FPGA PROTOTYPING OF CUSTOM GPGPUS

A Thesis
Presented to
The Academic Faculty

by

Nimit Nigania

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
College Of Computing

Georgia Institute of Technology
January 2014

FPGA PROTOTYPING OF CUSTOM GPGPUS

Approved by:

Professor Hyesoon Kim, Advisor
College Of Computing
Georgia Institute of Technology

Professor Sudhakar Yalamanchili
The School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Saibal Mukhopadhyay
The School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: January 2014

To my family,

and friends.

PREFACE

Prototyping new systems on hardware is a time consuming task with limited scope for architectural exploration. The aim of this work was to create a tool chain to prototype general purpose Graphics Processing Units (GPGPUs) on Field Programmable Gate Arrays (FPGAs). This hardware flow combined with the higher level simulation flow using the same source code allowed us to create a whole tool chain to study and build future architectures using new technologies quickly.

ACKNOWLEDGEMENTS

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

FPGA	Field Programmable Gate Arrays
GPGPU	General Purpose Graphics Processing Units
IP	Itellectual Property
SIMD	Single Instruction Multiple Data
MMU	Memory Management Unit
PLL	Phase Locked Loop
RAM	Random Access Memory
ROM	Read Only Memory
MSHR	Miss Status Handling Register

CHAPTER I

INTRODUCTION

1.1 Motivation

We are building a fast FPGA prototyping tool chain to explore various GPGPU based architectures. The tool chain allows us to also use the same code synthesized to verilog to be used for our simulation flow as well with SST. Field Programmable Gate Arrays have been used to prototype hardware designs or to do emulation. Since they are hardware implementation of the design they are very fast to run tests and simulations before actually taping out an ASIC. FPGA are used standalone also to implement highly parallel and configurable application specific designs. Having this emulation platform will allow us to run full feature applications which are generally the problem when running software simulations. For example to run an application on the simulator (MACSIM used in this study, [14]) we first need to trace the application and then run the trace on the simulator. Since the simulator is slow we trace only a small portion (hot loop) of the application. Whereas on the FPGA we can run the whole app without having to generate traces. For an application which has say a billion instructions running on the simulator like MACSIM which runs at generally 50 kips takes around 5hours where as running on the FPGA (200 MHz) might take us only a few seconds. The biggest difference or downside when running applications on an FPGA rather than an ASIC is the speed of the FPGA logic, for example the Altera Stratix III FPGA used for this work can run only till about 500MHz where as an ASIC can run at much higher speed. The modern day FPGA tools make prototyping (implementation and debugging) a painless task as compared to ASIC flow hence this approach was taken.

The aim of this wok was to create a tool chain to prototype general purpose graphics processing unit on FPGAs. This hardware flow combined with the higher level simulation flow using the same source code creates this whole tool chain to study future architectures using new technologies quickly. Most of the work in research has focused on either the simulation flow or the hardware flow. Having this flow for research will help to quickly explore hardware designs and show results on the real prototype. Although there are many research works who do publish results for simulation and hardware but this is generally a very long process.

1.2 Organization

The thesis is organized as follows:

Chapter 2 gives an overview of the overall system which was prototyped. It also discusses about the various tools used and also gives an overview of CHDL which is the programming language used to write the system design code.

Chapter 3 Describes each of the system components in more detail. We mainly cover some details of the ISA used, the design of the core, the cache and the memory controller. Many obvious micro-architecture details have been skipped to keep the descriptions brief. After discussing about the system components we then discuss about how we went on to integrate all these components and create example designs.

Chapter 4 shows the simulation setup along with some of the results obtained. We also show the resource consumption by the design on the FPGA board.

Chapter 5 finally concludes the work and discusses future directions.

CHAPTER II

DESIGN OVERVIEW AND TOOL CHAIN

2.1 System Overview

The aim of this work was to design a custom GP-GPU. Since we wanted the whole system to be customized, a custom ISA ([14]) called 'HARP' was used for this work. The ISA is a MIPS based ISA with support for several customizations which will be discussed in the next section. The key features are predication support, configurable instruction width, no. of general purpose and predicate registers and vector support. As for the design, the main components are the core, the cache and the memory controller- as can be seen in the Figure 1. The Figure 1 is a very simple design which we have used just to demonstrate different components. Each of these components will be discussed separately in the next section. We will also look at the different possible systems we can design in chapter 4. The core which implements the main pipeline and the memory management unit was written in 'CHDL' [14]. The cache was written in verilog but a few different versions also exist. The memory controller IP was generated using Altera's FPGA tools (Quartus II, [14]). Even though we have built a system using a particular configuration, we can use this flow to try many different scenarios. For example if we want to built a system which uses a new kind of memory system (example HMC [14]) all we would have to do is to replace the DDR2 controller which we are using right now with the IP of the new memory controller and everything else will remain the same. We can also add new instructions to the ISA and add support to the core (like support for multiple warps like in GPUs) keeping everything in the uncore part unchanged.

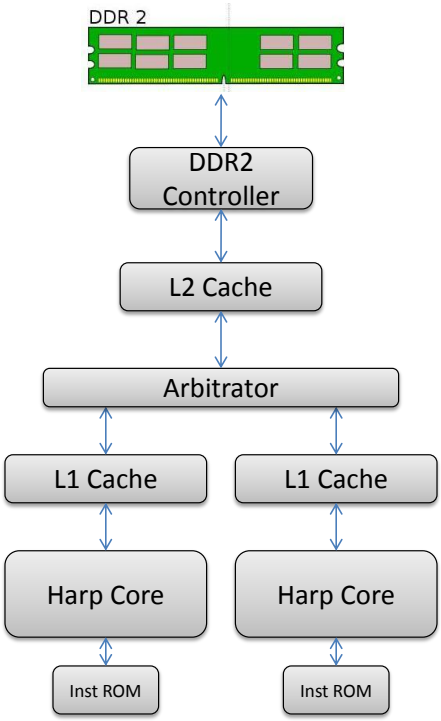

Figure 1: Basic system components

As we can see in Figure 1, our design supports multiple cores and each core can support SIMD hence creating a GPGPU. To enable support for multiple cores we have also designed the arbitrator which sends requests from the private caches of each core to the shared cache. Also similar to the way GPGPUs are designed; our design does not support coherence among the private caches of each of the cores. So the programmer should be aware of this fact and must write code accordingly. For the current work most benchmarks were written in HARP assembly but one of the future tasks as part of this project is to design a CUDA / OpenCL to HARP translator which would the allow us to run more general commercially available applications.

2.2 CHDL

CHDL [14]. CHDL is the custom environment used to write HDL (hardware description language) in C++. It has similarities to system C. We used CHDL so that as

previously mentioned the same code can be used for the FPGA flow as well as the simulation flow. CHDL is implemented as a source to source translator which translates high level C++ to netlist/Verilog HDL. CHDL is basically a set of C++ libraries used to support correct translation of code to its Verilog equivalent. Even though we write in C++ we still have to think to some extent as to how the verilog code will be generated and keep in mind that assignments to wires (represented a vector of Boolean) are continuous and we can assign the wire only once. More details of CHDL along with examples can be found in Appendix A. There are several advantages of taking the CHDL approach, as we write code in C++ it becomes easy to describe complex functions using simple code as compared to Verilog. Also since we already have a library of commonly used functions we can write code quickly, for example we have implementations of multiplexers, decoder, encoders, state machines etc. Obviously the down side of using this approach according to me is that since we can describe lot of complex functionality using this approach hence it might not always generate the most optimal verilog code. Although I expect the Verilog compiler to optimize the code and remove redundancies but I think we would be able to generate better code if we write directly in Verilog. The another biggest downside of using this approach is that it is very hard to debug, as the signal /variable names are lost in the code generated and we have to specify signals using special debug statements if we want to preserve the name to help us in debugging.

Figure 2 shows a simple example of code written using CHDL and compared with its verilog equivalent. The verilog code generated from CHDL is about 106 lines compared to about 10 lines in verilog. As can be seen this is a code for simple 5 bit counter. The verilog code is implied. As for the CHDL version, we represent the counter with as a vector of bool with 5 bits. The 'Reg' statements translates to

```
always@(posedge clk) output <= input;
```

Similarly if we want to implement basic operations we can directly use statements like that shown in Figure 3. This figure shows the implementation of a basic ALU where the final output is that coming out from the multiplexer which selects the correct output based on the 'op_select' which acts as the selector.

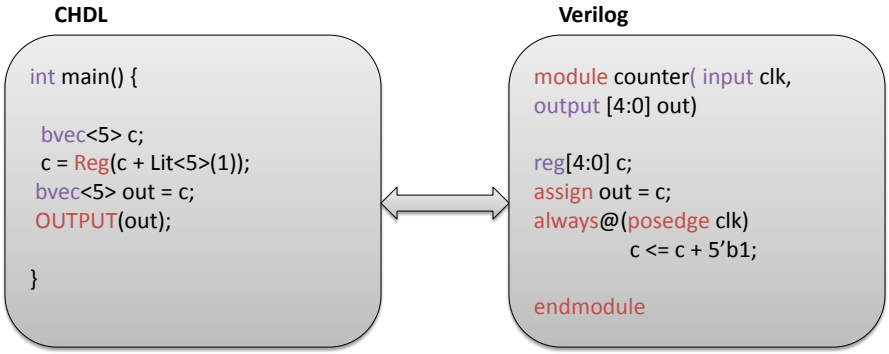

Figure 2: Simple Counter in CHDL and Verilog

```
bvec<N> a      = Input<N>("a");
bvec<N> b      = Input<N>("b");
bvec<ops> op = Input<ops>("op");

bvec<N> out;
vec<64, bvec<N>> mux_in;

mux_in[0x05] = -a;
mux_in[0x06] = ~a;
mux_in[0x07] = a & b;
mux_in[0x08] = a | b;
mux_in[0x09] = a ^ b;
mux_in[0x0a] = a + b;
mux_in[0x0c] = a * b;
mux_in[0x19] = mux_in[0x0f];
mux_in[0x1a] = mux_in[0x10];
mux_in[0x1b] = a;
mux_in[0x25] = b;

out = Wreg(enable, Mux(op, mux_in));
OUTPUT(out);
```

Figure 3: Basic ALU in CHDL

2.3 FPGA board and Development Tools

The FPGA board used for the studies and experiment is a Terasic DE3 board. It uses an Altera FPGA for prototyping. The FPGA used is a Stratix III 3SL150 FPGA with the following properties 142,000 logic elements (LEs); 5,499K total memory Kbits; 384 18x18-bit multipliers blocks; 736 user I/Os. The DE3 board also has support for DDR2 RAM, USB, leds, 7-segement display and connectors to stack multiple boards. Figure 4 shows a picture of the FPGA board used for this work.

We used Altera's FPGA tools for development purposes. The Quartus 11.3 tool was used to synthesize the HDL and program the FPGA device via USB. We used ModelSim for the RTL and gate level simulation experiments. As for other tools used we also used open source tools like 'iVerilog' to compile our verilog code and dump wave signals to a file which we then viewed using 'gtkwave'.

The DDR2 was used along with Altera's DDR2 memory controller IP. More about the DDR2 controller will be discussed in the chapter 3. The Quartus tool with a built in IP generator called 'Megacore wizard' was used to generate the memory controller IPs along with other commonly used IPs like PLLs for managing clocks.

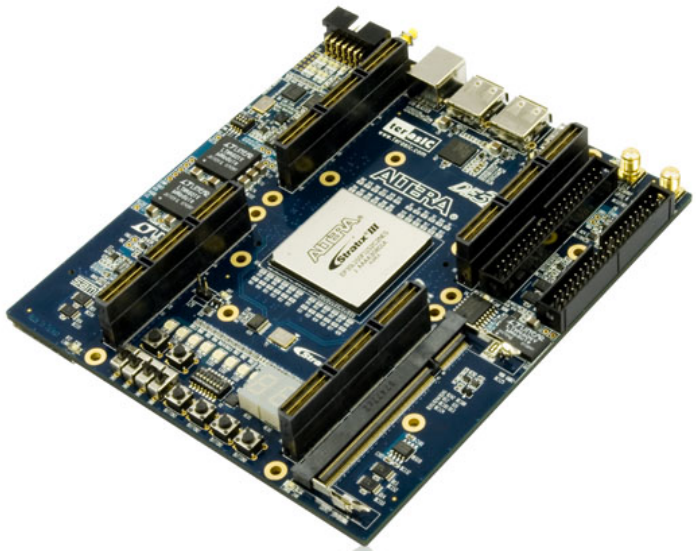

Figure 4: Altera Stratix III Board

CHAPTER III

SYSTEM DESIGN

3.1 System Components

3.1.1 Harp ISA

The ISA used as part of this work was a custom RISC based ISA. The ISA was developed under the name **HARP** which stands for Heterogeneous Architecture Research Project [14]. The main features of the ISA are: Full Predication, SIMD support and customizability. Many features of this ISA are customizable like the vector width, instruction length, number of general purpose and predicate registers etc. The main reason to do this was to add support for new instructions for future architectures and to also allow adapting the ISA quickly to various architectural configurations. There are several types of instructions depending on the number of arguments (register, immediate, predicate registers etc.) but all instructions are encoded in similar fashion. That is most significant bit of each instruction indicates if it's predicate or not, next field specifies the predicate register, the next field stands for the opcode and so on. More details can be found in [14].

The HARP ISA is also supported by its tool chain called Harptool which acts as an assembler, linker and emulator. For this work the benchmarks used were written directly in Harp Assembly which is very much similar to RISC based assembly programs. The benchmarks mainly written and used in our study for testing and performance purposes were as follows:

Single Lane apps:

- Vector Sum
- Sieve of Eratosthenes (finding prime numbers in a range of numbers)

- Bubble sort
- Exponential sum
- Matrix multiplication

SIMD apps:

- Vector Sum using coalesced and un-coalesced accesses
- Matrix multiplication

We varied the input size (array size for array sum, matrix size for matrixmul, input range for sorting) and ran it on our design which was used to not only test the overall design but also to get an idea of the performance. These are very nave applications compared to some of the complex benchmarks which are available out there but these do provided a good credibility to the design if not the performance as of now. One of the main focuses of writing these applications was to stress corner cases. One of the future work which is being done as part of this whole project is the design of a tool which can translate CUDA or OpenCL code to HARP assembly, once that is in place it will allow us to do more thorough testing.
The HARP ISA also in its current version supports only a single simple console I/O to help debug and display the output. Any store instruction at an address 0x8000 causes the harp core to send the required data to the display (7-segment display or LEDs or VGA).

3.1.2 Core

3.1.2.1 Core Pipeline

The base Harp Core design used as part of this work was a design written completely in CHLD by [14]. This section will give an overview of the base design of the core which was modified for use as part of this work. The Harp Core is an in-order issue

and out of order completion pipelined processor. The pipeline stages are mainly: Instruction Fetch, Decode / Register File access / Issue, Execute and WriteBack stage. The function of each of the stages is implied from their names and showed in Figure 5.

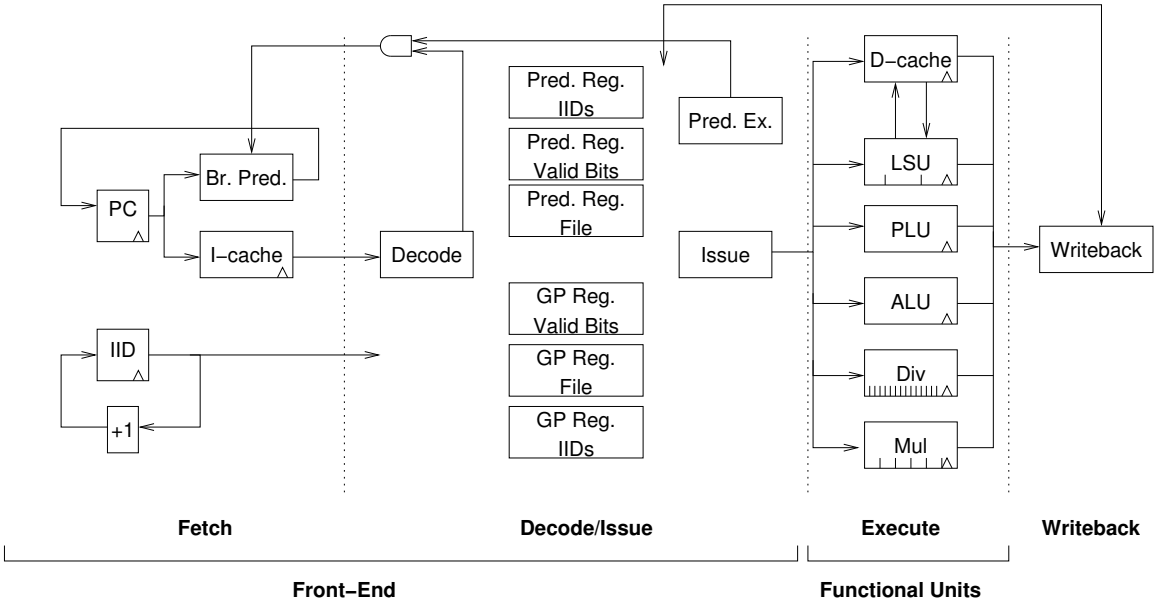

Figure 5: Harp Core Pipeline

The instruction fetch stage reads the instruction stored in the instruction ROM and passes it to the next stage. There is also a GHB (global history based) branch predictor and a BTB (branch target buffer). The PHT (pattern history table) of the GHB and BTB is index by XORing the PC and the branch history. The target branch address is obtained from the BTB and the direction from the PHT (pattern history table). Next, the decode stage decodes the instruction and reads all the registers required for each instruction. It also checks for dependencies and stalls the pipeline is the register it depends on is not updated by the responsible instruction yet. The prediction logic is also implemented here where we convert the instruction to a NOP depending on the value of the predicate register. There is no support for data forwarding in this design right now. To handle dependencies the design assigns a unique Instruction ID (IID) to each instruction and this is used to determine which

instruction is responsible to update the register file to its latest value. The execute stage has all the ALU units to implement the functions offered by the ISA. Additional modules like faster ALU units say for faster pipelined multiply, divide or floating point operations can be written in verilog and integrated with our current design as and when needed. This stage also has the memory management unit to handle memory accesses. The write-back stage writes back the updated data to the register files and updates the state of the registers so the dependent instructions can now progress.

Factors such as SIMD width, number of register, ROM size can be varied. We will discuss more about the SIMD support in the next part and the later sections will discuss more about the memory management unit. We also allow for a version of the core without any cache support but with each lane having a small RAM and a ROM on its own. This variation is just to explore different architecture possibilities.

3.1.2.2 SIMD support

Writing code in CHDL allowed us to easily extend of core design for SIMD support. The whole register file was instantiated as many times as number of lanes, hence converting each register to a vector register. For a given vector instruction the same operation was done for each WORD in the vector register. Once all the vector register are read they are sent to the execute stage. If the instruction involved an immediate then the same immediate was used for all the lanes.

In the execution unit each of the ALU units were instantiated as many times as number of lanes and their corresponding inputs were passed on from the decode stage. Everything else (checking for dependencies logic etc.) remained mostly the same as in the case of single lane (branch outcomes, predication outputs were determined only by the first lane). As of now the core does not support any mask operations so the same operation is done on all the words of the vector register. Adding support for

masking via mask registers and also support for branch divergence will be part of the future work. The only big change in order to support SIMD instructions was done in the memory management unit. To enable this, a complex load store queue shown in Figure 6 was designed as will be discussed in the next section. Also for SIMD support changes had to be done in the cache as the L1 cache now has to send and receive data at cache line granularity rather than word granularity, also the cache had to support masked writes in case of un-coalesced accesses.

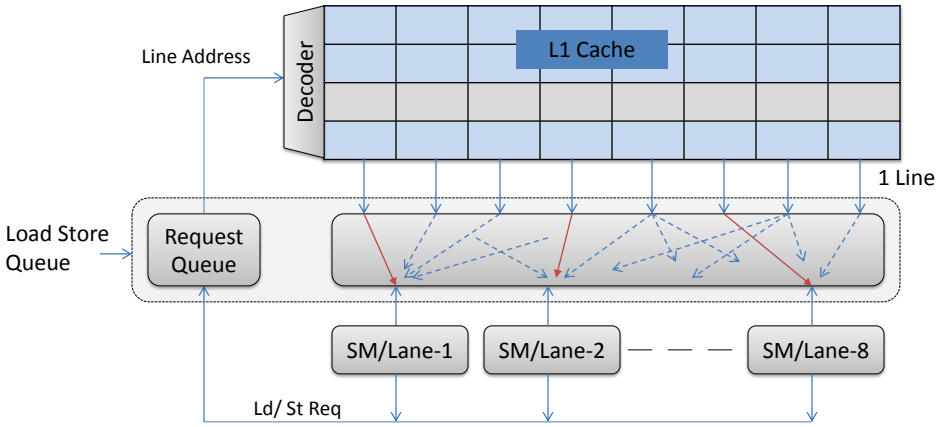

Figure 6: MMU or LoadStore Unit Interface

3.1.2.3 Memory management unit

The whole memory management unit and the cache were among more complex portions of the design. This section talks about the memory management unit / load-store queue / coalescing unit used in the execute stage of the core to send requests to the cache subsystem. The MMU can be divided into 4 major components as can be seen in Figure 7:

- Front End
- Load Queue
- Store Queue

- Cache Interface logic

We will now briefly discuss the functionality of each of the components for a SIMD Harp core. This SIMD support requires the address / data granularity of a cache line between MMU and cache. We also have a design variation without the MMU and blocking memory requests to save on logic resources.

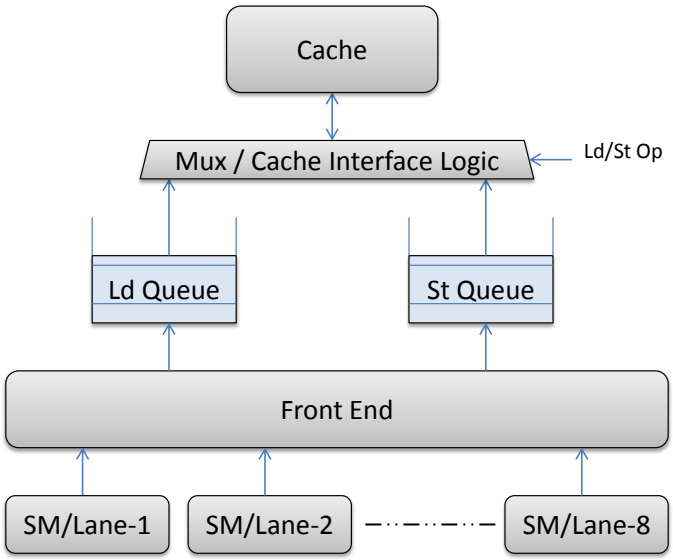

Figure 7: MMU Components

Front End: Receives the request from the core with addresses and data for each lane. It then tries to pack all lane requests to the same cache line together. Hence each entry in the load/store queue corresponds to a cache-line. For an un-coalesced request the front end keeps creating new entries and passing it on to the load or store queue. Along with the cache line address each entry also keeps information about the lanes requesting or writing that cache line along with their corresponding words offsets in the cache line.

Load Queue: All load requests are passed on to this queue. The first request for an instruction is called the leader and all the following requests derived from same

instruction (in case of un-coalesced) are followers. The leader checks if data for all lanes are loaded by waiting till its followers return the required data. The followers writes the data it receives from the memory to the corresponding lane data slot in the leader entry. Once data for all lanes are loaded (we use a loaded bit for each lane) we mark the entry as done and ready to retire. The load requests can be serviced in any order. We also enable LSF (Load Store Forwarding) feature where if an incoming load request sees data for the corresponding cache line in the store queue then it directly gets the data from the store queue entry and is marked as done. The current implementation is conservative in the sense it get data forwarded only when the whole cache line is being written by a store queue entry. If only a partial store is done to line we make the load entry wait until the store entry commits.

Store Queue: The store queue gets store data along with cache line address from the front end. It then checks if there is a pending load entry for the same address, if there is then we make this request wait until the matching load request is serviced to handle WAR hazard. Once the store request is ready to commit then we send the data along with the valid/mask bits for each word to the cache. We give higher preference to requests from load queue over store queue.

Cache Interface logic: This part of the MMU takes each pending entry from load or store queue and sends it to the cache. There is lot of switching logic involved mainly to handle requests from the store queue entry. We allow for cross lane stores wherein each lane can write to any word in a cache line except when they try to write the same word in the same line. We also have a broad cast logic built in which allows multiple lanes in the same request to ask for any/same word in the cache line.

3.1.3 Cache

We use a 2-level non-blocking, write-back, single-cycle latency cache for this design written completely in Verilog. A CHDL implementation was not written mainly because for the FPGA prototyping part we already had a base implementation of the cache design in verilog and for the software simulation part we already have a cache design tightly integrated with our simulation infrastructure. The input and output signals to the cache can be seen in Figure 8. The cache is parameterizable where we can configure the cache line size(32 bytes used for our design), address/data width and also make data input/output granularity to be word or cache line(for SIMD). Since we have designed a single cycle latency cache, we need an extra 2x clock do to tag read and write in the same clock cycle. This clock is fed in the top level system block generated via a PLL. The L1 cache is a direct mapped cache and the L2 is 4-way set associative cache. We will now discuss the main building blocks of the L1 cache, L2 is designed in the similar way except it uses a random cache line replacement policy and extra tag/data RAM.

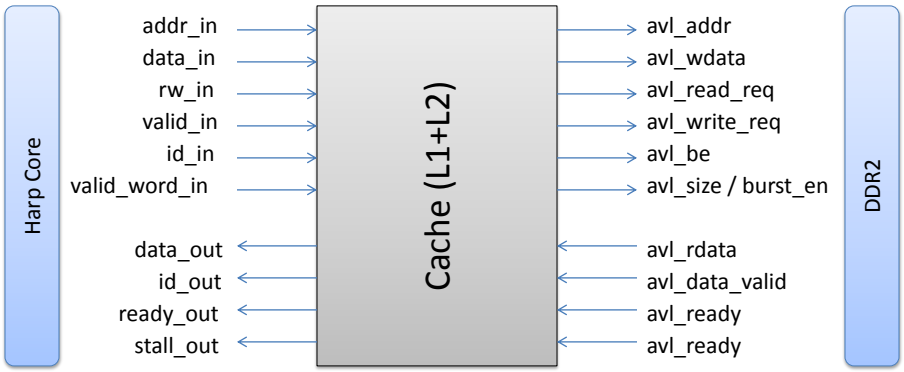

Figure 8: Cache Interface Signals for Core and Memory Sides

- **Data/Tag ram**: These storage elements hold the data and also tags along with dirty and valid bits. The actual data and tags are stored in 8 2-ported

32bit RAMs (8x32 = 256 bit cache line). We need 2 ports as we need to service request coming from the core as well as lower level memory (L2 cache).

- **MSHR** (Miss Status Handling Register): Since we implemented a non blocking cache we need an MSHR. This stores the request which miss in the cache and need to be stored till the cache lines it misses on is serviced by the lower level caches. This stores the data, addresses, read/write and core request id for each read/write operation. When a second request to the same cache line happens we stall the cache and store the new request in a separate data structure. This new request is serviced only after the matching entry in the MSHR is serviced. A future extension to improve the performance will be to allow some kind of a piggy back feature where instead of stalling on the same cache line we keep adding the requests to the same MSHR entry and service them all at once (though this will save us only a few cycles).

- **Non blocking FSM** (finite state machine): This state machine controls the overall operation of the cache. Since we are handling requests from the core and lower level caches, the FSM keeps track of whenever a new request arrives from either side. Depending on the request it issues control signals to the MSHR or the data/tag rams to update their state or generates a stall signal to tell the core to stop sending more requests.

- **Arbiter**: This is used for the multi-core versions of our system where each core has a private L1 cache and they share a L2 cache. The arbiter gets requests from L1 caches of all the cores and sends only one cache line request to the L2. It appends the core-id to the request-id coming from each core so it can return the data to the appropriate core when it is returned from the L2. Various arbitration schemes can be used, but for this work we used a priority encoder. The arbiter like other components is completely configurable allowing it to instantiate as

many L1 caches as possible with only a small change in the verilog code (input signals will be more for more caches).

Basic read and write operations are serviced in the following way.

Read/Write Hit operation: When a new request arrives in the cache we read the tag ram to see if there is a match (1st cycle of the 2x clock), if there is a hit we then send the data from the data ram to the core in the next cycle along with asserting the valid out bit for a load request. For a store request we update our data and tags (2nd cycle of 2x clock) and don't need to reply anything to the core.

Read/Write Miss operation: In case of a cache line miss the FSM allocates a new entry in the MSHR for this request. If the L2 cache is not stalled then it sends the request to the L2 and waits for the data. The L1 cache in the mean time can receive new requests if there is free space left in the MSHR. Each cache miss is treated in a similar way unless the new request is for the same line waiting in the MSHR. In this case we stall the L1 cache and wait until the matching MSHR entry is freed. Once the data from L2 arrives, it directly updates the cache data and tag rams. Then the FSM gets the MSRH entry responsible for the miss and issues it again, making it a cache hit this time. Once issued, the MSHR entry is freed and data along with valid output is sent to the core.

3.1.4 Memory Controller

The memory controller supported by the DE3 FPGA board used as part of this work is a DDR2 memory controller with operating frequency ranging from 125 MHz - 533 MHz. We used a 1GB DDR2 RAM for this work running at the 125MHz due to limitations set by other components of the system. The DDR2 is connected on the board via 200 pins for clock and control signals coming from the FPGA. Figure 9 shows the top view of the memory controller block, more details can be found here

[14]. The user side signal uses Altera's Avlon interface and this is the suffix given to these signals. The command generator receives the signals from the user side and passes it to the timing bank pool. The timing bank pool then checks for signal timings and if valid data is present in the data buffers. Then it passes the request to the arbiter which finally passes the command forward. The rank timer maintains rank specific information to maintain correct functionality.

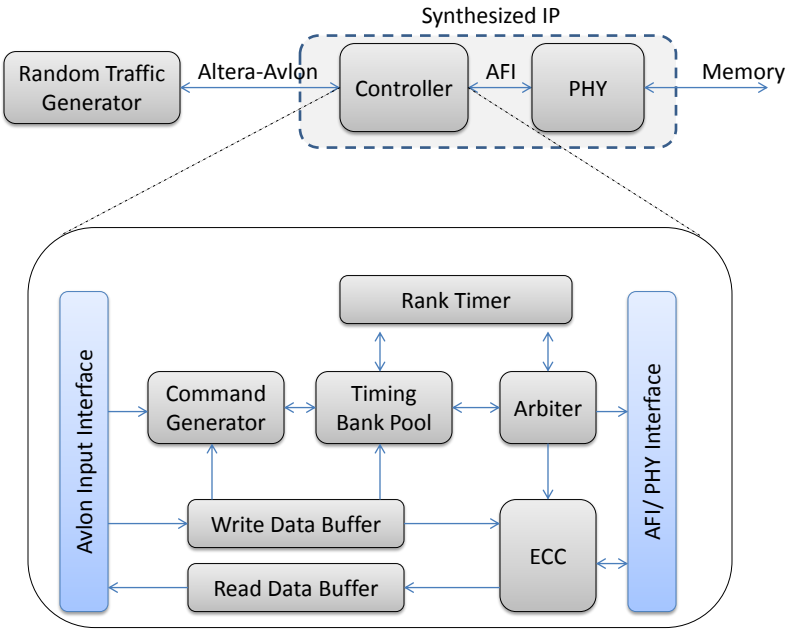

Figure 9: Test Setup for Memory Controller IP and Internal Block Diagram

The DDR2 controller IP was generated using the 'Altera's MegaWizard ' which automatically generates the IP along with some example files for simulation purposes. Although it might seem straight forward but creating this IP with the correct parameters set was very critical. Many times the RTL simulation (using a system verilog model for a DDR2 memory) of generated controller IP was found to be working but it failed on the board because one of the timing parameters was set incorrectly. Even using timing presets present in the Altera tool for the actual DDR2 DIMM used in the board were not correct and the correct parameters were later obtained from an

example DDR2 IP project from the FPGA board vendor (Terasic). Also since the number of pins which need to be connected to the DDR2 were significant they all had to be set very carefully again using the parameter obtained from board vendor which was also a confusing part as Altera by default recommended another configuration. The incorrect pin and parameter configurations were the main reasons behind failures seen initially. Many DDR IP parameters like the frequency and the row open/close policy were configurable via Altera's IP generator. Though we can customize the IP RTL itself but we can't reuse all of Altera's propriety IP. Different scheduling policies like FRFCFS etc. can be tried and can be one research direction going forward for different types of system architectures.

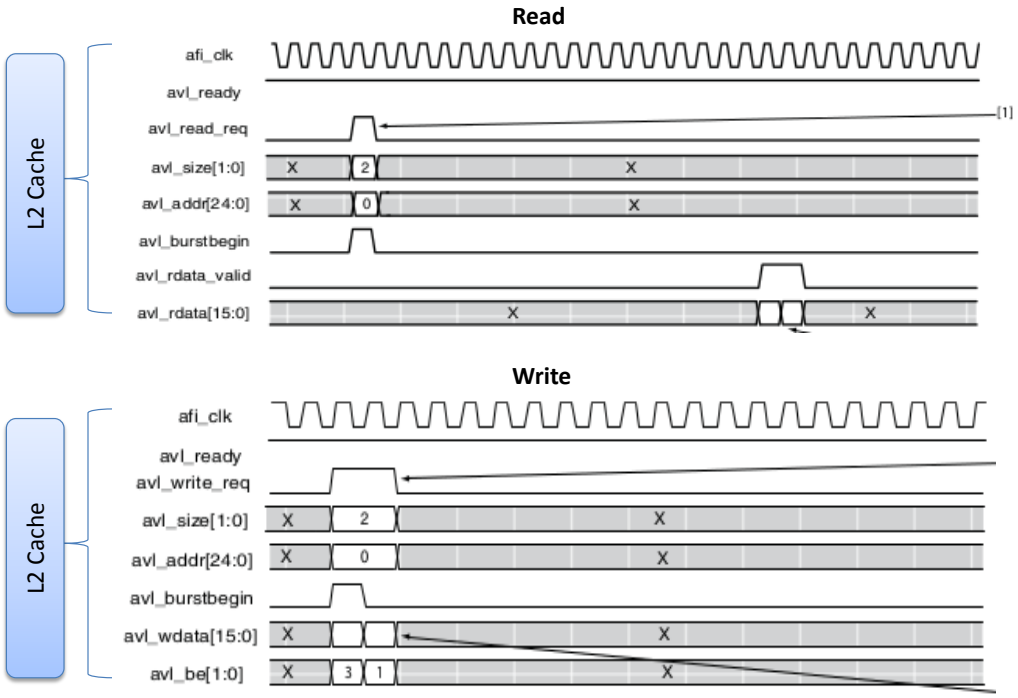

Figure 10: DDR cache side signals for read / write Operation

Before using the controller with the rest of the system it had to be tested using a random traffic generator (generated along with IP). It does a thorough test to make

sure the DDR2 runs for the parameters set on the board. We first tested the controller in RTL using a DDR2 memory model and then on the FPGA board. Once DDR2 was tested then we started integrating it with rest of the system. Since the cache interface is slight different from the user interface signals of the memory controller a memory controller wrapper block was written to interface them. The main difference was that the cache identifies each request via a unique ID where as the DDR2 just returns the data along with valid bit as can be seen in the DDR2 wave diagrams in Figure 10. This wrapper is mainly a FIFO based structure which stores the request along with the id of incoming request coming from the cache and sends the data returned from main memory back to the cache along with the original request id.

3.2 Putting it together: A Multicore and multilane GP-GPU system

Figure 11 shows the final system which was designed and tested on the FPGA board. We saw in previous sections in this chapter as to how each of the components namely the SIMD core, MMU, cache and the memory controller was designed. Now the aim of the work was to integrate each of these components to make the overall system. This although might sound simple to start with but was a fairly time consuming task as it was thorugh this process that many hidden bugs were discovered. Firstly, we had to test each module before integrating them; this was done by writing separate test cases for each of the verilog blocks. This led us know if we were getting the basic functionality out of the desired block. As many blocks were complex and had many inputs/outputs it was hard to test them. Then we started integrating the whole system and debug it in a step by step way. We first started by testing a single lane Harp core system and made sure this was working on the board and meeting all the timing constrains. For a system of this magnitude it is generally hard to test all the corner cases and even now there were some hidden bugs in the design. Since we tested our system by writing applications rather than creating stimuli or verilog testbench,

our tests were to a good extent thorough. First step was to do RTL simulation of

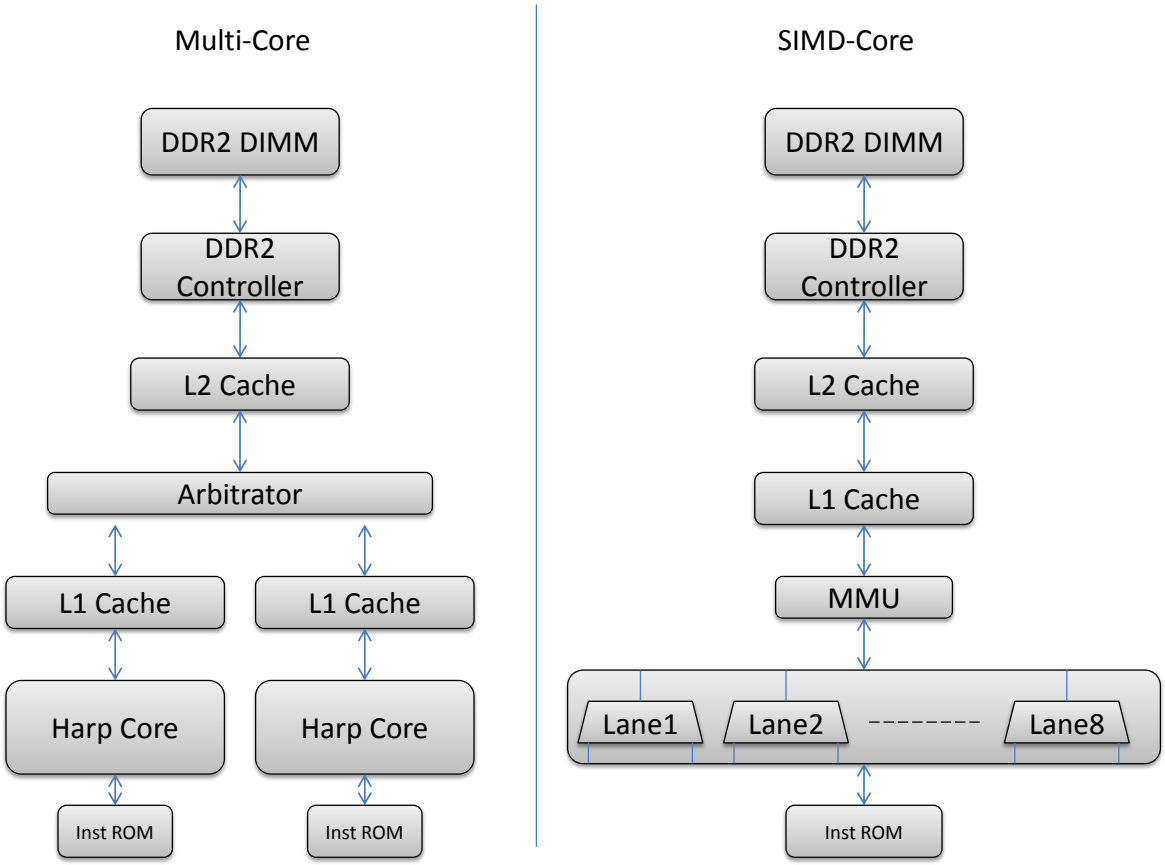

Figure 11: Possible System Designs

the whole system and make sure it was passing. Many functional bugs can be easily found and fixed at this stage. Next, we synthesize and do gate level simulation, this is hard to debug as signal names are changed or synthesized away. The best way to debug issues at this stage was to fix the warnings which were thrown by Altera's Quartus tool and this for me fixed many bugs in the design. Although many issues including the errors due to DDR2 timing issues were hard to fix and took a lot of time. After this stage, we test the system on the board by downloading the FPGA programming binary on the board. If it does not run even now then the reasons which I came across were mainly due to issues in the reset logic or the clock or the DDR

timings parameters. One good practice is to use PLLs to generate cleaner clocks with less jitter. Most of the problems in my design were fixed using the above process, in case if one is still not able to isolate issues then the next step is to use hardware debug IPs which can monitor signals on board and display it on the screen. This was also used to debug some issues in the DDR as part of this design. Once a single lane version of the whole system was working on the board we then tested a multiple lane version of the core using different tests to stress most of the commonly used cases, for example we wrote test cases for doing lot of un-coalesced memory requests and also coalesced requests to make sure if the complex memory management unit was working properly or not. Once the single core (one lane or SIMD) was tested on the board then it was easy to create multiple core versions by instantiating more instances of the core with each core running its own separate application. Figure 11 shows one such system, as we can see we have 2 cores each having its own private L1 cache and sharing a L2 cache. As mentioned previously we don't have support for coherence as of now. The extra component needed to make this multi-core system was the L1 cache arbitrator which schedules requests from different cores to the L2. For this work a priority arbiter was used but other possibilities can also be explored. To test we tried to run different combination of applications on each of the core accessing different data in the main memory. The results from the simulation experiments are discussed in the next section.

CHAPTER IV

EXPERIMENT RESULTS

4.1 Board and Test Environment

The FPGA board used for the studies and experiment is a Terasic DE3 board. It uses an Altera FPGA for prototyping. The FPGA used is a Stratix III 3SL150 FPGA with the following properties 142,000 logic elements (LEs); 5,499K total memory Kbits; 384 18x18-bit multipliers blocks; 736 user I/Os. The DE3 board also has support for DDR2 RAM, USB, leds, 7-segement display and connectors to stack multiple boards. Figure 4 shows a picture of the FPGA board used for this work. We use a 1GB DDR2 DIMM. The benchmarks as mentioned earlier were all written in Harp assembly and results of their run on the board are shown below.

4.2 Benchmarks Ran and Results

The basic parameters for the core and cache are set as shown in Table 1

Core Configuration	
Core Operating Freq:	62.5 MHz
DDR2 Operating Freq:	125 MHz
Instruction Width:	32 bits
L1 / L2 cache:	16KB / 128KB
General Registers:	16 32-bit Regs
Predicate Registers:	16
SIMD Lanes:	8
Instruction ROM:	1024KB

Table 1: Core Configuration.

After creating the basic building blocks of our system and we tried to run performance and functional tests on a few different basic variation of the design. Below we discuss 3 versions of the Harp Core and its performance for simple micro-benchmarks.

Single 1-Lane Harp Core:

First we ran the benchmarks on a simple 1-lane Harp Core. To make the design simple we removed the complex load/store queue so the core now sends blocking requests to the memory subsystem. Table 2 shows the performance of and Table 3 shows the logic utilization of this system.

Benchmark	IPC	Description
Vector Sum:	N/A (N/A)	Multiply 2 matrices of size 16x16
Sieve of Eratosthenes:	N/A (N/A)	finding prime numbers between 1 to 100
Bubble sort:	N/A (N/A)	Sort Numbers 1-10 using bubble sort
Matrix multiplication:	N/A (N/A)	Multiply two 8x8 matrices

Table 2: Single Core 1-Lane Performance

Logic Consumption:

Logic Utilization	
FPGA Utilization:	4% (3,661 / 113,600 ALUTs)
Combinational ALUTs:	13% (14,446 / 113,600 ALUTs)
Total block memory bits:	27% (1,505,792 / 5,630,976)

Table 3: Logic Utilization.

Dual 1-Lane Harp Core

Next we instantiate two of the 1-lane Harp cores above to form a multi-core system. Table 4 shows the performance of and Table 5 shows the logic utilization of this system.

Benchmark	IPC	Description
Vector Sum and Sieve:	N/A (N/A)	Run sum on core1 and sieve on core2
Bubble and Sieve:	N/A (N/A)	Run sum on core1 and sieve on core2
Matrix multiplication and Bubble:	N/A (N/A)	Multiply two 8x8 matrices

Table 4: Dual Core 1-Lane Performance

Single SIMD 8-Lane Harp Core

We also design and run an 8 Lane SIMD Core on the FPGA board. We used simple applications initially to test the complex coalescing unit. Once that was done we tried

Logic Utilization	
FPGA Utilization:	4% (3,661 / 113,600 ALUTs)
Combinational ALUTs:	13% (14,446 / 113,600 ALUTs)
Total block memory bits:	27% (1,505,792 / 5,630,976)

Table 5: Logic Utilization.

to run a compute internsive matrix multiplication code on the board. Comparing the performance numbers we can clearly see the advantage of using SIMD. This comes at the cost of much higher logic utilization due to the coalescing unit and the duplication of ALU blocks and register files for the SIMD core. Table 4 shows the performance of and Table 7 shows the logic utilization of this system.

Benchmark	IPC	Description
Matrix Multiplication:	N/A (N/A)	Multiply 2 matrices of size 16x16
Coalesced Vector Sum:	N/A (N/A)	Sum 100 numbers doing coalsced access
Un-Coalesced Vector Sum:	N/A (N/A)	Sum 100 numbers doing un-coalsced access

Table 6: Single Core SIMD 8-Lane Performance

Logic Utilization	
FPGA Utilization:	4% (3,661 / 113,600 ALUTs)
Combinational ALUTs:	13% (14,446 / 113,600 ALUTs)
Total block memory bits:	27% (1,505,792 / 5,630,976)

Table 7: Logic Utilization.

CHAPTER V

FUTURE WORK AND CONCLUSION

5.1 Related Work

5.2 Future Work

Since this work is part of a multiyear and multiperson project many extensions are planned for this work. As for changes in the core part of the design changes like adding support to handle multiple warps, branch divergence using mask registers will be one important feature which will make this design truly comparable to modern day GPGPUs. Also a feature like data forwarding can be added if we find that making the core more CPU like will be beneficial for certain kind of systems. The applications written right now were all in Harp assembly but to allow us to run of the shelf CUDA or OpenCL applications a software translator tool is being worked upon wich can generate Harp assembly from these binaries.

The next major part of the future work is to explore future systems using new memory technologies like the HMC. Building these systems would require only isolated changes to the existing design. For example to use a FPGA board with HMC we would only have to generate a new Memory Interface IP for the HMC (as the controller is integrated in the logic layer of the HMC) and use this with the rest of the system.

5.3 Conclusion

To conclude we showcased a tool chain and possible designs to allow quick prototyping of GPGPU designs on real hardware. Integrating the FPGA protyping flow with

software simulation infrastructure will allow us to explore future architectures at different levels of granulatiry. By creating a parameterized design we can change many aspects of our design to affect performance. The flexibility offered by CHDL was also shown which allowed us to easily add more features to our system if needed like SIMD support. We were able to see benefits of the SIMD version of the core using a coalescing unit over the single lane version. This work also discussed few different systems which were emulated on hardware with only minor changes to the base design flow.

REFERENCES

[1] Désarménien, J., "How to run TEX in french," Tech. Rep. SATN-CS-1013, Computer Science Department, Stanford University, Stanford, California, Aug. 1984.

[2] Fuchs, D., "The format of TEX's DVI files version 1," *TUGboat*, vol. 2, pp. 12–16, July 1981.

[3] Fuchs, D., "Device independent file format," *TUGboat*, vol. 3, pp. 14–19, Oct. 1982.

[4] Furuta, R. K. and MacKay, P. A., "Two TEX implementations for the IBM PC," *Dr. Dobb's Journal*, vol. 10, pp. 80–91, Sept. 1985.

[5] Knuth, D. E., "The WEB system for structured documentation, version 2.3," Tech. Rep. STAN-CS-83-980, Computer Science Department, Stanford University, Stanford, California, Sept. 1983.

[6] Knuth, D. E., *The TEX Book*. Reading, Massachusetts: Addison-Wesley, 1984. Reprinted as Vol. A of *Computers & Typesetting*, 1986.

[7] Knuth, D. E., "Literate programming," *The Computer Journal*, vol. 27, pp. 97–111, May 1984.

[8] Knuth, D. E., "A torture test for TEX, version 1.3," Tech. Rep. STAN-CS-84-1027, Computer Science Department, Stanford University, Stanford, California, Nov. 1984.

[9] Knuth, D. E., *TEX: The Program*, vol. B of *Computers & Typesetting*. Reading, Massachusetts: Addison-Wesley, 1986.

[10] Lamport, L., *LATEX: A Document Preparation System. User's Guide and Reference Manual*. Reading, Massachusetts: Addison-Wesley, 1986.

[11] Patashnik, O., *BibTEXing*. Computer Science Department, Stanford University, Stanford, California, Jan. 1988. Available in the BibTEX release.

[12] Patashnik, O., *Designing BibTEX Styles*. Computer Science Department, Stanford University, Jan. 1988.

[13] Samuel, A. L., "First grade TEX: A beginner's TEX manual," Tech. Rep. SATN-CS-83-985, Computer Science Department, Stanford University, Stanford, California, Nov. 1983.

[14] Schreier, R. and Temes, G. C., *Understanding Delta-Sigma Data Converters*. John Wiley and Sons Inc, 2005.

[15] Spivak, M. D., *The Joy of TEX*. American Mathematical Society, 1985.

INDEX

FPGA Prototyping Of Custom GPGPUs

Nimit Nigania

31 Pages

Directed by Professor Hyesoon Kim

This is the abstract that must be turned in as hard copy to the thesis office to meet the UMI requirements. It should *not* be included when submitting your ETD. Comment out the abstract environment before submitting. It is recommended that you simply copy and paste the text you put in the summary environment into this environment. The title, your name, the page count, and your advisor's name will all be generated automatically.