The Dissertation Committee for Kai Wang
certifies that this is the approved version of the following dissertation:

Improving Efficiency for GPUs with Decoupled Delegate

Committee:

Calvin Lin, Supervisor

Donald Fussell, Co-Supervisor

Chris Rossbach

Steve Keckler

Improving Efficiency for GPUs with Decoupled Delegate

by

Kai Wang

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2020

For my mother, Zhang Rong.

Acknowledgments

I'd like to thank the many people who helped me during my PhD studies. First, I'd like to thank my advisors, Calvin Lin and Don Fussell. They provided excellent mentorship and helped me grow intellectually. I am fortunate to have them as my advisors. Dr. Lin helped me to see the bigger picture behind specific research questions and helped me to understand the importance of a strong research methodology. I'm especially grateful for his guidance on how to write clear documents. Dr. Fussell helped me to look beyond the immediate research questions and helped me to connect dots between my own research and that of broader fields. I'm especially grateful for his guidance on how to present ideas clearly. The skills I acquired from my advisors will be immensely beneficial to my future career.

Next, I'd like to thank my committee members, Chris Rossbach and Steve Keckler. Dr. Rossbach pointed me to important related work for comparative study. Dr. Keckler raised interesting questions on work. Their value feedback helped me greatly during the process of improving my dissertation.

I'd like to thank my group members, Akanksha, Hao, Zhan, Chirag, Curtis, and Molly for their valuable feedback, and I also broadened my knowledge by learning from their research.

Finally, I'd like to thank my mother, Rong. I wouldn't be able to finish my degree without her support along the way.

Improving Efficiency for GPUs with Decoupled Delegate

Kai Wang, Ph.D.
The University of Texas at Austin, 2020

Supervisors: Calvin Lin
Co-Supervisor: Donald Fussell

GPUs are increasingly used for general-purpose computation. For many applications, GPUs achieve significant performance advantages over CPUs, largely due to GPUs' ability to exploit massive parallelism. However, massive parallelism can cause inefficiency for many operations, such as concurrent data structures. Symptoms include synchronization bottleneck and redundant overheads.

To make such operations efficient, our key strategy is to *decouple* them from the rest of GPU program. Then, we use a few threads acting as a *delegate* to perform the decoupled operations on behalf of all other threads. This reduces the synchronization for decoupled operations because fewer threads are used. In addition, the delegate amortizes overheads for other threads, similar to the way in which vector execution amortizes instruction overheads. The cost of our approach is the need for communication between the delegate and other threads. We develop innovative ways to reduce the communication so that the benefit strongly outweighs the cost. Based on the strategy, we propose three solutions for both regular and irregular workloads.

For regular GPU workloads, our solution reduces repetitive ALU OPs and instruction execution, while reducing memory latency with non-speculative prefetching. This approach enabled our solution to achieve 40.7% speedup and 20.2% energy reduction on average for 29 benchmarks. For lock-based workloads, our solution avoids destructive lock contention in global memory and thus achieves an average speedup of 3.6×, implemented entirely in software. Finally, we introduce a new GPU single source shortest path (SSSP) algorithm with a complex worklist; it offers many benefits compared to a simpler worklist but incurs significant overheads. However, our decoupled delegate approach reduces the overheads and makes the complex worklist design efficient for GPUs. Hence, our solution, implemented in software, achieved an average speedup of 2.8× over 226 graphs compared with state-of-the-art approaches.

Table of Contents

Acknowledgments **v**

Abstract **vi**

List of Tables **xii**

List of Figures **xiii**

Chapter 1. Introduction[1] **1**

1.1 Goal of this Thesis 3

1.2 Worklist-Based Algorithms—problem and strategy 3

1.3 Generalization of the Problem and Strategy 6

1.4 Cost of Decoupled Delegate 9

1.5 Contributions 9

Chapter 2. Related Work[2] **11**

2.1 Repetitive Computation Reduction 11

2.2 Memory Latency Reduction 12

2.3 Fine-Grained Synchronization 13

2.4 Sinlge Source Shortest Path 15

2.5 Delegation 16

[1] Portions of this chapter are based on the following publications:
Affine Computation for SIMT GPUs, ISCA 2017[134]
Fast Fine-Grained Global Synchronization on GPUs, ASPLOS 2019[133]
A Fast Work-Efficient SSSP Algorithm for GPUs, to be appear in PPOPP 2021
My contributions: I'm the first author for all three papers

[2] Portions of this chapter are based on the following publications:
Affine Computation for SIMT GPUs, ISCA 2017[134]
Fast Fine-Grained Global Synchronization on GPUs, ASPLOS 2019[133]
A Fast Work-Efficient SSSP Algorithm for GPUs, to be appear in PPOPP 2021
My contributions: I'm the first author for all three papers

Chapter 3. Decoupled Affine Computation[3] **20**
3.1 Background and Motivation 24
3.2 Our Solution 27
3.3 Implementation 30
3.3.1 **Expansion Units** 32
3.3.2 **Prefetching** 33
3.3.3 **Control Flow Handling** 35
3.3.4 **Divergent Affine Tuples** 36
3.4 Methodology 40
3.4.1 **Baseline Techniques** 41
3.4.2 **Benchmarks** 43
3.5 Evaluation 43
3.5.1 **Instruction Execution Reduction** 44
3.5.2 **Affine Instruction Coverage** 45
3.5.3 **Memory Latency Hiding** 46
3.5.4 **Energy Efficiency** 47
3.5.5 **Area Estimation** 48
3.6 Summary 49

Chapter 4. Decoupled Fine-Grained Synchronization[4] **51**
4.1 Motivation 52
4.2 Our Solution 53
4.2.1 Decoupled Program 55
4.3 Our Software Message Passing System 59
4.3.1 **Our Basic Algorithm** 59
4.3.2 **Our Optimized Algorithm** 61
4.4 Handling Nested Locks 63

[3] Portions of this chapter are based on the following publication:
Affine Computation for SIMT GPUs, ISCA 2017[134]
My contributions: I'm the first author for the paper

[4] Portions of this chapter are based on the following publication:
Fast Fine-Grained Global Synchronization on GPUs, ASPLOS 2019[133]
My contributions: I'm the first author for the paper

4.5 Methodology 66
4.6 Evaluation 69
4.6.1 **Performance** 69
4.6.2 **Bandwidth Benefits** 70
4.6.3 Latency Benefits 71
4.6.4 **Comparison Against Hardware Solutions** 72
4.7 Summary 74

Chapter 5. A GPU SSSP Solution with Decoupled Worklist[5] **75**
5.1 Background 78
5.1.1 **Work Scheduling** 80
5.1.2 **Δ -Stepping** 81
5.2 Motivation 83
5.2.1 **Design Consideration 1** 84
5.2.2 **Design Consideration 2** 84
5.2.3 **Design Consideration 3** 87
5.3 The Overview of Our Solution 88
5.4 A Single Bucket 92
5.5 Multiple Buckets 95
5.6 Dynamic Data Structure 98
5.7 Setting Δ dynamically 102
5.7.1 Finding the *Clip-Point* 106
5.7.2 Changing Δ Based on Utilization 107
5.8 Methodology 109
5.8.1 Graph Inputs 110
5.8.2 Evaluated Prior Implementations 111
5.9 Evaluation 113
5.9.1 Timing Results 114
5.9.2 Work Efficiency 117

[5] Portions of this chapter are based on the following publication:
A Fast Work-Efficient SSSP Algorithm for GPUs, to be appear in PPOPP 2021
My contributions: I'm the first author for the paper

5.9.3 Performance Analysis 118
5.10 Summary 125

Chapter 6. Conclusion[6] **126**

6.1 Why does the decoupled delegate approach work well on GPUs, and what other platforms may benefit from this approach? 127
6.2 Should we make irregular algorithms architectural efficient or algorithmic efficient? Or could we have both? 128

Bibliography **131**

[6] Portions of this chapter are based on the following publications:
Affine Computation for SIMT GPUs, ISCA 2017[134]
Fast Fine-Grained Global Synchronization on GPUs, ASPLOS 2019[133]
A Fast Work-Efficient SSSP Algorithm for GPUs, to be appear in PPOPP 2021
My contributions: I'm the first author for all three papers

List of Tables

3.1 Simulation Parameters 40

3.2 List of Benchmarks – G: GPGPU-sim distribution [10], R: Rodinia benchmark suite [22], C: CUDA SDK, P: Parboil benchmark suite [128] 41

4.1 GTX 1080 ti Specifications 66

4.2 **Latency and Total Execution Time** 72

4.3 **Speedup over respective baselines**—For HQL, the results are from Figure 12 of the paper [146]; the baseline is a simulated Radeon HD 5870 GPU. For BOWS, the results are from Figure 15 of the paper [40]; the baseline is a simulated GTX 1080ti. The HQL paper only provides results for the HT microbenchmark, and the BOWS paper only provides results for HT-1K and ATM-1K; unavailable results are left blank in the table. 73

5.1 RTX 2080 ti GPU 109

5.2 The Distribution of Graph Characteristics—count(% of 226 graphs) 111

5.3 Speedup of ADDS over prior implementations—the distribution of 226 graphs over speedup intervals 114

5.4 The execution time of ADDS, NearFar-OPT (**NF**), Gunrock-Bellman-Ford (BF), and nvGRAPH (NV). The speedup column is ADDS over NF. 115

5.5 Normalized vertex processing count of ADDS (lower the better) . . 117

List of Figures

1.1 (a) data parallelism for graph processing. (b) adding a worklist to the graph algorithm. (c) problems caused by using a worklist 2

1.2 Our Decoupled Delegate Strategy 4

1.3 Code sample: each thread increments an array element—instr0 and instr1 calculate the memory access address, and instr2 and instr3 load the data and perform the computation. 7

3.1 **Decoupling Repetitive Computations**: the figure illustrates our solution 21

3.2 Operand Values–Baseline GPU and Affine Computation 24

3.3 Example Kernel 26

3.4 Affine Values and Affine Tuples for 3 Threads 26

3.5 Percentage of Instructions Computing on Scalar Data and Thread IDs 27

3.6 Decoupled Kernels 29

3.7 Interaction Between the Affine Warp and the Non-Affine Warps . . 30

3.8 DAC Hardware Organization 31

3.9 Re-Convergence Stack for the Affine Warp 35

3.10 Divergent Base-Offset Pairs on SIMT Lanes 37

3.11 Using SIMT Entry as Divergent Condition 39

3.12 Speedup of CAE, MTA, and DAC over the Baseline GTX 480 GPU 42

3.13 Number of Warp Instructions Executed by DAC Normalized to the Baseline GPU 44

3.14 Affine Instruction Coverage of DAC and CAE 45

3.15 Percentage of Affine Global and Local Load Requests on DAC . . . 46

3.16 MTA Prefetcher Coverage 47

3.17 Energy Consumption of DAC Normalized to the Baseline GPU . . 48

4.1 Fine-grained mutual exclusion with **(a)** global locks (baseline) and **(b)** our solution 54

4.2 The basic data structure of a single message buffer and the basic algorithm for reading and writing 60

4.3 Sender Design—using local buffers for aggregated message write . 61

4.4 Receiver Design—using a single warp (the leader warp) for metadata accesses 62

4.5 **Synchronization Server for Two Nested Lock**—operations for handling an offloaded request from client that involves two server TBs 65

4.6 **Speedup of our solution over the state-of-the-art** 69

4.7 **L2 and DRAM traffic of our solution as a percentage (%) of the baseline**—The L2 traffic is the total cache-line accesses of global loads and stores and atomics, including misses to DRAM. The DRAM traffic includes both reads and writes.The traffic includes overhead due to non-coalesced accesses (i.e. unused words in cache lines) . . 70

5.1 An Example Graph—edges are directed with weight 79

5.2 Illustrate SSSP step-by-step 79

5.3 Δ -Stepping's Work Scheduling Data Structure 82

5.4 Implementing a List as Double Buffers 85

5.5 Execution Time against *C* for Two Graphs—the execution time is normalize to the minimum in the series; labels of the x-axis are power of 2 88

5.6 The Overview of Components' Functionalities 91

5.7 How Δ Affects Work Efficiency and Concurrency—pushing 4 vertices (a) to 4 buckets under 3 scenarios: when Δ =20 (c), it has best work efficiency; when increased to 40 (d), it improves concurrency;but when decreased to 5 (b), all vertices are clipped to the last bucket 101

5.8 This Experiment Plots Execution Time and Work Performed Against Δ — the choices of Δ are predetermined and fixed during execution; both *time* and *work* are normalized to the lowest point (lower the better); finally, the experiments are done using 32 buckets 103

5.9 The distribution of ADDS' speedup over NF-OPT correlating to graph degree 114

5.10 The distribution of ADDS' speedup over NF-OPT correlating to graph diameter 116

5.11 The correlation between speedup and work-efficiency (inverse of vertex count); both higher the better. 118

5.12 A.road-USA: s:3.09x, w:0.19x (s:speedup, w:work-efficiency), the figure plots the amount of parallelism (edge count) during the progress of execution (us) . . . 120

5.13 B.BenElechi1: s:4x, w:2.12x . . . 121

5.14 C.msdoor: s:5.57x, w:4x . . . 122

5.15 D.rmat22: s:2.29x, w:2.18x . . . 123

5.16 E.c-big: s:1.6x, w:3.35x . . . 124

Chapter 1

Introduction[1]

Historically, single thread performance (or ILP) has been the main focus of microprocessor design Although ILP is still at a premium even today for desktop and mobile CPUs, circa mid-2000s, the diminishing returns of complex superscalar CPU designs and the demise of Dennard scaling incentivized the industry to explore ways to the transistors on a chip. Evolved from dedicated 3D graphics accelerators, GPUs have emerged as attractive alternatives to CPUs for general–purpose applications.

GPUs deliver throughput by executing many lightweight scalar threads. The threads are organized hierarchically, first into thread blocks (*TBs*) and then into warps. At hardware level, threads within each warp are executed in lock-steps using a vector data path to amortize the instruction handling overhead. Warps within each TB are executed concurrently using fine-grained multi-threading to hide memory latency. This execution model is known as *single instruction multiple thread* (*SIMT*).

[1] Portions of this chapter are based on the following publications:
Affine Computation for SIMT GPUs, ISCA 2017[134]
Fast Fine-Grained Global Synchronization on GPUs, ASPLOS 2019[133]
A Fast Work-Efficient SSSP Algorithm for GPUs, to be appear in PPOPP 2021
My contributions: I'm the first author for all three papers

The key to GPU hardware efficiency is to focus on throughput instead of latency. This means the workloads running on GPUs must have massive parallelism. In general, GPU workloads achieve parallelism by assigning different data elements, such as array elements, to different threads for processing; this is known as *data parallelism*. For regular data structures such as simple arrays and matrices, GPUs are highly efficient due to convergent control flow and coalesced memory access.

The principle of data parallelism also applies to irregular data structures such as graphs. As Figure 1.1 (a) shows, vertices of a graph can be parallelized in a similar way as array elements. These irregular workloads are less efficient for GPUs due to divergent control flow and memory access. However, prior work has shown that GPUs have performance advantages over CPUs for many irregular workloads, and various hardware and software optimizations have been proposed to reduce divergence [85, 114, 64, 68, 115, 23, 119, 86, 105]. Therefore, it is still beneficial to use GPUs for irregular workloads.

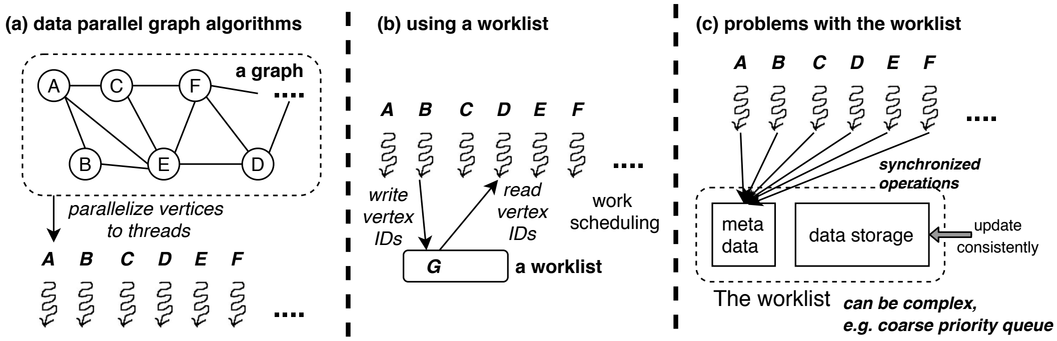


Figure 1.1: (a) data parallelism for graph processing. (b) adding a worklist to the graph algorithm. (c) problems caused by using a worklist

.

1.1 Goal of this Thesis

For both regular and irregular workloads, there are two fundamental requirements for data parallelism to work efficiently. First, there exist many data elements that can be assigned to many threads. Second, threads could process these data elements independently from each other.

However, many GPU workloads contain routines that violate the two requirements. Simply executing such routines with many data parallel threads would introduce inefficiency or even bottlenecks. This problematic approach is still adopted by existing GPU hardware and algorithm designs. The aim of this thesis is to find better ways to handle those routines. In particular, our three projects address three different manifestations of similar problems. All of our projects share the same strategy, namely the **decoupled delegate**.

In the rest of this introduction, we first discuss one of our projects in detail as an example to illustrate the common problem and our strategy. Then we extend the discussion to our other two projects.

1.2 Worklist-Based Algorithms—problem and strategy

The Problem As Figure 1.1 (b) shows, suppose a *worklist* is used for work scheduling and work distribution, where threads write vertex IDs to the worklist and read them out later in a certain order. It is evident that, unlike many vertices of the graph, only one worklist is globally accessed by all threads and threads' operations on the worklist are not independent from each other. Therefore, the worklist violates the two requirements mentioned earlier and does not work well under massive data parallelism.

To be specific, Figure 1.1 (c) shows that the worklist consists of data storage (e.g. for vertex IDs) and metadata. To update the globally shared data storage consistently, all threads must synchronize on metadata operations to avoid race conditions. The massive number of threads in GPUs leads to massive synchronization contention. Moreover, a worklist can be a complex data structure, such as a coarse priority queue, which would mean the metadata operations are complicated. Therefore, massive synchronization on complex metadata operations turns the worklist into a bottleneck, which hinders the graph processing operations.

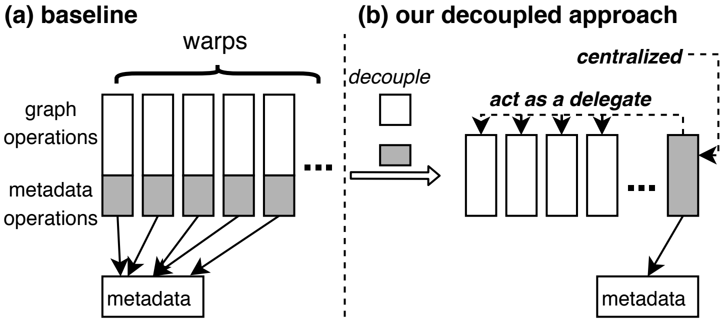


Figure 1.2: Our Decoupled Delegate Strategy

.

Our Strategy Figure 1.2 illustrates our strategy for solving the problem. In the baseline (a), all warps must perform both graph operations and worklist metadata operations, which leads to the problem identified earlier. In our solution (b), we decouple or separate the two types of operations and execute them on two set of warps. We use most of the warps to execute graph operations because they are suitable for massive data parallelism. To simplify discussion, assume we use only one warp to execute worklist metadata operations. The single warp would act as a *delegate* to handle metadata operations on behalf of all other warps.

he main rationale of our approach is to centralize metadata operations to fewer threads, namely the delegate warp. This approach has two benefits. First, metadata operations are centralized, which avoids synchronization across many threads. Second, in the baseline, each thread must pay metadata overheads for individual worklist reads/writes. By contrast, in our solution, the single delegate would amortize metadata overheads over many reads/writes, similar to the manner in which vector execution amortizes the instruction overheads. The reduced synchronization and overheads mean that most of the warps can focus on data parallel graph operations, without being hindered by bottlenecks.

Based on this approach, we develop a new *single source shortest path* (*SSSP*) algorithm for GPUs, implemented entirely in software. We chose the SSSP because work scheduling has a strong impact on its performance; in addition, SSSP is an extensively studied graph problem on GPUs. Without decoupling, existing algorithms adopt simple worklist designs to mitigate the problems mentioned earlier, but they suffer from poor *work scheduling quality*. This problem increases the total amount of work performed or causes hardware under-utilization (see Section 5.2). Our algorithm uses a more complex worklist design to achieve high-quality work scheduling, while making the complex design efficient for GPUs with the decoupled delegate.

We evaluated our algorithm along with seven prior GPU SSSP algorithms (e.g. nvGRAPH [101], Gunrock [135, 138]). Our algorithm achieved an average speedup of 2.8× on a set of 226 graphs, compared with the state-of-art (from Lonestar4.0 [15, 105]). We discuss this project in more detail in Chapter 5.

1.3 Generalization of the Problem and Strategy

We discussed our first project in the section above. To generalize, the problem is that certain operations are problematic for massive data parallelism. Our strategy is to decouple such operations, which are then centralized to a delegate so that fewer threads can be used to execute them. We now discuss how the problem and the strategy relate to our two other projects.

Fine-Grained Synchronization Our second project is a software solution that makes fine-grained locks more efficient on GPUs. Fine-grained locks are used to maintain mutual exclusion for updating globally shared data (items). Similar to the worklist problem, synchronization bottlenecks arise when too many threads attempt to update the same data item and thus contend on the same lock. Massive contention causes polling of lock variables in global memory and serialized critical-section execution.

Similar to our worklist delegate idea, our solution for this problem is to confine the scope of synchronization to fewer threads. In particular, we decouple the critical section and use delegates to execute it on behalf of others, where each delegate is a thread block (TB). Because synchronization is confined within each TB, we can use much faster local scratchpad memory to implement locks, which significantly eases the lock-polling and serialization problems. Non-delegate TBs now must offload their critical- section tasks to the appropriate delegate TB for execution, which is a form of communication. We implement an efficient inter-TB communication mechanism in software, to render such communication efficient. As a result, our solution achieves an average speedup of 3.6× for five benchmarks. We discuss this project in detail in Chapter 4.

The two projects we have discussed are for highly irregular workloads with synchronization bottlenecks. Perhaps surprisingly, our decoupled delegate approach also benefits typical regular workloads.

threads:	t0	t1	t2	t3	
instructions:	**dst operand value**				
0. MUL r0, 4, *TID*	0	4	8	12	**addr. calc.**
1. ADD r1, r0, *base*	100	104	108	112	*repetitive*
2. LOAD r2, [r1]	389	153	207	60	
3. ADD r3, r2, 1	390	154	208	61	

Figure 1.3: Code sample: each thread increments an array element—instr0 and instr1 calculate the memory access address, and instr2 and instr3 load the data and perform the computation.

.

Repetitive Computation Figure 1.3 illustrates a common GPU computation pattern: processing consecutive elements of an array. Notice that *instr0* and *instr1* are highly repetitive in terms of operand values, where adjacent threads are simply offset by 4. In practice, regular workloads have a substantial amount of such repetitive computation used for address calculation and control-flow condition evaluation. In a sense, the repetitive address calculations are overheads of actual computation on array data (*instr2* and *instr3*). On SIMT GPUs, each thread must pay the overheads for computing individual data elements, similar to the worklist metadata overhead problem. In this case, the overheads translate to ALU OPs at the scalar thread level and to instruction execution at the warp level.

Also similar to the worklist problem, we use the decoupled delegate strategy to amortize the overheads. Specifically, for each TB, we use a single warp as the delegate to perform repetitive computation on behalf of other warps in the TB. For example, the delegate executes the address calculation instructions once (see Section 3.1) and then generates cache-line addresses. These are then passed to other warps to be used for memory accesses. The benefit is that non-delegate warps do not have to execute instructions or perform ALU OPs for the repetitive computation. Therefore, our strategy improves performance and energy efficiency by reducing instruction count and ALU OPs.

In addition, our decoupled approach allows non-speculative prefetch to reduce memory latency. The delegate already produces cache-line addresses and can run ahead of other warps independently. Hence, it can retrieve data early from L2 or DRAM to the L1 cache. This principle is similar to *decoupled access/execution* [126]. As a result, our solution (implemented in hardware) achieved 40.7% speedup and 20.2% energy reduction on average for 29 benchmarks.

1.4 Cost of Decoupled Delegate

In general, the decoupled operations are integral parts of the original workloads, so the decoupled delegate must interact with others to work together cooperatively. This situation incurs a communication cost.

We employ two general measures to reduce the cost. First, we decouple the operations appropriately to reduce the required communication. For example, for *repetitive computation*, the delegate passes only cache-line address to other warps, instead of word addresses for individual threads, to reduce the interactions. Second, we maximize the efficiency of the communication. For *fine-grained synchronization*, we implement an optimized software solution for inter-TB communication. For *repetitive computation*, we add specialized hardware in each GPU core to facilitate passing results from the delegate to other warps.

1.5 Contributions

This thesis makes the following high-level contribution:

- Many GPU workloads have abundant parallelism but also contain operations that cause bottlenecks or inefficiency under massive parallelism. We develop a strategy to address the problem by decoupling such operations. They are then centralized to a delegate so that fewer threads can be used to execute them.

At the tactical level, the contributions of this thesis are three solutions that employ the decoupling strategy:

1. For regular workloads, we introduce a hardware solution that decouples scalar-like computations so that they can be performed by a single warp for each TB. This approach reduces redundancy and allows non-speculative prefetching. Hence, our solution improves performance and energy efficiency. Specifically, our solution achieves 40.7% speedup and 20.2% energy reduction on average for 29 benchmarks.

2. For workloads using global memory locks, we introduce a software solution that decouples the critical section so that it can be executed by a single TB instead of all threads. By doing this, our solution can use high-bandwidth and low-latency scratchpad memory for handling lock operations, which improves performance. Specifically, our solution achieves an average speedup of 3.6× over global memory locks.

3. We introduce a new GPU SSSP algorithm that uses a complex worklist to achieve high-quality work scheduling. To deal with the complexity, the principle is to decouple worklist management so that it can be handled by a few warps instead of all threads. For a set of 226 graphs, our algorithm achieves an average speedup of 2.8× compared with the state-of-art.

Chapter 2

Related Work[1]

This chapter summarizes related prior solutions. Section 2.1 discusses prior solutions that reduce repetitive computations on GPUs. Alternative prior solutions have exploited the repetitive behavior for prefetching; these are discussed in Section 2.2. By contrast, our solution achieves both goals with a unified mechanism, which yields wider benefits. Section 2.3 discusses prior solutions related to fine-grained synchronization. Section 2.4 discusses prior graph algorithm designs that solve the SSSP problem. Finally, Section 2.5 discusses how our solutions relate to prior work on delegation

2.1 Repetitive Computation Reduction

Previous work [48, 139, 143] has proposed a dedicated data path for scalar computation to eliminate redundancy and to improve performance and energy efficiency on SIMT GPUs. Collange, et al [27] introduce the notion of affine computation as a generalization of scalar computation. Kim, et al [65] extend this idea by

[1] Portions of this chapter are based on the following publications:
Affine Computation for SIMT GPUs, ISCA 2017[134]
Fast Fine-Grained Global Synchronization on GPUs, ASPLOS 2019[133]
A Fast Work-Efficient SSSP Algorithm for GPUs, to be appear in PPOPP 2021
My contributions: I'm the first author for all three papers

adding a functional unit that can perform affine branch and memory operations.

Our solution extends the special support for affine computation by decoupling its execution onto a separate warp. This approach (1) further reduces computational redundancy, (2) reduces the dynamic warp instruction count, and (3) hides memory latency through a form of non-speculative data prefetching.

Lee, et al [8] present a compiler-based technique to identify opportunities for scalar code to execute under divergent constraints in GPU workloads. Collange, et al [27] present a scalarizing compiler technique for mapping CUDA kernel to SIMD architectures. We build on their insights and present a compiler technique for identifying control-flow divergent conditions.

2.2 Memory Latency Reduction

For scalable speculative prefetching on GPUs, previous work [72, 120, 60, 59, 144] has built on the regularity of memory accesses across different GPU threads to infer prefetches, based on the observed behavior of a few threads. However, GPU prefetchers can sometimes be vexed by useless prefetches for inactive threads. This situation can cause cache pollution and other contention [72]. By contrast, our solution issues early memory requests non-speculatively as a part of the program execution. It does not suffer from mispredictions or early evictions.

Decoupled Access Execution (DAE) [126, 74, 41, 28] is a lightweight memory latency hiding mechanism for in-order processors. The main idea is to decouple memory instructions (the access stream) from other instructions (the execute stream), so that the access stream can bypass memory stalls and issue memory requests early. Arnau, et al. [7] decouple memory accesses from a fragment processor's tile queue, allowing a tile's memory requests to be issued before dispatch. Our solution employs decoupling to affine computations, both to reduce memory latency and to improve computational efficiency.

2.3 Fine-Grained Synchronization

Previous work propose designating one or more threads as servers (or delegates) to handle critical sections for multi-core CPUs with a cache-coherent shared memory interface [20, 83, 111, 129, 54, 104, 43] and for many-core CPUs with both cache coherence and hardware message-passing capability [109] (e.g. Tilera TILE serie [136]). While their designs differ, the principle of transforming synchronization into communication remains the same. The aim is to let clients offload the updates for the same data to the same server so that critical-section updates can be serialized at the servers.

Our work is the first to apply similar principles for GPUs. Since GPU architecture differs significantly from that of CPUs, our solution differs from previous work in several ways.

First, CPUs have fewer hardware threads than do GPUs. Hence, previous work has used individual threads as servers. Since conflicts are serialized to a single thread, no further synchronization is needed for processing requests. Because of the large thread/warp count of GPUs, our solution uses TBs as servers. Hence, when requests are processed, threads in the TB synchronize via their fast local scratchpad.

Second, CPUs have cache coherence and often also sophisticated on-chip interconnect as implicit hardware inter-core communication mechanisms. Previous work has employed software message-passing systems on top of these mechanisms for a relatively small number of threads. By contrast, our solution must be scalable for the much larger number of threads on GPUs, which lack such hardware support for inter-SM (inter-TB) communication.

On the other hand, several issues such as a large number of threads, no coherent L1 cache, and lower memory bandwidth per thread, render fine-grain locks a more severe problem on GPUs than CPUs [146, 40]. Therefore, our solution has greater performance improvement potential.

Yilmazer, et al. [146] propose a hardware-accelerated fine-grained lock scheme for GPUs, which adds support for queuing locks in L1 and L2 caches and uses a customized communication protocol to enable faster lock transfer and to reduce lock retries for non-coherent caches. ElTantawy et al. [40] propose a hardware warp scheduling policy that reduces lock retries by de-prioritizing warps whose threads are spin waiting. Hardware-accelerated locks have also been proposed for CPUs [131, 79, 147, 4].

By contrast, our solution does not require hardware modification. Moreover, a rough comparison with published results (see Section 4.6.4) suggests that our solution performs as well as, if not better than, previous hardware solutions. This is likely the result of our solution addressing the problem at higher level by using scratchpad memories for global synchronization.

2.4 Sinlge Source Shortest Path

Dijkstra's algorithm[38] uses a priority queue to process (i.e. relax) vertices according their tentative distance. This feature makes it the most work-efficient algorithm. By contrast, Bellman-Ford's algorithm [11] processes vertices without order. Originally, the advantage of Bellman-Ford is to allow negative edge weights.

There are parallel versions of both algorithms. Dijkstra's algorithm can be parallelized while persevering the exact ordering semantic. Crauser, et al. [29] propose an algorithm that discovers suitable vertices in the priority queue that can be processed in parallel; for example, if the first N vertices in the priority queue have the same distance, they can be removed in parallel. GPU-based parallel algorithms [84, 103] have also been proposed. Generally, these parallel algorithms are practical only for graphs with highly structured edge weights and connectivity.

Bellman-Ford's algorithm is much more straight-forward to parallelize, since it does not require a priority queue. Many GPU implementations have been proposed [51, 16, 130, 18, 117, 19, 124, 71, 101]. However, the disadvantage of Bellman-Ford is redundant work caused by processing vertices in an arbitrary order. Meyer, et al. [88] propose Δ -Stepping as a midway between Bellman-Ford and Dijkstra'. The algorithm processes vertices in approximate order, so it improves work efficiency when compared to Bellman-Ford; however, it does not use a priority queue, which makes parallelism possible.

Many GPU adoptions of Δ -Stepping have been proposed [135, 132, 12, 31, 9]. Ingeneral, these previous solutions render the work-scheduling data structure scalable for GPUs by simplifying the design. However, by doing so, they sacrifice the quality of work scheduling; in particular, they achieve sub-optimal work efficiency and concurrency, which limits their performance.

Our algorithm is a GPU adoption of Δ -Stepping. Compared with previous solutions, ours uses a more sophisticated work-scheduling mechanism that achieves better work efficiency and concurrency. At the same time, we implement a sophisticated mechanism to be efficient for GPUs. As a result, our solution performs far better than previous ones.

2.5 Delegation

Delegation is proposed by previous work [37, 42, 107, 121, 145, 54, 104, 109, 43, 108, 20, 83, 111, 116] as a way to avoid the inefficiency caused by locking for multi-core or multi-socket CPU systems. In this regard, our fine-grain lock project is closely related to prior work, and we have demonstrated that the notion of delegates can be adapted so that they work on GPUs. Our other two project uses delegation in a broader sense, and we have demonstrate that the idea can used beyond locking problems.

The main idea of delegation is that one or several threads act as servers that execute the critical section on behalf of other threads. Prior delegation proposals can be categorized into two main approaches—combiner and dedicated delegate.

As to the combiner approach [37, 42, 107, 121, 145, 54, 104, 109, 43], one thread is dynamically selected to act as the temporary delegate, i.e. combining;

the rationale is to avoid wasting a thread for handling critical section execution exclusively, especially for multi-core CPUs with a small hardware thread count.

As to the dedicated delegate approach [108, 20, 83, 111, 116], as the name suggests, delegates are dedicated server threads, which avoids the overhead of dynamical combining; this approach achieves better performance for systems with a moderate thread count. Lozi, et al. propose RCL [83, 108] with a drop-in replacement for locks but has additional complexities and overheads due to the need for supporting legacy programs. Roghanchi, et al. propose ffwd [116], which offers somewhat restricted functionalities, i.e. no support for certain legacy programming patterns, but achieves better performance when compared to RCL.

We now discuss how our projects are related to prior work.

Our fine-grain lock project is the first to adopt a dedicated delegate approach on GPUs. Due to the significant difference between GPU and CPU architectures, our solution differs from prior solution in two ways. First, each delegate is a thread block instead of a thread in our solution, and we take advantage of GPUs' fast local scratch-pad memories for locking within each delegate thread block. Second, most of the complexities of a delegate design lies in handling the interaction between the client and the delegate threads, where our solution uses an entirely different approach for GPU architecture.

To be specific, in both RCL [83, 108] and ffwd [116], each client CPU core owns a cache line for writing its request to the delegate (i.e. request buffer), and delegate polls those cache lines in round-robin to find and to retrieve clients' requests. Clearly, we could modify this approach for GPUs, e.g. each warp instead of a thread owns a request buffer (layout in memory as multiple cache lines), etc. but fundamentally, the idea of letting a client write a fixed location is not suitable for GPUs. First, GPUs have a large number of concurrent threads with sparsity in terms of requests to the delegate; therefore, polling through the request fields of threads/warps one by one is not efficient. Second, also due to sparisty, those requests would not be in consecutive memory locations, which is detrimental to coalesced memory accesses and L2 cache footprint. Therefore, in our solution, each delegate owns a concurrent read-write-able FIFO queue so that clients only write valid requests in consecutive locations. Furthermore, we apply various optimizations to promote coalesced memory accesses and to amortize overheads when writing and reading the FIFO queue. Therefore, our approach is more suitable for the highly parallel nature of GPU-like architecture.

As to the worklist manager in our SSSP project, although the manager can be considered as a delegate in a broader sense, the key difference is that the manager interacts with the clients at a much coarser granularity. To be specific, in both CPU and our own GPU lock-replacement delegate designs, the delegate handles individual threads' requests; clearly, this is unnecessary for the purpose of maintaining a coarse-grain priority queue (i.e. the buckets), since the manager does not care about the individual work items written by client threads. Instead, the manager only needs to determine when and which work-items (in a range) can be read out for maintaining consistency and ordering and to determine which client thread blocks are available for accepting work; in addition, features, such as dynamic delta adjustment, also only require coarse-grain information. Therefore, for our worklist solution, the manager acts as a controller for the clients that accesses various meta-data only instead of a true delegate in the traditional sense.

Chapter 3

Decoupled Affine Computation[1]

GPUs are optimized for regular data parallel workloads. Data parallelism is commonly expressed with single instruction multiple threads (SIMT), where data elements are parallelized to lightweight scalar threads, which are then grouped into vector warps at the hardware level (e.g. Nvidia GPUs) or at the assembly or compiler level (e.g. AMD GPUs). However, regular workloads often exhibit a high degree of repetitiveness in terms of address calculation and control flow. The repetitive computations must be duplicated to all threads or warps; in particular, it causes unnecessary instruction executions and ALU OPs.

Our main idea is simply to **de-duplicate** such computations. This is done by decoupling the original kernel into the *affine kernel* containing the repetitive portion (e.g. instr0, instr1 in Figure 1.3) and the *data parallel* (DP) kernel containing the true data parallel portion (e.g. instr2, instr3). As Figure 3.1 ① shows, for each thread block (TB), we launch N warps for the DP kernel, which is the same as for the original kernel, but only one warp for the affine kernel. The repetitive instructions are thus executed by a single warp instead of decoupling to N warps. Therefore,

[1] Portions of this chapter are based on the following publication:
Affine Computation for SIMT GPUs, ISCA 2017[134]
My contributions: I'm the first author for the paper

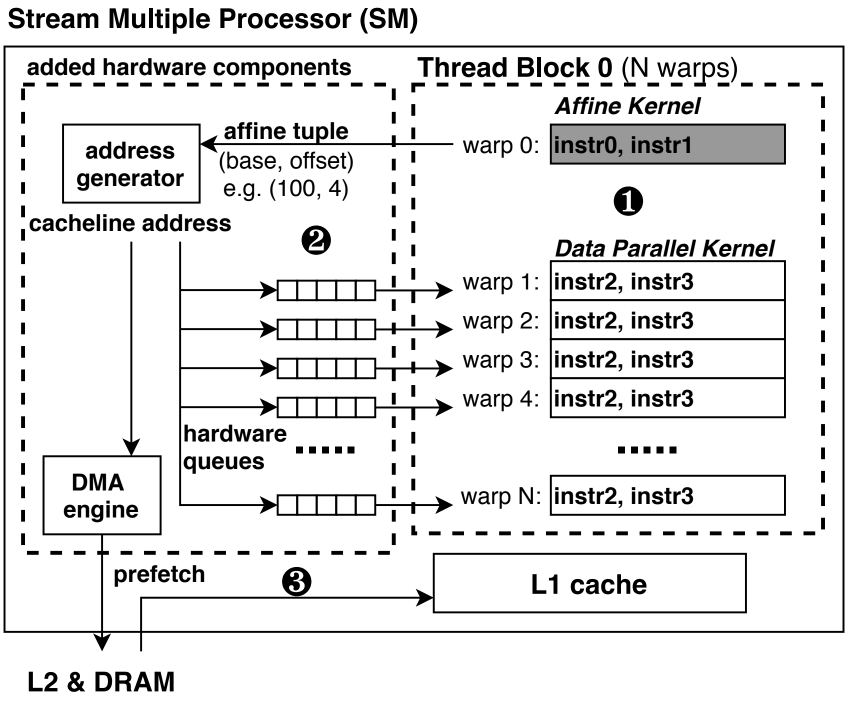


Figure 3.1: Decoupling Repetitive Computations: the figure illustrates our solution

.

our solution reduces the instruction execution count by a factor of N.

The two decoupled kernels work cooperatively as a single kernel, so the results produced by the affine warp must be passed efficiently to the DP warps. To facilitate such communication, we use a compact encoding format and add specialized hardware to each GPU core (SM), as shown in Figure 1.3 ②. For example, instr0 and instr1 calculate the memory addresses for array access. Because of the repetitiveness, the addresses can be encoded as a base-offset pair, such as (100, 4), known as the *affine tuple*. The affine warp produces the affine tuple and pass it to the *address generator*, which generates cache-line addresses and then buffers them in a set of *hardware queues*, one for each DP warp. In contrast to a baseline SIMT GPU, which calculates addresses thread-by-thread and then coalesces them into cache-line addresses, our solution generates cache-line addresses directly from the affine tuple. This approach improves the efficiency. In addition to address calculation, a similar mechanism is used for repetitive control flow condition evaluations (not shown). In this case, our solution generates predicate bit-masks directly.

In addition to de-duplication, our solution allows non-speculative prefetch of memory data. As Figure 3.1 ③ shows, the affine warp already produces cache-line addresses and it can be run ahead of DP warps due to decoupling. Therefore, we add a *DMA engine* that retrieves data early from L2 or DRAM to the L1 cache, which be accessed by DP warps later. The principle is similar to *decoupled access/execution* [126].

Some previous solutions have reduced repetition without decoupling. For example, AMD GPUs [6] use explicit scalar and vector instructions and a dedicated scalar data path so that repetitive instructions are handled with scalar ALU OPs instead of vector OPs. Kim, et al[65] propose a solution that achieves a similar effect for Nvida-like SIMT GPUs by detecting scalar executable instructions in run-time. The main drawback of previous solutions is that they de-duplicate only repetitive computation within each warp by transforming vector ALU OPs into scalar ones; repetitive instructions are still duplicated to all warps because they are not decoupled, and are executed redundantly. By contrast, our solution de-duplicates them at TB scope and thus reduces instruction executions.

In summary, our solution improves performance and energy efficiency by reducing dynamic instruction count, ALU OPs, and memory latency. For evaluation, we implement our solution on GPGPU-Sim simulating a GTX 480 GPU. For comparison, we also implement *CAE* [65], a previous repetition-reduction solution, and *MTA* [72], a previous speculative prefetching solution. We use a set of 29 GPGPU benchmarks, of which 18 are memory-bound and 11 are compute-bound. For all 29 benchmarks, our solution achieves 40.7% speedup and 20.2% energy reduction on average. For the 18 memory-bound benchmarks, our solution achieves a 44.7% mean speedup compared to the 16.7% achieved by MTA. For the 11 compute-bound benchmarks, our solution achieves a 34.0% mean speedup, compared to CAE's 11.0%.

The rest of this chapter is organized as follows. Section 3.1 formalizes the notion of repetitive computations with a previously proposed concept known as *affine computation.* Section 3.2 discusses our idea of decoupling affine computation. Section 3.3 shows how our solution can be implemented at the hardware and compiler levels. In Sections 3.4 and 3.5, we present the experimental evaluation of our solution.

3.1 Background and Motivation

Collange, et al [27] introduces *affine computatio*n to represent repetitive computations. The idea is based on the observation that GPU workloads often use scalar data, such as kernel parameters, and the thread ID to map memory accesses and control flow to threads. Therefore, the operand values of many ALU instructions often exhibit a high degree of regularity across threads. This characteristic can be exploited to reduce ALU operation.

	Baseline GPU					**Affine Computation**
Operands	**SIMT Lanes**					**Affine Tuple**
	0	1	2	...	31	(base, offset)
A	100	101	102	...	131	(100, 1)
B	200	201	202	...	231	(200, 1)
C=A+B	300	302	304	...	362	(300, 2)
D	2	2	2	...	2	(2 , 0)
E=CxD	600	604	608	...	724	(600, 4)

Figure 3.2: Operand Values–Baseline GPU and Affine Computation

Figure 3.2 shows the per-thread operand values of an ADD OP and an MUL OP as examples of the regularity. First, it is evident that the value of A starts at 100 in lane 0 and then increases by 1 with each successive thread. Hence, the entire vector value of A can be represented as an affine tuple (100, 1), where 100 is a base and 1 is the offset. Algebraically, an affine tuple represents values as a function of thread ID:

$$operand\_value = base + thread\_ID \times offset \tag{3.1}$$

Similarly, the scalar B can be represented as the tuple (200, 1). Next, the value of C (as the result of ADD OP) can be represented as the tuple (300, 2). Instead of performing one ALU per lane in vector computation, the C tuple can be derived with A and B affine tuples using just two additions. The first adds A's base to B's base; the second adds A's offset to B's offset, producing (300, 2). The C affine tuple can then be used as a source operand for subsequent affine computations.

The value of D can be represented as (2, 0), where the offset is 0. Such an operand is called a scalar operand. The MUL OP between C and D can also be calculated using affine arithmetic with two multiplications, one to multiply C's base with D's base and the other to multiply C's offset with D's base; the result is (600, 4). However, for affine MUL OP, one of the operands must be scalar; alternatively, vector computations must be used, because the result cannot be represented as an affine tuple.

Figure 3.3 shows a sample CUDA kernel in a relatively concrete example. Figure 3.4 shows how address *addrA* is derived entirely from affine arithmetic. The same applies to address *addrB* and predicate *p0* (not shown).

A sequence of affine computations can continue as long as both source and destination operands can be represented as affine tuples. Otherwise, affine tuples must be expanded into concrete values. For memory instructions with affine addresses (e.g. *addrA*) and for predicate computation instructions with affine operands

```
void example_kernel(int A[],int B[],
    int dim,int num)
{
int tid=blockIdx.x*blockDim.x+
    threadIdx.x;
for(int i=0;i<dim;i++)
{
int tmp=A[i*num+tid];
B[i*num+tid]=tmp+1;
}
}
```

(a) CUDA Code

```
 1  mul r0, blockIdx.x, blockDim.x;
 2  add tid, threadIdx.x, r0;
 3  mul r1, tid, 4;
 4  add addrA, A[], r1;
 5  add addrB, B[], r1;
 6  mov i, 0;
 7  LOOP:
 8  ld.global tmp, [addrA];
 9  add r2, tmp, 1;
10  st.global [addrB], r2;
11  add i, i, 1;
12  mul r3, num, 4;
13  add addrA, r3, addrA;
14  add addrB, r3, addrB;
15  setp.ne p0, dim, i;
16  @p0 bra LOOP;
```

(b) Pseudo Assembly Code

Figure 3.3: Example Kernel

	Operand Value			Affine Tuple
operand name	Thread 0	Thread 1	Thread 2	(Base, Offset)
A[]	0x80000	0x80000	0x80000	(0x80000,0x0)
#3, r1	0x0	0x4	0x8	(0x0,0x4)
#4, addrA	0x80000	0x80004	0x80008	(0x80000,0x4)
#12, r3	0x1000	0x1000	0x1000	(0x1000,0x0)
#13, addrA	0x81000	0x81004	0x81008	(0x81000,0x4)

Figure 3.4: Affine Values and Affine Tuples for 3 Threads

(e.g. #15), expansion can be handled efficiently in most cases. For example, *addrA*, has an offset of 4, and 32 consecutive threads of a warp can be serviced by a single cache-line address. Hence, a warp can be expanded by a single ALU operation. If an affine tuple cannot be expanded into predicate bit vectors or addresses, then it must be expanded into concrete vector values by evaluating function3.1 explicitly for each thread.

In addition to ADD and MUL, similar ALU operations (e.g. sub, shl, or

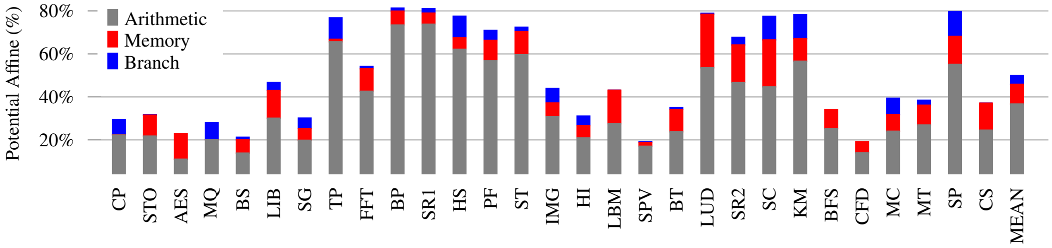


Figure 3.5: Percentage of Instructions Computing on Scalar Data and Thread IDs

mad) are supported. These simple operations constitute a large portion of computations on scalar data and thread IDs. They are often used for address and predicate bit vector computations for regular workloads, where memory accesses and control flows are generally not data-dependent. Figure 3.5 shows that for our 29 benchmarks, about half of the static instructions are potentially affine instructions. They are deemed such because two factors, namely control flow divergence and instruction type, can force them to execute in non-affine warps. Previous affine computation techniques [27, 65] cannot execute affine computation after control flow divergence. By contrast, our solution uses compile-time analysis and run-time mechanisms to execute affine instructions after limited forms of divergence.

3.2 Our Solution

In the previous section, we show that an affine computation (i.e. arithmetic for affine tuples) can replace vector computation for a warp instruction. Indeed, this is how the previous solution [65] exploited affine computations. Affine (functional) units are added to the GPU so that a warp instruction is issued to either a vector unit or an affine unit.

We recognize that greater efficiency can be gained by exploiting across multiple warps. The affine property (i.e. constant offset of values) often exists for adjacent warps, since they are assigned with consecutive thread IDs. Hence, it is possible to execute a single warp instruction with affine computation to replace instruction execution of multiple adjacent warps. Not only does this further reduce ALU operations but it also reduces the dynamic instruction count. Therefore, we conclude that the previous solution executes affine instructions redundantly for each warp.

Furthermore, since affine computations are often used for address calculations, it is possible to exploit them for non-speculatively prefetching memory data for latency reduction. To achieve this, our solution decouples affine instructions (i.e. instructions that are eligible for affine computation) and non-affine instructions into separate instruction streams. The non-affine stream is still launched as multiple warps (*non-affine warps*) for fine-grained multi-threading, while using vector computation, just as in the baseline SIMT GPU. The decoupling allows our solution to specialize in handling the affine instruction stream. Here, a single warp (the *affine warp*) is launched per TB for affine computation, so that affine instructions are not redundantly executed for each warp. Furthermore, the decoupling allows the affine warp to run ahead independently of non-affine warps to prefetch. We named our solution the decoupled affine computation (DAC).

Figure 3.6 shows that the original code from Figure 3.3 is compiled into two instruction streams. It is evident that memory accesses are decoupled into two

```
 1  LOOP:
 2  mul r0, blockIdx.x, blockDim.x;
 3  add tid, threadIdx.x, r0;
 4  mul r1, tid, 4;
 5  add addrA, A[], r1;
 6  add addrB, B[], r1;
 7  mov i, 0;
 8  LOOP:
 9  enq.data addrA;
10  enq.addr addrB;
11  add i, i, 1;
12  mul r3, num, 4;
13  add addrA, r3, addrA;
14  add addrB, r3, addrB;
15  setp.ne p0, dim, i;
16  enq.pred p0
17  @pred bra LOOP;
```

(a) The Affine Instruction Stream

```
1  LOOP:
2  ld.global tmp, deq.data;
3  add r2, tmp, 1;
4  st.global [deq.addr], r2;
5  @ deq.pred bra LOOP;
```

(b) The Non-Affine Instruction Stream

Figure 3.6: Decoupled Kernels

parts. The affine warp uses affine tuples to compute the memory addresses and then sends the affine tuples to the non-affine warps by enqueueing them to the address queue. The non-affine warps then dequeue the concrete values. For example, the store instruction in line 10 of the original code (*st.global[addrB], r2;*) is translated into line 10 in the affine instruction stream (*enq.addr addrB;*) and line 4 in the non-affine stream (*st.global [deq.addr], r2;*). Predicate computation instructions are handled in a similar manner.

The affine warp can run ahead of the non-affine warps to hide memory latency because the affine warp operates on read-only data, such as thread IDs and kernel parameters; it does not modify memory. Therefore, the affine warp can execute independently from the non-affine stream. More importantly, the affine warp fetches memory – but does not use it – on behalf of the non-affine warps, so the

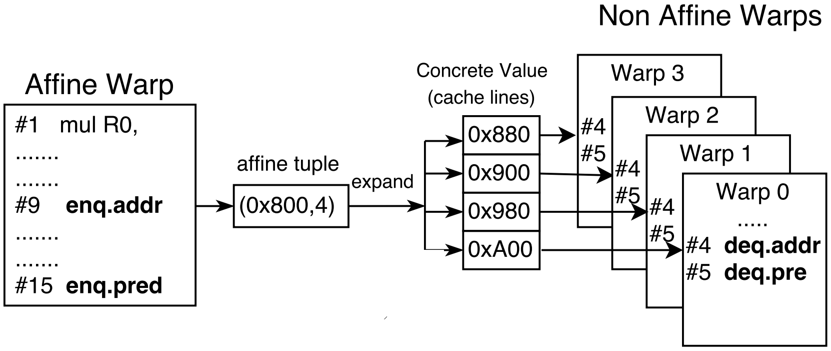


Figure 3.7: Interaction Between the Affine Warp and the Non-Affine Warps

affine warp can issue memory requests while bypassing stalls. For example, in Figure 3.6a, line 9 of the decoupled kernel loads the data pointed to by *addrA* in a loop. The affine warp can request *[addr]* for the next iteration without waiting for the requests of the previous iteration to finish, because only non-affine warps operate on data *[addr] (tmp+1)*. In other words, the original program's data dependence on *[addr]* is broken by executing the use of the data on the non-affine warps.

3.3 Implementation

Specialized hardware is added to support the affine stream at run-time. Specifically, the enqueue and dequeue instructions trigger hardware mechanisms that (1) expand affine tuples into concrete values and (2) coordinate the two streams at runtime. Figure 3.7 shows the interaction between the two instruction streams in hardware. The single affine warp sends a tuple for expansion when executing an enqueue instruction. An affine tuple is expanded into concrete values (cache-line addresses or predicate bit vectors) and buffered for each non-affine warp. Non-affine warps then retrieve the concrete values from buffer when executing dequeue instructions.

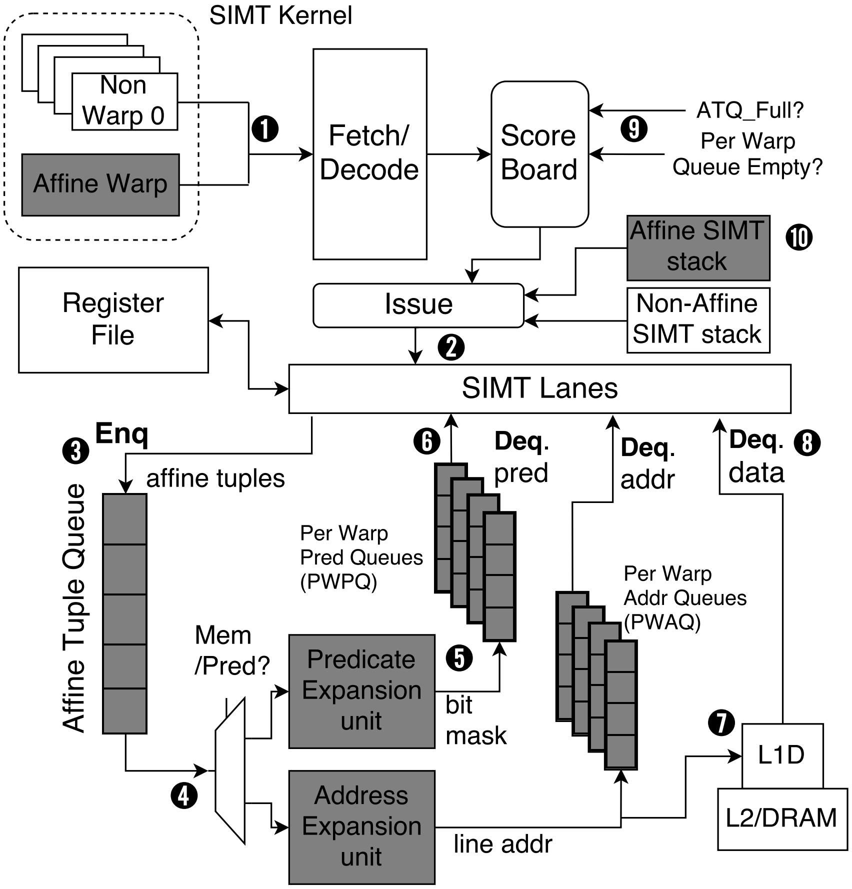


Figure 3.8: DAC Hardware Organization

The corresponding hardware is shown in Figure 3.8, with the baseline components appearing in white and the added components in gray.

3.3.1 Expansion Units

When the affine warp executes an *enq* instruction, the associated affine tuple is enqueued ③ to the tail of the *affine tuple queue* (ATQ). The *predicate expansion unit* or the *address expansion unit* then fetches the affine tuple from the head of the ATQ ④. Using the affine tuple, the expansion units generate predicate bit-masks or coarse-grain addresses for each non-affine warp. A predicate bit-mask is then, for example, enqueued ⑤ to the tail of the *per warp predicate queue* (PWPQ). As the name suggests, there is one PWPQ for each concurrent non-affine warp. Finally, when a non-affine warp executes a *deq.pred* instruction, the bit-mask is dequeued ⑥ from its PWPQ, and the bit-mask is used to set the predicate register. The process is similar for address expansion (*enq.addr*). Essentially, the expansion units are optimized the common cases for regular workloads.

The address expansion unit (AEU) generates cache-line addresses directly from the affine tuples, without generating addresses for individual threads. For example, with an offset of 4, adjacent warps access consecutive 128-byte cache lines; therefore, the AEU will generate a sequence of consecutive cache-line addresses for warps by accumulating 128 at a time from the starting address of the TB. The accumulation is always done by 128 at a time regardless of the offset value. For example, for an offset of 8, each warp receives two consecutive cache-line addresses. To indicate which word of the 128-byte data a non-affine thread should access, the

AEU generates a bit-mask that accompanies the cache-line address. For instance, an offset of 4 generates a bit-mask *111111...* to indicate that all 32 words are accessed. Similarly, an offset of 8 generates *101010...* to indicate the access of every other word in the region. The address and bit-mask are then pushed to the PWAQ as a compact record, which is then used by non-affine warps to access memory.

The predicate expansion unit (PEU) generates predicate bit vectors for the non-affine warps. Predicate bit vectors are generated by comparisons (e.g. greater than) between two operations. For a predicate computation to be decoupled, DAC requires that one operand (the *scalar operand*) be a scalar, where all threads in the same block have the same value. If the other operand is also a scalar, then only a single comparison is needed for all threads in the block. For our 29 benchmarks, this case constituted 64% of the decoupled predicate computations.

If the other operand is not a scalar, then as with the AEU an accumulation is performed. The idea is that if the first and last thread values of a warp are larger or smaller than the scalar operand, then due to the constant offset of the affine operand [65], all threads in between must have the same result. Thus, a convergent bit-mask is generated for a warp with only two comparisons. This case constitutes 93% of the decoupled predicate computations, including the scalar case. For the remaining 7%, the SIMT lanes are used to compare all 32 threads of a warp.

3.3.2 Prefetching

For the *enq.data* instruction (global and local loads), the AEU also sends requests to the L1 cache or the lower levels of the memory hierarchy on a miss.

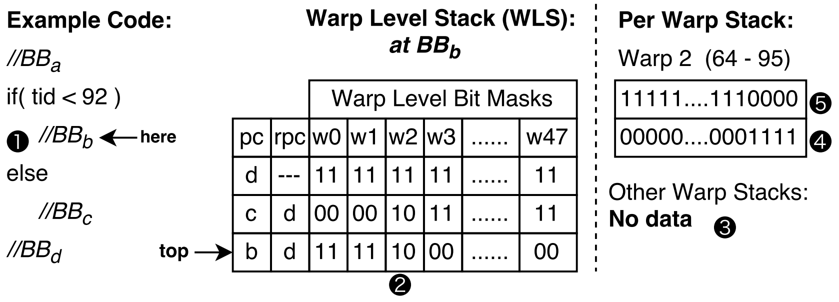
To avoid the eviction of requests that arrive before their demand accesses, DAC adds lock counters to the tag array, which temporarily disable replacement for a cache-line. The AEU locks cache lines upon issuing memory requests, and the non-affine warp unlocks cache lines upon access. Unlike speculative prefetching, the early requests are guaranteed to be accessed by the non-affine warp and eventually unlocked, so this locking is safe. Memory accesses that are not affine must be issued by the non-affine warps, but deadlock is avoided because the AEU can lock at most (N$-$1) sets of an N-way cache. It is possible to create contention between locked cache lines and non-affine cache lines, but we do not observe this to be a problem because usually only a small portion of the cache is locked at any time.

To avoid deadlock when all entries of a set are locked by prefetch and the non-affine on-demand access from the SIMT warp cannot be issued. At least one entry of a set must remain unlocked at any time, so that this line can be used by the non-affine access.

Early memory accesses can cause conflicts with barrier operations. To avoid conflicts, barrier instructions are replicated for both the affine and non-affine warps. The AEU handles barrier operations on behalf of the affine warp. When the affine warp executes a barrier instruction, the AEU disables expansion for the target non-affine blocks; the AEU only issues memory requests for non-affine blocks that pass the barrier. Affine warps themselves do not access memory; they only access read-only data, such kernel parameters. Therefore, they are not affected by these barriers.

3.3.3 Control Flow Handling

The affine and non-affine instruction streams work as a single kernel, so the control flow that affects affine instructions is replicated to both types of warps. A pair of corresponding statements in the affine and non-affine streams, namely “*if(tid¡bound) enq*” and “*if(tid¡bound) deq*,” is considered here as an example. The if-statement means that the affine warp should only enqueue and expand the tuple for non-affine threads that require the data (i.e. the non-affine warps with *tid* less than *bound*). In addition, the affine warp also requires control flow information of its own to run ahead of non-affine warps independently. Therefore, we equipped the affine warp with its own SIMT stack (the affine SIMT stack). The stack is a two-level design that exploits convergence at the warp level to reduce the need to check and update control flow on a thread-by-thread basis.



		Warp Level Bit Masks					
pc	rpc	w0	w1	w2	w3	......	w47
d	---	11	11	11	11	......	11
c	d	00	00	10	11	......	11
b	d	11	11	10	00	......	00

Figure 3.9: Re-Convergence Stack for the Affine Warp

Figure 3.9 depicts the affine SIMT stack. The functionality is similar to that of a baseline GPU and that of the non-affine SIMT stack. For the code example on the left, threads re-converge at *Basic Block D* (*BB_d*), and the affine warp is currently at *BB_b* ①. The *warp level stack* (WLS) encodes each non-affine warp’s bit vector with only 2 bits. The “11” term indicates that all threads in the warp are 1s; “00” denotes all 0s; and “10” denotes other settings. The *PC* and *re-convergence PC* (RPC) fields are shared by all warps. The “1” and “00” cases only require checking the WLS, without inspecting each thread’s bits. The WLS reduces the number of bits that are checked and updated. For the “10” case, *per warp stacks* (PWSs) are used for threads within a warp. On a Fermi GPU, 48 PWSs are used for concurrent warps on an SM. In the example, only warp 2 (*w2*) must update its PWS, and ④ and ⑤ show the content. All other warps use only WLS, and their PWSs have no data ③. The PWSs do not have PC and RPC fields, which reduces storage.

3.3.4 Divergent Affine Tuples

Control flow divergence affects whether an instruction is eligible for affine computation. In some cases, an eligible instruction may require more than one affine tuple due to divergence.

The code on the left of Figure 3.10 represents a case where a single tuple suffices even after control flow divergence. In this case, the affine warp will not enqueue *addr1* for inactive threads. All active threads’ *addr1*, however, are still computed by the same base and offset ①, so active threads still use the same affine tuple. In this case, it is sufficient to mask off the inactive threads when expanding the affine tuple; this is handled by the affine SIMT stack.

The code on the right represents a case where affine tuples become divergent. The common case is that threads compute addresses or predicates differently depending on whether boundary conditions are met. In the example, a thread’s

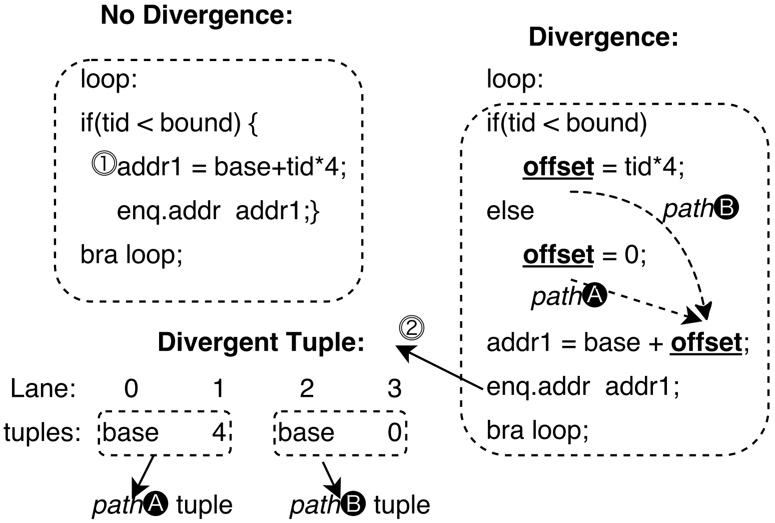


Figure 3.10: Divergent Base-Offset Pairs on SIMT Lanes

value for *offset* can be either *0* or *tid*4*, so addr1 has two affine tuples for all threads: *(base,4)* and *(base,0)*.

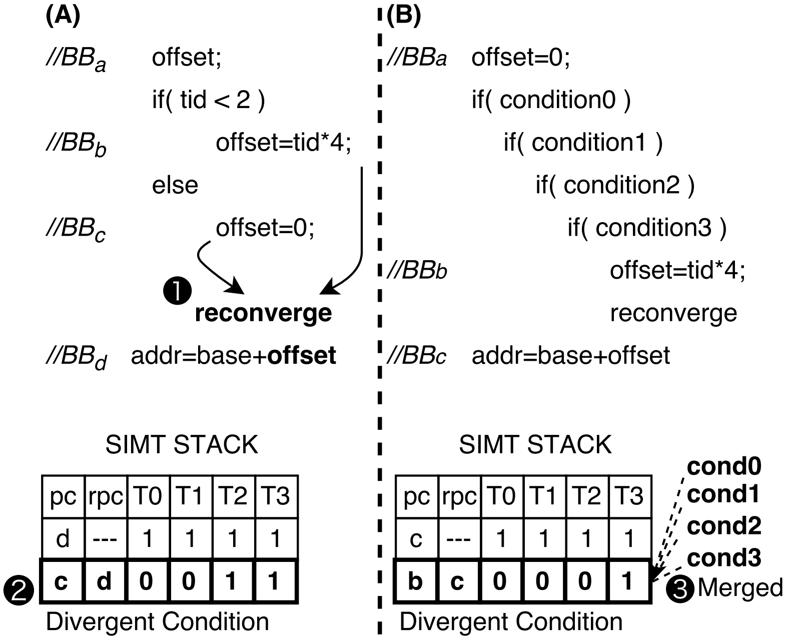
In general, at most two divergent conditions (or four tuples) can affect an affine operand; otherwise, the related affine instructions are not decoupled. A compile-time technique is used to detect divergent tuples. The main idea is to perform control-flow graph (CFG) analysis to identify and annotate which control flow conditions are responsible for creating divergent affine tuples. This information allows the compiler to decide whether certain affine instructions are eligible for decoupling. The annotate control flow instructions are used at run-time for the AEU to select whether to expand texttttTuple Ⓐ or `Tuple` Ⓑ.

The compiler technique operates as follows. Each operand is classified as one of three possible types: *scalar* (e.g. kernel parameters), *affine* (e.g. threadIdx), or *non-affine* (e.g. memory). These are listed in order from the most specific to the most general. The compiler initially assigns types to non-register operands. By creating a CFG and performing reaching definition analysis on the CFG, the initial types are iteratively propagated through register operands and instructions.

At each instruction, if more than one definition reaches a source operand, the most general type among the definitions is assigned to the source operand. The destination operand of an instruction is assigned the most general type among the source operands. Instructions with operations not supported by affine computation produce non-affine destination operands directly.

After the classification process, memory access and predicate computation instructions with scalar and affine type source operands are candidates for decoupling into the affine stream. We refer to instructions that define another instruction's source operands as *predecessors*. From each candidate memory and predicate instruction, the compiler recursively traverses the CFG backwards to checks its predecessor instructions for identifying divergent tuples. At each predecessor, the compiler recognizes divergent affine tuples when a source operand has two or more reaching definitions. For example, in Figure 3.11 (a), "`offset`" is defined at BB_b and BB_c before reaching BB_d; hence, "`addr`"has two affine tuples to expand. At run-time, for each thread, expansion units choose one of the affine tuples according to the thread's control flow. We call the conditions for making the choice *divergent affine conditions*.

Because the affine warp uses the affine SIMT stack to handle control flow, we used SIMT stack entries as divergent affine conditions. Figure 3.11(a) pro-



```
(A)
//BBa     offset;
          if( tid < 2 )
//BBb         offset=tid*4;
          else
//BBc         offset=0;
          reconverge
//BBd   addr=base+offset
```

```
(B)
//BBa  offset=0;
       if( condition0 )
          if( condition1 )
             if( condition2 )
                if( condition3 )
//BBb                  offset=tid*4;
                       reconverge
//BBc  addr=base+offset
```

SIMT STACK (A)

pc	rpc	T0	T1	T2	T3
d	---	1	1	1	1
c	**d**	**0**	**0**	**1**	**1**

SIMT STACK (B)

pc	rpc	T0	T1	T2	T3
c	---	1	1	1	1
b	**c**	**0**	**0**	**0**	**1**

Figure 3.11: Using SIMT Entry as Divergent Condition

vides the following example. The compiler identifies the re-convergence point of two reaching definitions' basic blocks ①. The last SIMT stack entry before re-convergence ② is the divergent affine condition, since it distinguishes threads that use BB$_b$'s definition from those that use BB$_b$'s.

Divergent reaching definitions can occur at an arbitrary predecessor of an affine instruction where expansions are required. Thus, DAC saves to a *dedicated divergent condition register file* (DCRF) the required SIMT stack entries (bit-vectors) at re-convergence points, so that they can be checked by expansion units later. As with the affine SIMT stack, the DCRF has a two-level structure but this is used as a register file rather than a stack. After detecting a divergent affine tuple, the re-convergence points are marked by the compiler, and a DCRF entry is allocated by the compiler.

3.4 Methodology

To evaluate performance, we used GPGPU-sim 3.2.2 [10], and to evaluate energy, we used GPUWattch[76]. The baseline GPU is modeled after a Fermi GTX 480 with the simulation parameters shown in Table 3.1. We use CACTI 5.3 [91] to model the energy overhead of DAC's added SRAM components.

Baseline GPU	
GPU	Fermi (GTX480), 15 SMs, 48 warps/SM
SM	32 SIMT lanes, 128KB register file
Scheduler	2 Schedulers/SM, Two Level Active [92]
L1	48 KB/SM, 4 Ways, 32 MSHRs
L2	769 KB, 6 Partitions, 8 Ways
GPU Prefetcher (MTA)	
Prefetch Buffer	16KB/SM (in addition to the 48KB L1)
Compact Affine Execution (CAE)	
Affine Units	2 Affine Units per SM (one per 16 lanes)
Decoupled Affine Computation (DAC)	
ATQ (per SM)	24 Entries, 392 bytes, 5.3 pJ/Access
PWAQ (per SM)	192 Entries, 1560 bytes, 3.4 pJ/Access
PWPQ (per SM)	192 Entries, 768 bytes, 1.5 pJ/Access
PWS (per SM)	8×48 Entries, 1536 bytes, 2.7 pJ/Access
PWS (per SM)	8×48 Entries, 1536 bytes, 2.7 pJ/Access

Table 3.1: Simulation Parameters

We simulate all benchmarks with SASS, which is the native instruction set executed directly on GPU hardware. GPGPU-sim parses the SASS assembly code produced by the CUDA tool-chain and generates PTXPLUS, which is the instruction set used by the simulator. PTXPLUS corresponds almost exactly to SASS; the conversion is merely syntactic.

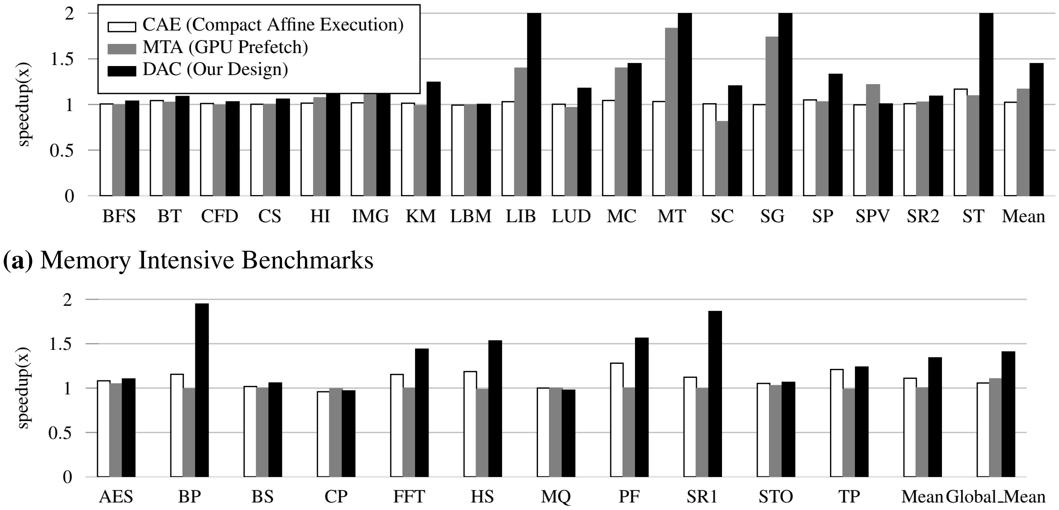
For DAC, the compiler's decoupling of kernels is performed on the PTX-PLUS instructions in the GPGPU-sim front-end before simulation starts.

Compute Intensive			**Memory Intensive (cont)**		
Name	Abbr.	Suite	Name	Abbr.	Suite
CP	CP	G	imghisto	IMG	G
STO	STO	G	histogram	HI	R
AES	AES	G	LBM	LBM	R
mri_q	MQ	G	SPMV	SPV	R
tpacf	TP	G	b+tree	BT	C
FFT	FFT	G	LUD	LUD	C
backprop	BP	C	sradv2	SR2	C
sradv1	SR1	C	stream cluster	SC	C
hotspot	HS	C	KMEANS	KM	C
pathfinder	PF	C	BFS	BFS	C
blackscholes	BS	P	CFD	CFD	C
Memory Intensive			monte carlo	MC	P
LIB	LIB	G	mersenne twister	MT	P
sgemm	SG	R	Scalar Product	SP	P
stencil	ST	R	Convolution Sep.	CS	P

Table 3.2: List of Benchmarks – G: GPGPU-sim distribution [10], R: Rodinia benchmark suite [22], C: CUDA SDK, P: Parboil benchmark suite [128]

3.4.1 Baseline Techniques

To evaluate both the computational and memory latency hiding aspects of DAC, we implement two other state-of-the-art designs. These are based on previously proposed techniques, which we now describe. In each case, we provisioned the techniques with extra hardware that we do not give to DAC.



(a) Memory Intensive Benchmarks

(b) Compute Intensive Benchmarks

Figure 3.12: Speedup of CAE, MTA, and DAC over the Baseline GTX 480 GPU

Compact Affine Execution (CAE) To evaluate DAC's efficiency in handling affine computations, we compare against CAE. The CAE augments the baseline GPU with an affine data path based on Kim et al's design [65]. CAE tracks affine operands at run-time to determine which warp instructions are eligible for affine computation. After fetch-decode, eligible warp instructions are sent not to the SIMT lanes but to the affine function units for execution. CAE improves efficiency by replacing vector computations with affine computations for threads within a warp.

GPU Prefetcher (MTA) To evaluate DAC's ability to hide memory latency, we compare it to a system that augments the baseline GPU with a data prefetcher based on Many-Thread Aware prefetching (MTA) [72]. MTA detects both intrawarp memory access offsets (e.g. load instructions in loops within a warp) and interthread offsets (e.g. load instructions issued by adjacent warps) for a few SIMT threads. The regularity is then speculatively generalized to all warps to achieve scalable prefetching. In addition, a throttling mechanism is used to control the aggressiveness of prefetching based on the number of evicted cache lines that are prefetched but not used by the GPU [72].

3.4.2 Benchmarks

We evaluate 29 benchmarks from four suites, as shown in Table 3.2. We divide them into two categories: memory-intensive and compute-intensive benchmarks. We consider a benchmark to be memory intensive if the baseline GPU can achieve a speedup of at least 1.5 when using a perfect memory system (i.e. no latency and unlimited bandwidth). The remaining benchmarks are considered to be compute-intensive.

3.5 Evaluation

Figure 3.12 shows the speedup of DAC, CAE, and MTA over the baseline GTX 480 GPU for our 29 benchmarks. DAC's geometric mean speedup of 1.40 is significantly better than either CAE's or MTA's. As expected, CAE provides benefits for just the compute-intensive benchmarks, whereas MTA provides benefits for just the memory-intensive benchmarks. Not only does DAC improve the performance of both classes of programs but it offers the best performance within each class of programs. For the compute-intensive benchmarks, DAC has a speedup of

1.34, while CAE achieves a speedup of 1.15. For the memory-intensive benchmarks, DAC produces a speedup of 1.44, whereas MTA achieves a speedup of 1.16.

3.5.1 Instruction Execution Reduction

Figure 3.13 shows that for the 29 benchmarks, DAC executes on average 0.74× as many warp instructions as the baseline GPU. Therefore, DAC reduces the dynamic instruction count by 26%, which in turn reduces execution time and improves energy efficiency. The effect is particularly evident for compute-intensive benchmarks. Only 4.6% of the instructions executed on DAC are affine instructions (see Figure 3.13), indicating that DAC does not require a dedicated affine functional unit.

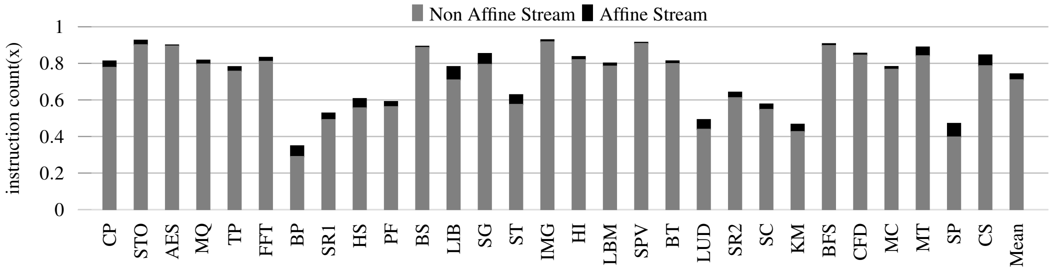


Figure 3.13: Number of Warp Instructions Executed by DAC Normalized to the Baseline GPU

With two affine units per SM, and two warp schedulers, our implementation of CAE doubles the affine instruction throughput compared to the baseline. By contrast, DAC executes a single affine instruction to replace nine instructions on the baseline GPU on average. Hence, it increases execution throughput for affine instructions by 9× over the baseline GPU.

3.5.2 Affine Instruction Coverage

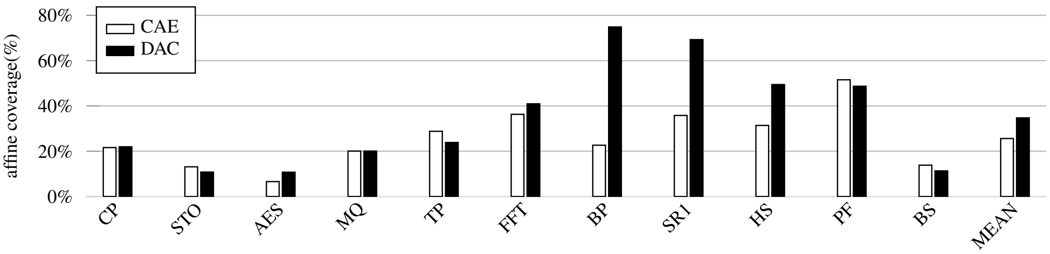


Figure 3.14: Affine Instruction Coverage of DAC and CAE

The coverage of affine instructions refers to the percentage of warp instructions executed by the baseline GPU that could be handled as affine instructions by CAE or DAC. For the 11 compute-intensive benchmarks, DAC achieves a geometric mean coverage of 34%, compared to 25% for CAE. These results are illustrated in Figure 3.14.

Because DAC identifies affine computations statically and uses SIMT lanes to execute affine instructions, DAC supports affine computations after limited control flow divergence; it uses offloaded affine SIMT stack entries to reduce the overhead. By contrast, CAE has no facilities for performing affine computations after divergence. Although the CAE scheme for identifying affine instructions is more flexible than that of DAC, CAE must use the SIMT lanes to expand any affine tuples involved in divergence back to vector values [65]. Moreover, CAE's affine functional unit uses a single ALU for offset computations, which requires all 32 threads of a warp to have the same offset pattern. For benchmarks, such as HT and BP, whose last-level dimension is smaller than 32, CAE can handle only scalar computations (i.e. an offset of 0), since the threads in a warp do not follow a single offset pattern.

3.5.3 Memory Latency Hiding

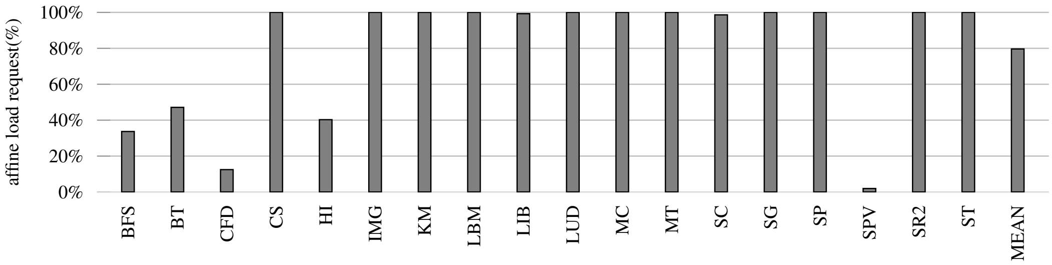


Figure 3.15: Percentage of Affine Global and Local Load Requests on DAC

DAC can hide memory latency because the affine warp can run ahead of non-affine warps and issue load requests, without waiting for previous requests to finish. An indicator for DAC's latency-hiding ability is the percentage of global and local load addresses that are produced by affine instructions, which can be issued by the affine warp. Figure 3.15 shows that for our memory-intensive benchmarks, an average of 79.8% of the global and local load requests are issued by the affine warp. Many benchmarks have close to 100% coverage, because regular SIMT workloads often use scalar data and thread IDs to map memory addresses for coalesced memory accesses.

For benchmarks such as BFS and BT, which make heavy use of indirect memory addresses to access complex data structures, DAC offers little performance improvement. In addition, benchmarks may be constrained by bandwidth, row buffer locality, or bank conflicts; in such cases, the affine warp might not run ahead sufficiently. Therefore, some benchmarks (e.g. LBM) show little performance improvement despite the high percentage of affine memory requests.

MTA and DAC use different mechanisms to hide memory latency. In DAC, affine memory requests are non-speculative and are generated by instruction executions of the affine warp. By contrast, MTA hides latency by speculatively issuing prefetch requests when triggered by on-demand memory accesses.

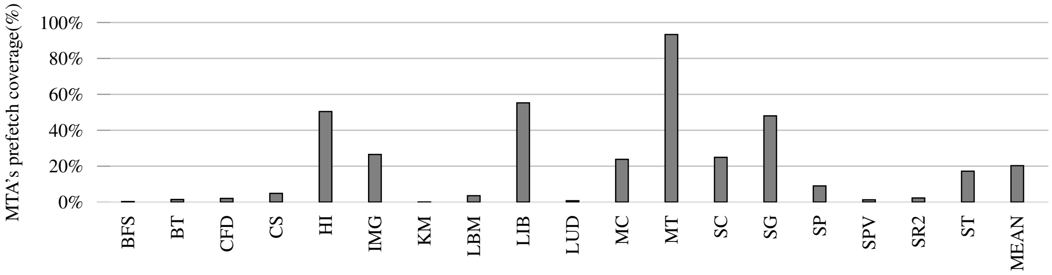


Figure 3.16: MTA Prefetcher Coverage

Many SIMT workloads have highly regular memory accesses, so the MTA prefetcher has high prediction accuracy. MTA's latency-hiding ability correlates with prefetcher coverage (see Figure 3.16), which is defined as the number of L2 and DRAM accesses covered by the prefetcher. It is evident that MTA's throttling mechanism reduces harmful prefetches, but it also reduces coverage when additional bandwidth is available in some cases. In certain other cases (e.g. SC), the throttling mechanism does not prevent cache pollution

3.5.4 Energy Efficiency

Figure 3.17 shows the total energy consumption (dynamic and static) of DAC normalized to the baseline GPU. For our 29 benchmarks, the geometric mean is 0.798. Thus, DAC reduces total energy by 20.2%, and it reduces dynamic energy

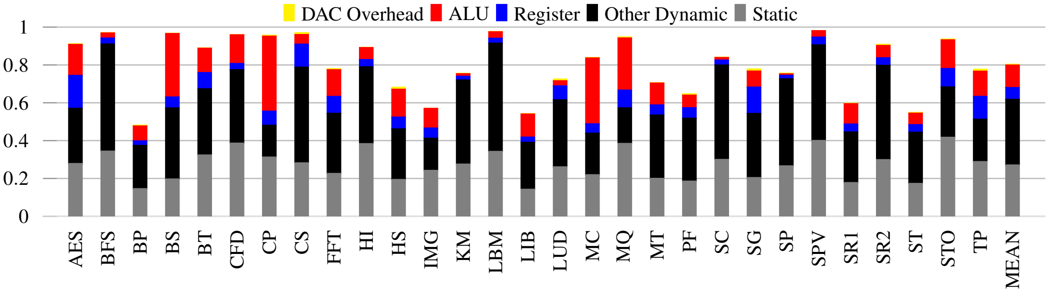


Figure 3.17: Energy Consumption of DAC Normalized to the Baseline GPU

alone by 18.4%. The major sources of savings are reduced ALU operations and reduced register accesses due to reduced dynamic instruction executions. DAC reduces the number of ALU operations by 44% and the energy consumption of ALUs by 34%. DAC also reduces the number of register accesses by 17% and the energy consumption of the register file by 32%. By reducing execution time, DAC reduces static energy consumption by 29%.

The overhead of DAC is only 0.96% of the dynamic energy consumption. Most of the overhead comes from the expansion of affine tuples. The expansion units are efficient since they typically use only one or two ALU operations to expand an affine tuple for a given warp.

3.5.5 Area Estimation

Most of the DAC hardware budget is allocated to expansion units, which add 2 ALUs per SM, and to the various SRAM components, which add 6 KB per SM. The ATQ has 24 entries, with a total size of 393 bytes. The per warp address queue (PWPQ) has 192 entries partitioned among warps, with a total size of 1560 bytes. Similarly, the PWPQ has 192 entries and a total size of 768 bytes.

The affine SIMT stack has a depth of 8. It has, first, a WLS with bit-masks, PC, and RPC, which requires 224 bytes; and second, PWSs with bit-masks only, which requires 1536 bytes. The DCRF has the same amount of storage as the affine SIMT stack.

We model the SRAM components using CACTI [91], which yields 0.21 mm^2 of estimated area per SM. We estimate the area of 2 ALUs with the model used in GPUWattch [76], which yields 0.16 mm^2 per SM. On a GTX 480, with a die size of 520 mm^2 [55], the area overhead is 1.06%.

3.6 Summary

In this chapter, we have demonstrated how two distinct ideas, namely affine computations and DAE, can be synergistically combined. Doing so greatly improves the performance and energy efficiency of SIMT GPUs.

First, specialized support for affine computations on SIMT GPUs has until now preserved the model in which a single instruction stream executes on all warps, which limits the redundancy reduction to within a single warp. By decoupling the affine computations to a separate affine instruction stream, DAC overcomes this limitation. A single affine warp can thus produce values for many non-affine warps and it reduces the warp instruction count.

Second, a naive implementation of DAE on GPUs would imply a doubling of the number of threads. However, because affine computations represent such a large reduction in computation, DAC focuses on affine memory accesses and adds one warp per SM to significantly hide memory latency.

The result is a system that improves performance and energy efficiency for both memory-intensive and compute-intensive workloads. The total energy consumption is reduced by 20.2%, and a speedup of 40.7% is achieved.

Chapter 4

Decoupled Fine-Grained Synchronization[1]

The GPU architecture is highly efficient for regular workloads, and displays significant performance advantage over traditional CPUs in that regard. Because of this success, people are interested in using GPUs for irregular workloads as well. The obstacle here is that irregular workloads tend to have operations not suitable for massive data parallelism. Among these, fine-grained lock-based synchronization is one of the most difficult problems to deal with.

To overcome the obstacle, we present a software solution that decouples lock-based operations so that specialized treatments can be used to bypass the limitations of GPU architecture. In particular, our solution can transform a global synchronization problem into a local synchronization problem, which can be then performed in fast scratch memories.

This chapter is organized as follows. Section 4.1 motives our solution by discussing the performance bottleneck caused by fine-grained locks. Sections 4.2, 4.2.1, 4.3 discuss various components of our solution. In Section 4.5 and 4.6, we present a detailed experimental evaluation of our solution.

[1] Portions of this chapter are based on the following publication:
Fast Fine-Grained Global Synchronization on GPUs, ASPLOS 2019[133]
My contributions: I'm the first author for the paper

4.1 Motivation

Global fine-grained synchronization enforces mutual exclusion when threads from multiple TBs update shared data in global memory. As Figure 4.1 (a) shows, the baseline protects the shared data with fine-grained locks implemented in global memory. When there are contentions on protected data, the critical-section execution of contending threads are serialized, and the threads that fail to acquire lock must retry lock variables continuously until success is attained. This process wastes memory bandwidth. While lock contentions can be a problem in general, they place a particularly heavy performance penalty on GPUs.

The performance bottleneck can be understood from both the throughput and latency perspectives. Regarding throughput, GPUs typically run tens of thousands of concurrent threads, which can lead to a massive number of lock retries. Unlike CPUs, GPUs do not have cache coherence that could allow lock retries to be confined to local caches (e.g. L1 caches). Hence, on GPUs, lock retries must directly access the global levels of memory hierarchy, which are L2 and DRAM. In addition, since fine-grained synchronization is used for irregular algorithms, lock retries are non-coalesced memory accesses. This situation further exacerbates the bandwidth waste, which in turn slows down all global memory traffic. Regarding latency, depending on the algorithm and the data-input used, it is possible for many threads to serialize their critical-section execution on a few protected data. Due to such serialization, the latency of lock operations can significantly affect performance. Unfortunately, the global memory system is optimized for throughput and has high latency; the latency of lock operations is further increased due to interference from lock retries.

4.2 Our Solution

Our solution utilizes fast scratchpad memories to handle lock operations and thus shields global memory from lock contentions. Scratchpad memories are local memories private to each thread block (TB). They are used in conjunction with message passing in global memory to achieve global synchronization that involves multiple TBs.

Specifically, our solution specializes only the critical-section part of the kernel with a message-passing model. The rest of the kernel (i.e. non-critical section) is left unchanged. This is done by separating an original kernel into two kernels: the *client kernel*, which handles the non-critical sections, and the *server kernel*, which executes the critical section on behalf of the client kernel. The two kernels run concurrently, as shown in Figure 4.1 (b).

Client threads can still update arbitrary protected data items. However, they do so indirectly, by offloading critical-section executions to server TBs via our software message-passing system. On the server side, the ownership of protected data is partitioned among server TBs so that each data item is accessible exclusively through a unique server TB. The partitioning is achieved by interleaving data items to server TBs, in a fine-grained manner, to avoid load imbalance. Client threads choose the appropriate server TBs as destinations of messages based on data IDs. Because data IDs have a one-to-one mapping to server TBs, all update requests for the same data are guaranteed to be sent to the same server TB, thus maintaining

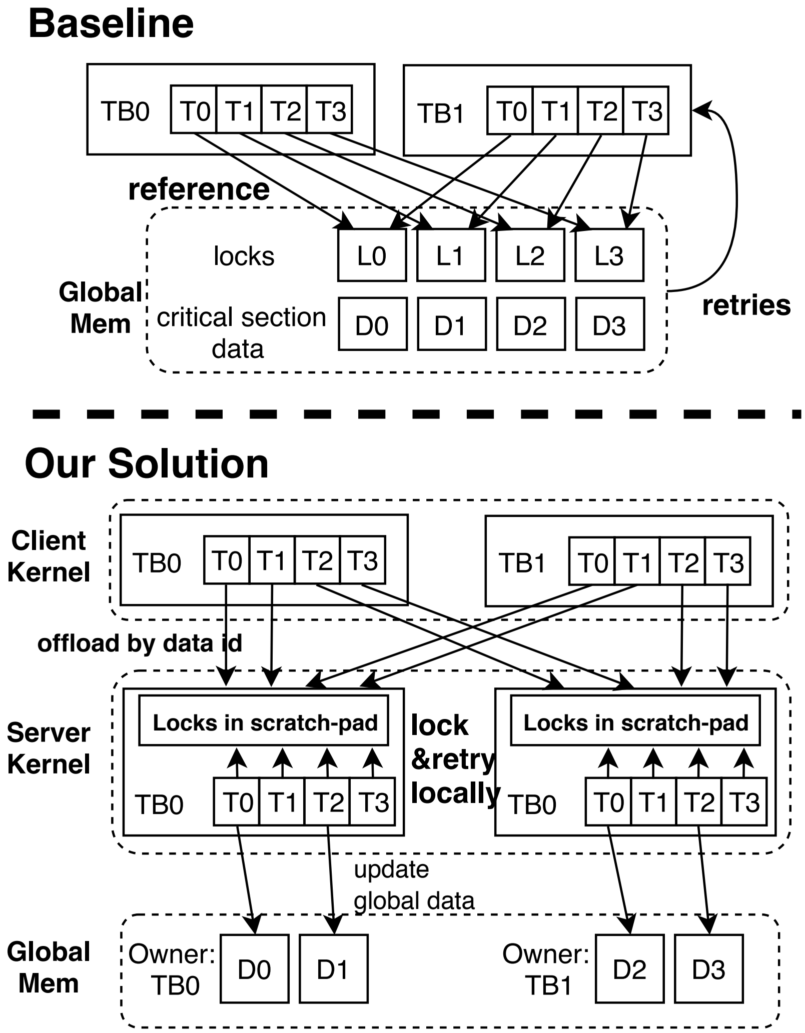


Figure 4.1: Fine-grained mutual exclusion with **(a)** global locks (baseline) and **(b)** our solution

.

mutual exclusion at the TB level. This allows threads within each server TB to use scratchpad memory locks to maintain mutual exclusion when servicing clients' requests in parallel. Scratchpad memories are high-bandwidth, low-latency on-chip SRAM that support word-granularity accesses, where accesses waste no bandwidth due to unused cache-line data. Thus, scratchpad memories are ideal for lock accesses and retries with irregular memory accesses.

Our solution improves performance by making better use of the GPU memory system. Compared to the baseline, which repeatedly issues lock retries to global memory when contentions occur, our solution does not waste global memory bandwidth. Client threads send offload messages via global memory only once, regardless of contentions. The localizing of lock operations in scratchpad memories means that useful global memory operations are no longer interfered with by lock retries; in addition, the latency of lock operations is reduced.

4.2.1 Decoupled Program

This section describes how the original program with global locks (Listing 4.1) would be transformed to a decoupled program (Listing 4.2) at the source-code level.

Listing 4.1 shows how a critical section is encapsulated into a function (line 6) and is protected by a try-lock loop. *Data_id*, which can be a single word or a data structure, refers to the data item to be updated. The *arg#* are additional arguments that are generated by computations in the non-critical sections and are then passed to the critical section.

```
void kernel(...){
// begin critical section
bool success = false;
do{
if(try_lock(data_id)){
critical_sec(data_id,arg0,arg1);
__threadfence();
unlock(data_id);
success = true;
}
}while(!success);
// end critical section
}
```

Listing 4.1: Original Kernel with Global Locks

Listing 4.2 shows how our software architecture uses two new procedures, *send_msg* and *recv_msg* (lines 2–3), to pass messages from a client TB to a server TB. The *dst* term in*send_msg* denotes the server TB. The message size, in words, corresponds to the number of arguments of the critical-section function.

The *client_kernel* (lines 5–16) corresponds to the original baseline kernel in Listing 4.1, where the critical section in the try-lock loop has been replaced with message sending to server TBs. The code at line 8 maps offloads work to server TBs based on data IDs. The mapping interleaves the ownership of data items to server TBs. This fine-grained partitioning provides better load balance among server TBs than would a coarse-grained partitioning. However, data IDs are dynamically generated by client TBs, depending on data inputs. Hence, load imbalance can still occur when many threads serialize on relatively few data items. Even in these scenarios, the original kernel with global locks suffers much more due to high latency global memory and the interference caused by lock retries.

```
 1 //procedure calls for message passing
 2 void send_msg(int dst,int data_id,any arg0,...);
 3      bool recv_msg(int& data_id,any& arg0,...);
 4
 5 void client_kernel(...){
 6      // execute non-critical section
 7      ...
 8
 9      //map data to server
10      int server_id = data_id % num_server_TB;
11
12      //offload critical section execution
13      send_msg(server_id,data_id,arg0,arg1);
14      // execute non-critical section
15      ...
16 }
17
18 void server_kernel(...){
19      //scratchpad memory locks
20      __shared__ int locks[4096];
21
22      //loop to handle client requests
23      bool terminate = false;
24      while(!terminate){
25           int data_id,arg0,arg1;
26           if(recv_msg(data_id,arg0,arg1)){
27                //received msg, do critical section
28                bool success = false;
29                do{
30                     if(try_lock_local(data_id)){
31                          critical_sec(data_id,arg0,arg1);
32 __                       threadfence_block();
33                          unlock_local(data_id);
34                          success = true;}
35                }while(!success);}
36           terminate=check_termination();}}
```

Listing 4.2: Pseudocode Code For Our Solution

The *server_kernel* (lines 18–40) executes the critical section on behalf of the clients. Hence, any try-lock loop in the original kernel is now in the server kernel (lines 29–36), which uses locks implemented in scratchpad memories rather than global memory. Since scratchpad memories have limited size, there can be a limited number of locks; multiple data IDs can be mapped (aliased) to the same lock. On modern Nvidia GPUs, for example, the maximum TB is 1K threads. We used 4K locks per server TB to reduce the chance of unnecessary serializations caused by aliasing.

Server threads execute a loop (lines 19–34) that listens to clients' messages and terminates when all clients are finished. The termination condition is a flag, set by clients, in global memory. It is only checked periodically by servers; the overhead of checking for termination is negligible because only one thread per TB checks the flag and then informs the other threads of the condition.

Our solution is mostly straightforward for programmers. For most cases, our code in Listing 4.2 can be used as a template; the programmer must insert code for both non-critical sections and critical sections at the indicated places. The server architecture for nested locks (discussed in Section 4.4 is more complex, but a template is also provided for that case.

The maximum occupancy of the GPU which refers to the number of TBs that can be executed concurrently can be determined by API calls. Some of those TBs are used by the server kernel, and the remaining TBs are used by the client kernel. The ratio of server-to-client TBs is based on the relative amount of work performed in the critical sections versus the non-critical sections. The programmer is responsible for setting this parameter based on the characteristics of the specific application. This aspect may require some tuning by the programmer.

4.3 Our Software Message Passing System

Message passing is achieved using a set of message buffers in global memory, shared between senders and receivers. For receivers (i.e. servers), there is a single buffer for each TB, which is used by all threads of that TB for receiving messages and is not accessed by threads in other receiver TBs. For senders (i.e. clients), threads choose the appropriate message buffer to write, based on data IDs. Each message buffer is a large array accessed as a circular buffer; we use 4K message entries as a default.

To implement message passing efficiently on GPUs, the main design consideration is scalability. Each message buffer can be concurrently read and written by thousands or more threads. Our main idea for addressing scalability is to take advantage of scratchpad memories and to manage buffers hierarchically.

4.3.1 Our Basic Algorithm

We first describe the basic algorithm for a single thread's read and write operations. Then, we describe our optimized implementation for scalability.

The message buffer has the metadata shown in Figure 4.2 and described here. The *write_idx* is atomically incremented by the sender to reserve a buffer index for writes. To determine whether the reserved index is free to write, the sender checks the *read_idx*, which is atomically incremented by the receiver after reads. The*bit-mask*k has one bit corresponding to each buffer location; it is set by the sender after the message data has been successfully written to L2 (i.e. after the memory barrier). It is checked by the receiver to find available messages for reads.

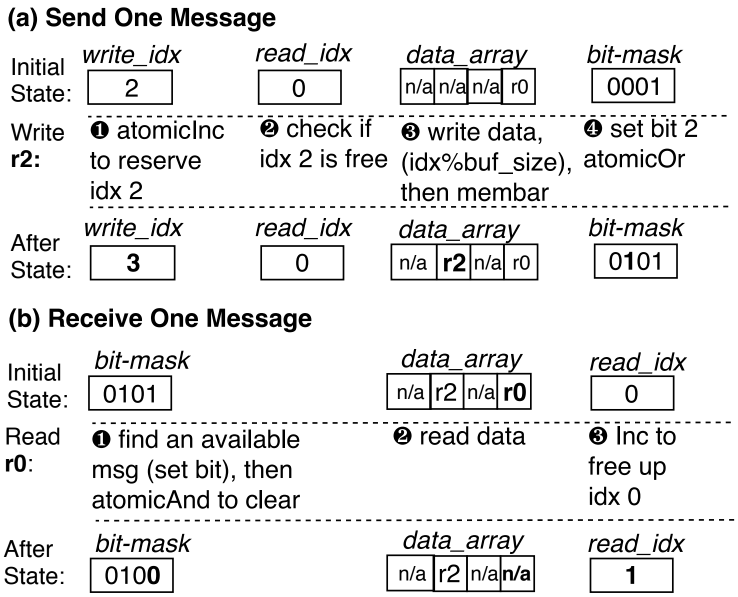


Figure 4.2: The basic data structure of a single message buffer and the basic algorithm for reading and writing

.

The bit-mask is needed because concurrent sender threads may finish writing out of order, such as in the figure, where buffer location 1 is reserved before location 2, but location 2 is written before location 1. Hence, the receiver must be able to determine whether a specific location is valid.

In our basic algorithm, each thread accesses message buffers individually, so each message send or receive incurs the overhead of accessing metadata in global memory (e.g. bit-mask) Moreover, the lanes of a sender warp may have different destination buffers (receiver TBs), and the lanes of the receiver warp might not read consecutive buffer locations. Therefore, memory accesses for message data may be non-coalesced.

4.3.2 Our Optimized Algorithm

Our optimized solution improves efficiency by amortizing the cost of global memory accesses over a number of messages and by promoting coalesced memory accesses.

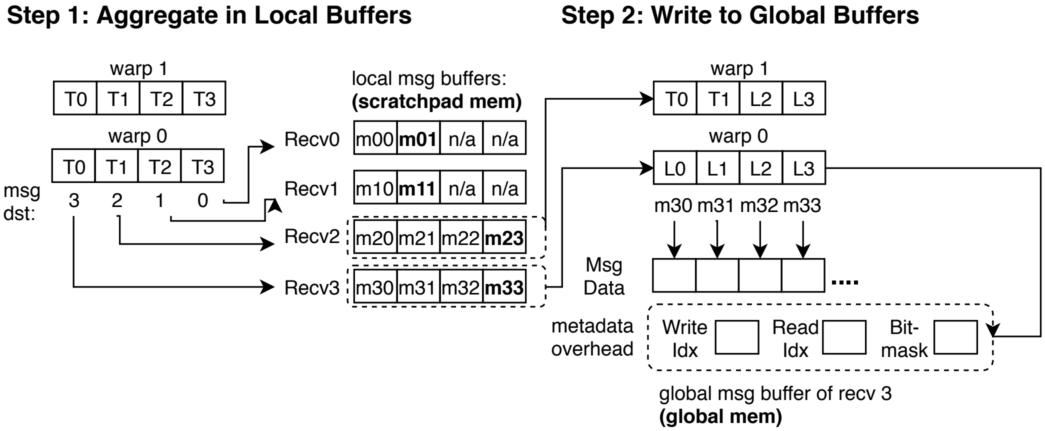


Figure 4.3: Sender Design—using local buffers for aggregated message write

.

Senders aggregate messages by collecting them in local buffers residing in scratchpad memory before they are written to global message buffers in bulk. Figure 4.3 illustrates the sender design. Each TB has a set of small message buffers in the scratchpad memory, with each local buffer corresponding to one receiver TB. Messages from multiple warps are aggregated in local message buffers before being written to global message buffers; hence, metadata overhead is amortized, and global memory accesses are typically coalesced. In addition, the metadata overhead for accessing *read_idx* (Figure 4.2 a) can be further reduced by keeping a local scratchpad copy and updating it lazily (not shown).

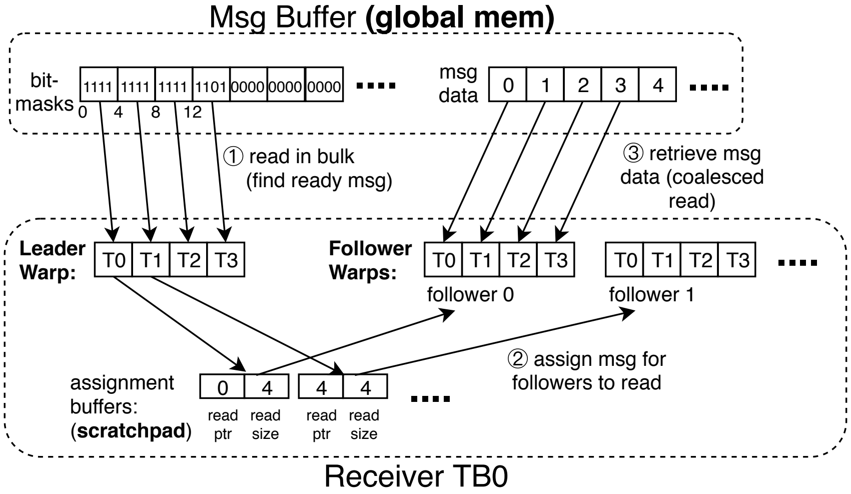


Figure 4.4: Receiver Design—using a single warp (the leader warp) for metadata accesses

.

Receivers aggregate message-passing metadata access by using a single warp, the *leader warp*, for each TB. This warp handles the metadata on behalf of the other warps of that TB, which we refer to as the *follower warps*. Figure 4.4 illustrates the receiver design. The leader warp discovers a number of ready messages by reading multiple bit-mask words in global memory at once. These messages are then assigned to follower warps using a set of *assignment buffers* in the scratchpad memory. The leader warp only reads the bit-mask, whereas the actual message data is read by follower warps. The messages assigned to each follower warp are stored in consecutive buffer locations in global memory, which means message data retrievals are coalesced memory accesses.

The leader warp can aggregate bit-mask reads of up to 1024 messages with single warp granularity global memory, because 32 lanes of the warp can read 32 bit-mask words, which each represent 32 messages. This feature greatly reduces the metadata overhead. In additional, other metadata overheads, such as resetting bit-masks, are performed in a similar aggregated manner (not shown).

In addition to the advantage of no lock tries granted by the overall software architecture, our message-passing system has additional bandwidth benefits compared to global memory lock operations. The main insight is that global memory lock operations must directly access specific lock variables that are spread throughout the address space. Therefore, memory accesses are inherently non-coalesced. By contrast, our solution handles locking indirectly and locally in the server TBs' scratchpad memories. Hence, a client's send messages are not bound to specific global memory addresses; therefore, these messages can be placed consecutively in circular buffers. This feature allows our optimized solution to perform coalesced reads and writes of global memory.

4.4 Handling Nested Locks

Listing 4.3 shows the original kernel code with two nested locks. The critical section manipulates two data items, so a thread must acquire the locks for both data items before entering the critical section.

```
// begin critical section

//data_id1 < data_id2
bool success1 = false;
bool success2 = false;
do{
     if(!success1){
          if(try_lock(data_id1))
               success1 = true; //acquired 1st lock
     }

     if(success1){ // acquire 2nd lock
          if(try_lock(data_id2)){
               critical_sec(data_id1,data_id2,...);
               __threadfence();
               unlock(data_id1);
               unlock(data_id2);
               success2=true;
          }
     }
}while(!(success1 && success2));
// end critical section
```

Listing 4.3: Original Kernel With Two Nested Lock

Just as with non-nested locks, our solution partitions data items among server TBs so that lock operations can be handled in scratchpad memories. As shown in Figure 4.5, server TB0 has ownership of data D0 and D1; hence, TB0 has exclusive access to the associated locks (L0 and L1) in the scratchpad memory. TB1 similarly has exclusive access to L2 and L3. The client's offload request now contains two data items, D0 and D2, belonging to two different server TBs.

Our solution lets a client send an offload message to the server TB that owns the first lock (TB0). TB0then acquires the remote lock (L2) from the other server TB (TB1). It does so by sending to TB1 a *request message* for L2, which tries to lock L2 locally. Once successful, TB1 sends a reply message back to TB0, temporarily granting ownership of L2 to TB0 and preventing other requests from

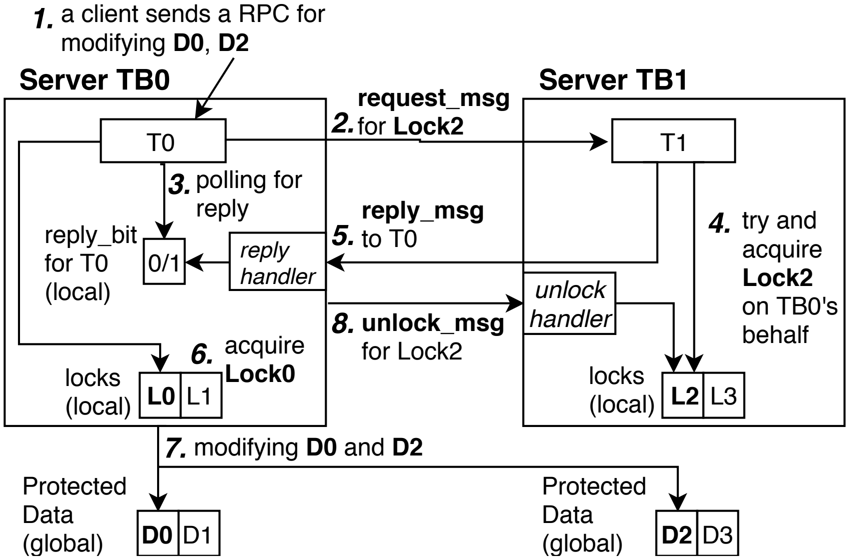


Figure 4.5: Synchronization Server for Two Nested Lock—operations for handling an offloaded request from client that involves two server TBs

modifying D2. Upon receiving the *reply message*, TB0 acquires L0 locally and executes the critical section. Once finished, TB0 sends an *unlock message* back to TB1 to unlock L2. As with non-nested locks, our solution handles lock retries in scratchpad memories so that server TBs send messages only once across the global memory.

To avoid deadlocks, we use a lock hierarchy to prevent circular dependences. The lock hierarchy is defined by the *global ID* of the locks to which they are mapped. Global ID uniquely identifies a local lock, where $global\_ID = server\_ID \times locks\_per\_server + local\_ID$. Furthermore, we avoided deadlock caused by insufficient buffer space availability by using different message channels for different message types. This approach is similar to the idea of using virtual channels to prevent protocol-level deadlocks.

The reply and unlock messages occur on the critical path of lock acquisition and release. Therefore, to reduce their latency, we replaced the receiver's leader warp (see Section 4.3.2) with a *reply handler warp* and an *unlock handler warp*. These two warps handle metadata in the same manner as the leader warp; however, instead of assigning messages to follower warps, they read message data directly and then perform their associated action directly. Their actions set reply bits or reset local locks. This feature reduces latency. The modification is possible because reply and unlock are simple tasks that are guaranteed to succeed without retry, so follower warps are not needed.

At the sender side, we did not aggregate reply and unlock messages in local buffers (see Section 4.3.2), because local buffering increases latency. Instead, the two types of messages are sent directly to global buffers

4.5 Methodology

Compute Capability	sm_61	Scratch-Pad Per SM	96KB
Shader Clock Rate	1.68 GHz	Max Scratchpad Per TB	48KB
SM Count	28	L2 Size	2.75MB
Max Threads Per SM	2048	L2 Cache Line Size	128 Byte

Table 4.1: GTX 1080 ti Specifications

We evaluated our solution on an Nvidia GTX 1080 ti GPU (Pascal, GP102) [97, 102] using CUDA toolkit version 9.2 with driver 396.37. The hardware specification is shown in Table 4.1. To gather kernel execution statistics, such as L2 and DRAM traffic, we used the Nvidia Profiler (nvprof) [99] that is provided with CUDA 9.2. The profiler replays kernel executions and periodically accesses hardware performance counters on the GPU to record statistics.

To evaluate our solution, we used five benchmarks: two microbenchmarks and three state-of-art implementations of relatively complex algorithms. Similar microbenchmarks are used by previous researchers who studied fine-grain locking[146, 40, 39] and transactional memories [45, 24, 44, 113, 142] on GPUs. We now describe each of our five baseline benchmark programs.

Hash Table (HT) HT is a microbenchmark. Threads insert elements into a hash table, with each hash-table entry being a linked list. Locks are used to provide mutual exclusion on entry updates. We used a large hash table and collision factors of 256, 1K, 32K, and 128K. The collision factor represents a pool of distinct entry references for threads to choose from randomly. Thus, small collision factors lead to many lock conflicts.

Bank Account (ATM) ATM is a microbenchmark with two nested locks. Each thread performs a transaction that withdraws funds from one account and deposits them into a second account. A lock is associated with each account, so each thread acquires two locks to perform a transaction. Similar to HT, threads randomly choose the source and destination accounts with collision factors 256, 1K, 32K, and 128K.

Minimum Spanning Tree (MST) MST finds a spanning tree that connects all vertices of a graph with minimum weight. Our baseline is a GPU implementation of Boruvka's algorithm from the newly released LonestarGPU 3.0 benchmark suite [69, 106]. Each thread works on a vertex of the graph and updates a data structure, called a “component.” Because multiple vertices may be mapped to the same component, fine-grain locks are used to provide mutual exclusion for component updates. We used as inputs the three largest graphs from the benchmark suites, namely rmat22 (power law), USA-road-d.USA (high-diameter), and r4-2e23 (random).

Stochastic Gradient Descent (SGD) SGD works on bipartite graphs, such as a movie rating graph with some vertices representing movies and other vertices representing users. Weighted edges between a user and a movie represent a rating. SGD predicts missing edges (ratings) based on existing edges. We use Kaleem et al’s [62] edge-lock implementation, where edges are assigned to threads and two nested locks are used to guard movies and users. We used three real-world inputs, namely Netflix (NF) [2, 13], Reuters (RT) [3, 77], and movie-lens 10M (ML) [1, 49].

Maxflow (MF) MF is a push-relabel algorithm that finds the maximum flow of a weighted graph, where edge weights represent network capacity. Nodes are parallelized to threads, and fine-grained locks are used to prevent the same node being worked by two or more threads. Our inputs are mesh graphs (2k×2k, 4k×4k, and 8k×4k) generated by a Washington generator[61].

4.6 Evaluation

This section describes our evaluation of the proposed solution. First, we present speedups over the current state-of-the-art; thereafter, we examine in detail the causes of the performance gap.

4.6.1 Performance

Figure 4.6 shows the speedup of our solution over state-of-the-art baseline implementations of each of our benchmarks. Our solution achieves a mean speedup of 3.6×. The reasons for the speedups we obtain are (1) reduction in global memory bandwidth consumption and (2) reduced lock operation latency.

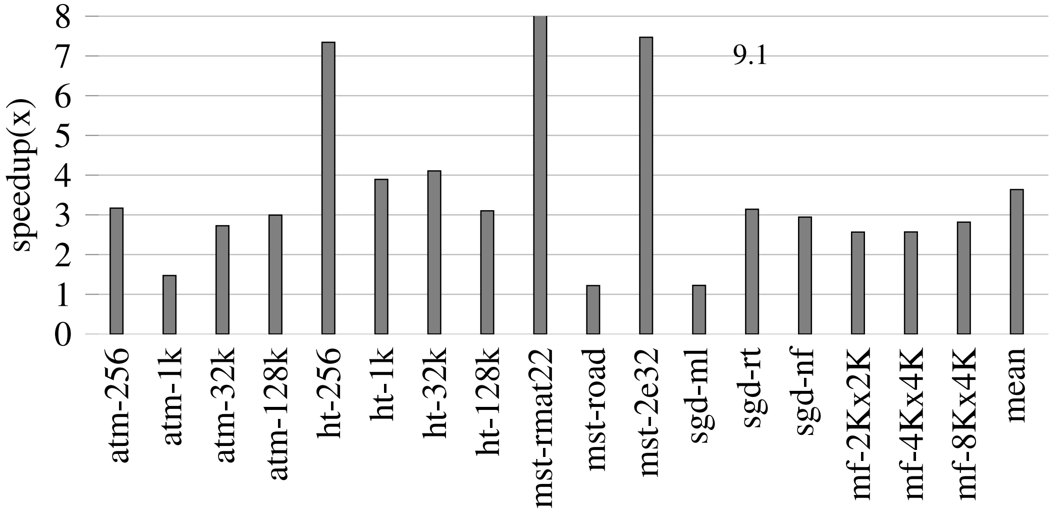


Figure 4.6: Speedup of our solution over the state-of-the-art

4.6.2 Bandwidth Benefits

Figure 4.7 shows the L2 and the DRAM traffic of our solution as a percentage of the baseline. The figure shows that our solution significantly reduces the amount of global memory traffic and thus alleviates the bandwidth bottleneck.

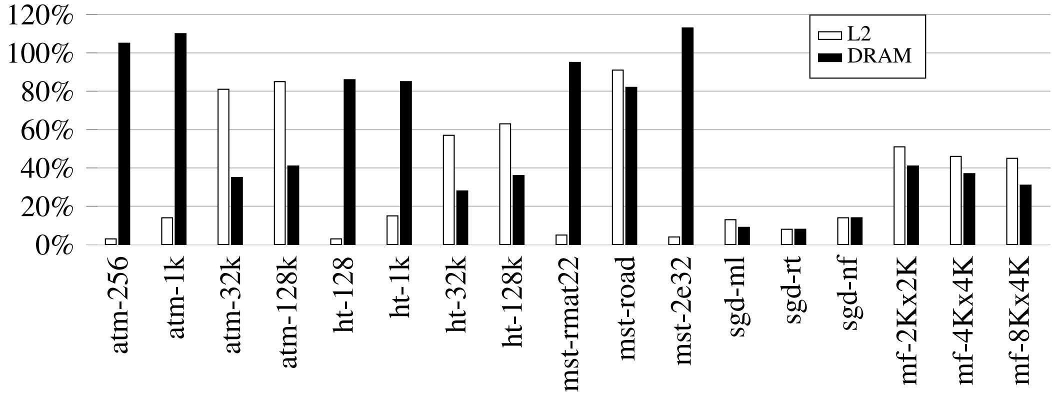


Figure 4.7: L2 and DRAM traffic of our solution as a percentage (%) of the baseline—The L2 traffic is the total cache-line accesses of global loads and stores and atomics, including misses to DRAM. The DRAM traffic includes both reads and writes.The traffic includes overhead due to non-coalesced accesses (i.e. unused words in cache lines)

Compared with the baseline, our solution improves efficiency by shielding global memory from lock retries, which are instead performed in scratchpad memories. In addition, our message-passing system contains optimizations that promote coalesced global memory accesses. By contrast, the baseline issued non-coalesced lock accesses directly to global memory.

Our scheme generally reduces both DRAM and L2 traffic, in most cases. However, our DRAM traffic reduction—compared with the baseline is greater than the L2 traffic reduction, because our message-passing system enjoys locality in the

L2 cache. Each receiver buffer has just one *write_idx* and *read_idx*, so access to those pointers causes L2 traffic but most likely results in a cache hit. Furthermore, for message data writes, when multiple senders make non-coalesced writes to the same buffer (same receiver) at similar times, they are likely to write to adjacent locations of the buffer, since the (circular) buffers are reserved incrementally for writing. Individually, each sender causes non-coalesced L2 accesses, but the cache lines of the buffer are evicted to DRAM and are read by the receiver with coalesced messages.

4.6.3 Latency Benefits

For benchmark-input combinations with high lock contention, Table 4.2 shows that execution time strongly correlates with latency. The table highlights the average latency between unlock and reacquisition, including measurement overhead. Latency is measured using the %globaltimer register, which is a nanosecond hardware timer that has a consistent time for all SMs.

For HT and MST, our solution handles locks and unlocks entirely through server TBs that access scratchpad memory. By contrast, the baseline model requires higher latency global memory, which is affected by memory contention caused by lock retries. Therefore, our solutions have significantly lower latencies. For our ATM solution, global memory is used to send messages to acquire and release locks. However, the critical path operations in global memory are not inhibited by lock retries, and certain lock transfers are handled in scratchpad memories. These characteristics mean our solution has lower latency than the baseline.

	Latency (Cycle)	
	Baseline	Our Solution
atm_256	24269	8292
atm_1k	18234	13831
ht_256	2192	248
ht_1k	2679	863
mst_rmat	3601	490
mst_r4	3537	533
	Latency (Norm.)	**Run Time (Norm.)**
	Our Solution	Our Solution
atm_256	0.34	0.31
atm_1k	0.75	0.68
ht_256	0.11	0.13
ht_1k	0.32	0.25
mst_rmat	0.14	0.11
mst_r4	0.15	0.13

Table 4.2: Latency and Total Execution Time

4.6.4 Comparison Against Hardware Solutions

We compare our solution with two previously proposed hardware solutions for improving the performance of global memory lock operations.*HQL* [146]] embeds hardware locks in the L1 and L2 caches, where cache tag entries act as queue locks. A cache-coherence-like protocol for lock operations between L1 and L2 is used.*BOWS* [40] is a warp scheduler that reduces retry traffic by deprioritizing warps that are spinning on locks.

Because HQL and BOWS are hardware solutions evaluated on simulators, a direct comparison is impractical. Table 4.3 provides a rough comparison between our solution and previous solutions, for common benchmarks; the speedup of our solution is shown together with published results for HQL and BOWS. Because of the numerous methodological and implementation differences, these numbers should be interpreted cautiously.

	Speedup over Baselines				
	HT-32	HT-128	HT-512	HT-1K	ATM-1K
HQL	10x	1.6x	1.1x	0.9x	
BOWS				1.3x	1.8x
Ours	18.3x	8.9x	4.0x	3.9x	1.5x

Table 4.3: Speedup over respective baselines—For HQL, the results are from Figure 12 of the paper [146]; the baseline is a simulated Radeon HD 5870 GPU. For BOWS, the results are from Figure 15 of the paper [40]; the baseline is a simulated GTX 1080ti. The HQL paper only provides results for the HT microbenchmark, and the BOWS paper only provides results for HT-1K and ATM-1K; unavailable results are left blank in the table.

At low lock count, HQL achieves speedup for HT because lock transfers are partially handled in the L1, which decreases latency compared to the baseline. The effect is similar to the use of scratchpad memories in our solution. However, hardware locks are bound to limited cache resources, namely, the cache capacity and the number of tags. Hence, the performance benefit of HQL decreases rapidly as the number of locks increases; at 1K, HQL degrades performance. Since our solution is implemented in software, it does not have these limitations. Our solution thus achieves much higher speedups and does not experience performance degradation at high lock counts.

BOWS improves performance by reducing lock retries. However, global synchronization is still handled in L2 and DRAM, which limits the performance gain, particularly for HT, compared to our solution. Our solution instead implements locks in scratchpad memories.

4.7 Summary

A common research trend is to add hardware support to render GPUs more efficient and effective for irregular computations. In this chapter, we have shown that in one respect, GPUs are already more efficient than is commonly recognized. With the right programming model, existing GPU hardware can support efficient fine-grained synchronization.

The main idea is to greatly reduce the use of slow global memory by distributing work to the faster local scratchpad memories. In particular, our solution uses global memory to distribute work to server TBs, each associated with a single scratchpad memory. Lock retries are then handled at the scratchpad memories, which are more efficient than global memory, particularly for non-coalesced memory accesses. To support this solution, we implemented an efficient software message-passing system built on top of global memory.

This new software architecture is straightforward for programmers. The main task is to decompose the critical sections from the rest of the code. For example, instead of writing a single kernel with a critical section, programmers implement two kernels, one representing client threads that execute the non-critical sections and make non-blocking procedure calls to the servers, and the other representing server threads that execute the critical sections on behalf of the clients.

We evaluated our solution on five irregular benchmarks, each with three different inputs. On Nvidia GTX 1080 ti GPUs, our solutions are on average 3.6× faster than the previous best state-of-the-art solutions for each problem.

Chapter 5

A GPU SSSP Solution with Decoupled Worklist[1]

Graph algorithms have abundant parallelism that can be exploited by the GPU hardware. Furthermore, there are optimizations [87, 105] to make them more regular in terms of control flow and memory accesses on GPUs.

Many graph algorithms benefit from using a worklist for work scheduling. The worklist is a globally shared data structure, so existing GPU algorithms use relatively simple worklist designs that suit the GPU architecture. However, such simple worklist designs sacrifice the quality of work scheduling, which limits the performance.

Our goal is to use a more complex worklist that performs high-quality work scheduling while also making the complex worklist efficient for GPU architecture. To achieve this, we decouple the original kernel into the *worker kernel* and the *worklist manager kernel*. The worker kernel is responsible for graph processing, whereas various tasks related to the worklist are delegated to the manager kernel. We launch only one TB for the manager kernel (*MTB*); all other hardware resources (TBs) are devoted to the worker kernel. The rationale is to centralize the worklist

[1] Portions of this chapter are based on the following publication:
A Fast Work-Efficient SSSP Algorithm for GPUs, to be appear in PPOPP 2021
My contributions: I'm the first author for the paper

management to a single MTB instead of letting all threads handle it directly. By doing this, we are able to implement a complex worklist while avoiding extensive synchronization between the many threads on the GPU.

Based on this approach, we develop a new *single source shortest path* (*SSSP*) for GPUs. We choose SSSP because it has been extensive studied on GPUs and work scheduling has a great impact on performance. To choose a baseline for comparison, we studied seven previous GPU SSSP algorithms (e.g. nvGRAPH [101], Gunrock [135, 138], etc.) on a set of 226 graphs, and the best performing algorithm is from **L**onestar**G**PU 4.0 [15, 105] (*LG-SSSP*), which is an adoption of Δ -stepping [88], a CPU SSSP algorithm. We will now briefly discuss and compare Δ -stepping, LG-SSSP , and our solution in turns.

The main idea of Δ -stepping is to use a worklist that supports approximate priority scheduling, which consists of many *buckets* (i.e. unordered lists, similar to the buckets used in radix-sort). The characteristics of the bucket data structure are as follows. First, each bucket can contain arbitrary number of vertex IDs, up to a *MAX SIZE*, and buckets grow and shrink dynamically during execution. It is evident that implementing buckets as arrays of MAX SIZE would waste too much memory. Therefore, the CPU algorithm implements buckets as linked lists to handle the sparsity. Second, the bucket data structure supports *multi-writer-multi-reader* (MWMR), which means that threads can simultaneously read and write the same buckets. This means that readers do not block writers or vice versa.

For GPUs, LG-SSSP **simplifies** the bucket-based worklist design. First, it uses arrays instead of link-lists for buckets. However, it is limited to using only two buckets instead of many to avoid wasting too much memory; the disadvantage of this apporach is reduced precision of priority scheduling and thus also **work efficiency**. Second, MWMR requires synchronization between writers and readers for many threads on the GPU, which is unscalable. Therefore, LG-SSSP does not support MWMR. Instead, it uses *double buffering* and global barrier, which allows readers and writers to update in different arrays and thus avoids reader-writer synchronization.

However, the disadvantage of not supporting MWMR is reduced **concurrency**. Work items (i.e. vertex IDs) written to the worklist cannot be read out and scheduled immediately but must be delayed until the global barrier. This scenario occurs because readers and writes do not access the same arrays.

Our solution is based on Δ -stepping, but we designed a worklist **efficient for GPUs without simplification**, where the decoupled and centralized worklist manager plays a key role. First, to support many buckets, we use **dynamic arrays** as buckets and implement a custom memory management scheme. All buckets share a common pool of pre-allocated memory, such as cudaMalloc. This memory is then managed by the worklist manager in run-time. To do this, each bucket (array) is used as a circular FIFO, so the manager monitors the read and write pointer of each FIFO bucket and then allocates (or de-allocates) memory to (or from) buckets accordingly. Second, to support MWMR, the worklist manager acts as a single delegate reader for graph-processing threads; this essentially transforms the problem into *multi-writer-single-reader*, which reduces synchronization requirements. The manager finds available vertex IDs in the worklist in bulk and then assigns them to graph-processing threads. To avoid the single manager becoming the bottleneck, the manager only accesses and updates the worklist's metadata rather than the actual content (i.e. vertex IDs). Third, our solution has a mechanism for setting the Δ in run-time. The Δ is the parameter that controls the granularity of buckets, which greatly affects the performance. The optimal Δ value differs for different graphs and GPU configurations. For both the CPU Δ -stepping algorithm and LG-SSSP, the Δ is statically chosen before execution with a simple heuristic, which often yields a non-optimal Δ . Our solution uses run-time information to choose a more optimal Δ dynamically. To do this, the worklist manager gathers statistics (e.g. hardware utilization and work efficiency) and feed them to a state machine to adjust the Δ value. To summarize, with many buckets, MWMR, and the run-time Δ mechanism, our solution significantly advances the state-of-art of GPU SSSP in terms of performance. Comparing to LG-SSSP (the best known solution), we achieve an average speedup of 2.8× on 226 graphs on a RTX 2080 ti GPU.

5.1 Background

This section discusses the basics of SSSP algorithm to provide necessary background.

SSSP works on weighted and directed graphs, as Figure 5.1shows. *Vertices* are connected together by *edges*, and the connection is unidirectional. In a trivial example, each vertex has only one or two neighboring vertices. However, in general graphs, each vertex can have any number of neighbors, and the distribution of the vertices' neighbor counts can be highly non-uniform.

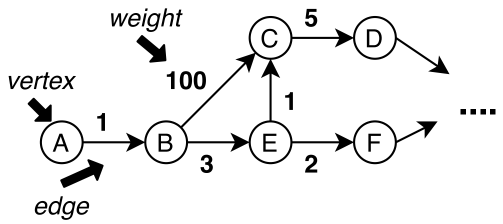
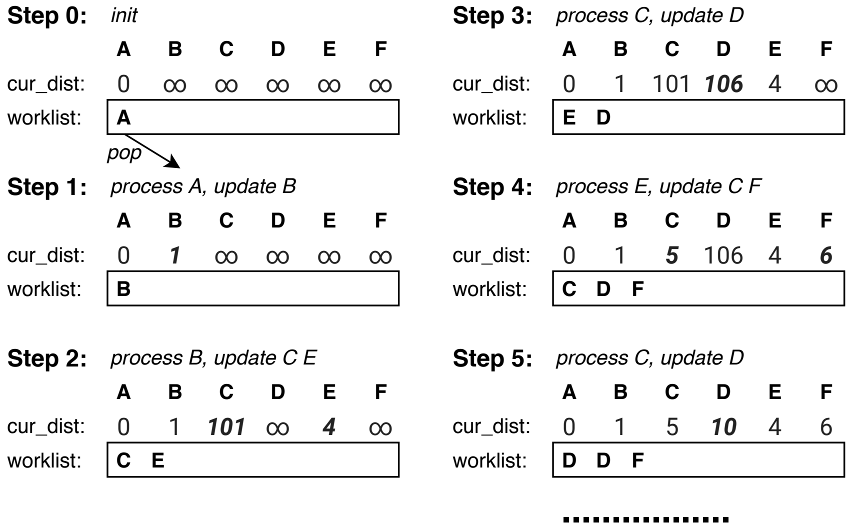


Figure 5.1: An Example Graph—edges are directed with weight

.

Each edge has a weight, which represents *distance*. The goal of SSSP is to find the shortest paths between a single source vertex (e.g. A) and every other vertex. For this calculation, each vertex is associated with a variable, textitcur_dist, which represents the currently best-known distance of that vertex. Initially, *cur_dist* of the source vertex is set to 0, whereas that of all other vertices is set to ∞, meaning an unknown distance. During execution, *cur_dist* is gradually refined until the shortest path is found.



Step 0: *init*

	A	B	C	D	E	F
cur_dist:	0	∞	∞	∞	∞	∞
worklist:	**A**					

pop

Step 1: *process A, update B*

	A	B	C	D	E	F
cur_dist:	0	***1***	∞	∞	∞	∞
worklist:	**B**					

Step 2: *process B, update C E*

	A	B	C	D	E	F
cur_dist:	0	1	***101***	∞	***4***	∞
worklist:	**C E**					

Step 3: *process C, update D*

	A	B	C	D	E	F
cur_dist:	0	1	101	***106***	4	∞
worklist:	**E D**					

Step 4: *process E, update C F*

	A	B	C	D	E	F
cur_dist:	0	1	***5***	106	4	***6***
worklist:	**C D F**					

Step 5: *process C, update D*

	A	B	C	D	E	F
cur_dist:	0	1	5	***10***	4	6
worklist:	**D D F**					

....

Figure 5.2: Illustrate SSSP step-by-step

.

Figure5.2 shows how a generic SSSP algorithm works. Each step processes one vertex by updating its neighbors' *cur_dist*. For example, in step 1, we select A and updateB, and B's new *cur_dist* is calculated as A's *cur_dist* (0) plus the edge weight (1). An updated vertex must be processed in subsequent steps so that the shortest path calculation can be propagated to its neighbors, and so on. A worklist is used to store the IDs of outstanding vertices to be processed. For example, in step 2, we push B's neighbors C and D to the worklist, and they are then popped from the worklist to processed i step 3 and 4.

5.1.1 Work Scheduling

In Figure 5.2, vertex C is processed twice, in step 3 and 5. The *cur_dist* of C in step 3 is derived from path ABC, which is not the shortest path, so it must be corrected in step 5 with the shortest path, ABEC. Alternatively, if we select E instead of C to process in step 3, and C in step 4, C will be processed only once, since its *cur_dist* is on the shortest path in the first try. This shows that the order in which vertices are selected in each step affects the total work performed.

To generalize, if the worklist is a *list* (as in Figure5.2), vertices are popped in arbitrary order; a vertex's *cur_dist* may not be on the shortest path when being processed and thus may be processed multiple times during execution. Alternatively, we can use a priority queue as the worklist, with *cur_dist* as the key (not shown), and the vertex with the smallest *cur_dist* is popped first. Vertices are guaranteed to be on the shortest path when being processed, so each vertex is processed only once during execution.

The choice of a list or a priority queue as the worklist corresponds to two well-known SSSP algorithms, namely those of *Bellman-Ford's* [11] and *Dijkstra's* [38]. Dijkstra's is the most work-efficient SSSP algorithm in that it performs the least amount of work. The difference in work efficiency between Bellman-Ford and Dijkstra can be up to the order of 10000×, especially for high-diameter graphs.

The disadvantage of Dijkstra's algorithm is that a priority queue has more overheads than a list in terms of data-structure complexity. More importantly, Dijkstra's algorithm is difficult to parallelize due to its requirement of processing vertices in strict order and the use of the priority queue. Generally, Dijkstra's algorithm is considered a serial algorithm. By contrast, for Bellman-Ford's, vertices in the list can be processed in an arbitrary order, so parallelism is simply achieved by assigning those vertices to threads to process. The available concurrency correlates the number of outstanding vertices in the worklist at a given time.

To summarize, the use of a worklist is a form of work scheduling. In general, there are three criteria to assess a work-scheduling method: **work efficiency**, **concurrency** (e.g. serial vs parallel), and **data-structure complexity** (e.g. list vs priority queue). Bellman-Ford's and Dijkstra's algorithms occupy the two ends of the spectrum for each criterion.

5.1.2 Δ -Stepping

eyer et al. [88] propose Δ -Stepping as a midway between Bellman-Ford's and Dijkstra's algorithms. The main idea is to replace the priority queue in Dijkstra's algorithm with an approximately ordering data structure that allows parallelism and thus improves concurrency, whereas the ordered work scheduling improves work efficiency compared to Bellman-Ford's approach.

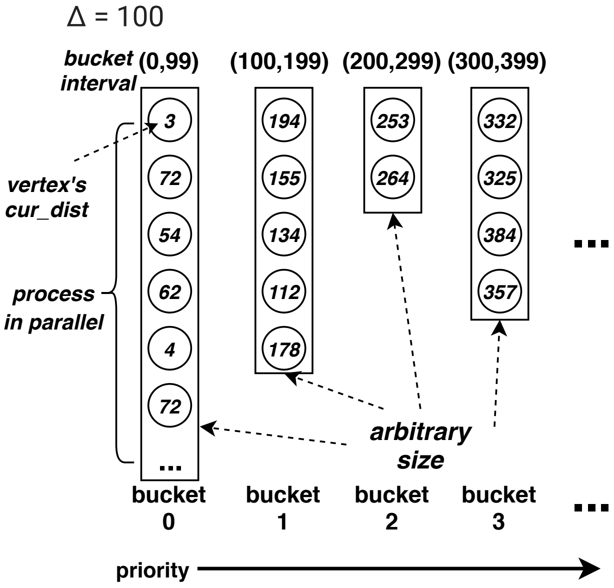


Figure 5.3: Δ -Stepping's Work Scheduling Data Structure

.

The approximately ordered data structure consists of many buckets (i.e. lists), as Figure 5.3 shows. Each bucket accepts vertices with *cur_dist* belonging to a certain range; the parameter Δ controls the interval. Essentially, the many-bucket data structure sorts the outstanding vertices in coarse granularity. Regarding work scheduling, vertices in the head bucket (e.g. bucket 0) are popped in arbitrary order and processed in parallel. When there is no vertex leaf, the next bucket (e.g. 1) becomes the head, and the process continues; this approach enforces approximate orders for vertices between buckets.

5.2 Motivation

We have described three SSSP algorithms: Dijkstra, Bellman-Ford, and Δ -stepping. The key difference between them is the work-scheduling method, which has a profound impact on performance for any hardware platform. In this work, we focus on developing a sound work-scheduling method for GPUs.

For GPU SSSP algorithms, work scheduling is a challenging problem. GPUs are massive parallel processors with tens of thousands of hardware threads; hence, the data structure used for work scheduling must be scalable to prevent a performance bottleneck. Furthermore, it should ideally provide enough concurrency to utilize the abundant hardware threads while also being work efficient.

To select a baseline for our study, we evaluated seven previous GPU SSSP algorithms on a set of 226 graphs (see Section 5.8 for details). The best-performing one is from LonestarGPU 4.0 [15, 105] (LG-SSSP), which is an adoption of a GPU adaptation of Δ -stepping. Recall the three criteria of workscheduling, namely work efficiency, concurrency, and data-structure complexity. The focus of LG-SSSP is to use a relativity simple data structure to achieve scalability on GPUs, but it sacrifices work efficiency and concurrency by doing so.

We now discuss the three design considerations when implementing Δ -stepping on GPUs. In addition, the limitations of LG-SSSP are described.

5.2.1 Design Consideration 1

In general, the total number of work items (i.e. outstanding vertex IDs) in all buckets is bounded by —*E*—, which refers to the number of edges. However, they are distributed to buckets non-uniformly, as shown in Figure 5.3. Furthermore, buckets grow and shrink dynamically during execution. If each bucket is implemented as a fix-sized array, all arrays must be large enough to expect the maximum usage case (—*E*—). When using many buckets, this wastes too much memory and thus becomes impractical for large graphs. Therefore, on CPUs, buckets are implemented as doubly linked lists to handle the dynamic sparsity[88].

The linked list is a poor choice. LG-SSSP compromises by implementing each bucket as a fix-sized array and by using just two buckets instead of many, to avoid wasting memory. This makes the work-scheduling data structure suitable for GPU architecture. The disadvantage is that work items are now grouped into only two buckets, which decreases the precision of ordering and thus decreases the work efficiency.

5.2.2 Design Consideration 2

Buckets are global data structures that are simultaneously accessed by all threads. Synchronization is needed when threads read (pop) or write (push) the buckets to avoid race conditions and to update the buckets consistently. There are three types of synchronization. First, when multiple threads write to the same bucket, writers must synchronize to ensure unique buffer locations are used. Second, multiple reader threads must synchronize to ensure unique locations are accessed. Third, readers must synchronize with writers to ensure that only properly written are read; for example, reading may happen before a location is properly written to the last-level cache.

This type of concurrent data structure is called the multiple-writer–multiple-reader (MWMR), which means that threads can read and write simultaneously. However, GPU algorithms, including LG-SSSP, typically do not use an MWMR concurrent data structure, since it requires too much synchronization and is thus unscalable for the large thread count of the GPU. Instead, concurrent data structures are implemented using a technique called double buffering to reduce synchronization, as shown in Figure 5.4.

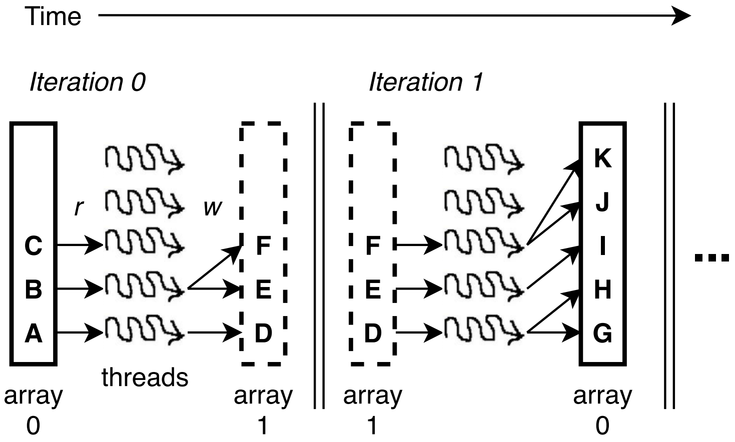


Figure 5.4: Implementing a List as Double Buffers

.

To simplify the discussion, we describe the use of a single list as the worklist. The key idea is to split a single list into two buffers: a reading buffer and a writing buffer. Then the graph is processed in iterations, called “super-steps.” For example, in iteration 0, threads read work items from *buffer 0* to process, and newly generated work items are written to buffer 1. After no work items are left in *array 0* and all threads are complete, the two arrays swap, so that *buffer 1* becomes the new reading buffer and *buffer 0* becomes the new writing buffer in iteration 1. The procedure repeats in following iterations. Iterations and buffer swaps are separated by a global barrier [140], which is a software-implemented barrier for all TBs in the grid.

With double buffering, reading actions are separated from writing actions because readers and writers access two different buffers. This removes the need for writer-reader synchronization. Only synchronization between writers is needed, instead of across all three types. Therefore, double buffering greatly reduces synchronization and simplifies the design, compared to MWMR. However, the disadvantage of double buffering is reduced concurrency, since newly generated work items can only be read in the next iteration. For example, in iteration 0, the newly written work items (D,E,F) are delayed for processing in iteration 1, even if idle threads are available in iteration 0. This scenario results in hardware under-utilization. By contrast, MWMR uses a single buffer, so that newly written work items can be read immediately without waiting; this approach achieves far superior concurrency.

In practice, double buffering is especially harmful for high-diameter graphs, where the execution is forced into many tiny iterations. For example, for the road.USA graph, the average work count per iteration is only 800, whereas an RTX 2080 GPU has 68K hardware threads.

5.2.3 Design Consideration 3

The third design consideration is how to set the parameter Δ , which strongly impacts performance. Recall that Δ affects the granularity of buckets. Choosing a smaller Δ results in fewer vertices in each bucket and thus reduces the amount of work that can performed in parallel. However, fine-grained buckets potentially reduce the total work due to precise ordering. Therefore, the choice of Δ controls the trade-off between available concurrency and work efficiency. The optimal choice – that is, the choice that yields the best performance – differs considerably across graphs and hardware. It depends on the weights and connectivity of the graph and the number of processing elements in the hardware.

LG-SSSP uses a simple heuristic [31] that often selects a far-from-optimal Δ . The Δ is chosen statically before execution, based on the average weight (W) and the average degree (D) of the graph input with the equation $\Delta = C*(W/D)$. *C*. The C term is a constant that remains the same for all graph inputs.

The equation cannot find a near-optimal Δ . We demonstrated this with the experiment shown in Figure 5.5. The idea is to make C a variable and then reversely find an optimal C for each graph. Figure 5.5 illustrates two points: first,the choice of Δ has a significant impact on performance; second, the optimal C for the two graphs are far away from each other. Hence, it is impossible to select a constant C suitable for all graphs.

In other words, static information (W and D) is not enough and we need more information and more sophisticated method to make the decision. Ideally,

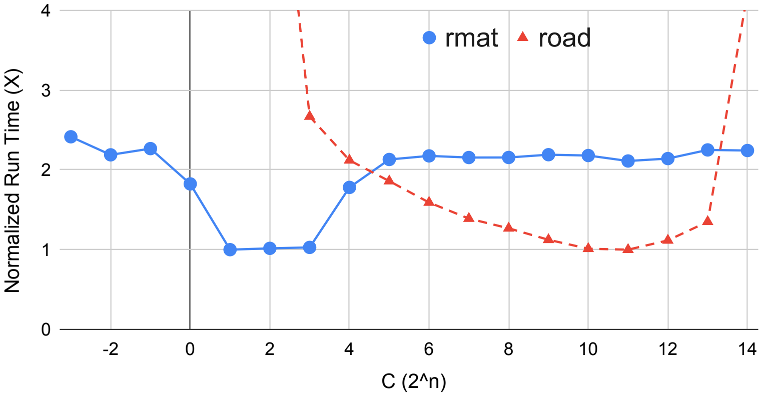


Figure 5.5: Execution Time against *C* for Two Graphs—the execution time is normalize to the minimum in the series; labels of the x-axis are power of 2

.

run-time information (e.g. hardware utilization and work efficiency) can be used to make a precise decision that yields a more optimal Δ .

In summary, the simplifications to the data-structure design improve the scalability, but at the expense of work efficiency and concurrency.

5.3 The Overview of Our Solution

Our solution is a GPU adoption of Δ -stepping. Compared to prior GPU adoptions, we use a new work-scheduling mechanism, which substantially improves the work efficiency and concurrency and therefore achieves a far superior performance. This is achieved by three key features.

First, we develop a dynamic data structure with customized memory management, which allows us to use many buckets instead of just two, to improve the work efficiency. Second, we do not use double buffering; instead, our bucket data structure is capable of MWMR, which improves concurrency. Third, we use a runtime mechanism that gathers statistics to dynamically derive a far more optimal Δ . While our work-scheduling mechanism has more sophisticated functionalities, the complexity of implementation is also substantially increased. Therefore, we need to find a way to deal with the complexity to implement the work scheduling efficiently.

Our main idea is to centralize work scheduling to a single thread block (manager TB, or *MTB*). All other thread blocks (worker TBs, or *WTBs*) are tasked with vertex processing. The rationale is to avoid letting all threads on the GPU handle the complex work-scheduling data structure directly; we isolate the synchronization and complexity to a single TB. The MTB and WTBs run in parallel while executing two different kernels of their respective tasks. Figure 5.6 shows the interactions between MTB and WTBs.

Worker threads process vertices and write the vertices' ID (*work items*) to buckets according to priority, but they cannot read from buckets directly. Instead, the MTB reads the bucket and assigns work to worker threads for processing. The MTB also maintains the priority of buckets by keeping track of and switching the head bucket. Essentially, the MTB is a delegate that performs read operations on behalf of worker threads; this transforms the MWMR problem into a MWSR problem (multi-writer–single-reader) because the MTB acts as a single reader.

To avoid the single reader becoming a bottleneck, the MTB only accesses the bucket *metadata* instead of the actual content (i.e. work items). For example, one such metadata indicates which bucket location is properly written to memory and valid for reading. This is a key requirement for a concurrent reader-writer data structure. The MTB probes the metadata to find valid work in the bucket, where one cache-line granularity memory access of metadata can resolve up to 8K bucket locations. Then MTB assigns the work to WTBs at a coarse granularity of up to 1K vertex IDs per assignment. To enhance the concurrency between metadata probing and work-assigning operations, we divide the two tasks between two separate warps in the MTB: the *probe-warp* and the *assign-warp*. Compared to a solution where all threads read and write buckets directly in MRMW fashion, our solution amortizes the overhead of metadata access to many reads instead of paying for each thread's (or warp's) read operation. Furthermore, our solution, being MRSW, does not need to resolve conflicts between multiple readers, such as colliding on the same bucket location. This characteristic simplifies the design.

The assign-warp also handles dynamic Δ stetting (see Section 5.7). Our Δ stetting mechanism relies on run-time statistics, the most important of which is how busy are the worker threads (i.e. hardware utilization). The assign-warp assigns work and thus can derive hardware utilization, which makes it the natural candidate for setting Δ .

Another task of MTB is memory management, which is handled by the *mm-warp*. From the perspective of worker threads, each bucket is a circular FIFO queue of size. However, there is no pre-allocated memory – such as cudaMalloc() – to back up the buckets; instead, all buckets share one pool of pre-allocated memory. This pool is dynamically allocated to and de-allocated from each bucket, on

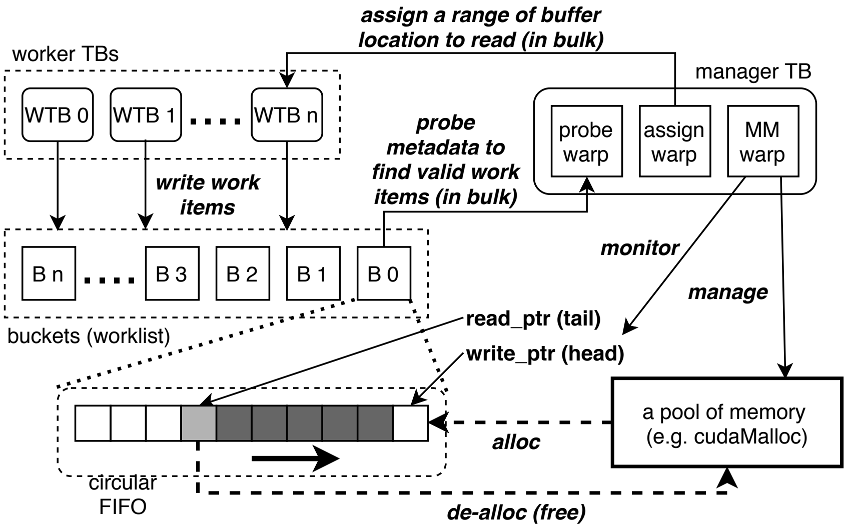


Figure 5.6: The Overview of Components' Functionalities

.

demand, by the mm-warp. The pool of memory is chopped into 256KB *blocks*, and the mm-warp uses bit-masks, stored in the fast scratchpad memory, to keep track of free blocks efficiently. Our memory-managing routine is customized for FIFO-like buckets, where the head and tail always grow linearly, and each written vertex ID is read out only once. There is no equivalent of malloc() and free() in our memory-manage routine because worker threads do not explicitly issue commands to the mm-warp. Instead, the mm-warp allocates and de-allocates blocks by monitoring the head and the tail of each FIFO bucket transparent to worker threads. By doing this, it avoids the overhead of communicating between work threads and the mm-warp over global memory. In addition, it allows the mm-warp to pre-allocate blocks to buckets in advance, because of FIFO's trivial access pattern, so that work threads do not stall on memory allocation when writing the buckets.

This rest of this chapter is organized as follows. We first discuss how a single bucket works and then expand the discussion to multiple buckets. To simplify the discussion, we first assume that bucket data buffers are static-allocated memory and without reuse. Then we discuss how the system would work with dynamic memory management.

5.4 A Single Bucket

A single bucket is an array used as a circular FIFO buffer. Our algorithm for preventing concurrent read/write race conditions is as follows.

Begin with *worker threads* (writers), multiple worker threads can write the bucket simultaneously, so each thread must find an unique buffer location to write. Each bucket has a reservation pointer (*resv_ptr*) (a global memory word). Threads atomically increment (atomicAdd) the *resv_ptr* before writes, and the return value (i.e. old value) is guaranteed to be unique for each calling threads. Next, we need to prevent the race condition between the reader and writers, in particular, the reader should only access buffer locations that have been fully written in memory.

For this purpose, each bucket has a number of write finished counters (*WFCs*) (global memory words). We call each N consecutive buffer locations a *segment* (N=256), each segment is associated with one WFC, initialized to 0. Here is how it works. Suppose a worker thread writes a buffer location in *segment A*, after the write, the thread execute a memory fence (__threadfence() in CUDA), which ensures that the data is properly written to the global level of memory hierarchy (e.g. the L2 cache on GPUs). After the fence, the worker thread atomically increment *segment*

A's *WFC*. Now suppose the reader read from *segment A*, it first checks the *WFC*. When *WFC*==N, it means all locations of the segment are fully written, so any location is valid for reading. Otherwise, when *WFC*¡N, the reader must also check the *resv_ptr*; we call the reserved location count inside the segment *resv_count* (i.e. *resv_ptr* minus the starting location the segment). There are following cases. Case 1, *WFC*¡*resv_count*, the reader cannot read the segment because writers may reserve and finish writing out of order; e.g. *WFC*==2 and *resv_count*==3, it is possible that locations 0, 1 and 2 are reserved, but only 0 and 2 are written, so we cannot know which locations are valid. Case 2, *WFC*¿*resv_count*, this cannot happen because a location must be reserved first before writing, so it is impossible to have more written locations than reserved locations. Finally, case 3, *WFC*==*resv_count*, any location up to *resv_count* is valid to read; this works because *resv_ptr* is loaded from memory **after** *WFC*, and thus the value of *resv_ptr* is at lease as recent as *WFC*; and since writes reserve location consecutively; therefore, if *WFC*==*resv_count*, the reserved locations must have been written. Furthermore, to avoid memory consistency issues, i.e. preventing the load requests for *WFC* and *resv_ptr* being reordered by the memory system, we place a memory fence between the two loads.

We have discussed our basic concurrent read/write algorithm. We now discuss our optimized implementation. In particular, we focus on reducing global memory metadata traffic by aggregating read/write operations and by utilizing GPUs' fast local scratch-pad memories. Recall that the *probe-warp* and the *assign-warp* in the MTB act together as the delegate reader for worker threads.

The probe-warp keeps a *read_ptr* for each bucket to remember the read progress. The *read_ptr* is private to the MTB, so it stored in the scratch-pad memory. The probe-warp finds valid buffer locations to read by checking *WFCs* of segments starting from *read_ptr*. WFCs are laid out consecutively in memory so that 32 threads of the probe warp can read 32 WFCs as a coalesced memory access. If any of the 32 segments is not full, then an additional access of *resv_ptr* is required. One design consideration is the segment size N, which trades off between overhead and concurrency. If N is small, then the overhead of each WFC check can only be amortized to small amount of reads; on the other hand, if N is too large, e.g. N equals total buffer size, then no read can be made if any work thread is in the process of writing (i.e. to be able to read, *WFC* must be equal to *resv_count*). Therefore, we choose N=256 as a balanced point, and the metadata checking overhead is amortized to 8K (32x256) reads. The probe-warp updates the *read_ptr* to the latest valid buffer location after each check.

The assign-warp checks the *read_ptr* and assigns valid buffer locations to worker threads, which then read out the buffered vertex IDs for processing. Assignments are to worker TBs rather than individual worker threads for efficiency. Each WTB (up to 1K threads on Nvidia GPUs) is associated with an assignment flag (*AF*)—a 64 bit global memory word. The *AF* is initialized to 0, meaning a WTB is available for accepting work; the assign-warp writes the flag, and the WTB reads the flag and resets it to 0. The bit fields of AF are shown below. Essentially, a WTB's assignment is a starting buffer index followed by a size.

The assign-warp polls *AFs* and the *read_ptr* to find available WTBs and available work. It is possible to have many available work but few available WTB temporarily, where more WTBs may become available shortly after; or vice versa; in such cases, it will create load imbalance if we just distribute all currently available work items to all currently available WTBs. Therefore, we set a lower bound and an upper bound for assignment size. The upper bound is 32 vertices per warp (or 1K per TB). The lower bound per warp is 32 divided by the average degree of the graph, rounded up to the next power of 2, so that each thread has at least 1 edge to process. Between the two bounds, the assign-warp choose a power of 2 value for per warp work based on available work items and available WTBs; it is specified by the *grain_shift* field in AF (i.e. 1¡¡*grain_shift*). WTBs are assigned in round robin order.

Each WTB also caches a local version of *AF* in scratch-pad memory to reduce accesses to global memory AF. Worker warps poll the local AF. If a warp finds the local AF to be 0, it then copies the assignment (if any) from the global AF to local AF; afterwards, each warp grabs a portion of assignment (specified by *grain_shift*) from the local AF until it is completed drained. Only one warp is allowed to access the global AF at a time to prevent unnecessary accesses; this is achieved by using a lock implemented in scratchpad memory.

5.5 Multiple Buckets

Our solution uses a fixed number of 32 buckets. Head and tail refer the current highest and lowest priority buckets (initially, bucket 0 and 31 are the head and tail). The head and tail bucket switch to next one in a circular way (e.g. 1 is now the head and 0 is the tail). Bucket switching is monotonically increment without backtracking; this is because the currently actively are always further away from the source vertex (in terms of distance).

The MTB controls bucket switching. The key design consideration is **when** to switch. One possible method is to switch after we have assigned all work items in the head bucket. However, work items from a bucket can generate more work to the same bucket upon being processed; so switching after assignment can result in premature bucket switch, where newly generated high priority work items are clipped to low priority buckets. In our experiment, this method often causes bucket switching to spiral out of control, where work items of multiple priorities are lumped together in the same bucket, and thus renders bucketing pointless.

Therefore, to slow down bucket switching, the MTB waits for assigned work items to finish processing in case more work items are enqueued to the head bucket. For this purpose, each bucket is associated with a done work counter (*DWC*)— a global memory word, initialized to 0. Suppose N work items from bucket A are assigned to a WTB, after all N work items are processed, the last warp that finishes its work atomically increase the *DWC* of A by N; so the *DWC* indicates how many work items from the bucket have been completely processed, and the overhead of updating DWC is amortized to each WTB's assignment (up to 1K work-items). On the MTB side, the probe-warp controls bucket switching; if the probe-warp finds no available work in the head bucket, then it switches the bucket if *DWC*==*resv_ptr*, i.e. all work items written to the head bucket (determined by *resv_ptr*) have been processed and no work new work item is written to the bucket (otherwise, *DWC*!=*resv_ptr*).

As an optimization, the MTB is allowed to assign (dequeue) work items from multiple high priority buckets (2 to 4) concurrently instead of just the head bucket; we call those buckets—*concurrent dequeue buckets* or *CDBs*. There are two reasons. First, the amount of work in a head bucket keeps decreasing, since fewer work items are being enqueued back the head bucket; and the MTB waits for the head bucket to completely finish before switching as mentioned earlier; so, in many cases, the parallelism is low toward the end of head bucket processing before switching happens. Therefore, the MTB is allowed to assign work items from the adjacent high priority bucket to avoid the low parallelism period. Second, our dynamic delta-picking mechanism can adjust the number of *CDBs*, in additional to adjust the delta, as a mean to balance between hardware utilization and work efficiency. Delta adjustment works a coarse-grain approach, while CDB adjustment works a complementary fine-grain approach (will be described in Section X). We choose 2 and 4 as min and max for CDBs. To support multiple CDBs, we use 4 probe-warps in the MTB for better probing concurrency (one probe-warp for each CDB), and we still use 1 assign-warp; when assigning work items, higher priority buckets are considered first, and lower priority buckets are considered only if there are available WTBs but no work in higher ones.

Finally, the algorithm terminates (i.e. converges) if all work items in the work storage are processed and no new work item is enqueued. The MTB is responsible for termination detection—when N buckets are switched consecutively, where N=2*NUM_ALL_BUCKETs (2x as a safety margin), and there is no work-item being probed nor assigned in any of the buckets. Upon termination, the assign-warp writes a special termination command (0xffffffffffffffff) to all WTBs' AF (assignment flag). WTBs check AFs for assignments anyway, so termination checks do not incur any new memory access overhead.

5.6 Dynamic Data Structure

We call the max number of active vertices of a graph at any time during execution (in all buckets)—*MAX_ACTIVE*. The distribution of vertices to buckets is non-uniform and unpredictable, and it changes during execution; in the worst case, all active vertices may cluster to the same bucket. Therefore, each bucket must be size of *MAX_ACTIVE*; if statically allocated, it uses too much memory and limits our ability to process large graphs. Our goal is to design a dynamic data structure while keeping the overhead minimal as if buckets are static arrays.

We statically allocate a pool of global memory and divided it into *data blocks* (64K of 32-bit words each), which are dynamically allocated to or de-allocated from each bucket based on actual usage.

The mechanisms discussed in previous sections work efficiently because each bucket is simply an array (used as a circular FIFO buffer); e.g. it allows probing, assigning and reserving consecutive locations, etc. We keep the array FIFO buffer data structure intact, so there is no change to those mechanisms. The only change is to the *buffer data array* (containing vertex IDs), which is now an array of pointers to actual *data blocks* (*bk_ptr*).

To access index *X* (e.g. 0xf0001) in the original *buffer data array* , the upper 16 bits of *X* (e.g. 0xf) is the index to the *bk_ptr* array; the retrieved *bk_ptr* is the starting location of the associated *data block* (e.g. 0x20000), which is then combined with the lower 16 bits of *X* to form the final index (e.g. 0x20001) to the pool of statically allocated global memory. So, essentially, the difference between a static array and our dynamic array is an additional translation step by reading the *bk_ptr* array resided in global memory.

To reduce translation overheads, WTBs cache the translation results in local scratch-pad memories. For each WTB, we implement a 4-entry directly mapped *translation cache* for each bucket. Each cache entry has a *tag*, which is the *bk_ptr* array index (e.g. 0xf for *X*), and a corresponding *result*, which is the *data block* ID (e.g. 0x2 for *X*). When performing a translation, the worker thread first check whether the *tag* of the cache entry matches its own *bk_ptr* array index; if they match, the cached *result* is used; otherwise, the thread reads the *bk_ptr* from global memory and updates the cache entry. To update an entry, the *tag* and the *result* must be modified together in an atomic step, since, otherwise, a cache access may happen in between and read stale *result*. To simplify design, we pack *tag* and *result* into a 32-bit word (16 bits for each); since reading or writing a word in memory is guaranteed to be atomic, it avoids using locks. WTBs' *translation caches* are only used for writing data. As to reading data, when the MTB assigns work to WTBs, the assign-warp performs translation and uses translated index in assignments; an assignment does not cross *data block* boundary. The MTB also has *translation caches* similar to WTBs.

The *mm-warp* in the MTB keeps track of free *data blocks* and modifies the *bk_ptr* array of each bucket to allocate/de-allocate *data blocks*. Since each bucket is just a FIFO, the *mm-warp* monitors the enqueue/dequueue progress to allocate/de-allocate in FIFO order; it is done automatically without worker threads' explicit involvement.

For allocations, the *mm-warp* records the current allocation progress of a bucket in the *alloc_ptr* (a scratch-pad memory variable) and checks a bucket's *resv_ptr* for enqueue progress; 32 threads of the mm-warp checks 32 buckets in parallel. We pre-allocate 2 *data blocks* ahead to hide allocation latency; so if the difference between *alloc_ptr* and *resv_ptr* is less than 2 blocks for a bucket, the *mm-warp* finds a free *data block* and then maps it to the bucket's *bk_ptr* array and increments the *alloc_ptr*. The status of *data blocks* is kept in a bitmask array in the scratch-pad memory; a free *data block* can be found quickly with 32 threads of warp checking in parallel. In addition, to further hide translation latency, the WTB prefetch 1 *data block* ahead from the *bk_ptr* array (which is pre-allocated) to the *translation cache*.

For de-allocations, the *mm-warp* needs to know when a *data block* can be safely freed in order not to destroy valid data items. In our solution, it can be easily determined due to the FIFO property of buckets, where an allocated *data block* can be freed if all its locations have been read once; so each *data block* is associated with a *read done counter* (or *RDC*, a global memory word). The last worker warp that finishes a WTB's assignment atomically increments the appropriate *RDC*; if a worker warp increments a *data block*'s *RDC* to 64K (block size), then the *data block* can be safely de-allocated from its current bucket; the worker warp first resets the related metadata (WDCs and RDC) of the *data block* and then set the last bit of the *bk_ptr* to 1 to indicate that it is free (note, the lower 16 bits of an allocated *bk_ptr* are 0s). Similar to *alloc_ptr*, the *mm-warp* also keeps a *dealloc-ptr* for each bucket to keep track of the FIFO de-allocation progress. The *bk_ptr* is checked by the *mm-warp* according to *dealloc-ptr*; if the last bit is set to 1, then the corresponding *data block* will be added back to the pool of free blocks.

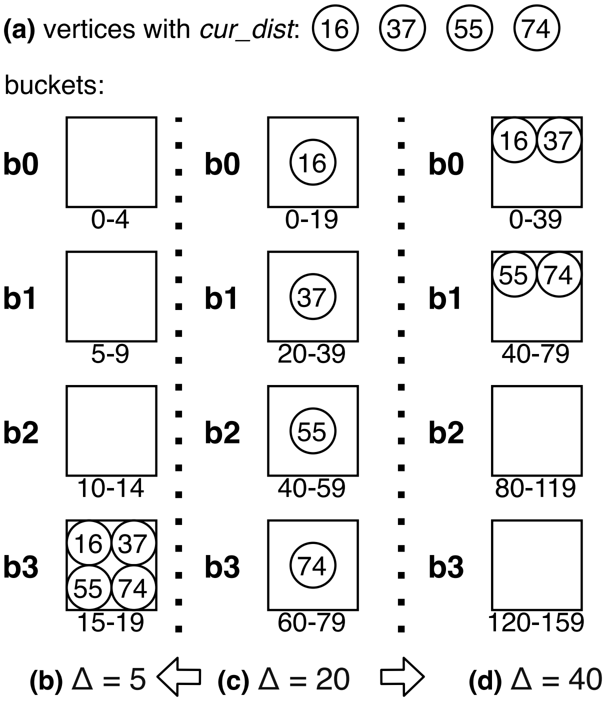


Figure 5.7: How Δ Affects Work Efficiency and Concurrency—pushing 4 vertices (a) to 4 buckets under 3 scenarios: when Δ =20 (c), it has best work efficiency; when increased to 40 (d), it improves concurrency;but when decreased to 5 (b), all vertices are clipped to the last bucket

5.7 Setting Δ dynamically

The choice of Δ has a significant impact on performance. This section first discusses how Δ affects performance in theory and in practices, and then based on the observation, we introduce a mechanism for setting Δ dynamically based on run time information.

Figure 5.7 illustrate how Δ affects performance in theory with a trivial example, where we push 4 vertices to 4 buckets under 3 different Δ choices. Let's start with the scenario in the middle, when Δ =20 **(c)**, it has the most precise ordering of vertices (i.e. closer to a priority queue) and thus gives best work efficiency. However, this Δ may not have enough concurrency to fully the hardware; in this case, increasing Δ **(d)** would increase the number of work items in each bucket (which can be processed in parallel) and therefore, improve concurrency.

The interesting case is when we decrease Δ to 5 **(b)**. One would assume decreasing Δ always improves ordering, but this assumption is only valid if the number of buckets is infinite. A finite number of buckets could only represent a limited range of priorities, so vertices out of the range would be **clipped** to the last bucket (e.g. *b3* in the figure). Therefore, we should avoid choose a too small Δ that causes too much clipping.

How Δ affects performance in practice We will now put theory into practice with realistic experiments. Figure 5.8 shows how performance and work efficiency correlate with Δ for 3 drastically different graphs. For each graph, we mark 3 Δ choices—the one that achieves best work efficiency (*best-work-point*), the one that

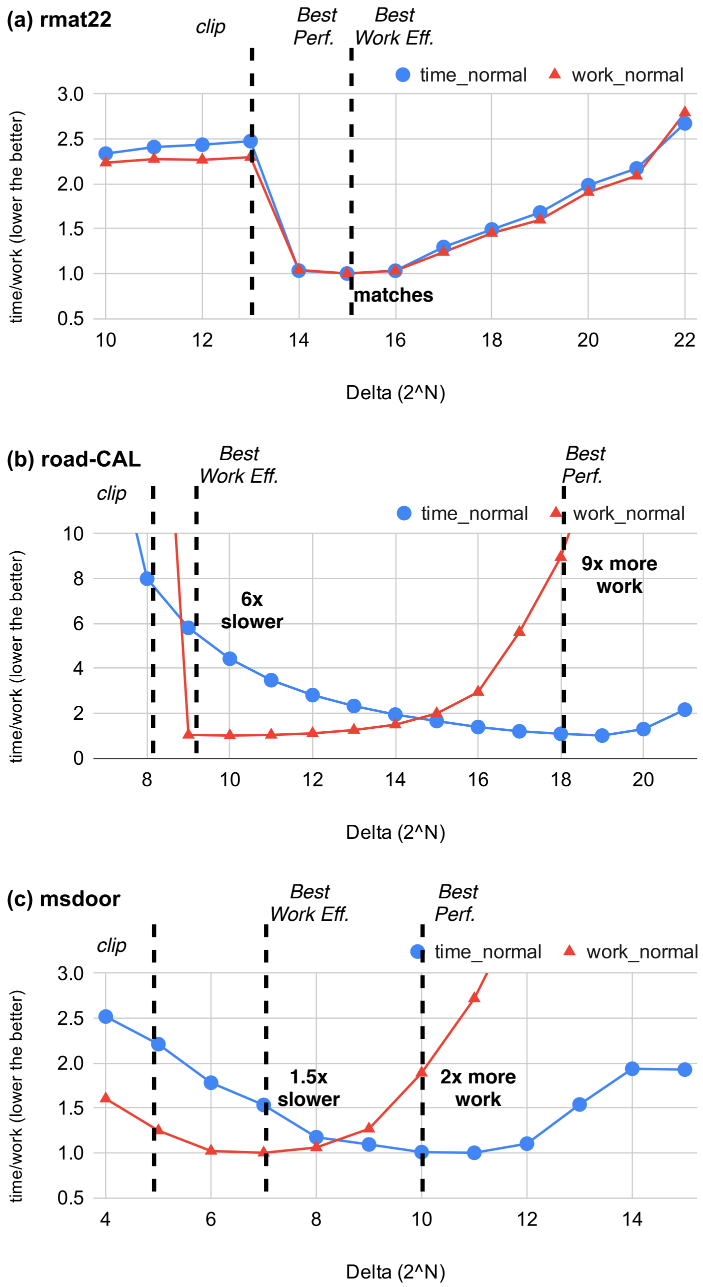


Figure 5.8: This Experiment Plots Execution Time and Work Performed Against Δ — the choices of Δ are predetermined and fixed during execution; both *time* and *work* are normalized to the lowest point (lower the better); finally, the experiments are done using 32 buckets

achieves best performance by balancing work efficient and concurrency (*best-perf-point*), and the one that causes clipping (*clip-point*), which roughly correspond to (**c**),(**d**), and (**b**) in Figure 5.7.

For the *RMAT* graph (a), the execution time correlates strongly with the amount of work performed. It indicates that there is enough work to keep the hardware fully utilized no matter the choice of delta, so Δ (*best-work-point*) with the least amount of work achieves the best performance (*best-perf-point*). The *ROAD* graph (b) is the opposite; the *best-work-point* causes severe hardware under-utilization, where the *best-perf-point* is 6× faster despite doing 9× more work; so hardware utilization is an important factor when choosing Δ for such graphs. The *MSDOOR* graph (c) is a midway between *RMAT* and *ROAD*, which the trade-off is much less extreme. Finally, for all 3 graphs, the *clip-point* always perform worse than the *best-work-point*, since it causes drastically more work without improving concurrency.

How to pick an optimal Δ Based on the above observations, we develop a run time mechanism that could automatically pick a near *best-perf-point* for a given graph. Our basic idea is as follows.

Before execution starts, we pick an initial Δ using a similar heuristic as previous solutions [31] (Section 5.2.3). Δ is rounded to power of 2. During execution, the MTB gathers run time information periodically and then makes the decision to either increase or to decrease Δ (i.e. shifting left or right). The process is a continuous feedback loop so that the MTB guides Δ closer to the optimal value at each period.

When shifting Δ , we should avoid the *clip-point* (or lower), which decreases work efficiency without improving concurrency. The *clip-point* can be determined by measuring the distribution of work items to buckets (Section 5.7.1 discusses the details). Also notice that in Figure 5.8, the *best-work-point* is always immediately to the right of the *clip-point*. So once we know the *clip-point*, we also know the *best-work-point*, which serves as the lower bound of Δ shifting.

Above the lower bound, we could safely assume that decreasing Δ increases work efficiency while decreasing concurrency, and vice versa. Our goal is to keep Δ near a point where the hardware is ***just about*** fully utilized. This point represents the optimal trade-off between work efficiency and concurrency. To understand the rationale, suppose we decrease Δ from the optimal point, it reduces concurrency so the GPU will have unused resources to accept more work; this makes the benefit of a smaller Δ (i.e. work saving) pointless. On the other hand, it is also pointless to increase Δ for more concurrency, since the hardware is already full utilization, where less work efficiency only hurts performance. Therefore, based on the rationale, we could determine whether the current Δ is optimal by measuring whether the current level of hardware utilization is near *just-about-utilized point*.

To measure hardware utilization, recall that the *assign-warp* in the MTB checks on the worker TBs and assigns work to them (Section 5.4), so the assign-warp knows how many work items are currently being processed by the workers, from which, we derive the current hardware utilization level (*cur_util*). We also define an under-utilization point and an over-utilization point (*util_low* and *util_high*) based on the total number of hardware threads of work TBs and the characteristics of the graph. The basic idea is to increase or to decrease Δ when *cur_util* is below *util_low* or above *util_high* so that *cur_util* is trapped near the *just-about-utilized point*.

To illustrate how the mechanism works, considering the graphs in Figure 5.8 again, for *RMAT* (a), Δ is firstly shifted to the *best-work-point* (i.e. the lower bound); at this point, *cur_util* is measured higher than *util_high*, and thus we attempt to decrease Δ , but Δ is already at the lower bound; in other words, Δ already gives most work efficiency, so decreasing Δ would not increase work efficient further. This pins down Δ at the point with both best work efficient and enough hardware utilization so that the optimal performance is achieved.

For *ROAD* (b), at the lower bound, there is severe hardware under-utilization, so *cur_util* is measured much lower than *util_low*, which causes Δ to increase. When Δ is near the *best-perf-point*, increasing Δ further will cause *cur_util* to exceed *util_high*, which decrease Δ , and vice versa; this keeps Δ near the optimal value.

The rest of this section discusses several important design considerations of our Δ setting mechanism in details.

5.7.1 Finding the *Clip-Point*

We define *N* as the number of work items pushed to a bucket during a period of time. We consider a Δ to be a *clip-point* if *N* of the tail bucket (or a bucket near tail) is a high percentage of the sum of *N* of all buckets. In particular, if the tail bucket's percentage is above a threshold *P*, then the current Δ should be increased to avoid clipping.

An appropriate threshold *P* should be chosen so that Δ immediate above *clip-point* achieves best work-efficiency (i.e. the *best-work-point*, the lower bound). Based on our experiments using 226 graphs, we determine that it is sufficient to use a constant *P* for all graphs; in particular, *P*=65% works well for most graphs when 32 buckets are used.

The MTB measures *N* periodically to calculate the tail bucket's percentage for adjusting the delta. To measures *N*, the MTB monitors the *resv_ptr* of each bucket in epochs. Recall that threads increase the *resv_ptr* when writing to a bucket, so *N* is the increment of *resv_ptr* between two consecutive epochs, i.e. the newly written items. In addition, whenever Δ changes, we skip the measurement of the following epoch in order for new Δ to take effect.

5.7.2 Changing Δ Based on Utilization

Our solution periodically measures *cur_util* (hardware utilization) and makes decision to increase or decrease Δ in order to guide Δ to an optimal point, where the GPU is just about fully utilized. For the idea to work properly, there are following challenges to deal with.

First, changing Δ too frequently has negative impacts on work efficiency because this causes work items of different priorities to mix up together in the same buckets. Ideally, once Δ is near the optimal point, it should remain stable instead of flip-flopping.

Second, *cur_util* fluctuates even if Δ is fixed. One scenario, for example, is when we start to process a new bucket (i.e. switching the head bucket) that has accumulated many work items that remain unprocessed previously; *cur_util* will temporally jumps and eventually falls down. In general, we should avoid changing Δ reactively based on temporal *cur_util* behaviors, especially when Δ is already optimal.

Third, after Δ change, *cur_util* changes gradually instead of immediately. In fact, the time it takes for the change to take full effect varies depending on many factors. So when guiding Δ to the optimal, we should avoid eager changes in order not to overshoot the target.

We employ the following measures to deal with the challenges.

First, recall that the assign-warp assigns work from multiple (*N*) high priority buckets instead of just the head bucket alone (Section 5.5). We could control the parameter *N* to adjust the trade-off between concurrency and work-efficiency (i.e. higher the *N* more concurrency, etc.). This is more fine-grained than changing Δ , so we use it as a form of immediate adjustment. In particular, *N* has a lower bound and a high bound (we choose them to be 2 and 4), we change *N* first based on current hardware utilization; if *N* is already at the bounds, we change Δ . By doing this, we reduce the frequency of Δ change.

Second, for changing *N*, we only use *cur_util* of the current epoch, but for changing Δ , we use the average value of several previous epochs. In addition, *util_low* and *util_high* (i.e. the bounds that trigger changes) are chosen conservatively so that there is a relative big gap between them. Those measures make Δ changes less sensitive to temporary *cur_util* fluctuations.

Third, after changing Δ , we must wait for *cur_util* to settle before changing Δ again to avoid overshooting the target. Of course, the key problem is to figure out the wait time. The time depends on the graph input and the current Δ value. Generally speaking, for a given graph, it takes longer for the *cur_util* to settle when the current Δ is larger, so the wait time must scale with Δ instead of being constant. Our solution is to count *head bucket switches*. The rationale is that a larger Δ means more work items in each bucket, and thus it takes more time to finish processing the head bucket before switching; therefore, the time scales naturally with Δ . In particular, after a Δ change, we wait for T head bucket switches before changing *delta* again; T=3 works best in our experiments.

5.8 Methodology

For evaluating ADDS and prior solutions, we run GPU implementations on an Nvidia RTX 2080 ti GPU (Turing, TU102) [100] with driver 440.44; our CUDA toolkit version 10.0 [102]. The GPU's specifications are listed in Table 5.1.

SM Count	68	Threads Per SM	1024
Max Clock Rate	1.75 GHz	GDDR6 Bandwidth	616 GB/s
DRAM Size	11 GB	L2 Size	5.5 MB
Scratchpad Per SM	48 KB	Compute Capability	7.5

Table 5.1: RTX 2080 ti GPU

We run shared memory and serial CPU implementations on a Intel Core i9-7900X CPU, which has 10 cores and 20 hardware threads running at 3.3 GHz.

5.8.1 Graph Inputs

We use a set of 226 graph inputs for our experiments. Those graphs are from two sources. Road network (e.g. road-USA), power law (e.g. RMAT22), and random (e.g. r4-2e23) graphs are from the Lonestar benchmark suite [69]. The remaining graphs are from the SuiteSparse Matrix Collection (formerly University of Florida) [32]; our selection criteria is as follows. We first select all weighted graphs with at least 100k vertices and 1M edges, and they must also fit in our GPU's memory (11GB); a few large graphs (e.g. HV15R) can run with ADDS but not with nvGRAPH and Lonestar's Near Far, since they use more memory than ADDS, so we exclude those graphs; as to Near Far, it allocates 3 arrays of —E— words for double buffering the near far pile, where as ADDS' dynamic data structure uses —E— words of memory in total; as to nvGRAPH, it is a black box implementation, so we are not sure about the reason. We then select graphs suitable for SSSP traversal, where at least 75% of vertices can be reached from a source vertex; for each graph, we find an appropriate source (from vertex 0 to 1000) and filer it on failure by marking vertices with BFS traversal. Finally, we convert the graph's negative edge weights (if any) to positive weights.

Table 5.2 shows the distribution of those graphs in terms of average degree and diameter.

As one can see, our input set contains graphs with a wide range of characteristics, and the distribution of graphs is relatively uniform. It allows us to evaluate ADDS and prior implementation in an unbiased manner.

Average Degree					
<4	4 - 8	8 - 16	16 - 32	32 - 64	>=64
17 (8%)	59 (26%)	34 (15%)	23 (10%)	71 (31%)	22 (10%)
Diameter					
<40	40 - 80	80 - 169	160 - 320	320 - 640	>=640
54 (24%)	33 (15%)	49 (22%)	29 (13%)	32 (14%)	29 (13%)

Table 5.2: The Distribution of Graph Characteristics—count(% of 226 graphs)

5.8.2 Evaluated Prior Implementations

Besides ADDS, we evaluated 6 other SSSP implementations—1 hardwired implementation and 5 implementations from well known and well maintained graph frameworks (3 on GPU and 2 on CPU). We now introduce each of them and describe the relevant modifications we made to their source code.

NearFar-OPT (NF-OPT) is a well optimized hardwired implementation of Near Far [31] from LonestarGPU 4.0 benchmark suite [15], which is the state-of-art GPU SSSP implementation. Our experiments focus on comparing ADDS with NF-OPT. We now briefly discuss some of key features of NF-OPT.

NF-OPT has an optimization [87, 105] that improves BSP for high diameter graphs. BSP requires a global barrier between super-steps to function properly. A basic GPU approach is to use kernel launch boundary as the global barrier, which induces latency by communicating with the driver over the PCI-E bus. High diameter graphs are especially sensitive to such latency as they tend to have little work in each super-step. NF-OPT use a software implemented global barrier [140] and launch just one kernel to handle all super-steps, which reduces the delay to just global memory latency. For example, it reduces the run time of road-USA graph from 760ms to 387ms with this approach. However, NF-OPT is fundamentally limited by low parallelism caused by BSP; in comparison, ADDS ditches BSP altogether, which further reduces road-USA run time to 125ms. In addition, NF-OPT uses threads in a warp to process vertices' edges cooperatively [87, 105] to improve load balance and coalesced memory accesses, and it has a procedure that removes duplicate (redundant) vertices IDs in the worklist; the two optimizations are also mentioned in the Near-Far paper [31]. ADDS also uses cooperative edge processing but no duplicate vertex ID removal, since it requires BSP.

We made changes to NF-OPT for our experiments. The original version requires the user to manually input a Δ value for a graph, otherwise, a default constant value (10000) is used; we changed it to use the equation (Δ =c*w/d, c=32) from the Near-Far paper instead. In addition, we added support for graphs with floating point weights (using a software implemented atomicMin from Gunrock1.0 [135], as does ADDS) and changed the warp level primitives (e.g. vote) to synced version (e.g. vote.sync) so that the code can work correctly with our Turing GPU[2].

Gunrock-BF is a **GPU** Bellman-Ford based implementation from Gunrock 1.0 [135] [138]. It uses a worklist to store vertex IDs and removes duplicate vertex IDs from the queue (called filter). The original Gunrock paper [135] uses a Near-Far based implementation instead of BF; we found it in an older of version of Gunrock.

[2] without the change, the SSSP kernel still finishes execution but produces wrong results

Gunrock-NF is a **GPU** Near-Far implementation from Gunrock 0.2. We evaluated both versions of Gunrock. For Gunrock-NF, we use c=32 for the Δ equation. We added support for floating point (to Gunrock 0.2) same as NF-OPT and ADDS. The filter for duplicate ID removal is commented out in Gunrock 0.2's source code; we uncommented it, which gives overall better performance.

nvGRAPH (NV) is a **GPU** linear-algebra based (semi-ring) SSSP implementation [101] from CUDA 10.0. It is a proprietary graph library from NVIDIA.

CPU-DS is a **shared memory** CPU delta-stepping [88] implementation from Galois 4.0 [70, 110]. It uses many very fine-grained buckets for priority. It expects a manually tuned Δ value from the user, so we changed it to use the same equation from Near-Far (Δ =c*w/d), except using c=1 instead of c=32, since CPUs require less parallelism.

Dijkstra is a **serial** implementation of Dijkstra's algorithm [**?**] from Galois 4.0, which implements the priority queue as binary heap.

5.9 Evaluation

In this section, we present the timing results of ADDS and prior implementations and analyze the performance differences with supporting experiments.

5.9.1 Timing Results

The timing results for all 226 graphs are shown in Table 5.4 (due to space limitation, Gunrock-NF, CPU-DS and Dijkstra are not shown). We summarize the speedup of ADDS over the 6 prior implementations in Table 5.3.

	<0.9x	0.9x - 1.1x	1.1x - 1.5x	1.5x - 2x	2x - 3x	3x - 5x	>=5x
NF-OPT	**8 (4%)**	**13 (6%)**	**27 (12%)**	**44 (19%)**	**54 (24%)**	**59 (26%)**	**21 (9%)**
Gun-NF	20 (9%)	2 (1%)	10 (4%)	14 (6%)	27 (12%)	40 (18%)	113 (50%)
Gun-BF	16 (7%)	3 (1%)	8 (4%)	8 (4%)	20 (9%)	49 (22%)	122 (54%)
NV	18 (8%)	4 (2%)	7 (3%)	12 (5%)	13 (6%)	24 (11%)	148 (65%)
CPU-DS	7 (3%)	2 (1%)	1 (%)	8 (4%)	28 (12%)	38 (17%)	142 (63%)
Dijkstra	2 (1%)	0 (%)	0 (%)	0 (%)	3 (1%)	14 (6%)	207 (92%)

Table 5.3: Speedup of ADDS over prior implementations—the distribution of 226 graphs over speedup intervals

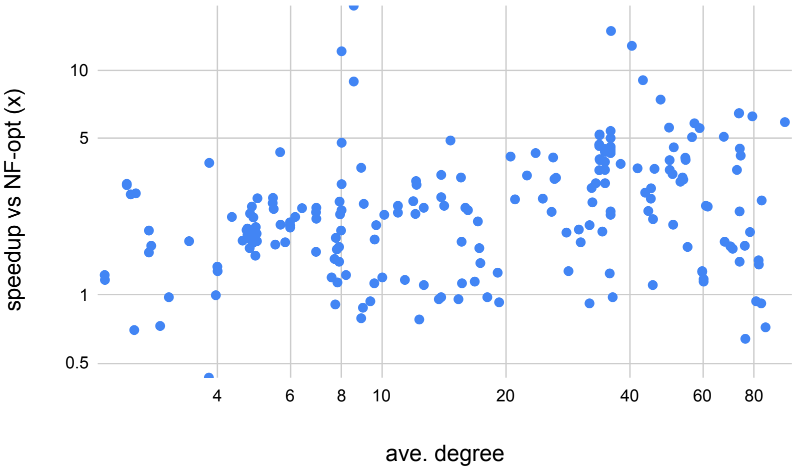


Figure 5.9: The distribution of ADDS' speedup over NF-OPT correlating to graph degree

.

Comparing to NF-OPT, ADDS achieves an average speedup of 2.9x; ADDS

	speedup	ADDS	NF	BF	NV
	(x)	(ms)	(ms)	(ms)	(ms)
al2010	1.75	5.9E+0	1.0E+1	7.3E+1	5.3E+1
atmosmodm	1.56	4.1E+0	6.4E+0	3.7E+1	1.4E+2
az2010	2.01	4.8E+0	9.7E+0	7.8E+1	4.5E+1
ca2010	1.97	1.1E+1	2.1E+1	2.4E+2	1.7E+2
engine	3.99	6.1E-1	2.4E+0	2.8E+0	5.3E+0
fl2010	2.48	5.4E+0	1.3E+1	7.5E+1	6.6E+1
ga2010	1.71	4.7E+0	8.1E+0	5.2E+1	5.4E+1
GAP-road	3.13	1.2E+2	3.8E+2	3.9E+4	3.9E+4
GL7d19	1.26	5.9E+0	7.4E+0	2.2E+1	1.9E+1
GL7d20	1.13	5.3E+0	6.0E+0	1.4E+1	9.6E+0
GL7d21	1.11	3.0E+0	3.3E+0	1.1E+1	6.0E+0
GL7d22	1.2	1.2E+0	1.4E+0	4.3E+0	2.7E+0
GL7d23	0.91	6.0E-1	5.5E-1	1.3E+0	1.1E+0
Hardesty1	2.73	7.6E+0	2.1E+1	7.7E+1	3.2E+2
ia2010	1.86	4.1E+0	7.7E+0	6.6E+1	4.2E+1
il2010	2.31	5.5E+0	1.3E+1	1.3E+2	8.4E+1
in2010	1.81	5.5E+0	1.0E+1	7.3E+1	5.5E+1
kron_17	1.91	1.6E+0	3.1E+0	2.0E+0	1.5E+0
kron_18	0.94	4.9E+0	4.6E+0	2.5E+0	2.9E+0
kron_19	0.92	7.2E+0	6.7E+0	7.7E+0	5.7E+0
kron_20	0.72	1.7E+1	1.2E+1	2.1E+1	1.3E+1
ks2010	1.95	5.7E+0	1.1E+1	8.0E+1	6.0E+1
mc2depi	1.34	1.7E+1	2.3E+1	1.7E+2	2.9E+2
mi2010	1.62	5.9E+0	9.7E+0	9.7E+1	6.3E+1
mn2010	1.98	5.7E+0	1.1E+1	9.4E+1	6.3E+1
mo2010	1.76	6.1E+0	1.1E+1	1.2E+2	7.9E+1
n4c6-b10	1.14	3.8E-1	4.3E-1	8.9E-1	8.2E-1
n4c6-b8	0.88	4.0E-1	3.5E-1	8.4E-1	9.2E-1
n4c6-b9	0.94	4.3E-1	4.0E-1	1.1E+0	7.8E-1
nc2010	1.9	5.2E+0	9.8E+0	7.2E+1	5.8E+1
neos3	1	3.0E+3	3.0E+3	2.7E+4	5.8E+4
ny2010	1.91	6.4E+0	1.2E+1	1.0E+2	7.7E+1
oh2010	1.77	5.7E+0	1.0E+1	8.7E+1	7.7E+1
ok2010	1.86	6.9E+0	1.3E+1	7.4E+1	6.7E+1
pa2010	2.22	5.5E+0	1.2E+1	1.0E+2	7.7E+1
pds-100	1.23	5.7E-1	7.0E-1	2.7E+0	2.2E+0
pds-90	1.17	4.9E-1	5.7E-1	2.3E+0	1.6E+0
r4-2e23	1.28	3.5E+1	4.5E+1	1.5E+2	3.2E+2
rmat20	1.64	6.1E+0	1.0E+1	1.5E+1	1.0E+1
rmat22	2.29	3.5E+1	8.0E+1	1.4E+2	2.4E+2
sx-stack	0.98	1.6E+1	1.6E+1	2.8E+1	1.9E+1
t2em	1.74	1.3E+1	2.3E+1	1.7E+2	4.0E+2
TF19	0.96	7.6E-1	7.3E-1	1.9E+0	2.2E+0
tn2010	1.88	6.5E+0	1.2E+1	7.9E+1	6.5E+1
tx2010	1.96	1.0E+1	2.0E+1	2.9E+2	2.3E+2
road-CAL	2.81	3.6E+1	1.0E+2	6.8E+2	1.3E+3
road-FLA	2.84	3.1E+1	8.7E+1	4.4E+2	6.3E+2
road-NY	1.66	9.9E+0	1.6E+1	8.4E+1	8.0E+1
road-USA	3.09	1.3E+2	3.9E+2	4.0E+4	4.0E+4
va2010	1.91	9.5E+0	1.8E+1	1.3E+2	8.9E+1
wi2010	1.8	4.3E+0	7.8E+0	6.4E+1	4.5E+1
wiki-talk	0.73	9.8E+0	7.2E+0	1.8E+0	1.7E+0
2cubes	1.62	4.7E+0	7.6E+0	2.2E+1	1.4E+1
af_0_k101	5.38	8.3E+1	4.5E+2	7.7E+2	9.9E+2
af_1_k101	5	8.6E+1	4.3E+2	7.7E+2	9.8E+2
af_2_k101	2.36	7.3E+1	1.7E+2	2.8E+2	4.5E+2
af_3_k101	2.28	7.6E+1	1.7E+2	2.3E+2	4.5E+2
af_4_k101	4.58	4.0E+1	1.9E+2	2.9E+2	5.8E+2
af_5_k101	4.55	4.1E+1	1.9E+2	2.9E+2	5.5E+2
af_shell10	14.98	8.1E+1	1.2E+3	1.7E+3	2.0E+3
af_shell1	4.51	1.2E+1	5.4E+1	1.3E+2	2.8E+2
af_shell2	4.41	1.2E+1	5.2E+1	1.3E+2	2.7E+2
af_shell3	4.42	1.2E+1	5.4E+1	1.0E+2	2.7E+2
af_shell4	4.49	1.2E+1	5.4E+1	9.4E+1	2.9E+2
af_shell5	4.31	1.2E+1	5.2E+1	1.2E+2	2.8E+2
af_shell6	4.3	1.2E+1	5.2E+1	1.3E+2	2.7E+2
af_shell7	4.29	1.2E+1	5.3E+1	1.1E+2	2.8E+2
af_shell8	4.33	1.2E+1	5.3E+1	1.0E+2	3.3E+2
af_shell9	4.3	1.2E+1	5.2E+1	9.9E+1	3.3E+2
apache2	1.8	1.2E+1	2.2E+1	5.8E+1	1.7E+2
ASIC_320k	1.23	2.3E+1	2.8E+1	2.9E+0	6.8E-1
ASIC_320ks	2.06	1.2E+0	2.5E+0	4.4E+0	5.5E+0
ASIC_680k	4.33	1.7E+1	7.4E+1	1.3E+0	1.3E+0
ASIC_680ks	1.74	1.4E+0	2.5E+0	5.2E+0	4.9E+0
atmosmodd	2.19	4.6E+0	1.0E+1	3.4E+1	1.1E+2
atmosmodj	2.45	4.1E+0	1.0E+1	3.2E+1	1.3E+2
atmosmodl	2.34	4.8E+0	1.1E+1	4.1E+1	1.6E+2
audikw_1	2.64	5.1E+1	1.3E+2	3.0E+2	3.4E+2
barrier2-10	5.18	8.9E-1	4.6E+0	2.6E+0	7.0E+0
barrier2-11	5.19	8.9E-1	4.6E+0	3.9E+0	9.3E+0
barrier2-12	5.15	9.1E-1	4.7E+0	4.2E+0	9.2E+0
barrier2-1	4.67	8.8E-1	4.1E+0	3.7E+0	8.9E+0
barrier2-2	4.61	8.9E-1	4.1E+0	1.7E+1	6.6E+0
barrier2-3	4.68	8.7E-1	4.1E+0	3.7E+0	6.7E+0
barrier2-4	3.6	1.1E+0	4.1E+0	3.7E+0	6.8E+0
barrier2-9	4.03	1.2E+0	4.7E+0	3.6E+0	1.3E+1
BenElechi1	4	1.4E+1	5.6E+1	1.0E+2	1.9E+2
bmw3_2	3.46	9.7E+0	3.4E+1	7.5E+1	6.9E+1
bmw7st_1	3.2	7.3E+0	2.3E+1	5.3E+1	4.3E+1
bmwcra_1	3.61	3.9E+0	1.4E+1	2.2E+1	2.7E+1
bone010	2.36	7.2E+0	1.7E+1	2.9E+1	2.2E+2
boneS01	3.27	1.1E+0	3.7E+0	8.8E+0	1.3E+1
boneS10	2.48	7.3E+0	1.8E+1	2.9E+1	2.2E+2
boyd2	3.88	8.8E+0	3.4E+1	1.0E+0	5.9E-1
Bump_2911	2.69	4.5E+1	1.2E+2	3.1E+2	9.0E+2
bundle_adj	12.87	9.9E+1	1.3E+3	2.8E+2	2.5E+2
c-73b	19.42	2.3E+0	4.5E+1	1.4E+0	1.0E+0
c-73	8.93	2.1E+0	1.9E+1	2.0E+0	8.7E-1
cage12	1.73	6.4E-1	1.1E+0	2.2E+0	4.2E+0
cage13	1.15	1.9E+0	2.2E+0	4.1E+0	8.5E+0
cage14	0.98	7.6E+0	7.4E+0	1.3E+1	3.2E+1
cage15	0.93	3.2E+1	3.0E+1	4.6E+1	1.4E+2
c-big	1.6	3.5E+0	5.6E+0	5.4E+0	4.9E+0
cfd2	4.1	3.9E+0	1.6E+1	3.6E+1	4.4E+1
circuit5M_dc	2.71	1.4E+1	3.8E+1	1.5E+2	2.0E+2
CO	1.25	2.0E+0	2.5E+0	4.4E+0	4.5E+0
cop20k_A	3.41	3.4E+0	1.2E+1	3.2E+1	2.5E+1
CoupCons3D	3.34	7.8E+0	2.6E+1	4.0E+1	6.5E+1
crashbasis	2.5	7.7E+0	1.9E+1	7.2E+1	6.5E+1
Cube_dt0	1.28	8.1E+1	1.0E+2	2.6E+2	7.2E+2
Cube_dt6	1.27	9.4E+1	1.2E+2	3.1E+2	8.0E+2
CurlCurl_1	2.62	2.3E+0	5.9E+0	1.3E+1	2.1E+1
CurlCurl_2	2.3	4.8E+0	1.1E+1	3.7E+1	6.9E+1
CurlCurl_3	3.22	7.4E+0	2.4E+1	7.0E+1	1.3E+2
CurlCurl_4	3.1	1.2E+1	3.6E+1	1.7E+2	3.0E+2
darcy003	2.11	1.2E+1	2.5E+1	1.5E+2	1.6E+2
degme	0.78	4.7E+1	3.6E+1	2.8E+0	2.5E+0
dgreen	0.92	3.9E+1	3.5E+1	6.9E+1	8.4E+1
dielFilterV2	9.05	1.7E+1	1.5E+2	3.0E+2	3.6E+2
dielFilterV3	1.37	3.7E+1	5.0E+1	9.3E+1	1.4E+2
d_pretok	2.05	8.6E+0	1.8E+1	5.0E+1	3.8E+1
Dubcova3	2.35	6.6E+0	1.6E+1	4.2E+1	1.6E+1
ecology1	2	1.9E+1	3.8E+1	1.5E+2	4.6E+2
ecology2	2.04	1.9E+1	3.9E+1	1.5E+2	4.6E+2
Emilia_923	2.18	1.7E+1	3.8E+1	1.1E+2	1.7E+2
F1	6.25	9.0E+0	5.6E+1	1.2E+2	1.1E+2
Fault_639	3.65	1.2E+1	4.4E+1	8.1E+1	1.6E+2
FEM_3D_2	4.29	1.8E+0	7.8E+0	1.3E+1	1.9E+1
filter3D	3.33	2.3E+0	7.7E+0	1.2E+1	1.3E+1
Flan_1565	0.64	1.0E+2	6.6E+1	2.9E+2	4.0E+2
Freescale1	1.68	1.7E+1	2.8E+1	1.7E+2	1.9E+2
Freescale2	1.45	1.2E+1	1.7E+1	8.3E+0	2.2E+1
FullChip	0.79	3.2E+2	2.6E+2	2.4E+1	3.5E+1
G3_circuit	1.72	6.9E+0	1.2E+1	4.2E+1	1.7E+2
Ga10As10	1.64	2.0E+0	3.3E+0	5.0E+0	3.6E+0
Ga19As19	1.73	2.9E+0	5.0E+0	1.3E+1	4.1E+0
Ga41As41	1.65	7.8E+0	1.3E+1	2.4E+1	8.6E+0
Ge87H76	1.61	2.0E+0	3.2E+0	1.0E+1	3.6E+0
Ge99H100	1.66	2.1E+0	3.5E+0	5.6E+0	3.9E+0
Geo_1438	2.99	2.0E+1	6.1E+1	9.9E+1	2.5E+2
Goodwin_095	3	8.2E+0	2.4E+1	4.3E+1	4.2E+1
Goodwin_127	2.59	2.0E+1	5.2E+1	1.3E+2	9.8E+1
gsm_106857	3.84	8.3E+0	3.2E+1	8.3E+1	7.2E+1
Hamrle3	0.43	7.9E+0	3.4E+0	4.9E+0	1.0E+1
helm2d03	3.12	4.9E+0	1.5E+1	4.4E+1	6.2E+1
hood	3.99	4.5E+0	1.8E+1	4.9E+1	4.4E+1
Hook_1498	3.67	3.3E+1	1.2E+2	2.4E+2	5.4E+2
imagesensor	2.45	2.8E+0	6.8E+0	1.6E+1	7.0E+0
inline_1	4.18	3.4E+1	1.4E+2	3.3E+2	4.2E+2
kkt_power	1.41	4.8E+1	6.7E+1	6.5E+1	4.8E+1
language	0.98	1.9E+0	1.8E+0	2.4E+0	2.3E+0
largebasis	2.45	3.2E+2	7.9E+2	2.3E+3	6.8E+3
ldoor	3.61	1.6E+1	5.8E+1	1.9E+2	5.0E+2
Lin	2.61	1.9E+0	5.0E+0	1.1E+1	2.0E+1
Long_dt0	1.18	9.7E+1	1.1E+2	3.0E+2	4.7E+2
Long_dt6	1.15	1.7E+2	2.0E+2	4.4E+2	7.0E+2
mac_fwd500	2.23	1.8E+1	4.1E+1	1.0E+2	8.2E+1
majorbasis	2.33	3.5E+0	8.1E+0	2.0E+1	2.5E+1
marine1	3.34	1.5E+1	5.1E+1	1.2E+2	9.7E+1
mario002	2.05	1.2E+1	2.4E+1	1.1E+2	1.6E+2
matrix_9	4.14	1.2E+0	5.1E+0	1.7E+1	9.2E+0
memchip	2.42	1.0E+1	2.5E+1	3.1E+2	3.0E+2
ML_Geer	4.49	6.7E+1	3.0E+2	2.2E+2	2.1E+3
ML_Laplace	6.46	1.2E+1	7.8E+1	6.6E+1	2.8E+2
msdoor	5.57	1.4E+1	7.5E+1	1.4E+2	3.9E+2
nlpkkt120	1.28	1.5E+1	2.0E+1	2.5E+1	2.5E+2
nlpkkt80	1.9	4.1E+0	7.7E+0	1.1E+1	5.3E+1
nv2	0.98	1.1E+2	1.0E+2	1.5E+2	7.7E+1
nxp1	2.44	1.1E+1	2.6E+1	1.1E+1	1.2E+1
offshore	1.39	1.2E+1	1.7E+1	3.2E+1	3.2E+1
ohne2	2.5	1.5E+0	3.8E+0	6.5E+0	8.4E+0
para-10	4.4	8.9E-1	3.9E+0	3.7E+0	8.5E+0
para-4	3.15	1.2E+0	3.8E+0	3.8E+0	8.2E+0
para-5	4.34	9.0E-1	3.9E+0	4.6E+0	8.5E+0
para-6	4.42	9.0E-1	4.0E+0	3.6E+0	8.5E+0
para-7	3.92	1.0E+0	4.0E+0	7.2E+0	8.5E+0
para-8	4.47	9.1E-1	4.1E+0	3.7E+0	8.5E+0
para-9	3.61	1.1E+0	3.9E+0	3.7E+0	8.5E+0
parabolic_fem	12.17	1.9E+1	2.3E+2	1.4E+2	2.4E+2
PFlow_742	4.55	2.9E+1	1.3E+2	2.6E+2	2.9E+2
power9	2.39	5.2E+0	1.2E+1	2.3E+1	1.4E+1
PR02R	2.06	2.0E+1	4.1E+1	6.5E+1	6.2E+1
pre2	2.55	4.2E+0	1.1E+1	1.4E+1	2.2E+1
pwtk	4.09	9.4E+0	3.9E+1	7.5E+1	1.1E+2
radiation	1.92	3.9E+0	7.5E+0	1.9E+1	6.7E+0
Raj1	1.5	5.1E+0	7.7E+0	5.9E+0	6.8E+0
rajat21	1.75	8.5E+0	1.5E+1	1.2E+1	1.6E+1
rajat24	2.57	8.6E+0	2.2E+1	6.8E+0	6.7E+0
rajat29	1.2	5.7E+1	6.8E+1	1.5E+1	1.4E+1
rajat30	1.13	6.8E+1	7.7E+1	1.6E+1	2.3E+1
rajat31	2.23	1.2E+1	2.6E+1	1.0E+2	9.9E+2
Serena	7.42	2.2E+1	1.7E+2	4.0E+2	3.6E+2
ship_003	5.07	1.1E+0	5.6E+0	6.3E+0	1.1E+1
shipsec1	5.05	1.2E+0	5.9E+0	7.5E+0	1.6E+1
shipsec5	5.82	1.4E+0	8.3E+0	1.1E+1	2.4E+1
shipsec8	5.54	9.6E-1	5.3E+0	6.5E+0	1.1E+1
Si41Ge41	1.43	3.6E+0	5.2E+0	7.1E+0	5.6E+0
Si87H76	1.11	4.5E+0	5.0E+0	9.2E+0	7.0E+0
SiO2	1.41	4.6E+0	6.5E+0	4.0E+0	3.8E+0
ss	2.67	1.1E+1	3.0E+1	4.1E+1	2.6E+2
StocF-1465	0.96	2.7E+2	2.6E+2	4.2E+2	3.0E+2
stomach	2.5	8.1E+0	2.0E+1	4.7E+1	5.9E+1
stormG2_1000	0.7	2.7E+0	1.9E+0	4.7E+0	1.2E+1
TEM152078	2.86	3.2E+0	9.1E+0	2.2E+1	1.3E+1
TEM181302	2.37	4.0E+0	9.4E+0	2.3E+1	1.7E+1
test1	3.15	5.4E+0	1.7E+1	3.9E+1	4.5E+1
thermal2	2.39	1.6E+1	3.9E+1	1.8E+2	4.9E+2
thermo_dK	3.43	8.7E+0	3.0E+1	9.2E+1	9.8E+1
thermo_dM	1.94	7.9E+0	1.5E+1	7.1E+1	7.5E+1
tmt_sym	4.77	2.9E+1	1.4E+2	4.3E+2	5.9E+2
tmt_unsym	2.7	1.1E+1	3.0E+1	7.1E+1	2.7E+2
torso1	6.44	1.1E+1	7.0E+1	3.8E+1	5.0E+1
torso2	3.69	8.3E+0	3.1E+1	1.0E+2	7.1E+1
torso3	2.13	6.0E+0	1.3E+1	2.8E+1	3.4E+1
tp-6	1.17	8.6E+1	1.0E+2	3.8E+0	3.7E+0
Transport	4.88	9.4E+0	4.6E+1	2.4E+2	3.6E+2
TSOPF_c30	3.29	9.0E+0	3.0E+1	2.1E+0	1.2E+0
turon_m	1.77	7.0E+0	1.2E+1	3.0E+1	3.0E+1
twotone	2.28	1.4E+0	3.2E+0	5.6E+0	5.9E+0
vas_1M	2.05	9.7E+0	2.0E+1	3.7E+1	1.4E+2
vas_2M	1.72	2.2E+1	3.8E+1	7.8E+1	3.6E+2
vas_4M	1.96	4.9E+1	9.7E+1	1.6E+2	1.2E+3
watson_1	1.94	6.3E-1	1.2E+0	2.7E+0	2.9E+0
watson_2	1.55	9.2E-1	1.4E+0	2.6E+0	3.3E+0
x104	5.89	1.6E+0	9.4E+0	1.2E+1	1.9E+1
xenon2	2.69	1.6E+0	4.3E+0	1.1E+1	1.5E+1

Table 5.4: The execution time of ADDS, NearFar-OPT (**NF**), Gunrock-Bellman-Ford (BF), and nvGRAPH (NV). The speedup column is ADDS over NF.

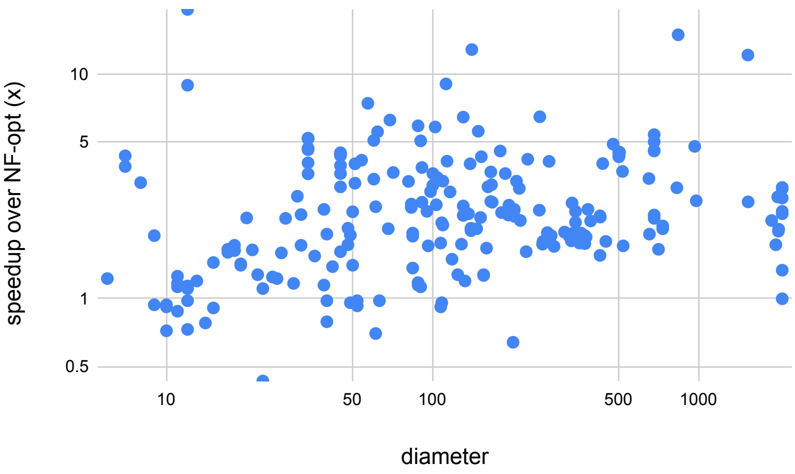


Figure 5.10: The distribution of ADDS' speedup over NF-OPT correlating to graph diameter

.

degrades performance (¡0.9x) for only 4% of graphs and achieves non-trivial speedup (¿1.5x) for 78% of graphs (up to 19x). Comparing to other 3 GPU implementations, ADDS achieves an average speedup of 5.8x, 9.6x and 13.4x over Gunrock-NF, Gunrock-BF and nvGRAPH respectively. The results show that ADDS achieves major performance gains on GPUs; the reason for speedup is improved work efficiency and increased parallelism, which will be examined in details later. It also shows that NF-OPT is the best performing one among prior implementations, so we focus on comparing ADDS with NF-OPT.

Comparing to CPU implementations, ADDS achieves an average speedup of 14.2x over Galois' shared memory CPU-DS and 34.4x over the serial Dijkstra's. It demonstrate the benefits of performing SSSP on GPUs.

Figure 5.9 and 5.10 show the speedup of ADDS over NF-OPT correlated to the average degree and the diameter of graphs respectively. As one can see, ADDS achieves speedups for all types of graphs without bias, where the distribution is relatively uniform. This is because ADDS optimizes on both parallelism and work-efficiency, and because our dynamic mechanism is able to automatically pick a suitable delta to balance between the two based on graphs' run time behavior.

5.9.2 Work Efficiency

Table 5.5 summarizes ADDS' total vertex process count normalized to others. NvGRAPH is not shown because it is a black box implementation.

	<0.25x	0.25x - 0.5x	0.5x - 0.75x	0.75x - 1x	1x - 1.5x	1.5-3x	>3x
NF-OPT	**10 (4%)**	**22 (10%)**	**13 (6%)**	**24 (11%)**	**75 (33%)**	**70 (31%)**	**12 (5%)**
Gun-NF	50 (22%)	25 (11%)	38 (17%)	35 (15%)	34 (15%)	36 (16%)	8 (4%)
Gun-BF	61 (27%)	21 (9%)	20 (9%)	28 (12%)	64 (28%)	25 (11%)	7 (3%)
CPU-DS	18 (8%)	21 (9%)	29 (13%)	18 (8%)	39 (17%)	37 (16%)	64 (28%)
Dijkstra	0 (%)	0 (%)	0 (%)	0 (%)	30 (13%)	49 (22%)	147 (65%)

Table 5.5: Normalized vertex processing count of ADDS (lower the better)

Comparing to NF-OPT, ADDS achieves non-trivial work saving (¡0.75x) for 26% of graphs. The saving is achieved by using 32 instead of just 2 buckets for more precise priority. For 36% of graphs, ADDS does noticeable more work (¿1.5x). The main reason is that ADDS' dynamic mechanism will pick a larger Δ if the GPU is under utilized; combined with asynchronous processing, it improves parallelism but leads to more work. In addition, ADDS does not have the duplicate vertex ID removal filter in NF-OPT, since it requires BSP to work. On the other hand, ADDS' multi bucket scheme can mitigate some of the work inefficiency; for 44 % of graphs, ADDS does similar amount of work (0.75x to 1.5x).

Our goal is to improve performance rather than work saving; if the improved hardware utilization outweighs the work-efficiency loss, it is still overall beneficial to performance. Therefore, on average, ADDS achieves 2.9x speedup over NF-OPT despite processing 1.55x more vertices. We now analyze ADDS' performance gains in details.

5.9.3 Performance Analysis

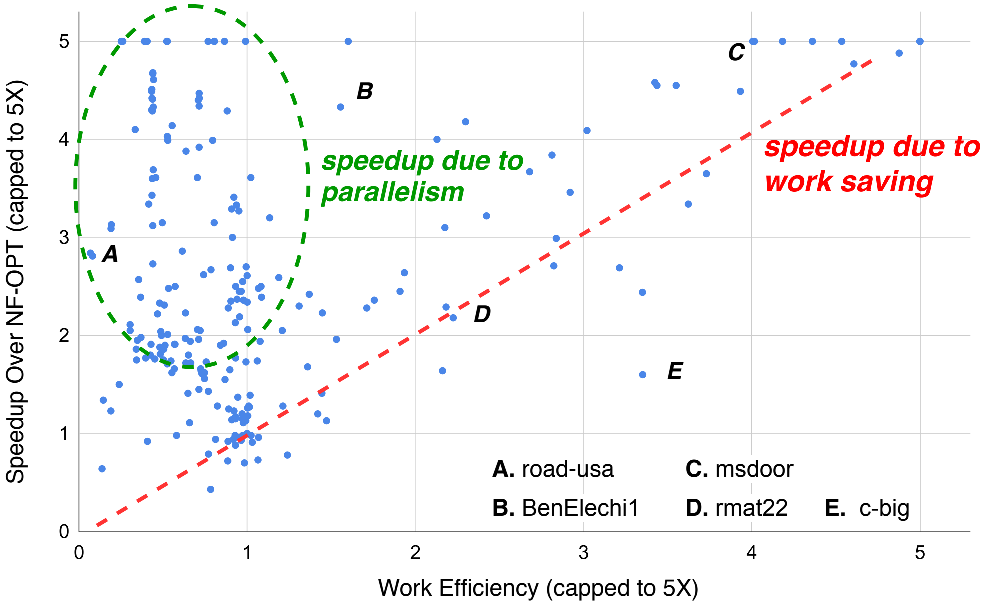


Figure 5.11: The correlation between speedup and work-efficiency (inverse of vertex count); both higher the better.

.

Figure 5.11 plots the correlation between speedup (ADDS over NF-OPT) and work saving for all 226 graphs. The diagonal line represents perfect correlation between work saving and speedup; so for graphs on or around this line (e.g. D.rmat22 and C.msdoor), the speedup is mainly due to work saving. For graphs in the upper left region (e.g. A.road-USA), ADDS does more work yet achieves speedups; many graphs are clustered in this region, which means NF-OPT under utilize hardware, while ADDS achieves speedup by increasing parallelism. For graphs in between the two region (e.g. B.BenElechi1), the speedup is due to both increased parallelism and work saving. Finally, the lower right region means that ADDS achieves work saving but, by doing so, decreases parallelism, so the speedup is lower than work saving; in fact, there is only 1 graph (E.c-bag) in this region far off the diagonal line. Figure 5.12 to 5.16 examines those regions in details.

Those figures plot the amount of parallelism during the course of execution in terms of edge count (vertex-count * average-degree). For ADDS, the vertex count is measured as the number of work items currently assigned to workers; for NF-OPT, it is measured as the number of work items to be processed in the worklist (the near pile) at the beginning of each BSP super-step.

Road-USA (Figure 5.12) represents one of extreme cases, where ADDS achieves 3x speedup yet does 5x more work. This is because NF-OPT's 2 bucket scheme works well in terms of work efficiency; however, it severely under-utilizes the GPU; in most of super-steps, the edge count is just less than 1K (the maximum is only 14K), where as the GPU has 68K threads. ADDS achieves much higher parallelism as the figure shows; this is because our concurrent reader writer bucket data structure allows newly active vertices to be processed immediately (asynchronous processing) so that vertices' processing can be overlapped; another reason is that ADDS' dynamic mechanism is able to increase Δ when hardware utilization is low.

Although ADDS trades off work efficiency for parallelism comparing to

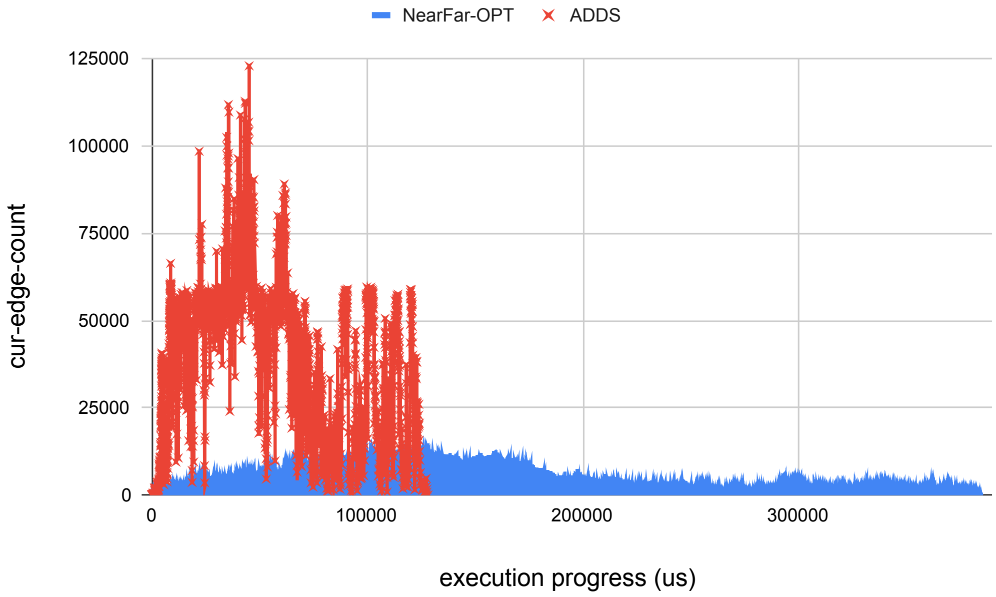


Figure 5.12: A.road-USA: s:3.09x, w:0.19x (s:speedup, w:work-efficiency), the figure plots the amount of parallelism (edge count) during the progress of execution (us)

.

NF-OPT, ordering is still extremely important for road network graphs. For example, Gunrock's Bellman-Ford implementation does 78x more work than ADDS while being 318x slower. Therefore, ADDS' dynamic mechanism works well, since it is able to pick a suitable Δ for better parallelism while avoiding degenerating ADDS into a Bellman-Ford solution.

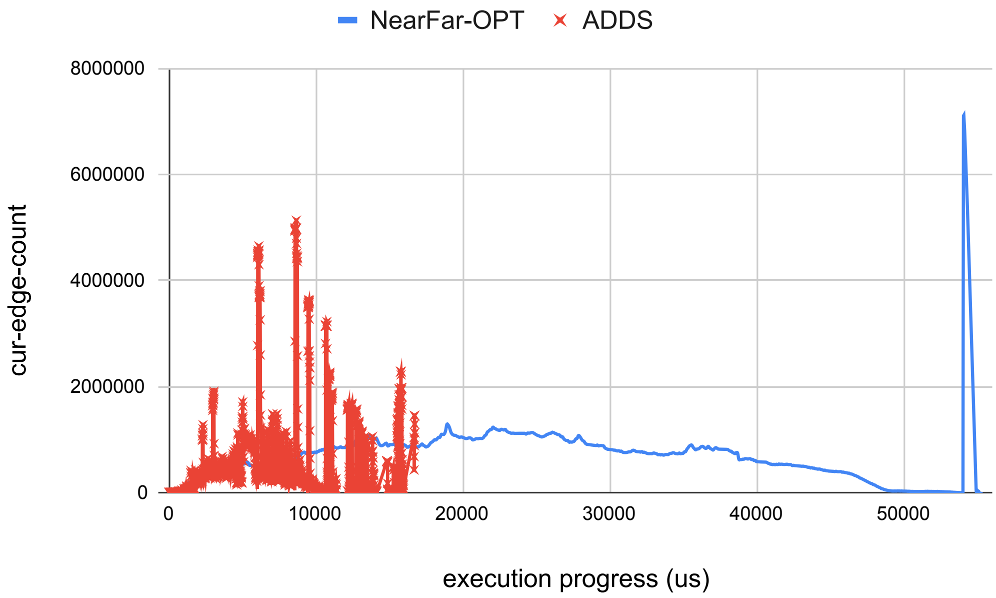


Figure 5.13: B.BenElechi1: s:4x, w:2.12x

.

BenElechi1 (Figure 5.13) represents cases where ADDS benefits both parallelism and work efficiency comparing to NF-OPT. In this case, NF-OPT still underutilizes the hardware (though better than road-USA). On the hand, ADDS' bucket data structure allows both concurrent-read-write and multi-bucketing; with a suitable Δ chosen by the dynamic mechanism, ADDS achieves 2x work saving while increasing also increasing parallelism, so the combined effect is a speedup of 4x.

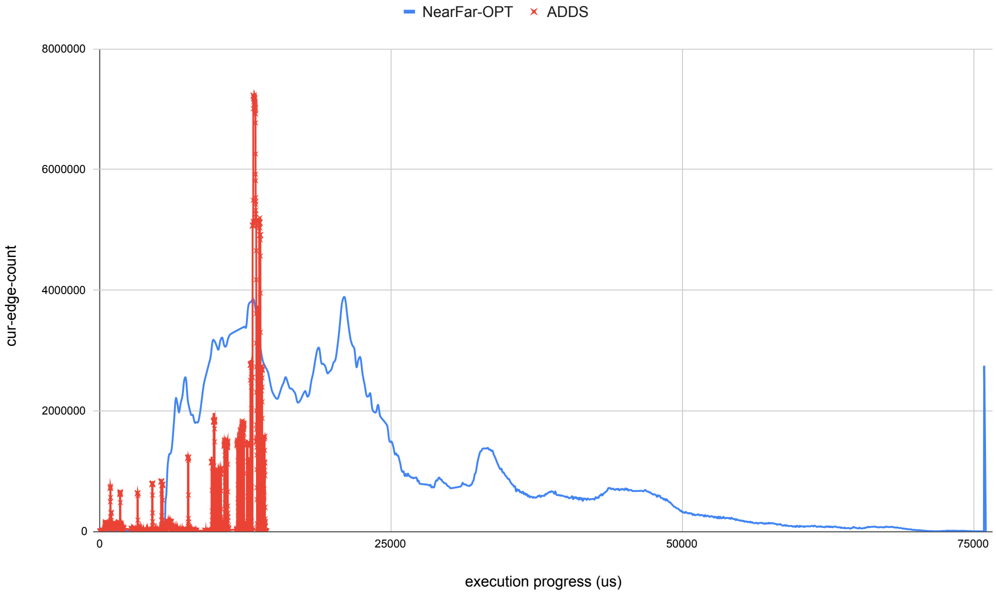


Figure 5.14: C.msdoor: s:5.57x, w:4x

.

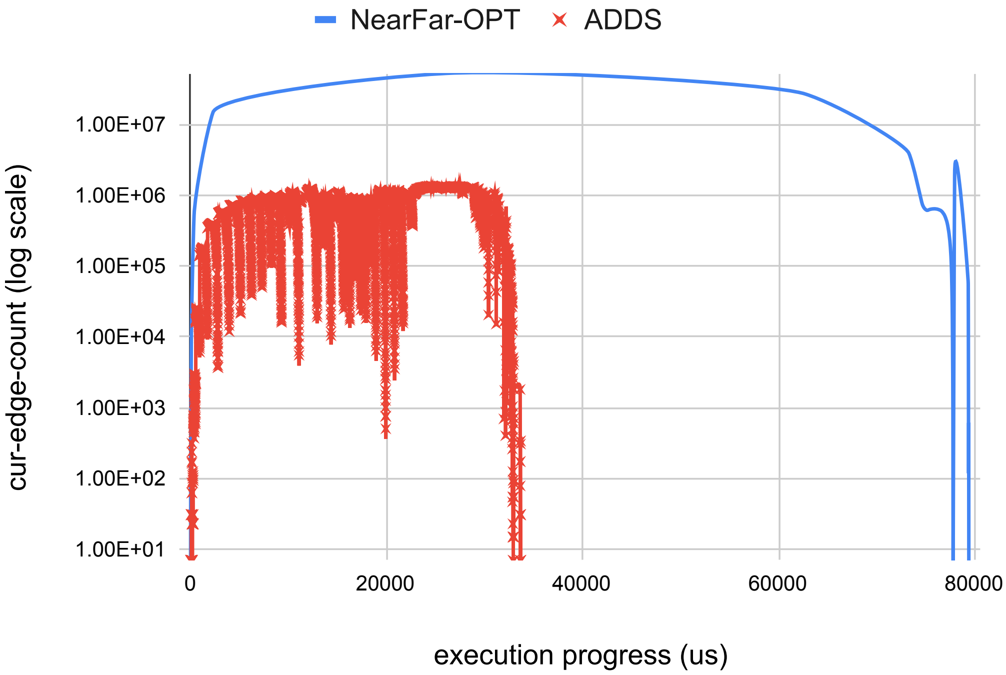


Figure 5.15: D.rmat22: s:2.29x, w:2.18x

.

For msdoor and rmat22 (Figure 5.14 and 5.15), NF-OPT achieves good hardware utilization, so ADDS achieves speedup mainly by being more work efficient, which is enabled by the multi-bucket capability. For msdoor, NF-OPT's parallelism is low during the last quarter of execution, so ADDS' speedup is still higher than work saving; while, for rmat22, both NF-OPT and ADDS are able to fully saturate the hardware, so the speedup correlates perfectly with work saving[3].

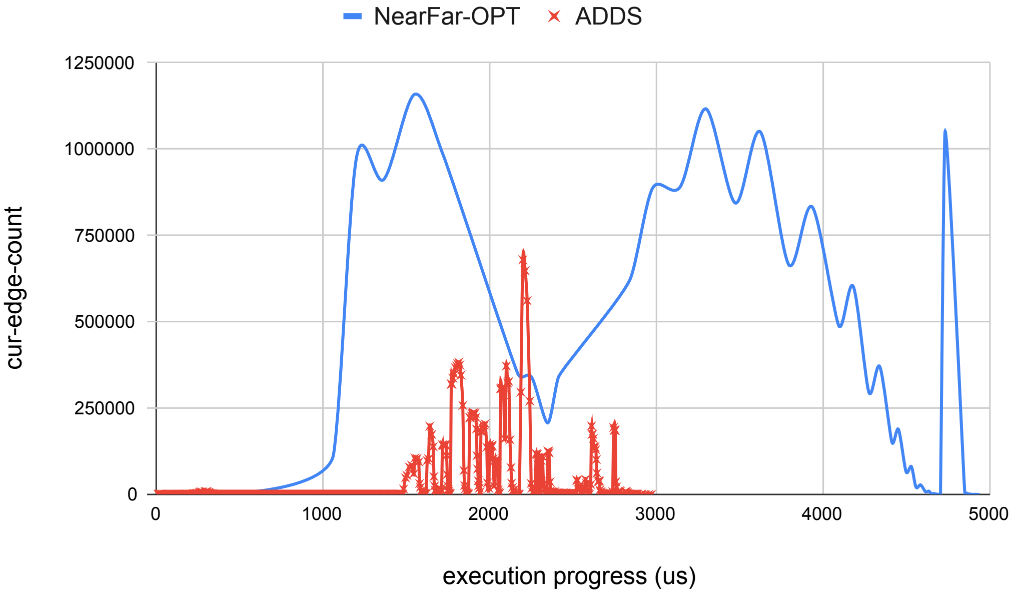


Figure 5.16: E.c-big: s:1.6x, w:3.35x

.

Finally, for c-big (Figure 5.16), ADDS achieves 3.35x work saving but a smaller speedup of 1.6x is achieved. One reason is that the total execution is short (3

[3] in the figure, cur-edge-count for NF-OPT is the amount of available work at the beginning of each BSP super-step, which is much larger than the GPU's thread count for the rmat graph; while, for ADDS, it is the currently assigned work; so the figure does not mean that NF-OPT processes more work concurrently than ADDS

ms), so ADDS' dynamic Δ is unable to increase the delta in time, so the parallelism is very low in the first half of execution.

5.10 Summary

In summary, ADDS' concurrent read-write multi-bucket design has the capability of improving parallelism and improving work-efficiency. ADDS' dynamic mechanism is able to utilize the multi-bucket properly by choosing a appropriate Δ according to the graph's run time characteristics.

Meanwhile our decoupled delegate approach (i.e. the worklist manager) amortizes metadata and reduces synchronization so that we can implement these advanced features efficiently on GPUs.

Therefore, ADDS is able to achieve good performance for a variety of graphs.

Chapter 6

Conclusion[1]

In this thesis, we have shown that the idea of *decoupled delegate* is applicable to various algorithm and hardware design problems for GPUs, namely, *scalar-like computation*, *fine-grained synchronization*, and *SSSP graph program*. Based on the decoupled delegate approach, we have presented solutions that advance the state-of-the-art for these 3 problems.

Our key idea is to *decouple* and to centralize appropriate computation routines to one or a few warps, instead of letting all warps handle them directly, where the decoupled warps collectively act as a *delegate* for other warps. In general, our decoupled delegate approach has two major benefits.

The first benefit is to avoid synchronization across a large number of threads/warps. For example, for fine-grained synchronization, the baseline solution lets all threads synchronize directly via locks in global memory. When there are lock contentions, a large number of threads create significant lock polling and serialization in the slow global memory. Our solution delegates the critical section execution

[1] Portions of this chapter are based on the following publications:
Affine Computation for SIMT GPUs, ISCA 2017[134]
Fast Fine-Grained Global Synchronization on GPUs, ASPLOS 2019[133]
A Fast Work-Efficient SSSP Algorithm for GPUs, to be appear in PPOPP 2021
My contributions: I'm the first author for all three papers

to a single thread block, which reduces the number of threads that access locks and allows much faster local scratchpad memories to be used for locking.

The second benefit is to amortize overheads. For example, to handle a concurrent worklist, a non-decoupled approach lets all threads access and update the worklist directly, so each thread must pay the various metadata overheads individually. Our solution performs worklist management with a delegate, which pays the metadata overheads once for a large number of worklist accesses/updates. In fact, the rationale is similar to amortizing instruction overheads with a vector unit, which is well known in the architecture community.

We end this thesis with several observations and questions pertaining to future researches for GPUs and beyond:

6.1 Why does the decoupled delegate approach work well on GPUs, and what other platforms may benefit from this approach?

Our decoupled approach achieves saving by amortizing overheads and/or reducing synchronization. In general, the amount of saving is proportional to the number of threads that require the said overheads and synchronization. Massive parallel processors, such as GPUs are good candidates in that regard.

On the other hand, the cost of our approach is the need of communication between regular threads and the delegate. Using our fine-grained synchronization solution as an example, the communication is done via the GPU's L2 cache, which is relatively efficient in a sense that it is still on-chip. Furthermore, we also amortize various communication overheads to reduce the cost, which is only possible if there are many requests in flight created by the many threads on the GPU.

Therefore, the massive parallel nature of GPUs makes the decoupled delegate a viable approach for appropriate problems.

As to other platforms, heterogeneous processors, such SoCs are also good candidates for the decoupled approach. For example, in our SSSP algorithm, the decoupled worklist manager can run on CPU cores, while the worker threads can run on GPU cores. For many SoC, the CPU and GPU cores share the last level cache, which can be used for communication. Furthermore, many modern SoCs also support features, such as fine-grained Shared Virtual Memory in OpenCL2.0, which allows atomic operations between CPU and GPU threads so that they can work cooperatively.

On the other hand, platforms, such as distributed systems have massive parallelism, but the interconnect is too slow. In this case, it is not viable to use a decoupled delegate for amortizing overheads across distributed processors, since the cost of communication would likely outweigh the saving. A more reasonable approach is to deploy a delegate on each processor.

6.2 Should we make irregular algorithms architectural efficient or algorithmic efficient? Or could we have both?

In general, when implementing irregular algorithms on GPUs, there is a tendency to focus exclusively on architectural efficiency, which is to promote convergent control flow and coalesced memory accesses. For example, prior GPU algorithms [135, 138, 69, 31, 52, 51, 16, 130, 18, 19] achieve this by adopting simplified concurrent data structures by avoiding the use of fine-grained locks. However, by doing so, it often sacrifices algorithmic efficiency.

For the SSSP problem, the prior state-of-art GPU algorithm (Near-Far) [31, 69] uses a simplified approximated priority queue with just two buckets and double buffering, which suffers from work efficiency and concurrency problems. As another example, the prior state-of-art GPU algorithm for the MaxFlow problem [52] is a lock-free implementation, which suffers from redundant work.

In this thesis, we have demonstrated that algorithmic efficient solutions can be made architecture efficient as well. Our new SSSP algorithm with a more sophisticated worklist achieves 2.8x speedup when compared to the prior state-of-art, and our lock-based MaxFlow implementation achieves 3x performance improvements. In fact, we achieve this without adding specialized hardware to GPUs, instead, the key is to employ an innovative programming paradigm.

Prior solutions use the data parallelism paradigm exclusively, i.e. executing the algorithm on a large number of homogenous threads. With this constraint, programmers have little choice but to simplify the algorithm to make it suitable for GPUs architecture. In contrast, we combine data parallelism with the decoupled delegate approach, which gives us the flexibility to adopt more complex algorithm designs. To be specific, the majority of threads work in data parallelism for architectural efficiency, while the complexity is shielded away by the delegate.

As to future research, our worklist design used for SSSP can be made general for other algorithms that benefit from approximate ordering. Our fine-grained locks can be adopted when designing lock-based GPU algorithms. More generally, our decoupled delegated approach can be used for other algorithm design problems, in particular, for making complex algorithm designs architecturally efficient on GPUs.

Furthermore, we believe our work opens a new playground for future GPU algorithm designs. We have demonstrated that the existing GPU hardware is capable of handling many complexity algorithms, while the key is to transform a complex algorithm in innovative ways to take advantage of the hardware instead of just simplifying the algorithm.

Bibliography

[1] Movielens 10m network dataset – KONECT, April 2017.

[2] Netflix network dataset – KONECT, April 2017.

[3] Reuters network dataset – KONECT, April 2017.

[4] J. L. Abelln, J. Fernndez, and M. E. Acacio. Glocks: Efficient support for highly-contended locks in many-core cmps. In *2011 IEEE International Parallel Distributed Processing Symposium*, pages 893–905, May 2011.

[5] Johnathan Alsop, Marc S. Orr, Bradford M. Beckmann, and David A. Wood. Lazy release consistency for GPUs. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, pages 26:1–26:13, Piscataway, NJ, USA, 2016. IEEE Press.

[6] AMD. Amd graphics cores next (gcn) architecture, 2012.

[7] José-María Arnau, Joan-Manuel Parcerisa, and Polychronis Xekalakis. Boosting mobile gpu performance with a decoupled access/execute fragment processor. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 84–93, Washington, DC, USA, 2012. IEEE Computer Society.

[8] Krste Asanovic, Stephen W. Keckler, Yunsup Lee, Ronny Krashinsky, and Vinod Grover. Convergence and scalarization for data-parallel architectures. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–11, Washington, DC, USA, 2013. IEEE Computer Society.

[9] Saman Ashkiani, Andrew Davidson, Ulrich Meyer, and John D. Owens. Gpu multisplit. *SIGPLAN Not.*, 51(8), February 2016.

[10] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 163–174, April 2009.

[11] RICHARD BELLMAN. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[12] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. Groute: An asynchronous multi-gpu programming model for irregular computations. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, page 235–248, New York, NY, USA, 2017. Association for Computing Machinery.

[13] James Bennett and Stan Lanning. The Netflix prize. In *Proc. KDD Cup*, pages 3–6, 2007.

[14] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, February 1984.

[15] M. Burtscher, R. Nasre, and K. Pingali. A quantitative study of irregular programs on gpus. In *2012 IEEE International Symposium on Workload Characterization (IISWC)*, pages 141–151, Nov 2012.

[16] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. A quantitative study of irregular programs on gpus. pages 141–151, 11 2012.

[17] Martin Burtscher and Keshav Pingali. *GPU Computing Gems Emerald Edition*, chapter 6, An Efficient CUDA Implementation of the Tree-Based Barnes Hut n-Body Algorithm, pages 75–92. Morgan Kaufmann, 2011.

[18] F. Busato and N. Bombieri. An efficient implementation of the bellman-ford algorithm for kepler gpu architectures. *IEEE Transactions on Parallel and Distributed Systems*, 27(08):2222–2233, aug 2016.

[19] F. Busato and N. Bombieri. An efficient implementation of the bellman-ford algorithm for kepler gpu architectures. *IEEE Transactions on Parallel and Distributed Systems*, 27(8):2222–2233, Aug 2016.

[20] Irina Calciu, Dave Dice, Tim Harris, Maurice Herlihy, Alex Kogan, Virendra Marathe, and Mark Moir. Message passing or shared memory: Evaluating the delegation abstraction for multicores. In *Proceedings of the 17th International Conference on Principles of Distributed Systems - Volume 8304*, OPODIS 2013, pages 83–97, Berlin, Heidelberg, 2013. Springer-Verlag.

[21] V. T. Chakaravarthy, F. Checconi, P. Murali, F. Petrini, and Y. Sabharwal. Scalable single source shortest path algorithms for massively parallel systems. *IEEE Transactions on Parallel and Distributed Systems*, 28(7):2031–2045, July 2017.

[22] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 44–54, Washington, DC, USA, 2009. IEEE Computer Society.

[23] G. Chen and X. Shen. Free launch: Optimizing gpu dynamic kernel launches through thread reuse. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 407–419, 2015.

[24] S. Chen and L. Peng. Efficient gpu hardware transactional memory through early conflict resolution. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 274–284, March 2016.

[25] S. Chen, L. Peng, and S. Irving. Accelerating gpu hardware transactional memory with snapshot isolation. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 282–294, June 2017.

[26] Sylvain Collange. Identifying scalar behavior in cuda kernels. Technical report, UCBL), 2011. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.405.3537&rep=re

[27] Sylvain Collange, David Defour, and Yao Zhang. Dynamic detection of uniform and affine vectors in gpgpu computations. In *Proceedings of the 2009 International Conference on Parallel Processing*, Euro-Par'09, pages 46–55, Berlin, Heidelberg, 2010. Springer-Verlag.

[28] Neal Clayton Crago and Sanjay Jeram Patel. Outrider: Efficient memory latency tolerance with decoupled strands. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 117–128, New York, NY, USA, 2011. ACM.

[29] Andreas Crauser, Kurt Mehlhorn, Ulrich Meyer, and Peter Sanders. A parallelization of dijkstra's shortest path algorithm. In *Proceedings of the 23rd International Symposium on Mathematical Foundations of Computer Science*, MFCS '98, page 722–731, Berlin, Heidelberg, 1998. Springer-Verlag.

[30] PTX ISA :: Cuda toolkit documentation.

[31] A. Davidson, S. Baxter, M. Garland, and J. D. Owens. Work-efficient parallel gpu methods for single-source shortest paths. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 349–359, May 2014.

[32] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), December 2011.

[33] D. Delling, A. V. Goldberg, A. Nowatzyk, and R. F. Werneck. Phast:

Hardware-accelerated shortest path trees. In *2011 IEEE International Parallel Distributed Processing Symposium*, pages 921–931, May 2011.

[34] Laxman Dhulipala, Guy Blelloch, and Julian Shun. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '17, page 293–304, New York, NY, USA, 2017. Association for Computing Machinery.

[35] Jeffrey R. Diamond, Donald S. Fussell, and Stephen W. Keckler. Arbitrary modulus indexing. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 140–152, 2014.

[36] Gregory Diamos, Benjamin Ashbaugh, Subramaniam Maiyuran, Andrew Kerr, Haicheng Wu, and Sudhakar Yalamanchili. Simd re-convergence at thread frontiers. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 477–488, New York, NY, USA, 2011. ACM.

[37] Dave Dice, Virendra J. Marathe, and Nir Shavit. Flat-combining numa locks. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, page 65–74, New York, NY, USA, 2011. Association for Computing Machinery.

[38] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math.*, 1(1):269–271, December 1959.

[39] A. ElTantawy and T. M. Aamodt. Mimd synchronization on simt architectures. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–14, Oct 2016.

[40] A. ElTantawy and T. M. Aamodt. Warp scheduling for fine-grained synchronization. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 375–388, Feb 2018.

[41] Roger Espasa and Mateo Valero. Decoupled vector architectures. In *Proceedings. Second International Symposium on High-Performance Computer Architecture*, pages 281–290, Feb 1996.

[42] Panagiota Fatourou and Nikolaos D. Kallimanis. A highly-efficient wait-free universal construction. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, page 325–334, New York, NY, USA, 2011. Association for Computing Machinery.

[43] Panagiota Fatourou and Nikolaos D. Kallimanis. Revisiting the combining synchronization technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 257–266, New York, NY, USA, 2012. ACM.

[44] W. W. L. Fung and T. M. Aamodt. Energy efficient gpu transactional memory via space-time optimizations. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 408–420, Dec 2013.

[45] Wilson W. L. Fung, Inderpreet Singh, Andrew Brownsword, and Tor M. Aamodt. Hardware transactional memory for gpu architectures. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 296–307, New York, NY, USA, 2011. ACM.

[46] Mark Gebhart, Stephen W Keckler, and William J Dally. A compile-time managed multi-level register file hierarchy. In *Proceedings of the 44th annual IEEE/ACM international symposium on microarchitecture*, pages 465–476. ACM, 2011.

[47] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Proceedings of the 7th International Conference on Experimental Algorithms*, WEA'08, page 319–333, Berlin, Heidelberg, 2008. Springer-Verlag.

[48] Syed Zohaib Gilani, Nam Sung Kim, and Michael J. Schulte. Power-efficient computing for compute-intensive gpgpu applications. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 445–446, New York, NY, USA, 2012. ACM.

[49] GroupLens Research. MovieLens data sets. `http://www.grouplens.org/node/73`, October 2006.

[50] Gpgpu-sim web page.

[51] Pawan Harish and P. J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *Proceedings of the 14th International Conference on High Performance Computing*, HiPC'07, page 197–208, Berlin, Heidelberg, 2007. Springer-Verlag.

[52] Z. He and B. Hong. Dynamically tuned push-relabel algorithm for the maximum flow problem on cpu-gpu-hybrid platforms. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–10, April 2010.

[53] Blake A. Hechtman and Daniel J. Sorin. Exploring memory consistency for massively-threaded throughput-oriented processors. *SIGARCH Comput. Archit. News*, 41(3):201–212, June 2013.

[54] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 355–364, New York, NY, USA, 2010. ACM.

[55] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[56] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada,

S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. V. D. Wijngaart, and T. Mattson. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *2010 IEEE International Solid-State Circuits Conference - (ISSCC)*, pages 108–109, Feb 2010.

[57] Intel. Introducing intel many integrated core architecture, 2011.

[58] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez. A scalable architecture for ordered parallelism. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 228–241, Dec 2015.

[59] Hyeran Jeon, Gunjae Koo, and Murali Annavaram. CTA-aware prefetching for GPGPU. Technical Report CENG-2014-08, Dept. of Electrical Engineering, University of Southern California, October 2014.

[60] Adwait Jog, Onur Kayiran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. Orchestrated scheduling and prefetching for gpgpus. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 332–343, New York, NY, USA, 2013. ACM.

[61] David S. Johnson and Catherine C. McGeoch, editors. *Network Flows and Matching: First DIMACS Implementation Challenge*. American Mathematical Society, Boston, MA, USA, 1993.

[62] Rashid Kaleem, Sreepathi Pai, and Keshav Pingali. Stochastic gradient descent on gpus. In *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs*, GPGPU-8, pages 81–89, New York, NY, USA, 2015. ACM.

[63] Roozbeh Karimi, David M. Koppelman, and Chris J. Michael. Gpu road network graph contraction and sssp query. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '19, page 250–260, New York, NY, USA, 2019. Association for Computing Machinery.

[64] Ji Kim and Christopher Batten. Accelerating irregular algorithms on gpgpus using fine-grain hardware worklists. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, page 75–87, USA, 2014. IEEE Computer Society.

[65] Ji Kim, Christopher Torng, Shreesha Srinath, Derek Lockhart, and Christopher Batten. Microarchitectural mechanisms to exploit value structure in simt architectures. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 130–141, New York, NY, USA, 2013. ACM.

[66] K. Kim, S. Lee, M. K. Yoon, G. Koo, W. W. Ro, and M. Annavaram. Warped-preexecution: A gpu pre-execution approach for improving latency hiding. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 163–175, March 2016.

[67] Keunsoo Kim, Sangpil Lee, Myung Kuk Yoon, Gunjae Koo, Won Woo Ro, and Murali Annavaram. Warped-preexecution: A GPU pre-execution approach for improving latency hiding. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 163–175, March 2016.

[68] J. Kloosterman, J. Beaumont, M. Wollman, A. Sethia, R. Dreslinski, T. Mudge, and S. Mahlke. Warppool: Sharing requests with inter-warp coalescing for throughput processors. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 433–444, 2015.

[69] Milind Kulkarni, Martin Burtscher, Calin Casçaval, and Keshav Pingali. Lonestar: A suite of parallel irregular programs. In *ISPASS '09: IEEE International Symposium on Performance Analysis of Systems and Software*, 2009.

[70] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, page 211–222, New York, NY, USA, 2007. Association for Computing Machinery.

[71] Shailendra Kumar, Alok Misra, and Raghvendra Tomar. A modified parallel approach to single source shortest path problem for massively dense graphs using cuda. pages 635–639, 09 2011.

[72] Jaekyu Lee, Nagesh B. Lakshminarayana, Hyesoon Kim, and Richard Vuduc. Many-thread aware prefetching mechanisms for gpgpu applications. In

Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO '43, pages 213–224, Washington, DC, USA, 2010. IEEE Computer Society.

[73] Sangpil Lee, Keunsoo Kim, Gunjae Koo, Hyeran Jeon, Won Woo Ro, and Murali Annavaram. Warped-compression: Enabling power efficient GPUs through register compression. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 502–514, 2015.

[74] Yunsup Lee. Decoupled vector-fetch architecture with a scalarizing compiler. Technical Report UCB/EECS-2016-117, EECS Department, University of California, Berkeley, May 2016.

[75] Yunsup Lee, Rimas Avizienis, Alex Bishara, Richard Xia, Derek Lockhart, Christopher Batten, and Krste Asanović. Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 129–140, New York, NY, USA, 2011. ACM.

[76] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. Gpuwattch: Enabling energy optimizations in gpgpus. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 487–498, New York, NY, USA, 2013. ACM.

[77] David D. Lewis, Yiming Yang, Tony G. Rose, and Fan Li. RCV1: A new

benchmark collection for text categorization research. *J. Machine Learning Research*, 5:361–397, 2004.

[78] Ang Li, Gert-Jan van den Braak, Henk Corporaal, and Akash Kumar. Fine-grained synchronizations and dataflow programming on gpus. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 109–118, New York, NY, USA, 2015. ACM.

[79] C. K. Liang and M. Prvulovic. MiSAR: minimalistic synchronization accelerator with resource overflow management. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 414–426, June 2015.

[80] J. Liu, J. Yang, and R. Melhem. SAWS: synchronization aware GPGPU warp scheduling for multiple independent warp schedulers. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 383–394, Dec 2015.

[81] L. Liu, M. Liu, C. J. Wang, and J. Wang. Compile-time automatic synchronization insertion and redundant synchronization elimination for gpu kernels. In *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*, pages 826–834, Dec 2016.

[82] Yuxi Liu, Zhibin Yu, Lieven Eeckhout, Vijay Janapa Reddi, Yingwei Luo, Xiaolin Wang, Zhenlin Wang, and Chengzhong Xu. Barrier-aware warp

scheduling for throughput processors. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, pages 42:1–42:12, New York, NY, USA, 2016. ACM.

[83] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 6–6, Berkeley, CA, USA, 2012. USENIX Association.

[84] Pedro J. Martín, Roberto Torres, and Antonio Gavilanes. Cuda solutions for the sssp problem. In *Proceedings of the 9th International Conference on Computational Science: Part I*, ICCS '09, page 904–913, Berlin, Heidelberg, 2009. Springer-Verlag.

[85] Jiayuan Meng, David Tarjan, and Kevin Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance: Extended results. u.va. Technical report, 2010.

[86] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable gpu graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, 2012.

[87] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable gpu graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, page 117–128, New York, NY, USA, 2012. Association for Computing Machinery.

[88] U. Meyer and P. Sanders. Delta-stepping: A parallelizable shortest path algorithm. *J. Algorithms*, 49(1):114–152, October 2003.

[89] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kauffman, San Francico, CA, 1997.

[90] Naveen Muralimanohar and Rajeev Balasubramonian. CACTI 6.0: A tool to understand large caches. Technical Report HPL 2009-85, HP Laboratories, 2009.

[91] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. Cacti 6.0: A tool to understand large caches. *University of Utah and Hewlett Packard Laboratories, Tech. Rep*, 2009.

[92] 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 *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 308–317, New York, NY, USA, 2011. ACM.

[93] R. Nasre, M. Burtscher, and K. Pingali. Data-driven versus topology-driven irregular computations on gpus. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 463–474, May 2013.

[94] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM*

Symposium on Operating Systems Principles, SOSP '13, page 456–471, New York, NY, USA, 2013. Association for Computing Machinery.

[95] Konstantinos Nikas, Nikos Anastopoulos, Georgios Goumas, and Nectarios Koziris. Employing transactional memory and helper threads to speedup dijkstra's algorithm. In *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, page 388–395, USA, 2009. IEEE Computer Society.

[96] NVIDIA. Fermi architecture whitepaper, 2009.

[97] NVIDIA. Geforce gtx 1080 whitepaper, 2016.

[98] NVIDIA. Nvidia tesla v100 gpu architecture, 2017.

[99] NVIDIA. Nvidia profiler user guide, 2018.

[100] NVIDIA. Nvidia turing gpu architecture, 2018.

[101] NVIDIA. The api reference guide for nvgraph, 2019.

[102] NVIDIA. Tuning cuda applications for turing, 2019.

[103] H. Ortega-Arranz, Y. Torres, D. R. Llanos, and A. Gonzalez-Escribano. A new gpu-based approach to the shortest path problem. In *2013 International Conference on High Performance Computing Simulation (HPCS)*, pages 505–511, July 2013.

[104] Y. Oyama, K. Taura, and A. Yonezawa. Executing parallel programs with synchronization bottlenecks efficiently.

[105] Sreepathi Pai and Keshav Pingali. A compiler for throughput optimization of graph algorithms on gpus. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, page 1–19, New York, NY, USA, 2016. Association for Computing Machinery.

[106] Sreepathi Pai and Keshav Pingali. A compiler for throughput optimization of graph algorithms on gpus. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 1–19, New York, NY, USA, 2016. ACM.

[107] Darko Petrović, Thomas Ropars, and André Schiper. On the performance of delegation over cache-coherent shared memory. In *Proceedings of the 2015 International Conference on Distributed Computing and Networking*, ICDCN '15, New York, NY, USA, 2015. Association for Computing Machinery.

[108] Darko Petrović, Thomas Ropars, and André Schiper. On the performance of delegation over cache-coherent shared memory. In *Proceedings of the 2015 International Conference on Distributed Computing and Networking*, ICDCN '15, New York, NY, USA, 2015. Association for Computing Machinery.

[109] Darko Petrović, Thomas Ropars, and André Schiper. Leveraging hardware message passing for efficient thread synchronization. *ACM Trans. Parallel*

Comput., 2(4):24:1–24:26, January 2016.

[110] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, and et al. The tao of parallelism in algorithms. *SIGPLAN Not.*, 46(6):12–25, June 2011.

[111] Kun Ren, Jose M. Faleiro, and Daniel J. Abadi. Design principles for scaling multi-core oltp under high contention. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1583–1598, New York, NY, USA, 2016. ACM.

[112] X. Ren and M. Lis. Efficient sequential consistency in GPUs via relativistic cache coherence. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 625–636, Feb 2017.

[113] X. Ren and M. Lis. High-performance gpu transactional memory via eager conflict detection. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 235–246, Feb 2018.

[114] M. Rhu and M. Erez. Capri: Prediction of compaction-adequacy for handling control-divergence in gpgpu architectures. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 61–71, 2012.

[115] T. G. Rogers, D. R. Johnson, M. O'Connor, and S. W. Keckler. A variable warp size architecture. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 489–501, 2015.

[116] Sepideh Roghanchi, Jakob Eriksson, and Nilanjana Basu. Ffwd: Delegation is (much) faster than you think. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 342–358, New York, NY, USA, 2017. Association for Computing Machinery.

[117] Mohsen Safari and Ali Ebnenasir. Locality-based relaxation: An efficient method for gpu-based computation of shortest paths. pages 43–58, 10 2017.

[118] Cuda sdk 4.2.

[119] A. Segura, J. Arnau, and A. González. Scu: A gpu stream compaction unit for graph processing. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 424–435, 2019.

[120] Ankit Sethia, Ganesh Dasika, Mehrzad Samadi, and Scott Mahlke. Apogee: Adaptive prefetching on gpus for energy efficiency. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 73–82, Piscataway, NJ, USA, 2013. IEEE Press.

[121] Ori Shalev and Nir Shavit. Predictive log-synchronization. *SIGOPS Oper. Syst. Rev.*, 40(4):305–315, April 2006.

[122] Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve. Efficient GPU synchronization without scopes: Saying no to complex consistency models. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 647–659, New York, NY, USA, 2015. ACM.

[123] Abhayendra Singh, Shaizeen Aga, and Satish Narayanasamy. Efficiently enforcing strong memory ordering in GPUs. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 699–712, New York, NY, USA, 2015. ACM.

[124] Dhirendra Pratap Singh, Nilay Khare, and Akhtar Rasool. Efficient parallel implementation of single source shortest path algorithm on gpu using cuda. 2016.

[125] I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt. Cache coherence for GPU architectures. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 578–590, Feb 2013.

[126] James E. Smith. Decoupled access/execute computer architectures. In *Proceedings of the 9th International Symposium on Computer Architecture (ISCA)*, pages 112–119, 1982.

[127] James E. Smith. Decoupled access/execute computer architectures. In *Proceedings of the 9th Annual Symposium on Computer Architecture*, ISCA '82, pages 112–119, Los Alamitos, CA, USA, 1982. IEEE Computer Society Press.

[128] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and W-m Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing.

Technical Report IMPACT 12-01, Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaigne, 2012.

[129] M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. *SIGPLAN Not.*, 44(3):253–264, March 2009.

[130] G. G. Surve and M. A. Shah. Parallel implementation of bellman-ford algorithm using cuda architecture. In *2017 International conference of Electronics, Communication and Aerospace Technology (ICECA)*, volume 2, pages 16–22, April 2017.

[131] E. Vallejo, R. Beivide, A. Cristal, T. Harris, F. Vallejo, O. Unsal, and M. Valero. Architectural support for fair reader-writer locking. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 275–286, Dec 2010.

[132] Hao Wang, Liang Geng, Rubao Lee, Kaixi Hou, Yanfeng Zhang, and Xiaodong Zhang. Sep-graph: Finding shortest execution paths for graph processing under a hybrid framework on gpu. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, page 38–52, New York, NY, USA, 2019. Association for Computing Machinery.

[133] Kai Wang, Don Fussell, and Calvin Lin. Fast fine-grained global synchronization on gpus. In *Proceedings of the Twenty-Fourth International Confer-*

ence on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19, pages 793–806, New York, NY, USA, 2019. ACM.

[134] Kai Wang and Calvin Lin. Decoupled affine computation for simt gpus. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 295–306, New York, NY, USA, 2017. ACM.

[135] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. Gunrock: A high-performance graph processing library on the gpu. *SIGPLAN Not.*, 51(8), February 2016.

[136] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C. Miao, J. F. Brown III, and A. Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27(5):15–31, Sept 2007.

[137] H. Wong, M. M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying gpu microarchitecture through microbenchmarking. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pages 235–246, March 2010.

[138] Y. Wu, Y. Wang, Y. Pan, C. Yang, and J. D. Owens. Performance characterization of high-level programming models for gpu graph analytics. In *2015 IEEE International Symposium on Workload Characterization*, pages 66–75, Oct 2015.

[139] Ping Xiang, Yi Yang, Mike Mantor, Norm Rubin, Lisa R. Hsu, and Huiyang Zhou. Exploiting uniform vector instructions for gpgpu performance, energy

efficiency, and opportunistic reliability enhancement. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 433–442, New York, NY, USA, 2013. ACM.

[140] Shucai Xiao and Wu Feng. Inter-block gpu communication via fast barrier synchronization. pages 1 – 12, 05 2010.

[141] Yunlong Xu, Lan Gao, Rui Wang, Zhongzhi Luan, Weiguo Wu, and Depei Qian. Lock-based synchronization for gpu architectures. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '16, pages 205–213, New York, NY, USA, 2016. ACM.

[142] Yunlong Xu, Rui Wang, Nilanjan Goswami, Tao Li, Lan Gao, and Depei Qian. Software transactional memory for gpu architectures. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 1:1–1:10, New York, NY, USA, 2014. ACM.

[143] Yi Yang, Ping Xiang, Michael Mantor, Norman Rubin, Lisa Hsu, Qunfeng Dong, and Huiyang Zhou. A case for a flexible scalar unit in simt architecture. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 93–102, Washington, DC, USA, 2014. IEEE Computer Society.

[144] Yi Yang, Ping Xiang, Mike Mantor, and Huiyang Zhou. Cpu-assisted gpgpu on fused cpu-gpu architectures. In *Proceedings of the 2012 IEEE 18th Inter-*

national Symposium on High-Performance Computer Architecture, HPCA '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society.

[145] P.-C. Yew, N.-F. Tzeng, and D. H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Trans. Comput.*, 36(4):388–395, April 1987.

[146] A. Yilmazer and D. Kaeli. HQL: a scalable synchronization mechanism for GPUs. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 475–486, May 2013.

[147] Weirong Zhu, Vugranam C Sreedhar, Ziang Hu, and Guang R. Gao. Synchronization state buffer: Supporting efficient fine-grain synchronization on many-core architectures. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 35–45, New York, NY, USA, 2007. ACM.