TECHNIQUES FOR MANAGING IRREGULAR CONTROL FLOW ON GPUS

by

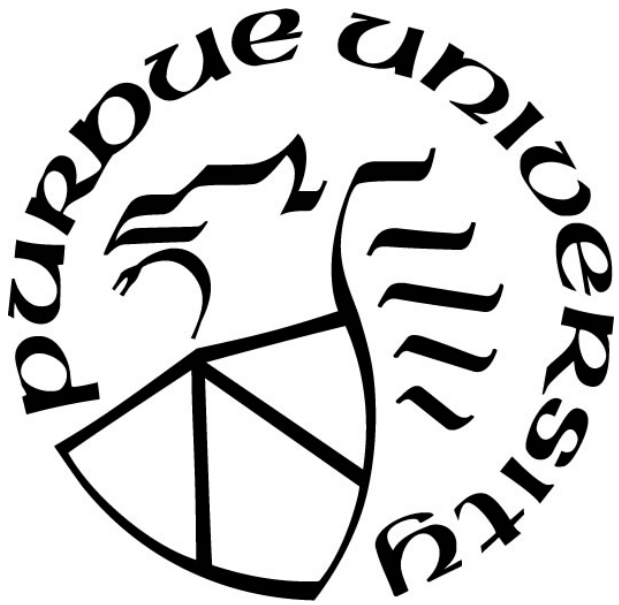
Jad Hbeika

A Thesis

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy



School of Electrical and Computer Engineering

West Lafayette, Indiana

May 2019

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Milind Kulkarni, Chair
School of Electrical and Computer Engineering

Dr. Timothy Rogers
School of Electrical and Computer Engineering

Dr. Samuel Midkiff
School of Electrical and Computer Engineering

Dr. Arun Prakash
School of Civil Engineering

Approved by:

Dr. Pedro Irazoqui
Head of the Graduate Program

I dedicate this work to my dad, Georges, my mom, Wadad, my brother, Imad, and my sister, Maya.

ACKNOWLEDGMENTS

I would like to express my great appreciation to professor Milind Kulkarni, my advisor, for his valuable guidance, continuous encouragement, and patience. I could have not imagined having a better advisor …

I would love also to thank the members of my PhD committee, Professor Samuel Midkiff, Professor Arun Prakash, and Professor Timothy Rogers, for their insightful comments and questions.

I thank my fellow lab mates for the simulating discussions and for all the fun we have had in the past years

This work was supported in part by the U.S. Department of Energy's (DOE) Office of Science, Office of Advanced Scientific Computing Research, Under DOE Early Career Award DE-SC0010295 and DOE DE-FC02-12ER26104

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Author: Hbeika, Jad. PhD
Institution: Purdue University
Degree Received: May 2019
Title: Techniques for Managing Irregular Control Flow on GPUs
Committee Chair: Milind Kulkarni

GPGPU is a highly multithreaded throughput architecture that can deliver high speed-up for regular applications while remaining energy efficient. In recent years, there has been much focus on tuning irregular applications and/or the GPU architecture to achieve similar benefits for irregular applications as well as efforts to extract data parallelism from task parallel applications. In this work we tackle both problems.

The first part of this work tackles the problem of Control divergence in GPUs. GPGPUs' SIMT execution model is ineffective for workloads with irregular control-flow because GPGPUs serialize the execution of divergent paths which lead to thread-level parallelism (TLP) loss. Previous works focused on creating new warps based on the control path threads follow, or created different warps for the different paths, or ran multiple narrower warps in parallel. While all previous solutions showed speedup for irregular workloads, they imposed some performance loss on regular workloads. In this work we propose a more fine-grained approach to exploit *intra-warp* convergence: rather than threads executing the same code path, *opcode-convergent threads* execute the same instruction, but with potentially different operands. Based on this new definition we find that divergent control blocks within a warp exhibit substantial opcode convergence. We build a compiler that analyzes divergent blocks and identifies the common streams of opcodes. We modify the GPU architecture so that these common instructions are executed as convergent instructions. Using software simulation, we achieve a 17% speedup over baseline GPGPU for irregular workloads and do not incur any performance loss on regular workloads.

In the second part we suggest techniques for extracting data parallelism from irregular, task parallel applications in order to take advantage of the massive parallelism provided by the GPU. Our technique involves dividing each task into multiple sub-tasks

each performing less work and touching a smaller memory footprint. Our framework performs a locality-aware scheduling that works on minimizing the memory footprint of each warp (a set of threads performing in lock-step). We evaluate our framework with 3 task-parallel benchmarks and show that we can achieve significant speedups over optimized GPU code.

1. INTRODUCTION

1.1 The Basics

GPUs were initially developed to enable real-time rendering in video games. While graphic acceleration is still among its main purpose, GPUs proved to be efficient in different computation domains and today we can find GPUs in many systems like: supercomputers, datacenters, laptops and even smart phones. The use of GPUs as general purpose computation platforms started around the year 2006 with the breakdown of the Dennard scaling.

1.2 Breakdown of the Dennard Scaling and its Effect

While improvements in architecture, microarchitecture, compilers and algorithms all led to computer systems increased performance over the last decades, the major improvements can be contributed to transistor scaling which allowed drastic raise in clock frequencies.

Dennard (1974) [1] concluded that voltage and current should be proportional to dimensions of a transistor. As transistors were scaling down in size the reduction of voltage allowed running circuits at higher frequency while keeping the dynamic power the same.

$$\text{Power} = \alpha C F V^2$$

C: capacitance

F: frequency

V: voltage

Alpha: percent rime switched

Dennard scaling ignored the leakage current and threshold voltage which could not be reduced further as transistor size were scaling further down. This created a “Power wall” that limited CPUs clock frequencies to around 4 GHZ around 2006.

Improving performance beyond this point relies mainly on new efficient architectures and the use of hardware accelerators. Since the “power wall” Multicore systems as well as the use of GPUs outside of the graphics world became hot research topics.

1.3 From ILP to TLP

For decades the main approach in exploiting parallelism was through extracting instruction level parallelism (ILP) from a single thread. Extracting ILP requires complex scheduling logic, branch prediction, larger caches etc. and aims at speeding up the execution of a single thread.

Over the recent years the attention shifted toward exploiting thread level parallelism (TLP) due to the diminished returns of further exploiting ILP [2]. Thread level parallelism relies on

1) Software developer’s effort to write their applications as a set of threads that can execute in parallel
2) Hardware support that allows execution of these threads in parallel

GPUs offer a throughput oriented architecture, that exploit TLP through fine grain multithreading. As shows in Figure 1, GPUs offers a high theoretical computation power compared to conventional CMPs. This theoretical computation power coupled with the programmability that improved with the introduction of new programming models opened new opportunities to speed up performance. Note that the margin between the theoretical peak GFLOP/sec of GPUs and CPUs shown in Figure 1 did shrink with the introduction of Xeon CPUs (2012) which are based on the Sandy bridge architecture and the introduction of dual-issue floating point units (2014).

The order of magnitude of speedup that GPUs offer over CPUs for regular applications does not hold for irregular applications. GPUs batch a group of threads and execute the batch, known as a *warp*, in lockstep in order to amortize the cost of fetch, decode and accessing the register files. Applications with threads following different control paths are known as *irregular applications.* Irregular applications, when running

on a GPU, alter the lockstep execution since different threads follow different paths. Baseline GPUs serialize the execution of diverging threads within a warp, resulting in TLP loss. Hence, a major challenge for GPUs is efficiently handling irregular control flow.

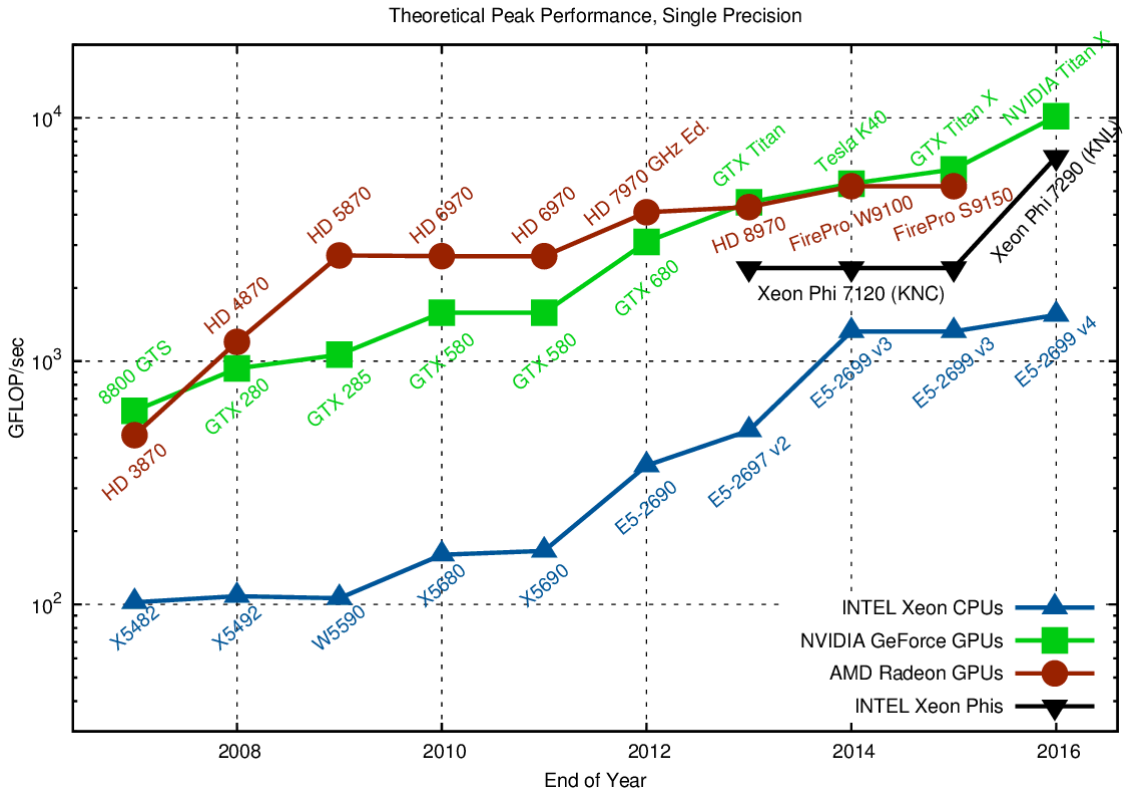


Figure 1: Theoretical peak GFLOP/sec for CPU vs GPU [36]

1.4 Contributions

The contributions of this thesis can be divided into two main parts:

1) The contributions of the first part titled "Opportunistic Inter-Path Re-convergence" are:
    a. We propose a more fine-grained approach to exploit *intra-warp* convergence by introducing the concept of *opcode-convergent threads* which execute the same instruction, but with potentially different operands.

b. We develop a compiler analysis based on longest common subsequence (LCS) [40] that identifies opcode convergence among divergent basic blocks and annotates the code so that it runs on our modified architecture.
c. We modify the GPU architecture so that opcode convergent instructions are executed as convergent instructions.
d. We quantify the percentage of common sub-blocks that exist among divergent basic blocks in a set of irregular GPU applications.
e. Using software simulation, we run irregular GPGPU workloads taking advantage of opcode convergence and we achieve 17% speedup over baseline architecture without introducing any performance loss on regular workloads.

2) The contributions of the second part titled "Locality-aware task-parallel execution on GPUs" are:
   a. We propose a locality-aware, task-queue abstraction for mapping task-parallel applications to GPUs.
      i. The proposed technique first extract data parallelism from the task parallel application by expanding task parallel work on the CPU}to generate a large number of tasks.
      ii. These tasks are then inserted into one or more task queues according to the type of computation they perform and, crucially, the locality properties of the tasks.
      iii. These queues are then merged into a single queue that is sent to the GPU, where they are executed in a data-parallel manner, with each task executing to completion on the GPU.
   b. We evaluate this queue abstraction on three applications: the fast multipole method, nearest-neighbor and two-point correlation. Our technique shows significant speedup for all three applications. And it yields to 50x speedup for one implementation of nearest neighbor.

1.5 Organization

The rest of the thesis is organized as follows:

- Chapter 2 provides a detailed discussion of the GPU architecture and the SIMT execution model.
- Chapter 3 provides an overview of the previously suggested solutions to the problem of control divergence in the literature.
- Chapter 4 describes our proposed solution "Opportunistic Inter-Path Re-convergence".
- Chapter 5 describes the simulation methodology, the three tools that we built (or modified in case of GPGPU-sim) and our experimental results.
- Chapter 6 is dedicated for our work on "Locality-aware task-parallel execution on GPUs", where we explain the proposed idea, the implementation and show the results of three task parallel applications.

2. OVERVIEW: THE GPU ARCHITECTURE AND THE SIMT EXECUTION MODEL

2.1 GPU Architecture

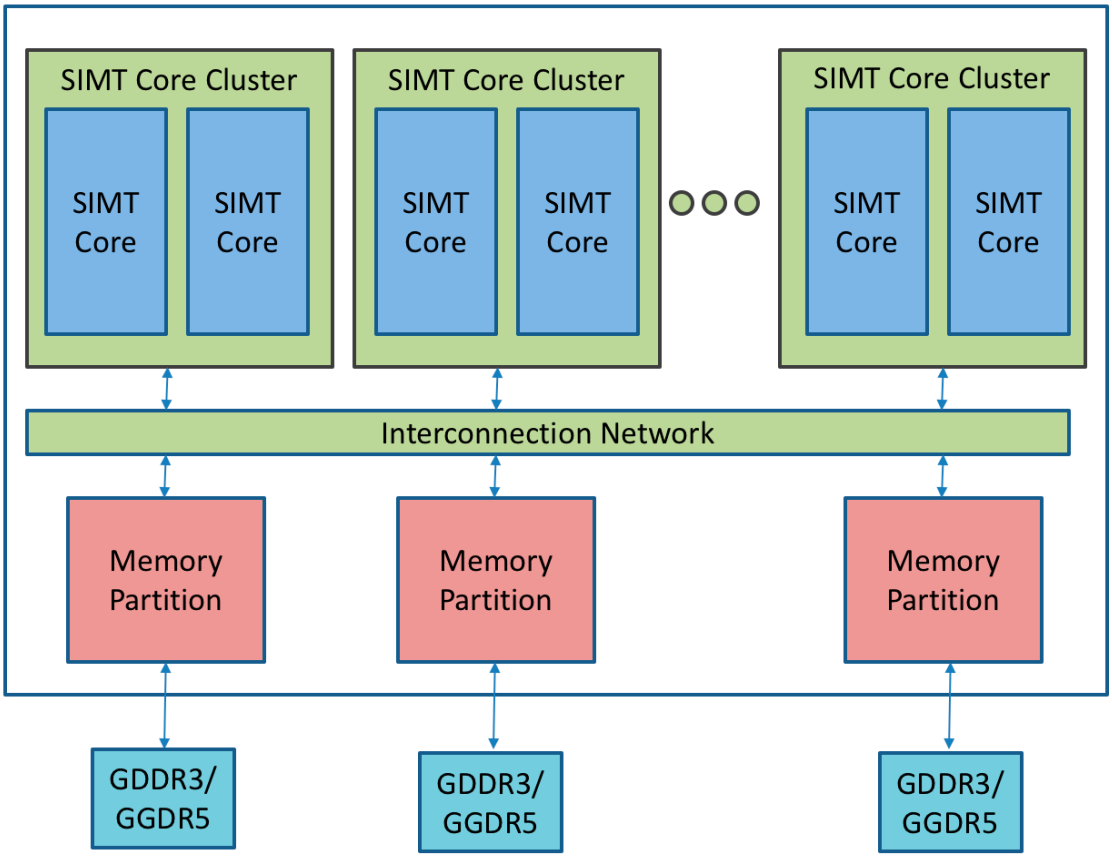


Figure 2: A generic GPU architecture

GPUs are throughput oriented machines; in other words, they get their improved performance, compared to superscalar out of order CPUs, from using heavy multithreading. In order to run thousands of threads, GPUs are made of many cores. Figure 3 below shows a modern Turing TU102 GPU by NVIDIA, which consist of 72 cores. A core in a GPU is usually made up of 32 (or 64) lanes which allows running 32(or 64) threads in lockstep in any given cycle. By context switching between different warps a set of 32(or 64) threads executing in lockstep each core in a GPU is capable of running thousands of threads. Threads executing on a given core can communicate results through

a scratchpad memory. Moreover, different warps running on a given core can synchronize using fast barrier operations. Each core has its own instruction cache and L1 data cache. Memory uses multiple channels in order to increase the data transfer rate. The parallelism in the memory system is a must in order to sustain the high computation throughput. The GPU cores and memory partitions are connected via an on-chip interconnection network.

Figure 3: Turing TU102 full GPU with 72 SM units.

2.2 Core Architecture

GPGPU cores are simpler cores than the cores used in a CPU. They do not implement any of the fancy architecture features that speedup the execution of single threads (i.e. no out of order execution, branch prediction, big caches). The high performance of regular applications running on GPUs is achieved through multithreading, as we mentioned earlier. Figure 4 shows the microarchitecture of a generic GPGPU core. Below is an explanation of each of the modules.

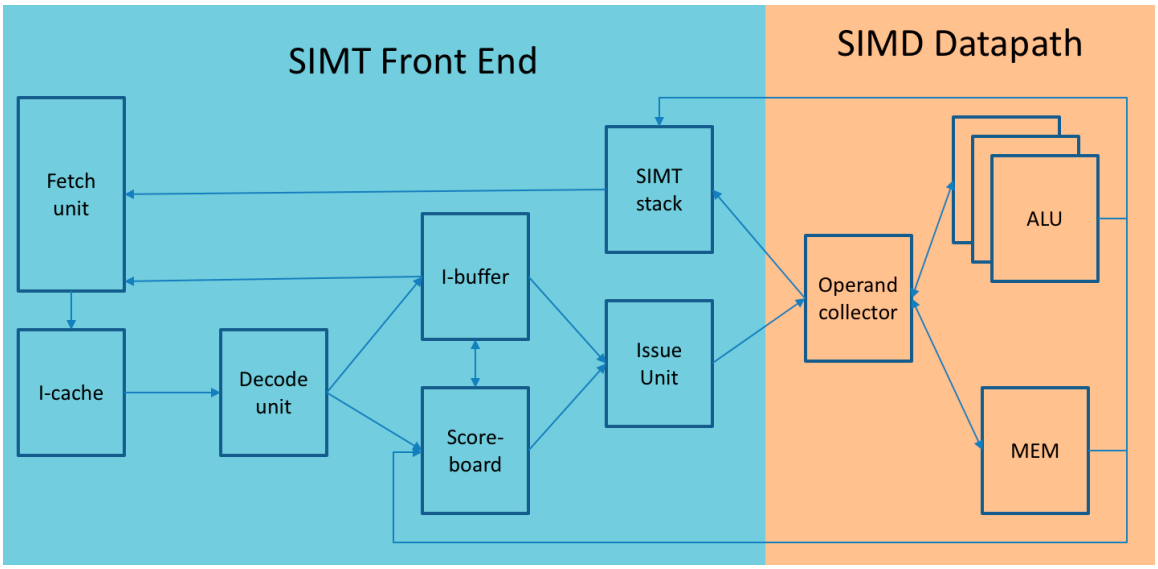


Figure 4: Microarchitecture of a generic GPGPU core

As discussed previously the threads are grouped in warps, consisting of 32 threads (usually) that execute in lockstep. A core will have multiple warps at any given time waiting to be scheduled. Each cycle, a warp is selected for scheduling. Having multiple warps helps hide long execution latencies. In other words, when a warp is waiting for an access to off chip memory, the core will be running different warps. Unfortunately, the number of warps that can "live" concurrently on a core is limited by the resources. A warp must ensure that the core has enough resources and registers before it starts executing.

Given that number of warps that can co-exist is limited by the resources available, and that we would like to hide latencies of long instructions, it becomes helpful if we can schedule subsequent instruction from a warp while earlier instructions from the same

warp have not yet completed. For this to be possible we need to know whether the next instruction of a warp is dependent on an earlier instruction that has not yet finished. This dependency information requires fetching the instruction from memory and decoding it to check whether it has such dependency. For this reason, GPUs use what is called an instruction buffer. Upon a cache fill or hit the instruction information is placed into the instruction buffer. The i-buffer haS dedicated slots for each warp. GPUs make use of a *scoreboard* to detect the dependencies. One naïve option of implementing the scoreboard would be to represent each register by one bit in the scoreboard and upon read or write each instruction checks the corresponding bits and if they are set to one the instruction is stalled. Unfortunately, this simple solution is not practical in GPUs because each warp in a GPU has 128 registers and a core can run up to 64 warps simultaneously, which means we need 8192 bits for the scoreboard. Moreover, because of the high number of warps running on a core, it could be that each warp has some instructions waiting on any of its four operands which means we may have up to 64(number of warps) *4(number of operands) = 256 scoreboard probes every cycle which would require 256 read ports.

A more feasible implementation would dedicate some entries in the scoreboard for each warp, usually around four entries. Upon fetching an instruction and placing it in the i-buffer, the scoreboard entries corresponding to the instruction's warp are checked. If any of the instructions' operands matches the scoreboard entry, then a ready bit in the buffer is set to 0 which means that the corresponding operand is not yet ready. when the entry is cleared from the scoreboard upon write back the ready bit in the i-buffer is set to 1. Once all the operands for a specific instruction in the i-buffer are ready. the scheduler starts considering the instruction for issuing.

Note that in the considered architecture, there are two scheduling decisions. The first is to select a warp and fetch its next instruction from the instruction cache and send it to the i-buffer. Usually this is done in a round robin manner among the warps that have no valid entries in the i-buffer. The second scheduling decision is selecting the instruction to be issued. This scheduler must select an instruction from the ready instructions in the i-buffer. Warp scheduling will be discussed in more detail in section 2.4.

Note that in the case of the GPGPU-sim simulator that we use to run my experiments, the i-buffer dedicates two entries per warp. Each i-buffer entry has two control bits, namely: a ready bit and a valid bit. The later indicates that the instruction is decoded but not yet issued. While the ready bit is set to 1 if the valid instruction is ready for execution, in other words if the data dependencies for this instruction are met and the instruction is ready to be issued.

The SIMT (single instruction multiple threads) execution model, used in GPU, provides the abstraction that each thread execute completely independently. This programming model is achieved via the use of a SIMT stack along with predication.

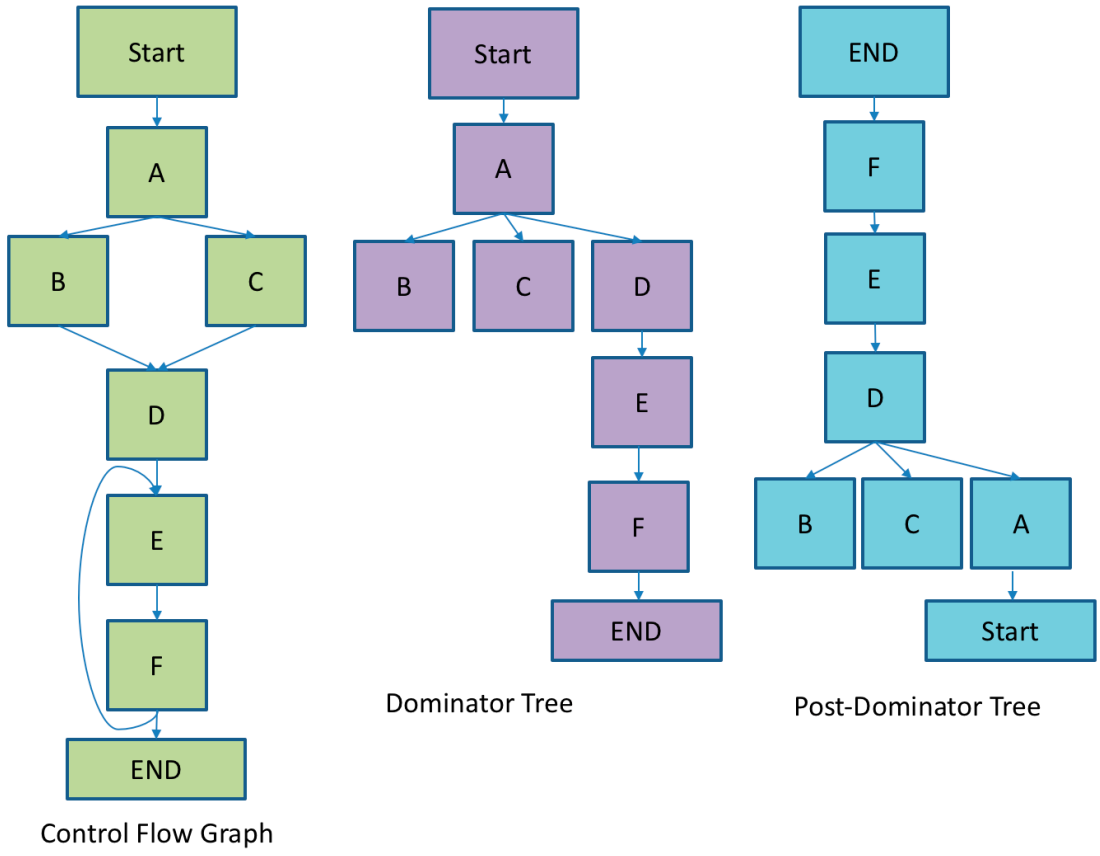


Figure 5: (A) Control flow graph. (B) Dominator tree (C) Post dominator tree.

As mentioned previously, threads in a warp execute in lockstep. However, upon a branch instruction different threads in a warp can follow different control path. This scenario is known as control divergence. Current GPUs handle control divergence by

serializing execution of threads that follow different control paths. While upon divergence it is possible for thread to stay divergent until the end of the program, this solution is not efficient. Hence, divergent threads in a GPU will try re-converge at the earliest possible opportunity. The *immediate post dominator* is the earliest re-convergence point that can be guaranteed at compile time. Note that some works [7] in the literature looked at earlier possible re-convergence points that can be defined at run-time.

The definition of post dominator is: an instruction 'b' is considered a post dominator of an instruction 'a' if every path from instruction 'a' to the end of the program has to go through 'b'. The immediate post dominator (IPDOM) is first instruction in the code that post dominates instruction 'a'.

2.3 SIMT Stack

It is through the use of a *SIMT stack* that GPUs handle serialized execution of divergent threads, re-convergence as well as the execution of nested branches. Figure 6 shows a CUDA code excerpt (Figure 7 shows its assembly equivalent). Different warp threads could diverge at the 'if' statement of line 6 as well as at line 8 at the nested 'if' statement. Figure 8 represents the control flow graph of this code excerpt. Note that the numbers in each basic block box of this control flow graph represent the mask of the threads of the currently running warp. A bit set to one represents the thread that actually execute the block while a bit set to zero represents a thread skipping the execution. Figure 9 represents the different states of the SIMT stack of this warp. Part a of Figure 9 represents the case where the execution is within basic block A and where all the bits of the mask are set to 1, which means that all the threads are executing this basic block. Once we reach the conditional statements at the end of basic block A, threads 1 through 3 take the branch while thread 4 goes in the fall through direction.

Upon this divergence we insert in the SIMT stack two new entries corresponding to the different paths that the threads are following (Figure 9a). Upon the insertion of these entries, threads 1,2,3 start executing block B while thread 4 is waiting. Every cycle, we compare the current PC to the re-convergence PC at the top entry of the SIMT stack.

If the two PCs are equal we pop this entry of the SIMT stack, since the divergent block would have reached its re-convergence point. Once we reach the end of basic block B we have another conditional statements and now the threads diverge again with thread 1 taking the jump while threads 2 and 3 going in the fall through direction.

Upon this ‘nested’ divergence, the SIMT stack is updated by popping the top entry and pushing three new entries representing basic blocks C, D, and E with their corresponding masks (Figure 9b). the execution continues with block C which is popped from the stack when it reaches its re-convergence point next block D is executed for threads 2 and 3 until it reaches its re-convergence PC. Figure 9c shows the SIMT stack while the program is executing block E which eventually will reach its re-convergence point and have its entry popped from the SIMT stack. At this point the execution will move to block F, and eventually all the threads will re-converge again once basic block F is done.

Figure 10 shows the serialized execution of threads upon divergence. it also shows the decreased lane utilization and the inefficiency of the SIMT programing model for applications with irregular control flow.

The literature describes different implementations of the SIMT stack which can be managed in software through the use of special dedicated instructions [37] or in hardware [6], [7]. In this work we consider the SIMT stack managed fully by hardware mainly because the SIMT stack in the simulator used in this work is hardware managed.

```
1.  Do {
2.    T1=tid*4;                 //A
3.    T2=T1+I;
4.    T3=DATA1[T2];
5.    T4=0;
6.    IF(T3 != T4) {
7.      T5=DATA2[T2];           //B
8.      IF(T5 != T4) {
9.        X +=1;                //C
10.     } ELSE {
11.       Y +=2;                //D
12.     }
13.   } ELSE{
14.     Z +=3;                  //F
15.   }
16.   I++;                      //G
17. }WHILE (I<N)
```

Figure 6: Example CUDA source code

```
1.   A:  mul.lo.u32     t1, tid, N;
2.       add.u32        t2, t1, I;
3.       ld.global.u32  t3, [t2];
4.       mov.u32        t4, 0;
5.       setp.eq.u32    p1, t3, t4;
6.       @p1 bra        F;
7.   B:  ld.global.u32  t5, [t2];
8.       setp.eq.u32    p2, t5, t4;
9.       @p2 bra        D;
10.  C:  add.u32        x, x, 1;
11.      bra            E;
12.  D:  add.u32        y, y, 2;
13.  E:  bra            G;
14.  F:  add.u32        z, z, 3;
15.  G:  add.u32        I, I, 1;
16.      setp.le.u32    p3, I, N;
17.      @p3 bra        A;
```

Figure 7: Example PTX assembly code

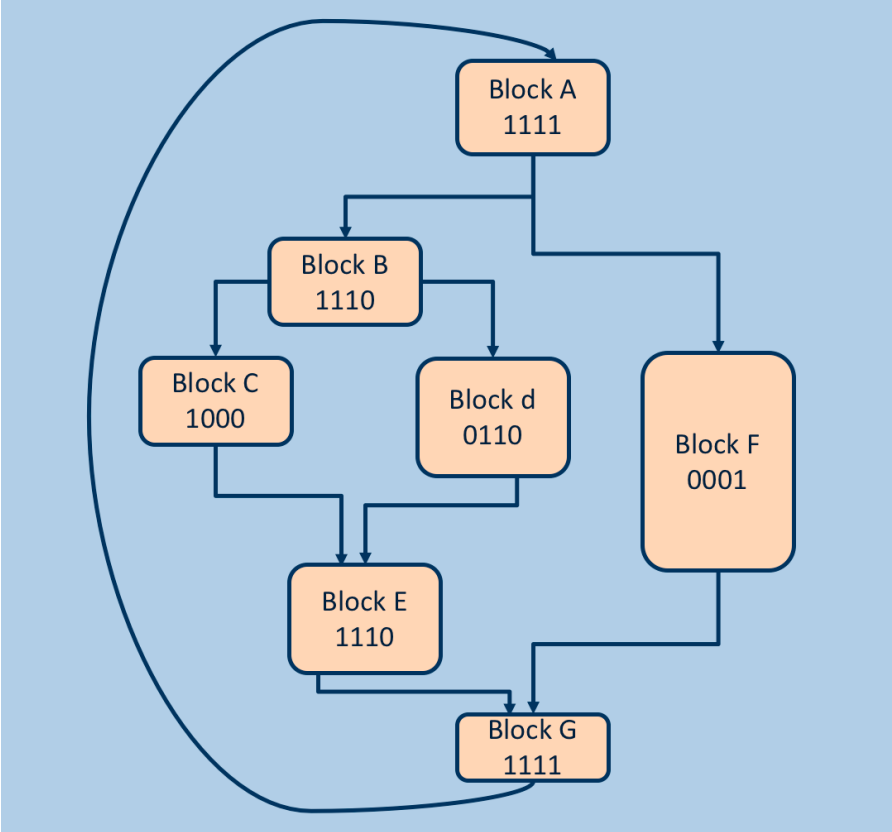


Figure 8: CFG of the sample code shown in fig 6

RPC	Next PC	Active Mask
-	G	1111
G	F	0001
G	B	1110

9-A

RPC	Next PC	Active Mask
-	G	1111
G	F	0001
G	E	1110
E	D	0110
E	C	1000

9-B

RPC	Next PC	Active Mask
-	G	1111
G	F	0001
G	E	1110

9-C

Figure 9: Different states of the SIMT stack

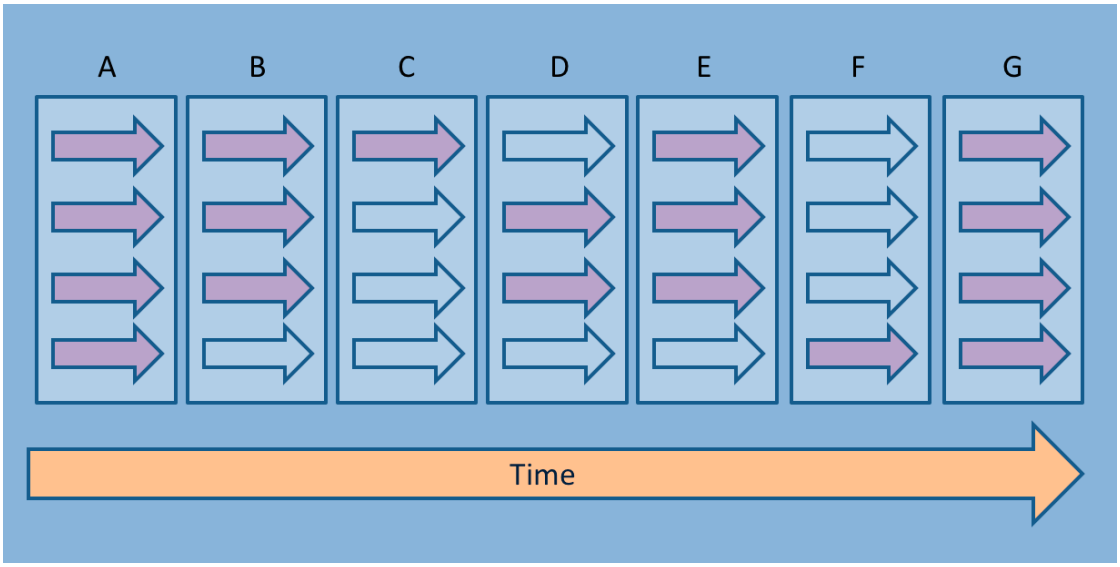


Figure 10: Timeline of the execution

2.4 Warp Scheduling

As mentioned earlier, GPUs are throughput oriented machines that use simple cores and heavy multithreading in order to hide the latency of slow memory operations. Two questions that arise are how many warps should live on a core at the same time so that the core remains busy at all time? And how should the warps be scheduled? The answer to both question would be simple if the memory system is ideal and responds to memory requests within a fixed time. If this were the case we could simply find out the number of needed warps by using a round robin scheduling algorithm to schedule the warps. The round robin scheduling will allow roughly equal time for each warp before being re-scheduled. Hence the minimum number of warps required to keep the core busy would be equal to the latency of the slowest instruction. However, the memory system is not ideal and the response to memory requests depends on the locality properties of the application. Furthermore, allowing a different warp to execute every cycle requires that each warp has its own registers. Otherwise we have to copy the state of the warp back and forth every time it is scheduled which is both energy and time consuming. The requirement of having dedicated registers for each running warp puts a limit on the number of warps that can live concurrently on a core because otherwise the chip area devoted for the registers will increase relative to the chip area devoted for the functional units which will limit the computation power of the chip. Thus the number warps is

limited by the resources and that is why GPUs allow scheduling instructions from the same warp even if previous instructions from the same warp have not finished executing. This will allow better hiding of latencies without increasing the number of warps and hence without increasing the number of required registers.

Since not all latencies can be hidden due to the limited number of warps that can live concurrently on a core, it becomes important to pick the scheduling algorithm that results in the least cache misses, i.e. a scheduling algorithm that takes advantage of locality. This has been the topic of considerable research. when different warps access nearby data at a similar point in the execution, it becomes beneficial that all these warps make equal progress to take advantage of the locality. Round robin warp scheduling can help taking advantage of locality in this case [3].

However, when different warps access disjoint data, like in the case of recursive tree based algorithms, it may be beneficial that a warp get scheduled repeatedly to maximize locality within the warp [4].

2.5 Operand Collector

Switching between warps every cycle requires physical registers to be devoted for each warp in order not to waste time and energy on moving state between memory and the register file. Devoting physical registers to each warp makes the register file big, it is almost 256 KB on recent GPUs like Kepler from NVIDIA.

One way to reduce the area of the register file is to use multiple banks of single-ported memories. GPUs use an operand collector to manage collecting the operands from the register file. The role of the operand collector is to help increase the bank level parallelism. This will be explained through the following example. Let us first consider we do not have operand collectors and that we just have a banked register file like the one shown in Figure 11.

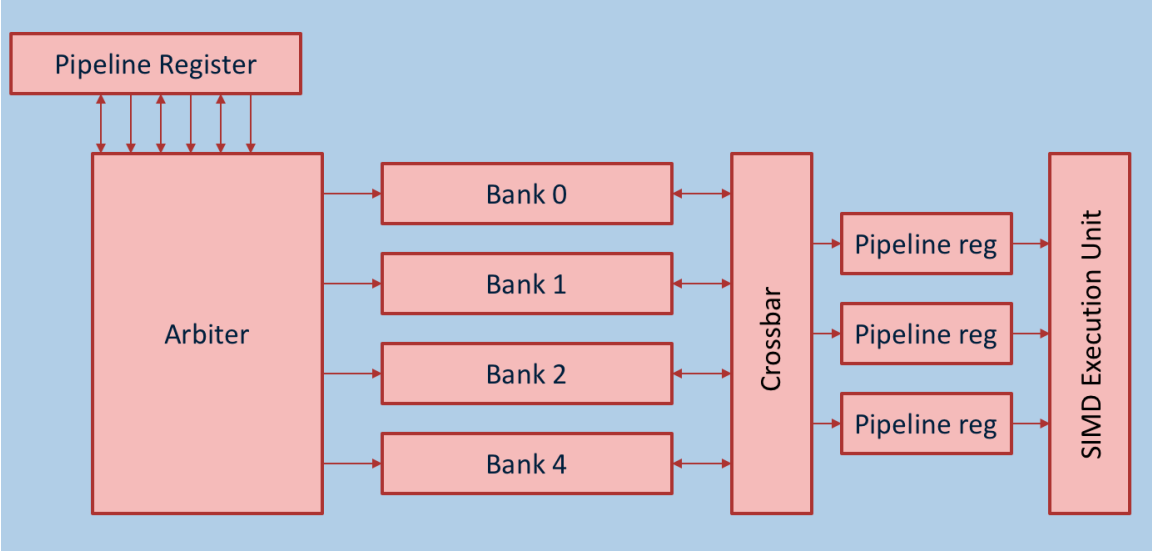


Figure 11: Architecture of naïve banked register file

In this microarchitecture we have 4 banks where each has a single port connected to a crossbar which route results to the pipeline registers. While this naïve microarchitecture will work fine, it will hurt performance due to little bank parallelism. To illustrate this let us consider having four warps running, the code show in the Figure 12 and let us consider the register layout shown in Figure 13

Cycle	Warp	Instruction
0	w0	mad r1, r2, r4, r6
1	w1	mad r1, r2, r4, r6
2	w2	mad r1, r2, r4, r6
3	w3	mad r1, r2, r4, r6

Figure 12: Issue time of different instructions

Bank 1	Bank 2	Bank 3	Bank 4
w0: r1	w0: r2	w0: r3	w0: r4
w0: r5	w0: r6	w0: r7	w1:r1
w1:r2	w1:r3	w1:r4	w1:r5
w1:r6	w1:r7	w2:r1	w2:r2
w2:r3	w2:r4	w2:r5	w2:r6
w2:r7	w3:r1	w3:r2	w3:r3
w3:r4	w3:r5	w3:r6	w3:r7

Figure 13: Distribution of registers across the banks of the register file

In clock cycle 1, warp zero instruction 1 can access both registers r2 and r4 however it cannot access register r6 because there is bank conflict with r2. In clock cycle 2, warp 0 accesses its last operand and now is ready to execute. In cycle 3, warp one accesses register r2 and r4 and again it cannot access r6 due to a conflict at bank 1. In cycle 4, warp 1 still cannot access r6 because the write of warp 0 conflict in bank 1 and the write has priority over the read. In cycle 5, warp 1 accesses register 6 and now it is ready to compute.

Now that warp 1 has all its operands, warp 2 can start accessing its operands, and it does access r4 and r6 at clock cycle. Warp r2 cannot access r2 in the same cycle due to a bank conflict in bank 4. In cycle 7, warp 2 still cannot access its 3rd operand due to the higher priority of the write of warp 1, which has a bank conflict with the read of r2 needed by warp 2. In cycle 8, warp 2 access its last operand and it is ready to issue. In cycle 9, warp 3 only accesses two of its operands and cannot access the 3rd due to bank conflict. In cycle 10, warp 2 writes its result to bank 3, which bans warp 3 from accessing its third operand. in cycle 11, warp 3 access its last operands and issues. Warp three writes its result back at cycle 13.

While this microarchitecture works fine, it is inefficient in terms of taking advantage of bank parallelism. GPUs use an operand collector which is similar to the architecture shown in Figure 14

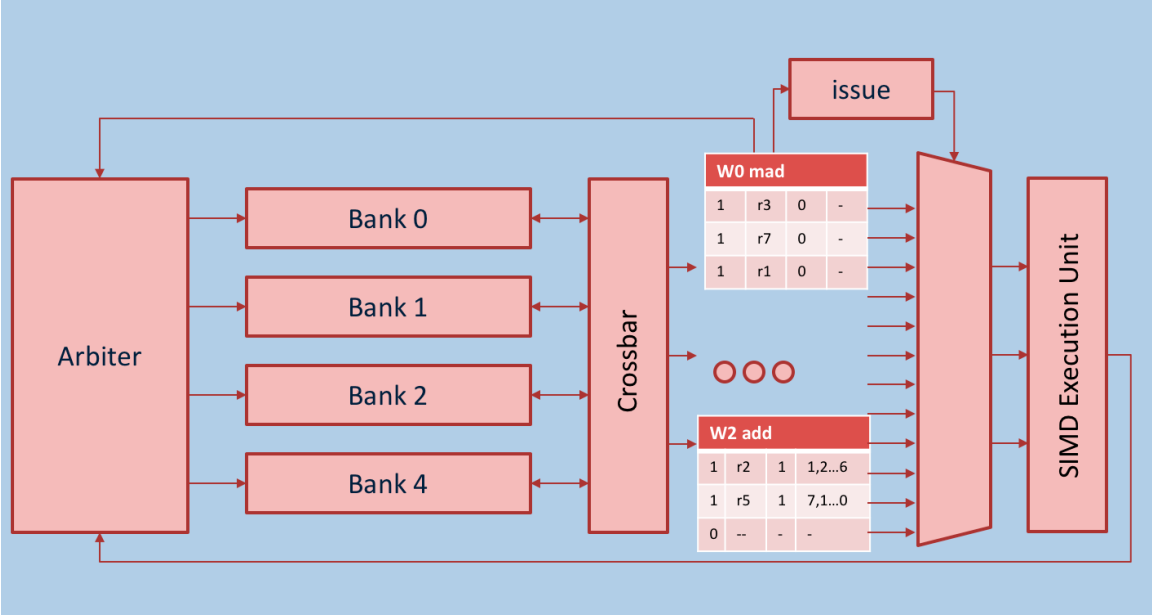


Figure 14: Operand collector microarchitecture (based on Tor M. Aamodt)

The major difference between this new microarchitecture and the previous one shown in Figure 11 is the presence of the collector units. Each issued instruction will be given one collector unit to collect its operands. This will allow multiple instructions to collect their operands in parallel and hence will take advantage of bank parallelism. Given the same example discussed above for the naïve microarchitecture, if we use the collector unit microarchitecture we get a 62% speedup. In cycle 1, only warp 0 has issued and warp 0 will read its first operand. In cycle 2, warp 0 reads its second operand and at the same time warp 1 can read r2 since it does not conflict with the bank that warp 0 is reading from. In cycle 3, warp 0 reads its third operand and is ready to execute, while warp 1 reads r6 and warp 2, which just issued, reads its first operand, r4. In cycle 4, warp 1 reads r4 and is ready to execute, warp 2 reads its second operand, which is r6, and warp 3, which just issued, reads its first operand, r4. In cycle 5, warp 0 writes back its result, warp 2 reads r2 from bank4, and warp 3 reads r2 from bank 3. In cycle 6, warp 1 writes its result while warp 3 reads its 3rd and last operand. In cycle 7, warp 2 writes its result and in cycle 8, warp 3 writes its result. In total the execution took 8 cycles while in the case of the naïve microarchitecture it took 13 cycles. Figure 15 and Figure 16 shows the timing in the naïve microarchitecture and of the operand collector respectively.

	Cycle												
Bank	1	2	3	4	5	6	7	8	9	10	11	12	13
4	w0:r4					w2:r6	w1:r1	w2:r2					
3			w1:r4						w3:r2	w2:r1	w3:r6		
2	w0:r2	w0:r6				w2:r4							
1			w1:r2	w0:r1	w1:r6				w3:r4				w3:r1

Figure 15: Timeline of accessing a naive register file

	Cycle							
	1	2	3	4	5	6	7	8
Bank 4			w0:r4	w2:r6	w2:r2	w1:r1		
Bank 3				w1:r4	w3:r2	w3:r6	w2:r1	
Bank 2	w0:r2	w0:r6	w2:r4					
Bank 1		w1:r2	w1:r6	w3:r4	w0:r1			w3:r1

Figure 16: Timeline of accessing the operand collector

Note that each collector unit has space to buffer all source operands of the allocated instruction. Given that we have multiple instructions and multiple operands per instruction, the arbiter is able to achieve high level of bank-level parallelism.

Note that the register layout affects the amount of bank level parallelism, hence previous works has suggested that the same registers from different warps be assigned to different banks. Why? Because with round robin scheduling most probably warps will be making similar progress, which means that at any point in time most warps will be executing the same instructions and hence accessing the same registers. If the equivalent registers from different warps are mapped to same banks, then we will have bank conflicts between different warps. That is why Narasiman et al. [3] suggested allocating equivalent registers to different banks.

The operand collector as explained above can cause a write after read hazard, if two instructions from the same warp are present in the operand collector at the same time and the second instruction writes a register that the first instruction reads. If the second instruction gets issued first because its operands were ready before the first instruction, then the first instruction will be reading a wrong value eventually (the value updated by instruction two). In order to prevent this write after read hazard, Mishkin et al. [5] suggested that instructions must be issued from the operand collector in program order.

The different schemes suggested by Mishkin et al. [5] resulted in some slowdown that reached a factor of two for some schemes.

3. CONTROL DIVERGENCE IN THE LITERATURE

3.1 Introduction

As discussed earlier GPUs serialize the execution of threads in a warp when different threads go in different direction upon a conditional statement. The SIMT stack is responsible of ensuring correct execution of divergent threads as well as re-convergence at the earliest re-convergence point that can identified at compile time which is known as the immediate post dominator.

The use of the SIMT stack as explained previously causes lower SIMD efficiency since only a subset of the lanes will be active at any divergent point. Moreover, the serialization is not required for correctness, in other words the threads within a warp do not have any implicit data dependency. In case of needed communication, this has to be done explicitly through the use of barriers and shared memory.

The SIMT stack area per warp is 32 by 64 = 2048 bits. A SIMT stack is required for each running warp, So in order to allow 64 warps to run concurrently we need 64 * 2048 = 16 KB. In the following section, we explain the major works from the literature that tackled the problem of control divergence:

3.2 Dynamic Warp Formation (DWF)

GPUs use fine grained multithreading to hide memory latencies, this means that hundreds of threads are running usually on a given core. These threads are divided into warps, where a warp is a set of threads that execute in all threads. All warps usually execute the same code and not only the threads within one warp, is this due to the way threads are spawned. In current GPUs each warp handles divergence among its own thread. The observation that Fung et al. [6] make in this work is that among the hundreds of threads that run on a core each divergent branch target is executed by large number of threads that are scattered among the different warps. Thus, they suggest forming new warps dynamically upon divergence. The newly formed warps all share the same target upon the divergent branch, i.e these dynamically formed warps are convergent.

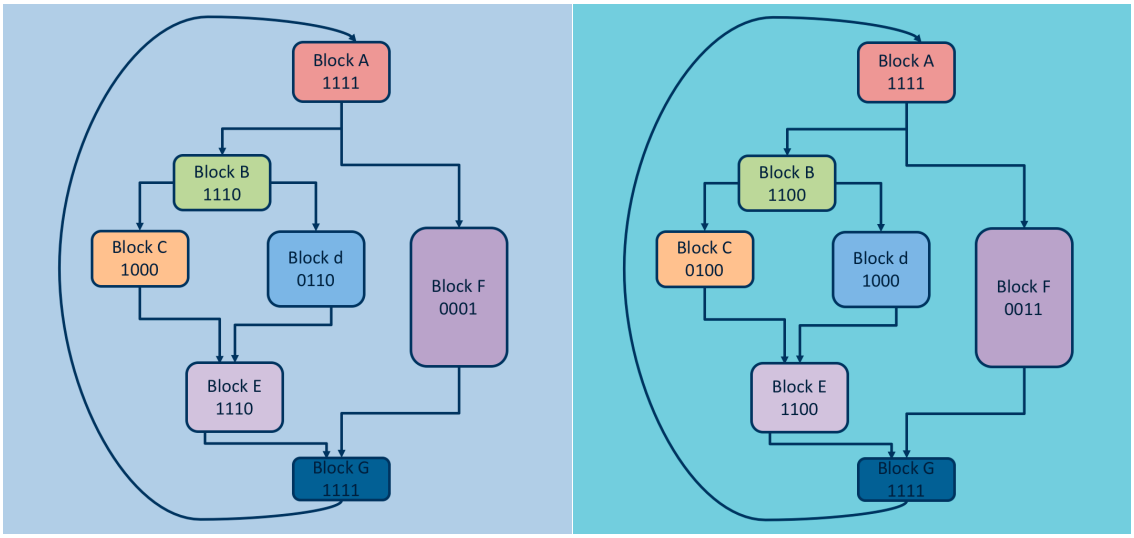


Figure 17: DWF example: 2 CFGs showing 2 warps diverging differently

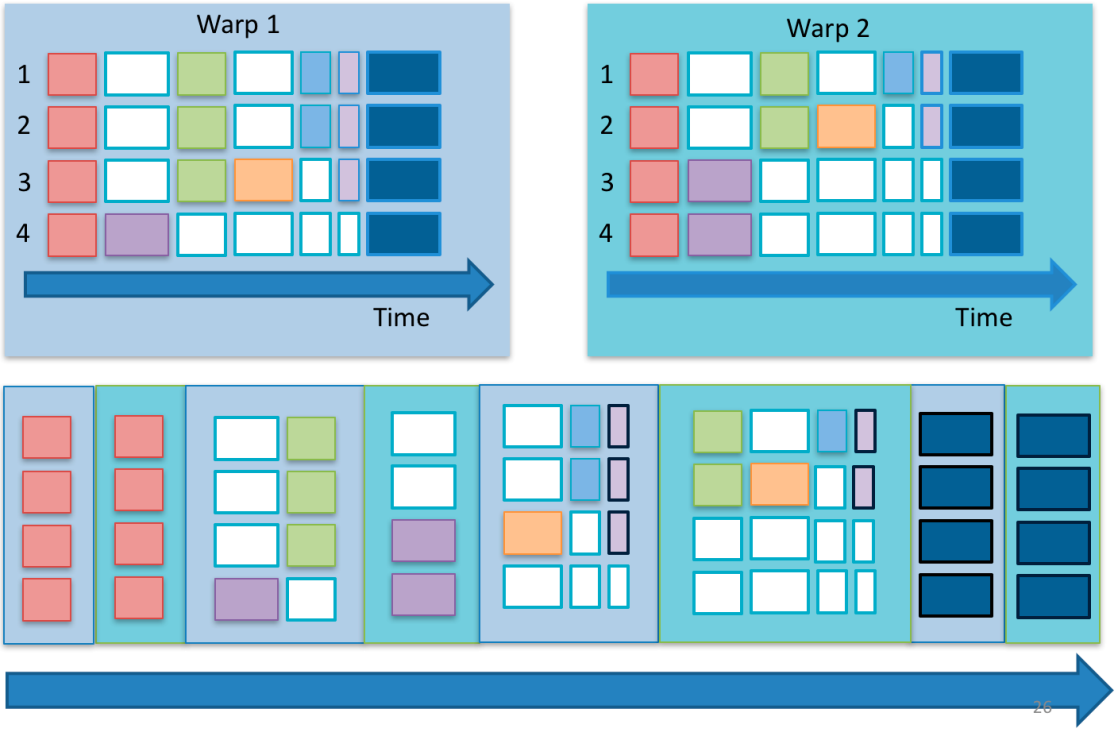


Figure 18: DWF example: (A) Basic Blocks executed by warp 1. (B) Basic Blocks executed by warp 2. (C) Execution Flow of 2 warps

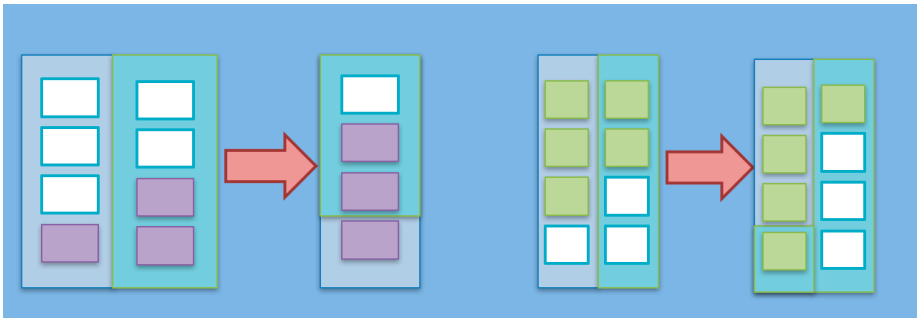


Figure 19: DWF example : Warp compaction

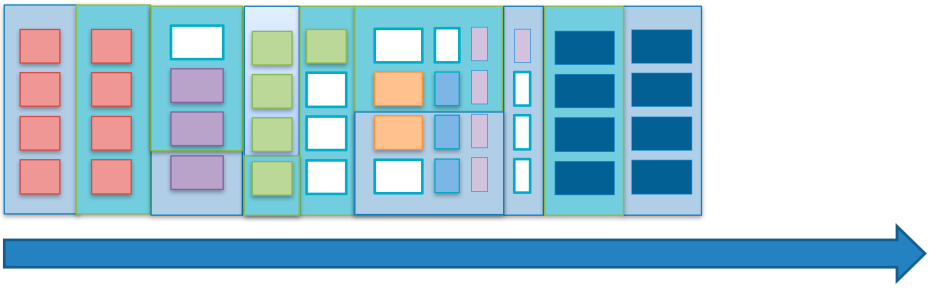


Figure 20: DWF example: Execution flow using DWF

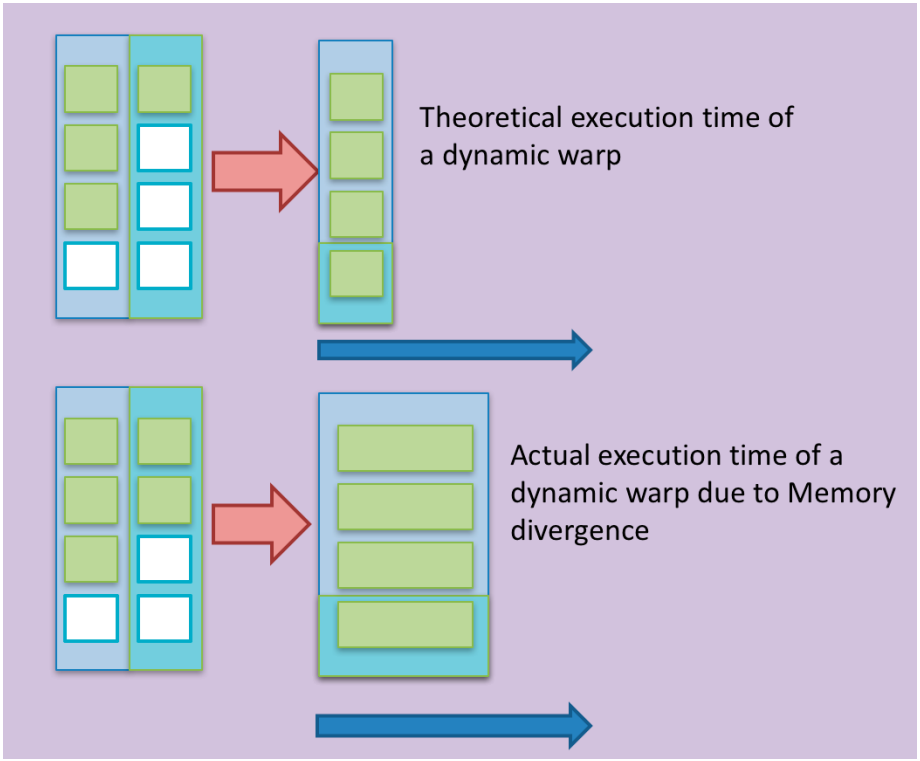


Figure 21: DWF example: The effect of Memory divergence

Figure 17 and 18 shows an example of Dynamic warp formation. Having two warps 1 and 2 each constituting of four threads. Both warps are convergent at block A. At the end of block ‘A’, the first three threads of warp ‘1’ jump to block ‘B’ while the fourth thread jumps to block ‘F’. Warp ‘2’ has its first two thread jump to block ‘B’ and the other two threads jump to ‘F’. Furthermore, at the end of block ‘B’ the first thread of warp ‘1’ jumps to block ‘C’ and the 2nd and the 3rd threads jump to block ‘D’. Warp ‘2’ also diverges at the end of block ‘B’ with thread 3 jumping to block ‘C’ and thread 4 jumping to block ‘D’. By looking at each basic block of the control flow graph shown in Figure 1,7 we can see that if divergence is dealt with across different warps we can get a better lane utilization. For example block F will be executed once for 1 thread of warp ‘1’ and another time for 2 threads from warp ‘2’. Hence the total number of threads executing block ‘F’ is three threads which is lower than the number of threads per warp. Thus if these threads belonged to 1 warp we could have executed block ‘F’ once instead of two times. Similarly block ‘C’ can be executed once instead of twice, and the same applies to block ‘D’. Note that with dynamic warp formation, block ‘B’ and block ‘E’ will be executed twice like in the case of static warp because the number of threads executing these blocks is five, which is bigger than the number of threads per warp, so we need two warps to execute these blocks, and thus DWF does not help with these two basic blocks (Figure 19 shows how threads can be merged into dynamic warps)
Figure 20 shows the theoretical execution time of Dynamic warp formation vs the baseline GPU architecture and Figure 21 shows that some dynamic warps will take extra time executing due to memory divergence resulting from compacting threads from different warps.

DWF requires changes to the register file. When grouping different threads from different static warps into the newly formed warps, these new threads perform the same instructions but they need to access different registers. In their paper on DWF, Fung et. al restricted threads to stick to their original assigned lane when moved to a new warp. Otherwise the register file needs to be designed to have a crossbar (Figure 22-B), which complicates the register file access, Moreover, if two threads from same ‘home’ lane are grouped into the same warp, these two threads will have a bank conflict since each lane is assigned a bank. Even with the restriction that the threads stick to their home lanes, the

traditional register (Figure 22 – A) file is not sufficient. Registers accessed by the threads of initially created warps are at the same offset within the lane, thus we need one decoder. However, in newly dynamic formed warps, the offset to access registers in a warp is not the same in each lane, thus a common decoder for all lanes is not possible (Figure 22-c).

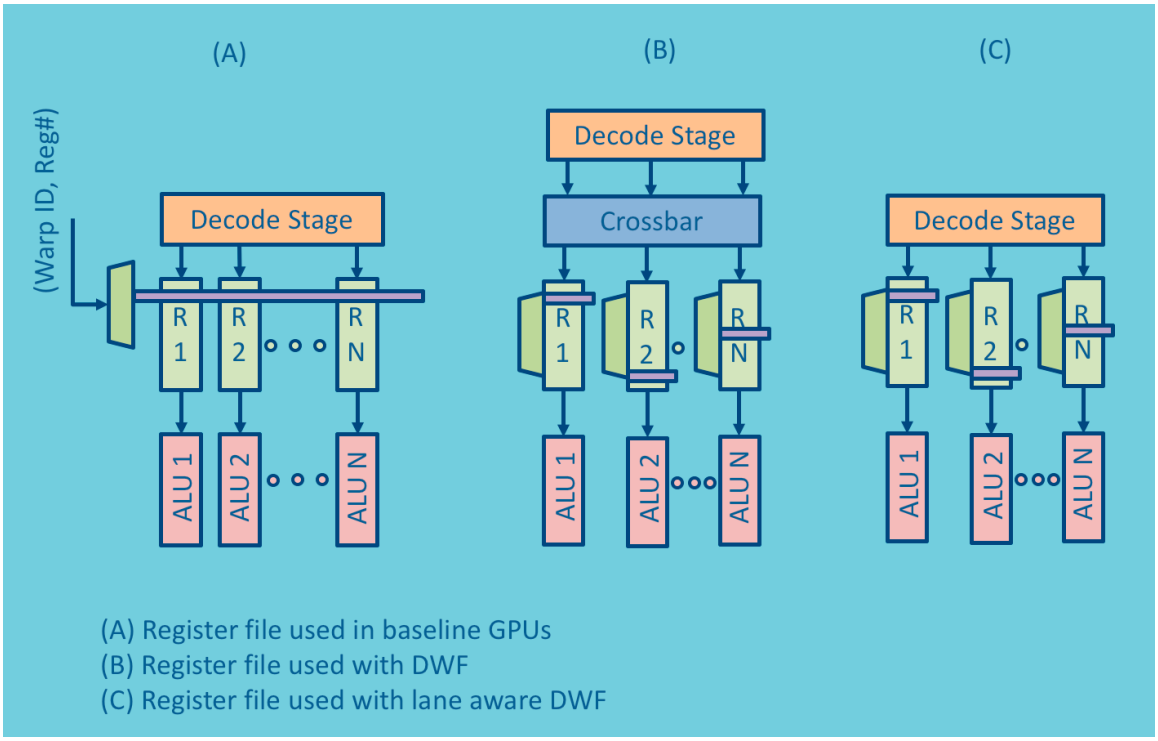


Figure 22: Register file of DWF (based on Figure 10 from Fung et al. [6])

The first major drawback of dynamic warp formation is that some threads can starve while waiting to be grouped with other threads in a new warp. These threads will not progress because no other threads are following the control path. The second major drawback of Dynamic warp formation is increased memory divergence. In the GPU world, when different threads from a warp load from memory it could be the case that some threads hit in cache while other miss and since the SIMT programming model requires the threads of a given warp to execute in lockstep, the threads that hit in cache must wait for the threads that miss in cache. This phenomenon is known as memory divergence. It turned out that grouping threads from different warp will increase the memory footprint of the warp which will increase the probability of memory divergence to occur. Moreover, even if all thread miss in cache, the probability that the different threads belonging to different warps to access nearby data is low which will increase non

coalesced memory accesses and shared memory conflicts. These drawbacks of dynamic warp formation causes slow down for many applications despite increasing the lane utilization. One last drawback of Dynamic warp formation is that applications that require implicit synchronization between the threads of a static warp will execute incorrectly.

One solution that was proposed by the authors of DWF is to rearrange the threads back to their 'home' warps when executing the non-divergent portions of the application. Moreover, many scheduling techniques were studied to avoid thread starvation.

3.3 Thread Block Compaction (TBC)

Thread block compaction [7] is based on the same observation as dynamic warp formation. However, it limits the warp formations to threads belonging to the same core. Thread block compaction creates the new warps mainly by modifying the SIMT stack so that it encompasses all warps executing in the same core. New warps are formed upon a divergent branch. Note that synchronizing all threads of a block at each divergent branch reduces the available TLP. In other words, the scheduler may not find instructions to execute since all the threads are waiting for synchronization. Thread block compaction restrict the compaction of threads to threads belonging to the same thread block. This restriction is a compromise between the TLP availability and the SIMD utilization. In other words, if we do not regroup thread into new warps, the lane utilization will be low because of the divergence. And if we synchronize all the threads in a core upon divergence in order to create the new warps we end up with the TLP loss explained above. By restricting the compaction within a block, TBC claim that other blocks could be executing and moving forward while the threads in a divergent block are waiting to synchronize at the divergent branch. Figure 24 below shows the high level operations of Thread block compaction.

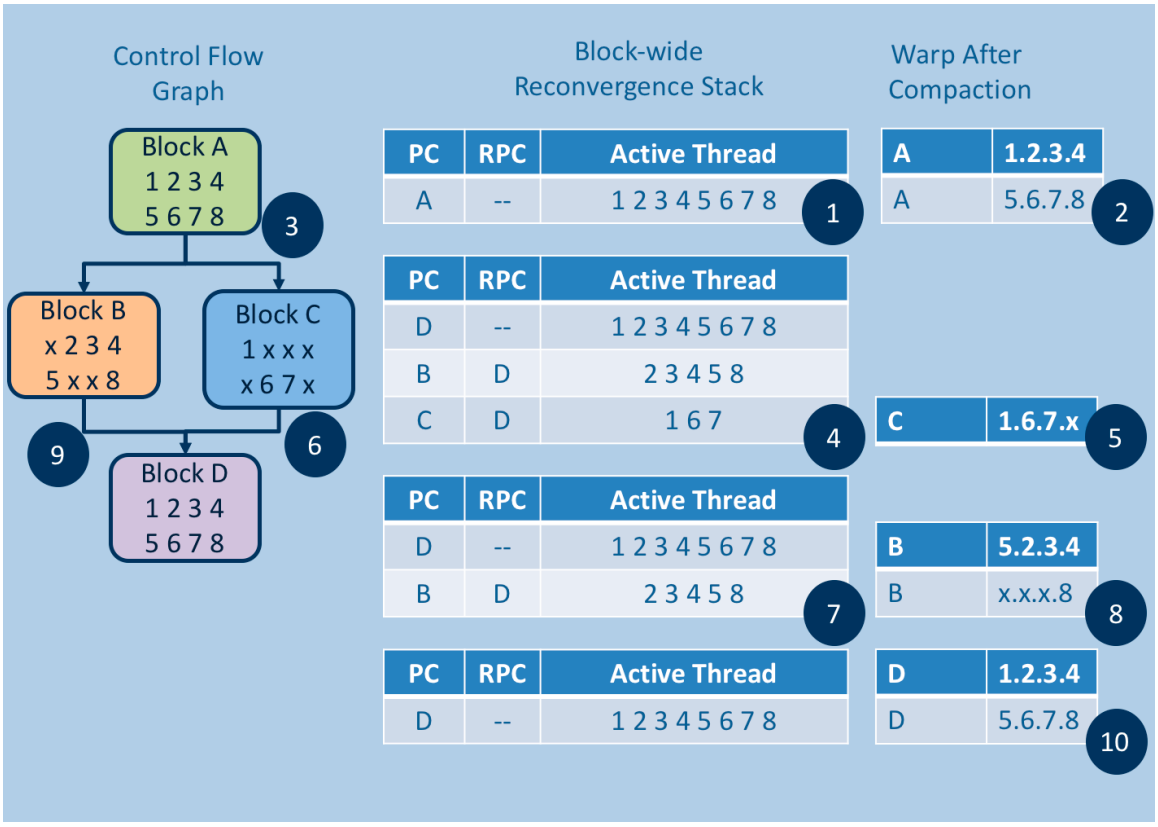


Figure 23: High level operation of TBC

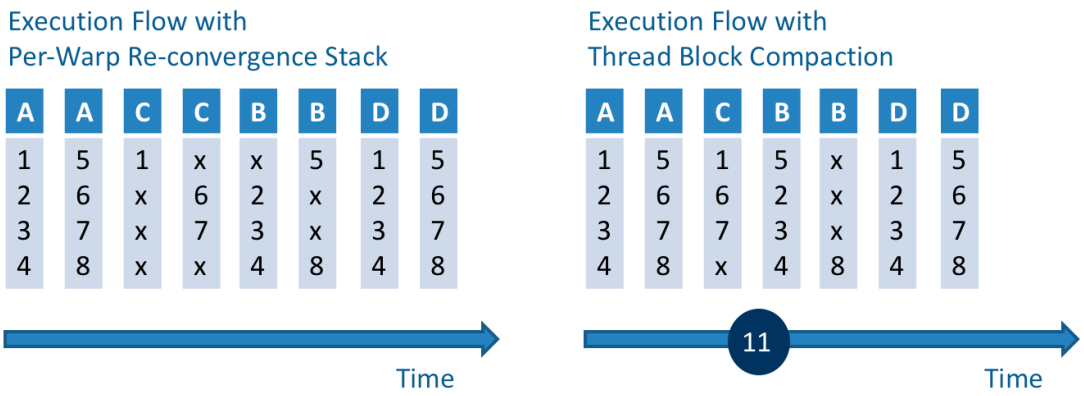


Figure 24: TBC execution Time (based on Figure 7 from Fung et al. [7])

Part B of Figure 23 shows the control flow graph as well as the block wide re-convergence SIMT stack of example code shown in Figure 23(A). The two warps executing each has 4 threads. While executing block A there is no divergence and both warps are scheduled independently. At the end of block A both warps synchronize since this is a potential point of divergence. When both warps execute the branch instruction

two new entries are inserted in the stack. Each of these new entries contain all the threads jumping to its corresponding PC, even if these threads belonged previously to different warps. The threads are then compacted into new warps. When these new warps reach their re-convergence point their corresponding entries in the stack are popped. So first we have one warp executing block C and then we have two warps executing block B. when block B terminates the threads are reassigned to their initial warps.
Figure 24 compares the execution time of traditional per warp re-convergence stack and the execution time of thread block compaction. In this example the execution time is reduced by 12.5% using TBC.

3.4 Large Warp Microarchitecture (LWM)

Like Thread Block Compaction, LWM [3] extends each SIMT stack so that it handles more than one warp. However unlike Thread block compaction, which synchronizes threads at the branch instructions and at the re-convergence points, the group of warps in LWM all execute in lockstep. In other words, they are synchronizing at each instruction. This compaction at every instruction reduces thread level parallelism even more than in the case of thread block compaction. This lockstep execution of warps within a group allows performing compaction using predicate instructions.
Large warp microarchitecture tracks register dependency at thread granularity in order to allow some warps to execute before some other warps belonging to the same group. This decreases the thread level parallelism loss caused by the lockstep execution.

3.5 Simultaneous Warp Interweaving

As shown in Figure 25 simultaneous branch interweaving [8] support issuing 2 different instructions where the instructions come from warps that are divergent so that they fill the branch gaps. This complicates the front-end of the GPU.

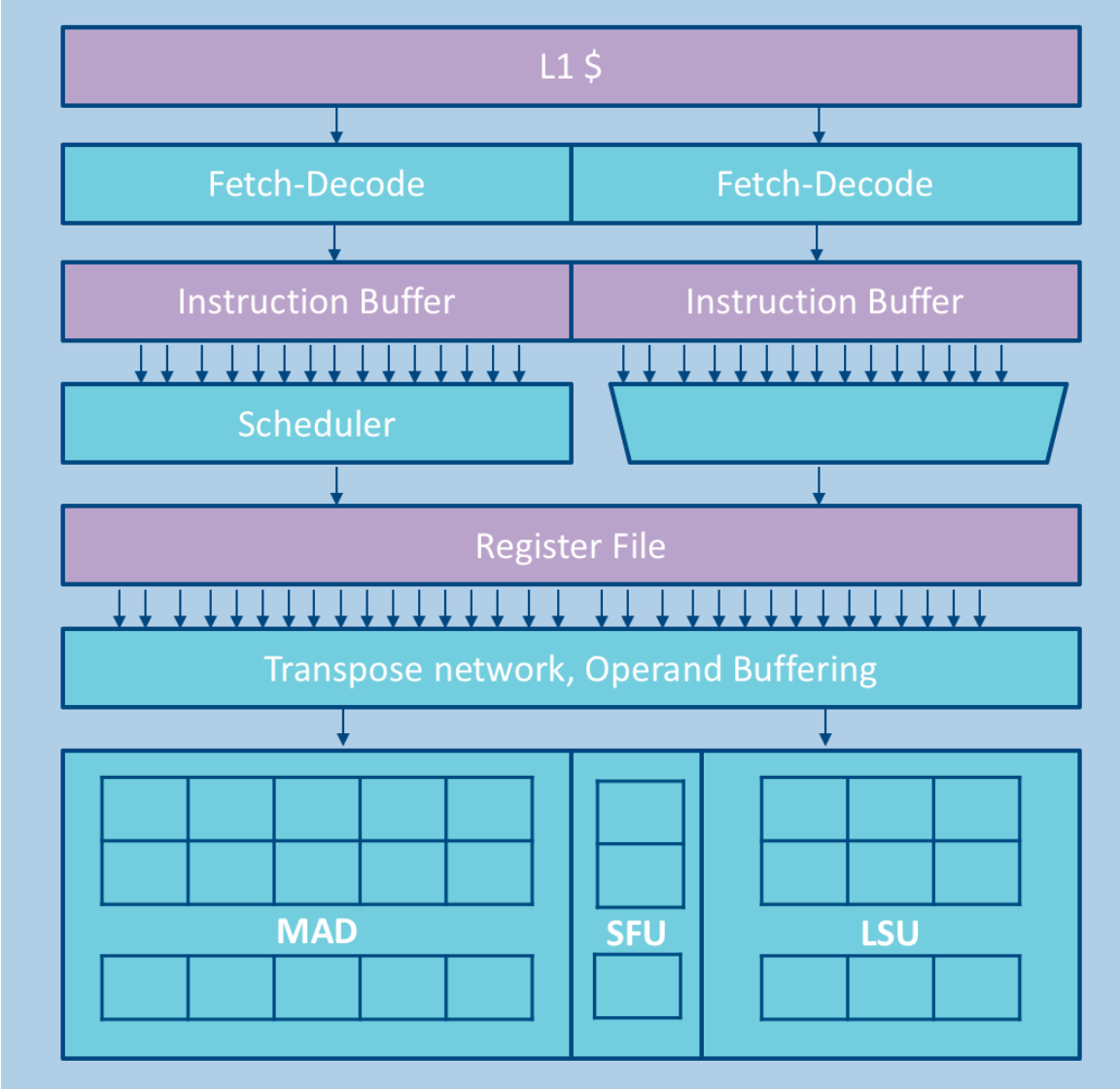


Figure 25: Simultaneous Branch Interweaving micro-architecture (based on Figure 3 from Brunie et al.)

3.6 Conditional Streams

Different works focused on using warp compaction in software in order to improve SIMD efficiency. Conditional streams [9], has its kernel splits it data steam into many streams based on the result of a divergent branch and then each sub-stream is processed by a different kernel which merge back at re-convergence.

3.7 SIMD Lane Permutation

As mentioned earlier the problem with most thread compaction techniques is that threads must remain in the same lane when a new dynamic warp is formed otherwise the register file state of the thread must be moved to a different lane. Rhu and Erez [10] observe that threads that take the same path tend to belong mostly to the same home lane and thus they conclude that the sequential mapping of thread IDs to consecutive threads in a warp is not optimal for any dynamic warp formation technique. So in this work they propose different thread mapping and improve the rate of compaction.

3.8 Multi-Path Parallelism

Upon divergence threads with same branch targets are grouped into warp splits. Warp splits are executed sequentially in baseline GPU. This serialization is not required for correctness. It is just used because it allows using a simple hardware and because graphics applications which GPUs were first built for were mostly regular application with little control divergence. Actually different warp splits can even execute in parallel because each thread has its own registers. Allowing an interleaved execution of these warp splits will give the warp scheduler more scheduling opportunities in order to hide the long latency memory instructions. In other words, interleaving the execution of divergent paths will boost thread level parallelism without improving the SIMD efficiency.

Multi-path execution boosts performance of memory bound applications, where the cores are idle waiting for memory accesses and were the limited resources of the cores does not allow more warps to co-exist and hide each other’s latencies.

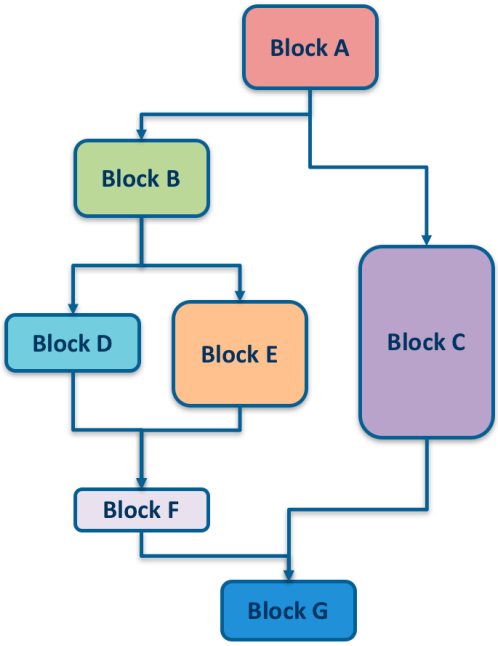


Figure 26: DPE example: Control Flow Graph

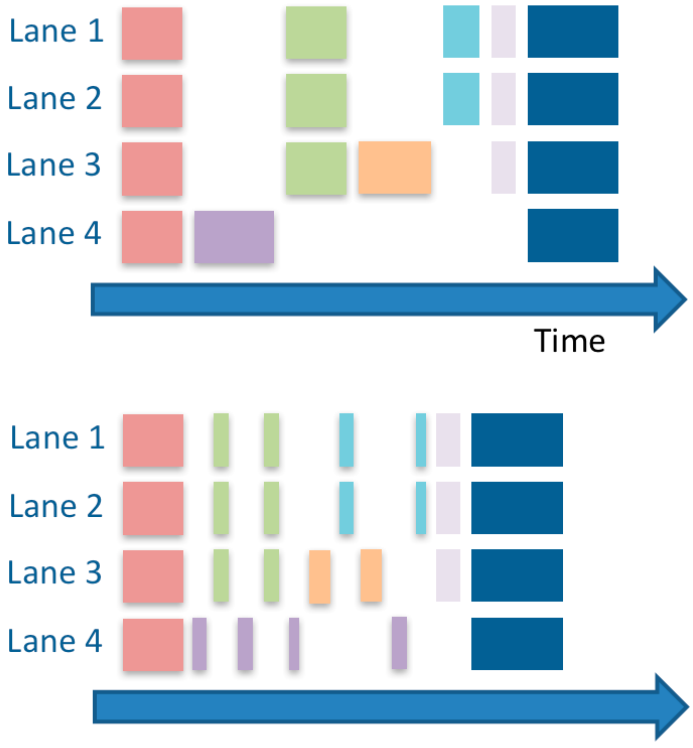


Figure 27: DPE example: Execution flow

Figure 27-A shows the performance when using baseline GPU while Figure 27-B shows the performance when allowing interleaving between different divergent blocks.

So while block C is executing in case it is waiting for memory access the threads that went in the other direction of the branch (block B) can start executing. similarly, block D and Block E have their execution interleaved, while D waits on its long latency access, threads that went in the E direction are executing.

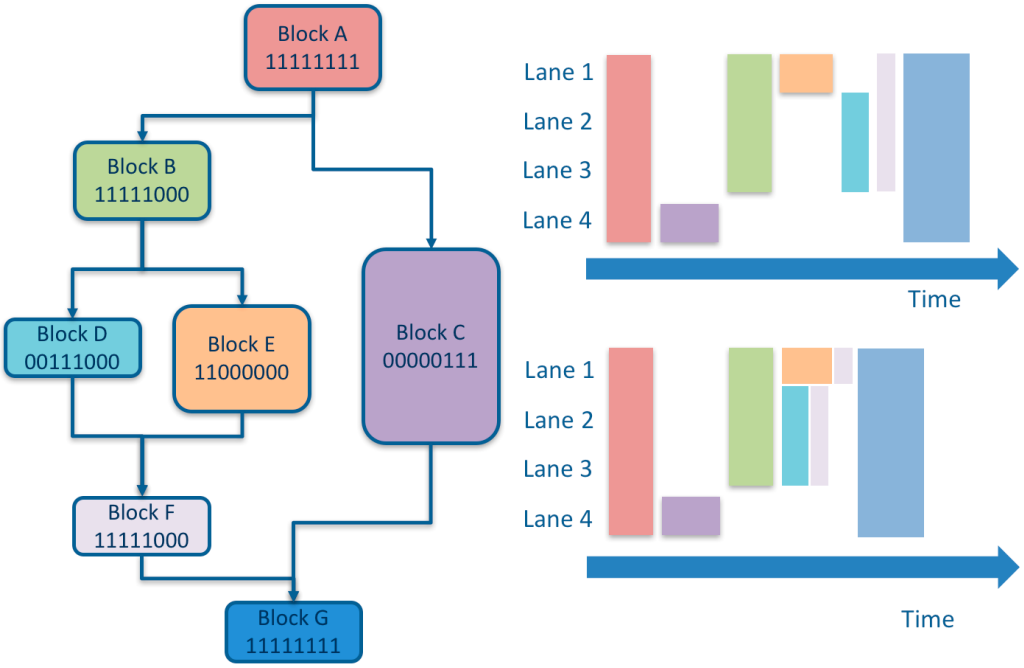


Figure 28: Dynamic Warp Subdivision example

3.9 Dynamic Warp Subdivision

Dynamic warp subdivision [11] split a divergent warp into two warp splits that execute in parallel. In order to achieve this, the re-convergence stack is extended with a warp split table. Dynamic warp subdivision allows warps splits also at memory divergence when different threads in a warp have different memory access latencies due to some threads hitting in cache while others missing. The threads that hit in level one data cache form a warp split and are allowed to move forward while the complementary warp split is waiting for its memory access to resolve. Note that in this case the threads that move forward may pre-fetch for the warp split that is still waiting for its access to resolve. As shown in Figure 28 above the warp splits are allowed to go beyond the immediate post dominator (block F in Figure 28) and not converge with the other warp-

splits. This may create the pre-fetch effect we discussed previously but it also risks pipeline under-utilization.

3.10 Dual Path Execution and Multi Path Execution

Dual path execution [12] restricts DWS so that each warp executes only two concurrent warp-splits. This restriction allows capturing most of the benefits of DWS while keeping the hardware simple. DPE requires some extension for the SIMT stack so that both entries of a divergent branch are at the top of the stack. In other words, so that the warp scheduler can interleave the execution of both branch target. While DPE does not increase the SIMD utilization it does offer higher level of parallelism to the warp scheduler which helps at better hiding the latency of long memory accesses. Other than the SIMT re-convergence stack, DPE requires some extensions to the scoreboard so that dependencies are tracked for each warp split separately. Multi path execution [13] follows DPE but gets rid of the dual path limitation.

3.11 Likely Convergence Points

As we discussed earlier, the immediate post dominator is the earliest re-convergence point that it is safe to re-converge at and that can be identified at compile time. However, re-convergence can happen earlier than IPDOM in reality. Figure 29 shows a situation where threads can re-converge at an earlier point than the IPDOM. The IPDOM of both block A and block C is block F however in most cases the threads can re-converge at block E since 73% of the times block C jumps to block E and 25% of the time block B jumps to block E and 96% of the times block E jumps back to block A. Hence, the path from block C to block D is rarely followed and in most cases at runtime we can re-converge at block E.

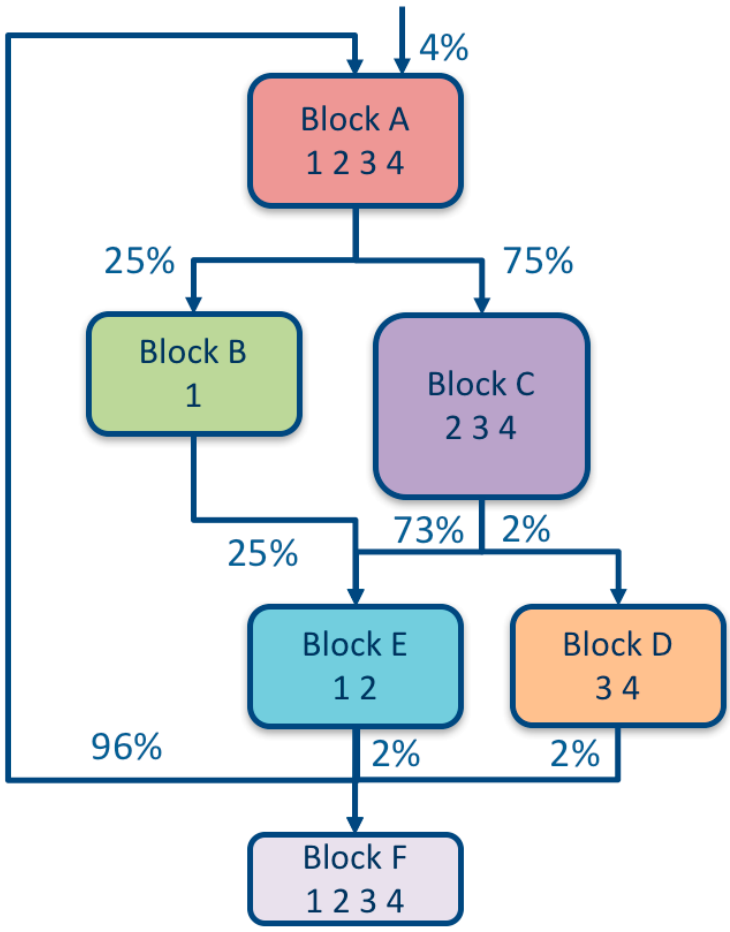


Figure 29: CFG annotated with path probabilities

Fung et al. proposes extending the re-convergence stack with LCP (likely re-convergence points) and a pointer that points to the stack position of a special likely convergence entry that is created when a branch has an LCP that is not equal to the IPDOM. When a warp diverges, three entries are added to the stack. The first one is the likely point of convergence. The other two correspond to the two sides of the divergent branch as shows in Figure 29-a. Now when the warp diverges at block C, two new entries are created (4). When the execution reaches the LPC, the entry of E is removed from the stack (5). When thread 3 and thread 4 reaches block F, the stack is popped (7) . Then block B executes and its entry is popped once the PC = LPC. Finally, the likely convergence entry is executed until it reaches the IPDOM.

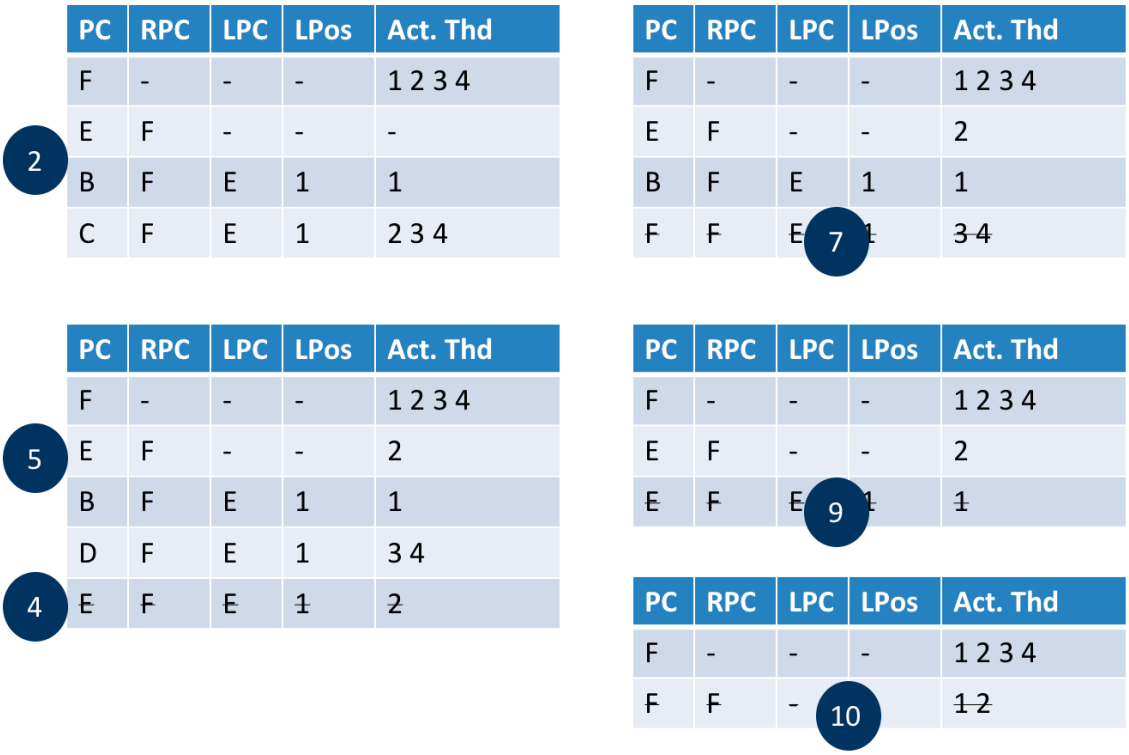


Figure 30: SIMT stack of the Likely convergence example (based on Figure 3.19 from Aamodt et al.)

3.12 Vector Thread Architecture

The vector thread architecture [14] combines MIMD and SIMD, i.e. the front end can fetch instructions from the shared L1 instruction (Figure 31) cache and issue an instruction to be executed across all threads or it can switch to a MIMD architecture where a lane fetches instructions from its private level zero cache and executes its own instructions. Vector thread architecture have comparable efficiency to SIMT machines when running regular applications but outperforms GPUs when executing irregular applications.

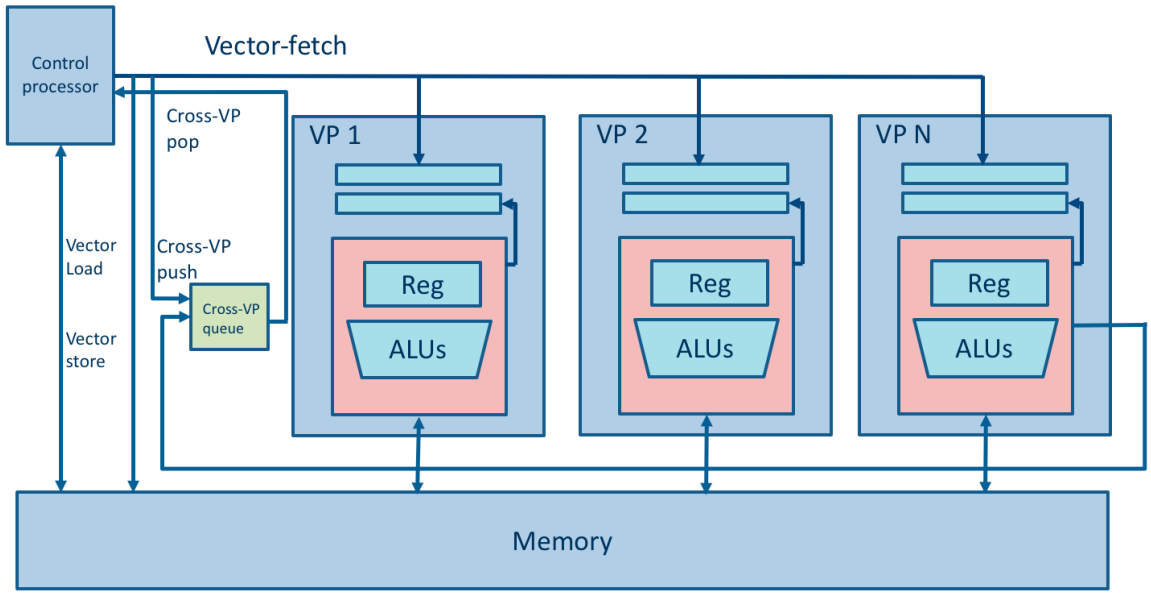


Figure 31: Abstract model of a vector-thread architecture (base on Figure 1 of Krashinsky et al.)

3.13 Temporal SIMT

Temporal SIMT [15] machines (Figure 32) are similar to vector thread machines in the sense that they allow each lane to execute in MIMD fashion. Each lane, however, runs one warp. In other words, the warp with all its threads is time multiplexed in one lane and thus the machine amortizes the control flow overhead across time, while traditional GPUs amortize the overhead across the lanes.

3.14 Variable Warp Sizing

In variable warp sizing, Rogers et al. [16] propose narrow warps, called slices, where each slice consist of 4 lanes. Each slice contains fetch and decode units. These slices can be grouped into gangs. If the application has no branch divergence, the warps belonging to a gang execute in lockstep and use a shared fetch unit and a shared level one of cache. Every time the control flow diverges the gang splits into multiple gangs. The split gang can split further if it counters divergence again. Splitting can continue until each warp is in its own gang. Upon reaching this point, each single warp

gang start using its own fetch unit and private level zero cache. The gangs are merged back when the PC are equal in different gangs

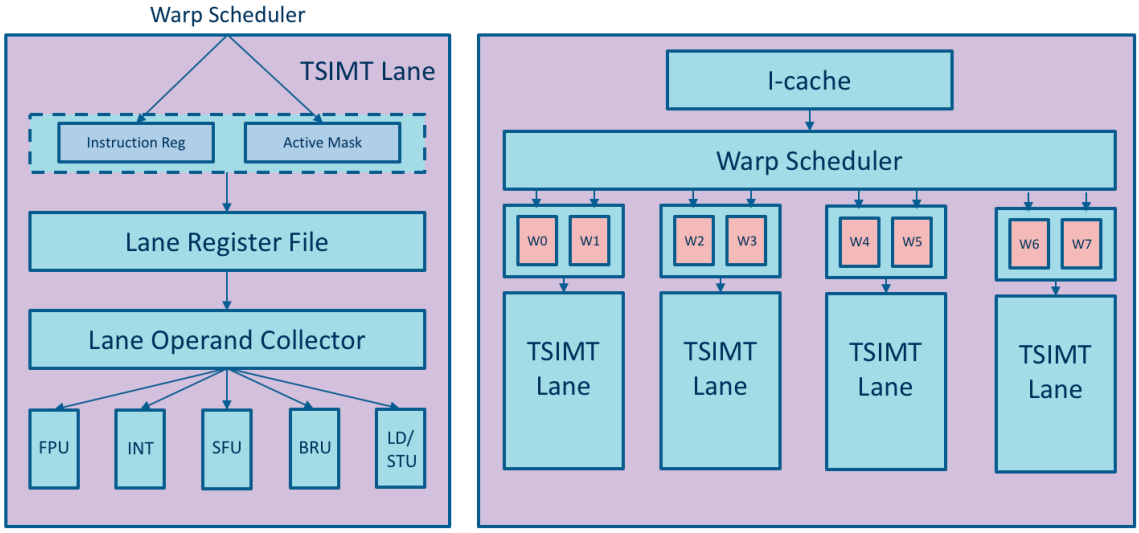


Figure 32: Temporal SIMT micro-architecture (based on Figure 1 from Keckler et al.)

3.15 SIMT Stack Alternatives and Divergence Management:

The size of the SIMT stack scales with the number of threads per warp as well as the number of warps that co-exist in a specific core. The area requirement of a baseline re-convergence stack is 32*64 bits. While this is not huge in comparison to the size of the register file, many proposals discussed alternative mechanisms of implementing the re-convergence stack, mainly in order to take advantage of the area dedicated to the stack when an application does not diverge.

Among these proposals is the AMD's GCN [37] which proposes the use of a scalar register file whose registers are used as predication registers to control the execution in each warp. The compiler uses this structure to emulate the re-convergence stack in software. In the absence of divergence, the compiler can make use of the registers for other computations. Moreover, they minimize the size of the stack by prioritizing the execution of the path that has lower number of active threads.

Another proposition was a stack-less SIMT coupled with temporal SIMT architecture [39]. In this proposition threads execute in lockstep when they are convergent and shift to a MIMD execution upon divergence. The instruction set architecture is extended with a

sync-warp instruction that the compiler places at the re-convergence point, so that divergent threads re-converge. The compiler analyzes the code to determine convergent and divergent regions and detects the branches that are taken by all threads by following these steps:

1) All basic blocks are considered thread invariant at the beginning
2) Mark all threads dependent on thread ID as thread variant.
3) Iteratively mark all instructions dependent on thread-variant instructions as thread variants as well
4) All instructions in a BB that are control dependent on a thread variant branch are also thread variant

This stack-less technique does not catch all possible re-convergence points in the case of nested divergent branches, but performs comparably to the SIMT stack solution when the applications are fairly regular.

Old GPUs actually did not use a full SIMT stack. Instead they used predication. Predication is a known technique in the CPU world used to reduce pipeline stalls due to a control hazard. In the world of CPU, when a branch is miss-predicted, the penalty is large. The CPU stalls the pipeline and starts fetching instructions from the branch target. With the use of hardware branch prediction, the number of stalls decreases, however the rate of miss-prediction remains between 5 and 10 percent, which can slow down a processor by as much as 30%. Predication solves this problem by removing the branch instruction itself and executing both possible paths of the branch. The instructions, however, will have a predicate tag appended by the compiler and will only be allowed to commit their results and change the program state if their corresponding predicate is true. The value of the predicate will be only true in one of the paths. Predication relies on parallel executions of two instructions, each belonging to a different side of the branch, and thus it requires an architecture that feature multiple instruction issue. The following example (Figure 33) illustrates the idea of predication. In a baseline CPU, the code is divided into 4 blocks, while with the use of predication it will executed as a single block (Figure 33- c). Note that the two predicates are complimentary of each other, thus only one can be true at a given time and thus only one path will commit its results.

In GPUs, predication can be used as a low overhead way to implement simple conditional code sections. It saves the overhead of pushing and popping from the re-convergence stack.

Using predication allows the compiler to interleave the execution of the both paths coming out of the branch. This increases the Tthread level parallelism like the DPE work mentioned above without the advanced SIMT stack that DPE requires. Note that this is possible with predication because divergent blocks use different predication registers.

Predication increases the pressure on the register file, which will result in a lower number of warps that can co-exist.

Predication affects the dynamic number of instructions. It results in removing the pop and push instructions required to manage the re-convergence stack. On the other hand, checking the uniform branches increases the number of dynamic instructions.

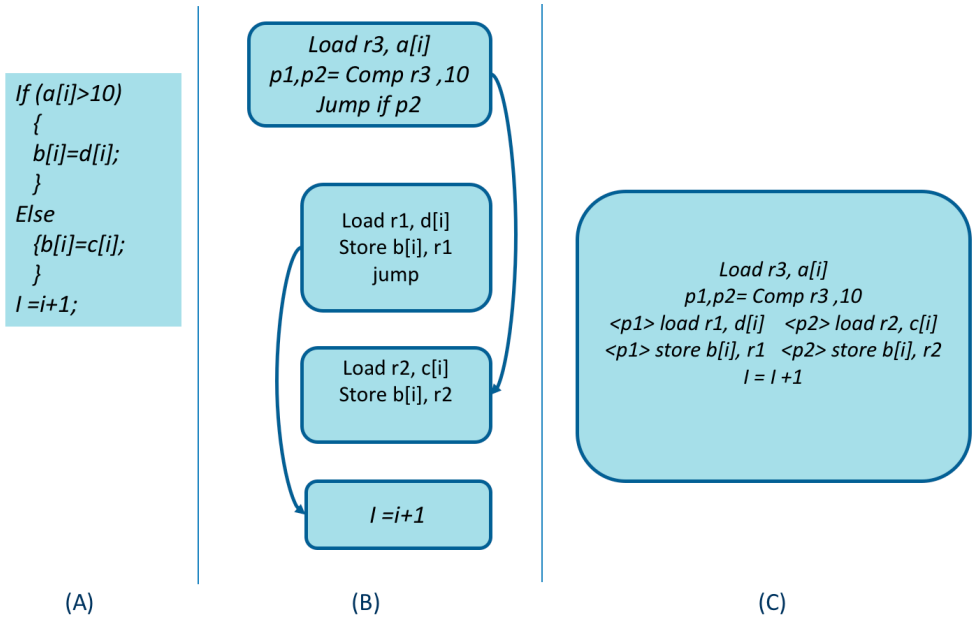


Figure 33: (B) Baseline CFG. (C) CFG using predication

4. OPPORTUNISTIC INTER-PATH RE-CONVERGENCE

4.1 Introduction

GPGPUs deliver high speed up for regular applications while remaining energy efficient. The single instruction multiple thread (SIMT) execution model used in GPUs amortizes the energy and bandwidth cost per instruction over a set of threads executing in lockstep, known as a warp, Moreover. GPUs use fine-grained multithreading to hide long off-chip latency. This high performance-energy promise does not hold for irregular control divergent applications. When different threads in a warp decide to jump to different targets upon executing a branch instruction the lockstep execution is altered. Current GPUs serialize the execution of threads following different paths within a given warp.

While this solution is simple and could be handled using a simple SIMT re-convergence stack in hardware, this technique leads to lower SIMD efficiency and needless serialization.

Different solutions were proposed in the literature to deal with control divergence. These solutions can be divided into three major categories:

- The first is based on warp compaction: GPU is a highly multithreaded throughput architecture, which means that any core has hundreds of threads running at the same time and executing the same kernel. Hence any target of a branch instruction is reached by many threads but that are scattered among multiple warps. Hence it makes sense, to compact threads that follow the same control path into new warps upon a divergent branch.
- The second family of works, focused on improvements to the SIMT re-convergence stack: The observation behind these works is that divergent threads within a warp are independent and thus their execution can be interleaved. The interleaved execution makes use of the idle cycles. These techniques are helpful mainly because the number of warps that can co-exist is limited by the resources which makes it hard for the scheduler to hide long off-chip latency.

- The third family of works adds a MIMD capability to traditional GPUs: In other words, the GPU performs in a SIMD mode the convergent parts of a warp and upon divergence it switches to a MIMD mode which allows all threads to make progress and not be serialized.

Detailed description of these different solutions is presented in chapter 3. While all these techniques improve the performance of some applications with divergent control flow, they had some drawbacks like increased non-coalesced memory accesses in case of thread compaction works which results in slow down for some applications, reduction in lock-stepping effect and hence GPGPUs' bandwidth and energy amortization in case of adding MIMD capabilities. While some inefficiency for irregular workloads may be unavoidable some works imposed some loss on regular workloads.

In this work we introduce the concept of *opcode-convergent threads* which are threads that execute the same instructions but could be accessing different operands. We find out that divergent basic blocks exhibit substantial opcode convergence. For example, in the case of LUD 65%, of the divergent instructions are actually opcode convergent. This means that upon serializing the execution of two divergent basic blocks, many instructions are re-executed in the same order but potentially accessing different operands.

This high percentage of opcode convergence among divergent blocks is due to the simple instruction set used in GPUs. GPUs are RISC machines. The large semantic gap between the high level language used in writing the application, CUDA in our case, and the assembly code running on the GPU results in basic blocks that exhibit a lot of opcode convergence even though the divergent basic blocks could be performing completely different computation at a high level.

We build a compiler that analyzes divergent blocks and identify the common streams of opcodes using the longest common subsequence (LCS) algorithm. We modify the GPU architecture so that these common instructions are executed as convergent instructions. Using software simulation, we achieve a 17% speedup over baseline GPGPU for irregular workloads and we do not incur any performance loss on regular workloads.

In this work we make the following contributions:

a. We propose a more fine-grained approach to exploit *intra-warp* convergence by introducing the concept of *opcode-convergent threads* which execute the same instruction, but with potentially different operands.
b. We develop a compiler analysis based on *longest common subsequence (LCS)* [40] that identifies opcode convergence among divergent basic block and annotates the code so that it runs on our modified architecture.
c. We modify the GPU architecture so that opcode convergent instructions are executed as convergent instructions.
d. We quantify the percentage of common sub-blocks that exist among divergent basic blocks in a set of irregular GPU applications.
e. Using software simulation, we run irregular GPGPUs workloads taking advantage of opcode convergence and we achieve 17% speedup over baseline architecture without introducing any performance loss on regular workloads.

4.2 Background

GPUs consists of multiple streaming multiprocessors (SM) each consisting of a set of cores. Hundreds or even thousands of threads run on each core and are divided into warps, where a warp is a set of threads that execute in lockstep.

GPUs provide an abstract execution model known as single instruction multiple threads (SIMT) while executing on Single Instruction Multiple Data (SIMD) units

The lockstep execution of threads within a warp allows a single instruction fetch and decode as well as wide vector read/write access to register files. Similarly, memory accesses from different threads can be coalesced if accessing consecutive addresses. Hence, through SIMD execution GPUs can amortize the energy and bandwidth cost per instruction over a wide set of threads in a warp.

When executing a branch instruction, different threads within a warp may jump to different targets. This situation is known as *control divergence*. Upon control

divergence, GPUs serialize the execution of diverging basic blocks which causes TLP (Thread-level Parallelism) loss. Divergent threads re-converge at the immediate post dominator (IPDOM), which is the safest and earliest re-convergence point that can be identified at compile time. For example, the IPDOM of the branch instruction at the end of basic block A, in Figure 34 is the first instruction in block D.

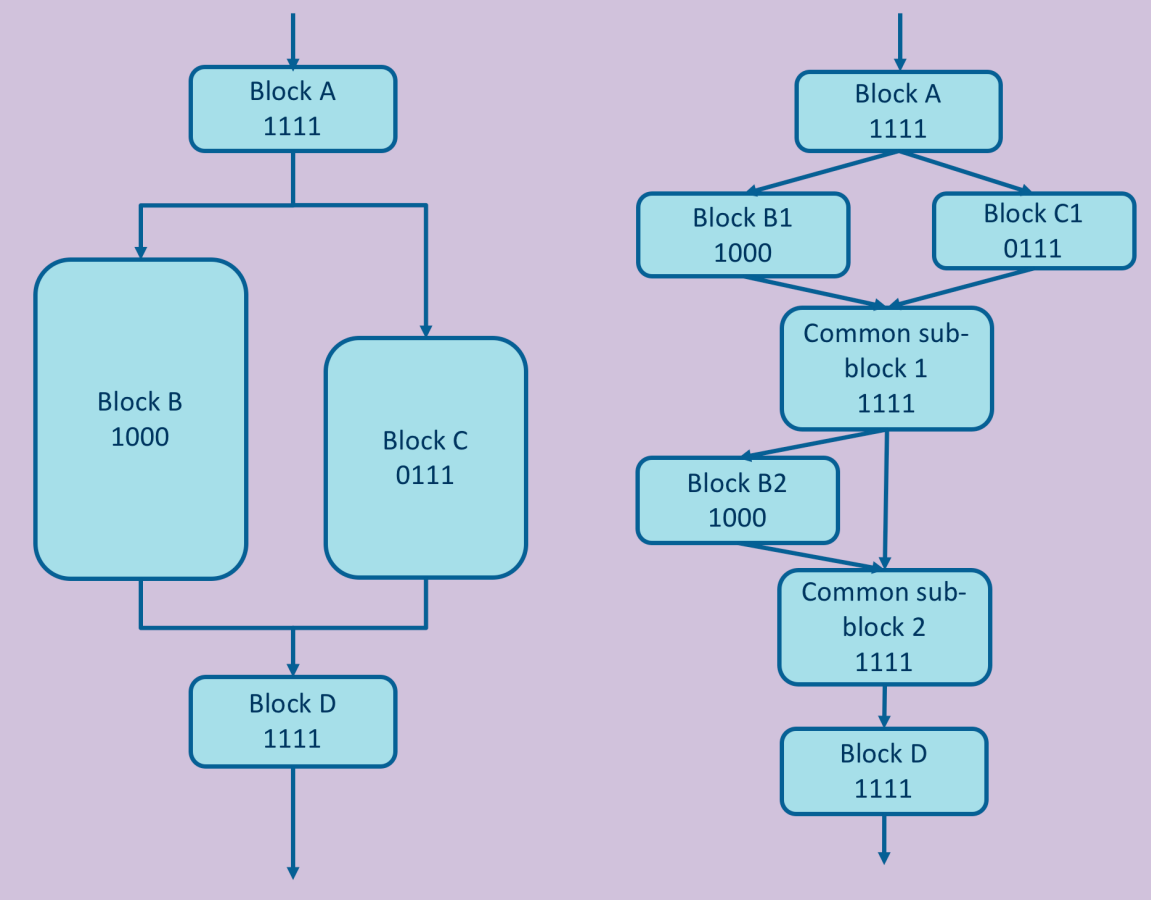


Figure 34: (A) CFG of original code VS (B) CFG of transformed code

GPUs make use of a SIMT-stack to handle divergence and re-convergence in hardware or can use predication in software as discussed in section 3.15. In this work, we consider the SIMT- stack solution mainly because the simulator we use, GPGPU-sim [41], implements the SIMT-stack solution. Every time the threads in a warp diverge two entries are pushed into the stack representing the two different control paths. An entry in the stack consists of three elements: the current PC, the active mask which helps tracking

which threads are active at each path and a re-convergence PC (RPC). At any given time only one control path is executing which correspond to the entry at the top of the stack. Once the current PC equals the re-convergence PC an entry is popped from the stack and execution is moved to the next entry.

When a warp starts executing the first entry is pushed to the stack with PC=0 , the RPC pointing to the end of the program, and an active mask of all ones indicating that all threads are active (Figure 35-a). When the divergent branch instruction at the end of block A executes, two new entries are pushed into the stack representing the two paths (Figure 35-b).

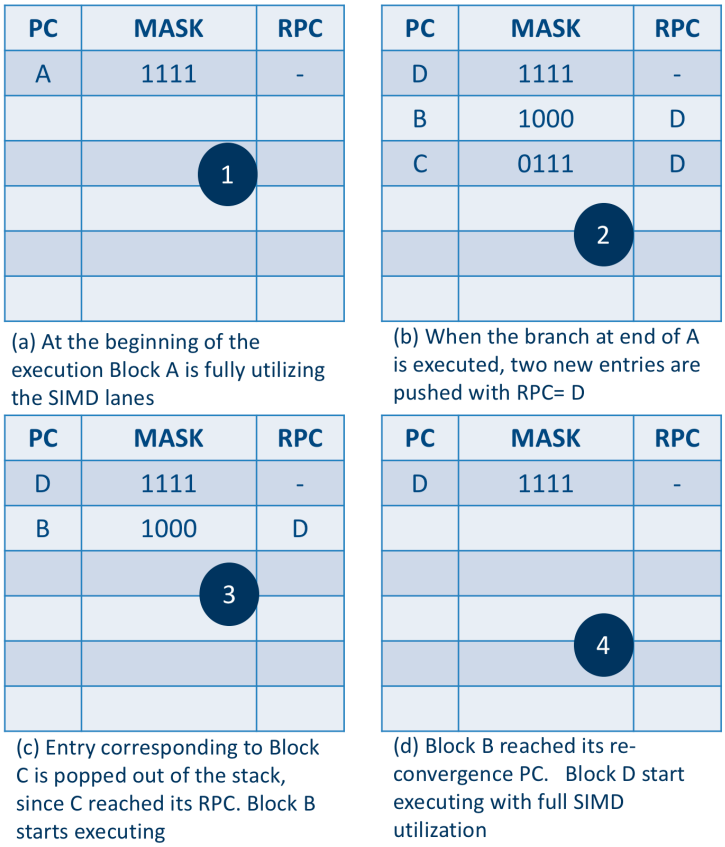


Figure 35: The different states of the SIMT stack

The RPC of both entries is set to the starting PC of block D. Next the execution moves to the path that is on top of the stack, block C in this case. Once current PC = RPC this entry is popped out of the stack and the execution of Block B starts (Figure 35-c). Once block B reaches the re-convergence point, block D starts executing and at this points all the threads are active.

4.3 Opcode-Convergent Threads

GPGPUs' SIMT execution model is ineffective for workloads with irregular control-flow because of the serial execution of divergent paths. In this work we propose a new fine-grained approach to exploit _intra-warp_ convergence. The key insight of this work lies in the observation that divergent basic blocks executed serially share a lot of their opcodes. This is due to two facts:

- First GPUs are RISC machines, **thus two basic blocks will inevitably have 'many' common instructions**, due to the limited number of instructions offered by the ISA. Note that even if these two blocks are performing completely different computation at a high level, their assembly version will have many common instructions, due to the large semantic gap between a high level language, CUDA in our case, and a RISC ISA
- Second, having one compiler applying the same set of optimizations and using the same code generation technique to generate both blocks, will result in many of the common instructions appearing in the same order. **Thus two divergent basic blocks will most probably have 'many' common sub-blocks.**

By serializing the execution of divergent blocks, the SIMT execution model is unjustifiably serializing the execution of these sub-blocks. In theory these sub-blocks can be executed in lockstep since they execute the same code on different lanes. Executing these common sub-blocks in lockstep will minimize the TLP loss caused by control divergence.

In order to execute these sub-blocks in lockstep we first need to identify these blocks. For that we wrote a compiler analysis that identify these common sub-blocks and annotate them. Moreover, we need some hardware modifications so that these common blocks access their corresponding operands as well as some modifications to the SIMT re-convergence stack.

Because this technique does not involve different warps to minimize the TLP loss, it is not expected to increase non coalesced memory accesses, an artifact that showed up in many previous works. Moreover, the suggested technique is also not expected to slow

down regular workloads since the compiler analysis will not identify any common sub-blocks in regular workloads; in other words, we end up executing the same old code. The suggested technique re-defines thread convergence from thread executing the same instructions to threads executing the same opcode. This new definition makes a given set of thread, less divergent. These 'less divergent' threads can still benefit from all previously suggested techniques for even less TLP loss. Hence this technique is complementary to all previously suggested solutions.

4.4 Compiler Support

In the following section, we explain the compiler we built to identify and annotate the common sub-blocks that lie within divergent basic blocks. The compiler that we developed is a source to source compiler that reads NVIDIA PTX assembly and generates a transformed/ or annotated PTX assembly.

- The compiler first builds the control flow graph (CFG) of the input code, where a CFG is a representation using graph notation of all the paths that might be traversed by a thread (check Figure 34).

- The second step is to identify the basic blocks that could potentially diverge at runtime. Two basic block may diverge at runtime and hence get executed serially, if they share the same immediate dominator (IDOM) and the same immediate post dominator (IPDOM).

  Where the *post dominator* of a node X is defined to be any node Y in the program such that every path from node X to the end of the program must go through node Y. Y is an *immediate post dominator* of X if Y is the unique node that does not post dominate any other post dominator of X. Similarly, Y *dominates* X if Y is any node in the graph such that every path from the start of the program to node X must go through node Y. A node Y *strictly dominates* a node X if Y dominates X and Y does not equal X. Y is an *immediate dominator* of a node X if Y is the unique node that strictly dominates X but does not strictly dominate any other node that strictly dominates X. Figure 34(a) shows that block B and block C share the same IDOM which is block A and share the same IPDOM which is block D. Hence threads could

potentially diverge at the end of block A, which results in serial execution of block B and block C. Note that if we have divergence within block B and/or block C the execution of these divergent blocks will be serialized within the execution of Block B and/or block C. Further discussion on nested divergence and how we deal with it can be found in section 4.9.

- Step three of our compiler is to identify the common opcodes between the divergent blocks. For that we use a widely known algorithm in computer science, the Longest Common Subsequence (LCS) algorithm. Where LCS is the problem of finding the longest subsequence common to all sequences in a set of sequences (two sequences in our case). The explanation of this algorithm is given in Appendix A. Figure 36(a) shows the CFG of Figure 34 but with the instructions of blocks B and C. Figure 36(b) shows the resulting CFG after running our compiler analysis.

Figure 37 shows the LCS table when the compiler works on finding the common instructions between blocks B and Block C. Note that the arrows shown in Figure 37 are used by a trace-back procedure that deduce the common subsequences between the two streams by following the arrows backward, starting from the last cell in the table. Each diagonal arrow in the table represents a common instruction and the numbers represent the size of the longest common subsequence.

By running this compiler analysis on irregular GPU benchmarks we found that the percentage of common instructions between divergent blocks could be as high as 85% as in the case of one set of divergent blocks in LU decomposition. More results are discussed in the results section.

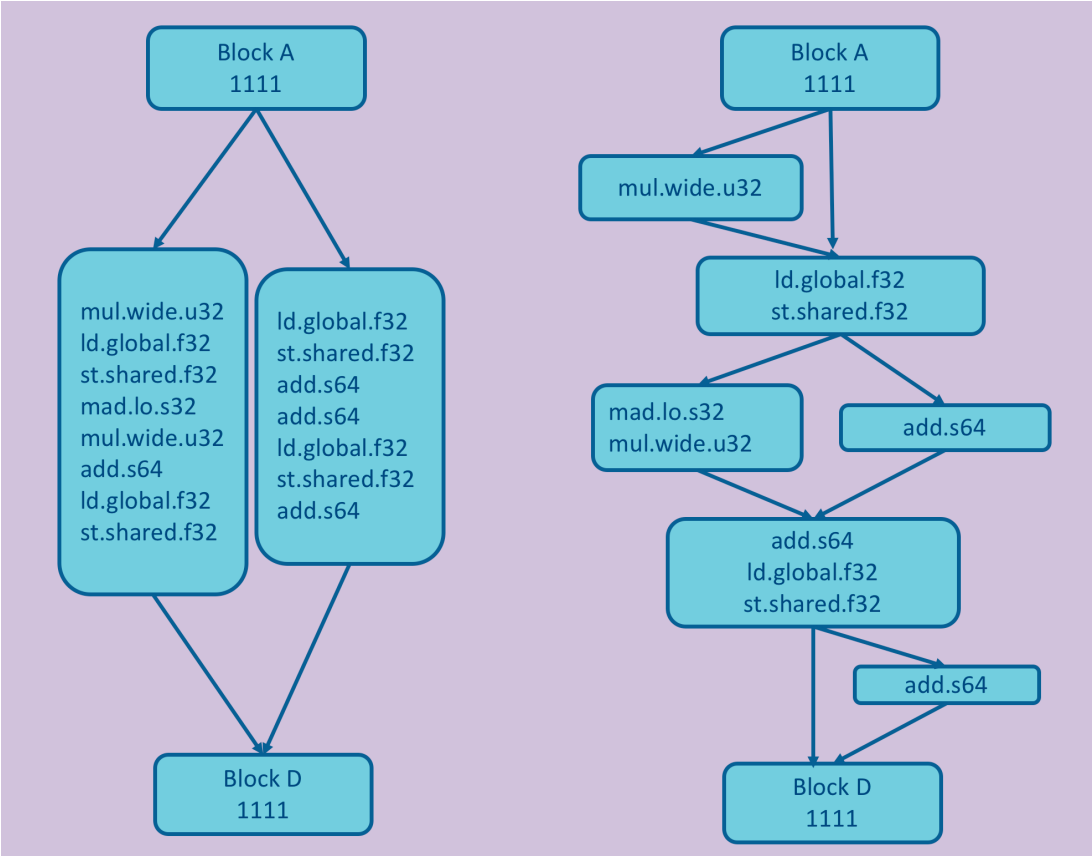
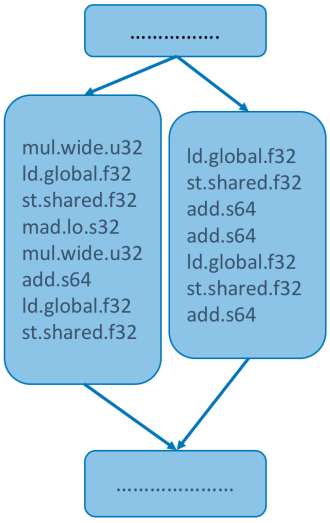


Figure 36: (A) CFG and its corresponding transformed CFG (B)



		Mul	Ld	St	Mad	Mul	Add	Ld	st
	0	0	0	0	0	0	0	0	0
Ld	0	0	1	1	1	1	1	1	1
St	0	0	1	2	2	2	2	2	2
Add	0	0	1	2	2	2	3	3	3
Add	0	0	1	2	2	2	3	3	3
Ld	0	0	1	2	2	2	3	4	4
St	0	0	1	2	2	2	3	4	5
Add	0	0	1	2	2	2	3	4	5

Figure 37: LCS table and its corresponding CFG

```
* st.shared.f32 [%rl2+ …//209
* add.s64 %rl54,… //210
* .loc 2 87 1 //211
* ld.global.f32 %f14, … //212
* st.shared.f32 [%rl2 … //213
* .loc 2 91 1 //214
* mov.u32 %r33, %c … //215
* shl.b32 %r34, %r33, 4; //216
* add.s32 %r35, %r34, %r261;
* add.s32 %r36, %r … //217
* mad.lo.s32 %r37
* add.s32 %r38, %r37 … //218
* .loc 2 93 1 //219
* mul.wide.s32 %rl55, %r38 //220
* add.s64 %rl56, %rl1, … //221
* ld.global.f32 %f15, … //222
* st.shared.f32 [%rl3 … //223
* add.s64 %rl57, %rl56, %rl40;
*
*
* add.s64 %rl58, %rl … //227
* .loc 2 93 1
* ld.global.f32 %f16, … //228
* st.shared.f32 [%rl3] …//229
* add.s64 %rl59, %rl58, %rl40;
*
*
*
* add.s64 %rl60, %rl58 … //234
* .loc 2 93 1
* ld.global.f32 %f17, … //235
* st.shared.f32 [%rl3 … //236
* add.s64 %rl61, %rl59, %rl40;
*
*
* add.s64 %rl62, %rl59 … //240
* .loc 2 93 1
* ld.global.f32 %f18, … //241
* st.shared.f32 [%rl3 …  //242
* add.s64 %rl63, %rl61, %rl40;
* add.s32 %rl64, %rl61 … //247
* .loc 2 93 1
*
* ld.global.f32 %f19, … //248
* st.shared.f32 [%rl3 …//249
* add.s64 %rl65, %rl63, %rl40;
*
* add.s64 %rl66, %rl63 … //253
* ld.global.f32 %f20 … //254
* st.shared.f32 [%rl3 …//255
* add.s64 %rl67, %rl65, %rl40;
* add.s64 %rl68, %rl65 …//259
```

```
* st.shared.f32 [%rl … //80
* add.s64 %rl95, %rl94,…  //81
* .loc 2 72 1 //82
* ld.global.f32 %f38, …  //83
* st.shared.f32 [%rl2 …  //84
* .loc 2 78 1 //85
* mov.u32 %r89, %ctaid.x; //86
* shl.b32 %r90, %r89, 4; //87
*
* add.s32 %r91, %r72, …  //89
*
* add.s32 %r92, %r91, 16; //91
* .loc 2 78 1 //92
* mul.wide.u32 %rl96, %r92, //93
* add.s64 %rl97, %r … //94
* ld.global.f32 %f39 …  //95
* st.shared.f32 [%rl4] …  //96
* add.s32 %r95, %r92, %r256;
* .loc 2 78 1
* mul.wide.u32 %rl98, %r95, 4;
* add.s64 %rl99, %rl1 …  //98
*
* ld.global.f32 %f40, … //100
* st.shared.f32 [%rl … //101
* shl.b32 %r98, %r256, 1;
* add.s32 %r99, %r92, %r98;
* .loc 2 78 1
* mul.wide.u32 %rl100, %r…
* add.s64 %rl101, %rl1 … //103
*
* ld.global.f32 %f4 … //105
* st.shared.f32 [%rl4+ … //106
* mad.lo.s32 %r102, %r2 ...
* .loc 2 78 1
* mul.wide.u32 %rl10 …
* add.s64 %rl103, … //108
*
* ld.global.f32 %f42 … //110
* st.shared.f32 [%rl4 … //111
* shl.b32 %r105, %r256, 2;
* add.s32 %r106, %r92, %r105;
* .loc 2 78 1
* mul.wide.u32 %rl104 …
* ld.global.f32 %f4 … //115
* st.shared.f32 [%rl4 … //116
* mad.lo.s32 %r109, %r2 …
* mul.wide.u32 %rl …;
* add.s64 %rl107,… //118
* ld.global.f32 %f4… //120
* st.shared.f32 … //121
* mul.wide.u32 %rl108, …
* add.s64 %rl109, % … //123
```

Figure 38: LCS applied to divergent blocks in Cholesky

4.5 ISA Support

As mentioned earlier, the opcode convergent blocks potentially need to access different operands based on their 'home basic block'. This issue can be solved through different techniques in software and/or in hardware. The technique we are suggesting in this work is to extend the ISA so that each instruction in the initial ISA has an equivalent instruction that accesses double the number of operands. In this way, the compiler can re-write the code using these instructions as shown in Figure 39 below. The common ADD instruction between the two divergent blocks is now replaced with an ADD that can read 4 operands and write to 2 different registers. Similarly, for the SUB instruction. We save the mask of the IDOM of divergent blocks in a special register that we call, the *IDOM register mask register*. This mask is used by each lane along with the SIMT stack mask to know whether it should access the first set of operands or the second set. Further discussion of the SIMT stack can be found in section 4.7.

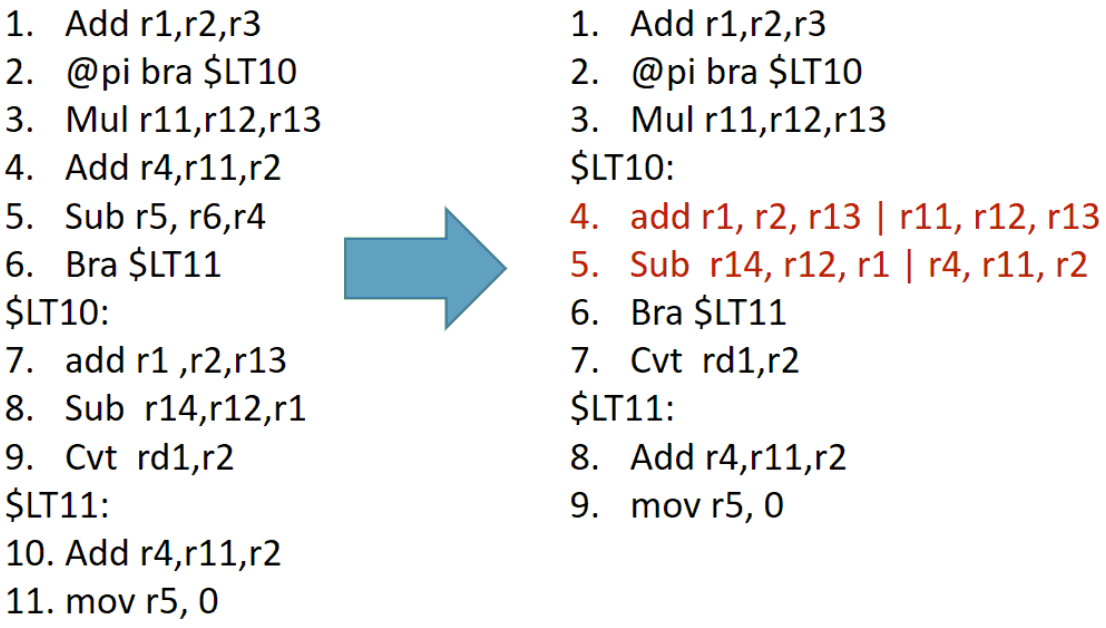


Figure 39: Original piece of code VS its transformed version

4.6 Scoreboard Modifications

In order to increase multithreading opportunity current GPUs allows a warp to issue instructions back to back. Issuing instructions back to back from the same warp requires tracking RAW and WAW data dependencies within each warp. GPUs use scoreboards to check for data dependencies. Scoreboards are usually implemented as Content Addressable Memory (CAM) structures like the one. The CAM structure is indexed using the warp ID and the register ID. Before issuing an instruction the scoreboard must be consulted to check for any RAW or WAW hazard. If no hazard exists, the instruction is considered for scheduling. Once scheduled, the scoreboard should be updated to show the destination register of the scheduled instruction as pending. Similarly, upon the completion of a write back the scoreboard entry for the corresponding register is cleared.

In our *opcode_convergent_thread* execution model, instructions may access up to six registers and not only three and definitely we must check all six dependencies before deciding whether an instruction is a valid candidate for scheduling or not.
Moreover, the dependencies should be tracked based on the original path that the register in question belongs to, otherwise we could be missing multi-threading opportunities.

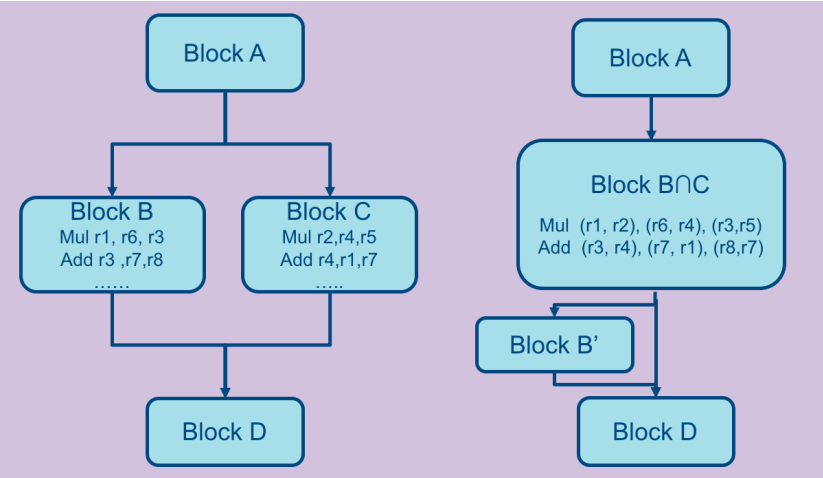


Figure 40: Transformed CFG shows a false data dependency

Figure 40 above helps making this last point clear. Register r1 and register r2 are written in the first instruction of the merged block (B∩C). The following instruction is an ADD and it reads r1. However, looking at the original code (Figure 40 A) we notice that r1 is written in original block B while it is read in original block C, so there is no real dependency.

If we track dependency using one scoreboard there is no way to Figure out this case and we may incorrectly stall block (B∩C) and lose multithreading, Hence the need for two different scoreboards. This design solution is similar to the solution used in Dual Path Execution [12] but it requires some extensions in our case as we will explain next. In our design we can identify three different types of basic blocks: the convergent basic blocks, the divergent basic blocks and the merged basic blocks. The scoreboard should be updated and checked differently based on the type of the basic block that the instruction in question belongs to. Following are the different scenarios for updating and checking the scoreboards and they are based on the type of the block that the instruction belongs to:

1. Instructions in a “Convergent block”:
    a. When writing to a register the corresponding bit should be set in both scoreboards because dependent instructions in both sides of the CFG needs to see this register as pending if the current block diverges before the write back is complete. Similarly, when a write back is done, the pending bits should be updated in both scoreboards.
    b. When scheduling an instruction from a convergent block it is enough to check one scoreboard since both scoreboards are identical at this point of execution
2. Instructions in a “divergent block”:
    a. When writing to a register only the scoreboard that belongs to the currently executing side of the branch should be updated, otherwise we may create a false dependency when the execution moves to the other side of the branch.

Similarly, we should only update one scoreboard upon the completion of a write-back. That being said, we should update both scoreboards upon re-convergence to a "convergent block" as follows. Instructions executing after re-convergence must see the pending writes from both divergent flows that re-converged, hence we must copy all the set bits in the first scoreboard to the second scoreboard and vice versa.

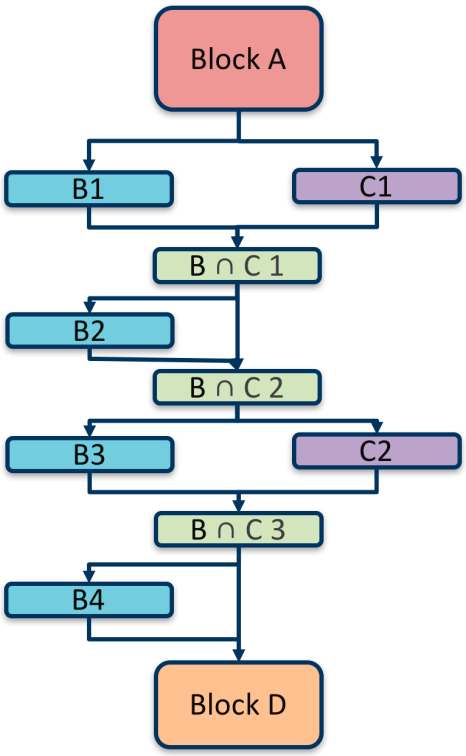
b. When scheduling an instruction from a divergent block we should check the scoreboard that corresponds to the side of the branch that we are currently executing

3. Instructions in "merged block"
    a. Merged instructions access different registers based on the side of the branch they belong to in the original CFG. That being said, upon a write both scoreboards will be updated, however it is not the same update. Each scoreboard will set the register bit of the register that was written on the side of the branch that corresponds to this scoreboard. When a write-back is done only the corresponding scoreboard has its corresponding register bit reset. By updating register in this way we avoid false dependences. Once again, like in the case of divergent blocks, we have to unify both scoreboards upon re-convergence.
    b. When scheduling an instruction from a "merged block" both scoreboards will be checked but each will check only the registers that its corresponding "divergent block" in the original CFG would have checked

4.7 SIMT Stack Modifications

Upon merging the divergent paths using our compiler analysis, we could have instructions interleaving the 'opcode convergent blocks' i.e, instructions that are not common to both divergent paths. For example, the multiply instruction in Figure 36(b) which is only needed by the left path. One would think that due to such non-common instructions the compiler must add branch instructions to the transformed code so that these non-common instructions gets executed just by their corresponding lanes. While

this will definitely work, the added branch instructions are not needed and they will cause an extra overhead.



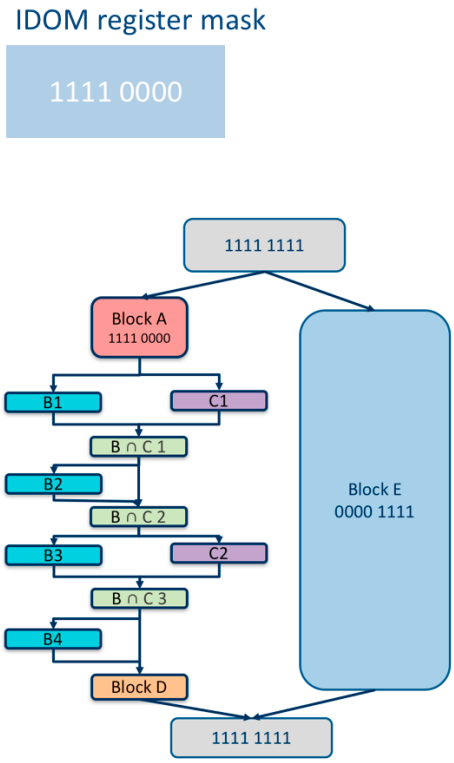
PC	MASK	RPC
D	1111	-
B4	1110	D
B∩C 3	1111	D
C2	0001	B∩C 3
B3	1110	B∩C 3
B∩C 2	1111	B3
B2	1110	B∩C 2
B∩C 1	1111	B2
C1	0001	B∩C 1
B1	1110	B∩C 1

Figure 41: The modified SIMT stack upon divergence at end of block A

The key insight here is that all these added branch instructions will resolve like the original conditional branch for any given thread. Hence we suggest a small modification to the SIMT re-convergence stack to take advantage of this observation. Upon divergence the SIMT stack will not push only two entries like in the case of the traditional SIMT stack described above, instead it will push the whole control flow that lies between the divergent branch and its IPDOM in the original CFG. So in case of the CFG shown in Figure 41(A), the SIMT stack upon executing the divergent branch at the end of block A will look as shown in Figure 41(B).

Merged instructions should be told which set of operands to use. One easy solution is to add the mask of IDOM of the diverging branch into a special register Figure 42. And then use the mask in the SIMT stack to select the operand set for each lane. If the lane has a zero bit in the “IDOM register” mask then this lane should not be executing otherwise, a 1 means that this lane should be active. A zero in the SIMT stack mask

refers to accessing operand set number 1 and a 1 in the SIMT stack refers to accessing operand set number 2.



PC	MASK	RPC
D	1111 0000	-
B4	1110 0000	D
B∩C 3	1110 0000	D
C2	0001 0000	B∩C 3
B3	1110 0000	B∩C 3
B∩C 2	1110 0000	B3
B2	1110 0000	B∩C 2
B∩C 1	1110 0000	B2
C1	0001 0000	B∩C 1
B1	1110 0000	B∩C 1

Figure 42: Re-convergence stack of *opcode_convergent_threads*

For example, consider Figure 42, where we have eight threads per warp. Only the first 4 lanes execute block A upon the first divergence. When block A starts executing and we diverge again at the end of block A and we push the whole entry in the stack as shown in Figure 42, the Mask of block A should be saved in the IDOM mask register. Now when the merged instructions of block B ∩ C 1 starts executing the mask is 1110 0000 but the zero of lane 4 does not mean must be idle but it means that lane four must execute the merged block while accessing the first set of operands while lanes 5 through 8 should be idle and lanes 1 through 3 should execute the merged block accessing the second set of operands. In order to know that the 0 in the mask corresponding to lane 4 should be treated differently than the zeros of lane 5 through 8 we compare the mask to the IDOM mask register. in the IDOM mask register lane 4 has a 1 bit while lane 5 through 8 has zeros.

Note that, if the SIMT stack gets too big because of pushing the whole control flow that lies between the IDOM and the IPDOM of a divergent branch the compiler can decide to insert explicit branch instructions. Each explicitly inserted branch if inserted at the middle of the entry can divide the number of needed entries by 2.

4.8 Register File Modifications

The register file modifications are minimal. We just increase the size of each collector unit so that they can hold up to 6 values corresponding to the 6 operands that are accessed by a merged instruction.

4.9 Nested Divergence

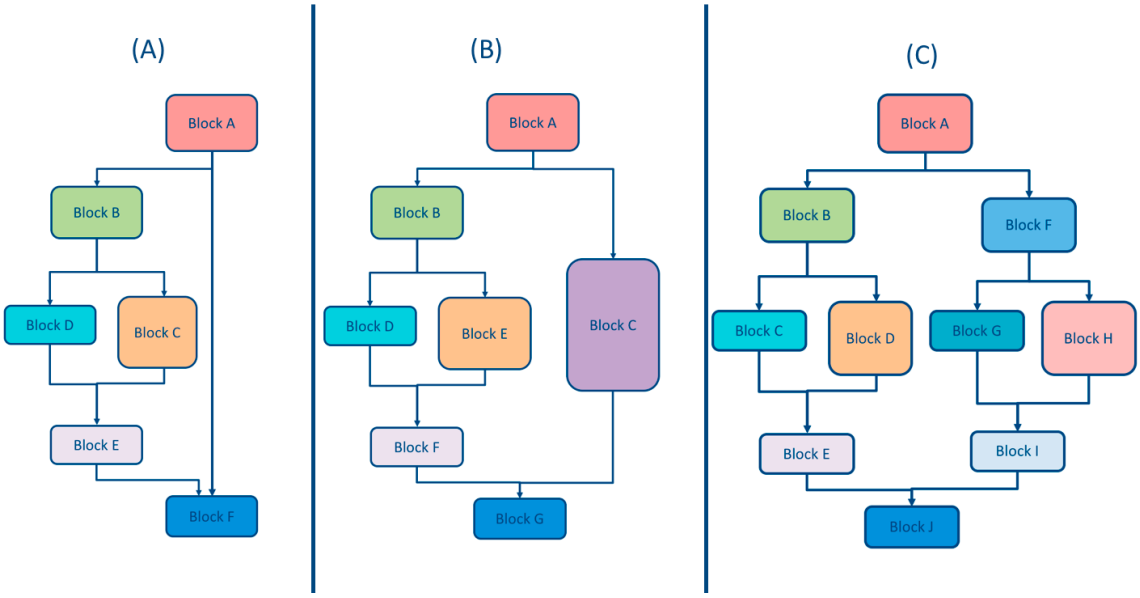


Figure 43: Nested Divergence Scenarios

It could happen that divergent paths are nested within other divergent paths as shown in the Figure 43. Merging instructions from two divergent paths required extending each instruction in the ISA with an extra operand set in order to accommodate two paths. Similarly, if we decide to merge instructions from more paths, in case of nested divergence, the ISA must be further extended to accommodate the newly merged paths. For example, if we merge instructions from blocks D, E, and C in Figure 43-B these

merged instructions must offer three operand sets to accommodate the three merged paths.

After examining the benchmarks listed in section 5.2, we found out that the occurrence of nested 2-way branch structures within 2-way branch structures is rare thus, merging instructions from more than two paths will further complicate the ISA and eventually the microarchitecture without offering major benefits in term of performance. Thus, we decided not to merge instructions from more than two paths. However, we still have to deal with nested divergence and the compiler must choose which paths to merge. Consider Figure 43-B, since we are using one *IDOM-mask register*, the compiler must either merge block D with block E and in this case the *IDOM-mask register* would save the mask of block B, or the compiler must merge blocks B and F with block C and in this case the *IDOM-mask register* would save the mask of block A.
The compiler simply decides between these two options based on the number of instructions that could be potentially saved by each merging scenario.

Note that in case the number of potentially saved instructions is equal the compiler prioritize merging at the outer level (i.e. blocks B and F with block C) because there is a higher probability that threads diverge at the outer level since the number of threads at the outer divergent branch is greater than or equal to the number of threads at the inner divergent branch.

Note that in case we want to merge both the outer and the inner divergent blocks we can either save two IDOM masks, namely the mask of block A and the mask of Block B or we can simply add an extra mask to the SIMT re-convergence stack that helps each lane figure out which operand set it must access.
Regarding Figure 43-C the compiler compares the number of saved instructions if blocks C and D are merged together and blocks G and H are merged together with the number of instructions saved if blocks B and E are merged with blocks F and I. Similarly to figure 43-B, the compiler prioritize merging the outer divergent blocks over the inner divergent blocks in case of a tie.

5. OPPORTUNISTIC INTER-PATH RE-CONVERGENCE IMPLEMENTATION AND RESULTS

5.1 Methodology

Implementing this idea required the use of three tools: the compiler we built and explained in section 4.4, a modified version of GPGPU-sim, and finally, a trace-based simulator that we built to time the execution of the annotated code. Following we explain the methodology used and how these three tools interacts together.

First, our source to source compiler reads the PTX assembly code generated by NVCC and generates a PTX code with opcode convergent instructions annotated as described in section 4.4.

Second, GPGPU-sim reads the annotated PTX and executes the application. GPGPU-sim is modified to generate a detailed time trace for each instruction running in each warp. By detailed time trace we mean the time each instruction spends in each pipeline stage.

Third, our trace base simulator reads the traces generated by GPGPU-sim in stage two and re-simulates the timing by merging the annotated instructions. This trace base simulator simulates the round robin warp scheduler present in GPUs and GPGPU-sim, it models the three pipelines (SP, SFU and MEM) that are present in GPGPU-sim as well as the operand collector.

This trace base simulator was tested with un-annotated traces and it generated exactly the same timing as GPGPU-sim.

Since the trace-based simulator does not simulate the register file we decided to be conservative in timing the register file accesses. In case of running an operand convergent instruction, the simulator waits for the 2 sets of operands as if they are accessed sequentially. For example, if a merged ADD instruction initially waits for 20 cycles to collects its operands when it is executed in the left path and 15 cycles to collects its operands when executed in the right path. The merged version of this ADD instruction will have to wait for 20+15 = 35 cycles for its operands to become ready. This assumption is conservative, because the register file is banked and the access of both sets could be overlapped.

Since the trace-based simulator does not simulate the memory system we decided to be conservative and not merge memory instructions. Memory instructions timing is exactly the same as that in GPGPU-sim since these instructions are never annotated. Note that this is conservative because if loads were merged we could amortize the cost of fetch, decode and in some cases the loads could be coalesced.

5.2 Benchmarks:

We used a selection of benchmarks from the following benchmark suites: Parboil [19], Rodinia [17], the benchmarks given with GPGPU-sim [18] and some irregular task parallel benchmarks from [27]. The benchmarks were selected based on the ability of GPGPU-sim to execute them to completion.

Name	Description	Ref
BFS	breadth first search	[17]
BH	Barnes Hut	[27]
Cholesky	Cholesky Decomposition	[27]
FMM	Fast Multipole Method	[27]
LPS	Laplace Solver	[18]
LUD	LU decomposition	[17]
NW	Needlemen Wunsch	[17]
NN	Nearset Neighbour	[27]
PC	Point Correlation	[27]
Stencil	3D Stencil operation	[19]

The studied benchmarks can be divided into four categories:

- Regular benchmarks like FMM. These Benchmarks do not exhibit any control divergence.
- Irregular benchmarks with empty non-taken path like BFS and NW.
- Irregular benchmarks with two-path branch structures but that rarely diverge in runtime like NN, PC, and BH.

- Irregular benchmarks with both taken and non-taken paths non empty and which actually diverges at runtime like Stencil, LPS, LUD, and Cholesky.

5.2 Experimental Results

The goal of opcode-convergent threads is to decrease the TLP loss upon divergence. It does so by taking advantage of common opcode sequences used in both the taken and non-taken paths of a branch. The applications studied were first analyzed statically to check the potential of merging instructions. As expected, regular applications or irregular applications with one-path branch structure, like BFS, did not show any potential for merging. However, irregular applications with two-path branch structure showed potential for merging (Figure 44). Note that these static numbers may or may not actually translate to speedup upon execution because threads in a warp may not diverge at runtime. In other cases, a low percentage of overlap may translate to high speed-up in case the divergent blocks happen to be wrapped inside a loop. Hence the role of the static analysis numbers reported in Figure 44 is just to prove the validity of the observation mentioned in section 4.3 above: divergent blocks, even if performing completely different computation, will have substantial percentage of common sub-sections when considered at assembly level.

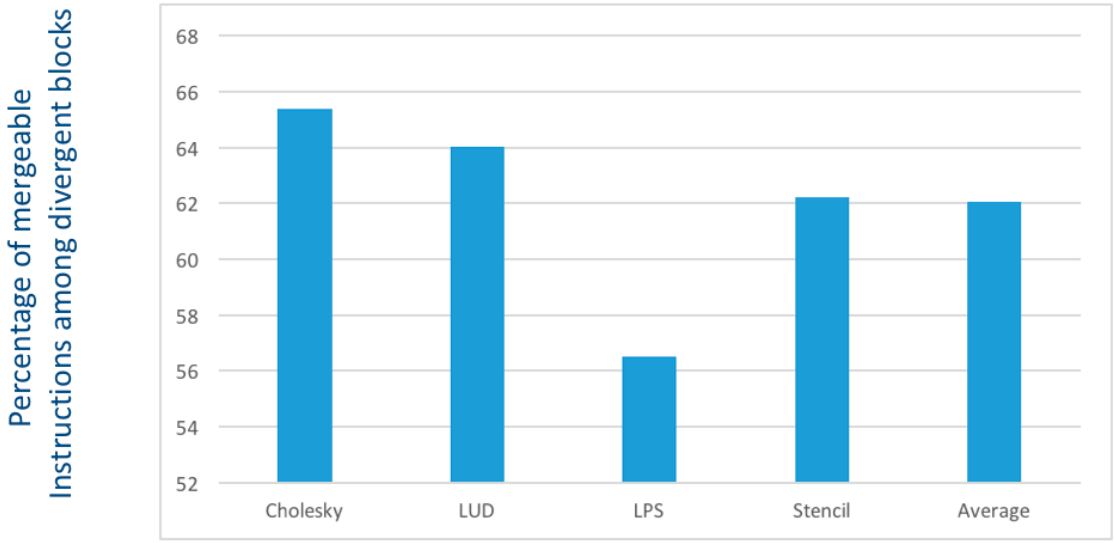


Figure 44: Static analysis: Percentage of mergeable instructions

Among the studied irregular applications, Cholesky and LUD showed the highest percentages of common opcode sequences among their divergent blocks. The highest percentage was 85% in one set of divergent blocks in Cholesky, however the overall overlap among all the divergent blocks within Cholesky was 65%. Note that Barnes Hut had one set of potentially mergeable divergent blocks with a 62% overlap, however at runtime Barnes Hut did not show any speed-up (around 1%) because the threads rarely diverged.

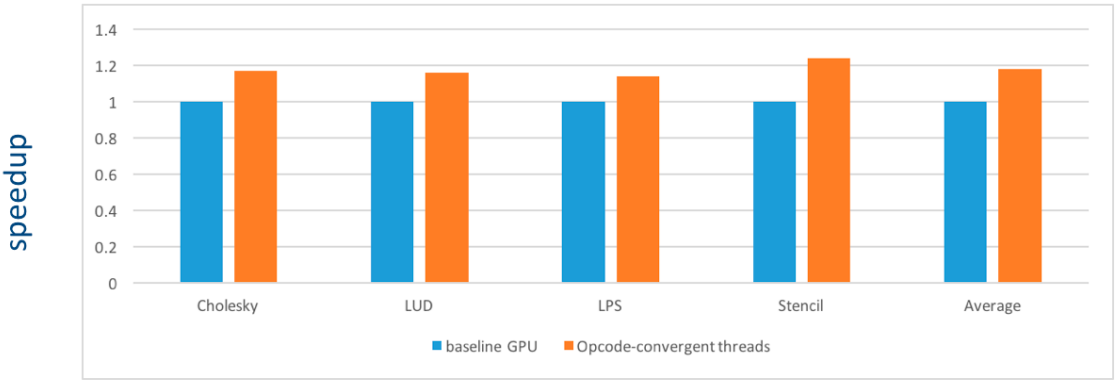


Figure 45 : Speedup upon exploiting opcode convergence

Figure 45 Shows the performance gains due to merging instructions from divergent paths. On average the speedup for irregular applications with 2 path branch structure is 17% while we did not see any slowdown for regular applications or irregular applications with 1 path branch structures. The highest speedup was seen with stencil and it was 22%. These speedup numbers are prone to increase if the register file is modified so that the access to the two sets of operands required by a merged instruction is allowed to overlap. Figure 46 shows the number of saved instructions and on average we saved 21% on the number of executed instructions.

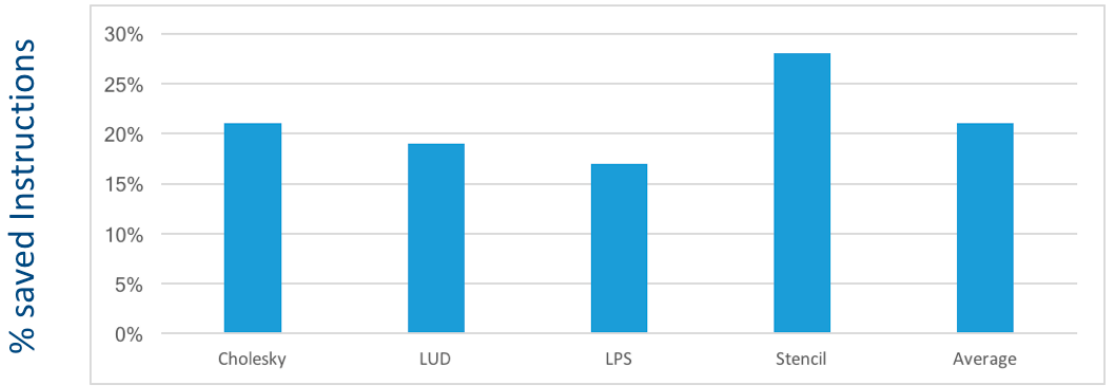


Figure 46: Percentage of saved instructions

The speedup we got is mainly due to executing fewer instructions after merging similar opcodes from divergent basic blocks. However, the new scheduling resulting from the merging of common sub-blocks and serializing the execution of non-common sub-blocks increases the level of parallelism available for the warp scheduler and thus could be contributing to the speedup. It is known that because divergent blocks are independent, interleaving their execution increases parallelism and performance. The idea was studied extensively by [12] and proved to be effective. While DPE interleaves the execution of divergent blocks at the granularity of instruction and the traditional GPU "interleaves" the execution at the granularity of the divergent basic blocks, a subsequence of our execution model results in interleaving the execution at the granularity of "sub-block" which increases the level of available parallelism to the warp scheduler compared to the state of the art GPUs. Every time the execution moves to the complementary side of a branch it is highly probable that the warp scheduler finds a ready instruction waiting to be scheduled since the data dependencies will be met by then. Since our execution model results in more frequent jumping between the two sides of a divergent branch the scheduler will have more opportunities at scheduling instructions and hiding latencies. For that we measured the percentage of idle cycles where the scheduler could not find an instruction ready to issue and we saw a 3% improvement in the number of these idle cycles compared to executing the same application under the traditional SIMT execution model.

5.3 Conclusion

This work presents a more fine-grained approach to exploit *intra-warp* convergence: rather than threads executing the same code path, *opcode-convergent threads* execute the same instruction, but with potentially different operands. Based on this new definition we find that divergent control blocks within a warp exhibit 62% opcode convergence. We build a compiler that analyzes divergent blocks and identifies the common streams of opcodes. Moreover, we suggest modifications to the ISA, the SIMT stack and the scoreboard to handle the execution of opcode convergent threads. Using software simulation, we achieve a 17% speedup over baseline GPU for irregular workloads and do not incur any performance loss on regular workloads. The proposed technique is complementary to all previously suggested techniques that deals with control divergence. It does not increase the memory footprint of the warps which is a drawback of many previous works. Simulation showed that opcode_convergent_threads ends up executing 21% less instructions compared to baseline SIMT execution model. Moreover, merging divergent basic blocks gives the warp scheduler more scheduling opportunities which boost thread level parallelism. On average the number of idle issue cycles decreased by 3%

6. LOCALITY-AWARE TASK-PARALLEL EXECUTION ON GPUS

6.1 Introduction

GPGPUs have proven themselves to be a cost-effective way of accelerating applications. The single-instruction, multiple-thread (SIMT) execution model of GPUGPUs provides massive amounts of parallelism while remaining energy efficient. The hardware of the GPU is limited to keep power consumption low. Most prominently, the SIMT execution model requires that all threads in a warp perform the same instruction at the same time to enjoy parallelism; if different threads do different work, some threads sit idle and parallelism is lost. Second, there is relatively little hardware support for hiding latency---the core cannot execute instructions out of order, nor are there forwarding networks to help mitigate the penalty of long-latency instructions---instead, the GPU relies on massive multithreading to hide latency, keeping hundreds or even thousands of threads in context to swap in and out during long-latency operations. Perniciously, this means that not only does the GPU support massive parallelism, it needs massive parallelism for effective execution. Finally, caches are small compared to their CPU counterparts, especially when considered on a per-thread basis (thousands of threads spread a 1 MB cache very thin!)

As a result of these limitations, not all applications can execute efficiently on GPUs. GPUs are well-suited to regular, data-parallel applications, where the well-structured computation is performed on different pieces of data. In these applications, the similarity of the computations performed in parallel means that there are not substantial penalties for executing in a SIMT manner. Moreover, the data-parallel nature of the application means that it is easy to generate enough parallelism to fill the GPU, allowing for effective multithreading.

However, for task parallel applications, where parallelism arises from independent, distinct tasks that run simultaneously, GPUs are not nearly as attractive a target.

- First, the fork-join nature of task parallelism does not map well to standard GPU programming models such as CUDA [32] or OpenCL [30].
- Second, even if there is data parallelism in these applications (either because individual tasks have data parallel work, or because there is some parallel outer loop in the application), there may not be enough parallelism to effectively use the GPU's resources.
- Third, even if the tasks of the program could be mapped to the GPU, the limited memory subsystem of the GPU can lead to poor performance in data-heavy tasks.

There has been recent work on turning task parallelism into data parallelism to map task-parallel applications to hardware (including GPUs) that is made for data parallelism [26, 33, 34]. These proposals either require hardware changes [26, 33] or target fine-grained data parallelism in SIMD units [34]. None of these approaches consider locality.

In this work, we propose a locality-aware, task-queue abstraction for mapping task parallel applications to GPUs. The basic approach is to expand task parallel work on the CPU to generate a large number of tasks. These tasks are then inserted into one or more task queues according to the type of computation they perform and, crucially, the locality properties of the tasks. These queues are then merged into a single queue that is sent to the GPU, where they are executed in a data-parallel manner, with each task executing to completion on the GPU. This model has several features. First, by expanding out the task-parallel work on the CPU, we avoid needing to handle task-parallelism on the GPU; instead, once execution begins on the GPU, it is purely data parallel. Second, because the task queues are partitioned based on operation type, the tasks that execute simultaneously are computationally similar, promoting efficient SIMT execution. Finally, the locality-aware nature of the queues promotes tasks in the same queue having overlapping memory footprints, reducing cache pressure and hence improving performance relative to a locality-unaware approach.

We evaluate this queue abstraction on three applications: a task-parallel implementation of the fast multipole method, and two data mining applications that

feature a mix of data-parallelism and task-parallelism, nearest-neighbor and two-point correlation. In all three cases, we demonstrate that our locality-aware approach delivers better performance than a locality-agnostic one. For the mixed applications, not only do we show that our locality-aware approach is better than the locality-agnostic approach, but we also show that in the absence of very large amounts of data parallelism (for example, only 200,000 data-parallel iterations), our task-parallel approach, by exploiting additional parallelism, is also significantly faster than the best-available implementations, which exploit only data parallelism.

The remainder of this chapter is organized as follows.

Section 6.2 discusses some previous works that considered different programing models for GPUs.

Section 6.3 provides background on GPGPU programming and task parallelism.

Section 6.4 discusses our basic task-queue-based technique for exploiting data parallelism in task-parallel applications.

Section 6.5 discusses the need for locality aware scheduling of subtasks. Section 6.6 discusses the implementation, Section 6.7 evaluates our system on the four applications mentioned above, and Section 6.8 concludes.

6.2 Related Works

Due to the increasing prevalence of hardware resources for data parallelism (GPUs, SIMD units, etc.), there has been significant recent interest in techniques for mapping task-parallel computations to data parallel hardware. Gaster and Howes propose a abstraction for executing Cilk-style task-parallel programs on GPUs, where the GPU hardware manages queues for each type of task, and provides support for de-queueing and in-queuing new tasks [26]. Orr et al. presented an instantiation of the channels model and showed its efficacy on several small Cilk-style programs [33]. Both of these approaches require hardware support, and hence are not suitable to executing task-parallel programs on commodity data-parallel hardware. Moreover, the channels abstraction does not consider locality between tasks; it only concerns itself with grouping together tasks with similar computation.

More recently, Ren et al. described a series of code transformations that transform task-parallel algorithms into recursive algorithms: recursive algorithms where each method invocation performs a block of tasks, rather than a single task [34]. These blocks can be executed efficiently in a vectorized manner. While Ren et al.'s general approach---transforming independent, parallel tasks into data-parallel blocks of tasks---is similar to ours, their technique does not apply in our setting
for two reasons:

1) they target vector units on CPUs, and hence can support code transformations that require fine-grained interleaving of SIMD and scalar operations;

2) more importantly, the applications they study only manipulate the stack, and hence their technique does not have to account for locality considerations.

In a more general sense, models for executing work-queues on GPUs have been studied extensively in the literature. The persistent threads model [28] proposes maintaining a CPU-managed work-queue along with a specially-designed GPU kernel where a limited number of threads each run a simple get-work/execute-work loop until the software-managed queue is empty. This style of programming can be conducive to expressing idioms, such as producer-consumer dependences, that data-parallel programs are ill-suited to capture. This model has been used to implement several work-queue-style applications [23, 31]. For the most part, persistent-thread applications do not consider locality in mapping tasks to threads. Moreover, unlike in a persistent thread model, our approach does not attempt to limit the number of threads executing on the GPU; instead, the entire queue of tasks is sent to the GPU at the same time, to maximize the effectiveness of hardware multithreading.

Chen at al. do consider locality concerns in a persistent-thread-like model [24]. In their programming model, tasks in a task-queue can generate new tasks that may operate on similar data to the parent tasks. They use a compiler-based code transformation to map child tasks to the threads that executed the parent task, to promote reuse. In contrast, our

approach considers locality between the tasks that are mapped to the same warp---in other words, locality between tasks that are executed by different threads, rather than consecutively on a single thread---in an attempt to improve memory coalescing and minimize cache pressure. Wu et al. perform affinity scheduling, mapping tasks with overlapping footprints to the same Streaming Multiprocessor (SM) [36]; this notion of locality is far more coarse-grained than the warp-focused affinity scheduling we pursue. Moreover, neither Chen et al. nor Wu et al. focus on the type of task-parallel applications that we tackle.

Goldfarb et al. looked at mapping tree traversal applications to the GPU, as with two of our example benchmarks [27]. They adopt a fully data-parallel approach, meaning that their implementation relies on inputs with a large number of traversals to obtain good performance. Moreover, they do not consider locality between threads, relying on ad hoc, programmer-provided scheduling decisions. Liu et al. expanded on this work by developing a hybrid scheduling framework that attempts to reschedule traversals based on similarity [29]. As in Goldfarb et al.'s work, Liu et al. only consider fully data-parallel implementations, and rely on high degrees of data parallelism for their scheduling to be effective. Their implementations, which represent highly optimized GPU implementations of tree traversal applications, form the baseline for our experiments.

6.3 Background and Motivation

6.3.1 GPU Architecture and Limitations

A typical GPU consists of multiple streaming multiprocessor units (SMs), each of which features multiple simple cores, a register file, an L1 cache, and a shared memory used by threads within the same thread block to communicate (in NVIDIA GPUs, the shared memory and L1 cache share the same hardware structure, which can be partitioned between the two in different ratios, depending on the workload). There is also a shared L2 cache among all the SMs on the GPU. Finally, there is an off-chip, global memory accessible by all the SMs.

Execution on a GPU consists of a kernel that is expressed in terms of a thread grid ---a set of threads that execute in parallel to complete the kernel. The thread grid is executed across one or more of the SMs on the GPU. To facilitate this execution, the threads in the grid are partitioned into multiple thread blocks. While different thread blocks may execute on different SMs, all the threads in a single block are guaranteed to execute on a single SM. As a result of this thread partitioning, threads in a thread block can communicate through shared memory, but threads in a grid can only communicate through global memory.

Within a thread block, execution proceeds by dividing the block into warps: groups of 32 threads that execute simultaneously on the SM's 32 cores. These threads execute in a lockstep, SIMT (single instruction multiple thread) manner, for efficiency: all threads must be executing the same instruction for computations to be performed in parallel, and if some threads want to execute different instructions, some of the 32 cores sit idle until the threads in the warp return to executing the same instruction. Paired with this control divergence is memory divergence: if multiple threads in the warp issue a load, all the threads must wait until all of the loads complete before proceeding. Hence, if some loads miss in the cache, the entire warp can stall for a long time before resuming execution.

As a final complication, GPU cores are extremely simple---in order, no forwarding networks, no branch prediction, etc. Instead, performance is maintained in the face of pipeline stalls and memory divergence through massive multithreading: NVIDIA's latest Kepler GPUs can keep up to 64 warps (2048 threads) in context at the same time, and will context switch between these warps on stalls. Note that this massive multithreading means that a GPU's memory system is much smaller than a CPU's on a per thread basis: a CPU hardware context (core) has access to a 64 KB private L1 cache, and a ~ 1 MB L2 cache and ~6 MB L3 cache shared among 4-12 cores. In contrast, an SM's 16-64KB of L1 cache is shared among up to two thousand threads, and its ~1.5 MB of L2 cache is shared among all of the SMs in the system. On the flip side, while the

amount of memory per thread may be small, the GPU features extremely high throughput to keep the threads fed.

These hardware features conspire in destructive ways when writing programs that do not have very well-structured computation and memory accesses:

- A GPU needs large amounts of parallelism to sustain throughput under memory stalls.
- Memory stalls are more likely due to the SIMT architecture if different threads in the same warp have divergent memory footprints, as each SIMT-coalesced memory access is more likely to result in cache misses.
- These memory stalls require even more parallelism to hide the resulting latency, which places even more pressure on the memory system.

Unsurprisingly, then, while GPU implementations attain extremely high speedups over CPU implementations for data-parallel, regular applications, as programs become less regular and less data-parallel, speedups become harder to attain. This work describes one approach to achieve good speedup for a class of irregular, task-parallel applications.

```
1.  corr(KDNode n, Point p, float r) {
2.  if (!canCorrelate(n, p, r)) return;
3.  else
4.       if (n.isLeaf && dist(n, p) < r)
5.            p.count.accum(1);
6.       else if (!n.isLeaf){
7.            spawn corr(n.left, p, r);
8.            spawn corr(n.right, p, r);
9.       }
10. }
```

Figure 47: Task-parallel implementation of point correlation

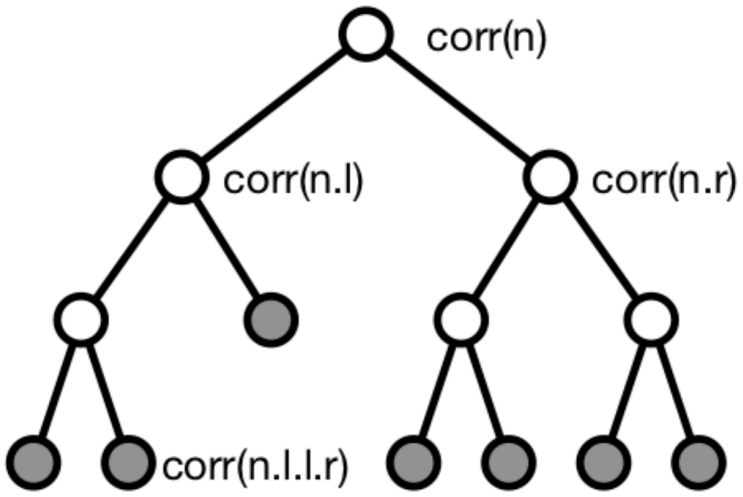


Figure 48: Computation tree for point-Correlation

6.3.2 Task Parallelism

In this work, we consider mapping task-parallel applications on GPUs. In particular, we consider recursive and task-parallel applications where parallelism arises from executing multiple recursive function invocations simultaneously. Figure 47 shows a task-parallel implementation of a recursive algorithm to compute two- point correlation. The algorithm takes a point p and traverses a kd-tree structure to determine how many points in a metric space are within a specified radius r of that point. Because the kd-tree is a binary tree, each subtree can be searched independently, as indicated by the use of the spawn keyword (we borrow the keyword from Cilk [22], perhaps the most well-known task-parallel programming languages). Following the approach of Ren et al. [34], rather than using sync and return values to perform the final correlation computation, we instead accumulate into a Cilk-style reducer [25]. In section 6.4, we explain how we use this reduction approach in our implementations.

Two-point correlation has substantial amounts of parallelism, but, nevertheless, is poorly suited to mapping to a GPU: the parallel operations do not arise from a data parallel loop; the parallel operations have very different memory footprints; and the computation itself is highly irregular. While some data-parallelism does arise because this task-parallel

computation can be repeated for different points, there may not always be enough data parallelism to provide the parallelism the GPU requires. The best-available GPU implementation of point correlation [29] relies on massive data parallelism to effectively manage the irregularity of the computation.

6.4 Data Parallel GPU Execution of Task Parallel Code

This section presents our basic technique for extracting data parallelism from task-parallel programs. It shares some basic similarities with the approach of Orr et al. [33] and Ren et al. [34], in that it "expands" out the task parallel work to generate enough tasks that can subsequently be executed in a data parallel manner. Unlike the prior two approaches, though, our approach focuses on cooperation between the CPU and GPU to generate the necessary tasks.

6.4.1 Basic Technique

The key insight underpinning our approach is that the execution of a recursive, task-parallel program with no syncs can be viewed as an execution tree: a pro- gram begins at the root of the tree, and at each spawn call generates two leaves: one for the spawn, and one for the continuation. If there is no continuation (i.e., the spawn is the last operation in the function), then only one leaf is generated. Hence, in a code like the point correlation code of Figure 47, non-base-case executions generate interior nodes of the tree with two leaves each, for each of the spawn calls, and base case invocations of the corr method create leaves of this execution tree. Figure 48 shows an example of the execution tree, with base-case tasks shaded gray. Some of the nodes are labeled with the node argument passed to the task. Note that the nature of the point correlation algorithm means that the execution tree corresponds to the actual tree that is being traversed: each method invocation corresponds to operating on a subtree of the overall kd-tree. This connection will become pertinent in Section 6.5.

Because of the nature of task-parallel execution, the nodes in this computation tree can be executed in any order, provided that ancestors always execute before descendants. As nodes are executed, the "frontier" of non-executed nodes represents the

set of tasks that are left to be executed (so, for example, in a Cilk-style runtime system, the contents of the threads' deques are this frontier).

Our execution model, then, is straightforward. Figure 48 illustrates the steps.

1. Partially expand the computation tree on the CPU, until a sufficient frontier is generated. (The nodes inside the red rectangle of Figure 49)

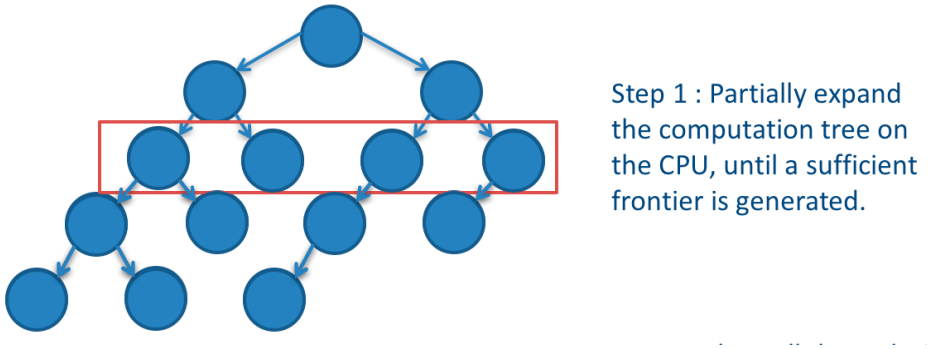


Figure 49: Partial expansion of the computation tree by the CPU

2. Place all the tasks in this frontier into a queue (shown in step 2 of Figure 50). Note that each of these tasks, by definition, is independent from the others. Moreover, each task can be executed to completion sequentially, executing the entire subtree rooted at that task.

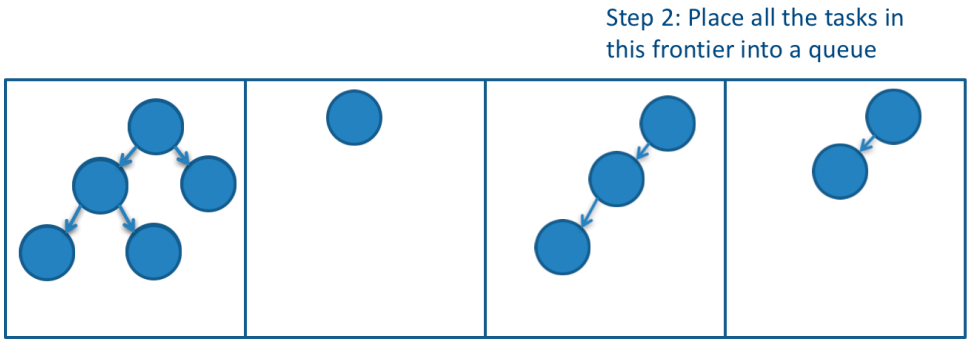


Figure 50: Different tasks are assigned to different queues

3. Execute this task queue in a data parallel manner on the GPU, with each task maintaining its own reduction result. Because of the nature of reduction computations [24], each of these tasks can perform its reductions independently.

4. Return the reduction objects to the CPU to be combined to produce the final result.

Note that because of the recursive nature of the tasks in the applications we consider, each of the tasks has a fairly similar computational fingerprint. This similarity between tasks helps reduce control divergence. Memory divergence, though, is another issue, as Section 6.5 elaborates.

6.4.2 Generating GPU Task Queues

The process of enqueuing tasks into the task queue for execution on the GPU is straightforward, and can be accomplished by a basic code transformation. Because spawned tasks are independent of their execution, it is also safe to not execute them, and instead defer their execution until a later point in time. Hence, during sequential execution of a task-parallel program on the CPU, whenever a threshold is hit, spawned tasks are not executed, but are instead enqueued onto a task queue that will be sent to the GPU. Figure 51 shows a version of our point correlation example that performs this enqueuing.

```
1.  corr(KDNode n, Point p, float r) {
2.      if (!canCorrelate(n, p, r)) return;
3.      else
4.          if (n.isLeaf && dist(n, p) < r)
5.              p.count.accum(1);
6.          else if (!n.isLeaf)
7.              if (!thresholdMet)
8.                  spawn corr(n.left, p, r);
9.                  spawn corr(n.right, p, r);
10.             else
11.                 gpuQ.addTask(n.left, p, r);
12.                 gpuQ.addTask(n.right, p, r);}
13. }
```

Figure 51: Transformed point correlation algorithm to enable GPU task queue

Note that we do not attempt to perform the CPU work in parallel, though it could reasonably be parallelized; because the work up to the frontier represented by the task queue is small compared to the overall work in the program, executing this work

sequentially is a minor overhead, and simplifies implementation.

6.4.3 Mixing Data Parallelism and Task Parallelism

As mentioned in Section 6.3.2, some applications have a mix of data and task parallelism. In particular, applications such as point correlation often have a data parallel outer loop (in this case, iterating over multiple points), where each iteration of that data parallel loop performs task-parallel work.

Integrating such algorithms into our framework is simple: we merely execute the data-parallel outer loop sequentially, and, upon entering a task parallel iteration, execute it according to the scheme above. Because most of the work is performed by the tasks enqueued into the GPU queue, the data parallel outer loop will "finish," having enqueued most of its computation into the GPU queue. We can then execute the GPU queue and proceed as before. This transformation is safe, since the data-parallel iterations are independent of one another, and each iteration's task-parallel tasks are also independent, hence all the tasks, even if they arise from different iterations, are independent.

Note that in the particular case of point correlation, this execution strategy leads to poor data locality. Suppose we want to process 1000 points. Suppose, further, that the enqueuing threshold for the task parallel computation is two levels deep in the tree (as in Figure 49). Then each point's execution will lead to the creation of four tasks, touching four different subtrees of the kd-tree, and overall 4000 total tasks will be created, with 1000 tasks touching each subtree. However, if the tasks are placed into the GPU queue in order, then each of a point's tasks will be placed contiguously into the queue. Because these threads are likely to be placed in the same warp during GPU execution, each warp's memory footprint will span the entire tree, leading to very poor locality, and hence poor performance. The next section discusses how to solve this problem.

6.5 Scheduling for Locality

The execution strategy outlined in the previous section solves the initial problem

of executing a task-parallel application on hardware built for data-parallelism. But, as pointed out in Section 6.3.1, while GPUs are very efficient data-parallel execution engines, their efficiency comes with several drawbacks. In particular, the memory resources of the GPU are not well-matched to the massive parallelism that efficient execution requires: the GPU features substantial bandwidth (al- lowing the threads to be fed), but very small cache resources. As a result, GPUs work well in streaming workloads or workloads with small reuse footprints. But in workloads with large amounts of reuse but also large footprints, the small caches can dramatically reduce performance. Indeed, to avoid the thrashing that can result from too many threads contending for the same small caches, it is often necessary to reduce an SM's warp count from 64 (the maximum supported) to only four or five [4]. Unfortunately, the kinds of irregular, task-parallel workloads we target are precisely workloads that feature substantial parallelism (due to our execution strategy), but potentially-large memory footprints (the trees traversed in point correlation, for example, can feature millions of nodes). Another problem arises in combating memory divergence: while keeping the footprint of a block to a minimum to avoid cache thrashing is important, it is also important to ensure that the threads of a single warp do not encounter widely varying memory latencies: if one thread in a warp encounters a cache miss while the others do not, all the threads pay the penalty of that cache miss.

To address these problems, we propose locality-aware queue scheduling. Orr et al. proposed a multi-queue strategy for executing task parallel program where different queues correspond to different types of computations (to reduce control divergence) [33]. Instead, we propose to use multiple queues where different queues correspond to different locality domains: an abstract notion of the region of memory a task might access. We create a separate queue for each such locality domain, and tasks expected to access similar regions of memory will be assigned to the same queue. By placing threads from the same locality domain together, we promote threads in the same warp touching similar pieces of memory, both reducing footprints and decreasing memory divergence.

Locality-aware scheduling requires some understanding of the memory access patterns of the tasks that are being scheduled. This is inherently an application- specific property:

various features of the application, and the specific task, might be used to map the task to a particular locality domain. To support this type of scheduling, we add an additional parameter to the addTask hook, where the programmer can pass the result of evaluating a simple function to compute the locality domain for the scheduled task.

In many cases, it is straightforward to determine a task's locality domain. For example, in tree-based benchmarks, we originally start with independent tasks each accessing the whole tree i.e. we start with one locality domain. When subdividing these tasks into independent subtasks, each traversal gets divided into multiple traversals each accessing a part of the tree. This suggests a very simple representation of each locality domain: the node the task is invoked on, which is the root of the subtree it will access. Thus, the two addTask calls from Figure 51 can be replaced with the following:

a. gpuQ.getQueue(n.left).addTask(n.left, p, r);
b. gpuQ.getQueue(n.right).addTask(n.right , p, r);

Note the effect of this implementation on the mixed data- and task-parallelism scenario from Section 6.4.3. If there are four different subtrees that an enqueued task could access, there will be four separate queues. Each point will enqueue its four tasks onto the four separate queues, and task from different points that access the same subtree will be placed next to each other in each of the queues, promoting locality.

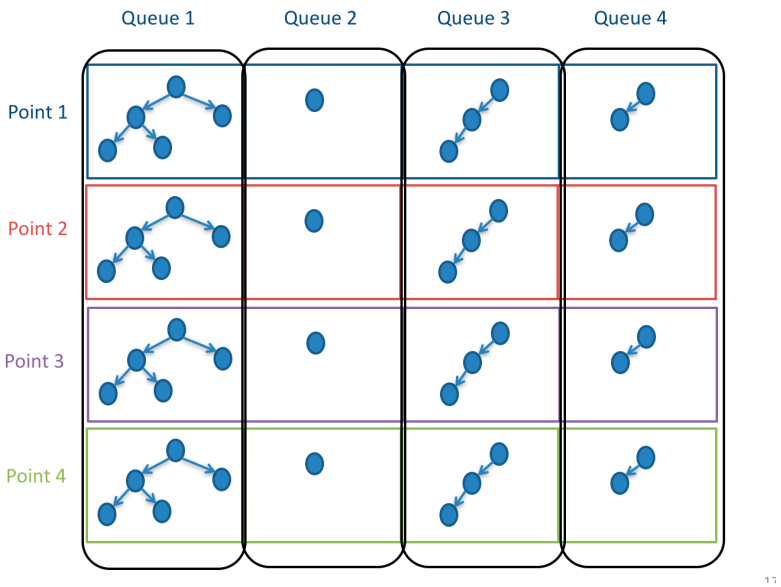


Figure 52: Locality aware queue

6.6 Implementation

6.6.1 Determining the Queue Threshold

The pseudocode in Figure 51 uses an arbitrary threshold for determining when to stop expanding out the computation tree and to instead enqueue tasks into the GPU queue for later execution. In general, the threshold is application- specific, and possibly input-specific. However, we have found that a good rule of thumb is to ensure that the memory footprint of each task is fairly small. Recall that GPUs have limited per-thread memory resources. Hence, if the tasks in the task queue have large footprints, each thread will demand substantial amounts of memory, increasing the per-block memory footprint and lowering performance. Moreover, by keeping tasks relatively small, the likelihood that tasks diverge significantly during execution is reduced, helping further mitigate control divergence. Finally, by keeping individual task footprints small, the total number of tasks increases (because the CPU waits longer before hitting the enqueue threshold), creating enough parallelism to keep the GPU busy. So, for example, in our point correlation example, we might set the threshold at a depth such that the subtree visited by each enqueued task is relatively small.

6.6.2 Queue Merging

If there are many locality-based queues, the overhead of sending each queue to the GPU separately can be prohibitive. Instead, once the queues are constructed by the CPU expansion of the computation tree are complete, the queues are merged into a single queue and sent to the GPU. Because the threads in the uni- fied queue are still ordered according to their locality-based queues, the schedule of execution will still be consistent with the locality-aware grouping.

6.6.3 Queue Size Reduction

If there are too many tasks in the queues after expansion, there may be too much parallelism for the GPU, resulting in more scheduling overhead. To avoid this, we implement a queue size reduction optimization. Rather than passing the task queue to the GPU to be executed with a simple do-all loop, where each iteration gets mapped to a separate thread, we can instead rewrite the task queue loop so that each thread processes two (or more) tasks from the queue (essentially, by strip mining the loop that processes the task queue). Because each thread will execute consecutive tasks in the task queue, the locality-aware scheduling policy will promote those two tasks' having overlapping memory footprints, thus not increasing the memory footprint of a given thread, coarsening the computation without increasing cache pressure.

Note that this general strategy: of having a smaller number of threads exe- cute a larger number of tasks by scheduling multiple tasks per thread, is similar to the approach advocated by persistent threads [28]. A key difference is that persistent threads approaches dynamically schedule tasks to threads, while we statically schedule tasks to threads. While dynamic scheduling can be more flexible, our static scheduling has two advantages:

1) static scheduling means that the multiple tasks scheduled to each thread a are guaranteed to come from the same locality domain, improving locality.

2) static scheduling means that we avoid the runtime overhead of managing the task queue.

6.7 Evaluation

We evaluate our locality-aware task scheduling approach on three task-parallel applications: fast multipole method (FMM), point correlation (PC), and nearest- neighbor (NN). The CPU on which the experiments are conducted has 2 AMD Opteron 6164 HE processors, each of which has 12 cores running at 1.7 GHz, with 32 GB of system memory. The GPU on which the experiments are run is an nVidia Tesla K20C with 5120 MB of RAM and 2496 CUDA cores. The runtimes of our implementations include the time spent on the CPU to generate tasks and communicate the task queue to the GPU, as well as the time to retrieve the reduction objects from the GPU and complete the reduction. In other words, our implementations' runtimes are directly comparable to GPU-only execution.

6.7.1 Fast Multipole Method

The fast multipole method is a fast approximation algorithm for the n-body problem. It operates by performing a bottom-up traversal of an quad-tree, at each level of the tree computing for each subtree at that level the forces con- tributed by the bodies in that subtree on neighboring subtrees. The task paral- lelism in this program arises because the subtrees can be processed in parallel.

For FMM, we compare a locality-agnostic implementation of our approach to one where subtrees represent locality domains, and hence tasks that operate on the same subtree are grouped together. Figure 53 shows the speedup of the locality-aware approach to the locality-agnostic approach, with two different task granularities: one where tasks originate 6-levels deep in the tree, and one where they originate 7-levels deep. We see that with the coarser-grained tasks, being locality-aware provides a 1.34× speedup. With finer-grained tasks, where grouping together similar tasks results in a smaller footprint that better utilizes the GPU's caches, we can achieve a 2.36× speedup over the locality-agnostic implementation.

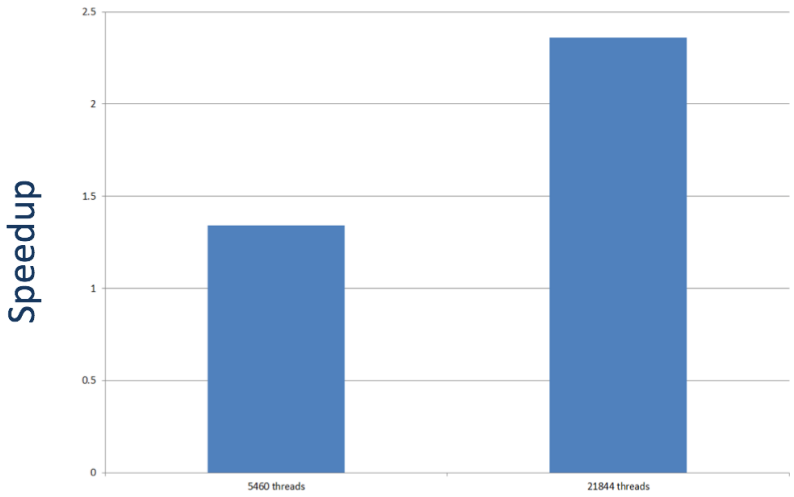


Figure 53: Speedup of locality-aware FMM over locality-agnostic FMM

6.7.2 Point Correlation

Point correlation, our running example, is a mixed data- and task-parallel benchmark: each of a set of independent points traverses a tree (data parallelism), and those points can traverse the tree in parallel by processing independent subtrees simultaneously (task parallelism). Unlike for FMM, where we compare the locality-aware and locality-agnostic implementations of our framework, for PC, we also compare against an optimized GPU baseline [29]. This GPU baseline exploits only data parallelism.

Figure 54 shows the speedup of our locality-agnostic and locality-aware task queue implementations over the data-parallel baseline. We varied both the enqueuing threshold (a given number of trees is the number of tasks per point we generate) and the number of traversals we perform. We see that the locality- aware implementation is consistently faster than the locality-unaware implementation. Indeed, the locality-aware implementation is consistently faster than the baseline data-parallel implementation. Further, we see the effect of the enqueuing threshold on performance: for this particular application, generating four tasks per point provides a good balance between overhead (more tasks means more work generating tasks and performing reductions on the CPU) and locality. Finally, we see that when there are only a small number of traversals, the

data-parallel implementation is significantly slower than our mixed implementation, as we are able to generate additional parallelism to keep the GPU busy, achieving a speedup of almost 3× over the optimized baseline.

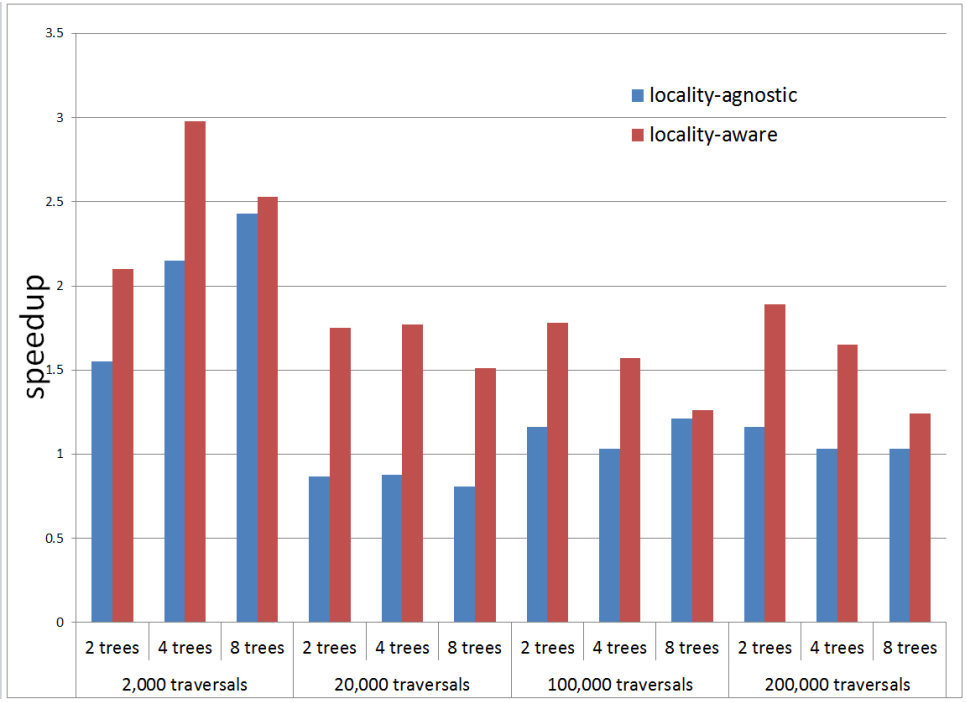


Figure 54: Speedup of PC over data-parallel GPU baseline

6.7.3 Nearest-Neighbor

We perform a similar experiment as PC for our nearest-neighbor benchmark, again comparing our locality-agnostic and locality-aware implementations to an optimized, data-parallel-only baseline. Figure 55 shows again that our mixed implementations are consistently faster than the data-parallel-only implementation, and that adding locality awareness consistently adds performance. We again see the effect of being able to exploit additional parallelism: for 20K point inputs, our locality-aware task queue implementation is over 50× faster than the baseline.

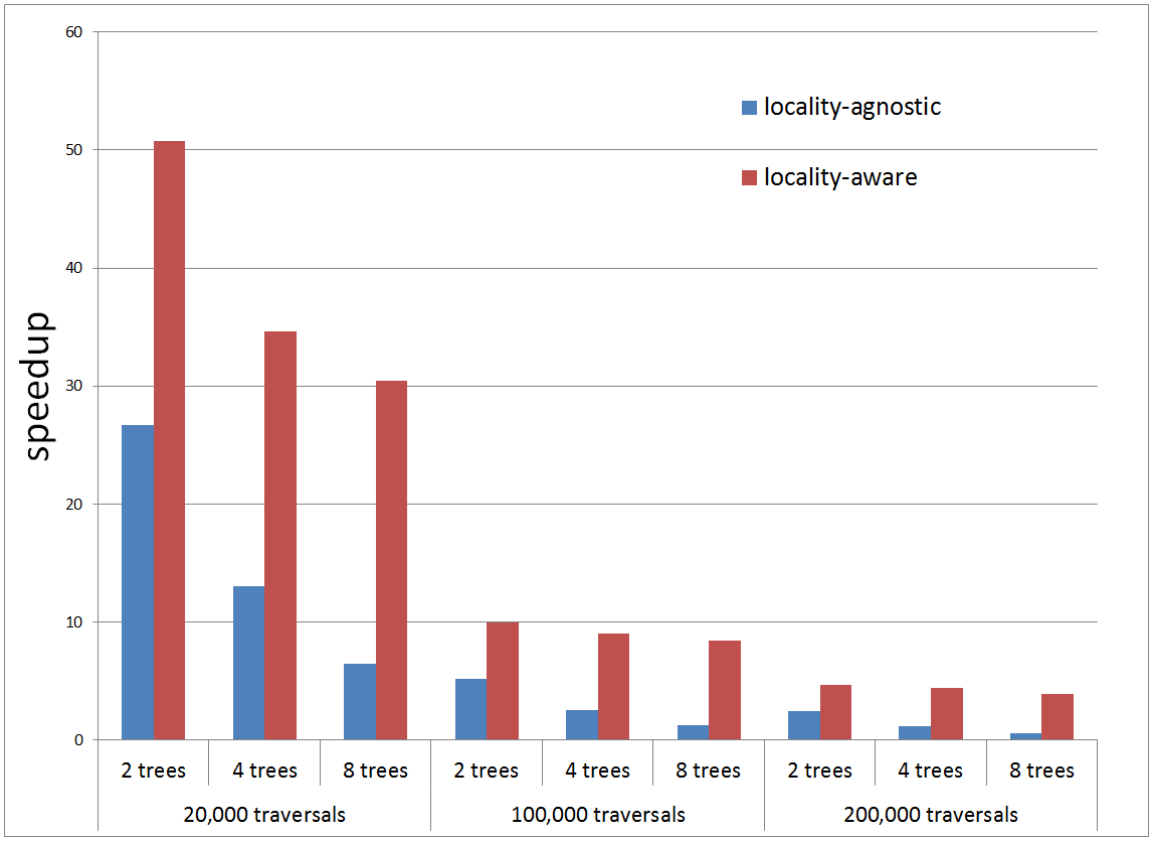


Figure 55: Speedup of NN over data-parallel GPU baseline

6.8 Conclusion

This work presented an approach to exploiting task-parallelism on GPUs by performing partial execution on the CPU to generate a queue of data-parallel tasks that can be executed on the GPU. We then showed how multiple such task queues can be used to add locality-awareness, with the goal of shrinking the memory footprint of individual warps to reduce cache pressure. Preliminary results are promising, showing not only that our approach to exploiting task-parallelism can result in efficient implementations, but also that locality- awareness can significantly boost performance. Indeed, for an implementation of nearest-neighbor, our approach yields a performance improvement of 50× over an optimized data-parallel implementation. Future work could explore methods of defining locality domains for applications where statically defining the domain is not possible.

APPENDIX A.

Longest Common Subsequence

LCS is the problem of finding the longest subsequence common to all sequences in a set of sequences, 2 sequences in our case.

A sub-sequence is a sequence that appears in the same relative order, but not necessarily contiguous.

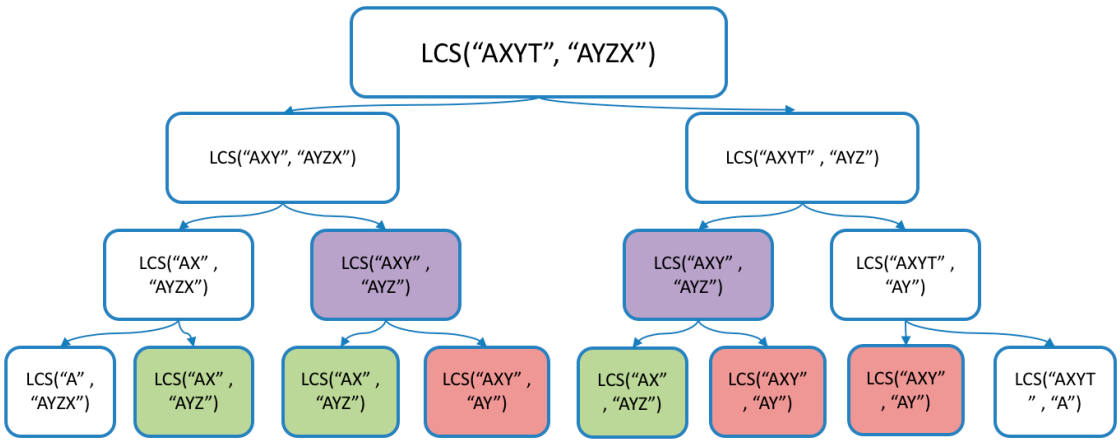


Figure 56: LCS example

LCS can be solved using dynamic programming because it has both an optimal substructure and the sub-problems are overlapping.

- It has an optimal substructure because the optimal solution can be constructed from the optimal solutions of the subsequence of the sequences in question.
- The sub-problems are overlapping because the space of sub-problems is small and any algorithm that solves the problem use the solutions of the sub-problems over and over again rather than generating new sub-problems

The solution using dynamic programming requires saving the solutions of the sub-problems in a table as shown in Figure 37 above.

So given the sequences X=AXYT and Y=AYZX

If the last characters of the 2 sequences are equal, then

- LCS(X[0,m], Y[0,n]) = 1+ LCS(X[0,m-1], Y[0,n-1]) (1)

If the last 2 characters are not equal, then

- LCS(X[0,m], Y[0,n]) = MAX(LCS(X[0,m-1], Y[0,n]) , LCS(X[0,m], Y[0,n-1])) (2)

So it is clear from equations (1) and (2) above that LCS has an optimal substructure.

$$LCS(X_i, Y_j) = \begin{cases} \emptyset & \text{if } i = 0 \text{ or } j = 0 \\ LCS(X_{i-1}, Y_{j-1}) \frown x_i & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \text{longest}\{LCS(X_i, Y_{j-1}), LCS(X_{i-1}, Y_j)\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

Now if we draw the tree of the solution (Figure 56) for the strings given above we find that some that some of the sub-problems are repeated. all the nodes with same fill color (other than white) in Figure 56 solve the same sub-problem.

Thus if the sub-problem is solved once and its result is saved we do not have to re-compute the LCS of the same strings when we re-encounter the same sub-problem.

APPENDIX B.

PTX instruction set:

PTX stands for parallel thread execution. It is a pseudo assembly language used in the programming environment of NVIDIA.

Given a CUDA program the NVCC compiler generates PTX. Later the graphics driver has a compiler that translates PTX into a binary code that can run on NVIDIA's GPUs.

PTX instructions generally have from zero to four operands, plus an optional guard predicate appearing after an '@' symbol to the left of the opcode.

Integer Arithmetic instructions:

Add	Addition
Sub	Subtract
Mult	Multiply
Addc	Add 2 values with carry-in and optional carry out
Subc	Subtract one value from another with borrow-in and optional borrow-out
Mad	Multiply 2 values and add a third value
Mul24	Multiply 2 24-bit integer value
Mad24	Multiply two 24-bt integer values and add a third value
Sad	Sum of absolute differences
Div	Divide one value by another
Rem	The remainder of integer division

Abs	Absolute value
Neg	Arithmetic negate
Min	Minimum of two values
Max	Find the maximum of two values

Floating point instructions:

Add	Addition
Sub	Subtract
Mult	Multiply
Fma	Fused multiply add
Mad	Multiply 2 values and add a 3rd value
Div	Divide one value by another
Abs	Absolute value
Neg	Arithmetic negate
Min	Minimum of 2 values
Max	Maximum of 2 values
Rcp	Take the reciprocal of a value
Sqrt	Take the square root of a value
Rsqrt	Take the reciprocal of the square root of a value

Sin	Find the sine of a value
Cos	Find the cosine of a value
Lg2	Find the log, base 2, of a value
Ex2	Find the base-2 exponential of a value

Comparison and selection instructions:

Set	Compare two numeric values with a relational operator, and optionally combine this result with a predicate value by applying a Boolean operator.
Setp	Compare 2 numeric values with a relational operator, and optionally combine this result with a predicate value by applying a Boolean operator
Selp	Select between source operands, based on the value of the predicate source operand
slct	Select one source operand, based on the sign of the third operand

Logic and shift instructions:

And	Bitwise AND
Or	Bitwise OR
Xor	Bitwise exclusive OR
Not	Bitwise negation; one's complement
Cnot	c/c++ style logical negation
Shl	Shift bits left, zero-fill on right

shr	Shift bits right, sign or zero fill on left

Data movement and conversion instructions:

Mov	Set a register variable with the value of a register variable or an immediate value
Ld	Load a register variable from an addressable state space variable
St	Store a register to an addressable state space variable.
Cvt	Convert a value from one type to another

Texture instructions (provides access to texture memory):

Tex	Perform a texture memory lookup

Control flow instructions:

{}	Instruction grouping
@	Predicated execution
Bra	Branch to a target and continue execution there
Call	Call a function, recording the return location
Ret	Return from function to instruction after call
Exit	Terminate a thread

Parallel synchronization and communication instructions:

Bar	Signal arrival at a barrier and wait
Membar	Memory barrier
Atom	Atomic reduction operations for thread-to-thread communication
Red	Reduction operations on global and shared memory
vote	Vote across thread group

Miscellaneous instructions:

Trap	Perform trap operation
Brkpt	breakpoint
Pmevent	Performance monitor event

REFERENCES

[1] Dennard scaling : Dennard, Robert H.; Gaensslen, Fritz; Yu, Hwa-Nien; Rideout, Leo; Bassous, Ernest; LeBlanc, Andre (October 1974). "Design of ion-implanted MOSFET's with very small physical dimensions" (PDF). *IEEE Journal of Solid-State Circuits*. **SC-9** (5).

[2] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate versus IPC: the end of the road for conventional microarchitectures. In ISCA ’00: Proceedings of the 27th Annual International Symposium on Computer Architecture, pages 248–259, 2000.

[3] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt. Improving GPU performance via large warps and two-level warp scheduling. In Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO), pages 308–317, 2011.

[4] Timothy G. Rogers, Mike O’Connor, and Tor M. Aamodt. Cache-conscious wavefront scheduling. In Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO), 2012

[5] Michael Mishkin, Nam Sung Kim, and Mikko Lipasti. Write-after-read hazard prevention in GPGPUSIM. In Workshop on Deplicating, Deconstructing, and Debunking (WDDD), June 2016.

[6] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic warp for- mation and scheduling for efficient GPU control flow. In Proc. of the ACM/IEEE Interna- tional Symposium on Microarchitecture (MICRO), pages 407–420, 2007.

[7] Wilson W. L. Fung and Tor M. Aamodt. Thread block compaction for efficient SIMT control flow. In Proc. of the IEEE International Symposium on High-Performance Computer Architecture (HPCA), pages 25–36, 2011.

[8] Nicolas Brunie, Sylvain Collange, and Gregory Diamos. Simultaneous branch and warp inter- weaving for sustained GPU performance. In Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA), pages 49–60, 2012.

[9] Ujval J. Kapasi et al. Efficient conditional operations for data-parallel architectures. In Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO), pages 159–170, 2000.

[10] Minsoo Rhu and Mattan Erez. Maximizing SIMD resource utilization in GPGPUs with SIMD lane permutation. In Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA), 2013b.

[11] Jiayuan Meng, David Tarjan, and Kevin Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA), pages 235–246, 2010.

[12] Minsoo Rhu and Mattan Erez. The dual-path execution model for efficient GPU control flow. In Proc. of the IEEE International Symposium on High-Performance Computer Architecture (HPCA), pages 591–602, 2013a

[13] Ahmed ElTantaway, Jessica Wenjie Ma, Mike O'Connor, and Tor M. Aamodt. A scalable multi-path microarchitecture for efficient GPU control flow. In Proc. of the IEEE Interna- tional Symposium on High-Performance Computer Architecture (HPCA), 2014.

[14] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic. The vector-thread architecture. In Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA), pages 52–63, June 2004.

[15] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco. GPUs and the future of parallel computing. Micro, IEEE, 31(5):7–17, September 2011.

[16] Timothy G. Rogers, Daniel R. Johnson, Mike O'Connor, and Stephen W. Keckler. A vari- able warp size architecture. In Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA), 2015

[17] S.Che,M.Boyer,J.Meng,D.Tarjan,J.Sheaffer,S.-H.Lee, and K. Skadron. Rodinia: A Benchmark Suite for Hetero- geneous Computing. In *IEEE International Symposium on Workload Characterization (IISWC-2009)*, October 2009.

[18] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Sim- ulator. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2009)*, April 2009.

[19] IMPACT Research Group. The Parboil Benchmark Suite. http://www.crhc.uiuc.edu/IMPACT/parboil.php, 2007.

[20] A. Gharaibeh and M. Ripeanu. Size Matters: Space/Time Tradeoffs to Improve GPGPU Applications Performance. In *2010 International Conference for High Performance Computing, Networking, Storage and Analysis (SC-2010)*, November 2010.

[21] T. D. Han and T. Abdelrahman. Reducing Branch Diver- gence in GPU Programs. In *4-th Workshop on General Pur- pose Processing on Graphics Processing Units (GPGPU-4)*, March 2011.

[22] Robert D. Blumofe , Christopher F. Joerg , Bradley C. Kuszmaul , Charles E. Leiserson , Keith H. Randall , Yuli Zhou, Cilk: an efficient multithreaded runtime system, Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming, p.207-216, July 19-21, 1995, Santa Barbara, California, USA

[23] N. Capodieci, P. Burgio, "Efficient implementation of genetic algorithms on gp-gpu with scheduled persistent cuda threads", *Parallel Architectures Algorithms and Programming (PAAP) 2015 Seventh International Symposium on*, pp. 6-12, Dec 2015.

[24] Guoyang Chen , Xipeng Shen, Free launch: optimizing GPU dynamic kernel launches through thread reuse, Proceedings of the 48th International Symposium on Microarchitecture, December 05-09, 2015, Waikiki, Hawaii

[25] Matteo Frigo , Pablo Halpern , Charles E. Leiserson , Stephen Lewin-Berlin, Reducers and other Cilk++ hyperobjects, Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures, August 11-13, 2009, Calgary, AB, Canada

[26] Benedict R. Gaster , Lee Howes, Can GPGPU Programming Be Liberated from the Data-Parallel Bottleneck?, Computer, v.45 n.8, p.42-52, August 2012

[27] Michael Goldfarb , Youngjoon Jo , Milind Kulkarni, General transformations for GPU execution of tree traversals, Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, November 17-21, 2013, Denver, Colorado

[28] K. Gupta, J. A. Stuart and J. D. Owens. A Study of Persistent Threads Style GPU Programming for GPGPU Workloads, InPar-2012

[29] Jianqiao Liu , Nikhil Hegde , Milind Kulkarni, Hybrid CPU-GPU scheduling and execution of tree traversals, Proceedings of the 2016 International Conference on Supercomputing, June 01-03, 2016, Istanbul, Turkey

[30] Munshi, A.: OpenCl parallel computing on the GPU and CPU . In: SIGGRAPH

[31] Rupesh Nasre , Martin Burtscher , Keshav Pingali, Data-Driven Versus Topology-driven Irregular Computations on GPUs, Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, p.463-474, May 20-24, 2013

[32] NVIDIA. CUDA . http://www.nvidia.com/object/cuda_home_new.html

[33] Marc S. Orr , Bradford M. Beckmann , Steven K. Reinhardt , David A. Wood, Fine-grain task aggregation and coordination on GPUs, Proceeding of the 41st annual international symposium on Computer architecuture, June 14-18, 2014, Minneapolis, Minnesota, USA

[34] Bin Ren , Youngjoon Jo , Sriram Krishnamoorthy , Kunal Agrawal , Milind Kulkarni, Efficient execution of recursive programs on commodity vector hardware, Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, June 13-17, 2015, Portland, OR, USA

[35] Bo Wu , Guoyang Chen , Dong Li , Xipeng Shen , Jeffrey Vetter, Enabling and Exploiting Flexible Task Assignment on GPU through SM-Centric Program Transformations, Proceedings

of the 29th ACM on International Conference on Supercomputing, June 08-11, 2015, Newport Beach, California, USA

[36] Rupp Karl, CPU, GPU and MIC Hardware Characteristics over time, https://www.karlrupp.net/

[37] AMD Southern Islands Series Instruction Set Architecture. AMD, 1.1 ed., December 2012

[38] Tor M. Aamodt, Wilson Wai Lun Fung, and Timothy G. Rogers, General-Purpose Graphics Processor Architectures. Morgan & Claypool Publishers, 2018. Print

[39] Krste Asanovic, Stephen W. Keckler, Yunsup Lee, Ronny Krashinsky, and Vinod Grover. Convergence and scalarization for data-parallel architectures. In Proc. Of the ACM/IEEE International Symposium on Code Generation and Optimization (CGO), 2013.

[40] Thomas H. Cormen, Charles E. Leireson, Ronald L. Rivest, and Clifford Stein, Introduction to Algorithms: Edition 3. MIT press, 2009. Print

[41] GPGPU-Sim. http://www.gpgpu-sim.org

[42] GPGPU-Sim Manual http://ww.gpgpu-sim.org/manual