Compiling for a Multithreaded Horizontally-Microcoded Soft Processor Family

by

Ilian Tili

A thesis submitted in conformity with the requirements for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

© Copyright 2013 by Ilian Tili
Abstract

Compiling for a Multithreaded Horizontally-Microcoded Soft Processor Family

Ilian Tili
Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto
2013

Soft processing engines make FPGA programming simpler for software programmers. TILT is a multithreaded soft processing engine that contains multiple deeply pipelined and varying latency functional units. In this thesis, we present a compiler framework for compiling and scheduling for TILT. By using the compiler to generate schedules and manage hardware we create computationally dense designs (high throughput per hardware area) which make compelling processing engines. High schedule density is achieved by mixing instructions from different threads and by prioritizing the longest path of data flow graphs. Averaged across benchmark kernels we can achieve 90% of the theoretical throughput, and can reduce the performance gap relative to custom hardware from 543x for a scalar processor to only 4.41x by replicating TILT cores up to a comparable area cost. We also present methods of quickly navigating the design space and predicting the area of hardware configurations.
Acknowledgements

First, I want to thank my supervisor Prof. Greg Steffan for his advice, and direction during my research. I gained valuable research experience under his supervision.

I want to thank professors Natalie Enright Jerger, Jason Anderson, and Paul Chow for being part of my defense committee, and for providing feedback on my thesis. Thanks go also to Alberto Leon-Garcia for acting as a defense chair of the committee.

I also want to thank the students in our research group, and in the lab for their feedback and useful suggestions.

Finally, I would like to thank all of my family for their unconditional support throughout the years, and for their help in achieving my goals.
# Contents

1 Introduction ................................................................. 1
   1.1 TILT Processor ....................................................... 2
   1.2 TILT Compilation and Tuning ........................................ 2
   1.3 Research Goals ...................................................... 3
   1.4 Limitations .......................................................... 4
   1.5 Organization and Collaboration ..................................... 4

2 Related Work ................................................................... 5
   2.1 VLIW Soft Processors ................................................. 6
   2.2 Multithreaded Soft Processors ....................................... 7
   2.3 Scheduling .............................................................. 8

3 TILT Architecture ........................................................... 11
   3.1 Memory Architecture .................................................. 12
   3.2 Functional Units ....................................................... 15
   3.3 Interconnect ............................................................ 15
   3.4 Control ................................................................. 15
   3.5 Data Input/Output ..................................................... 16
   3.6 Predication ............................................................. 17
   3.7 Summary ............................................................... 17

4 LLVM Framework .......................................................... 18
   4.1 Instruction Parsing ..................................................... 19
      4.1.1 getelementptr (GEP) instruction ........................... 19
      4.1.2 fpext ......................................................... 20
      4.1.3 Stores .......................................................... 20
      4.1.4 Load ............................................................ 21
      4.1.5 Compares ...................................................... 21
4.1.6 Branches .................................................. 22
4.1.7 Binary Instructions ....................................... 22
4.1.8 Function Calls ............................................ 23
4.1.9 Compare Mux ............................................. 23
4.2 Dependence Tracking ......................................... 25
4.2.1 Read After Write ......................................... 25
4.2.2 Write After Read ......................................... 25
4.2.3 Write After Write ........................................ 26
4.2.4 CMP Dependences ........................................ 26
4.3 Summary ...................................................... 27

5 Compiling for TILT ........................................... 28
5.1 Scheduling Approaches ..................................... 28
5.1.1 Greedy .................................................... 30
5.1.2 Prioritized Greedy (PG) ................................. 31
5.1.3 Greedy Mix (GM) ......................................... 32
5.1.4 Prioritized Greedy Mix (PGM) ......................... 33
5.1.5 Symmetric ................................................ 33
5.1.6 Longest Path ............................................. 33
5.1.7 Grouped PGM ............................................ 34
5.1.8 Grouped PGM Longest Path ............................ 35
5.2 Benchmarks .................................................. 36
5.3 Scheduling Results ......................................... 38
5.3.1 Black Scholes ............................................. 39
5.3.2 Gaussian Blur ............................................ 40
5.3.3 N Body .................................................. 41
5.3.4 SRAD .................................................... 41
5.3.5 Hodgkin-Huxley (HH) .................................. 42
5.3.6 HDR ...................................................... 42
5.3.7 Matrix Multiply .......................................... 43
5.3.8 Matrix Inverse .......................................... 43
5.3.9 Planck Distribution .................................... 44
5.3.10 Conclusion ............................................... 44
5.4 Memory Allocator for DFG ................................. 46
5.5 Summary ..................................................... 48
### 6 Hardware Results

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Evaluation Details</td>
<td>49</td>
</tr>
<tr>
<td>6.2</td>
<td>TILT Comparisons</td>
<td>50</td>
</tr>
<tr>
<td>6.3</td>
<td>Reducing the Gap with Custom Hardware</td>
<td>50</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Base configuration</td>
<td>50</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Area &amp; Throughput Comparisons</td>
<td>53</td>
</tr>
<tr>
<td>6.3.3</td>
<td>Replication Results</td>
<td>55</td>
</tr>
<tr>
<td>6.4</td>
<td>Port Sharing</td>
<td>57</td>
</tr>
<tr>
<td>6.5</td>
<td>Schedule Compression</td>
<td>60</td>
</tr>
<tr>
<td>6.6</td>
<td>Summary</td>
<td>62</td>
</tr>
</tbody>
</table>

### 7 Navigating TILT Design Space

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>Simulator design space reduction</td>
<td>63</td>
</tr>
<tr>
<td>7.2</td>
<td>Predicting Hardware Area</td>
<td>66</td>
</tr>
<tr>
<td>7.3</td>
<td>Evaluation of architecture predictions</td>
<td>70</td>
</tr>
<tr>
<td>7.3.1</td>
<td>Evaluating Compute Density Prediction</td>
<td>71</td>
</tr>
<tr>
<td>7.4</td>
<td>Summary</td>
<td>73</td>
</tr>
</tbody>
</table>

### 8 Conclusions

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1</td>
<td>Contributions</td>
<td>74</td>
</tr>
<tr>
<td>8.2</td>
<td>Future Work</td>
<td>75</td>
</tr>
</tbody>
</table>

Bibliography | 76   |
### List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>FU Usage Count Statistics.</td>
<td>36</td>
</tr>
<tr>
<td>5.2</td>
<td>Statistics on DFG Nodes.</td>
<td>38</td>
</tr>
<tr>
<td>5.3</td>
<td>FU Latencies.</td>
<td>38</td>
</tr>
<tr>
<td>5.4</td>
<td>Memory Savings on different benchmarks.</td>
<td>47</td>
</tr>
<tr>
<td>6.1</td>
<td>Estimated Performance of Replicating TILT Architectures.</td>
<td>55</td>
</tr>
<tr>
<td>6.2</td>
<td>Details of densest TILT designs per benchmark.</td>
<td>56</td>
</tr>
<tr>
<td>6.3</td>
<td>Estimated Performance of Replicating the Overall Densest Core.</td>
<td>56</td>
</tr>
<tr>
<td>7.1</td>
<td>Simulator space reduction.</td>
<td>65</td>
</tr>
</tbody>
</table>
List of Figures

3.1 TILT Architecture Diagram ................................................. 11
3.2 Different memory configurations for TILT ............................. 13
3.3 Horizontally Microcoded Instruction Memory .......................... 16

4.1 Illustration of getelementptr (GEP) instruction. .................... 19
4.2 Illustration of fpext instruction. ........................................ 20
4.3 Illustration of Store instruction. ........................................ 21
4.4 Illustration of Compare and Branch instructions. .................... 21
4.5 Illustration of predication through CMP_MUX instructions ........ 24
4.6 Illustration of WAW dependence. ........................................ 26

5.1 Example data flow graph and schedules illustrating different scheduling approaches ................................................. 29
5.2 Scheduler results for Black Scholes. .................................... 39
5.3 Scheduler results for Gaussian Blur. ..................................... 40
5.4 Scheduler results for N Body. ............................................ 41
5.5 Scheduler results for SRAD. .............................................. 41
5.6 Scheduler results for HH. .................................................. 42
5.7 Scheduler results for HDR. ................................................ 43
5.8 Scheduler results for Matrix Multiply. ................................ 43
5.9 Scheduler results for Matrix Inverse. ................................. 44
5.10 Scheduler results for Planck Distribution. ........................... 45
5.11 Averaged scheduler results .............................................. 45

6.1 Throughput averaged across benchmarks for a base design (FU Mix 1-1-1-1-1-1) with 1 to 64 banks and depths of 128,256,512 and 1024 words. 51
6.2 Total TILT area for a base design (FU Mix 1-1-1-1-1-1) with 1 to 64 banks and depths of 128,256,512 and 1024 words. 51
6.3 Compute density (throughput/area) averaged across benchmarks for a base design (FU Mix 1-1-1-1-1-1) with 1 to 64 banks and depths of 128,256 and 512 words. ................................................................. 52
6.4 Throughput of a soft scalar CPU, single-thread base TILT (having one bank), 4-bank base TILT (the densest overall, with multiple threads), and custom hardware implementations. ............................... 53
6.5 Area of base TILT (1 of each FU), an aggressive 12-FU instance, the densest-per-benchmark instance, and estimated area of custom hardware. 54
6.6 Area breakdown for the densest TILT configuration for each benchmark. 54
6.7 Port Sharing results for Black Scholes. ............................ 58
6.8 Illustration of schedule compression and potential savings ............. 60
6.9 Results for schedule compression on Black Scholes and Gaussian Blur benchmarks ......................................................... 61

7.1 Instruction width calculation. ........................................ 67
7.2 Error distributions for the hardware area prediction model. ............... 70
7.3 Results showing accuracy of predicting densest design. .................... 72
7.4 Accuracy evaluation for the prediction of the overall densest architecture. 73
7.5 Accuracy evaluation for the prediction of the base densest architecture. 73
Field Programmable Gate Arrays (FPGAs) are recently becoming more popular alternatives for accelerating computationally intensive applications. They have attracted commercial interest from companies such as Intel and IBM. Intel has released Stellarton \[37\], an Intel Atom processor that is integrated with an FPGA from Altera, while IBM has several products that use FPGAs such as Data Power \[36\] and Netezza \[35\].

FPGA programming requires an expert hardware designer, and long compile times are needed to produce a design or to rebuild it after making a modification. To ease the programming task for non-expert hardware programmers, systems are emerging that can map high-level languages such as C and OpenCL to FPGAs via compiler-generated circuits and soft processing engines. One approach is behavioural synthesis which converts a high-level language into a hardware circuit. This would create custom circuits for specific parts of the program’s source code. Another approach is to use soft processing engines, which are similar to CPUs, and are an example of an overlay architecture. These are familiar to programmers, can be reprogrammed quickly without rebuilding the FPGA image, and by their general nature can support multiple software functions in a smaller area than the alternative of multiple per-function synthesized circuits. Furthermore, compelling processing engines can be chosen by high-level synthesis systems to implement loops or common functions. However, for soft processing engines to be compelling they must be computationally dense, that is they must achieve high throughput per area. For simple CPUs with simple functional units (FUs), it is relatively straightforward to achieve good utilization, and it is not overly-detrimental if smaller units with latencies of a few cycles such as an integer adder are under-utilized. In contrast, processors that contain larger, more deeply-pipelined, more numerous, and more varied types of FUs can be quite challenging to keep busy.
1.1 TILT Processor

We introduce TILT (Thread and Instruction Level parallel Template architecture), which is a soft-processor that can be customized to run different floating-point applications while maximizing the hardware utilization. Since we are dealing with floating-point units that have very deep pipelines, we rely on the compiler to highly utilize the hardware by generating a schedule that uses ILP (Instruction Level Parallelism) and TLP (Thread Level Parallelism) to increase pipeline utilization while trying to keep the hardware complexity at a minimum. Our instructions are considered Horizontal Microcode (HM) since they are used to control low level signals such as multiplexers for read/write ports or sub-banks of memory. Horizontal Microcode encodes several operations in the same cycle, and is used to control low level components of the processor. HM is similar to Very Long Instruction Word (VLIW), except VLIW operates at a higher level of abstraction than HM. TILT is organized by having FUs and memory banks that are connected by a crossbar network which allows reading inputs into the FUs and writing outputs to memory banks. TILT also contains an instruction memory that controls the execution schedule.

To improve an application’s throughput on TILT we can tune the number of FUs as well as the depth and number of ports in memory banks, which allows us to exploit more ILP and TLP in an application. Since each thread executes its own instructions, adding more threads requires more instruction storage. TILT designs can be configured by changing the type and number of instances of FUs, the number and depth of memory banks, and the amount of instruction storage, which depends on the previous variables and the number of threads. This creates a large design space that is a product of all the possible configuration parameters.

There is a tension between making a faster processing engine—which can easily be achieved by adding more resources—and reducing hardware area, which comes at the expense of the performance. We decided to create a configurable architecture with minimal hardware support that requires the compiler to generate schedules, and manage resources in order to keep the hardware area at a minimum. The main challenge of this work is to design a compiler that can generate high throughput schedules and maximize hardware utilization of our soft processing engine.

1.2 TILT Compilation and Tuning

With TILT we want to find out if it is possible to reduce hardware complexity by using the compiler to statically schedule for a hardware configuration. Given a TILT configuration
and a benchmark, we want to find a schedule with the highest throughput (i.e. for a given number of threads have the shortest possible schedule). We need to find which algorithms work better for scheduling multiple threads, while respecting architectural limitations such as memory bank read and write ports. The scheduling problem involves scheduling for multiple functional units that are pipelined and have varying latencies (5-28 cycles). The number of operations issued or completed in a cycle in the same bank cannot exceed the number of read or write ports associated with that bank. This type of problem can be solved only through heuristics, as there are no efficient algorithms for it.

TILT is highly parameterized, and we can tune it to a single application or a group of applications. In either case, we are dealing with a large design space, and it would be tedious to build all designs in a brute-force manner. It would be useful if the compiler can be used to effectively predict the design space of TILT architectures, and their performance. By parameterizing the compiler with different architecture options, we can evaluate how a benchmark application will perform in a certain architecture. In addition, we have the ability to estimate the area of a TILT configuration, which helps when it is required to balance between throughput and area. Using the compiler has the advantage of saving hardware compilation time, as these calculations can be done without recompiling hardware continuously. Besides supporting single applications, we can recommend architectures that are efficient in supporting multiple applications, and such a process requires more extensive design space searches. We also evaluate how the compiler along with architecture enhancements contribute to optimizing, and improving the efficiency of TILT.

1.3 Research Goals

The objective of this work is to use the compiler, knowledge of the target application or applications, and a configurable hardware architecture to recommend a high density compute engine. Our research goals are the following:

1. to design an efficient scheduling approach for a multithreaded, multi-memory-banked, horizontally-microcoded architecture with multiple, variable-latency FUs;

2. to determine designs that will maximize hardware utilization by exploring the design space of functional unit instances, number and size of memory banks, and numbers of threads;

3. to estimate the effectiveness of TILT designs in closing the performance gap with custom hardware implementations;
4. to design a method for predicting the area and performance of different TILT configurations without performing full CAD compilation on each design.

1.4 Limitations

In this work we do not schedule for the marshalling of data between external memory and TILT memory banks. This support is ongoing work by other group members. We assume it is done in the background using spare memory ports and managed by a fetching mechanism such as CoRAM [25] or LEAP scratchpads [13]. We do not support full C code but expect the source code to be included into one C function, similar to a kernel in OpenCL. All variables have to be declared inside the function or they can be passed as function parameters. We support stack variables of floating-point type or pointers to these types of variables. Dynamically allocated variables are not supported; we need to know in advance the size of a variable. We do not yet support nested loops although they could be unrolled. We expect an outer loop whose body would be transformed into a function that would be executed by independent threads. Predication is used to support if-else branches.

1.5 Organization and Collaboration

This thesis is organized as follows. Chapter 2 discusses work related to soft processors and compiler scheduling. Chapter 3 introduces TILT’s hardware architecture and explains its main components. Chapter 4 describes the LLVM framework used to compile applications for our processor. Chapter 5 presents different scheduling approaches, their performance on benchmarks and a memory allocator algorithm. Chapter 6 shows hardware results and how these combined with the compiler can be used to navigate the design space to find efficient designs. It also evaluates architecture optimizations such as Port Sharing and Schedule Compression. Chapter 7 describes and evaluates how we can simplify and accelerate design space exploration. Lastly, Chapter 8 summarizes and draws conclusions of our work.

Part of the work presented in this thesis was implemented by fellow Master’s student Kalin Ovtcharov. He was responsible for the Verilog implementation of the processor which is described in Chapter 3, with the exception of Branch Predication which was designed by both of us. The hardware results presented in Chapter 6 and 7 were gathered by compiling his implementation of the processor with different parameters, but the analysis of these results and area prediction models were solely the author’s work. Part of the work in this thesis was published in FPL 2013 [55].
Chapter 2

Related Work

In this section we cover related work in two main fields: soft processors and scheduling. We will discuss VLIW Soft Processors and Multithreaded Soft Processors in more detail. We consider our processor Horizontally Microcoded (HM) as it has instructions that are used to perform low-level control of hardware such as multiplexers. A horizontal instruction can encode several operations that can be executed in parallel by the processor. The main difference with VLIW is the level of abstraction, VLIW operates at a higher level than HM. There are fewer implementations of HM processors, with No Instruction Set Computer (NISC) \[38\] \[61\] being one of them. NISC does not have a decoding stage and uses a set of control signals to drive the data path components. Their data path can be easily reconfigured and they take into account application requirements when allocating processor resources. We will discuss VLIW processors as they are very similar to HM, with the only difference of supporting a more high-level instruction set. Soft processors are designed for use in FPGAs, and they are easily customizable by adding or removing components to improve performance or area. NIOS \[16\] supports floating-point via an extension, and up to six stages of pipelining. MicroBlaze \[64\] is pipelined with 3 or 5 stages, and supports floating-point operations. ARM Cortex-M1 \[51\] is a 3-stage soft processor but does not have floating-point support. There are also Vector Processors that operate on vectors of data by executing the same operation on each. VESPA \[67\] contains a scalar 3-stage MIPS, and a vector coprocessor that can be tuned to reduce area and match application requirements. VIPERS \[68\] also uses a scalar core, and a configurable number of vector lanes. VEGAS \[24\], VENICE \[60\] are other vector processors that improve on performance and area of VIPERS.

Besides soft processors, High Level Synthesis (HLS) is another popular approach that operates by mapping software code to hardware that implements the same functionality. In this area there have been many academic releases: LegUp \[23\], GAUT \[45\],
ROCCC [62], xPilot [26], and commercial products such as C2H [15] and OpenCL [17] by Altera, Liquid Metal by IBM [58], and Vivado ESL Design by Xilinx [65]. HLS is application specific and it’s unlikely that the hardware can be reused by a different application unless regenerated. It is possible to integrate HLS with a processing engine so that part of the computation can be executed on the engine.

2.1 VLIW Soft Processors

VLIW processors improve performance by using a long instruction to issue multiple independent operations in a cycle. The compiler is used to find parallelism and generate the VLIW instruction, therefore it is possible to gain performance without having hardware complexity to support it.

VLIW-SCORE [40] is similar to our architecture in that it has a VLIW, pipelined architecture for utilizing floating-point units, and relies on a compiler to schedule operations via software pipelining. One key difference between the two architectures is storage organization: VLIW-SCORE organizes storage as a pair of operand memories in front of each functional unit, with a time-multiplexed network connecting functional unit outputs to operand-memory inputs; in contrast, we organize storage as banks of memory having contiguous per-thread address spaces. For VLIW-SCORE the FU mix (the number of instances of different FU types) is tuned to match the ratio of the activation counts of the different FUs in the data flow graph of the target application. We choose the FU mix based on the density of the whole design and do not decide only based on the data flow graph (DFG).

VPFPAP [44] implements variable precision floating-point arithmetic with a custom numerical representation, focusing on using VLIW instructions to comprise transcendental functions from basic arithmetic operations. They perform speedup comparison to a software library and another custom processor. Our architecture uses predefined floating-point functions from Altera, and does not focus on the implementation details. It is more focused on finding the most appropriate configuration (FU mix, number of threads, number of banks) that can balance the throughput of the application and resource requirements on FPGA.

Saghir et al. [59] support custom computation units in their datapath. The VLIW instructions are decoded in the 2nd stage (Instruction Decode), and dispatched for execution to the appropriate units. Most of their units execute in 1 cycle. They use RAM blocks to implement multi-ported register files. They show that customizing the datapath achieves higher performance but this comes at a hardware cost which must be weighed
Chapter 2. Related Work

Jones et al. [39] present an architecture where a 4-way VLIW processor is connected to a register file. Groups of several machine instructions are synthesized directly into hardware functions, achieving super-linear speedup. These hardware functions are also connected to the shared register file. For each processor there are 2 read ports, and 1 write port in the register file. Multiplexing is a problem for clock speed on their VLIW architecture, and using a shared register file is expensive. They measure the impact of multiplexer complexity of an N-wide VLIW on performance and area requirements. In TILT, we perform a broad evaluation of the design space for all the components: FU area, multiplexing, memory requirements. Our approach does not generate custom hardware for software instructions, but we want to choose the best FU configuration for the hardware, and are not completely application specific.

The ρ-Vex [18] is a VLIW that can be configured at design time by changing the number and type of functional units, the number of multiported registers, and latencies. The 2-issue machine has 2 Arithmetic Logic Units (ALUs), 2 multipliers, and also a memory and control unit. The 4 issue machine has double the resources. They use a bit stream to reconfigure the processor. The register file is implemented using BlockRAMs. Reconfiguration is used to change the functionality of the system with the goal of improving utilization based on the application. Applications with larger ILP benefit from a wider issue VLIW.

2.2 Multithreaded Soft Processors

Multithreading is useful at improving pipeline utilization by reducing stall cycles. It is more area efficient than replicating a single-threaded processor, and is a good way of utilizing long pipelines.

Moussali et al. [52] describes microarchitectural techniques for supporting multithreading in soft processors. Multithreading is implemented by the hardware that chooses operations from threads based on availability. On the other hand, in our work we use the compiler to create the multithreaded schedule. Their register file supports multiple threads using multiple ports. Each thread has its own memory bank, and it is not connected to other threads. They support custom execution units for demanding operations such as floating-point multiplication or addition.

Fort et al. [32] present a multithreaded soft processor on an FPGA, and their goal is to reduce the area usage on the FPGA. Similar to our approach it tries to maximize the compute density. Their architecture resembles a real CPU with several stages, while
ours does not have that complexity.

Labrecque and Steffan [41] showed that there are significant benefits of multithreaded soft processors over single threaded counterparts. Fine-Grained Multithreading (FGMT) is used to execute an instruction from a different thread at each pipeline stage. An area efficiency metric is used to evaluate the processors, and with multithreading, these processors no longer require hazard detection logic which can provide improvements in both area and frequency. Deep pipelines are more favourable for multithreaded soft processors.

Manjikian et al. [47] outlines the design issues for a dual-issue superscalar processor in programmable logic including the use of a multiported register file that implements replication.

CUSTARD [28] is a customizable multithreaded FPGA soft processor. It is a load/store RISC architecture supporting the full MIPS integer instruction set. Their multithreading support maintains independent thread contexts. A multiported register file is used to store data per thread. It is parameterizable in number of ports and registers per thread. The application code is statically analyzed by the compiler which generates custom instructions to accelerate the frequently performed computations by implementing them in dedicated datapaths. We do not create custom instructions but use scheduling and FU mix to improve computational density. Their multithreading is performed by having context switches between threads. Our architecture does not store state about the threads which are controlled by the compiler.

2.3 Scheduling

The goal of TILT is to reduce hardware complexity by using the compiler to schedule for variable-latency FUs with multiple threads. By moving this task to the compiler, it is more important that it efficiently schedules instructions. The problem of scheduling is well known and has been studied for several decades. Scheduling problems [22] are classified using 3 fields: machine environment, job characteristics and optimality criterion. Machine environments can be: identical parallel, uniform parallel or unrelated parallel. Job characteristics include: job preemption, precedence relations, batching, etc. The optimality criteria determines what function should be optimized, and usually minimizes the sum of completion times, delay penalty or lateness. In our case, we are dealing with a problem that executes on unrelated parallel machines (functional units) with precedence relations (data dependences between operations of the DFG), and our objective is to minimize the completion times. There are also a few additional requirements since
our functional units have different latencies, and are pipelined. A survey on scheduling has been done by Graham et al. [33]. Research has shown that resource constrained scheduling [21] and scheduling for pipelined machines [19] is NP complete for 2 machines. Therefore, heuristic methods are employed to solve this problem.

Fisher [31] describes one of the earliest list scheduling algorithms that supports priority, and is used to compile microcode for a processor. Lam [43] shows that software pipelining is a viable alternative for scheduling VLIW processors. Using software heuristics, near optimal results can be obtained for loops containing intra- and inter-iteration dependences. Software Pipelining creates code for the loop’s prologue, where a new iteration is created in each cycle; the steady state where several iterations are in progress at the same time; and the epilogue which completes the iterations that are in progress. The initiation interval of iterations must be the same and the schedule must be identical. The goal is to find the smallest initiation interval with a valid schedule. Modulo variable expansion is used to allocate multiple registers to variables that are used in loop iterations. This removes inter-iteration dependences, and lengthens the loop steady state. In our implementation we assume that each loop iteration is an independent thread, and there are no inter-iteration dependences. This allows us to assign separate memory spaces and storages to threads. Instead of generating code for prologue and epilogue, we create a fully specified schedule, resembling the steady state of a loop, which contains code for a chosen number of threads. This full schedule can be repeated in a loop to execute more iterations for a multiple number of the threads scheduled. Since our architecture can have varying area costs based on number of threads supported, we choose the best number of threads based on their effect on the whole architecture’s area. Scheduling more threads in a fully specified manner always improves throughput, approaching the ideal throughput of a schedule, but comes at an increasing area cost to support these threads.

Rau and Glaeser [56] present a scheduling algorithm for a loop body which finds the best initiation interval of a continuous loop. They also introduce the concept of a scratchpad memory and a crossbar that can send data from the output of a Processing Element to the input of another one. To enable modulo scheduling the paper introduces a polycyclic processor which uses arbitrary delays in the interconnect that allow the results to reach the Processing Element’s inputs on time for the next computation.

Iterative Modulo Scheduling [57] performs scheduling by attempting to find a valid schedule for a certain initiation interval (II), and then increases the interval until it finds a valid modulo schedule that does not violate resource and dependence constraints. A modular schedule requires the resources to be available for all the operations. In our
experiments, we schedule loops with large code footprints. In such cases it is more
difficult to find a repeating pattern in the schedule. By scheduling all threads together
(i.e. unrolling the loop body) we increase the throughput at the expense of instruction
memory.

Software Pipelining has also been applied to multicores [29] and takes into considera-
tion resource constraints, inter-thread dependences, and synchronization. List scheduling
is also used for Global Multi-Threaded scheduling [54].

Trace Scheduling [30] uses list scheduling to compact microcode by finding traces (loop
free sequences) of operations from different basic blocks, and scheduling them close to
each other to increase ILP and reduce code length. This method works with both acyclic
graphs or graphs that contain loops and it requires compensation code to be generated
to preserve code correctness. Our scheduler works on the DFG, and only preserves data
flow dependences, thus removing any information about control flow with the exception
of if-else branches that are predicated. Our approach is similar to Trace Scheduling, since
it exposes all the available parallelism in the DFG. On the other hand we do not require
profile information about the most popular trace, and prioritize based on slack time to
schedule operations.

Superblock scheduling [49] is a method based on traces. A superblock is a trace that
has no side entrances; control may leave from many exit points but only enters from one.
Superblock formation starts by finding traces, and modifies the code to eliminate side
entrances. The creation of superblocks increases the amount of ILP available, and list
scheduling is performed on the dependence graph, which achieves speedup over the base
configuration.

Scheduling for TILT is similar to previous approaches since it extracts parallelism
from the DFG, and finds a high utilization schedule by using Horizontal Microcode. Our
scheduling is based on list scheduling, but it is extended to support TILT configuration
parameters and architectural constraints; we schedule multiple independent threads for
deeply pipelined FUs and deal with benchmarks with large loop bodies.
Chapter 3

TILT Architecture

Figure 3.1: The TILT Architecture, consisting of a banked multi-ported memory system and FUs connected by crossbar networks. Execution is controlled by an instruction memory.

TILT (Thread and Instruction Level parallel Template architecture) is an FPGA-based compute engine designed to highly-utilize multiple, varied, and deeply-pipelined FUs, assuming a programming model that provides many threads. Our approach to achieving high utilization in TILT is to leverage (i) thread-level parallelism (TLP), (ii) instruction-level parallelism (ILP), and (iii) static compiler analysis and scheduling. TILT is highly-configurable, allowing us to explore many architectural possibilities, and to discover the configurations that lead to the best utilization and performance. Our goal is to find the densest architecture, which is the one that maximizes throughput while minimizing hardware area. To avoid unnecessary hardware complexity we want to use the compiler to predict the appropriate hardware configuration, and allow it to generate a static schedule without requiring a hardware scheduler. We face a very large design
space that involves multiple Arithmetic Logic Units (ALUs) of significant and varying pipeline depth, thread data storage requirements and interconnect between components. To justify why we chose a multithreaded hardware architecture that supports both ILP and TLP we consider the following types of architectures:

**Single-Threaded Single-Issue:** This architecture requires minimal thread storage (for 1 thread only) and can only issue to one functional unit at a time. The hardware area would be low, but the utilization would also be low due to pipeline stalls.

**Single-Threaded Multiple-Issue:** This architecture adds support for more functional units which exploits the ILP within a thread, but does not completely remove the pipeline stalls issue.

**Multithreaded Single-Issue:** This architecture would have minimal functional units support but it would have high utilization due to the instructions from different threads keeping the pipeline busy.

After considering the alternatives presented above we decided that the best architecture is a combination of the above approaches, one that supports both TLP and ILP. The exact configuration of the architecture would depend on the application. For that reason we use the compiler to help with the design space exploration, and selection of a dense and efficient architecture. As summarized in Figure 3.1, there are four main components of the TILT architecture: memory, the functional unit (FU) array, crossbar networks (for reading FU inputs and writing FU output results), and control (including instruction memory). Memory is used to store thread data, therefore its configuration depends on the amount of storage required per thread, the number of threads and banks of a TILT configuration, and the amount of ILP of a memory bank. The number of instances of FUs can be tuned to improve the throughput of an application and reduce the hardware area of a configuration, while remaining fully-connected and hence capable of executing other applications as well. The interconnect and control, are mostly dictated by the selection of memory architecture and FU organization.

### 3.1 Memory Architecture

TILT is designed to support two forms of parallelism: (i) *Thread-Level Parallelism* (TLP) and (ii) *Instruction-Level Parallelism* (ILP). TLP assumes that multiple parallel threads are provided by the programming model. For this work we assume that threads are completely independent and have no cross-thread synchronization. TLP is attractive because operations from different threads can always execute in parallel. In TILT, threads are mapped to independent memory banks without the need for keeping the banks coher-
Figure 3.2: Instances of TILT with: (a) two FUs, two banks (2B), one write port (1W) and two read ports (2R) per bank, hence supporting only 1-wide ILP per bank; (b) extended with an additional bank, deeper banks, and an additional FU, supporting greater TLP; (c) instead extended with two additional read ports per bank, hence supporting 2-wide issue-ILP per bank; (d) extended to two write ports and four read ports per bank, hence supporting both 2-wide issue-ILP and 2-wide completion-ILP per bank.

In order to be efficient, it is also possible for data from multiple threads to reside in a single memory bank, but data within a thread cannot reside across banks. Since FUs are typically pipelined—and in this work they are deeply pipelined up to 28 stages (square root operation)—TLP is an easy way to exploit the pipeline parallelism available in the datapath without requiring hazard detection logic nor extra result forwarding lines. Figure 3.2 shows four simple example instances of TILT, focusing on components and connections for data flow and storage.

2B/1W/2R Figure 3.2(a) shows a simple two-bank (2B) TILT that supports two-wide TLP (we have two banks and assume a thread will reside in each) but only one-wide ILP per thread, meaning that only a single operation from a single bank can issue or complete in a given cycle. Assuming FUs that each have one output and two inputs, then a bank...
must support one write and two reads every cycle (1W/2R) to avoid stalling. Assuming that memories will be constructed using Block Random Access Memories (BRAMs) that typically have only two ports each, this requires that each bank be composed of two BRAMs connected as shown such that the write value is copied to one port of each BRAM, and the remaining port of each BRAM provides one of the two read values—i.e., the BRAM contents are replicated.

3B/1W/2R, Deeper Banks, Additional FU  Figure 3.2(b) illustrates several extensions over (a). First, an additional bank is added, providing increased TLP by supporting an additional thread, but requiring wider read-side and more numerous write-side multiplexing. Second, the depth of each bank is doubled (i.e., by implementing each bank using twice the BRAMs), allowing a greater number of threads to reside within each bank by providing extra storage capacity. Third, an additional FU is added, supporting increased TLP, but requiring more multiplexers on both the read and write sides (note that the number of banks need not match the number of FUs). These three axes alone comprise a large and interesting design space.

2B/1W/4R  Figure 3.2(c) instead extends each bank by two additional BRAMs (shaded) such that this architecture is now capable of two-wide issue-ILP: each bank now supports four read ports (4R), hence two operations can be issued each cycle from each bank from a single thread. However, this architecture still supports only one write port (1W) per bank, and hence can only complete one operation per bank each cycle. Of what use is a datapath that each cycle can issue two operations but only complete one operation? Since FUs may have varying pipeline depth (between 5 and 28-stages), operations that are issued in the same cycle may complete in different cycles. While two-wide issue-ILP from a single thread cannot be sustained when only one-wide completion-ILP is supported, occasionally exploiting issue-ILP can potentially boost overall performance and be worth the additional hardware area.

2B/2W/4R  Figure 3.2(d) further extends each bank with support for an additional write port via four additional BRAMs (shaded), hence providing both two-wide issue-ILP and two-wide completion-ILP; in other words, each bank can potentially issue and complete two operations every cycle. While this architecture is the most flexible, it is also expensive—requiring twice the BRAMs of (c) in addition to wider multiplexing. However, this design can reduce the amount of write-side multiplexing: in the example shown, write-side multiplexing is eliminated because the supported completion-ILP matches the total number of FUs, hence each FU writes to its own sub-bank of BRAMs. However, this introduces the complexity that the most recently-written value for a given location might
reside in one of the two sub-banks. Rather than implement a “self-directed” multi-ported memory such as the Live Value Table approach [42], we instead rely on the compiler to direct the FU-input multiplexers to the appropriate sub-bank.

### 3.2 Functional Units

The TILT architecture can target FUs of any type and data width, although for this work we assume FUs that require at most two 32-bit inputs and one 32-bit output. FUs can be pipelined to any depth, and different FUs can have different pipeline depths. There can also be multiple instances of any FU type. Therefore, there is potential for tuning the FU mix to match the needs of the application. We will evaluate the effectiveness of different FU mixes, allowing us to additionally account for the area cost and throughput impact of different FU selections. We also focus on applications that require IEEE-754 single-precision (32-bit) floating-point functional units.

### 3.3 Interconnect

As illustrated in Figures 3.1 and 3.2, TILT requires two crossbar networks to route (i) FU inputs and (ii) FU outputs, from and to the memory system BRAMs allowing for a fully connected architecture; that is, every FU input and FU output has access to every memory bank. This added flexibility allows TILT to be fully software-programmable.

The routing of the interconnect is accomplished via multiplexers as shown in Figure 3.2, and the width and number of multiplexers grows quickly with memory system banks and FU count. Interconnect can comprise a significant fraction of the total area of TILT.

### 3.4 Control

TILT is a Horizontally Microcoded architecture, meaning that each ”instruction” encodes several operations that may issue in parallel in the same cycle as illustrated in Figure 3.3(b). The instruction encoding provides a sub-instruction for each architected FU as illustrated in Figure 3.3(a), and its width depends on several parameters described below as well as the number of FUs. The encoding for each FU specifies three operand addresses (two read and one write) for accessing memory or I/O, read and write port identifiers to direct any multiplexers in the FU-input or FU-output interconnects (including selecting the correct sub-bank of BRAMs in the case of replicated BRAMs), a thread identifier for selecting which predicate bit to access during multiplex operations (described in Section 3.6), an opcode for controlling the FUs, and a valid flag for in-
Figure 3.3: Illustration of the instruction encoding and an example of a full instruction memory illustrating the presence of explicit instructions for each functional unit and execution cycle.

dicating a valid sub-instruction. The address, port and thread ID widths depend on the configuration selected. For example, for 8B/2W/4R, $8 \times 2 = 16$ write ports and $8 \times 2 \times 4 = 64$ read ports will exist providing access to the 8 memory banks. Based on the above parameters the compiler is used to generate the Instruction Memory illustrated in Figure 3.3(b) with a long instruction that is the combination of the sub-instructions for each FU and specifies the execution for all the cycles.

### 3.5 Data Input/Output

TILT provisions an extra I/O port that can be used to connect another component to enable data transfer between external memory and TILT memory banks. This port can be used to connect a Direct Memory Access (DMA) engine or a mechanism such as CoRAM [25] or LEAP scratchpads [13] for transferring data in the background using the interconnect. Data transfers can be controlled by separate instructions specifying local and external memory addresses along with type of transfer (in or out of TILT). Currently we do not schedule I/O operations as part of the execution schedule for our benchmarks, and assume this would be done in the background. For our densest base design (1 FU instance for each type, 4 banks of 512 word depth) we have on average 45% empty schedule slots (i.e., available memory ports) that could be used for I/O instructions. Additional memory depth could be allocated to support double-buffering, allowing data transfers without impacting application performance. TILT currently assumes that all required data are resident in memory at the start of execution, and TILT could be signalled to
begin execution when this is true. More aggressively, schedules could be adjusted to attempt to overlap and tolerate the latency of external memory requests, but in the simple case TILT would have to stall whenever a required data item has not yet arrived. These more aggressive implementations are left for future work.

3.6 Predication

To avoid complex scheduling/branching hardware, TILT supports predication by implementing compare (CMP) and multiplex (MUX) instructions in a separate FU (CMP,MUX). The MUX is similar to a phi function in static single assignment (SSA) scheduling [14]. In data flow analysis, phi is used to indicate that the value of a variable at a point in the program depends on the path taken to reach that point. The Compare operation takes as input two floating-point operands, and a flag indicating the comparison type. An array of predicate flag bits, containing a 1-bit entry for each thread’s flag, is used to store the comparison results. To support if-else branches, we schedule instructions from both sides of the branch, and after they finish executing we save the results of the correct side of the branch. For each variable modified inside the branch, the MUX instruction is used to take the data saved in temporary locations from both sides of the branch, and copy the valid instance to the final destination. The MUX reads the input operands directly from thread memory, accesses the corresponding flag in the predicate flag array, and based on the flag writes the correct result to the output memory. Due to the low latency and area cost, the CMP,MUX FU accesses operands from the memory banks by sharing the read and write ports with one of the least-utilized FUs depending on the application. Since each thread has only 1 entry for predicate flags, a set of CMP and MUX instructions must execute as a pair before the flag bit can be overwritten by the next CMP. To alleviate this constraint, the compiler needs to find other parallel operations to execute.

3.7 Summary

In this chapter we presented the TILT architecture and its main components: memory, functional units, interconnect and control. FUs are connected to the memory banks through the interconnect, execution is controlled by using Horizontally Microcoded instruction memory, and branches are supported through predication. The compiler’s job is to organize these components to provide efficiency for applications running on TILT. The next chapter will give more details on how we leverage LLVM to compile programs for this architecture.
Chapter 4

LLVM Framework

In this chapter we describe the aspects of LLVM that we use when compiling for TILT and additional support that we added to make it useful for TILT compilation. LLVM [9] is a compiler infrastructure that includes front-ends for languages such as C, C++, Objective C, an optimizer with multiple passes such as Dead Code Elimination, Global Value Numbering, Loop Invariant Code Motion, and a target independent code generator. For the work in this thesis, we use the intermediate representation (IR) generated by the frontend and implement a Data Flow Graph (DFG) Generator compiler pass, that runs on the IR and generates a custom representation of the DFG which is used in the rest of our tools.

The LLVM compiler framework (version 3.1 [9]) was used to generate data flow graphs from the C programming language. The Data Flow Graph Generator is implemented as a compiler pass [12], which runs on the intermediate representation that is generated by the compiler frontend. The DFG Generator expects the source code to be included in one C function, similar to a kernel in OpenCL. All variables have to be declared inside the function or they can be passed as function parameters. We support stack variables of floating-point type or pointers to these types of variables. Dynamically allocated variables are not supported, i.e. we need to know in advance the size of a variable. Support for other types such as integer or char can be added easily but it was not required for our purposes. We do not support loops, and it is assumed that the body of the loop would be transformed into a function that would be executed by each thread. The DFG Generator supports if-else branches by using predication. Since we are targeting compute-intensive applications, branch intensive applications will suffer a performance hit.
4.1 Instruction Parsing

The LLVM IR [10] is composed of a large number of instructions. For the purposes of our DFG Generator pass we support the following subset of instructions:

- getelementptr (GEP)
- fpext
- Store
- Load
- Compare
- Branch (Br)
- Binary Operations
- Function Calls

In the following section, we will provide a brief overview of how these instructions are used by LLVM and explain how the DFG Generator handles them including some challenges when parsing certain instruction.

4.1.1 getelementptr (GEP) instruction

```
%26 = load float* %t.m_NA, align 4
%div56 = fdiv float %sub55, %26
%arrayidx57 = getelementptr inbounds [4 x float]* %f, i32 0, i32 1
store float %div56, float* %arrayidx57, align 4
...
%arrayidx53 = getelementptr inbounds [4 x float]* %f, i32 0, i32 1
%25 = load float* %arrayidx53, align 4
%sub54 = fsub float %24, %25
```

Figure 4.1: Illustration of getelementptr (GEP) instruction.

This instruction is used by LLVM to calculate the address of an array element or struct component. It takes as parameters a pointer to a variable and a number of indexes that are interpreted as the offsets into the various dimensions of the data structure. From
these parameters it creates a pointer to a memory location that can be used by loads or stores. In Figure 4.1, f is a 2 dimensional array and `getelementptr` is used to calculate a pointer to the array location f[0][1].

The pointers generated by this instruction can cause memory aliasing, which creates a problem for the DFG parser. In the example of Figure 4.1, `arrayidx53` and `arrayidx57` point to the same location f[0][1]. In the first 4 lines of IR code, a value is stored into the memory location pointed by `arrayidx57`, which aliases to f[0][1]. Later a value is loaded from that same location but under the alias `arrayidx53`, which implies data flow between the `fdiv` and `fsub` operations that must be recorded. To solve the problem we maintain a map data structure that has an entry for each register generated by `getelementptr` and a corresponding memory location. For the sample code 4.1, we would have 2 entries: `arrayidx57→f[0][1]` and `arrayidx53→f[0][1]`. When generating data flow dependences we look-up other locations that map to the same alias and check if they are executing an operation that generates data flow.

### 4.1.2 `fpext`

```
%58 = load float* %total, align 4
%conv151 = fpext float %58 to double
%add152 = fadd double %conv151, 6.000000e+01
```

Figure 4.2: Illustration of `fpext` instruction.

LLVM’s `fpext` instruction takes as parameters a floating-point value and another floating-point type to cast the previous value to. The second type should be larger than the input value so that the value can be extended. In Figure 4.2, the float value from register 58 is extended to a double value and stored in register `conv151`.

The `fpext` instruction is used to convert between data types, so it does not have a real impact on data flow. During the DFG generation, we have to bypass these instructions until the data flow reaches an instruction that performs computations. When parsing the code in Figure 4.2, we keep track of the operation `fadd` and its 2 inputs are register 58 and the constant 6.0e+01, therefore bypassing the conversion.

### 4.1.3 Stores

An LLVM store instruction can store an LLVM register value specified in the first parameter to a memory location specified by the second parameter. It is also used in the IR to store constant values to memory.
Stores are the only way to write data to a memory variable (non-register), therefore they are important in data flow tracking. In Figure 4.3, a value computed by \texttt{fmul} is stored into \texttt{sum} and later loaded into register \texttt{59} which is used in a compare instruction. In this case the store is used to create data flow from \texttt{fmul} to \texttt{fcmp}. In addition to data flow, we need to perform renaming of the stores that are part of a branch, for purposes of branch predication. Each store that is performed inside a branch is renamed to a different destination with a unique name. Since we always execute both sides of a branch, we cannot write to the original memory address until we know the outcome of the branch. A table is used to keep track of all the memory locations that were renamed so a \textit{Compare Mux} instruction can be inserted to select the correct value at the join point of the branch.

4.1.4 Load

LLVM uses loads to move values from memory into registers so that operations can be executed on those values. Loads are one way of creating data flow in the IR. A load takes as a parameter a pointer to the memory address where the data is located and an optional align parameter to indicate the alignment of the memory address.

In Figure 4.3, the input to the \texttt{fcmp} instruction is loaded from the \texttt{sum} memory address. The \texttt{load} is used to generate data flow from the \texttt{fmul} instruction to the \texttt{fcmp} by going through the \texttt{sum} memory location. This behaviour is recorded in the DFG.

4.1.5 Compares
An LLVM compare instruction returns a boolean value based on the results of the comparison between two operands that are compared based on a condition code such as greater than (OGT), less than (OLT), and saves the result of that comparison in a register. A full list of condition codes can be found on the LLVM documentation website [2][3]. In Figure 4.4, the condition code is OGT and it is checking if the value in register 9 is greater than the floating-point constant 0 (zero). Compare instructions are used in LLVM IR to compute branch outcomes. In the DFG Generator these are treated as any other computation operation, where we keep track of the data flow into the two operands of the cmp. The result of the compare is used by the Compare Mux operations inserted to support predication (see Section 4.2.4). A dependence is inserted between the cmp and corresponding Compare Mux operations.

4.1.6 Branches

A branch is used to change the control flow to different basic blocks of LLVM IR. There are 2 types: unconditional and conditional branches. Unconditional branches take as parameter the label of the target basic block. Conditional branches take as parameters a condition value and 2 possible target labels. In Figure 4.4 the condition calculated by cmp is used to determine whether the control flow will go to the if.then or the if.else block. For every if-else branch, we keep track of a mapping between the compare operation that evaluates the result of the branch and the Basic Blocks that are the possible destinations of that branch. This is used later for predication.

4.1.7 Binary Instructions

This refers to all instructions that have 2 input operands. Examples of these are: Add, Subtract, Multiply, Divide. In LLVM IR these operations take as input a register value or a constant and the result is stored in another temporary register. LLVM uses the SSA [14] format for instructions, and each operation that produces a result stores it in a register named according to SSA, which generates a new register name for each assignment.

In the DFG Generator the operations of these instructions are converted to DFG nodes. Each node has two input edges that represent the operands of this instruction and connect to the nodes generating those operands. If an input is a constant or it is not produced by a computation (such as a first time load from memory), there is no edge but the data flow is implied. Outgoing edges points to other nodes that utilize the generated result. Results that are stored to memory and never loaded by a subsequent operation
inside the code do not have an outgoing edge.

### 4.1.8 Function Calls

Our processor does not support function calls by using calling conventions or parameter passing. Instead, it has a predefined list of functions that are implemented in hardware as functional units. These functions are: Exponent, Square Root, Logarithm, and Absolute value. In LLVM IR they appear as function calls, take a single input parameter and save their results in a SSA register. They are represented as nodes in the DFG with one input edge and one output edge.

### 4.1.9 Compare Mux

We use predication to support if-else branches in TILT, which is implemented by scheduling both sides of a branch separately, and then selecting the correct results based on the outcome of the branch comparison. This approach is similar to the phi function in static single assignment (SSA) scheduling [14]. Most of the computations’ results are stored in SSA registers, but for those instructions that save results to memory we need to perform renaming if a memory location is being stored to from instructions inside a branch. At the end of the branch, we insert a Compare Mux instruction (similar to a select instruction [46]), which takes as parameters: i) an array of predicate flag bits with the result of the branch comparison, ii) the temporary memory locations used in each side of the branch, iii) and the final memory location where the result will be copied. Based on the outcome of the branch, it copies the correct result while disregarding the computation of the other side of the branch. Figure 4.5 illustrates how we handle if-else branches by showing the source code, corresponding IR, and the DFG that we generate after inserting a Compare Mux instruction, and after making the changes required to support predication.
void cmpmux_example(float* f_vector)
{
float beta_mK;
float t_m_K;
float sum6;

sum6 = t_m_K+60.0;
if (sum6 > 0)
{
    beta_mK = 0.125*sum6;
}
else
{
    beta_mK = -0.125*sum6;
}

f_vector[0] = beta_mK;
return;
}

(a) Sample source code.

(b) CFG of source code. We see that on both sides of the branch there is a store to the same variable beta_mK.

(c) DFG of source code. The first row of each box contains the operation ID, and in brackets the high level representation of the operation. The second row indicates the latency of each outgoing edge, which corresponds to the latency of the current operation (add takes 7 cycles).

Figure 4.5: This is an example of an if-else branch where both sides write a result to the same variable beta_mK. The writes on both sides of the branch will be redirected to temporary memory locations. This is reflected in the DFG where beta_mK has been replaced by beta_mK_tmp1 and beta_mK_tmp2. The CMP_MUX instruction has been inserted in the DFG, and it depends on 3 operations (3 incoming edges): the compare that generates the result and 2 predicated operations that compute the variables.
4.2 Dependence Tracking

To create a correct DFG representation of the source code, we need to keep track of all the dependences and flow of data between operations. This section will explain how we deal with different cases of these dependences. Before starting to search for dependences between operations, we create a mapping for all the aliases created by LLVM’s GEP instruction. In LLVM’s IR, every time an address is calculated using GEP, it creates a new alias for the pointer, so we need to record these aliases and use them to track data flow. In addition, we assign a unique number to each variable and LLVM register to generate a flat address space for the memory accessed by the DFG. This means that the DFG does not distinguish between variables and array element accesses.

4.2.1 Read After Write

Results produced by a calculation in LLVM are either stored to a temporary register or written to memory by using a store instruction. Registers are assigned in the SSA format, which creates a new register name for each assignment, therefore making it easier to track the data flow; alternatively memory locations can be read and written by different instructions of the control flow graph (CFG), so we need to keep track of data flow across different basic blocks. A data structure, we call the Store Tracking data structure, is updated with information every time an operation writes its output to a memory location. When we iterate through each instruction, a check is performed to find out if each input is a memory location or a register. For registers we add a dependence to the operation that writes its output to that register. The SSA format guarantees that there is only 1 such operation. If the input is a memory location, we have to refer to the Store Tracking data structure and find the ID/name of the operation that will be the last one to store a value to that memory location. In this case we indicate a dependence on that operation.

4.2.2 Write After Read

To allow us to keep track of these dependences another table is kept with all the loads that have been executed in previous instructions of the CFG. When an operation is storing a value to memory, it checks the load table and if there is an entry which loads from the same memory location a dependence is inserted indicating that it has to wait for the previous value to be loaded before overwriting it. The exact delay is determined later based on the architecture’s read and write latencies.
4.2.3 Write After Write

To exploit parallelism, the execution schedule of DFG operations does not have to match the CFG’s order of operations. Therefore, to preserve program correctness it is necessary to execute writes to the same memory location in the same order that they appear in the code. To track these dependences, we use the Store Tracking data structure again, to search for other stores that occur earlier than the current store (in CFG program order), and add a dependence between the current operation and them. For this type of dependence we need to wait for the full execution time of the previous operation before overwriting the data. Initially it seems unnecessary to keep track of WAW dependences if there is no use (i.e. LOAD) of the value in between, which is a case that would have been handled by what was described in the previous section (WAR). This is not the case in the code of Figure 4.6.

```c
1: int test(float* in1){
2:    float a0=1.5, a1=2.5, a2=3.5;
3:    float s0=5.4;
4:    *
5:    *in1 = a0 + a1;
6:    *in1 = a2 + s0;
7:    return 0;
8: }
```

Figure 4.6: Illustration of WAW dependence.

In this case there are 2 writes to the same pointer (in1), passed in as a function parameter. Logically only the result of the second write should be stored into that location. Therefore we need to keep track of the program order of the statements in lines 5 and 6 such that the correct result is stored.

4.2.4 CMP Dependences

Hardware support is required to implement branching through predication. In our architecture we decided to provide registers that store the result of a compare instruction: true or false. The number of registers determines how many branches can be executed at the same time. Initially, we simulated the option where only 1 branch can be executed across all threads for a branch intensive benchmark, but the throughput was very low. In our hardware architecture, we support a single storage location per thread to be used for branch result codes. A slight delay is introduced for applications with a high number
of branches such as Hodgkin-Huxley, but it is hidden by using additional threads. This limitation requires that a pair of \texttt{CMP} and \texttt{CMP\_MUX} are executed in order without any intervening \texttt{CMP} that would overwrite the previous results. To prevent this problem we do two things: a dependence edge is inserted between a \texttt{CMP} and its corresponding \texttt{CMP\_MUX}; and another edge is inserted between a \texttt{CMP\_MUX} and the next \texttt{CMP} in program order. The last edge is inserted to prevent the next \texttt{CMP} from being scheduled before the \texttt{CMP\_MUX} has a chance to read the result code.

\section*{4.3 Summary}

In this chapter, we described the LLVM framework and how we parse a subset of its instructions to generate a DFG for our compiler. We described how dependences are tracked in the IR and converted to edges in the DFG, including the ones related to branch predication. DFG generation is an important step necessary for the rest of our experiments and in the next chapter we will show results gathered after scheduling the applications’ DFGs.
Chapter 5

Compiling for TILT

In the following chapter we will develop a scheduling approach for TILT with the objective of highly utilizing variable-latency, pipelined functional units by executing multiple independent threads. We start with a simple scheduling algorithm, which is based on list scheduling, and improve it to create our final approach. For our benchmarks we evaluate the performance of these approaches, and assess which variants perform better. Lastly, we introduce a memory allocator that is used to reduce benchmark memory usage.

5.1 Scheduling Approaches

Software Pipelining [43] is a well known approach that creates code for the loop’s prologue, steady state, epilogue and overlaps loop iterations to improve resource utilization. In our scheduler we assign threads to independent loop iterations but do not have prologue and epilogue code. We create a full schedule for all the threads that could be repeated just like the steady state code.

All of our scheduling algorithms are based on list scheduling [43], [31], which iteratively places—into the earliest free spot in the schedule—operations with no predecessors or whose predecessors have already been scheduled. The algorithm variants presented in the following sections differ by changing: the way that they schedule operations across multiple threads and the way DFG operations are prioritized. These different approaches can be combined together to generate more efficient schedules. As these algorithms are heuristics, there is never a guarantee that a certain approach will always produce the best scheduling results. Nevertheless, some approaches are in general more effective than others. The following describes each algorithm in detail, illustrated with schedules for the example data flow graph shown in Figure 5.1(a). For all of the scheduling algorithms, the scheduling is performed by keeping a reservation table [27] with entries for each func-
Figure 5.1: An example data flow graph (a), and schedules for four threads (1-4), assuming delays of 4 cycles for add/sub and 3 cycles for mult/div. In (b), (c) and (d) colors distinguish operations from different threads. In (e) colors distinguish between scheduling groups, while in (f) they distinguish longest path operations from the rest. The schedule entries indicate thread number and operation id (e.g. 2:E means thread 2, operation E), executing at specific cycle in an FU. By performing round robin scheduling among threads, the schedule in (c) becomes 1 cycle shorter than the schedule in (b). Schedule (c) uses by default a group of size 4, while schedule (e) uses a group size of 2 (see 5.1.7). In this case a group size of 4 generates a shorter schedule than a group size of 2. Figure (f) shows the benefit of longest path scheduling, by saving 3 cycles over PGMs schedule (c). A symmetric schedule is shown in (d).
scheduling operations of threads that share a bank, we have to also keep track of, and avoid read/write port conflicts. Since operations have different latencies, two operations starting at different cycles may have a conflict when writing results at the same cycle. To find a free spot in the schedule, the algorithms search the table of the corresponding functional unit, and find a suitable cycle to issue the instruction where none of the resource constraints are violated.

5.1.1 Greedy

![Algorithm 1: Greedy Scheduling.](image)

This scheduling approach, listed in Algorithm 1, is based on list scheduling. It will take as input the DFG of a benchmark, and it will start by scheduling the operations that have no predecessors. At each iteration it will check for new ready operations whose predecessors were scheduled, and it will find the earliest free spot where that operation can be scheduled. The process stops when all the operations of a thread have been scheduled. To schedule multiple threads, the same procedure is repeated but every thread will have to be scheduled in the free spots left by the previously scheduled threads. This heuristic stops when all the threads are scheduled. Our intuition indicated this may not be the best approach. Since our architecture has a limited number of FUs, we thought it could be important to prioritize certain operations when scheduling with limited resources. By scheduling threads in order, we believed that subsequent threads were not being treated fairly by being scheduled in the leftover schedule spots of the previous threads. This thinking motivated the approaches that follow.
5.1.2 Prioritized Greedy (PG)

This greedy approach, listed in Algorithm 2, gives priority to operations with lower slack time [50]: the cycle difference in placement of an operation in As Late As Possible (ALAP) and As Soon As Possible (ASAP) schedules, ignoring resource constraints. ASAP schedules operations as soon as predecessors have completed, while ALAP schedules operations as late as possible in the schedule without increasing the total schedule latency beyond that of ASAP. The overall effect of slack-based priority is to decide wisely between two “ready” operations, favouring the operation that is more likely to increase the total schedule latency if unchosen. Depending on the parallelism of the DFG, it is likely there will be more than one ready operation for scheduling at a certain iteration of the algorithm. The Greedy approach did not use priorities when selecting operations for scheduling. By using priorities we can chose in a non-arbitrary way which operation gets scheduled first, which is useful especially when the number of functional units is limited. To ensure fairness when using operation priorities we use a cycle counter, and hold back operations from being scheduled to make sure all the operations that could have started at a certain cycle are in the ready list before proceeding. This prevents an operation with a lower priority from being scheduled before another operation of higher
priority simply because the latter was not inserted in the ready list at the time when the
former was scheduled. Looking at Figure 5.1 (a), operations G and C are both ready
immediately, but G has the least slack and higher priority and is hence scheduled first in
Figure 5.1 (b). This method schedules each thread completely before moving on to the
instructions of the next thread. This approach does not yet address the problem with
threads being scheduled in order, and this will be addressed in the next section.

5.1.3 Greedy Mix (GM)

| Data: DFG |
| Result: Instruction Schedule |
| Create ASAP schedule for DFG operations |
| Create ALAP schedule for DFG operations |
| Calculate Slack (ALAP-ASAP) for operations in DFG |
| Create readyList[] array with a readyList entry for each thread |
| for t ← 0 to Threads − 1 do |
| foreach Operation in DFG do |
| if operation has no predecessors then |
| Insert in readyList[t] |
| end |
| end |
| cycleCounter ← 0 |
| while readyList array not empty do |
| Sort readyList[] entries by priority for each thread |
| for i ← 0 to (maximum list size across all readyList[] entries) − 1 do |
| for t ← 0 to Threads − 1 do |
| if readyList[t] contains an operation at index i then |
| if earliest cycle operation can start <= cycleCounter then |
| Schedule operation in earliest free spot |
| Mark for removal from readyList[t] |
| end |
| end |
| end |
| Remove marked entries from readyList[] for all threads |
| cycleCounter ← cycleCounter + 1 |
| for t ← 0 to Threads − 1 do |
| foreach Operation in DFG not already scheduled for t do |
| if all predecessors are scheduled then |
| Insert in readyList[t] |
| end |
| end |
| end |

**Algorithm 3:** Prioritized Greedy Mix.

In the previous approaches, a thread was being scheduled completely before moving
to the next thread. This variant of scheduling selects instructions to schedule in a round-
robin fashion across threads, as shown in the sample schedule of Figure 5.1 (c). Ready operations from different threads are considered for scheduling by giving each thread a chance to schedule operations instead of making them wait for the previous threads to be fully scheduled. We believed this approach would be more effective because it reduces the waiting time before a thread can start executing. By fully scheduling each thread, there may be lower priority operations in a previous thread that postpone a higher priority operation of the current thread. In such cases, postponing the finish time of the previous thread has a small effect when compared with the improvement in total finish time, which is controlled by the last scheduled thread in the Greedy approach. With Greedy Mix we schedule the operations more fairly across threads.

5.1.4 Prioritized Greedy Mix (PGM)

This is a combination of the previous two approaches that selects instructions to schedule in a round-robin fashion across threads, and assigns priorities to operations based on slack time. It is outlined in Algorithm 3. We expected this approach to improve on the previous, by combining operation priorities within a thread and better scheduling across threads. The reasoning behind these approaches was explained in their respective sections.

5.1.5 Symmetric

The symmetric scheduling algorithm (sample shown in Figure 5.1 (d)) performs list scheduling on the operations of a single thread, to minimize its schedule duration (eg., a duration of 9 cycles in the example). Subsequent threads share the same schedule/code, and each is scheduled to begin at the earliest possible cycle where there are no resource conflicts with the previous thread. This method has the benefit of minimal program storage, but rarely results in a dense execution schedule.

5.1.6 Longest Path

Inspired by Xu et al. [66], this approach first schedules instructions that are part of the longest dependence chain within each thread—accounting for FU latency and scheduling across threads—then schedules the remaining instructions. A path is defined as one that starts at an instruction with no prior dependences (i.e. no predecessors) and ends at a node with no successors. The longest path is a path whose edge weights (representing latencies) have the maximum sum over all possible paths in a DFG. The instructions are then separated into two sets: $Set1$, those in the longest path plus all of their predecessors (there may be predecessors not part of longest path); and $Set2$, the remaining instruc-
tions. The algorithm schedules \( \textit{Set1} \) first across all threads, and then schedules \( \textit{Set2} \). The intuition is that giving higher priority to operations in the longest path will reduce overall schedule length. In Figure 5.1 (f) operations A through E are part of the longest path and are scheduled before the other operations F through H. This approach can be combined with any of the previous approaches. For example we can use Prioritized Greedy Mix to schedule the operations in \( \textit{Set1} \), and then in the same way schedule the operations in \( \textit{Set2} \).

### 5.1.7 Grouped PGM

```plaintext
Data: DFG
Result: Instruction Schedule
Create \textbf{ASAP schedule} for DFG operations
Create \textbf{ALAP schedule} for DFG operations
Calculate \textbf{Slack} (ALAP-ASAP) for operations in DFG
Create \textbf{readyList[]} array with a readyList entry for each thread

\textbf{for} groups \( \leftarrow 0 \) to \( \text{Threads/GroupSize} \) \textbf{do}
  \textbf{for} \( t \leftarrow \text{first thread of group to last thread of group} \) \textbf{do}
    \textbf{foreach} Operation in DFG \textbf{do}
      \textbf{if} operation has no predecessors \textbf{then}
        \textbf{Insert in readyList}[t]
    \textbf{end}
  \textbf{end}

\textbf{cycleCounter} \( \leftarrow 0 \)
\textbf{while} readyList array not empty \textbf{do}
  Sort readyList[] entries by priority for each thread of group
  \textbf{for} \( i \leftarrow 0 \) to \( \text{(maximum list size across all readyList[]} \text{ entries of group)−1} \) \textbf{do}
    \textbf{for} \( t \leftarrow \text{first thread of group to last thread of group} \) \textbf{do}
      \textbf{if} readyList[t] contains an operation at index \( i \) \textbf{then}
        \textbf{if} earliest cycle operation can start \( \leq \) cycleCounter \textbf{then}
          Schedule operation in earliest free spot
          Mark for removal from readyList[t]
        \textbf{end}
      \textbf{end}
  \textbf{end}

Remove marked entries from readyList[] for all threads of group
\textbf{cycleCounter} \( \leftarrow \) cycleCounter + 1
\textbf{for} \( t \leftarrow \text{first thread of group to last thread of group} \) \textbf{do}
  \textbf{foreach} Operation in DFG not already scheduled for \( t \) \textbf{do}
    \textbf{if} all predecessors are scheduled \textbf{then}
      \textbf{Insert in readyList}[t]
  \textbf{end}
\textbf{end}
\textbf{end}

\textbf{Algorithm 4}: Grouped Prioritized Greedy Mix.
When we changed from Greedy to Greedy Mix, we went from scheduling one thread at a time to scheduling all threads round-robin (i.e., all of them at a time). What if there is a better way to schedule between these two extremes. To evaluate this possibility we add the concept of thread groups: for group size $G$, this algorithm schedules instructions using PGM fully for the first $G$ threads until done, then moves on to the next $G$ threads. In Figure 5.1 (e), the group containing threads 1 and 2 is scheduled completely (i.e., all instructions A to H), and then the next group containing threads 3 and 4 is scheduled. By scheduling all instructions from a group of threads before moving on to the next group, this version of PGM can reduce intra-group FU conflicts and hence produce more efficient schedules. Our algorithm tries all possible group sizes, from 1 to total number of threads, and selects the best one. The pseudo-code for this approach is listed in Algorithm 4.

5.1.8 Grouped PGM Longest Path

This approach combines Grouped PGM with the Longest Path approach. The operations in a DFG are first separated into 2 sets, as explained in the Longest Path subsection 5.1.6. Each of these sets is then scheduled for the required number of threads using the Grouped Prioritized Greedy Mix approach, therefore applying all the methods that we have described in the previous paragraphs.
Table 5.1: FU Usage Count Statistics.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>AddSub</th>
<th>Mul</th>
<th>Div</th>
<th>Sqrt</th>
<th>Exp</th>
<th>CMP MUX</th>
<th>Log</th>
<th>Abs</th>
<th>Total</th>
<th>Data Requirement (words)</th>
<th>Predicated Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Black Scholes</td>
<td>23</td>
<td>32</td>
<td>9</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>79</td>
<td>64</td>
<td>4</td>
</tr>
<tr>
<td>Gaussian Blur</td>
<td>40</td>
<td>51</td>
<td>7</td>
<td>1</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>104</td>
<td>128</td>
<td>0</td>
</tr>
<tr>
<td>N Body</td>
<td>9</td>
<td>9</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>20</td>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>SRAD</td>
<td>20</td>
<td>19</td>
<td>12</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>55</td>
<td>64</td>
<td>4</td>
</tr>
<tr>
<td>Hodgkin-Huxley</td>
<td>36</td>
<td>24</td>
<td>28</td>
<td>0</td>
<td>15</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>115</td>
<td>128</td>
<td>71</td>
</tr>
<tr>
<td>HDR</td>
<td>18</td>
<td>12</td>
<td>9</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>42</td>
<td>64</td>
<td>0</td>
</tr>
<tr>
<td>Matrix Multiply</td>
<td>12</td>
<td>16</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>28</td>
<td>64</td>
<td>0</td>
</tr>
<tr>
<td>Matrix Inverse</td>
<td>55</td>
<td>92</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>148</td>
<td>128</td>
<td>0</td>
</tr>
<tr>
<td>Planck Distribution</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>16</td>
<td>0</td>
</tr>
</tbody>
</table>

5.2 Benchmarks

We measure TILT using a set of benchmark applications from different areas such as: neuroscience simulations, financial applications, physics simulations, mathematical applications or image processing. Some of these benchmarks had to be rewritten to remove features that are not currently supported by our DFG parser. For Gaussian Blur we performed loop unrolling. No changes were made to Black Scholes and Matrix Inverse. For the rest of the benchmarks the body of a loop was assumed to be the code for a single thread. TILT is not designed for branch intensive applications, and we do not have support for memory intensive applications, so we do not evaluate such applications. We use benchmarks from various sources and not a specific suite for the following reasons: i) we preferred compute intensive applications as opposed to data intensive (i.e. a large loop body but a smaller amount of data), ii) we preferred using floating-point benchmarks to match our architecture, and iii) we wanted to test a variety of functional units including transcendental functions (exp, log).

The benchmarks will be described in the following paragraphs, and their FU usage statistics are displayed in Table 5.1.

**Black Scholes (BSc) option pricing** The BSc [20] model is based on a partial differen-
tial equation used to approximate the value of European call and put options, accounting for variables such as stock price, risk-free interest rate, and volatility. Its implementation requires the following FUs: add/sub, multiply, divide, exponent, logarithm, square root, compare and absolute value FUs.

**Gaussian Blur** This filter [1] is used extensively in image processing and computer vision. It also known as an averaging operator and optimal for image smoothing [53]. The coefficients for blurring are generated using a Gaussian distribution function, and then each pixel value is multiplied by the corresponding coefficient value. This benchmark requires add/sub, multiply, divide, exponent, and square root FUs.

**N-body Simulation** This is a method [7] used to approximate the movement of a system of bodies that are interacting with each other. In an astrophysical simulation each body could be a galaxy or star that interacts through the gravitational force. For this benchmark we use code from the All Pairs N-Body Algorithm, which updates the acceleration for each pair of bodies due to their interaction with each other. The benchmark requires the following FUs: add/sub, multiply, divide, and square root.

**Speckle Reducing Anisotropic Diffusion (SRAD)** SRAD is a diffusion method for ultrasonic and radar imaging applications based on partial differential equations (PDEs). It is used to remove locally correlated noise, known as speckles, without destroying important image features [69] [8]. SRAD includes several steps that perform image processing. The benchmark that we use for evaluation contains computation steps that gather statistics on the image, and compute mathematical operations such as derivatives, gradient and laplacian. The benchmark requires the following FUs: add/sub, multiply, divide, and compare.

**Hodgkin-Huxley (HH)** The Hodgkin-Huxley neuron simulation [34] describes the electrical activity across a patch of a neuron membrane, where the voltage across the membrane varies as ions flow radially through each compartment. The computation involves solving four first-order differential equations using Euler’s method to iteratively compute simple finite difference, and the implementation requires add/sub, multiply, divide, exponent, and compare FUs.

**Matrix Multiply** This code is taken from the MESA library [11] [6]. It performs the multiplication of two 4x4 matrices of floating-point numbers. The benchmark executes the body of the loop that multiplies each row by the corresponding column and saves the sum into the destination variable. The benchmark requires the following FUs: add/sub and multiply.

**Matrix Inverse** This code is also taken from MESA which is a graphics library [11] [5].
The code performs the inversion of a 4x4 matrix of floating-point numbers. At first it computes the determinant of the matrix, and then calculates the inverse value for every entry in the matrix. The benchmark requires the following FUs: add/sub, multiply, and divide.

**High Dynamic Range (HDR)** This is used to generate images that contain a greater range of luminance than the image that can be captured by standard cameras [48]. This benchmark operates on the pixel values of 3 images taken at different exposures: bright, medium, and dark. It combines the 3 values and then normalizes them to produce a new image of a High Dynamic Range. The benchmark requires the following FUs: add/sub, multiply, divide, and square root.

**Planck Distribution** This benchmark is part of Livermore loops. It is used to calculate the intensity of radiation emitted by a body. It is a simple loop that requires the following FUs: add/sub, multiply, divide and exponent.

### 5.3 Scheduling Results

**Table 5.2: Statistics on DFG Nodes.**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>On Longest Path</th>
<th>Dependences of Longest Path</th>
<th>Others</th>
<th>Total operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Black Scholes</td>
<td>25</td>
<td>47</td>
<td>7</td>
<td>79</td>
</tr>
<tr>
<td>Gaussian Blur</td>
<td>16</td>
<td>25</td>
<td>63</td>
<td>104</td>
</tr>
<tr>
<td>N Body</td>
<td>12</td>
<td>4</td>
<td>4</td>
<td>20</td>
</tr>
<tr>
<td>SRAD</td>
<td>21</td>
<td>28</td>
<td>6</td>
<td>55</td>
</tr>
<tr>
<td>HH</td>
<td>21</td>
<td>64</td>
<td>30</td>
<td>115</td>
</tr>
<tr>
<td>HDR</td>
<td>8</td>
<td>6</td>
<td>28</td>
<td>42</td>
</tr>
<tr>
<td>Matrix Multiply</td>
<td>4</td>
<td>3</td>
<td>21</td>
<td>28</td>
</tr>
<tr>
<td>Matrix Inverse</td>
<td>12</td>
<td>57</td>
<td>79</td>
<td>148</td>
</tr>
<tr>
<td>Planck Distribution</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>

**Table 5.3: FU Latencies.**

<table>
<thead>
<tr>
<th>FU Type</th>
<th>Add/Sub</th>
<th>Mult</th>
<th>Div</th>
<th>Sqrt</th>
<th>Exp</th>
<th>Log</th>
<th>CMP_MUX</th>
<th>Abs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latencies</td>
<td>7</td>
<td>5</td>
<td>14</td>
<td>28</td>
<td>17</td>
<td>21</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

This section shows scheduling results for the benchmarks used in this study and discusses which algorithms perform better. For the configurations in the following sections we will refer to the FU mix using 6 digits each indicating the number of Add/Sub-Multiply-Divide-Square Root-Exponent-Logarithm units (e.g. 2-3-1-1-1-1 means 2 Add/Sub, 3 Multiply, 1 Divide and so on). CMP_MUX and Abs FUs are not indicated in the FU Mix. They have a very small area, and are implemented as part of other FUs such as Exp or
Chapter 5. Compiling for TILT

Log. In some of the explanations we will refer to data in Table 5.2 about the number of operations in the Longest Path of the DFG.

To run our compiler we provide as input the DFG, a configuration file with FU latencies, and a list of parameters (FU Mix, Threads, Banks) that we want to simulate. The compiler schedules the DFG using different algorithms, and records schedule length, and throughput. For these results, we assume banks with 2 read ports and 1 write port, meaning they issue and complete 1 operation each cycle. Each thread resides in its own memory bank and no sharing is performed among threads. The read latency for getting operands from memory banks is 8 cycles, write latency is 5 cycles, and a pair of write-read operations can be overlapped by issuing the read 1 cycle after a write has been issued. We have used the FU latencies in Table 5.3. The floating-point multiplier uses dedicated FPGA resources, which explains its lower latency of operation. Throughput, reported in operations-per-cycle (OPC), is calculated by dividing the total number of operations scheduled across all threads by the schedule length in cycles. Theoretical throughput for an FU mix is calculated as the throughput we could achieve if there were infinite threads (equivalent to having no DFG dependences), and is limited by the busiest FU of the DFG. It is the highest possible throughput for a specific combination of FU instances and FU usages (determined by DFG). Since the TILT parameter space is vast, the discussion only includes the base configuration (1 FU instance for each type), and the configuration that has the highest compute density which varies between benchmarks. In cases where these two configurations are the same only one is shown in the results. Compute density is calculated as the ratio of throughput to the hardware area of a configuration, and will be discussed in the next chapter.

5.3.1 Black Scholes

Figure 5.2: Scheduler results for Black Scholes.
As seen in Figure 5.2(a), the heuristics that produce better results in this case are Longest Path and Greedy Mix. The Grouped PGM Longest Path approach gives the best schedule throughput for this benchmark. The DFG contains 25 Longest Path nodes and 47 of their dependences, which comprises 72 out of 79 total nodes. Greedy and Prioritized Greedy perform poorly by themselves.

In the 2-3-1-1-1-1 configuration the number of FUs is proportional to the DFG usage of 23 AddSubs and 32 Multiplies. As seen in Figure 5.2(b), for a smaller number of threads, all algorithms have similar performance, due to the FU mix which relieves the bottleneck in scheduling. At a high number of threads such as 64, Prioritized Greedy Mix and Grouped PGM Longest Path perform better. This is due to the high amount of Add, Sub and Mult operations in the Longest Path, and as the number of threads increases it becomes important to prioritize them.

5.3.2 Gaussian Blur

For the results of this benchmark shown in Figure 5.3, the Greedy Mix approach improves throughput more than the other approaches. The best scheduling approaches to get a high throughput for any number of threads are Greedy Mix and Greedy Mix with Longest Path (LP). Prioritizing operations has a negative effect on the schedule performance. There is a problem with the priority mechanism—used to ensure fairness when using operation priorities— which holds back operations from being scheduled to make sure all the operations with the same start cycle are in the ready list before proceeding. This problem affects only Gaussian Blur and further analysis showed that by delaying the scheduling of a thread’s operations, another thread can fill several cycles not allowing for a fair interleaving of operations between threads. A similar trend is seen in the 3-3-1-1-1-0 configuration.
5.3.3 N Body

Figure 5.4: Scheduler results for N Body.

For both results shown in Figure 5.4 Grouped PGM Longest Path performs better. Using Greedy Mix or Prioritized Greedy by themselves does not improve the schedule. We are dealing with a small DFG (only 20 nodes), where 16 of 20 nodes are included in the Longest Path set. Therefore the Longest Path algorithm has the most significant effect on the schedule throughput. Greedy Mix is not as effective, due to the small size of the DFG, which implies that there is not a significant delay between each thread scheduled. The same conclusions hold for the 2-2-1-1-0-0 configuration.

5.3.4 SRAD

Figure 5.5: Scheduler results for SRAD.

For the results shown in Figure 5.5, Grouped PGM Longest Path gives us the best performance results over different number of threads. The main reason is because 49
nodes out of 55 are part of the Longest Path. The rest of the algorithms perform well for smaller number of threads but at higher number of threads Grouped PGM Longest Path dominates.

### 5.3.5 Hodgkin-Huxley (HH)

![Graph showing scheduler results for HH.](image)

The results in Figure 5.6, show that the approaches that produce better results in this case are Greedy Mix and Longest Path. The DFG of this benchmark contains 21 nodes in the Longest Path and 64 nodes that are dependences of the Longest Path nodes which is a total of 85 out of 115 DFG nodes. For this reason applying the Grouped PGM Longest Path heuristic produces better results. Since the DFG is relatively large with 115 nodes it helps to mix operations between different threads.

### 5.3.6 HDR

In the results of Figure 5.7(a), Greedy Mix and Grouped PGM Longest Path perform well. Algorithms that do not use the Greedy Mix approach produce a less efficient schedule. This benchmark performs 3 symmetrical computations on different data but only 1 group of the symmetric operations is identified as part of the Longest Path of the DFG. This includes 14 out of 42 nodes, which is not a majority. Therefore, using Longest Path alone does not improve the schedule due to the symmetry, but it can be scheduled more efficiently if the Greedy Mix approach is enabled.

In the 2-1-1-0-0 configuration, shown in Figure 5.7(b), Grouped PGM Longest Path (which combines all the approaches) performs best. Greedy is very efficient for high number of threads, but not for small numbers.
Chapter 5. Compiling for TILT

5.3.7 Matrix Multiply

The DFG for Matrix Multiply contains 4 symmetrical operations that perform multiplication and addition. Similarly to the HDR benchmark, Grouped PGM Longest Path gives better results for this benchmark as shown in Figure 5.8.

5.3.8 Matrix Inverse

For this benchmark using Greedy Mix and Longest Path algorithms helps produce faster schedules, as seen in Figure 5.9(a). The DFG contains 148 operations (the largest from these benchmarks), therefore mixing operations from different threads is useful. The Longest Path operations (68/148) are a considerable part of the DFG, which explains why this heuristic is also necessary. In the 1-2-1-0-0-0 configuration Grouped PGM
Longest Path (which combines all the approaches) and Greedy perform well as shown in Figure 5.9(b). We notice a particular behaviour where the Greedy Mix algorithm performs much worse in this configuration. To understand the reason behind this, we note that Matrix Inverse has an FU Usage of 55 AddSub, 92 Multiply, and 1 Divider. By increasing the number of multipliers to 2, but leaving AddSub at 1 we have allowed the schedulers to exploit parallelism among multiply operations and improve the overall schedule compared to 1-1-1-0-0-0, but the AddSub unit has turned into a bottleneck. Since the multiply operations are dependent on the adds and subtractions, at high numbers of threads Greedy Mix increases the finish time of a thread by scheduling other threads in the single AddSub unit. In this occasion it is not a good idea to use Greedy Mix. From looking at other simulation results not shown here, this behaviour repeats whenever the number of Multipliers is higher than the number of AddSub units.

5.3.9 Planck Distribution

Planck Distribution has a very small DFG. We see that all the algorithms produce almost identical throughputs in Figure 5.10(a) and Figure 5.10(b). Having a small DFG and functional units with deep pipeline stages makes it easy for multiple threads to be scheduled with negligible effect from the scheduling algorithm.

5.3.10 Conclusion

As seen in the scheduling results of different benchmarks described in the previous sections, different approaches can be useful depending on the nature of the application’s DFG, and hardware constraints. Despite these differences, the combined approach which
we call the Grouped PGM Longest Path approach is the most effective approach overall, and achieves 90% of the theoretical throughput for 64 threads. This is seen in Figure 5.11, where we show the throughput of each scheduling algorithm after averaging across all the results of the previous section, and normalizing between zero and one, where one is the theoretical throughput. Normalizing the results reduces the apparent improvement over the Greedy approach, but depending on the configuration simulated there are significant
improvements when using Grouped PGM Longest Path.

## 5.4 Memory Allocator for DFG

LLVM IR uses SSA format to name the registers that store the results produced by each instruction in the code. By allocating a memory location to each SSA variable no memory is reused, leading to high memory usage for a DFG. We can perform memory allocation to reuse locations and drastically reduce the memory required by the DFG. In this section, I will describe the algorithm used to assign memory locations to the DFG operations. The memory allocation is performed based on the DFG, before knowing the schedule of operations. Therefore we can only make assumptions about information that is present in the DFG. Memory allocation does not affect the DFG, therefore scheduling can be performed without having to know if memory allocation was performed or not. The IR code can contain references to i) temporary registers (SSA format), ii) and memory variables allocated in the source code. We assign a distinct location to each memory variable and never reuse it to preserve correctness. Alternatively, locations assigned to temporary registers can be reused as soon as all the operations that read from that location are done reading the operand. The method shown in Algorithm 5, performs memory allocation assuming there is a separate address space for each thread, so multiplying by the number of threads is required to calculate the total memory usage of a benchmark.

- First we need to create an Availability Matrix (NxN entries) which contains an entry for each pair of vertices in the DFG. An entry $e[i][j]$ of the matrix is set to true, if all the paths exiting node $i$ have to go through node $j$ before reaching the end of the paths in the graph. Otherwise it is set to false. This similar to computing the postdominators of the nodes in the DFG.

- Second, for each node in the DFG, we keep track of the total number of edges exiting that node. This is required to figure out when the value of this node is no longer required, and its location can be reused.

- To perform memory allocation we visit each node in topological order, which makes sure a node is visited only after all of its predecessors have been assigned a memory location. The free list is used to keep memory locations so they can be reused when it is possible. A memory location from the free list is reused only if it was previously used by a node postdominated by the current node. This prevents reusing locations that may not be safe to use (i.e. could be still in use by other nodes). A new location is allocated only when it is not possible to reuse one from the free list.
Data: DFG
Result: Memory allocation
Assign permanent memory locations to source code variables
Create Availability Matrix based on Postdominator Information
Create an empty freeList
Create EdgeCount table with number of edges exiting each node;
forall the DFG nodes in Topological order do
  read current;
  foreach Incoming Edge do
    Decrement count of source vertex in EdgeCount;
    if count is zero then
      Deallocate memory by putting it in freeList
    end
  end
  foreach Entry in freeList do
    if entry was previously used by a node postdominated by this node then
      Remove from freeList and use memory location for current node;
      break
    end
    if no memory location assigned from freeList then
      allocate new location and use;
    end
  end
end

Algorithm 5: Memory Allocation for DFG.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Original Usage</th>
<th>Usage after allocation</th>
<th>% Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Black Scholes</td>
<td>93</td>
<td>38</td>
<td>59.14</td>
</tr>
<tr>
<td>Gaussian Blur</td>
<td>114</td>
<td>101</td>
<td>11.40</td>
</tr>
<tr>
<td>N-Body Interaction</td>
<td>28</td>
<td>21</td>
<td>25</td>
</tr>
<tr>
<td>SRAD</td>
<td>64</td>
<td>42</td>
<td>34.38</td>
</tr>
<tr>
<td>Hodgkin-Huxley</td>
<td>148</td>
<td>84</td>
<td>43.24</td>
</tr>
<tr>
<td>High Dynamic Range</td>
<td>68</td>
<td>50</td>
<td>26.47</td>
</tr>
<tr>
<td>Matrix Multiply</td>
<td>47</td>
<td>39</td>
<td>17.02</td>
</tr>
<tr>
<td>Matrix Inverse</td>
<td>155</td>
<td>95</td>
<td>38.71</td>
</tr>
<tr>
<td>Planck Distribution</td>
<td>13</td>
<td>11</td>
<td>15.38</td>
</tr>
</tbody>
</table>

In Table 5.4 we show the results with the memory savings after applying our memory allocator. The savings range from 11.4% for Gaussian Blur to 59.14% for Black Scholes. For Gaussian Blur and Matrix Multiply we achieve small reductions since their DFGs contain respectively 5 and 4 parallel paths and the allocator cannot make assumptions
for parts of the DFG that are not connected to each other, thus cannot optimize across them. Black Scholes and Hodgkin-Huxley contain longer paths of dependent operations in their DFGs, thus allowing for a higher amount of storage locations to be reused.

5.5 Summary

In this chapter we presented and evaluated different scheduling approaches that vary in how they schedule operations across multiple threads and in prioritizing DFG operations. These approaches were evaluated on nine benchmarks, and we concluded that an approach combining multiple heuristics is better on average. We also presented a memory allocation algorithm that reduces memory usage for the benchmark applications. These scheduling approaches are used to gather the results in the next chapter and are combined with hardware results.
Chapter 6

Hardware Results

In this section we will combine hardware results with scheduler results to help us illustrate the trade-off between throughput and area by navigating the design space for a base design (FU Mix 1-1-1-1-1-1). We show area breakdowns for TILT and estimate the potential of replicating TILT cores compared to the custom hardware implementations. Lastly, we evaluate the potential of architecture optimizations such as Port Sharing and Schedule Compression.

6.1 Evaluation Details

We use Altera’s Quartus 12.0 targeting the Stratix IV EP4SE530H40C2 FPGA, a device of the highest available speed grade. All designs were implemented using generic Verilog-2001 without any Altera-specific modules except for the FUs. We report area in equivalent Adaptive Logic Modules (eALMs), to roughly account for the actual silicon area of any ALMs, BRAMs, and Digital Signal Processing (DSP) Blocks used by each design: Wong et al. [63] report that an M9K BRAM has an area equivalent to 28.7 ALMs, an M144K BRAM is 267 ALMs, and one 18-bit DSP block is 14.875 ALMs. We use fully pipelined IEEE-754-compliant 32-bit floating-point units generated by Altera’s MegaWizard Plugin Manager. All designs reported support an operating frequency ($F_{MAX}$) of at least 200MHz, where the exponent FU is typically the limiting component.

We use compute density to identify designs that balance between throughput and area usage. Compute density (CD) is calculated as the ratio of throughput to hardware area, and designs with the highest CD are considered the most efficient. CD is reported in $\text{OPC}/(10k \text{ eALMs})$, where the units of 10k eALMs are used to avoid very small decimal numbers. The densest per benchmark TILT configuration is found by calculating the CD over the design space of a benchmark and selecting the one with highest CD.
6.2 TILT Comparisons

We compare TILT with the performance and area of a soft scalar CPU, and with custom hardware implementations of benchmarks. The soft scalar CPU is similar to a NIOS having support for floating-point units via the custom instruction interface, such that only one operation issues at a time in a non-pipelined (blocking) fashion. This processor has a read latency of 4 cycles, and a write latency of 2 cycles. This implementation is a good baseline for comparison since it is implemented using the same building blocks, and we can use it to show the benefits of supporting ILP and TLP on TILT compared to a single issue processor. For the custom hardware implementations, each application datapath is fully laid-out in hardware, and fully pipelined such that an independent thread can potentially reside in each pipeline stage of the design. Every operation in the data flow graph has a dedicated FU instantiated for it, and all independent and conditional paths are evaluated in parallel. We use a regression model based on the area of the functional units, to approximate the area of the custom hardware designs without implementing them in hardware. We do not include in our estimate the cost of a data fetching mechanism, and assume that all the data is already residing in memory. An external mechanism such as CoRAM [25] or LEAP scratchpads [13] would be used to transfer data in the background. The throughput (OPC) of custom hardware is calculated assuming there is a thread for each pipeline stage of the DFG’s critical path. The custom hardware is a representation of the most application specific implementation, and gives us an upper bound on the throughput of a benchmark as well as good reference for the hardware area.

6.3 Reducing the Gap with Custom Hardware

6.3.1 Base configuration

In the following figures, we evaluate the throughput, area and compute density of a base configuration (one of each FU) by varying the number of memory banks from 1 to 64 and depth from 128 to 1024 32-bit words. These results are averaged across all the benchmarks, and they are used to find an architecture with the highest density averaged over all the benchmarks for an FU mix of 1 each. Each benchmark’s data requirement may vary, so we started at 128 word banks which provides enough capacity for the largest benchmarks (Hodgkin-Huxley, Matrix Inverse). To calculate the throughput we have chosen the highest number of threads that each benchmark can fit inside a bank without causing throughput deterioration (e.g. having 8 threads-per-bank instead of 4
can produce lower throughput due to schedule conflicts).

![Figure 6.1: Throughput averaged across benchmarks for a base design (FU Mix 1-1-1-1-1-1) with 1 to 64 banks and depths of 128, 256, 512 and 1024 words.](image)

In Figure 6.1 we see that throughput improves almost linearly when banks are increased from 1 to 16 for a 128 words-deep design, from 1 to 8 banks for a 256 words-deep design, and from 1 to 4 banks for the 512 and 1024 words-deep designs. Adding more banks increases the amount of TLP that we can extract from a benchmark until it reaches a point where TLP does not improve throughput due to the FU Mix becoming a bottleneck. For larger designs, the maximum throughput is achieved more quickly since deeper banks can accommodate more threads at the same time.

In Figure 6.2 we see that area grows very quickly with the number of banks, mainly due to

![Figure 6.2: Total TILT area for a base design (FU Mix 1-1-1-1-1-1) with 1 to 64 banks and depths of 128, 256, 512 and 1024 words.](image)
to crossbar area. This makes 64 bank designs extremely costly. There is only a slight increase due to bank depth, which shows that it is not a very important factor compared to crossbar area.

As shown in Figure 6.3, for each different bank depth configuration adding more banks improves density up to 8 banks for the 128 words-deep design, up to 4 banks for the 256 and 512 words-deep designs, and up to 2 banks for 1024 word designs. This indicates that the improvement in throughput facilitated by the extra banks is worth the extra area used. The density drops for 16 or more banks in the 128 words-deep design, for 8 or more banks in the 256 and 512 words-deep designs, and for 4 or more banks in 1024 words-deep designs. In these cases, adding more banks causes slight improvement in throughput while the area increases quickly. We also notice that each bank depth has a different curve. The 512 words-4 banks design is the computationally densest overall, while the 256 words-4 banks design is second by a slight difference. On two different extremes, we have the 128 words designs, and the 1024 words designs. The 128 words designs have lower area, but they do not have enough hardware resources to support a high number of threads, therefore their throughput and densities are low. The 1024 words designs have higher area than all the other designs. Although they can support a high number of threads, their density is lower due to their large area. We see a bell shaped graph within each group of results for the same bank depth, and across different depths of the same number of banks (e.g. 4 banks across 128 to 1024 word banks). This shows how each configuration parameter (banks and depth) affects density and the densest design balances throughput and the area controlled by the number of these resources.
6.3.2 Area & Throughput Comparisons

In this section, we will compare the throughput of a scalar soft processor, TILT, and custom hardware. Also, we will compare the area of TILT to custom hardware implementations, and analyze area breakdown for the densest per benchmark TILT configuration.

![Bar chart comparing throughput of a soft scalar CPU, single-thread base TILT (having one bank), 4-bank base TILT (the densest overall, with multiple threads), and custom hardware implementations.](chart)

In Figure 6.4, we compare the performance of TILT with that of the soft scalar CPU and the estimated custom hardware implementations. First, we observe that the performance difference between the soft scalar CPU and custom hardware implementations is close to three orders of magnitude. The single-thread instance of TILT (with one of each FU) improves on the soft scalar CPU by supporting pipelined execution within and across the FUs, limited by the dependences in the data flow graph of the single thread. Throughput, measured in operations-per-cycle (OPC), is improved from 0.12 to 0.27 OPC on average. For N Body, SRAD and Planck Distribution, throughput for the soft scalar CPU and single-thread TILT is similar, indicating limited pipeline parallelism within a single-thread for those applications. We also show the throughput of the 4 bank-512 words depth base TILT, which has an average throughput of 2.02 OPC. On average across all benchmarks, the custom hardware implementations are 32.6x faster than this dense version of TILT. However, TILT is capable of executing all of the benchmarks while an independent custom hardware implementation exists for each application.

Figure 6.5 compares the area (eALMs) of the base TILT, the densest TILT per benchmark (see Table 6.2 for details), and the custom hardware. Base TILT contains one of each FU types, and 4 banks of depth 512 words each (1 word equals 32 bits) that can fit...
different numbers of threads for each benchmark depending on their data requirements. It is also significantly smaller than the custom hardware implementations, except for N Body, Matrix Multiply and Planck Distribution—those applications use few FU types and have relatively small data flow graphs. The densest TILT instance for each benchmark varies widely in size. For Black Scholes, the densest TILT is slightly larger than base TILT for having 1 extra AddSub and 2 extra Multiply units. For Gaussian Blur, it contains 2 extra AddSub and 2 extra Multiply units. For the remaining applications, the densest TILT is smaller due to the lower amount of FUs required by each.

Figure 6.5: Area of base TILT (1 of each FU), an aggressive 12-FU instance, the densest-per-benchmark instance, and estimated area of custom hardware.

Figure 6.6: Area breakdown for the densest TILT configuration for each benchmark.
Figure 6.6 shows the area breakdown for the densest TILT configuration for each benchmark. We see that FUs comprise roughly half of the area for all of the designs, which is a large fraction devoted to hardware that does the “real work” of computing. Hardware devoted to control (including instruction memory) and data memory is modest, accounting for less than 20% of area combined. Interconnect comprises more than 30% of area; crossbar-interconnect is what allows TILT to support multiple applications therefore it is necessary. Choosing designs by compute density favors those with a high throughput to area ratio. Since FUs are the component which has a big impact on the throughput of a design, these comprise the majority of the area in the densest designs. On the other hand it is not reasonable to dedicate a large amount of resources to data memory and control. The improvements from those components are minor compared to the FUs.

In this section we showed how TILT performs compared to a scalar soft processor, and custom hardware for all the benchmarks. Area comparisons showed that base TILT and the densest per benchmark are on average much smaller than the custom hardware implementations. Area breakdown indicated that FUs dominate the cost while interconnect ranks second, and could benefit from improvements.

### 6.3.3 Replication Results

The following results will show the potential of TILT when its cores are replicated. We perform this calculation using 3 versions of TILT: i) the densest per benchmark, ii) the base densest, and iii) the densest overall. The base densest is a design with an FU Mix of 1-1-1-1-1-1 and 4 banks of 512 word depth. The densest overall has an FU Mix of 2-2-1-1-1-1 and 4 banks of 512 word depth. The area of the custom hardware is used as an upper bound on the number of replicated cores for each benchmark.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Densest per benchmark</th>
<th>Base Architecture</th>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OPC</td>
<td>OPC Total OPC</td>
<td>OPC Slo(\text{down})</td>
</tr>
<tr>
<td>Black Scholes</td>
<td>3.09</td>
<td>2</td>
<td>10.18</td>
</tr>
<tr>
<td>Gaussian Blur</td>
<td>4.93</td>
<td>3</td>
<td>14.79</td>
</tr>
<tr>
<td>N Body</td>
<td>2.92</td>
<td>1</td>
<td>2.92</td>
</tr>
<tr>
<td>SRAD</td>
<td>2.02</td>
<td>7</td>
<td>14.12</td>
</tr>
<tr>
<td>HH</td>
<td>2.69</td>
<td>12</td>
<td>32.23</td>
</tr>
<tr>
<td>HDR</td>
<td>2.53</td>
<td>4</td>
<td>10.11</td>
</tr>
<tr>
<td>Matrix Multiply</td>
<td>1.37</td>
<td>7</td>
<td>9.59</td>
</tr>
<tr>
<td>Matrix Inverse</td>
<td>2.24</td>
<td>15</td>
<td>33.57</td>
</tr>
<tr>
<td>Planck Distribution</td>
<td>2.48</td>
<td>1*</td>
<td>2.48</td>
</tr>
<tr>
<td>Mean</td>
<td>2.92</td>
<td>14.43</td>
<td>4.41x</td>
</tr>
</tbody>
</table>
Table 6.2: Details of densest TILT designs per benchmark.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>FU Mix</th>
<th>Threads</th>
<th>Banks</th>
<th>Depth (words)</th>
<th>Schedule Length (cycles)</th>
<th>Throughput</th>
<th>Compute Density (OPC/(10k eALMs))</th>
<th>Area eALMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Black Scholes</td>
<td>2-3-1-1-1-1-1</td>
<td>64</td>
<td>8</td>
<td>512</td>
<td>993</td>
<td>5.09</td>
<td>3.23</td>
<td>15772</td>
</tr>
<tr>
<td>Gaussian Blur</td>
<td>3-3-1-1-1-0</td>
<td>32</td>
<td>8</td>
<td>512</td>
<td>675</td>
<td>4.93</td>
<td>3.36</td>
<td>14669</td>
</tr>
<tr>
<td>N Body</td>
<td>2-2-1-1-0-0</td>
<td>64</td>
<td>4</td>
<td>512</td>
<td>439</td>
<td>2.92</td>
<td>4.19</td>
<td>6954</td>
</tr>
<tr>
<td>SRAD</td>
<td>1-1-0-0-0</td>
<td>32</td>
<td>4</td>
<td>512</td>
<td>809</td>
<td>2.02</td>
<td>4.86</td>
<td>4151</td>
</tr>
<tr>
<td>HH</td>
<td>1-1-0-1-0-0</td>
<td>16</td>
<td>4</td>
<td>512</td>
<td>685</td>
<td>2.69</td>
<td>4.61</td>
<td>5830</td>
</tr>
<tr>
<td>HDR</td>
<td>2-1-1-0-0-0</td>
<td>32</td>
<td>4</td>
<td>512</td>
<td>532</td>
<td>2.53</td>
<td>3.97</td>
<td>6356</td>
</tr>
<tr>
<td>Matrix Multiply</td>
<td>1-1-0-0-0-0</td>
<td>16</td>
<td>2</td>
<td>512</td>
<td>327</td>
<td>1.37</td>
<td>6.87</td>
<td>1995</td>
</tr>
<tr>
<td>Matrix Inverse</td>
<td>1-2-1-0-0-0</td>
<td>8</td>
<td>4</td>
<td>256</td>
<td>529</td>
<td>2.24</td>
<td>4.83</td>
<td>4631</td>
</tr>
<tr>
<td>Planck Distribution</td>
<td>1-2-1-0-1-0</td>
<td>64</td>
<td>4</td>
<td>512</td>
<td>155</td>
<td>2.48</td>
<td>3.94</td>
<td>6283</td>
</tr>
</tbody>
</table>

Table 6.3: Estimated Performance of Replicating the Overall Densest Core.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Overall Densest Architecture</th>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Benchmark</td>
<td>OPC eALMs</td>
<td>Cores</td>
</tr>
<tr>
<td>Black Scholes</td>
<td>3.11 11144 3</td>
<td>9.32 8.48x</td>
</tr>
<tr>
<td>Gaussian Blur</td>
<td>2.52 11144 5</td>
<td>12.59 8.26x</td>
</tr>
<tr>
<td>N Body</td>
<td>2.92 11144 1*</td>
<td>2.92 6.86x</td>
</tr>
<tr>
<td>SRAD</td>
<td>2.61 11144 2</td>
<td>5.22 9.77x</td>
</tr>
<tr>
<td>HH</td>
<td>2.87 11144 6</td>
<td>17.22 6.68x</td>
</tr>
<tr>
<td>HDR</td>
<td>2.64 11144 2</td>
<td>5.28 7.95x</td>
</tr>
<tr>
<td>Matrix Multiply</td>
<td>2.74 11144 1</td>
<td>2.74 10.22x</td>
</tr>
<tr>
<td>Matrix Inverse</td>
<td>2.87 11144 6</td>
<td>17.24 8.58x</td>
</tr>
<tr>
<td>Planck Distribution</td>
<td>2.48 11144 1*</td>
<td>2.48 2.42x</td>
</tr>
<tr>
<td>Mean</td>
<td>2.75 11144 1*</td>
<td>2.48 65.89</td>
</tr>
</tbody>
</table>

In Table 6.1, we show results estimating how fast each benchmark can run compared to the custom hardware implementation if the cores of our processor are replicated. The number of replicated cores is determined by calculating how many TILT cores can fit inside the area of the custom hardware implementation. In cases where the custom hardware is smaller than the processor core, we assume replication of 1 (denoted by an asterisk *). The Total OPC is calculated by multiplying the number of cores by the OPC of a single TILT core, which is computed through schedule simulation. Slowdown is calculated by dividing the OPC of the custom hardware by the Total OPC of the benchmark running on replicated cores. These results do not take into account interconnection between cores or resource availability on the FPGA, but are a numerical estimate to demonstrate the potential of this method. By replicating the densest design per benchmark, we have replication factors ranging from 1 to 15. The low replication factors occur for the benchmarks with very small custom hardware area. In these cases, only a small number of cores can be replicated before exceeding the area of custom hardware, which is our upper bound on total area of replicated cores. Conversely, higher replication factors correspond
to benchmarks with very high custom hardware areas. The slowdown compared to custom implementations ranges from 2.42x to 7.76x for a geometric mean of 4.41x. More details on the densest-per-benchmark configuration are shown in Table 6.2.

Again in Table 6.1, we look at a more general case, where we replicate a base architecture of 1 FU each that can support all the benchmarks and is tuned to the average, not a specific benchmark. For this architecture, the slowdown factor ranges from 2.92x for Planck Distribution to 17.03x for Matrix Multiply for a geometric mean of 9.03x. Although these numbers are not as favorable as the application specific cores, they show that a general and minimal core performs well although some throughput is lost at the expense of increased programmability.

In Table 6.3, we show similar results, but in this case we have created a configuration that is the densest over the whole design space for these benchmarks. This design has an FU Mix of 2-2-1-1-1-1 with 4 banks of 512 word depth. It has an area of 11144 eALMs which is higher than the base architecture discussed previously (10065 eALMs). The replication factors range from 1 to 6, which are lower than for the base architecture but in this case we have a configuration with a higher average OPC. The slowdown ranges from 2.42x to 10.22x for a geometric mean of 7.21x which is better than the base architecture.

In this section we showed the potential performance of replicating TILT. By replicating densest-per-benchmark cores, we achieved the best performance: 4.41x slower than custom hardware. By considering more general designs that support all benchmarks we achieve 7.21x slowdown for an overall densest core, and 9.03x for a base densest core.

6.4 Port Sharing

By analyzing hardware area results we notice that the interconnect is a significant percentage of the total area. Port Sharing would reduce the area of the interconnect by combining multiple functional units such that they share the same connection to the memory banks instead of having separate connections. This would come at the expense of a small multiplexer that controls which FU is using the connection to read or write results at a particular time. In terms of scheduling, this means that operations from FUs that share the same port cannot issue or complete at the same cycle. To generate a proper schedule, these operations have to be treated as if they execute in the same FU. By scheduling in this manner, we reduce the effective amount of available FU issue slots, which has a potential to increase schedule length especially for designs with a high number of scheduled threads. We evaluate this method on the Black Scholes benchmark,
which uses the highest number of functional units, and hence has the biggest potential for improvement on crossbar area.

We simulate the effect of Port Sharing on the area of Black Scholes by gathering hardware area results on a design with 4 FUs: Add/Sub, Multiply, Div, Square Root. From these results, we extract the area of the read and write crossbar. Port Sharing would reduce the complexity of the crossbar, therefore we estimate its cost by using the results of crossbar configurations with a smaller number of FUs. The extracted crossbar area is combined with the area of the other components to calculate the total hardware area of a configuration. To calculate schedule lengths for this experiment the operations of the 3 least popular FUs of this benchmark, Square Root, Log and Exponent, were scheduled under the same FU to simulate the limitations of sharing a port among FUs.

Figure 6.7: Effects of Port Sharing on hardware area, throughput and density for the top 20 designs of Black Scholes. The designs on the X-axis are specified by this format: AddSub-Mult-Div-Sqrt-Exp-Log-Threads-Banks.
In Figure 6.7, we show the top 20 Black Scholes designs (sorted by compute density). Graphs 6.7 (a) to (d) show the changes in crossbar area, throughput, total area, and compute density for these designs. By sharing ports we get definite savings in crossbar area (Figure 6.7(a)) from 650 eALMs in an 8 FU-4 Banks design (2-2-1-1-1-1-32-4) to 2432 eALMs in an 11 FU-8 Banks design(2-3-1-1-3-1-64-8). This result coincides with the fact that crossbar area grows quickly as we add more FUs and banks to it.

Analyzing the throughput graph in Figure 6.7(b), we see drops in throughput up to 0.95 OPC for the 3-3-2-1-2-64-8 design. This is caused by the increased number of conflicts in the scheduler when 3 FUs share the same port. For the 3-3-2-1-1-2-64-8 design we see a slight increase in throughput. Although this is counter-intuitive, we are using a scheduling heuristic for an NP-complete problem so in some cases the results are sub-optimal. In our scheduling algorithms, we take into account read and write port limitations of a design in addition to FU availability. The problem is caused by a long latency operation which is scheduled much earlier but completes (writes the result) at the same time as a shorter operation that is scheduled afterwards. In this case, the shorter operation has to be postponed, thus delaying the whole schedule. With a smaller number of FUs the first operation gets delayed by other operations, which removes the conflict and creates a faster schedule. An attempt to correct this problem, by keeping on hold all operations that write at the same cycle and scheduling them based on their priority, did not work effectively and introduced delays in other parts of the schedule. We can only conclude that this problem can be repaired on a case by case basis, and not in an algorithmically efficient manner.

In Figure 6.7(c) we show the area savings for the whole design (in eALMs) when Port Sharing is enabled. These numbers range from 343 to 2254 eALMs, and in relative terms up to 12% savings in total area. These savings are not consistent across all the designs, and are more pronounced for the larger designs. Unfortunately, it does not translate to improvements in the compute density of designs. In Figure 6.7(d), we see that the compute density (CD) of the original top 20 designs changes from -0.37 (drop) to 0.36 (increase). The top 2 designs experience a drop in CD, which is caused by throughput loss due to Port Sharing. While there are a few designs that gain CD, due to area savings and very small changes in throughput, it is not enough to improve on the previously best CD.

In conclusion, Port Sharing is an effective technique for saving on the crossbar area of a design but also comes at a considerable cost to the schedule performance of a design. Thus, it is not effective in improving the compute density of a design.


6.5 Schedule Compression

![Diagram](image)

Figure 6.8: Illustration of potential savings in instruction memory. The first figure shows the scheduled slots (highlighted) and empty slots. In the second figure, the empty slots are compressed to a single entry (checkerboard pattern) and separate storage and PC is used for each FU.

By observing instruction schedules of different benchmarks, we noticed that the instruction schedule of each FU contains regions that have multiple consecutive gaps where no instruction is issued. Based on that, it is worth exploring the option of compressing those empty cycles to reduce instruction memory storage requirements. The BlockRAMs used to store instruction memory can be configured in different width to depth ratios [4]. When the instruction schedule is considered as one whole unit it can produce inefficient memory configurations, since the memory depth depends on the schedule length which could depend on a single FU. Figure 6.8(a) illustrates an instruction memory that would benefit from compression. In this optimization, we assume that a group of empty (NOP) instructions can be collapsed into a single instruction containing the length in cycles until the next valid instruction. A separate Program Counter (PC) will be required for each FU, and it will wait for a specified number of cycles before executing the next instruction in the memory. An illustration of how it would look after compression is shown in Figure 6.8(b). In this case, the instruction memory will no longer resemble Horizontal Microcode.

To evaluate the effect of schedule compression, results were gathered for the Black Scholes and Gaussian Blur benchmarks. For each FU instance, we calculate separately the depth of the required instruction memory assuming it has been compressed. Ideally this approach will produce smaller memories of different sizes for each FU instance. The
Chapter 6. Hardware Results

Figure 6.9: Effects of schedule compression on Instruction Memory Area, Total Design Area and Compute Density. Figures (a)-(c) show results for Black Scholes, and (d)-(f) show results for Gaussian Blur. The designs on the X-axis are specified by this format: AddSub-Mult-Div-Sqrt-Exp-Log-Threads-Banks.

The total area of the instruction memory unit is calculated by summing the areas of individual memories.

For Black Scholes, as shown in Figure 6.9(a), we save from 99 eALMs (approximately 8%) to 1123 eALMs in area (approximately 52%). The area savings for the whole design, shown in Figure 6.9(b), range from 0.6% to 7.1%. The instruction memory consists an average 7.6% of the total design area. The improvement in CD (see Figure 6.9(c)) ranges from 0.01 to 0.23 OPC/(10k eALMs), which translates to a maximum of 7.6% improvement in CD.

For Gaussian Blur, as shown in Figure 6.9(d), we save from 43 eALMs (approximately 9%) to 1655 eALMs in area (approximately 57%). The area savings for the whole design, shown in Figure 6.9(e), range from 0.3% to 8.6%. The impact is diminished due to the instruction memory consisting only of an average 8.2% of the total design area. The
improvement in (see Figure 6.9(f)) CD ranges from 0.01 to 0.27 OPC/(10k eALMs), which translates to a maximum of 9% improvement.

This optimization method has the benefit of not affecting design throughput as it simply changes the schedule’s storage method. Our experiment demonstrates the potential of schedule compression, although the exact savings will depend on the implementation and possible overheads of separate instruction memories and program counters.

6.6 Summary

In this chapter, we presented a combination of hardware and compiler results illustrating the trade-offs between throughput and area. We also presented area results, and estimated the potential of a replicated TILT core compared to the custom hardware implementations. Lastly we evaluated Port Sharing and Schedule Compression, two potential architecture optimizations where Schedule Compression was the most promising one. In the next chapter we will describe methods used to accelerate and simplify the design search process for finding dense designs.
Chapter 7

Navigating TILT Design Space

As we have mentioned in previous chapters, TILT is a flexible architecture that can be configured to support different applications. The current architecture supports up to 6 different functional units\(^1\), and in the future, may be expanded to a higher number of units. The substantial number of functional units, and the adaptability to different applications create a vast design space that needs to be considered when finding the densest design configuration for a specific application. A brute force approach to design exploration would involve using the simulator to create schedules for every configuration of FUs, threads and banks. It would also require using Quartus to compile every hardware configuration in order to extract the hardware areas. Although we can tolerate the time required to run the scheduler even for very large design spaces, it is infeasible to wait for the hardware synthesis times where a single design can take on the order of hours. On the other hand, the scheduler simulation time of the whole design space can be completed in less than an hour in several cases. For the reasons mentioned above, we need a method to reduce the design space search, and to predict which designs have a higher potential for a specific application. After reducing the design space to a limited number of designs, it is feasible to run the schedule simulator and Quartus synthesis to verify the predicted results.

7.1 Simulator design space reduction

The design space of the benchmark applications that can run on TILT varies in size and can grow to become very large. For each FU mix, we simulate 7 different numbers of threads \(1, 2, 4, 8, 16, 32, 64\), and for each of these we simulate different numbers of threads

\(^1\)CMP, MUX and Abs are not counted in the 6 FUs, since they are incorporated inside other FUs.
per bank which effectively changes the number of banks from the current number of threads (1 thread-per-bank) to 1 bank (all threads in 1 bank). For example, a design with 16 threads would be simulated with 16, 8, 4, 2, 1 banks. When counted this creates a total of 28 pairs of thread-bank combinations for each FU mix. For our simulations we have a self-imposed limit of 3 instances for each FU. For a benchmark, such as Black Scholes, with 6 FU types, the total number of FU mixes that need to be simulated is $3^6 = 729$. In order to reduce the amount of designs that have to be simulated, we do not need to simulate all the possible FU mixes but only those that have a potential to improve the throughput of a DFG. For every benchmark, we start with an FU mix of 1 instance for all FUs (e.g. 1-1-1-1-1-1 for Black Scholes). We calculate the average utilization of an FU by dividing the number of operations in the DFG with the number of instances of an FU. The FU with the highest average utilization is considered a bottleneck and it is incremented by 1. The process continues until we reach our self-imposed limit of 3 instances for each FU. Such process is illustrated in Algorithm 6.

<table>
<thead>
<tr>
<th>Data: DFG usages</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Result:</strong> List of relevant FU Mixes</td>
</tr>
<tr>
<td>Create total usages[] array with number of DFG operations for each FU type</td>
</tr>
<tr>
<td>Create fu instances[] array with an entry for each FU type initialized to 1</td>
</tr>
<tr>
<td>Create average utilization[] array with an entry for each FU type</td>
</tr>
<tr>
<td>Create list fumixes list for adding results</td>
</tr>
<tr>
<td>MaxUnits = 3</td>
</tr>
<tr>
<td>TotalFUs = number of FU Types</td>
</tr>
<tr>
<td>Add fu instances to list fumixes</td>
</tr>
<tr>
<td>while True do</td>
</tr>
<tr>
<td>for fu ← 0 to TotalFUs − 1 do</td>
</tr>
<tr>
<td>average utilization[fu] = total usages[fu] / fu instances[fu]</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>MaxFU ← with index of maximum of average utilization</td>
</tr>
<tr>
<td>if fu instances[MaxFU] &lt; MaxUnits then</td>
</tr>
<tr>
<td>fu instances[MaxFU] = fu instances[MaxFU] + 1</td>
</tr>
<tr>
<td>Add fu instances to list fumixes</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>Break</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>

**Algorithm 6:** FU Mix calculator.

By following the algorithm, the simulator design space is reduced significantly. In Table 7.1, we show the full design space for our benchmarks, by counting the total
number of FU Mixes, and total number of designs after combining an FU Mix with all combinations of threads and banks. We also show the reduced number of FU Mixes, and total designs after using Algorithm 6. As seen in Table 7.1, the number of FU mixes that we have to simulate for Black Scholes is reduced from 729 to 5, which is equivalent to a reduction from 20412 to 140 designs after taking into account threads and banks combinations. Gaussian Blur, N Body, HH, HDR, and Planck Distribution have design spaces in the thousands of designs and these are reduced to a few hundreds. The percentage of savings ranges from 44 % for Matrix Multiply to 99 % for Black Scholes. Matrix Multiply has a small amount of savings because it starts with a smaller design space, and the reduction does not appear significant. For Black Scholes, the schedule simulation time decreases from 15 hours to 5 minutes, and for HH from 3 hours to 16 minutes. Although using only the operation count for each FU to figure out the bottleneck is not ideal due to DFG dependences, this method works well since we are simulating multiple independent threads which increases the number of independent operations in the schedule and reduces the overall effect of dependences on the schedule. This approach is very useful in finding TILT designs of a high density for a benchmark, and we verified that by using this algorithm we always encounter the design with the highest density in our benchmarks. This method creates more savings when dealing with applications that require a high number of FUs since such applications have a larger design space. It is more helpful when the FU usage in the DFG is heavily weighted towards a smaller subset of FUs. In such cases, the number of FU Mixes that have to be simulated is smaller. As seen in our results, HH and Planck Distribution which have a more balanced ratio of FUs are reduced to, respectively, 8 and 7 FU Mixes after the algorithm is used. Matrix Inverse, which heavily utilizes 2 out of 3 FUs, is reduced to 4 FU Mixes.

Table 7.1: Simulator space reduction.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Full Design Space</th>
<th>Reduced Design Space</th>
<th>Savings</th>
</tr>
</thead>
</table>
|                  | #FU Mixes | # Designs (FUMix-thr-
|                  | #FU Mixes | # Designs (FUMix-
|                  | #FU Mixes | % Simulated | % Not Simulated | # Not Simulated |
| Black Scholes    | 729    | 20412 | 5   | 140 | 0.69 | 99.31 | 20272 |
| Gaussian Blur    | 243    | 6804  | 5   | 140 | 2.06 | 97.94 | 6664  |
| N Body           | 81     | 2268  | 5   | 140 | 6.17 | 93.83 | 2128  |
| SRAD             | 27     | 756   | 6   | 168 | 22.22 | 77.78 | 588   |
| HH               | 81     | 2268  | 8   | 224 | 9.88 | 90.12 | 2044  |
| HDR              | 81     | 2268  | 5   | 140 | 6.17 | 93.83 | 2128  |
| Matrix Multiply  | 9      | 252   | 5   | 140 | 55.56 | 44.44 | 112   |
| Matrix Inverse   | 27     | 756   | 4   | 112 | 14.81 | 85.19 | 644   |
| Planck Distribution | 81   | 2268  | 7   | 196 | 8.64 | 91.36 | 2072  |
7.2 Predicting Hardware Area

Prediction of hardware area is a method that has the potential to save a considerable amount of time from design space exploration. To gather the data used to derive the following models, full TILT configurations were compiled using Quartus, and the area of different components was extracted from the compilation reports. The TILT processor consists of the following main components: a) Functional Unit Array; b) Instruction Memory Unit; c) Memory Banks and d) Interconnect.

**Functional Unit Array** The area of the Functional Unit Array mainly depends on the number and types of functional units instantiated. Each functional unit requires a certain number of ALMs (Adaptive Logic Modules) and some units also utilize Digital Signal Processing (DSP) Blocks. We use the linear regression model in Equation 7.1, to predict the total number of ALMs required by the Functional Unit Array. To train the regression for this model, area cost values were gathered from the different benchmarks since they require different subsets of the available functional units. A total of 750 different configurations are used to create the model. The average percent error for this formula is 1.13%.

\[
FU_{Area} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \theta_4 x_4 + \theta_5 x_5 + \theta_6 x_6
\]

\[
x_1 = \text{Number of AddSub Units}
\]

\[
x_2 = \text{Number of Multiply Units}
\]

\[
x_3 = \text{Number of Divide Units}
\]

\[
x_4 = \text{Number of Square Root Units}
\]

\[
x_5 = \text{Number of Exponent-Compare Units}
\]

\[
x_6 = \text{Number of Logarithm-Abs Units}
\]

\[
\begin{align*}
\theta_0 &= 116.876 \\
\theta_1 &= 596.692 \\
\theta_2 &= 315.282 \\
\theta_3 &= 582.177 \\
\theta_4 &= 615.676 \\
\theta_5 &= 718.929 \\
\theta_6 &= 1782.818
\end{align*}
\]
Through observation it was possible to find a linear formula for calculating the number of DSPs required by a configuration (Equation 7.2).

\[
DSP\_Count = \begin{cases} 
4 \times \text{Mult} + 16 \times \text{Div} + 8 \times \text{Log} & \text{if Exp} = 0 \\
4 \times \text{Mult} + 16 \times \text{Div} + 8 \times \text{Log} + 21 \times \text{Exp} + 1 & \text{if Exp} \geq 1
\end{cases}
\]

\(\text{Mult} = \text{Number of Multiply Units}\)
\(\text{Div} = \text{Number of Divide Units}\)
\(\text{Exp} = \text{Number of Exponent-Compare Units}\)
\(\text{Log} = \text{Number of Logarithm-Abs Units}\)

\textbf{Instruction Memory Unit} The Instruction Memory Unit is mainly comprised of a ROM which is generated using M9K memory blocks. We use a variable length instruction word whose length depends on several architectural parameters, and this determines the width of the instruction memory. Figure 7.1 illustrates the instruction width calculation for a single FU. The total width is found by multiplying by the number of functional units. The schedule length, which is determined by the scheduler’s results, is used as the instruction memory’s depth. We observe that Quartus chooses M9K configurations that have enough depth to fit the whole instruction memory. After that, it uses the corresponding width of that configuration to calculate the amount of blocks required to achieve the total width required for the instruction word. Based on the above, a formula is derived to estimate the number of M9K blocks used in this component.

A similar approach is used to calculate the amount of memory blocks used by the Memory Banks which store the thread data for the applications. TILT only supports data memory depths that are a power of 2 and a width of 32 bits. Therefore it is simple to calculate the amount of memory blocks required by the Memory Banks. For bank depths that are less than 4096 we use the following formula (Equation 7.3):

\[
\text{Block\_Count} = \lceil \text{Bank Depth} \times 32/\text{(M9K Size)} \rceil \times \text{Read ports} \times \text{Banks} \times \text{Write Ports}
\]
For bank depths that are greater or equal to 4096, M144K blocks are used and the formula has to be adjusted so it uses $M144K\text{Size} (144 \times 1024)$ instead of $M9K\text{Size} (8 \times 1024)$.

**Interconnect** The interconnect consists of the read crossbar and write crossbar. It is used to read results from memory banks into the functional units, and to write the results from them back to the memory banks. The amount of wiring used in the interconnect is proportional to the number of FUs, number of memory banks connected with them, and the depth of the memory banks connected which depends on the number of threads per bank. By using these features, a polynomial regression model was created to predict the area of the interconnect in eALMs.

\[
\text{ReadCrossbar\_Area} = \theta_0 + \theta_1 \log(\gamma) + \theta_2 \alpha \times \beta + \theta_3 \alpha^2 + \theta_4 \beta^2 + \theta_5 \alpha^3 + \theta_6 \alpha \times \beta \times \log(\gamma)
\]

$$\alpha = \text{Total Number of FUs}$$

$$\beta = \text{Number of Banks}$$

$$\gamma = \text{Number of Memory Blocks used}$$

\[
\begin{align*}
\theta_0 &= 5079.593 \\
\theta_1 &= 41.931 \\
\theta_2 &= 4757.458 \\
\theta_3 &= 1096.258 \\
\theta_4 &= 1365.484 \\
\theta_5 &= -1027.969 \\
\theta_6 &= 3536.846
\end{align*}
\]

(7.4)

The regression in Equation 7.4 was derived to calculate the area of the read crossbar. It is a polynomial equation involving 3 variables: total number of FUs, number of banks and number of memory blocks. By observing the data, the total number of FUs was the parameter that had the largest effect on the area. Therefore it is assigned a squared, and cubic term in the equation. The number of banks has a slightly smaller impact so it has a squared term. The number of memory blocks has a smaller impact than the other two parameters, therefore we calculate a logarithm of its value, and use it in the formula. To model the dependence on combinations of both FUs and number of banks we have a term with their product. We also have a term with the triple product of FUs, number of banks and logarithm of number of memory blocks. This is trained on 272 samples of the design space and it predicts with an average error of 7.11%. If we train on half of the
space (136 random samples), and test our model on the other half, the prediction error percentage varies only 2%.

\[ \text{WriteCrossbar Area} = \theta_0 + \theta_1\alpha + \theta_2\beta + \theta_3\log(\gamma) + \theta_4\alpha \times \beta + \theta_5\alpha^2 + \theta_6\beta^2 + \theta_7\alpha^3 + \theta_8\beta^3 + \theta_9\alpha \times \beta \times \log(\gamma) \]

\[ \alpha = \text{Total Number of FUs} \]
\[ \beta = \text{Number of Banks} \]
\[ \gamma = \text{Number of Memory Blocks used} \]

\[ \theta_0 = 2956.5294 \]
\[ \theta_1 = 445.3052 \]
\[ \theta_2 = 1253.6812 \]
\[ \theta_3 = -9.5533 \]
\[ \theta_4 = 1113.562 \]
\[ \theta_5 = -823.8821 \]
\[ \theta_6 = -3244.7555 \]
\[ \theta_7 = 636.4358 \]
\[ \theta_8 = 3058.2745 \]
\[ \theta_9 = 3331.911 \]

The regression in Equation 7.5 was derived to calculate the area of the write crossbar. It is polynomial equation involving the same three variables as the previous equation but a different polynomial. In this case we have a linear term for each of total number of FUs, number of banks and logarithm of number of memory blocks. We have a term with the product of total number of FUs and number of banks. In addition there is a squared and a cubed term for total number of FUs and number of banks. Finally there is a term with the triple product of FUs, number of banks and logarithm of memory blocks. This is trained on 272 samples of the design space and it predicts with an average error of 4.15%. If we train on half of the space (136 random samples), and test our model on the other half, the prediction error percentage varies less than 1%. The formulas presented in this section will be used to predict area and their evaluation is presented in the following sections.
7.3 Evaluation of architecture predictions

By using all the formulas that were described in Section 7.2, a calculator was implemented to predict the hardware area in eALMs for a specific hardware configuration. The input data to the calculator is: the FU Mix, Number of Threads, Number of Banks, Bank Depth, Number of Read and Write Ports, and Schedule Length. We evaluate the accuracy of the prediction model, by predicting the cost of all the configurations in a benchmark’s design space, and plotting a distribution of the percent error of these predictions in Figure 7.2.

In the results for Black Scholes, HDR, Matrix Inverse, N Body, HH, and Planck Distribution all the errors are between -10% and +10%. We consider this to be acceptable since there is some variability even when simulating the same configuration multiple times.
times. For the Gaussian Blur benchmark there is a considerable number of designs that are in the range of 14-20% error. Most of these are designs with 64 threads which translates to a long instruction schedule, precisely 6721 cycles. For these designs, the CAD tool is performing optimizations to reduce the size of the instruction memory, which leads to our estimate being larger than the measured values. The same thing happens with SRAD where a considerable number of designs have a 12-18% error, with a schedule length of 3416 cycles. One design prediction for Matrix Multiply is in the 18% error range. The read crossbar model does not predict accurately for designs that have a high number of memory banks and a high number of memory blocks due to the low amount of training data that was available for such configurations. This occurs because the CAD tool has problems compiling large designs, and sometimes it fails completely, therefore not producing much data for these configurations. Most likely these designs would not be attractive to implement due to their inefficient use of area. From the results in Figure 7.2, we deduce that the hardware area prediction model is fairly accurate for the design space of our benchmarks (on average 3.5% over all benchmarks).

7.3.1 Evaluating Compute Density Prediction

In this section, we evaluate the effectiveness of the hardware area prediction formula when it is used to calculate the computationally densest design for a benchmark. We use the simulated throughput values generated from the compiler, and combine them with the predicted hardware areas to calculate compute density (CD). In Figure 7.3, the designs are sorted by the predicted compute density, and we note how many times the predicted densest design corresponds to the measured densest design. For the following benchmarks we correctly predict the densest design: Black Scholes, HDR, Matrix Inverse, N Body, HH, Matrix Multiply, and Planck Distribution. For SRAD, we predict the 2nd best design as the densest, while the measured densest design is ranked 2nd. Using the predicted densest for SRAD gets us 2.5% less density than the best. For Gaussian Blur the predicted densest design has 0.0015% less density than the best. It is the 2nd design by a very small difference.

We use our hardware area prediction formula to predict the densest overall design for all the applications in our benchmark set. To evaluate the effectiveness of this formula we will compare its results with measured results for the hardware area of part of the design space. The calculated densest design had an FU Mix 2-2-1-1-1-1, with 4 banks of depth 512 words. The set of designs measured for verification, included the following FU Mixes: 1-1-1-1-1-1, 2-2-1-1-1-1, 3-3-1-1-1-1. It also included combinations of 1 to 64
Figure 7.3: Graphs showing the measured compute density of designs after sorting the data based on the predicted compute density. The prediction is accurate when the densest predicted design matches the densest measured design.

banks (incrementing in powers of 2), and bank depths of 128, 256, 512, 1024 words. In Figure 7.4(a) we show the top 20 designs sorted by predicted compute density. We see that the densest design is predicted correctly in this case. In Figure 7.4(b) we show the percent error distribution graph for the gathered results which ranges from -5% to 6%, and the average percentage of error is 2.4%. We have also evaluated our formula for the prediction of the base densest design (FU Mix 1-1-1-1-1-1). In this case, we collected results for combinations of 1 to 64 banks (incrementing in powers of 2) with bank depths of 128, 256, 512, 1024 words. In Figure 7.5(a) we show the results for CD prediction and the design that we predicted to be the densest is actually the 2nd best when measured. The density of the top 2 designs differs by 0.0238 (1.2%), and the very small difference can explain why our prediction is incorrect. In Figure 7.5(b), the error percentage ranges from -4% to 6%, and the average percentage of error is 2.63%.
We have shown that our hardware area prediction method has acceptable levels of accuracy when predicting design area. Using this method to choose the densest design produces accurate results with the exception of cases where there is a small difference between the density of top designs. In such cases our prediction is no more than 2.5% different from the measured densest design.

**7.4 Summary**

In this chapter, we described a method for reducing the design space search to a relevant subset of the FU mixes. We also presented and evaluated formulas for predicting the area of TILT configurations, which have a low error percentage compared to measured results. This generates accurate results for design area prediction of TILT.
Chapter 8

Conclusions

We presented how we compile and schedule applications for a configurable soft processing engine we call TILT. By using a scheduling approach that prioritizes the Longest Path of the DFG, and mixes instructions across threads we achieved higher throughput for our benchmarks. TILT can be configured by changing the number of Functional Units, number and depth of memory banks, and number of threads, which presented us with a large design space. We explored the design space to find a balance between performance and hardware area, a process that was illustrated by finding computationally dense designs for individual benchmarks or for all of them as a set. TILT designs showed considerable performance improvement over an in-order NIOS processor (22.65x) and closed the gap with custom hardware to 4.41x on average. Our analysis of area showed that the interconnect is a considerable fraction of TILT and requires further improvement. Regarding future architecture improvements, schedule compression—where empty slots are compressed to reduce memory requirements—seems more promising than port sharing. Lastly, we presented methods for reducing the design space search process both on the compiler and hardware side, and evaluated the accuracy of these methods which produced reliable results for hardware area prediction and compiler design space reduction.

8.1 Contributions

This thesis makes the following contributions:

1. We evaluate different scheduling approaches for scheduling applications on TILT and suggest an effective static scheduling approach that on average produces schedules with highest throughput over our benchmarks for a deeply pipelined soft processor and achieves 90% of the theoretical throughput for 64 threads.
2. We perform design space exploration of different hardware configurations, and find designs with a high compute density (i.e. maximize ratio between throughput and area) for our benchmarks. These designs usually have an FU mix that approximates the ratio of operations in the DFG; they contain 2 to 8 banks with 8 to 64 threads, showing that TLP is an important factor in utilizing TILT.

3. We evaluate the potential performance and area of replicating TILT cores compared to custom hardware implementations. We find that the overall densest architecture performs 22.65x faster than an in-order NIOS, and 7.21x slower than the custom hardware implementations when replicated. Replicating the densest TILT per benchmark reduces the gap to an average of 4.41x compared to custom hardware.

4. We present methods of reducing the design space search required to find the computationally dense designs, and formulas that predict hardware area with a low error percentage of 3.5% on average over all benchmarks.

5. We provide an LLVM-based compiler framework that can be used in future research to evaluate architectures similar to TILT.

8.2 Future Work

Ways of extending this work involve both compilation and hardware architecture. The compiler could be extended to support more programming features, such as calls to functions defined in code instead of hardware. We support only one level of nesting for if-else branches, and this could be extended by changing both compiler and hardware support for predication. Another feature would be to support indirection, allowing code to compute addresses such as array indices during runtime. We currently assume that each thread is independent and there is no communication between threads, so in the future there could be support for synchronization between threads. Currently, TILT schedules do not take into consideration data transfers between TILT and external memory. When a data transfer component is implemented it would require support from the scheduler to coordinate computations with data movement. Another possibility is to perform scheduling for a multicore TILT where there would be dependences between the computations across the cores. In terms of architecture, it could be useful to explore ways of reducing crossbar area which was a resource bottleneck especially in large designs. Reducing the connectivity between FUs and memory banks would lead to a more application specific TILT but can potentially improve area and throughput. It would also be useful to find a way of estimating throughput of an application without having to fully schedule it.
Bibliography


