Spring 2015

Exploiting intra-warp address monotonicity for fast memory coalescing in GPUs

Hector Rodriguez-Simmonds
Purdue University

Graduate School Form 30
(Updated 11/20/2014)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Hector Rodriguez-Simmonds

Entitled
Exploiting Intra-Warp Address Monotonicity for Fast Memory Coalescing in GPUs

For the degree of Master of Science in Electrical and Computer Engineering

Is approved by the final examining committee:

MITHUNA S. THOTTETHODI

ANAND RAGHUNATHAN

MILIND KULKARNI

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification/Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

MITHUNA S. THOTTETHODI

Approved by Major Professor(s): _____

_____

Approved by: Michael R. Melloch 03/30/2015

Head of the Department Graduate Program Date

EXPLOITING INTRA-WARP ADDRESS MONOTONICITY FOR FAST MEMORY COALESCING IN GPUS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Héctor E. Rodríguez-Simmonds

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science in Electrical and Computer Engineering

May 2015

Purdue University

West Lafayette, Indiana

ACKNOWLEDGMENTS

First of all I'd like to thank my advisor Dr. Mithuna Thottethodi. Your grounded, pertinent advice and direction have been invaluable. Thank you for talking with me, advising, teaching, and mentoring me. Thank you for letting the student come first. Thank you to my committee Dr. Milind Kulkarni and Dr. Anand Raghunathan for their feedback and time.

I'd like to acknowledge the help of Prashanth Purnananda. Your help at the last minute helped tremendously! Thanks to Dr. Mark C. Johnson for his help throughout the circuit design and test portion of this work. I'd like to thank my collaborator Calvin Holic. I'd like to thank my close friends Nadra Guizani and Tanmay Prakash for keeping me going throughout this process. I also want to thank fellow members of my lab: Eric Villaseñor and Tim Pritchett for empathizing with my struggles to complete this thesis.

I would also like to acknowledge the help and support offered by my friends as well as my Engineering Education (ENE) cohort and the faculty of ENE for their guidance. I want to thank my family for continually encouraging me. Another thank you goes out to the staff of the graduate office for making all the administrative things I had to do to get my masters much more bearable. Also, thank you to the janitorial staff of ECE, especially Teresa for chatting with me in the hallway during many late nights working. I'd also like to acknowledge my funding sources: The Purdue Doctoral Fellowship and Dr. Monica Cox through the REACH Scholars program.

Lastly and most importantly I want to acknowledge the help and support of my mother, Emma. Thank you for dealing with me throughout this entire escapade. Thank you. Thank you. Thank you.

TABLE OF CONTENTS

Page

LIST OF FIGURES

ABSTRACT

Rodríguez-Simmonds, Héctor Enrique M.S.E.C.E., Purdue University, May 2015. Exploiting Intra-Warp Address Monotonicity for Fast Memory Coalescing in GPUs. Major Professor: Mithuna S. Thottethodi.

Graphics Processing Units (GPUs) are growing increasingly popular as general purpose compute accelerators. GPUs are best suited for applications which have abundant data parallelism wherein the computation expressed as a single thread can be applied over a large set of data items. One key constraint that affects application performance on GPUs is that the underlying hardware is single-instruction, multiple data (SIMD) hardware which requires parallel instructions from the multiple threads to execute in a lock-step manner. The benefits of lock-step execution can be seriously degraded if the threads diverge (because of memory or branches). Specifically in the case of memory, the addresses from each thread in a SIMD "wavefront/warp" must be coalesced to enable parallel memory access to minimize divergence.

The general problem of coalescing assumes arbitrary address distribution which can be slow. This thesis aims to exploit intra-warp address monotonicity (as measured in a recent study by Holic) to achieve fast memory coalescing. Holic's study reveals the intra-warp addresses are monotonically increasing or decreasing in the common case. The key contributions of this thesis are twofold. First, I design novel hardware coalescing mechanisms to achieve fast-coalescing and quantify the area/delay of my coalescing designs. Second, I quantify the impact of fast-coalescing on overall GPU performance for a suite of GPU benchmarks.

1. INTRODUCTION

Graphics Processing Units (GPUs) are growing increasingly popular as general purpose compute accelerators. GPUs are best suited for applications which have abundant data parallelism wherein the computation expressed as a single thread can be applied over a large set of data items. Such general-purpose GPUs (GPGPUs) exploit the abundant data-level parallelism available across threads to tolerate the latencies of individual threads by executing other threads in the interim.

One key constraint that affects application performance on GPUs is that the underlying hardware is single-instruction, multiple data (SIMD) hardware which requires parallel instructions from the multiple threads to execute in a lock-step manner. The benefits of lock-step execution can be seriously degraded if the threads diverge (because of memory or branches). Specifically in the case of memory, the addresses from each thread in a SIMD "wavefront/warp" must be coalesced to enable parallel memory access to minimize divergence.

Consider the problem of memory coalescing. The memory requests from each thread in the warp that targets the same cache block must be merged to avoid unnecessary repeated accesses. In general, this is the problem of detecting unique addresses (at the cache block granularity) from among the addresses accessed by each warp. (In reality there are other considerations because of cache banking. Two addresses that are unique may still cause divergence if they cause a bank conflict which precludes parallel access.) The general problem of coalescing addresses the worst-case challenge of arbitrary address distribution. That is, the problem is one of finding unique cache block addresses among all addresses in a warp (typically 32 or 64 addresses).

Further simplifications can be made based on the observation that there is an upper bound on unique accesses that can be coalesced because of hardware bandwidth. For example, in a memory system in which no more than four unique addresses

can be accessed, coalescing can be reduced to detecting four unique cache blocks (rather than all unique cache blocks). In spite of such simplifications, coalescing is expensive because it fundamentally requires comparisons of all pairs of addresses. My evaluations using Synopsis Design compiler reveals that the coalescer logic delay can exceed 100 FO4[1] units.

This thesis aims to exploit intra-warp address monotonicity (as measured in a recent study by Holic) to achieve fast memory coalescing. Holic's study reveals the intra-warp addresses are monotonically increasing or decreasing in the common case. Formally, if the threads in a warp are numbered in increasing order from 1 to 32 (or 64), monotonicity requires that the addresses from the threads are either non-increasing or non-decreasing in thread order (i.e., $\forall i, j | 0 \leq i, j \leq 31, i > j \implies A[i] \leq A[j]$ for monotonically decreasing addresses and $i > j \implies A[i] \geq A[j]$ for monotonically increasing addresses.)

I make the key observation that monotonicity can be leveraged to achieve faster coalescing. Specifically, unique cache blocks can be determined solely by comparing neighboring addresses (i.e., comparing the memory request address from thread i to that of thread $(i+1)$ and $(i-1)$) because duplicate addresses must be consecutive. One can offer a simple proof-by-contradiction to the possibility that identical block addresses (say $A[i]$ and $A[i+2]$) are non-adjacent by assuming an intermediate address $A[i+1]$. In the trivial case where $A[i+1] = A[i]$, transitivity ensures that the duplicate addresses are in fact adjacent (because $A[i] = A[i+2]$). If on the other hand, we assume that $A[i+1] \neq A[i]$ we can show that monotonicity is violated. My evaluations show that leveraging the above property can significantly reduce coalescer delays.

While the above insight works for warps where the addresses are monotonic, coalescers must indeed work for the general case. To that end we propose to use the fast-coalescing approach speculatively in parallel with the traditional general coalescer. If the address distribution is monotonic, we can proceed with the output of the fast

[1]FO4 refers to the 'fan-out 4' delay of an inverter driving four other inverters, which is a technology independent delay unit.

coalescer. However, if the addresses are not monotonic, one must fall back on the slower traditional coalescer. In either case, redundant cache block accesses are guaranteed to be eliminated. This approach requires fast detection of monotonicity among the addresses of a warp. Fortunately, monotonicity detection requires neighbor-to-neighbor address comparison (in this case subtraction, rather than equality testing) and neighbor-to-neighbor comparison to ensure monotonicity.

The key contributions of this thesis are twofold. First, I design novel hardware coalescing mechanisms to achieve fast-coalescing and quantify the area/delay of my coalescing designs. Second, I show that the abundant latency tolerance for the benchmarks I consider results in minimal direct speedup even if latency of memory accesses is increased due to slow coalescing. However, I offer evidence that my fast-coalescing technique places less demand on the latency tolerance mechanisms; which preserves the ability to use latency tolerance for other unavoidable long-latency operations. I show that with reduced latency tolerance, the performance benefits of fast-coalescing increase.

2. BACKGROUND

This chapter discusses Graphics Processing Units (GPUs), how they're used in general purpose applications, and includes a brief discussion about the relevant terms and architectural features of GPUs. The primary architectural features of the two main design houses for recent GPUs, AMD and NVIDIA, are mostly the same. For the purposes of this thesis I use NVIDIAs terminology and programming model, Compute Unified Device Architecture (CUDA), unless otherwise stated.

2.1 Graphics Processing Unit Background

Graphics Processing Units (GPUs) are highly parallel computing structures. They were originally developed for graphical applications [1]. Graphics applications are inherently highly parallelizable. Therefore, the hardware structures that computed them were more efficiently implemented as Single Instruction Multiple Data (SIMD) structures. In contrast, Central Processing Units (CPUs) are Single Instruction Single Data (SISD). As GPUs are highly parallel structures, many general-purpose applications and algorithms that are also easily parallelizable have been implemented or ported to GPUs. The use of GPUs in these general-purpose applications has been steadily increasing in the last decade as both hardware and software implementations by AMD and NVIDIA have made programmability on these platforms simpler, more efficient, and more robust [1, 2].

APIs such as CUDA that abstracted GPU-specific structures helped alleviate programmer effort. The increase in programmability of GPUs as well as their evolution to more flexible general-purpose hardware has led to General Purpose Graphics Processing Units (GPGPUs) [3]. For example, NVIDIA uses "warps" as their construct for handling SIMD instruction types. Each of these warps contains one instruction

and up to 32 data elements on which to perform this instruction in lockstep. In constrast to CPU designs, GPU memory latencies are conceled by interlacing warps. CPUs use large caches to conceal these latencies. Smaller caches allow GPUs to dedicate more chip area to execution units, the main element responsible for the highly parallelizable feature of GPUs. The memory subsystem of a GPU is of particular interest to this research. I will now cover NVIDIAs memory subsystem.

2.1.1 Memory types

Using CUDA, NVIDIA allows the use of their GPGPUs in many programming languages [4]. In the following section I discuss the different memory types on GPGPUs and their relevance to my memory access coalescer.

Global Memory

This memory, which resides on the device, can be accessed through "32-, 64-, or 128-byte memory transactions" [4]. The compute capability of the GPU will change the transaction size [4]. Memory accesses within a warp are coalesced together into accesses that align with these memory transaction sizes. If multiple thread's accesses cannot be coalesced into one memory transaction, more accessess will be generated. The more accesses that are generated, the more unused data is transferred. This reduces instruction throughput.

Maximizing this type of coalescing with respect to the device's compute capability is important to keep in mind when designing applications. Devices with compute capability 2.x or higher are able to cache memory transactions. Creating data structures of appropriate sizes to fit within the GPUs memory transaction size is important to maximizing coalescing [4].

Texture and Surface Memory

Both of these memories reside in device memory. They are cached in texture cache. Only on texture cache misses does the memory fetch go to surface memory. The texture cache is optimized for "threads of the same warp that read from memory locations that are close together" [4].

Local Memory

This memory is located in device memory as well. Local memory is used automatically by the hardware if Kernels use more registers than available, when array structures would use up excessive register space, and when arrays are indexed with inconsistent quantities [4]. Local memory is optimized to offer consecutive 32-bit word reads by consecutive thread IDs in a warp. Also compute capability dependant, these types of memory accesses mirror global memory policies and are cached in L1 and L2 [4].

Constant Memory

Also residing in device memory, this memory observes caching policies depending on compute capability. The CUDA programming guide states that for devices with compute capability 1.x, this memory is accessed through an exclusive read-only cache that is constant. This cache is "shared by all functional units" [4].

Shared Memory

This on-chip memory shared by the L1 cache in devices with compute capability 2.x [4]. It can be arranged as 48KB of shared memory and 16KB of L1 cache or as 16KB of shared memory and 48 KB of L1 cache [4]. It is arranged into 16 banks. It can handle 8-, 16-, and 32-bit strided accesses. Accesses larger than 32-bits per thread generate bank conflicts. In the case of a 32-bit access, each warp's memory access

is split into two half-warp accesses such that there is no conflict between threads in the first or second half. In the event of a non-atomic instruction writes to a shared memory location, it is undefined which thread will perform the write [4].

Summary

Of the five types of memory in GPGPUs, the Rodinia GPGPU benchmark suite I'm using primarily used Global and Shared memory. Rodinia includes 18 benchmark applications. All of the benchmarks used Global memory, texture was accesed by 3 of them, local memory by one, constant memory by one, and shared memory by 10 of them. Together, the two types account for 85% of total memory accesses.

2.1.2 The Challenge of Memory Coalescing

Coalescing multiple memory accesses into one access is important in order to fully utilize the bandwidth of SIMD memory architectures. Duplicate memory requests made by multiple threads in a block can be avoided using coalesced memory accesses. The most heavily utilized memories in GPGPUs are cached and benefit greatly from maximizing their memory bandwidth.

Since these coalesced accesses must be found and given to the memory subsystem before the cache (or TLB), the latency requirements for coalescing hardware structures is very tight. Assuming arbitrary address distribution, finding unique memory addresses requires $\mathcal{O}(n^2)$ accesses. This hardware does not scale well and is very slow. In this thesis I implement a coalescer that can handle an arbitrary address distribution. The impact of this coalescer has a significant penalty. Rotating out to another warp in order to hide these memory latencies increases the hardware-warp-contexts that must be used. Doing this has its own costs. GPUs do not currently switch out warps for coalescing, so this option isn't discussed any further.

2.1.3 Monotonicity of Accesses

Previous work by my collaborator Calvin Holic has shown that a large fraction of warps have monotonic memory accesses. For the Rodinia benchmarks 11 of them were 100% monotonic. Four of the remaining seven had at least 98% monotonicity. Of the remaining three, two (bfs and mummergpu) had monotonicity rates of at least 85% and the last (gaussian) had a monotonicity rate of only 25%.

Our goal is to exploit these monotonicity findings in the creation of hardware that will efficiently coalesce memory accesses.

3. DECOUPLING PERFORMANCE AND NON-REDUNDANT CACHE BLOCK ACCESSES IN COALESCER DESIGN

Recall from (1) that my approach relies on address monotonoity within warps to achieve faster coalescing in the common case. In this chapter, I describe the baseline coalescing circuits which guarantee the reduction of redundant cache block accesses (3.1), the faster coalescing circuits for monotonic intra-warp addresses (3.2), and finally the combined design.

3.1 The Baseline Coalescer

In order to find unique addresses without assuming monotonicity, the baseline coalescer must do an all-to-all comparison of all 32 addresses (in the case of NVIDIA's CUDA architecture) across a warp. The first address must be compared with all other addresses. The second with all the subsequent addresses, the third with all subsequent, and so on. This would result in $\sum_{i=1}^{n-1} i$ comparisons where n is the number of addresses being compared. For example, an NVIDIA warp of 32 addresses would take $\sum_{i=1}^{31} i = 512$ comparisons. This coalescer guarantees no redundant cache block accesses without needing monotonicity to be true. Later, I show the precise hardware organization of the baseline coalescer in section 4.1.3.

3.2 A Fast Coalescer for Monotonic Address Coalescing

The fast coalescer performs a neighbor-to-neighbor comparison of the 32 addresses in a warp. This comparison yeilds a vector that defines the addresses within the warp that are unique. This comparator relies on monotonicity being true in order

for its results to reduce redundant cache block accesses. Assuming monotonicity is maintained across the addresses in the warp, only adjacent addresses need to be compared in order to find which of these in a warp are unique. If monotonicity across the addresses in the warp is preserved, this comparator would take $n-1$ comparisons. In the case of an NVIDIA warp comprised of 32 threads/addresses, 31 comparisons would happen. Later, I show the precise hardware organization of the fast neighbor-to-neighbor coalescer in section 4.1.4.

3.3 Monotonicity Detector

The monotonicity detector is responsible for determining whether the warp is monotonic. This circuit takes in the 32 (or 64) addresses that make up a warp and compares two addresses at a time starting from the first. The first and second address are compared for three things:

1. Whether the two addresses are equal
2. Whether the first address is larger
3. Whether the second address is larger

The second and third addresses are then compared for the same 3 things. The same pattern continues successively until the $n-1$ and n comparisons.

Each comparison yeilds three outputs: equality, 1st address greater, 2nd address greater. If across all 31 comparisons it is found that either the addresses are equal or that the 1st address is always greater, the warp is monotonic. If across all 31 comparisons it is found that either the addresses are all equal or that the 2nd address is always greater, then this warp is monotonic. However, if within a warp there is at least one occurence where both the second and third condition are met, the warp is not monotonic. The hardware organization for the monotonicity detector is shown later in section 4.1.5.

3.4 Putting it together

The full coalescer is created by incorporating the baseline coalescer, the fast monotonically dependent coalescer, and the monotonicity detector together. Fastest coalescing is achieved when monotonicity can be leveraged to find unique memory addresses within a warp. The intuition here is that the baseline coalescer will take the longest in finding unique addresses. The fast coalescer performs minimal neighbor-to-neighbor comparisons and would thus be a very fast circuit but depends on monotonicity to reduce redundant cache block accesses. The monotonic detector assists the fast coalescer to determine if the addresses it deems unique are actually unique. In the event monotonicity holds true for the warp, the results of the fast coalescer can be used to find unique memory addresses. If monotonicity does not hold for the warp, the baseline coalescer can be used to find unique addresses.

Based on my collaborators findings that a vast majority of memory accesses on the Rodinia benchmarks were monotonic, running all three circuits simultaneously ensures fast and correct results are generated for memory coalescing. In the event a warp is not monotonic, unique addresses can still be coalesced into a memory access, albeit taking longer to compute than in the monotonic case.

4. METHODOLOGY

My evaluation methodology has (1) a delay analysis component to analyze the advantage of monotonicity-based coalescing, and (2) architectural evaluation via simulation of execution time.

For the first component, I designed and analyzed the coalescer and monotonicity detection circuits. Specifically, in order to coalesce memory accesses efficiently I created a neighbor-to-neighbor address comparator as described in 3.2 and a naive all-to-all address comparator described in (3.1). I also created an address monotonicity detector (3.4) which is necessary to recognize cases where the monotonicity-based fast-coalescer is inadequate. These three circuits were created in both 32-address and 64-address variants to handle NVIDIA's 32 thread Warp and AMD's 64 thread vectors, respectively.

Synopsis' Design Compiler was used to analyze these circuits and acquire timing and area information. To avoid dependence on the specific technology library, the delays of these circuits are also reported in FO4 units. (The "fan-out 4" delay of an inverter driving four other inverters is a technology independent delay unit.)

For the second component I use a GPU simulator to characterize the performance of the monotonicity coalescer in GPGPU applications. This timing information is also used in our simulations to drive the coalescer delays in GPGPU-SIM in order to investigate how memory access latencies were affected.

4.1 Toolchain

Verilog was used to implement all of the circuits used in this thesis. Functional verification was performed using ModoelSim SE-64 10.1b. Delay analysis was computed using Synopsis Design Compiler.

4.1.1 Synopsis Design Compiler

Synopsis Design Compiler version G-2012.06 on a 64-bit Red Hat Enterprise Linux system was used to analyze the delay of each circuit. The Synopsis SAED_EDK90_CORE 90nm Digital Standard Cell Library was used to synthesize the circuits [5]. This cell library was chosen because of the limited options available; it most closely resembles the 45nm process NVIDIA currently uses in their Fermi architecture GPU's. My collaborator, Calvin Holic, acquired his monotonicity statistics by modeling an NVIDIA Tesla C2050, a Fermi architecture chip. This cell library was created for educational and training purposes [5].

There are 27 characterization models included in the SAED EDK 90 cell library that vary process, voltage, and temperature conditions. Figure 4.2 contains all the options available. The 'typ_ht' model was chosen for this project because it most closely modeled typical conditions for the circuits on these cards. This model operates at 125 centigrade, has a power supply voltage of 1.2V, and has a typical-typical (TT) process corner (NMOS-PMOS) [5]. Because we are interested in more recent technology nodes, the delay analysis of the circuits are also reported in FO4 (the delay of an inverter with a fanout of four equally-sized inverters) units.

By default Synopsis Design Compiler is configured to optimize Verilog designs within boundaries. To avoid a chain of inverters from being optimized out, each inverter was instantiated in its own Verilog module. These modules were then connected as shown in figure 4.1. Circumventing Design Compiler's optimizations in this way allowed me to measure the delay of the middle inverter.

The following parameters were used to synthesize the circuit and generate timing and area information. This TCL script reads the source Verilog files, compiles them with the highest effort, and generates timing and area reports into a file. Compiling with the highest effort makes Design Compiler try "still more gate minimization strategies. The "tool adds gate composition to the process and allocates more CPU time than medium effort" [6]. I found that for the "compile" command "-map_effort

medium" and "-map_effort high" made no difference in the synthesized design for the critical path.

```
read_verilog SOURCE_VERILOG.sv
read_verilog SOURCE_VERILOG.sv
link
uniquify

set_driving_cell -lib_cell NBUFFX2 {address0}
set_driving_cell -lib_cell NBUFFX2 {address1}
set_driving_cell -lib_cell NBUFFX2 {address2}
                .
                .
                .
set_driving_cell -lib_cell NBUFFX2 {addressn}

compile -map_effort high
report_timing -path full -delay max -max_paths 25 -nworst 1 > FILE
report_area >> FILE
write_file -format verilog -hierarchy -output SCHEMATIC_FILE.v
```

4.1.2 FO4 circuit

We decided to report delays for the comparators and monotonicity detector using the fanout-of-4 inverter (FO4) standard to normalize against advances in process technologies, feature sizes, process corners, etc [7]. As shown in figure 4.1 one inverter feeds four inverters which in turn feed four more. In the diagram I've separated the three sets of inverters with group numbers.

I decided to use group two's inverter delay because it most closely captured realistic operating conditions of our circuits by being loaded on both sides. The implementation of this Verilog circuit involved creating this tree of inverters in separate modules so as to disallow Design Compiler from optimizing away the separate inverters.

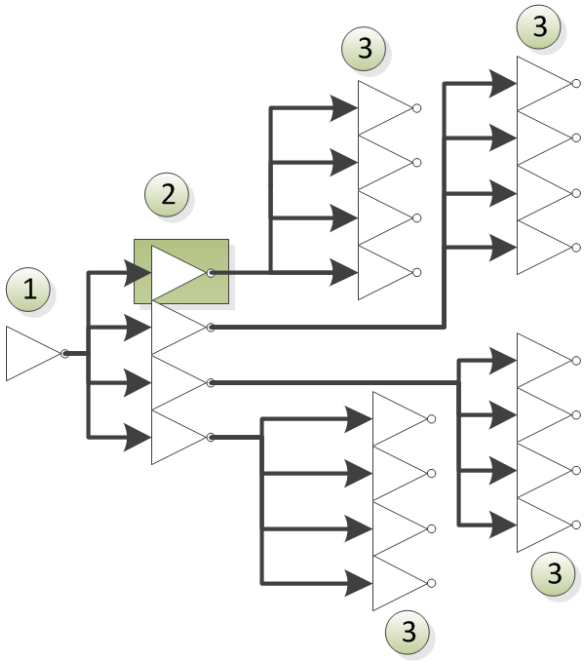

Fig. 4.1. FO4 circuit schematic

4.1.3 The Baseline Coalescer

As mentioned in (3.1) this comparator is attempting to find unique memory addresses within the warp without assuming the addresses are laid out monotonically. This results in an all-to-all comparison of addresses. I implemented the circuit using the naive approach. Figure 4.3 contains a schematic of my design. I used an equality comparator module comprised of XOR gates across every bit to compute if two addresses are the same. If the result of this comparison is 0, the addresses are identical. If the result is a 1, the addresses are different. The current address is compared against all other remaining addresses in the warp in this way. Once this is completed the results of this addresses comparison with all subsequent addresses are ANDed together. If all addresses are different, the output of each comparison is a 1, and the AND will output a 1 to signifying this address is unique.

Fig. 4.2. Base Characterization Corners from [5]

Corner Name	Process (NMOS proc. – PMOS proc.)	Temperature (^{0}C)	Power Supply (V)	Notes	Library Name Prefix
TTNT1p20v	Typical - Typical	25	1.2	Typical corner	Typ
TTHT1p20v	Typical - Typical	125	1.2	Typical corner	typ_ht
TTLT1p20v	Typical - Typical	-40	1.2	Typical corner	typ_ltl
SSNT1p08v	Slow - Slow	25	1.08	Slow corner	max_nth
SSHT1p08v	Slow - Slow	125	1.08	Slow corner	max_hth
SSLT1p08v	Slow - Slow	-40	1.08	Slow corner	max_lth
FFNT1p32v	Fast - Fast	25	1.32	Fast corner	min_nt
FFHT1p32v	Fast - Fast	125	1.32	Fast corner	min_ht
FFLT1p32v	Fast - Fast	-40	1.32	Fast corner	min
Middle Voltage Operating Conditions					
TTNT0p75v	Typical - Typical	25	0.75	Typical corner	typ_tm
TTHT0p75v	Typical - Typical	125	0.75	Typical corner	typ_htm
TTLT0p75v	Typical - Typical	-40	0.75	Typical corner	typ_ltm
SSNT0p65v	Slow - Slow	25	0.65	Slow corner	max_tm
SSHT0p65v	Slow - Slow	125	0.65	Slow corner	max_htm
SSLT0p65v	Slow - Slow	-40	0.65	Slow corner	max_ltm
FFNT0p85v	Fast - Fast	25	0.85	Fast corner	min_tm
FFHT0p85v	Fast - Fast	125	0.85	Fast corner	min_htm
FFLT0p85v	Fast - Fast	-40	0.85	Fast corner	min_ltm
Low Voltage Operating Conditions					
TTNT0p08v	Typical - Typical	25	0.8	Typical corner	typ_ntl
TTHT0p08v	Typical - Typical	125	0.8	Typical corner	typ_htl
TTLT0p08v	Typical - Typical	-40	0.8	Typical corner	typ_ltl
SSNT0p07v	Slow - Slow	25	0.7	Slow corner	max_nt
SSHT0p07v	Slow - Slow	125	0.7	Slow corner	max
SSLT0p07v	Slow - Slow	-40	0.7	Slow corner	max_lt
FFNT0p09v	Fast - Fast	25	0.9	Fast corner	min_ntl
FFHT0p09v	Fast - Fast	125	0.9	Fast corner	min_htl
FFLT0p09v	Fast - Fast	-40	0.9	Fast corner	min_ltl

4.1.4 Fast Neighbor-to-Neighbor Coalescer

As discussed in (3.2) the neighbor-to-neighbor comparator compares adjacent memory addresses in a warp to find which are unique as quickly as possible. The schematic of this circuit is in 4.4. The implementation of this comparator is as follows: Each adjacent address is compared bit by bit via an XOR operation using a comparator module. The result of each bits' comparison is then ORed together in the module. If the result of this operation is found to be a 1, the address is unique and can be coalesced into one memory access. This result is placed in the unique address vector.

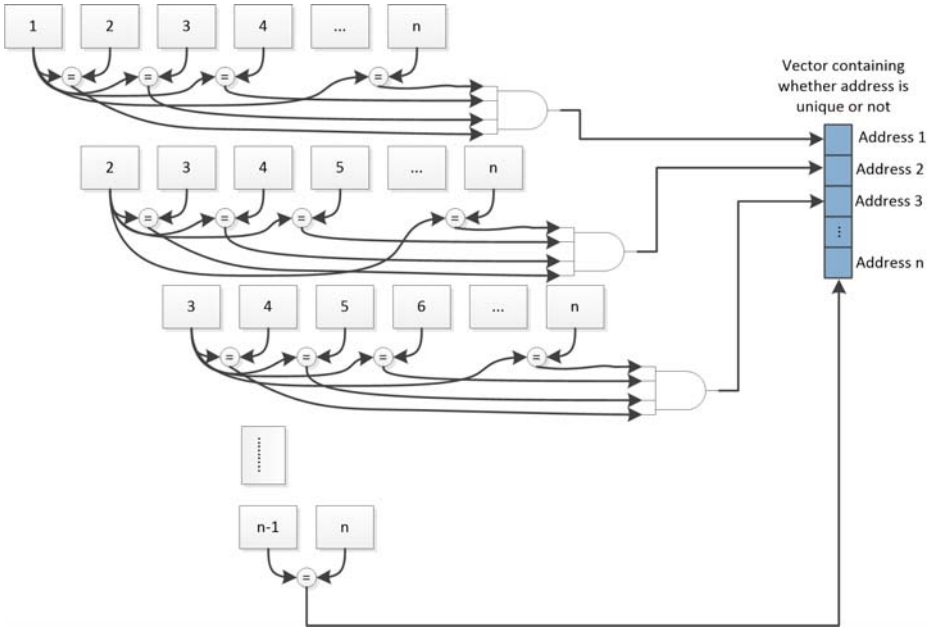

Fig. 4.3. Baseline all-to-all comparator

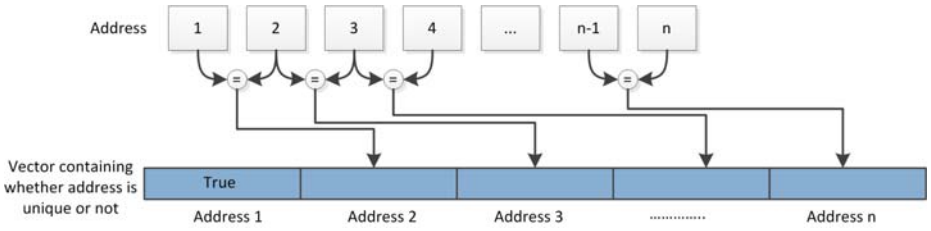

Fig. 4.4. Fast Neighbor-to-Neighbor Comparator

4.1.5 Monotonicity Detector

As discussed in (3.3) the monotonicity detector must take as input 32 addresses (1 warp), each 32-bits wide, and output whether this warp is monotonic or not.

The monotonicity detector is comprised of three main parts: 1

- A subtraction unit that takes in two 32-bit addresses and calculates which of the two is greater
- An equality unit that takes in two 32-bit addresses and calculates if these two are equal

- Glue logic that instantiates 32 (or 64) subtraction and equality units and uses these to calculate monotonicity across all addresses.

This circuit was implemented in Verilog using a Carry-lookahead adder. The subtractor unit is based on a 4-bit Carry-lookahead adder [8]. A schematic of the circuit is included in figure 4.5. Figure 4.6 contains the internal hierarchical design of the CLA subtractor module. The adder is made into a subtractor by inverting the second address fed into the unit and feeding a carry-in of 1 into the circuit. The creation of this entire circuit necessitated both addresses and the result to be extended by one bit, from 32-bit to 33-bits. This allows for the most significant bit (MSB) of the subtraction to denote whether the result was positive or negative, thus allowing me to know which of the two input numbers was larger. If the MSB is a 1 the bigger address is in the subtrahend. If the result is a 0, the bigger address is in the minuend.

The equality unit uses XOR gates between each bit in the address to see if they are the same. The output of these results are then ORed together and if the result of this ORing is 0 the addresses are equal. Otherwise, the addresses are not equal. The result of this unit is then negated to make the continuing logic easier.

Each set of addresses is fed to each of these modules. Starting with the first address, it is compared against the second. The subtraction and equality units find out whether the first or second address is greater and whether the addresses are equal to one another. The result of the subtraction is ORed twice: once each with the results of the equality unit. This tells us if the pattern between these addresses is a $>=$ or $<=$. The results of these OR's are output for each set of addresses. The next set of addresses, the second and third, go through this same process, as well as all subsequent pair of addresses.

The 2 outputs of each of these comparisons ($>=$ and $<=$) are then ANDed together in the following way: all $>=$ outputs are ANDed together and all $<=$ outputs are ANDed together. If either of the results of these AND's is a 1 (found via an OR gate) the addresses maintain a monotonic behavior.

With the addresses extended, a simple 4-bit Carry-lookahead adder was found online and expanded to compute 8-bits, then to do 16, then to the compute all 33 bits. Once the adder was working, I inverted the second of the two inputs and fed in a Carry-in of 1 at the beginning of the circuit, effectively creating a subtractor.

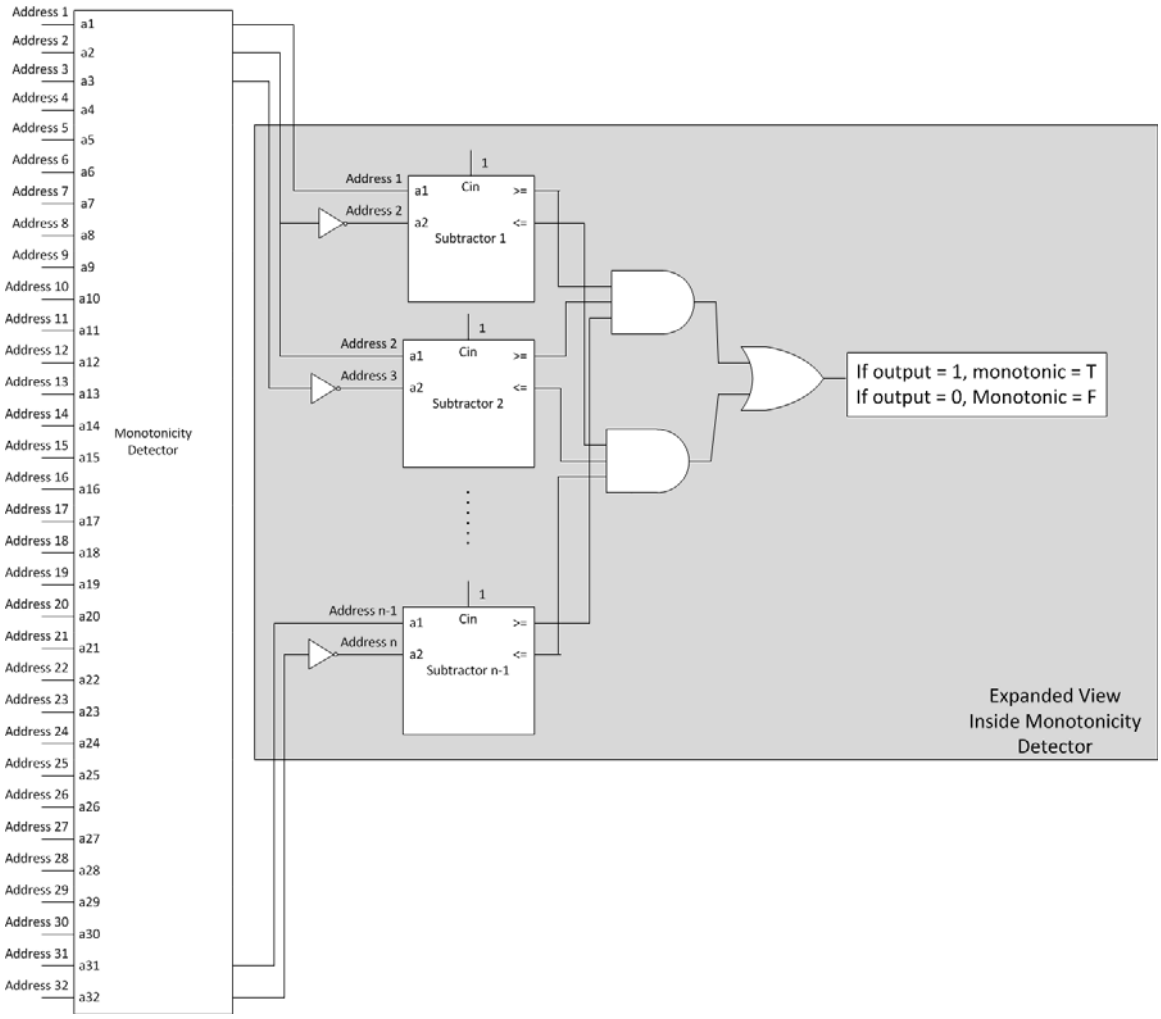

Fig. 4.5. Monotonicity Detector Circuit

4.2 Simulator

In order to characterize the performance of the monotonicity coalescer in GPGPU applications, I use the simulator from the University of British Columbia, GPGPU-

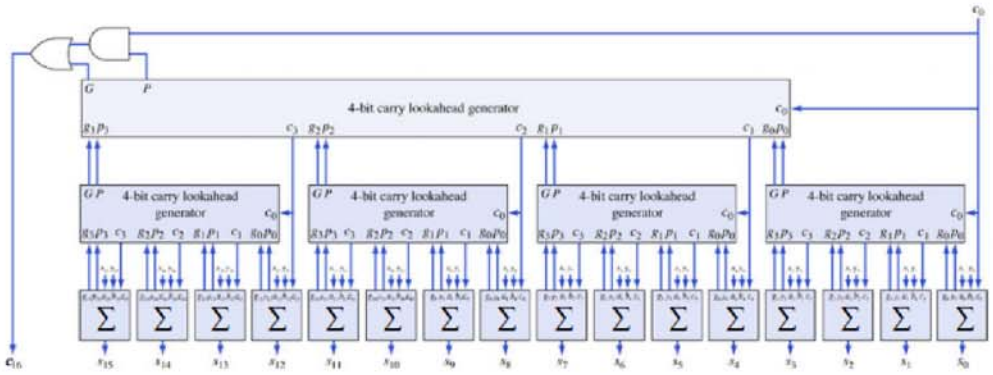

Fig. 4.6. Hierarchical Carry Lookahead Adder from [9]

Sim. This detailed simulator provides the ability to test and measure the effects our coalescer has on the performance of a GPU. The benchmarks and simulator used mirror those of my collaborator, Calvin R. Holic. I use the same computers that Calvin did to run my simulations: a cluster of Intel Xeon E5310 servers running Red Hat Enterprise Linux Workstation release version 6.5. The CUDA benchmarks were compiled with NVIDIA CUDA Toolkit version 4.0.17, just like my collaborator [10].

The simulator was altered as follows: In the fetch stage of each core's pipeline where warp instructions are sent to the core's scheduler, warp instructions were arrested and held for 1 or 3 cycles. The addresses of each thread's memory access for this Warp instruction were forwarded from the instruction parameters and analyzed for monotonicity. If this Warp instruction was found to be monotonic, it was delayed for 1 cycle before being marked ready for issue by the scheduler to this core. If this Warp instruction was found to be break monotonicity, this warp instruction was delayed for 3 cycles before being marked ready for the scheduler to schedule it to this core.

At this point in the pipeline, warps are selected to be issued by the scheduler based on whether or not they're ready. The round robin scheduler employed by GPGU-Sim allows for warps that are not ready to be skipped and other warps to be issued instead [11].

4.3 Benchmarks and Data Sets

The Rodinia benchmark suite version 2.4, created at the University of Virginia, was used to analyze the workload on the GPUs. This benchmark suite was chosen because it helps computer architects explore the architectures of GPUs by providing a diverse set of applications that cover the gamut of typical computation seen by GPGPUs [12,13]. For these benchmarks I used the same configurations as my collaborator. These configurations allow an ample number of memory accesses within a realistic simulation time. Simulation times for specific applications in the suite ranged from one day to one week. CFD Solver is the only benchmark included in the suite that I did not run. This is because it would run for over nine days and crash. Results for the remaining 18 benchmarks are found in the following subsections [10].

4.3.1 Back Propagation (backprop)

As discussed on the Rodinia documentation website for this benchmark, this machine-learning algorithm trains a neural-network. The first phase of the training computes values. In the second phase, these values are compared against the requested values and their difference is used as input in the first phase to train the neural-network. I used the largest of the included data sets to get sizable memory accesses. This data set is larger than that used in real hardware, so it will be sufficient for measuring the effects of the coalescer. [10, 12].

4.3.2 Breadth-First Search (BFS)

This application traverses millions of vertices in a graph. This type of algorithm is common to many scientific and engineering applications. The configuration used for my testing was the largest available in this Rodinia version at 1 million graph vertices [10, 12]. Similar to Backprop, this data set is larger than that used by real

hardware so it should be sufficient for the purposes of measuring the effects of the coalescer.

4.3.3 B+ Tree (b+tree)

B+ is a graph traversal algorithm used primarily in database searches [14]. These searches can be parallelized using GPUs without having to alter the structure of the B+ data structure to check for records. Only one benchmark configuration is included with Rodinia so this is the one used in my experiments.

4.3.4 Gaussian Elimination (gaussian)

This benchmark solves "for all the variables in a linear system" row by row [15]. The algorithm synchronizes iterations but can calculate the values in each iteration in parallel [15]. I match my collaborators configuration of a matrix that's 1024x1024 to thoroughly gather experimental data for this work [10].

4.3.5 Heart Wall Tracking (heartwall)

This medical imaging application reconstructs approximated full shapes of mice heart walls by tracking sample points over 104 ultrasound images of a mice's heart. There are two highly parallel portions of this program: SRAD (discussed later on) and Tracking. The program uses these to remove noise from the images and to track movement of the walls through the images, respectively [15]. I use Rodinia's default configuration in running the benchmark to collect enough data and to be able to run the benchmark within a reasonable amount of time [10].

4.3.6 HotSpot (hotspot)

This application uses a grid layout to estimate "processor temperature based on an architectural floor plan and simulated power measurement" [15]. It solves a series of differential equations to estimate the temperatures [12].

4.3.7 K-means (kmeans)

This benchmark is primarily used in data mining. It identifies related data points and clusters them together to compute a new centroid of this cluster. The data set chosen for my experiments mirrors that of my collaborator. He chose this configuration to fit within reasonable simulation times (4 days in this case) [10].

4.3.8 LavaMD (lavaMD)

This benchmark calculates the potential energy and relocation of particles based on the forces between them in 3D space [15]. My collaborator used the default configuration included with Rodinia at a runtime of 4 days [10]. To run these benchmarks within a reasonable time frame, I use the same configuration.

4.3.9 Leukocyte (leukocyte)

This program tracks white blood cells in video microscopy. The algorithm relies primarily on Gradient Inverse Coefficient of Variation and Motion Gradient Vector Flow to track the cells [12]. I match my collaborators configurations for running these benchmarks to optimize time spent running them and the amount of memory accesses generated by the benchmark [10].

4.3.10 LU Decomposition (lud)

LUD is a matrix algebra algorithm that solves a set of linear equations. When optimized properly this classical algorithm benchmark "exhibits significant inter-thread sharing and row and column dependencies." High inter-thread communication is accomplished through advanced data structures and is important for performance gains in parallel architectures [13]. My collaborator used the largest included configuration to run this benchmark, a 2048x2048 matrix, to increase the number of memory accesses [10].

4.3.11 MUMmerGPU (mummergpu)

As per the documentation on the Rodinia website, this application is a local-sequence alignment program. The GPU is used to align many request sequences to a single request. The query is performed in a pairwise, parallel manner [13]. My collaborator changed the code so that it would run properly on the simulator. The offending call to the GPU which the simulator couldn't handle was changed to a call to the CPU instead. In my experiments I use the same configuration as my collaborator, 50,000 25-character sequences, to collect my results [10].

4.3.12 Myocyte (myocyte)

Myocyte models heart muscle cells (myocytes) in order to research heart failure. The model is comprised of differential equations that are solved in parallel at a very fine grain, similar to ILP [15]. I use the default configuration settings included with Rodinia, just like my collaborator. This default simulates 100 milliseconds [10].

4.3.13 k-Nearest Neighbors (nn)

This dense linear algebra data mining application finds "the k-nearest neighbors from an unstructured data set" [15]. It calculates the Euclidean distance between a

target point and many other points in parallel. This benchmark had to be modified in order to run on the simulator, as my collaborator states [10]. A CUDA call: cudaMemGetInfo() had to be changed to use hardcoded memory information. The data set used for this benchmark is the largest option with 32M records, just like my collaborator.

4.3.14 Needleman-Wunsch (nw)

This is a bioinformatics DNA sequencing and matching application that uses Dynamic Programming. It tries to maximize the score of the path through a matrix that represents potential DNA sequences. Once the optimal path is found, the path is traced backwards to find the optimal alignment of the sequences [15]. My collaborator chose to follow the same 2048x2048 configuration used in [12, 13]. Growing the input size caused memory constraint issues, so I use the same configuration as him [10].

4.3.15 Particle Filter (particlefilter)

Another medical imaging application; Particle Filter uses a structured grid to track objects in video. While it can be used in any video tracking application, this implementation was optimized for tracking leukocyte and myocardial cells [15]. My collaborator chose to use an input set of 400K as per [16]. Since I'm also interested in generating memory accesses to study the effects of the coalescer, I use the same configuration as him [10].

4.3.16 PathFinder (pathfinder)

Pathfinder is a grid traversal algorithm that uses dynamic programming in finding a path on a 2-dimensional grid [15]. The input size used for this application during my simulations was 400K. This input size is used by [16] on real hardware and by

my collaborator during his experiments. This input size was sufficient for him to characterize memory accesses and it is sufficient for my purposes.

4.3.17 Speckle Reducing Anisotropic Diffusion (SRAD)

"SRAD is used in ultrasonic and radar imaging applications" [15]. It removes speckles in an image without removing important features. Speckle removal is accomplished by using a partial differential equation algorithm. [12]. Rodinia includes two versions of this benchmark. My collaborator used the second version which makes more use of the GPUs shared memory. I match my collaborators choice to run this benchmark for 10 iterations, instead of 100 used in [12], due to time constraints.

4.3.18 Streamcluster (streamcluster)

Streamcluster is a dense linear algebra benchmark. The benchmark tries to find the number of medians for points such that every point is near its center. Originally from the Parsec benchmark suite, this version was parallelized by making the "pgain" function parallel [15]. This function calculates the trade-off of creating a new point against the savings of minimizing distances of two points x and y for all points. For my experiments I use the simmedium input data size used in [17] mirroring my collaborators choice due to time constraints [10].

5. RESULTS

The key results of this thesis are summarized below.

- I designed fast memory fetching hardware to reduce redundant cache block accesses that can coalesce accesses across a warp significantly faster when using observed monotonic access trends, but still finds unique accesses when monotonicity does not hold.
- Even though the address monotonicity property largely holds for the Rodinia benchmark suite, coalescing designs did not increase performance of these benchmarks. I offer preliminary evidence that the latency of slow coalescing is hidden by the GPU's natural latency tolerance mechanism. With reduced latency tolerance, the performance gap widens.

I expand on each of the above results below.

5.1 Delay Results

5.1.1 FO4 circuit and clock period selection

To calibrate FO4 delays, I used the the circuit shown in 4.1 from section (4.1.2). To avoid corner cases where the inverter does not drive other inverters (e.g., group three) or are not driven by other inverters (group one), we considered the delay of inverters from group 2 to be an FO4 delay. All other delays are normalized to this FO4 delay unit.

We assume that a single clock cycle should accommodate the monotonicity detection circuit which amounts to 48 FO4 and 56 FO4 for 32 and 64-wide vectors. These numbers also correspond to observed trends wherein modern CPUs (with about 16-20

FO4 delays) use a 3GHz clock speed. GPUs which are typically clocked at 1/3rd of that delay (e.g. 1GHz) should have thrice that clock period. Further, because these delays are approximate we also explored above and below the 48 FO4 delay estimate to 40 and 56 FO4 delays per cycle.

5.1.2 Coalescing Hardware

The results for all 3 coalescing hardware components are discussed below.

The Baseline Coalescer

My implementation of this circuit takes 56 FO4s, for 32 addresses. It takes 112 FO4s, to complete for 64 addresses.

Fast Neighbor-to-Neighbor Coalescer

This circuit is extremely fast for 32 and 64 addresses. The delays are the same because the critical path for either circuit is only 4 elements deep in order to compute results. This translates to 5 FO4s latency for these circuits. As we see below, the clock critical computation for our design is the monotonicity detector rather than the neighbor-to-neighbor coalescer.

Monotonicity Detector

This circuit takes 46 FO4s to compute monotonicity for 32 addresses, and 55 FO4s for 64 addresses. The delay results for the circuits are listed in the tables below in cycles. As you can see, the delays significantly grow with the number of threads in a warp.

In general, the delay results support the key claims that fast-coalescers can exploit monotonicity to speed up coalescing.

Table 5.1.
Delay results for 32 address warp

Clock Delay (FO4)	Naïve Coalescer (cycles), [55.92 FO4]	Fast coalescer (if monotonic)
40	2	2
48	**2**	**1**
56	1* (56 vs. 55.92)	1

Table 5.2.
Delay results for 64 address warp

Clock Delay (FO4)	Naïve Coalescer (cycles), [112.25 FO4]	Fast coalescer (if monotonic)
40	3	2
48	**3**	**2**
56	3	1 (56 vs. 55.17)

5.2 Architectural Simulation Results

Simulations using GPGPUSim (recall the Tesla-like configuration from 4) are shown in 5.1. The simulations plot normalized execution time (lower is better) on the Y axis and benchmarks from the Rodinia suite on the X axis. The key result is that there is no meaningful performance difference between the fast and slow coalescers. Even the ideal coalescer which always completes in 1-cycle is no different from the slow coalescer which always takes 3 cycles.

To understand this surprising result, consider the two possible implementations of coalescing. One alternative is to implement it as a blocking function wherein the whole pipeline stalls for 3 cycles whenever the slow coalescing has to occur. This approach would indeed show the slow down because of coalescer latencies that are exposed. The second and more sophisticated alternative, which is the one I implemented, does indeed delay the affected warp by 3 cycles. However, it allows for other warp contexts to swap in and use the pipeline instead. Such a design allows for the GPUs natural latency tolerance mechanisms to hide the latency of coalescing. However, if the fixed number of warp contexts is reduced, coalescer latency eats into the available latency tolerance. To further explore if this hypothesis is valid, I evaluate

the performance with a reduced number of warp contexts, which reduces the available latency tolerance.

Simulations for a subset of benchmarks with a reduced number (24) of warp contexts were run. Indeed, we saw the gap between the traditional coalescer and the ideal coalescer increase significantly relative to the baseline configuration (albeit from a small base). Figure 5.2 shows the increase in the gap for the subset of benchmarks.

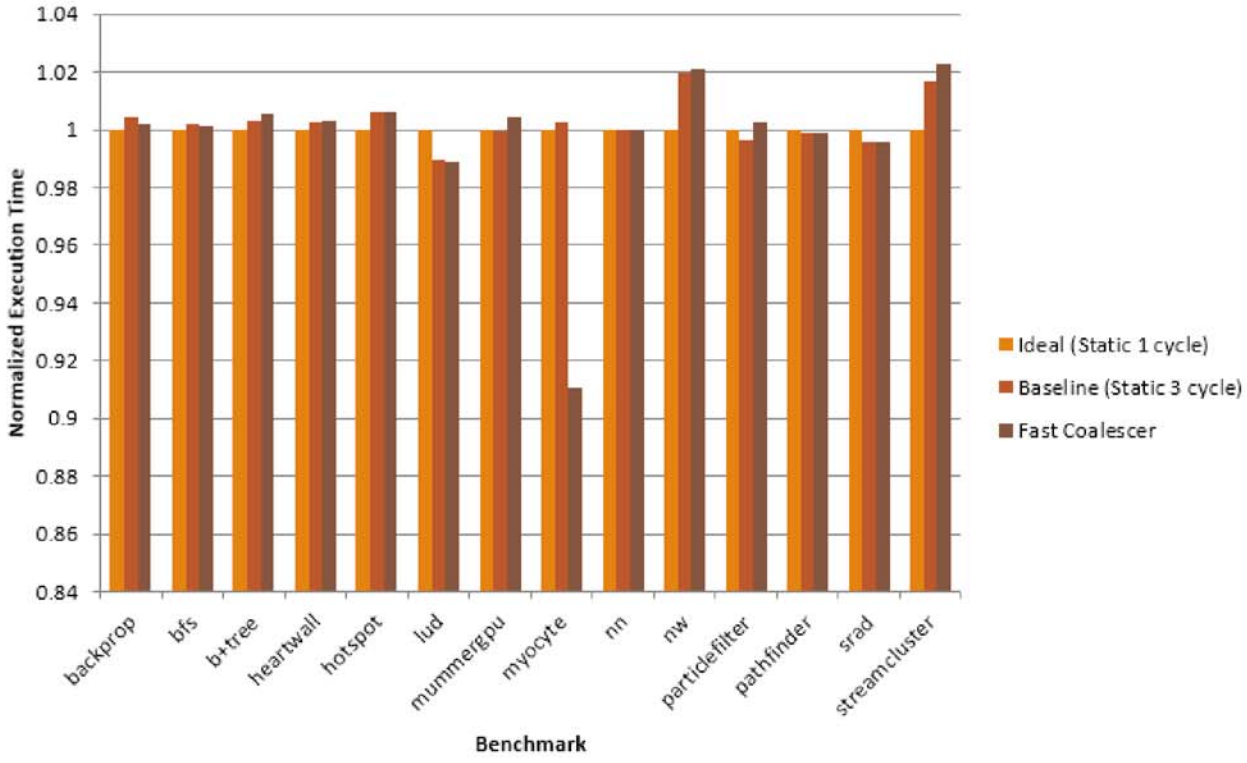

Fig. 5.1. Benchmark Performance

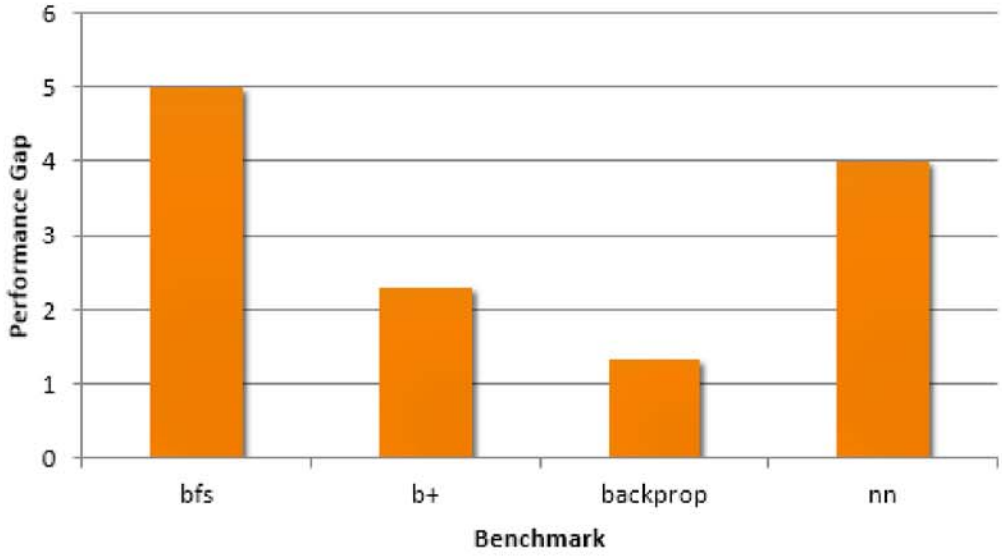

Fig. 5.2. Impact of reduction in latency tolerance (reducing number of warp contexts from 48 to 24)

6. RELATED WORK

My collaborator Calvin Holic investigated the monotonicity of intra-warp memory accesses in his thesis [10]. He showed that the majority of Rodinia benchmarks have very high rates of monotonicity (greater than 98%). He speculated that these observations could be used to influence the design of memory coalescing hardware that would only require neighbor-to-neighbor comparisons within a warp in order to find unique addresses.

Bakhoda et.al. discuss their extensions to GPGPU-Sim in [18]. They discuss Cooperative Thread Array (CTA) distribution and saturation characteristics, interconnection designs, memory coalescing hardware, and caching of the different memories on a GPU. With respect to the simulator, they discuss the warp scheduler. They state: "Every 4 cycles the round robin scheduler issues a warp that is ready. The simulator skips warps, such as those waiting on global memory accesses, and schedules those that have threads that are ready for execution. Therefore, throughput is maintained and long latency operations are tolerated in this way." They mention that for certain applications, performance can be improved by alleviating contention in the memory system. This can be alleviated by not running the full number of threads allowed by hardware.

They go on further to discuss the intra-warp memory coalescing they implement. They group memory accesses into 32 byte payloads that saturate the memory system. Grouping accesses that fetch from contiguous memory regions fully utilize the bandwidth of the memory system and increase performance [4] . They do not use memory access characteristics to inform their coalescing hardware. In this paper they discuss their implementation of an inter-warp coalescer that satisfies redundant memory requests made by multiple warps running on the same shader core.

Jang et.al. investigated techniques for improving memory efficiency of applications in [19]. They create a model that captures memory access patterns. By using characteristic memory access patterns within loop nests they create a guide that can be used to optimize software for GPU architectures.

7. SUMMARY

Graphics Processing Units (GPUs) have increasing relevance for accelerating general purpose computing by exploiting data-level parallelism (DLP). GPUs are able to achieve energy-efficient computation by leveraging the SIMT programming model on SIMD hardware. A result of this choice is that any divergence can significantly degrade performance.

Memory coalescing is a key function that is necessary to minimize unnecessary memory accesses and prevent memory-related warp divergence. The general problem of memory coalescing requires all-to-all comparisons of memory addresses from all threads in a warp, which can be slow. We leverage the observations of a previous study that found that the intra-warp addresses are largely monotonic.

The main contribution of my thesis is the design of a coalescer that is fast when warp addresses are monotonic (the common case) and still eliminates redundant cache block accesses when they are not. Delay analysis shows that the fast coalescer is significantly better than the traditional coalescer for both 32-wide and 64-wide warps.

While the effect on bottomline performance (as measured by simulation) is minimal, it is my conjecture that the traditional coalescer latencies are eating into the available latency tolerance. A more careful architectural study on the opportunity cost of eating into available latency tolerance is left for future work.

LIST OF REFERENCES

LIST OF REFERENCES

[1] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, "From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming," *Parallel Computing*, vol. 38, no. 8, pp. 391–407, Aug. 2012. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0167819111001335

[2] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, "A performance study of general-purpose applications on graphics processors using CUDA," *Journal of Parallel and Distributed Computing*, vol. 68, no. 10, pp. 1370–1380, Oct. 2008. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0743731508000932

[3] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007. [Online]. Available: http://dx.doi.org/10.1111/j.1467-8659.2007.01012.x

[4] *CUDA C Programming Guide*, NVIDIA, http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[5] SYNOPSYS ARMENIA Educational Department, "Digital Standard Cell Library SAED_EDK90_CORE DATABOOK," p. 165, 2011.

[6] Synopsys, *2014 Design Compiler User Guide*, version i-2013.12-sp4 ed., 2014, no. June.

[7] D. Harris and R. Ho, "The fanout-of-4 inverter delay metric," *... Manuscript. http://odin. ac. ...*, pp. 4–5, 2003. [Online]. Available: http://www-vlsi.stanford.edu/papers/dh_vlsi_97.pdf

[8] Jeya, "4 bit carry look ahead adder in verilog," http://tjeyamy.blogspot.com/2012/02/4-bit-carry-look-ahead-adder-in-verilog.html, February 2012.

[9] E. Ackerman and M. Karimi, *Hierarchical Carry Lookahead Adder*, 2007, http://www.cs.sfu.ca/CourseCentral/150/eyal/lectures/CLA.pdf.

[10] C. R. Holic, "Characterizing the intra-warp address distribution and bandwidth demands of gpgpus," Master's thesis, Purdue University, 2014.

[11] *GPGPU-Sim 3.x Manual*, GPGPU-Sim, http://gpgpu-sim.org/manual/index.php/GPGPU-Sim_3.x_Manual.

[12] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, Oct 2009, pp. 44–54.

[13] S. Che, J. Sheaffer, M. Boyer, L. Szafaryn, L. Wang, and K. Skadron, "A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads," in *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, Dec 2010, pp. 1–11.

[14] J. Fix, A. Wilkes, and K. Skadron, "Accelerating braided b+ tree searches on a gpu with cuda," *Proceedings of the 2nd Workshop on Applications for Multi and Many Core Processors: Analysis, Implementation, and Performance (A4MMC)*, 2011.

[15] *Rodinia:Accelerating Compute-Intensive Applications with Accelerators*, Rodinia, https://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php/Main_Page.

[16] J. Shen and A. L. Varbanescu, "A detailed performance analysis of the openmp rodinia benchmark," Delft University of Technology, PDS Technical Report PDS-2011-011, 2011. [Online]. Available: http://www.pds.ewi.tudelft.nl/fileadmin/pds/reports/2011/PDS-2011-011.pdf

[17] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '08. New York, NY, USA: ACM, 2008, pp. 72–81. [Online]. Available: http://doi.acm.org/10.1145/1454115.1454128

[18] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, April 2009, pp. 163–174.

[19] B. Jang, D. Schaa, P. Mistry, and D. Kaeli, "Exploiting memory access patterns to improve memory performance in data-parallel architectures," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 22, no. 1, pp. 105–118, Jan 2011.