

# **Allocation-oriented Algorithm Design with Application to GPU Computing**

A Dissertation

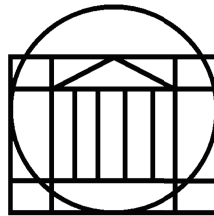
Presented to

the faculty of the School of Engineering

and Applied Science

at the

University of Virginia



In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Duane G. Merrill III

December 2011

## ABSTRACT

The wide data-parallelism of GPU processor design facilitates the execution of many concurrent, fine-grained tasks with unprecedented performance and efficiency. However, contemporary opinion regards GPU architecture as being poorly suited for many parallelizations that impose dynamic dependences between processing elements. This dissertation specifically attends to problems whose efficient solutions require *cooperative allocation*, i.e., their dependences stem from dynamic data placement within shared structures. The contribution of this thesis is a set of design patterns and reusable, tunable software primitives that foster the construction of cooperative, allocation-oriented parallelizations that are significantly faster, more efficient, more flexible, and more performance-portable than the state of the art.

Whereas concurrent CPU programs typically leverage fine-grained atomic operations for coordinating data placement, we demonstrate the advantages of *parallel prefix sum* as a high performance, bulk-synchronous alternative for cooperative allocation. We construct very efficient algorithms for global and local prefix sum by employing *flexible granularity coarsening* techniques for properly balancing serial and parallel workloads. The resulting efficiency gains permit *kernel fusion* techniques

whereby application-specific logic can be incorporated within our prefix sum constructs with little or no overhead.

To demonstrate the viability of our methods, we construct cooperative GPU implementations for a variety of parallel list-processing primitives including reduction, prefix scan, duplicate removal, histogram, and reduce-by-key. We evaluate their performance across a wide spectrum of problem sizes, types, and target architectures. To achieve performance-portability, we present a *policy-based autotuning* design idiom in which we leave implementation decisions having opaque performance consequences unbound within the program text. Finally, we showcase high performance implementations for two archetypal problems in computer science: sorting and breadth-first search (BFS). We achieve excellent performance for both genres, demonstrating multiple factors of speedup over prior parallelizations for GPU and conventional multi-core platforms.

## ACKNOWLEDGMENTS

To Jenny, my eternal confidante,  
without whose love, support, and companionship  
my purpose would surely be lost

To my parents, to my family,  
through their sacrifice and achievements  
my opportunities were possible

To my committee,  
for gladly suffering my impenetrable prose  
and helping me to better communicate

To Andrew, my advisor and mentor,  
for the means to improve myself,  
for letting my interests run on such a long leash,  
and for the wisdom that this is all just a bit of history repeating

To Kevin Skadron and the Lava Lab,  
for letting me monopolize their toys

To David Luebke,  
for plugging me in

To Kim Hazelwood,  
for supporting third-chances

To Mark Morgan,  
for showing me how to program  
and what Fred Brooks was writing about

To Michael, Nathan, and the NVR team,  
for access  
and the opportunity to have real-world impact

And to my friends and colleagues,  
Karp, Anh, Zach, Ben,  
Chris, Jen, Sam, Nate, Brian, Mike,  
Howie, Kar, Sal, Yan, and Avenish



## TABLE OF CONTENTS

Abstract .....	i
Acknowledgments.....	iii
Table of Contents .....	iv
List of Figures .....	vii
List of Tables .....	ix
List of Listings .....	x
Chapter 1 <i>Cooperative Allocation</i> .....	1
1.1    Introduction .....	1
1.2    Dissertation Roadmap .....	5
1.3    Cooperative Algorithm Design .....	6
1.3.1    Suitability of atomic operations.....	9
1.3.2    Suitability of iterative stencil application.....	12
1.3.3    Suitability of prefix sum .....	15
1.4    Contributions.....	17
1.4.1    Prefix Scan.....	17
1.4.2    Sorting .....	18
1.4.3    Graph Traversal .....	19
1.4.4    Design Idioms.....	20
1.5    Chapter Summary.....	21
Chapter 2 <i>GPU Machine and Programming Models</i> .....	22
2.1    Introduction .....	22
2.2    Abstract Machine Model.....	22
2.3    Programming Model .....	25
2.4    Performance Modeling and Analysis .....	27
2.5    Performance Pitfalls .....	29
2.5.1    Non-uniform memory access costs.....	29
2.5.2    Thread divergence .....	31
2.6    On the Suitability of GPU Architecture .....	32
2.6.1    Contention .....	32
2.6.2    Conclusions from workload and architectural studies.....	33
2.7    Chapter Summary.....	36
Chapter 3 <i>Granularity coarsening &amp; policy-based tuning</i> .....	38
3.1    Introduction .....	38
3.1.1    Performance benefits .....	40
3.1.2    Software reusability benefits .....	41
3.2    Tunable Concurrency .....	42

3.2.1	Expressing all available concurrency is counterproductive .....	42
3.2.2	The insulation of “expression” from “mapping” is counterproductive .....	44
3.3	Granularity Coarsening .....	48
3.3.1	CTA serialization.....	49
3.3.2	Thread serialization .....	53
3.3.3	Involvement with the type system .....	56
3.4	Tuning via the Type System.....	57
3.4.1	A simple example: data-parallel copy .....	58
3.4.2	Analysis of performance landscape across GPU architecture .....	63
3.4.3	Effectiveness of auto-tuning .....	67
3.5	Chapter Summary .....	69
Chapter 4	<i>Parallel Prefix Scan</i> .....	72
4.1	Introduction .....	72
4.2	Background .....	75
4.2.1	Prefix scan .....	75
4.3	Global CTA Decomposition.....	78
4.3.1	Scan-then-propagate .....	79
4.3.2	Reduce-then-scan.....	80
4.3.3	Two-level reduce-then-scan .....	81
4.4	Local Prefix Scan .....	82
4.4.1	Reduced-conflict Brent-Kung (RCBK) .....	82
4.4.2	Sequential-reduce-then-scan (SRTS) .....	85
4.4.3	SIMD Optimizations .....	86
4.4.4	Evaluation .....	88
4.5	Kernel Fusion .....	91
4.5.1	Segmented scan .....	92
4.5.2	Duplicate removal.....	93
4.5.3	Reduce-by-key .....	93
4.5.4	Histogram .....	94
4.6	Chapter Summary .....	95
Chapter 5	<i>Radix Sorting</i> .....	97
5.1	Introduction .....	97
5.2	Background .....	99
5.2.1	GPU sorting applications.....	99
5.2.2	Parallel sorting networks and the impracticality of output-oriented design. ....	100
5.2.3	The radix sorting method.....	102
5.2.4	Parallel radix sorting.....	103
5.2.5	GPU parallelizations of other sorting methods.....	106

5.3	Our Radix Sorting Strategy .....	107
5.3.1	“Multi-scan” prefix sum as allocation runtime.....	107
5.3.2	Multi-scan downsweep kernel operation .....	113
5.4	Optimizations .....	116
5.4.1	Composite scan.....	116
5.4.2	Early exit.....	117
5.4.3	Flexible tuning.....	118
5.5	Analytical Model.....	119
5.6	Evaluation.....	120
5.6.1	Configuration and methodology .....	121
5.6.2	Overall sorting rates.....	122
5.6.3	Resource utilization .....	123
5.6.4	Computational workloads.....	124
5.6.5	Memory workloads.....	125
5.6.6	Key diversity.....	126
5.7	Chapter Summary.....	128
Chapter 6	<i>Sparse Graph Traversal</i> .....	129
6.1	Introduction .....	129
6.2	Background .....	131
6.2.1	Sparse graph representation.....	132
6.2.2	Sequential BFS .....	133
6.2.3	Parallel BFS.....	134
6.3	Benchmark Suite .....	140
6.3.1	Graph Datasets.....	140
6.3.2	Logical Frontier Plots .....	141
6.4	Micro-benchmark Analyses .....	143
6.4.1	Isolated Neighbor Gathering .....	143
6.4.2	Isolated Status-lookup .....	148
6.4.3	Coupling of Gathering and Lookup.....	150
6.4.4	Concurrent Discovery.....	152
6.5	Single-GPU Parallelizations.....	157
6.5.1	Expand-contract (out-of-core vertex queue).....	157
6.5.2	Contract-expand (out-of-core edge queue).....	159
6.5.3	Two-phase (out-of-core vertex and edge queues) .....	161
6.5.4	Hybrid.....	161
6.5.5	Evaluation.....	162
6.6	Multi-GPU Parallelization.....	167
6.6.1	Design.....	167

6.6.2	Evaluation .....	168
6.7	Chapter Summary .....	170
Chapter 7	<i>Conclusion</i> .....	172
7.1	Summary .....	172
7.2	Limitations and Future Work .....	174
7.2.1	The CTA serialization idiom .....	174
7.2.2	Static metaprogramming .....	175
7.2.3	Sorting .....	176
7.2.4	Graph traversal .....	176
	References .....	177
	Index .....	186

## LIST OF FIGURES

Fig. 1.	Simple output-oriented stencil parallelizations .....	7
Fig. 2.	Run-length decoding having dynamic input dependences .....	8
Fig. 3.	Copy throughput as a function of decreasing global atomic workload .....	10
Fig. 4.	Copy throughput as a function of decreasing shared atomic workload .....	11
Fig. 5.	Example $O(n)$ global reduction .....	13
Fig. 6.	Example $O(n^2)$ sorting network .....	13
Fig. 7.	Example $O( V ^2)$ sparse graph traversal (BFS) .....	14
Fig. 8.	Search space dataflow for 4-queens problem .....	14
Fig. 9.	Example of exclusive prefix sum for computing scatter offsets for run-length decoding .....	16
Fig. 10.	Recursive Brent-Kung construction for prefix sum of 16 inputs .....	16
Fig. 11.	Example GPU processor organization .....	23
Fig. 12.	Bulk-synchronous kernel invocation and global data flow. ....	26
Fig. 13.	Alternative dataflow constructions for 8-element prefix sum .....	43
Fig. 14.	Example CTA decompositions for a data-parallel transformation .....	49
Fig. 15.	“Copy” kernel instruction overhead vs. CTA granularity .....	50
Fig. 16.	“Copy” kernel utilized bandwidth vs. problem size $n$ .....	50
Fig. 17.	Example CTA decompositions for global reduction .....	51
Fig. 18.	Cooperative instruction overhead vs. CTA granularity .....	52
Fig. 19.	Recursive, pair-wise parallelization of local CTA reduction .....	53

Fig. 20. Recursive, three-phase parallelization of local CTA reduction.....	54
Fig. 21. Alternative memory layouts for raking reduction in shared memory .....	56
Fig. 22. “Copy” kernel performance histograms of tuning configurations.....	61
Fig. 23. Performance histograms of tuning configuration “strength” .....	66
Fig. 24. Global reduction performance comparison .....	68
Fig. 25. Examples of prefix sum variants. ....	75
Fig. 26. Alternative constructions for 16-element prefix sum .....	77
Fig. 27. Example operation of a fully-recursive <i>scan-then-propagate</i> CTA decomposition .....	79
Fig. 28. Example operation of a fully-recursive <i>reduce-then-scan</i> CTA decomposition	80
Fig. 29. Example operation of a <i>two-level reduce-then-scan</i> CTA decomposition.....	81
Fig. 30. Example operation of Blelloch exclusive scan.....	83
Fig. 31. Example operation of our <i>reduced-conflict Brent-Kung</i> (RCBK) exclusive scan .....	84
Fig. 32. Example operation of our conflict-free <i>sequential-reduce-then-scan</i> (SRTS) scan .....	85
Fig. 33. The operation of an unrolled, divergence-free three-level SIMD “warpscan”...	87
Fig. 34. Global prefix sum throughput .....	88
Fig. 35. Global computational overhead.....	88
Fig. 36. Scan kernel utilized bandwidth.....	89
Fig. 37. Scan kernel computational overhead.....	89
Fig. 38. Kernel fusion of application code into scan kernels.....	91
Fig. 39. Segmented scan throughput.....	93
Fig. 40. Duplicate removal throughput.....	93
Fig. 41. Reduce-by-key throughput .....	94
Fig. 42. Histogram throughput.....	94
Fig. 43. The traditional split operation.....	103
Fig. 44. Procedures for distribution sorting .....	105
Fig. 45. A typical recursive CTA decomposition for prefix scan.....	109
Fig. 46. A fixed, two-level CTA decomposition for prefix scan .....	109
Fig. 47. GTX 285 memory wall.....	112
Fig. 48. Free cycles within a downsweep scan kernel .....	112
Fig. 49. Free cycles within a downsweep sorting scan/scatter kernel .....	112
Fig. 50. Intra-CTA multi-scan operation incorporating fused binning and scatter logic	114
Fig. 51. GTX285 keys-only and key-value pair radix sorting rates.....	122
Fig. 52. GTX285 saturated sorting rates for various radix bits $d$ .....	122
Fig. 53. Resource utilization for 32-bit key-value sorting .....	123
Fig. 54. Computational workload for 32-bit key-value sorting .....	124
Fig. 55. Memory workload for 32-bit key-value sorting .....	125

Fig. 56. Memory vs. compute workload ratios for individual sorting kernels.....	126
Fig. 57. Sorting performance with varying key entropy .....	127
Fig. 58. Sorting performance for key distributions with banded ranges.....	127
Fig. 59. Example CSR representation.....	132
Fig. 60. Example sparse graph, corresponding CSR representation, and frontier evolution for a BFS beginning at source vertex $v_0$ .....	135
Fig. 61. Sample frontier plots of logical vertex and edge-frontier sizes.....	142
Fig. 62. Alternative neighbor-gathering strategies .....	145
Fig. 63. Neighbor-gathering behavior.....	147
Fig. 64. Status-lookup behavior .....	149
Fig. 65. Comparison of lookup vs. gathering.....	150
Fig. 66. Comparison of isolated vs. fused lookup and gathering.....	151
Fig. 67. Example of redundant adjacency list expansion due to concurrent discovery .	152
Fig. 68. Actual expanded and contracted queue sizes without local duplicate culling..	153
Fig. 69 Redundant work expansion incurred by variants of our <i>two-phase</i> BFS implementation .....	154
Fig. 70 Percentages of false-negatives incurred by status-lookup strategies.....	156
Fig. 71 BFS traversal performance and workloads.....	162
Fig. 72. Sample <i>wikipedia-20070206</i> traversal behavior.....	163
Fig. 73. NVIDIA C2050 traversal throughput. ....	165
Fig. 74. Average multi-GPU traversal rates.....	169
Fig. 75. Multi-GPU sensitivity to graph size and average out-degree $d$ for uniform random graphs.....	170

## LIST OF TABLES

Table 1. Throughput limits of NVIDIA GPUs .....	28
Table 2. Max achievable DRAM bandwidth .....	62
Table 3. Corpus of tuning benchmarks .....	63
Table 4. Between-configs slowdown variance .....	65
Table 5. Within-configs slowdown variance .....	65
Table 6. Average bandwidth utilization of <i>all</i> 128MB tuning configurations.....	67
Table 7. Average bandwidth utilization of <i>best</i> 128MB tuning configurations .....	67
Table 8. Saturated 32-bit sorting rates .....	121

Table 9. Suite of benchmark graphs .....	141
Table 10. Single-socket performance comparison.....	164

## LIST OF LISTINGS

Listing 1. A straightforward kernel subroutine for CTAs to copy tiles .....	59
Listing 2. A tuning policy type for data-parallel copy .....	59
Listing 3. A generalized, policy-based kernel subroutine for CTAs to copy tiles of elements .....	60
Listing 4. Blelloch PRAM exclusive scan algorithm.....	83
Listing 5. The simple sequential breadth-first search algorithm.....	132
Listing 6. A simple quadratic-work, vertex-oriented BFS parallelization.....	134
Listing 7. A linear-work BFS parallelization constructed using a global vertex-frontier queue.....	134
Listing 8. A linear-work, vertex-oriented BFS parallelization for a graph that has been partitioned across multiple processors .....	138
Listing 9. GPU pseudo-code for warp-based, strip-mined neighbor-gathering.....	144
Listing 10. GPU pseudo-code for fine-grained, scan-based neighbor-gathering.....	144
Listing 11. GPU pseudo-code for a localized, warp-based duplicate-detection heuristic. .....	155

## Chapter 1

### *Cooperative Allocation*

#### 1.1 INTRODUCTION

*Efficiency.* We use the term to describe how well we spend our time and effort. Computer science is in many regards a study of efficiency. The algorithm encodes how to solve a problem, the machine automates the solution for us, and the efficiency of their pairing determines the practicality of the whole endeavor.

In order to improve machine efficiency, the current trend in processor architecture is to embrace wider parallelism. With an emphasis on more processing elements per chip, microprocessors can deliver increasingly higher throughput while maintaining energy efficiency. Contemporary graphics processors – GPUs – are at the leading edge of this trend. GPUs have evolved from fixed-function hardware into fully-programmable processors capable of general-purpose computation. In contrast to mainstream CPU architecture, GPUs provision tens of thousands of data-parallel threads.

GPUs have captured mindshare for their impressive peak throughput. Modern GPUs are capable of trillions of floating point operations per second (teraFLOPS) from a single microprocessor. They deliver very high throughput on parallel computations, but



require large amounts of fine-grained concurrency to do so. Unfortunately the physical design tradeoffs for such wide parallelism can penalize algorithms having irregular, dynamic, and cooperative behavior. GPU architecture is particularly sensitive to load imbalance among processing elements and serialization from contended accesses to memory.

This dissertation focuses on a particular class of problems for which GPUs are perceived as being poorly-suited. The common theme amongst these problems is that their solutions require *cooperative allocation*. We use this term to convey the notion that dynamic data placement within shared structures plays a central role in their operation. Parallel sorting, graph traversal, search space exploration, and duplicate removal are commonplace examples of this problem genre. These problems all expose abundant fine-grained concurrency, but their allocation (or relocation) behavior imposes global task dependences upon otherwise independent operations. Threads must cooperate with each other simply to determine where they can write their outputs.

Although the hardware is very efficient at scale, the traditional algorithmic techniques for solving these problems are often not. Parallel programming coursework typically illustrates multithreaded cooperation using fine-grained synchronization, specifically atomic read-modify-write operations. These mechanisms work by serializing updates within shared data structures. This type of serialization is particularly expensive for GPUs in terms of efficiency and performance. Although mutual exclusion may be suitable for the coarse-grained parallelism common to conventional CPU processors, it does not scale well to tens of thousands of threads. Furthermore, the occurrence of localized serialization between threads is typically more costly for GPU hardware. In

particular, many threads may be penalized when only a few are forced to wait for contended access.

As a result, straightforward implementations of many cooperative problems often demonstrate poor performance, particularly when compared to their CPU-based counterparts. The underlying goal of this work is to distill efficient allocation-oriented algorithms and design idioms for GPU processor architecture.

In particular, we focus on *prefix sum* as an alternative mechanism for data placement. Parallel prefix sum is an algorithmic primitive that can be used to compute space reservations for threads given their individual allocation requirements. Once these output offsets are known, threads can perform contention-free writes into shared structures. Cooperative allocation using local and global applications of efficient prefix sum is the central theme of this dissertation.

Prefix sum is also useful within optimization steps meant to improve GPU utilization by reorganizing sparse and uneven workloads into dense and uniform ones. Any mechanism for reorganization must be efficient: the opportunistic performance benefits must outweigh the additional overhead. When such optimizations become worthwhile, the GPU performance landscape for many irregular problems can be drastically improved.

However, prior implementations for GPU prefix sum were inflexible and inefficient. They existed at the global level to be invoked by the host CPU program, serving as black-box subroutines around which application-specific computation needed to be suspended and resumed. They were expensive because their inputs and outputs had

to be placed in off-chip global memory, making them suitable only for problems large enough to warrant enlisting the entire processor.

Furthermore, reusable implementations of prefix sum for small problems residing in on-chip shared memory were virtually non-existent. The scalability of many problems often hinges on hierarchical algorithms and data structures, and the ability to allocate within small structures shared by a localized group of threads is often as desirable as for large global ones.

In the absence of efficient and convenient GPU prefix sum, common practice for allocation-oriented parallelizations is to either (a) implement them using inefficient atomic operations; (b) avoid dynamic placement altogether by using work-inefficient data-parallel algorithms instead; or (c) simply not bother using the GPU at all. As processor design evolves towards wider data parallelism, all three would make poor usage of GPU-like throughput computing cores.

This dissertation is characterized by two overlapping research agendas. The first is the development of techniques for prefix scan that are several factors more efficient than prior work. The second is the practical application of these techniques and other idioms within a variety of algorithmic primitives and applications, substantially improving their performance.

In particular, we showcase our efforts by addressing two dynamic problems fundamental to computer science: sorting and breadth-first search (BFS). Both are common building blocks for more sophisticated algorithms. Both have abundant concurrency. Both elicit workloads representative of many problem genres: sorting encompasses list processing, partitioning, and ordering behavior; and BFS exemplifies

dynamic workload management and pointer-chasing. And both are simple enough that we can analyze their behavior in depth.

## 1.2 DISSERTATION ROADMAP

This dissertation is organized into the following chapters:

- Chapter 1 continues with a discussion of cooperative parallelization idioms for GPUs. We characterize the types of problems that benefit from efficient allocation-oriented design. Furthermore, we illustrate the inefficiency of atomic operations, the traditional mechanism for dynamic allocation.
- Chapter 2 describes the GPU abstract machine model, programming model, and performance pitfalls that guide the design patterns and idioms we develop in this dissertation. We also review the contemporary attitude regarding the use of GPUs for cooperative problems, underscoring the need for better solution strategies.
- Chapter 3 describes two prominent design idioms that we incorporate throughout this dissertation. We leverage *granularity coarsening* to balance the ratio of serial versus parallel workloads such that concurrency scales with processor width rather than problem size. *Policy-based tuning* allows us to tailor the granularity and other algorithm configuration options to match the specific input and target processor via the programming language’s type system.
- Chapter 4 presents our investigation of global and local algorithms for prefix scan<sup>1</sup>. By improving the efficiency of prefix scan past the point of being memory-bound, we leverage the *kernel fusion* design idiom to construct more sophisticated list-

---

<sup>1</sup> Whereas prefix sum specifically incorporates the addition operator, *prefix scan* is the generalization for arbitrary binary associative combining operators

processing primitives (e.g., segmented scan, duplicate removal, histogram, and reduce-by-key) with little or no extra overhead.

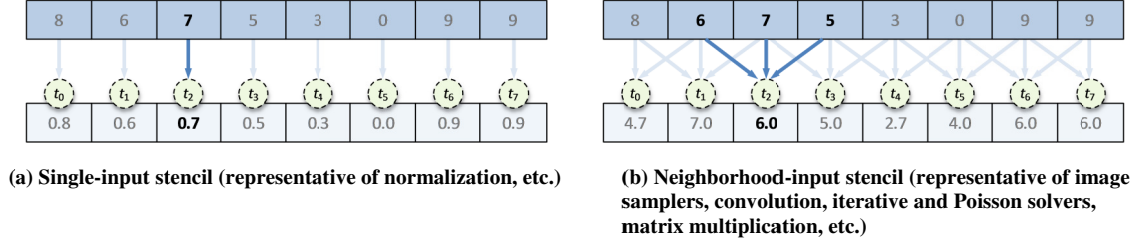
- Chapter 5 describes our parallelization of high performance radix sorting.
- Chapter 6 presents our parallelization of breadth-first search for sparse graphs.
- Chapter 7 concludes by summarizing the contributions and limitations of the work presented within this dissertation, commenting on opportunities for future research.

### 1.3 COOPERATIVE ALGORITHM DESIGN

GPUs are designed for data-parallel transformations, i.e., threads within kernel programs read input items from a global memory, process them independently, and write their results back to memory.

In particular, the GPU machine and programming models favor *stencil* kernel designs. This fundamental parallelization idiom is output-oriented. The stencil kernel is written such that each thread produces a specific item in the output dataset, regardless of what is computed by other threads. For stencil threads, the output location for each result is expressed as a static function of thread identifier. (E.g., “thread  $t_6$  produces  $output_6$ ”.)

Furthermore, a given stencil thread is typically not intended to reference the entire input. The specific inputs needed by each thread to compute the stencil operation are either (a) referenced directly via thread identifier, or (b) referenced indirectly by other directly-referenced inputs. As illustrated in Fig. 1, the stencil inputs for each thread are often regular and structured, belonging to a small finite neighborhood within the input dataset.

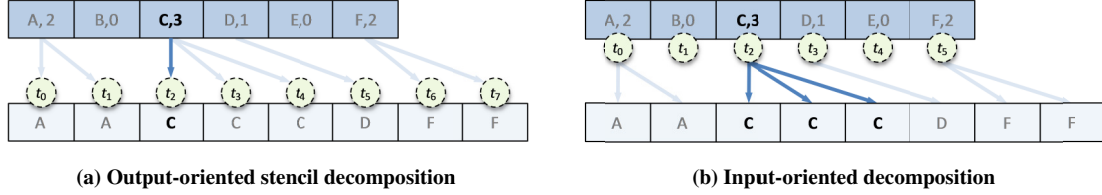


**Fig. 1.** Simple output-oriented stencil parallelizations. (Emphasis on dependences for thread  $t_2$ .)

Algorithm design becomes interesting when we want to perform data transformations having global input dependences, i.e., the result destined for each output location may depend on knowledge of any/every input element. Even the number of output items may be variable. Many useful primitives have this characteristic. Reduction, sorting, histogram, map-reduce, and duplicate-removal are commonplace examples.

However, a single stencil kernel is unsuitable for performing such transformations. Pure stencil kernels require independent computation with no cooperative dependences between threads. Such dependences may stem from:

- **Shared data dependences.** For example, the overall work for a global reduction problem is minimized when intermediate partial results can be shared between threads.
- **Allocation dependences.** For example, threads must coordinate when concurrently placing consecutive elements into a shared buffer. In this case, threads are not interested in *what* is computed by their peers, but rather *how many*. Cooperative allocation dependences between threads are an artifact of shared-memory architecture, and constitute the primary subject of this dissertation.



**Fig. 2.** Run-length decoding having dynamic input dependencies. (Emphasis on dependencies for thread  $t_2$ .)

Without cooperation, each output-oriented stencil thread would be required to read up to the entire problem. As an example, Fig. 2a illustrates a hypothetical output-oriented stencil decomposition for run-length decoding. Each input tuple contains a character and number of times it is to be repeated. The desired transformation is to expand these tuples into the output array. Each stencil thread must look at  $O(n)$  inputs to determine what should be written to its output location. This decomposition is further complicated because it requires prior knowledge of how many threads to launch, i.e., how many output items will be produced.

Instead, an input-oriented decomposition is often more intuitive for problems having global allocation dependencies. Fig. 2b illustrates the input-oriented decomposition for our run-length decoding example. Threads are mapped onto input items, and each thread is tasked with determining where its result should be placed. The input-oriented idiom has the advantage of not needing a preprocessing step to determine how many threads to invoke.

However, the input-oriented decomposition has the consequence of increasing the task granularity of individual threads. Threads are no longer responsible for producing exactly one item. In our example, arbitrarily large repeat-counts can lead to significant workload imbalance among threads. Furthermore, the shared-memory architecture

would still require each thread to inspect  $O(n)$  other inputs simply to determine where to begin its output.

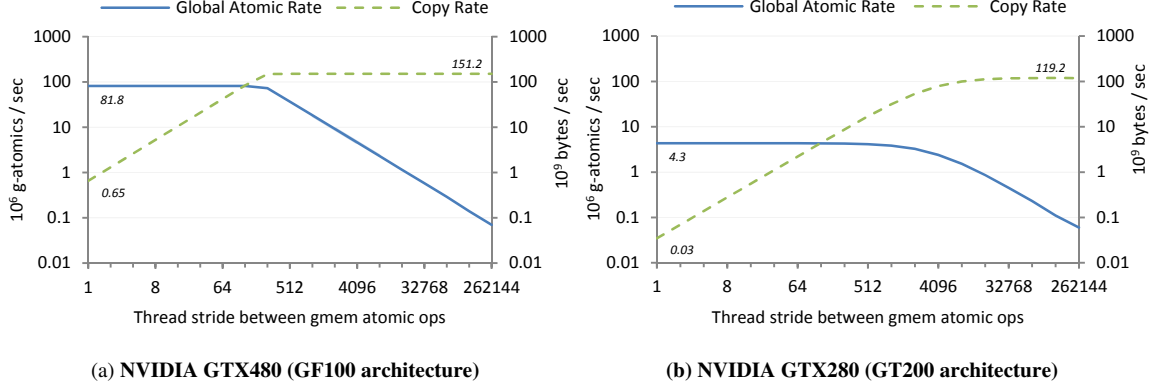
Regardless of input/output-oriented thread assignment, we would prefer to avoid the overall quadratic workload for cooperative problems that arises when each thread must independently inspect  $O(n)$  input items. In the following subsections, we describe the suitability of three strategies for doing just that: (1) input-oriented atomic updates; (2) output-oriented iterative stencil application; and (3) input-oriented prefix sum.

### *1.3.1 Suitability of atomic operations*

Fine-grained synchronization is attractive for solving many problems having allocation dependences. We can use atomic addition to trivially implement our input-oriented example of run-length decoding (with relaxed ordering of output-runs). With a global counter initialized to zero, threads can determine locations for their output-runs via atomic addition on the shared counter using their repeat-count as the addend. The operation returns the counter’s previous value which can then be used as a base scatter offset for the calling thread.

Although atomics often provide ease of implementation, they can incur dramatic processor underutilization. We illustrate atomic overhead by coupling it with a trivial “copy” kernel whose threads simply read and write their 32-bit elements from global input and output arrays. After loading an input, each thread performs an atomic addition on a shared counter in off-chip global memory to provide a corresponding allocation workload. We use 32M-element arrays, large enough to saturate the processor cores of the last three generations of NVIDIA GPU architectures (GF100, GT200, and G92).



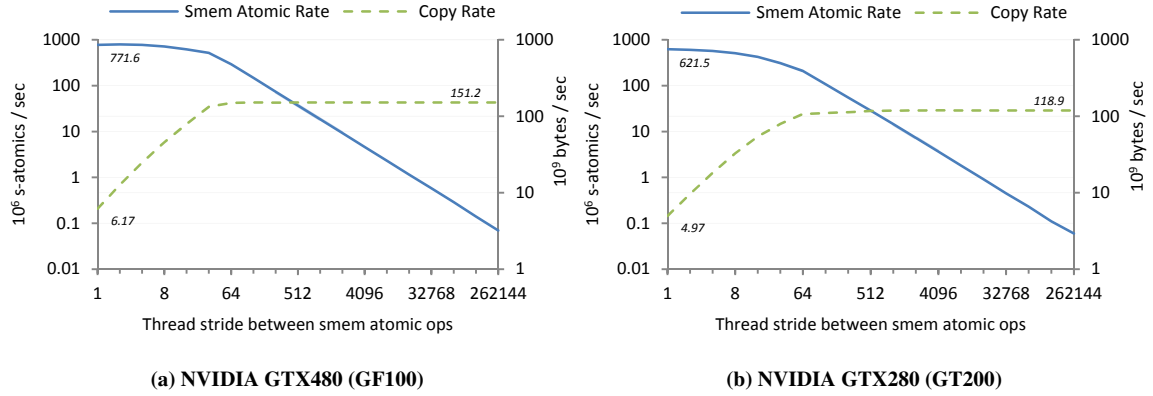


**Fig. 3.** Copy throughput as a function of decreasing global atomic workload. (Shown in log-log scale.)

The plots in Fig. 3 illustrate the response of overall copy throughput as we geometrically decrease the global atomic workload. This workload ranges from fine-grained synchronization (where every thread performs an atomic addition after loading its input) to only one atomic addition for every  $2^{18}$  threads. As the atomic workload decreases, we observe a transition from being rate-limited by atomic addition to being rate-limited by global memory bandwidth.

By progressively relieving the memory subsystem from atomic overhead, GTX480 copy throughput increases until it saturates at 151 GB/s (the peak achievable bidirectional bandwidth for the processor). However, the copy kernel only achieves 650 MB/s at the finest granularity of atomic synchronization, a 233x slowdown versus memory-saturated throughput. Fig. 3b reveals the situation is even worse for the older GTX280: fine-grained synchronization incurs a 3,970x slowdown.

On the other hand, atomic usage can be harmless when used sparingly. We observe the GTX480 is rate-limited at 82M global atoms/sec. As atomic workload is decreased past bandwidth saturation, the average thread latency no longer decreases correspondingly. Instead it is locked to the rate at which items can be copied through



**Fig. 4.** Copy throughput as a function of decreasing shared atomic workload. (Shown in log-log scale.)

memory. When atomic addition is no longer the limiting factor, measured atomic throughput begins to decrease along with decreased atomic workload. These sparse atomic operations have negligible overall performance impact: copy throughput remains at a plateau regardless of shrinking atomic workload.

Because processor cores can perform infrequent atomic reservation without penalty, we can leverage *software combining* [109] to batch many local requests into a single atomic operation. For example, threads local to a processor core would aggregate local counts into a single addend. On the GTX480, one thread can then perform a single global atomic reservation representing counts from 256 others without reducing overall throughput. As described further in Chapter 6, we make use of global atomics for aggregating allocation requests in our methods for GPU graph traversal.

The challenge for such “reservation-in-bulk” is computing the local aggregate addend. Unfortunately local atomic operations within on-chip shared memory are also not particularly efficient. Fig. 4 plots similar rate-limit transitions, this time with decreasing shared-memory atomic workloads. For both GTX480 and GTX280, we see

that copy throughput suffers a 24x slowdown when every resident thread performs a shared memory atomic operation.

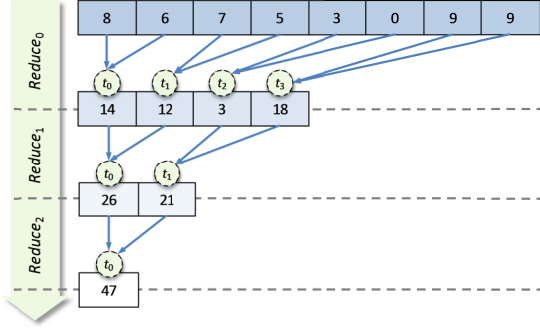
The implication is that fine-grained contention, in global or nearby shared memory, is simply too expensive for practical application. In Chapter 4, we show that we can construct functionality comparable to fine-grained shared-memory atomics using local prefix sum, and do so with zero slowdown under the cover of memory I/O.

Atomic operations have a second drawback in that access order is arbitrary. In our example, the output-runs may not appear in the same order as their corresponding inputs. This issue of unstable ordering precludes atomic operations from many problems. Many list-processing problems have stable ordering constraints where a given output allocation must be relative to the location of the input item (e.g., variants of array compaction, duplicate-removal, counting sort, etc.). Even when stable ordering is not required, an arbitrary ordering of output items can often destroy any inherent localities that were present in the input dataset.

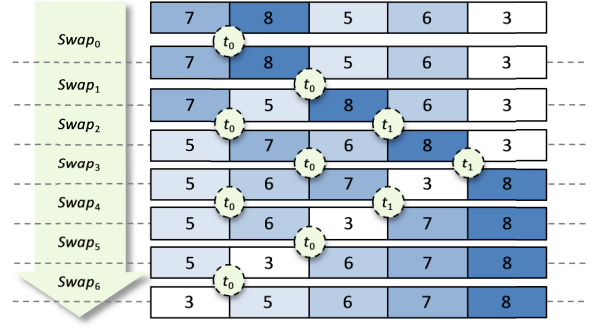
### ***1.3.2 Suitability of iterative stencil application***

In short, we prefer parallelizations that avoid contended updates to shared data structures. The *iterative stencil pattern* is a common, straightforward approach for orchestrating global dataflow without such contention. The idea is to string together a series of homogenous stencil kernels, effecting a global transformation from many iterations of neighborhood dataflow.

This static strategy works well for some problems. For global reduction, we can iterate simple stencils in which threads reduce pair-wise neighbors. As illustrated in Fig. 5, threads within each kernel invocation read pairs of consecutive inputs, reduce them,



**Fig. 5.** Example  $O(n)$  global reduction constructed from pair-wise stencil kernels.

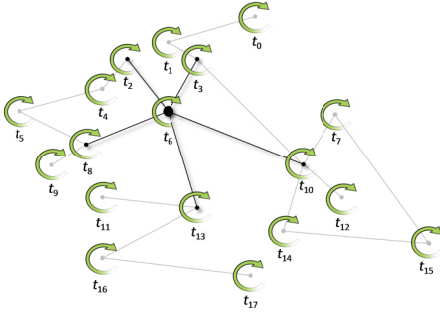


**Fig. 6.** Example  $O(n^2)$  sorting network constructed from pair-wise swapping stencil kernels.

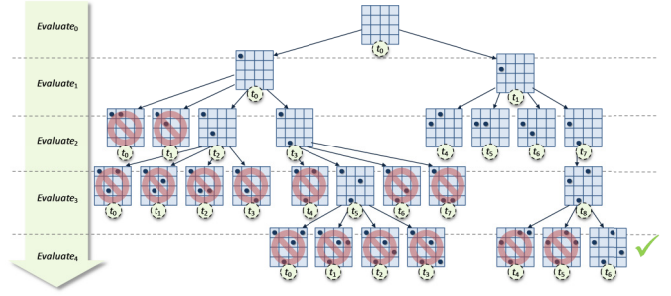
and then write out the results. This repeats until the last pair is reduced. The entire process requires amortized  $O(n)$  work, which is optimal for reduction.

Iterative stencil application is less effective for other problems. Although it may lead to functional solutions, the overall work performed can be asymptotically inferior to alternative parallelizations having dynamic allocation behavior. For example, *sorting networks* are sorting strategies where the sequence of comparisons is statically known in advance. As such, they can be implemented using static stencils that conditionally swap specific elements. Fig. 6 illustrates a simple parallelization where threads perform pair-wise neighbor-swapping.

Unfortunately this particular sorting network is isomorphic to bubble/insertion sorting and has  $O(n^2)$  work complexity. Methods based upon Batcher’s bitonic and 0-1 sorting networks [10] improve the complexity to  $O(n \log^2 n)$ . Although work-optimal  $O(n \log n)$  networks are known [3], they have extremely large practical constants due to embedded expander graphs within the dataflow circuit. At scale, these asymptotically and/or practically inefficient networks are not well-suited for problem sizes large enough to saturate the GPU.



**Fig. 7.** Example  $O(|V|^2)$  sparse graph traversal (BFS). Neighbor-expansion stencil kernels map one thread per vertex.



**Fig. 8.** Search space dataflow for 4-queens problem. Homogenous stencil application is unsuitable for arbitrary nested/recursive parallelism.

We would prefer work-optimal  $O(n \log n)$  comparison-based or  $O(n)$  position-based solutions, particularly for sorting problems large enough to saturate the GPU. Efficient methods like quicksort, sample sort, and radix sort all rely on dynamic partitioning. They are constructed from iterative passes that place variable numbers of keys into different output bins. This allocation behavior does not lend itself to output-oriented stencil designs whose threads rigidly produce specific output items. In Chapter 5, we describe the work-efficient parallelization of radix sorting using prefix sum.

Other problems suffer a similar fate. The breadth-first search of sparse graphs can also be implemented via repeated data-parallel stencil application. During each search time step, this approach launches one thread per vertex in the graph to see if it was visited in the previous iteration. This parallelization is illustrated in Fig. 7. When a thread discovers its vertex has been marked, it marks the neighbors of that vertex. For a graph having  $n$  vertices and  $m$  edges, the overall work complexity is quadratic  $O(n^2 + m)$ . Every vertex is inspected during every iteration, and there may be  $n$  BFS iterations in the worst case.

We would prefer  $O(n+m)$  linear-work graph traversal, which means we need a mechanism for tracking the dynamic *frontier* of discovered-but-unexplored vertices between BFS iterations. The implication is that GPU threads would need to cooperate in order to allocate and place newly-discovered vertices within a shared work queue. We describe work-efficient BFS parallelizations constructed from prefix sum in Chapter 6.

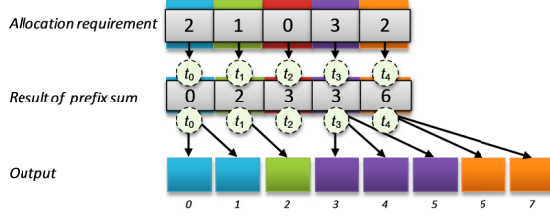
Finally, iterative stencil application completely fails for nested and recursive parallelism. Such problems exhibit arbitrary expansion and contraction of work items, implying global allocation dependences.

Consider the well-known  $n$ -queens “toy” search space exploration problem illustrated in Fig. 8. The problem is to identify valid configurations for placing  $n$  chess queens on an  $n \times n$  chessboard. The general solution strategy progressively places more queens on the chessboard, concurrently evaluating configurations having the same number of queens. The number of subsequent configurations produced by each evaluation is a dynamic quantity, unknown until runtime. A given configuration may expand into one, ten, a hundred more, or none at all. Unlike quadratic sorting and graph search, the output locations for each evaluation depend on the quantities of subsequent configurations produced by other concurrent evaluations.

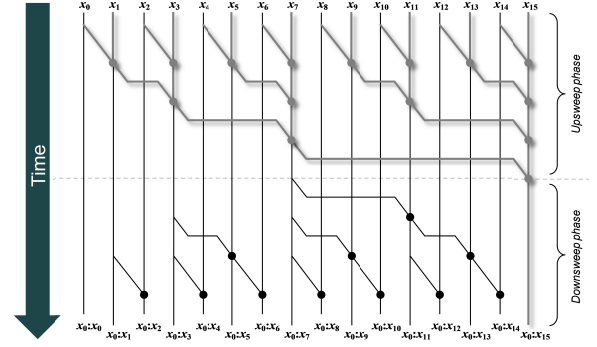
In short, the repetitive application of homogenous stencils can be suitable for managing well-structured global data dependences, but not for dynamic allocation dependences within arbitrarily-sized data structures.

### 1.3.3 Suitability of *prefix sum*

As a facilitator of cooperative allocation, prefix sum combines the best features of both atomic allocation and repeated stencil application. Like atomic updates, it facilitates



**Fig. 9.** Example of exclusive prefix sum for computing scatter offsets for run-length decoding. Input order is preserved.



**Fig. 10.** Recursive Brent-Kung construction for prefix sum of 16 inputs. Live values are carried on “wires” and addition operators are performed at pads.

input-oriented cooperative allocation with linear work. Like repetitive stencil application, it preserves input order and avoids contended updates.

As a list-processing primitive, prefix sum produces an output list where each element is computed to be the sum of the numbers occurring earlier in the input list. The utility of using prefix-sum for performing input-oriented, cooperative allocation is straightforward. We take the allocation requirements for each thread, put them in a list, and feed it to prefix sum. Prefix sum then computes the offsets for where each thread should start writing its output elements.

Fig. 9 illustrates prefix sum in the context of run-length expansion. In this example, the thread  $t_0$  wants to produce two items,  $t_1$  one item,  $t_2$  zero items, and so on. The prefix sum computes the scatter offset needed by each thread to write its output element. Thread  $t_0$  writes its items at offset zero,  $t_1$  at offset two,  $t_3$  at offset three, etc.

The history of prefix sum is rooted in circuit design for carry-over adders. Fig. 10 illustrates a 16-element Brent-Kung [20] circuit construction for inclusive prefix sum. This construction, along with many others, is statically recursive in nature. This implies

that we can orchestrate the global dataflow of prefix sum within GPU memory via the repeated application of stencil kernels having only neighborhood dependences.

Throughout this dissertation, we promote prefix sum as an efficient, high performance primitive for solving problems having dynamic allocation dependences.

## 1.4 CONTRIBUTIONS

This dissertation makes contributions in the following areas:

### 1.4.1 Prefix Scan

Prefix scan is a fundamental list-processing primitive for computing recurrence relations. In the form of parallel prefix sum, it is a useful mechanism by which concurrent threads can cooperatively compute scatter offsets for writing data into shared structures. Our prefix scan work makes the following contributions:

- **Parallelization strategies.** We present new variants of local scan and segmented scan that are 1.8x computationally-efficient than prior work. At the global level, we present a *reduce-then-scan* decomposition that requires 25% less global memory traffic and only a constant amount global storage for intermediate results. Our strategies fully leverage the GPU’s global memory bandwidth and will scale with future bandwidth improvements. We demonstrate 1.7x and 3.8x speedups over prior work [113] for global and segmented global scan, respectively.
- **Parallel primitives.** We have constructed BackForty [6], an open-source C++ library of fundamental data transformations for the NVIDIA CUDA parallel computing framework [34]. We employ our prefix scan techniques and design idioms to construct implementations of scan, segmented scan, reduction, list compaction, duplicate removal, and histogram. Furthermore, our implementations



of sorting and reduction-by-key allow users to express computation in the familiar map-reduce model of parallel task decomposition [38]. These algorithms demonstrate several factors of speedup over prior work [35, 113] for a diversity of problem sizes and data types.

#### *1.4.2 Sorting*

The need to rank and order data is pervasive, and many algorithms are fundamentally dependent upon sorting and partitioning operations. Our sorting work makes the following contributions:

- ***Parallelization strategy.*** We present a GPU parallelization for radix sorting passes that is constructed within a “multi-scan runtime” for computing multiple concurrent prefix sums, one for each partitioning bin. The granularity of our approach is more tunable than prior work, requiring memory traffic that is inversely proportional to the number of radix bits per digit. This provides flexibility for future improvements in computational throughput. We also describe a novel optimization for early termination that significantly improves performance for commonplace sorting problems whose key distributions have low variance.
- ***High performance.*** Our tunable implementation achieves multiple factors of speedup over prior GPU sorting implementations across all generations programmable NVIDIA GPUs. We demonstrate sustained sorting rates in excess of 1.2 billion 32-bit keys/sec and 342 million 64-bit keys/sec. These sorting rates are the fastest published for any fully-programmable microarchitecture. Put in context, contemporary CPU parallelizations achieve 240 million 32-bit keys/sec

[98] and reconfigurable FPGAs have demonstrated 250 million 64-bit keys/sec [73].

### 1.4.3 Graph Traversal

Breadth-first search (BFS) is a core primitive for graph traversal. It is representative of many computations whose memory accesses and workload distributions are irregular and data-dependent, and serves as a computational kernel within a number of benchmark suites. Our BFS work makes the following contributions:

- ***Parallelization strategy.*** We present a GPU parallelization for breadth-first search that performs an asymptotically-optimal linear amount of work. Our approach is the first to incorporate fine-grained parallel adjacency list expansion. We also introduce local duplicate detection techniques for avoiding race conditions that create redundant work. We also describe the first design for multi-GPU graph traversal.
- ***Empirical performance characterization.*** We present detailed analyses that isolate and analyze the expansion and contraction aspects of BFS throughout the traversal process. We reveal that serial and warp-centric expansion techniques described by prior work significantly underutilize the GPU for important classes of sparse graph datasets. We also show that, counter intuitively, the fusion of neighbor expansion and inspection within the same kernel often yields worse performance than performing them separately.
- ***High performance.*** We demonstrate excellent performance on a broad spectrum of structurally diverse synthetic and real-world graphs. Our implementation achieves traversal rates in excess of 3.3 billion and 8.3 billion traversed edges per

second (TE/s) for single and quad-GPU configurations, respectively. Put in context, contemporary parallel implementations for single-socket and quad-socket multi-core CPUs have demonstrated 0.7 billion and 1.3 billion TE/s respectively for similar datasets [2].

#### 1.4.4 Design Idioms

- ***Kernel fusion and the prefix sum “allocation runtime”.*** Throughout this dissertation, we advocate a kernel-fusion design idiom where we construct variants of global prefix sum, embedding within them problem-specific logic that will realize behavior for sorting, list compaction, graph traversal, etc. This is an inversion of the usual pattern for program composition promoted by Blelloch [17], i.e., where application logic calls down into prefix sum as a subroutine.
- ***Granularity coarsening.*** The data parallel programming paradigm encourages programmers to express all of the available concurrency inherent to their problem. This leads to substantial inefficiencies from redundant operations and unnecessary rounds of communication. Instead, our approach is to construct parallelizations where logical threads are a multiple of machine width, not problem size. We do this by increasing the granularity, i.e., amount of serial work performed by each thread, warp, and CTA.
- ***Templated “policy-based” tuning via the type system.*** The GPU programming model forces programmers to make implementation decisions that have opaque performance consequences. We show diverse and non-intuitive performance landscapes for thousands of program variants all implementing the same algorithmic strategies, yet parameterized with different configurations for thread

blocking, parallel widths and steps (task concurrency and granularity), cache modifiers for data movement, algorithm selection, etc. Our philosophy is to leave these decisions unbound within the program text, allowing the programmer (or the compiler, or the runtime) to specialize them for specific target microarchitectures and problem instances. By incorporating the formal type system into our tuning methodology, we are able to co-optimize application code alongside reusable library components.

## **1.5 CHAPTER SUMMARY**

Contemporary opinion is that GPU architecture is not well suited for problems that require dynamic and irregular data movement within shared data structures. They lack the practical atomic read-modify-write mechanisms that multithreaded algorithms have traditionally leveraged for cooperative allocation. Unfortunately many problems exist for which the only known efficient solutions require dynamic, fine-grained data allocation and/or relocation. Instead of performing contended updates for cooperative allocation, we focus on parallelization strategies that incorporate GPU-friendly algorithms for prefix sum.

## Chapter 2

### *GPU Machine and Programming Models*

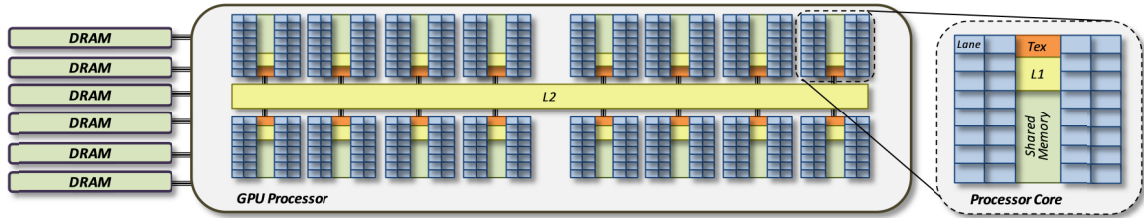
#### **2.1 INTRODUCTION**

This background chapter discusses the general perception of suitability of GPU architecture for cooperative workloads. The prevailing attitude is that GPUs are poor environments for problems having dynamic data movement, i.e., cooperative allocation within shared structures.

To provide important context for this discussion, we first review the abstract GPU machine and programming models, focusing on details pertaining to cooperation and contention. We also we discuss performance pitfalls that can lead to inefficient algorithm design, providing intuition for the design patterns and idioms we develop in this dissertation. Furthermore, we describe our philosophy of rate-limited performance analysis, the means by which we evaluate the practical efficiencies of our constructions.

#### **2.2 ABSTRACT MACHINE MODEL**

The “soul” of GPU architecture is very different from conventional multi-core CPU design. The abstract GPU machine model is fundamentally geared towards high-throughput (versus low-latency) computing. As a parallel architecture, it is designed to



**Fig. 11.** Example GPU processor organization.

process large quantities of concurrent, fine-grained tasks. GPUs are often classified as SPMD (single program, multiple data) processors in that tens of thousands of hardware-scheduled execution contexts, or *threads*, run copies of the same *kernel* program.

Fig. 11 illustrates contemporary GPU processor organization. High performance GPUs contain tens of processor cores, each comprising tens of homogeneous processing elements or data-paths called *lanes*. Processor cores use SIMD (single instruction, multiple data) and SMT (simultaneous multithreading) techniques to map threads of execution onto these lanes.

SIMD techniques are architecturally efficient in that one hardware unit for instruction-issue can service many data paths. However, GPUs are not exclusively SIMD machines. A pure SIMD design connotes a single instruction stream for the entire processor. SIMD has two scalability issues that prevent its application across thousands of threads: signal propagation delay and underutilization. Cross-chip signal propagation delay can result in undesirable timing skew amongst large numbers of SIMD elements that are intended to execute in lock-step. Underutilization is caused by thread divergence, i.e., when only a subset of threads takes a conditional branch, stalls on a memory access, is granted exclusive access to words in memory, etc. The remaining lanes sit idle when this occurs.

Instead, GPUs typically implement fixed-size SIMD groupings of threads called *warps*. The width of the warp (e.g., 32 threads) corresponds loosely to the number of SIMD lanes per processor core. Distinct warps are not run in lockstep and may diverge. Using SMT techniques, each processor core maintains and schedules amongst the execution contexts of many warps. The degree of GPU multithreading is often an order of magnitude higher than for conventional architectures. Instead of two or four instruction streams, GPU cores typically multiplex 30-50 warp contexts.

This style of SMT enables GPUs to “hide” latency by switching amongst warp contexts when architectural, data, and control hazards would normally introduce stalls. The result is a more efficient utilization of physical processing elements. Maximal instruction throughput occurs when the number of thread contexts is much greater than the aggregate number of SIMD lanes per processor. As such, the GPU’s throughput response to workload size makes the processor appear wider than it physically is, particularly at the point where performance saturation occurs.

The high degree of multithreading relieves GPU microarchitecture from latency-reducing techniques such as out-of-order execution, branch prediction, speculative execution, etc. As a result, the latencies of individual operations are comparatively much higher than on modern CPUs. Although this compromise is affordable for dependence-free computation, it compounds the expense of synchronization between threads, particularly serialization from fine-grained<sup>2</sup> atomic operations.

Communication between threads is achieved by reading and writing data to various shared memory spaces. The machine model exposes three levels of explicitly

---

<sup>2</sup> When describing an operation or task, we use the term *fine-grained* to connote that the amount of computation entailed is very small compared to the data needed to perform it. Often, the quantities of fine-grained operations and threads performing them are roughly equivalent (as opposed to batched or aggregated in some way).

managed storage that vary in terms of visibility and latency: per-thread registers, *shared memory* local to a collection of warps running on the same processor core, and a large *global memory* in off-chip DRAM that is accessible by all threads. Threads must explicitly move data from one memory space to another.

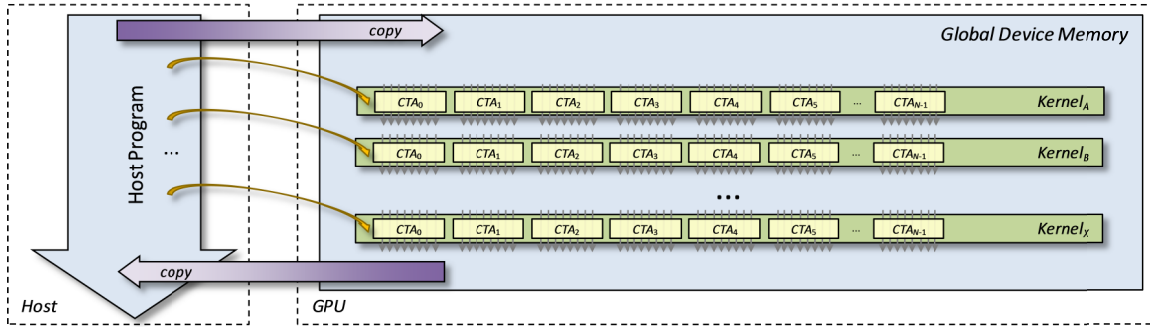
Unlike traditional CPU architecture, GPUs do not implement data caches for the purpose of maintaining the program's working set in nearby, low-latency storage. Doing so requires an expensive write-coherent cache hierarchy in which the last-level cache constitutes the majority of on-chip storage. Rather, the inverse is true for GPUs: the cumulative register file comprises the bulk of on-chip storage. A much smaller, read-only cache hierarchy often exists for the primary purpose of smoothing over irregular memory access patterns. The local exchange of intermediate computations amongst nearby threads must be explicitly managed in shared memory.

Furthermore, contemporary GPUs are not interrupt-driven. Instead of servicing and reacting to external events in their environment, their emphasis is on throughput-oriented processing. As such, they are not designed to run general-purpose operating systems. Rather, modern GPUs serve as peripheral accelerators on which CPU *host programs* can manage data and dispatch work. The physical GPU interface is via a motherboard or backplane interconnect such as PCI-express [94].

## 2.3 PROGRAMMING MODEL

Similar to SPMD programming models like MPI [86] and PVM [108], a kernel is an imperative function executed by a collection of logical threads. These logical threads are mapped onto hardware threads by a scheduling runtime, either in software or hardware.





**Fig. 12.** Bulk-synchronous kernel invocation and global data flow.

Thread behavior can be specialized by thread identifier, allowing threads to determine which portion of the problem to operate on.

Threads read their inputs from global device memory, perform some finite computation, and write their results back to global device memory. Global memory is only guaranteed consistent after kernel termination. As depicted in Fig. 12, the host orchestrates global data flow by sequentially invoking new kernel instances, each of which is presented a consistent view of the results from the previous kernel instance.

Grouping constructs help facilitate problem decomposition in a way that is convenient for mapping blocks of logical threads onto physical processor cores. The CUDA [34] programming framework exposes two levels of grouping: a *cooperative thread array* (CTA) that will be assigned to the same processor core, and a *grid* of homogeneous CTAs that encapsulates all of the threads for a given kernel. Thread execution can be further specialized by CTA identifier as well as CTA and grid dimensions.

Cooperation amongst threads is based on the bulk-synchronous model [116]: coherence in shared memory spaces is achieved through synchronization barriers. Different programmatic barriers exist for different memory spaces. CTA-wide barrier instructions exist for local shared memory. Global memory is guaranteed to be consistent

at the boundaries between sequential kernel invocations. An important consequence is that problems with global dependences often require multiple kernel invocations.

## 2.4 PERFORMANCE MODELING AND ANALYSIS

Empirical performance models are important for developing intuition regarding the actual costs of various aspects of program behavior. However, GPU models of algorithm performance can be difficult to construct and parameterize. Parallel performance models commonly describe runtime as a function of three parameters:

- 1) The problem input size  $n$
- 2) The number  $p$  of scalar tasks actively making progress
- 3) Additive, empirically determined duration-constants having units time/operation, e.g.,  $\text{time}_{load} + \text{time}_{mul} + \text{time}_{store}$

The idea is to compose a model of individual task time from component operations and then account for the mapping of  $n$  tasks onto a system with degree of parallelism  $p$ . Unfortunately this approach is difficult to parameterize for deeply multithreaded architectures. Both effective parallelism and the duration-constants are actually dynamic quantities.

For example, the effective parallelism  $p$  corresponds neither to physical processor cores nor to logical threads. Increasing the number of active warps per core from one warp to two will typically decrease overall runtime because the physical cores were previously undersubscribed. However, increasing the number of active warps per core from 31 to 32 will likely have negligible effect. The effective parallelism is determined by the processor saturation point, which is dependent on the exact numbers and types of architectural hazards experienced by the instantaneous workload.

Model	Architecture	Bidirectional DRAM Bandwidth ( $10^9$ bytes / sec)	Instruction Throughput ( $10^9$ scalar thread- instructions / sec)	Global Atomic Throughput ( $10^6$ atomics / sec)	Local Atomic Throughput ( $10^6$ atomics / sec)
GTX580	GF110	177	791	90.0	908
GTX480	GF100	159	672	81.8	771
Tesla C2050	GF100	128	514	67.5	591
GTX285	GT200	137	354	4.5	721
GTX280	GT200	125	311	4.3	623
9800 GTX+	G92	63	235	5.0	n/a

**Table 1.** Throughput limits of NVIDIA GPUs. Bandwidth is maximum-achievable from auto-tuning (§3.4.1). Global atomic throughput is to a single shared word. Local atomic throughput is to a single shared word per CTA.

Similarly, individual task latencies in saturated conditions are not what they would be if measured in isolation. For example, the latency measured in clock cycles for a local CTA-wide reduction will depend on how many other CTAs are resident on the processor core and what they are doing.

The difficulties of such empirical parameterization suggest an alternative model. Instead of models constructed from notions of parallelism and task duration, the high degree of multithreading allows us to construct simple rate-limited performance models for most saturating workloads. Rather than modeling individual task time, we simply decouple the component operations into separate workloads then determine which is rate-limiting the entire computation. In the copy example from Section 1.3.1, all computation on the GTX480 processor is limited by 82M atomics/sec regardless of the presence of load and store instructions. Table 1 lists various rate-limits for saturating workloads for the GPUs most frequently evaluated within this dissertation.

This dissertation primarily focuses on problems large enough to warrant the wide parallelism of the GPU. We use this simple approach throughout to characterize and analyze large-scale workloads. Unless otherwise specified, we generally consider the performance modeling of small, fleeting workloads to be the subject of future work.

## 2.5 PERFORMANCE PITFALLS

Efficient algorithm design requires an understanding of potential performance pitfalls. In general, *architectural mismatches* are a class of performance casualties that result when software components make assumptions that do not align with the structure of the system within which they operate [47, 107]. We describe two of the more prominent mismatches between the GPU machine and programming models that lead to inefficient implementations: variable memory access cost from SIMD access patterns; and thread divergence.

### 2.5.1 *Non-uniform memory access costs*

The programming model presents programmers with several physical memory spaces, each having a single method of accessing the words within. Without knowledge of the hardware, the programmer can only presume a uniform cost for references across each memory space.

GPU hardware often violates this presumption, causing a mismatch with the programming model. Performance is particularly sensitive to memory access patterns. A straightforward, naive treatment of memory can often lead to significant underutilization, a performance aspect not reflected in GPU programming languages [21, 62, 65, 81, 82, 89, 110]. One can often do better by designing algorithms around more complex memory models that accurately reflect the behavior of the underlying hardware. However, it is possible to worsen the situation if these assumptions do not match the specific hardware at hand.

Snyder’s notion of architecture mismatch [107] was concerned with tree-shaped memory topologies that resulted in different service latencies for different memory

locations. The problem is not quite the same for the GPU programming model. Rather, the mismatch stems from SIMD access patterns. Performance variances are less about access location and more about the specific pattern of references being made by other threads within the warp. We next describe how these variations can arise in both off-chip global memory and on-chip shared memory.

***Global memory access patterns.*** Global memory performance is affected by *coalescing*. For a SIMD instruction that accesses global memory, the individual accesses for each thread can be combined/bundled together by the memory subsystem into a single memory transaction if every reference falls within the same contiguous global memory segment. The performance discrepancies between coalesced and non-coalesced accesses can be as large as an order of magnitude. Bus transactions are on the order of 128 bytes, making it particularly wasteful if each thread induces a separate transaction for a single 4-byte memory reference.

To improve coalescing, we demonstrate the effective use of local prefix sum for reorganizing work among CTA threads. By locally rearranging tasks within the CTA, the threads within individual warps have better locality of reference. We show this optimization provides significantly better throughput for sorting and graph traversal.

***Shared memory access patterns.*** Local shared memory performance is affected by *bank conflicts*. Physical memories often aggregate individual cells into larger units of sequentially-accessible storage. Performance is highest when warps of threads make “broadside” accesses into these banks, i.e., each thread within the warp accesses a word within a different bank. Bank conflicts occur when threads within the same warp access

different words within the same memory bank, causing the individual accesses to be serialized by the hardware.

Parallel prefix sum is a cooperative problem that can be vulnerable to significant slowdown from excessive bank conflicts. In this work, we develop several variants of local prefix sum that avoid bank conflicts without additional instruction overhead. We use the programming language’s type system to abstract the rules for bank conflicts, providing flexibility and portability for our solutions.

### 2.5.2 *Thread divergence*

The GPU programming model presents an abstraction of concurrent, threaded execution. Each logical thread behaves as if it has its own program counter, register set, and stack space. This abstraction implies two properties of thread behavior. The first is that threads are presumably free to pursue their own independent path through the program. The second is that threads are scheduled fairly and generally proceed at a uniform rate.

The SIMD nature of the underlying hardware violates these presumptions, causing a mismatch with the programming model. In reality, logical threads are grouped into warps of execution. A single program counter is shared by all threads within the warp. Warps, not threads, are free to pursue independent paths through the kernel program.

To provide the illusion of individualized control flow, the execution model must transparently handle *branch divergence*. This situation occurs when a conditional branch instruction would redirect a subset of threads down the taken path, leaving the others to continue the fall-through path. Because threads within the warp proceed in lockstep fashion, the warp must necessarily execute both halves of the branch, masking off SIMD

lanes where appropriate. The two prevalent mechanisms for implementing such conditional execution are compiler-inserted instruction predicates and hardware-managed divergence stacks.

These mechanisms can lead to an inherently unfair scheduling of logical threads. In the worst case, only one logical thread may be active while all others effectively occupy the remaining SIMD lanes, yet perform no work. The GPU's relatively large SIMD widths exacerbate the problem. Branch divergence can impose an order of magnitude slowdown in overall computing throughput.

This SIMD aspect of GPU architecture has similar performance repercussions when a subset of the warp must stall for other reasons. Contended accesses from atomic operations are a relevant example. In contrast, prefix sum constructions are largely free of control flow divergence and contention, allowing our work to avoid control and data hazards that would otherwise lead to idle SIMD lanes.

## **2.6 ON THE SUITABILITY OF GPU ARCHITECTURE**

There is a general perception that GPUs are poorly-suited for cooperative parallelism, i.e., when dependences exist among tasks that will be performed by different processing elements. This is reflected in contemporary workload and architectural studies that have expressed concern over the GPU's ability to handle various forms of dynamic contention.

### ***2.6.1 Contention***

In the PRAM (parallel random-access machine) model of parallel computation [45, 49], cooperation is realized by the exchange of data through shared memory spaces. Parallel algorithm performance on real hardware is heavily influenced by the amount of serialization that arises from contention in shared memory spaces.

Serialization from contention may be implicitly introduced by the hardware or explicitly by the program. The physical networking between processing elements and memory imposes implicit contention: there will be hard limits on the number of simultaneous accesses to the same memory location. As previously described, GPUs incorporate various forms of combining networks having specific rules regarding transaction coalescing and bank conflicts.

Furthermore, cooperative algorithms explicitly introduce contention. They must synchronize concurrent accesses around updates to shared data in order to maintain a consistent view of global state. Barriers and atomic read-modify-write operations are two common mechanisms for such explicit synchronization. Coarse-grained barriers ensure all threads have progressed to the same point before letting any continue. Conversely, fine-grained atomics allow threads to manage a consistent view of shared data while unrelated tasks proceed unimpeded.

Nearly all modern processors support a form of atomic read-modify-write operation, e.g., test-and-set, fetch-and-add, compare-and-swap, load-linked/store-conditional, etc. Although atomics implemented by most GPU microarchitecture, their performance is incompatible with wide data parallelism. To illustrate, the simple data movement kernels we present in Section 1.3.2 incur 200-1700x slowdown with the introduction of atomic workloads.

### ***2.6.2 Conclusions from workload and architectural studies***

The lack of efficient fine-grained atomics contributes to the notion that GPUs are less practical for cooperative parallelizations. Without atomics, parallelizations requiring contended access would seem to require *fundamentally different algorithms* than those



that perform well on mainstream multi-core CPUs. This disparity is revealed by GPU benchmark suites. In comparing their heterogeneous Rodinia benchmark suite [27] with the multi-core PARSEC suite [14], Che *et al.* note that fine-grained synchronization figures much more prominently in the latter, despite principle component analysis showing that both sets of workloads cover similar application spaces [28].

“Different” often has a negative connotation for two reasons. The first is simply that it requires new effort. Cooperative parallelizations designed for CPUs are not suitable for GPU architecture. Alternative, atomic-free strategies must be developed

The second is that it carries an implied uncertainty, a non-obvious factor for how best to implement cooperation. The inefficiencies of existing methods are reflected in several ways. First, the performance speedups we demonstrate within this dissertation indicate the significant headroom for improvement. Additionally, the workload studies we review below report that GPU architecture is lacking in features that facilitate parallelizations with explicit contention. Such studies are important tools for distilling important architectural features.

In their rebuttal of GPU performance claims, Lee et al. express several criticisms relating to a perceived dearth of support for fine-grained thread cooperation [76]. Their survey of throughput problems reports lackluster GPU performance for many list, tree, and graph-based algorithms when evaluated alongside competent and comparable multi-core implementations and processors. With respect to contention, they advocate atomic support for histogram parallelization [119] and recommend specialized vector read-modify-write operations for improved atomics within SIMD lanes<sup>3</sup>. They suggest tree-

---

<sup>3</sup> We note that histogram is fundamentally a counting problem. The amortized contention within counting networks has been provably shown to be significantly lower than with single-variable shared counters [44].

structured search [70] would benefit from a fine-grained ability to expand work among SIMD lanes, i.e., to cooperatively enlist nearby threads for related tasks<sup>4</sup>. They observe the radix sorting passes of prior work [98] to incur excessive instruction overhead from inefficient prefix sum and speculated that GPU SIMD width is too wide. They advocate a coherent cache hierarchy for improved cross-core communication using atomics. Finally they recommend hardware accelerated task queues for fine-grained workload management, pushing the burden of contention from software into hardware.

With regard to parallel graph computation, Bader et al. posited fine-grained atomic operations as a critically important architectural feature for task and status management [8, 7]. Hong et al. have suggested the absence of efficient GPU atomics precludes the construction of shared queues needed for work-optimal graph traversal [63, 64]. Instead, they advocate a quadratic-work method for BFS that avoids contention altogether, a method only suitable for a narrow regime of sparse graphs having small constant diameter. Furthermore they only recommend the GPU for select traversal phases having abundant bulk concurrency.

With the exception of the linear-work BFS parallelization by Luo et al. [79], all prior published GPU implementations have implemented the quadratic method to avoid the challenges of dynamic queue management. In fact, Hussein *et al.* performed quadratic BFS on the GPU largely to avoid the cost of transferring the GPU-resident graph to the CPU just for BFS and then back again [66].

A common criticism from GPU application benchmarking efforts is that performance would often be improved by fine-grained task and data reorganization for improved balance and memory reference locality, an optimization not trivially

---

<sup>4</sup> In Chapter 6, we demonstrate cooperative enlistment for BFS using efficient local prefix sum.

implemented without atomics. Che *et al.* [28], Bakhoda *et al.* [9], and Kerr *et al.* [69] show that many benchmark workloads exhibit low *activity factor* (the average fraction of thread that are active at a given time) and low *memory efficiency* (the fraction of explicit warp accesses versus the actual number of memory transactions). These statistics indicate irregular control flow and memory references, both of which are the result of sub-optimal mappings between threads and data items.

With a perfect mapping of data and threads, the analyses by Zhang *et al.* indicate several factors of potential speedup for many benchmark applications [122]. Their G-Streamline framework is able to achieve some of this speedup via dynamic data reordering and job swapping between threads. The difficulties of fine-grained thread cooperation, however, led them to a pipelined implementation where the CPU handles the details of these transformations in advance. This reflects the notion that fine-grained data and job reorganization is impractical within the GPU itself.

As a result of these studies, there is a strong feeling that GPUs are not well-suited for parallel algorithms having dynamic data structures, dynamic workloads, and dynamic access patterns. Unfortunately, the known work-efficient algorithms for many important problems often have one or more of these characteristics. The constructions we describe within this dissertation serve as existence proofs that many of these concerns can be adequately addressed using counting networks, i.e., variants of prefix sum.

## 2.7 CHAPTER SUMMARY

The GPU is capable of efficiently executing large quantities of concurrent, ultra-fine-grained tasks. It derives much of its efficiency from SIMD hardware where a single instruction stream drives the same computation on multiple data elements. The high-

throughput philosophy of GPU architecture engenders wide SIMD, heavy multithreading, and long instruction latencies. These characteristics are particularly problematic for the dynamic, fine-grained synchronization mechanisms that multithreaded algorithms have traditionally leveraged for making coherent updates to shared memory.

Without practical mechanisms for atomic read-modify-write operations, the prevailing attitude is that GPUs are poor environments for dynamic data movement. Furthermore, workload studies have revealed that the opportunistic rearrangement of tasks and/or data would significantly reduce the burden of the architecture's restrictive access patterns and sensitivity to load imbalance. As a result, alternatives to atomic operations (such as prefix sum) have significant opportunity to improve the performance of parallelizations having allocation dependences.

## Chapter 3

### ***Granularity coarsening & policy-based tuning***

#### **3.1 INTRODUCTION**

Parallel programming is difficult. We can generalize the inherent challenges of parallel programming as stemming from two related sources: *expressing* parallelism, and *mapping* the expression of parallelism onto real hardware. The former encapsulates the creative aspects of devising and authoring a clean, concise, and correct description of concurrent tasks. The latter comprises the practical aspects of compiling and scheduling such descriptions of computation and data movement onto the underlying hardware for efficient execution.

The twin burdens of expression and mapping have historically fallen separately upon the shoulders of the programmer and the compiler/runtime, respectively. For sequential programs, compilers have largely succeeded in providing performance-portability across variations in microarchitecture, and have done so without explicit guidance from the programmer.

However, the effectiveness of this arrangement is unlikely to continue as contemporary processor architecture embraces ever-increasing parallelism. To achieve

performance-portability across diverse problem types and microarchitecture variants, we argue that parallel programs should incorporate *flexible granularity coarsening*, a sliding scale of parallel versus sequential computation. This allows the expensive aspects of communication and the redundant aspects of data parallelism to scale with the width of the processor rather than the problem size. We show this idiom to be critically important for obtaining good performance, particularly for GPU parallelizations that are cooperative in nature. In this chapter, we describe our philosophy of expressing tunable concurrency with an eye toward producing good performance across a diversity of GPU hardware and problem instances.

The requirement for flexible granularity (or flexible program-composition in general) complicates the manner in which we express concurrency. The program must be expressed in terms of both serial and concurrent behavior. We want to leave unbound both: (1) the number of steps each phase is to be run; and (2) the width of parallelism for each phase. Our approach incorporates aspects of metaprogramming, i.e., programmer effort is split among two aspects: (1) expressing the *template program*, i.e., a general description how the target machine is to perform its computation; and (2) expressing the *metaprogram*, i.e., rules and guidance for the compiler to follow when mapping the template program onto a specific problem and processor.

In this fashion, we can author the “general shape” of an implementation, leaving many of the performance-sensitive details unbound. Our approach uses parametric *policy types* that describe how the compiler should expand, couple, and select from various phases of sequential and parallel computation. In general, we use such tuning policy to

insulate both the programmer and the program text from implementation details having opaque performance consequences.

### *3.1.1 Performance benefits*

Metaprogramming allows us to explore the space of reasonable tuning policies, evaluating the performance of thousands of alternative program specializations. From the perspective of the programmer, it is a relief to be freed from many of the tuning decisions that are necessary to concretize a cooperative parallelization, yet are largely opaque in terms of their performance impact for a target processor that may not yet even be known.

To demonstrate, we expend the programming effort to implement only a single, generic high-level implementation for the following primitives: parallel copy, reduction, scan, and reduce-by-key. We then explore the tuning space of these four problems for a variety of data types and problem sizes across three generations of NVIDIA GPUs (GF100, GT200, and G92). Our results show:

- A large performance variance among reasonable specializations (which programmers could be expected to implement explicitly)
- That we can identify specializations that maximally utilize the underlying processor for many combinations of problem type and architecture
- That the highest performing specializations are different for distinct architecture versions, data types, and problem sizes
- That no single specialization for a given problem performs very well across all data types, problem sizes, and architectures

### 3.1.2 *Software reusability benefits*

Furthermore, our tuning approach dramatically improves software reusability. Software reuse is a critical aspect of good software development practice. However, GPU computing has received criticism for lack of software reuse [91]. The current trend in GPU library development is to provide developers with repositories of high-level data transformations (e.g., global reduction, sort, etc.) that can be invoked by sequential code on the host platform.

While these “host-side” primitives unburden the programmer from writing any parallel device software, libraries of “device-side” subroutines do not exist. For example, there are no collections of device subroutines for performing local reduction or local prefix sum within a CTA. This is an aspect of GPU software development that has heretofore been neglected.

We believe the dearth of reusable software components for constructing GPU kernels corresponds to a lack of performance flexibility. The performance of kernel code can be significantly affected by problem type, problem size, and specific GPU processor architecture. There is little value in providing libraries of reusable device subroutines that cannot be tailored for the specific problem and processor at hand.

Our explicit use of the type system for template metaprogramming provides us with the interface flexibility needed for software reusability as well as performance tuning. Reusable subroutines can be co-optimized with the enclosing kernel source. All of the kernels described within this dissertation are composed from our BackForty library of reusable, tunable device subroutines for common CTA activities (e.g., workload management, data movement, variants of local reduction and scan, etc.) [6].



### 3.2 TUNABLE CONCURRENCY

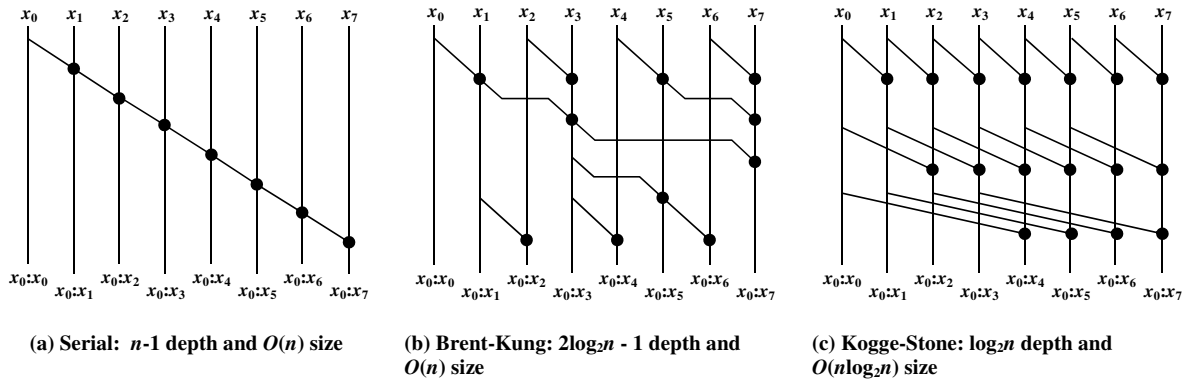
In this section, we further describe our perspective on the difficulties of parallel programming and the necessity for design idioms that facilitate the flexible coupling of diverse parallel and serial algorithmic phases. In particular, we focus on two prevalent attitudes regarding parallel software development that we feel are ultimately counterproductive:

- It is simpler (and thus preferable) to express all available concurrency within software.
- It is simpler (and thus preferable) to insulate the expression of such concurrency from the process of efficiently mapping it onto the underlying hardware.

#### 3.2.1 *Expressing all available concurrency is counterproductive*

Imperative algorithms for asynchronous, multithreaded models of computation are notoriously hard to construct, prove correct, maintain, and debug. The inherent non-determinism often leads to unanticipated interactions that are difficult to diagnose. Over the years, many programming models, languages, and APIs have been designed with the intent of simplifying the expression of parallel algorithms.

Many of these abstractions are designed for the program to specify *all available concurrency*. For example, SISAL [83], MultiLisp [54], and VHDL [67] are well-known declarative languages for expressing data dependences. These dependences dictate the global flow of computation, and all independent operations can proceed in parallel. In a similar vein, OpenMP [36], CUDA [34], and map-reduce [38] are examples of popular imperative paradigms for specifying data-parallel operations to be performed on every data element. The abstract GPU machine model supports this idiom through thread



**Fig. 13.** Alternative dataflow constructions for 8-element prefix sum

virtualization, i.e., the decoupling of logical threads from hardware threads. Programmers are encouraged to construct data-parallel task decompositions that instantiate a unique logical thread for every data item.

This idiom of parallel expression simplifies many decisions for the programmer. It allows them to remain oblivious to hardware details and focus on encoding a single parallelization that simply expresses the smallest granularity of concurrent tasks.

However, this style of task decomposition has important consequences for cooperative problems. When logical threads scale with input size, so does the amount of communication through memory. Communication between logical threads often results in the same data being loaded back into registers on the same processor core, yet at the expense of many clock cycles and costly synchronization for correctness. We would prefer not to move such data at all. This implies that communication overhead should scale with physical processing elements, not problem size.

Furthermore, much of the instruction workload also scales with logical threads. Local computation within a CTA typically involves computing conditional predicates, performing offset calculations, initializing local variables and shared memory, etc. Many

of these operations are identical across CTAs. For example, thread  $t_i$  in one CTA is likely to have the same activation schedule and access the same shared memory locations as thread  $t_i$  in all other CTAs. These identical instructions are effectively redundant when they are ultimately executed on the same SIMD lanes. When the number of CTAs scales with problem size, this redundant computation does as well.

We can apply the idiom of *granularity coarsening* to reduce the presence of unnecessary computation and communication. We do this by increasing the granularity, i.e., amount of serial work performed by each thread, warp, and CTA. Our goal is to construct parallelizations where logical threads are a multiple of machine width, not problem size. As we further illustrate in Section 3.3, granularity coarsening significantly improves our ability to efficiently map the implementation onto various underlying hardware.

### 3.2.2 The insulation of “expression” from “mapping” is counterproductive

A clear, concise, elegant, and correct program is not particularly useful if it does not map well to the specific processor it is to be executed on. For sequential computation, the responsibility of constructing this mapping has traditionally fallen on the shoulders of the optimizing compiler with little to no visibility from the program. For parallel programs, a philosophy of complete insulation from the mapping process is less useful for achieving both portability and performance. At worst, it is counterproductive. In this section we discuss three aspects of mapping that would benefit from explicit guidance from the program: algorithm selection, scheduling, and variable concurrency.

**Algorithm selection.** For many problems, no single parallelization is best across all processor architectures and input sizes, types, and data. As discussed in Chapter 2,

different processor genres can require fundamentally different algorithms for solving the same problem, and the preference of one algorithm over another can depend on problem size and data type [5].

In an ideal world, we would like our compilers to be able to: (a) detect that a program implements a particular algorithm; and (b) discover an alternative parallelization that might be better suited for the underlying hardware. For example, we might want to detect that the program expresses a work-efficient parallelization of prefix sum having depth  $2\log_2 n$  (e.g., Fig. 13b) and replace it with a shorter, work-inefficient construction having depth  $\log_2 n$  (e.g., Fig. 13c) when the problem size drops below the warp width.

In an ideal world, we would like our compilers to be able to: (a) detect that a program implements a particular algorithm; and (b) discover an alternative parallelization that might be better suited for the underlying hardware. However, it is extremely difficult for compilers to synthesize alternative, fungible parallelizations, particularly for problems having non-trivial data dependences. In the general case, it is impossible [95, 96].

This motivates an alternative paradigm having a less opaque relationship between the expression of the parallel program and its compilation, e.g., one in which the compiler is provided with algorithmic alternatives and rules for guiding selection among them based upon problem type and target processor.

***Scheduling and resource management.*** The challenges of mapping programs onto parallel hardware extend beyond algorithm selection and choice. Even when the basic outline of an algorithm is a good fit for the underlying machine model, an efficient scheduling of tasks on one processor can result in significant underutilization on another. This is exacerbated on contemporary GPUs, where the hardware resources provisioned

for each thread (e.g., registers, shared memory, etc.) are intimately intertwined with co-scheduling, i.e., thread blocking.

Logical threads are dispatched onto processor cores by CTA. The number of resident, active CTAs per core is limited by the core's resources, namely the aggregate register file, local shared memory, and scheduling contexts. For example, the NVIDIA GF100 architecture provisions 32K 32-bit registers, 48KB shared memory, and scheduling resources for 1,536 threads per core. The configuration space for thread blocking is quite large, including such alternatives as:

- a) Three resident 512-thread CTAs (1536 threads/core), 16KB shmem per CTA, 21 registers per thread
- b) Six resident 128-thread CTAs (768 threads/core), 8KB shmem per CTA, 42 registers per thread
- c) Eight resident 64-thread CTAs (512 threads/core), 6KB shmem per CTA, 64 registers per thread

What should the program specify? The performance consequences are opaque. More resident threads does not necessarily imply greater throughput if computation or memory is already saturated. More independent CTAs can provide a greater diversity of instantaneous thread behavior for better core utilization. The same diversity, however, can be harder on read-only cache hierarchies. More CTAs also reduces the amount of shared scratch available to each for local cooperation.

Furthermore, the complex relationships between these details explicitly affect the imperative behavior of threads, e.g., the locations in shared memory that threads must read and write from. On one hand, we can encode these relationships directly within our

kernel programs, having each thread dynamically compute many of the derivative details it will need (e.g, offsets, strides, etc.) from parameters supplied by the host program. Alternatively, we can encode these relationships statically using the type system, allowing the much of this information to be computed at compile time.

The compiler can do a much better job of code generation if the thread blocking information can be specified at compile time. Without knowledge of the desired number of resident threads per core, the compiler must perform conservative register allocation under the assumption that the core may be fully occupied with threads. By specifying a combination of desired CTA occupancy and CTA size that is below the maximum thread residency for the core, the compiler can allocate more registers per thread. This can significantly improve performance by minimizing costly spills and lowering dynamic instruction overhead via common subexpression elimination.

In our evaluations, we illustrate large performance variances among reasonable thread blocking configurations that programmers could be expected to implement explicitly. Unfortunately there is little information to guide selection from the space of diverse, yet functionally-equivalent alternatives. Without precise analytical models for complex and data-dependent scheduling interactions on specific target architecture, empirical performance tuning is a compelling approach for optimizing hierarchical thread blocking.

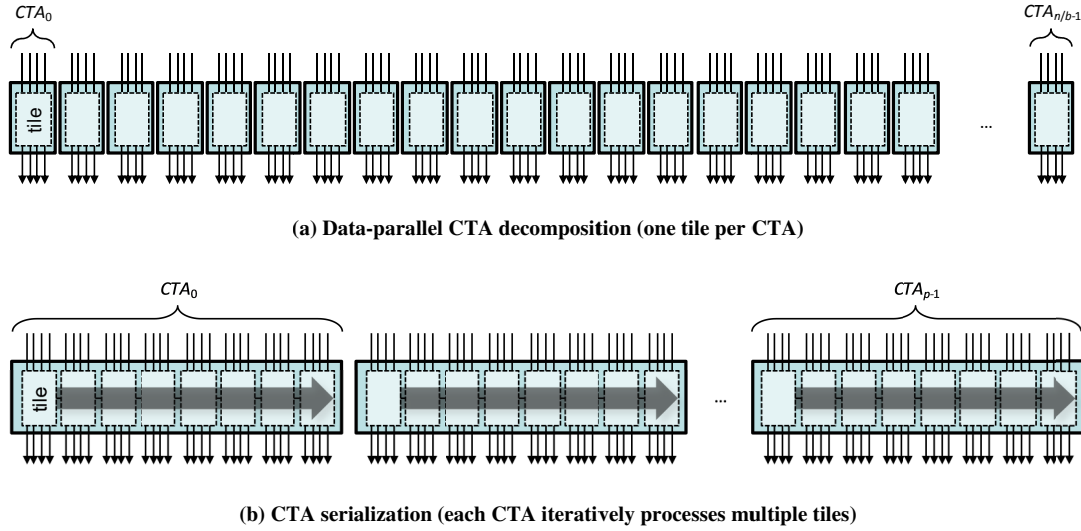
***Variable concurrency.*** Parallel programming adds an important facet to performance tuning: the amount of concurrency expressed. As a deeply multithreaded architecture, GPUs are designed to be saturated with concurrency, a feature that is ostensibly well-suited to the programming idiom of expressing all available concurrency.

From the perspective of mapping programs onto hardware, this idiom is attractive in several ways. First, the approach ensures that the concurrency expressed by a given program is both maximal and scales with problem size. These two properties are useful for achieving strong and weak scaling, respectively. Second, the idiom provides good portability. It abstracts away the physical details of processor cores and SIMD widths that may vary across GPUs. Finally, the oversubscription of processing elements with short-lived tasks helps ensure good load balancing and overall utilization.

As we show in the following section, however, this simplistic approach leads to substantial inefficiencies that stem from redundant computations and unnecessary rounds of communication between logical threads. Instead, we advocate design approaches where programmers explicitly express both serial and cooperative phases of their algorithms and rules for how they should be coupled. Although such flexible granularity coarsening complicates the expression of the program, we prefer to leave the granularity of serial work performed by CTAs, and warps, and threads unbound until the compiler and/or runtime maps their operation onto the target hardware.

### 3.3 GRANULARITY COARSENING

This section describes two important design idioms for applying granularity coarsening with respect to CTAs, warps, and threads: *CTA-serialization* and *thread serialization*. We make extensive use of these two patterns throughout this dissertation. Along the way, we illustrate examples of unnecessary overheads that are incurred by programs that are rigidly constructed to express all available concurrency.



**Fig. 14.** Example CTA decompositions for a data-parallel transformation. Tile size  $b=4$  elements.

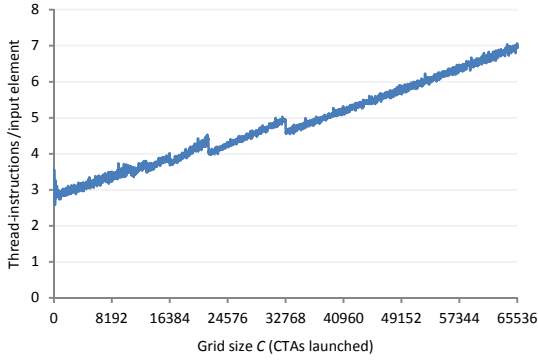
### 3.3.1 CTA serialization

The CUDA programming model encourages data-parallel decompositions where the number of threads, and thus the number of CTAs, scales with problem size. Fig. 14a illustrates this for a simple data-parallel transformation (e.g., copy). Each CTA processes exactly one tile<sup>5</sup> of data, typically where the number of data elements  $b$  in a tile corresponds to the number of threads in a CTA. For a given problem of size  $n$  and scheduling granularity  $b$ , the kernel will launch a grid of  $C = n/b$  CTAs.

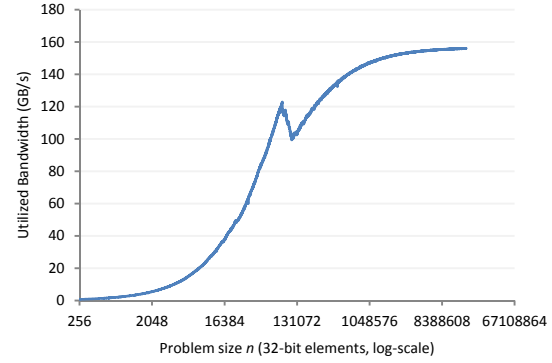
Fig. 14b illustrates an alternative CTA decomposition for the same data-parallel problem in which the number of CTAs launched  $C$  is constant. The tile-processing logic for each CTA is wrapped within in a while-loop. When  $C$  is a fixed multiple of cores  $p$ , each CTA is responsible for serially processing  $O(n/(pb))$  tiles. Because  $C$  is  $O(p)$ , the number of logical threads scales with processor width instead of problem size.

<sup>5</sup> To avoid further overloading of the term “block”, we use *tile* to describe a block of input data that a CTA is designed to process to completion before terminating or obtaining another block of input.





**Fig. 15.** “Copy” kernel instruction overhead vs. CTA granularity for 64M-element datasets (GTX480)

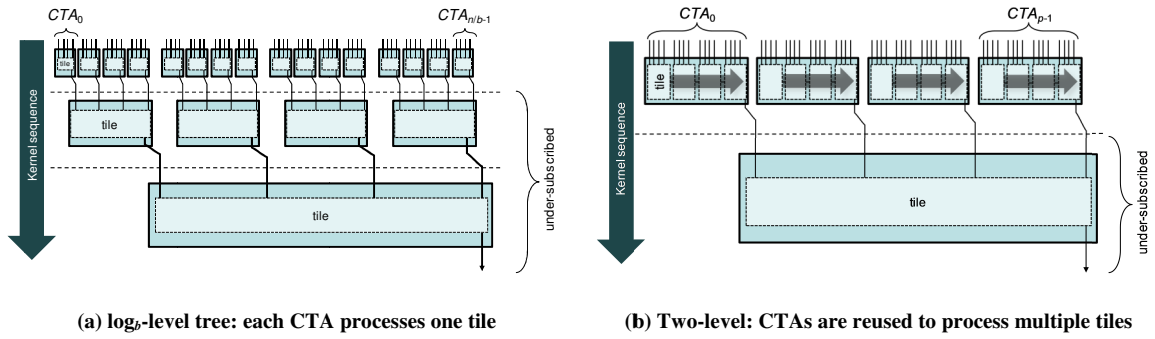


**Fig. 16.** “Copy” kernel utilized bandwidth vs. problem size  $n$  (no CTA serialization, GTX480)

We illustrate the effectiveness of this technique for a trivial data-parallel “copy” kernel. Threads simply read and write their 32-bit elements from one global array to another. We use 64M-element arrays, large enough to saturate the GTX480 memory subsystem. Fig. 15 plots the number of dynamic thread-instructions executed per input element as a function of the number of CTAs launched by the kernel. We vary the CTA count from the minimum number needed to occupy the processor ( $8p=120$  CTAs) to fully data-parallel ( $n/b = 64K$  CTAs where  $b=1024$ ).

We observe that dynamic instruction overhead increases linearly with the number of CTAs invoked. With fewer CTAs, the computational savings from reduced concurrency and increased serial processing are substantial. Compared to the strictly data-parallel extreme on the right hand side, restricting the amount of concurrency to the width of the processor reduces the overall computational workload by 57%.

Two factors contribute to these savings. First, the reduced number of logical threads lowers the overall thread-setup overhead. This includes instructions for loading the kernel parameters into registers, computing the offset of the CTA’s first tile, the offset



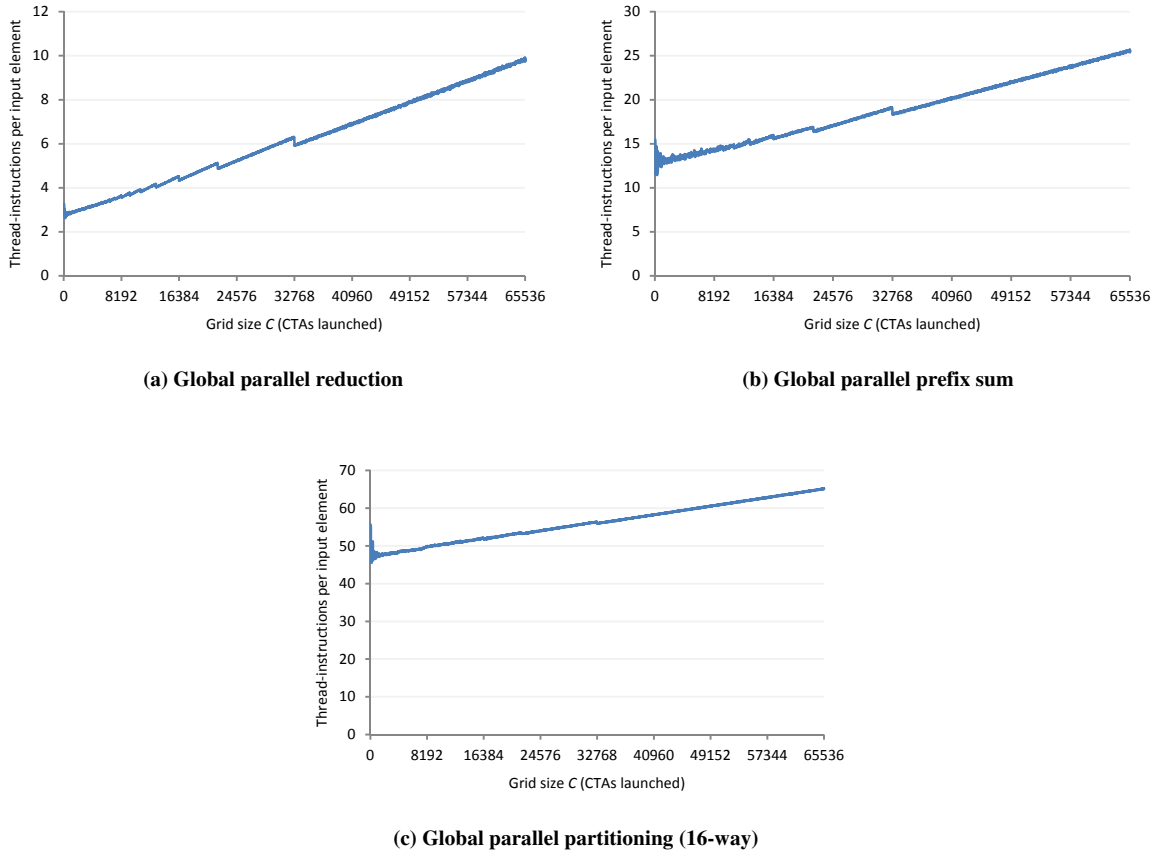
**Fig. 17.** Example CTA decompositions for global reduction. CTAs are comprised of four threads. Tile size  $b=4$  elements.

of the thread into that tile, etc. Second, the compiler can hoist operations out of the tile-processing loop, further reducing the workload per input element.

This CTA serialization idiom is also particularly effective for recursive decompositions. Fig. 17a illustrates the traditional recursive data-parallel decomposition for parallel reduction. Each CTA computes a partial reduction from its tile of  $b$  elements. The host program further invokes  $\log_b n - 1$  reduction kernels to reduce these partial reductions into a single aggregate result.

However, GPUs are only efficient when the problem size is large enough to saturate the processor. This is rarely true for the interior of the reduction tree. For example, the second level of a 64M element reduction tree with branching factor  $b=1024$  contains only 64K elements. Unfortunately the memory subsystem for the GTX480 only saturates for inputs larger than 8M elements (Fig. 16). The second and third kernel invocations leave the GPU undersubscribed. Only the first reduction kernel is capable of fully utilizing the processor.

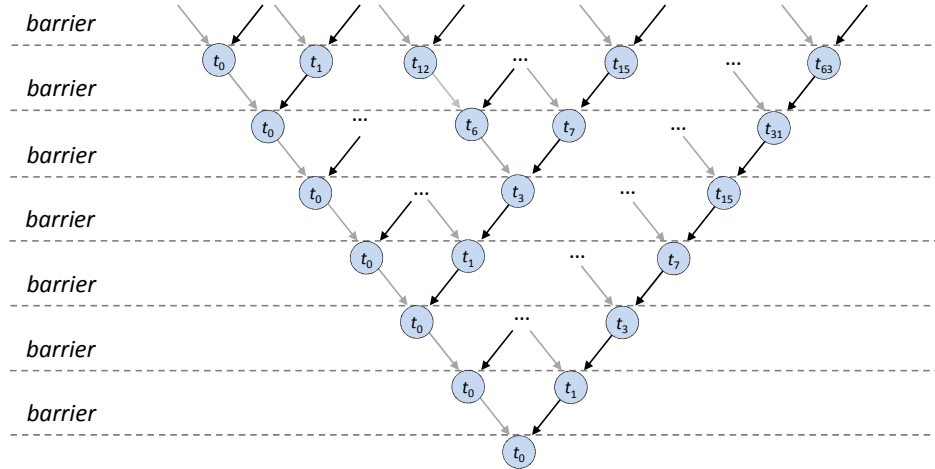
As an alternative, Fig. 17b illustrates the CTA serialization idiom as applied to our reduction example. When  $C$  is a constant multiple of  $p$  and roughly the same order of



**Fig. 18.** Cooperative instruction overhead vs. CTA granularity (GTX480)

magnitude as  $b$ , we only need a single-CTA kernel to reduce one tile of  $C$  partials. This two-level CTA decomposition finishes the inefficient part of parallel reduction as quickly as possible.

Furthermore, the cost of aggregating partial reductions between tiles is much lower. For sequentially-processed tiles, we can simply leave these partials in registers instead of exchanging them through global memory. We obviate  $O(n/b)$  global memory reads and writes at a savings of 2-4 instructions per round-trip (offset calculations, load, store). Instead, we only require  $O(C)$  global communication for partials, where  $C$  is now independent of  $n$ .



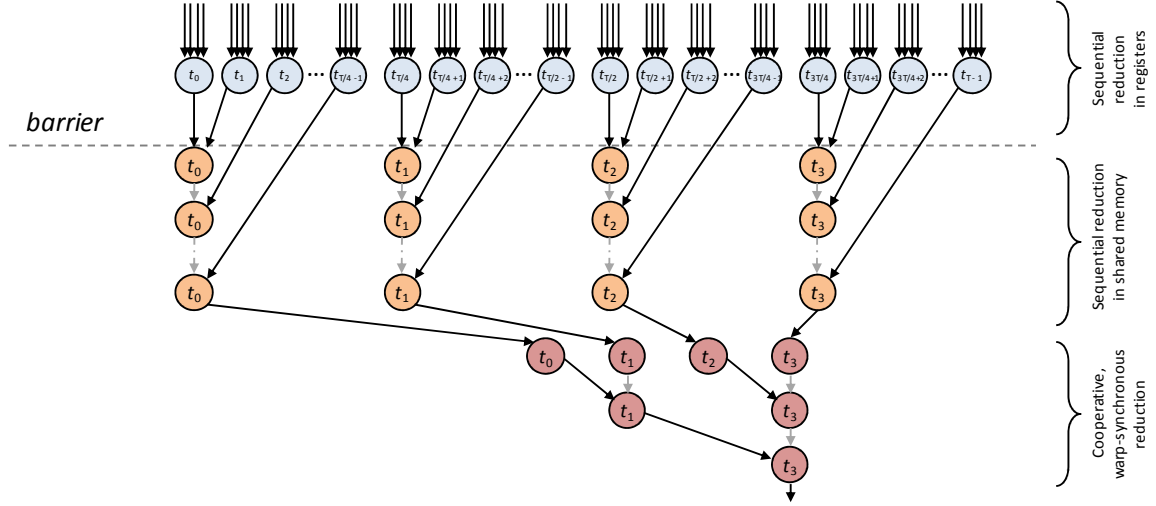
**Fig. 19.** Recursive, pair-wise parallelization of local CTA reduction. Lighter dataflow arrows indicate partials left in registers.

The two-level decomposition is also more energy-efficient. The energy required to move a 32-bit word through DRAM is currently on the order of 2,000 pJ [13]. The energy cost of leaving it in a register and simply reading it later is roughly 1.7 pJ.

Fig. 18 illustrates the effectiveness of CTA serialization for the cooperative problems of global reduction, prefix sum, and multi-way partitioning (for radix sorting). By only invoking as many CTAs as can be actively resident on the processor, we demonstrate computational savings of 67% for reduction, 42% for prefix sum, and 27% for partitioning.

### 3.3.2 Thread serialization

In this subsection, we discuss the merits of granularity coarsening for local cooperation with the CTA. The programming model's hierarchical memory spaces and grouping constructs encourage the decomposition of globally-cooperative problems into independent subproblems (tiles) that can be processed in nearby shared memory with much better locality.



**Fig. 20.** Recursive, three-phase parallelization of local CTA reduction. Lighter dataflow arrows indicate partials left in registers.

When expressed at their finest granularity, the task dependences for many cooperative parallelizations comprise binary trees of communication through shared memory spaces. Reduction and prefix sum are commonplace examples. At each timestep, the expressed concurrency is geometrically decreasing (or increasing). Fig. 19 illustrates such pair-wise reduction as mapped onto threads within a CTA.

Despite its simplicity and abundant concurrency, this parallelization is quite inefficient on GPU architecture. Each of the  $b-1$  reduction operators has an operand that needs to be written, synchronized, and read from shared memory. After performing an operator, threads must also evaluate a conditional to determine whether they will be active in the subsequent level. For example, a 1024-thread CTA requires 4,224 thread-instructions<sup>6</sup> to reduce a tile of  $b=1024$  elements.

A much better fit is the generic, three-phase construction illustrated in Fig. 20. Each phase seeks to either increase the amount of sequential work within a given storage

<sup>6</sup> The actual width of the final five reduction levels is the warp-width  $w_{\text{SIMD}}=32$ , regardless of deactivated threads.

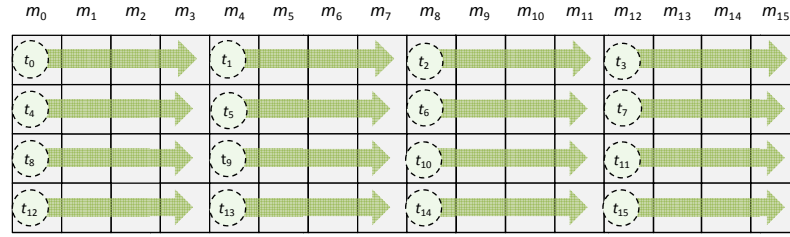
class (e.g., registers, shared memory, etc.) or exploit a particular aspect of the abstract machine model (e.g., lock-step thread progress within the warp):

- 1) ***Sequential reduction in registers.*** This phase decouples the tile size  $b$  from the CTA size  $p_{\text{CTA}}$ . Each thread loads  $b/p_{\text{CTA}}$  items. It is important that this phase be wide enough to saturate the global memory subsystem with requests. The loaded elements are sequentially reduced in registers without read, write, and barrier instructions.
- 2) ***Sequential reduction in shared memory.*** We place the partials from the previous step into shared memory, invoke a barrier, and then reduce the parallelism to the SIMD width  $w_{\text{SIMD}}$  of the processor core. One warp then serially *rakes*<sup>7</sup> over the shared partials for  $p_{\text{CTA}}/w_{\text{SIMD}}$  steps without write and barrier instructions.
- 3) ***Cooperative, warp-synchronous reduction.*** Finally, the single raking warp performs a synchronization-free, pair-wise reduction in shared memory of the partial reductions computed in the previous phase. We exploit the lock-step SIMD behavior of threads within the same warp to avoid explicit barrier synchronization.

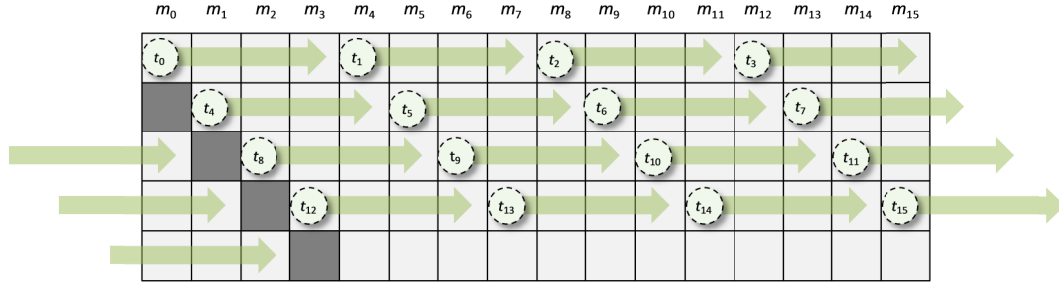
This construction only requires one barrier-synchronized exchange through shared memory that is accompanied by a single conditional for reducing the degree of parallelism. All other steps are free of conditionals, and the bulk of the reduction operators (first phase) are free of any shared memory overhead. Compared with the pair-wise example, this three-phase construction only requires 1,440 thread-instructions to reduce a tile of  $b=1024$  elements using a 128-thread CTA with  $w_{\text{SIMD}}=32$ , a savings of 67%.

---

<sup>7</sup> *Raking* is a strategy for assigning a set of threads  $p$  to process a much larger data set. Each thread is assigned an even-share of consecutive inputs to process serially, i.e., the stride between threads is  $p$  and the stride between elements for a given thread is 1. Alternatively, *strip mining* iteratively stripes the mapping of threads across the input set, i.e., the stride between threads is 1 and the stride between elements for a given thread is  $p$ .



(a) Raking reduction with four-way bank conflicts



(b) Conflict-free raking reduction with padding (four bytes of padding for every sixteen 32-bit partials placed into shared memory)

**Fig. 21.** Alternative memory layouts for raking reduction in shared memory (phase<sub>2</sub> of three-phase local reduction). The total number of partials to rake  $p_{CTA} = 64$ . Warp-width  $w_{SIMD} = 16$  threads. Each raking thread reduces four 32-bit partials. Local shared memory is comprised of 16 banks having a stride of four bytes per bank.

This example serves to illustrate the importance of expressing the “general shape” of cooperation from multiple algorithmic phases. However, we do not want to bind these phases to any particular widths and depths when authoring our programs. In this example, the tile size, CTA size, and warp size are the unbound tuning parameters that ultimately dictate the number of steps to statically unroll each phase. They also dictate the size and layout of shared memory needed for thread communication. We prefer to bind these parameters after empirically tuning for a specific problem and target architecture.

### 3.3.3 Involvement with the type system

The layout of shared memory for second phase of local reduction warrants additional discussion. The shared, physical memory per GPU core is comprised of sequentially-

accessible storage banks. Bank conflicts arise when threads within the same warp access different words residing in the same memory bank. A bank can only service one thread at a time. This results in the undesirable serialization of otherwise concurrent memory accesses.

Without proper padding, many of the concurrent reads made by threads in our raking warp would target the same memory banks. For example, the raking threads in Fig. 21a experience four-way bank conflicts for every read from shared memory, causing each read instruction to be replayed four times. The number of memory banks is a multiple of the stride between raking threads. Alternatively, the padding in Fig. 21b ensures no bank conflicts, as the number of banks and thread stride are relatively prime.

The formal data type being reduced affects the following aspects of shared memory layout: (1) the placement of padding; (2) the placement-offset for storing each partial reduction into shared memory; and (3) the raking offsets for raking threads. Continuing the example in Fig. 21, local reduction of 64-bit doubles would require eight bytes of padding for every eight doubles placed into shared memory. For 8-bit characters, our example requires no padding: all 64 characters fit within one row of memory banks.

In short, our tuning decisions, the problem type, and the target architecture all affect the data types we use to organize communication through shared memory.

### 3.4 TUNING VIA THE TYPE SYSTEM

Our design idiom for tuning via the type system uses the language’s support for template-based metaprogramming to ease the burden of granularity selection and algorithmic choice. We author our parallel algorithms such that they can be specialized by tuning



policy. We express such tuning policy using reflective C++ types. Our kernel procedures are parameterized to accept such policy types as template parameters. Within the procedure, we can then express various aspects of its behavior in terms of the information carried within the policy type.

In this fashion, we can author the “general shape” of an implementation, leaving many of the performance-sensitive details unbound. We can then use (and reuse) this code later by binding it with a tuning configuration policy that matches the specific problem at hand. The configuration policy guides the compiler in unrolling and generating well-tuned code.

Because the policy is statically known to the compiler, we eliminate the need for any runtime decision-making with each logical thread. The overhead of runtime decision-making (e.g., how many loads to unroll) is particularly costly on GPU-like architectures having tens or hundreds of thousands of resident threads.

#### ***3.4.1 A simple example: data-parallel copy***

Consider data-parallel copy as a trivial example. As one of the simplest stencil kernels, threads simply load elements from a global input array and write them to equivalent locations within the output array. Listing 1 illustrates a “concrete” tile-copying subroutine in which a CTA copies a tile of 32-bit floats. Each thread loads and stores exactly one float.

In practice, the ostensibly simple copy operation incorporates quite a few tuning decisions that are opaque in terms of their performance impact for any given architecture and problem type. Lines 2-14 in Listing 2 illustrate a parametric type *Policy* that can be specialized in the following tuning dimensions:

**Listing 1.** A straightforward kernel subroutine for CTAs to copy tiles of 32-bit floats from one global array to another

---

**Template parameters:** None

**Formal parameters:**

- Global input and output arrays *d\_in*, *d\_out*
- Offset *tile\_offset* into *d\_in*/*d\_out* of the tile to be copied
- Optional limit *guarded\_elements* on the number of tile elements to copy

**Other:**

- Global variable *thread\_id* for thread identifier
- Global variable *cta\_size* for CTA-size in threads

---

```

1  __device__ void CopyTile(
1  float *d_in,
2  float *d_out,
3  size_t cta_offset,
4  size_t guarded_elements = cta_size)
5  {
6  if (thread_id < guarded_elements) {
7
8      // Load tile data
9      float data =
10         d_in[tile_offset + thread_id];
11
12     // Store tile data
13     d_out[tile_offset + thread_id] =
14         data;
15 }
16 }
```

---

**Listing 2.** A tuning policy type for data-parallel copy, followed by an example parameterization of that type specialized for large-problems of 8-byte elements on the GF100 architecture.

---

```

1  // Tuning policy type
2  template <
3      // Problem instance type parameters
4      typename T,
5      int ARCHITECTURE,
6
7      // Tunable parameters
8      int LOG_THREADS,
9      int LOG_LOAD_VEC_SIZE,
10     int LOG_LOADS_PER_TILE,
11     ld::CacheModifier READ_MODIFIER,
12     st::CacheModifier WRITE_MODIFIER,
13     bool WORK_STEALING>
14     struct Policy;
15
16 // Example policy parameterization
17 // tuned for 8-byte data, large-size
18 // problems
19 typedef Policy<unsigned long long,
20     GF100, 8, 7, 1, 0, ld::cg,
21     st::cg, true>
22     LargeProblemPolicy8B;
```

---

- a) ***The number of loads per thread per tile.*** This allows us to increase the number of outstanding loads issued before stores at the expense of increased register pressure. Reasonable configurations include  $2^0$ ,  $2^1$ , and  $2^2$  loads per thread per tile.
- b) ***The number of items per load.*** Current NVIDIA GPUs support vector-loads of up to four component elements. Reasonable configurations include  $2^0$ ,  $2^1$ , and  $2^2$  elements per vector load.
- c) ***The number of threads per CTA.*** Reasonable configurations include powers-of-twos ranging from  $2^5$  to  $2^{10}$  threads.

**Listing 3.** A generalized, policy-based kernel subroutine for CTAs to copy tiles of elements from one global array to another.

<p><b>Template parameters:</b></p> <ul style="list-style-type: none"> <li>• Tuning policy type <i>Policy</i> having the following type definition fields: <ul style="list-style-type: none"> <li>◦ <i>T</i> (data type to be copied)</li> <li>◦ <i>SizeT</i> (data type for offsets)</li> </ul> </li> <li>and enumerated constant fields: <ul style="list-style-type: none"> <li>◦ <i>LOADS_PER_TILE</i> (number of loads per tile)</li> <li>◦ <i>LOAD_VEC_SIZE</i> (elements per load)</li> <li>◦ <i>THREADS</i> (number of threads per CTA)</li> </ul> </li> </ul>	<p><b>Formal parameters:</b></p> <ul style="list-style-type: none"> <li>• Global input and output arrays <i>d_in</i>, <i>d_out</i></li> <li>• Offset <i>tile_offset</i> into <i>d_in/d_out</i> of the tile to be copied</li> <li>• Optional limit <i>guarded_elements</i> on the number of tile elements to copy</li> </ul> <p><b>Other:</b></p> <ul style="list-style-type: none"> <li>• Device function <i>LoadTileValid()</i> for reading each thread's tile portion</li> <li>• Device function <i>StoreTileValid()</i> for writing each thread's tile portion</li> </ul>
--	--

---

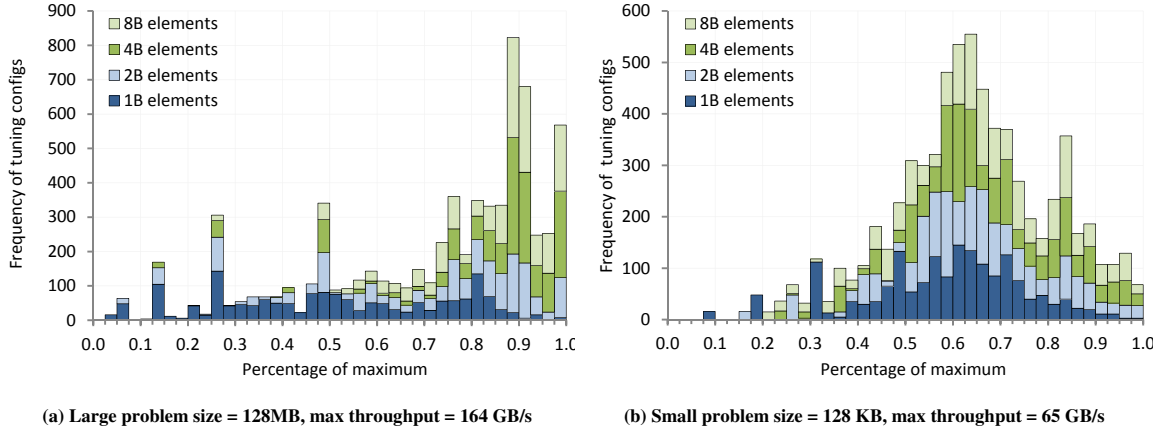
```

1  template <typename Policy>
2  __device__ void CopyTile(
3      typename Policy::T *d_in,
4      typename Policy::T *d_out,
5      typename size_t tile_offset,
6      typename size_t guarded_elements =
7          Policy::ELEMENTS_PER_TILE)
8  {
9      // Tile data
10     typename Policy::T
11         data[Policy::LOADS_PER_TILE]
12         [Policy::LOAD_VEC_SIZE];
13
14     // Load tile
15     LoadTileValid <
16         Policy::LOADS_PER_TILE,
17         Policy::LOAD_VEC_SIZE,
18         Policy::THREADS,
19         Policy::READ_MODIFIER>(
20         data, d_in + tile_offset, guarded_elements);
21
22     // Store tile
23     StoreTileValid<
24         Policy::LOADS_PER_TILE,
25         Policy::LOAD_VEC_SIZE,
26         Policy::THREADS,
27         Policy::WRITE_MODIFIER> (
28         data, d_out + tile_offset, guarded_elements);
29 }

```

---

- d) **Work-stealing.** As algorithmic variants, we can either: (a) provide each CTA with an even-share of input tiles; or (b) allow CTAs to “steal” tiles of work using bulk atomic-addition as described in Chapter 1.3.1.
- e) **Caching directives.** These modifiers affect cache behavior during loads and stores. Current NVIDIA GPUs expose up to four variants: default caching at L2



**Fig. 22.** “Copy” kernel performance histograms of tuning configurations binned by normalized slowdown with respect to the maximum throughput achieved (NVIDIA GTX 480).

and L1 levels; no caching; cache in global L2 using smaller cache lines; and tagging for preferential eviction.

Listing 3 illustrates a templated copy subroutine that expresses the “general shape” of tile-copying. This procedure is not bound to a specific type of copy-element. In addition, each thread loads and stores a tunable number of elements. Such tuning details are encapsulated within the template parameter type *Policy*. An example “concrete” tuning policy that we have identified for copying large lists of 8-byte elements is shown in Listing 2, lines 18-20.

Fig. 22 illustrates the diversity of the corresponding performance landscape for the current NVIDIA GF100 architecture (GTX480). These tuning options enumerate a configuration space of 1,728 tuning variants per data type, per problem size. We evaluate these specializations for a pair of “large” and “small” representative workloads: 128MB and 128KB. Furthermore, we explore the configuration space for 1-byte, 2-byte, 4-byte, and 8-byte data types for each problem size. We normalize the throughputs of each tuning configuration against the maximum observed for its problem size and plot the resulting slowdown histograms.

<b>Table 2.</b> Max achievable DRAM bandwidth ( $10^9$ Bytes/s)			
	<i>GTX480</i>	<i>GTX280</i>	<i>9800 GTX+</i>
<i>Unidirectional (even-share)</i>	163.4	135.6	67.8
<i>Unidirectional (steal)</i>	168.6	63.6	42.6
<i>Bidirectional (even-share)</i>	153.6	125.4	61.7
<i>Bidirectional (steal)</i>	163.7	85.3	55.5

The large problem size (Fig. 22a) is representative of datasets large enough to saturate the memory subsystem. In general, the GTX480 is somewhat forgiving at this problem size, i.e., it is skewed to the right. On average, 25% of all configurations achieve more than 90% of the maximum achievable throughput (164 GB/s). However, we observe that it is relatively much more difficult to achieve this performance when copying 1-byte characters. Only 2% of configurations achieve more than 90% of maximum on 1B problem instances.

The performance for the small problem size (Fig. 22b) is much more diverse. Only 6% of all specializations fall within 90% of the maximum throughput (65 GB/s). For the various problems discussed throughout this dissertation, we generally observe that it is comparatively harder to find tuning configurations that are well-suited to small, fleeting workloads.

We also observed the configurations corresponding to the straightforward implementation specified in Listing 1 were not particularly competitive. For the large 128MB problems instances, the best 4-byte, 1-load, vector-1 configurations perform at less than 90% of the maximum achievable bandwidth. For the small 128KB instances, these configurations only achieve 65% of maximum. It is not obvious to the programmer that this “concrete” implementation would perform so poorly.

Table 3. Corpus of tuning benchmarks				
<i>Benchmark</i>	<i>Description</i>	<i>Kernel tuning dimensions</i>	<i>Tuning configs per problem instance</i>	<i>Total sample evaluations</i>
<i>Global copy</i>	One kernel.	Copy kernel: a, b, c, d, e	1,728	124,416
<i>Global reduction</i>	Two kernels (“upsweep” and “spine”). Each CTA within the “upsweep” computes a partial reduction of its portion. A single-CTA “spine” kernel further reduces these partials.	Upsweep kernel: a, b, c, d Spine kernel: a, b, c	8,748	104,976
<i>Global prefix sum</i>	Three kernels (“upsweep”, “spine”, and “downsweep”). See Chapter 4.	Upsweep kernel: a, b, c Spine kernel: a, b, c Downsweep kernel: a, b, c	157,464	11,337,408
<i>Reduce-by-key</i>	Three kernels (“upsweep”, “spine”, and “downsweep”). See Chapter 4.	Upsweep kernel: a, b, c Spine kernel: a, b, c Downsweep Kernel: a, b, c	157,464	11,337,408

Finally, we use this tunable kernel to determine the maximum-achievable DRAM bandwidths for each of our three of our evaluation GPUs (GTX480, GTX280, and 9800 GTX+). We use these throughputs, listed in Table 2, to evaluate memory-bound implementations throughout this dissertation.

### 3.4.2 Analysis of performance landscape across GPU architecture

In this section, we explore the cumulative tuning landscape for several data-parallel and cooperative problems across the last three generations of NVIDIA GPU architecture. Our results show:

- Large performance spread across reasonable specializations
- Specializations themselves have large performance variance across different GPUs, problem types, and problem sizes
- No single specialization for a given problem performs exceedingly well across all data types, problem sizes, and architectures

Our evaluation is comprised of the following four benchmark problems: *global copy*, *global reduction*, *global prefix sum*, and *global reduce-by-key*<sup>8</sup>. Table 3 lists the kernels that comprise each benchmark and the dimensions along which we can tune each kernel. For example, the *reduce-by-key* benchmark has three kernels, each of which can be tuned by loads-per-thread, items-per-load, and number-of-threads-per-CTA ( $a$ ,  $b$ , and  $c$  from the previous section). With three kernels and 54 tuning specializations per kernel, the benchmark has an overall tuning domain of 157,464 tuning configurations.

Our investigation evaluates how different tuning policies respond to different problem instances (where a problem instance is a specific combination of data type, problem size, and GPU architecture). We evaluate the performance of each tuning configuration across a sample space of 72 problem instances constructed from combinations of the following:

- Four data types (1-byte, 2-byte, 4-byte, and 8-byte elements)
- Six problem sizes (128 KB, 512 KB, 2MB, 8MB, 32MB, and 128 MB)
- Three GPU architectures (NVIDIA GF100, GT200, G92 represented by GTX480, GTX280, and 9800 GTX+ GPUs)

We are interested in gauging how performance varies *between* configurations as well as *within* configurations. These two properties intuitively correspond to configuration “strength” and “consistency”, respectively.

We normalize our performance samples to the interval  $[0,1]$  so that we may generalize behavior across problem instances. For every problem instance, we identify the tuning configuration that provides the best sample performance. (For example,

---

<sup>8</sup> Reduce-by-key is the third phase of the map-reduce paradigm (after mapping and sorting) [38]. Given a list of key-value pairs, it is analogous to a segmented reduction over the values where the segments are defined by regions of consecutive, identical keys.

Table 4. Between-configs slowdown variance ( $s^2_B$ )				
	GTX480	GTX280	9800 GTX+	All GPUs
<i>Copy</i>	0.52	0.08	0.48	0.40
<i>Reduction</i>	0.74	0.15	0.31	0.41
<i>Scan</i>	0.58	0.42	0.31	0.83
<i>Reduce-by-key</i>	0.53	0.38	0.25	0.91

Table 5. Within-configs slowdown variance ( $s^2_W$ )				
	GTX480	GTX280	9800 GTX+	All GPUs
<i>Copy</i>	0.03	0.04	0.14	0.07
<i>Reduction</i>	0.03	0.04	0.11	0.06
<i>Scan</i>	0.03	0.02	0.09	0.06
<i>Reduce-by-key</i>	0.01	0.01	0.03	0.02

reducing 128 MB of 4-byte integers on GT200 maximally proceeds at 169 GB/s.) We then normalize the performance samples of all configurations for that problem instance in terms of relative slowdown against this “best” performance.

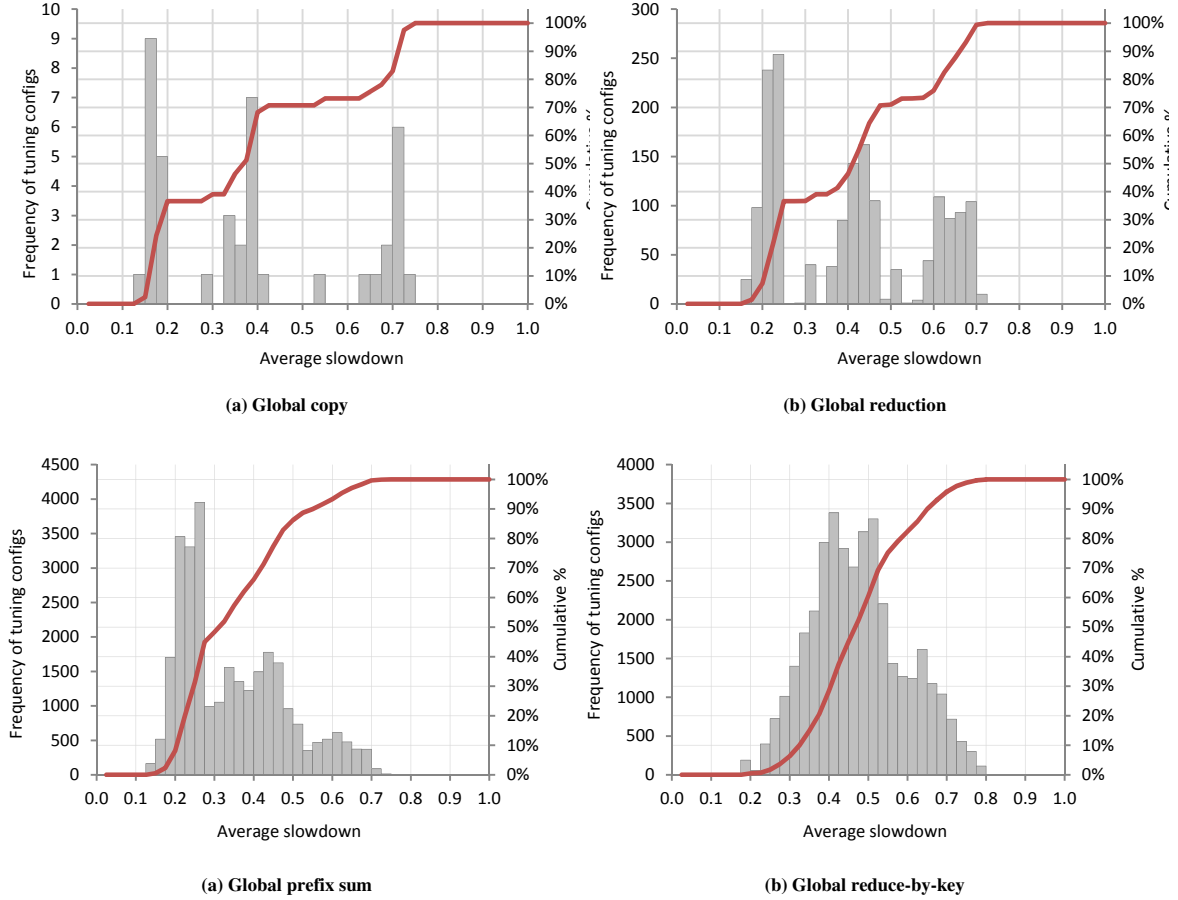
We use the statistical metrics *between-group variance* ( $s^2_B$ ) and *within-group variance* ( $s^2_W$ ) for analyzing the diversities of configuration strength and consistency, respectively [41]. The between-group variance is a measure of the variability of configuration means around the grand mean. The within-group variance is a weighted average of configuration variance, with weights determined by the number of problem instance samples in each configuration.<sup>9</sup>

Table 4 and Table 5 present the between-group and within-group variances, respectively. The large ratios of  $s^2_B/s^2_W$  indicate that the broad majority of overall variation between pairings of configurations and problem instances is due to differences *between* configurations, i.e., certain configurations are innately better or worse than others. The performance-slowdown histograms in Fig. 23 graphically illustrate the ample performance variation amongst tuning configurations by binning configurations by their average slowdown.

Furthermore, Table 4 also reveals that some architectures are relatively more pliant than others. For example, the variances among tuning configurations are much

<sup>9</sup> These metrics are used when performing statistical analysis of variance (ANOVA) to determine whether a set of groups are significantly dissimilar.





**Fig. 23.** Performance histograms of tuning configuration “strength”. Configurations are binned by the harmonic mean of their normalized slowdown across all problem instances.

lower for problem instances on the GTX280 than for the newer GTX480, particularly for the *reduction* benchmark.

Despite being dwarfed by between-groups variance, the within-groups variance  $s^2_w$  is also fairly significant. For example, the within-groups deviation  $s_w$  for *prefix sum* across all GPUs is  $\sqrt{0.6} = 24\%$ . This implies that performance is also strongly related to problem instance, and that it will be relatively difficult to find tuning configurations that are universally better than others.

The histograms in Fig. 23 corroborate the absence of tuning configurations that perform well across the entire sample space of problem instances. “Well-rounded”

Table 6. Average bandwidth utilization of <i>all</i> 128MB tuning configurations			
	<i>GTX480</i>	<i>GTX280</i>	<i>9800 GTX+</i>
<i>Copy</i>	0.72	0.43	0.45
<i>Reduction</i>	0.61	0.32	0.35
<i>Scan</i>	0.59	0.46	0.47
<i>Reduce-by-key</i>	0.31	0.16	0.16

Table 7. Average bandwidth utilization of <i>best</i> 128MB tuning configurations			
	<i>GTX480</i>	<i>GTX280</i>	<i>9800 GTX+</i>
<i>Copy</i>	1.00	0.99	0.99
<i>Reduction</i>	0.96	0.88	0.95
<i>Scan</i>	0.97	0.97	0.94
<i>Reduce-by-key</i>	0.67	0.38	0.33

tuning configurations do not exist. For example, no single configuration for *copy* averages more than 75% of the maximum-achievable performance across problem instances. For *reduction*, *prefix-sum*, and *reduce-by-key*, the best all-purpose configurations only average 73%, 73%, and 83% of what we can maximally achieve.

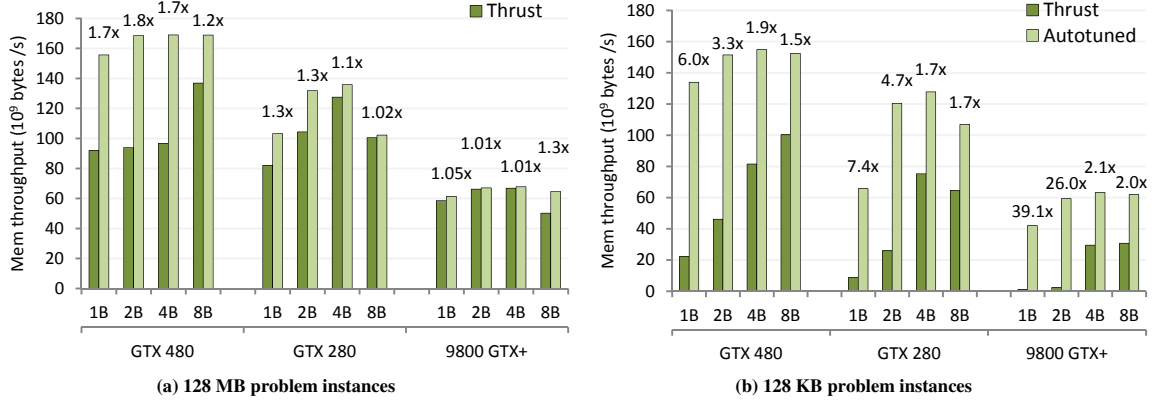
### 3.4.3 Effectiveness of auto-tuning

In this section, we evaluate how successful our tuning exploration is at identifying good code specializations.

For large saturating problem sizes, we would like our memory-bound problems to proceed at the maximum-achievable DRAM bandwidth for each device. Because of the heavily overlapped nature of the GPU, we would expect that all memory-bound specializations would yield equal performance.

However, Table 6 reveals this not to be the case. It presents the average bandwidth utilization of tuning configurations paired with 128MB problem instances, normalized to the DRAM bandwidth presented in Table 2. The implementations that should be bandwidth-bound at this problem size (namely *copy*, *reduction*, and *prefix sum*) are nowhere near maximum bandwidth utilization.

Our autotuning search is quite effective at finding configurations that perform at peak or near-peak bandwidth. Selecting among only the best-performing configurations



**Fig. 24.** Global reduction performance comparison between our autotuned and the “concrete” Thrust implementations.

for each of the 128MB problem instances, Table 7 shows that we can identify policy configurations that perform exceptionally well for each data type (1B – 8B). Even for our compute-bound problem (*reduce-by-key*), our best-performing configurations are more than twice as fast.

We further illustrate the need for specialization by comparing our tuned global reduction kernels against those provided by the Thrust library of GPU primitives [113]. Up until this point, we have only compared our best specializations with our own average specializations. This raises the question of whether our average specializations are representative of concrete implementations “in the wild.” Of the Thrust library of parallel primitives, only the global reduction implementation shares the same overall parallelization strategy<sup>10</sup>.

Fig. 24 illustrates our autotuned reduction performance advantage over the Thrust implementation for both saturating 128MB and fleeting 128KB problem instances. For large, GF100-based problems instances, the Thrust performances align with our average configuration performance. In relation, our tuned specializations achieve a harmonic

<sup>10</sup> Their implementations of scan, reduce-by-key, duplicate-removal, etc., all impose significantly larger memory workloads than our parallelizations. For data-parallel copy, they defer to the CUDA *cudaMemcpy()* API.

mean speedup of 1.6x. Their large-problem performance is relatively much better for the older GT200 and G92 architectures. We only achieve 1.14x and 1.08x speedups for those GPUs, respectively.

Fig. 24b illustrates the importance of autotuning for small problem sizes. For this subset of problem instances, the Thrust performance is representative of our grand-mean configuration slowdown of 0.6 across all reduction problems. In relation, our tuned specializations achieve harmonic mean speedups of 2.4x, 2.6x, and 3.9x for the GF100, GT200, and G92 architectures, respectively.

### 3.5 CHAPTER SUMMARY

We believe the days of writing concrete device code for the GPU are numbered. If mainstream software developers are to embrace the GPU computing paradigm, the paradigm itself must embrace performance-portability. At this juncture, many programmers appear willing to port their applications to the bulk-synchronous, data-parallel programming model. However, they will certainly balk at having to reconstruct their implementations in order achieve good utilization and efficiency when they wish to adapt them to different problem types and/or new processor architectures.

This reflects our own experience while constructing reusable library primitives. This chapter described the two related design idioms that we have developed in order to achieve good performance-portability across a diversity of problem types and target architectures: flexible granularity coarsening and tuning via the type system.

*Flexible granularity coarsening.* Many parallel programming paradigms encourage programmers to express every last bit of concurrency inherent within their problem in hope that the compiler and/or runtime will efficiently schedule it onto the

underlying hardware. Our profiling revealed that doing so often leads to substantial inefficiencies from redundant operations and unnecessary rounds of communication. This is particularly true for cooperative problems where the programmer dictates explicit communications patterns, making it difficult or impossible for the compiler to restructure such data flow in more efficient ways.

As an alternative, we demonstrated the utility of an idiom where programmers express serial and cooperative phases of the algorithm, and rules for how these phases should be coupled by the compiler. Our goal is for the concurrency expressed by the compiled implementation to scale with the width of the target architecture, not problem size.

***Tuning via the type system.*** When authoring GPU programs, there are many decisions that must be made in order to construct a concrete program, yet have performance consequences that are opaque to the programmer. Achieving performance-portability implies leaving these details unbound in the program text.

Among such decisions is the sliding scale of granularity coarsening described above, which influences the layout of shared memory spaces that cooperating threads communicate through. As such, the tuning process necessarily involves the programming language’s type system. As a mechanism for expressing rules for compiler to construct valid programs, the type system is both suitable and convenient for guiding the compilation of specialized implementation variants.

In the course of our investigation, we validated two important arguments that underscore the usefulness of our policy-based tuning idioms:

- 1) Inflexible, concrete implementations are often incapable of delivering good performance across the domain of problem instances they might be expected to address. It is particularly hard to simultaneously achieve good performance from a single implementation on both large, saturating workloads and small, fleeting workloads.
- 2) By expressing only the general “shape” of the solution in our program text, our autotuning approach consistently discovers good program specializations for the specific problem instance at hand.

Despite the added complexity of having to reason about both the execution of the program and the execution of the compiler, we feel that long-term benefits of performance-portability will be worth the effort.

## Chapter 4

### *Parallel Prefix Scan*

#### 4.1 INTRODUCTION

Software developers rely on algorithmic primitives as basic building blocks for solving more complex problems. A particularly useful primitive for list processing applications on parallel machines is *prefix scan* (also known *prefix reduction* or simply as *scan*). Given a list of input elements and a reduction operator, scan produces an output list where each element is computed to be the reduction of the elements occurring earlier in the input list. Implementations of parallel scan support a wide variety of problem domains, e.g., sorting, stream compaction, construction of trees, cooperative queue management, solving recurrence relations, etc. [16, 17, 61]

A salient characteristic of scan parallelizations is that the computational granularity of concurrent tasks is miniscule, often comprising only a single binary instruction (e.g., addition). This aspect of scan makes it particularly amenable to fine grained computational environments, e.g., directly within electronic circuitry as well as within software for wide parallel architectures such as vector and GPU processors.

The primary performance consequence of such small computational granularity is that global memory bandwidth should be the limiting hardware resource. However, we find this is not true of current GPU implementations of parallel scan: they either make inefficient use of device memory accesses, exhibit high dynamic instruction counts, or both [15, 35, 42].

We argue that common data-parallel programming patterns and design idioms are responsible. The GPU programming model fundamentally encourages programmers to decompose problems in ways that map unique threads onto individual data elements, i.e., the numbers of threads and their corresponding grouping constructs scale with problem size.

However, prefix scan is not a data-parallel problem. It is a cooperative one. Its efficient computation requires intermediate computations to be shared amongst parallel processing elements. As a result, approaches incorporating a data-parallel style of thread assignment impose unnecessary memory traffic in proportion to problem size. To efficiently implement prefix scan on GPU architecture, we must increase the amount of serial work (*granularity coarsening*) in order for communication overheads to scale with processor width, not problem size.

Furthermore, simply improving scan efficiency to the point where it is bandwidth-bound does not go far enough. In isolation, any bandwidth-bound parallelization is just as fast as the next. However, further reducing the dynamic instruction count provides room for other computations to be performed under the covers of memory latency as well. The more efficient the scan implementation, the more application-specific computation we can *fuse* into it without incurring additional performance overhead.



Throughout this dissertation, we advocate a design idiom of *kernel-fusion* where we construct variants of global prefix scan, embedding *within* them problem-specific logic that will realize behavior for sorting, duplicate removal, graph traversal, etc. This is an inversion of the usual pattern for program composition where application logic would call down into prefix sum as a subroutine.

Our work as described in this chapter makes contributions in the following areas:

***Local parallelization strategies.*** We investigate two variants of intra-CTA prefix scan having different degrees of granularity coarsening. One prioritizes low-latency for small, fleeting problems and the other high-efficiency for large, saturating problems. Their computational overheads are up to 1.8x lower than prior work, making them bandwidth-bound and thus suitable for kernel fusion.

***Global parallelization strategies.*** Our global scan implementations employ a *two-level reduce-then-scan* CTA decomposition that imposes 25% less global memory traffic and only requires a constant amount global storage for intermediate results. We demonstrate 1.7x and 3.8x speedups for global scan and segmented-scan, respectively.

***Parallel primitives.*** To demonstrate the utility of kernel fusion with efficient prefix sum, we have constructed BackForty [6], an open-source C++ library of fundamental list-processing transformations for the NVIDIA CUDA parallel computing framework [34]. We provide high performance implementations of scan, segmented scan, duplicate removal, histogram, and reduce-by-key that achieve several factors of speedup over prior work [35, 113] across many diverse problem sizes and data types.

[8, 6, 7, 5, 3, 0, 9]	[0, 8, 14, 21, 26, 29, 29]
(a) Input	(b) Exclusive prefix sum
[8, 14, 21, 26, 29, 29, 38]	[0, 8, 14, 0, 5, 0, 0]
(c) Exclusive prefix sum	(d) Segmented exclusive prefix sum for segment flags [1, 0, 0, 1, 0, 1, 0]

**Fig. 25.** Examples of prefix sum variants.

## 4.2 BACKGROUND

### 4.2.1 Prefix scan

Prefix scan is a higher-order function that consumes an  $n$ -element input list  $(x_0, \dots, x_{n-1})$  and a binary associative combining operator  $\oplus$ . It produces an output list  $(y_0, \dots, y_{n-1})$  where

$$\begin{aligned}
 y_i &= \bigoplus_{0 \leq a < i} x_a && \text{when } 0 < i < n \\
 &= id_{\oplus} && \text{when } i = 0
 \end{aligned}$$

Multiple variations of scan exist, all having a prefix-dependency characteristic where the  $i^{\text{th}}$  output element is a function of the previous input elements. The version described above, *exclusive scan*, does not incorporate the  $i^{\text{th}}$  input element within the  $i^{\text{th}}$  output reduction. As such, exclusive scans rely upon the existence of an identity element  $id_{\oplus}$  having the property that  $x_a \oplus id_{\oplus} = x_a$ . For example,  $id_+=0$  for addition,  $id_*=1$  for multiplication, etc.

*Inclusive scan* is similar with the exception  $y_i = \oplus(x_0, \dots, x_i)$ . *Reverse scan* (also known as *backward scan*) processes the input elements with a “postfix dependency”, i.e.,  $y_i = \oplus(x_{i+1}, \dots, x_{n-1})$ . *Segmented scan* is a composition of scan instances: the input is a sequence of list segments, typically delineated by marker flags, each of which is to be

scanned separately. *Prefix sum* connotes a prefix scan using addition as the binary combining operator.

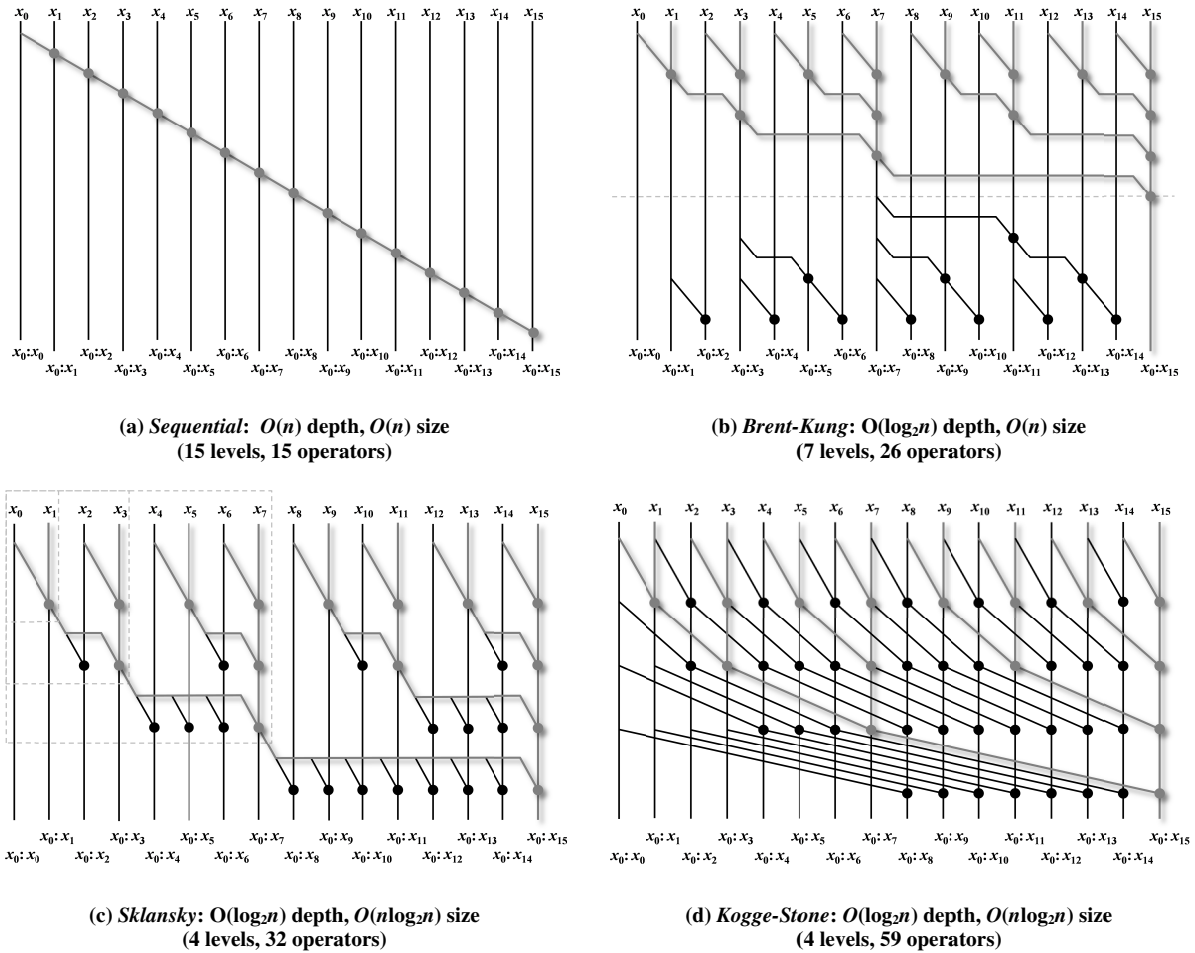
The long history of prefix sum is rooted in circuit design for parallel adders and counting networks. Scan circuits are fundamental to the operation of fast adder hardware [20, 105]. Software-based scan was popularized more than two decades ago as an algorithmic building block for vector and array processor architectures [16, 17, 26].

***Scan parallelizations.*** As per the *boolean circuit model* of parallel computation [19, 117], scan parallelizations are evaluated by their *size* and *depth* complexities. The size complexity of a circuit family (as a function of the input size  $n$ ) is a measure of the total number of operations performed. When consuming large, saturating inputs on the GPU ( $p \ll n$ ), overall runtime will be dictated by size complexity (also known as *work complexity*).

The depth complexity is a measure of the length of the longest path from an input value to an output, and is a performance indicator for how long the computation will take given an unlimited number of parallel processing elements  $p$ . For small, fleeting problems ( $p > n$ ), runtime will be dictated by depth complexity.

Prefix scan can be thought of as a composition of  $n$  binary reductions. Although these reductions could be performed separately, it is much more efficient to compose them together in such a manner such that they share as many intermediate computations as possible, reducing the overall size of the scan circuit.

The design space for parallel prefix circuits, i.e., all possible superpositions of the parallel reduction trees, is quite large. Fig. 26 illustrates four common constructions for prefix sum: sequential, Brent-Kung [20], Sklansky [105], and Kogge-Stone [74]. Like



**Fig. 26.** Alternative constructions for 16-element prefix sum. Computation proceeds from top to bottom, pads connote binary reduction operations, and lines indicate dataflow dependencies.

many prefix scan constructions, they are recursive in nature and thus easily adaptable to level-synchronous architectures such as the GPU.

Prefix scan has  $O(n)$  work-complexity: all  $n$  input elements must be read and the sequential implementation imposes only linear-work. Prefix networks are also subject to the following size-depth tradeoff: for a given network of size  $s$  gates and depth  $d$  levels,  $d+s \geq 2n-2$  [106]. The amount by which a given prefix network misses the depth-size lower bound of  $2n-2$  is called its *deficiency*. A network with zero deficiency is called *depth-size optimal* (DSO). It is easy to see that the sequential prefix network is DSO.

However, work-efficient DSO prefix networks become progressively difficult to find (or disappear altogether) as the depth-constraint is made smaller.

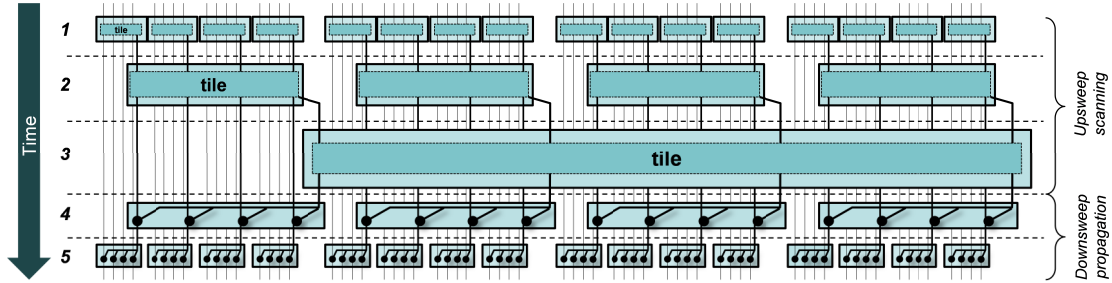
This tradeoff between work-efficiency and low-latency has important consequences for GPU prefix scan. Reducing the logical depth of local prefix constructions at the expense of additional operators often worsens overall performance. GPU processor cores are intended to be over-saturated with logical threads, causing performance to be rate-limited by overall work. Thus smaller, deeper prefix constructions are often preferable to larger constructions having shallower “logical” depths. Our most efficient scan implementations incorporate long, deep phases of sequential work (Fig. 26a), whereas prior work is constructed from work-inefficient Kogge-Stone subroutines (Fig. 26d) [80, 102].

#### 4.3 GLOBAL CTA DECOMPOSITION

A one-to-one mapping between levels of prefix scan dataflow and bulk-synchronous GPU kernels would be impractical. Storing every intermediate result back to global memory would be prohibitively expensive, particularly for constructions having sub-optimal work-complexity.

Instead, the thread grouping hierarchy allows us to take advantage of the recursive nature of many scan constructions. Consecutive kernels can be used to recursively process fixed-size tiles of work locally within shared memory, ultimately communicating a much smaller subset of intermediate values through global memory between kernel invocations.

Prior GPU scan implementations have used fully-recursive approaches for decomposing work into fixed-size tiles among CTAs. In this section, we review two such



**Fig. 27.** Example operation of a fully-recursive *scan-then-propagate* CTA decomposition in which each CTA processes  $b=4$  values.

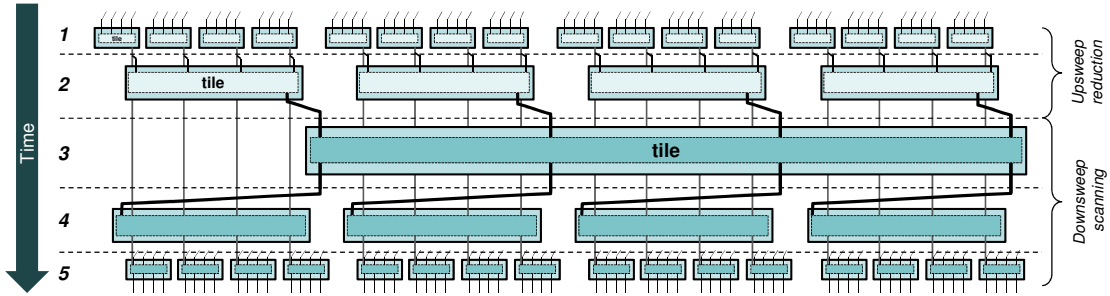
variants (which we term them *scan-then-propagate* and *reduce-then-scan*) and then present our own variation (*two-level reduce-then-scan*) which requires less memory traffic and provides better overall processor utilization.

#### 4.3.1 Scan-then-propagate

The data flow within the Brent-Kung construction (Fig. 26b) can be thought of as existing in a balanced binary-tree communication network. In the *upsweep phase*, the leaves are supplied with the input elements and the interior nodes progressively accumulate partial reductions upwards towards the root. In the *downsweep phase*, the accumulated partial reductions are then propagated back downwards from the root, with the right child also accumulating the partial from the left.

This *scan-then-propagate* approach is generalizable to arbitrary bases. Recursive blocks of operators can be replaced with “tiles” of localized  $b$ -element upsweep scans and  $b$ -way downsweep distributions. The scan implementations by Sengupta et al. [80, 101, 102] and Thrust [113] employ this approach, typically with  $b=1024$  elements per tile.

Fig. 27 illustrates this operation with  $b=4$  elements per tile. Each upsweep tile reads  $b$  inputs and writes  $b$  scan results back to global memory. The downsweep



**Fig. 28.** Example operation of a fully-recursive *reduce-then-scan* CTA decomposition in which each CTA processes  $b=4$  values.

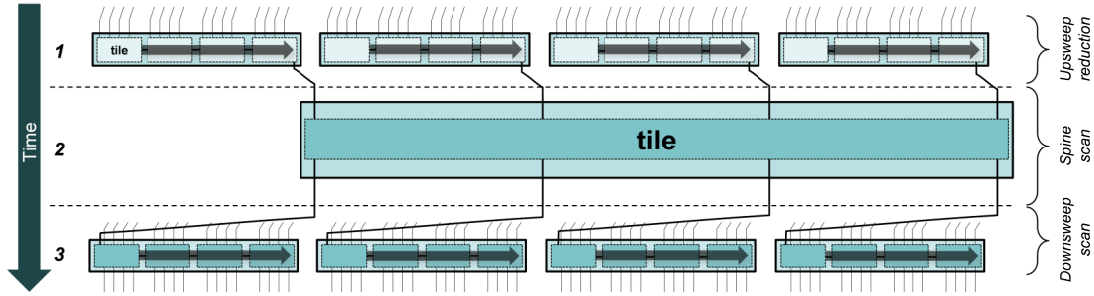
propagation phase unwinds the recursion. Each “fix-up” tile reloads the  $b$  intermediate results from the corresponding upsweep scan tile, aggregates its incoming value from the preceding level into each, and writes the updated  $b$  values back out to global memory.

Both upsweep and downsweep phases comprise complete  $b$ -ary trees having  $\log_b n$  kernel launches and  $(n-1)/(b-1)$  tiles. Each tile requires  $2b$  memory accesses, resulting in  $4b(n-1)/(b-1)-2b$  overall memory traffic.

#### 4.3.2 Reduce-then-scan

The *reduce-then-scan* decomposition is similar in that it also entails  $\log_b n$  levels of kernels, but instead executes reduction kernels during the upsweep phase followed by scan kernels during the downsweep phase. This technique was first popularized for the Cray Y-MP by Chatterjee et al. [26] and demonstrated more recently for the GPU by Dotsenko et al. [42].

Fig. 28 illustrates this operation with  $b=4$  elements per tile. Each reduction tile reads  $b$  inputs, aggregates them, and writes a single intermediate result back to global device memory. The intermediate values computed during the reduction kernels are not saved and must be recomputed later. The downsweep phase unwinds the recursion. Each



**Fig. 29.** Example operation of a *two-level reduce-then-scan* CTA decomposition in which each CTA processes  $b=4$  values.

tile performs a scan of the  $b$  partial reductions used as inputs to the corresponding upsweep reduction tile, seeded with the partial reduction from the preceding level.

As with scan-then-propagate, an  $n$ -element reduce-then-scan scan problem will require  $2(n-1)/(b-1)-1$  tiles. This approach has a performance advantage in that each upsweep tile only produces one output (instead of  $b$ ), halving the memory traffic for the upsweep. The result is  $3b(n-1)/(b-1)-b$  overall memory traffic. At the expense of performing some redundant calculations during the downsweep phase, the reduce-then-scan strategy moves 25% fewer bytes through global memory than scan-then-propagate.

#### 4.3.3 Two-level reduce-then-scan

We employ our *CTA-serialization* idiom (Chapter 3.3.1) to implement a *two-level reduce-then-scan* decomposition of CTAs. Instead of allocating a unique CTA for every tile, we simply dispatch a fixed number of  $C$  CTAs in which threads are “reused”. We choose  $C$  large enough to saturate all GPU cores.

Fig. 29 illustrates the operation of a two-level reduce-then-scan implementation with  $b=4$  elements per tile. The outer-level upsweep reduction produces an inner-level scan problem (the “spine”) having  $C$  inputs. The spine is small enough to scan using a single CTA. Once completed,  $C$  downsweep CTAs are dispatched to perform the



independent outer-level scan tiles, seeded with the appropriate aggregate from the spine. When reducing or scanning sequential tiles, each CTA simply carries the aggregate partial reduction from one tile to the next within registers.

The result is that an entire  $n$ -element computation requires only  $3n+3C$  global memory accesses. In comparison with the fully-recursive approaches, the advantages of our two-level strategy are threefold:

- i. Asymptotically fewer kernel launches: three versus  $O(\log_b n)$ . Most importantly, the undersubscribed interior is completed as quickly as possible.
- ii. Asymptotically fewer global memory accesses for intermediate values:  $3C$  versus  $O(n)$ .
- iii. A constant amount of temporary storage (versus  $O(n)$  intermediate storage).

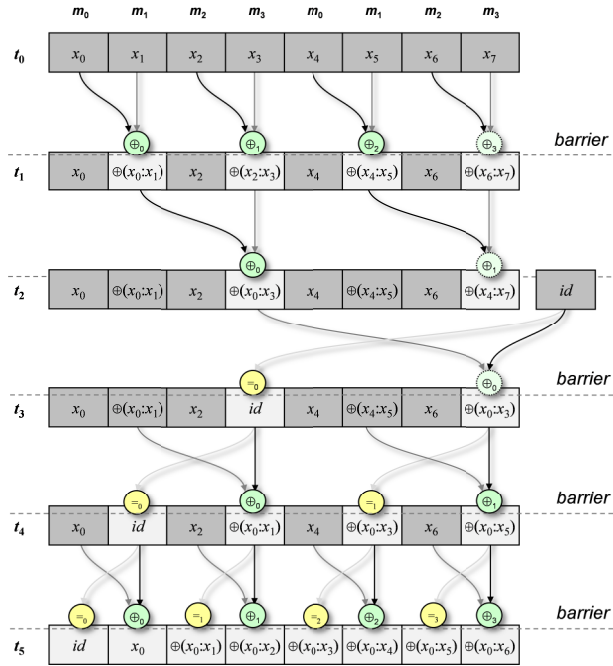
In independent work on parallel compaction, Billeter et al. have proposed similar two-level CTA decompositions [15].

#### 4.4 LOCAL PREFIX SCAN

As described in the previous section, our global scan comprises two types of CTA-wide tile-processing routines: local reduction during upsweep and local scan during downsweep. Chapter 3.3.2 describes our parallelization strategy for reducing upsweep tiles. In this section, we present and evaluate two variants for scanning downsweep tiles, each having different degrees of granularity coarsening: *reduced-conflict Brent-Kung* (RCBK) and *sequential-reduce-then-scan* (SRTS).

##### 4.4.1 Reduced-conflict Brent-Kung (RCBK)

Scan isomorphs to the Brent-Kung construction are commonly implemented using the Blelloch PRAM algorithm presented in Listing 4 [16]. Unfortunately this parallelization



**Fig. 30.** Example operation of Blelloch exclusive scan algorithm on  $n=8$  elements using four threads. Same-colored data flow arrows indicate potential bank conflicts.

**Listing 4.** Blelloch PRAM exclusive scan algorithm

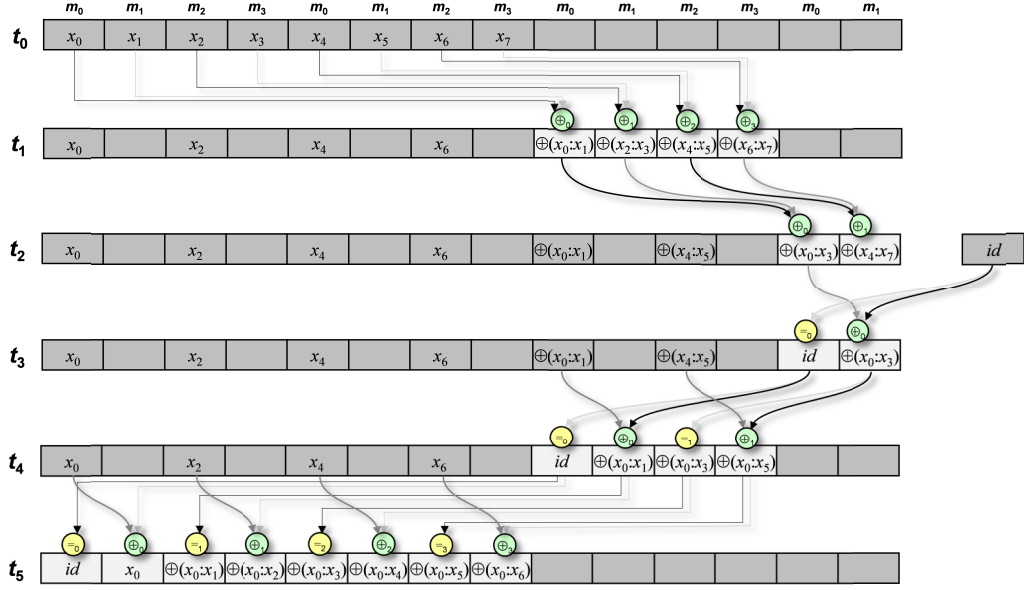
```

1  for d := 0 to  $\log_2 n - 2$ 
2  do
3    for k := 0 to n-1 by  $2^{d+1}$  in parallel
4    do
5      // upsweep into parent
6       $m[k + 2^{d+1} - 1] :=$ 
7         $m[k + 2^{d+1} - 1] + m[k + 2^d - 1];$ 
8    od
9  od
10
11  $m[n - 1] := m[n/2 - 1];$ 
12  $m[n/2 - 1] := id;$ 
13
14 for d :=  $\log_2 n - 2$  downto 0
15 do
16   for k := 0 to n-1 by  $2^{d+1}$  in parallel
17   do
18     temp :=  $m[k + 2^d - 1];$ 
19
20     // downsweep into left child
21      $m[k + 2^d - 1] := m[k + 2^{d+1} - 1];$ 
22
23     // downsweep into right child
24      $m[k + 2^{d+1} - 1] := temp +$ 
25        $m[k + 2^{d+1} - 1];$ 
26   od
27 od

```

produces progressively worse bank conflicts in GPU shared memory, leading to excessive thread serialization. Fig. 30 illustrates this behavior for  $w_{\text{SIMD}} = 4$  threads and four shared memory banks. Every memory access performed in time step  $t_1$  incurs two-way conflicts in banks  $m_1$  and  $m_3$ , and every access in  $t_2$  incurs perfectly degenerate four-way conflicts in  $m_3$ .

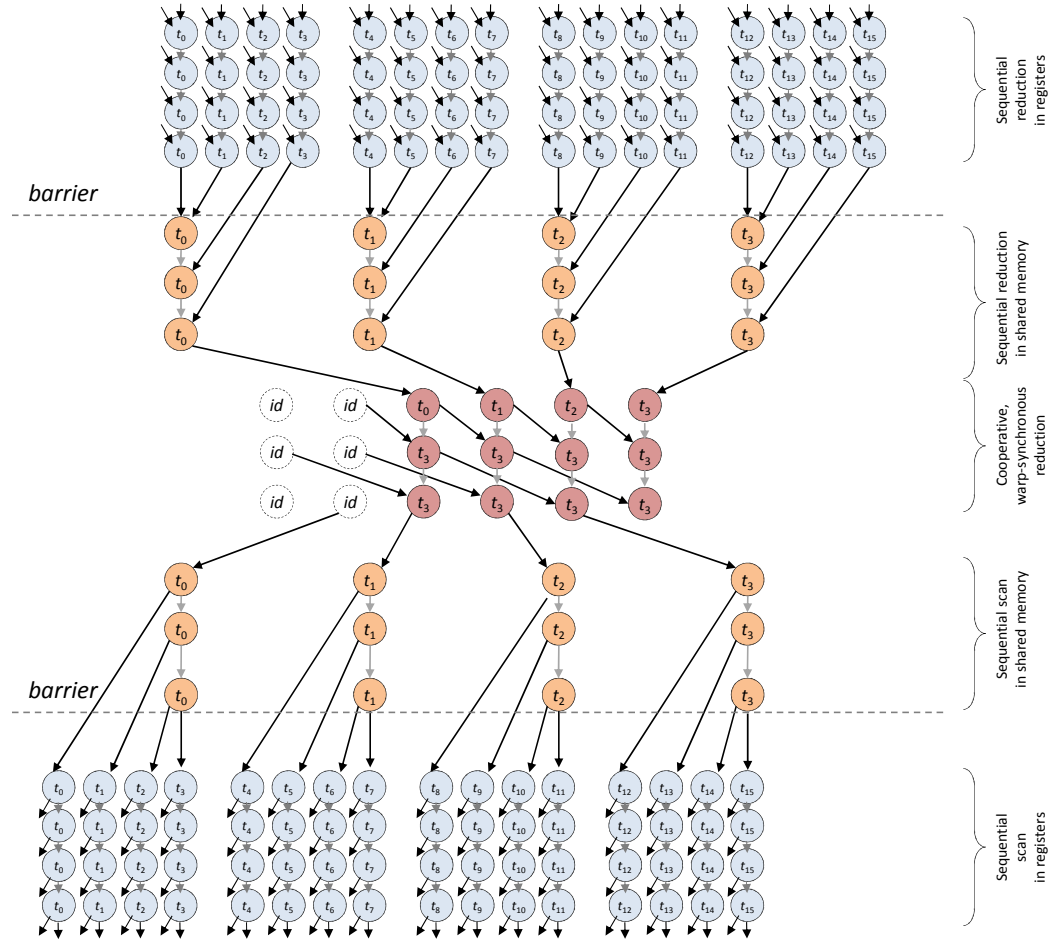
In prior work, Harris et al. addressed this problem with the insertion of aperiodic padding cells having geometric progression factor  $1/32$ . In the limit, this approach requires  $O(n/32)$  extra padding. However, the additional instruction overhead for compensating the addressing offsets resulted in overall slowdown compared to the conflict-ridden original. [80]



**Fig. 31.** Example operation of our *reduced-conflict Brent-Kung* (RCBK) exclusive scan algorithm on  $n=8$  elements using four threads. Same-colored data flow arrows indicate 2-way bank conflicts.

As an alternative, we present a different PRAM isomorph of Brent Kung in Fig. 31 that only suffers two-way bank conflicts. Whereas each step of the Brelloch algorithm operates by reusing the storage of the previous, our approach does not. Instead, we use  $n-2$  extra memory cells for the storage of partial reductions and maintain a two-element stride between threads for all memory accesses. Unlike geometric padding approaches that dynamically compute addressing offsets as a function of thread-rank, ours can be unrolled as static constants.

This RCBK parallelization is designed for low-latency scans. It carries a higher instruction overhead than our SRTS parallelization (presented in the next subsection) because threads must barrier and conditionally deactivate at each time step. However, it maintains a much wider average degree of parallelism than SRTS, making it preferable for quickly retiring undersubscribed, single-tile scenarios such as the upper-level spine kernel in Fig. 29.



**Fig. 32.** Example operation of our conflict-free *sequential-reduce-then-scan* (SRTS) scan algorithm on  $n=64$  elements using 16 threads and a warp width  $w_{\text{SMD}}=4$  threads.

#### 4.4.2 Sequential-reduce-then-scan (SRTS)

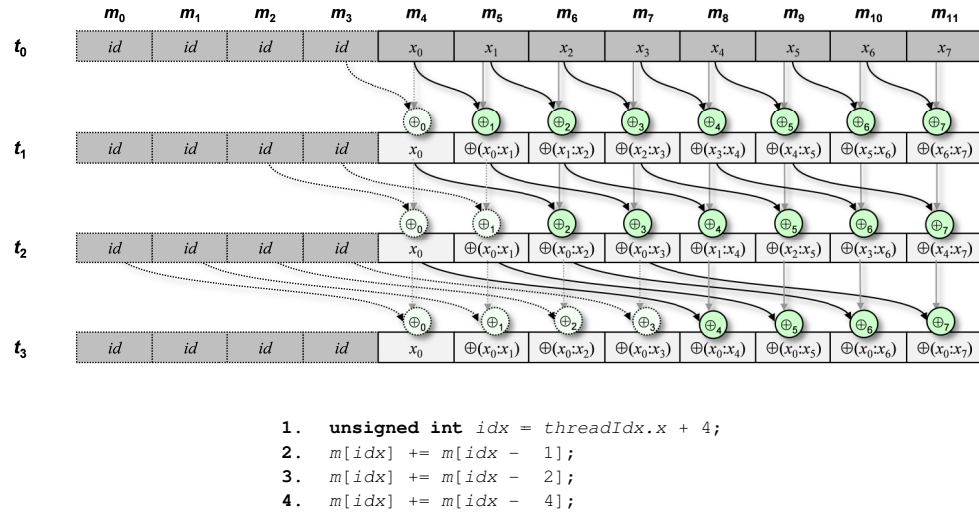
Fig. 32 illustrates our sequential-reduce-then-scan (SRTS) parallelization for local prefix scan. This parallelization prioritizes efficiency, employing the thread-serialization techniques we described for local reduction in Chapter 3.3.2. Similarly, SRTS incorporates different phases of computation, each designed to either increase the amount of sequential work within a given storage class (e.g., registers, shared memory, etc.) or exploit a particular aspect of the abstract machine model (e.g., lock-step thread progress within the warp):

- 1) *Sequential reduction in registers.* The accumulated partial reductions are then placed in a grid of shared memory.
- 2) *Sequential reduction in shared memory.* The parallelism is reduced to a single warp which performs a sequential raking reduction in each thread. Shared memory is periodically padded to avoid bank conflicts (Chapter 3, Fig. 21).
- 3) *Cooperative warp-synchronous scan.* The single raking warp performs a synchronization-free scan of the partials reduced in the previous phase. We describe this “warpscan” procedure in more detail in the next subsection.
- 4) *Sequential scan in shared memory.* The raking warp performs a sequential scan of the original partials placed into the grid, seeded with the exclusive prefixes computed by the warpscan.
- 5) *Sequential scan in registers.* The entire CTA reactivates, each thread performing a sequential reduction of its inputs, seeded with its exclusive prefix computed by the raking warp in the shared grid.

Dotsenko et al. have previously demonstrated similar granularity coarsening for local prefix scan [42]. Whereas we only implement a single shared memory raking phase, their approach incorporates several (32-wide, 8-wide, and 1-wide stages). The overhead from extra exchanges between raking grids and unused SIMD lanes prevents their implementation from being bandwidth-bound.

#### 4.4.3 SIMD Optimizations

The Kogge-Stone construction (Fig. 26b) works by progressively building partial reductions from consecutive inputs. The strategy is easily implemented in software for PRAM architectures [61], including GPUs [37, 59]. Unfortunately the construction



**Fig. 33.** The operation of an unrolled, divergence-free three-level SIMD “warpscan” for an input size  $n = 8$ .

suffers from suboptimal work complexity and write-after-read anti-dependences that typically necessitate double-buffering for safe operation.

However, Kogge-Stone variants can be very efficient when the problem size is smaller than or equal to the SIMD width. Fig. 33 illustrates an intra-warp construction commonly known as “warpscan” [80, 101].

The lock-step progression of threads precludes any anti-dependence hazards or undesired read/write interleavings. No explicit programmatic synchronization is needed. Furthermore, the addition of shared memory padding populated by identity values can preclude control flow divergence by relieving threads from having to conditionally deactivate during successive time steps. (There is no benefit for such deactivation during warpscan – the SIMD lanes are occupied regardless.)

Because of its speed and efficiency, warpscan is an attractive building block for constructing hybrid scan strategies. As an optimization, both RCBK and SRTS parallelizations incorporate a single warpscan at the “center” of their computations where

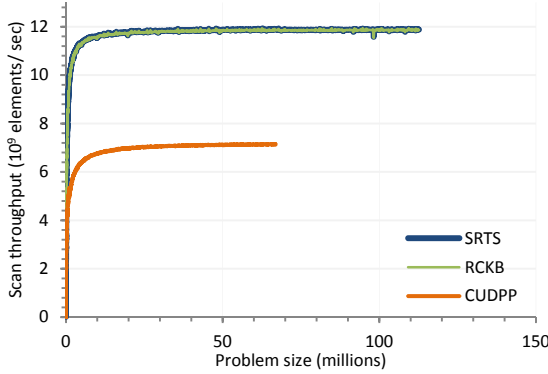


Fig. 34. Global prefix sum throughput (overall)

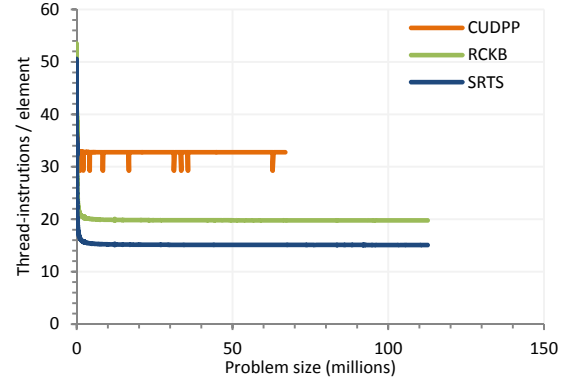


Fig. 35. Global computational overhead (overall)

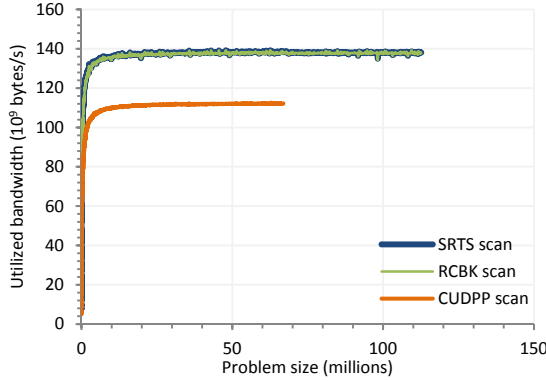
the concurrency drops below the warp size. However, the profligate use of warpscan as a recursive tiling subroutine would be prohibitively expensive.

#### 4.4.4 Evaluation

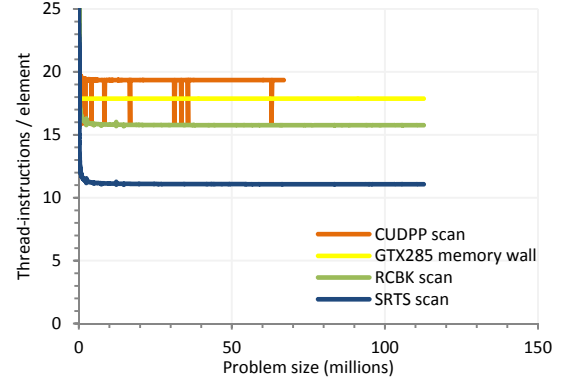
We evaluated our RCBK and SRTS scan constructs in the context of 32-bit global prefix sum alongside the reference CUDPP v1.1 implementation. We used an NVIDIA GTX285, sampling performance measurements directly from the GPU’s hardware counters (which do not include driver and staging overheads). We present performance data for 2,000 problem-instances having problem-size sampled log-normally from the interval  $[2^5, 2^{27}]$ .

**Overall scan throughput.** Fig. 34 plots global throughput as a function of problem size. This analysis reflects the cumulative elapsed time for all kernel invocations executed. For large saturating problem sizes (32M+ elements), CUDPP averages 7.1B elements/sec. Our SRTS and RCKB-based scans both average 11.9B elements/sec, a speedup of 1.7x.

By construction, the CUDPP implementation is less efficient due to its *scan-then-propagate* CTA decomposition. Other performance and utilization concerns aside, our



**Fig. 36.** Scan kernel utilized bandwidth



**Fig. 37.** Scan kernel computational overhead

*two-level reduce-then-scan* implementations should only be 1.3x faster by virtue of imposing only  $\frac{3}{4}$  as much memory traffic. The remainder of CUDPP slowdown derives from processor underutilization from inner kernel launches and excessive computational overhead.

**Overall computational overhead.** Fig. 35 plots the number of instructions per element scanned as a function of problem size. For saturating problem sizes, CUDPP averages 32.7 thread-instructions/element. Our SRTS and RCKB-based scans average 14.9 and 19.0 thread-instructions/element, respectively. Our SRTS implementation is 1.7x and 1.3x more efficient than CUDPP and RCKB, respectively.

**Scan kernel throughput.** Drilling down into the tile-scanning kernels, Fig. 36 plots the utilized bandwidth for each as a function of problem size. For saturating problem sizes, the CUDPP scan kernel averages 112 GB/s, 81% bandwidth utilization. Our SRTS and RCKB-based scans both average 138 GB/s, matching the peak-achievable for the GPU (Chapter 2, Table 1).

**Scan kernel computational overhead.** Fig. 37 confirms the CUDPP scan kernel to be compute-bound at 19.3 thread-instructions per 32-bit element. It is 8% higher than



the GTX285 memory wall for scan kernels, which is plotted below in yellow at 17.9 thread-instructions/element<sup>11</sup>. Our SRTS and RCKB-based scans are both bandwidth-bound, averaging only 15.8 and 10.9 thread-instructions/element, respectively. Our SRTS kernel is 1.8x more efficient than CUDPP and provides substantial opportunity for kernel fusion.

***Single-CTA tile-processing latency.*** Low-latency prefix scan parallelizations are preferable for fleeting workloads incapable of saturating GPU cores. For a single CTA on an otherwise idle GPU, we used cycle counters to measure the average number of cycles required to locally scan tiles of 512 elements.

Interestingly enough, the more efficient (yet deeper) SRTS parallelization is also quicker on the NVIDIA GT200 architecture (GTX280):

- RCBK: 2,441 cycles
- SRTS: 1,472 cycles

However, the opposite is true for the newer GF100 architecture (GTX480):

- RCBK: 1,452 cycles
- SRTS: 1,660 cycles

Because shared memory is relatively “further away” on GF100, more local parallelism is needed to overlap the additional latency. The disparity in performance response between the two architectures underscores the importance of having flexible primitives capable of autotuned algorithm selection.

---

<sup>11</sup>  $(30 \text{ cores} * 8 \text{ SIMD lanes per core} * 1.48\text{GHz clock} * 8 \text{ bytes of traffic per 32-bit element}) / (159\text{GB/s}) = 17.9 \text{ thread-instrs/element.}$

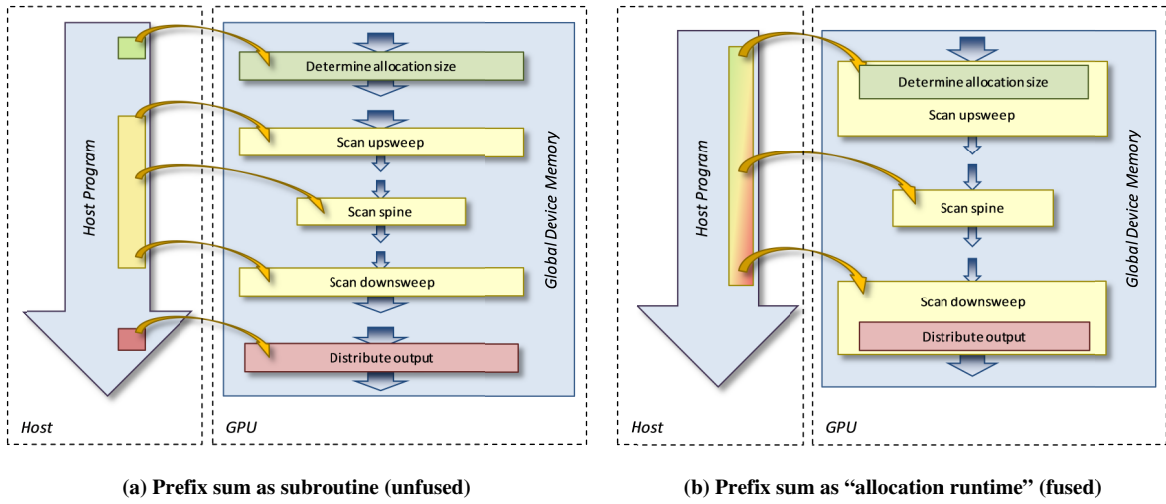


Fig. 38. Kernel fusion of application code into scan kernels

#### 4.5 KERNEL FUSION

Prefix scan was originally promoted as a software primitive for vector and array processor architectures more than two decades ago [17, 61]. While GPU architectures share many traits with these forebears, the relatively new combination of deeply multithreaded scheduling and local scratchpad memories has altered the algorithm design landscape.

These features enable *kernel fusion*, a design idiom where functional encapsulation is broken and multiple dataflow transformations are encoded within the same kernel procedure. Within the context of this dissertation, kernel fusion leverages three key ideas:

- 1) Bandwidth-bound kernels leave compute resources underutilized. However, the deep multithreading of the GPU makes it easy to fill these cycles with useful work, particularly when that work can be done within local scratch memories.

- 2) Live state is expensive to move out to DRAM only to have it read back in again by the next kernel. We reduce instruction counts, bandwidth demand, and energy by retaining such state in registers or shared memory for the next computation.
- 3) Bandwidth-bound prefix sum kernels form a nice runtime abstraction for performing allocation-oriented stream transformations. Fig. 38 illustrates the fusion of allocation problems *within* prefix sum kernels. This saves the overhead of writing allocation counts out to global memory, invoking a separate kernel, and then reading the computed reservation offsets back in again.

As examples of such kernel fusion, this section presents derivatives of prefix sum that implement segmented scan, duplicate removal, reduce-by-key, and histogram. We demonstrate the advantages of this idiom by comparing our performance with equivalent functionality implemented within the Thrust library of parallel primitives [113].

#### 4.5.1 *Segmented scan*

The *segmented scan* problem is a composition of independent scan instances. Typically these subproblems are concatenated within a single large input array and are delineated by a second input array of head-flags. Segmented scan is a particularly useful primitive for many top-down partitioning problems, e.g., parallel quicksort, acceleration structures, etc. [16, 26, 92].

Fig. 39 compares our B40C segmented scan throughput versus Thrust for 128MB problem sizes on the NVIDIA GTX280. We exhibit a 3.5x harmonic mean speedup across data types.

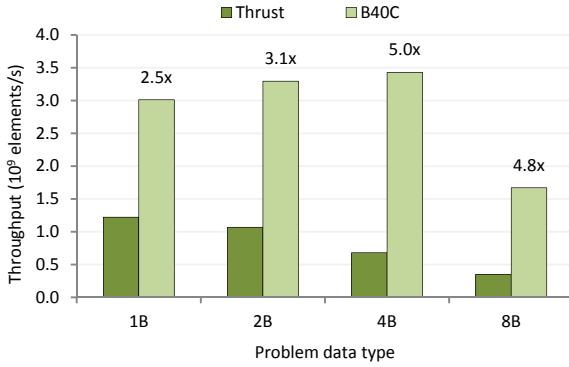


Fig. 39. Segmented scan throughput (GTX280)

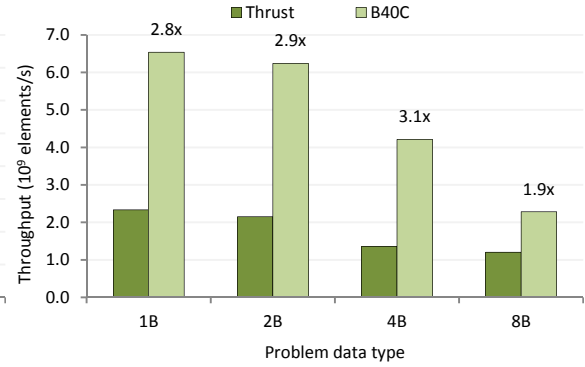


Fig. 40. Duplicate removal throughput (GTX280)

#### 4.5.2 Duplicate removal

Duplicate removal entails a straightforward fusion with prefix sum. Tiles of keys are read and local flags are generated to mark discontinuities between keys and their predecessors. We then perform a prefix sum over the flags to compute the scatter offsets for writing the corresponding keys into the output stream. The flag vector is never wholly realized in global memory: we simply (re)generate the flags for both upsweep and downsweep kernels.

Fig. 40 compares our B40C segmented scan throughput versus Thrust for 128MB problem sizes on the NVIDIA GTX280. We exhibit a 2.6x harmonic mean speedup across data types.

#### 4.5.3 Reduce-by-key

*Reduce-by-key* is the third phase of the map-reduce paradigm (after mapping and sorting) [38]. Given a list of key-value pairs, it is analogous to a segmented reduction over the values where the segments are defined by regions of consecutive, identical keys.

We perform reduce-by-key using a variant of segmented prefix scan. Tiles of keys are read and local head-flags are generated where discontinuities are observed. We

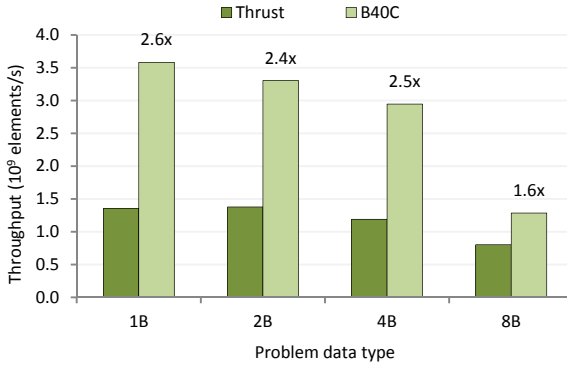


Fig. 41. Reduce-by-key throughput (GTX280)

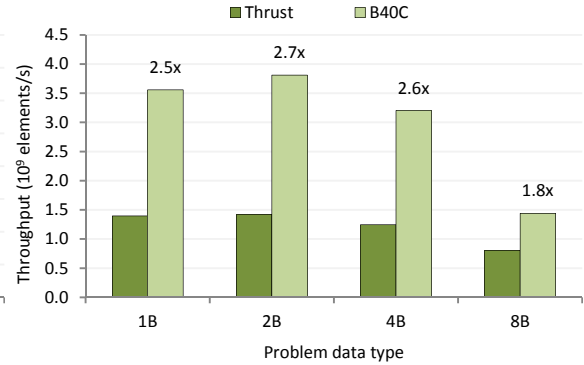


Fig. 42. Histogram throughput (GTX280)

then perform a segmented scan over the corresponding values and head-flags, yet we also compute a prefix sum of the head-flags. The scanned head-flags now convey the scatter offsets for writing the corresponding value reductions into the output stream.

Fig. 41 compares our B40C segmented scan throughput versus Thrust for 128MB problem sizes on the NVIDIA GTX280. We exhibit a 2.2x harmonic mean speedup across data types.

#### 4.5.4 Histogram

GPUs have been criticized for their lack of suitable atomics for performing contended updates to histogram counts [76]. An alternative strategy that is perhaps better suited to bulk-synchronous architectures entails first sorting the keys and then performing a variant of reduce-by-key where the associated values are all implicitly 1. As we describe in Chapter 5, GPUs are particularly adept at high performance sorting. Furthermore, the early-exit sorting optimizations we present are well-suited to the types of low-entropy key distributions common to histogram problems.

Fig. 42 compares our B40C histogram construction throughput versus Thrust for 128MB problem sizes for previously-sorted datasets on the NVIDIA GTX280. We exhibit a 2.3x harmonic mean speedup across data types.

We note that we can radix sort uniform-random 32-bit keys on the GTX280 at 534M keys/s. Paired with our consecutive-histogram rate of at nearly 3B keys/s, we can compute histograms of at an overall rate of 450M keys/s. This is a 100x speedup versus the naive approach of global atomics, which is rate-limited at 4.5M atomics/s (Chapter 2, Table 1).

#### 4.6 CHAPTER SUMMARY

Parallel prefix scan is core algorithmic primitive for constructing parallel programs. Due to the relatively fine-grained ratio of computation to operand size (e.g., one addition 32-bit addition instruction versus 12 bytes of operands and results), GPU parallelizations of prefix scan ought to be bandwidth-bound. However, this was not the case with prior work.

In this chapter, we presented new implementations for local prefix sum that are 1.7x more computationally-efficient than previous parallelizations. Our constructions make extensive use of the granularity coarsening idioms we developed in Chapter 2. These design idioms additionally provide the dual benefits of software-reuse and tuning flexibility. Furthermore, our global scan primitives are 25% more efficient with memory bandwidth and less apt to incur processor underutilization.

The *kernel fusion* design idiom seeks to maximally utilize both memory and computational resources on the GPU. Because the computational overhead of prefix sum now resides below the memory wall, we have created an inflection point in the design

space for many cooperative problems where kernel fusion within prefix sum becomes worthwhile. To illustrate the effectiveness such fusion, we have constructed high performance implementations of segmented scan, duplicate removal, histogram, and reduce-by-key that provide several factors of speedup over contemporary parallelizations within Thrust [113].

## Chapter 5

### *Radix Sorting*

#### 5.1 INTRODUCTION

High performance sorting is particularly desirable for GPU architectures. As a fundamental algorithmic primitive, sorting often plays an explicit role in more sophisticated algorithms. Algorithms that produce or transform data frequently need to subsequently rank, organize, or partition that data in some fashion. While GPU architectures are particularly adept at data-independent transformations, it is less obvious that they would be equally adroit at sorting, a list-processing operation that is inherently cooperative. Because of the large problem sizes typical of GPU applications, inefficient sorting can be a major bottleneck of overall application performance.

In addition, sorting can play a performance-enhancing role in many parallelizations of serial algorithms. Discretionary sorting has the potential to serve as a “bandwidth amplifier” for problems involving pointer-chasing and table lookups. Reorganizing either tasks or data can realize better mappings between the two that exhibit significantly improved spatial and temporal locality. Sorting (or binning) is a common technique for smoothing otherwise incoherent memory accesses. Similarly, local sorting



within distributed settings can play a crucial role in preparing data on one processor for batched distribution to its peers.

Although parallel sorting methods are highly concurrent and have small computational granularities<sup>12</sup>, sorting on GPUs has been perceived as challenging, particularly in comparison with conventional multi-core CPU architecture. As a list-processing transformation, sorting has irregular and global data-dependences. The placement of a given input item will depend upon the value of every other input element. As discussed in §2.6, such fine-grained global allocation dependences are representative of the cooperative workloads that many researchers feel poorly suited for GPU architecture. The work described within this chapter refutes this popular opinion, providing evidence that GPUs are exceptional platforms for sorting operations.

In particular, we focus on the problem of sorting large sequences of elements, specifically sequences comprised of hundreds-of-thousands or millions of fixed-length numeric keys. The methods we present within this chapter address two problem variants: (a) 32-bit integer keys paired with 32-bit satellite values; and (b) 32-bit keys only. Our solution strategy generalizes for other problem types as well: as a C++ template implementation, our algorithm can be parameterized using any C++ primitive type as key types and arbitrary user-defined structures for value types.

Our work as described in this chapter makes contributions in the following areas:

***Parallelization strategy.*** We present a GPU parallelization for radix sorting passes that is constructed within a “multi-scan runtime” for computing multiple concurrent prefix sums, one for each partitioning bin. The granularity of our approach is more tunable than prior work, requiring memory traffic that is inversely proportional to

---

<sup>12</sup> E.g., comparison operations for comparison-based sorting, or shift and mask operations for radix sorting.

the number of radix bits per digit. This provides flexibility for future improvements in computational throughput. We also describe a novel optimization for early termination that significantly improves performance for commonplace sorting problems having banded key diversity.

***High performance.*** Our tunable implementation achieves multiple factors of speedup over prior GPU sorting implementations across all generations programmable NVIDIA GPUs. We demonstrate sustained sorting rates in excess of 1.2 billion 32-bit keys/sec and 342 million 64-bit keys/sec. To our knowledge, these sorting rates are the fastest published for any fully-programmable microarchitecture. Put in context, state-of-the-art CPU parallelizations achieve 240 million 32-bit keys/sec [98] and reconfigurable FPGAs have demonstrated 250 million 64-bit keys/sec [73].

***Impact.*** Our radix sorting implementation is incorporated within the Thrust parallel template library [113]. Thrust is a high-profile, productivity-oriented library that is bundled with the NVIDIA CUDA software development toolkit [34]. Furthermore, tuned versions of our implementation are specifically incorporated within the AMBER 11 molecular simulation tools [22], the NVIDIA Optix ray tracing engine [88], and the LibBSC lossless compression suite [60].

## 5.2 BACKGROUND

### 5.2.1 GPU sorting applications

Sorting is germane to many problems in computer science. As an algorithmic primitive, sorting facilitates many problems including binary search, finding the closest pair, determining element uniqueness, finding the  $k^{\text{th}}$  largest element, and identifying outliers [33, 72]. The use of sorting for reorganizing sparse data structures figures prominently in

sparse matrix-matrix multiplication [12]. For large-scale problems on distributed memory systems (e.g., graph algorithms for clusters and supercomputers [32]), sorting plays an important role in improving communication efficiency by coalescing messages between nodes. In Chapter 6, we leverage sorting for batching communication between GPUs for multi-node graph traversal.

Recent literature has demonstrated many applications of GPU sorting. Sorting is a procedural step during the construction of acceleration data-structures, such as octrees [75], KD-trees [124], and bounding volume hierarchies [92]. These structures are often used when modeling physical systems, e.g., molecular dynamics, ray tracing, collision detection, visibility culling, photon mapping, point cloud modeling, particle-based fluid simulation, n-body systems, etc. GPU sorting has found many applications in image rendering, including shadow and transparency modeling [104], Reyes rendering [123], volume rendering via ray-casting [68], particle rendering and animation [31, 71], ray tracing [46], and texture compression [23]. GPU sorting has also been demonstrated for parallel hashing [4], database acceleration [51, 57, 58], data mining [52], and game engine AI [103].

### ***5.2.2 Parallel sorting networks and the impracticality of output-oriented design***

The GPU machine model is designed for output-oriented, stencil-based decompositions. Threads are logically defined by the specific output elements they are to produce, and kernel programs statically encode input and output locations as a function of thread rank (independent of the computation of other threads).

However, this output-oriented focus can pose difficulties for sorting, a problem having global input dependences. The value to be placed at a given output location is

dependent upon the value of every location within the input list. Without cooperation from other threads, all  $n$  threads would need to perform their own  $O(n)$  selection algorithm [33] to determine what value to write, resulting in quadratic overall workload. Cooperation is a critical component of any efficient sorting parallelization.

It is possible to construct sorting from the repeated application of output-oriented kernels, exclusively. Such solutions are called *sorting networks*. The threads in each stencil kernel perform pair-wise swapping operations. Because the input and output locations of each thread are encoded as a function of thread rank, the sequence of comparisons and flow of data through memory are statically known in advance.

Unfortunately, known sorting network constructions are asymptotically and/or practically inefficient. The simple pair-wise swapping example in §1.3.2, Fig. 6 implements  $O(n^2)$  work and is isomorphic to bubble/insertion sort. Variations of Batcher’s bitonic sorting network have size  $O(n\log^2 n)$  [10]. Although variants of the AKS sorting network [3] have optimal size  $O(n\log n)$ , they have extremely large big- $O$  constants that prevent their practical usage. It is an open question as to whether practical  $O(n\log n)$  size sorting networks exist.

Instead, we advocate an input-oriented decomposition for implementing work-optimal sorting. From the thread perspective, we want to logically associate tasks with specific elements in the input stream. In the context of partitioning-based sorting methods, each thread gathers its key, determines which partition that key belongs, and then must cooperate with other threads to determine where the key should be relocated. By shifting the focus to input items, these problems can all be reduced to the problem of cooperative allocation.

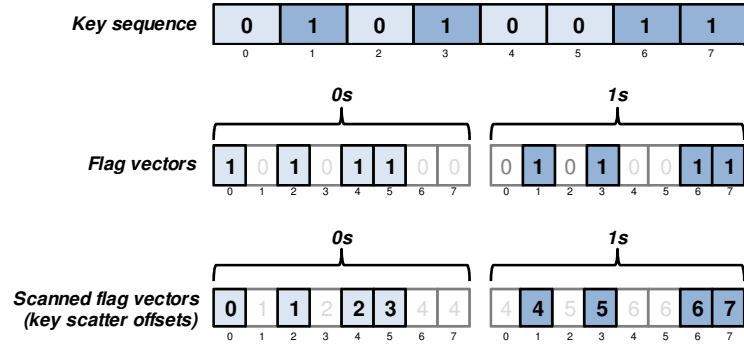
### 5.2.3 The radix sorting method

Radix sorting is currently the fastest approach for sorting 32- and 64-bit keys on both CPU and GPU processors [84, 98]. The method relies upon a positional representation for keys, i.e., each key is comprised of an ordered sequence of numeral symbols (i.e., *digits*) specified from least-significant to most-significant. For a given input sequence of keys and a set of rules specifying a total ordering of the symbolic alphabet, the radix sorting method produces a lexicographic ordering of those keys.

The process works by iterating over digit-places within the keys from least-significant to most-significant. For each digit-place, the method performs a stable *distribution sort* of the keys based upon their digit at that digit-place in order to partition the keys into radix  $r$  distinct buckets. Given an  $n$ -element sequence of  $k$ -bit keys,  $d$ -bit digits, and  $r = 2^d$ , a radix sort of these keys will require  $k/d$  passes of distribution sorting.

The asymptotic work complexity of the distribution sort is  $O(n)$ . Each of the  $n$  input items needs comparing with only a fixed number of radix digits. With a fixed number of digit-places, the entire radix sorting process is also  $O(n)$ . When a key is relocated, its global relocation offset is computed as the number of keys with “lower” digits at that digit place plus the number of keys having the same digit, yet occurring earlier in the sequence.

Radix sorting has a variable granularity of computation, i.e., it can trade more computation for less memory traffic. Increasing  $d$  (the number of bits per radix-digit) decreases the total number of digit-place passes that need iterating over. For example, a sort of 32-bit integers can be performed using thirty-two 1-bit distribution passes (each partitioning into  $r=2$  bins). By using 4-bit distributions instead (each partitioning into  $r=16$  bins), we can perform the same sort in only eight overall passes.



**Fig. 43.** The traditional split operation: a decoding step combined with prefix scan reveals the scatter offsets required to enact a radix  $r = 2$  distribution sort on the first digit-place of an input sequence.

However, the number of bins  $r$  scales exponentially with  $d$ . This implies that linear decreases in global memory traffic will result in super-linear increases in dynamic instruction counts and local storage requirements.

#### 5.2.4 Parallel radix sorting

The fundamental component of the radix sorting method is the distribution sort in which  $n$  keys are scattered into  $r$  bins. Parallel radix sorting is comprised of  $k/d$  iterations of parallel distribution sorting passes.

Because key distribution is typically unknown, the sizes and memory layout of the bins for a given distribution pass must be dynamically determined. There are two strategies for constructing bins: (1) using blocks that are allocated online and linked with pointers; and (2) contiguous allocation in which offsets and lengths are computed *a priori* using a parallel prefix scan algorithm. Most research is focused on the latter: the ability to perform contention-based allocation is non-existent or severely expensive on many parallel architectures (e.g., vector, array, and GPU processors), and traversing linked structures can carry stiff performance penalties.

**Scan-based approach.** In its simplest form, a distribution sort can be implemented using a binary *split* primitive [26] comprised of two prefix scans over two  $n$ -element binary encoded flag vectors: the first initialized with 1s for keys whose digit was 0, the second to 1s for keys whose digit was 1. The two scan operations are dependent: the scan of the 1s vector can be seeded with the number of zeros from the 0s scan. After the scans, the  $i^{\text{th}}$  element in the appropriate flag vector will indicate the relocation offset for the  $i^{\text{th}}$  key. An alternative is to perform one large scan over the concatenation of the two vectors, as shown in Fig. 43.

As described in Fig. 44, a naive GPGPU distribution sort implementation can be constructed by simply invoking a parallel prefix scan primitive between separate *decoding* and *scatter* kernels. The decoding kernel would be used to create a concatenated flag vector of  $rn$  elements in global memory. After scanning, the scatter kernel would redistribute the keys (and values) according to the scan results.

This approach suffers from an excessive memory workload that scales with  $2^d/d$ . As such, the overall memory workload will be minimized when the number of radix digit bits  $d = 1$ . This provides little flexibility for tuning the sorting granularity to minimize (and overlap) the memory and computational workloads.

**Histogram-based approach.** As an alternative, practical sorting implementations have used a histogram-based strategy [43, 121]. For typical parallel machines, the number of parallel processors  $p \ll n$ . This makes it natural to distribute the input sequence amongst processors. Using local resources, each processor can compute an  $r$ -element histogram of digit-counts. By only sharing these histograms, the global storage requirements are significantly reduced. A single prefix scan of these histograms provides

<i>Naive GPU distribution sort</i>		
<i>Kernel</i>	<i>Read I/O Workload</i>	<i>Write I/O Workload</i>
1. Decode keys and compute flag vectors	$n$ keys	$nr$ counts
2. Flag scan: upsweep reduction	$nr$ counts	(insignificant)
3. Flag scan: spine scan	(insignificant)	(insignificant)
4. Flag scan: downsweep scan	$nr$ counts + (insignificant)	$nr$ offsets
5. Scatter keys to appropriate bin	$nr$ offsets + $n$ keys	$n$ keys
<b>Total I/O for all <math>k/d</math> passes:</b>		$(k/d) (5n(2^d) + 3n)$

<i>GPU histogram-based distribution sort [5,7]</i>		
<i>Kernel</i>	<i>Read I/O Workload</i>	<i>Write I/O Workload</i>
1. Locally sort blocks at current digit-place into digit-segments	$n$ keys	$n$ keys
2. Compute block histograms of digit counts	$n$ keys	$nr/b$ counts
3. Histogram scan: upsweep reduction	$nr/b$ counts	(insignificant)
4. Histogram scan: spine scan	(insignificant)	(insignificant)
5. Histogram scan: downsweep scan	$nr/b$ counts + (insignificant)	$nr/b$ offsets
6. Scatter sorted digit-segments of keys to appropriate bin	$nr/b$ offsets + $n$ keys	$n$ keys
<b>Total I/O for all <math>k/d</math> passes:</b>		$(k/d) (5n(2^d)/b + 7n)$

<i>Our GPU allocation-oriented distribution sort</i>		
<i>Kernel</i>	<i>Read I/O Workload</i>	<i>Write I/O Workload</i>
1. Allocation scan: upsweep reduction (locally decode and reduce flag counts)	$n$ keys	(insignificant)
2. Allocation scan: spine scan of flag counts	(insignificant)	(insignificant)
3. Allocation scan: downsweep scan (locally decode and scan flag counts, scatter keys)	$n$ keys + (insignificant)	$n$ keys
<b>Total I/O for all <math>k/d</math> passes:</b>		$(k/d) (3n)$

**Fig. 44.** Procedures for distribution sorting, described as sequences of kernel launches. The I/O models of memory workloads are specified in terms of  $d$ -bit radix digits, radix  $r = 2^d$ , local block size of  $b$  keys, and an  $n$ -element input sequence of  $k$ -bit keys.

each processor with the base digit-offsets for its block of keys. These offsets can then be applied to the local key rankings within the block to distribute the keys.

**Prior GPU approaches.** Current radix sort implementations for the GPU use this approach, treating each CTA as a logical processor operating over a fixed-size block of  $b$  keys [97, 56, 102]. The procedure of Satish et al. is representative of this approach, and is reviewed in Fig. 44.

Because of the decision to keep  $b$  constant, the number of CTA “processors” grows with problem size and the overall memory workload still scales exponentially with  $d$ , although significantly reduced by common block-sizes of 128-1024 keys. This elicits



a global minimum in which there exists an optimal  $d$  to produce a minimal memory overhead for a given block size. For example, when block size  $b = 512$  and key size  $k = 32$ , a radix digit size  $d = 8$  provides minimal memory overhead. As a point of comparison, their implementation imposes an explicit memory workload of 56.6 words per key (where words and keys are 32-bits,  $d = 4$  bits, and  $b = 1024$  keys).

***Our parallelization strategy.*** Briefly, our distribution strategy uses *kernel fusion* to collapse the naive separate decoding and scattering kernels into the prefix scan kernels themselves. Furthermore, we use the *CTA serialization* idiom outlined §3.3.1 to construct a reduce-then-scan approach that requires three kernels, regardless of problem size. We further describe the details of these two idioms and our implementation in §5.3.

Our strategy can operate with a radix digit size  $d \leq 4$  bits on current NVIDIA GPUs before exponentially-growing demands on local storage prevent us from saturating the device. With  $d = 4$  and  $k = 32$ -bit keys-only sorting, our algorithm requires the memory subsystem to explicitly process only 24 words per key, a 2.4x reduction in memory workload.

### 5.2.5 GPU parallelizations of other sorting methods

Radix sorting methods make certain positional and symbolic assumptions regarding the bitwise representations of keys. A comparison-based sorting method is required for a set of ordering rules in which these assumptions do not hold. A variety of comparison-based, top-down partitioning and bottom-up merging strategies have been implemented for the GPU, including quicksort [24, 58], most-significant-digit radix sort [56], sample-sort [39, 77], and merge sort [97]. The number of recursive iterations for these methods

is logarithmic in the size of the input sequence, typically with the first or last 8-10 iterations being replaced by a small local sort within each CTA.

There are several contributing factors that have historically given radix sorting methods an advantage over their comparison-based counterparts. First, comparison-based sorting methods must have work-complexity  $O(n \log_2 n)$  [72], making them less efficient per key as problem size grows. Second, for problem sizes large enough to saturate the device (e.g., several hundred-thousand or more keys), a radix digit size  $d \geq 4$  will result in fewer digit passes than recursive iterations needed by the comparison-based methods performing binary partitioning. Third, the amount of global intermediate state needed by these methods for a given level in the tree of computation is proportional to the width of that level, as opposed to a small constant amount for our radix sort strategy. Finally, parallel radix sorting methods guarantee near-perfect load-balancing amongst GPU cores, an issue of concern for comparison-based methods involving pivot selection

### 5.3 OUR RADIX SORTING STRATEGY

Our radix sorting strategy strives to obtain maximal overall system utilization for a given target architecture. Our goal is to reduce the aggregate memory workload and permit flexible radix sorting granularity  $d$  to maximize processor utilization.

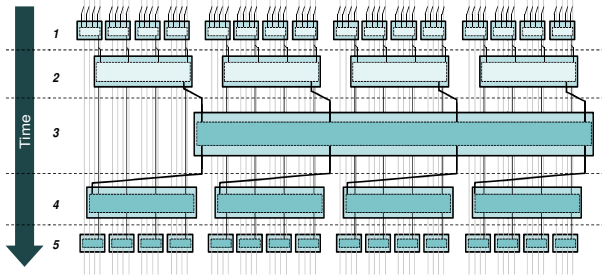
#### 5.3.1 “Multi-scan” prefix sum as allocation runtime

We have generalized our prefix scan implementation for *multi-scan*, i.e., to compute multiple, dependent scans concurrently in a single pass. This allows us to efficiently compute the prefix sums of radix  $r > 1$  flag vectors without imposing any significant additional workload upon the memory subsystem. We rely on the idioms of kernel fusion and CTA serialization to construct multi-scan.

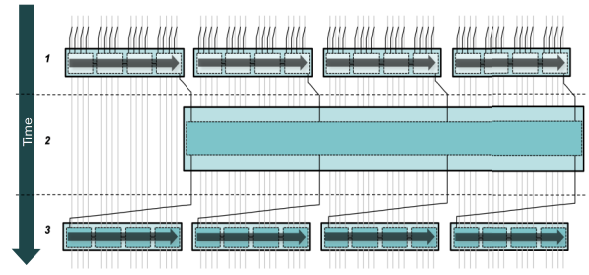
Multi-scan is related to, but different from the segmented scan problem [16]. The segmented-scan problem connotes a single vector comprised of multiple segments, each segment an independent scan problem. For radix sorting purposes, the segments that would correspond to bin-allocations are not actually independent: each has prefix dependences on prior bins. Furthermore, we cannot afford to construct such a vector of  $r$  segments in global memory. Instead, we generate and consume local portions of our flag-vector scan problems in parallel. In addition, these scans are not completely independent: their cumulative reductions are concatenated and scanned as well, resulting in a total ordering of partition offsets.

***Kernel fusion.*** Kernel fusion allows us to collapse the outer decode and scatter kernels from the naive approach. The idea is simple: reduce aggregate memory workload by co-locating sequential steps in the stream pipeline within a single kernel. We implement kernel fusion by inserting our own digit-decoding and key-scattering logic directly into the kernels for prefix scan. The flag values obtained by a thread when decoding a given key can be passed directly via registers to upsweep or downsweep scan logic. Similarly, the ranking results computed by downsweep scan can be locally conveyed to scattering logic for relocating keys and values. Additionally, the keys themselves need not be re-read from global memory for scattering; they were obtained earlier by the downsweep decoding logic within the same kernel closure.

The overall amount of memory traffic is dramatically reduced because we remove the need to move flags through global memory. The elimination of the corresponding load/store instructions also increases the computational efficiency, further allowing our



**Fig. 45.** A typical recursive CTA decomposition for prefix scan. In this example with block size  $b=4$  keys, the scan requires five kernel launches: two upsweep reduction kernels and three downsweep scan kernels.



**Fig. 46.** A fixed, two-level CTA decomposition for prefix scan requires only three kernel launches, regardless of input size. In this example, we reuse four CTAs to process multiple tiles of  $b=4$  keys.

algorithm to exploit those resources. Instead of simply serving as procedural subroutine, prefix scan becomes “runtime” that drives the entire distribution sorting pass.

**CTA serialization.** As illustrated in Fig. 45, a fully-recursive Brent-Kung scan computation has a height of  $O(2\log_b n)$  kernel launches when each CTA is assigned to a fixed-size block of  $b$  elements. This is currently the prevailing strategy for GPU prefix scan [42, 101]. Unfortunately the interior kernels of the “hourglass” do not provide enough work to saturate the processor during their execution. As the least-efficient phase of computation, we want to dispose of this work as quickly as possible.

Our distribution sorting kernels are derived from our *two-level reduce-then-scan* primitives outlined in §4.3. As illustrated in Fig. 46, we compose scan using only three kernels regardless of problem size: an upsweep reduction, a spine scan, and a downsweep scan. Instead of allocating a unique thread for every input key, the first and third kernels dispatch a fixed-size grid of  $C$  CTAs in which threads are re-used to process the input in successive “tiles” of  $b$  keys each. Partial reductions are accumulated in thread-private registers as tiles are processed serially.

There are several important benefits to restricting the amount of parallel work. Our approach requires only a single kernel launch to perform a small, constant  $O(rC)$  amount of interior work. We eliminate  $O(n/b)$  global memory reads and writes at a savings of 2-4 instructions per round-trip (offset calculations, load, store). Finally, any static computation common to each tile of keys can be hoisted, computed once, and reused.

**Kernel stages.** Our three multi-scan kernels listed in Fig. 44 operate as follows:

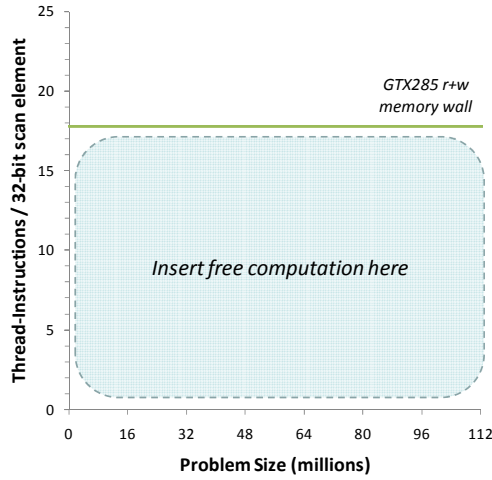
- 1) *Upsweep reduction.* For a multi-scan distribution sorting pass, the upsweep reduction kernel reduces  $n$  inputs into  $rC$  partial reductions. In our implementation, the reduction threads employ a *loop-raking* strategy [18] in which each thread accumulates flags from consecutive tiles, similar to Harris et al. [90]. For each tile, a thread gathers its key, decodes the digit at the current digit-place, and increments the appropriate flag (kept in private registers). After processing their last tile, the threads within each CTA cooperatively reduce these private flags into  $r$  partial reductions, which are then written out to global device memory in preparation for the spine scan.
- 2) *Spine scan.* The single-CTA, spine scan serves to scan the partial reduction contributions from each of the  $C$  bottom-level CTAs. Continuing our theme of multiple, concurrent scans, we have generalized it to scan a concatenation of  $rC$  partial reductions.
- 3) *Downsweep scan/scatter.* In the downsweep scan/scatter kernel, CTAs perform independent scans of their tile sequence, seeded with the partial sums computed by the spine scan. For each tile, threads re-read their keys, re-decode them into local

digit flags, and then scan these flags using the local multi-scan strategy described in the next subsection. The result is a set of  $r$  prefix sums for each key that are used to scatter the keys to their appropriate bins. This scatter logic is also responsible for loading and similarly redistributing any paired satellite values. The  $r$  aggregate counts for each digit are serially carried into the next  $b$ -sized tile.

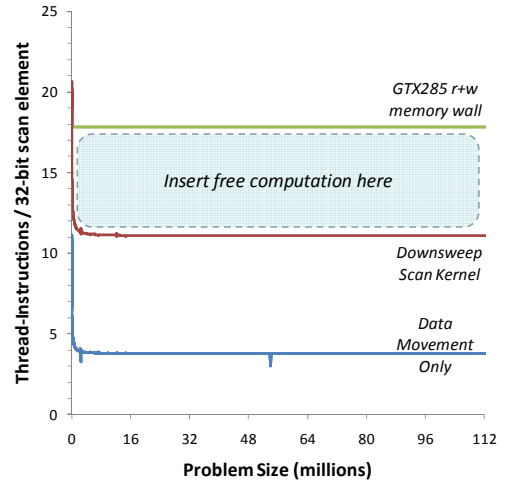
As described in Fig. 44, only a constant number of memory accesses are used for the storage of intermediate results, and there are no longer any coefficients that are exponential in terms of the number of radix digit bits  $d$ . This implies that memory workload will monotonically decrease with increasing  $d$ , positioning our strategy to take advantage of any additional computational power that may allow us to increase  $d$  in the future.

***Flexible radix sort granularity.*** As described in Chapter 4, prefix sum is a memory-bound operation that affords a “bubble” of idle cycles within which we can fuse in sorting logic with little incremental overhead. Furthermore, our multi-scan approach allows us to tune the computational granularity (i.e., number of radix digits  $d$ ) to better fill this bubble of idle cycles.

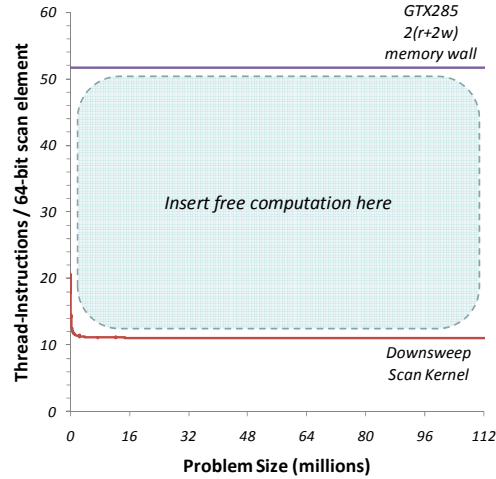
For example, Fig. 47 depicts the “bubble” of free computation below the GTX285 *memory wall*, i.e., the ideal 17.8 thread-cycles that can be executed per 32-bit word copied in and out by the memory subsystem. For the same memory workload, Fig. 48 shows the bubble for a prefix scan downsweep scan kernel after accounting for data movement and scan instructions [85].



**Fig. 47.** GTX 285 memory wall. At an ideal  $354 \times 10^9$  thread-instructions/s and  $159 \times 10^9$  bytes/s, the GTX285 can overlap 17.8 instructions with every two words of memory traffic.



**Fig. 48.** Free cycles within a downsweep scan kernel that moves two words while executing 4 data movement and 8 local scan instructions for each input element.



**Fig. 49.** Free cycles within a downsweep sorting scan/scatter kernel that reads two words, writes two words, and has a write partial-coalescing penalty of two words.

As shown in Fig. 49, the ideal bubble is tripled for a downsweep scan kernel with a memory workload sized to distribute key-value pairs. It must read a pair of two words, write a pair of two words, and pay a partial coalescing penalty of two words. (As we discuss in Section 4, key-scattering produces up to twice as much memory traffic due to partial coalescing. Additionally, the bubble is even larger in practice due to the slightly lower achievable bandwidth rates.)

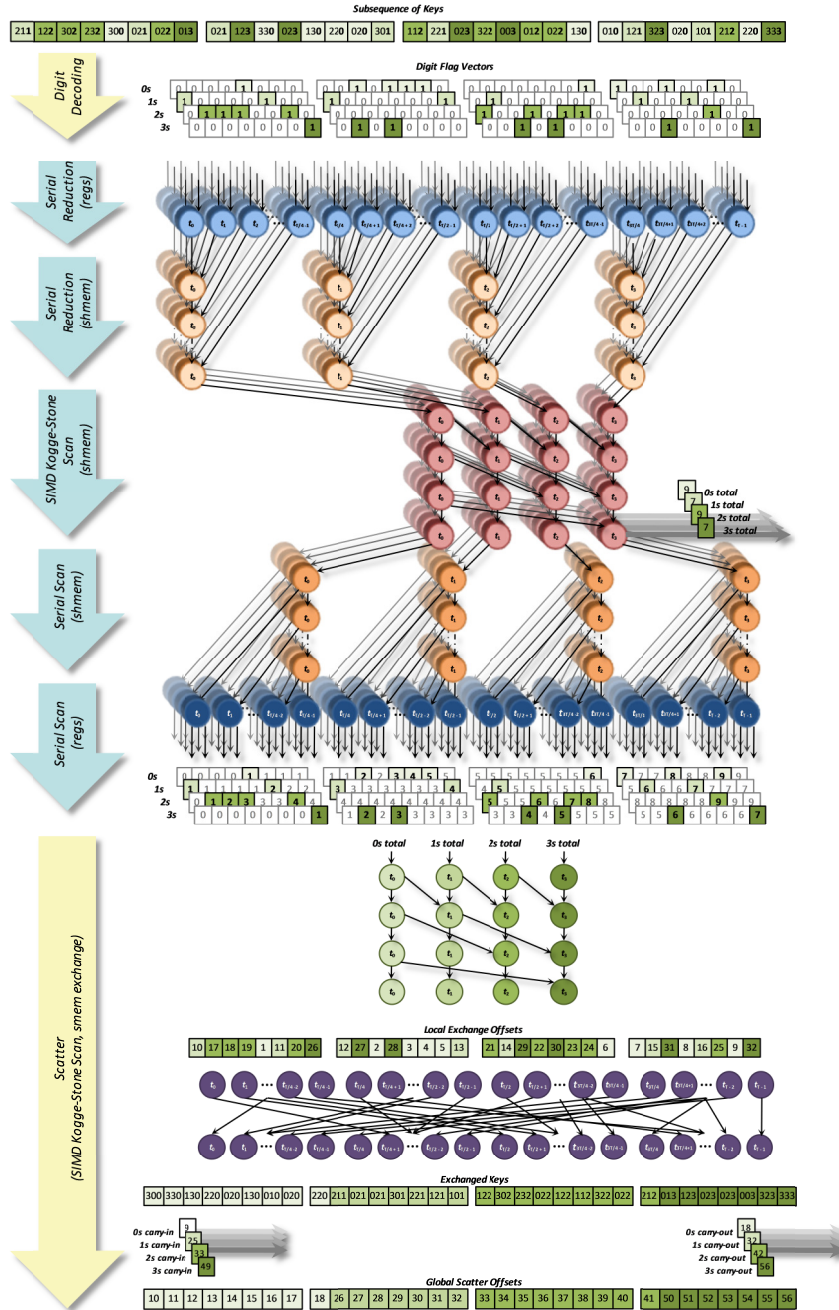
The result is a large window that not only allows us to construct distribution sorting *inside* of prefix scan, but to be more flexible with the granularity of sorting computation as well.

### 5.3.2 Multi-scan downsweep kernel operation

The multi-scan downsweep kernel is the most sophisticated of all three kernels. The downsweep must efficiently perform  $r$  local prefix sums concurrently. Fig. 50 illustrates computation from the point of a single CTA processing a particular tile of input values:

- 1) **Digit decoding.** Threads within the decoding logic collectively read  $b$  keys, decode them according to the current digit-place, and create the private-register equivalent of  $r$  flag vectors of  $b$  elements each.
- 2) **Local multi-scan.** The scan logic is replicated  $r$ -times, ultimately producing  $r$  vectors of  $b$  prefix sums each: one for each of the  $r$  possible digits. It is implemented as a flexible hierarchy of reduce-then-scan strategies composed of three phases of upsweep/downsweep operation:
  - a. *Thread-independent* processing in registers, shown in blue. This phase serves to transition the problem from the tile size  $b$  into a smaller version that will fit into shared memory and back again. This provides flexibility in terms of making maximal use of the available register file and for facilitating different memory transaction sizes (e.g., 1/2/4-element vector load/stores), all without impacting the size of the shared-memory allocation.





**Fig. 50.** Intra-CTA multi-scan operation incorporating fused binning and scatter logic. Computation and time flow downward for an input block size of  $b = 32$  keys, a radix  $r = 4$  digits, and a warp-size  $w = 4$  threads. The scan stages are labeled in light blue, the fused stages in yellow. Circles indicate the assignment of a given thread  $t_i$  to a binary addition task. Flag vector encoding is not shown. The blue thread-independent processing phase is shown to perform vector-2 loads/stores.

b. *Inter-warp cooperation*, shown in orange. In this phase, the threads within a single warp independently reduce/scan though the partial reductions placed in shared memory by the other warps. This serial raking process

transitions the problem size into one that can be cooperatively processed by a single warp and back again, and is similar to the scan techniques described by Dotsenko et al. [42]. This phase provides flexibility in terms of facilitating shared memory allocations of different sizes, supporting alternative SIMD warp sizes, and accommodating arbitrary numbers of warps per CTA. For example, we double the GT200 tile size for the newer GF100 architecture because of the increased amount of shared memory per processor core.

- c. *Intra-warp cooperation*, shown in red. For a given warp-size of  $w$  threads, the intra-warp phase implements  $\log_2 w$  steps of a Kogge-Stone scan [74] in a synchronization-free SIMD fashion. The  $r$  running digit totals from the previous tile are carried into this SIMD “warpscan”, incorporated into the prefix sums of the current tile’s elements, and new running totals are carried out for the next tile, all in local shared memory.

- 3) **Key scattering.** The scatter operation is provided with the tile of keys, their local ranks/prefix sums, the tile’s digit totals, and the incoming running digit totals. Although each scatter thread could use this information to distribute the same keys that it obtained during decoding, doing so would result in poor write coherence. Instead we implement a key exchange. We use the local ranks to scatter keys into a pool of local shared memory, repurposing the raking storage. Then consecutive threads can acquire consecutive keys and scatter them to global device memory with a minimal number of memory transactions. We compute a SIMD prefix sum

of the local digit totals in order to determine the locations of each of the  $r$  segments of newly-coherent keys within this pool.

Although our two-phase scatter procedure is fairly expensive in terms of dynamic instruction overhead and arbitrary bank conflicts, it is much more efficient than the sorting phase implemented by Satish et al. [97]. Their sorting phase performs  $d$  iterations of binary-split, exchanging keys (and values)  $d$  times within shared memory, whereas our approach only exchanges them once.

## 5.4 OPTIMIZATIONS

Our implementation incorporates three important optimizations for improving the efficiency and utility of the radix sorting method: *composite scan*, *early-exit*, and *flexible tuning*. Composite scans exploit bitwise parallelism for much more efficient computation. Early-exit often allows our implementation to skip unnecessary distribution passes for sorting problems having less-than-uniform key distributions. Tuning flexibility facilitates the discovery of program specializations that fit well with the specific target architecture and sorting problem at hand.

### 5.4.1 *Composite scan*

In order to increase the computational efficiency of our implementation, we employ a method for encoding multiple binary-valued flag vectors into a single, composite representation. By using the otherwise unused high-order bits of the flag words and the bitwise parallelism of addition, our *composite scan* technique allows us to compute several logical scan tasks while only incurring the cost of a single parallel scan.

For example, by breaking a tile of keys into *subtiles* of 256-element multi-scans, the scan logic can encode up to four digit flags within a single 32-bit word, with one byte

used for each logical scan. The bit-wise parallelism of 32-bit addition allows us to effectively process four radix digits with a single composite scan. To process the sixteen logical arrays of partial flag sums when  $d = 4$  bits, we therefore only need local storage for 4 scan vectors.

#### 5.4.2 *Early exit*

In many sorting scenarios, the input keys reflect a *banded* distribution. For example, the upper bits are often all zero in many integer sorting problems. Similarly the sign and exponent bits may be homogenous for floating point problems. If this is known *a priori*, the sorting passes for these corresponding digit-places can be explicitly skipped. Unfortunately this knowledge may not be available for a given sorting problem or there may be abstraction layers that prevent application-level code from being aware that a radix-based sorting method is being used.

To provide the benefits of fewer passes for banded keys in a completely transparent fashion, we implement a novel, *early-exit* decision check at the beginning of the downsweep kernel in each distribution sorting pass. By inspecting the partition offsets output by the spine scan kernel, the downsweep CTAs can determine if all keys have the same bits at the current digit place. If there are no partition offsets within the range  $[1, n-1]$ , the downsweep kernel is able to terminate early, leaving the keys in place.

Some passes cannot be short-circuited, however. When sorting signed or floating point keys, the first and last passes must be executed in full, if only to perform the “bit-twiddling” necessary for these types to be radix-sorted. For example, the most-significant bits for signed integer types need to be flipped before the first pass and after the last.

### 5.4.3 *Flexible tuning*

Radix sorting exhibits a fundamental tradeoff with regard to tile size  $b$  versus processor core occupancy. Both the tile size and the number of radix digits  $r$  will determine the local storage requirements (e.g., register and shared memory allocations) for a given CTA. Increasing the tile size has two effects: it increases the write coherence for scattering keys; and it comparatively lowers the relative overheads from the work-inefficient portions of our local scan. However, too large a tile size will prevent the processor cores from being occupied by enough CTAs to cover shared-memory latencies when only raking warps are active.

For different data types, this performance cliff occurs at different tile sizes for different architectures, and is dependent upon register and shared memory availability and pipeline depths. For example, the GT200 architecture allows us to unroll two subtiles per tile due to the amount of shared memory provisioned per core. Furthermore, performance tuning reveals better throughput using 128-thread CTAs where each thread processes two keys. In contrast, we can unroll four subtiles per tile on newer GF100 processors having larger cores, and better performance is achieved using 64-thread CTAs where each thread processes four keys. For both architectures, we must halve the tile size when sorting 64-bit keys because the storage required for scattering keys to local shared memory is doubled. Similarly, we can double the tile size when sorting 16-bit shorts and 8-bit chars.

As described in Chapter 3, we establish rules for generating such tuning policies for different combinations of problem type (i.e., key/value types) and processor architecture. These tuning policies are expressed as C++ types when are then used to parameterize our templated sorting implementation. We rely on the compiler for

template expansion, constant propagation, and loop unrolling in order to produce an executable assembly that is well-tuned for the specifically targeted hardware and problem type.

## 5.5 ANALYTICAL MODEL

The computational workload for distribution sorting passes can be decomposed into two portions: work that scales directly with the size of the input (i.e., moving and decoding keys); and work that scales proportionally with the size of the input multiplied by the number of radix digits (i.e., the  $r$  concurrent scans). Because the computational overhead of the downsweep scan kernel dominates that of the upsweep reduction kernel, we base our granularity decisions upon modeling the former.

We model downsweep kernel work in terms of the following tasks: (1) data-movement to/from global memory; (2) digit inspection and encoding of flag vectors in shared memory; (3) shared-memory scanning; (4) decoding local rank from shared memory; (5) and locally exchanging keys and values prior to scatter. For a given key-value pair, each task incurs a fixed cost  $\alpha$  in terms of thread-instructions. The flag-encoding and scanning operations will also incur a per-pair cost of  $\beta$  instructions per composite scan. We model cumulative thread-instruction count using:

$$\begin{aligned} instrs_{\text{scankernel}}(n, r) = n & \left( \alpha_{\text{mem}} + \alpha_{\text{encflags}} + \alpha_{\text{scan}} + \alpha_{\text{decflags}} + \alpha_{\text{exchange}} + \right. \\ & \left. \frac{r}{4} (\beta_{\text{encflags}} + \beta_{\text{scan}}) \right) \end{aligned}$$

We can use instrumentation to determine these coefficients for a given architecture / processor family. For example,  $instrs_{\text{scankernel}}(n, r) = n (51.4 + 1.0r)$  for

sorting pairs of 32-bit keys and values on the NVIDIA GT200 architecture. The instruction costs per pair are (not including warp-serializations):

$$\begin{aligned}\alpha_{\text{mem}} &= 6.3 & \alpha_{\text{scan}} &= 10.7 & \alpha_{\text{exchange}} &= 14.7 \\ \alpha_{\text{encflags}} &= 5.5 & \alpha_{\text{decflags}} &= 13.9\end{aligned}$$

The instruction costs per pair per composite scan are:

$$\beta_{\text{encflags}} = 2.6 \qquad \beta_{\text{scan}} = 1.4$$

Minimizing this parameterized function for GT200, the radix sorting granularity with the lowest computational overhead is  $d = 4$  bits ( $r = 16$  radix digits). Parameterizations for G80, G92, and GF100 architectures yield the same granularity.

Although  $d=4$  is minimal for the GTX285, the model predicts that the downsweep kernel will still be compute-bound: the overhead per pair is 67.4 instructions, which exceeds the memory wall of 52 instructions illustrated in Fig. 49. For this specific processor, the performance will strictly be a function of compute workload. However, as increases in computational throughput continue to outpace increases in memory bandwidth, it is likely that the bubble of memory-boundedness will eventually provide us with room to increase  $d$  past the minimum computational workload without penalty in order to reduce overall passes.

## 5.6 EVALUATION

This section presents the performance of our allocation-oriented radix sorting strategy. We present our own performance measurements of the implementation by Satish et al. (via CUDPP v1.1 [35]), which is representative of the current state of the art in GPU

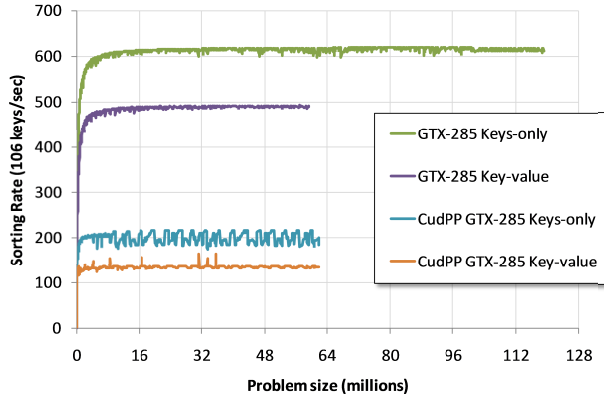
Table 8. Saturated 32-bit sorting rates for input sequences larger than 16M elements				
Device	Keys-only Sorting Rate ( $10^6$ 32-bit keys/s)		Key-Value Sorting Rate ( $10^6$ 32-bit pairs/s)	
Name	CUDPP Radix	Our Radix (speedup)	CUDPP Radix	Our Radix (speedup)
NVIDIA GTX 580		1182		882
NVIDIA GTX 480	349	1005 (2.9x)	257	775 (3.0x)
NVIDIA Tesla C2050	270	742 (2.7x)	200	581 (2.9x)
NVIDIA GTX 285	199	615 (2.8x)	134	490 (3.7x)
NVIDIA GTX 280	184	534 (2.6x)	117	449 (3.8x)
NVIDIA Tesla C1060	176	524 (2.7x)	111	333 (3.0x)
NVIDIA 9800 GTX+	111	265 (2.0x)	82	189 (2.0x)
NVIDIA 8800 GT	83	171 (2.1x)	63	129 (2.1x)
NVIDIA Quadro FX5600	66	147 (2.2x)	55	110 (2.0x)
Intel 32-core Knight's Ferry MIC [9]		560		
Intel quad-core i7 [7]		240		
Intel Q9550 quad-core [8]		138		

sorting. We also contrast our sorting performance with contemporary x86-based many-core sorting results [30, 98, 99].

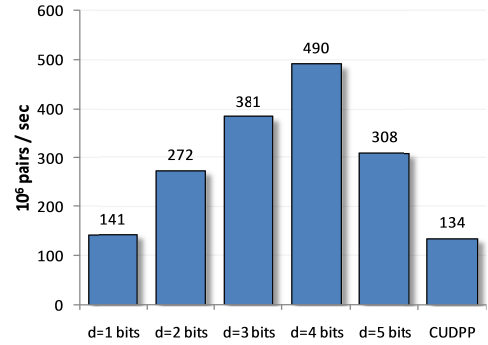
### 5.6.1 Configuration and methodology

Our primary test environment consisted of a Linux platform with an NVIDIA GTX285 GPU running the CUDA 3.2 compiler and driver framework. Our analyses are derived from measurements taken over a suite of ~3,000 randomly-sized problem sequences (up to 128M elements). Each sorting problem is initialized with 32-bit keys and values sampled from a uniformly random distribution. We primarily evaluate key-value pair sorting, but also report results for keys-only sorting. Our measurements for elapsed time, dynamic instruction count, warp serializations, memory transactions, etc., are taken directly from GPU hardware performance counters. Our analyses are reflective of *in situ* sorting problems: they preclude the driver overhead and the overheads of staging data





**Fig. 51.** GTX285 keys-only and key-value pair radix sorting rates.



**Fig. 52.** GTX285 saturated sorting rates for various radix bits  $d$  (32-bit key-value pairs)

to/from the accelerator, allowing us to directly contrast the individual and cumulative performance of the stream kernels involved.

### 5.6.2 Overall sorting rates

Fig. 51 plots the measured radix sorting rates exhibited by our  $d = 4$  bits implementation and the CUDPP primitive. We observe that the radix sorting performances plateau into steady-state as the GPU’s resources become saturated. In addition to exhibiting 3.7x and 2.8x speedups over the CUDPP implementation on the same device, our key-value and key-only implementations provide smoother, more consistent performance across the sampled problem sizes. Table 8 presents speedups over CUDPP of 2.0-3.8x for all generations of NVIDIA GPU processors.

Recent publications have set a precedent of comparing the sorting performances of the “best available” implementations for GPU and CPU architecture [30, 98, 99]. The saturated sorting rates on these devices for input sequences of 16M+ keys are denoted in Table 8. Using our method, all of the NVIDIA GPUs outperform the Intel Xeon Core2 Quad Q9550 CPU and, perhaps more strikingly, our sorting rates for previous-generation

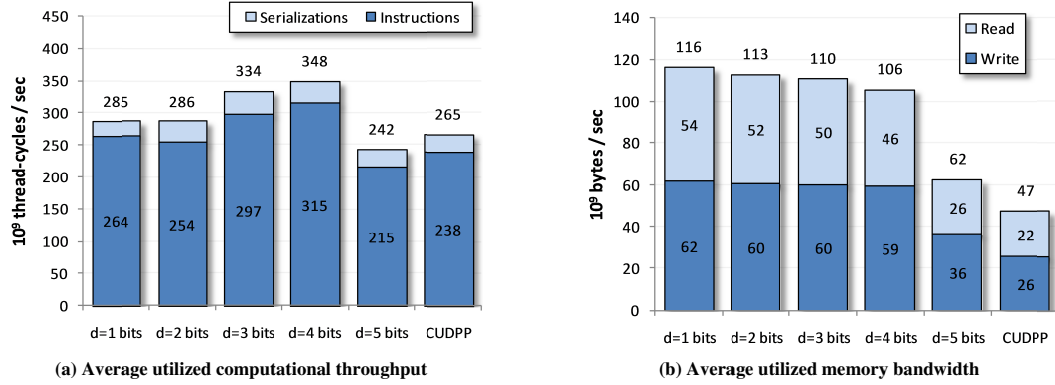


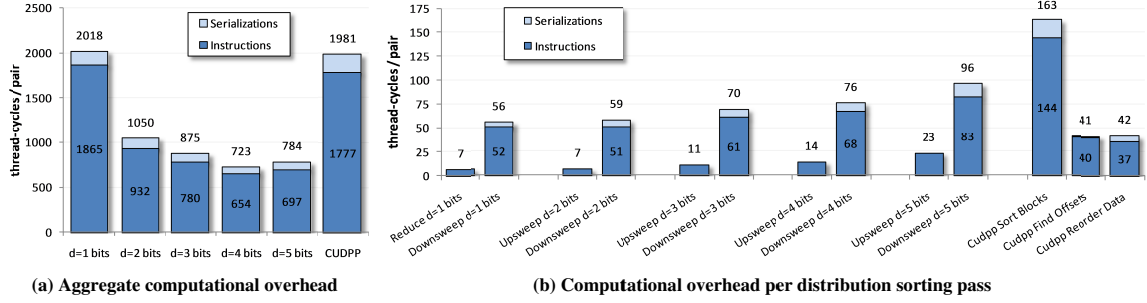
Fig. 53. Resource utilization for 32-bit key-value sorting on GTX285.

GT200 GPUs exhibit performance on par with or better than the unreleased Intel 32-core Knights Ferry.

Fig. 52 illustrates the effects of increasing the number of radix digit bits  $d$  (and thus decreasing the number of distribution sorting passes) for  $1 \leq d \leq 5$  bits. We observe that throughput improves as  $d$  increases for  $d < 5$ . When  $d \geq 5$ , two issues conspire to impair performance, both related to the exponential growth of the  $r = 2^d$  radix digits that need scanning. The first is that the cumulative computational workload is no longer decreasing with reduced passes. As predicted by the model, the two bottom-level kernels are compute-bound under this load. Continuing to increase the overall computational workload will only result in progressively larger slowdowns. The second issue is that the increased local storage requirements (i.e., registers and shared memory) prevent processor saturation: the occupancy per GPU core is reduced from 640 to 256 threads.

### 5.6.3 Resource utilization

Fig. 53 shows the computational and memory throughputs realized by our variants and the CUDPP implementation. The GTX285 provides realistic maximums of  $354 \times 10^9$  thread-cycles/s and  $136\text{--}149 \times 10^9$  bytes/s, respectively. Our 3-bit and 4-bit variants



**Fig. 54.** Computational workload for 32-bit key-value sorting on GTX285.

achieve more than 94% of this computational throughput. The 1-bit, 2-bit, and CUDPP implementations have a mixture of compute-bound and memory-bound kernels, resulting in lower overall averages for both workloads. The 5-bit variant illustrates the effects of under-occupied processor cores: its kernels are compute-bound, yet it only utilizes 68% of the available computational throughput.

#### 5.6.4 Computational workloads

Our five variants in Fig. 54a corroborate our model of computational overhead versus digit size: workload decreases with the number of passes until the cost of scanning radix digits becomes dominant at  $d = 5$ . This overhead is inclusive of the number of thread-cycles consumed by scalar instructions as well as the number of stall cycles incurred by the warp-serializations that primarily result from the random exchanges of keys and values in shared memory. The 723 thread-cycles executed per input pair by our 4-bit implementation may seem substantial, yet efficiency is 2.7x that of the CUDPP implementation.

Fig. 54b presents the computational overheads of the individual kernel invocations that comprise a distribution sorting pass. For  $d > 2$ , we observe that the workload deltas between scan kernels double as  $d$  is incremented, scaling with  $r$  as

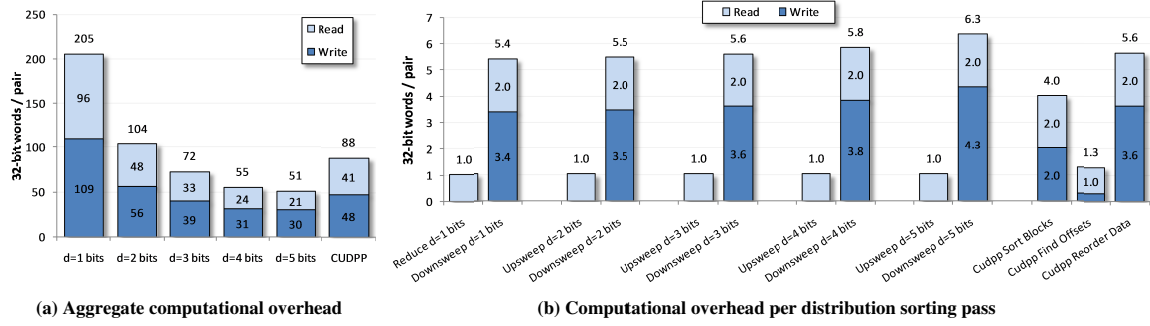


Fig. 55. Memory workload for 32-bit key-value sorting on GTX285.

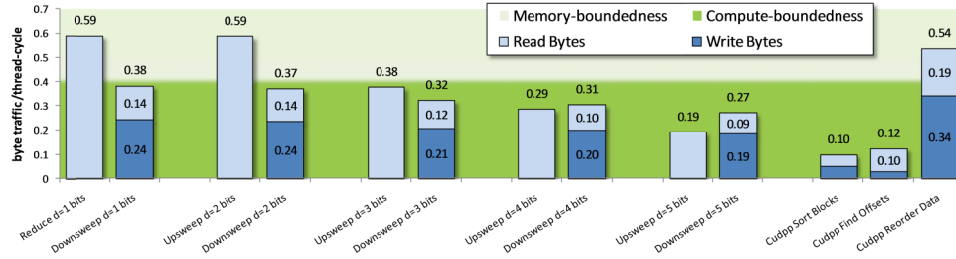
expected and validating our model of instruction overhead. Our 1-bit and 2-bit variants do not follow this parameterized model: the optimizing compiler produces different code for them because flag vector encoding requires only one composite scan.

### 5.6.5 Memory workloads

The overall memory workloads for these implementations are shown in Fig. 55a. We confirm that our memory overhead monotonically decreases with increasing  $d$ . As predicted by our memory traffic model (Fig. 44), the memory workload of the 4-bit CUDPP implementation is 1.6x that of our 4-bit variant.

Fig. 55b illustrates the memory overheads for a single distribution sorting pass, broken down by kernel and granularity. Although our scatter instructions logically write two 32-bit words per pair, we observe that the hardware issues separate transactions when threads within the same half-warp scatter keys to different memory segments. Extra memory bandwidth is used when this occurs: the memory subsystem rounds up to a full-sized transaction (e.g., 32B / 64B / 128B), even though only a portion of it may contain actual data.

On the GT200 architecture, this extra traffic scales with the number of partitions  $r$ , starting at 70% overhead when  $r = 2$ . For  $r = 16$ , this overhead exceeds our explicit



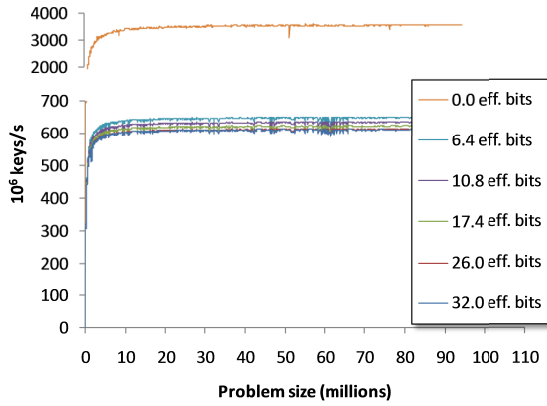
**Fig. 56.** Memory vs. compute workload ratios for individual sorting kernels. The two-tone backdrop demarcates the memory wall for the GTX285 (i.e., 0.45 bytes/cycle), illustrating the degree to which kernels are memory-bound or compute-bound.

memory traffic model (Fig. 44) by 28%, whereas the CUDPP implementation incurs only a 22% cumulative increase.

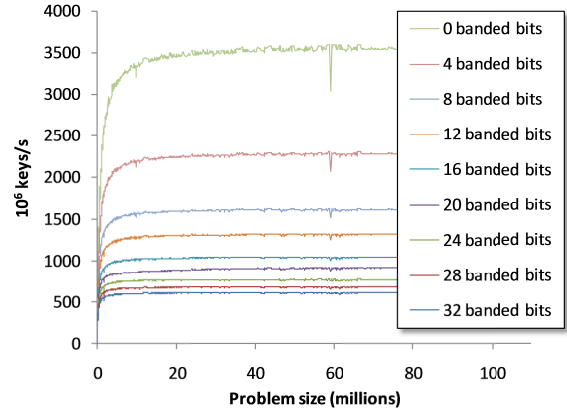
GPU processors are less efficient when consecutive kernels oscillate between being memory-bound and being compute-bound. Fig. 56 illustrates the corresponding workload ratios for each of the stream kernels relative to the GTX285 memory wall. We see that for  $d > 2$ , our distribution sorting streams only contain compute-bound kernels. The CUDPP implementation contains a mixture of memory-bound and extremely compute-bound kernels, resulting in an overall underutilization of hardware

### 5.6.6 Key diversity

The distribution of key-bits can have an impact on the overall sorting performance. The traditional expectation for radix sorting is that a perfectly uniform distribution of key bits will yield worst-case performance because it produces the highest number of fragmented scatter transactions. As the bits become less uniform and certain digits become more likely than others, the expected number of memory segments touched per SIMD-write decreases. The increased efficiency typically results in a corresponding performance improvement. We evaluate two classes of key distributions having less-than-uniform variance: decreased entropy and key banding.



**Fig. 57.** Sorting performance with varying key entropy (32-bit keys-only, GTX285).



**Fig. 58.** Sorting performance for key distributions with banded ranges (32-bit keys-only, GTX285)..

**Decreased entropy.** The reduction of bitwise entropy within randomly selected keys is a common technique for reducing key variance [112]. The idea is to generate keys that trend towards having more 0s than 1s bits by repeatedly bitwise-ANDing together uniformly-random bits. For each bitwise-AND, the number of 1s bits per key decreases. An infinite number of bitwise-AND iterations results in a completely uniform key-distribution of all 0s. Fig. 57 confirms that our implementation’s performance improves as keys become less random. Our evaluation of entropy reduction reveals a range of improvement of up to 11% for keys-only sorting, and the early-exit optimization is activated at zero-effective bits.

Interestingly enough, our performance improvement is not a result of lower memory workload. These kernels are compute-bound; the speedup is instead gained from the elimination of warp-serialization hazards that stem from bank conflicts incurred during key exchange.

**Key banding.** Banding (or “clumping”) is perhaps a more likely form of reduced key diversity. In these scenarios, keys are differentiated only by narrow bands of key

bits. Fig. 46 illustrates the effectiveness of our early-exit optimization. Without specifying any explicit information to the sorting implementation, we evaluate sorting performance on key distributions whose keys differ only by variously-sized “banded” bit fields.

For GTX285 keys-only sorting, we observe that the cost of a distribution pass is reduced by 83% when short-circuited. Depending on the number of passes that exit early, sorting rates can be improved by up to 5.8x for 32-bit keys. As an example, consider an array of 32-bit integers containing the same nominal information as a similarly-sized array of 8-bit characters. Our implementation will discover this banding information transparently and sort these 32-bit keys at a rate of 1.6 billion keys/s on the GTX285, a speedup of 2.6x.

## 5.7 CHAPTER SUMMARY

We have presented efficient radix-sorting strategies for ordering large sequences of fixed-length keys (and values) for GPU architecture. Our performance results demonstrate multiple factors of speedup over existing GPU implementations, and we believe our implementations to be the fastest published for any fully-programmable microarchitecture.

These results motivate a style of flexible algorithm design for GPU stream architectures that can maximally exploit the memory and computational resources, yet easily be adapted for a diversity of underlying hardware configurations. Our allocation-oriented framework provides us substantial flexibility with respect to radix sorting granularity. Our approach is well positioned to take advantage of increasing computational throughputs that outpace improvements in memory bandwidth.

## Chapter 6

### *Sparse Graph Traversal*

#### 6.1 INTRODUCTION

Algorithms for analyzing sparse relationships represented as graphs provide crucial tools in many computational fields ranging from genomics to electronic design automation to social network analysis. In this chapter, we explore the parallelization of one fundamental graph algorithm on GPUs: breadth-first search (BFS). BFS is a common building block for more sophisticated graph algorithms, yet is simple enough that we can analyze its behavior in depth. It is also used as a core computational kernel in a number of benchmark suites, including Parboil [93], Rodinia [27], and the emerging Graph500 supercomputer benchmark [111].

BFS is representative of a class of algorithms for which it has been hard to obtain significantly better performance from parallelization. When parallelized, the cooperative and dynamic nature of the problem introduces concerns of contention, load imbalance, and underutilization on multithreaded architectures [2, 78, 118]. Both the wide SMT and SIMD parallelism of GPUs can be particularly performance-sensitive to these issues.



Prior work has advocated two key architectural features for facilitating parallel graph algorithms: deep multithreading and fine-grained synchronization [2, 7, 8]. As a mechanism for overlapping computation with memory latency, multithreading is especially valuable for sparse graph workloads. Optimizing memory usage is non-trivial because memory access patterns are determined by the arbitrary structure of the input graphs. Because high memory latencies are often unavoidable, it is often more advantageous to hide such latency with multithreading than attempting to minimize it using the cache hierarchy.

The second feature is fine-grained synchronization, specifically atomic read-modify-write operations. Such algorithms have incorporated atomic mechanisms for coordinating the dynamic placement of data into shared data structures and for arbitrating contended status updates. On paper, modern GPU architectures provide both features. However, the performance overhead from atomic serialization is often unacceptably high. As we illustrated in Chapter 1.3.1, the incorporation of fine-grained atomic operations can reduce overall throughput by two or three orders of magnitude.

Continuing our dissertation theme, we argue that that prefix sum is a more suitable mechanism for dynamic data placement within shared structures. Such structures are necessary for work-efficient graph traversal. Furthermore, efficient prefix sum enables optimizations that reorganize sparse and uneven workloads into dense and uniform ones in all phases of graph traversal.

Our work as described in this chapter makes contributions in the following areas:

***Parallelization strategy.*** We present a GPU BFS parallelization that performs an asymptotically optimal linear amount of work. It is the first to incorporate fine-grained

parallel adjacency list expansion. We also introduce local duplicate detection techniques for avoiding race conditions that create redundant work. We demonstrate that our approach delivers high performance on a broad spectrum of structurally diverse graphs. To our knowledge, we also describe the first design for multi-GPU graph traversal.

***Empirical performance characterization.*** We present detailed analyses that isolate and analyze the expansion and contraction aspects of BFS throughout the traversal process. We reveal that serial and warp-centric expansion techniques described by prior work significantly underutilize the GPU for important graph genres. We also show that the fusion of neighbor expansion and inspection within the same kernel often yields worse performance than performing them separately.

***High performance.*** We demonstrate that our methods deliver excellent performance on a diverse body of real-world graphs. Our implementation achieves traversal rates in excess of 3.3 billion and 8.3 billion traversed edges per second (TE/s) for single and quad-GPU configurations, respectively. To put these numbers in context, recent state-of-the-art parallel implementations achieve 0.7 billion and 1.3 billion TE/s for similar datasets on single and quad-socket multicore processors [2].

## 6.2 BACKGROUND

We consider graphs of the form  $G = (V, E)$  with a set  $V$  of  $n$  vertices and a set  $E$  of  $m$  directed edges. Given a source vertex  $v_s$ , the goal of BFS is to traverse the vertices of  $G$  in breadth-first order starting at  $v_s$ . Each newly-discovered vertex  $v_i$  will be labeled by (a) its distance  $d_i$  from  $v_s$  and/or (b) the predecessor vertex  $p_i$  immediately preceding it on the shortest path to  $v_s$ .

$$\mathbf{A} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} \quad \begin{array}{l} C = [0, 1, 1, 2, 0, 2, 3, 1, 3] \\ R = [0, 2, 4, 7, 9] \end{array}$$

**Fig. 59.** Example CSR representation: column-indices array  $C$  and row-offsets array  $R$  comprise the adjacency matrix  $A$ .

---

**Listing 5.** The simple sequential breadth-first search algorithm for marking vertex distances from the source  $s$ . Alternatively, a shortest-paths search tree can be constructed by marking  $i$  as  $j$ 's predecessor in line 11.

---

**Input:** Vertex set  $V$ , row-offsets array  $R$ , column-indices array  $C$ , source vertex  $s$   
**Output:** Array  $dist[0..n-1]$  with  $dist[v]$  holding the distance from  $s$  to  $v$   
**Functions:** *Enqueue*( $val$ ) inserts  $val$  at the end of the queue instance. *Dequeue*() returns the front element of the queue instance.

---

```

1  Q := {}
2  for i in V:
3      dist[i] := ∞
4  dist[s] := 0
5  Q.Enqueue(s)
6  while (Q != {}) :
7      i = Q.Dequeue()
8      for offset in R[i] .. R[i+1]-1 :
9          j := C[offset]
10         if (dist[j] == ∞)
11             dist[j] := dist[i] + 1;
12         Q.Enqueue(j)

```

---

Fundamental uses of BFS include identifying all of the connected components within a graph; finding the diameter of tree; and testing a graph for bipartiteness [33]. More sophisticated problems incorporating BFS include identifying the reachable set of heap items during garbage collection [29]; belief propagation in statistical inference [50], finding community structure in networks [87], and computing the maximum-flow/minimum-cut for a given graph [66].

### 6.2.1 Sparse graph representation

For simplicity, we identify graph vertices using integer indices, i.e.,  $v_0 \dots v_{n-1}$ . The pair  $(v_i, v_j)$  indicates a directed edge in the graph from  $v_i \rightarrow v_j$ , and the adjacency list  $A_i = \{v_j \mid (v_i, v_j) \in E\}$  is the set of neighboring vertices adjacent from vertex  $v_i$ . We treat undirected graphs as symmetric directed graphs containing both  $(v_i, v_j)$  and  $(v_j, v_i)$  for each undirected edge. In this paper, all graph sizes and traversal rates are measured in terms of directed edge counts.

We represent the graph using an adjacency matrix  $\mathbf{A}$ , whose rows are the adjacency lists  $A_i$ . The number of edges within sparse graphs is typically only a constant factor larger than  $n$ . We use the well-known compressed sparse row (CSR) sparse matrix format to store the graph in memory consisting of two arrays. As illustrated in Fig. 59, the column-indices array  $C$  is formed from the set of the adjacency lists concatenated into a single array of  $m$  integers. The row-offsets  $R$  array contains  $n + 1$  integers, and entry  $R[i]$  is the index in  $C$  of the adjacency list  $A_i$ .

We store graphs in the order they are defined. We do not perform any offline preprocessing in order to improve locality of reference, improve load balance, or eliminate sparse memory references. Such strategies might include sorting neighbors within their adjacency lists; sorting vertices into a space-filling curve and remapping their corresponding vertex identifiers; splitting up vertices having large adjacency lists; encoding adjacency row offset and length information into vertex identifiers; removing duplicate edges, singleton vertices, and self-loops; etc.

### 6.2.2 *Sequential BFS*

Listing 5 describes the standard sequential BFS method for circulating the vertices of the input graph through a FIFO queue that is initialized with  $v_s$  [33]. As vertices are dequeued, their neighbors are examined. Unvisited neighbors are labeled with their distance and/or predecessor and are enqueued for later processing. This algorithm performs linear  $O(m+n)$  work since each vertex is labeled exactly once and each edge is traversed exactly once.

---

**Listing 6.** A simple quadratic-work, vertex-oriented BFS parallelization

---

**Input:** Vertex set  $V$ , row-offsets array  $R$ , column-indices array  $C$ , source vertex  $s$   
**Output:** Array  $dist[0..n-1]$  with  $dist[v]$  holding the distance from  $s$  to  $v$

---

```

1  parallel for (i in V) :
2      dist[i] := ∞
3  dist[s] := 0
4  iteration := 0
5  do :
6      done := true
7      parallel for (i in V) :
8          if (dist[i] == iteration)
9              done := false
10             for (offset in R[i] .. R[i+1]-1) :
11                 j := C[offset]
12                 dist[j] = iteration + 1
13             iteration++
14  while (!done)

```

---



---

**Listing 7.** A linear-work BFS parallelization constructed using a global vertex-frontier queue.

---

**Input:** Vertex set  $V$ , row-offsets array  $R$ , column-indices array  $C$ , source vertex  $s$ , queues  
**Output:** Array  $dist[0..n-1]$  with  $dist[v]$  holding the distance from  $s$  to  $v$   
**Functions:** *LockedEnqueue*( $val$ ) safely inserts  $val$  at the end of the queue instance

---

```

1  parallel for (i in V) :
2      dist[i] := ∞
3  dist[s] := 0
4  iteration := 0
5  inQ := {}
6  inQ.LockedEnqueue(s)
7  while (inQ != {}) :
8      outQ := {}
9      parallel for (i in inQ) :
10         for (offset in R[i] .. R[i+1]-1) :
11             j := C[offset]
12             if (dist[j] == ∞)
13                 dist[j] = iteration + 1
14                 outQ.LockedEnqueue(j)
15             iteration++
16  inQ := outQ

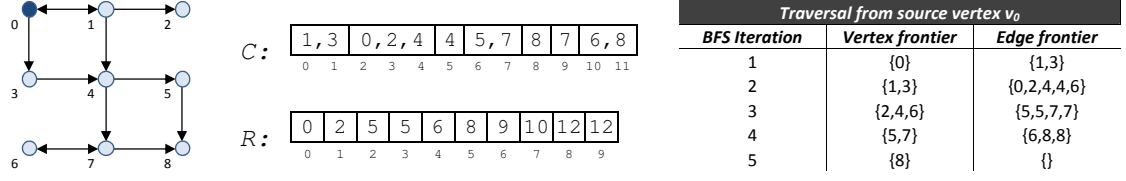
```

---

### 6.2.3 Parallel BFS

The FIFO ordering of the sequential algorithm forces it to label vertices in increasing order of depth. Each depth level is fully explored before the next. Most parallel BFS algorithms are *level-synchronous*, i.e., each level may be processed in parallel as long as the sequential ordering of levels is preserved. An implicit race condition can exist where multiple tasks may concurrently discover a vertex  $v_j$ . This is generally considered benign since all such contending tasks would apply the same  $d_j$  and give a valid value of  $p_j$ .

Structurally different methods may be more suitable for graphs with very large diameters, e.g., algorithms based on the method of Ullman and Yannakakis [114]. Such alternatives are beyond the scope of this paper.



**Fig. 60.** Example sparse graph, corresponding CSR representation, and frontier evolution for a BFS beginning at source vertex  $v_0$ .

As illustrated in Fig. 60, each iteration of a level-synchronous method identifies both an edge and vertex *frontier*. The edge-frontier is the set of all edges to be traversed during that iteration or, equivalently, the set of all  $A_i$  where  $v_i$  was marked in the previous iteration. The vertex-frontier is the unique subset of such neighbors that are unmarked and which will be labeled and expanded for the next iteration. Each iteration logically expands vertices into an edge-frontier and then contracts them to a vertex-frontier.

**Quadratic-work parallelizations.** The simplest parallel BFS algorithms inspect every edge or, at a minimum, every vertex during every iteration. These methods perform a quadratic amount of work. A vertex  $v_j$  is marked when a task discovers an edge  $v_i \rightarrow v_j$  where  $v_i$  has been marked and  $v_j$  has not. As Listing 6 illustrates, vertex-oriented variants must subsequently expand and mark the neighbors of  $v_j$ . Because the process may require  $n$  BFS iterations in the worst case, the overall work complexity is  $O(n^2 + m)$ .

Quadratic parallelization strategies have been used by almost all prior GPU implementations. The static assignment of tasks to vertices (or edges) trivially maps to the data-parallel GPU machine model. Each thread's computation is completely independent from that of other threads. Harish *et al.* [55] and Hussein *et al.* [66] describe vertex-oriented versions of this method. Deng *et al.* present an edge-oriented implementation [40].

Hong *et al.* [63] describe a vectorized version of the vertex-oriented method that is similar to the CSR sparse matrix-vector (SpMV) multiplication approach by Bell and Garland [11]. Warps rather than individual threads are assigned to vertices. During neighbor expansion, the SIMD lanes of an entire warp are used to strip-mine<sup>13</sup> the corresponding adjacency list.

These quadratic methods are isomorphic to iterative SpMV in the algebraic semi-ring where the usual  $(+, \times)$  operations are replaced with  $(\min, +)$ , and thus can also be realized using generic implementations of SpMV [48].

***Linear-work parallelizations.*** A work-efficient parallel BFS algorithm should perform  $O(n+m)$  work. To achieve this, each iteration should examine only the edges and vertices in that iteration’s logical edge and vertex-frontiers, respectively.

Frontiers may be maintained *in-core* or *out-of-core*. An in-core frontier is processed online and is never wholly realized. On the other hand, a frontier that is managed out-of-core is fully produced in off-chip DRAM (global memory) for consumption by the next BFS iteration after a global synchronization step.

Implementations typically prefer to manage the vertex-frontier out-of-core. Less global data movement is needed because the average vertex-frontier is smaller by a factor of  $\bar{d}$  (average out-degree). As described in Listing 7, each BFS iteration maps tasks to unexplored vertices in the input vertex-frontier queue. Their neighbors are inspected and the unvisited ones are placed into the output vertex-frontier queue for the next iteration.

---

<sup>13</sup> Strip mining entails the sequential processing of parallel batches, where the batch size is typically the number of hardware SIMD vector lanes.

Research has traditionally focused on two aspects of this scheme: (1) improving hardware utilization via intelligent task scheduling; and (2) designing shared data structures that incur minimal overhead from insertion and removal operations.

The typical approach for improving utilization is to reduce the task granularity to a homogenous size and then evenly distribute these smaller tasks among threads. This is done by expanding and inspecting neighbors in parallel. Logically, the sequential-for loop in line 10 of Listing 7 is replaced with a parallel-for loop. The implementation can either: (a) spawn all edge-inspection tasks before processing any, wholly realizing the edge-frontier out-of-core; or (b) carefully throttle the parallel expansion and processing of adjacency lists, producing and consuming these tasks in-core.

In recent BFS research, Leiserson and Schardl [78] designed an implementation for multi-socket CPU systems that incorporates a novel multi-set data structure for tracking the vertex-frontier. They implement concurrent neighbor inspection, using the Cilk++ runtime to manage the edge-processing tasks in-core.

For the Cray MTA-2, Bader and Madduri [7] describe an implementation using the hardware's full-empty bits for efficient queuing into an out-of-core vertex frontier. They also perform adjacency-list expansion in parallel, relying on the parallelizing compiler and fine-grained thread-scheduling hardware to manage edge-processing tasks in-core.

Luo *et al.* [79] present an implementation for GPUs that relies upon a hierarchical scheme for producing an out-of-core vertex-frontier. To our knowledge, theirs is the only prior attempt at designing a work-efficient BFS algorithm for GPUs. Their GPU kernels logically correspond to lines 10-13 of Listing 7. Threads perform serial adjacency list



---

**Listing 8.** A linear-work, vertex-oriented BFS parallelization for a graph that has been partitioned across multiple processors. The scheme uses a set of distributed edge-frontier queues, one per processor.

---

**Input:** Vertex set  $V$ , row-offsets array  $R$ , column-indices array  $C$ , source vertex  $s$ , queues

**Output:** Array  $dist[0..n-1]$  with  $dist[v]$  holding the distance from  $s$  to  $v$

**Functions:**  $LockedEnqueue(val)$  safely inserts  $val$  at the end of the queue instance

---

```

1  parallel for i in V :
2      distproc[i] := ∞
3      iteration := 0
4      parallel for (proc in 0 .. processors-1) :
5          inQproc := {}
6          outQproc := {}
7          if (proc == Owner(s))
8              inQproc.LockedEnqueue(s)
9              distproc[s] := 0
10     do :
11         done := true;
12         parallel for (proc in 0 .. processors-1) :
13             parallel for (i in inQproc) :
14                 if (distproc[i] == ∞)
15                     done := false
16                     distproc[i] := iteration
17                     for (offset in R[i] .. R[i+1]-1) :
18                         j := C[offset]
19                         dest := owner(j)
20                         outQdest.LockedEnqueue(j)
21         parallel for (proc in 0 .. processors-1) :
22             inQproc := outQproc
23             iteration++
24     while (!done)

```

---

expansion and use an upward propagation tree of child-queue structures in an effort to mitigate the contention overhead on any given atomically-incremented queue pointer.

**Distributed parallelizations.** It is often desirable to partition the graph structure amongst multiple processors, particularly for datasets too large to fit within the physical memory of a single machine. Even for shared-memory SMP platforms, recent research has shown it to be advantageous to partition the graph amongst the different CPU sockets; a given socket will have higher throughput to the specific memory managed by its local DDR channels [2].

The typical partitioning approach is to assign each processing element a disjoint subset of  $V$  and the corresponding adjacency lists in  $E$ . For a given vertex  $v_i$ , the inspection and marking of  $v_i$  as well as the expansion of  $v_i$ 's adjacency list must occur on the processor that owns  $v_i$ . Distributed, out-of-core edge queues are used for communicating neighbors to remote processors. Listing 8 describes the general method. Incoming neighbors that are unvisited have their labels marked and their adjacency lists expanded. As adjacency lists are expanded, neighbors are enqueued to the processor that owns them. The synchronization between BFS levels occurs after the expansion phase.

It is important to note that distributed BFS implementations that construct predecessor trees will impose twice the queuing I/O as those that construct depth-rankings. These variants must forward the full edge pairing  $(v_i, v_j)$  to the remote processor so that it might properly label  $v_j$ 's predecessor as  $v_i$ .

Yoo *et al.* [120] present a variation for BlueGene/L that implements a two-dimensional partitioning strategy for reducing the number of remote peers each processor must communicate with. Xia and Prasanna [118] propose a variant for multi-socket nodes that provisions more out-of-core edge-frontier queues than active threads, reducing the contention at any given queue and flexibly lowering barrier overhead.

Agarwal *et al.* [2] describe a two-phase implementation for multi-socket systems that implements both out-of-core vertex and edge-frontier queues for each socket. As a hybrid of Listing 7 and Listing 8, only remote edges are queued out-of-core. Edges that are local are inspected and filtered in-core. After a global synchronization, a second phase is performed to filter edges from remote sockets. Their implementation uses a

single, global, atomically-updated bitmask to reduce the overhead of inspecting a given vertex’s visitation status.

Scarpazza *et al.* [100] describe a similar hybrid variation for the Cell BE processor architecture. Instead of separate contraction phase per iteration, processor cores perform edge expansion, exchange, and contraction in batches. DMA engines are used instead of threads to perform parallel adjacency list expansion. Their implementation requires an offline preprocessing step that sorts and encodes adjacency lists into segments packaged by processor core.

***Our parallelization strategy.*** In comparison, our BFS strategy expands adjacent neighbors in parallel; implements out-of-core edge and vertex-frontiers; uses local prefix-sum in place of local atomic operations for determining enqueue offsets; and uses a best-effort bitmask for efficient neighbor filtering. We further describe the details in Section 6.5.

## 6.3 BENCHMARK SUITE














### 6.3.1 Graph Datasets

Our benchmark suite is composed of the thirteen graphs listed in Table 1. We generate the square and cubic Poisson lattice graph datasets ourselves. The *random.2Mv.128Me* and *rmat.2Mv.128Me*<sup>14</sup> datasets are constructed using GTgraph [53]. The *wikipedia-20070206* dataset is from the University of Florida Sparse Matrix Collection [115]. The remaining datasets are from the 10<sup>th</sup> DIMACS Implementation Challenge [1].

One of our goals is to demonstrate good performance for large-diameter graphs. The largest components within these datasets have diameters spreading five orders of

---

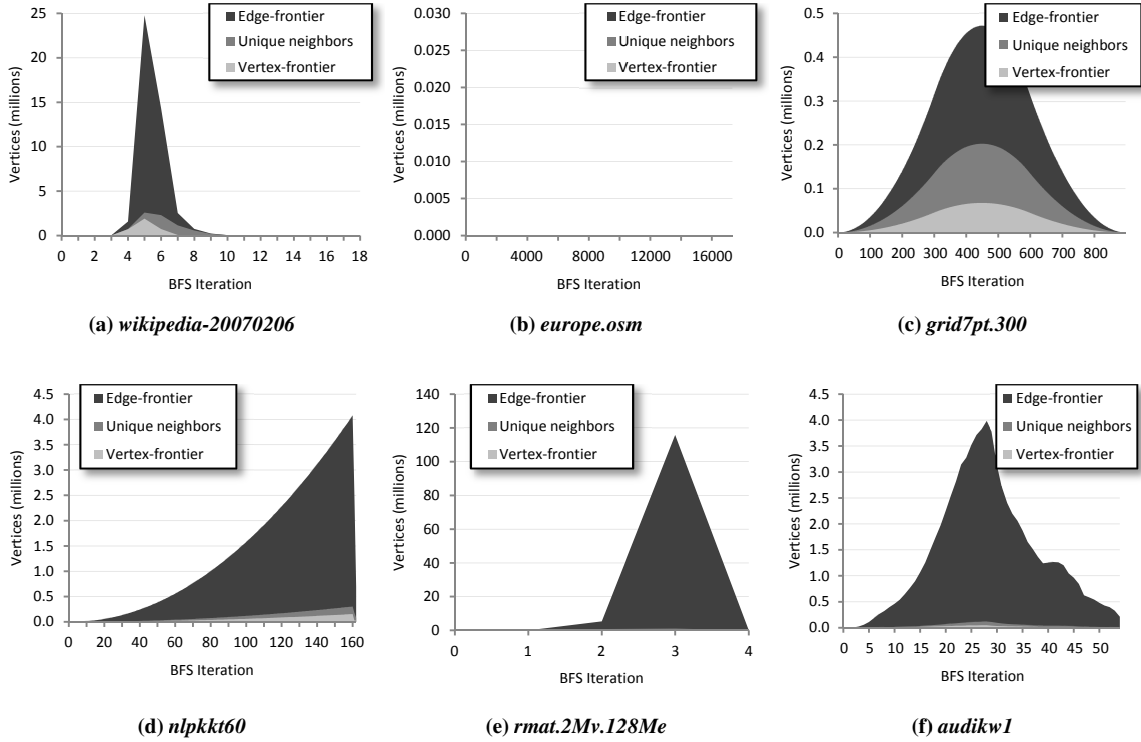
<sup>14</sup> RMAT graphs are synthetic graph constructions having power-law degree distributions and “small-world” connectivity [25].

Table 9. Suite of benchmark graphs						
Name	Sparsity Plot	Description	$n$ ( $10^6$ )	$m$ ( $10^6$ )	$\bar{d}$	Avg. Search Depth
europe.osm		European road network	50.9	108.1	2.1	19314
grid5pt.5000		5-point Poisson stencil (2D grid lattice)	25.0	125.0	5.0	7500
hugebubbles-00020		Adaptive numerical simulation mesh	21.2	63.6	3.0	6151
grid7pt.300		7-point Poisson stencil (3D grid lattice)	27.0	188.5	7.0	679
nlpkkt160		3D PDE-constrained optimization	8.3	221.2	26.5	142
audikw1		Automotive finite element analysis	0.9	76.7	81.3	62
cage15		Electrophoresis transition probabilities	5.2	94.0	18.2	37
kkt_power		Nonlinear optimization (KKT)	2.1	13.0	6.3	37
coPapersCiteseer		Citation network	0.4	32.1	73.9	26
wikipedia-20070206		Links between Wikipedia pages	3.6	45.0	12.6	20
kron_g500-logn20		Graph500 RMAT ( $A=0.57, B=0.19, C=0.19$ )	1.0	100.7	96.0	6
random.2Mv.128Me		$G(n, M)$ uniform random	2.0	128.0	64.0	6
rmat.2Mv.128Me		RMAT ( $A=0.45, B=0.15, C=0.15$ )	2.0	128.0	64.0	6

magnitude. Graph diameter is directly proportional to average search depth, the expected number of BFS iterations for a randomly-chosen source vertex.

### 6.3.2 Logical Frontier Plots

Although our sparsity plots reveal a diversity of locality, they provide little intuition as to how traversal will unfold. Fig. 61 presents sample *frontier plots* of logical edge and



**Fig. 61.** Sample frontier plots of logical vertex and edge-frontier sizes during graph traversal.

vertex-frontier sizes as functions of BFS iteration. Such plots help visualize workload expansion and contraction, both within and between iterations. The ideal numbers of neighbors expanded and vertices labeled per iteration are constant properties of the given dataset and starting vertex.

Frontier plots reveal the concurrency exposed by each iteration. For example, the bulk of the work for the *wikipedia-20070206* dataset is performed in only 1-2 iterations. The hardware can easily be saturated during these iterations. We observe that real-world datasets often have long sections of light work that incur heavy global synchronization overhead.

Finally, Fig. 61 also plots the duplicate-free subset of the edge-frontier. We observe that a simple duplicate-removal pass can perform much of the contraction work from edge-frontier down to vertex-frontier. This has important implications for

distributed BFS. The amount of network traffic can be significantly reduced by first removing duplicates from the expansion of remote neighbors.

We note the direct application of this technique does not scale linearly with processors. As  $p$  increases, the number of available duplicates in a given partition correspondingly decreases. In the extreme where  $p = m$ , each processor owns only one edge and there are no duplicates to be locally culled. For large  $p$ , such decoupled duplicate-removal techniques should be pushed into the hierarchical interconnect. Yoo *et al.* demonstrate a variant of this idea for BlueGene/L using their MPI set-union collective [120].

## 6.4 MICRO-BENCHMARK ANALYSES

A linear BFS workload is composed of two components:  $O(n)$  work related to vertex-frontier processing, and  $O(m)$  for edge-frontier processing. Because the edge-frontier is dominant, we focus our attention on the two fundamental aspects of its operation: *neighbor-gathering* and *status-lookup*. Although their functions are trivial, the GPU machine model provides interesting challenges for these workloads. We investigate these two activities in the following analyses using NVIDIA Tesla C2050 GPUs.

### 6.4.1 *Isolated Neighbor Gathering*

This analysis investigates serial and parallel strategies for simply gathering neighbors from adjacency lists. The enlistment of threads for parallel gathering is a form task scheduling. We evaluate a spectrum of scheduling granularity from individual tasks (higher scheduling overhead) to blocks of tasks (higher underutilization from partial-filling). We show the serial-expansion and warp-centric techniques described by prior work underutilize the GPU for entire genres of sparse graph datasets.

**Listing 9.** GPU pseudo-code for warp-based, strip-mined neighbor-gathering

**Input:** Vertex-frontier  $Q_{vfront}$ , column-indices array  $C$ , and the offset  $cta\_offset$  for the current tile within  $Q_{vfront}$   
**Functions:**  $WarpAny(pred_i)$  returns true if any  $pred_i$  is set for any thread  $t_i$  within the warp.

```

1  GatherWarp(cta_offset, Q_vfront, C) {
2      volatile shared comm[WARPS][3];
3      {r, r_end} =
4          Q_vfront[cta_offset + thread_id];
5      while (WarpAny(r_end - r)) {
6
7          // vie for control of warp
8          if (r_end - r)
9              comm[warp_id][0] = lane_id;
10
11         // winner describes adjlist
12         if (comm[warp_id][0] == lane_id) {
13             comm[warp_id][1] = r;
14             comm[warp_id][2] = r_end;
15             r = r_end;
16         }
17
18         // strip-mine winner's adjlist
19         r_gather = comm[warp_id][1] + lane_id;
20         r_gather_end = comm[warp_id][2];
21         while (r_gather < r_gather_end) {
22             volatile neighbor = C[r_gather];
23             r_gather += WARP_SIZE;
24         }
25     }
26 }
```

**Listing 10.** GPU pseudo-code for fine-grained, scan-based neighbor-gathering

**Input:** Vertex-frontier  $Q_{vfront}$ , column-indices array  $C$ , and the offset  $cta\_offset$  for the current tile within  $Q_{vfront}$   
**Functions:**  $CtaPrefixSum(val_i)$  performs a CTA-wide prefix sum where each thread  $t_i$  is returned the pair  $\{\sum_{k=0}^{i-1} val_k, \sum_{k=0}^{CTA\_THREADS-1} val_k\}$ .  $CtaBarrier()$  performs a barrier across threads within the CTA.

```

1  GatherScan(cta_offset, Q_vfront, C) {
2      shared comm[CTA_THREADS];
3      {r, r_end} =
4          Q_vfront[cta_offset + thread_id];
5      // reserve gather offsets
6      {rsv_rank, total} =
7          CtaPrefixSum(r_end - r);
8      // process fine-grained batches of
9      // adjlists
10     cta_progress = 0;
11     while ((remain =
12         total - cta_progress) > 0)
13     {
14         // share batch of gather offsets
15         while((rsv_rank < cta_progress +
16             CTA_THREADS) && (r < r_end))
17         {
18             comm[rsv_rank-cta_progress] = r;
19             rsv_rank++;
20             r++;
21         }
22         CtaBarrier();
23         // gather batch of adjlist(s)
24         if (thread_id <
25             Min(remain, CTA_THREADS))
26         {
27             volatile neighbor =
28                 C[comm[thread_id]];
29         }
30         cta_progress += CTA_THREADS;
31         CtaBarrier();
32     }
33 }
```

For a given BFS iteration, our test kernels simply read an array of preprocessed row-ranges that reference the adjacency lists to be expanded and then load the corresponding neighbors into local registers. (For full BFS, we do not perform any preprocessing.)





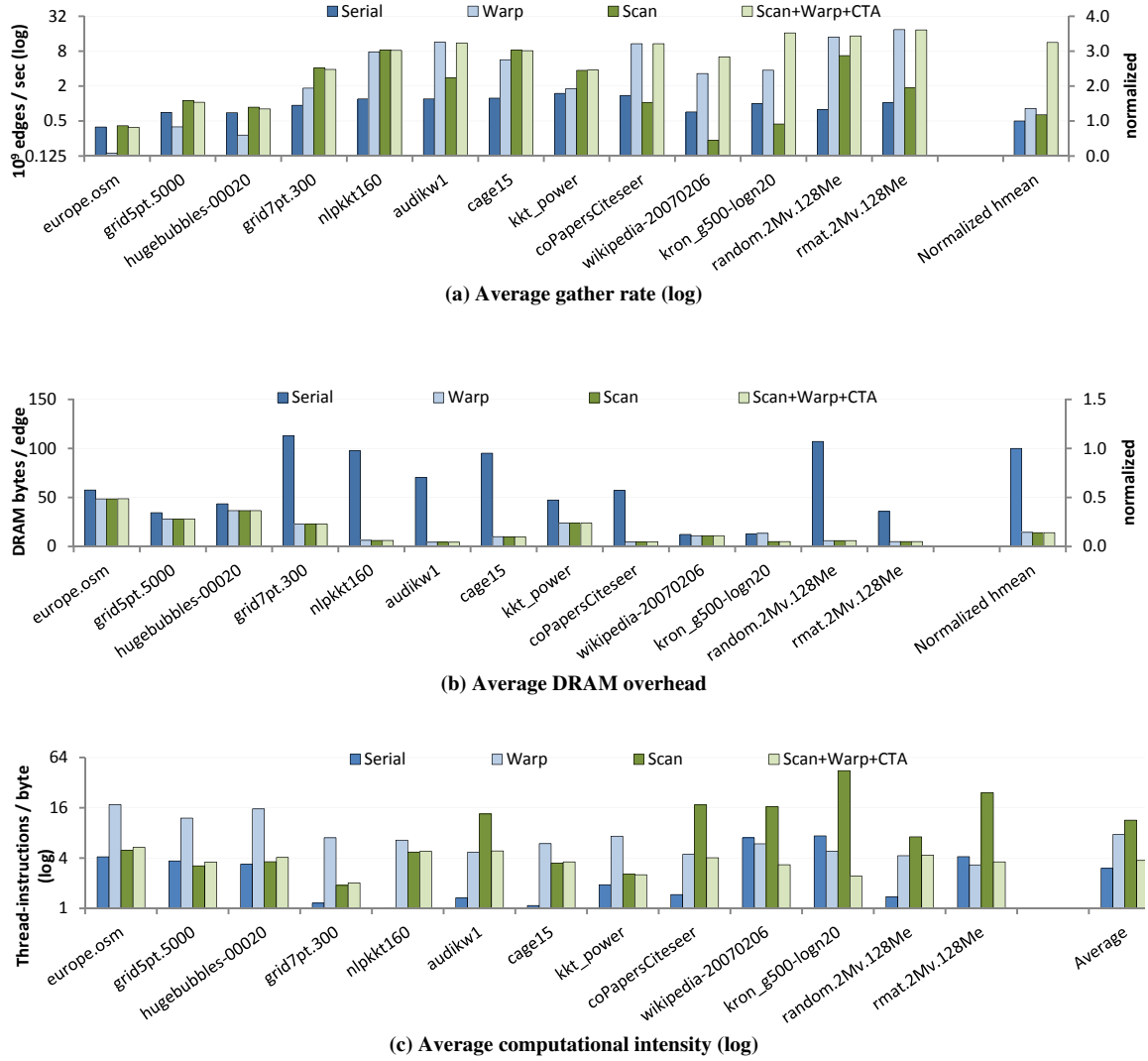
Compared to the two previous strategies, the entire CTA participates in every read. Any workload imbalance between threads is not magnified by expensive global memory accesses to  $C$ . Instead, workload imbalance can occur in the form of underutilized cycles during offset-sharing. The worst case entails a single thread having more neighbors than the gather buffer can accommodate, resulting in the idling of all other threads while it alone shares gather offsets.

***Scan+warp+CTA gathering.*** We can mitigate this imbalance by supplementing fine-grained *scan-based* expansion with coarser CTA-based and warp-based expansion. We first apply a CTA-wide version of *warp-based* gathering. This allows threads with very large adjacency lists to vie for control of the entire CTA, the winner broadcasting its row-range to all threads. Any large adjacency lists are strip-mined using the width of the entire CTA. Then we apply *warp-based* gathering to acquire portions of adjacency lists greater than or equal to the warp width. Finally we perform *scan-based* gathering to acquire the remaining “loose ends”.

This hybrid strategy limits all forms of load imbalance from adjacency list expansion. Fine-grained scan-based distribution limits imbalance from SIMD lane underutilization. Warp enlistment limits offset-sharing imbalance between threads. CTA enlistment limits imbalance between warps. And finally, any imbalance between CTAs can be limited by oversubscribing GPU cores with an abundance of CTAs and/or implementing coarse-grained tile-stealing mechanisms for CTAs to dequeue tiles<sup>15</sup> at their own rate.

---

<sup>15</sup> We term *tile* to describe a block of input data that a CTA is designed to process to completion before terminating or obtaining more work.



**Fig. 63.** Neighbor-gathering behavior. Harmonic means are normalized with respect to serial-gathering.

**Analysis.** We performed 100 randomly-sourced traversals of each dataset, evaluating these kernels on the logical vertex-frontier for every iteration. Fig. 63a plots the average edge-processing throughputs for each strategy in log-scale. The datasets are ordered from left-to-right by decreasing average search depth.

The *serial* approach performs poorly for the majority of datasets. Fig. 63b reveals it suffers from dramatic over-fetch. It plots bytes moved through DRAM per edge. The

arbitrary references from each thread within the warp result in terrible coalescing for SIMD load instructions.

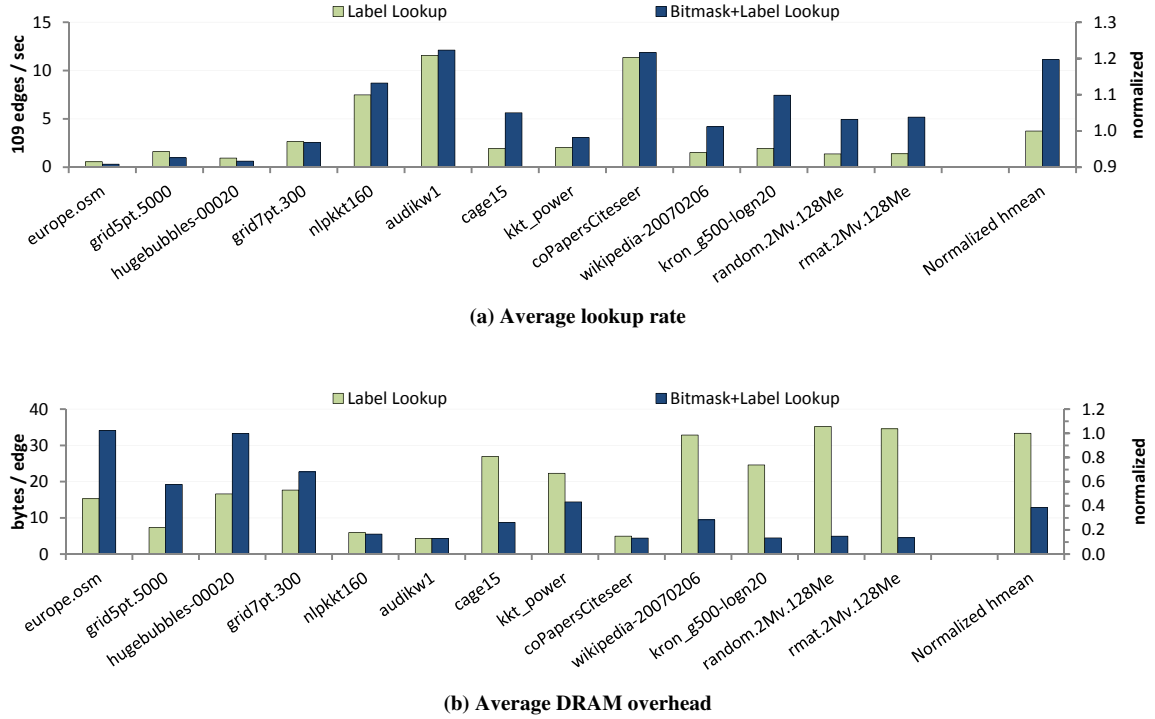
The *warp-based* approach performs poorly for the graphs on the left-hand side having  $\bar{d} \leq 10$ . Fig. 63c reveals that it is computationally inefficient for these datasets. It plots a log scale of computational intensity, the ratio of thread-instructions versus bytes moved through DRAM. The average adjacency lists for these graphs are much smaller than the number of threads per warp. As a result, a significant number of SIMD lanes go unused during any given cycle.

Fig. 63c also reveals that that *scan-based* gathering can suffer from extreme workload imbalance when only one thread is active within the entire CTA. This phenomenon is reflected in the datasets on the right-hand side having skewed degree distributions. The load imbalance from expanding large adjacency lists leads to increased instruction counts and corresponding performance degradation.

Combining the benefits of bulk-enlistment with fine-grained utilization, the hybrid *scan+warp+cta* demonstrates good gathering rates across the board.

#### 6.4.2 Isolated Status-lookup

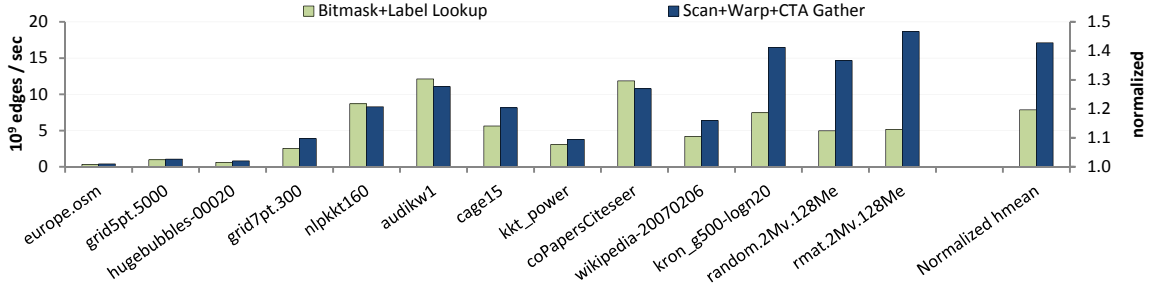
*Status-lookup* is the other half to neighbor-gathering; it entails determining which neighbors within the edge-frontier have already been visited. This section describes our analyses of status-lookup workloads, both in isolation and when coupled with neighbor-gathering within the same kernel. Although performing them separately requires more explicit memory traffic, we reveal that coupling the two within the same kernel invocation can cause TLB issues resulting in markedly worse performance.



**Fig. 64.** Status-lookup behavior. Harmonic means are normalized with respect to simple label-lookup.

Our strategy for status-lookup incorporates a bitmask to reduce the size of status data from a 32-bit label to a single bit per vertex. CPU parallelizations have used atomically-updated bitmask structures to reduce memory traffic via improved cache coverage [2, 100]. Because we avoid atomic operations, our bitmask is only a conservative approximation of visitation status. Bits for visited vertices may appear unset or may be “clobbered” due to false-sharing within a single byte. If a status bit is unset, we must then perform a second read to check the corresponding label to ensure the vertex is safe for marking. This scheme relies upon capacity and conflict misses to update stale bitmask data within the read-only texture caches.

Similar to the neighbor-gathering analysis, we isolate the status-lookup workload using a test-kernel that consumes the logical edge-frontier at each BFS iteration. Despite having much smaller and more transient last-level caches, Fig. 64 confirms the technique



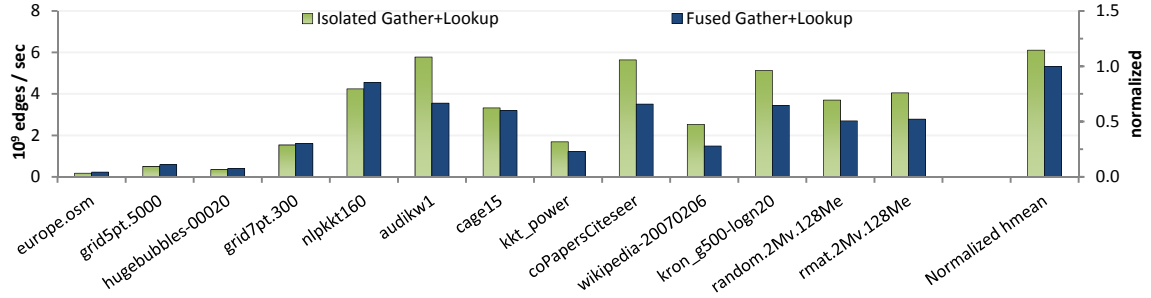
**Fig. 65.** Comparison of lookup vs. gathering.

can reduce global DRAM overhead and accelerate status-lookup for GPU architectures as well. The exceptions are the datasets on the left having a hundred or more BFS iterations. The bitmask is less effective for these datasets because texture caches are flushed between kernel invocations. Without coverage, the inspection often requires a second label lookup which further adds delay to latency-bound BFS iterations. As a result, we skip bitmask lookup for fleeting iterations having edge-frontiers smaller than the number of resident threads.

Fig. 65 compares the throughputs of lookup versus gathering workloads. We observe that status-lookup is generally the more expensive of the two. This is particularly true for the datasets on the right-hand side having high average vertex out-degree. The ability for neighbor-gathering to coalesce accesses to adjacency lists increases with  $\bar{d}$ , whereas accesses for status-lookup have arbitrary locality.

#### 6.4.3 Coupling of Gathering and Lookup

A complete BFS implementation might choose to fuse these workloads within the same kernel in order to process one of the frontiers online and in-core. We evaluate this fusion with a derivation of our *scan+warp+cta* gathering kernel that immediately inspects every gathered neighbor using our bitmap-assisted lookup strategy. The coupled kernel



**Fig. 66.** Comparison of isolated vs. fused lookup and gathering.

requires  $O(m)$  less overall data movement than the other two put together (which effectively read all edges twice).

Fig. 66 compares this fused kernel with the aggregate throughput of the isolated gathering and lookup workloads performed separately. Despite the additional data movement, the separate kernels outperform the fused kernel for the saturating benchmarks (right two-thirds of the chart). However, the extra data movement of separate kernels results in net slowdown for the latency-bound datasets having limited bulk concurrency (left-hand side).

The fused kernel likely suffers from TLB misses experienced by the neighbor-gathering workload. The column-indices arrays occupy substantial portions of GPU physical memory. Sparse gathers from them are apt to cause TLB misses. The fusion of these two workloads inherits the worst aspects of both: TLB turnover during uncoalesced status lookups.

The implication is that fused approaches are preferable for fleeting BFS iterations having edge-frontiers smaller than the number of resident threads. For graphs with abundant concurrency, however, the fusion of neighbor expansion and inspection yields worse performance than performing them separately.

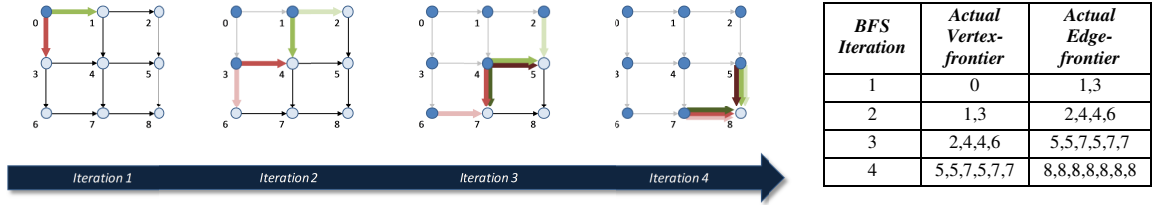


Fig. 67. Example of redundant adjacency list expansion due to concurrent discovery

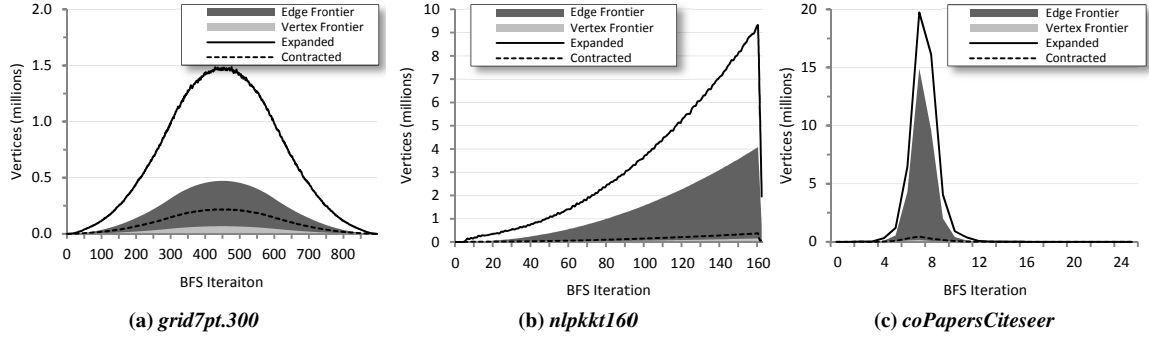
#### 6.4.4 Concurrent Discovery

Duplicate vertex identifiers within the edge-frontier are representative of different edges incident to the same vertex. This can pose a problem for implementations that allow the benign race condition. Adjacency lists will be expanded multiple times when multiple threads concurrently discover the same vertices via these duplicates. Without atomic updates to visitation status, we show the SIMD nature of the GPU machine model can introduce a significant amount of redundant work.

**Effect on overall workload.** Prior CPU parallelizations have noted the potential for redundant work, but concluded its manifestation to be negligible [78]. Concurrent discovery on CPU platforms is rare due to a combination of relatively low parallelism (~8 hardware threads) and coherent L1 caches that provide only a small window of opportunity around status-inspections that are immediately followed by status updates.

The GPU machine model, however, is much more vulnerable. If multiple threads within the same warp are simultaneously inspecting same vertex identifier, the SIMD nature of the warp-read ensures that all will obtain the same status value. If unvisited, the adjacency list for this vertex will be expanded for every thread.

Fig. 67 demonstrates an acute case of concurrent discovery. In this example, we traverse a small single-source, single-sink lattice using fine-grained cooperative expansion (e.g., Listing 10). For each BFS iteration, the cooperative behavior ensures



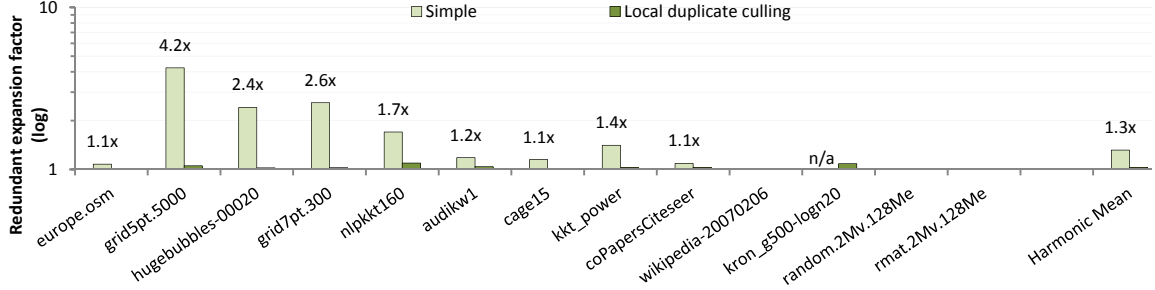
**Fig. 68.** Actual expanded and contracted queue sizes without local duplicate culling, superimposed over logical frontier sizes.

that all neighbors are gathered before any are inspected. No duplicates are culled from the edge frontier because SIMD lookups reveal every neighbor as being unvisited. The actual edge and vertex-frontiers diverge from ideal because no contraction occurs. This is cause for concern: the excess work grows geometrically, only slowing when the frontier exceeds the width of the machine or the graph ceases to expand.

We measure the effects of redundant expansion upon overall workload using a simplified version of the *two-phase* BFS implementation described in Section 6.5. These expansion and contraction kernels make no special effort to curtail concurrent discovery. For several sample traversals, Fig. 68 illustrates compounded redundancy by plotting the actual numbers of vertex identifiers expanded and contracted for each BFS iteration alongside the corresponding logical frontiers. The deltas between these pairs reflect the generation of unnecessary work.

We define the *redundant expansion factor* as the ratio of neighbors actually enqueued versus the number of edges logically traversed. Fig. 69 plots the redundant expansion factors measured for our *two-phase* implementation, both with and without extra measures to mitigate concurrent discovery. The problem is severe for spatially-descriptive datasets. These datasets exhibit nearby duplicates within the edge-frontier





**Fig. 69** Redundant work expansion incurred by variants of our *two-phase* BFS implementation. Unlabeled columns are  $< 1.05x$ .

due to their high frequency of convergent exploration. For example, simple two-phase traversal incurs 4.2x redundant expansion for the 2D lattice *grid5pt.5000* dataset. Even worse, the implementation altogether fails to traverse the *kron\_g500-logn20* dataset which encodes sorted adjacency lists. The improved locality enables the redundant expansion of ultra-popular vertices, ultimately exhausting physical memory when filling the edge queue.

This issue of redundant expansion appears to be unique to GPU BFS implementations having two properties: (1) a work-efficient traversal algorithm; and (2) concurrent adjacency list expansion. Quadratic implementations do not suffer redundant work because vertices are never expanded by more than one thread. In our evaluation of linear-work serial-expansion, we observed negligible concurrent SIMD discovery during serial inspection due to the independent nature of thread activity.

In general, the issue of concurrent discovery is a result of false-negatives during status-lookup, i.e., failure to detect previously-visited and duplicate vertex identifiers within the edge-frontier. Atomic read-modify-write updates to visitation status yield zero false-negatives. As alternatives, we introduce two localized mechanisms for reducing false-negatives: (1) *warp culling* and (2) *history culling*.

---

**Listing 11.** GPU pseudo-code for a localized, warp-based duplicate-detection heuristic.

---

**Input:** Vertex identifier *neighbor*

**Output:** True if *neighbor* is a conclusive duplicate within the warp's working set.

---

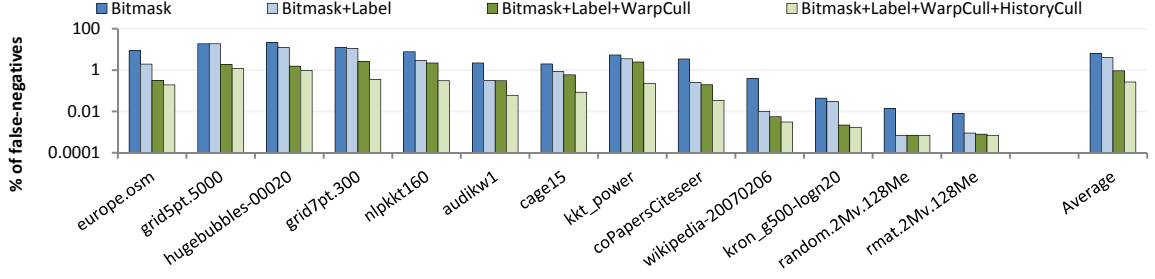
```

1  WarpCull(neighbor) {
2      volatile shared scratch[WARPS][128];
3      hash = neighbor & 127;
4      scratch[warp_id][hash] = neighbor;
5      retrieved = scratch[warp_id][hash];
6      if (retrieved == neighbor) {
7          // vie to be the "unique" item
8          scratch[warp_id][hash] = thread_id;
9          if (scratch[warp_id][hash] !=
10             thread_id)
11          {
12              // someone else is unique
13              return true;
14          }
15      }
16      return false;
17  }
```

---

**Warp culling.** Listing 11 describes this heuristic for preventing concurrent SIMD discovery by detecting the presence of duplicates within the warp's immediate working set. Using shared-memory per warp, each thread hashes in the neighbor it is currently inspecting. If a collision occurs and a different value is extracted, nothing can be determined regarding duplicate status. Otherwise threads then write their thread-identifier into the same hash location. Only one write will succeed. Threads that subsequently retrieve a different thread-identifier can safely classify their neighbors as duplicates to be culled.

**History culling.** This heuristic complements the instantaneous coverage of warp culling by maintaining a cache of recently-inspected vertex identifiers in local shared memory. If a given thread observes its neighbor to have been previously recorded, it can classify that neighbor as safe for culling.



**Fig. 70** Percentages of false-negatives incurred by status-lookup strategies.

**Analysis.** We augment our isolated lookup tests to evaluate these heuristics. Kernels simply read vertex identifiers from the edge-frontier and determine which should not be allowed into the vertex-frontier. For each dataset, we record the average percentage of false negatives with respect to  $m - n$ , the ideal number of culled vertex identifiers.

Fig. 70 illustrates the progressive application of lookup mechanisms. The bitmask heuristic alone incurs an average false-negative rate of 6.4% across our benchmark suite. The addition of label-lookup (which makes status-lookup safe) improves this to 4.0%. Without further measure, the compounding nature of redundant expansion allows even small percentages to accrue sizeable amounts of extra work. For example, a false-negative rate of 3.5% for traversing *kkt\_power* results in a 40% redundant expansion overhead.

The addition of warp-based culling induces a tenfold reduction in false-negatives for spatially descriptive graphs (left-hand side). The history-based culling heuristic further reduces culling inefficiency by a factor of five for the remainder of high-risk datasets (middle-third). The application of both heuristics allows us to reduce the overall redundant expansion factor to less than 1.05x for every graph in our benchmark suite.

## 6.5 SINGLE-GPU PARALLELIZATIONS

A complete solution must couple expansion and contraction activities. In this section, we evaluate the design space of coupling alternatives:

- 1) *Expand-contract*. A single kernel consumes the current vertex-frontier and produces the vertex-frontier for the next BFS iteration.
- 2) *Contract-expand*. The converse. A single kernel contracts the current edge-frontier, expanding unvisited vertices into the edge-frontier for the next iteration.
- 3) *Two-phase*. A given BFS iteration is processed by two kernels that separately implement out-of-core expansion and contraction.
- 4) *Hybrid*. This implementation invokes the *contract-expand* kernel for small, fleeting BFS iterations, otherwise the *two-phase* kernels.

We describe and evaluate BFS kernels for each strategy. We show the *hybrid* approach to be on-par-with or better-than the other three for every dataset in our benchmark suite.

### 6.5.1 *Expand-contract (out-of-core vertex queue)*

Our *expand-contract* kernel is loosely based upon the fused gather-lookup benchmark kernel from Section 6.4.3. It consumes the vertex queue for the current BFS iteration and produces the vertex queue for the next. It performs parallel expansion and filtering of adjacency lists online and in-core using local scratch memory.

A CTA performs the following steps when processing a tile of input from the incoming vertex-frontier queue:

- 1) Threads perform local warp-culling and history-culling to determine if their dequeued vertex is a duplicate.

- 2) If still valid, the corresponding row-range is loaded from the row-offsets array R.
- 3) Threads perform coarse-grained, CTA-based neighbor-gathering. Large adjacency lists are cooperatively strip-mined from the column-indices array C at the full width of the CTA. These strips of neighbors are filtered in-core and the unvisited vertices are enqueued into the output queue as described below.
- 4) Threads perform fine-grained, scan-based neighbor-gathering. These batches of neighbors are filtered and enqueued into the output queue as described below.

For each strip or batch of gathered neighbors:

- i. Threads perform status-lookup to invalidate the vast majority of previously-visited and duplicate neighbors.
- ii. Threads with a valid neighbor  $n_i$  update the corresponding label.
- iii. Threads then perform a CTA-wide prefix sum where each contributes a 1 if  $n_i$  is valid, 0 otherwise. This provides each thread with the scatter offset for  $n_i$  and the total count of all valid neighbors.
- iv.  $Thread_0$  obtains the base enqueue offset for valid neighbors by performing an atomic-add operation on a global queue counter using the total valid count. The returned value is shared to all other threads in the CTA.
- v. Finally, all valid  $n_i$  are written to the global output queue. The enqueue index for  $n_i$  is the sum of the base enqueue offset and the scatter offset.

This kernel requires  $2n$  global storage for input and output vertex queues. The roles of these two arrays are reversed for alternating BFS iterations. A traversal will generate  $5n+2m$  explicit data movement through global memory. All  $m$  edges will be streamed into registers once. All  $n$  vertices will be streamed twice: out into global frontier queues

and subsequently back in. The bitmask bits will be inspected  $m$  times and updated  $n$  times along with the labels. Each of the  $n$  row-offsets is loaded twice.

CTAs perform two or more local prefix-sums per tile. One is used for allocating room for gather offsets during scan-based gathering. We also need prefix sums to compute global enqueue offsets for every strip or batch of gathered neighbors. Although GPU cores can efficiently overlap concurrent prefix sums from different CTAs, the turnaround time for each can be relatively long. This can hurt performance for fleeting, latency-bound BFS iterations.

### 6.5.2 *Contract-expand (out-of-core edge queue)*

Our *contract-expand* kernel filters previously-visited and duplicate neighbors from the current edge queue. The adjacency lists of the surviving vertices are then expanded and copied out into the edge queue for the next iteration.

A CTA performs the following steps when processing a tile of input from the incoming edge-frontier queue:

- 1) Threads progressively test their neighbor vertex identifier  $n_i$  for validity using (i) status-lookup; (ii) warp-based duplicate culling; and (iii) history-based duplicate culling.
- 2) Threads update labels for valid  $n_i$  and obtain the corresponding row-ranges from  $R$ .
- 3) Threads then perform two concurrent CTA-wide prefix sums: the first for computing enqueue offsets for coarse-grained warp and CTA neighbor-gathering, the second for fine-grained scan-based gathering.  $|A_i|$  is contributed to the first prefix sum if greater than  $WARP\_SIZE$ , otherwise to the second.

- 4) *Thread*<sub>0</sub> obtains a base enqueue offset for valid neighbors within the entire tile by performing an atomic-add operation on a global queue counter using the combined totals of the two prefix sums. The returned value is shared to all other threads in the CTA.
- 5) Threads then perform coarse-grained CTA and warp-based gathering. When a thread commandeers its CTA or warp, it also communicates the base scatter offset for  $n_i$  to its peers. After gathering neighbors from  $C$ , enlisted threads enqueue them to the global output queue. The enqueue index for each thread is the sum of the base enqueue offset, the shared scatter offset, and thread-rank.
- 6) Finally, threads perform fine-grained scan-based gathering. This procedure is a variant of Listing 10 with the prefix sum being hoisted out and performed earlier in Step 4. After gathering packed neighbors from  $C$ , threads enqueue them to the global output. The enqueue index is the sum of the base enqueue offset, the coarse-grained total, the CTA progress, and thread-rank.

This kernel requires  $2m$  global storage for input and output edge queues. Variants that label predecessors, however, require an additional pair of “parent” queues to track both origin and destination identifiers within the edge-frontier. A traversal will generate  $3n+4m$  explicit global data movement. All  $m$  edges will be streamed through global memory three times: into registers from  $C$ , out to the edge queue, and back in again the next iteration. The bitmask, label, and row-offset traffic remain the same as for *expand-contract*.

Despite a much larger queuing workload, the *contract-expand* strategy is often better suited for processing small, fleeting BFS iterations. It incurs lower latency because

CTAs only perform local two prefix sums per block. We overlap these prefix-sums to further reduce latency. By operating on the larger edge-frontier, the *contract-expand* kernel also enjoys better bulk concurrency in which fewer resident CTAs sit idle.

### 6.5.3 *Two-phase (out-of-core vertex and edge queues)*

Our *two-phase* implementation isolates the expansion and contraction workloads into separate kernels. Our micro-benchmark analyses suggest this design for better overall bulk throughput. The expansion kernel employs the *scan+warp+cta* gathering strategy to obtain the neighbors of vertices from the input vertex queue. As with the *contract-expand* implementation above, it performs two overlapped local prefix-sums to compute scatter offsets for the expanded neighbors into the global edge queue.

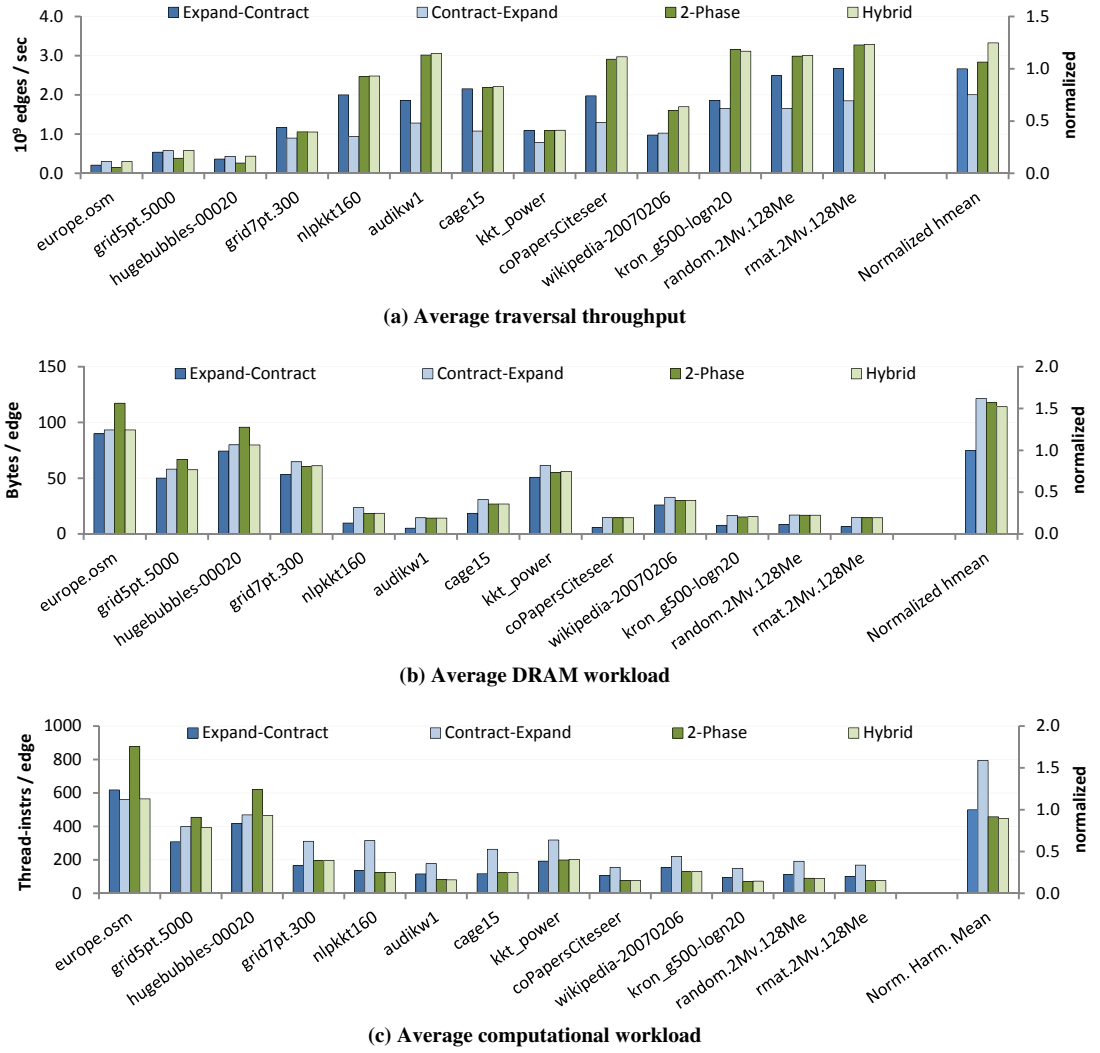
The contraction kernel begins with the edge queue as input. Threads filter previously-visited and duplicate neighbors. The remaining valid neighbors are placed into the outgoing vertex queue using another local prefix sum to compute global enqueue offsets.

These kernels require  $n+m$  global storage for vertex and edge queues. A *two-phase* traversal generates  $5n+4m$  explicit global data movement. The memory workload builds upon that of *contract-expand*, but additionally streams  $n$  vertices into and out of the global vertex queue.

### 6.5.4 *Hybrid*

Our hybrid implementation combines the relative strengths of the *contract-expand* and *two-phase* approaches: low-latency turnaround for small frontiers and high-efficiency throughput for large frontiers. If the edge queue for a given BFS iteration contains more vertex identifiers than resident threads, we invoke the *two-phase* implementation for that



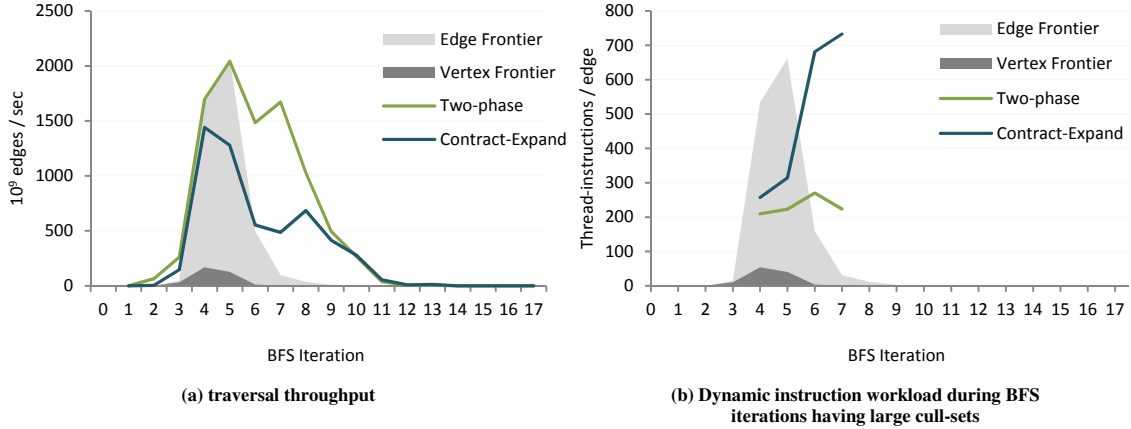


**Fig. 71** BFS traversal performance and workloads. Harmonic means are normalized with respect to the *expand-contract* implementation.

iteration. Otherwise we invoke the *contract-expand* implementation. The hybrid approach inherits the  $2m$  global storage requirement from the former and the  $5n+4m$  explicit global data movement from the latter.

### 6.5.5 Evaluation

Our performance analyses are constructed from 100 randomly-sourced traversals of each dataset. Fig. 71 plots average traversal throughput. As anticipated, the *contract-expand* approach excels at traversing the latency-bound datasets on the left and the *two-phase*



**Fig. 72.** Sample *wikipedia-20070206* traversal behavior. Plots are superimposed over the shape of the logical edge and vertex-frontiers.

implementation efficiently leverages the bulk-concurrency exposed by the datasets on the right. Although the *expand-contract* approach is serviceable, the *hybrid* approach meets or exceeds its performance for every dataset.

**The importance of work compaction.** With in-core edge-frontier processing, the *expand-contract* implementation is designed for one-third as much global queue traffic. The actual DRAM savings are substantially less. We only see a 50% reduction in measured DRAM workload for datasets with large  $\bar{d}$ . Furthermore, the workload differences are effectively lost in excess over-fetch traffic for the graphs having small  $\bar{d}$ : their small adjacency lists only occupy a small proportion of the memory transactions used to retrieve them.

The *contract-expand* implementation performs poorly for graphs having large  $\bar{d}$ . This behavior is related to a lack of explicit workload compaction before neighbor gathering. Fig. 72 illustrates this using a sample traversal of *wikipedia-20070206*. We observe a correlation between large contraction workloads during iterations 4-6 and significantly elevated dynamic thread-instruction counts. This is indicative of SIMD

Graph Dataset	Sequential CPU <sup>†</sup>	State-of-the-art parallel CPU		NVIDIA Tesla C2050 ( <i>hybrid strategy</i> )			
	10 <sup>9</sup> TE/s	10 <sup>9</sup> TE/s	Speedup vs. sequential CPU	Distance Labeling		Predecessor Labeling	
				10 <sup>9</sup> TE/s	Speedup vs. sequential CPU	10 <sup>9</sup> TE/s	Speedup vs. sequential CPU
europe.osm	0.029			0.31	11x	0.31	11x
grid5pt.5000	0.081			0.60	7.3x	0.57	7.0x
hugebubbles-00020	0.029			0.43	15x	0.42	15x
grid7pt.300	0.038	0.12 <sup>††</sup>	3.0x	1.1	28x	0.97	26x
nlpkt160	0.26	0.47 <sup>††</sup>	1.8x	2.5	9.6x	2.1	8.3x
audikw1	0.65			3.0	4.6x	2.5	4.0x
cage15	0.13	0.23 <sup>††</sup>	1.8x	2.2	18x	1.9	15x
kkt_power	0.047	0.11 <sup>††</sup>	2.2x	1.1	23x	1.0	21x
coPapersCiteseer	0.50			3.0	5.9x	2.5	5.0x
wikipedia-20070206	0.065	0.19 <sup>††</sup>	2.7 x	1.6	25x	1.4	22x
kron_g500-logn20	0.24			3.1	13x	2.5	11x
random.2Mv.128Me	0.10	0.50 <sup>†††</sup>	5.0 x	3.0	29x	2.4	23x
rmat.2Mv.128Me	0.15	0.70 <sup>†††</sup>	4.6 x	3.3	22x	2.6	18x

**Table 10.** Single-socket performance comparison.

GPU speedup is in regard to sequential CPU performance. <sup>†</sup>3.4GHz Core i7 2600K. <sup>††</sup>2.5 GHz Core i7 4-core, distance-labeling [78]. <sup>†††</sup>2.7 GHz Xeon X5570 8-core, predecessor labeling [2].

underutilization. The majority of active threads have their neighbors invalidated by status-lookup and local duplicate removal. Cooperative neighbor-gathering becomes much less efficient as a result.

*Distance vs. predecessor labeling.* Table 10 presents *hybrid* traversal performance for distance and predecessor labeling variants. The performance difference between variants is largely dependent upon  $\bar{d}$ . Smaller  $\bar{d}$  incurs larger DRAM over-fetch which reduces the relative significance of added parent queue traffic. For example, the performance impact of exchanging parent vertices is negligible for *europe.osm*, yet is as high as 19% for *rmat.2Mv.128Me*.

*Comparison of our hybrid strategy with prior work.* It is challenging to contrast traversal performance for CPU and GPU architectures. The construction of high performance CPU parallelizations is outside the scope of this work, and published studies of CPU traversal have not reported performance results for all of the datasets in our benchmark corpus.

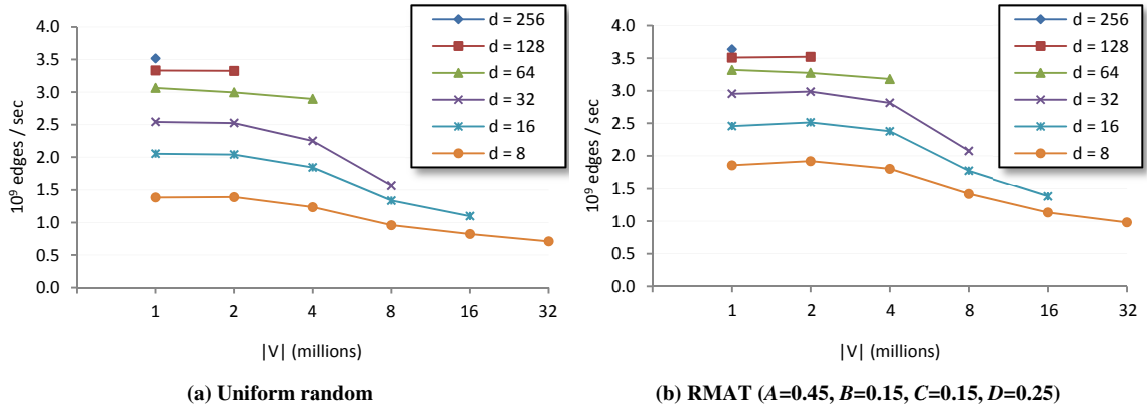


Fig. 73. NVIDIA C2050 traversal throughput.

However, we make the observation that parallel CPU implementations have been unable to achieve linear performance scaling with respect to processor cores [2, 78]. Hedging in favor of CPU performance, we compare our GPU traversal performance with an implementation of the sequential method and then assume a hypothetical CPU parallelization with perfect linear scaling per core. As such, we consider a 4-8x GPU speedup versus our sequential CPU implementation on a state-of-the-art 3.4GHz Intel Core i7 2600K (Sandybridge) as being competitive with contemporary parallel CPU traversal<sup>16</sup>.

If we were to assume 4x scaling across all four 2600K CPU cores, our C2050 traversal rates would outperform the CPU for all benchmark datasets. In addition, the majority of our graph traversal rates exceed 12x speedup, the perfect scaling of three such CPUs. At the extreme, our average *wikipedia-20070206* traversal rates outperform the sequential CPU version by 25x, i.e., eight CPU-equivalents.

<sup>16</sup> Our sequential implementation exceeds the single-threaded results reported by Leiserson et al. [78] and Agarwal et al. [2] despite having fewer memory channels

We also note that our methods perform well for large and small-diameter graphs alike. Comparing with sequential CPU traversals of *europe.osm* and *kron\_g500-logn20*, our *hybrid* strategy provides an order-of-magnitude speedup for both.

In comparing with state-of-the-art GPU implementations, we evaluated the quadratic implementation provided by Hong *et al.* [63] on our benchmark datasets. Their work-inefficient, quadric method suffers from high overhead and was not competitive on even the lowest diameter graphs in our experimental corpus. At best, their implementation achieved an average 2.1x slowdown for *kron\_g500-logn20*. At worst, a 2,300x slowdown for *europe.osm*. For *wikipedia-20070206*, a 4.1x slowdown.

We use a previous-generation NVIDIA GTX280 to compare our implementation with the results reported by Luo *et al.* for their linear parallelization [79]. We achieve 4.1x and 1.7x harmonic mean speedups for the referenced 6-pt grid lattices and DIMACS road network datasets, respectively.

***Uniform-random and RMAT-scaling.*** Fig. 73 further presents C2050 traversal performance for synthetic uniform-random and RMAT datasets having up to 256 million edges. Each plotted rate is averaged from 100 randomly-sourced traversals.

Our maximum traversal rates of 3.5B and 3.6B TE/s occur with  $\bar{d} = 256$  for uniform-random and RMAT datasets having 256M edges, respectively. The minimum rates plotted are 710M and 982M TE/s for uniform-random and RMAT datasets having  $\bar{d} = 8$  and 256M edges. Performance incurs a drop-off at  $n=8$  million vertices when the bitmask exceeds the 768KB L2 cache size.

## 6.6 MULTI-GPU PARALLELIZATION

Communication between GPUs is simplified by a unified virtual address space in which pointers can transparently reference data residing within remote GPUs. PCI-express 2.0 provides each GPU with an external bidirectional bandwidth of 6.6 GB/s. Under the assumption that GPUs send and receive equal amounts of traffic, the rate at which each GPU can be fed with remote work is conservatively bound by  $825 \times 10^6$  neighbors / sec, where neighbors are 4-byte identifiers. This rate is halved for predecessor-labeling variants.

### 6.6.1 Design

We implement a simple partitioning of the graph into equally-sized, disjoint subsets of  $V$ . For a system of  $p$  GPUs, we initialize each processor  $p_i$  with an  $(m/p)$ -element  $C_i$  and  $(n/p)$ -element  $R_i$  and  $Labels_i$  arrays. Because the system is small, we can provision each GPU with its own full-sized  $n$ -bit best-effort bitmask.

We stripe ownership of  $V$  across the domain of vertex identifiers. Striping provides good probability of an even distribution of adjacency list sizes across GPUs. This is particularly useful for graph datasets having concentrations of popular vertices. For example, RMat datasets encode the most popular vertices with the largest adjacency lists near the beginning of  $R$  and  $C$ . Alternatives that divide such data into contiguous slabs can be detrimental for small systems: (a) an equal share of vertices would overburden first GPU with an abundance of edges; or (b) an equal share of edges leaves the first GPU underutilized because it owns fewer vertices, most of which are apt to be filtered remotely. However, this method of partitioning progressively loses any inherent locality as the number of GPUs increases.

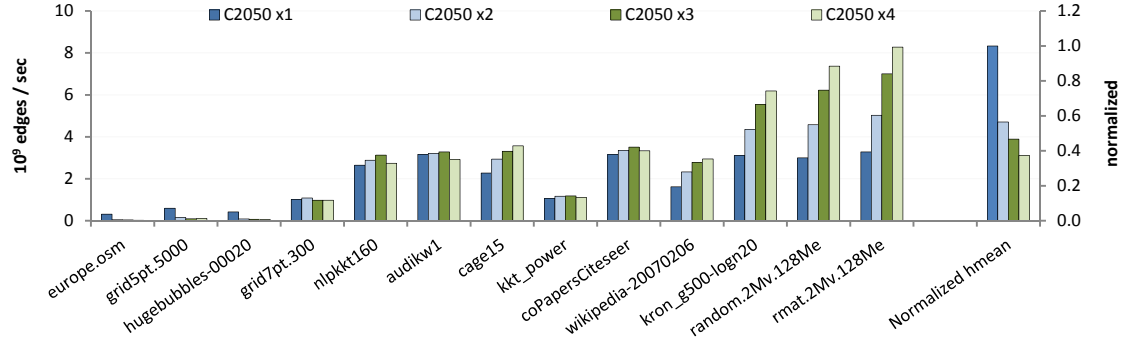
Graph traversal proceeds in level-synchronous fashion. The host program orchestrates BFS iterations as follows:

- 1) Invoke the *expansion* kernel on each GPU<sub>*i*</sub>, transforming the vertex queue  $Q_{vertex_i}$  into an edge queue  $Q_{edge_i}$ .
- 2) Invoke a fused *filter+partition* operation for each GPU<sub>*i*</sub> that sorts neighbors within  $Q_{edge_i}$  by ownership into  $p$  bins. Vertex identifiers undergo opportunistic local duplicate culling and bitmask filtering during the partitioning process. This partitioning implementation is analogous to the three-kernel radix-sorting pass described in Chapter 5.
- 3) Barrier across all GPUs. The sorting must be completed on all GPUs before any can access their bins on remote peers. The host program uses this opportunity to terminate traversal if all bins are empty on all GPUs.
- 4) Invoke  $p-1$  *contraction* kernels on each GPU<sub>*i*</sub> to stream and filter the incoming neighbors from its peers. Kernel invocation simply uses remote pointers that reference the appropriate peer bins. This assembles each vertex queue  $Q_{vertex_i}$  for the next BFS iteration.

The implementation requires  $(2m+n)/p$  storage for queue arrays per GPU: two edge queues for pre and post-sorted neighbors and a third vertex queue to avoid another global synchronization after Step 4.

### 6.6.2 Evaluation

Fig. 74 presents traversal throughput as we scale up the number of GPUs. We experience net slowdown for datasets on the left having average search depth  $> 100$ . The cost of global synchronization between BFS iterations is much higher across multiple GPUs.



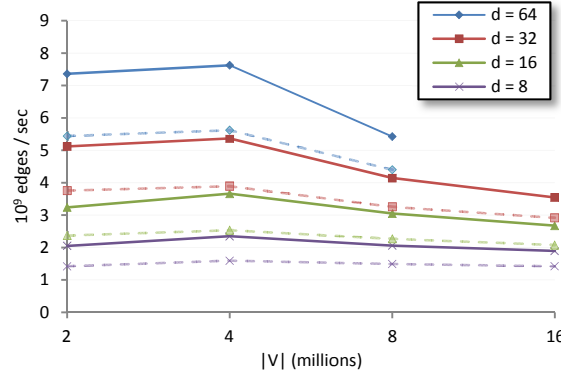
**Fig. 74.** Average multi-GPU traversal rates. Harmonic means are normalized with respect to the single GPU configuration.

We do yield notable speedups for the three rightmost datasets. These graphs have small diameters and require little global synchronization. The large average out-degrees enable plenty of opportunistic duplicate filtering during partitioning passes. This allows us to circumvent the PCI-e cap of  $825 \times 10^6$  edges/sec per GPU. With four GPUs, we demonstrate traversal rates of 7.4 and 8.3 billion edges/sec for the uniform-random and RMAT datasets respectively.

As expected, this strong-scaling is not linear. For example, we observe 1.5x, 2.1x, and 2.5x speedups when traversing *rmat.2Mv.128Me* using two, three, and four GPUs, respectively. Adding more GPUs reduces the percentage of duplicates per processor and increases overall PCI-e traffic.

Fig. 75 further illustrates the impact of opportunistic duplicate culling for uniform random graphs up to 500M edges and varying out out-degree  $\bar{d}$ . Increasing  $\bar{d}$  yields significantly better performance. Other than a slight performance drop at  $n=8$  million vertices when the bitmask exceeds the L2 cache size, graph size has little impact upon traversal throughput.





**Fig. 75.** Multi-GPU sensitivity to graph size and average out-degree  $\bar{d}$  for uniform random graphs using four C2050 processors. Dashed lines indicate predecessor labeling variants.

To our knowledge, these are the fastest traversal rates demonstrated by a single-node machine. The work by Agarwal et al. is representative of the state of the art in CPU parallelizations, demonstrating up to 1.3 billion edges/sec for both uniform-random and RMAT datasets using four 8-core Intel Nehalem-based XEON CPUs [2]. However, we note that the host memory on such systems can further accommodate datasets having tens of billions of edges.

## 6.7 CHAPTER SUMMARY

This chapter has demonstrated that GPUs are well-suited for sparse graph traversal and can achieve very high levels of performance on a broad range of graphs. We have presented a parallelization of BFS tailored to the GPU's requirement for large amounts of fine-grained, bulk-synchronous parallelism.

Furthermore, our implementation performs an asymptotically optimal amount of work. While quadratic-work methods might be acceptable in certain very narrow regimes [63, 64], they suffer from high overhead and did not prove effective on even the lowest diameter graphs in our experimental corpus. Our linear-work method compares

very favorably to state-of-the-art multicore implementations across our entire range of benchmarks, which spans five orders of magnitude in graph diameter.

Beyond graph search, this chapter distills several general themes for implementing sparse and dynamic problems for the GPU machine model:

- In contrast to coarse-grained parallelism common on multicore processors, GPU kernels cannot afford to have individual threads streaming through unrelated sections of data. Groups of GPU threads should cooperatively assist each other for data movement tasks.
- Fusing heterogeneous tasks does not always produce the best results. Global redistribution and compaction of fine-grained tasks can significantly improve performance when the alternative would allow significant load imbalance or underutilization.
- The relative memory traffic from global task redistribution can be less costly than anticipated. The data movement from reorganization may be insignificant in comparison to the actual over-fetch traffic from existing sparse memory accesses.
- It is useful to provide separate implementations for saturating versus fleeting workloads. Hybrid approaches can leverage a shorter code-path for retiring underutilized phases as quickly as possible.

## Chapter 7

### *Conclusion*

#### **7.1 SUMMARY**

This dissertation has addressed many of the challenges inherent to the construction of cooperative parallelizations for GPU architecture. Despite contemporary opinion to the contrary, we have shown GPU architecture to be exceptionally well-suited for computations having fine-grained, dynamic allocation dependences between concurrent tasks. Our primary examples are parallel sorting and graph traversal, two archetypal applications for this problem genre. Our implementations for both of these problems achieve the fastest published performance on any fully-programmable microarchitecture.

The ability to cooperatively reserve space within shared data structures is a fundamental aspect of parallel computing. Although atomic operations are presently the conventional tool for implementing concurrent data placement, our analyses show they are incongruous with the bulk-synchronous and SIMD nature of the GPU machine model. We demonstrate prefix sum as a superior alternative for implementing cooperative allocation among many parallel threads.

The development of efficient parallelizations for prefix sum was critical to this research. Our efficiency stems from *flexible granularity coarsening*, the ability to provide proper balance between serial and parallel phases of computation for the target architecture. By reducing the computational overhead of local prefix sum by several factors, we created an inflection point in the design space for many cooperative problems where it now becomes feasible to:

- Benefit from *kernel fusion*, i.e., the colocation of application-specific logic within prefix sum kernels with significantly reduced (or negligible) overhead
- Perform fine-grained workload redistribution

To demonstrate the viability and generality of our designs, we constructed cooperative GPU implementations for a variety of parallel list-processing primitives, evaluating their performance across a wide spectrum of problem sizes, types, and target architectures.

However, it became clear that “concrete” implementations are simply not performance-portable, particularly if we want to reuse intra-CTA subroutines for common tasks. Out of necessity, we developed a higher-level programming abstraction for *policy-based tuning* where the programmer expresses the “general shape” of their solution, leaving many of the performance sensitive details unbound. We found the C++ type system to be useful as a mechanism for specializing code generation via template metaprogramming, particularly as our tuning decisions affect data structure and layout within shared memory. Our autotuning results demonstrate the ability to consistently discover good specializations for the specific problem instance at hand.

## 7.2 LIMITATIONS AND FUTURE WORK

### 7.2.1 *The CTA serialization idiom*

CTA serialization has several drawbacks. The increased granularity of computation can lead to load imbalance within and among GPU cores. If the scheduling hardware within each core is unfair, warps within one CTA may repeatedly be given preference over those of another when either have ready candidates. The result is a long tail of processor underutilization after the preferred CTAs have completed and only disfavored CTAs have work remaining. We can curtail this effect to some degree by slightly oversubscribing each core with a constant amount of additional CTAs.

Register pressure can also become an issue for kernels having more complicated tile-processing logic. This is caused by increased register live ranges, a side effect of hoisting local variables from the tile processing loop. The combination of excessive register pressure and aggressive common subexpression elimination (CSE) can lead to expensive spills to off-chip memory.

As a compiler optimization, CSE is particularly advantageous for traditional CPU architecture where registers can be spilled to and recovered from nearby, high speed L1 cache. However, it is often more advantageous for GPU threads to simply recompute a result than to let it spill. An interesting area of research for future investigation would be the tighter integration of CSE with register allocation, perhaps along with analytical cost models and program analysis of how many active threads will be affected.

As another avenue of future work, CTA serialization is sufficiently simple to be automated by the compiler as an optimization step. The compiler simply needs to wrap the data-parallel kernel code within a while-loop, sequentially virtualizing the concurrent CTAs expressed by the programmer.

### 7.2.2 *Static metaprogramming*

The static metaprogramming techniques we describe in this dissertation are useful for achieving good performance on an abstract machine model where the overheads of runtime decision-making are substantially magnified by parallelism. However, metaprogramming under the current model of compilation has two drawbacks. Both are related to the fact that this extra programmer-supplied detail (e.g., the relationships between unrolling steps, tile sizes, and GPU architectures) is lost after the high-level source is compiled down into an intermediate representation.

First, library developers of GPU primitives cannot possibly hope to distribute code in the form of precompiled binaries. The number of specializations that would arise from simply compiling the cross-product of numeric data types across today’s existing architectures would result in untenable library bloating. As an example of binary distribution bloat, the CUDPP library redistributable is 294MB [35].

Instead, library providers must distribute high-level sources that can be `#included`, similar to the C++ standard template library. This may place an unwanted burden on library authors who may not want to share the details of their high-level source code, or on developers who may not want to contend with lengthier compile times.

Second, the metaprogramming approach is subject to performance regression by omission. Although code specialization by data type can be driven by well-defined sets of traits (e.g., representation size, signed/unsigned, etc.), specialization by architecture-version requires the compiler to be aware of all potential target processor configurations at compile-time. New architectures are currently released around every 18 months.

A better solution might entail a mechanism for the programming model to retain metaprogramming directives within the intermediate representation, allowing just-in-time

compilation by the loader/driver to specialize and unroll executable code for the specific target processor it has been deployed with.

### **7.2.3 *Sorting***

This dissertation presents very efficient radix sorting. However, comparison-based methods are required for sorting problems where a lexicographic ordering of keys does not exist. They are also more desirable when the key data type is sufficiently long and/or the input size of the sorting problem is relatively small.

A prospective avenue of future research would be to investigate the design of comparison-based, top-down partitioning strategies, e.g., multi-pivot quicksort. Our concurrent allocation strategies based upon parallel prefix sum should be directly applicable to the problem of constructing and tracking the dynamic, recursive partitioning of sorting inputs.

### **7.2.4 *Graph traversal***

Our multi-GPU implementation of sparse BFS leverages duplicate removal to significantly cut down the number of vertex-identifiers transmitted between GPUs. An interesting extension of this technique would be to push such duplicate removal into the hierarchical interconnect of large scale systems (such as those evaluated by the Graph500 benchmark [111]). Using a randomized overlay network having  $\log_2 p$  expected communication hops, one can imagine the entire system collectively acting as a progressive filter for eliminating the vast majority of edges before they ever arrive at their authoritative ownership nodes.

## REFERENCES

- [1] 10th DIMACS Implementation Challenge:  
*<http://www.cc.gatech.edu/dimacs10/index.shtml>*. Accessed: 2011-07-11.
- [2] Agarwal, V. et al. 2010. Scalable Graph Exploration on Multicore Processors. *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (New Orleans, LA, USA, Nov. 2010), 1-11.
- [3] Ajtai, M. et al. 1983. An  $O(n \log n)$  sorting network. *Proceedings of the fifteenth annual ACM symposium on Theory of computing* (New York, NY, USA, 1983), 1–9.
- [4] Alcantara, D.A. et al. 2009. Real-time parallel hashing on the GPU. *ACM SIGGRAPH Asia 2009 papers* (New York, NY, USA, 2009), 154:1–154:9.
- [5] Ansel, J. et al. 2009. PetaBricks: a language and compiler for algorithmic choice. *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2009), 38–49.
- [6] BackForty Computing: Fast and efficient software primitives for GPU computing:  
*<http://code.google.com/p/back40computing/>*. Accessed: 2011-08-25.
- [7] Bader, D.A. and Madduri, K. Designing Multithreaded Algorithms for Breadth-First Search and st-connectivity on the Cray MTA-2. *2006 International Conference on Parallel Processing (ICPP'06)* (Columbus, OH, USA), 523-530.
- [8] Bader, D.A. et al. On the Architectural Requirements for Efficient Execution of Graph Algorithms. *2005 International Conference on Parallel Processing (ICPP'05)* (Oslo, Norway), 547-556.
- [9] Bakhoda, A. et al. 2009. Analyzing CUDA workloads using a detailed GPU simulator. (Apr. 2009), 163-174.
- [10] Batcher, K.E. 1968. Sorting networks and their applications. *Proceedings of the April 30–May 2, 1968, spring joint computer conference* (New York, NY, USA, 1968), 307–314.
- [11] Bell, N. and Garland, M. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (New York, NY, USA, 2009), 18:1–18:11.
- [12] Bell, N. et al. 2011. *Exposing Fine-Grained Parallelism in Algebraic Multigrid Methods*. Technical Report #NVR-2011-002. NVIDIA Corporation.



- [13] Bergman, K. et al. 2008. *ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems* Peter Kogge, Editor & Study Lead.
- [14] Bienia, C. et al. 2008. The PARSEC benchmark suite: characterization and architectural implications. *Proceedings of the 17th international conference on Parallel architectures and compilation techniques* (New York, NY, USA, 2008), 72–81.
- [15] Billeter, M. et al. 2009. Efficient stream compaction on wide SIMD many-core architectures. *Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), 159–166.
- [16] Blelloch, G.E. 1990. *Prefix Sums and Their Applications*. Synthesis of Parallel Algorithms.
- [17] Blelloch, G.E. 1989. Scans as primitive parallel operations. *IEEE Transactions on Computers*. 38, 11 (Nov. 1989), 1526-1538.
- [18] Blelloch, G.E. et al. Solving linear recurrences with loop raking. 416-424.
- [19] Borodin, A. 1977. On Relating Time and Space to Size and Depth. *SIAM Journal on Computing*. 6, 4 (1977), 733-744.
- [20] Brent, R.P. and Kung, H.T. 1982. A Regular Layout for Parallel Adders. *Computers, IEEE Transactions on*. C-31, 3 (Mar. 1982), 260 -264.
- [21] Buck, I. et al. 2004. Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph.* 23, 3 (Aug. 2004), 777–786.
- [22] Case, D.A. et al. 2010. *Amber 11*. University of California.
- [23] Castaño, I. 2007. High Quality DXT Compression using CUDA. NVIDIA.
- [24] Cederman, D. and Tsigas, P. 2010. GPU-Quicksort: A practical Quicksort algorithm for graphics processors. *J. Exp. Algorithmics*. 14, (Jan. 2010), 4:1.4–4:1.24.
- [25] Chakrabarti, D. et al. 2004. R-MAT: A Recursive Model for Graph Mining. *SIAM International Conference on Data Mining* (2004).
- [26] Chatterjee, S. et al. 1990. Scan primitives for vector computers. *Proceedings of the 1990 ACM/IEEE conference on Supercomputing* (Los Alamitos, CA, USA, 1990), 666–675.
- [27] Che, S. et al. 2009. Rodinia: A benchmark suite for heterogeneous computing. *2009 IEEE International Symposium on Workload Characterization (IISWC)* (Austin, TX, USA, Oct. 2009), 44-54.

- [28] Che, S. et al. 2010. A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads. (Dec. 2010), 1-11.
- [29] Cheney, C.J. 1970. A nonrecursive list compacting algorithm. *Commun. ACM*. 13, (Nov. 1970), 677–678.
- [30] Chhugani, J. et al. 2008. Efficient implementation of sorting on multi-core SIMD CPU architecture. *Proc. VLDB Endow.* 1, (Aug. 2008), 1313–1324.
- [31] Cohen, J.M. et al. 2010. Interactive fluid-particle simulation using translating Eulerian grids. *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2010), 15–22.
- [32] Cong, G. et al. 2010. Fast PGAS Implementation of Distributed Graph Algorithms. *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (New Orleans, LA, USA, Nov. 2010), 1-11.
- [33] Cormen, T.H. et al. 2001. *Introduction to Algorithms*. MIT Press.
- [34] CUDA: [http://www.nvidia.com/object/cuda\\_home\\_new.html](http://www.nvidia.com/object/cuda_home_new.html). Accessed: 2011-08-25.
- [35] cudpp - CUDA Data Parallel Primitives Library - Google Project Hosting: <http://code.google.com/p/cudpp/>. Accessed: 2011-07-12.
- [36] Dagum, L. and Menon, R. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*. 5, (Mar. 1998), 46-55.
- [37] Daniel Horn 2005. Stream reduction operations for GPGPU applications. *GPU Gems 2 : Programming Techniques for High-Performance Graphics and General-purpose Computation*. Addison-Wesley. 573-589.
- [38] Dean, J. and Ghemawat, S. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM*. 51, 1 (Jan. 2008), 107–113.
- [39] Dehne, F. and Zaboli, H. 2010. Deterministic Sample Sort For GPUs. *CoRR*. abs/1002.4464, (2010).
- [40] Deng, Y. (Steve) et al. 2009. Taming irregular EDA applications on GPUs. *Proceedings of the 2009 International Conference on Computer-Aided Design* (New York, NY, USA, 2009), 539–546.
- [41] Devore, J. 1999. *Applied statistics for engineers and scientists*. Duxbury Press.
- [42] Dotsenko, Y. et al. 2008. Fast scan algorithms on graphics processors. *Proceedings of the 22nd annual international conference on Supercomputing* (New York, NY, USA, 2008), 205–213.

- [43] Dusseau, A.C. et al. 1996. Fast parallel sorting under LogP: experience with the CM-5. *Parallel and Distributed Systems, IEEE Transactions on*. 7, 8 (Aug. 1996), 791–805.
- [44] Dwork, C. et al. 1997. Contention in shared memory algorithms. *J. ACM*. 44, 6 (Nov. 1997), 779–805.
- [45] Fortune, S. and Wyllie, J. 1978. Parallelism in random access machines. *Proceedings of the tenth annual ACM symposium on Theory of computing* (New York, NY, USA, 1978), 114–118.
- [46] Garanzha, K. and Loop, C. 2010. Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing. *Computer Graphics Forum*. 29, 2 (Jun. 2010), 289–298.
- [47] Garlan, D. et al. 1995. Architectural mismatch: why reuse is so hard. *IEEE Software*. 12, 6 (Nov. 1995), 17–26.
- [48] Garland, M. 2008. Sparse matrix computations on manycore GPU's. *Proceedings of the 45th annual Design Automation Conference* (New York, NY, USA, 2008), 2–6.
- [49] Goldschlager, L.M. 1982. A universal interconnection pattern for parallel computers. *J. ACM*. 29, 4 (Oct. 1982), 1073–1086.
- [50] Gonzalez, J. et al. 2009. Residual Splash for Optimally Parallelizing Belief Propagation. *Journal of Machine Learning Research - Proceedings Track*. 5, (2009), 177–184.
- [51] Govindaraju, N. et al. 2006. GPUTeraSort: high performance graphics co-processor sorting for large database management. *Proceedings of the 2006 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2006), 325–336.
- [52] Govindaraju, N.K. et al. 2005. Fast and approximate stream mining of quantiles and frequencies using graphics processors. *Proceedings of the 2005 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2005), 611–622.
- [53] GTgraph: A suite of synthetic random graph generators:  
<https://sdm.lbl.gov/~kamesh/software/GTgraph/>. Accessed: 2011-07-11.
- [54] Halstead, Jr., R.H. 1985. MULTILISP: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst*. 7, 4 (Oct. 1985), 501–538.
- [55] Harish, P. and Narayanan, P.J. 2007. Accelerating large graph algorithms on the GPU using CUDA. *Proceedings of the 14th international conference on High performance computing* (Berlin, Heidelberg, 2007), 197–208.

- [56] He, B. et al. 2007. Efficient gather and scatter operations on graphics processors. *Proceedings of the 2007 ACM/IEEE conference on Supercomputing* (New York, NY, USA, 2007), 46:1–46:12.
- [57] He, B. et al. 2009. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.* 34, (Dec. 2009), 21:1–21:39.
- [58] He, B. et al. 2008. Relational joins on graphics processors. *Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2008), 511–524.
- [59] Hensley, J. et al. 2005. Fast Summed-Area Table Generation and its Applications. *Computer Graphics Forum.* 24, 3 (2005), 547–555.
- [60] High performance block-sorting data compression library:  
<https://github.com/IlyaGrebnev/libbsc>. Accessed: 2011-09-23.
- [61] Hillis, W.D. and Steele, G.L. 1986. Data parallel algorithms. *Communications of the ACM.* 29, 12 (Dec. 1986), 1170-1183.
- [62] HLSL (Windows): [http://msdn.microsoft.com/en-us/library/bb509561\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/bb509561(v=vs.85).aspx). Accessed: 2011-08-25.
- [63] Hong, S. et al. 2011. Accelerating CUDA graph algorithms at maximum warp. *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming* (New York, NY, USA, 2011), 267–276.
- [64] Hong, S. et al. 2011. Efficient Parallel Graph Exploration for Multi-Core CPU and GPU. (New York, NY, USA, 2011), to appear.
- [65] Hou, Q. et al. 2008. BSGP: bulk-synchronous GPU programming. *ACM SIGGRAPH 2008 papers* (New York, NY, USA, 2008), 19:1–19:12.
- [66] Hussein, M. et al. 2007. On Implementing Graph Cuts on CUDA. *First Workshop on General Purpose Processing on Graphics Processing Units* (Boston, MA, Oct. 2007).
- [67] IEEE Computer Society 2009. IEEE Standard VHDL Language Reference Manual. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*. (2009), c1 -626.
- [68] Kainz, B. et al. 2009. Ray casting of multiple volumetric datasets with polyhedral boundaries on manycore GPUs. *ACM SIGGRAPH Asia 2009 papers* (New York, NY, USA, 2009), 152:1–152:9.
- [69] Kerr, A. et al. 2009. A characterization and analysis of PTX kernels. (Oct. 2009), 3-12.

- [70] Kim, C. et al. 2010. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. *Proceedings of the 2010 international conference on Management of data* (New York, NY, USA, 2010), 339–350.
- [71] Kipfer, P. et al. 2004. UberFlow: a GPU-based particle engine. *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (New York, NY, USA, 2004), 115–122.
- [72] Knuth, D.E. 1998. *The Art of Computer Programming, Volume 3: (2nd ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc.
- [73] Koch, D. and Torresen, J. 2011. FPGASort: a high performance sorting architecture exploiting run-time reconfiguration on fpgas for large problem sorting. *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays* (New York, NY, USA, 2011), 45–54.
- [74] Kogge, P.M. and Stone, H.S. 1973. A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations. *IEEE Transactions on Computers*. C-22, 8 (Aug. 1973), 786-793.
- [75] Kun Zhou et al. 2011. Data-Parallel Octrees for Surface Reconstruction. *IEEE Transactions on Visualization and Computer Graphics*. 17, 5 (May. 2011), 669-681.
- [76] Lee, V.W. et al. 2010. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. *Proceedings of the 37th annual international symposium on Computer architecture* (New York, NY, USA, 2010), 451–460.
- [77] Leischner, N. et al. 2010. GPU sample sort. *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)* (Atlanta, GA, Apr. 2010), 1-10.
- [78] Leiserson, C.E. and Schardl, T.B. 2010. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures* (New York, NY, USA, 2010), 303–314.
- [79] Luo, L. et al. 2010. An effective GPU implementation of breadth-first search. *Proceedings of the 47th Design Automation Conference* (New York, NY, USA, 2010), 52–55.
- [80] Mark Harris et al. 2007. Parallel Prefix Sum (Scan) with CUDA. *GPU Gems 3*. Addison-Wesley. 573-589.
- [81] Mark, W.R. et al. 2003. Cg: a system for programming graphics hardware in a C-like language. *ACM Trans. Graph.* 22, 3 (Jul. 2003), 896–907.
- [82] McCool, M. 2004. *Metaprogramming GPUs with Sh*. A K Peters.

- [83] McGraw, J. et al. 1985. *SISAL: Streams and iteration in a single assignment language, language reference manual version 1.2*. Lawrence-Livermore-National-Laboratory.
- [84] Merrill, D. and Grimshaw, A. 2011. High Performance and Scalable Radix Sorting: A case study of implementing dynamic parallelism for GPU computing. *Parallel Processing Letters*. 21, 02 (2011), 245-272.
- [85] Merrill, D. and Grimshaw, A. 2009. *Parallel Scan for Stream Architectures*. Technical Report #Technical Report CS2009-14. Department of Computer Science, University of Virginia.
- [86] Message Passing Interface Forum 2009. *MPI: A Message-Passing Interface Standard Version 2.2*.
- [87] Newman, M. and Girvan, M. 2004. Finding and evaluating community structure in networks. *Physical Review E*. 69, 2 (Feb. 2004).
- [88] NVIDIA Optix Ray Tracing Engine: <http://developer.nvidia.com/optix>. Accessed: 2011-09-23.
- [89] OpenGL Overview: <http://www.opengl.org/about/overview/>. Accessed: 2011-08-25.
- [90] Optimizing parallel reduction in CUDA: 2007. [http://developer.download.nvidia.com/compute/cuda/1\\_1/Website/projects/reduction/doc/reduction.pdf](http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf). Accessed: 2009-12-12.
- [91] Owens, J.D. et al. 2008. GPU Computing. *Proceedings of the IEEE*. 96, 5 (May. 2008), 879-899.
- [92] Pantaleoni, J. and Luebke, D. 2010. HLBVH: hierarchical LBVH construction for real-time ray tracing of dynamic geometry. *Proceedings of the Conference on High Performance Graphics* (Aire-la-Ville, Switzerland, Switzerland, 2010), 87-95.
- [93] Parboil Benchmark suite: <http://impact.crhc.illinois.edu/parboil.php>. Accessed: 2011-07-11.
- [94] PCI-SIG 2010. *PCI Express Base 3.0 Specification*.
- [95] Rice, H.G. 1953. Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society*. 74, 2 (1953), pp. 358-366.
- [96] Rogers, H. 1987. *Theory of recursive functions and effective computability*. MIT Press.

- [97] Satish, N. et al. 2009. Designing efficient sorting algorithms for manycore GPUs. *2009 IEEE International Symposium on Parallel & Distributed Processing* (Rome, Italy, May. 2009), 1-10.
- [98] Satish, N. et al. 2010. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. *Proceedings of the 2010 international conference on Management of data* (New York, NY, USA, 2010), 351–362.
- [99] Satish, N. et al. 2010. *Fast Sort on CPUs, GPUs and Intel MIC Architectures*. Intel Labs.
- [100] Scarpazza, D.P. et al. 2008. Efficient Breadth-First Search on the Cell/BE Processor. *IEEE Transactions on Parallel and Distributed Systems*. 19, 10 (Oct. 2008), 1381-1395.
- [101] Sengupta, S. et al. 2008. *Efficient parallel scan algorithms for GPUs*. Technical Report #NVR-2008-003. NVIDIA.
- [102] Sengupta, S. et al. 2007. Scan primitives for GPU computing. *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware* (Aire-la-Ville, Switzerland, Switzerland, 2007), 97–106.
- [103] Shopf, J. et al. 2008. March of the Froblins: simulation and rendering massive crowds of intelligent and detailed creatures on GPU. *ACM SIGGRAPH 2008 classes* (New York, NY, USA, 2008), 52–101.
- [104] Sintorn, E. and Assarsson, U. 2008. Real-time approximate sorting for self shadowing and transparency in hair rendering. *Proceedings of the 2008 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2008), 157–162.
- [105] Sklansky, J. 1960. Conditional-Sum Addition Logic. *IEEE Transactions on Electronic Computers*. EC-9, 2 (Jun. 1960), 226-231.
- [106] Snir, M. 1986. Depth-size trade-offs for parallel prefix computation. *Journal of Algorithms*. 7, 2 (Jun. 1986), 185-201.
- [107] Snyder, L. 1986. Type architectures, shared memory, and the corollary of modest potential. *Annual Reviews Inc.* 289–317.
- [108] Sunderam, V.S. 1990. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*. 2, (Dec. 1990), 315-339.
- [109] Tang, P. and Yew, P.-C. 1990. Software combining algorithms for distributing hot-spot addressing. *Journal of Parallel and Distributed Computing*. 10, 2 (1990), 130 - 139.
- [110] Tarditi, D. et al. 2006. Accelerator: using data parallelism to program GPUs for general-purpose uses. *SIGOPS Oper. Syst. Rev.* 40, 5 (Oct. 2006), 325–335.

- [111] The Graph 500 List: <http://www.graph500.org/>. Accessed: 2011-07-11.
- [112] Thearling, K. and Smith, S. 1992. An improved supercomputer sorting benchmark. *Proceedings of the 1992 ACM/IEEE conference on Supercomputing* (Los Alamitos, CA, USA, 1992), 14–19.
- [113] Thrust - Code at the speed of light - Google Project Hosting: <http://code.google.com/p/thrust/>. Accessed: 2011-08-25.
- [114] Ullman, J. and Yannakakis, M. 1990. High-probability parallel transitive closure algorithms. *Proceedings of the second annual ACM symposium on Parallel algorithms and architectures - SPAA '90* (Island of Crete, Greece, 1990), 200-209.
- [115] University of Florida Sparse Matrix Collection: <http://www.cise.ufl.edu/research/sparse/matrices/>. Accessed: 2011-07-11.
- [116] Valiant, L.G. 1990. A bridging model for parallel computation. *Commun. ACM*. 33, 8 (Aug. 1990), 103–111.
- [117] Valiant, L.G. 1976. Universal circuits (Preliminary Report). *Proceedings of the eighth annual ACM symposium on Theory of computing* (New York, NY, USA, 1976), 196–203.
- [118] Xia, Y. and Prasanna, V.K. 2009. Topologically Adaptive Parallel Breadth-first Search on Multicore Processors. *21st International Conference on Parallel and Distributed Computing and Systems (PDCS'09)* (Nov. 2009).
- [119] Yang, Z. et al. 2008. Parallel Image Processing Based on CUDA. (2008), 198-201.
- [120] Yoo, A. et al. A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L. *ACM/IEEE SC 2005 Conference (SC'05)* (Seattle, WA, USA), 25-25.
- [121] Zagha, M. and Blelloch, G.E. 1991. Radix sort for vector multiprocessors. *Proceedings of the 1991 ACM/IEEE conference on Supercomputing* (New York, NY, USA, 1991), 712–721.
- [122] Zhang, E.Z. et al. 2011. On-the-fly elimination of dynamic irregularities for GPU computing. *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2011), 369–380.
- [123] Zhou, K. et al. 2009. RenderAnts: interactive Reyes rendering on GPUs. *ACM SIGGRAPH Asia 2009 papers* (New York, NY, USA, 2009), 155:1–155:11.
- [124] Zhou, K. et al. 2008. Real-time KD-tree construction on graphics hardware. *ACM SIGGRAPH Asia 2008 papers* (New York, NY, USA, 2008), 126:1–126:11.



## INDEX

activity factor .....	36
architectural mismatch .....	29
atomic read-modify-write operations.....	9, 33
autotuning .....	67
bank conflict.....	30, 56
barrier.....	26, 33
between-group variance .....	64
BFS	
concurrent discovery .....	151
neighbor-gathering .....	143
redundant expansion factor .....	153
status-lookup .....	148
boolean circuit parallel model.....	76
branch divergence .....	See SIMD divergence
bulk-synchronous parallel model (BSP) .....	26
coalescing.....	30, 112
combining .....	33
software combining.....	11
contention.....	32
cooperative thread array (CTA) .....	26
CTA serialization .....	50, 81, 106, 109
data-parallel.....	6
dependences	
allocation dependences .....	7
shared data dependences .....	7
distribution sort .....	102
divergence .....	See SIMD divergence
DRAM.....	25
fine-grained .....	See granularity
frontier (BFS).....	15, 135
global memory .....	25
granularity .....	24
granularity coarsening.....	39, 48
grid (of CTAs).....	26
host.....	25
input-oriented decomposition .....	8
kernel.....	23

kernel fusion.....	91, 108
lanes (SIMD).....	See SIMD lanes
level synchronous.....	134
loop raking .....	See raking
memory efficiency .....	36
memory wall .....	90, 111
meta-programming.....	39
output-oriented decomposition .....	6
prefix scan.....	76
Brent-Kung .....	76
deficiency .....	77
definition.....	75
depth-size optimal (DSO) .....	77
downsweep phase.....	79
exclusive scan .....	75
inclusive scan .....	75
prefix sum .....	3, 15, 76
sequential .....	76
Sklansky .....	76
upsweep phase .....	79
prefix scan global parallelizations	
reduce-then-scan .....	80
scan-then-propagate .....	79
two-level reduce-then-scan .....	81, 109
prefix scan local parallelizations	
Blelloch.....	82
reduced-conflict Brent Kung (RCBK) .....	84
sequential-reduce-then-scan (SRTS) .....	85
radix sort .....	102
composite scan .....	116
digit .....	102
digit-place .....	102
early-exit .....	117
multi-scan.....	107
raking .....	55
RMAT synthetic graphs.....	140
segmented scan .....	92
shared memory.....	25
SIMD divergence .....	32
SIMD lanes .....	23
simultaneous multithreading (SMT) .....	23
single-instruction, multiple-data (SIMD).....	23
single-program, multiple-data (SPMD) .....	23
sorting network .....	13, 101
split primitive .....	104
stencil kernel .....	6

iterative stencil pattern .....	12
strip mining .....	55
thread .....	23
thread serialization .....	53, 85
tile .....	49
tuning policy .....	57
warp .....	24
warp scan .....	86, 87
within-group variance .....	64
work complexity .....	76