ABSTRACT

KOTHIYA, MAYANK VINODBHAI. Understanding the ISA impact on GPU Architecture. (Under the direction of Huiyang Zhou.)

The wide spread acceptance of GPU for parallel computation has created the demand for general purpose capabilities in GPU. In response, Industry is coming up rapidly with better architecture to support general purpose processing on GPUs. NVIDIA has come up with Tesla, Fermi and Kepler architecture. General Purpose Graphics Processing Units (GPGPU) are widely being used in many different application domains such as neural networks, matrix computations, graph algorithms etc. [1]. This work studies the evolution of ISA from Tesla architecture to Fermi architecture.

General Purpose GPU Simulator (GPGPUSim) currently doesn't support Native ISA for Fermi architecture [2], however it supports native ISA for previous generation of GPU architecture (Tesla). Our contribution is to extend GPGPUSim to use Native ISA for performance/functional simulation. Also native control flow information can be very useful to generate reference model for hardware implementation. GPGPUSim also doesn't use control flow information present in Native ISA. It extracts that information from PTX assembly instruction analysis. Our simulator extension uses control flow information present in Native ISA.

The methodology for extending GPGPUSIM involves studying the simulator and modifying it to support the target set of benchmarks. The methodology to understand program control flow involved a review of patents and it provided insight for required hardware support to execute Native ISA. In order to validate the understanding gained from these literature study, GPGPUSim is extended and verified on selected benchmarks for Fermi architecture. Various instruction and its semantics were found out by correlation of CUDA (Compute Unified Device Architecture-programming language for NVIDIA's GPGPU Architecture) Code, PTX code and Native ISA.

The dynamic instruction count difference between Tesla ISA and Fermi is around 26% (observed on selected benchmarks). The dynamic instruction count difference reveals interesting insight of the difference between two ISAs. This difference is mainly caused by instruction fusion (For example, combining one multiply and one add into single instruction), predication and uniform branching support, 32 bit multiplication and better addressing modes in Fermi architecture which doesn't require significant changes in hardware architecture.

Understanding the ISA impact on GPU Architecture

by
Mayank Vinodbhai Kothiya

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the degree of
Master of Science

Computer Engineering

Raleigh, North Carolina

2014

APPROVED BY:

_____
Dr. Huiyang Zhou
Committee Chair

_____ _____
Dr. James Tuck Dr. Eric Rotenberg

DEDICATION

To my family, friends and teachers.

BIOGRAPHY

Mayank Kothiya was brought up in Surat, India. He has obtained his Bachelors in Electronics and Communication Engineering from Institute of Technology, Nirma University in 2010. He worked for NVIDIA Bangalore for two years. Thereafter he joined Computer Engineering MS program at NC State University in fall 2012. After graduating from NC State, he will be joining Oracle, Santa Clara as hardware engineer.

ACKNOWLEDGEMENTS

I would like to thank my parents for helping and motivating me to pursue higher studies without which none of this would have been possible. I would also like to thank them inspiring me through their own journey of life and teaching me the right values as I grew up and making me what I am today. I would also like to thank my two sisters who have always been my support system.

I would like to thank my advisor Dr Huiyang Zhou for all the encouragement, motivation and support without which this work would not have been possible. I am very fortunate that I got a chance to work with him and learn a lot more about GPUs as well as learning some valuable lessons in life which I will carry with me and follow throughout my life.

I would like to thank my committee members Dr. Eric Rotenberg and Dr. James Tuck for providing critical feedback on my work and helping to solve some bugs.

I would like to thank Chao Li, Hongwen Dai and Ping Xiang for insightful discussion while working on this thesis. I would also like to thank all my friends for their support and encouragement throughout this journey. I would like to specially thank Shreyas Bhardwaj for proof reading my thesis and helping me to edit it. I would also like to thank Abhijeet Deshpande, Kirti Bhanushali, Sandesh Saokar, Vikrant Srivastava and Ajay Ramesh for moral support during this journey.

TABLE OF CONTENTS

LIST OF TABLES ..... **vii**

LIST OF FIGURES ..... **viii**

1 INTRODUCTION AND MOTIVATION **1**

1.1 Motivation on using GPUs: 1

1.2 GPU General Architecture: 2

1.3 SM (or SIMT Core) Architecture and Pipeline: 4

1.4 Related Work: 6

2 BACKGROUND ON GPGPUSIM **7**

2.1 GPGPUSim Software Architecture: 8

2.2 GPGPUSim extension to support Fermi architecture at Native ISA: 10

3 CONTROL INSTRUCTIONS MANAGEMENT **13**

3.1 Predication: 13

3.2 SIMT Stack based implementation: 14

3.2.1 Token Structure and Types: 15

3.3 IF THEN ELSE and WHILE/FOR Management: 18

3.3.1 “IF THEN-ELSE” branch management: 18

3.3.1.1 Predication based Implementation: 18

3.3.1.2 SIMT Stack based Implementation: 19

3.3.2 While/For Loop branch management in Fermi Native ISA: 23

3.4 Break/Pre-break Management: 27

3.5 Control Flow Management Algorithm: 33

4 BENCHMARKS AND INSTRUCTION SUPPORT **37**

4.1 Instruction specific SASS to PTXPlus Conversion: 37
5 RESULTS 44
5.1 Analysis and Results: 44
5.1.1 MM (matrix multiplication): 45
5.1.2 DWT benchmark: 48
5.1.3 NN Benchmark- Kernel1: 50
5.1.4 Pathfinder Benchmark- Kernel1: 52
5.1.5 B+tree benchmark: 55
6 CONCLUSION 58
REFERENCES 59

LIST OF TABLES

Table 1 Token Structure 15
Table 2 Various Tokens [5] 17
Table 3 Native Assembly Code 21
Table 4 SIMT Stack during Program execution 22
Table 5 SIMT Stack during Program execution 26
Table 6 SIMT Stack Management for Figure 10/11 31
Table 7 Total Instruction Count (MM) 45
Table 8 Code analysis (MM) 47
Table 9 Total Instruction Count (DWT) 48
Table 10 Code Analysis (DWT) 49
Table 11 Total Instruction Count (NN) 51
Table 12 Code Analysis 51
Table 13 Total instruction count 52
Table 14 Code analysis 1 53
Table 15 Code analysis 2 54
Table 16 Total Instruction count 55
Table 17 Code analysis 1 56
Table 18 Code analysis 2 56
Table 19 % Savings in instruction count for different benchmarks 57

LIST OF FIGURES

Figure 1: GPU General Block Diagram [2] 3
Figure 2 SIMT Core Architecture and Pipeline [2] 5
Figure 3 PTX and PTXPlus Compilation flow [2] 8
Figure 4 Software Architecture of GPGPUSim [2] 10
Figure 5 Predication for IF-THEN ELSE Type of branch 18
Figure 6 “C language Program” 19
Figure 7 Fermi Architecture Native ISA Translation for program in Figure 6 21
Figure 8 C language Program with “for” loop 24
Figure 9 Native Assembly Code 25
Figure 10 C/CUDA Kernel 27
Figure 11 Native ISA code for Figure 10 28
Figure 12 Flow chart of control instructions 36
Figure 13 Dynamic Instruction Count 44
Figure 14 Dynamic Instruction Count for Tesla vs Fermi (MM) 45
Figure 15 Instruction Percentage for Tesla vs Fermi (MM) 46
Figure 16 Dynamic Instruction Count for Tesla vs Fermi (DWT) 48
Figure 17 Instruction Percentage for Tesla vs Fermi (DWT) 49
Figure 18 Dynamic Instruction Count for Tesla vs Fermi (NN) 50
Figure 19 Instruction Percentage for Tesla vs Fermi (NN) 51
Figure 20 Dynamic Instruction Count for Tesla vs Fermi (pathfinder) 52
Figure 21 Instruction Percentage for Tesla vs Fermi (pathfinder) 53
Figure 22 Dynamic Instruction Count for Tesla vs Fermi (b+tree) 55
Figure 23 Instruction Percentage for Tesla vs Fermi (b+tree) 56

1 Introduction and Motivation

1.1 Motivation on using GPUs:

GPUs are the processors that are mainly designed and used to render graphics applications e.g. animations, video and game rendering. Due to the nature of their applications, GPUs are very useful for the programs that involve parallel computations. NVIDIA is leading the effort of building GPUs that can be used for general purpose computation apart from graphics rendering. GPGPUs are very useful for applications that involve parallel computation and GPGPUs are finding their usefulness in many different application domains such as neural network computations, graph search algorithms, sorting algorithms, Monte Carlo simulations, ray tracing, weather predications and genome computations to name a few [1]. The main advantage of GPU comes from performance and power efficiency to perform parallel computation. Performance comes from running multiple threads in parallel whereas energy efficiency comes from simplifying hardware design. CPU executes the program serially on each data whereas GPU performs computation on multiple data in parallel. CPU fetches and decodes instruction for each data separately and performs the computation on each data. Considerable energy is wasted in a CPU due to repetition of unnecessary computation. Furthermore CPU pipelines are complex Out of Order to achieve higher instructions per cycle (IPC). Due to above constraints, parallel computation on CPU is power hungry and performance inefficient. GPU achieves power and energy efficiency by computing several threads at the same time. GPU executes several threads in a group. This thread group is alternatively known as warp. Warp can have 32(NVIDIA GPUs)/64(AMD GPUs) threads executing in lock step. Each warp has a single program counter and all threads within warp

shares this program counter. With one instruction fetch and decode, 32/64 data computation can be done in parallel depending on number of available ALUs. GPU saves fetch and decode energy via combining threads in a warp and it achieves faster performance by doing computation for multiple threads on different ALUs at the same time.

This chapter presents introduction to GPU architecture. The next chapter presents an overview of GPGPUSim. These two chapter forms baseline to understand the work accomplished in this thesis. Chapter 3 presents algorithm and hardware support to manage various control instructions in GPGPU/SIMT architecture. Chapter 3 is the backbone of the work accomplished in this thesis. Chapter 4 provides introduction to benchmarks used in this study and the implementation details of various instruction. Finally, chapter 5 presents results and analysis.

1.2 GPU General Architecture:

The program to be executed on GPU is known as “kernel” in Compute Unified Device Architecture (CUDA) community. GPU kernel is divided in thread blocks before it can be executed on the GPU. CUDA programming model explains further details regarding GPU programming which are out of scope for this document. Figure 1 shows the architecture of GPU. The different units of GPU are Thread Block Scheduler, SIMT Core and Interconnection Network. Thread Block Scheduler is responsible for dispatching threads to SIMT Cores at thread block granularity. Single Instruction Multiple Thread (SIMT) Core is the main execution unit or processor. It is also known as streaming multiprocessors (SM). SIMT Core is responsible for executing all the computation of the program. It is connected with interconnection network which can access on chip L2 cache or off chip global memory in case of LD/ST miss in L1 cache. SIMT Core also informs Thread Block Scheduler regarding the status of current thread block being executed. Generally in GPU, there will be one Thread Block Scheduler and multiple SIMT Cores to execute as many threads in parallel as possible. Thread

Block Scheduler assigns threads to SIMT Core in round robin manner or any other possible mechanism at thread block granularity.

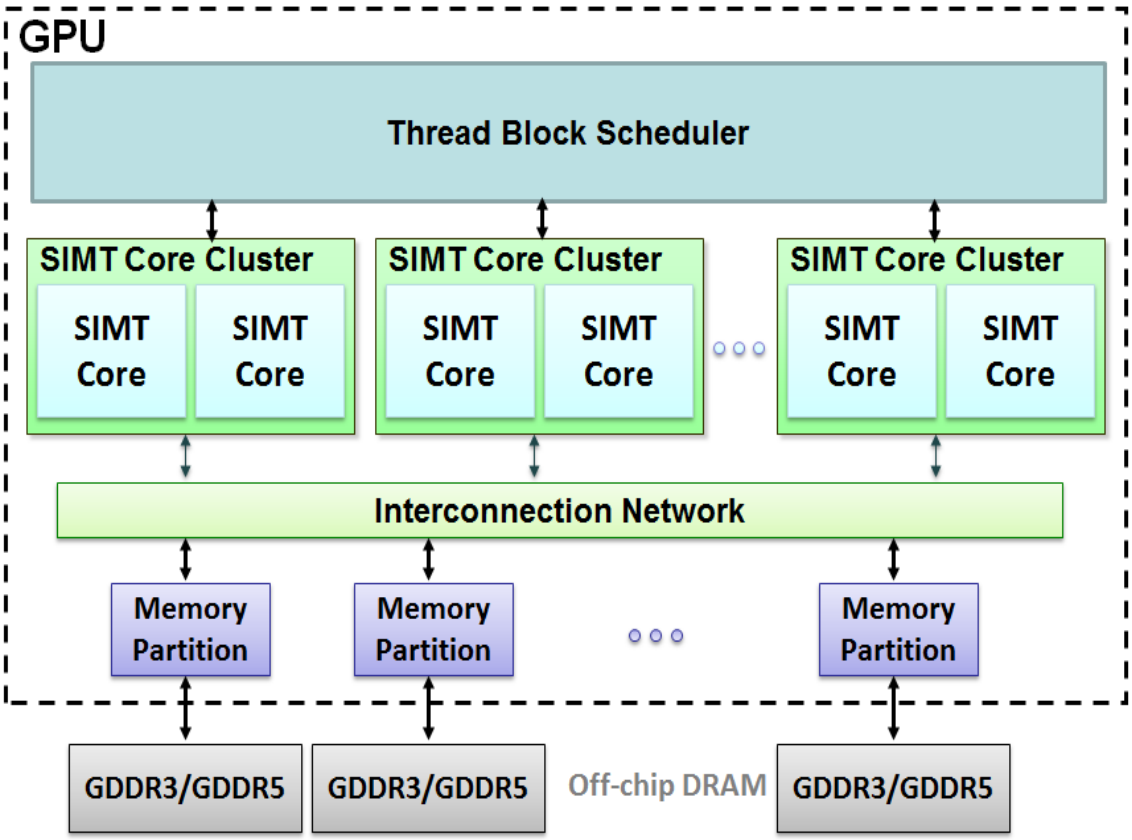

Figure 1: GPU General Block Diagram [2]

1.3 SM (or SIMT Core) Architecture and Pipeline:

Figure 2 shows the block diagram of GPU microarchitecture for SIMT Core. Fetch unit, Decode Unit, Scoreboard, SIMT Stack, Issue Unit, Register Read, ALU (Arithmetic and Logic Unit), Memory, SFU (Special function unit), L1 cache, constant cache, texture cache, shared memory are the different blocks of SIMT Core (not all shown in the Figure 2). As mentioned previously, Thread Block Scheduler dispatches "thread block" to SIMT Core and threads are executed on SIMT Core at warp granularity. SIMT Core divides thread block to create multiple warp and all these warp can execute in parallel to each other. For example if the number of threads in thread block are 256 then SIMT Core will create 8 warps considering threads within a single warp is 32. Each thread in thread block/warp is identified via unique thread identifier. Once the threads are allocated and warp creation is complete, SIMT Core will start the execution.

- Fetch unit: This unit is responsible for sending the Instruction request to instruction cache. Fetch unit interacts with SIMT Stack to find out the program counter (PC) of the next instruction.
- Instruction cache (IC): If the request from the fetch unit is hit in IC then it will send instruction to decode unit immediately otherwise it will keep the request in Miss Status Holding Register (MSHR) and forward the data to decode unit whenever it is available.
- Decode unit: This unit will decode the instruction and forward it to I-Buffer. It forwards the source and destination register information to scoreboard as well as forwards instruction type, target address (for branches) and other control flow relevant information to SIMT Stack.
- SIMT Stack: SIMT Stack is responsible for managing the control flow related instructions. It is also responsible for providing "Next PC". The more details of SIMT Stack will be explained in later chapters.

- Scoreboard: It is used to support ILP (Instruction level parallelism). Depending on the complexity of the scoreboard, multiple independent instruction can be executed in parallel. Scoreboard keeps track of pending register writes in order to stall the younger dependent instructions.
- I-Buffer: It holds the information of the decoded instructions of all warps. Warp scheduling policy decides the order of instruction issue to execution and memory stages. There are various policy that can be used to decide the order of instruction issue and most basic policy is round robin scheduling.
- Register File: To support multiple threads within a warp, register file is vectored with two read ports and one write port (this may vary). Operand collector architecture for register file is a different mechanism to manage instruction operands.

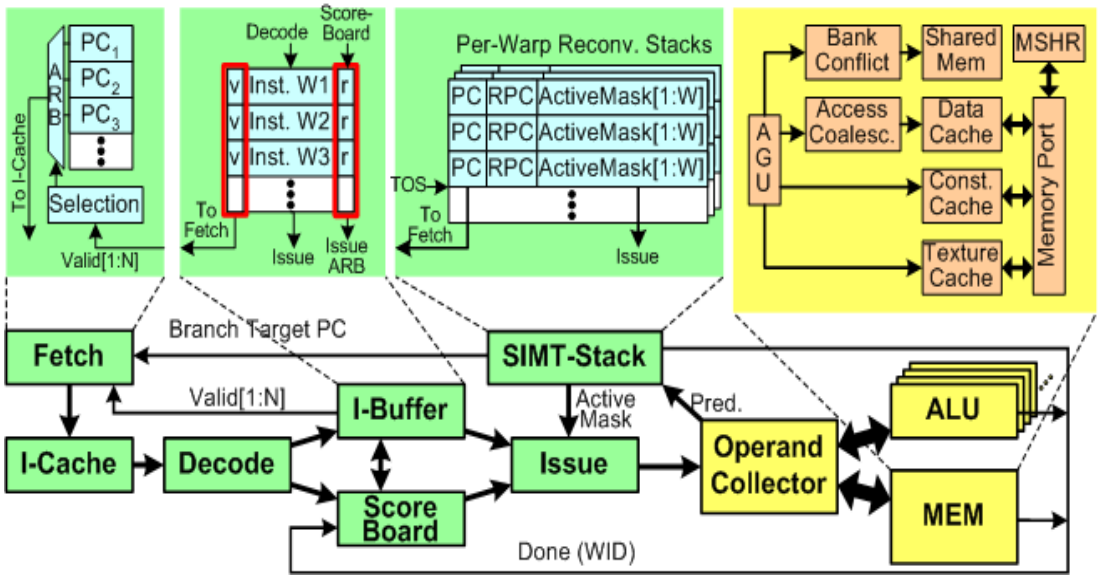

Figure 2 SIMT Core Architecture and Pipeline [2]

- Backend Units: After issuing the instruction it goes through register read and execution stages and at the end of the execution, scoreboard is updated to reflect the register write. The dependent instruction can now be issued according to warp scheduling policy. If the number of ALUs (N) are lesser than warp size then each cycles N threads will be processed and total duration of processing a single warp will be warp size divide by N. For instance if the number of ALUs are 8 and warp size is 32 then it will take 32/8 = 4 cycles to process a single warp. The number of special function units is generally much lesser than ALUs because of infrequent use which leads to slower execution for programs involving special function units. (E.g. sine, cosine, reciprocal, square root).
- Shared Memory: Apart from register file, GPU's SM also has shared memory (also known as scratchpad memory). This memory is programmable by the user and it is used to keep the data which is frequently used by different threads of a thread block. It is beneficial as it reduces the total number of memory accesses to global memory (which has access latency in hundreds of cycles).

1.4 Related Work:

GPGPUSim was developed by UBC and it supported Native ISA (sass) and PTX ISA for Tesla architecture [1]. Later on, they added support for PTX ISA even for Fermi architecture but they haven't supported the simulator for Native ISA of Fermi architecture. Nickolls et al [3] have published patent on the design of Unanimous branch mechanism. This thesis refers the mechanism of unanimous branch from the patent. Coon et al. [5] have published patent on mechanism of managing indirect function call instruction and hardware design. The work in this thesis also refers to this patent to understand indirect branch mechanism. Asfermi [9] has also done some work to explore the semantics of Fermi ISA.

2 Background on GPGPUSim

GPGPUSim is a General Purpose GPU Simulator that can simulate any CUDA/Open Computing Language (openCL) applications with cycle level performance accuracy. CUDA is a programming language developed by NVIDIA to simplify parallel programming on GPUs and it is quite similar to C. CUDA is gaining popularity due to its ease in parallel programming and performance improvement achieved by the same on GPUs. Cuda/C program is compiled by NVIDIA's "NVIDIA Cuda Compiler (NVCC)" compiler and it generates PTX (Parallel thread execution), SASS and ELF files. PTX is the intermediate representation of the program whereas SASS is native ISA (Instruction Set Architecture). ELF file contains information regarding constant data to be used by program. GPGPUSim currently supports PTX/SASS execution for GT200 (Testa Architecture) but it supports only PTX execution for Fermi architecture. Our work extends GPGPUSim to support Native ISA for Fermi architecture. The native ISA execution will better approximate the correlation of performance result compared to real GPU.

Figure 3 shows the pictorial representation of compilation and execution flow for PTX/SASS ISA. NVCC and ptxas compilers together generates executable which can run on hardware. "cuobjdump" extracts PTX, SASS and ELF files from this executable. This PTX can be used directly on GPGPUSim. PTXPlus file is generated using ELF file and SASS file. SASS is converted to PTXPlus using "cuobjdump_to_ptxplus" (parser tool which is part of GPGPUSim). ELF file is used to parse constant data which will be a copied in the same PTXPlus file. Once PTXPlus file is generated, GPGPUSim can execute it for functional or performance simulation. There is one to one mapping from SASS to PTXPlus. In order to avoid developing and maintaining two functional simulators (one for SASS and one for PTX), SASS is converted to PTXPlus representation. PTX and PTXPlus both uses the same parser and functional simulation engine.

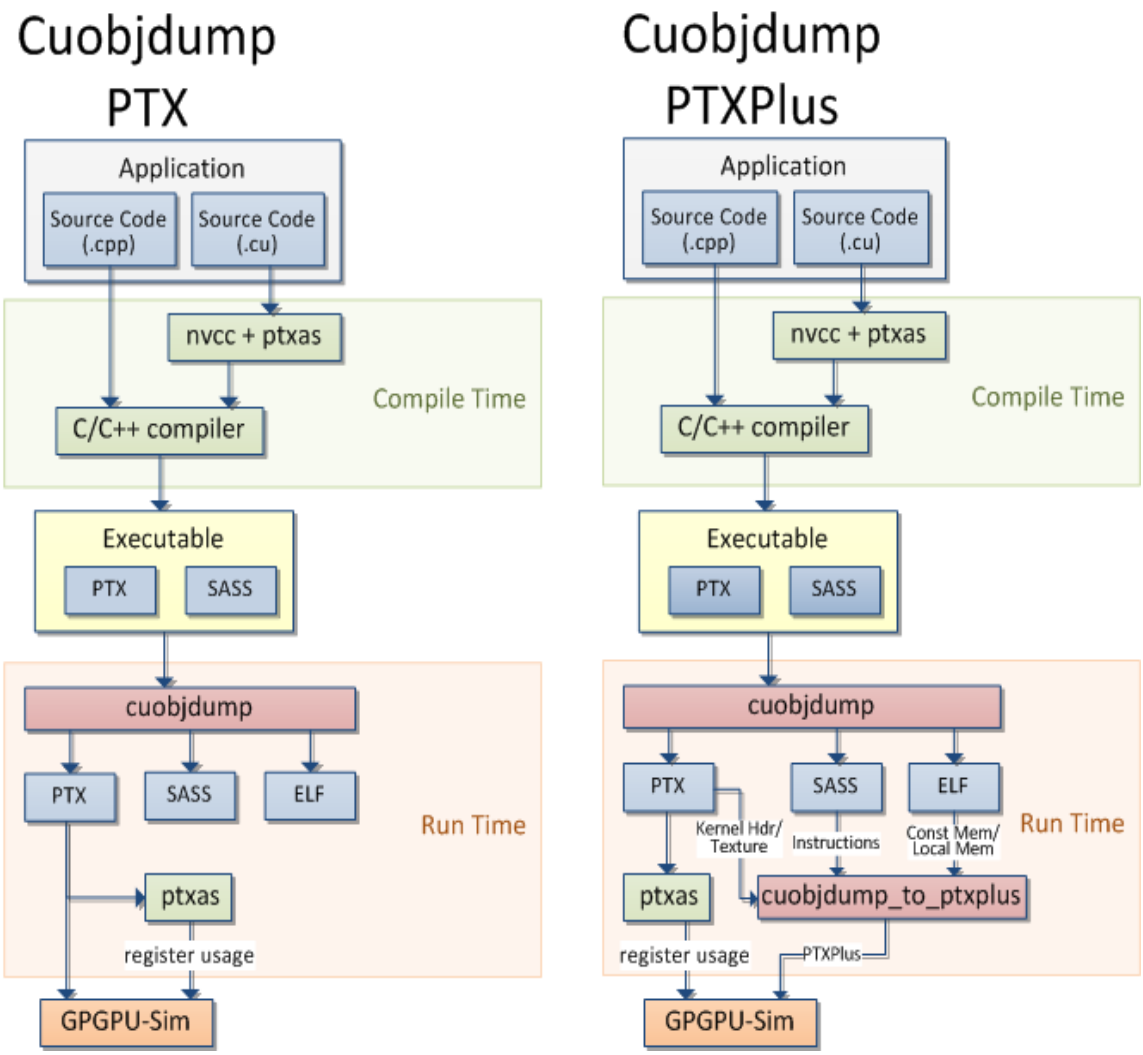

Figure 3 PTX and PTXPlus Compilation flow [2]

2.1 GPGPUSim Software Architecture:

Figure 4 shows the various classes used in creating GPGPUSim. “function_info” holds all the instructions after parsing it via PTX/PTXPlus parser. It also has symbol table which has a mapping of symbols and corresponding values. (E.g. registers, constants) “ptx_instruction”

class holds information regarding instruction after it is parsed from PTX parser. It holds information such as instruction type, instruction opcode, base and type modifiers, operands etc. This class is instantiated within “function_info” class. Figure 4 captures the relation between different classes. It shows which class has been instantiated within which class. “function_info” has a function which performs control flow analysis to determine post dominators. The information obtained from control flow is used in managing “SIMT Stack”. The work in this thesis doesn’t use information generated from control flow analysis because it relies on the information extracted from Native ISA. “Kernel_info” class holds information regarding kernel parameters, grid/block dimension and whether the kernel has been launched or not. “ptx_thread_info” class holds information regarding particular thread such as thread ID, register values, program counter of the threads as instructions are executed one after another. It also keeps information such as per thread local memory. “SIMT_Stack” class is instantiated within “core_t” class. It has the stack that is used to manage various control flow instructions. “ptx_cta_info” keeps the information regarding various threads within thread block. It keeps track of the number of threads within cta and the number of threads that have finished the execution.

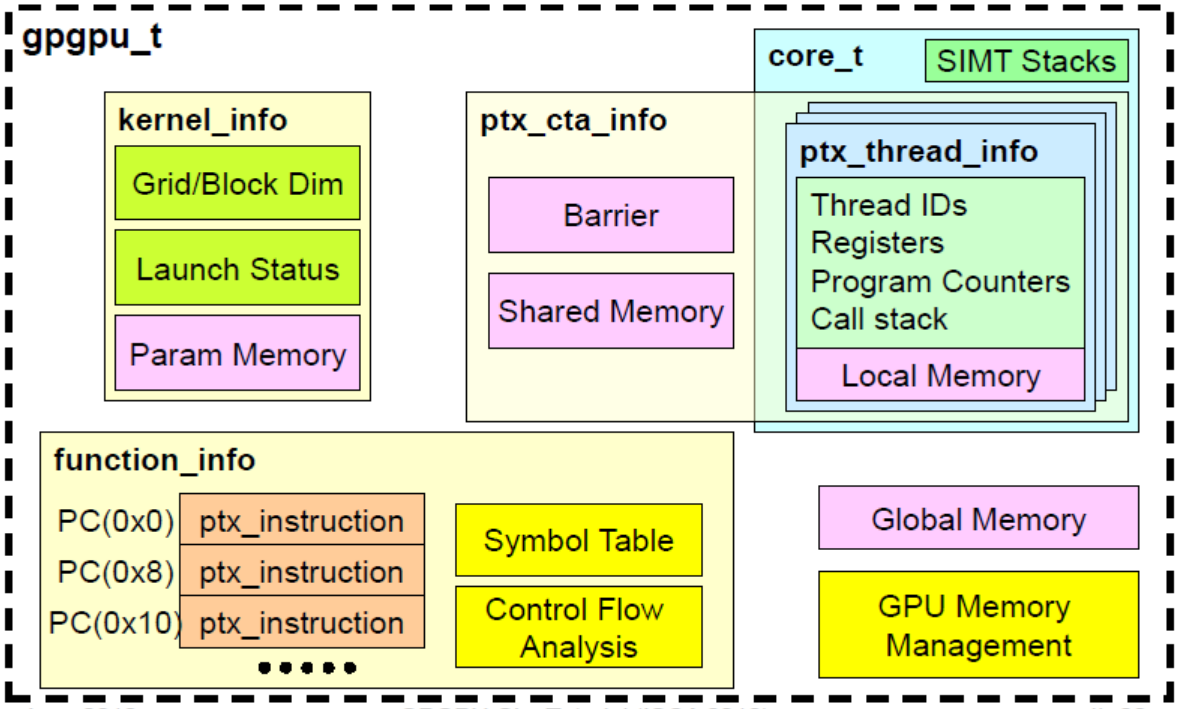

Figure 4 Software Architecture of GPGPUSim [2]

2.2 GPGPUSim extension to support Fermi architecture at Native ISA:

These are various steps involved in adding support for new instruction for Fermi architecture in GPGPUSim.

- Modification to SASS Parser: This section describes changes required in "cuobjdump_to_ptxplus" tool to support new instruction.
- Modification to PTX/PTXPlus Parser: This section describes the changes required in PTX or PTXPlus parser to support new instruction.
- Modification to Functional simulator: This section explains the details of modifying various section of GPGPUSim to model functional execution for the instruction under

consideration.

This section describes generic changes related to specific section described above in order to support new instruction in GPGPUSim.

Modification to SASS parser:

SASS parser is built using Flex/Bison which allows flexibility in modifying parser as the architecture evolves. The parsed instruction are stored in data structures before converting them to PTXPlus format. “CuobjdumpInst” class is the structure to hold information for one instruction. It has members such as instruction opcode, base modifiers, type modifiers, operands and various functions which operates on the member variables of this class (files: cuobjdumpInst.cc/cuobjdumpInst.h). “CuobjdumpInstList” class holds the all the instructions of the program. This class creates list of instructions using “CuobjdumpInst” class. Once all the instructions have been parsed, the PTXPlus file is generated by member function of “CuobjdumpInstList” class.

Cuobjdump_to_ptxplus/sass.l (lexer file):

This is lexer file which defines various token and passes then to parser. To support new instruction, token should be created for instruction keyword. Depending on the base and type modifier keywords, additional tokens may or may not be required.

Cuobjdump_to_ptxplus/sass.y (parser file):

This is the parser file which defines parsing rule/grammar for instruction. Depending on the syntax of instruction to be added, parser rule may or may not need any changes. New tokens which are added to lexer file also have to be defined in parser file and parsing rule has to be modified to support new token. And also according to parser rule modification, relevant function should be called to hold the instruction in relevant data structure for later processing.

Cuobjdump_to_ptxplus/CuobjdumpInstList.cc-CuobjdumpInstList.h-CuobjdumpInst.cc-CuobjdumpInst.h:

There four files defines classes which holds instruction and are used to generate PTXPlus file. For any new instruction, relevant code might have to be modified in multiple of these files. Specific details are out of scope for this document.

Modification to PTX/PTXPlus Parser:

This parser is also built using Flex/Bison to allow flexibility. This is similar to SASS parser except lexer file is “ptx.l” and parser file is “ptx.y”. After parsing the instruction, the data is stored in “func_info” class. This class stores information of kernel in the form of function. It has members which stores instructions (list of “ptx_instruction” class), kernel parameters and symbol table (class “symbol_table”). The modification will generally be done in “ptx_instruction” class if it’s instruction specific or in “operand_info” class if it is operand specific. “ptx_parser.cc” contains functions which are used by parser (ptx.y) to push instruction specific data into “func_info” class. Once the instructions are stored in func_info class correctly, they will be used for functional/performance simulation.

Modification to Functional simulator:

Adding a new instruction to GPGPUSim requires adding the functionality of instruction for correct functional simulation. Three different files will require modification for this purpose.

1. Adding instruction function information in opcodes.def
2. Adding instruction function definition in “instructions.h”
3. Adding instruction function definition in “instructions.cc”

3 Control Instructions Management

This chapter provides insight into the mechanism to support control instruction in GPGPU/SIMT architecture. Section 3.1 and 3.2 presents introduction to different mechanisms of managing branch instruction. Section 3.3 presents branch management methodology of GPGPU for IF-THEN ELSE type of branches and WHILE/FOR type of loops. It also presents detailed examples for branches and loop structures. Section 3.4 presents an example to manage “break” instruction. Finally section 3.5 presents algorithm in form of flow chart which covers rest of the control instruction present in NVIDIA Fermi architecture.

This section presents the various methods of code generation for branch instructions and hardware mechanism to support the same in GPUs. There are two ways in which branch instructions are handled in GPUs.

- Predication
- SIMT Stack based implementation

3.1 Predication:

Predication is generally used to implement IF-THEN ELSE type of structures. NVIDIA GPU has $p0-$p7 predicate registers which are used for general purpose code generation. $p7 predicate register, also known as predicate true (pt) is always true. To implement control dependent block using predication, control condition is set in predicate register and each instruction in that block is then pre-attached with predicate register. The threads which fail the predicate condition doesn’t execute that instruction. For example when “IF” part of the condition is pre-attached with $p0 then “ELSE” part will be pre-attached with !$p0. Predicated execution is efficient as it does not use conditional branch which eliminates the

need for stack access. The code example for the same is illustrated in the next section.

BRA.U:

".U" flag for branch instruction suggests that the instruction is unanimous branch and it doesn't require SIMT stack for the functionality and correctness. Unanimous branch indicates that most likely all threads will either be taken or non-taken, however it is not necessary. Unanimous branch relies on predication support provided by hardware. All the instruction of the "taken" and "non-taken" path are predicated. There are three difference scenarios that can happen with unanimous branch.

1. All threads follow the taken path. In this scenario, "Next PC" is set to target address.
2. All threads are follow non-taken path. In this scenario, "Next PC" is set to non-taken path's address.
3. Threads are diverging in two different paths. In scenario 3 also, "Next PC" is set to non-taken path.

Unanimous branch never modifies active mask. This is an important aspect which makes unanimous branch is more efficient than SIMT stack when there is no divergence. [3]

3.2 SIMT Stack based implementation:

SIMT Stack based mechanism is useful because predication based mechanism is only beneficial in certain situations. There can be three different scenarios for branch instruction based warp divergence in SIMT Stack based implementation. No divergence if all threads are either taking the branch (scenario 1) or not taking the branch (scenario 2). Divergence when threads are going in different directions (scenario 3). SIMT Stack is not required if threads are

not diverging. When threads are diverging, one path information is pushed onto the stack while second path will be executed. The code example is explained with details in the later sections.

3.2.1 Token Structure and Types:

There are various types of control instruction that changes the flow of program execution. For instance conditional/unconditional branches, break, function calls. Managing these instructions in CPU based execution is easier compared to GPU based execution. In GPU based execution managing control flow is relatively complex due to SIMT execution model of GPUs. When CPU executes control instruction, it can directly branch to target address or next PC according to the outcome of control instruction. There is a possibility of warp divergence (different threads in a warp taking different paths) in GPU which may lead to serial execution for different paths.

Table 1 Token Structure

ACTIVE MASK	RECONVERGENCE PC	ENTRY TYPE

The hardware support required for control instruction is added in the form of stack. The content of stack will depend on the type of instruction. The general term for one stack entry is token and Table 1 shows the structure of the token. The “ACTIVE MASK” is 32 bit in size (assuming warp size of 32) and it stores the 1 bit information per thread regarding thread state (active/inactive). “RE-CONVERGENCE PC” is a program counter value and it is common for all threads within a warp. “ENTRY TYPE” differentiates various types of instruction. Each SIMT Core also have per warp “Active Mask” and “Active PC” apart from stack components described above. Per warp “Active Mask” disables the threads that are not active and per

warp “Active PC” is program counter for each warp.

Table 2 shows the various types of instructions and its token information.

- SSY (entry 1 in Table 2) instruction is generally used to push the re-convergence point information. The token pushed onto the stack includes the Active Mask of warp when instruction is encountered, re-convergence point information which is a part of instruction semantic and “token type” which will be SSY.
- The unconditional branch (entry 2 in Table 2) instruction does not require any token to be pushed onto stack as it is non-divergent.
- The conditional branch can be divergent as well as non-divergent. In case of non-divergent conditional branches (entry 2 in Table 2), token is not required. In case of conditional divergent branch (entry 3 in Table 2), stack support may be required (backward vs forward branch).
    - Forward direct branch will generally have two paths, taken path and non-taken path. Taken path (THEN block) is the one to be pushed onto the stack while non-taken path (ELSE block) is the one to be executed first. The token pushed onto stack contains Taken Mask (which is stored in “Active Mask”), Taken PC (which is stored in “re-convergence PC”) and DIV type (which stored in “Token type”).
- BRX instruction is used to support virtual function calls or indirect function calls. BRX instruction is indirect branch and can have more than two target addresses. The execution for different paths is serialized. Generally there is always a PRET instruction before BRX instruction for which PRET token (entry 8 in Table 2) is pushed onto stack. When there is more than one destination for BRX instruction, token shown in entry 5 Table 2 is pushed onto stack. The “Re-convergence PC” field of pushed token is “Active PC” of warp. Once the execution of a particular target address is over, the same BRX instruction is executed again and this time it branches to another of remaining targets

(it requires hardware support apart from stack which is not mentioned here). When there is only one target left, no token is pushed onto stack. Since PRET token was pushed onto stack initially, that token is popped at this time, execution is resume from the PC set by PRET token.

- CALL instruction (entry 7 in Table 2) is non-divergent instruction. While encountering call instruction, call token is pushed onto stack. The Active mask will be the mask at the time instruction is encountered and next PC is used as Re-convergence PC. The last instruction of function call is RETURN and when RETURN instruction is encountered, call token is popped from the stack. PREBRK instruction ((entry 9 in Table 2) is used to support break/continue instruction of C language.

Table 2 Various Tokens [5]

Entry ID	Instruction	Divergent?	Token Pushed on stack		
			Token Type	Active Mask	Re-convergence PC
1	SSY	Never	SSY	Active Mask	Target PC
2	BRA	N	------	------	------
3	BRA	Y	DIV	Taken Mask	Taken PC
4	BRX	N	-----	------	------
5	BRX	Y(1)	DIV	Non taken Mask	Active PC
6	BRX	Y(2)	DIV	Non taken Mask	Non Taken PC
7	CALL	------	CALL	Active Mask	Non Taken PC
8	PRET	------	PRET	Active Mask	Target PC
9	PREBRK	------	PBK	Active Mask	Target PC

3.3 IF THEN ELSE and WHILE/FOR Management:

The various types of branches observed in high level programming languages and their translation to equivalent Native assembly for Fermi ISA is presented in this section. In addition, this section provides regarding hardware support for those branches.

1. "IF THEN-ELSE" Type of branch code Layout in Native ISA and hardware support required in Fermi Native ISA
2. While/For loop type of branch code layout in Native ISA and hardware support required in Fermi Native ISA.

3.3.1 "IF THEN-ELSE" branch management:

There are two hardware mechanisms by which "IF THEN-ELSE" block can be supported.

- Predication based implementation (also BRA.UNI as mentioned earlier) and
- SIMT Stack based implementation.

3.3.1.1 Predication based Implementation:

In predication based implementation, IF-THEN block and ELSE block, both are predicated with condition. Program will only execute for threads which passes the condition and rest of the

```
Psuedo Code:
ISETP.GT.AND.U32 P0, pt, Index, x1, pt;
@P0   d_c [index] = d_a [index] + d_b[index] ;     //THEN Block
@!P0  d_c [index] = d_a [index] * d_b [index] ;    //ELSE Block
d_e [index] = d_c [index] + d_d [index] ;
```

Figure 5 Predication for IF-THEN ELSE Type of branch

thread will be masked. Predication is efficient when number of instruction in IF-THEN block and ELSE block is less. The example is presented in Figure 5. As seen in the Figure 5 THEN block is masked with “@P0” condition and “ELSE” block is masked with “@!P0” condition.

```
int index = blockIdx.x*blockDim.x + threadIdx.x;
        if (index > x1)                                          // Section 1 - THEN
        {
                d_c [index] = d_a [index] + d_b[index] ;
        }
        Else                                                     //Section 1 - ELSE
        {
                d_c [index] = d_a [index] * d_b [index] ;
        }

        d_e [index] = d_c [index] + d_d [index] ;
```

Figure 6 “C language Program”

3.3.1.2 SIMT Stack based Implementation:

Figure 6 shows “IF THEN-ELSE” program and Figure 7 shows Native assembly code (pseudo code) generated for the same in Fermi architecture. Multiple threads runs in lockstep due to SIMD nature of execution in GPUs and they should not diverge too much to maintain good hardware utilization. Thread divergence should be re-converged at the earliest to achieve better hardware utilization. The code layout of IF-THEN ELSE block in Native ISA is such that ELSE section precedes THEN section as shown in Figure 7. In case of “IF THEN-ELSE” block, earliest re-convergence can happen immediately after “ELSE” block is over. This point is marked as “RECONVERGENCE” label in Figure 7. In order to support divergence at branch and

re-convergence at the end of “ELSE” block (Figure 6), “SSY” instruction is added in the Native program by compiler which has the information of nearest re-convergence point for the branch instruction. Now while executing Native ISA, if the conditional forward branch instruction is encountered then there can be three different scenarios:

1. Threads are diverging. In this case, a new entry is created on stack. It contains “Active Mask” and Target Address of “If block” and token type which will be “DIV” (Divergence Token—Name Taken from NVIDIA patent). The “Active Mask” and “Active PC” of the warp are set to “ELSE” block values. The program starts executing the “ELSE” block until it encounters POP instruction. (“.S” flag in Fermi Architecture which is inserted by compiler) At this point, top entry is popped from the stack and “Active Mask” and “Active PC” values are set using the values read from top of stack (“Then” block). Now “Then” block is executed until it also encounters POP instruction at which point token is popped (SSY Token) from the stack and “Active Mask” and “Active PC” are set using these values. At this points threads which diverged at branch instruction are re-converged again and execution continues for the rest of the program.

2. Threads are not diverging and all threads are taking going to “Then” block

3. Threads are not diverging and all threads are taking “Else” block.

In 2nd and 3rd case, new stack entry creation is not required. The value of “Active Mask” will not change and the value of Active PC is set according to whether all threads are going to “Then” block or “Else” block.

Table 3 and Table 4 are shown here for more details and illustration purposes. Table 3 shows the native program code and comment section explain meaning of relevant instruction from perspective of the discussion presented so far. Table 4 presents the status of the stack as program is executed from the first instruction till it exits.

```
                    SSY RECONVERGENCE;
                    setp.le.and P0, index, x1;
                    @P0 BRA section 1- THEN:
Section 1 – ELSE:
                    ….
                    ….
                    “POP Stack” (NOP.S)
Section 1- THEN:
                    ….
                    ….
                    “POP Stack” (NOP.S)
RECONVERGENCE:
```

Figure 7 Fermi Architecture Native ISA Translation for program in Figure 6

Table 3 Native Assembly Code

PC	Instruction	Comment
0000	MOV R1, c [0x1] [0x100];	
0020	SSY 0xf0;	SSY Instruction
0028	…..	…..
0090	@P0 BRA 0xb8;	Branch corresponding to Outer IF Then-Else
0098	LD.E R3, [R2];	ELSE PART
00a0	LD.E R4, [R4];	ELSE PART
00a8	IADD R2, R4, R3;	ELSE PART
00b0	ST.E.S [R6], R2;	ELSE PART. Notice ".S" flag.
00b8	LD.E R5, [R4];	IF THEN PART
……	….	….
00e8	NOP.S CC.T;	Last Instruction of Outer If. Notice ".S" flag.
00f0	LD.E R3, [R10];	Threads Synchronizes at this point.
00f8	LD.E R4, [R8];	….
0100	LD.E R2, [R6];	Go till Exit.
….	….	….
0140	EXIT;	

Table 4 SIMT Stack during Program execution

Entry ID	PC	Active Mask	TOS	TOS-1	TOS-2
1	0x000	11111111	Empty	Empty	Empty
2	0x020	11111111	Empty	Empty	Empty
3	0x028	11111111	SSY, 0xFF, 0xF0	Empty	Empty
4	0x090	11111111	SSY, 0xFF, 0xF0	Empty	Empty
5	0x098	11110000	DIV, 0x0F, 0xB8	SSY, 0xFF, 0xF0	Empty
Execute this way up to the end of “ELSE” block. POP token (“.S”) flag indicates popping stack entry.					
	…..				
6	0x0B0	11110000	DIV, 0x0F, 0xB8	SSY, 0xFF, 0xF0	Empty
7	0x0B8	`00001111	SSY, 0xFF, 0xF0	Empty	
Execute this way up to the end of “Then” block.					
..					
8	0x0E8	`00001111	SSY, 0xFF, 0xF0	Empty	
9	0xF0	11111111	Empty		

3.3.2 While/For Loop branch management in Fermi Native ISA:

While loop implementation at assembly level generally uses backward branch. The implementation challenge arises in GPU due to SIMT execution model. There are two different scenarios that can happen.

- Threads are not diverging. When the loop condition is independent of thread ID or in other words the threads are not diverging at backward branch, the branch is taken until it is false. In this case the native assembly code will not generate any synchronization points and there will not be any “SSY” instruction. It will not create any entry onto stack at all.
- Threads are diverging. When the loop condition is dependent on thread ID and each thread may execute the different number of loop iteration. In this scenario, threads are disabled one by one as they fail the loop test and remaining threads of the warp continues. The loop is executed until none of the threads are active. Once all threads are done, Active mask is restored correctly and next instructions in sequence starts executing. The generated code will have “SSY” instruction at the beginning and corresponding “.S” flag at the end of while loop. This is shown in the assembly program in Figure 9. The C code corresponding to assembly code is shown in Figure 8. Each thread executes different number loop iterations. The manipulation of stack as the program executes is shown in Table 5. As shown in Figure 9, threads are disabled one by one until none of the threads are active. At this time, next instruction is executed which pops token from the stack and execution starts from PC indicated by popped stack entry. Execution continues till it reaches exit instruction. To correctly restore the “Active Mask” once all threads fails the loop test, per backward branch “while mask” is required. “while_mask” maintains value of “Active mask” that is observed while executing the branch for the very first time.

```
int i = blockDim.x * blockIdx.x + threadIdx.x;
   int j;
      if (i < N)
      {
                  C[i] = 1;
                  For (j=1;j<=i;j++)
                  {
                    C[i] = C[i] + j;
                  }
          }
```

Figure 8 C language Program with “for” loop

```
        Address     Binary Code                 Assembly Instruction
        /*0000*/   /*0x00005de428004404*/  MOV R1, c [0x1] [0x100];
        /*0008*/   /*0x94001c042c000000*/  S2R R0, SR_CTAid_X;
        /*0010*/   /*0x84009c042c000000*/  S2R R2, SR_Tid_X;
        /*0018*/   /*0x20009c0320044000*/  IMAD.U32.U32 R2, R0, c [0x0] [0x8], R2;
        /*0020*/   /*0xb021dc231b0e4000*/  ISETP.GE.AND P0, pt, R2, c [0x0] [0x2c], pt;
        /*0028*/   /*0x000001e780000000*/  @P0 EXIT;
        /*0030*/   /*0xa0211c4340004000*/  ISCADD R4, R2, c [0x0] [0x28], 0x2;
        /*0038*/   /*0x00001de218fe0000*/  MOV32I R0, 0x3f800000;
        /*0040*/   /*0x0421dc23188ec000*/  ISETP.LT.AND P0, pt, R2, 0x1, pt;
        /*0048*/   /*0x00401c8590000000*/  ST [R4], R0;
        /*0050*/   /*0x000001e780000000*/  @P0 EXIT;
        /*0058*/   /*0x04209c034800c000*/  IADD R2, R2, 0x1;
        /*0060*/   /*0x0400dde218000000*/  MOV32I R3, 0x1;
        /*0068*/   /*0xc000000760000000*/  SSY 0xa0;  Sync Instruction
        /*0070*/   /*0x0d215e0418000000*/  I2F.F32.S32 R5, R3;
        /*0078*/   /*0x0430dc034800c000*/  IADD R3, R3, 0x1;
        /*0080*/   /*0x00501c0050000000*/  FADD R0, R5, R0;
        /*0088*/   /*0x0c21dc231a8e0000*/  ISETP.NE.AND P0, pt, R2, R3, pt;
        /*0090*/   /*0x600001e74003ffff*/  @P0 BRA 0x70;   Backward Branch of “For”
loop
        /*0098*/   /*0x00001df440000000*/  NOP.S CC.T; “.S” means Popping Stack Entry
        /*00a0*/   /*0x00401c8590000000*/  ST [R4], R0;
        /*00a8*/   /*0x00001de780000000*/  EXIT;
```

Figure 9 Native Assembly Code

Table 5 SIMT Stack during Program execution

Active PC	Active Mask	Stack	Note
0x008	: 11111111111111111111111111111111		
0x010	: 11111111111111111111111111111111		
0x018	: 11111111111111111111111111111111		
….			
0x050	: 11111111111111111111111111111111		@P0Exit
0x058	: 01111111111111111111111111111111		
0x060	: 01111111111111111111111111111111		
0x068	: 01111111111111111111111111111111		SSY Inst
0x070	: 01111111111111111111111111111111	Stack mask: 01111111111111111111111111111111 rp: 0x0a0 SYNC	
0x078	: 01111111111111111111111111111111	Stack mask: 01111111111111111111111111111111 rp: 0x0a0 SYNC	
0x080	: 01111111111111111111111111111111	Stack mask: 01111111111111111111111111111111 rp: 0x0a0 SYNC	
0x088	: 01111111111111111111111111111111	Stack mask: 01111111111111111111111111111111 rp: 0x0a0 SYNC	
0x090	: 01111111111111111111111111111111	Stack mask: 01111111111111111111111111111111 rp: 0x0a0 SYNC	
0x070	: 00111111111111111111111111111111	Stack mask: 01111111111111111111111111111111 rp: 0x0a0 SYNC	
0x078	: 00111111111111111111111111111111	Stack mask: 01111111111111111111111111111111 rp: 0x0a0 SYNC	
0x080	: 00111111111111111111111111111111	Stack mask: 01111111111111111111111111111111 rp: 0x0a0 SYNC	
0x088	: 00111111111111111111111111111111	Stack mask: 01111111111111111111111111111111 rp: 0x0a0 SYNC	
0x090	: 00111111111111111111111111111111	Stack mask: 01111111111111111111111111111111 rp: 0x0a0 SYNC	
……			
0x070	: 00000000000000000000000000000011	Stack mask: 01111111111111111111111111111111 rp: 0x0a0 SYNC	
0x078	: 00000000000000000000000000000011	Stack mask: 01111111111111111111111111111111 rp: 0x0a0 SYNC	
0x080	: 00000000000000000000000000000011	Stack mask: 01111111111111111111111111111111 rp: 0x0a0 SYNC	
0x088	: 00000000000000000000000000000011	Stack mask: 01111111111111111111111111111111 rp: 0x0a0 SYNC	
0x090	: 00000000000000000000000000000011	Stack mask: 01111111111111111111111111111111 rp: 0x0a0 SYNC	
0x070	: 00000000000000000000000000000001	Stack mask: 01111111111111111111111111111111 rp: 0x0a0 SYNC	
0x078	: 00000000000000000000000000000001	Stack mask: 01111111111111111111111111111111 rp: 0x0a0 SYNC	
0x080	: 00000000000000000000000000000001	Stack mask: 01111111111111111111111111111111 rp: 0x0a0 SYNC	
0x088	: 00000000000000000000000000000001	Stack mask: 01111111111111111111111111111111 rp: 0x0a0 SYNC	
0x090	: 00000000000000000000000000000001	Stack mask: 01111111111111111111111111111111 rp: 0x0a0 SYNC	
0x098	: 01111111111111111111111111111111	Stack mask: 01111111111111111111111111111111 rp: 0x0a0 SYNC	NOP.S Inst
0x0a0	: 01111111111111111111111111111111		
0x0a8	: 01111111111111111111111111111111		

3.4 Break/Pre-break Management:

Break/Pre-break instruction is generally used by compiler when generating native code for programs that use break/continue keyword of C/CUDA language. Apart from stack components discussed so far, break instruction requires the usage of “Disable Mask”. This mask is used to disable threads in the loop which are done while rest of the loop still keeps on executing. The example here tries to illustrate the stack management for the program which uses pre-break/break instructions.

```
__global__ void kernel_update(float * A, float * B,
float * C, unsigned int N, unsigned int * hasproxy)
{
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
if (i >= N)
return ;
float dij = 2.0;
    if (!hasproxy[i]) //Initialize array with 0 or 1
    //randoml of size thread count (Total no of threads)
    {
        for ( unsigned int j = 0; j < N; j++ )  {
          if ( j == i )
          continue;
          if ( (i * N + j) > (i*j + i))  {
             dij = A[i] * (B[j] - B[i]) ;
             break;
          } }
    }
    C[i] = B[i] + dij ;
}
```

Figure 10 C/CUDA Kernel

Figure 11 Native ISA code for Figure 10

```
/*0000*/   /*0x00005de428004404*/  MOV R1, c [0x1] [0x100];
/*0008*/   /*0x94001c042c000000*/  S2R R0, SR_CTAid_X;
/*0010*/   /*0x84009c042c000000*/  S2R R2, SR_Tid_X;
/*0018*/   /*0x20009c0320044000*/  IMAD.U32.U32 R2, R0, c [0x0] [0x8], R2;
/*0020*/   /*0xb021dc03188e4000*/  ISETP.LT.U32.AND P0, pt, R2, c [0x0] [0x2c], pt;
/*0028*/   /*0x000021e780000000*/  @!P0 EXIT;
/*0030*/   /*0x0820de036000c000*/  SHL R3, R2, 0x2;
/*0038*/   /*0x4000000768000004*/  PBK 0x150;
/*0040*/   /*0xc0301c0348004000*/  IADD R0, R3, c [0x0] [0x30];
/*0048*/   /*0x00001c8580000000*/  LD R0, [R0];
/*0050*/   /*0xfc01dc031a8e0000*/  ISETP.NE.U32.AND P0, pt, R0, RZ, pt;
/*0058*/   /*0x400001e740000003*/  @P0 BRA 0x130;
/*0060*/   /*0xb3f1dc03190e4000*/  ISETP.EQ.U32.AND P0, pt, RZ, c [0x0] [0x2c], pt;
/*0068*/   /*0x800001e740000002*/  @P0 BRA 0x110;
/*0070*/   /*0xfc001de428000000*/  MOV R0, RZ;
/*0078*/   /*0x0021dc03190e0000*/  ISETP.EQ.U32.AND P0, pt, R2, R0, pt;
/*0080*/   /*0xc000000760000001*/  SSY 0xf8;
/*0088*/   /*0x000001f440000000*/  @P0 NOP.S CC.T;
/*0090*/   /*0x00211c0320040000*/  IMAD.U32.U32 R4, R2, R0, R2;
/*0098*/   /*0xb0215c0320004000*/  IMAD.U32.U32 R5, R2, c [0x0] [0x2c], R0;
/*00a0*/   /*0x1441dc031b0e0000*/  ISETP.GE.U32.AND P0, pt, R4, R5, pt;
/*00a8*/   /*0x000001f440000000*/  @P0 NOP.S CC.T;
/*00b0*/   /*0x90009c4340004000*/  ISCADD R2, R0, c [0x0] [0x24], 0x2;
/*00b8*/   /*0x80311c0348004000*/  IADD R4, R3, c [0x0] [0x20];
```

```
        /*00c0*/    /*0x90301c0348004000*/   IADD R0, R3, c [0x0] [0x24];
        /*00c8*/    /*0x00209c8580000000*/   LD R2, [R2];
        /*00d0*/    /*0x00001c8580000000*/   LD R0, [R0];
        /*00d8*/    /*0x00411c8580000000*/   LD R4, [R4];
        /*00e0*/    /*0x00209d0050000000*/   FADD R2, R2, -R0;
        /*00e8*/    /*0x08409c0058000000*/   FMUL R2, R4, R2;
        /*00f0*/    /*0x00001de7a8000000*/   BRK;
        /*00f8*/    /*0x04001c034800c000*/   IADD R0, R0, 0x1;
        /*0100*/    /*0xb001dc031a8e4000*/   ISETP.NE.U32.AND P0, pt, R0, c [0x0]
[0x2c], pt;
        /*0108*/    /*0xa00001e74003fffd*/    @P0 BRA 0x78;
        /*0110*/    /*0x90301c0348004000*/   IADD R0, R3, c [0x0] [0x24];
        /*0118*/    /*0x00009de219000000*/   MOV32I R2, 0x40000000;
        /*0120*/    /*0x00001c8580000000*/   LD R0, [R0];
        /*0128*/    /*0x00001de7a8000000*/   BRK;
        /*0130*/    /*0x90301c0348004000*/   IADD R0, R3, c [0x0] [0x24];
        /*0138*/    /*0x00009de219000000*/   MOV32I R2, 0x40000000;
        /*0140*/    /*0x00001c8580000000*/   LD R0, [R0];
        /*0148*/    /*0x00001de7a8000000*/   BRK;
        /*0150*/    /*0x08001c0050000000*/   FADD R0, R0, R2;
        /*0158*/    /*0xa0309c0348004000*/   IADD R2, R3, c [0x0] [0x28];
        /*0160*/    /*0x00201c8590000000*/   ST [R2], R0;
        /*0168*/    /*0x00001de780000000*/   EXIT;
```

Table 6 SIMT Stack Management for Figure 10/11

Entry ID	PC	Active Mask	Disable Mask	TOS	TOS-1	TOS-2
1	0x18	111111111111	0x000			
2	..					
3	0x38	111111111111	0x000			
4	0x40	111111111111	0x000	PBK, 0xFFF, 0x150		
5	..					
6	0x58	111111111111	0x000	PBK, 0xFFF, 0x150		
7	0x60	001100011110	0x000	BRA, 0xCE1, 0x130	PBK, 0xFFF, 0x150	
8	..					
9	0x80	001100011110	0x000	BRA, 0xCE1, 0x130	PBK, 0xFFF, 0x150	
10	0x88	001100011110	0x000	SSY, 0x31E, 0xF8	BRA, 0xCE1, 0x130	PBK, 0xFFF, 0x150
11	0x90	000000011110	0x000	SSY, 0x31E, 0xF8	BRA, 0xCE1, 0x130	PBK, 0xFFF, 0x150
12	..					
13	0xa8	000000011110	0x000	SSY, 0x31E, 0xF8	BRA, 0xCE1, 0x130	PBK, 0xFFF, 0x150
14	0xb0	000000011110	0x000	SSY, 0x31E, 0xF8	BRA, 0xCE1, 0x130	PBK, 0xFFF, 0x150
15	..					
16	0xf0	000000011110	0x000	SSY, 0x31E, 0xF8	BRA, 0xCE1, 0x130	PBK, 0xFFF, 0x150
17	0xf8	001100000000	0x01E	BRA, 0xCE1, 0x130	PBK, 0xFFF, 0x150	
18	...					
19	0x108	001100000000	0x01E	BRA, 0xCE1, 0x130	PBK, 0xFFF, 0x150	
20	0x78	001100000000	0x01E	BRA, 0xCE1, 0x130	PBK, 0xFFF, 0x150	
21	0x80	001100000000	0x01E	BRA, 0xCE1, 0x130	PBK, 0xFFF, 0x150	
22	0x88	001100000000	0x01E	SSY, 0x300, 0xF8	BRA, 0xCE1, 0x130	PBK, 0xFFF, 0x150
23	..					
24	0xf0	001100000000	0x01E	SSY, 0x300, 0xF8	BRA, 0xCE1, 0x130	PBK, 0xFFF, 0x150
25	0xf8	000000000000	0x31E	BRA, 0xCE1, 0x130	PBK, 0xFFF, 0x150	
26	None of threads are active here due to which a token is popped from the stack.					
27	0x130	110011100001	0x01E	PBK, 0xFFF, 0x150		
28	…					
29	0x148	110011100001	0x01E	PBK, 0xFFF, 0x150		
30	0x150	111111111111	0x000			
31	…					
32	Till exit					

The Figure 10 shows the C/CUDA kernel. It shows the use of continue and break keywords. Figure 11 shows the Native Code for the same kernel. Table 6 shows the change in SIMT stack as kernel is executed. This example shows the method of managing “prebreak” and “break”

instruction encountered in GPU programs. When program reaches “prebreak” instruction during execution, it pushes “PBK” token onto stack as shown in Table 6 entry 4. The program continues to execute until it reaches “brk (break)” instruction. At this point, token is popped from the stack. If the popped token is any token other than “PBK” (table 3.8 entry 17->18), disable mask is set and those threads are disabled. Active Mask and PC are set from popped token. This process continues for “brk” instruction until popped token is “PBK”. When popped token is “PBK”, disable mask will be reset again and Active PC and Mask are set using values from popped token. ”@P0 NOP.S CC.T” instruction disables the threads for which @P0 is true (via resetting the “Active Mask”).

3.5 Control Flow Management Algorithm:

Figure 12 shows the control flow management algorithm. It presents the SIMT stack management flow for various type of instructions. Once instruction is fetched, it is decoded and instruction type, target address is forwarded to stack by decode stage. In this implementation, Stack also has information of current “Active mask” and “Active PC” for each warp. (Referring to per warp “Active Mask” and “Active PC”. They are not part of token.) All the steps mentioned in this section refers to Figure 12.

- Stack look up is performed (step 3), when none of the threads are active according to active mask (step 2). If the stack is empty according to step 3, execution is over for that particular warp, if not, stack is popped and Active mask/PC are set using values received from popped stack entry. If few threads are active in step 2 then it proceeds to step 6 where type of the instruction is checked.
- If the type of instruction is conditional branch according to step 6 then flow proceeds to step 7. Depending on backward/forward branch, flow proceeds to step 11/step 10. For/While loops are implemented using backward branches whereas IF-THEN ELSE type of control structures are implemented using forward branches. For/While loop can be of two types. Thread ID dependent and thread ID independent. Thread ID independent while loops are non-divergent and thread ID dependent while loops are divergent (All thread might run different iterations of the loop). In any case, managing any of the backward branches doesn’t add any entry to the stack. In case of non-divergent backward branch, branch is taken and no entry is added to the stack. This will continue until all the threads fails the loop test. In case of divergent branch, when a particular thread fails loop test condition, active mask for that thread is disabled and backward branch is taken until none of the threads are active. Once all threads fail the loop test condition, active mask is restored using “while_mask”.

- If the branch in step 7 is forward branch then flow moves to step 10 where it checks whether it's Uniform branch or normal branch. As discussed earlier, Uniform branches doesn't use stack and they relies on predication support. If the threads are diverging (as shown in step 18), then PC is incremented to next PC in instruction sequence. If the threads are not diverging for uniform branch then depending on the direction of threads PC is set to either PC+1 (step 33) or Target PC (step 34). Uniform branches improves efficiency of branch divergence by avoiding use of stack.
- If the branch observed in step 10 is not uniform branch then its normal conditional backward branch instruction which requires stack support in case of divergence. If the threads are diverging (step 21) then token is pushed onto stack. The token includes "taken mask" as active mask, "taken pc" as re-convergence PC, "DIV" as token type and "non-taken mask" as "non-taken mask" of token. The active PC of the warp is set to non-taken PC (which is PC+1) and active mask is set to non-taken mask. If the threads are not diverging at step 13, then according to direction of all threads PC is set to either Taken or non-taken path.
- If the instruction in step 6 is not conditional branch then it is tested for various other type of instructions. Step 8 checks whether instruction is unconditional branch or not. If step 8 is true then active PC is set to target PC and stack doesn't need to be modified because it's unconditional branch.
- Step 12 checks for whether the instruction is any of SSY/PBK/CALL. If any of these three cases, token is pushed onto stack. The token type will be SSY or PBK or CALL, the Re-convergence PC for SSY and PBK is extracted from instruction and for CALL instruction its PC+1. The active mask pushed onto the stack for all three instruction will be the active mask of the warp at the time the instruction is encountered.
- Step 22 checks if instruction is RETP instruction. This instruction is generally encountered at the end of the function call. For this instruction, Token is popped from the top of stack. Active mask and Active PC are set using the values from the popped

token.

- Step 25 checks if the instruction is EXIT. If the exit instruction is encountered and stack is empty then execution is over but if the stack is not empty then stack is popped (step 30) and active PC and active mask are set using values popped from the stack.
- Step 35 checks whether it's instruction with ".S" flag which indicates to pop the entry from the stack. Whether instruction is predicated or nor (step 36) also makes the difference with ".S" flag. In case if the instruction is predicated (step 38), stack is not popped. Only active mask will be disabled for the threads which follow the predicate condition. If the instruction is not predicated then stack is popped and active mask and active PC are set using the values from the popped entry of stack.
- Step 40 checks if the instruction is break (BRK). The predicated or non-predicated BRK instruction are managed differently. For non-predicated BRK instruction, stack is popped and entry type is checked. If the entry type is PBK then active mask and active PC are set using popped entry and disable mask will be reset. If the popped entry in not PBK then active mask and active PC are set using popped entry but disable mask doesn't change. If this instruction is predicated PBK then check is performed (step 44) to find out if any of the threads are active according to predicate condition. If none of the threads are active then entry is popped and active PC/active mask are set using popped token. If some of the threads are active then disable mask is set for those threads.

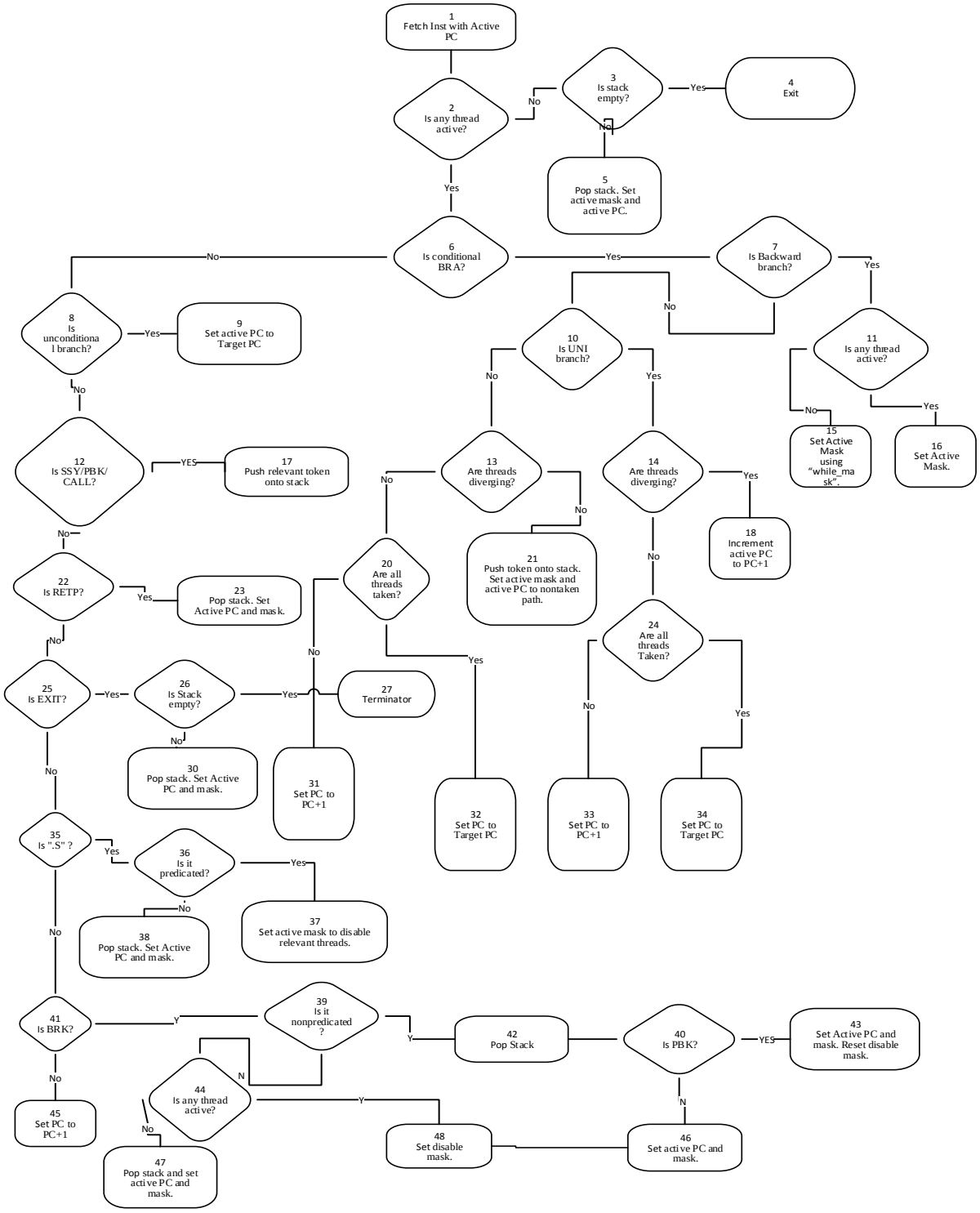

Figure 12 Flow chart of control instructions

4 Benchmarks and Instruction Support

This chapter discusses the various benchmarks that were implemented for Fermi architecture. The benchmark were selected so as to get broad range of different GPGPU applications. This naturally resulted in selecting benchmarks from mix of Rodinia benchmark suite, NVIDIA GPU Computing SDK and GPGPUSim benchmark suite.

This section provides discussion of various benchmarks.

Matrix Multiplication:

The better performance for matrix based calculation is always useful because of large application base of matrix math. This benchmark uses shared memory to optimize matrix multiplication performance.

Neural Network:

Neural network applications have large matrix and vector calculations. They are highly suitable to optimize on GPUs because of the parallel nature of application.

Apart from the benchmark described above, the following benchmarks were used. Dwt (NVIDIA SDK), b+tree (rodinia benchmark suite), pathfinder (rodinia benchmark suite).

4.1 Instruction specific SASS to PTXPlus Conversion:

This section describes the instruction specific changes that were done to support Fermi architecture. The format described for all upcoming example has SASS and PTXPlus representation of the instruction.

1. Parameter c [0x1] [0x100] is unknown and the register R1 is never used in any program so it has been ignored for simulation. This instruction is always first instruction for any program for Fermi architecture.

SASS	MOV R1, **c [0x1] [0x100];**
PTXPlus	mov.u32 $r1, **0x00000000;**

2. This is thread block id in x dimension. The naming convention is modified so that it's similar to PTX.

SASS	S2R R0, **SR_CTAid_X**;
PTXPlus	cvt.u32.u32 $r0, **%ctaid.x**;

3. This is thread ID in x dimension. The naming convention is modified so that it's similar to PTX.

SASS	S2R R2, **SR_Tid_X**;
PTXPlus	cvt.u32.u32 $r2, **%tid.x**;

4. This is PBK instruction. It is generally used in conjunction with "BRK" instruction. This instruction is used to support "break" and "continue" keywords of C/CUDA language. Both parser were modifier to add support for this instruction.

SASS	**PBK** 0x150;
PTXPlus	**pbk.u32** 0x00000150;

5. This is global load for Fermi architecture. The parser was modified to identify different types of LD instruction.

SASS	**LD** R0, [R0];
PTXPlus	**ld.global.u32** $r0, $r0;

6. RZ stands for "register zero." Since its value is always zero, parser is modified to reflect that.

SASS	MOV R0, **RZ**;
PTXPlus	mov.u32 $r0, **0x00**;

7. This is SSY instruction and its usefulness has been described in previous sections.

SASS	**SSY** 0xf8;
PTXPlus	**ssy.u32** 0x000000f8;

8. Predication (@P0 or !@P0) semantic support has been added. Also ".S" flag suggests to pop the stack. The meaning of CC.T is not known but it doesn't seem to affect anything for functional simulation based on benchmarks used. NOP stands for no operation.

SASS	**@**P0 NOP.S CC.T;
PTXPlus	**@$p0 nop.s**;

9. This is BRK instruction and its usefulness is explained with PBK instruction information.

SASS	**BRK;**
PTXPlus	**brk.u32;**

10. This is global store for Fermi architecture. The parser was modified to identify different types of LD instruction.

SASS	**ST** [R2], R0;
PTXPlus	**st.global.u32** $r2, $r0;

11. ISCADD" is shift and add instruction. This is new instruction with Fermi. This required changes at all levels from parser to functional simulation. Also c [0x0][0x28] is kernel parameter and it's representation has been changed in PTXPlus as shown in the instruction.

SASS	**ISCADD** R5, R4, **c [0x0] [0x28]**, 0x2;
PTXPlus	**iscadd.u32** $r5, $r4, **k [0x0018]**, 0x00000002;

12. “STS” is store to shared memory. The addressing mode and instruction opcode both were added to support this type of instruction.

SASS	**STS [R12+0x180], R11;**
PTXPlus	**st.shared.u32 $r12, 0x00000180, $r11;**

13. “LDS” is Load from shared memory. The addressing mode and instruction opcode both were added to support this type of instruction.

SASS	**LDS R12, [R9+0x2e80];**
PTXPlus	**ld.shared.u32 $r12, $r9, 0x00002e80;**

14. “.U” flag indicates is UNI branch. Parser were modified to identify the base modifier as UNI branch.

SASS	@P0 **BRA.U** 0x860;
PTXPlus	@$p0 **bra.uni.u32** l0x00000860;

15. “c [0x2] [0x4] is constant register. This required changes in the parser for instruction as well as parsing the constant values from “elf” file.

SASS	MOV R10, **c [0x2] [0x4];**
PTXPlus	mov.u32 $r10, **constant2[0x0004]**;

16. LDU is LD Uniform. Support was required from parser to functional simulation.

SASS	**LDU R10, [R5];**
PTXPlus	**ldu.global.u32 $r10, $r5, 0x00;**

17. LD_LDU is two load instructions mixed together. Support was required from parser to functional simulation. Also addressing mode support was required.

SASS	LD_LDU.32.32 R16, R15, [R17], [R5+0x4];
PTXPlus	ldu.global.u32 $r16, $r15, $r17, $r5, 0x00000004;

18. F2F is float to float conversion instruction. Modulus operation support has been added and it required changes from parser till functional simulation.

SASS	F2F.F32.F64 R6, **|R8|;**
PTXPlus	cvt.f32.f64 $r6, **\|$r8|**;

19. TRUNC base modifier support was added from parser till functional simulation.

SASS	**F2F.TRUNC** R8, R8;
PTXPlus	**cvt.rz.u32** $r8, $r8;

20. Constants such as 0x3f000 appearing floating instruction (FADD, FMUL and FSETP) were extended to identify correct float value.

SASS	@P0 FMUL R8, R8, **0x3f000**;
PTXPlus	@$p0 mul.f32 $r8, $r8, **0x3f000000**;

21. This is range reduction operator for exponent of two operation. Support was added from parser to functional simulator to support it.

SASS	**RRO.EX2** R7, R8;
PTXPlus	**rro.f32** $r7, $r8;

22. MUFU is multi-function operator and EX2 stands for exponent of two. MUFU keyword support was added in SASS parser.

SASS	**MUFU.EX2** R9, R7;
PTXPlus	**ex2.f32** $r9, $r7;

23. MUFU is multi-function operator and RCP stands for reciprocal. MUFU keyword support was added in SASS parser.

SASS	**MUFU.RCP** R7, R7;
PTXPlus	**rcp.f32** $r7, $r7;

24. c [0x10] [0xc] is a constant and support was added to parse this correctly.

SASS	FFMA R7, R9, R7, **c [0x10] [0xc]**;
PTXPlus	fma.f32 $r7, $r9, $r7, **constant1_Z17executeFirstLayerPfS_S_[0x000c]**;

25. SEL keyword support was added to SASS parser. This instruction selects value from two register based on P0.

SASS	**SEL R8, R8, c [0x10] [0x10], P0;**
PTXPlus	**selp.u32 $r8, $r8, constant1_Z17executeFirstLayerPfS_S_[0x0010], $p0;**

26. This is 64 bit load and it stores values in two 32 bit register. Also it's loading values from constant and its load from constant memory. Changes were done in parser and functional simulator.

SASS	**LDC.64 R2, c [0x2] [R5+0x8];**
PTXPlus	**mov.const.u32.doublereg $r2, constant2[$r5+0x0008];**

27. The changes were done to support this type of addressing mode.

SASS	**LD_LDU.32.32 R15, R12, [R11+0x2a4], [R0+0x24];**
PTXPlus	**ldu.global.u32 $r15, $r12, $r11, 0x000002a4, $r0, 0x00000024;**

28. This is compare and select instruction. The value is third source operand is compared with zero and based on the outcome of those two, source 1 or source 2 is stored in destination. The instruction support required modification from SASS parser to functional simulator.

SASS	**ICMP.LT R6, R8, RZ, R4;**
PTXPlus	**icmp.lt.s32 $r6, $r8, 0x00, $r4;**

29. This is min or max instruction. Third source operands determines min or max operation. (!pt) is for max operation and (pt) is for min operation. This instruction

support required modification from SASS parser to functional simulator.

SASS	**IMNMX R5, R5, R6, !pt;**
PTXPlus	**imnmx.u32 $r5, $r5, $r6, !pt;**

30. The min flag indicates minimum operation. This instruction selects min from source operand 1 and source operand 2 and adds it to source operand 3. This instruction support required modification from SASS parser to functional simulator.

SASS	**VADD.S32.S8.MIN R7, R0, 0x1, R7;**
PTXPlus	**vadd.min.s32.s8 $r7, $r0, 0x00000001, $r7;**

31. This is predicate set instruction. It uses predicate registers as source operands and sets the destination predicate. AND flag specifies the operation. This instruction support required modification from SASS parser to functional simulator. In this case P1 = (P2 & P1 & P0).

SASS	**PSETP.AND.AND P1, pt, P2, P1, P0;**
PTXPlus	**psetp.and.and.s32 $p1, pt, $p2, $p1, $p0;**

32. This is min operation with accumulation. Min values is selected from source operand 1 and source operand 2 and added to source operand 3. This instruction support required modification from SASS parser to functional simulator. For example, R11 = min (R10, R8) + R11.

SASS	**VMNMX.ACC R11, R10, R8, R11;**
PTXPlus	**vmnmx.acc.u32 $r11, $r10, $r8, $r11;**

5 Results

5.1 Analysis and Results:

The comparison between Tesla and Fermi (Native ISA) provides quite interesting insight into the difference between them. The number of instructions that are executed with Fermi are 26 % less than number of instructions that are executed on Tesla. Figure 13 shows the dynamic instruction count difference between Tesla and Fermi for different benchmarks.

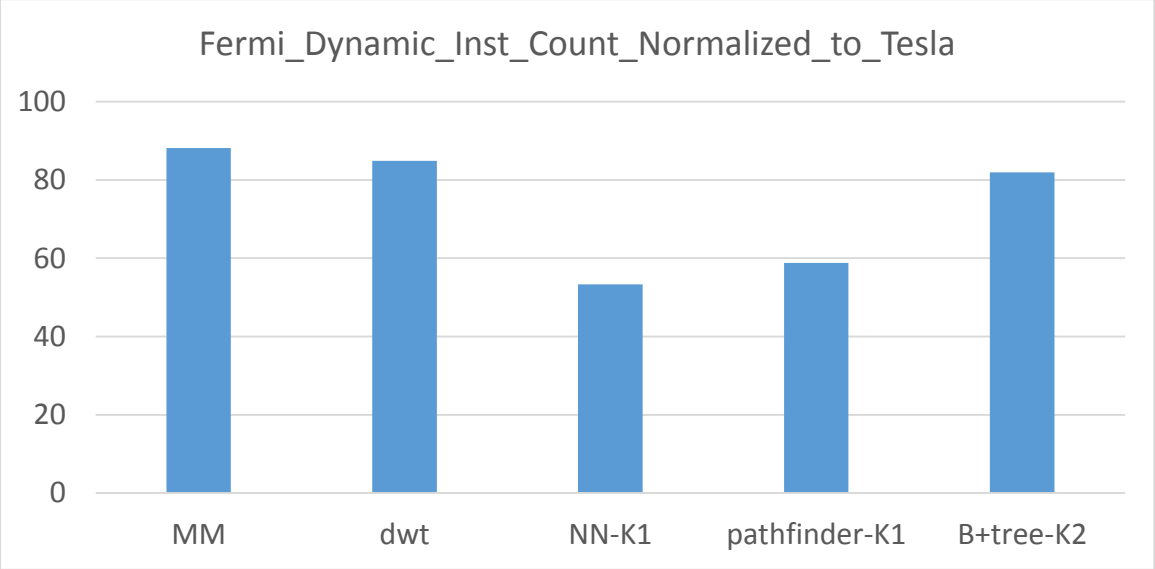

Figure 13 Dynamic Instruction Count

This section examines each benchmark in detail to understand the difference between Tesla and Fermi architecture.

5.1.1 MM (matrix multiplication):

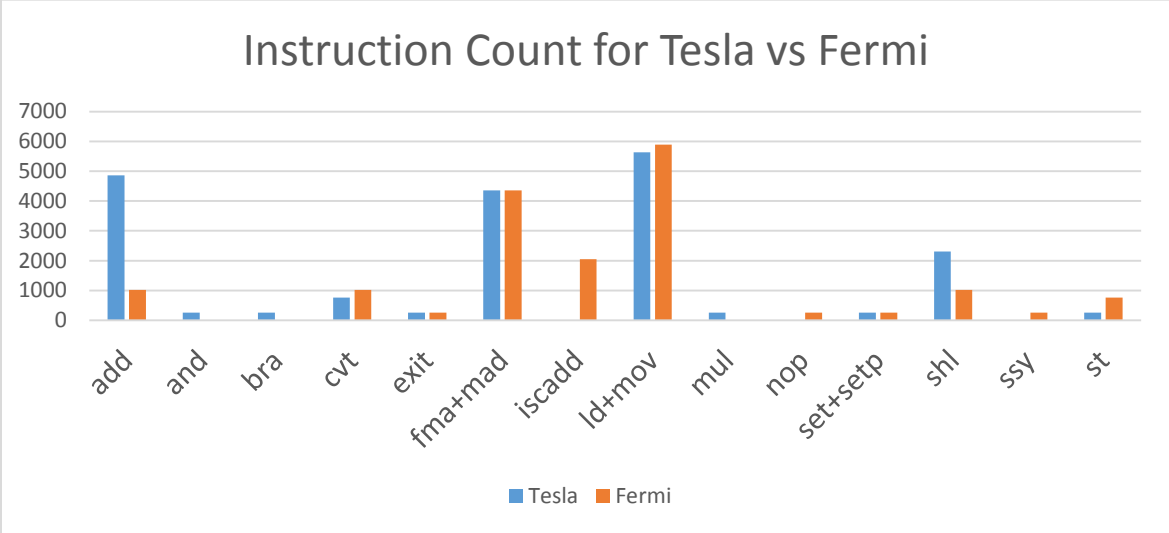

Figure 14 Dynamic Instruction Count for Tesla vs Fermi (MM)

Table 7 Total Instruction Count (MM)

Tesla_gpu_sim_inst	Fermi_gpu_sim_inst
19456	17152

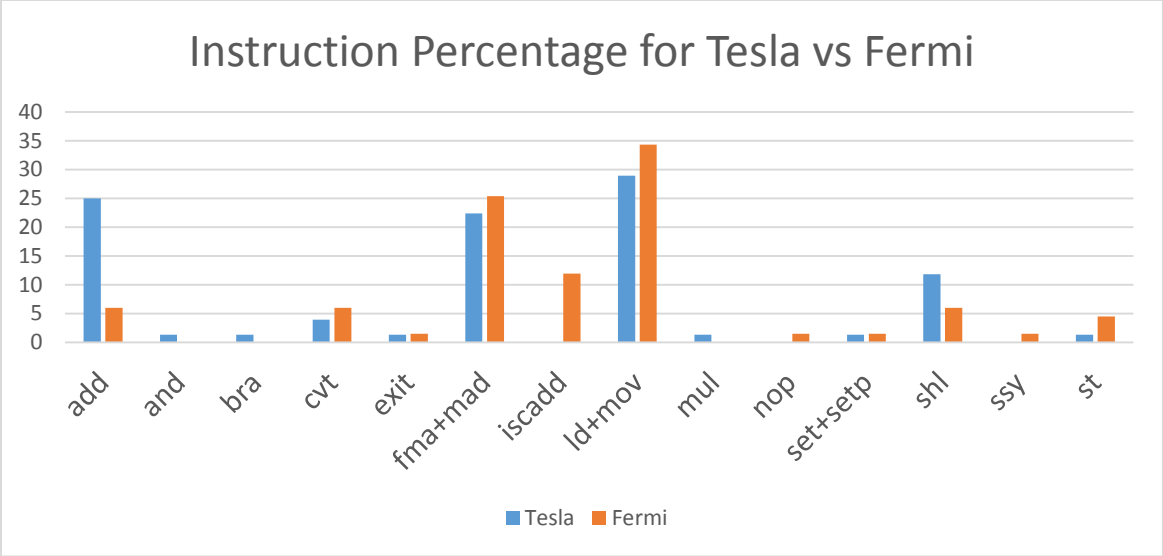

Figure 15 Instruction Percentage for Tesla vs Fermi (MM)

Table 7 shows the absolute instruction count for Tesla and Fermi architecture for matrix multiplication benchmark. Figure 14 shows the instruction wise distribution and Figure 15 shows the percentage of each instruction. “ADD” and “SHL” (shift left) instruction contributes major part for Tesla architecture. Fermi architecture uses “ISCADD” (shift and add) instruction to do the same. Fermi architecture just uses one instruction (ISCADD) to do the work that takes two instructions (SHL and ADD) in Tesla architecture. Table 8 shows code section for CUDA Code, Tesla Native ISA program and Fermi Native ISA program. Fermi and Tesla architecture difference is shown here via C/CUDA code section bold marked in Table 8. (“A[a + WA * ty + tx]”)**.** Calculation of this code section takes different number of instruction in Tesla native code and Fermi native code which is also shown in the right side in the Table 8. It takes five instruction to do the calculation in Tesla whereas it only takes three instructions to do the same in Fermi (Relevant instructions highlighted in bold). Overall it takes 11.8 % lesser instructions in Fermi architecture to do this computation. This optimization can be used during address computation of any array.

Table 8 Code analysis (MM)

C/CUDA Code:

```
bx = blockIdx.x;
by = blockIdx.y;
tx = threadIdx.x;
ty = threadIdx.y;
aBegin = WA * BLOCK_SIZE * by;
aEnd   = aBegin + WA - 1;
aStep  = BLOCK_SIZE;
bBegin = BLOCK_SIZE * bx;
bStep  = BLOCK_SIZE * WB;
Csub = 0;
    for (int a = aBegin, b = bBegin;
            a <= aEnd;
            a += aStep, b += bStep)
      {
        __shared__ float
As[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float
Bs[BLOCK_SIZE][BLOCK_SIZE];

        AS(ty, tx) = A[a + WA * ty + tx];
        BS(ty, tx) = B[b + WB * ty + tx];
        __syncthreads();
        ……
      }
```

Tesla:

SHL R1, R2, 0x4; R1= ty * WA (or WB);
IADD32 R5, R2, R0; R5 = ty + bx;
IADD32 R4, R1, R4; R4 = aBegin + ty * WA;
SHL R5, R5, 0x4; //R5 = R5 * 16;
IADD32 R4, R3, R4; //R4 = aBegin + ty * WA + tx;
IADD32 R5, R3, R5; //R5 = bBegin + ty * WB + tx;
SHL R4, R4, 0x2; R4= R4 * 4;
IADD R1, R3, R1;
SHL R5, R5, 0x2; R5 = (b+ WB*ty + tx) * 4;
IADD R4, g [0x4], R4; Address for A[]
GLD.U32 R4, global14 [R4]; Load R4 = A[]
R2A A1, R1, 0x2;
IADD R1, g [0x5], R5; R1 = address for B[]

Fermi:

ISCADD R4, R3, R4, 0x4; // = ty* WA + aBegin ;
IADD R5, R2, R3; // R5 = bx + ty;
IMAD.U32.U32 RZ, R1, RZ, RZ; //Bubble
IADD R4, R12, R4; //R4 = tx + a (aBegin) + WA * ty;
ISCADD R6, R5, R12, 0x4; //R6 = bx* BLOCK_SIZE (bBegin) + ty* WB + tx;
ISCADD R5, R4, c [0x0] [0x20], 0x2; // R5 = A[];
ISCADD R4, R6, c [0x0] [0x24], 0x2; //R4 = B[];

5.1.2 DWT benchmark:

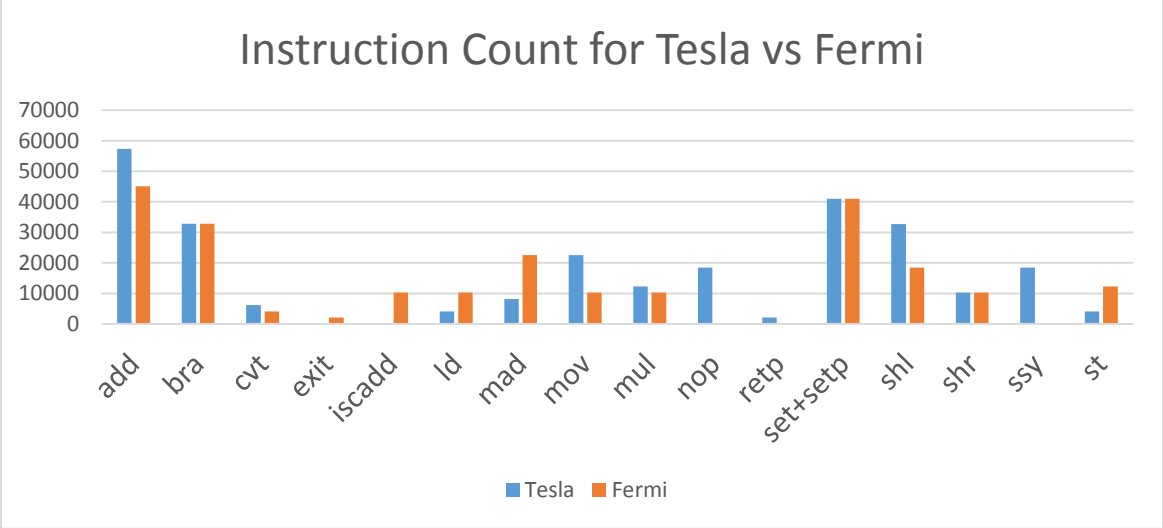

Figure 16 Dynamic Instruction Count for Tesla vs Fermi (DWT)

Table 9 Total Instruction Count (DWT)

Tesla_gpu_sim_inst	**Fermi_gpu_sim_inst**
270248	229312

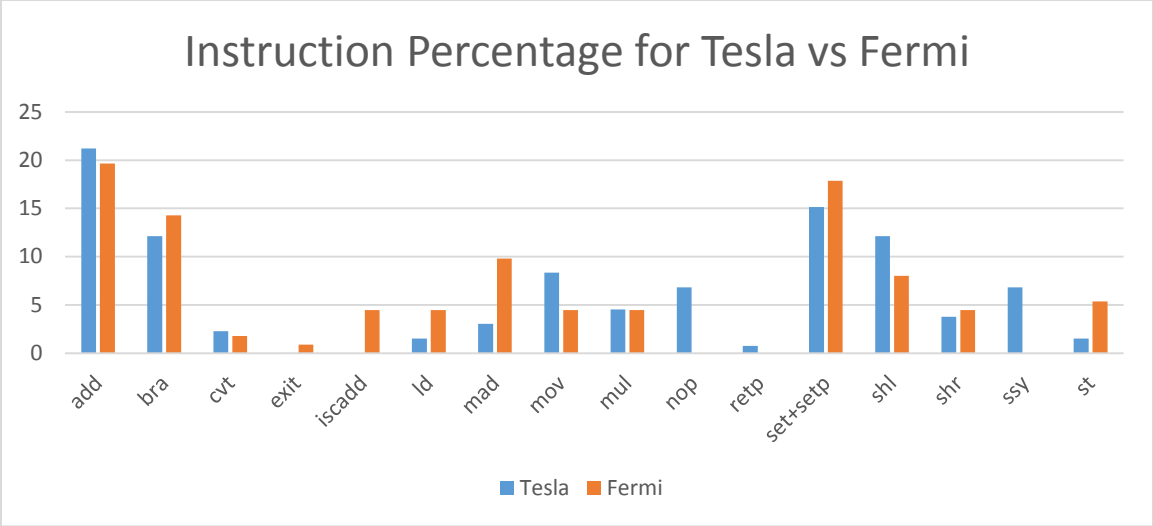

Figure 17 Instruction Percentage for Tesla vs Fermi (DWT)

Table 10 Code Analysis (DWT)

For (……..)
{
if(tid < num_threads)
{
…….
}
}
	Tesla Code:
	/*0160*/ SSY 0x230;
	/*0168*/ BRA C0.NE, 0x230;
	……
	/*0198*/ SHR R8, R0, 0x4;
	……
	/*0230*/ NOP.S;
	
	Fermi Code:
	/*0150*/ ISETP.GE.U32.AND P0, pt, R2, R5, pt;
	/*0158*/ @P0 BRA.U 0x200;
	/*0160*/ @!P0 IADD R9, R3, R4;
	……
	/*01f8*/ @!P0 STS [R10], R8;

The instruction count difference between Fermi and Tesla architecture for this benchmark comes from ISCADD instruction and predication based uniform branching. The sample program of predication based uniform branching is shown in Table 10. To implement "IF-THEN" conditional block, Tesla architecture uses stack based mechanism. Stack based mechanism relies on pushing and popping of tokens which requires additional instruction (SSY and NOP.S shown in Tesla section of Table 10). In case of Fermi architecture, it relies on predication based uniform branching which doesn't require this additional instruction. In this benchmark, this particular "IT-THEN" conditional block is in "FOR" loop and that causes instruction count difference between these two architecture.

5.1.3 NN Benchmark- Kernel1:

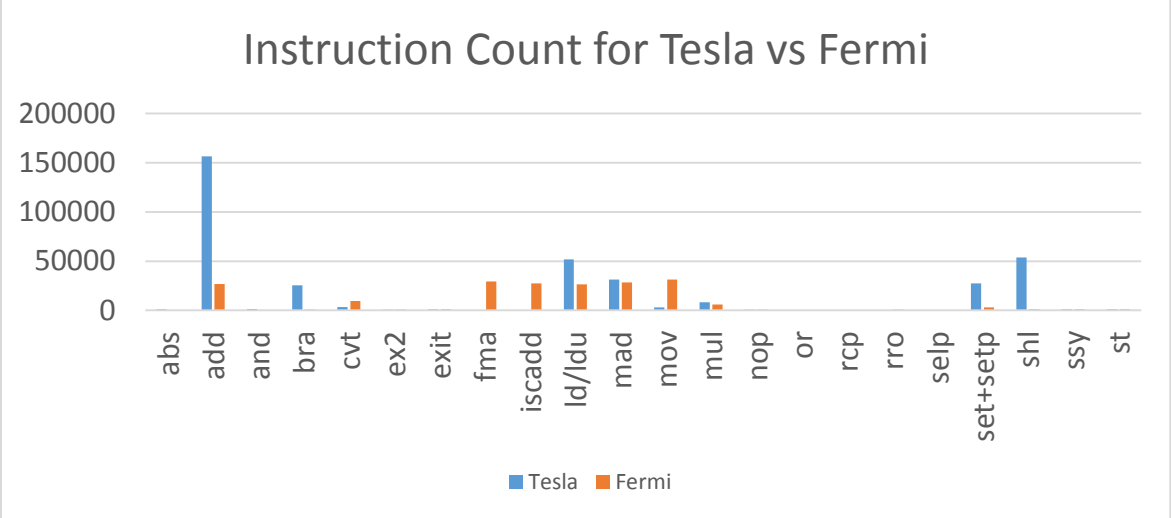

Figure 18 Dynamic Instruction Count for Tesla vs Fermi (NN)

Table 11 Total Instruction Count (NN)

Tesla_gpu_sim_inst	Fermi_gpu_sim_inst
367920	196180

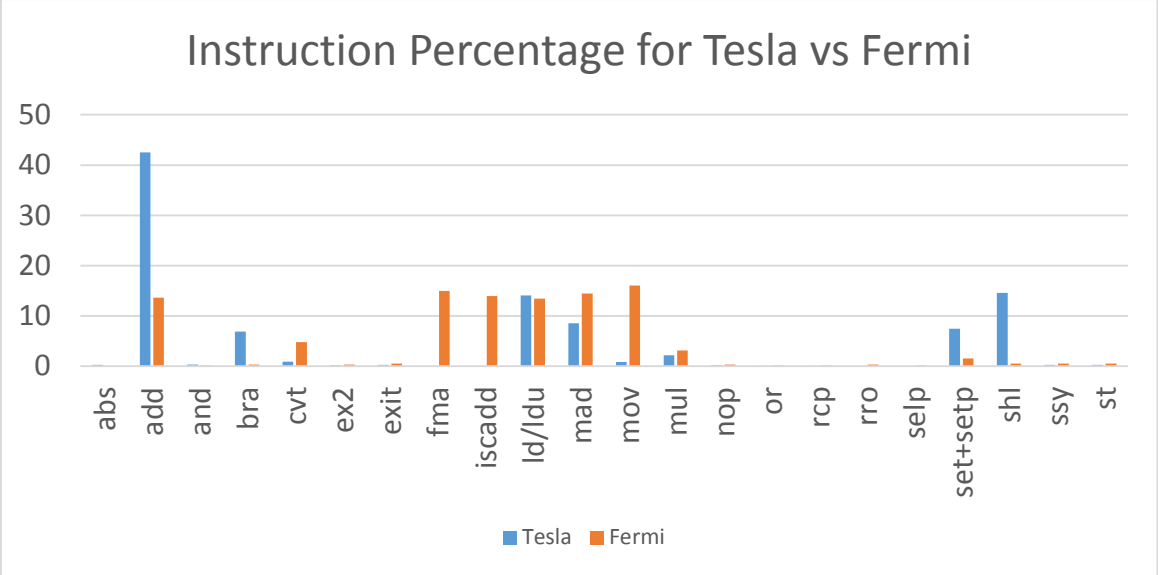

Figure 19 Instruction Percentage for Tesla vs Fermi (NN)

Table 12 Code Analysis

Fermi:	Tesla:
LD_LDU.32.32 R16, R15, [R17], [R5+0x4];	GLD.U32 R7, global14 [R9]; GLD.U32 R8, global14 [R8];

Apart from ISCADD instruction, LD_LDU, 32 bit IMAD and Loop Unrolling are responsible for the instruction count difference between Tesla and Fermi for this benchmark. Table 12 shows the example of LD_LDU instruction. It takes only one instruction to load two different register in Fermi compared to two instructions in Tesla. IMAD example will be covered in the later benchmark. Also because of loop unrolling, Tesla has more ISETP and BRA instruction compared to Fermi.

5.1.4 Pathfinder Benchmark- Kernel1:

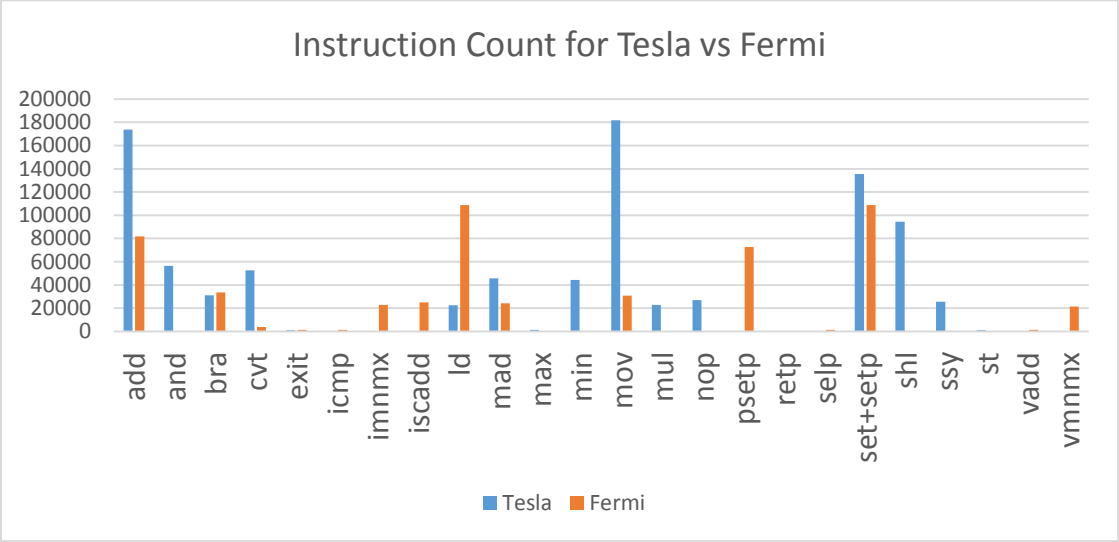

Figure 20 Dynamic Instruction Count for Tesla vs Fermi (pathfinder)

Table 13 Total instruction count

Tesla_gpu_sim_inst	Fermi_gpu_sim_inst
916460	538600

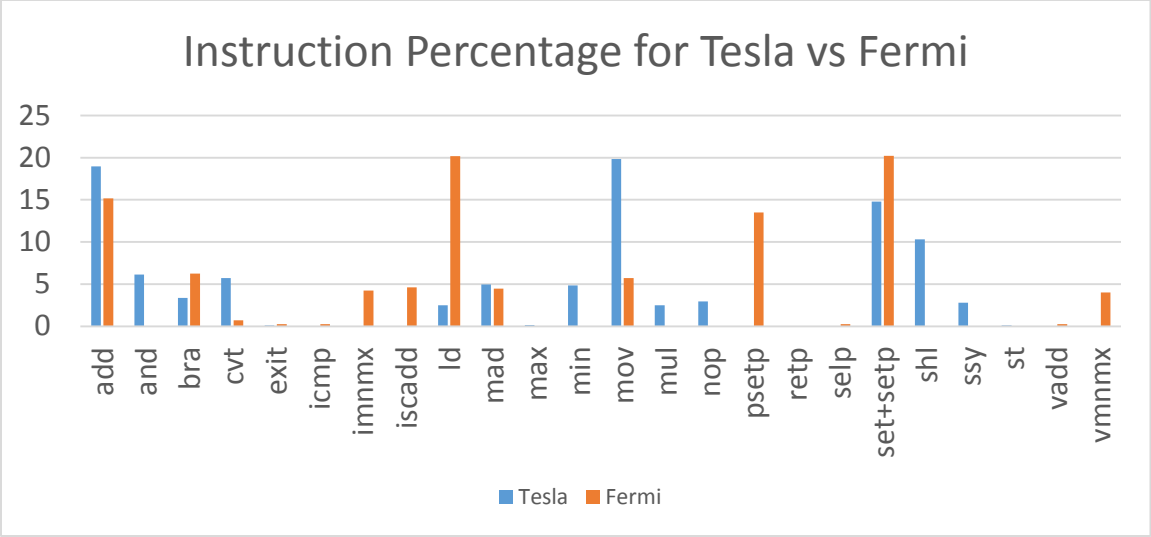

Figure 21 Instruction Percentage for Tesla vs Fermi (pathfinder)

Table 14 Code analysis 1

C/CUDA Code: If(IN_RANGE(tx, i+1, BLOCK_SIZE-i-2) && isValid) // (i+1 < tx < BLOCK_SIZE-i-2) && isValid	Tesla: /*0158*/ IADD32I R0, R1, 0x1; (i+1) /*0160*/ IADD32I R7, -R1, 0xfe; R7 = BLOCK_SIZE-i-2; /*0168*/ ISET.S32 R2, R0, R3, LE; (i+1) < tx /*0170*/ ISET.S32 R7, R3, R7, LE; (tx < BLOCK_SIZE-i-2) **/*0178*/ LOP.AND R2, R2, R7; IN_RANGE(,,)** **/*0180*/ I2I.S32.S32 R2, -R2; Invert** **/*0188*/ LOP.AND.C0 o [0x7f], R5, R2; (isValid & IN_RANGE)** Fermi: /*0128*/ IADD R8, -R3, 0xfe; R8 = BLOCK_SIZE-i-2; /*0130*/ IADD R6, R3, 0x1; R6 = i+1; /*0138*/ ISETP.LE.AND P1, pt, R0, R8, pt; if (tx <= BLOCK_SIZE-i-2) /*0140*/ ISETP.LE.AND P0, pt, R6, R0, pt;if (i+1 <= tx) /*0148*/ **PSETP.AND.AND P1, pt, P2, P1, P0; P1 = (P2&P1) & P0** /*0150*/ PSETP.AND.AND P0, pt, pt, pt, !pt; P0 = pt & pt & !pt; //Resetting “Computed”

Table 15 Code analysis 2

```
if( ……..) {
            computed = true;
            int lft = prev[W];
            int up = prev[tx];
            int right = prev[E];
            int shortest = MIN(left, up);
            shortest = MIN(shortest, right);
            int index = cols*(startStep+i)+xidx;
            result[tx] = shortest + gpuWall[index];
}
```

Fermi:

```
/*0158*/     @!P1 BRA.U 0x1c0;
 /*0160*/    @P1 IADD R8, R3, c [0x0] [0x38];
(startStep+i)
 /*0168*/    @P1 SHL R12, R0, 0x2; tx*4
 /*0170*/    @P1 PSETP.AND.AND P0, pt, pt, pt, pt; P0
= pt && pt && pt;
 /*0178*/    @P1 IMAD R8, R8, c [0x0] [0x30], R2;
index = cols*(startStep+i)+xidx;
 /*0180*/    @P1 LDS R9, [R12];  up = prev[tx];
 /*0188*/    @P1 ISCADD R10, R8, c [0x0] [0x24], 0x2;
r10 = gpuWall[index];
 /*0190*/    @P1 LDS R8, [R5]; left = prev[W];
 /*0198*/    @P1 LD R11, [R10]; gpuWall[index]
loading
 /*01a0*/    @P1 IMNMX R8, R9, R8, pt;  R8..shortest
= min (left, up);
 /*01a8*/    @P1 LDS R10, [R7]; Loading right = prev[E]
 /*01b0*/    @P1 VMNMX.ACC R11, R10, R8, R11;
R11 = min(shortest-R8,right-R10)+gpuWall[index];
 /*01b8*/    @P1 STS [R12+0x400], R11; result[tx] =
shortest+ gpuWall[index];
```

Tesla:

```
/*0190*/    SSY 0x228; (SSY before IF-THEN)
 /*0198*/    MOV R2, R124;
 /*01a0*/    BRA C0.EQ, 0x228; (Jump if out)
 /*01a8*/    IADD32 R7, g [0xa], R1; (startStep+i)
 /*01ac*/    MOV32 R2, g [0x8]; R2 = cols
 /*01b0*/    IMUL.U16.U16 R8, R2L, R7H; cols
*(startstep+i)
 /*01b8*/    IMAD.U16 R8, R2H, R7L, R8;
 /*01c0*/    SHL R8, R8, 0x10;
 /*01c8*/    IMAD.U16 R2, R2L, R7L, R8;
 /*01d0*/    IADD R7, R2, R4;
 /*01d8*/    R2A A3, R3, 0x2;
 /*01e0*/    G2R.U32 R2, g [A1+0xc].U32;
 /*01e8*/    SHL R7, R7, 0x2;
 /*01f0*/    IMIN.S32 R8, g [A3+0xc], R2;
 /*01f8*/    IADD R2, g [0x5], R7;
 /*0200*/    IMIN.S32 R7, g [A2+0xc], R8; shortest
 /*0208*/    GLD.U32 R2, global14 [R2];
gpuWall[index}
 /*0210*/    IADD R2, R2, R7;
 /*0218*/    R2G.U32.U32 g [A3+0x10c], R2;
 /*0220*/    MVI R2, 0x1;
 /*0228*/    NOP.S;
```

ISCADD, PSETP and VMNMX are the major instructions that are responsible for the instruction count difference between Tesla and Fermi architecture for pathfinder benchmark. As shown in Table 14, it takes three instructions to compute the “IF” condition for Tesla architecture whereas it only takes one instruction to do the same in Fermi. Table 15 shows example of

VMNMX instruction. Tesla architecture takes two instruction to compute minimum and do the addition whereas Fermi only takes one.

5.1.5 B+tree benchmark:

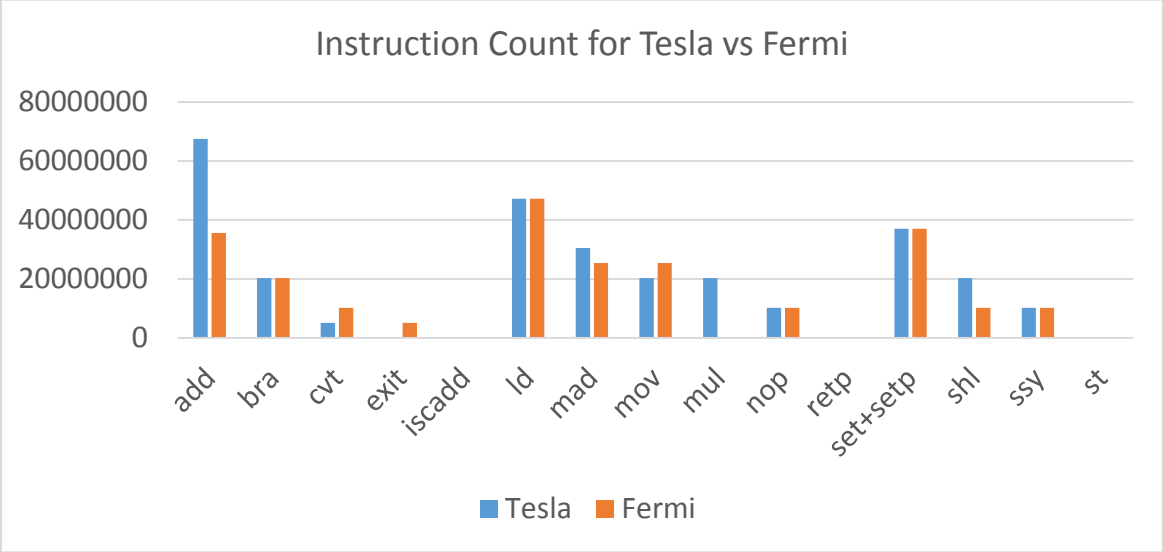

Figure 22 Dynamic Instruction Count for Tesla vs Fermi (b+tree)

Table 16 Total Instruction count

Tesla_gpu_sim_inst	Fermi_gpu_sim_inst
289028114	236755316

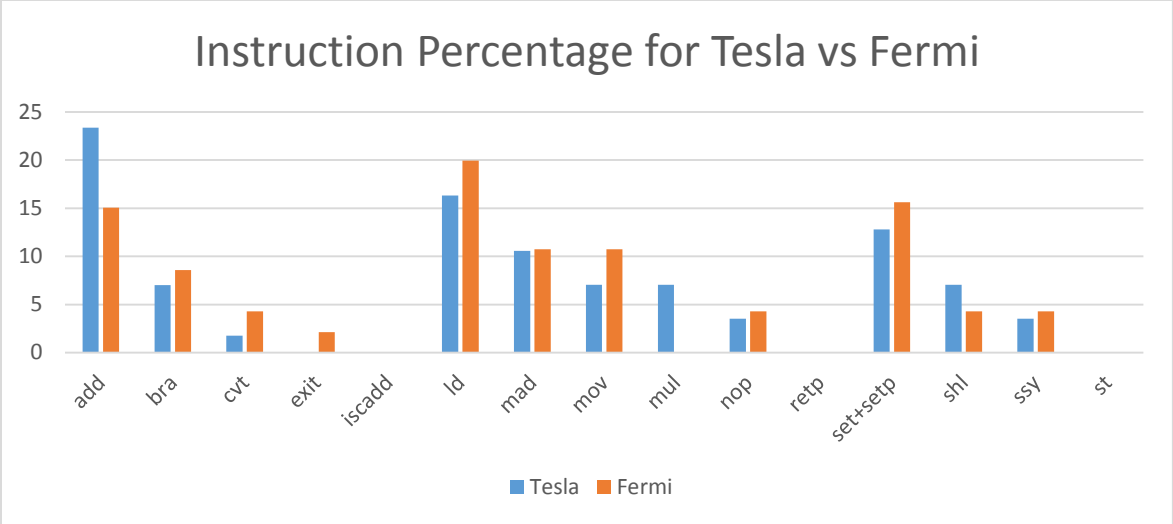

Figure 23 Instruction Percentage for Tesla vs Fermi (b+tree)

Table 17 Code analysis 1

Tesla:	Fermi:
/*0048*/ MVI R6, 0xff4; /*0050*/ IMUL.U16.U16 R7, R5L, R6H; /*0058*/ IMAD.U16 R7, R5H, R6L, R7; /*0060*/ SHL R7, R7, 0x10; /*0068*/ IMAD.U16 R5, R5L, R6L, R7;	IMAD.U32.U32 R6, R6, 0xff4, R7;

Table 18 Code analysis 2

Tesla:	Fermi:
/*0070*/ IADD32 R6, g [0x5], R5; /*0078*/ IADD32I R7, R6, 0x7f8; /***irrelavant instruction***/ /*0088*/ GLD.U32 R7, global14 [R7];	/*0080*/ IADD R8, R6, R3; /*0088*/ LD R6, [R8+0x7f8];

B+tree benchmark shows 18% lesser instruction count compared to Fermi architecture. The major instruction count difference comes from 32 bit IMAD instruction support in Fermi architecture. Table 17 shows example of this. It takes one instruction in Fermi architecture to do the multiplication and addition whereas it takes five instruction to do the same in Tesla architecture. Table 18 shows another example from this benchmark. The semantic of LD instruction in Fermi architecture allows lesser instruction count in Fermi compared to Tesla.

Table 19 shows the percentage of saving in instruction count for above benchmarks. NQ indicates non-quantifiable numbers for those benchmarks but they are responsible for instruction count reduction. The number in the table indicates the reduction in instruction percentage in Fermi architecture compared to Tesla due to instruction fusion, uniform branch etc.

Table 19 % Savings in instruction count for different benchmarks

% saving in inst count	ISCADD	IMAD	BRA.U/NOP.S	LD_LDU	VMNMX	PSETP
MM	10.53	-	-	-	-	-
Dwt	3.79	-	13.64	-	-	-
NN-K1	7.44	NQ	-	7.17	-	-
pathfinder-K1	2.72	7.88	2.79	-	2.35	11.90
B+tree-K2	-	14.74	3.93	NQ	-	-

NQ: Not quantifiable

6 Conclusion

The work done in this thesis correlates the ISA impact between Tesla and Fermi architecture of NVIDIA GPUs. There are many improvements in Fermi architecture compared to Tesla from ISA perspective which results in 26 % lesser dynamic instruction count in Fermi architecture. The improvements in dynamic instruction count comes from Instruction fusion (merging two instructions into one for example ISCADD, VMNMX, PSETP, LD_LDU), 32 bit multiplication (IMAD) and predication based uniform branching. Instruction fusion doesn't require any significant changes in hardware architecture (register file design) but it saves energy because of just using one instruction to do the work instead of two. Tesla architecture itself have three operand instructions so it doesn't require significant changes to register file design. Predication based uniform branching is effective as it can skip IF-THEN sections based on runtime behavior of threads within a warp. These mechanism can be implemented in SIMD architecture and it improves performance without impacting hardware design significantly. As a future work, even more benchmarks can be tested to cover entire Fermi ISA. NVIDIA has also come up with Kepler architecture and it will be interesting to understand ISA evolution from Fermi to Kepler.

REFERENCES

[1] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, Tor M. Aamodt, Analyzing CUDA Workloads Using a Detailed GPU Simulator, In proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 163-174, Boston, MA, April 26-28, 2009.

[2] GPGPUSim Tutorial. [Online]. Available: http://gpgpu-sim.org/manual/index.php5/GPGPU-Sim_3.x_Manual

[3] Nickolls et al. United States Patent 2011/0072249: UNANIMOUS BRANCH INSTRUCTIONS IN A PARALLEL THREAD PROCESSOR (Assignee NVIDIA Corp.), March 2011

[4] Coon et al. United States Patent US007877585Bl: STRUCTURED PROGRAMMING CONTROL FLOW IN A SIMD ARCHITECTURE. Assignee NVIDIA Corporation, Santa Clara, CA (US)

[5] Coon et al. United States Patent US008312254B2: INDIRECT FUNCTION CALL INSTRUCTIONS IN A SYNCHRONOUS PARALLEL THREAD PROCESSOR. Assignee NVIDIA Corporation, Santa Clara, CA (US)

[6] Coon et al. United States Patent US007761697Bl: PROCESSING AN INDIRECT BRANCH INSTRUCTION IN A SIMD ARCHITECTURE. Assignee NVIDIA Corporation, Santa Clara, CA (US)

[7] PARALLEL THREAD EXECUTION ISA VERSION 3.1.

[8] Nicholas Wilt. The CUDA Handbook: A Comprehensive Guide to GPU Programming.

[9] https://code.google.com/p/asfermi/