v) \in E\).

### 3.5.4 Buffer Constraints

We now model the buffer constraints of the scheduling problem. Consider the buffer requirements of an edge \((u,v) \) in the stream graph. To estimate this, we begin with the live ranges of the values (tokens) produced by filter \( u \). We use a simple buffer allocation
policy where the buffers are allocated once before a particular steady state iteration is begun. The buffers continue to remain allocated until the steady state iteration is complete. i.e. the buffer allocation will not be changed during the steady state iteration. In other words, a buffer that has been allocated at the beginning of the iteration, will remain allocated until the end of the iteration, regardless of when in the iteration, the buffer has been consumed. This model also implies that buffers are not shared across edges in the StreamIt graph as suggested in [50]. A similar model has been used in [25, 26, 40].

Consider the values (tokens) produced by the \( k \)th firing of filter \( u \) in the software pipelined loop. These will be used by \( k' \)th instances of filter \( v \), where \( k' \in [\hat{k}_{\text{start}}, \hat{k}_{\text{end}}] \), and

\[
\hat{k}_{\text{start}} = \left\lceil \frac{k \times O_{uv} + 1 + m_{uv} - I_{uv}}{I_{uv}} \right\rceil \mod K_v
\]

\[
\hat{k}_{\text{end}} = \left\lceil \frac{(k + 1) \times O_{uv} + m_{uv} - I_{uv}}{I_{uv}} \right\rceil \mod K_v
\]

If we wish to express these instances as instances within a particular iteration, we can write the instances of filter \( v \) that depend on values produced by the \( k \)th firing of filter \( u \) in iteration \( j \) (i.e. \( 0 \leq k < K_u \)) as:

\[
k'_r = \left\lceil \frac{k \times O_{uv} + r + M_{uv} - I_{uv}}{I_{uv}} \right\rceil \mod K_v
\]

\[
\hat{j}_{\text{lead},r} = \left\lfloor \frac{1}{k_v} \left\lceil \frac{k \times O_{uv} + r + M_{uv} - I_{uv}}{I_{uv}} \right\rceil \right\rfloor , \quad \forall r \in [1, O_{uv}]
\]

Which is to say that the instance numbers \( k'_r \) of the filter \( v \) from iterations \( j + \hat{j}_{\text{lead},r} \) depend on the values produced by the \( k \)th instance of filter \( u \) in iteration \( j \). Thus these values must be preserved at least for a time interval given by \((\max_r(\hat{j}_{\text{lead},r}) + 1) \times T\). During this time, there will be \( O_{uv} \times K_u \) tokens added to the buffer for every \( I_{uv} \) tokens. Also, the iteration distances between various filters have been defined by the \( f_{k,v} \) in Equation (3.1). Thus, we may write the buffer requirements as:

\[
b_{uv} \geq O_{uv} \times K_u \times (f_{k'_r} - f_{k,u} + 1), \quad \forall k \in [0, K_u), \quad \forall (u, v) \in E, \quad \forall r \in [1, O_{uv}] \quad (3.13)
\]
Thus, we can now constrain the total buffer requirement of the software pipelined schedule as:

$$\sum_{(u,v) \in E} b_{uv} \leq B_{\text{max}}$$  \hspace{1cm} (3.14)$$

The value of $B_{\text{max}}$ can be set to the amount of device memory available on the GPU, in which case the problem becomes a constraint formulation. Alternatively, it is possible to minimize the objective function of $\sum_{(u,v) \in E} b_{uv}$ to achieve the schedule which has the minimum buffer requirement, in which case the problem becomes an ILP optimization problem rather than merely a constraint problem. We only discuss the constraint form of the problem in our work since the memory available on the current GPUs is quite high (512MB – 4096MB). Thus we have presented a linear constrained framework for software pipelining of a stream graph on to a GPU.

### 3.6 Code Generation

The constraints for the software pipelining problem are been generated for optimal execution configurations obtained from the profiling and optimal configuration selection phase. Once the constraints have been generated, we then use CPLEX [1], and industrial strength ILP solver to solve the ILP formulation. Since our ILP formulation is more of a constraint problem, rather than an optimization problem, the solution time for solving the ILP formulation is lower than if it were an optimization problem.

The solution to the ILP gives the schedule of the StreamIt program on the SMs of the GPU. The schedule may require that different SMs execute different filters. However, although all the multiprocessors are required to have a single kernel entry point, the individual multiprocessors may diverge with no performance penalty. We utilize this fact to generate code that allows different SMs to execute code corresponding to different filters. Once the $w_{k,v,p}$, $o_{k,v}$, and the $f_{k,v}$ have been determined from the solution of the ILP, we generate a single software pipeline kernel, with the parts required to execute on individual multiprocessors separated out using a `switch` statement. Within each multiprocessor, the instances of filters are orchestrated to execute in increasing order of the $o_{k,v}$ variables corresponding to them. When $o_{k,v}$ values of two filters are
equal it implies that the two instances of filters are not related by a producer–consumer relationship within the same II or software pipelined kernel. i.e., even if there is a producer–consumer relationship between two filters that have the same \( o_{k,v} \) value, the constraint in Equation (3.12) ensures that they are separated by at least one II by maintaining a difference of at least one between the \( f_{k,v} \) values of the filters in consideration. We use the predicated kernel-only code generation schema described in [54], within each case of the switch statement, with the staging predicate implemented as an array, similar to the scheme described in [40].

Figure 3.7 shows the skeleton of the code generated for the example software pipelined loop shown in Figure 3.1(d), which has been software pipelined assuming four SMs and thus consists of four blocks, one for each SM. The \texttt{__device__} variable \texttt{__stages_gpu} is an array that contains the staging predicated and is used in the software pipelined kernel to selectively turn stages on or off as described in [54]. It is set by the CPU before each kernel invocation to the appropriate values to reflect whether the software pipelined loop is executing in the prologue, steady or epilogue states.

The kernel itself consists of a switch statement, indexed by the \texttt{blockIdx} variable, which enables different blocks (which are executing across different SMs of the GPU) to follow different execution paths which correspond to the work that has been allocated to them. The iteration number is passed to each of the work functions in the software pipelined loop so that the work functions of the filters can perform the required rotation of the buffers depending on the iteration. The second argument to the work functions is the instance number of the filter, which facilitates the respective work functions to write or read from the appropriate offsets in their buffers. This software pipelined kernel which is executed by the GPU is repeatedly called by the CPU (host) control thread, after adjusting the staging predicate appropriately whenever necessary.

3.7 Buffer Layout Optimizations

As described in Section 2.1, accesses to device memory need to be coalesced in order to effectively utilize the high bandwidth that the device memory is capable of providing. We illustrate the problem with an example. Figure 3.8(a) shows the execution of a
filter which has a pop rate of 4, executing with 4 threads on a device with memory organized as 8 banks. The buffer is organized sequentially, in the natural FIFO ordering of elements. Thus thread₀ pops the first four elements, thread₁ the next four and so on. But the accesses by thread₀ and thread₂, which occur at time instant t₀, both hit bank B₀, causing these accesses to be serialized. Similar collisions occur with respect to thread₁ and thread₃ at time t₀. Clearly, as can be seen in Figure 3.8(b), which shows the conflicts at time t₁, these collisions occur at every memory access, reducing
the utilization of the memory bus and degrading performance. The problem is not restricted to memory reads alone, as writes also suffer from the same conflict pattern, further degrading performance.

The problem can be alleviated by streaming the entire working set of all the threads into the shared memory in each multiprocessor. The shared memory is also organized as banks, and collisions still degrade performance on shared memory, but the degradation is small, since the shared memory has an access latency of 1 cycle as compared to the 600 – 800 cycle latency of the global memory. However, this approach does not scale well for filters with large working sets. For example, a filter that pushes 64 int s and pops the same number of int s at each firing, can then be executed with a maximum of 32 threads, since each multiprocessor has only 16KB of shared memory. The low number of threads still causes performance degradation due to the memory latency that has now been exposed due to low levels of multi-threading. Clearly, the buffer layout
3.7. Buffer Layout Optimizations

A layout that causes no Bank Conflicts at time $t_0$:

(a) A Layout that causes no Bank Conflicts at time $t_0$

A layout that causes no Bank Conflicts at time $t_1$:

(b) A Layout that causes no Bank Conflicts at time $t_1$

Figure 3.9: A Conflict-free Data Layout

scheme needs to be chosen carefully to minimize bank conflict.

We leverage the fact that the production rate is equal to the consumption rates on a given channel between two filters when considered across one entire steady state, in a steady state schedule, to optimize the buffer layout. Consider the example program shown in Figure 3.8. The conceptual idea is to lay the buffers out so that the data elements that are accessed together by the threads of the program executing in lock-step (i.e., in a SIMD fashion) are placed in a contiguous region of memory. This idea is depicted in Figure 3.9 for the same program as that shown in Figure 3.8. As can be seen in Figure 3.9(a), all threads access distinct banks avoiding any bank conflict at time $t_0$. Similarly, for times $t_1$ and above, no bank conflicts occur as shown in Figure 3.9(b).

While the example in Figure 3.9 shows a simplified view of our buffer layout scheme, we now present a more complete example. Consider the stream graph shown in Figure 3.10. Filter A has base push and pop rates of 2 and filter B has base push and pop rates of 1. Also, filter A is to be executed with 256 threads, while filter B is
to be executed with 128 threads. So there would be 1 (multi-threaded) instance of A in the steady state schedule and 4 (multi-threaded) instances of B. The data elements are labeled in Figure 3.10 with the superscript indicating which filter the data elements is produced or consumed by as well as whether the date is produced or consumed by that filter, while the subscript indicates the position of the data element in the natural FIFO ordering. For instance a label of $d_{1}^{A_o}$ indicates that the data element is the element number one in the FIFO ordering and is popped by filter A. Similarly a label of $d_{128}^{B_u}$ indicates that it is the element number 128 in the FIFO ordering and is pushed by filter B.

We organize the buffers such that the first 128 elements of the buffer contains the first popped elements for each of the 128 threads, as shown in Figure 3.10, where we show the layout of each of the buffers in the stream graph, and what threads produce and consume into each of the positions in the buffer. We group threads into clusters of 128 threads, since this is the gcd of the thread block sizes that we consider. Each thread of each cluster of 128 threads, pops or pushes the first token it consumes or produces from contiguous locations in memory. In this way, it is guaranteed that no bank conflicts can occur between accesses of threads in the same block. Note that it is not necessary for the groups of 128 accesses to be coalesced. In fact it is enough if accesses by a warp which consists of 32 threads to be coalesced. However, we chose to coalesce the accesses of 128 threads, since it is the GCD of 128, 256, 384 and 512, which are the block sizes that we consider in this work.

It is sufficient if the very first input buffer to the stream graph is shuffled, since after that, we can modify the push() and pop() routines to ensure that the data values in the buffers internal to the stream graph are communicated in a consistent manner. Formally, we define the shuffle function in terms of the number of tokens pushed or popped in one entire steady state execution of the stream graph as follows

$$s(i) = b\left(\left\lfloor \frac{i}{128} \right\rfloor + (i \mod 128) \times \left( o \times k \times \frac{n_t}{128} \right)\right)$$

(3.15)

where $o$ is the base pop rate (or the base push rate) of the consumer (or producer) of the edge under consideration, $k$ is the firing rate of the multi-threaded consumer (or
Buffer Layout Optimizations

In the steady state schedule, \( n_t \) is the number of threads the consumer (or the producer) is to be executed with. \( s(i), \forall i \in [0, n_t \times o \times k) \) denotes the value value at the \( i^{th} \) position of the shuffled buffer, while \( b(i) \) denotes the value at the \( i^{th} \) position of the serial or FIFO ordering of the same buffer.

The index \( j \) of the \( n^{th} \) element popped by a thread whose thread index is \( tid \) in the \( k^{th} \) instance of a filter with pop rate \( u \) is given by:

\[
j = 128 \times n + \left\lfloor \frac{tid}{128} \right\rfloor \times 128 \times o + (tid \mod 128), \quad n < o \tag{3.16}\]

Similarly the index \( k \) of the \( m^{th} \) element pushed by a thread whose thread index is \( tid \) is given by:

\[
...
in the $k^{th}$ instance of a filter with push rate $u$ is given by:

$$k = 128 \times m + \left\lfloor \frac{\text{tid}}{128} \right\rfloor \times 128 \times u + (\text{tid} \mod 128), \ m < u$$  \hspace{1cm} (3.17)

The first term in both equations sets up the offset for the push or pop. The second term
serves to organize the pushes into groups of 128, and the final term indicates exactly
where in the current block of 128, the element needs to be pushed to or popped from.

With this buffer layout scheme, we totally avoid all bank conflicts and obviate the
need to use shared memory. Further, the efficiency of the scheme is oblivious to the
push and pop rates of the individual filters.

Before we conclude the discussion on the optimized buffer layout, we note that
the buffer layout optimization is not applicable for filters that peek past the elements
that they actually consume, since the residual elements on the channel pose a problem.
For such filters, we first stage the entire working set into the shared memory on each
SM using a sequence of coalesced reads and then operate on the buffer, localizing all
the bank conflicts on the shared memory, which are 1-cycle conflict [3], rather than the
600 – 800 cycle conflicts at the device memory. This approach is scalable to peeking
filters which have a larger working set than permitted by the 16KB of shared memory.
Fortunately, all the benchmarks evaluated have peeking filters whose working sets fit
into the 16KB of shared memory on each SM.

### 3.8 Experimental Evaluation

We have implemented the proposed compilation methodology on the StreamIt com-
piler tool-chain. In this section, we demonstrate the effectiveness of our methodology
and compare it with two alternative approaches to solve the problem:

1. A software pipelined version that does not coalesce accesses to device memory.
   Such a comparison brings out the benefits of the proposed buffer layout scheme
   alone.

2. A serialized execution model, where all filters execute a Single Appearance Sched-
   ule (SAS) [13, 34, 45], the execution spanning across all multiprocessors available,
3.8. Experimental Evaluation

with the maximum level of multi-threading supported by the GPU. However, since a SAS schedule typically has the maximum buffer requirement among all schedules of a steady state solution, the buffer requirements of the SAS schedule are constrained to that of the software pipelined schedule to ensure a fair comparison between the two approaches. Comparing our approach with a single appearance schedule helps to get an idea of the performance benefits due to the software pipelined execution model.

3.8.1 Experimental Methodology

We use benchmarks distributed along with the StreamIt compiler toolkit version 2.1.1 to evaluate our scheme \[9\]. The details of each benchmark and a brief description is provided in Table 3.1. We compile each benchmark as described in Figure 3.3 for the CUDA framework. The \textit{nvcc} compiler is then used to compile the resulting sources to an executable form. We choose to target 16 blocks to match the 16 multiprocessors available. We only consider programs with stateless filters in this study, since stateful filters have an implicit self-dependence between one firing and the next and hence are not data parallel. We deal with programs which consist of stateful filters in Chapter 4.

Each benchmark was compiled and executed on a machine with dual quad-core Intel Xeon processors, running at 2.83 GHz, with 16 GB of FB-DIMM main memory. The machine runs Linux, with kernel version 2.6.18, and the NVIDIA driver version 173.14.09. We have used a GeForce 8800 GTS 512 GPU, with 512 MB of device memory for our experiments. In the results presented later, we do not report execution times, but report only the speedups relative to a single threaded CPU executing the same program. We define the speedup as:

\[
\text{speedup} = \frac{t_{\text{host}}}{t_{\text{gpu}}},
\]

where \(t_{\text{host}}\) is the time taken for executing same program on the host CPU mentioned, with a single thread of execution and \(t_{\text{gpu}}\) is the time taken for executing the same program on the GPU. The CPU version of the program was compiled using the uniprocessor back-end of the StreamIt compiler suite. The C sources produced by the
Table 3.1: Benchmarks Evaluated

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Filters</th>
<th>Peeking Filters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitonic</td>
<td>58</td>
<td>0</td>
<td>Bitonic sorting network for sorting 8 integers</td>
</tr>
<tr>
<td>BitonicRec</td>
<td>61</td>
<td>0</td>
<td>Recursive implementation of the bitonic sorting network for sorting 8 integers</td>
</tr>
<tr>
<td>DCT</td>
<td>40</td>
<td>0</td>
<td>8x8 Discrete Cosine Transform</td>
</tr>
<tr>
<td>DES</td>
<td>55</td>
<td>0</td>
<td>Implementation of the DES encryption algorithm</td>
</tr>
<tr>
<td>FFT</td>
<td>26</td>
<td>0</td>
<td>Fast Fourier Transform implementation</td>
</tr>
<tr>
<td>Filterbank</td>
<td>53</td>
<td>16</td>
<td>Filter bank to perform multi-rate signal processing</td>
</tr>
<tr>
<td>FMRadio</td>
<td>67</td>
<td>22</td>
<td>Software FM Radio with 10-band equalizer</td>
</tr>
<tr>
<td>MatrixMult</td>
<td>43</td>
<td>0</td>
<td>Blocked matrix multiply</td>
</tr>
</tbody>
</table>

uni-processor back-end were compiled with gcc using optimization level -O3.

The timing information for the GPU version was collected using the event mechanism provided by the CUDA framework, while the timing information for the CPU version was collecting using the standard UNIX syscall gettimeofday(). The timing measurements reported as speedups over the base CPU version, with both versions doing the same amount of work.

3.8.2 Experimental Results

We evaluate the following aspects of our software pipelining implementation in our experiments:

1. The effect of coarsening the granularity of the software pipelined schedule on the execution time, where coarsening implies that we repeat the software pipelined schedule an integral number of times in each GPU kernel call.

2. Comparison of our methodology with a software pipelining solution that does not coalesce accesses to device memory, i.e., without the buffer optimizations
3.8. Experimental Evaluation

Figure 3.11: Effect of Coarsening on Performance

proposed in Section 3.7.

3. Comparison with a serialized (one filter at a time, but fully data parallel SAS schedule, which coalesces accesses to device memory.

3.8.2.1 Effect of Coarsening the Granularity of the Software Pipelined Schedule

Figure 3.11 shows the effect of coarsening the granularity on the execution time. The results for each benchmark are plotted along with the geometric mean as the last set of bars in Figure 3.11 and Figure 3.12. In the SWPn schedule, each instance of a filter is iterated n times to increase the granularity of the GPU kernel. Thus, an SWPn kernel performs n times as much work in each iteration as the original software pipelined kernel. This does not affect the optimality of the schedule, since the delay of each filter is increased by the same proportion, thereby leaving the work distribution still uniform. However, this has the effect of smoothing out any anomalies in the schedule, caused by the execution times of filters being non-deterministic, due to memory access arbitration among the multiprocessors. This also has the effect of increasing the amount of work done per kernel invocation, amortizing the cost of launching kernels on the GPU over more iterations. We observe from Figure 3.11 that the optimized SWP schemes achieve
a speedup of 1.87X to 36.83X for the different benchmark programs. Further the gains start to plateau between SWP8 and SWP16 for all benchmarks. This is due to the fact that the kernel launch overhead is fixed and its effect becomes negligible when larger amounts of work are performed per kernel invocation.

3.8.2.2 Comparison with SAS Schedules and Software Pipelined Execution Without Buffer Layout Optimizations

We now compare the performance of our scheme with two other schemes. SWPNC is the same as our implementation, except that memory access coalescing is not done, i.e., the buffer layout scheme described in Section 3.7 is not applied, using a serial or FIFO ordering instead. For this, the profile runs are also executed without memory access coalescing. However, if the number of threads with which the filter is to be executed is such that the working set (the push and the pop set) can fit into shared memory, then we bring in the entire working set into shared memory using coalesced reads. Note that the performance gains from this approach is highly dependent on the push and pop characteristics of the filter. The Serial scheme is such that every filter is run as a separate kernel in a SAS schedule. We fix the number of blocks with which a filter executes to 16
— same as with the SWP scheme — and set the number of threads so that the buffer usage is less than or equal to the SWP scheme compared here, which is SWP8. No buffer sharing is performed in all our schemes; in other words, a buffer allocated for a channel is not shared with another. All buffers are allocated at the beginning of the run and are not freed until the program completes execution. Table 3.2 shows the buffer requirements of the optimized software pipelined schedule which has been coarsened 8 times (SWP8) for each benchmark. Although the serial scheme executes filters in a serial fashion, we have implemented this scheme to coalesce device memory accesses. The optimized software pipelining scheme (the SWP scheme) performs better than the alternative schemes in all benchmarks, except for the MatrixMult and the DCT benchmarks, where the serial version performs marginally better. These benchmarks display a phased behavior, with each phase having a splitter which exposes a large amount of task parallelism. However very little work is done in the task parallel branches before a joiner that combines the results of the task parallel branches and passes them on to the next phase. These joiners and splitters are bandwidth hungry in nature, since all they do is move data around, without any computation. Our scheme does not take these second order effects into account. We believe this to be skewing the actual execution times of the filters, leading to an imbalanced work distribution. The study of such phased behavior in stream programs, is an interesting area of further research, but is beyond the scope of this work.

SWPNC performs poorly in all benchmarks, yielding a maximum speedup of only 1.22X, except in the Filterbank and FMRadio. The poor performance can be explained as being caused by non-coalesced accesses causing an under-utilization of the memory bus. The high speedups of 11.59 and 31.78 in the Filterbank and FMRadio benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>SWP8</th>
<th>SWPNC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitonic</td>
<td>5308416</td>
<td>4472832</td>
</tr>
<tr>
<td>BitonicRec</td>
<td>29360128</td>
<td>59768832</td>
</tr>
<tr>
<td>DCT</td>
<td>29360128</td>
<td>59768832</td>
</tr>
<tr>
<td>DES</td>
<td>59768832</td>
<td>92602368</td>
</tr>
<tr>
<td>FFT</td>
<td>25165824</td>
<td>7471104</td>
</tr>
<tr>
<td>Filterbank</td>
<td>25165824</td>
<td>7471104</td>
</tr>
<tr>
<td>FMRadio</td>
<td>1671168</td>
<td>92602368</td>
</tr>
<tr>
<td>MatrixMult</td>
<td>92602368</td>
<td>92602368</td>
</tr>
</tbody>
</table>

Table 3.2: The Buffer Requirements for Each Benchmark. All sizes are in Bytes.
Table 3.3: Solution Times and Relaxation for the Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Solution Time (s)</th>
<th>% Relaxation from MII</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitonic</td>
<td>161</td>
<td>4.5</td>
</tr>
<tr>
<td>BitonicRec</td>
<td>122</td>
<td>3.0</td>
</tr>
<tr>
<td>DCT</td>
<td>178</td>
<td>4.5</td>
</tr>
<tr>
<td>DES</td>
<td>26</td>
<td>1.5</td>
</tr>
<tr>
<td>FFT</td>
<td>4</td>
<td>7.0</td>
</tr>
<tr>
<td>Filterbank</td>
<td>28</td>
<td>1.5</td>
</tr>
<tr>
<td>FMRadio</td>
<td>2</td>
<td>7.0</td>
</tr>
<tr>
<td>MatrixMult</td>
<td>25</td>
<td>0.5</td>
</tr>
</tbody>
</table>

are due to the fact that in these benchmarks, the entire push and pop working set of all the threads fits into the shared memory for each filter and hence is brought into shared memory and written back from shared memory, using a series of coalesced accesses. Bank conflicts do occur in shared memory, however, these are conflicts are 1-cycle conflicts and do not impose a severe performance penalty, yielding a net performance gain.

The low speedup values for the Bitonic, BitonicRec are due to the fact that these benchmarks perform very little work in each of the filters. The benchmarks consist of compare-exchange filters and splitters and joiners. The compare-exchange filters only perform a comparison as computation. Thus, these benchmarks are extremely bandwidth intensive, with almost no computation behind which the memory latency can be hidden.

The FFT benchmark achieves low speedup due to a large number of filters in the benchmark which have register requirements that far exceed the 8192 registers available on each SM, even when they are compiled with a register constraint of 64 registers per thread. For instance, the CombineDFT filter in the FFT benchmark requires arrays of 8, 16, 32 and 64 floats. The nvcc compiler does not place these arrays in the register files, instead placing them in the device memory in a thread local location. Thus, accesses to these arrays cause long latency accesses to device memory limiting
the speedup obtained. The DES and MatrixMult benchmarks also have filters which have high register requirements that exceed the number of available registers. However, in these benchmarks, the number of such filters is small and do not significantly limit the performance gains on the GPU.

3.8.2.3 Efficacy of the ILP formulation

We conclude the discussion on the experimental results with a mention of the efficiency of the compilation process. We have used CPLEX version 9.0, running on Linux to solve the ILP. The machine used was a quad processor Intel Xeon 3.06GHz, with 4GB of RAM. However, for all the solution times reported, CPLEX was running as a single threaded application and hence did not make use of the SMP available. The ILP solution process which is the most complex step of the compilation trajectory, is quite fast for all the benchmarks. The methodology we used to solve the ILP was to determine the II as $$\max(\text{ResMII}, \text{RecMII})$$. Once this was done, the solver was allowed 20 seconds to attempt a solution with this II. If it failed to find a solution in 20 seconds, the II was relaxed by 0.5% and the process was repeated until a feasible solution was found. It should also be noted that the compilation time is dominated by the \texttt{nvcc} compiler which takes 1 – 2 orders of magnitude more time. As shown in Table 3.3, all of the benchmarks took less than 30 seconds to solve, except for Bitonic, BitonicRec and DCT, which took 161, 122 and 178 seconds respectively. All solutions were found within a 5% relaxation on the II, except for FFT and FMRadio, both of which required a 7% relaxation. Thus the proposed solution is quite efficient and finds solutions that are quite close to the MII. Also, the solution time is within three minutes for all the benchmarks evaluated, with most of the benchmarks requiring less than 30 seconds for a solution. Note that although the MII forms a lower bound for the II. It does not guarantee that a schedule exists at the MII.

The NP-Hardness of solving an Integer Linear Program raises the question as to whether this approach scales to larger programs. We expect large StreamIt programs to be modular and hence the ILP formulation may be applied to each module separately.

---

2RecMII was 0 for all the benchmarks, since none of the benchmarks in the set of benchmarks provided with the StreamIt distribution had feedback loops and we have not considered stateful filters.
thus yielding efficient solutions even for larger programs

3.9 Summary

We have described a methodology for compiling StreamIt programs for execution on GPUs which includes determines the optimal execution configuration for the program as well as orchestrates the software pipelined execution of the StreamIt program. The scheduling and work assignment is formulated as an Integer Linear Program (ILP) which is solved using CPLEX (an industrial strength solver). We also suggest a buffer layout technique that makes effective use of the available memory bandwidth on the GPU. The scheme has been implemented in the StreamIt compiler (version 2.1.1), which has been made publicly available at the StreamIt home page [9].

The suggested methodology performs well, yielding upto 36.83X speedup over an optimized single threaded CPU execution. The benefits of the software pipelined execution as well as the buffer layout optimization is brought out by comparing the performance of the scheme with a naïve software pipelining implementation, which does not coalesce buffer accesses and with a Serialized SAS schedule which has the same buffer requirements as the optimized software pipelined execution scheme. The naïve SWP scheme yields an average speedup of 1.87X; the Serial SAS scheme yields an average speedup of 3.73X, while the optimized software pipelined execution yields an average speedup of 5.02X across the StreamIt benchmark suite. This clearly highlights the impact of software pipelining as well as the buffer layout optimization on performance. The ILP methodology is also quite efficient, with most of the benchmarks taking less than 30 seconds for solution, while the results are at most within 7% of the best possible solution in terms of the Initiation Interval (II) achieved.

The proposed methodology handles only stateless filters. Although programs with stateful filters are rare, extending the framework to support programs with stateful filters would be interesting. Also, while the GPU performs computation, the CPU cores are idle. With multi-core platforms becoming ubiquitous, leveraging the computational resources of the CPU cores could lead to significant performance gains and remains an interesting problem. We address this in greater detail in Chapter 4.
Chapter 4

Synergistic Execution of Stream Programs on Multicores with Accelerators

If I could solve all the problems myself, I would.

THOMAS ALVA EDISON

United, united, united we stand; United we never shall fall
United, united, united we stand; United we stand one and all

JUDAS PRIEST

4.1 Introduction

The ubiquity of GPUs as powerful stream processing engines makes a compelling case for compiling stream programs for execution on GPUs. Chapter 3 described our approach for compiling programs written in StreamIt, a high level stream programming language, for execution on the GPU. However, this approach suffers from the following limitations:

- The approach does not support StreamIt programs with stateful filters. While programs with stateful filters are rare, they still need to be supported for completeness.
The approach utilizes only the GPU to perform computation while the the CPU cores are left idle. Utilizing the CPU cores to perform useful work could yield significant performance benefits.

In this chapter we try to address these two limitations and leverage the computational capabilities of the CPU to execute stateful filters and to provide a synergistic execution framework for StreamIt programs.

The task of executing programs across the CPU cores and the GPU is complicated by the following factors:

- The CPU and the GPU operate on disjoint address spaces. Neither can the GPU access any of the host memory space, nor can it initiate transfers into or out of the host memory.

- The CPU can initiate DMA transfers into or out of the device memory, but the DMA bandwidth is limited and needs to be modeled as a resource in order to ensure that the DMA channel does not become a bottleneck.

- The DMA latency needs to be kept off the critical path; in other words it needs to be overlapped with computation so that the effect of the DMA transfers does not add to the critical path of the program.

- The buffer layout scheme proposed in Section 3.7 enables the GPU to utilize its memory bandwidth effectively. However, on the CPU, such a layout causes a large number of cache misses owing to the strided access pattern it results in. Thus the buffers need to be transformed into the appropriate form when data is moved between the GPU and the CPU.

- Finally, the work has to be partitioned between the CPU cores and the GPU which have vastly different compute capabilities. While GPUs are extremely efficient for data parallel code, they result in poor performance while executing code with control dependencies or if the code has register requirements that exceed the size of the register files. The CPU is better suited to execute such sections of code owing to its ability to exploit instruction level parallelism and out of order execution. The problem of work partitioning between the CPU and GPU is not amenable to
simple graph partitioning heuristics such as those described in [38], due to the fact that the weight of the nodes are not constant, but depend on which partition they are assigned to, since the execution time of the same code section might be different on the CPU and on the GPU.

In order to address these challenges, we formulate the partitioning problem, including the buffer transformation operations, as an integer linear program (ILP). This approach accurately models the DMA bandwidth as a resource to ensure that all resources (CPU cores, GPU SMs and DMA bandwidth) are utilized in a balanced manner. Once the partitioning is complete, we execute the program in a software pipelined fashion which overlaps the DMA with useful computation on the CPU and GPU. We also develop a simple and efficient heuristic partitioning technique that compares favorably with the optimal partitions obtained from the ILP formulation.

The rest of this chapter is organized as follows: Section 4.2 describes the impact of buffer layout on performance and suggests minor changes to the buffer layout scheme suggested in Chapter 3. Section 4.3 motivates our approach with an illustrative example. In Section 4.4 we provide an overview of the proposed methodology. Section 4.5 and Section 4.6 respectively, describe the ILP formulation and the heuristic algorithm for partitioning the StreamIt graph. Sections 4.8 and 4.9 detail the modulo scheduling and code generation steps of the compilation process. In section Section 4.10 we present the results of experimentally evaluating our approach and Section 4.11 provides a summary of our synergistic approach.

### 4.2 Buffer Layout Considerations

As mentioned in Chapter 2 accesses to GPU memory must be coalesced in order to effectively utilize the high memory bandwidth available. To this end, we proposed a buffer layout scheme in Section 3.7 in Chapter 3. While this buffer layout scheme helps in improving the performance on the GPU, it adversely affects the performance, if used as is, owing to the strided access pattern seen by the CPU, which causes a large number of cache misses.

The effect of data layout on performance is shown in Table 4.1, which gives the
execute time for accessing 32 MB of data on the CPU and the GPU, using both the coalesced (shuffled) and serial buffer layouts. Clearly, the shuffled layout degrades performance by more than an order of magnitude on the CPU. This is because the caches are effectively rendered useless due to the strided access pattern. The serial layout incurs a similar performance loss on the GPU. Thus, it is essential that the buffers be transformed to the layout that is most suited for the CPU and the GPU before they operate on the data.

We term the process of transforming a serially laid out buffer into the shuffled layout as a shuffle operation, while the inverse operation of transforming a shuffled buffer into a serial layout as a deshuffle operation.

We modify the buffer layout proposed in Section 3.7 slightly. The original buffer layout grouped data elements popped by 128 contiguous threads together, since it was the GCD of 128, 256, 384 and 512, which were the block sizes we considered. However, it is not necessary that the grouping be done at the granularity of 128 elements. The CUDA documentation only requires that the accesses be coalesced within a warp, which translates to 32 accesses. Thus, for the synergistic execution, we formally define the shuffle function as:

$$s(i) = b \left( \left\lfloor \frac{i}{ws} \right\rfloor + (i \mod ws) \times \left( o \times k \times \frac{n_t}{ws} \right) \right)$$

where $o$, $k$, $n_t$ and $ws$ are the pop rate of the consumer on the edge under consideration, the firing rate of the consumer in the steady state schedule, the number of threads the consumer is executing with and the warp size of the device respectively;

---

\[1\] We refer to the buffer layout described in Section 3.7 as the shuffled buffer layout.

---

<table>
<thead>
<tr>
<th></th>
<th>Serial (Uncoalesced) Access (ms)</th>
<th>Shuffled (Coalesced) Access (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>14.55</td>
<td>187</td>
</tr>
<tr>
<td>GPU</td>
<td>176.6</td>
<td>8.1</td>
</tr>
</tbody>
</table>

Table 4.1: Execution times with serial (Uncoalesced on the GPU) and shuffled (Coalesced on the GPU) accesses
4.2. Buffer Layout Considerations

$s(i)$, $\forall i \in [0, o \times k \times n_t]$ denotes the value at the $i^{th}$ position in the shuffled buffer and $b(i)$ denotes the value of the $i^{th}$ element in the natural FIFO ordering of the buffer.

Assuming that all filters on the GPU are executed with a number of threads which is a multiple of the warp size $ws$, the index $j$ of the $n^{th}$ element pushed (popped) by a thread whose $threadid$ is $tid$, in the $m^{th}$ instance of the filter, with a push (pop) rate $o$ is given by:

$$j = off + ws \times n + \left\lfloor \frac{tid}{ws} \right\rfloor \times ws \times o + (tid \mod ws), \quad n < o$$

where $off = m \times o \times n_t$ is the offset for the $m^{th}$ instance of the filter.

Thus we have changed the granularity of the shuffle operation from 128 to 32 elements. This is in order to enable the GPU to perform the shuffle and deshuffle operations efficiently and is explained in greater detail in Section 4.7.

In general, all buffers for the filters assigned to the CPU are laid out in a deshuffled fashion, while the buffers for the filters executing on the GPU are laid out in a shuffled fashion. However peeking filters, i.e., the filters that inspect tokens on their input FIFO that they do not immediately consume, cannot utilize their buffers effectively in a shuffled layout, since there will always be some residual elements on their input FIFOs from a previous iteration. Laying these buffers in a shuffled manner while ensuring coalesced accesses by threads would require a lot of data duplication to maintain the “moving window” of data as seen by the individual threads. Thus, the input and output buffers for the peeking filters are always laid out in a sequential (deshuffled) manner, regardless of whether they are to be executed on the CPU or the GPU. When such peeking filters are to be executed on the GPU, in order to avoid uncoalesced accesses at the device memory, we stage the entire working set into shared memory before beginning execution of such filters. The output is also produced in a deshuffled fashion and is staged into shared memory to avoid uncoalesced writes. After the peeking filter has completed execution, we move the entire output set into device memory using a series of coalesced writes. This approach works well for peeking filters with small working sets. If the working set of a peeking filter is too large to be accommodated in the 16KB of shared memory available, we fix the filter to execute on the CPU.
4.3 A Motivating Example

We now provide an example that serves to highlight the need for a synergistic partitioning of StreamIt programs across the CPU cores and the GPU. Consider the simple StreamIt graph shown in Figure 4.1(a), which consists of a pipeline with five filters. Assume that the firing rate of each filter is unity. The execution time of each filter on the CPU and the GPU, and the latencies for transferring the required data for each channel is shown in Figure 4.1. For simplicity, we assume that the shuffle and deshuffle costs associated with the edges are zero, although we take them into account in the final formulation. Filter B is a stateful filter which can be executed only on the CPU. We now need to partition the remaining filters optimally to achieve the minimum possible lower bound for the Initiation Interval (MII) for the software pipelined schedule, where MII is as defined in Section 3.2 except that now the DMA channel is also modeled as a resource. If the graph contains one or more feedback loop structures, then each feedback loop can easily be fused into a stateful filter. Since stateful filters and feedback loop structures govern RecMII which cannot be reduced, our approach partitions...
4.3. A Motivating Example

Figure 4.2: The Software Pipelined Kernel for the Partitioning shown in Figure 4.1 (d). For simplicity, only one CPU core and one GPU SM is assumed.

... the remaining filters to execute on the CPU or the GPU such that the resulting ResMII is as close to RecMII as possible. Thus, the partitioning strategy tries to minimize the ResMII, by appropriately partitioning the nodes between the CPU and the GPU and modeling the DMA channel as a resource, which also contributes to ResMII.

Assume a single GPU SM and a single CPU core. Figure 4.1(b) shows the MII if we naively map filter B on the CPU and execute all the other filters on the GPU. This partition incurs DMA transfer costs from the GPU to the CPU (between filter A and B) and again from the CPU to the GPU (between filter B and C). As can be seen, a load imbalance also exists in this case, resulting in an MII of 75.

Figure 4.1(c) shows the partitioning obtained if we try a greedy strategy by moving a filter to the partition (either the CPU or the GPU), where it is most beneficial to execute the filter. In this case, although the load has been well distributed between the CPU and the GPU, the DMA channel is over utilized, yielding an MII of 70. Finally, the partition depicted in Figure 4.1(d) achieves the lowest possible MII of 45 and is optimal. This example demonstrates that simple heuristics do not work well and that a more intelligent partitioning strategy that takes into account the DMA and buffer transformation latencies is called for. Also, graph partitioning heuristics proposed in [38], based on...
static weights on the nodes do not help, since the weights of the nodes are not static and depend on which partition each filter is assigned to.

Figure 4.2 shows how we propose to software pipeline the program as well as the edges representing DMA transfers, once the optimal partitioning as in Figure 4.2(d) is obtained. Again, for simplicity and clarity, the shuffle and deshuffle operations that may be required are not shown.

4.4 Overview of the Compilation Methodology

Figure 4.3 depicts the work-flow of the approach proposed to compile StreamIt program for synergistic execution across the CPU cores and the GPU SMs. We first profile the individual filters across both the CPU and the GPU in order to obtain accurate estimates of execution time for each filter as well as to determine the optimal execution configuration for each filter on the GPU as described in Section 3.4 in Chapter 3. The profiling and execution configuration phase is the same described in Chapter 3, except that we also determine the execution time for each filter when executed on the CPU cores. Once the profiling is completed and the optimal execution configurations have been determined for each of the filters, we then proceed to the partitioning phase. We accomplish the partitioning in two phases:

1. We first partition the StreamIt graph into two ways: One for the CPU cores and the other for the GPU SMs. This implies that a filter (all its instances) in the StreamIt graph is assigned either the CPU cores or to the GPU SMs. It is possible to consider the partitioning at the level of filter instances rather than at the level of entire filters. However, such a formulation leads to unnecessary complications, since the buffers associated with the filters must be allocated in both the CPU and GPU address spaces and data transfers must be handled at the level of filter instances. Also, since the number of instances of each filter could be in the order of a few tens to a few hundreds, the ILP formulation and the heuristic partitioner becomes unnecessarily complex. Hence we restrict our attention in our work to partitioning only at the level of filters. Further, in our approach, stateful filters and peeking filters with large working sets (described later) are fixed to the
CPU and are not allowed to be assigned to the GPU. We refer to this step as *task partitioning* or *filter partitioning*.

2. Once the task partitioning step has been completed, we then partition the instances of individual filters across the CPU cores or the GPU SMs, depending on which partition the filter has been assigned to. We refer to this step as *instance partitioning*.

We propose two different approaches for both the levels of partitioning. One is based on an exact integer linear programming formulation and the other based on a heuristics to provide for near-optimal partitioning.

Once the partitioning has been completed, we then orchestrate the execution of the StreamIt program across the GPU and the CPU cores by generating a modulo schedule and adding the appropriate DMA transfers and shuffle and deshuffle operations at the required stages of the modulo schedule. Finally, we generate a mixture of CUDA and C code that is compiled by the *nvcc* and the host compiler for the platform respectively, yielding the final executable.
4.5 ILP Formulation of the Partitioning Problem

In this section, we describe the ILP formulation we use for obtaining an optimal partition of the StreamIt graph. We use an ILP formulation for both the task partitioning and the instance partitioning steps.

4.5.1 Task Partitioning

4.5.1.1 Definitions

We begin the derivation of the ILP formulation for the task partitioning by defining the symbols used in the ILP formulation. Many of these symbols are the same as those used in Section 3.5. Nonetheless, we repeat them here for the sake of completeness and easy reference. As before, in our notation, variables in the ILP formulation are represented using lower case letters and constants using upper case letters.

- \( V \) is the set of all nodes in the stream graph. All the nodes are considered to be multi-threaded as described in Chapter 3.
- \( E \) is the set of all directed edges in the stream graph.
- \( C_{\text{fixed}} \subseteq V \) is the set of all filters fixed to the CPU. This includes all stateful filters and peeking filters whose working sets are larger than the size of the shared memory on the GPU. \( C_{\text{fixed}} \) is known before the partitioning formulation and is based purely on the characteristics of the filter and the size of the shared memory region in the target GPU.
- \( \text{PEEK}_v, \quad \forall v \in V \) are 0-1 constants which are set to 1 if and only if the filter \( v \) is a peeking filter.
- \( \text{gpu}_v, \quad \forall v \in V \) are 0-1 variables. \( \text{gpu}_v \) is set to 1 by the solver if and only if node \( v \) is assigned to the GPU.
- \( \text{shuf}_{uv} \geq 0, \quad \forall (u, v) \in E \) and \( \text{deshuf}_{uv} \geq 0, \quad \forall (u, v) \in E \) are real valued variables that denote the shuffle and deshuffle costs respectively, associated with a partition, for each edge \( (u, v) \in E \).
4.5. ILP Formulation of the Partitioning Problem

• DC(u, v) and SC(u, v), ∀(u, v) ∈ E are functions that return the deshuffle and shuffle cost associated with an edge (u, v) respectively, which depends on the amount of data transferred on the edge (u, v), but is a constant for a given edge in the stream graph.

• tran_{u,v} ≥ 0, ∀(u, v) ∈ E are real valued variables that denote the DMA transfer cost associated with a partition for each edge (u, v) ∈ E.

• cost_{shuf} ≥ 0 is a real valued variable indicating the total shuffle and deshuffle cost associated with a partition.

• cost_{DMA} ≥ 0 is a real valued variable that denotes the total cost of DMA transfers associated with a partition.

• KGPU_v and CPU_v, ∀v ∈ V are constants that denote the number of instances of a filter v ∈ V, in one steady state iteration of the software pipelined schedule, when a filter v is mapped onto the GPU and the CPU respectively.

• N_{cpus} and N_{gpus} are constants which denote, respectively, the number of CPU cores and GPU SMs available.

• D_{cpu}(v) and D_{gpu}(v), ∀v ∈ V are real valued constants that denote the delays of one instance of a filter (multi-threaded filter in the case of D_{gpu} and a version that executed 128 iterations in the case of D_{cpu}) v on the CPU and the GPU respectively, which are obtained through profiling.

• TGC(u, v) and TCG(u, v) are functions that return the DMA transfer cost associated with the edge (u, v), from the GPU to the CPU and from the CPU to the GPU respectively. This is a function of the amount of data transferred across the edge over one steady state iteration.

4.5.1.2 Formulation of the Task Partitioning Problem

We now describe the constraints for the ILP formulation for the task partitioning of the filters between the CPU cores and the GPU. A task partitioning is specified by the
values of the $\text{gpu}_v$ variables. First, we ensure that all filters in the set $C_{\text{fixed}}$ are kept on the CPU, by the following constraint:

$$\text{gpu}_v = 0, \forall v \in C_{\text{fixed}} \tag{4.1}$$

We model the DMA costs incurred by the cut edges, i.e., the edges with one end on the GPU and the other on the CPU, or vice versa as:

$$\text{tran}_{u,v} \geq (\text{gpu}_u - \text{gpu}_v) \times \text{TGC}(u,v), \forall (u,v) \in E \tag{4.2}$$

$$\text{tran}_{u,v} \geq (\text{gpu}_v - \text{gpu}_u) \times \text{TCG}(u,v), \forall (u,v) \in E \tag{4.3}$$

Note that at most one of the above two inequalities would be active as the RHS of the other would yield a negative value. Also, when both $u$ and $v$ are scheduled on the CPU (or both on the GPU), the RHS of the inequalities $4.2$ and $4.3$ would be 0. The total DMA cost associated with a partition is modeled as:

$$\text{cost}_{\text{DMA}} = \sum_{(u,v) \in E} \text{tran}_{u,v} \tag{4.4}$$

Inequalities (4.2) and (4.3) ensure that the $\text{tran}_{u,v}$ variables are greater than or equal to the DMA costs associated with the edge $(u,v)$. Since the objective function is to minimize the Initiation Interval (II), and by the constraints on the II given in Inequalities (4.10), (4.11) and (4.12) these values will be pushed to their lowest permissible levels during the course of the optimization.

We now model the shuffle and deshuffle costs associated with a partition. The conditions for an edge $(u,v) \in E$ to require a shuffle or deshuffle operation are as follows:

1. When filter $u$ is assigned to the GPU and $v$ is assigned to the CPU, a deshuffle operation is needed if and only if filter $u$ does not peek; this is because if filter $u$ peeks, then it would produce its outputs in a deshuffled order already.

---

2In this ILP formulation, it is ensured that all variables get a non-negative value (the non-negativity constraints are not shown). Thus if the RHS is negative, the constraint is subsumed by $\text{tran}_{u,v} \geq 0$. 
2. When both filters \( u \) and \( v \) are assigned to the GPU, a deshuffle operation is required if and only if filter \( v \) peeks and filter \( u \) does not; this is because there is no need to deshuffle between a pair of peeking filters connected by a producer-consumer relationship, since the output of the producer would be in a deshuffled order already, by virtue of it being a peeking filter.

3. When filter \( u \) is assigned to the CPU and \( v \) is assigned to the GPU, a shuffle operation is needed if and only if filter \( v \) does not peek. The reasoning is similar to the reasoning behind condition 1, i.e., if the filter \( v \) peeks, then it would require it buffers to be organized in a serial fashion, this eliminating the need for a shuffle operation.

4. When both filters \( u \) and \( v \) are assigned to the GPU, a shuffle operation is necessary if and only if filter \( u \) peeks and filter \( v \) does not. Again, the reasoning is similar to the reasoning behind condition 2, i.e., if a filter \( u \) peeks, it produces its outputs in a serial ordering, which is not suitable for use by filter \( v \), which requires its buffers to be in a shuffled layout for efficient execution, thus requiring a shuffle operation between the execution of filter \( u \) and filter \( v \).

Based on these conditions, we model the shuffle and the deshuffle costs as follows. Specifically, inequalities (4.5) and (4.6) model the deshuffle costs according to the conditions 1 and 2 above, while inequalities (4.7) and (4.8) model the shuffle costs according to the shuffle conditions 3 and 4 above.

\[
\text{deshuf}_{u,v} \geq (\text{gpu}_u - \text{gpu}_v) \times (1 - \text{PEEK}_u) \times \text{DC}(u,v), \quad \forall (u,v) \in E \quad (4.5)
\]

\[
\text{deshuf}_{u,v} \geq (\text{gpu}_u + \text{gpu}_v - 1) \times \text{PEEK}_v \times (1 - \text{PEEK}_u) \times \text{DC}(u,v), \quad \forall (u,v) \in E \quad (4.6)
\]

\[
\text{shuf}_{u,v} \geq (\text{gpu}_u - \text{gpu}_v) \times (1 - \text{PEEK}_v) \times \text{SC}(u,v), \quad \forall (u,v) \in E \quad (4.7)
\]

\[
\text{shuf}_{u,v} \geq (\text{gpu}_u + \text{gpu}_v - 1) \times \text{PEEK}_u \times (1 - \text{PEEK}_v) \times \text{SC}(u,v), \quad \forall (u,v) \in E \quad (4.8)
\]

Note that in Equation (4.5), Equation (4.6), Equation (4.7) and Equation (4.8) \( \text{PEEK}_u, \text{PEEK}_v, \text{SC}(u,v) \) and \( \text{DC}(u,v) \) are constants. We now model the total shuffle and
deshuffle cost by the following constraint:

\[
\text{cost}_{\text{shuf}} = \sum_{(u,v) \in E} \text{shuf}_{u,v} + \sum_{(u,v) \in E} \text{deshuf}_{u,v} \tag{4.9}
\]

Having modeled each of the shuffle and deshuffle cost and the DMA costs, we now model the constraints on the Initiation Interval II as follows:

\[
\begin{align*}
\text{II} & \geq \text{cost}_{\text{DMA}} \tag{4.10} \\
\text{II} & \geq \frac{1}{N_{\text{gpus}}} \left( \sum_{v \in V} (\text{gpu}_v \times D_{\text{gpu}}(v) \times \text{KGPU}_v) + \text{cost}_{\text{shuf}} \right) \tag{4.11} \\
\text{II} & \geq \frac{1}{N_{\text{cpus}}} \left( \sum_{v \in V} ((1 - \text{gpu}_v) \times D_{\text{cpu}}(v) \times \text{KCPU}_v) \right) \tag{4.12}
\end{align*}
\]

Again, as mentioned in Chapter 3 if the execution time of any instance of any filter is beyond the ResMII obtained, then it is impossible to obtain a schedule at any II less than the execution time of that filter. We resolve this problem in a similar fashion as in Chapter 3, i.e., we conservatively increase the multiplicities of the filters by an integral factor until the execution time of no filter dominates the II on both the CPU and the GPU partitions. Note that our approach is conservative in the sense that it might increase the multiplicities beyond the minimum required by the partition that is chosen, since it is performed prior to the actual partitioning. Thus every filter is assured to have a delay that will never dominate the II regardless of how the stream graph is partitioned.

Having thus modeled the II to be the maximum of the load on the GPU, the load on the CPU and the DMA load, we can now achieve an optimal II by using an objective function which minimizes II subject to the constraints developed.

### 4.5.2 Instance Partitioning

We now describe the ILP formulation for partitioning the instances of the filters that have been assigned to the CPU or to the GPU across the CPU cores or the GPU SMs respectively, from the formulation in Section 4.5.1. As mentioned earlier, in our approach, all instances of a filter are assigned either to the CPU cores or the GPU SMs depending
4.5. ILP Formulation of the Partitioning Problem

on the partitioning decision taken in the task partitioning step. In this phase, partition-
ing the instances of the filters, rather than the filters themselves, allows us to perform
the partition at a much finer granularity.

4.5.2.1 Definitions

We begin the description of the ILP formulation by defining the symbols used in the
ILP:

- $\text{Nodes}_{\text{CPU}} \subseteq V$ is the set of all nodes assigned to the CPU
- $\text{Nodes}_{\text{GPU}} \subseteq V$ is the set of all nodes assigned to the GPU. Note that since the
  nodes are partitioned between the CPU and the GPU, $\text{Nodes}_{\text{GPU}} = V - \text{Nodes}_{\text{CPU}}$
- $w_{\text{CPU}}(k, v, p), \forall v \in \text{Nodes}_{\text{CPU}}, \forall k \in K_{\text{CPU}}, \forall p \in [0, N_{\text{cpus}})$ are 0-1 vari-
  ables which denote that the $k^{th}$ instance of a filter $v$ has been assigned to CPU
core $p$

4.5.2.2 Formulation of the Instance Partitioning Problem

We first describe the instance partitioning formulation for the CPU cores. To ensure
that every instance of every filter is mapped on to exactly one CPU core, we model the
following constraint:

$$\sum_{p \in [0, N_{\text{cpus}})} w_{\text{CPU}}(k, v, p) = 1, \forall v \in \text{Nodes}_{\text{CPU}}, \forall k \in [0, K_{\text{CPU}}(v))$$ (4.13)

Now to ensure that no core is loaded beyond the Initiation Interval $\Pi$, achieved as in
Section 4.5.1, we use the following constraint:

$$\sum_{v \in \text{Nodes}_{\text{CPU}}} \sum_{k \in [0, K_{\text{CPU}}(v)]} (w_{\text{CPU}}(k, v, p) \times D_{\text{CPU}}(v)) \leq \Pi, \forall p \in [0, N_{\text{cpus}})$$ (4.14)

By solving these constraints, a partitioning of the instances of the filters assigned to the
CPU can be obtained which satisfies the $\Pi$ that has been determined. A similar for-
mulation is used to partition the instances of the filters assigned to the GPU across the
Chapter 4. Synergistic Execution of Stream Programs

GPU SMs, as well as to partition the instances of the shuffle and deshuffle operations across the GPU SMs.

4.6 An Efficient Heuristic Partitioning Algorithm

While the ILP formulation yields optimal solutions to the partitioning problem and is useful to compare other methods with a known lower bound on the II, it is not quite suited for use in a production environment, since the execution time for solving the ILP could be very large, since the problem of solving an Integer Linear Program is an NP-Hard problem. This motivates the development of a heuristic algorithm for the task partitioning. Intuitively, we would expect the nodes assigned to the CPU to be the nodes most beneficial to execute on the CPU. We therefore define

\[
\text{Speedup}_{\text{cpu}} = \frac{D_{\text{gpu}}(v)}{D_{\text{cpu}}(v)}, \quad \text{for each filter } v, \quad \text{where } D_{\text{cpu}}(v) \text{ and } D_{\text{gpu}}(v) \text{ are as defined in Section 4.5.1.}
\]

A high value for this metric implies that the filter is better suited for execution on the CPU than on the GPU. A value greater than 1 for this metric indicates the filter can be executed faster on the CPU than on the GPU. Now, taking the intuition further, we would expect not just the nodes with the highest Speedup\_cpu values to be assigned to the CPU but also some of their neighboring nodes which could possibly have low values of Speedup\_cpu, as the cost of DMA transfers and the shuffle and deshuffle costs (if necessary) would otherwise dominate and offset the benefits. Going by this intuition, we propose the heuristic described in Algorithms 3 and 4. The procedure PARTITION is the main entry point for the partitioner. The sets CPU\_Nodes and GPU\_Nodes form the initial partition with CPU\_Nodes = C_{fixed} and with GPU\_Nodes = V - C_{fixed}. We assume that all filters in C_{fixed} execute on the CPU cores and all the remaining filters are initially assumed to execute on the GPU when the algorithm begins. Filters are moved from the GPU to the CPU partition over the course of the algorithm depending on the reduction in the II achieved. Thus all the II calculations in the algorithm are done by taking these fixed nodes on the CPU into account, in addition to whatever other nodes might be added. The II calculations also take into account the shuffle and deshuffle costs as well as the DMA transfer costs associated with a partition. The first step of the partitioning is to get the set of clusters of nodes...
Algorithm 3 Heuristic Partitioning Algorithm

1: procedure PARTITION(CPUNodes, GPUNodes)
2:    S ← Nodes sorted in decreasing order of Speedup_cpu
3:    Clusters ← GETCLUSTERS(S)
4:    bestCluster ← REFINESCULUSTERS(Clusters)
5:    CPUNodes ← CPUNodes ∪ bestCluster
6:    GPUNodes ← GPUNodes − bestCluster
7: end procedure
8:
9: procedure GETCLUSTERS(S) \(\triangleright\) Returns a set of clusters
10:    clusters ← ∅
11:    while \((|S| = 0) \land (\text{numNodes} < \text{thresh} \times |V|)\) do
12:        curCluster ← ∅
13:        root ← First Node in S
14:        S ← S − root
15:        curCluster ← curCluster ∪ root
16:        GROWCLUSTER(curCluster, root)
17:        clusters ← clusters ∪ curCluster
18:        numNodes ← numNodes + |curCluster|
19:        return curCluster
20: end while
21: end procedure
22:
23: procedure GROWCLUSTER(curCluster, root)
24:    newRoot ← The node among all the predecessors of root, which is most
25:        beneficial to be added to CPUNodes in terms of reduction in II
26:    curCluster ← curCluster ∪ newRoot
27:    GROWCLUSTER(curCluster, root)
28:    newRoot ← The node among all the successors of root, which is most
29:        beneficial to be added to CPUNodes in terms of reduction in II
30:    curCluster ← curCluster ∪ newRoot
31:    GROWCLUSTER(curCluster, newRoot)
32: end procedure

that are beneficial to be moved to the CPU. This is achieved by a call to the procedure GETCLUSTERS. This procedure takes as input the set of all nodes sorted in decreasing order of their Speedup_cpu values. It then calls the recursive procedure GROWCLUSTER on these nodes to obtain a set of clusters. Each of these clusters is a set of contiguous filters in the StreamIt graph. Also, the clusters formed in this step are not necessarily disjoint. Note that each call to GROWCLUSTER starts with the assumption that the CPUNodes = \(C_{\text{fixed}}\) and GPUNodes = \(V − C_{\text{fixed}}\) and starts growing the graph with
Algorithm 4 Heuristic Partitioning Algorithm (...contd.)

33: procedure \text{REFINECLUSTERS}(\text{clusters})
34: \hspace{1em} while true do
35: \hspace{2em} tryAgain ← false
36: \hspace{2em} srtClust ← clusters sorted in increasing order of II they achieve
37: \hspace{2em} for i ← 0, |srtClust| do
38: \hspace{3em} Choose a \( j > i \) such that it results in the maximum reduction
39: \hspace{3em} in II by calling \text{SMARTFUSION}(srtClust[i], srtClust[j]).
40: \hspace{2em} if no such \( j \) exists then
41: \hspace{3em} continue
42: \hspace{2em} else
43: \hspace{3em} newFusion ← \text{SMARTFUSION}(srtClust[i], srtClust[j])
44: \hspace{3em} srtClust ← srtClust − srtClust[i]
45: \hspace{3em} srtClust ← srtClust − srtClust[j]
46: \hspace{3em} srtClust[i] ← newFusion
47: \hspace{3em} clusters ← srtClust
48: \hspace{3em} tryagain ← true
49: \hspace{3em} break
50: \hspace{2em} end if
51: \hspace{2em} end for
52: \hspace{2em} if tryagain is false then
53: \hspace{3em} return clusters[0]
54: \hspace{2em} end if
55: \hspace{2em} end while
56: \hspace{2em} end procedure
57:

58: procedure \text{SMARTFUSION}(\text{clusterA}, \text{clusterB})
59: \hspace{1em} sortedFuse ← \text{clusterA} \cup \text{clusterB}
60: \hspace{1em} with nodes sorted in decreasing order of $\text{Speedup}_{\text{cpu}}$
61: \hspace{1em} bestII ← \infty
62: \hspace{1em} fused ← \emptyset
63: \hspace{1em} for i ← 0, |sortedFuse| do
64: \hspace{2em} if II by adding \text{sortedFuse}[i] to CPU < \text{bestII} then
65: \hspace{3em} fused ← fused \cup \text{sortedFuse}[i]
66: \hspace{2em} end if
67: \hspace{1em} end for
68: \hspace{1em} return fused
69: \hspace{1em} end procedure

the \textit{root} node that has been passed as the parameter. The \textit{threshold} mentioned in line 11 determines how much of the graph is covered by the process of growing the clusters. Higher values result in a larger number of clusters. We set this parameter to 2.5 in all our experiments since it provides a reasonable compromise between the time to form
4.6. An Efficient Heuristic Partitioning Algorithm

The set of clusters and the efficiency of our heuristic approach.

Each cluster in the set of clusters returned by GROWCLUSTERS may be considered to be a candidate partition, independent of the other clusters in the set, with the filters in the cluster being assigned to the CPU and the rest of the filters assigned to the GPU. Thus each cluster in the set returned by GROWCLUSTERS has an II associated with it, which is simply the II if all the filters in the cluster are assigned to the CPU, and the rest of the filters are assigned to the GPU, the II calculation being done taking into account the the fixed nodes, the shuffle and deshuffle costs as well as the DMA transfer costs associated with that partition as mentioned before. Note that the clusters are not necessarily disjoint, however, since each of these clusters is a candidate partition independent of the other clusters, this is not an issue.

Growing clusters in this fashion has the limitation that it can only move contiguous regions in the stream graph to the CPU. It will not be able to capture partitions such as that shown in Figure 4.1(d). To overcome this limitation, we follow up the cluster growing phase with a cluster fusion phase. The procedure REFINEC LUSTERS performs this operation, through a smart fusion process. The smart fusion process, described in the procedure SMARTFUSION, essentially takes two clusters and greedily merges the parts of them that result in the maximum reduction in II. Thus the procedure REFINEC LUSTERS iteratively tries to apply SMARTFUSION to the set of clusters until no further benefits can be obtained from the fusion. It then returns the fused cluster with the lowest II value.

The final CPU partition is thus the set of fixed CPU nodes, along with the nodes in the best cluster; while the final GPU partition consists of all remaining nodes in the stream graph. Note that the fixed CPU nodes are never considered to be moved to the GPU by the partitioning algorithm.

Algorithm 3 thus takes care of task partitioning and determines the sets of CPU nodes and the GPU nodes. For the instance partitioning step, we use Metis [36] as a set partitioner to partition the instances of filters across the CPU cores or the GPU SMs. We do this by adding zero weight edges between the instances of the filters, since we are only concerned with load balancing. We also partition the instances of the shuffle and deshuffle operations across the GPU SMs using the same technique. This completes the
instance partitioning step.

Once the partitioning is completed, using either the ILP partitioning approach or the heuristic algorithm described in Algorithm 3, the StreamIt program is then modulo scheduled as described in Section 4.8 and the final CUDA and C code for the StreamIt program is generated as described in Section 4.9.

### 4.7 The Shuffle and Deshuffle Operations

As mentioned in Section 4.2, the shuffle and deshuffle operations themselves are always assigned to the GPU, since it can perform these operations much faster than the CPU, since the GPU has a much higher memory bandwidth than the CPU.

However, if a naïve approach to shuffling or deshuffling is taken, it results in bank conflicts on the GPU, under-utilizing the high memory bandwidth available on the GPU. Thus the challenge is to perform the shuffle and deshuffle operations on the GPU, without incurring any bank conflicts.

We propose a technique to perform the shuffle and deshuffle operations on the...
4.7. The Shuffle and Deshuffle Operations

GPU without incurring any bank conflicts at the device memory by staging the shuffle set into the shared memory of the SMs and incurring all bank conflicts at the shared memory. These conflicts are 1-cycle conflicts, as compared to device memory conflicts, which take 600–800 cycles to resolve [3], and thus do not cause significant performance issues. To ensure that all accesses to device memory are coalesced, we must shuffle (or deshuffle) at least warpsize × warpsize elements at one go, since this would mean that there are at least warpsize elements that are contiguous in the serial (shuffled) ordering. Thus, we define an instance of a shuffle or deshuffle operation to operate on 1024 elements. We also scale the steady state execution counts of all filters to ensure that the number of elements exchanged on each buffer in one steady state iteration is a multiple of 1024. These instances, which shuffle or deshuffle 1024 elements, form the basic schedulable unit for the shuffle and deshuffle operations.

Figure 4.4 describes the shuffle operation on the GPU. In the example shown in the figure, we shuffle a serially laid out buffer that contains 1024 elements. For simplicity, we assume that the number of elements exchanged across the buffer over one entire steady state iteration is also 1024. Larger buffers are shuffled in a similar fashion. The
addresses shown in the figure only indicate the offset from a suitably aligned start address. As can be seen in Figure 4.4, the serial buffer is accessed such that every warp accesses a set of contiguous addresses while reading from the input device memory. However, the writes by these warps into the shared memory cause bank conflicts. Once the entire buffer has been brought into the shared memory, they are then written out onto the output buffer using a series of coalesced writes. Thus all bank conflicts are incurred at the shared memory instead of the device memory. As mentioned earlier, stalls due to non-coalesced accesses to shared memory take 1-cycle to resolve, which is significantly lower than the 600 – 800 cycles required to resolve a stall at the device memory.

Figure 4.5 describes the deshuffle operation, which is similar to the shuffle operation, except that the data is first staged into shared memory using a series of coalesced reads, which also write into the shared memory in a coalesced manner. The data in the shared memory is then written such that all bank conflicts are caused only when the threads read from the shared memory due to the strided memory access pattern by the threads, but no bank conflicts are caused when the threads write to the device memory in a coalesced manner.

4.8 Modulo Scheduling

We now describe the modulo scheduling algorithm that we use assign the instances of the filters to stages in the software pipelined kernel in order to ensure correct execution. The stage assignment process essentially serves to set up the iteration differences between the various filters, while taking into account the DMA transfer latency and the shuffle and deshuffle latencies associated with the edges of the stream graph. Essentially, our algorithm considers nodes of the stream graph in a topologically sorted ordering. This is always possible since the only structure that can cause the StreamIt graph to be cyclic is the feedback-loop structure. As mentioned earlier we assume that all feedback-loop structures in the StreamIt graphs have been collapsed to a single stateful filter. Thus assigning stage numbers to filters according to their topological sort order ensures that each node is assigned a stage only after all its predecessors have
been assigned stages. Our method assigns a stage number to each node based on the following rules:

1. For an edge \((u, v)\) if \(u\) is assigned to the CPU and \(v\) is assigned to the GPU or vice versa, then \(\text{stage}(v) \geq \text{stage}(u) + 2\).

2. For an edge \((u, v)\), if both \(u\) and \(v\) are assigned to the CPU or to the GPU, then \(\text{stage}(v) \geq \text{stage}(u) + 1\).

3. If an edge \((u, v)\) requires a shuffle or a deshuffle operation, then the \(\text{stage}(v)\) as computed using Rule 1 or 2 above must be further increased by one.

Rule 1 ensures that the stages of producer and consumer nodes that are across devices are separated by at least 2 to ensure that the DMA operation can be inserted in the intermediate stage. Rule 2 ensures that if the nodes are assigned to the same partition, then the stages are separated by 1, since we do not synchronize within a software pipelined kernel. Finally, Rule 3 ensures that if a shuffle or deshuffle operation is required on an edge, then the stages of the producer and consumer nodes are separated by 1 stage more than that mandated by Rule 1 or 2, so that a shuffle operation can then be inserted in the intermediate stage. The software pipelined kernel shown in Figure 4.2 obeys these rules.

Once the stages for the nodes have been assigned, the stages to the DMA operations and the shuffle operations are assigned. Our method assigns stages to the DMA and shuffle operations such that DMA transfers \textit{into} the GPU occur \textit{As Late As Possible (ALAP)}, while DMA transfers \textit{out of} the GPU occur \textit{As Soon As Possible (ASAP)}. This is done to conserve the limited memory space available on the GPUs at the cost of increased memory usage in the CPU address space.

It is important to note that this algorithm will fail if there are cycles in the StreamIt graph, since no topological sort is possible in that case. While StreamIt has a \textit{feedback loop} construct, none of the benchmarks distributed with the StreamIt tool-chain utilize this construct. We assume that even if the StreamIt graph contains feedback loops, they can be \textit{fused} \cite{25} into a single stateful filter before the modulo scheduling algorithm is invoked.
4.9 Code Generation

The StreamIt compiler is a source to source compiler, capable of generating C-like code. We have modified the compiler to generate CUDA code for the filters assigned to the GPU and C code for the filters assigned to the CPU. The compiler also takes the buffer layout into consideration while generating code for the CPU and the GPU. The CUDA code generated by the StreamIt compiler is then compiled to native code by the `nvcc` and `gcc` compilers for the GPU and CPU respectively.

The code generation scheme we use is the *predicated kernel only* code schema described in [54]. Both the GPU code and the CPU code have a local variable that acts as the *staging predicate*, allowing each stage to be turned on or off as required, for the prologue and the epilogue of the software pipelined schedule.

For each CPU core, we create a CPU *worker* thread at the beginning of the program execution. We also create one other thread on the CPU cores, designated as The *main thread* which is responsible for orchestrating the execution of all the CPU threads, the DMA transfers and the GPU calls. It executes the following actions in a loop:

1. Shift the staging predicate appropriately if executing in the prologue or the epilogue phase.
2. Issue an asynchronous call to execute the GPU kernel.
3. Wake up the worker threads.
4. Issue asynchronous calls to initiate the necessary DMA transfers.
5. Wait for completion of the worker threads and the GPU kernel call.

The CPU code is also predicated by the *threadid* such that each worker thread executes *only* the work assigned to it. Further, the threads are *pinned* to the CPU cores to ensure that the scheduler does not migrate them to another processor. These worker threads perform the following actions indefinitely:

1. Wait for work.
2. Execute the kernel.
3. Notify completion to the main thread.

Each SM on the GPU has access to the index of the block it is executing through a variable called `blockIdx`. We set the number of blocks to match the number of SMs on the GPU. Thus the `blockIdx` variable has a one-to-one mapping with the SMs and can be used to separate the code to be executed by each SM.

The code generation scheme also performs an additional post-processing pass after the modulo scheduling to reduce the number of function calls to be executed on the GPU and the CPU. Specifically, if an SM in the GPU (or a core on the CPU) is to execute \( n \) contiguous instances \( m, m+1, \ldots, m+n-1 \) of a given filter \( v \), then it is compiled as a single call to the work function of \( v \), rather than \( n \) calls. This has been done because the compile time of the `nvcc` compiler that we use to compile the CUDA code for the GPUs increases exponentially with the length of the GPU kernel code. Thus, reducing the number of calls to the filter work functions on the GPU, greatly reduces the compile time and memory requirements of the `nvcc` compiler. Using this technique, we were able to compile within a few minutes, programs that had previously caused the compiler to run out of memory after trying to compile for a few hours.

### 4.10 Experimental Evaluation

We now present the results of the experimental evaluation of our methodology. We begin by describing the experimental setup and then present the experimental results and discuss them in detail.

#### 4.10.1 Experimental Methodology

We have implemented both the ILP partitioner as well as the heuristic partitioner and the simple software pipelining method described in Section 4.8 in the StreamIt compiler framework version 2.1.1, publicly available from the StreamIt website [9]. The compiler emits CUDA and C code which are then compiled by the `nvcc` and `gcc` compilers for the GPU and CPU respectively. All experiments reported in this section were performed on machines with two quad core Intel Xeon E5440 processors running at 2.83 GHz with 16GB of FB-DIMM RAM and a graphics card based on the GeForce 8800 GTS.
Table 4.2: Details of the Benchmarks Evaluated

| Benchmark        | Filters | $|C_{fixed}|$ | Description                                           |
|------------------|---------|----------|------------------------------------------------------|
| Bitonic          | 58      | 0        | Bitonic sorting network for sorting 8 integers       |
| Bitonic-Rec      | 61      | 0        | Recursive Implementation of the bitonic sorting network |
| ChannelVocoder   | 55      | 1        | Vocoder Implementation                               |
| DCT              | 40      | 0        | 8x8 DCT Implementation                               |
| DES              | 55      | 0        | Implementation of the DES crypto algorithm           |
| FFT-C            | 26      | 0        | Coarse Grained FFT                                   |
| FFT-F            | 99      | 0        | Fine Grained FFT                                     |
| Filterbank       | 53      | 0        | Filter bank for multi-rate signal processing         |
| FMRadio          | 67      | 0        | Software FM Radio with equalizer                     |
| MatrixMult       | 43      | 0        | Blocked matrix multiply                              |
| MPEG2Subset      | 39      | 1        | A subset of the MPEG2 Decoder                        |
| TDE              | 29      | 0        | Time Delay Equalization phase from Ground Moving Target Indicator |

512 GPU with 512 MB of device memory. The Linux operating system, with kernel version 2.6.24 was installed on these machines. The GPUs were driven by the NVIDIA driver version 180.11 and the CUDA 1.1 tool-chain was used to build the applications. Table 4.2 provides the details of each benchmark that was evaluated. The benchmarks were all taken from the StreamIt benchmark suite [9] and the benchmarks are the same as the benchmarks evaluated in Chapter 3 except that some benchmarks which were not supported by the approach described in Chapter 3 have been added.

For the ILP partitioner, the task partitioning problem was allowed to run until completion, with no time limits, since it is a minimization problem and we would like to have the minimal II possible. For the instance partitioning using the ILP approach, the problem is a constraint solution problem. We thus allotted a time of 60 seconds for the
4.10. Experimental Evaluation

solver to find a solution at the minimal II obtained from the solution to the task partitioning problem. If the solver was unable to find a solution within 60 seconds, we relax the II by 0.5% and try solving the instance partitioning problem again. We repeat this process until we obtain a valid instance partitioning.

We report the performance results of our approach using the speedup over a single threaded CPU execution as a metric. We define the speedup as $speedup = \frac{t_{cpu}}{t_{gpu}}$, where $t_{cpu}$ is the time taken for a single threaded CPU execution and $t_{syn}$ is the time taken for a synergistic execution across the GPU and the CPU cores, with both executions performing an identical amount of work. The single threaded CPU code was generated by the uni-processor backend of an unmodified StreamIt compiler and compiled with gcc with an optimization level of -O3.

4.10.2 Experimental Results

We evaluate the following aspects of the proposed compilation methodology:

1. The performance and accuracy of the heuristic partitioner as compared to the ILP partitioner in terms of the II achieved by them.

2. The performance of the code generated with the heuristic partitioner in comparison with the performance of the code generated with the ILP partitioner in terms of actual execution time.

3. Comparison of the synergistic approach to simpler heuristics.

4. The scalability of the approach: Performance of the approaches when applied to the compilation for a Tesla C1060 GPU.

4.10.2.1 Efficiency of the Heuristic Partitioner

The II values obtained from the ILP and heuristic task partitioner are compared in Table 4.3. Note that these are static II values determined by the partition and used by the compiler framework to construct the modulo schedule. The partitioning was carried out assuming 4 CPU cores and 16 GPU SMs. The results indicate that the heuristic partitioner performs quite well with an average increase in II of 9.05%. Further, the
Table 4.3: Performance of the Heuristic Partitioner Compared to the ILP Partitioner in Terms of Achieved II

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>II (ILP) (ms)</th>
<th>II (Heur)(ms)</th>
<th>%Degrade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitonic</td>
<td>0.07878</td>
<td>0.08270</td>
<td>4.97</td>
</tr>
<tr>
<td>Bitonic-Rec</td>
<td>0.1206</td>
<td>0.14397</td>
<td>19.4</td>
</tr>
<tr>
<td>ChannelVocoder</td>
<td>8.9430</td>
<td>10.1270</td>
<td>13.24</td>
</tr>
<tr>
<td>DCT</td>
<td>1.6550</td>
<td>1.7472</td>
<td>5.57</td>
</tr>
<tr>
<td>DES</td>
<td>0.4262</td>
<td>0.4546</td>
<td>6.67</td>
</tr>
<tr>
<td>FFT-C</td>
<td>0.3310</td>
<td>0.4050</td>
<td>22.37</td>
</tr>
<tr>
<td>FFT-F</td>
<td>0.4283</td>
<td>0.4433</td>
<td>3.48</td>
</tr>
<tr>
<td>Filterbank</td>
<td>0.7290</td>
<td>0.7858</td>
<td>7.79</td>
</tr>
<tr>
<td>FMRadio</td>
<td>0.2080</td>
<td>0.2170</td>
<td>4.34</td>
</tr>
<tr>
<td>MatrixMult</td>
<td>1.2997</td>
<td>1.4229</td>
<td>9.48</td>
</tr>
<tr>
<td>MPEG2Subset</td>
<td>1.9188</td>
<td>1.9913</td>
<td>3.78</td>
</tr>
<tr>
<td>TDE</td>
<td>14.6469</td>
<td>15.7518</td>
<td>7.54</td>
</tr>
</tbody>
</table>

Increases in II are well within 10% for 9 out of 12 benchmarks. A few benchmarks show higher degradations, up to 22.3%. The large degradation in Bitonic-Rec is primarily due to the fact that the II is extremely small. The ChannelVocoder benchmark is an interesting case. The optimal solution in this case is one which does not involve any shuffle or deshuffle operations. Thus, any deviation from the optimal would incur some shuffle or deshuffle operations and would therefore increase the II. Although our heuristic takes the shuffle operations into account while calculating the II, it fails to obtain a solution which does not require any shuffle or deshuffle operations. FFT-C owes its large degradation to the fact that it is a very small application, with only 26 filters and a relatively small II. Increasing the coverage threshold, mentioned in Algorithm 3 to 5 results in an improvement for FFT-C bringing down the degradation to about 10%.

Next we report the execution times for the partitioning using the ILP formulation and the heuristic partitioner. The heuristic partitioning method took a total of 35.9 seconds for all the benchmarks. This is over two orders of magnitude lower than the time taken by the ILP solver, which took a total of 2379 seconds over all the benchmarks.
4.10. Experimental Evaluation

Figure 4.6: Performance of the ILP vs. Heuristic Partitioner

Also, given that the heuristic partitioner has been implemented in Java (which is the language used for the StreamIt compiler), the implementation and hence the execution time could be optimized further.

4.10.2.2 Execution Using the Heuristic and ILP Approaches

Having demonstrated that the heuristic partitioner compares favorably with the ILP partitioner in terms of the II achieved, we now compare the actual execution time of the code generated by both techniques. We apply the appropriate level of coarsening as described in Section 3.8 in Chapter 3. A coarsening of n essentially executes the software pipelined kernel n times, thereby amortizing the cost of the GPU kernel launch and the cost of waking up the worker threads. The numbers reported in Figure 4.6 are all for the best coarsening, which may vary across benchmarks and depends on the buffer requirements of each benchmark.

Figure 4.6 shows the results with the applications being compiled for 4 CPU cores and 16 SMs. Clearly, the heuristic partitioner performance is quite comparable to the ILP partitioner. It is surprising that in the case of the Bitonic-Rec, FFT-C, Filterbank and FMRadio benchmarks, the heuristic partitioner actually performs better than the ILP partitioner, albeit by a very small margin. This is despite the fact that the IIs achieved...
Chapter 4. Synergistic Execution of Stream Programs

by the heuristic partitioner are larger than the IIs achieved by the ILP partitioner for these benchmarks. We attribute this to two factors. First, as demonstrated in [55], the dynamic factors which are not considered by the ILP formulation could be affecting the performance. For instance, neither the ILP nor the heuristic partitioner takes second order effects into account, such as the interaction between filters executing concurrently on the different SMs of the GPU or on the different cores of the CPU. While the profile runs try to model the contention for bandwidth by executing the same filter on all the GPU SMs or CPU cores, it is impossible to predict how the execution of another filter on a different core or SM would interact with the execution of a filter on a given core or SM. Secondly, as mentioned in Section 4.9, we perform a post-processing pass to reduce the number of device calls. The ILP partitioner tends to assign instances to the CPU cores or SMs in an arbitrary fashion, whereas the Metis partitioning tends to keep consecutive instances of filters together on the same core or SM. This results in a large number of calls being merged into a single call, when the heuristic partitioner is used, thereby eliminating the overhead of some function calls. However, the ILP partitioner on the other hand, attempts to achieve a better load balancing without giving any consideration for successive instances being assigned to the same CPU core or GPU SM. As a result, the ILP partitioning is unable to get much benefit from the post-processing optimization pass for reducing the number of function calls. Both these factors account for the heuristic partitioning performing slightly better than the ILP partitioner in some cases. Overall, the ILP partitioner yields a geometric mean speedup of 6.84 over the single threaded execution, while the heuristic partitioner yields a geometric mean speedup of 6.68 across the entire benchmark suite.

4.10.2.3 Comparison with Simpler Heuristics

We now compare the performance of the synergistic partitioning approach with other approaches. Specifically, we evaluate the performance of a naïve partitioning where only the nodes that are in $C_{\text{fixed}}$ are executed on the CPU cores, while all other nodes are executed on the GPU. We refer to this approach as “Mostly GPU”. For programs without stateful filters, this is the same partitioning as done in Chapter3. In the second scheme, called “CPU Only”, all the nodes are partitioned for execution on the CPU.
cores. In all cases, we report speedups relative to single threaded CPU execution.

Figure 4.7 compares the performance of the three schemes. In most of the benchmarks, our synergistic heuristic execution performs significantly better than the CPU-Only and Mostly-GPU approaches, resulting in up to 51.96X speedup and a geometric mean speedup of 6.68X across the benchmark suite over a single threaded CPU execution. Compared to this, the CPU-Only and Mostly-GPU schemes achieved respectively a speedup of 1.44X and 5.62X only. Thus, the synergistic execution yields an 18.9% improvement over the Mostly-GPU approach.

Next, we discuss the performance of the CPU-Only scheme in detail, where our instance partitioning approach yields an average speedup of 1.44X. The Bitonic and Bitonic-Rec benchmarks perform poorly. These benchmarks are extremely bandwidth intensive. The only work that the filters in these benchmarks do is to compare and exchange values, with no other computation. Thus, these benchmarks will hit the bandwidth limitations with any scheme. Other benchmarks, notably DES and FFT-E, show a performance degradation (speedups less than 1) when compiled for multiple CPUs. We attribute this to the increased number of cache misses due to the partitioning being done across the CPU cores in a cache oblivious fashion. The baseline — the single threaded CPU execution — executes all the filters in a single appearance schedule on

![Figure 4.7: Comparison of Synergistic Execution with other schemes](image)
the same core. Thus the producer-consumer locality results in a large degree of cache reuse, which leads to better performance in a single threaded CPU execution. Our partitioning scheme partitions the instances of the filters without considering where the producer instances of the filter are scheduled. Benchmarks like MatMul, MPEG2Subset and TDE display no speedup at all for the CPU Only scheme for the same reason. It has been demonstrated in [60] that cache aware optimizations on StreamIt programs can yield large performance gains. Benchmarks with peeking filters, such as Filterbank, FMRadio and ChannelVocoder, yield large speedups on the CPU. While the cache oblivious partitioning hurts the performance of these programs too, the effect is more than offset by the fact that executing them 128 times or more at one go allows for a large amount of cache reuse owing to the peek set of these filters.

Our primary objective in this work has been to build a synergistic execution framework for executing stream programs across the GPU and the CPU cores. We have not taken cache considerations into account during the CPU partitioning phase. The cache considerations are irrelevant for the GPU, since they do not have a multilevel memory hierarchy in general. Our framework can easily be extended to cover this by considering two versions of CPU profile data for each filter — one with a cold cache and the other with a warm cache — and using the appropriate versions of the profile data to yield an optimal partition. With this, the proposed approach can also be used for executing StreamIt programs on multi-cores.

4.10.2.4 Scalability Study: Compiling for the Tesla C1060 GPU

We now study the scalability of the proposed approaches for synergistically executing stream programs across the GPU SMs and the CPU cores by studying the performance of each scheme on a Tesla C1060 GPU. The Tesla C1060 GPU has the same basic architecture as the GeForce 8800 GTS 512. The key differences are each SM in the Tesla C1060 is equipped with 16384 32-bit registers, while the 8800 GTS 512 has only 8192 32-bit registers per SM. The Tesla C1060 consists of 30 SMs as compared with the 16 SMs of the 8800 GTS 512. The Tesla C1060 also has a 512-bit interface to the 4GB of device memory, while the 8800 GTS 512 has a 256-bit interface to its 512MB of device memory.
Table 4.4: Performance of the Heuristic Partitioner compared to the ILP partitioner in terms of achieved II on a Tesla C1060

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>II (ILP) (ms)</th>
<th>II (Heur)(ms)</th>
<th>%Degrade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitonic</td>
<td>0.6051</td>
<td>0.6366</td>
<td>5.2</td>
</tr>
<tr>
<td>Bitonic-Rec</td>
<td>0.1084</td>
<td>0.1109</td>
<td>2.28</td>
</tr>
<tr>
<td>ChannelVocoder</td>
<td>7.2246</td>
<td>7.6567</td>
<td>5.98</td>
</tr>
<tr>
<td>DCT</td>
<td>1.3275</td>
<td>1.5780</td>
<td>18.87</td>
</tr>
<tr>
<td>DES</td>
<td>0.5155</td>
<td>0.5597</td>
<td>8.59</td>
</tr>
<tr>
<td>FFT-C</td>
<td>0.7603</td>
<td>1.2178</td>
<td>60.18</td>
</tr>
<tr>
<td>FFT-F</td>
<td>0.3150</td>
<td>0.3400</td>
<td>7.92</td>
</tr>
<tr>
<td>Filterbank</td>
<td>0.6068</td>
<td>0.6386</td>
<td>5.24</td>
</tr>
<tr>
<td>FMRadio</td>
<td>0.1646</td>
<td>0.1769</td>
<td>7.48</td>
</tr>
<tr>
<td>MatrixMult</td>
<td>0.8321</td>
<td>0.9790</td>
<td>17.66</td>
</tr>
<tr>
<td>MPEG2Subset</td>
<td>1.2778</td>
<td>1.3378</td>
<td>4.7</td>
</tr>
<tr>
<td>TDE</td>
<td>8.9769</td>
<td>9.8364</td>
<td>9.58</td>
</tr>
</tbody>
</table>

For evaluating the performance on the Tesla C1060 we used a test-bed consisting of dual quad core Intel Xeon 64-bit processors running at 2.83GHz. The machine was connected to a 1U Tesla S1070 unit, which internally consists of four Tesla C1060 GPUs, each of which can be individually utilized. We performed our experiments using 4 CPU cores and only one of the four Tesla C1060 units in the S1070. This is due to the fact that each of the four Tesla C1060 GPUs in a Tesla S1060 unit is treated as a distinct GPU, with no hardware support for data transfer from one GPU to another in the Tesla S1070 unit. As such, if multiple Tesla C1060 units in the Tesla S1070 are to be used, the problem becomes one of scheduling across multiple GPUs, which although is an interesting area of future work, is beyond the scope of this thesis.

Table 4.4 compares the II achieved by the ILP partitioner and the heuristic partitioner. Note that these numbers are provided in order to compare the heuristic and ILP partitioners only. Comparing these numbers with Table 4.3 would not be fair, since the work done in one II in the two cases might not be the same owing to our approach increasing the multiplicity of the schedule if any one filter proves to be a bottleneck;
Figure 4.8: Performance of the partitioning schemes on the Tesla C1060

i.e., the execution time of the filter exceeds the MII that has been obtained, making it impossible to find a schedule at the MII. As can be seen from Table 4.4, the heuristic approach performs well with the exception of FFT-C, where the heuristic approach yields a 60.18% degradation over the optimal solution. Increasing the threshold mentioned in Algorithm 3 to 5 reduces the degradation to under 30%. Overall, the heuristic partitioner yields an average degradation of 12.81% across the entire benchmark suite. The ILP partitioner took a total 2143 seconds across the entire benchmark suite, while the heuristic partitioner required a total of only 38 seconds across the benchmark suite. The solution time for the ILP does not increase much even when the number of SMs is increased. Also, the heuristic algorithm provides reasonable performance while requiring a couple of orders of magnitude less time than the ILP solver, even when the number of SMs is large.

We now present the performance results obtained by executing the programs on a platform equipped with a Tesla C1060. The rest of the parameters of the test-bed are the same as mentioned earlier. As before, all the benchmarks were compiled to target four CPU cores. Figure 4.8 shows the performance obtained from the ILP partitioner, the heuristic partitioner and a naïve partitioner that partitions all the filters except the filters with persistent state onto the GPU. The Bitonic and Bitonic-Rec benchmarks show
4.10. Experimental Evaluation

Figure 4.9: Comparison of the Performance of our approaches on a Tesla C1060 and a GeForce 8800 GTS 512

very low speedups again, owing to them being memory bandwidth intensive and also due to the fact that the II values for these benchmarks are extremely small as can be seen in Table 4.4. For the rest of the benchmarks the heuristic partitioner yields good performance. For the ChannelVocoder benchmark, the heuristic partitioner performs slightly better than the ILP partitioner. As explained in Section 4.10.2.3, this is due to second order effects such as the exact interleaving of the filters, which are not considered in the ILP formulation. Also, the heuristic partitioner assigns filter instances in a regular fashion as opposed to the fairly random assignment by the ILP partitioner so that a large number of device function calls are eliminated, yielding some performance benefits. Overall, the heuristic partitioner performs quite well, yielding a geometric mean speedup of 8.73X across the entire StreamIt benchmark suite, which is significantly better than the speedups of 8.25X and 9.41X achieved by the Mostly GPU and ILP partitioning strategies respectively.

We provide a comparison of the performance of our scheme in terms of execution time on both the Tesla C1060 and the GeForce 8800 GTS 512 in Figure 4.9 for ease of reference.

The results on the Tesla do not scale proportionately with the number of SMs due
to the following reasons:

1. The SMs in the 8800 GTS 512 are clocked at 1625 MHz, while the SMs in the Tesla C1060 are clocked at a lower speed of 1300 MHz, possibly due to the more complex floating point unit on the Tesla, which supports double precision arithmetic as opposed to the floating point unit on the 8800 GTS, which supports only single precision arithmetic.

2. The overhead of launching a kernel on the Tesla is higher than on the 8800 GTS 512, possibly owing to the fact that the Tesla is connected to the host system through an HBA card on a 16X PCI-E bus rather than directly to the PCI-E bus as is the case with the 8800 GTS 512.

3. The 8800 GTS 512 has a memory bandwidth of 64 GB/s, while the Tesla C1060 has a memory bandwidth of 102 GB/s. A proportional increase in memory bandwidth with the number of SMs would be 120 GB/s, which is not the case.

To conclude, both the ILP partitioner and the heuristic partitioning approaches result in a balanced task partitioning and yield a synergistic execution of StreamIt programs across the GPU SMs and the CPU cores. Further, we have demonstrated that our approaches scale well even when the number of SMs is almost doubled, yielding good performance on the Tesla C1060 accelerator.

4.11 Summary

We have presented a framework for compiling StreamIt programs synergistically across both the CPU cores and the GPU SMs. The partitioning problem was formulated as an ILP, which models the DMA channel as a resource and thus takes into account the finite DMA bandwidth available on the system. The ILP formulation also considers the necessary buffer transformation operations when data is to be moved from the CPU address space to the GPU address space or vice-versa. We have also presented a heuristic algorithm for the partitioning of the StreamIt program. The heuristic partitioning scheme performs well in comparison with the optimal solutions obtained by the ILP and yields partitions which are within 10% of the optimal, for nine out of the twelve
4.11. Summary

benchmarks evaluated, while requiring 2 – 3 orders of magnitude lesser time than the ILP partitioner. In terms of actual execution times as well, the heuristic partitioner performs well, yielding a 6.68X speedup on an average across the StreamIt benchmarks over an optimized single threaded CPU execution while the optimal partitions yield a 6.84X speedup.

Our evaluation of the approach on a platform with a Tesla C1060 GPU indicates a geometric mean speedup of 8.73X and 9.41X over an optimized single threaded CPU execution using the ILP partitioning technique and the heuristic partitioning technique respectively. The partitioning time for both the ILP and the heuristic approach is not significantly affected by the increased number of SMs on the Tesla C1060 as compared to the GeForce 8800 GTS 512.
Chapter 5

Related Work

We now discuss some work related to this thesis. Specifically, we discuss the earlier work on stream programming languages and frameworks in Section 5.1, while processor architecture proposals that are suited for stream programming are discussed in Section 5.2. Prior effort on programming on GPUs is discussed in Section 5.3.

5.1 Stream Programming

There is a significant body of research related to stream programming paradigms, languages and frameworks. A survey of stream programming can be found in [61]. LUSTRE [17] is a synchronous data flow language that defines the output at a time $t$ as a function of the inputs available at or before time $t$. In other words the output is defined by a set of equations which depend on the input received only at or before time $t$. LUSTRE programs are conservatively analyzed for deadlocks during compile time.

SIGNAL [24] is a programming language designed to program real-time systems again using a synchronous data flow model. The representation of a system in SIGNAL is designed to be very close to the actual mathematical or graphical representation of the real-time system. Both LUSTRE and SIGNAL have a multiform notion of time rather than rely on a centralized clock.

The ESTEREL language [12, 15] is a real-time imperative language for describing concurrent systems. The aim of ESTEREL is to develop a rigorous formal model of real-time computation, while offering semantics that simplify the programming of real-time
systems. All these languages are designed with verifiability in mind and are not very well optimized. Also, these languages tend to be more control-oriented than StreamIt.

The STREAM language \[19\] which is also based on the dataflow model of computation has been proposed for concurrent programming. It is designed for formally specifying, reasoning about and transforming hardware designs at the conceptual, register and gate level. As such, the constructs and primitives of STREAM are best suited for description and easy verifiability of hardware implementations of the algorithm unlike StreamIt.

Lee et. al. have developed the Ptolemy framework \[22, 44\], which is a simulation and development framework for embedded and real-time systems. The framework also includes support for the Synchronous Data Flow (SDF) model of computation, along with static and dynamic scheduling algorithms for SDF graphs. The SDF model is less restrictive than the StreamIt model, since filters are allowed to have any number of inputs and outputs. Thus, it is possible to construct SDF graphs which have no legal schedule. The semantics of StreamIt do not allow for the creation such graphs, except for the feedback loop construct, which could result in a deadlock, if too few elements are buffered on the back edge.

Gao et. al. have studied a variant of SDF called Regular Stream Flow Graphs (RSFG) and demonstrated how to construct RSFGs such that it is always possible to schedule them at compile time \[23\]. Govindarajan et. al. have focused their efforts on scheduling RSFGs so as to obtain rate-optimal schedules on an unbounded number of processors using a polynomial time algorithm as well as obtaining schedules that have minimum buffer requirements as well as being rate-optimal using a Linear Programming formulation.

StreamC and KernelC \[49\] are extensions to the C programming language specifically designed for the Imagine \[11, 33, 39\] stream processor. The two languages complement each other, with StreamC controlling the dispatch of data parallel kernels, which themselves are written in KernelC to the ALU clusters in the Imagine processor, which is described in some detail in Section \[5.2\].

The StreamIt language \[65\] has recently revived the interest in the dataflow model
of computation. Techniques for compiling StreamIt on to the RAW [64] have been studied in [25, 26]. Compilation strategies for compiling StreamIt for the CellBE accelerator architecture have been proposed in [40, 69]. However, none of these approaches are directly applicable for compiling StreamIt programs targeting GPUs, since the GPU is a highly data parallel, multi-threaded device as opposed to the CellBE and RAW architectures which are essentially MIMD architectures with specialized interconnects.

General optimization techniques for StreamIt programs have also been studied. Lamb et al. [42] have proposed the linear analysis of StreamIt programs and transformation of the computation into the frequency domain, whenever beneficial to speedup the computation by a reduction in the floating point operations. Agrawal et al. [10] have extended the framework for the analysis of stateful filters. These optimizations are complementary to the work done in this thesis and may also be applied to the approaches studied in this thesis for greater performance benefits. For instance, the linear analysis and frequency domain transformations studied in [42] may be effectively applied to the compilation of StreamIt programs on GPUs using the existing optimized frameworks such as CUFFT [18] for performing FFT and inverse FFT on the GPU. Sermulins et. al. have proposed cache optimizations for StreamIt programs for uni-processors [60]. These techniques may be extended to allow for cache aware partitioning of StreamIt programs across the CPU cores as mentioned in Chapter 4.

5.2 Stream Architectures

With general purpose processors reaching the limits of performance for single threaded applications and due to their inherent inability to exploit the characteristics of stream programs to deliver performance, several architectures designed from the ground up for stream processing have emerged. These architectures are characterized by:

- A high bandwidth memory subsystem, possibly with multiple levels in the memory hierarchy with software managed scratchpad memories and scratch register files.
• Simple processing cores, optimized for area, so that a larger number of them can fit into a package.

• Large distributed register files.

• Restricted instruction fetch and execution mechanisms among the processing elements.

• A high bandwidth, and often low latency, interconnect between the processing elements.

The Imagine architecture is one such proposal [11, 33, 39]. The Imagine processor consists of a set of arithmetic clusters, connected by a high speed interconnect network. Each cluster is similar in architecture to a clustered VLIW architecture with a local register file and scratchpad memory. The programming interface is a set of extensions to the C programming language called StreamC and KernelC as mentioned earlier in Section 5.1. A 64-bit stream processor for scientific computation applications has also been proposed [68], which has an architecture similar to the Imagine processor.

The TRIPS processor [37, 58, 59] is another effort at designing a high performance architecture with exposed wire delays. Although it is designed to be a polymorphous architecture and not specifically a stream processor, it is quite capable of achieving high levels of performance when executing streaming applications as demonstrated in [58].

The RAW processor [63, 64] is a tiled architecture that has been specifically designed with stream processing in mind. It consists of a set of 16 MIPS 4000 style processing cores connected by a mesh network. Each core has specialized programmable routers that are designed to carry results at a high bandwidth and low latency. As mentioned earlier, the StreamIt language has backends designed for the RAW processor [25, 26]. A compiler for translating general purpose C programs, exploiting the instruction level parallelism, for the RAW processor has been described in [46].

GPUs have recently become Turing complete computation engines capable of executing branches and loops. They can be considered as high throughput streaming processors. However, unlike the other proposals, the GPU has the advantage of being a commodity computing engine rather than advanced research prototypes, making
it ubiquitous. Truly, every computer today has a GPU for accelerating graphics and media applications. Frameworks like OpenCL are promising since their goal is to harness the power of these accelerators in a universal and platform independent manner.

5.3 General Purpose and Stream Programming on GPUs

The large computation throughput offered by GPUs have long attracted research attention focused on harnessing this throughput in a universal fashion. While several applications like ray tracing have been ported to the GPU using specialized techniques, we restrict our discussion in this section to frameworks that ease the development of applications for execution on the GPU.

One of the earliest proposals provided abstractions for encapsulating data parallel structures, a shader program and semaphores on the GPU. The programmer still had to write the code for the shader program using a low level shader language, but the task of mapping the computation to the graphics pipeline was automated. We believe this proposal to be the first to attempt automation of translating programs for the GPU. Also, it is to be noted that at the time of this proposal, graphics processors had not yet evolved into the Turing complete processing engines that they are today. They were fixed-function pipelines, with a few programmable stages called shaders. The programs that could be executed by these stages had several restrictions on the length of the program, the number of registers used and a lack of support for looping constructs.

Brook is another proposal that suggests a higher level abstraction of streams. With this approach the programmer is required to organize his computation as a set of data parallel kernels with communication streams. While the language provided an abstraction for the communication streams, the programmer still had to manage the communication between kernels using these streams.

The Accelerator framework takes the approach to an even higher level of abstraction. It proposes the use of an object oriented structure called the data parallel array. A data parallel array is similar to the standard array construct provided by most programming languages, except that its elements cannot be accessed in an indexed
manner. The Accelerator framework provided mechanisms for converting between a standard array and a data parallel array. The GPU could only operate on the data parallel array, which has predefined methods associated with it. The programmer may operate on the data parallel array using these predefined operations. The framework would generate the corresponding shader code for the operations at run-time using a lazy evaluation and execution scheme to bundle as much work as possible into a single GPU call. This approach while elegant, is not easily extensible, since the programmer would need to know the internal workings of the data parallel array, as well as write in shader language, if he wished to extend the gamut of operations provided by the data parallel array.

With the Direct3D 10 specification \cite{14} and the shift in GPU architecture towards a set of streaming processors rather than fixed function pipelines, it is no longer necessary to program in shader language to harness the computational power of GPUs. Frameworks like CUDA \cite{3} and CTM \cite{2} provide C like programming environments for programming GPUs. The OpenCL \cite{8} framework promises to provide a unified programming interface for all kinds of data parallel architectures.
Chapter 6

Conclusions

We now provide a summary of the work done as part of this thesis and also highlight some interesting areas of future research along the lines of this thesis.

6.1 Summary

In this thesis we have described a framework that software pipelines the execution of StreamIt programs for execution on the GPU. The approach harnesses the parallelism inherent in the StreamIt graphs at various levels:

- The data parallelism in the stateless filters is effectively mapped to the data parallelism offered by the hardware multi-threaded SIMD execution model of the SMs on the GPU.

- The task and pipeline parallelism in the StreamIt graph is exploited by a software pipelining approach which partitions the work across the SMs of the GPU, while setting up appropriate iteration differences between the various filters, exploiting pipeline parallelism.

We have proposed a profile-driven methodology to determine the optimal level of data parallelism for the filters of the StreamIt program for execution on the GPU.

We have formulated the partitioning and scheduling of StreamIt programs in a software pipelined schedule across the SMs of the GPU as an integrated Integer Linear Program (ILP). The ILP formulation takes into account the specific challenges posed...
by the programming model and architecture of the GPU and also schedules with the limited non-virtualized memory of the GPU in mind.

We have also proposed a novel buffer layout scheme for StreamIt programs, to effectively utilize the high bandwidth available on the GPU. The layout scheme ensures that the data parallel threads executing on the GPU access the memory in a coalesced manner. The buffer layout optimization is essential for achieving good performance on the GPU as is demonstrated by our experimental results.

We have implemented and evaluated our proposals using the StreamIt 2.1.1 compiler framework. The results indicate a maximum speedup of 33.82X, with an geometric mean speedup of 5.02X across a set of StreamIt benchmarks, over an optimized single threaded CPU execution.

In order to effectively utilize the CPU cores as well as the GPU in a synergistic manner and also to provide support for StreamIt programs with stateful filters, we extend and enhance the framework developed for synergistic execution of StreamIt programs across the CPU cores and the GPU SMs. The approach poses the following challenges:

- Effective load balancing across the GPU SMs and the CPU cores.
- The disjoint address spaces of the CPU and GPU require DMA transfers in order to communicate data from one to another.
- The buffer layout optimizations are detrimental to performance on the CPU, owing to a large number of cache misses caused by the strided access pattern when the program is executed using a single thread on the CPU cores.

In order to address these challenges, we have proposed an Integer Linear Programming formulation for the partitioning problem which models the limited DMA bandwidth as a resource and also considers the cost of transforming the buffers into the layout that is optimal for the partition where the data is accessed, whenever data is communicated between the CPU cores and the GPU.

We have also proposed a heuristic partitioning algorithm for performing the partition, which compares well with the optimal solutions obtained by the solution to the
ILP, typically yielding a partition which is under 10% from the optimal partition, while requiring 2 – 3 orders of magnitude less time to compute the partition.

In order to efficiently transform the buffers into the required layout, we have developed a novel technique that performs these operations on the GPU SMs, without incurring bank conflicts at the device memory, instead staging all bank conflicts into the much faster shared memory.

We have adapted a modulo scheduling algorithm for scheduling the computation on the respective partitions while completely hiding the DMA latency behind the computation. Our experiments using an implementation of our synergistic approach on the StreamIt compiler framework demonstrate maximum speedups of 51.96X and 70.18X over a single threaded CPU execution on the GeForce 8800 GTS 512 and the Tesla C1060 GPUs, with a geometric mean speedup of 6.84X and 9.41X respectively.

Thus we have developed and evaluated a framework for the efficient compilation of stream programs for execution on GPUs.

# 6.2 Future Work

An interesting area of future work would be extend and enhance the framework to take advantage of multiple GPUs on a single system. This poses some challenges from the perspective of communication between different GPUs, since all communications between GPUs must be orchestrated by the control thread running on the host processor and must be staged into the host memory first.

Allocation of work to the CPU in a cache aware manner, while extending the ideas presented in [60] to specifically optimize for our approach would also be an interesting idea to follow on. This could potentially improve the performance of our synergistic approach to compiling StreamIt programs. This would probably necessitate the use of multiple profile runs of the filters on the CPU. One run with a cold cache and another with a warm cache. The appropriate execution times must then be chosen by the partitioner depending on the partition being considered.

While we use profiling to determine the execution time of each filter on the GPU and the CPU, analytical modeling of these devices could help in reducing the amount
of work required for compilation. While an analytical model for superscalar CPUs has been proposed in [35], a simplified model would probably be more appropriate, given the regular structure of StreamIt programs which have high spatial locality in their access patterns, but low temporal locality owing to the low degree of reuse. On the other hand [30] proposes an analytical model for GPUs, but as of this writing, the text of the paper is not available. It is also unclear as to whether the regular access patterns owing to the stream model of computation would benefit from a simpler model. Use of such analytical models could simplify the compilation process considerably.

Another interesting area of future research is to adaptively migrate parts of the StreamIt program between the CPU cores and the GPU depending on the load on the CPU cores. This can be accomplished with the support of a run-time system and would help in adapting the framework proposed in this thesis for systems which are not dedicated for the execution of stream processing applications.
References


REFERENCES


[33] Ujval Kapasi, William J. Dally, Scott Rixner, John D. Owens, and Brucek Khailany.
REFERENCES


[41] Monica Lam. Software Pipelining: An Effective Scheduling Technique for VLIW


REFERENCES


