UC Riverside
UC Riverside Electronic Theses and Dissertations

Title
High Performance Vertex-Centric Graph Analytics on GPUs

Permalink
https://escholarship.org/uc/item/26g8702n

Author
Khorasani, Farzad

Publication Date
2016

Copyright Information
This work is made available under the terms of a Creative Commons Attribution License, availalbe at https://creativecommons.org/licenses/by/4.0/

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA
RIVERSIDE

High Performance Vertex-Centric Graph Analytics on GPUs

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Farzad Khorasani

December 2016

Dissertation Committee:

Dr. Rajiv Gupta, Chairperson
Dr. Laxmi N. Bhuyan
Dr. Zizhong Chen
Dr. Nael Abu-Ghazaleh

Copyright by
Farzad Khorasani
2016

The Dissertation of Farzad Khorasani is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

This dissertation would not have been possible if it wasn't for the help of my advisor, my professors, friends and collaborators, and my family.

First, I would like to express my gratitude to my academic prophet, my advisor, Prof. Rajiv Gupta, for leading me through the hardships of this path. I have no doubt it was Dr. Gupta's trust in my abilities and his experience and guidance that made this dissertation possible. I consider myself a lucky person for the opportunity of working with Dr. Gupta. *Thanks Dr. Gupta!*

I would like to thank Dr. Bhuyan, for the collaboration and guidance during my research. I would like to thank my dissertation committee members Dr. Chen and Dr. Abu-Ghazaleh for their valuable feedback and support.

I would like to thank the members of our research group who were always willing to help me during these years: Kishore, Min, Changhui, Yan, Sai, Mehmet, Vineet, Amlan, Keval, Zack, Bo, and Bryan. I also extend my gratitude to all the teachers I have had throughout my life.

I would like to thank my family, my mother Soheila Bahadori, my father Parviz Khorasani, and my brother Roozbeh Khorasani for their help and support throughout my life. I would like to thank my wife Dänae G. Khorasani for her constant love and support.

Finally, I would like to acknowledge the support of National Science Foundation by providing grants CCF-0905509, CCF-0963996, CNS-1157377, CCF-1318103, CCF-1423108, and CCF-1524852 to UC Riverside.

To my parents and my brother for all the love and support,

and to my wife Dänae.

ABSTRACT OF THE DISSERTATION

High Performance Vertex-Centric Graph Analytics on GPUs

by

Farzad Khorasani

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, December 2016
Dr. Rajiv Gupta, Chairperson

Massive parallel processing power of GPU's presents an attractive opportunity for accelerating large scale vertex-centric graph computations. However, the inherent irregularity and large sizes of real-world power law graphs creates many challenges. Lock-step execution by threads within a SIMD group restricts exploitable parallelism, the limited GPU's DRAM size restricts the sizes of graphs that can be offloaded to the GPU, and the limited inter-GPU communication bandwidth necessitates judicious use available bandwidth. This dissertation addresses all of these challenges.

We present *Warp Segmentation* that greatly enhances GPU device utilization by dynamically assigning appropriate number of SIMD threads to process a vertex while employing the compact CSR representation to maximize the graph size that can be held in GPU global memory. Prior works can either maximize graph sizes (e.g., VWC [34]) or device utilization (e.g., CuSha [42]). We scale graph processing over multiple GPUs via *Vertex Refinement* that dynamically collects and transfers only the updated boundary vertices leading to dramatically reduced amount of inter-GPU data transfer. Existing multi-GPU techniques (Medusa [94], TOTEM [24]) perform high degree of wasteful vertex transfers.

Since processing all vertices at every iteration wastes much of GPU's computation power, we present a work-efficient solution that processes only those vertices during an iteration that were activated in the previous iteration. We employ an effective task expansion

strategy that avoids intra-warp thread underutilization. For multi-GPU graph computation, we present *permissive partitioning* to dynamically balance load across GPUs. Also, as recording vertex activeness requires additional data structures, to manage the graph storage overhead, we introduce *vertex grouping* that enables trade-off between memory consumption and work efficiency.

Finally, to apply the proposed solutions to other irregular applications, we generalize our techniques and present *Collaborative Context Collection* (CCC) and *Collaborative Task Engagement* (CTE). CCC is a software/compiler technique to enhance the SIMD-efficiency in loops containing thread divergence. CTE abstracts away the complexities of a rather complicated technique using a CUDA C++ device side template library and balances load across threads within a SIMD group.

Contents

List of Figures x

List of Tables xiii

1 Introduction 1
- 1.1 Dissertation Overview . . . 3
  - 1.1.1 Addressing the Warp Efficiency and Scalability Challenges . . . 4
  - 1.1.2 Enabling Work Efficiency for the Vertex-Centric Model . . . 5
  - 1.1.3 Extending Techniques to Other GPU Applications . . . 7
- 1.2 Dissertation Organization . . . 9

2 Warp Efficiency and Scalability 10
- 2.1 Warp Efficiency via Warp Segmentation . . . 13
  - 2.1.1 Motivating Study . . . 14
  - 2.1.2 Warp Segmentation . . . 16
- 2.2 Graph Processing Framework for WS . . . 19
  - 2.2.1 Core Processing Procedure . . . 19
  - 2.2.2 Framework Interface . . . 22
- 2.3 Scaling via Vertex Refinement . . . 23
  - 2.3.1 Inefficiency of Existing Inter-GPU Communication . . . 24
  - 2.3.2 Data Structure Organization . . . 24
  - 2.3.3 Vertex Refinement . . . 27
- 2.4 Performance Evaluation . . . 29
  - 2.4.1 Warp Segmentation Performance Analysis . . . 29
  - 2.4.2 Vertex Refinement Performance Analysis . . . 33
- 2.5 Summary . . . 38

3 Enabling Work-Efficiency 40
- 3.1 Data Structures for Work-Efficiency . . . 43
  - 3.1.1 Recording Vertex Activeness using Bitmasks . . . 45
  - 3.1.2 Vertex Grouping . . . 47
- 3.2 Warp Efficiency with Dynamic Thread Assignment . . . 49
- 3.3 Permissive Partitioning for Inter-GPU Load Balance . . . 52

3.4 KiTES and its Interface . . . 55
3.5 Evaluation of Techniques . . . 57
3.5.1 Single-GPU Performance Analysis . . . 57
3.5.2 Multi-GPU Performance Analysis . . . 62
3.6 Summary . . . 66

4 Generalizing Techniques **67**
4.1 Collaborative Context Collection . . . 68
4.1.1 CCC Core Procedure . . . 70
4.1.2 CCC Transformations . . . 76
4.1.3 CCC Optimizations . . . 83
4.1.4 CCC Implementation . . . 86
4.1.5 Experimental Evaluation . . . 89
4.2 Collaborative Task Engagement . . . 97
4.2.1 Motivation: Inefficiency of Static Task Decomposition Methods . . . 99
4.2.2 Collaborative Task Engagement . . . 104
4.2.3 Experimental Evaluation . . . 113
4.2.4 Sensitivity Analysis: varying coarse-grained task sizes . . . 117
4.3 Summary . . . 119

5 Related Work **120**
5.1 Graph Processing on GPUs . . . 120
5.2 SIMD Thread Divergence . . . 123

6 Conclusions and Future Work **128**
6.1 Contributions . . . 128
6.2 Future Directions . . . 130

Bibliography **132**

List of Figures

2.1 An example graph with 5 vertices and 8 edges and its CSR representation. . 13
2.2 Degree distribution for graph vertices. 14
2.3 An example: reduction in VWC with assumed warp size of 8 and first 6 neighbors belonging to one vertex and last 2 belonging to another. 16
2.4 Reduction using Warp Segmentation with the same scenario in Figure 2.3. . 17
2.5 Discovering segment size and the index within segment by warp lanes for the graph in Figure 2.1. Warp size is assumed 8. 17
2.6 Framework's graph processing procedure pseudo-algorithm. Assumed warp size is 32. Shared memory pointers in the program code are declared with `volatile` qualifier. 20
2.7 User-specified structures and functions for SSWP. 22
2.8 Organization of data structures in multi-GPU processing. 25
2.9 An example of online vertex refinement stages. 28
2.10 Profiled average warp execution efficiency of Warp Segmentation compared to VWC's. *SSSP* is the benchmark. 32
2.11 Processing-time break down into computation time and communication time for the Vertex Refinement (VR) compared to ALL and MS. Computation time is the total duration of kernel execution, and communication time is the total duration of inbox/outbox management plus inter-device memory copies. For each benchmark, the times are normalized with respect to the longest time. Note that this times cannot be used to infer the overall speedup due to asynchronicity of devices. 36
2.12 The scalability of our framework over graphs with different number of edges and densities for *SSSP* benchmark. All the graphs are Rmat created with parameters $a = 0.45$, $b = 0.25$, and $c = 0.15$. y axis is the processing time (lower is better). 39

3.1 The percentage of updated vertices and their connected edges across iterations. 41
3.2 An example directed graph, its CSR representation, and the main components of its CSC representation. 44
3.3 The frequency distribution of differences between source and destination indices of edges of 2 real-world graphs. 47

3.4 The effect of *vertex grouping* with ratio 2 on the size of the CSC representation of the example graph in Figure 3.2. 48
3.5 Iterative Vertex-Centric Graph Processing. 50
3.6 A simplified example demonstrating our dynamic thread assignment strategy. 51
3.7 The distribution of active edges on 3 GPUs across iterations in WS-VR [41]. 53
3.8 An example of *permissive partitioning* using the graph in Figure 3.2a and the resulting data structures for each device. 54
3.9 A sample use of KiTES to execute user-defined SSSP with 3 GPUs. 56
3.10 Per-iteration kernel execution time for KiTES and Warp Segmentation (WS) for PR and SSSP on LiveJournal. 59
3.11 Average warp execution efficiency profiled for KiTES and Warp Segmentation for SSSP and PR. 60
3.12 The effect of *vertex grouping* on the GPU's DRAM consumption and the performance of the procedure for two algorithms. The graph is *RMD25V200E*. 63
3.13 The speedup of KiTES over WS-VR in multi-GPU graph processing with and without *permissive partitioning*. 64
3.14 The effect of *permissive partitioning* on the distribution of active edges between 3 GPUs across iterations. *RMD41V536E* is the input graph. 65

4.1 An example: BFS graph processing visualization in CUDA [30]. 71
4.2 Applying Collaborative Context Collection to the program in Figure 4.1 eliminates warp execution inefficiency. 73
4.3 Applying CCC on the BFS CUDA kernel in Figure 4.1a. 74
4.4 A grid-stride loop applied to the BFS CUDA kernel in order to make it accessible by CCC. The maximum theoretical occupancy is assumed 100%. 77
4.5 An example demonstrating the transformation of a CUDA device function (BFS processing of a vertex's neighbors) with variable trip-count to a form accessible by CCC. 79
4.6 An example demonstrating the transformation of a recursive CUDA device function (*cuckoo hashing on GPU* [3]) by CCC. 80
4.7 Transformation of a loop with unknown trip-count (*Cuckoo Hashing on GPU* [3]) by CCC. 81
4.8 Variation generation CUDA device function in Fractal Flame [77] from Iterated Function System (IFS) class. 83
4.9 SSSP graph processing CUDA kernel from [30] containing a coalesced global memory access to the `costs` buffer in the divergent path. We preserve the coalescence in CCC by excluding the memory access from the divergent path. 84
4.10 12CCC Framework operates alongside NVCC. 86
4.11 A PTX sample code inside the repetitive section and the resulted graph from connecting definition and usage of virtual registers. 88
4.12 The kernel execution speedup provided by CCC. For benchmarks with iterative GPU kernel launches (BFS and SSSP) the speedup is measured based on the aggregation of kernels. 92

4.13 Warp execution efficiency comparison for kernels with and without CCC. For BFS and SSSP the warp execution efficiency is averaged across all the kernel launches. 92
4.14 CCC performance enhancement compared to the original divergent kernel over different amount of intra-warp divergence (and hence workload imbalance). The divergent path contains 20 FMAD operations. 96
4.15 Sensitivity of CCC against different execution paths lengths plotted for two different amounts of intra-warp divergence. 97
4.16 An example — Sparse Matrix-Vector Multiplication (SpMV) CUDA kernel with a CSR matrix using 1D decomposition. Intra-warp load imbalance induces warp inefficiency and performance loss. 100
4.17 Warp execution visualization in sub-warp decomposition (with width 4) for the example in Figure 4.16. Sub-warp decomposition attempts to exploit parallelism inside coarse-grained tasks. 102
4.18 Visualization of the SpMV CUDA kernel in Figure 4.16a after applying CTE. 105
4.19 GPU pseudo-code for CCC. 107
4.20 Expressing the nested pattern in Fig. 4.16a CUDA C++ kernel in CTE form using our template library interface. 110
4.21 The kernel execution speedup of CTE and sub-warp decomposition over 1D decomposition for matrix operations on real-world matrices. 114
4.22 Profiled warp execution efficiency of CTE, sub-warp decomposition, and 1D decomposition kernels for experiments in Figure 4.21. 115
4.23 The kernel execution speedup of CTE and sub-warp decomposition over 1D decomposition for Fast Multiple Method [43] and Dynamical Quadrature Grids [54] with different inputs. 116
4.24 The kernel execution speedup of CTE and sub-warp decomposition over 1D decomposition for different graph applications and inputs. 117
4.25 Kernel execution duration (left plot) and Warp execution efficiency (right plot) for decomposition methods when the task sizes vary linearly and quadratically proportional to the lane index. Map and reduce portion of the fine-grained tasks each contain 20 FMAD instructions. For the LINE scenario, the coarse-grained task size is calculated with $4 \times laneID$ while for the QUAD scenario it is calculated with $\frac{laneID^2}{8}$. Task sizes for the sub-warp decomposition are calculated using their sub-warp index. 118

List of Tables

2.1 The percentage of useful vertex data among all the transferred data when all the vertices (ALL) or the maximal subset of them (MS) are copied from one GPU to another. In this two-GPU configuration, the graph under the examination is an Rmat graph with approximately 40 million vertices and 470 million edges. 25
2.2 Graphs used in single-GPU experiments – across benchmarks the size ranges in MBytes for CSR and CW representations. Sizes exceeding GPU's global memory capacity are bolded. 30
2.3 Raw running times (ms) of Warp Segmentation (WS) and VWC including kernel executions and host-device data transfers for different algorithms and different graphs. 31
2.4 Speedup ranges of Warp Segmentation over VWC excluding data transfer times. Since both methods use CSR representation, their data transfer times are equal. 32
2.5 The speedup of Warp Segmentation over CuSha's [42] CW for *large* graphs. The shards reside inside the host pinned buffers (x means graph is small - fits in GPU memory). 33
2.6 The speedup of Warp Segmentation over CuSha's [42] CW for *small* graphs. The shards reside inside the GPU's global memory (x means graph is large - requires host memory). 33
2.7 Graphs for multi-GPU experiments: Top 6 graphs used in experiments with 3 GPUs; rest used with 2 GPUs. 34
2.8 The speed-up of VR over ALL and MS for three-GPU and two-GPU configurations. 35
2.9 The speedup of our framework when scaling to more GPUs: From 2 to 3 GPUs for the top 6 graphs; and From 2 to 3 and from 1 to 2 GPUs for the rest of the graphs. 38

3.1 The memory required for CSR representation of four directed graphs [50] and their additional CSC representation. 47
3.2 Graphs for single-GPU evaluations and their representation sizes (in MB). For undirected graphs, CSC size is zero. 57

3.3 Raw execution time (ms) of KiTES (KT) in comparison with Warp Segmentation's (WS) [41] and Concatenated Windows's (CW) from CuSha [42] when all the graphs reside inside the GPU's global memory. 58
3.4 Execution times (ms) of KiTES (KT), Warp Segmentation (WS) [41] and Concatenated Windows (CW) [42] including host to device copy time. While WS and KiTES can fit the graph inside the GPU, CuSha must hold graphs in host pinned memory. 61
3.5 Execution time (ms) of KiTES (KT), Warp Segmentation (WS) [41] and Concatenated Windows (CW)[42] including host to device copy time. KiTES has to apply vertex grouping to fit the representation inside the GPU's DRAM. 62
3.6 Graphs used for multi-GPU experiments: Top 4 graphs employed for experiments with 3 GPUs and bottom 4 graphs for experiments with 2 GPUs. . . 63
3.7 Speedup provided by KiTES over WS-VR [41]. Top 4 entries use 3 GPUs and bottom 4 entries use 2 GPUs. 64

4.1 The CCC overhead in terms of resource usage (per thread). Underlined entry results from spilling two excessive registers into local memory (L1 cache) via `-maxrregcount` compiler option. The maximum theoretical occupancy is 100% in all cases. 94
4.2 Kernel warp execution efficiency of CUDA applications exposed to different inputs with 1D and sub-warp decomposition methods. The efficiency of kernels not only varies from one sub-warp width to another (the best in each row is underlined), it is also well below 100%. 103

Chapter 1

Introduction

Graph analytics have become one of the critical members in the suite of infrastructures dealing with big data processing. The need for efficient large scale graph processing has grown due to the importance of applications involving graph mining and graph analytics. Iterative vertex-centric processing model is one of the most popular and a methodical yet straightforward approach to representing graph algorithms. It has been incorporated in software systems such as PowerGraph [27], GraphLab [53], GraphChi [47] and others. The designer only needs to focus on the interaction of a vertex with its neighbors during an iteration. The underlying system orchestrates the execution of the vertex-centric methods on graph vertices over multiple iterations until convergence.

The deployment of GPUs as general purpose accelerators that started a decade ago has now become mainstream. Today GPUs serve as an essential processing platform for speeding-up data parallel computations. GPUs contain thousands of execution units and sufficient memory bandwidth that makes them well-suited for graph applications requiring massive parallelism. However, using GPUs for efficient graph processing has remained a challenging open problem. Even though GPUs provide a massive amount of parallelism with the potential to outperform general purpose CPUs, the underlying architecture imposes restrictions and introduces challenges in processing irregular real-world power law graphs. Thus, the challenges this thesis seeks to address are as follows.

– **Lock-step traversal of SIMD threads** restricts exploitable parallelism on GPUs. The SIMD architecture demands repetitive processing patterns on regular data which is contrary to the irregular nature of real-world graphs. All the threads inside a SIMD group (i.e., all the warp lanes) execute the same instruction at any given time. The presence of conditionals – such as if-else code blocks – causes thread divergence because a conditional may evaluate to true for some warp lanes and false for other lanes. In this situation, the warp takes all the divergent paths, while disabling non-relevant lanes inside every path. That is, the warp scheduler fetches instructions for all the divergent paths while the execution stage is performed only for a number of threads that are active in the path. As a result, a portion of the available processing power goes unutilized for the duration of divergence, diminishing the SIMD execution benefits. On the other hand, real-world graphs usually exhibit an irregular degree distribution known as power-law in which a great portion of the vertices have a few neighbors and a small portion of the vertices have a very high number of neighbors. Previously introduced task decomposition schemes assign one [30] or a fixed number [34] of threads to process vertices; thus, the mismatch between static decomposition and unpredictable irregularity of the computation in graph processing leads to the problem of underutilization of GPUs and thus limits performance.

– **Limited amount of available DRAM** presents another challenge in processing of large graphs. High performance GPUs come in the form of discrete GPUs and are equipped with high bandwidth GDDR5 or HBM off-chip memories. However, available memories are fixed, limiting the maximum size of the graph that can be kept at the device side and efficiently processed by the GPU. Therefore, the applicability of solutions with high storage overhead such as CuSha [42] is limited. In addition, although there has been research on utilizing host memory as auxiliary storage [78] to hold a larger graph, the resulting unavoidable performance drop is large due to the comparatively low communication bandwidth between the host and the GPU device.

– **Comparatively low inter-GPU bandwidth** makes processing very large graphs over multiple GPUs a challenge. A natural approach for processing very large graphs that do not fit inside one GPU is to partition them and process the partitions using multiple GPUs. However, during this out-scaled processing of the graph computation over multiple GPUs, the devices have to communicate via PCIe links which have low transfer rates compared to the rate at which GPUs access their own DRAM. Addressing this challenge requires a carefully-designed inter-device communication scheme. However, multi-GPU graph processing frameworks such as TOTEM [24] and Medusa [94] suffer from redundant inter-GPU communication data which makes the PCIe links a major bottleneck that limits achievable performance.

In this thesis, we address above challenges in high performance vertex-centric graph analytics on GPUs. This dissertation develops a dynamic task decomposition scheme to overcome the SIMD efficiency problem in irregular graph processing, while maximizing the allowable graph size by employing space-efficient CSR representations. It further extends the techniques to eliminate computation redundancy, and generalizes them via template libraries and compiler techniques to be deployed easily in other GPU applications that exhibit similar issues.

1.1 Dissertation Overview

The vast computing power of GPUs makes them an attractive platform for accelerating large scale data parallel computations such as popular graph processing applications. However, the inherent irregularity and large sizes of real-world power law graphs makes effective use of GPUs a major challenge. In this dissertation, we develop techniques based on CUDA platform that greatly enhance the performance and scalability of iterative vertex-centric graph processing on GPUs. In the vertex-centric model of graph computation, vertices update their value at every iteration using the vertex values seen at the other end of an edge. The computed values must be reduced to get the new vertex content. When

there is no change to the content of vertices at an iteration, the computation has converged, and the procedure terminates.

1.1.1 Addressing the Warp Efficiency and Scalability Challenges

We introduce techniques for efficient scaling of iterative graph algorithms to larger graphs using multiple GPUs. The graphs are stored in the space-saving CSR form that allows processing large graphs. To overcome the SIMD execution inefficiency in existing CSR-based graph processing methods, we present *Warp Segmentation*, a novel technique that assigns appropriate number of warp threads to process vertices with irregular number of neighbors on-the-fly. To scale the graph processing over multiple GPUs, we introduce *Vertex Refinement* that collects and transfers only those vertices that are boundary and recently updated. *Vertex Refinement* maximizes the inter-device bandwidth utilization efficiency.

Warp Segmentation for Efficient Warp Execution

Real-world graphs often exhibit power-law degree distribution, i.e. the number of neighbors a vertex owns vary greatly from one vertex to another. This makes existing methods such as Virtual Warp Centric (VWC) [34] that statically assign a fixed number of threads to vertices vulnerable to the intra-SIMD underutilization. We present *Warp Segmentation*, a novel method that greatly improves intra-warp utilization by dynamically assigning appropriate number of SIMD threads to process vertices with irregular-sized neighbors while employing compact CSR representation to maximize the graph size that can be kept inside the GPU global memory. This is in contrast to prior vertex-centric methods such as CuSha [42] that use G-Shards and CW representations requiring up to 2.5x more memory than CSR in order to boost the SIMD efficiency. Warp Segmentation assigns a warp to process a group of vertices and let threads iterate over the expanded list of neighbors. When the set of neighbors for the vertices is viewed as expanded, each thread can visit a neighbor, and then reduce the computed value with other threads inside the SIMD group that have processed a neighbor belonging to the same vertex. Therefore, at every iteration

all the threads execute the compute function without underutilization, and participate in a parallel reduction with appropriate threads. As a result, warp utilization is increased and the processing time is reduced. Warp Segmentation delivers average speedups of 1.29x to 2.80x over VWC.

Vertex Refinement for Efficient Inter-GPU Communication

We further scale graph processing to make use of multiple GPUs while proposing *Vertex Refinement* to address the challenge of judiciously using the limited bandwidth available for transferring data between GPUs over the PCIe bus. Whereas existing multi-GPU techniques (Medusa [94] and TOTEM [24]) perform high degree of wasteful vertex transfers, *Vertex Refinement* picks out, packs, and transfers only the updated boundary vertices thus dramatically reducing inter-GPU data transfer volume. *Vertex Refinement* essentially works as a fused stream compaction. To perform it inside the same GPU kernel by warp threads efficiently, at the end of the computation, threads get assigned to vertices. Threads check vertices for updates, each producing a predicate. Using techniques such as intra-warp binary reduction and prefix sum, intra-warp data propagation using shuffle, and warp-aggregated atomics, threads with true predicate effectively write the updated vertex index and content into the outbox buffer. Our design achieves up to 2.71x performance improvement compared to inter-GPU vertex communication schemes used by other multi-GPU techniques (i.e., Medusa and TOTEM).

1.1.2 Enabling Work Efficiency for the Vertex-Centric Model

The above vertex-centric solution lacks work-efficiency because, at the expense of being generic, it processes all vertices at every iteration. As a result GPU's SIMD power is wasted on processing inactive vertices that do not result in any change in vertex values. We remedy this issue by enabling *work-efficiency* when processing the graph on one or more GPUs. Our solution processes only those vertices that are activated in the previous iteration and hence their values are subject to change. Our experiments show that enabling work

efficiency enhances the performance of the procedure by up to 5.46x and 1.67x for single and multi-GPU configurations respectively over multiple algorithms and inputs. Below are the techniques used to overcome the challenges induced as the side effects of enabling work efficiency.

Vertex Grouping

Enabling work efficiency for the vertex-centric model needs a mechanism to keep track of the propagation of updates which necessitates storing outgoing neighbors for the vertices of the graph alongside its incoming neighbors. In directed graphs, we utilize the CSC representation of the graph and store it alongside the CSR representation. However, this increases space required by 1.5x to 1.98x and thus limits the maximum allowable size of the graph that can be kept inside the GPU. To attenuate the impact of graph storage overhead on limited GPU DRAM, we introduce *vertex grouping* as a technique that enables trade-off between memory consumption and work efficiency in our solution. *Vertex grouping* groups consecutive vertices in the CSC representation and represents them as a single entity, therefore, if there are multiple edges between vertices in two groups, they are represented by only one edge between entities. This results in reduced space consumption but also lower work efficiency since activation of any one vertex in the group leads to processing of all the vertices in the group during the next iteration.

Warp Efficiency with Dynamic Thread Assignment

In Warp Segmentation, the list of accessed neighbors for the set of vertices assigned to the warp are placed consecutively in the memory. However, by enabling work efficiency, only the neighbors assigned to active vertices need to be visited requiring the warp to access disjoint locations inside the memory. For a SIMD-efficient kernel operation we require gathering of active neighbors and performing reduction on them. This is achieved by an effective task expansion strategy that avoids intra-warp thread underutilization. Threads iterate over a concatenated view of the neighbors for active vertices and utilize high per-

formance SIMD primitives such as intra-warp binary reduction and binary prefix sum to realize the actual neighbor locations and indices. Our experiments show that this method sustains a high warp execution efficiency and is 82.4% on average.

Permissive Partitioning for Inter-GPU Load Balancing

Without work efficiency, since all the vertices are processed at every iteration, static partitioning of the graph is enough to balance the load across devices. However, when work efficiency is enabled, the amount of load for each GPU dynamically changes with each iteration. In a given iteration, one GPU may end up with a great number of active vertices and edges while another GPU may have much lower number of active graph components. This creates inter-device load imbalance in graph processing. To deal with this problem, for multi-GPU graph computations, we present *permissive partitioning* to achieve a maximally balanced load across GPUs. *Permissive partitioning* allows partitions stored on GPUs to overlap as much as the GPUs' available DRAM allows. During the iterative graph computation, while GPUs are busy processing the graph, the host asynchronously calculates the approximate borders for the set of vertices that will lead to the best load balance across devices. Accordingly, the sets of vertices assigned to the GPUs change in the next iteration. This scheme enhances the overall multi-GPU graph processing performance by up to 20%.

1.1.3 Extending Techniques to Other GPU Applications

GPU's SIMD architecture is a double-edged sword confronting parallel tasks with control flow divergence. On the one hand, it provides a high performance yet power-efficient platform to accelerate applications via massive parallelism; however, on the other hand, irregularities induce inefficiencies due to the warp's lockstep traversal of all diverging execution paths. This is true not only for graph applications, as we described earlier, but also for other programs that exhibit characteristics such as thread divergence or irregular load distribution. Sparse matrix vector multiplication (SpMV), parallel hashing, and ray

tracing are a few examples of such programs. We extend two SIMD efficiency enhancement techniques and make them applicable to GPU kernels beyond graph computation.

Collaborative Context Collection

We present a software (compiler) technique named *Collaborative Context Collection* (*CCC*) that increases the warp execution efficiency when faced with thread divergence incurred either by different intra-warp task assignment or by intra-warp load imbalance. CCC collects the relevant registers of divergent threads in a warp-specific stack allocated in fast shared memory, and restores them only when the perfect utilization of warp lanes becomes feasible. We propose code transformations to enable applicability of CCC to program segments such as recursive functions and loops with variable trip-count. We also introduce optimizations to reduce the cost of CCC and to avoid device occupancy limitation or memory divergence. We experiment with CCC on real-world applications, and analyze it under multiple scenarios using synthetic programs. CCC improves the warp execution efficiency of real-world benchmarks by up to 56% and achieves up to 3.08x speedup compared to the original programs.

Collaborative Task Engagement

Nested patterns are one of the most frequently occurring algorithmic themes in GPU applications where coarse-grained tasks are constituted from a number of fine-grained ones. However, efficient execution of irregular nested patterns, with coarse-grained tasks that substantially vary in size, has remained an open problem for the GPUs SIMT architecture. Existing methods, similar to what we observe in graph computation domain, rely on static task decomposition where one or a fixed number of threads inside the SIMD grouping (warp) carry out the fine-grained tasks. These approaches fail to provide portable performance across diversity of irregular inputs. Moreover, due to intra-warp load imbalance, they incur warp underutilization. We generalize our dynamic decomposition scheme for graph processing and introduce it as a software technique called *Collaborative Task En-*

gagement (*CTE*) that, unlike previous methods, achieves sustained high warp execution efficiencies across irregular inputs and provides portable performance. CTE assigns a group of coarse-grained tasks to the warp and allows threads inside the warp to carry out the expanded list of fine-grained tasks collaboratively. In multiple rounds, all the warp threads perform mapping portion of fine-grained tasks and participate in a reduction phase with appropriate lanes to reduce calculated values. This strategy avoids over-subscription or under-subscription of threads while preserving the benefits of parallel reduction. We prepared a CUDA C++ device-side template library for developers to easily express nested patterns in GPU kernels using our technique. Our experiments show that CTE delivers up to 37% warp execution efficiency improvement and gives up to 1.51x speedup over sub-warp decomposition with the best sub-warp width.

1.2 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 presents *Warp Segmentation* for SIMD-efficiency and *Vertex Refinement* for effective multi-GPU scalability of vertex-centric graph processing. Chapter 3 gives our solution to enable work-efficiency and offers a dynamic task assignment scheme. Chapter 3 also presents *Vertex Grouping* and *Permissive Partitioning* to overcome challenges introduced as side-effects of supporting work-efficiency. Chapter 4 generalizes two of our techniques and extends them as *Collaborative Context Collection* and *Collaborative Task Engagement* in forms of compiler optimization and template library respectively. Chapter 5 discusses the related work and Chapter 6 concludes the thesis by giving a summary of our work as well as directions for future work.

Chapter 2

Warp Efficiency and Scalability

Due to their ability to represent relationships between entities, graphs have become the building blocks of many high performance data analysis algorithms. A wide variety of graph algorithms are iterative in nature – during each iteration vertices update their state based upon states of neighbors connected by edges using a computation procedure until the graph state becomes stable. The inherent data parallelism in an iterative graph algorithm makes many-core processors, with underlying SIMD hardware such as GPUs, an attractive platform for accelerating the algorithms. However, efficient mapping of real-world power law graphs with irregularities to symmetric GPU architecture is a challenging task [55].

This chapter presents techniques that enhance the scalability and performance of vertex-centric graph processing on multi-GPU systems. This is achieved by fully exploiting the resources available on a multi-GPU system as follows:

- The irregular nature of power law graphs makes it difficult to balance load across threads leading to underutilization of SIMD resources. We address the device underutilization problem by developing *Warp Segmentation* that dynamically assigns appropriate number of SIMD threads to process a vertex with irregular-sized neighbors. Our experiments show that the warp execution efficiency of warp segmentation exceeds 70% while for the well known VWC [34] technique it is around 40%.

- For scaling performance to large graphs, they must be held in the global memories of GPUs in the system. To maximize the graph sizes that can be held in global memories, a compact graph representation must be used. Therefore *Warp Segmentation* makes use of the compact CSR representation. To tolerate the long latency of non-coalesced memory accesses that arise while accessing the neighbors of a vertex in CSR, warp segmentation keeps the GPU cores busy by scheduling other useful operations that compute the segment size and lane's intra-segment index.

- Since large graphs must be distributed across the global memories of multiple GPUs, processing at each GPU requires values of neighboring vertices that reside on other GPUs. Here we must make judicious use of the limited bandwidth available for transferring data between GPUs via the PCIe bus. We introduce an approach based upon parallel binary prefix sum that dynamically limits the inter-GPU transfers to only include updated vertices. In contrast, existing multi-GPU techniques perform high degree of wasteful vertex value transfers.

Our solution maximizes the graph sizes for which high performance can be achieved by fully utilizing GPU resources of SIMD hardware, memory, and bandwidth.

Let us briefly consider the related works and see how our approach overcomes their drawbacks. First, we consider the prominent single GPU techniques for vertex-centric graph processing, namely VWC [34] and CuSha [42]. Virtual-Warp Centric (VWC) [34] is the state-of-the-art method that uses the compact CSR representation and is inevitably prone to load imbalance when processing real-world graphs due to high variation in degrees of vertices. When the size of the virtual warp is less than the number of neighbors for a vertex, the virtual warp needs to iterate over the neighbors forcing other virtual warps within the warp that are assigned to vertices with fewer neighbors to stay inactive. When the size of the virtual warp is greater than the the size of the neighbors for a vertex, a great portion of the virtual warp is disabled. Both cases lead to underutilization of SIMD resources and poor warp execution efficiency. In addition, discovering the best virtual warp size for every

graph and every expressed graph algorithm requires multiple tries. CuSha [42] addresses the drawbacks of VWC, namely warp execution inefficiencies and non-coalesced accesses, but at the cost of using G-Shards and CW graph representations which are 2x-2.5x larger than the CSR representation due to vertex replication. *In contrast, Warp Segmentation uses the compact CSR representation while delivering high SIMD hardware utilization.* In warp segmentation the neighbors of warp-assigned vertices are grouped into segments. Warp lanes then get assigned to these neighbors and recognize their position inside the segment and the segment size by first performing a quick binary search on the fast shared memory content and then comparing their edge index with corresponding neighbor indices. When the segment size and the position in the segment are known for the lanes, user-defined reduction can be efficiently performed between neighbors of a vertex without introducing any intra-warp load imbalance. It will be shown in experiments that WS outperforms VWC by 1.29x−2.80x on average.

Next let us consider the related works on multi-GPU graph processing [94, 24]. Given a partitioning of a graph across multiple GPUs, these techniques underestimate the importance of efficient inter-device communication and do not effectively utilize the PCIe bandwidth. This is a significant problem because, the PCIe bus, as the path to communicate data from one GPU to another GPU, is tens of times slower than GPU global memory. Previous multi-GPU techniques either copy the whole vertex set belonging to one GPU to other GPUs at every iteration [94], or they identify boundary vertices in a pre-processing stage and make GPUs exchange these subsets of vertices in every iteration [24, 25]. In both approaches, a great number of vertices that are exchanged between devices is redundant. *In contrast, we propose Vertex Refinement, a new strategy that enables our framework to efficiently scale to multiple GPUs.* Vertex Refinement refines and transfers only those vertices that are updated in the previous round and are needed by other devices. It consists of two stages: *online* and *offline*. In the offline stage, boundary vertices are recognized and marked during pre-processing. In the online stage, we exploit parallel binary prefix sum to refine updated vertices from not-updated ones on-the-fly. A vertex is transferred to another

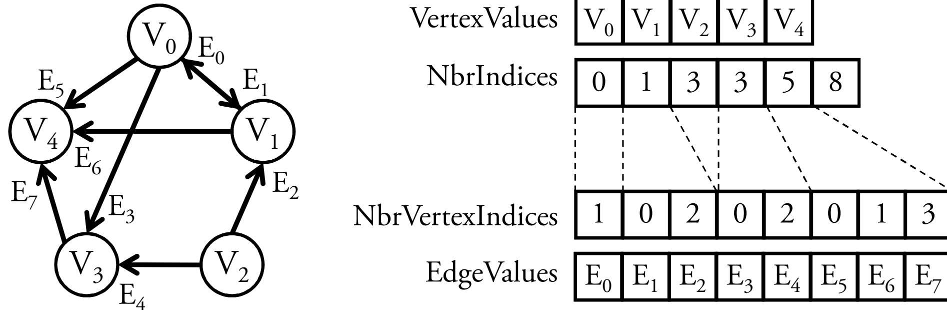


Figure 2.1: An example graph with 5 vertices and 8 edges and its CSR representation.

device only if it is marked and refined by the online stage. Thus, Vertex Refinement eliminates wasteful communication and provides higher multi-GPU performance and provides exclusive speedup of up to 2.71x over other multi-GPU vertex communication schemes.

The remainder of this chapter is organized as follows. We first present *Warp Segmentation* and the interface of the framework we created to express and execute the graph vertex-centric algorithms. Then we describe efficient scaling of our framework to multiple GPUs via Vertex Refinement. Finally, we evaluate the performance of our methods.

2.1 Warp Efficiency via Warp Segmentation

In this section we present *Warp Segmentation* (WS) that eliminates intra-warp load imbalance and enhances execution efficiency for processing a graph in CSR form. CSR is a compact form suitable for representing large and sparse graphs in a minimum space. Due to its space-efficiency, CSR is a good choice to hold large graphs inside the limited GPU memory.

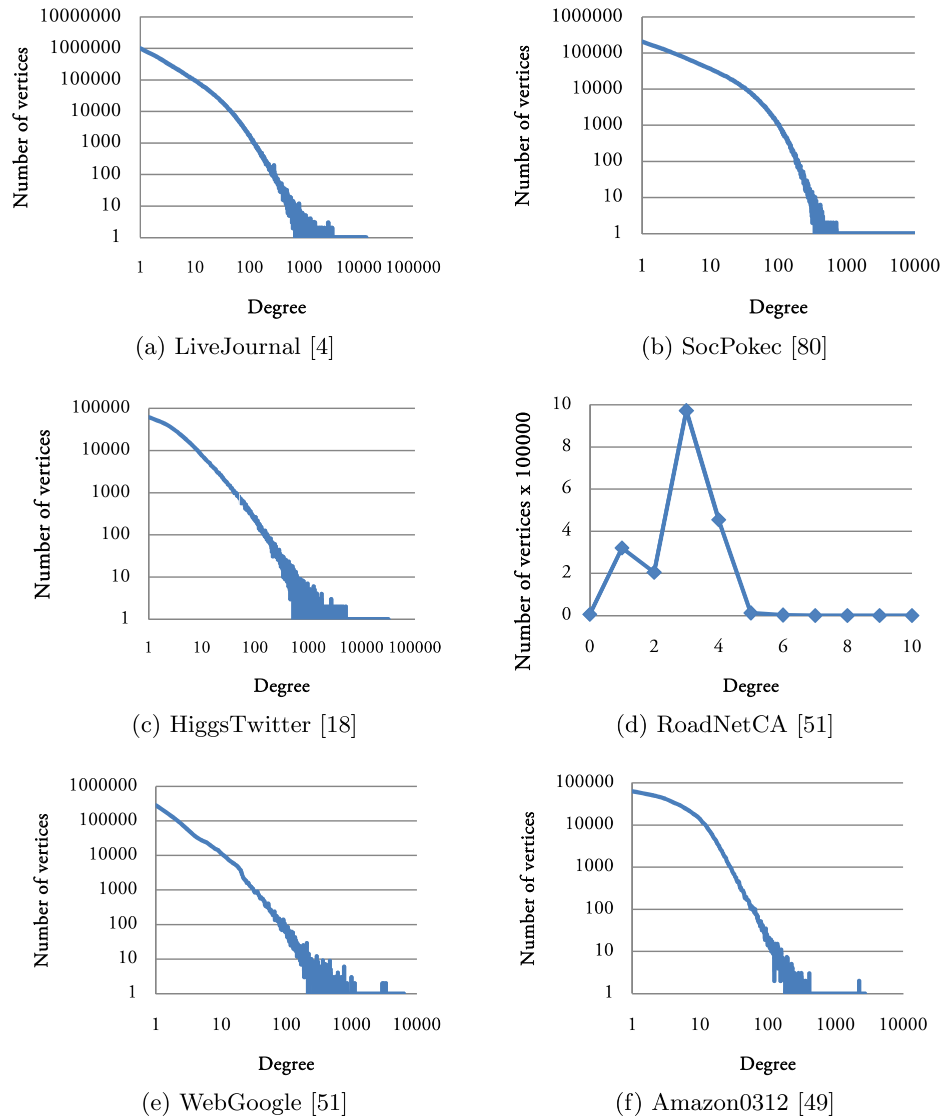
As Figure 2.1 shows, CSR consists of 4 arrays:

- *VertexValues* holds the content of the ith vertex in its ith element.
- *NbrVertexIndices* holds the indices for a vertex's neighbors in a contiguous fashion.
- *NbrIndices* holds a prefix sum of the number of neighbors for vertices. The ith

vertex's neighbors inside *NbrVertexIndices* start at $NbrIndices[i]$ and end before $NbrIndices[i+1]$.

- *EdgeValues* holds the edge values corresponding to the neighbors inside *NbrVertexIndices*.

2.1.1 Motivating Study



(a) LiveJournal [4]

(b) SocPokec [80]

(c) HiggsTwitter [18]

(d) RoadNetCA [51]

(e) WebGoogle [51]

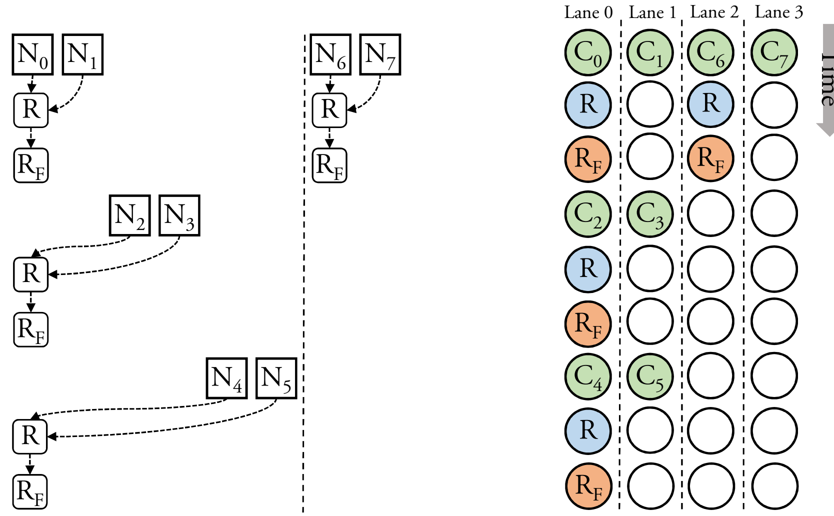
(f) Amazon0312 [49]

Figure 2.2: Degree distribution for graph vertices.

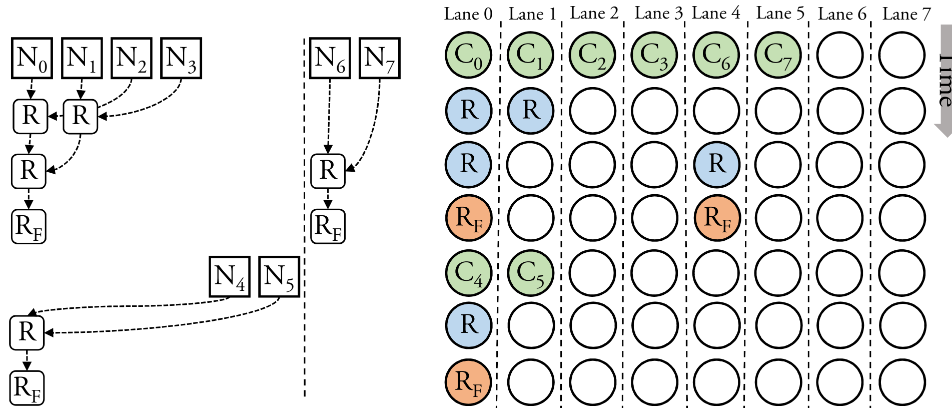
To motivate the need for WS, we first describe the drawbacks of the Virtual-Warp Centric (VWC) [34] method that also uses the CSR representation. VWC divides the SIMD group (warp, in CUDA terms) with the physical length of 32 into smaller virtual warps with fixed lengths (2, 4, 8, 16, or 32). Virtual warp size is kept the same throughout the graph processing. Each virtual warp is assigned to process one vertex and its incoming edges. As an enhancement of the original work [34], Khorasani et al. proposed a generalized form of VWC in [42] in which threads of the virtual warp are involved in reduction over the computed values. However, real-world graphs often exhibit power-law degree distribution, as it is evident in Figure 2.2, i.e. the number of neighbors a vertex owns vary greatly from one vertex to another. Thus, due to fixed number of virtual lanes involved in a reduction, VWC unavoidably suffers from underutilization:

- If the virtual warp is *smaller* than the vertex's number of neighbors, it will have to iterate over the vertex's connected edges hence dragging with it other virtual warps that have already finished their jobs (see the example in Figure 2.3a).
- If the virtual warp has a size that is *larger* than the number of neighbors for a vertex, a portion of virtual warp's lanes stays *idle* during the reduction leading to underutilization (see the example in Figure 2.3b).

This motivates the need for a technique that, independent of inner graph structure, takes minimum number of reduction steps in a SIMD environment, i.e. Warp Segmentation. Note that VWC suffer from the SIMD load imbalance in the same way PRAM-style thread assignment [30] does. In both PRAM-style and VWC, assigning fixed number of SIMD threads to process one vertex and its edges leads to thread-idling due to highly irregular vertex degree distribution. This fixed number in the former is exactly one while in the latter it can be a power of 2.



(a) VWC with Virtual Warp size 2.



(b) VWC with Virtual Warp size 4.

Figure 2.3: An example: reduction in VWC with assumed warp size of 8 and first 6 neighbors belonging to one vertex and last 2 belonging to another.

2.1.2 Warp Segmentation

To remedy the drawbacks of fixed-sized virtual warps, we propose *Warp Segmentation* (WS) technique. In WS, a warp is assigned to a group of 32 consecutive vertices and their connected edges. When warp lanes process edges iteratively, those that process edges belonging to one vertex—i.e. having the same destination index—form a *segment*. By knowing the segment size and the index inside the segment, lanes can participate in the appropriate reduction of segment, minimizing the total number of reduction steps.

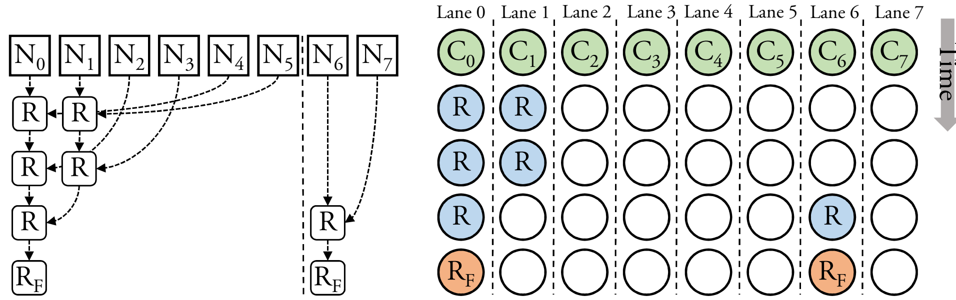


Figure 2.4: Reduction using Warp Segmentation with the same scenario in Figure 2.3.

Figure 2.4 shows the reduction in WS in an example scenario. In this example, first six lanes belong to one segment and two last lanes belong to another. The minimum number of reduction steps in this case is $\lceil \log_2 6 \rceil = 3$ which is also the case in WS. As Figure 2.3 shows, on-the-fly efficient reduction procedure in WS leads to better utilization of SIMD resources compared to VWC. In addition, WS does not need any pre-processing or trial-and-error for the best configuration determination.

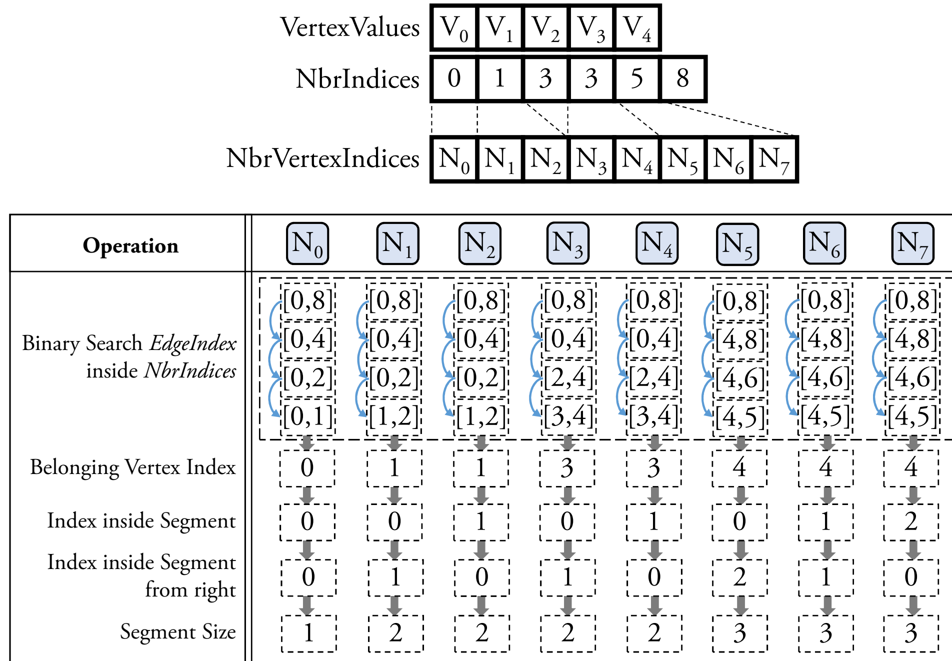


Figure 2.5: Discovering segment size and the index within segment by warp lanes for the graph in Figure 2.1. Warp size is assumed 8.

The key feature of WS is its fast determination of the segment a lane belongs to and the index of the lane within the segment. The step-by-step approach shown in Figure 2.5 illustrates this. Warp lanes perform a binary search over *NbrIndices* elements for their assigned edge index. Since *NbrIndices* elements are already fetched to the fast shared memory of the GPU, the binary search is performed quickly. After $\log_2(warpSize)$ steps, the starting position of the resulting search boundary shows the vertex index to which the edge belongs. Knowing the vertex index, the lane's index inside the segment and the segment size is retrieved using *NbrIndices* array. The distance of the holding edge index from the vertex's corresponding *NbrIndices* element reveals the position of the vertex in the segment. The difference between the holding edge index and the next vertex's corresponding *NbrIndices* element, minus one, yields the distance of the lane from the end of the segment. Addition of these two distances plus one represents the segment size.

WS is based upon the vertex-centric paradigm where in every iteration the shared memory serves as a scratchpad for vertices. The shared memory regions corresponding to vertices are: initialized by the vertex content within the global memory, modified depending upon the edges connected to the vertex using appropriate reductions, and at the end of the iteration, the updated values are pushed back to the global memory. Two alternatives for the intra-warp reduction in WS are possible. The first one is to use atomics to survive the concurrent modifications of the vertices as in [42]. However, this alternative imposes heavy use of atomics on shared memory locations on top of CSR's inherent non-coalesced neighbor accesses. The second alternative is processing a groups of vertices by one thread block. However, this approach necessitates multiple synchronization primitive across the thread block that degrade the performance. WS assigns a set of vertices to GPU's architectural SIMD grouping (warp) and performs efficient reductions hence *it avoids shared memory atomic operations alongside any explicit synchronizations throughout the kernel.*

The reduction in WS can be viewed as a form of intra-warp segmented reduction but without a *head flags* array, consisting of two main steps. First, warp lanes identify the vertex index via a fast binary search. Second, they discover the intra-segment index and the

segment size. Also, note that these two sets of operations are independent from the neighbor vertex value hence can be used to cover the latency of the inevitable non-coalesced access. The thread exploits instruction level parallelism by simultaneously executing non-dependent instructions. Thus, GPU cores are kept busy performing operations while neighbor's vertex value is on its way.

2.2 Graph Processing Framework for WS

Next we describe the framework that uses the graph processing procedure based on WS. Then, we present the interface functions that allow easy expression of graph algorithms by non-expert users.

2.2.1 Core Processing Procedure

Figure 2.6 shows the graph processing procedure. The convergence of iterative graph processing is controlled via a variable passed between the host and the device. If no thread updates this variable, it means the algorithm has converged and no more iterations are needed. In the outer-most `for` loop, according to the WS paradigm, each warp is assigned to process a contiguous set of vertices with the size equivalent to the warp size (32 for current CUDA devices). A warp task during one iteration is to process its assigned vertices. This task can be broken down into three major steps.

First step. In this step (lines 11 to 15 in Figure 2.6) threads of a warp fetch 32 elements of *VertexValues* and initialize the designated shared memory region for vertex values using user-provided initialization function. The threads also put $32+1$ corresponding elements of *NbrIndices* into another shared memory buffer. Using the *NbrIndices* starting and ending element, warp lanes can recognize the region within *EdgeValues* and *NbrVertexIndices* arrays that belongs to the assigned group of vertices.

```
0.  converged = false;
1.  while( !converged ) {
2.    converged = true;
3.    parallel-for warp w {
5.      __shared__ Vertex V[blockDim];
6.      __shared__ Vertex tLocal_V[blockDim];
7.      __shared__ uint NIdx[blockDim];
8.      w_V = V + warpOffsetWithinCTA;
9.      w_tLocal_V = tLocal_V + warpOffsetWithinCTA;
10.     w_NIdx = NIdx + warpOffsetWithinCTA;
        /* 1st major step */
11.     initVertex(w_V+laneID, VertexValues+globalTID);
12.     w_NIdx[laneID] = NbrIndices[globalTID];
13.     startEIdx = w_NIdx[0];
14.     endEIdx = NbrIndices[warpGlobalOffset+32];
        /* 2nd major step */
15.     for( currEIdx = startEIdx + laneID;
         currEIdx < endEIdx;
         currEIdx += 32 ) {
16.       nbrIdx = NbrVertexIndices[currEIdx];
17.       srcV = VertexValue[nbrIdx];
18.       belongingVIdx = binarySearch(currEIdx, w_NIdx);
19.       inSegID = min( laneID,
           currEIdx - w_NIdx[belongingVIdx] );
20.       SegSize= inSegID + 1 + min( 31 - laneID,
           ( (belongingVIdx==31)?endEIdx:
           w_NIdx[belongingVIdx+1] ) - currEIdx - 1 );
21.       ComputeNbr( srcV, EdgeValues+currEIdx,
           w_tLocal_V+laneID );
22.       reduceInsideSegment( w_tLocal_V+laneID,
           inSegID, SegSize );
23.       if( inSegID==0 )
24.         ReduceVertices( w_V+belongingVIdx,
             w_tLocal_V+laneID );
25.     }
        /* 3rd major step */
26.     if( IsUpdated( w_tLocal_V+laneID,
         VertexValues+globalTID ) ) {
27.       atomicExch( VertexValues+globalTID,
           w_tLocal_V[laneID] );
28.       converged = false;
29.     }
30.   }
31.   sync_device_with_host();
32. }
```

Figure 2.6: Framework's graph processing procedure pseudo-algorithm. Assumed warp size is 32. Shared memory pointers in the program code are declared with `volatile` qualifier.

Second step. This step involves iteration of warp lanes over the elements of the *EdgeValues* and *NbrVertexIndices* arrays region (lines 15 to 25 in Figure 2.6). Warp lanes perform a user-provided compute function with the fetched neighbor vertex value and the connected edge value and save the outcome in a local shared memory buffer (line 21). Besides, every warp lane must discover which of 32 vertices that are assigned to the warp owns the processed edge and neighbor. This involves a $\log 32 = 5$ stepped binary search on fetched *edgeIndices* in the shared memory (line 18). Using the resulting vertex index, warp lanes can be grouped into segments, each segment corresponding to one vertex. Each lane identifies its position within the segment and the size of the segment it belongs to (lines 19 and 20). Therefore warp lanes can execute user-provided reduction function in parallel (line 22). Finally, the first lane in each segment performs the reduction function over the outcome and associated element in the shared memory region for vertex values (lines 23 and 24). Warp lanes perform these steps iteratively until all the edges for the set of vertices are processed.

Third step. In this step, the warp lanes compare the content of designated shared memory region for vertex values with the corresponding *VertexValues* elements using the user-provided function (line 26). If the function returns true, the vertex content inside the global memory will be updated.

Once all the vertices are processed, the framework executes another iteration of the algorithm on all the graph vertices if any vertex in the current iteration is updated. Graph processing with WS method dynamically determines the proper size for reduction based on the segment size and it is guaranteed that the number of steps for parallel reduction will never exceed five ($\log warpSize$).

Note that the memory transactions in all the steps are *coalesced* except for accessing the neighbor vertex value (line 17), which is inherent in the compact graph representation like CSR. However by moving "binary search" and "segment realization" functions (lines 18 to 20) before the neighbor computation function, we exploit instruction level parallelism to hide the latency associated with the non-coalesced memory access.

2.2.2 Framework Interface

```
0.  struct Edge{ uint BW; };
1.  typedef unsigned int Vertex;
2.  inline __device__ void initVertex(
     volatile Vertex* initV, Veretx* V ){
3.     *initV = *V;
4.  }
5.  inline __device__ void ComputeNbr(
     Vertex SrcV, Edge* E, volatile Vertex* localV ) {
6.    *localV = min( SrcV, E->BW );
7.  }
8.  inline __device__ void ReduceVertices(
     volatile Vertex* firstV, Veretx* secondV ){
9.    *firstV = max( *firstV, *secondV );
10. }
11. inline __device__ bool IsUpdated(
     volatile Vertex* computedV, Veretx* V ){
12.   return ( *computedV > *V );
13. }
```

Figure 2.7: User-specified structures and functions for SSWP.

In addition to trivial input/output handling functions, type definition for the vertex, and the structure definition for the edge, our framework accepts the following user specified functions:

- *InitVertex* initializes the vertex at the beginning of an iteration.
- *ComputeNbr* is performed for every neighbor vertex.
- *ReduceVertices* acts as the reduction function between the results of *ComputeNbr* for two neighbors of a vertex.
- *IsUpdated* verifies if a vertex has updated during the current iteration.

Figure 2.7 illustrates the usage of the framework by showing the functions for Single Source Widest Path (SSWP) algorithm as an example. SSWP requires a variable for expressing the edge bandwidth and another variable for specifying maximum visible bandwidth by the

vertex from the source. In SSWP, during multiple rounds, the content of a vertex is updated by the maximum bandwidth it observes picked from the minimums between incoming edges and corresponding neighbors. As Figure 2.7 shows, this algorithm can be easily expressed in our framework via the above processing functions. First, the vertex content inside the shared memory is initialized by the most updated content of the vertex. Second, for each neighbor a local value is computed, which in this case is the minimum between every connecting edge bandwidth and its corresponding source vertex visible bandwidth. Third, these values are reduced two-by-two using the reduction function and the result is saved to the first argument content. For SSWP, reduction function selects the maximum of visible values through neighbors. Also, at the end of the third step of the processing procedure, the reduction function is executed for the initialized vertex and the final reduction result. Finally, in the fourth step, the framework verifies if the vertex should be updated. If the *IsUpdated* function returns true—which in case of SSWP is observing a greater bandwidth to the source—the content of the vertex inside global memory is replaced with the reduced vertex content at the current iteration. If any vertex is updated, the host must executes another iteration.

2.3 Scaling via Vertex Refinement

To handle larger graphs we must scale our method to use multiple GPUs that provide more memory and processing resources. Although graph partitioning strategies for GPUs have been explored, inter-GPU data transfer efficiency has not received adequate attention. Given a partitioning, for scaling of graph processing to be effective, we must make good use of limited PCIe bandwidth. We show the inefficiency of existing techniques and then present Vertex Refinement that avoids redundant data exchange between GPUs.

2.3.1 Inefficiency of Existing Inter-GPU Communication

Existing multi-GPU generic graph processing schemes divide the graph in two or more partitions and assign each partition to one GPU. Graph vertices completely fall into partitions while there can be edges that pass the partition boundaries. Due to these boundary edges, a GPU needs to be informed of the vertex updates happening in other GPUs. To keep the content of its assigned vertices held inside other GPUs updated, the GPU needs to transfer vertices belonging to its own partition over the PCIe bus. PCIe data transfer rate is tens of times slower than that of GPU global memory; thus extra care must be taken to transfer only necessary data so as not to waste PCIe precious bandwidth.

Nonetheless, since implementing a mechanism to efficiently manage queues in GPU's massively multithreaded environment is challenging, previous works choose simple but inefficient approaches. Medusa [94] copies all the vertices belonging to one device to other devices at every iteration. We refer to this solution as the ALL method. TOTEM [24, 25] pre-selects the boundary vertices in a pre-processing stage but similar to Medusa copies the boundary vertices after every iteration. We refer to this solution as Maximal Subset (MS) method. Both of these methods suffer from wastage of PCIe bandwidth because usually only a small portion of the vertices are updated during each iteration. Table 2.1 shows the ratio of useful transferred vertices—vertices that are updated in the last iteration—to all the vertices that are transferred in such schemes. Such low percentages motivate the need for a new solution that utilizes limited PCIe bandwidth economically.

To eliminate the overhead of transferring unnecessary vertices between devices, our framework performs Vertex Refinement in two steps: *offline* and *online*. We first describe the required data structures and then present the two-staged refinement procedure.

2.3.2 Data Structure Organization

To process a graph with multiple GPUs, our framework divides the vertices and their associated edges into partitions and assigns each partition to one GPU, so that each

Graph Algorithm	ALL %	MS %
Breadth-First Search (BFS)	10.43	12.21
Connected Components (CC)	9.55	11.19
Circuit Simulation (CS)	2.34	2.39
Heat Simulation (HS)	31.29	36.66
Neural Network (NN) [5]	15.66	18.34
PageRank (PR) [65]	10.38	13.65
Single Source Shortest Path (SSSP)	13.65	15.99
Single Source Widest Path (SSWP)	3.14	3.68

Table 2.1: The percentage of useful vertex data among all the transferred data when all the vertices (ALL) or the maximal subset of them (MS) are copied from one GPU to another. In this two-GPU configuration, the graph under the examination is an Rmat graph with approximately 40 million vertices and 470 million edges.

GPU processes a continuous set of vertices. Since the processing time is mostly affected by the memory accesses associated with gathering the values of neighbor vertices, determining the boundaries of vertex partitions depends upon the total number of edges that vertices of each subset hold. In our scheme, vertices of each partition will have roughly the same number of edges in order to provide a balanced load between GPUs. Each GPU will hold relevant subset of *NbrIndices*, *NbrVeretxIndices*, and *EdgeValues* but will contain a full version of *VertexValues* array. This organization allows each device to process vertices belonging to its own partition as long as vertices inside *VertexValues* that belong to other GPUs are updated during an iteration.

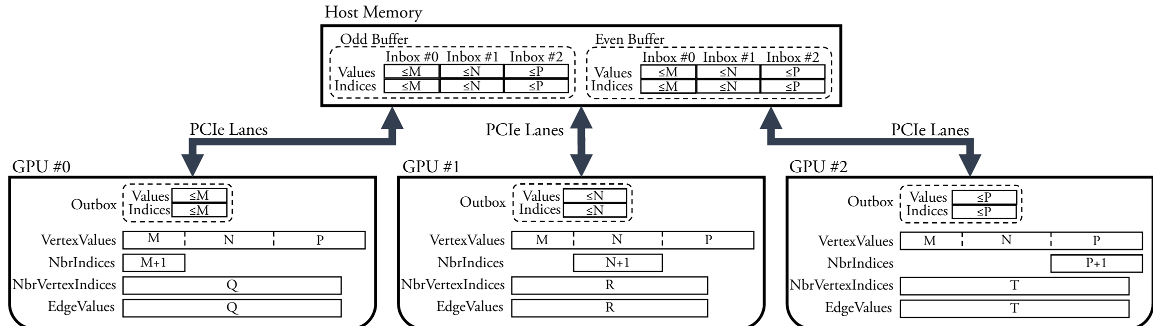


Figure 2.8: Organization of data structures in multi-GPU processing.

In addition to CSR representation buffers, each GPU will hold one *Outbox* buffer that is filled with updated vertex indices and vertex values of the GPU-specific division. As shown in Figure 2.8, we keep the inboxes inside host pinned buffers. In other words, the set of host buffers is similar to a hub that are filled by devices. At the start of an iteration, a device accesses inboxes corresponding to other devices and updates its own *VertexValues* array. Also at the end of an iteration, the device transfers its own outbox content to device's corresponding inbox. Moreover, we apply double buffering technique by alternating read buffers and write buffers. In an odd (even) iteration, devices read from the odd (even) inbox buffers and copy their outbox to their designated even (odd) inbox buffer. In summary:

- *Inbox and outbox buffers* are vital for a fast data transfer between GPUs. Direct peer-device memory access as an alternative will introduce significant performance penalty due to non-coalesced transactions over PCIe bus [86]. In contrast, inbox and outbox buffers allow the collection of necessary data together and hence accelerate the inter-device communication.
- *Using Host memory as the hub* not only reduces memory constraint pressure for GPUs, but is also beneficial when more than two GPUs are processing the graph. A device copies its own outbox to a host buffer only once. In contrast, if there is no intermediate host buffer, the device has to copy the outbox to each of the other GPUs' inboxes causing unnecessary traffic over connected PCIe lanes since the same data are passed over more than once. Our experiments show that using host as the hub is always beneficial in reducing the communication traffic and overall multi-GPU processing time in comparison to using inbox and outbox buffers residing inside the GPUs.
- *Double buffering* eliminates the need for additional costly inter-device synchronization barriers between data transfers and kernel executions. For instance, when device A grabs inbox buffer content of the device B during an iteration, since device B is going to fill another inbox buffer in the current iteration, needless of synchronizing with device B we will be sure that device A does not receive corrupted data.

If there are two GPUs processing the graph, during the runtime our framework queries the available global memory on the GPUs. If there is enough memory to hold the pertained part of the graph plus both the odd and even inboxes belonging to the other device, the framework puts the inboxes inside the GPUs global memory. Otherwise, it chooses host pinned buffers for this purpose.

2.3.3 Vertex Refinement

Offline Vertex Refinement. In this pre-processing stage, the framework scans *NbrVertexIndices* elements and identifies the boundary vertices: those that are being accessed by edges of one division while belonging to another division. For such a vertex, we set the most significant bit of its corresponding element inside *NbrIndices* buffer. During the online refinement, if a vertex is not a boundary vertex, it will be filtered out. Note that this bit will be ignored during other computations that involve *NbrIndices* buffer. Also during this stage, the framework can determine the maximum size to allocate for inbox and outbox buffers.

Online Vertex Refinement via parallel binary prefix sum. As opposed to Offline Vertex Refinement, Online Vertex Refinement happens on-the-fly inside the GPU kernel. At the last level of graph processing, lanes of a warp examine warp-assigned vertices for updates, each producing a binary predicate associated with one vertex. If this predicate is true and at the same time the vertex is marked during the offline stage, the vertex is required to be transferred to other devices.

By means of `__any()` intrinsic, we first verify if any of the warp lanes has an eligible vertex to transfer. If yes, warp lanes quickly count the total number of updated vertices inside the warp via intra-warp binary reduction and realize the number of updated vertices in lower lanes via intra-warp binary prefix sum. For a fast computation of binary reduction and inclusive binary prefix sum, our framework utilizes Harris et al. approach [31] in which `__popc()` and `__ballot()` CUDA intrinsic functions are exploited. Having total number of

updated vertices, one lane in the warp atomically adds it to a moving index inside the global memory, which its returned value specifies the starting position in the designated outbox buffer to write the warp's updated vertex indices and values. In other words, a lane reserves a region inside the Outbox for eligible warp lanes. The starting position of the region is shuffled to other lanes in the warp via `__shfl()` intrinsic, and lanes with updated vertex fill up the buffer using this position plus their intra-warp prefix sum. Figure 2.9 presents an example showing online vertex refinement procedure.

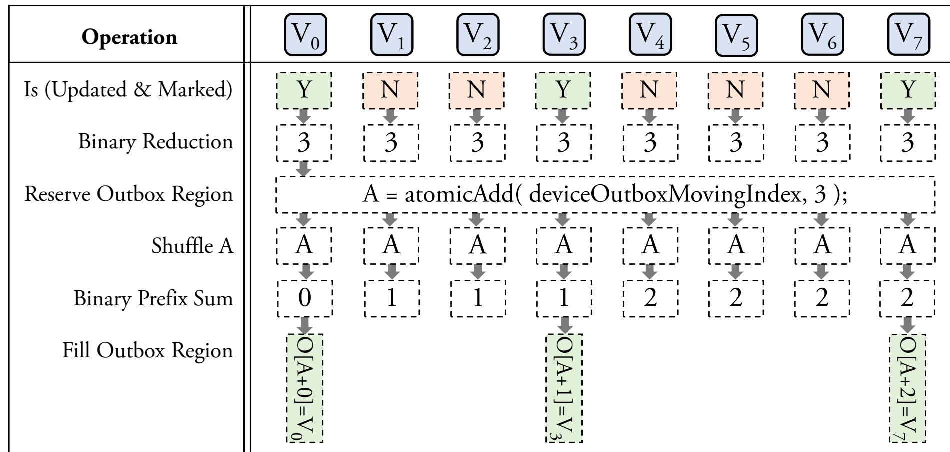


Figure 2.9: An example of online vertex refinement stages.

An alternative to above approach is extending the binary reduction and the binary scan to the CTA; however, we did not find this alternative faster since it required two synchronizations across the thread-block. Whereas in our approach the atomic addition is performed by only one lane in the warp which avoids heavy contention for the atomic variable.

When processing in an iteration is done, the moving index determines how much of the device outbox buffer has been filled. We significantly reduce the communication time by transferring the content of this buffer to the corresponding inbox buffer *only with the length specified by the moving index.* At the beginning of the next iteration, in order to have newly updated vertex values from other devices, each device distributes the content of

other devices' inboxes *only with the length specified by their associated moving indexes* over its own *VertexValues* array.

In summary, Offline Vertex Refinement identifies boundary vertices and Online Vertex Refinement recognizes the vertices that updated in the previous iteration. The combination of two yields the set of updated boundary vertices and maximizes the inter-device communication efficiency.

2.4 Performance Evaluation

The system we performed experiments on has 3 NVIDIA GeForce GTX780 GPUs each having 12 Kepler Streaming Multiprocessors and approximately 3 GBytes of global memory. The first GPU is connected to the system with PCIe 3.0 16x while the rest are operating at 4x speed. The single-GPU experiments are reported from the GPU with the highest PCIe bandwidth. We compiled and ran all programs for Compute Capability 3.5 on Ubuntu 14.04 64-bit with CUDA 6.5 and applied the highest optimization level flag.

2.4.1 Warp Segmentation Performance Analysis

In this section, we analyze the performance of Warp Segmentation on a single GPU. We use the graphs shown in Table 2.2 for experiments in this section. In the table, graphs with prefix *RM* refer to Rmat [11] graphs created with parameters $a = 0.45$, $b = 0.25$, and $c = 0.15$. Rmat graphs are known to imitate the characteristics of real-world graphs such as power-law degree distribution. The graph with prefix *ER* is a uniformly random (Erdős-Rényi) graph. Other graphs are extracted from real-world origins and are publicly available at SNAP dataset [50]. Graphs in Table 2.2 cover a wide range of sizes with different densities and characteristics.

Input Graph	N. Vertices	N. Edges	CSR size	CW size
RM33V335E	33 554 432	335 544 320	1611-3087	**5503-8321**
ComOrkut [90]	3 072 441	234 370 166	962-1912	**3762-5649**
ER25V201E	25 165 824	201 326 592	1007-1913	**3322-5033**
RM25V201E	25 165 824	201 326 592	1007-1913	**3322-5033**
RM16V201E	16 777 216	201 326 592	940-1812	**3288-4966**
RM16V134E	16 777 216	134 217 728	671-1275	2215-**3355**
LiveJournal [4]	4 847 571	68 993 773	315-610	1123-1695
SocPokec [80]	1 632 803	30 622 564	136-265	496-748
HiggsTwitter [18]	456 631	14 855 875	63-124	240-360
RoadNetCA [51]	1 965 214	5 533 214	38-68	96-149
WebGoogle [51]	875 713	5 105 039	27-51	85-130
Amazon0312 [49]	400 727	3 200 440	16-30	53-80

Table 2.2: Graphs used in single-GPU experiments – across benchmarks the size ranges in MBytes for CSR and CW representations. Sizes exceeding GPU's global memory capacity are bolded.

Warp Segmentation vs VWC – Performance Comparison.

First we compare the performance of WS method against VWC with graphs shown in Table 2.2 for the benchmarks in Table 2.1. Table 2.3 presents the raw processing time for the completion of all the benchmarks over all the graphs for both methods. We experimented on VWC with all the possible virtual warp sizes (2, 4, 8, 16, and 32) hence its processing times are specified in ranges. Table 2.4 shows the average speedup of WS compared to VWC over input graphs and benchmarks. In comparison with VWC, WS shows better performance across all the graphs and all the benchmarks. WS speedup over VWC when averaged across all the input graphs and benchmarks ranges from 1.29x to 2.80x.

To further examine the effectiveness of WS against VWC method, as the state-of-the-art CSR based generic graph processing method, we profiled both our framework and VWC over different graphs for warp execution efficiency. Figure 2.10 shows the average warp execution efficiency (predicted and non-predicted combined) over all the iterations of graph processing with *SSSP* benchmark. It is evident from the figure that for different graphs, best warp execution efficiency for VWC happens in different virtual warp sizes. For example

Input Graph		BFS	CC	CS	HS	NN	PR	SSSP	SSWP
RM33V335E	WS	1257	1118	1629	2812	1416	6056	2882	5505
	VWC	1428-1811	1270-1680	2012-2562	3501-4412	2030-2506	6563-8275	3237-3959	6740-8268
ComOrkut	WS	403	351	4162	681	904	4290	1398	931
	VWC	455-664	382-572	5566-8847	692-1056	989-1634	6296-13334	1515-2519	1029-1626
ER25V201E	WS	837	644	704	8330	773	5004	2181	2462
	VWC	976-1385	710-1045	748-1313	9499-16047	805-1160	5287-6095	2386-3505	2574-3756
RM25V201E	WS	845	835	1052	4782	1023	3856	1802	4216
	VWC	933-1231	935-1233	1287-1709	5619-8716	1190-1529	4183-5491	2080-2653	4787-5991
RM16V201E	WS	667	663	959	1762	840	3762	1625	2998
	VWC	750-907	746-908	1187-1438	2058-2337	984-1159	4043-4526	1800-2230	3403-4284
RM16V134E	WS	512	514	660	1244	572	4068	1159	2028
	VWC	591-820	592-822	850-1218	1539-2133	691-913	4448-5656	1402-1832	2427-3267
LiveJournal	WS	172	154	535	346	2061	2326	446	772
	VWC	215-296	201-273	807-1084	378-536	2297-4746	2498-4043	619-814	1059-1345
SocPokec	WS	75	66	121	226	464	1145	194	194
	VWC	90-107	80-106	175-203	264-329	614-761	1302-2817	237-327	236-314
HiggsTwitter	WS	48	37	117	75	159	483	100	77
	VWC	54-170	49-178	157-495	95-294	192-812	927-2433	113-432	98-355
RoadNetCA	WS	386	330	1694	41	193	55	465	1077
	VWC	480-3400	493-3437	2392-23659	45-301	191-1668	62-448	619-4402	118-5619
WebGoogle	WS	41	36	61	15	84	109	63	108
	VWC	81-109	75-99	124-186	23-35	124-167	145-248	113-172	186-288
Amazon0312	WS	17	17	263	81	41	44	33	38
	VWC	25-46	26-46	419-797	142-237	42-78	63-110	63-90	57-92

Table 2.3: Raw running times (ms) of Warp Segmentation (WS) and VWC including kernel executions and host-device data transfers for different algorithms and different graphs.

with *RoadNetCA*, a 2D mesh of intersections and roads, virtual warp size 2 yields the best results due to special structure of the graph; while it leads to the poorest performance for other graphs. On the other hand, WS exhibits a steady warp execution efficiency (71.8% on average) regardless of the graph. WS warp execution efficiency is 1.75x-3.27x better than VWC when averaged across all graphs. This result proves the SIMD efficiency of WS over fixed-width intra-SIMD thread assignment in VWC.

Warp Segmentation Performance against CW.

We present the speedup of WS over CW having large graphs in Table 2.5 and having small graphs in Table 2.6. For the large graphs, CW representation cannot fit the whole graph inside GPU global memory. For these combinations, CuSha fails; therefore, as

Averages Across Input Graphs		Averages Across Benchmarks	
BFS	1.27x−2.60x	RM33V335E	1.23x−1.56x
CC	1.33x−2.90x	ComOrkut	1.15x−1.99x
CS	1.43x−3.34x	ER25V201E	1.09x−1.69x
HS	1.27x−2.66x	RM25V201E	1.15x−1.57x
NN	1.21x−2.70x	RM16V201E	1.16x−1.41x
PR	1.22x−2.68x	RM16V134E	1.22x−1.69x
SSSP	1.31x−2.76x	LiveJournal	1.29x−1.99x
SSWP	1.28x−2.80x	SocPokec	1.27x−1.77x
		HiggsTwitter	1.34x−4.78x
		RoadNetCA	1.24x−9.90x
		WebGoogle	1.79x−2.69x
		Amazon0312	1.53x−2.68x

Table 2.4: Speedup ranges of Warp Segmentation over VWC excluding data transfer times. Since both methods use CSR representation, their data transfer times are equal.

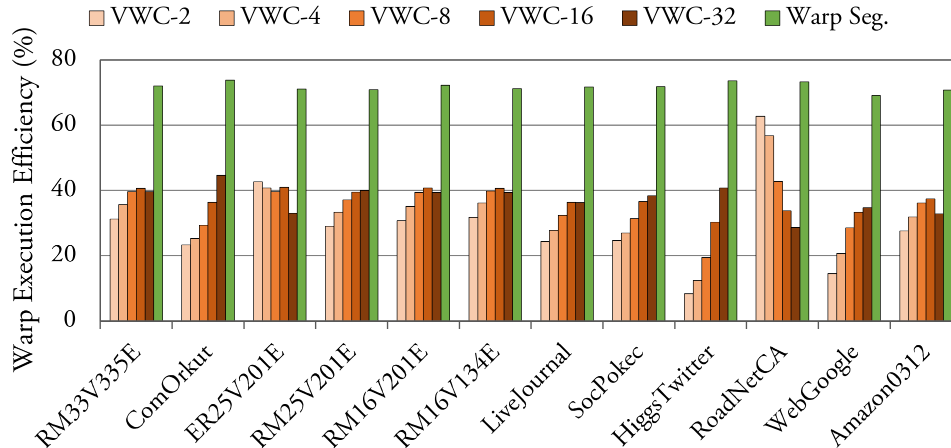


Figure 2.10: Profiled average warp execution efficiency of Warp Segmentation compared to VWC's. *SSSP* is the benchmark.

a straightforward workaround, we kept vertex value and small auxiliary buffers inside the GPU global memory and put shards at mapped pinned buffers inside the host. For large graphs, CW processing time is significantly higher than our method's due to involvement of PCIe bus, limiting the scalability of CW representation. Also for the small graphs, although CW provides fully regular access patterns, it incurs larger memory footprints. In addition, our framework covers the latency of CSR-inherent irregular accesses, therefore we observe

Input Graph	BFS	CC	CS	HS	NN	PR	SSSP	SSWP
RM33V335E	3.41	3.21	8.44	14.14	4.02	5.38	4.36	4.66
ComOrkut	5.11	5.91	1.72	10.76	5.23	6.85	7.92	5.72
ER25V201E	3.47	3.36	6.20	10.43	3.72	2.59	4.46	4.34
RM25V201E	3.07	2.76	7.71	9.65	3.55	3.54	3.99	4.14
RM16V201E	3.45	3.06	6.53	8.42	3.87	4.50	4.63	4.41
RM16V134E	x	x	3.19	4.97	x	3.93	x	x
Average	3.70	3.66	5.63	9.73	4.08	4.47	5.07	4.65

Table 2.5: The speedup of Warp Segmentation over CuSha's [42] CW for *large* graphs. The shards reside inside the host pinned buffers (x means graph is small - fits in GPU memory).

Input Graph	BFS	CC	CS	HS	NN	PR	SSSP	SSWP
RM16V134E	0.74	0.80	x	x	0.88	x	0.67	0.56
LiveJournal	1.06	1.21	0.74	1.10	1.03	0.60	0.86	0.82
SocPokec	0.92	1.02	1.04	0.81	0.41	0.34	0.73	0.67
HiggsTwitter	1.48	2.30	1.48	1.64	2.20	1.19	1.65	2.03
RoadNetCA	0.67	1.13	0.98	0.92	1.02	1.20	0.76	0.91
WebGoogle	0.58	0.82	0.78	1.74	1.69	0.59	0.61	0.74
Amazon0312	1.05	1.47	0.39	0.91	0.91	0.97	1.21	1.20
Average	0.93	1.25	0.90	1.19	1.16	0.82	0.93	0.99

Table 2.6: The speedup of Warp Segmentation over CuSha's [42] CW for *small* graphs. The shards reside inside the GPU's global memory (x means graph is large - requires host memory).

near par performance, as shown by averages in Table 2.6.

2.4.2 Vertex Refinement Performance Analysis

Next we analyze the performance of our framework when it is scaled to multiple GPUs. First we present the speedup provided by Vertex Refinement compared to existing methods over very large input graphs, and analyze its cost and benefits. For the experiments in this section, we created 12 Rmat and Erdős-Rényi graphs with different sizes and densities, shown in Figure 2.7. 6 of these graphs can be fit inside two of our GPUs and 6 require three GPUs. Finally, we analyze the performance when smaller graphs are processed on multiple GPUs.

Input Graph	N. Vertices	N. Edges
RM54V704E	54 525 952	704 643 072
ER50V671E	50 331 648	671 088 640
RM50V671E	50 331 648	671 088 640
RM46V671E	46 137 344	671 088 640
RM46V603E	46 137 344	603 979 776
RM41V536E	41 943 040	536 870 912
RM41V503E	41 943 040	503 316 480
ER39V469E	39 845 888	469 762 048
RM39V469E	39 845 888	469 762 048
RM37V469E	37 748 736	469 762 048
RM37V436E	37 748 736	436 207 616
RM35V402E	35 651 584	402 653 184

Table 2.7: Graphs for multi-GPU experiments: Top 6 graphs used in experiments with 3 GPUs; rest used with 2 GPUs.

Vertex Refinement Performance Comparison.

To better realize the importance of data communication strategy and the efficiency of Vertex Refinement, we have implemented two other inter-device communication methods in our framework. The first method is the straightforward solution that copies all the vertices belonging to one device to other devices at every iteration. We refer to this solution as *ALL*. The second one is the maximal subset method where vertices that belong to one device and can be accessed by another device are identified in a pre-processing stage. During the iterative execution, only these vertices are communicated to other devices. We refer to this method as *MS*. We compare these methods with Vertex Refinement - *VR*. Note that to better realize the benefits of VR, for all the inter-device communication methods, we keep intra-device processing style intact. In other words, underlying graph processing method is WS for all the experiments in this section.

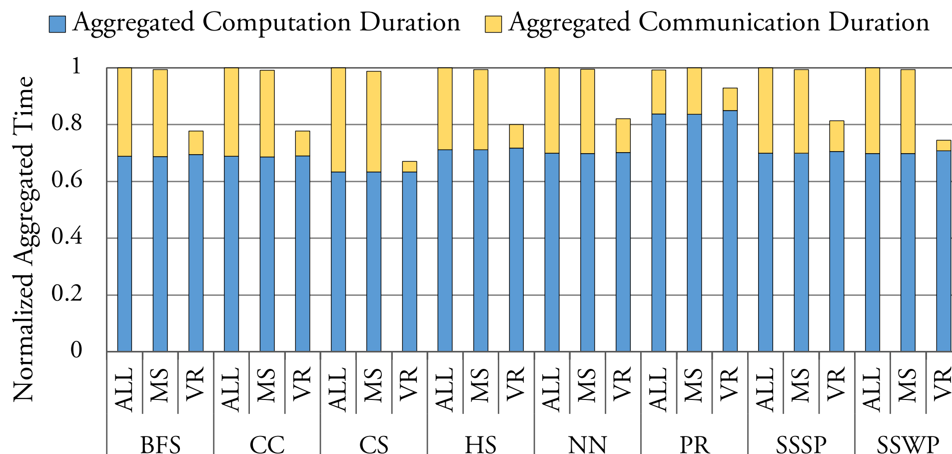
Table 2.8 shows the speedup of our framework when VR is employed over ALL and MS, for all the graphs and benchmarks. In all cases, our solution performs better than other methods. When averaged over all the graphs and benchmarks, our approach provides 1.81x and 1.30x speedups over ALL and 1.77x and 1.28x speedups over MS for three-GPU

Input Graph		BFS	CC	CS	HS	NN	PR	SSSP	SSWP
RM54V704E	over ALL	1.85	1.81	2.53	1.64	1.66	1.48	1.75	2.03
	over MS	1.82	1.78	2.46	1.59	1.63	1.47	1.71	1.98
ER50V671E	over ALL	1.64	1.36	2.19	1.55	1.43	1.22	1.72	2.02
	over MS	1.67	1.4	2.24	1.49	1.48	1.21	1.76	2.05
RM50V671E	over ALL	1.83	1.76	2.51	1.68	1.63	1.49	1.72	1.98
	over MS	1.78	1.74	2.47	1.6	1.6	1.36	1.68	1.93
RM46V671E	over ALL	1.78	1.77	2.48	1.7	1.62	1.43	1.72	1.98
	over MS	1.75	1.74	2.42	1.64	1.6	1.41	1.69	1.93
RM46V603E	over ALL	1.84	1.82	2.58	1.67	1.67	1.43	1.79	2.07
	over MS	1.81	1.8	2.51	1.59	1.64	1.37	1.75	2.01
RM41V536E	over ALL	1.89	1.84	2.71	1.62	1.69	1.44	1.8	2.1
	over MS	1.82	1.81	2.63	1.58	1.66	1.39	1.75	2.04
RM41V503E	over ALL	1.29	1.29	1.61	1.23	1.21	1.18	1.24	1.35
	over MS	1.27	1.28	1.57	1.21	1.2	1.15	1.21	1.32
ER39V469E	over ALL	1.21	1.06	1.49	1.18	1.14	1.19	1.23	1.21
	over MS	1.23	1.09	1.53	1.16	1.17	1.15	1.25	1.24
RM39V469E	over ALL	1.29	1.3	1.64	1.28	1.21	1.39	1.26	1.38
	over MS	1.28	1.28	1.61	1.24	1.2	1.29	1.23	1.35
RM37V469E	over ALL	1.26	1.26	1.6	1.24	1.2	1.23	1.22	1.36
	over MS	1.25	1.26	1.57	1.21	1.19	1.18	1.2	1.33
RM37V436E	over ALL	1.33	1.32	1.66	1.27	1.22	1.25	1.28	1.41
	over MS	1.31	1.29	1.63	1.23	1.22	1.24	1.26	1.39
RM35V402E	over ALL	1.32	1.31	1.72	1.28	1.23	1.21	1.25	1.41
	over MS	1.3	1.29	1.66	1.23	1.22	1.2	1.22	1.38

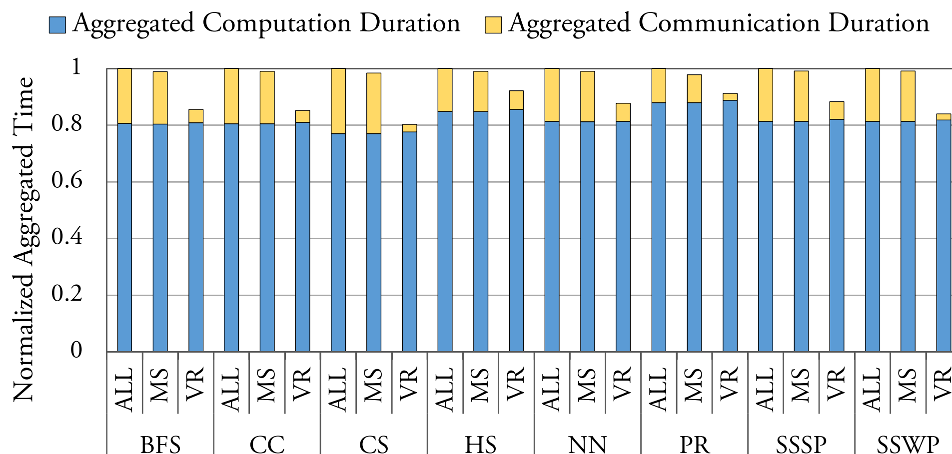
Table 2.8: The speed-up of VR over ALL and MS for three-GPU and two-GPU configurations.

and two-GPU configurations respectively.

In Figure 2.11, we analyzed the cost of VR queue management versus the savings it provides (in terms of eliminating redundant inter-device vertex communication) by breaking down the processing time into computation time and communication time. To create this plot, we measured the time for each and every kernel execution, memory copy, and outbox-loading/inbox-unloading. Aggregated computation duration refers to the total duration of GPU kernel executions, whereas aggregated communication time refers to the total duration of copies and/or box handling kernels.



(a) RM54V704E graph with 3 GPUs.



(b) RM41V503E graph with 2 GPUs.

Figure 2.11: Processing-time break down into computation time and communication time for the Vertex Refinement (VR) compared to ALL and MS. Computation time is the total duration of kernel execution, and communication time is the total duration of inbox/outbox management plus inter-device memory copies. For each benchmark, the times are normalized with respect to the longest time. Note that this times cannot be used to infer the overall speedup due to asynchronicity of devices.

First, it is evident from both plots in Figure 2.11 that MS is not an effective solution for reducing communication overhead. In fact, in one case (*PR* in Figure 2.11a) the overhead of outbox handling overcomes the benefits of pre-selection. Second, unlike MS, VR significantly reduces the total communication duration by refining vertices on-the-fly while adding negligible overhead to the computation duration. Note that even though in VR the vertex information is communicated accompanying its index, the communication duration is still much less compared to ALL and MS for all the cases. We note that another overhead of VR is higher register consumption. In our configuration, compared to ALL and MS, the kernel with Online Vertex Refinement requires two more 32-bit registers per thread, asking 30 in total. Since Nvidia Kepler architecture allows a thread to consume up to 32 registers without limiting occupancy, this higher register pressure does not hurt the performance. Third, by comparing Figure 2.11a and Figure 2.11b, we notice that more time is spent on the communication by employing more GPUs. By adding another GPU, each device needs to send and receive more vertex information to and from more devices, signifying VR's supremacy even further. Especially in the 3-GPU configuration, using host as the hub supports reducing inter-device traffic by passing the data over PCIe only once.

Scaling to multiple GPUs for smaller graphs.

To observe the effect of scaling graph processing procedure from one or two GPUs to three GPUs, we experimented our framework with smaller graphs and more GPUs and reported the speedups in Table 2.9. As this table shows, the performance does not scale linearly as we add more GPUs. This is due to comparatively slow PCIe paths and also imperfect load division between different GPUs. Also, the speedup of adding more GPUs greatly depends on the graph algorithm. For example, in PageRank (PR) the chances that a vertex is updated during an iteration is relatively high (especially in earlier iterations) thus more vertices have to be transferred from one GPU to another. As a result, we observe lower speedups in PageRank compared to other algorithms when adding more GPUs.

Input Graph	GPUs	BFS	CC	CS	HS	NN	PR	SSSP	SSWP
RM41V503E	3 vs. 2	1.39	1.38	1.32	1.23	1.21	1.12	1.32	1.35
ER39V469E	3 vs. 2	1.36	1.11	1.44	1.19	1.33	1.09	1.28	1.26
RM39V469E	3 vs. 2	1.3	1.37	1.42	1.22	1.28	1.13	1.42	1.29
RM37V469E	3 vs. 2	1.21	1.27	1.32	1.24	1.38	1.19	1.34	1.43
RM37V436E	3 vs. 2	1.22	1.18	1.32	1.17	1.21	1.18	1.28	1.34
RM35V402E	3 vs. 2	1.5	1.37	1.42	1.23	1.29	1.14	1.33	1.39
RM33V335E	3 vs. 1	1.75	1.56	1.95	1.33	1.52	1.1	1.55	1.59
	2 vs. 1	1.27	1.24	1.4	1.07	1.12	1.06	1.21	1.2
ComOrkut	3 vs. 1	1.65	1.81	1.95	1.28	1.97	1.43	1.96	1.85
	2 vs. 1	1.19	1.31	1.65	1.15	1.36	1.32	1.4	1.39
ER25V201E	3 vs. 1	1.5	1.55	1.44	1.19	1.38	1.18	1.48	1.58
	2 vs. 1	1.14	1.33	1.16	1.07	1.13	1.15	1.11	1.19
RM25V201E	3 vs. 1	1.47	0.96	1.56	1.29	1.38	0.93	1.29	1.17
	2 vs. 1	1.08	0.97	1.26	1.1	1.08	1.01	1.07	0.94
RM16V201E	3 vs. 1	1.45	1.64	1.74	1.3	1.56	1.02	1.42	1.6
	2 vs. 1	1.26	1.36	1.44	1.12	1.21	1.06	1.17	1.34
RM16V134E	3 vs. 1	1.36	1.58	1.86	1.36	1.47	1.19	1.46	1.66
	2 vs. 1	1.21	1.21	1.44	1.14	1.11	1.12	1.12	1.31

Table 2.9: The speedup of our framework when scaling to more GPUs: From 2 to 3 GPUs for the top 6 graphs; and From 2 to 3 and from 1 to 2 GPUs for the rest of the graphs.

We also present the effect of the graph characteristics (graph size and density) on the scalability of our framework in Figure 2.12. By comparing large graphs and small graphs in Figure 2.12, we observe that as the graphs get larger with greater number of edges, adding more GPUs produces greater reductions in graph processing time. In addition, higher density in larger graphs signifies the reduction in the processing time when scaling to multiple GPUs by downsizing inter-device vertex transfer volumes.

2.5 Summary

This chapter introduced a CUDA-based solution for efficient scaling of iterative graph algorithms to larger graphs and multiple GPUs. The graphs are stored in the space-saving CSR form that allows processing large graphs. To overcome the SIMD execution inefficiency in existing CSR-based graph processing methods, this chapter proposed Warp

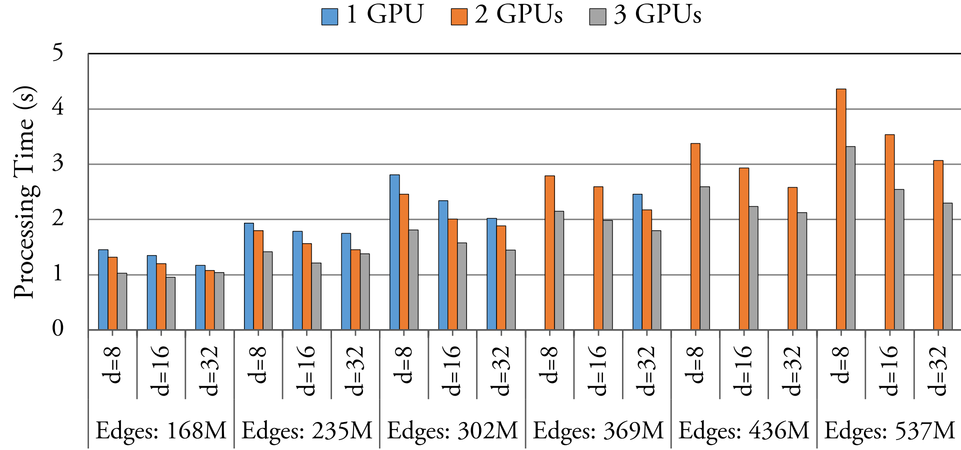


Figure 2.12: The scalability of our framework over graphs with different number of edges and densities for *SSSP* benchmark. All the graphs are Rmat created with parameters $a = 0.45$, $b = 0.25$, and $c = 0.15$. y axis is the processing time (lower is better).

Segmentation, a novel technique that assigns appropriate number of warp threads to process vertices with irregular-sized neighbors on-the-fly and provides 1.29x−2.80x speedup over state-of-the-art Virtual Warp-Centric method. It also offered Vertex Refinement in order to scale the graph processing over multiple GPUs. Vertex Refinement efficiently collects and transfers only those vertices that are boundary and recently updated. Vertex Refinement maximizes the inter-device bandwidth utilization efficiency and enables up to 2.71x exclusive speedup over existing multi-GPU communication schemes.

Chapter 3

Enabling Work-Efficiency

Proposed solution in the previous chapter, due to being generic and being able to express a wide variety of algorithms, processes *all* the vertices and visits *all* the neighbors belonging to vertices iteratively. However, in reality only a subset of vertices and their incoming edges need to be processed in each iteration. To study this issue, we measured the percentage of updated vertices and their connected edges across iterations for two well-known graph algorithms, and plotted the results in Figure 3.1. It is evident from these plots that in most iterations the vast majority of the vertices do not cause any changes in the values, and thus need not be processed. In other words, the necessary volume of work to carry out is only the integral of presented plots. The redundant calculations account for a great deal of SIMD power and memory bandwidth being wasted.

In another class of GPU graph processing solutions ([60, 16, 84]) this problem is mitigated by focusing on vertex and edge frontiers. These solutions are categorized as push-based methods [63] where the computation is invoked alongside outgoing edges. Nevertheless, push-based approaches have a few disadvantages compared to vertex-centric model which limit their usage. First, push-based methods have to deal with the multiple writers issue, hence for a correct functionality they require *atomics* in the main computation routine which restricts the framework expressiveness and complicates the description of the desired

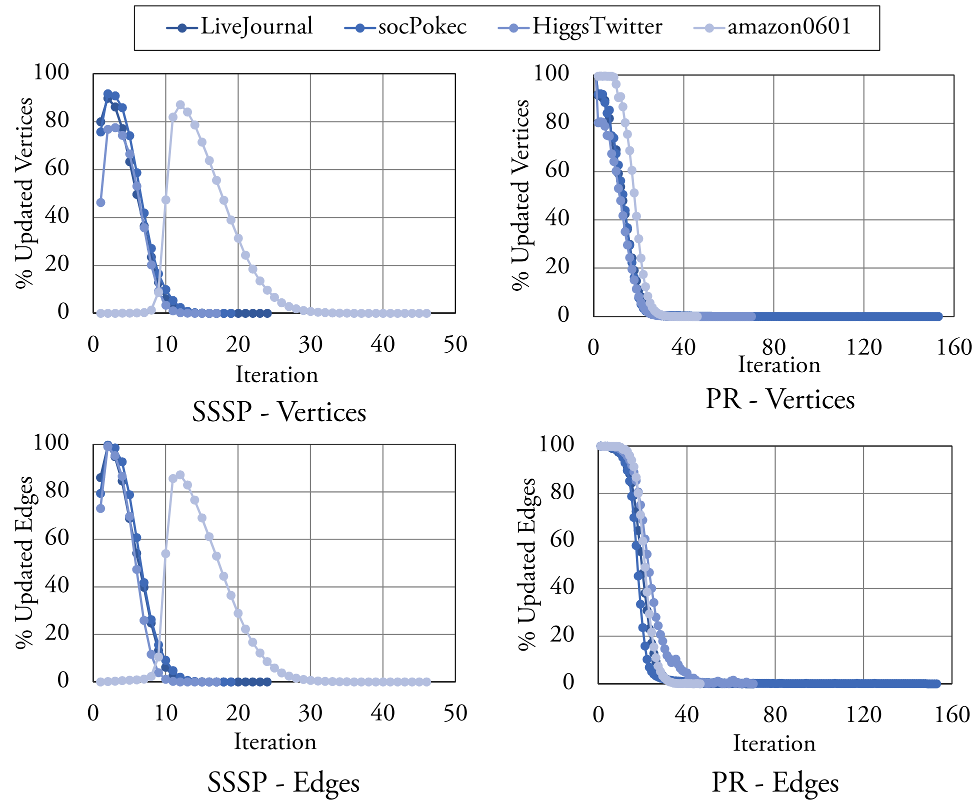


Figure 3.1: The percentage of updated vertices and their connected edges across iterations.

algorithm. On the other hand, vertex-centric model follows the single writer semantics and does not use atomics. Second, unlike vertex-centric methods, push-based approaches inhibit natural out-scaling. A GPU has to perform remote atomics on scattered locations of the destination device's memory resulting in significant performance loss. Finally, it is easier to *think like a vertex* and express an algorithm in the vertex-centric model as opposed to *think like a frontier*. Henceforth, the focus of this chapter and this thesis is vertex-centric model.

This chapter presents a mechanism to keep track of *active* vertices– the subset of vertices that are subject to change in the current iteration– and subsequently process only those, providing a vertex-centric yet work-efficient design that functions according to the propagation of vertex *activities* in the graph. This mechanism records the vertex

activeness within an auxiliary *bitmask* data structure in which each bit corresponds to one vertex. During an iteration, threads retrieve the content of assigned vertex's bit in the bitmask, and process only those that are activated. Processing an active vertex during an iteration, if the vertex is updated, activates the vertices sitting at the other end of its outgoing edges. Compared to frontier-based approaches, bitmasks incur much lower memory overhead, facilitate scaling the solution to multiple GPUs, and eliminate the need for– and consequently the cost of– a duplicate elimination strategy.

Adding work-efficiency to vertex-centric graph processing naturally requires a selective vertex processing approach. In other words, only those vertices that are active will be processed in the current iteration. Accompanied with the irregularity of the real-world graphs [55], it necessitates a task expansion strategy that not only accounts for visiting vertices with irregular number of neighbors, but also distributes these irregular adjacency lists– which may be located disjointly in memory– appropriately among threads. We introduce a *dynamic task assignment* strategy that utilizes high performance GPU primitives to efficiently map threads inside the warp to the adjacency list elements of the active vertices. Using this technique, the kernel avoids intra-warp thread starvation during neighbor expansion and sustains a high SIMD efficiency.

Moreover, enabling work-efficiency for vertex-centric computation of a directed graph requires us to accompany its CSC representation with its primary CSR representation. This places burden on limited available GPU global memory and limits the size of the graph that can be held inside and processed by a GPU. We propose *vertex grouping* technique that creates a trade-off between the work-efficiency and the consumed amount of DRAM. In *vertex grouping*, the vertices in the CSC representation are grouped together and multiple edges between the groups are represented only by one edge. As the number of vertices inside a group increases, the size of the CSC representation is reduced. However, on the other hand, all the vertices inside the vertex group are activated together which introduces some work-inefficiency.

A common approach for scaling a graph computation to multiple GPUs is to

partition the graph and store each partition inside a GPU's DRAM. However, this scheme can lead to inter-device load imbalance since during an iteration GPUs can have significantly different number of active vertices to process. We mitigate this issue using *permissive partitioning* technique. In *permissive partitioning*, the graph partitions stored in the GPUs have overlap to an extent the global memory of devices allow. Therefore, during an iteration, vertex activeness distribution is analyzed using the constructed bitmask and the regions at which GPUs process the graph are resized accordingly so as to provide a maximally balanced load between GPUs.

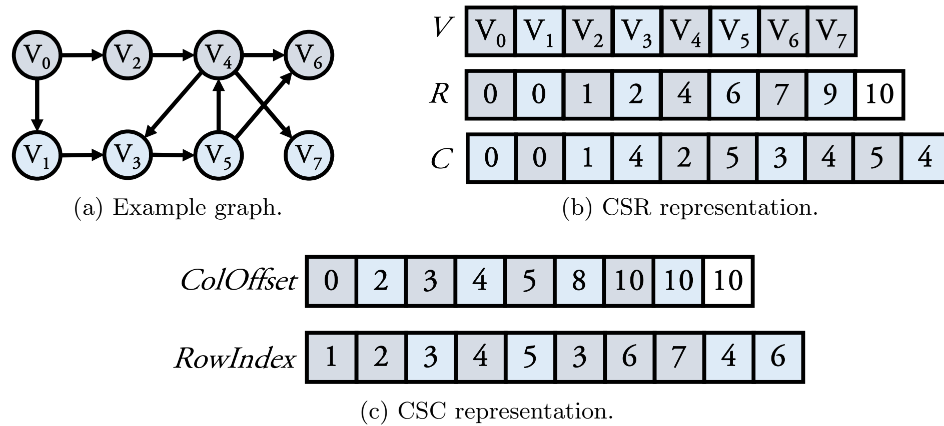
We packaged aforementioned methods in a CUDA C++ template library named KiTES allowing its ultimate integration with the user's code– unlike existing frameworks which are implemented as stand-alone programs. While KiTES abstracts away the complication of the GPU command enqueuing and kernel orchestration via its API, it gives users freedom to control various execution settings such as specific GPUs that process the graph, the synchronicity of the computation, etc. Most importantly, the user supplies the desired device function objects which are passed to the main GPU kernels and constitute the heart of the computation. Our experiments show that KiTES provides up to 5.46x and 1.67x speedup over the non-work-efficient solution for single and multi-GPU configurations over multiple algorithms and inputs respectively.

The remainder of the chapter is organized as follows: Section 3.1 presents the required data structures for enabling work-efficiency. Section 3.2 and Section 3.3 describe techniques to overcome the warp efficiency and scalability challenges induced as work-efficiency side-effects. Section 3.4 discusses the KiTES library collecting these techniques. Section 3.5 gives the results of our evaluation and Section 3.6 summarizes the chapter.

3.1 Data Structures for Work-Efficiency

In iterative vertex-centric graph processing, a vertex's value is subject to change only if the value of at least one of the vertices at the other end of its incoming edges changes.

In the existing vertex-centric work-inefficient approach [42, 41], updates are captured automatically since all the vertices– regardless of the changes in the graph in previous iterations– are processed in each iteration. However, to enable work-efficiency, the set of vertices that are activated by the changes in the current iteration must be identified. Therefore, the set of outgoing neighbors for a vertex have to be discoverable by threads inside the kernel with minimum overhead. Towards this end, for directed graphs, we accompany the Compressed Sparse Row (CSR) representation of the graph with primary components of the Compressed



(a) Example graph.

(b) CSR representation.

(c) CSC representation.

Figure 3.2: An example directed graph, its CSR representation, and the main components of its CSC representation.

Sparse Column (CSC) format, as shown in Figure 3.2. An array named *ColOffset* with size $|V| + 1$ stores the prefix sum of the number of outgoing neighbors for vertices (i.e., the frequency of non-zero elements in columns of the adjacency matrix), and another array named *RowIndex* with size $|E|$ that stores the index of the outgoing neighbors. In this representation, the indices of vertices pointed to by a vertex are placed contiguously and their location inside *RowIndex* is easily retrievable by an access to *ColOffset* at vertex's index position.

As an example, consider the vertex V_4 in Figure 3.2. The 4th and 5th elements of R array in Figure 3.2b tell us that the set of V_4's incoming neighbors start from the 4th element in C and end before the 6th element in C, which gives vertices 2 and 5 as indices

for vertices sitting at the other end of incoming edges. Now, for the outgoing neighbors, the 4th and 5th elements in $ColOffset$ array reveal that the set of outgoing neighbors start from 5th element in $RowIndex$ and end before the 8th element in $RowIndex$, thus vertices with indices 3, 6, and 7 are at the other end of V_4's outgoing edges. Please note that if the graph is undirected, its CSC representation is not necessary due to the graph's symmetric adjacency matrix.

3.1.1 Recording Vertex Activeness using Bitmasks

We utilize two auxiliary bitmask data structures in order to keep track of active vertices. Both bitmasks reside inside the GPU global memory and the corresponding bits of the bitmasks are associated with one vertex in the graph. Setting a bit in the bitmask means that the corresponding vertex will be active and thus will be processed in the next iteration. During a given iteration, one of the bitmasks is written to for recording the active vertices that will be processed during the next iteration, and the other one contains the activeness of vertices for the current iteration, i.e. it was written in the previous iteration. At the end of each iteration, bitmasks switch their roles, and before invoking the main computation kernel, the bitmask to be written is reset.

Our approach for keeping track of the activeness of vertices to provide work-efficiency has the following advantages:

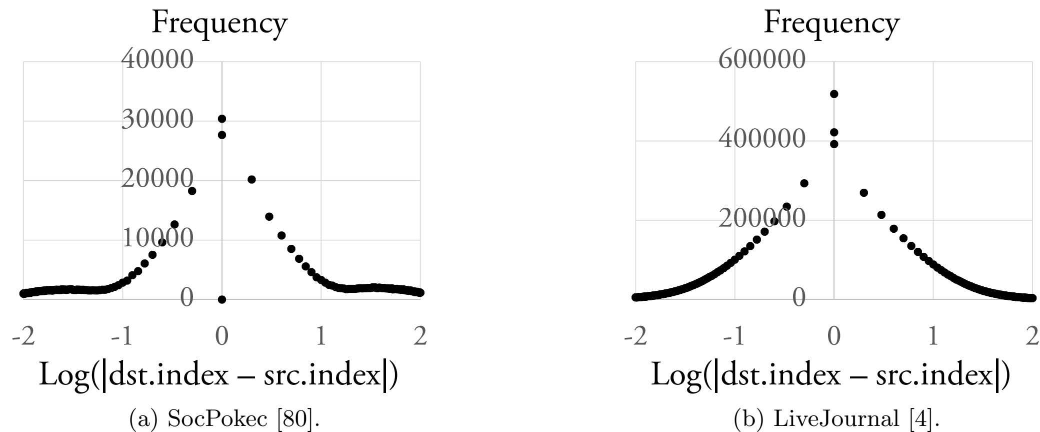
- **Minimum memory overhead**. Only two bits are needed for a vertex. The upper-limit for the space to store the bitmaskss is $2 \times 4 \times \lceil \frac{|V|}{32} \rceil$ bytes. In comparison, frontier-based approaches require buffers to collect active frontier in which each element needs to be the size of vertex index. Without pre-processing, the algorithm must set aside storage to keep producer and consumer frontiers each of which in general can be as large as the number of edges, thus leading to overall storage overhead upper-limit of $2 \times |E| \times sizeof(vIdx)$ bytes. To put this into perspective, consider the LiveJournal [4] graph that has around 4 million vertices and 34 million edges. The required storage to

store bitmasks would be only 1 MB while frontiers require 272 MB of space (assuming vertex indices are 4 bytes long each).

- **Automatic duplicate elimination**. To provide work-efficiency, frontier-based approaches require schemes to eliminate duplicate indices in the new frontier. Hashing is the preferred method in this case and is utilized to imitate the behavior of an unordered set. However, in addition to consuming compute resources, hashing inevitably introduces non-coalesced memory accesses that reference irregular memory positions. Moreover, hashing approaches requires additional data structures. On the other hand, using bitmasks instinctively eliminates the duplicates since setting an already set bit in the bitmask has no adverse effect (i.e., no duplicates are created).

- **Facilitates multi-GPU scalability.** To enable work-efficiency in a multi-GPU processing environment, during each iteration each device needs to know the set of vertex indices assigned to it that are activated by vertices at other GPUs. A GPU can easily do this by *OR*ing its bitmask section with the corresponding bitmask section in other GPUs. In comparison, in frontier-based approaches, explicit vertex indices need to be copied across devices which requires duplicate elimination both before inter-device transfer to save PCIe bandwidth and after inter-device transfer to avoid redundant processing.

As mentioned earlier, in our method, each vertex of the graph is assigned to one specific bit inside a bitmask. Considering that the global memory writes in Kepler architecture (and also later architectures) are cached in L2, and in currently available CUDA-enabled GPUs L2 holds 32 bytes per cache line, setting bits for the vertices inside the bitmask has the probability of hitting the cache if at least one of 255 adjacent vertices have recently been marked. There is a considerably high likelihood of a hit given that nearby vertex indices in real-world graphs are more likely to be neighbors compared to indices that are far from each other. Figure 3.3 illustrates this phenomenon by plotting the difference of vertex indices of source and destination of edges belonging to two real-world



(a) SocPokec [80]. (b) LiveJournal [4].

Figure 3.3: The frequency distribution of differences between source and destination indices of edges of 2 real-world graphs.

Graph	Algorithm	$\|CSR\| + \|V\| + \|E\|$	$\|CSC\|$
LiveJournal [4]	BFS	315 MB	295 MB
	SSSP	591 MB	
socPokec [80]	BFS	136 MB	129 MB
	SSSP	258 MB	
HiggsTwitter [18]	BFS	63 MB	61 MB
	SSSP	123 MB	
amazon0601 [49]	BFS	17 MB	15 MB
	SSSP	30 MB	

Table 3.1: The memory required for CSR representation of four directed graphs [50] and their additional CSC representation.

graphs. The high frequency of data points distributed close to the y axis confirms our claim. Moreover, atomic operations used to set bits of the bitmask within the CUDA kernel have fire-and-forget semantics. Thus, the warp scheduler can hide their latency by moving on and executing further instructions (from same warp or other warps) after an atomic request.

3.1.2 Vertex Grouping

We previously mentioned that directed graphs in our solution require CSC representation to be processed in a work-efficient manner. *RowIndex* and *ColOffset* arrays in

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0
3	0	1	0	0	1	0	0	0
4	0	0	1	0	0	1	0	0
5	0	0	0	1	0	0	0	0
6	0	0	0	0	1	1	0	0
7	0	0	0	0	1	0	0	0

(a) Graph's adjacency matrix before *vertex grouping*.

	0	1	2	3
0	1	0	0	0
1	1	0	1	0
2	0	1	1	0
3	0	0	1	0

ColOffset	0	2	3	6	6	
RowIndex	0	1	2	1	2	3

(b) Graph's adjacency matrix and its CSC representation after *vertex grouping*.

Figure 3.4: The effect of *vertex grouping* with ratio 2 on the size of the CSC representation of the example graph in Figure 3.2.

CSC impose storage overhead especially in discrete GPUs with fixed amount of DRAM. Table 3.1 compares the required memory to store CSR format plus the vertex and edge arrays with the required memory to store CSC representation for multiple graphs and two algorithms. Based on the table, adding CSC incurs 1.5x-1.97x memory overhead. This high memory overhead can potentially limit the size of the graph that can be processed by the GPU. To ameliorate this issue and enable processing of larger graphs, we propose *vertex grouping* technique to be applied onto these two buffers.

In *vertex grouping*, fixed number of vertices with consecutive indices are grouped and represented as one entity in *RowIndex* and *ColOffset* arrays. If vertex A in the graph has an edge to vertex B, the vertex group containing A will have an edge to the vertex group containing B. Using this technique, if there are multiple edges between vertices in two vertex groups, they will be represented by one element inside the *ColOffset* array, hence reducing the array size, as demonstrated by the example in Figure 3.4. The amount of the reduction in the *ColOffset* array depends on the structure of the graph. However, with a grouping ratio of *Ratio*, the space used by *RowIndex* will be reduced by a factor of *Ratio*. Note that when vertices are grouped, the bitmasks shrink according to the grouping ratio.

Vertex grouping provides a knob that can be turned to perform trade-off between memory consumption and work efficiency. By representing a group of vertices as a single combined entity we save space; however, if any vertex inside a group is activated, all the vertices inside the group must be processed in the next iteration, i.e. the computation will include some redundancy. The parameter that controls this trade-off is the number of vertices per group. The more the number of vertices inside a group, the less memory is consumed to store two aforementioned arrays, and the more the redundant computation performed. Note that a work-inefficient approach sits at the end of this spectrum where the whole adjacency matrix is represented as one entity.

3.2 Warp Efficiency with Dynamic Thread Assignment

Figure 3.5 presents the core computation routine in our solution. Until the graph computation converges, the host side launches GPU kernels in which parallel threads are assigned to active vertices. A local version of the vertex content is initialized using the user-provided device function (line 3). Then, for all the incoming neighbors of the active vertex a neighbor visitation function (line 5) and a reduction function (line 6) are executed both of which are provided by the user for the given graph computation. After visiting all the neighbors, another function produces a boolean predicate (line 8) which signals the update of the global vertex value and the activation of the vertex's outgoing neighbors. Essentially, the procedure utilizes these predicates created inside the kernel to determine the propagation of computation.

In this procedure, the neighbor visitation and reduction routine (lines 4 to 7) dominate the kernel's execution time; thus, its SIMD parallelization strategy for the GPU environment is of great importance. Once having introduced the notion of vertex activeness, the main challenge of parallelization is *load balancing*. Vertices assigned to threads inside a warp may or may not be active, which makes the set of to-be-visited adjacency lists in C array disjoint. Please note that work-inefficient schemes do not have this issue. In

```
input :Graph G
1  while G is not converged do
2      foreach v_active ∈ G.V_active do
3          v_local = LocalVertexInitialization(v_active)
4          foreach incoming neighbor nbr_in to v_active do
5              v_partial = VisitNeighbor( nbr_in )
6              Reduce( v_local, v_partial )
7          end
8          if UpdatePredicate(v_local,v_active) then
9              v_active = v_local
10             foreach outgoing neighbor nbr_out from v_active do
11                 Activate nbr_out vertex for the next iteration
12             end
13         end
14     end
15 end
```

Figure 3.5: Iterative Vertex-Centric Graph Processing.

Warp Segmentation [41], for example, threads inside the warp that are assigned to process consecutive vertices access consecutive locations in R array (Figure 3.2b) and read the neighbor indices placed consecutively in C array. Since all the vertices are supposed to be processed at every iteration, adjacent locations are being accessed in R and C arrays which makes it trivial to avoid intra-warp thread underutilization. In addition, employing simpler static assignment approaches [30, 34] makes the kernel susceptible to load imbalance in presence of irregular graphs. Hence an effective assignment strategy needs to be devised.

Next we present a *dynamic thread assignment* technique to overcome the above challenge. In our method, threads inside the warp iterate over a packed view of the neighbors belonging to different vertices . Therefore, in each round, each thread inside the SIMD group is assigned to visit and process one neighbor, avoiding intra-warp load imbalance and leading to a sustained high warp execution efficiency.

Figure 3.6 illustrates the implementation details of our dynamic thread assignment technique via an example. First, the number of neighbors that need to be visited is calculated for active vertices – threads inside the warp compute the intra-warp binary

V: V_0 V_1 V_2 V_3 R: 0 1 3 6 7 C: n_0 n_1 n_2 n_3 n_4 n_5 n_6

1	`vIdx`	0	1	2	3
2	`nbrSize = R[ vIdx+1 ] - R[vIdx ]`	1	0	3	0
3	`loc = Intra-warp binary scan( active? )`	0	1	1	2
4	`ps = Intra-warp prefix sum( nbrSize )`	1	1	4	4
5	`totalLoad = shfl( ps , warpSize-1 )`	4	4	4	4
6	`Shared Ra[ loc ] = R[vIdx ]`	0	3		
7	`Shared loadPS[ loc ] = ps - nbrSize`	0	1	∞	∞
8	`eVirIdx = { 0, 1, . . ., totalLoad-1 }`	0	1	2	3
9	`vLoc = binary search( eVirIdx, loadPS )`	0	1	1	1
10	`adjLstOffset = Ra[ vLoc ]`	0	3	3	3
11	`adjLstIdx = virIdx - loadPS[ vLoc ]`	0	0	1	2
12	`nbrIdx = C[ adjLstOffset + adjLstIdx ]`	n_0	n_3	n_4	n_5

Figure 3.6: A simplified example demonstrating our dynamic thread assignment strategy.

prefix sum [31] using the vertex activeness binary predicate (line 3) so that threads with active vertices realize their relative location between similar threads inside the warp. Then threads perform an intra-warp inclusive prefix sum [86] using the number of neighbors for active vertices (line 4), and shuffle the last lane's element to determine the total number of neighbors to visit (line 5). We also utilize two on-chip shared memory buffers to collect the row offset array elements and the exclusive prefix sum of number of neighbors both for active elements (lines 6 and 7).

Now, by knowing the total number of neighbors, threads can iterate over the loads without underutilization. The iteration index can be considered as a virtual neighbor index that has to be mapped to an actual neighbor inside the loop. Threads identify the active vertex location corresponding to their assigned virtual index using a binary search over the buffer containing the prefix sum of the number of neighbors (line 9), and figure out the adjacency list element index and offset inside C array (lines 10 and 11) using the information collected inside the shared memory. Therefore, threads find their assigned neighbors and continue the computation which involves a compute function with the neighbor content and reducing it with other neighbors. This technique essentially enables a one-to-one mapping between a thread and the neighbors of an active thread thus avoiding thread underutilization. The same scheme is used for activating a vertex's outgoing neighbors too.

3.3 Permissive Partitioning for Inter-GPU Load Balance

In the approach proposed in Section 2.3 as well as in existing multi-GPU graph processing solutions [94, 24], in order to distribute the computation across multiple GPUs, the graph is statically partitioned and each partition is assigned to one GPU. To maximize the allowable size of the input graph, a partition that is assigned to a GPU stays on the device throughout the iterative graph processing duration. However, this scheme can lead to inter-device load imbalance and GPU idling. This is because as the iterations go forward, the set of active vertices constantly change. In an iteration, a GPU that holds a partition of the graph and is assigned to process vertices in that particular partition may contain only a few active vertices and edges, while another GPU may end up with a considerably high number of active graph components. To illustrate this argument, in Figure 3.7 we have plotted the number of active edges for three devices with static partitioning across iterations for two graph algorithms. The deviation between the number of active edges across devices during iterations is clear from these plots. This issue can potentially introduce device underutilization, since, assuming similar GPU characteristics, all the GPUs have to wait for the GPU with the most active graph components to finish its job in an iteration. Essentially in work-efficient solutions a GPU's load is mainly determined by the *activeness* of vertices.This motivates the need for our technique: *permissive partitioning*.

In *permissive partitioning* we allow graph partitions being held by GPUs to *overlap* as much as device memories allow – see Figure 3.8. This enables us to dynamically slide the borders using which the GPUs determine the corresponding graph partitions they compute. The direction and the amount by which borders slide are determined on-the-fly during the iteration by observing the distribution of the active vertices in the bitmask. If a device in the current assignment has considerably higher number of active vertices and edges compared to other devices, the border that specifies its computation region is shrunk so that all devices have possibly equivalent or at least closer number of active graph components. Therefore,

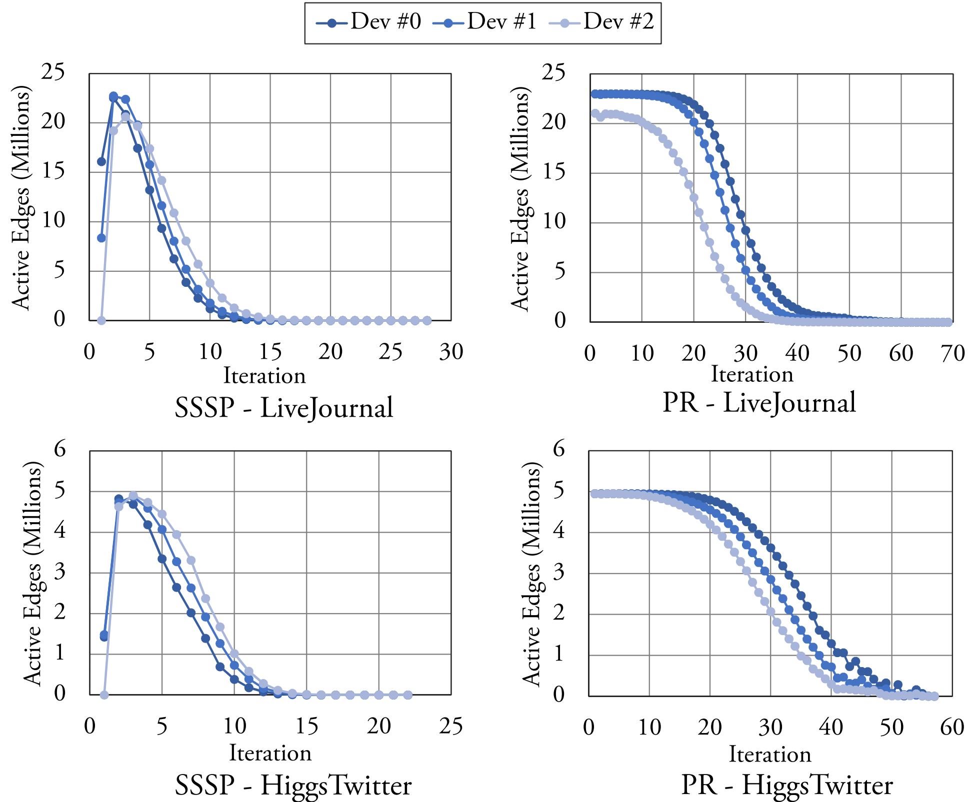


Figure 3.7: The distribution of active edges on 3 GPUs across iterations in WS-VR [41].

it provides a more balanced load and thus diminishes device underutilization. We employ *permissive partitioning* with the following *offline* and *online* phases.

Offline Phase

After estimating number of edges per device (as in static partitioning), we query the devices' available DRAM and then expand the initial static vertex-range estimate to use the additional device memory available. We maximize the usage of GPU's DRAM by storing as much of the graph data as possible. We also count the average number of edges per vertex for every 128 consecutive vertices, the same size as the thread-block dimension in our solution [1]. The resulting array is used in the *online* phase described next.

[1]Since our kernels are warp-centric, 128 as the thread-block size gives the kernel full theoretical occupancy, and simultaneously, minimizes the inter-warp load imbalance.

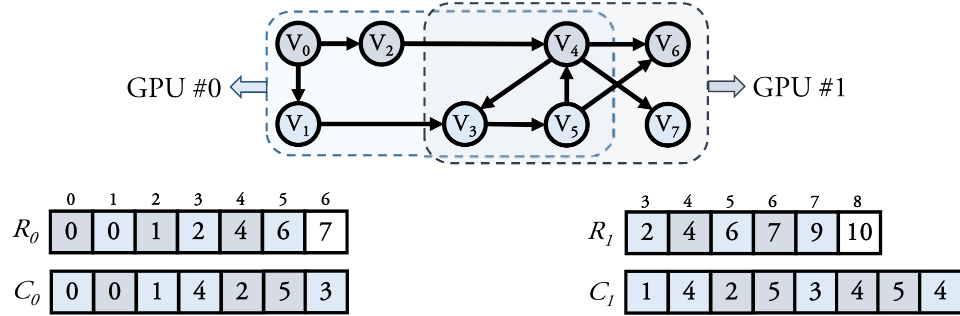


Figure 3.8: An example of *permissive partitioning* using the graph in Figure 3.2a and the resulting data structures for each device.

Online Phase

At the beginning of each iteration of the iterative graph processing procedure, the number of active vertices in every group of 128 vertices (the same size as the sections in the *offline* phase) are counted by the GPU. Counting kernel utilizes the already existing bitmask and native population count primitive. Our profiling shows that GPUs spend less than half a percent of their aggregated kernel execution time on this counting kernel. The kernel writes the results directly into a pre-allocated host pinned buffer. After enqueuing all the commands to GPUs, the host waits for the events associated with the bitmask counting kernel. When this kernel ends, the host calculates an array containing the prefix sum of the element-wise multiplication outcome of active number of vertices and the average number of edges per vertex in that section. When this array is created, the total approximate number of active edges can be found at the end of the array, and therefore, the fair amount of edges per device that balances load can be easily calculated. The host discovers the regions standing for the fair loads by binary searching the constructed array– which has a logarithmic time complexity. For each device, the host slides the border based upon the fair load value. If the load falls outside of the vertex region held by the device, the end of the region becomes the assignment border for the next iteration. Using *permissive partitioning*, at every iteration each device receives an approximately equal– or at least fairer– number of active edges. Note that since the bitmask counting kernel is enqueued earlier than other

commands, the host and the devices execute concurrently. Our experiments show that the host carries out its computation much sooner than when the devices finish the iteration; hence, the devices are not blocked.

3.4 KiTES and its Interface

We packaged the introduced techniques of our solution in a CUDA C++11 template library named KiTES. This design maximizes the solution's portability and ease-of-use. KiTES abstracts away the complexities of dealing with CUDA API for data structure management and command buffer orchestration for GPUs, as well as the subtleties of GPU kernel development. KiTES gets compiled by NVCC alongside the code that includes and uses it therefore the developer need not worry about installing the framework or satisfying dependencies.

We illustrate the simplicity of its usage via an example. Figure 3.9 shows an actual sample code utilizing KiTES to run SSSP algorithm on the requested graph using 3 GPUs. Inside the `main` function, the user instantiates a CSR graph constructed from the specified edge-list (lines 6-10). The property of the graph and the preprocessing function for the graph are also supplied during the construction. The template arguments determine the types of the vertex and the edge used to hold the graph and also the storage space for the graph. In this example, the final form of the CSR graph will be collected inside the host pinned memory.

Lines 11 to 18 define four device functions with lambda expressions [2] for: vertex initialization, neighbor visitation, partial value reduction, and update predicate creation. These device functions describe the procedure to perform on a vertex during an iteration so as to carry out SSSP iteratively. Line 19 in Figure 3.9 specifies the GPUs with IDs 0, 1, and 2 as a device group. KiTES allows the user's code to determine the processing platforms as a combination of all or a number of processing devices. At line 20 the processing function

[2] CUDA 7.5 and 8.0 allow *GPU lambdas* – device function objects that are defined in the host code.

```
1  #include <kites.cuh>
2  #include "pre_process_func_def_dec.hpp"
3  int main(){
4    using vT = uint;
5    using eT = uint;
6    kites::graph_csr<vT, eT, kites::mem::pinned> grph(
7     kites::input_graph_form::edge_list_s_d,
8     "graph_edgelist.dat",
9     kites::graph_property::directed,
10    pre_process_func_per_edge_sssp<vT, eT> );
11   auto vInitF  = [] __device__ ( volatile vT& locV,
12    vT& glob ) { locV = globV; };
13   auto nbrVstF = [] __device__ ( volatile vT& parV,
14    vT& nbrV, eT& connE ) { parV = nbrV + connE; };
15   auto parRedF = [] __device__ ( volatile vT& lV,
16    volatile vT& rV ) { if( lV > rV ) lV = rV; };
17   auto upPredF = [] __device__ ( volatile vT& newV,
18    vT& oldV ) { return newV < oldV; };
19   kites::device::nv_gpu devs{ 0, 1, 2 };
20   kites::process< kites::launch::async >(
21    grph, devs, vInitF, nbrVstF, parRedF, upPredF );
22   kites::sync_with( devs );
23   kites::io::output( devs, grph, "outVertices.dat" );
24   return 0; }
```

Figure 3.9: A sample use of KiTES to execute user-defined SSSP with 3 GPUs.

is called by specifying the graph object, device object, and the GPU device functions. This essentially means that the specified graph will be processed by the set of GPUs specified as the compute device.

Behind the scene, the library transfers the graph data into GPUs' DRAM and organizes the graph processing procedure while utilizing specified device functions iteratively. The function call is specified as asynchronous to the host in the template argument, which makes the call non-blocking. After submitting the process request, the host can synchronize with the devices and output the vertex content into a file (or alternatively another buffer or vector). As you can see, KiTES has effectively hidden the complexity of the design including the scheduling of commands to GPUs, kernel execution details, data transfer, buffer allocation and deallocation, etc.

Input Graph	$\lvert V \rvert$	$\lvert E \rvert$	$\lvert CSR \rvert$	$\lvert CSR \rvert + \lvert CSC \rvert$
Large				
RMD33V330E	33.0M	330.0M	1584-3036	3036-4356
RMD30V250E	30.0M	250.0M	1240-2360	2360-3360
RMD25V250E	25.0M	250.0M	1200-2300	2300-3300
Medium				
RMU33V330E	33.0M	330.0M	1584-3036	1584-3036
ComOrkut [90]	3.0M	234.3M	962-1912	962-1912
RMD25V200E	25.0M	200.0M	1000-1900	1900-2700
Small				
LiveJournal [4]	4.8M	68.9M	315-610	610-886
SocPokec [80]	1.6M	30.6M	136-265	265-387
HiggsTwitter [18]	0.4M	14.8M	63-124	124-184
RoadNetCA [51]	1.9M	5.5M	38-68	38-68
WebGoogle [51]	0.8M	5.1M	27-51	51-72
Amazon0601 [49]	0.4M	3.2M	16-32	32-45

Table 3.2: Graphs for single-GPU evaluations and their representation sizes (in MB). For undirected graphs, CSC size is zero.

3.5 Evaluation of Techniques

This section presents performance evaluation of introduced techniques in two parts. The first part focuses on the performance of the solution on a single GPU and the second part analyzes the techniques when multiple GPUs are employed.

3.5.1 Single-GPU Performance Analysis

In this section, we present an analysis of our solution's performance on a single GPU. We use the graphs listed in Table 3.2. Graphs with *RMD* and *RMU* prefixes are directed and undirected Rmat [11] graphs respectively, created with parameters $a = 0.45$, $b = 0.25$, and $c = 0.15$ while the rest are real-world graphs from SNAP dataset [50].

To compare the performance of our solution with Warp Segmentation's [41] and CuSha's [42], we categorize the graphs in Table 3.2 into small, medium, and large groups. For small graphs, all the graph components reside in the device memory in all three methods. For medium graphs, CuSha has to put the graphs inside the host pinned memory. Finally,

Input Graph		BFS	CC	CS	HS	NN	PR	SSSP	SSWP
LiveJournal	KT	130	122	329	311	1631	1110	297	672
	WS	172	154	535	346	2061	2326	446	772
	CW	183	187	378	382	2116	1129	386	631
SocPokec	KT	65	60	114	170	216	603	171	177
	WS	75	66	121	226	464	1145	194	194
	CW	69	67	126	184	183	363	133	127
HiggsTwitter	KT	58	54	128	72	143	559	133	77
	WS	48	37	117	75	159	483	100	77
	CW	71	85	152	123	350	574	165	156
RoadNetCA	KT	71	211	1477	43	142	62	124	350
	WS	386	330	1694	41	193	55	465	1077
	CW	238	374	1666	37	196	66	341	927
WebGoogle	KT	29	29	39	23	43	71	43	82
	WS	41	36	61	15	84	109	63	108
	CW	24	30	48	26	59	65	34	61
Amazon0312	KT	18	23	109	71	34	45	34	39
	WS	19	18	278	84	45	46	38	41
	CW	20	26	85	76	41	45	37	49
Avg. Speedup	WS	1.87	1.11	1.48	1.05	1.53	1.38	1.62	1.45
	CW	1.50	1.37	1.09	1.18	1.42	0.93	1.32	1.39

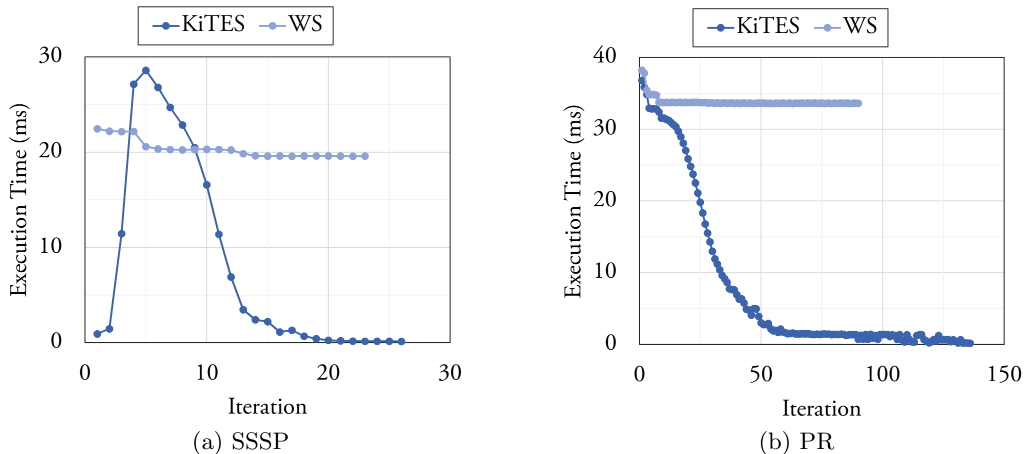
Table 3.3: Raw execution time (ms) of KiTES (KT) in comparison with Warp Segmentation's (WS) [41] and Concatenated Windows's (CW) from CuSha [42] when all the graphs reside inside the GPU's global memory.

large graphs are directed graphs for which KiTES applies vertex grouping to fit the graph inside the device memory.

Small Graphs

Table 3.3 presents the execution time provided by our method in comparison with Warp Segmentation (WS) [41] and Concatenated Windows (CW) from CuSha [42] framework. All the graphs are inside the GPU's global memory. As you can see, KiTES speeds up the iterative procedure by up to 1.87x. When averaged across all the graphs and algorithms, the speedup of KiTES over WS and CW are 1.43x and 1.27x respectively.

To illustrate the supremacy of our solution over work-inefficient methods, we measured the WS and KiTES kernel execution times across iterations over the graph processing



(a) SSSP

(b) PR

Figure 3.10: Per-iteration kernel execution time for KiTES and Warp Segmentation (WS) for PR and SSSP on LiveJournal.

duration for PageRank and SSSP on LiveJournal graph, and plotted the results in Figure 3.10. As we can see, in both cases the execution time for WS stays approximately the same since all the vertices are being visited and all the edges are being processed in each iteration, regardless of the updates. However, in our solution the kernel execution time significantly varies across iteration since it processes only active vertices. An interesting observation is the matching patterns in Figure 3.10 and Figure 3.1 which demonstrates that the amount of time the kernel spends on the graph is a function of active graph components. In addition, in Figure 3.10 we see that the peak kernel execution time for KiTES is greater than the roughly constant WS's kernel execution time. This is due to the majority of vertices being active in those iterations such that the overhead of reading from the bitmask and writing to it exceeds the benefits it provides. However, this happens only in a few iterations while in most iterations enabling the work-efficiency is highly beneficial. Also, we can see that using WS the computation can take less iterations to converge which is a minor positive side-effect of work-inefficiency.

To further examine the effectiveness of our dynamic thread assignment strategy, we profiled the average warp execution efficiency of our kernels and compared it with WS's and

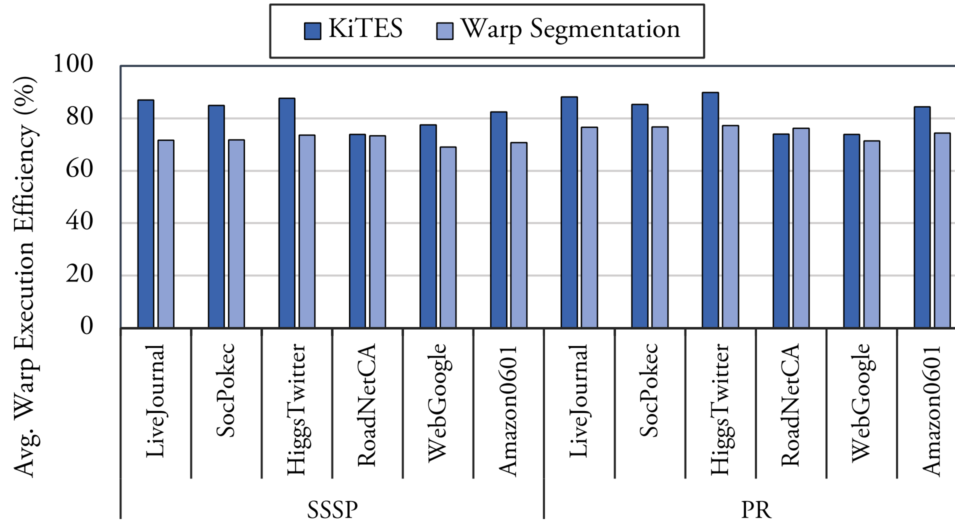


Figure 3.11: Average warp execution efficiency profiled for KiTES and Warp Segmentation for SSSP and PR.

plotted the results in Figure 3.11. It is evident that even though, unlike Warp Segmentation, the disjoint segments of the adjacency list array are being accessed, our strategy handles the introduced challenge effectively and avoids warp underutilization. Due to a longer kernel code, KiTES's profiled warp execution efficiencies are usually higher than Warp Segmentation's and are on average 82.4%.

Medium Graphs

Table 3.4 compares the execution time of KiTES with WS's and CW's for graphs for which both KiTES and WS can keep inside the GPU's DRAM but in CW the shards have to be put inside a pinned mapped host memory. Since CW's execution time in this case is heavily affected by communications across PCIe bus with much slower bandwidth, the speedup provided by our method over CW is now much bigger. When averaged, KiTES is 7.07x and 1.32x faster than CW and WS respectively.

Input Graph		BFS	CC	CS	HS	NN	PR	SSSP	SSWP
RMU33V330E	KT	793	906	1011	2889	900	3139	1612	4026
	WS	1164	1123	1530	2716	1589	5016	2476	5330
	CW	3958	3594	12852	38296	6404	26685	10672	21960
ComOrkut	KT	332	344	3212	603	751	3803	996	866
	WS	403	351	4162	681	904	4290	1398	931
	CW	2059	2074	7159	7328	4728	29387	11072	5325
RMD25V200E	KT	603	683	809	4011	616	3020	1148	2769
	WS	827	820	1019	4487	970	3810	1805	4066
	CW	2531	2247	7744	41415	3327	13754	6877	16427
Avg. Speedup	WS	1.35	1.15	1.36	1.06	1.51	1.33	1.50	1.29
	CW	5.13	4.43	8.17	11.91	6.27	6.93	7.91	5.85

Table 3.4: Execution times (ms) of KiTES (KT), Warp Segmentation (WS) [41] and Concatenated Windows (CW) [42] including host to device copy time. While WS and KiTES can fit the graph inside the GPU, CuSha must hold graphs in host pinned memory.

Large Graphs

Next, we show the performance on graphs for which our method has to apply vertex grouping to keep the graph inside device memory. KiTES enables vertex grouping with ratios 2, 4, 32, or 64, all of which are integer powers of two so as to facilitate the implementation. Our experiments revealed that vertex grouping ratios of 32 and 64 provide better speedups compared to ratios higher than 4 and lower than 32. This is because, for smaller vertex grouping ratios the overhead of dynamic thread assignment offsets the benefits of lower amounts of redundant computation. In contrast, for ratios 32 and 64 there is no need for dynamic thread assignment since all the threads inside the warp become assigned to active vertices or otherwise all retire early in the kernel. Ratios higher than 64 are not desired due to very high redundancy. During buffer allocation, KiTES prioritizes smaller vertex groups so as to provide higher performance.

Table 3.5 shows the execution time of our method KT, WS, and CW for large graphs. On average, KiTES's speedups over CW and WS are 5.97x and 1.19x respectively. As we can see, KiTES still provides superior performance over WS, although its speedup is

Input Graph		BFS	CC	CS	HS	NN	PR	SSSP	SSWP
RMD33V330E	KT	1077	898	1465	5006	1319	4340	2432	5013
	WS	1143	1103	1689	5868	1550	4907	2663	5342
	CW	3692	3474	14238	82862	6231	26131	11557	21475
RMD30V250E	KT	806	761	941	4729	956	3265	1682	4589
	WS	1064	963	1342	4788	1112	4230	2056	4816
	CW	3203	2542	10242	44050	3792	15288	7988	19697
RMD25V250E	KT	771	781	743	5404	671	3208	1355	4258
	WS	941	907	1088	4605	976	3977	1840	4770
	CW	2898	2458	8304	42044	3387	14596	6642	19128
Avg. Speedup	WS	1.20	1.22	1.35	1.01	1.26	1.22	1.23	1.08
	CW	3.72	3.45	10.59	11.22	4.58	5.08	4.80	4.36

Table 3.5: Execution time (ms) of KiTES (KT), Warp Segmentation (WS) [41] and Concatenated Windows (CW)[42] including host to device copy time. KiTES has to apply vertex grouping to fit the representation inside the GPU's DRAM.

now lower due to the some computation redundancy introduced by vertex grouping.

To further observe the effect of *vertex grouping*, we manually enabled it for one of the medium graphs, and examined the execution times with different vertex grouping ratios. Figure 3.12 presents representation size coupled with the speedup over Warp Segmentation for different vertex grouping ratios and two graph algorithms. The plot confirms the trade-off between the memory consumption and the performance enabled by *vertex grouping*. The more vertices there are inside each group, the lower is the memory consumption but more redundant computation is present that leads to a bigger performance penalty. Also, the amount of reduction in size is highly dependent on the structure of the graph. If the distribution of elements in the adjacency matrix of the graph is uniform, the adjacency matrix becomes denser after reduction lowering the reduction achieved by vertex grouping.

3.5.2 Multi-GPU Performance Analysis

Next we analyze KiTES's scalability when multiple GPUs are being deployed for the graph processing task. For experiments in this part, we utilize directed and undirected Rmat graphs with different sizes and densities presented in Table 3.6.

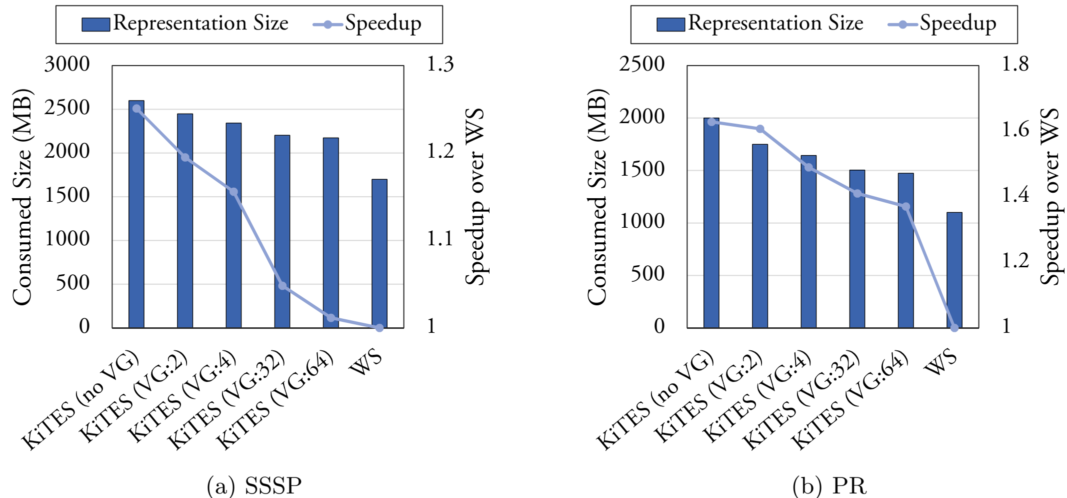


Figure 3.12: The effect of *vertex grouping* on the GPU's DRAM consumption and the performance of the procedure for two algorithms. The graph is *RMD25V200E*.

Input Graph	N. Vertices	N. Edges
RMU50V671E	50 331 648	671 088 640
RMD50V671E	50 331 648	671 088 640
RMD46V603E	46 137 344	603 979 776
RMD41V536E	41 943 040	536 870 912
RMU39V469E	39 845 888	469 762 048
RMD39V469E	39 845 888	469 762 048
RMD37V436E	37 748 736	436 207 616
RMD35V402E	35 651 584	402 653 184

Table 3.6: Graphs used for multi-GPU experiments: Top 4 graphs employed for experiments with 3 GPUs and bottom 4 graphs for experiments with 2 GPUs.

Table 3.7 gives the speedup provided by KiTES over WS-VR framework [41] (the multi-GPU version of WS) with the same configuration. This table shows the superiority of our method and also the importance of work-efficiency when scaling out to multiple processing devices. For two- and three-GPU configurations, the average speedups provided by KiTES are 1.29x and 1.37x respectively.

To better understand the effect of permissive partitioning, we plotted the speedups of KiTES over WS-VR [41] with and without *permissive partitioning* for a number of graphs

Input Graph	BFS	CC	CS	HS	NN	PR	SSSP	SSWP
RMU50V671E	1.43	1.32	1.41	1.12	1.43	1.40	1.66	1.34
RMD50V671E	1.39	1.30	1.34	1.08	1.39	1.44	1.62	1.36
RMD46V603E	1.30	1.26	1.35	1.17	1.46	1.34	1.59	1.27
RMD41V536E	1.34	1.29	1.28	1.14	1.57	1.42	1.67	1.36
RMU39V469E	1.29	1.29	1.27	1.10	1.34	1.29	1.55	1.28
RMD39V469E	1.24	1.28	1.30	1.04	1.30	1.33	1.54	1.24
RMD37V436E	1.26	1.25	1.33	1.03	1.38	1.39	1.51	1.25
RMD35V402E	1.21	1.22	1.28	1.06	1.27	1.38	1.53	1.31

Table 3.7: Speedup provided by KiTES over WS-VR [41]. Top 4 entries use 3 GPUs and bottom 4 entries use 2 GPUs.

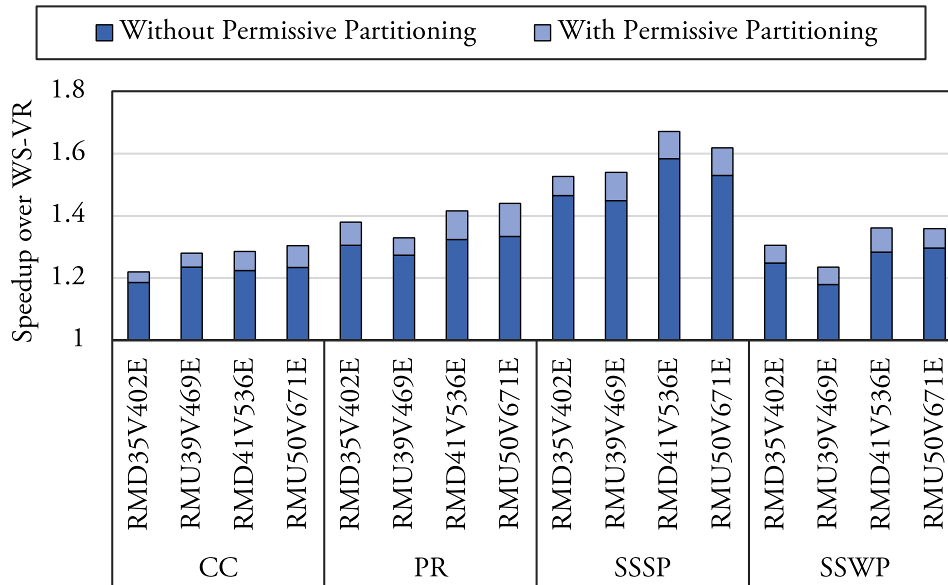
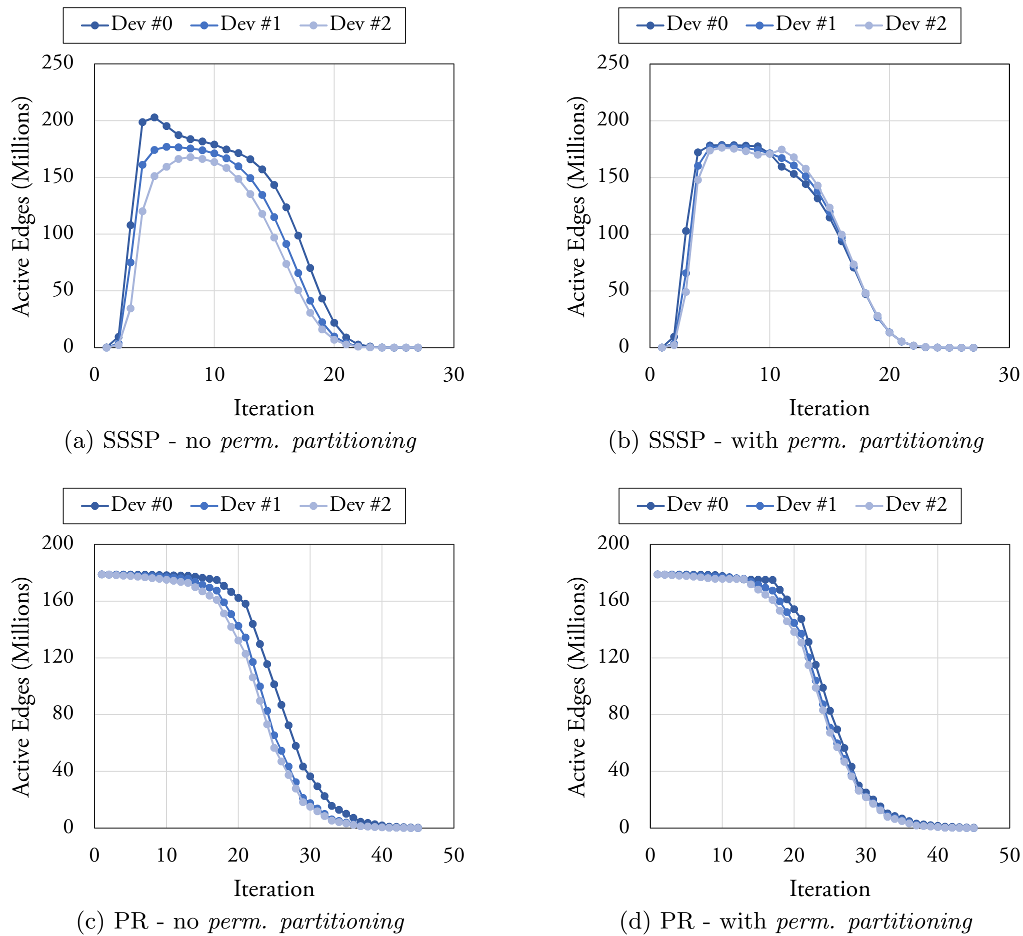


Figure 3.13: The speedup of KiTES over WS-VR in multi-GPU graph processing with and without *permissive partitioning*.



(a) SSSP - no *perm. partitioning*

(b) SSSP - with *perm. partitioning*

(c) PR - no *perm. partitioning*

(d) PR - with *perm. partitioning*

Figure 3.14: The effect of *permissive partitioning* on the distribution of active edges between 3 GPUs across iterations. *RMD41V536E* is the input graph.

and two algorithms in Figure 3.13. This not only tells the benefits of *permissive partitioning*, but also reveals how much of the multi-device performance superiority is coming from only the work-efficiency. Based on the information in Figure 3.13, for two-GPU configuration 17% of the speedup comes from *permissive partitioning* while for three-GPU configuration this percentage is slightly higher and, on average, is 20%. As more devices are deployed to carry out the graph processing task, the benefits of *permissive partitioning* rise. This is because with more GPUs, the load imbalance between multiple devices is greater and so is the need for permissive partitioning.

We also measured the distribution of the loads with and without *permissive partitioning* across iterations for two graph algorithms on a sample graph and plotted the results in Figure 3.14. As you can see, enabling *permissive partitioning* has reduced the gaps between plots belonging to different devices meaning that there is less deviation between GPU's load volumes. The small remaining gap in load volumes between devices when using *permissive partitioning* is due the delay in applying vertex assignments. Since the activeness distribution information is obtained by reading the bitmask created in the previous iteration, and the result of the calculation is going to be applicable in the next iteration, there is always one-iteration delay in the feedback. However, as we can see, this delay only mildly disrupts the balance in distribution.

3.6 Summary

This chapter presented our solution to enable work-efficiency in GPU vertex-centric graph processing by keeping track of active vertices. It employs a dynamic task assignment strategy to efficiently map threads to disjoint elements of adjacency lists during the expansion, avoiding intra-warp thread underutilization. *Vertex grouping* technique mitigates the memory consumption overhead issue and enables processing of bigger graphs on the GPU with limited DRAM size. In addition, *permissive partitioning* provides dynamic inter-GPU load balancing. The collection of these techniques are gathered in a CUDA C++ template library named KiTES to facilitate their deployment in user applications. Our experiments show that KiTES provides up to 5.46x and 1.67x speedup over the work-inefficient solution for single and multi-GPU configurations over multiple algorithms and inputs respectively.

Chapter 4

Generalizing Techniques

This chapter discusses the feasibility of extending techniques that we developed for graph analytics to other domains of GPU computing. More specifically, it introduces two generic techniques, inspired by the previously discussed graph processing methods, to enhance the SIMD execution efficiency of CUDA applications. This chapter consists of the following two sections.

The first section proposes *CCC*, a compiler technique for CUDA programs that boosts the warp execution efficiency upon divergence. CCC collects tasks at warp granularity and remedies inefficiency due to intra-warp load imbalance or dissimilar task assignment. To enhance the applicability of CCC, this section presents transformations to make common code patterns accessible to CCC and develop optimizations to increase the CCC performance. It discusses automating CCC at the CUDA PTX level via a source-to-source compiler and enbed it into a framework that operates alongside NVCC and allows CCC application by annotating loops and divergent paths in CUDA C/C++ kernels. CCC increases the warp execution efficiency of real-world applications containing divergent execution paths by up to 56% and provides speedup of up to 3.08x.

The second section of this chapter proposes *Collaborative Task Engagement* (*CTE*), a novel task decomposition technique to efficiently process irregular nested parallel patterns in GPUs. Unlike previous methods, warp threads in CTE pass over the expanded list of

fine-grained tasks, making it resilient against input irregularities. In order to abstract away the complexities of the CUDA implementation, CTE is developed as a CUDA C++ device-side template library for easy-expression of nested patterns. CTE is resilient against irregularities and provides up to 1.51x speedup over the best sub-warp decomposition width and exhibits excellent warp execution efficiency.

4.1 Collaborative Context Collection

Graphics Processing Units (GPUs) have become the essential part of high-performance and power-efficient parallel computing. Recent emergence of programming APIs such as CUDA and OpenCL have played an important role in enabling general purpose parallel computing using GPUs. A key to the success of such APIs is the freedom they provide to define different execution paths for the threads inside a SIMD group (warp or wavefront); while the underlying architecture manages the resulting complexities. To emphasize this aspect of GPUs, they are also called SIMT (Single Instruction Multiple Thread) microarchitecture devices. However, the combination of this attribute with GPUs inherent lack of fine-grained task parallelism support may result in a significant performance loss.

All the threads inside a SIMD group (i.e., all the warp lanes) execute one unique instruction at a time. The presence of conditionals—such as due to if-else code blocks—causes *thread divergence* because a conditional may evaluate to true for some warp lanes and false for other lanes. In this situation, the warp takes all the divergent paths, while disabling non-relevant lanes inside every path. That is, the warp scheduler fetches instructions for all the divergent paths while the execution stage is active only for a number of corresponding threads. As a result, a portion of the available processing power goes unutilized for the duration of divergence, diminishing the SIMD execution benefits.

Microarchitectural and compiler solutions have been suggested to remedy thread divergence. While most microarchitectural methods are based on warp formation [22, 59, 68] or warp compaction [23, 66, 82], the compiler solutions rely on the warp lanes majority vote

to gain partial warp execution enhancement [28, 13, 21]; however, full warp utilization is usually out-of-reach. The compiler based solutions require program-specific and input-specific information about divergence behavior to *speculate* on scheduling divergent code blocks and lack systematic application procedure. Other software approaches disrupt the GPU kernel autonomy by offloading data/task reordering onto CPU [92], implement global locks with heavy contention for global queues [81], or accept errors in the output by ignoring the task of minorities [74]. The above limitations of software methods make them often inapplicable and unreliable.

We propose Collaborative Context Collection (CCC), a software (compiler) technique for GPUs that enables efficient execution of warps in presence of dissimilar intra-warp task assignment or intra-warp load imbalance. Unlike previous solutions, CCC does not rely on heuristics to increase warp utilization; instead, it accumulates tasks to provide maximum warp execution efficiency. CCC utilizes the fast shared memory of the Streaming Multiprocessor (SM) to collect threads' *context* which includes the content of the thread-private registers that are sufficient to describe the thread's task inside the divergent path. *Context collection* provides all the warp lanes with homogeneous tasks and hence allows efficient warp execution. CCC is primarily aimed at removing divergence from repetitive GPU code blocks (e.g., loops). Threads in a warp are provided with a warp-specific stack in shared memory. In each iteration, if there are insufficient contexts of the divergent path in the stack to keep unemployed threads busy, warps collect their context on the stack. Otherwise, each unemployed thread grabs a context from the shared memory and all the warp lanes execute the divergent branch. This eliminates warp underutilization since the warp lanes follow the *all-or-none* principle for taking the divergent path. By collecting tasks at the warp granularity, CCC avoids need for any syncing or fencing operations. CCC exploits fast CUDA intrinsics to implement intra-warp binary reduction and prefix sum necessary for collaborative context storing and restoring.

We further present transformations to extend the applicability of CCC to many common code patterns with intra-warp load imbalance or dissimilar task assignment, such

as loops with varying or unknown trip-count and recursive functions. We also provide optimizations for CCC to increase performance in certain situations and to eliminate or reduce the CCC possible side-effects such as memory divergence or theoretical occupancy limitation. Finally, we implemented CCC in a software framework that allows annotating the repetitive patterns and the divergent paths in CUDA C/C++ kernels. Annotations are then replaced and transferred to the Nvidia CUDA Compiler (NVCC) generated intermediate PTX where CCC is applied to PTX code via our source-to-source PTX compiler. Then, the framework feeds the transformed code to the rest of the compilation chain.

The rest of this section is organized as follows. We first define the thread divergence problem and describe CCC. Then we present transformations to apply CCC or to convert common code patterns to representations accessible by CCC. We also discuss CCC optimization techniques elaborate upon CCC implementation as a compiler framework. Finally we evaluate CCC performance and analyze its sensitivity.

4.1.1 CCC Core Procedure

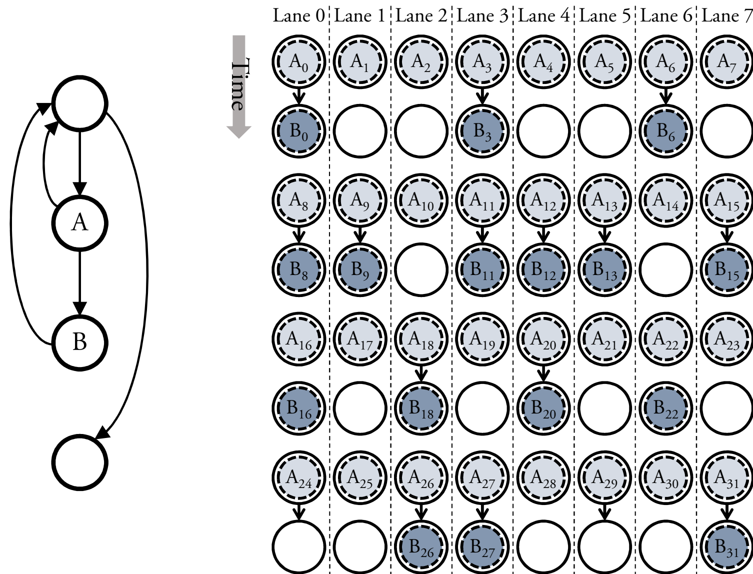
Here, we first explain the *thread divergence* problem and then discuss our solution.

Thread Divergence Problem Overview

The SIMT microarchitecture in GPUs provides a parallel processing platform that groups fine-grained threads into warps. A warp owns only one active Program Counter (PC) at a given time, allowing the hardware to schedule one instruction for execution by multiple execution units (SM cores, in terms of CUDA) using only one instruction fetch and decode. This design reduces the die size and power consumption while providing massive parallelism. However, warp lanes must run in lockstep. Therefore, specifying different execution paths for threads of a warp—also called warp lanes—results in the traversal of all the divergent branches by them. For a divergent branch, the processor masks off inactive threads while holding their reconvergence PC in a hardware divergence stack [43]. Until the warp's active PC reaches the reconvergence PC, masked off threads stay inactive; thus

```
1  __global__ void CUDA_kernel_BFS(
2  const int numV, const int curr, int* levels,
3  const int* v, const int* e, bool* done ) {
4     for(
5     int vIdx = threadIdx.x + blockIdx.x * blockDim.x;
6     vIdx < numV;
7     vIdx += gridDim.x * blockDim.x ) {
8        bool p = levels[ vIdx ] == curr; // Block A.
9        if( p )
10          process_nbrs( vIdx,
11          curr, levels, v, e, done ); // Block B.
12    } }
```

(a) The CUDA kernel for iterative BFS graph processing.



(b) Kernel CFG. (c) Resulting warp execution visualization.

Figure 4.1: An example: BFS graph processing visualization in CUDA [30].

some reserved execution units are not utilized, causing warp execution inefficiency.

Figure 4.1 illustrates the above using BFS adapted from [30]. Each thread is assigned processing of a number of vertices. If a vertex is updated in the previous CUDA kernel invocation, the thread must update vertex's neighbors. This condition leads to thread divergence as it can evaluate to true for some warp lanes and to false for others. Threads that do not execute divergent branch must wait for other threads in the warp to finish processing `block B` as illustrated in Figure 4.1c.

Boosting Warp Efficiency with CCC

To eliminate thread divergence due to imbalanced load/task assignment to warp lanes, we propose Collaborative Context Collection (CCC). CCC increases the warp execution efficiency for kernels containing repetitive diverging tasks with independent iterations. This pattern is common in GPU thread task assignment. The BFS graph processing CUDA kernel shown in Figure 4.1a matches this model and Figure 4.1b depicts the corresponding control flow graph (CFG). Later in Section 4.1.2 we introduce transformations enabling a wide variety of GPU algorithms to be expressed in this form.

In the above program model, threads inside the repetitive code block (the loop) iterate over divergent tasks. The key idea of CCC is keep collecting tasks corresponding to threads of a warp until there are tasks to keep all threads busy. To establish a connection between the divergent task and its required data, we define a **context** as the minimum set of variables that can fully describe the functionality of the task if the task is carried out by another thread. For a GPU thread, these variables are a subset of thread's registers (that are thread-private).

In CCC, at every iteration, if there are insufficient divergent tasks to keep all the warp lanes busy, then lanes that are assigned such tasks collect their context in the **context stack**. Context stack is a warp-specific shared memory region for collecting unprocessed task contexts. After stacking the contexts, the warp moves to the next iteration without entering the divergent branch. Later, if the aggregation of tasks that are stacked and the tasks assigned to the warp lanes in the current iteration exceeds the warp size, lanes without a task can grab a context from the shared memory, and the entire warp executes the divergent branch. Thus, warp lanes execution discipline upon divergence is **all-or-none** which avoids warp underutilization. Figure 4.2 demonstrates the impact of CCC on the divergent CUDA program in Figure 4.1 by visualizing the execution of a warp.

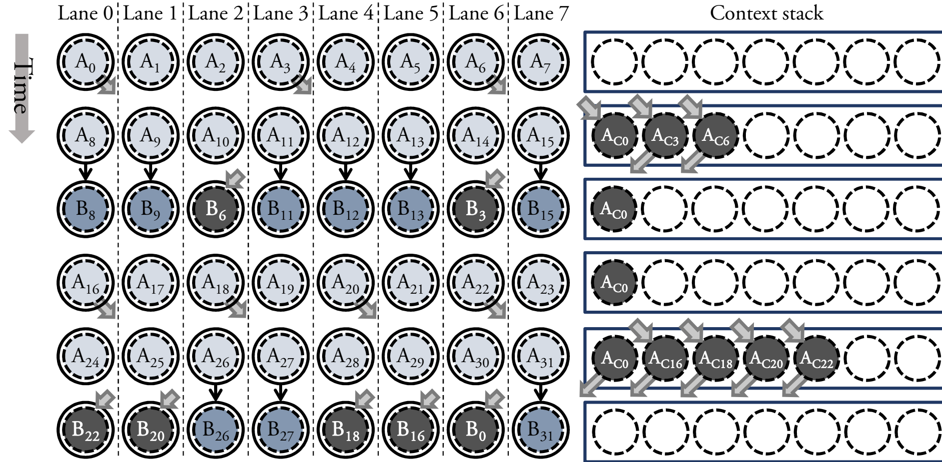


Figure 4.2: Applying Collaborative Context Collection to the program in Figure 4.1 eliminates warp execution inefficiency.

CCC's Efficient CUDA Implementation

Let us first precisely define *context*. We refer to a thread's task context as a set of its designated registers that are

1. *defined* prior to divergent task path as a function of thread-specific special registers including `%tid`, `%laneid`, and lane masks such as `%lanemask_eq`; and

2. *used* inside the divergent task path.

Note that a context may contain special registers themselves. This definition enables CCC to distinguish the minimal subset of thread's registers that need to be collected and retrieved at every iteration.

Next we describe CCC's application to the CUDA BFS kernel. Figure 4.3 shows CUDA BFS kernel in Figure 4.1a after applying CCC. The first highlighted section (lines 4-7 in Figure 4.3) is initialization of variables and the stack for CCC. The context stack consists of `vIdx` in Original CUDA code (Figure 4.1a) since it is the only variable that depends on the thread index and is used in the divergent region. CCC uses shared memory —the fastest memory after thread-private registers— for collecting contexts. The stack is marked `volatile` to pass data between warp lanes without any need for adding synchronization

```
1  __global__ void CUDA_kernel_BFS_CCC(
2  const int numV, const int curr, int* levels,
3  const int* v, const int* e, bool* done ) {
4     volatile __shared__ int cxtStack[ CTA_WIDTH ];
5     int stackTop = 0;
6     int wOffset = threadIdx.x & ( ~31 );
7     int lanemask_le = getLaneMaskLE_PTXWrapper();
8     for(
9     int vIdx = threadIdx.x + blockIdx.x * blockDim.x;
10    vIdx < numV;
11    vIdx += gridDim.x * blockDim.x ) {
12       bool p = levels[ vIdx ] == curr; // Block A.
13       int jIdx = vIdx;
14       int pthBlt = __ballot( !p );
15       int reducedNTaken = __popc( pthBlt );
16       if( stackTop >= redNTaken ) { // All take path.
17          int wScan = __popc( pthBlt & lanemask_le );
18          int pos = wOffset + stackTop - wScan;
19          if( !p ) jIdx = cxtStack[ pos ]; // Pop.
20          stackTop -= reducedNTaken;
21          process_nbrs( jIdx,
22          curr, levels, v, e, done ); // Block B.
23       } else { // None take path.
24          int wScan = __popc( ~pthBlt & lanemask_le );
25          int pos = wOffset + stackTop + wScan - 1;
26          if( p ) cxtStack[ pos ] = jIdx; // Push.
27          stackTop += warpSize - reducedNTaken; } } }
```

Figure 4.3: Applying CCC on the BFS CUDA kernel in Figure 4.1a.

or fencing primitives. `volatile` qualifier inhibits unsolicited optimization of references and enforces sequential consistency between the threads of a warp accessing the shared memory. Also, all the threads initialize the context stack top to zero indicating no context has been collected yet.

The second highlighted region in Figure 4.3 (lines 14-15) is executed in every iteration. Threads count the total number of lanes for which the predicate for taking divergent path is false. This is the total number of lanes inside the warp that will be idle during the divergent branch in the original program. If the result is less than or equal to the number of stacked contexts, it means all unemployed lanes can restore context, i.e., the warp can take divergent path without underutilization (starting from line 17). Otherwise, full warp utilization is not possible by taking the divergent path; hence, threads with task collect their context into the stack (starting from line 24) and the warp moves to the next

iteration without executing the divergent branch. In the third highlighted section (lines 17-20 in Figure 4.3) unemployed threads calculate the stack index from which they pop contexts. After popping, all the threads move down the stack. Note that it is necessary to check the predicate before popping the stack (line 19) since only those warp threads that do not have any task to perform should grab an existing context. Those that have a task simply execute the task context they already hold. The fourth highlighted section (lines 24-27) is the counterpart of the third section for pushing contexts. Finally, there is another section (not shown) for executing leftover task contexts in the stack after finishing all the iterations (such section is present in every program discussed).

For simplicity, the stack size (number of elements) for each warp is made the same as the warp size, although the number of stacked contexts will never exceed warp size minus one because if it does, it means that in a previous iteration all the warp threads could have been utilized but were not. Note that CCC requires iterations of the repetitive pattern to be independent of each other so the reordering of iterations preserves program semantics. Therefore, barriers and memory fences, as long as they do not disrupt this feature, can be used in the iterative code segment. Finally, if a register is written inside the divergent path and the write operation can be expressed in form of a associative reduction function that has an atomic operation counterpart, to apply CCC, the register needs to be transferred to a shared memory buffer. Accesses to the register will be replaced with accesses to the corresponding shared memory buffer; especially the writes inside the divergent path should be made atomically.

The key methods that make CCC feasible yet fast are:

- counting the total number of warp lanes with the false predicate–this is a form of intra-warp binary reduction; and
- realizing the stack position to/from which a thread needs to store/restore the context–this is a stream compaction problem that we solve using inclusive intra-warp binary prefix sum (scan).

We employed Harris and Garland's methods [31] for both intra-warp binary reduction and scan. Both methods utilize `__popc()` and `__ballot()` CUDA intrinsics and translate into very few binary operations.

4.1.2 CCC Transformations

To provide unique tasks for all the warp lanes, CCC requires repetition over the divergent code block. However, a GPU kernel in its original form may not expose this pattern readily. Therefore to extend the applicability of CCC, in this section, we provide transformations to enable application of CCC to many common forms of GPU kernel patterns.

Task Repetition with Grid-Stride Loops

It is a well known software technique to launch the GPU kernel with exactly enough threads so that all the Streaming Multiprocessors are occupied. In this technique, threads inside the GPU kernel iterate over assigned tasks using a loop. The BFS CUDA code in Figure 4.4a employs this technique in order to allow a thread to iterate over multiple vertices. Figure 4.4 shows an example of such transformation and required changes in the host and the device code. Enabling task repetition with *grid-stride loops* is similar to persistent threads technique introduced in [2] in which enough residing GPU threads are invoked inside the kernel. However, in a grid-stride loop, the assignment of tasks to threads is predetermined rather than being controlled by a shared queue.

A grid-stride loop can transform a kernel with divergence into a form suitable for CCC. Beside this, it usually exhibits two other benefits that increase the performance. First, it eliminates the overhead associated with scheduling new thread-blocks since the same thread-blocks that are invoked initially do the whole computation by iterating over the tasks. Second, it lowers the inter-warp load imbalance effect. A thread-block is not finished until all its associated warps terminate. Load imbalance between different warps of

```
1  __global__ void CUDA_kernel_BFS(
2  const int numV, const int curr, int* levels,
3  const int* v, const int* e, bool* done ) {
4     int vIdx = threadIdx.x + blockIdx.x * blockDim.x;
5     if( vIdx < numV ) {
6        bool p = levels[ vIdx ] == curr;
7        if( p )
8           process_nbrs( vIdx,
9           curr, levels, v, e, done ); } }

10 int main() { // Host side program.
11    . . . .
12    int gridD = ceil( numV / blockD );
13    gpuKernel <<< gridD, blockD >>> // Kernel launch.
14    ( numV, kernelIter, lev, v, e, done );
15    . . . . }
```

(a) Before transformation.

```
1  __global__ void CUDA_kernel_BFS_with_gridstride_loop(
2  const int numV, const int curr, int* levels,
3  const int* v, const int* e, bool* done ) {
4     for(
5     int vIdx = threadIdx.x + blockIdx.x * blockDim.x;
6     vIdx < numV;
7     vIdx += gridDim.x * blockDim.x ) {
8        bool p = levels[ vIdx ] == curr;
9        if( p )
10          process_nbrs( vIdx,
11          curr, levels, v, e, done ); } }

12 int main() { // Host side program.
13    . . . .
14    int gridD = nSMs * maxThreadsPerSM / blockD;
15    gpuKernel <<< gridD, blockD >>> // Kernel launch.
16    ( numV, kernelIter, lev, v, e, done );
17    . . . . }
```

(b) After transformation.

Figure 4.4: A grid-stride loop applied to the BFS CUDA kernel in order to make it accessible by CCC. The maximum theoretical occupancy is assumed 100%.

one thread-block may cause one warp to hold hostage the entire thread-block and disallow further thread-block scheduling. This issue is also referred to as *warp-level divergence* [89]. Grid-stride loops greatly reduce this effect by allowing assigned tasks to be processed by warps that are done with an iteration. The cost of a grid-stride loop is using a register to keep the status of the loop iteration index.

Loops with Variable Trip-Count

A common load assignment pattern is to assign a coarse-grained task to each GPU thread via a loop. In this pattern, the loop trip-count determines the amount of fine-grained tasks, i.e. the load volume, for the thread. The thread iterates over the load until it is done. This approach is specifically prevalent in GPU graph processing (pioneered in [30]) in which threads are assigned to graph vertices and process all the vertex's neighbors using a loop. Figure 4.5a exhibits this assignment pattern. Although this load assignment strategy provides code readability, it can introduce heavy load imbalance and warp execution inefficiency. For instance, in case of GPU graph processing, since different vertices can have very different number of neighbors, different threads have to iterate different number of times over the loop. In real-world power law graphs the load imbalance can be especially high.

Loops with variable trip-count can be expressed in form of a loop with uniform trip-count containing a divergent path, and hence, benefit from CCC similar to the example in Figure 4.1. Figure 4.5b depicts this transformation applied onto the BFS CUDA device function shown in Figure 4.5a. First, warp lanes reduce the largest trip-count using the butterfly shuffle instruction (as in [86]), and select the resulting value as the uniform trip-count. Then, the code block inside the loop is wrapped by a condition check that verifies if the iteration is less than the thread's original trip-count.

```
1  __device__ void process_nbrs(
2  const int vIdx, const int curr, int* levels,
3  const int* v, const int* e, bool* done ) {
4     int eIdx = v[ Idx ];
5     int nNbrs = v[ vIdx + 1 ] - eIdx;
6     for( int nbrIdx = 0; nbrIdx < nNbrs; ++nbrIdx )
7        process_nbr( curr, eIdx + nbrIdx,
8        levels, v, e, done); }
```

(a) Before transformation.

```
1  __device__ void process_nbrs(
2  const int vIdx, const int curr, int* levels,
3  const int* v, const int* e, bool* done ) {
4     int eIdx = v[ Idx ];
5     int nNbrs = v[ vIdx + 1 ] - eIdx;
6     int UniCount = intra_warp_reduce_max( nNbrs );
7     for( int nbrIdx = 0; nbrIdx < UniCount; ++nbrIdx )
8        if( nbrIdx < nNbrs )
9           process_nbr( curr, eIdx + nbrIdx,
10          levels, v, e, done); }
```

(b) After transformation.

Figure 4.5: An example demonstrating the transformation of a CUDA device function (BFS processing of a vertex's neighbors) with variable trip-count to a form accessible by CCC.

Recursive Device Functions

CUDA allows recursion on device functions that provides intuitive ways to express algorithms. Cuckoo hashing on GPU [3] is one example as shown in Figure 4.6a. In cuckoo hashing a key-value pair is provided with multiple hash functions. On every insertion endeavor, the pair is inserted into the bucket pointed to by one of hash functions using an atomic exchange. The return value of the atomic operation yields the content of the bucket right before insertion. If this content holds another key-value pair, the returned pair has to be rehashed with an altered hash function. The recursion of insertion endeavor goes on until the returned bucket had been empty.

While some threads might be successful inserting their key-value pair into the table in the very first try, other threads in the same warp might take the divergent path over and over again causing overall warp underutilization. CCC can be applied to recursive device functions as well to eliminate the load imbalance. Figure 4.6b shows the resulting

```
1  __device__ bool try_insert(
2  ulonlong* pos, ulonglong& kvp ) {
3     kvp = atomicExch( pos, kvp );
4     return ( uint )( kvp >> 32 ); }

5  __device__ void insert_KVPair( ulonglong kvp,
6  uint loc, const uint tSize, ulonglong* table ) {
7     uint retKey = tryInsert( table + loc, kvp );
8     bool p = retKey != EMPTY_KEY;
9     if( p ) {
10       loc = find_next_location( retKey, loc, tSize );
11       insert_KVPair( kvp, loc, tSize, table ); } }

12 __global__ void generate_hash_table(
13 const int nKVPairs, const uint tableSize,
14 const ulonglong* kvpairs, ulonglong* table) {
15    for(
16    int eIdx = threadIdx.x + blockDim.x * blockIdx.x;
17    eIdx < nKVPairs;
18    eIdx += blockDim.x * gridDim.x ) {
19       ulonglong kvp = kvpairs[ eIdx ];
20       uint key = ( uint )( kvp >> 32 );
21       uint loc = hash_func( key, tableSize, 0 );
22       insert_KVP( kvp, loc, tableSize, table ); } }
```

(a) Before transformation.

```
1  __device__ void insert_KVPair_CCC( ulonglong kvp,
2  uint loc, const uint tSize, ulonglong* table ) {
3     uint retKey = tryInsert( table + loc, kvp );
4     bool p = retKey != EMPTY_KEY;
5     int redNtaken = intra_warp_binary_reduce( !p );
6     if( stackTop >= redNtaken ) {
7        pop( p, retKey, loc, kvp );
8        loc = find_next_location( retKey, loc, tSize );
9        insert_KVPair( kvp, loc, tSize, table );
10    } else {
11       push( p, retKey, loc, kvp ); } }
```

(b) After transformation. Operations and variables related to context stack are shortened for brevity.

Figure 4.6: An example demonstrating the transformation of a recursive CUDA device function (*cuckoo hashing on GPU* [3]) by CCC.

code after applying CCC. Before taking the divergent path, which contains the call to the recursive function, warp lanes count the predicates (line 5), and take the divergent path if all of them can be fully utilized. Otherwise, threads that have to call the recursive function stack their contexts, and the warp exits the function to grab fresh key value pairs and repeat the procedure.

Similar to the previous transformation shown, warp lanes discipline for calling the recursive function is *all-or-none* and thus enabling maximum warp utilization. Note that this solution focuses on *single recursion* where the recursive function contains a single call to itself. Having multiple references to itself (*multiple recursion*) and also dynamic parallelism are task generation problems that are beyond the scope of this work.

Loops with Unknown Trip-Count

```
1  __device__ void insert_KVPair_noRec( ulonglong kvp,
2  uint loc, const uint tSize, ulonglong* table ) {
3     uint retKey = tryInsert( table + loc, kvp );
4     bool p = retKey != EMPTY_KEY;
5     while( p ) {
6        loc = find_next_location( retKey, loc, tSize );
7        retKey = tryInsert( table + loc, kvp );
8        p = retKey != EMPTY_KEY; } }
```

(a) Before transformation.

```
1  __device__ void insertKVPair_noRec_CCC( ulonglong
2  kvp, uint loc, const uint tSize, ulonglong* table ) {
3     uint retKey = tryInsert( table + loc, kvp );
4     bool p = retKey != EMPTY_KEY;
5     int redNtaken = intra_warp_binary_reduce( !p );
6     while( stackTop >= redNtaken ) {
7        pop( p, retKey, loc, kvp );
8        loc = find_next_location( retKey, loc, tSize );
9        retKey = tryInsert( table + loc, kvp );
10       p = retKey != EMPTY_KEY;
11       redNtaken = intra_warp_binary_reduce( !p ); }
12    if( p ) push( p, retKey, loc, kvp ); }
```

(b) After transformation. Operations and variables related to context stack are shortened for brevity.

Figure 4.7: Transformation of a loop with unknown trip-count (*Cuckoo Hashing on GPU* [3]) by CCC.

The transformation for recursive functions naturally extends to the loops with unknown trip-count. For example, codes in Figures 4.7 are functionally equivalent counterparts of recursive GPU hashing functions in Figures 4.6 written using a `while` loop. Similar to recursive functions, warp lanes count the predicates before taking the path (Figure 4.7b line 5). Then inside the `while` loop, taskless lanes pop contexts (line 7), all the warp lanes perform the divergent task, and count the predicate again (line 11). Threads of a warp stay inside the loop as long as there are enough tasks to keep them all busy. When the warp gets out of the loop, lanes that hold task push the contexts into the stack.

Again, all the warp lanes either get inside and stay inside the `while` loop, or all of them get out, respecting the *all-or-none* discipline.

Nested and Multi-Path Context Collection

CCC can be applied in a nested manner to a divergent path containing intra-warp divergence. In this scenario, a separate stack collects the context of the parent divergent path and another stack collects the child's.

An example of this scenario is again CUDA BFS kernel. While neighbors of a vertex are visited only when it is updated in the previous GPU kernel invocation (Figure 4.1a line 9), variable trip-count for the inner loop (Figure 4.5a line 6), due to irregularity of the graph, creates load imbalance inside the divergent path. For this problem, nested CCC collects the context for the outer and the inner divergent paths independently, and executes each path only when enough contexts of that particular path exist.

Now consider the example of Iterated Function Systems (IFS) inside CUDA kernels. Figure 4.8 presents a device function in Fractal Flames GPU program [77] in which each thread executes a random function variation. For such cases, multi-path CCC assign a separate context stack to each task. Threads collect the contexts for each divergent branch separately, and warp lanes execute a path only if enough contexts of a certain task, that can provide full warp efficiency, are available.

```
1  __device__ cuFloatComplex variation_gen(
2  const float x, const float y, const uint var ) {
4     switch( var ) {
5     case 0: // Linear variation.
6        return make_cuFloatComplex( x, y );
7     case 1: // Sinusoidal variation.
8        return make_cuFloatComplex( sinf(x), sinf(y) );
9     . . . .
10    case 7: // Power variation.
11       float theta = atanf( x / y );
12       float len = sqrtf( x * x + y * y );
13       float sinTh = sinf( theta );
14       float mul = powf( len, sinTh );
15       float r = mul * cosf( theta );
16       float i = mul * sinf( theta );
17       return make_cuFloatComplex( r, i ); } }
```

Figure 4.8: Variation generation CUDA device function in Fractal Flame [77] from Iterated Function System (IFS) class.

4.1.3 CCC Optimizations

In this section, we discuss techniques that can be implemented on top of CCC to further optimize the performance in certain situations.

Context Compression: Reducing Context Storage and Context Saving/Restoring Overhead

If in a context, a register value can be computed from another register's value with a computationally inexpensive operation(s), i.e. one can be expressed as a trivial compute-only function of another, only one of them needs to be collected during storing. For the restoration, one register content is then derived from the other one using the function. This optimization reduces shared memory consumption and lowers storing/restoring overhead.

An example can be found in Figure 4.6 where the context includes three variables: `kpv`, `loc`, and `retKey`. Since `retKey` can be recomputed from `kvp` using only a shift operation (Figure 4.6a line 4), the context is compressed by saving one less variable. Excluded variable (`retKey`) will be reconstructed from available context variable (`kpv`) using this shift operation during the pop procedure.

Memory Divergence Avoidance

Applying CCC reorders the iterations of the loop. In the original program, a group of iterations with consecutive indices get assigned to threads with consecutive global indices; but employing CCC may disrupt this assignment. This becomes an issue when there are global/host memory reads and writes in the divergent path that are in a direct relationship with the iteration index. The original coalesced and cache-friendly memory accesses may lose the notion of locality due to context collection and retrieval. The introduction of memory divergence can hurt the performance.

```
1  __global__ void CUDA_kernel_SSSP(
2  const int numV, int* bitMask, int* costs, int* Ua,
3  const int* v, const int* e, const int* eValues ) {
4     for(
5     int vIdx = threadIdx.x + blockIdx.x * blockDim.x;
6     vIdx < numV;
7     vIdx += gridDim.x * blockDim.x ) {
8        int container = bitMask[ vIdx >> 5 ];
9        bool p = ( container >> ( vIdx & 31 ) ) & 1;
10       if( p ) { \\ Divergent path.
11          int vCost = costs[ vIdx ];
12          visit_nbrs( vIdx, vCost, costs, Ua,
13          v, e, done, eValues ); } } }
```

(a) Without optimization.

```
1  __global__ void CUDA_kernel_SSSP(
2  const int numV, int* bitMask, int* costs, int* Ua,
3  const int* v, const int* e, const int* eValues ) {
4     for(
5     int vIdx = threadIdx.x + blockIdx.x * blockDim.x;
6     vIdx < numV;
7     vIdx += gridDim.x * blockDim.x ) {
8        int container = bitMask[ vIdx >> 5 ];
9        bool p = ( container >> ( vIdx & 31 ) ) & 1;
10       int vCost;
11       if( p ) vCost = costs[ vIdx ];
12       if( p ) { \\ Divergent path.
13          visit_nbrs( vIdx, vCost, costs, Ua,
14          v, e, done, eValues ); } } }
```

(b) With Optimization (before transformation).

Figure 4.9: SSSP graph processing CUDA kernel from [30] containing a coalesced global memory access to the `costs` buffer in the divergent path. We preserve the coalescence in CCC by excluding the memory access from the divergent path.

Figure 4.9a presents an example of such scenario: Single-Source Shortest Path (SSSP) in a graph using CUDA [30]. In this example, threads are assigned to process vertices iteratively. A vertex is processed only if its corresponding bit in the `bitMask` buffer is set. In the divergent path, accesses to the `costs` array have locality and are coalesced. However, if we apply CCC to this kernel, as we did in Figure 4.3, due to reordering of iterations, this memory access will not be necessarily coalesced. In other words, nearby memory locations may be accessed at distant iterations which wastes memory bandwidth.

CCC can avoid memory divergence by taking the memory access out of the diverging path to the non-divergent block and stacking the memory content alongside the context. In other words, excluding the coalesced memory access from the path and executing it with the path predicate, as shown in Figure 4.9b. Therefore, CCC can be applied to the new divergent path similar to Figure 4.3 while preserving coalesced access pattern. The only difference is that now the context includes the memory content (`vCost`) as well.

Prioritizing the Costliest Branches

When there are multiple divergent paths taken by the warp lanes, context collection for all the branches may exceed SM's available shared memory and limit the theoretical GPU occupancy. Sometimes it might not even be possible to launch the kernel with a context stack for each and every path, due to requested capacity exceeding the limit. For example, applying CCC to all or most of the divergent paths inside the device function in Figure 4.8 is not possible or limits the occupancy because of limited available shared memory.

To avoid restricting the occupancy due to excessive use of shared memory, and at the same time, to avoid intra-warp divergence as much as possible, we prioritize the costliest branches, i.e. those branches for which context collection provides the most benefits. The cost of taking a divergent branch is proportional to the volume of operations inside the branch plus how infrequently it is visited by the warp lanes; later in Section 4.1.5 we verify this argument. Therefore, CCC applies to the longest branches with the least probability

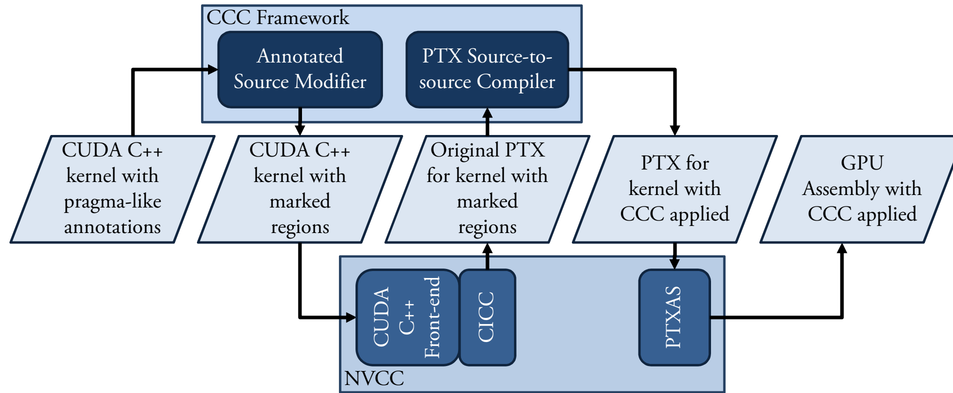


Figure 4.10: CCC Framework operates alongside NVCC.

of traversal. In case of the example in Figure 4.8, the path belonging to latest variations are longer and more expensive; hence, are more suitable candidates for CCC.

4.1.4 CCC Implementation

To implement CCC, we designed a framework based on a combination of user-provided annotations identifying the paths for CCC application and an automatic compilation chain intervention that transforms the code. The user annotates the repetitive pattern and the divergent path inside the CUDA C++ kernel. Then, the framework operates alongside NVCC and applies CCC to the code automatically, as shown in Figure 4.10. Each transformation described in section 4.1.2 is specified by a different annotation. For example, for the code in Figure 4.1a a user needs to insert only `#CCC for const` above the `for` loop and specify the divergent code block by putting `#CCC if` above the `if` condition evaluation line. The first part of the framework, Annotated Source Modifier, marks these specific regions to transfer them to the PTX level. The second part of the compiler, PTX Source-to-source Compiler, applies CCC and the optimizations described in Section 4.1.3.

Annotated Source Modifier

This part of the framework analyzes the code, identifies specific user-specified directives in CUDA C++ source code, and enables recognition of these patterns inside the PTX code. The frameowrk replaces and also inserts assembler statement `asm` with `volatile` keyword to mark the repetitive section and the beginning and the end of the divergent code path region. `asm` statement allows arbitrary code to propagate into and appear inside the PTX code, without even necessarily being a valid PTX statement. In addition, `volatile` keyword prevents optimization on specified assembly statement and preserves the relative order of instructions before and after the statement. These two properties of `asm volatile()` enables marking PTX code regions of interest from inside the CUDA C++ kernel. As shown in Figure 4.10, the framework feeds the marked CUDA kernel into NVCC front-end— which is tightly bound to CICC (LLVM-based optimizer and PTX generator)— to yield the PTX code.

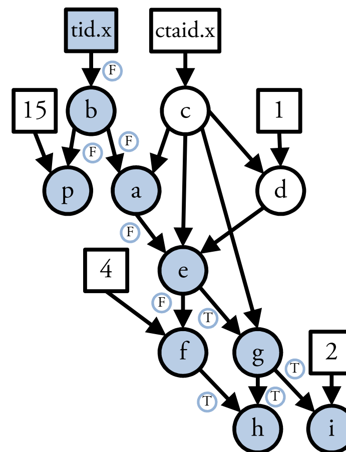
PTX Source-to-source Compiler

The second part of the framework receives a PTX source with distinct annotations that mark regions of interest including the beginning and the end of the divergent code block (similar to lines 9 and 13 in Figure 4.11a) and the immediate path after the repetitive pattern. This part outputs the PTX code with CCC and its optimizations applied, and sends the resulting PTX code to the rest of the compilation chain. Applying our technique at the PTX level is advantageous since the PTX code has a closer-to-machine assembly-like form with a limited number of instructions and directives. This facilitates reasoning about the functionality, throughput, inputs, and output of each instruction.

To apply CCC to the PTX code, It is necessary to recognize the context variables corresponding to a divergent path. To identify context registers, we generate a data dependence graph in which every node represents a PTX virtual register and every edge stands for the def-use link that defines the destination node and uses the source node. The

```
1  mov.u32       c,  %ctaid.x;
2  mov.u32       b,  %tid.x;
3  min.s32       a,  b,  c;
4  add.s32       d,  c,  1;
5  mad.s32       e,  c,  a,  d;
6  mul.wide.s32  f,  e,  4;
7  setp.le.s32   p,  b,  15;
8  %p  bra       POST_PATH_LBL;
9  #CCC_if_marked_begin
10 add.s32       g,  e,  c;
11 sub.s32       h,  g,  f;
12 add.s32       i,  g,  2;
13 #CCC_if_marked_end
14 POST_PATH_LBL:
```

(a) PTX code.



(b) Resulting graph.

Figure 4.11: A PTX sample code inside the repetitive section and the resulted graph from connecting definition and usage of virtual registers.

BFS traversal starting from thread-specific special registers induces sub-graphs of virtual registers (directly and indirectly) affected by them. Figure 4.11a shows a sample PTX code and Figure 4.11b presents its corresponding graph, in which the induced sub-graph is highlighted. In addition, the framework assigns every edge a boolean property for which a true value specifies if the def-use link between two virtual registers is established inside the divergent path. The framework then identifies a virtual register as a context if and only if, inside the sub-graph, its corresponding node's incoming edges are assigned false and at least one of its outgoing edges is assigned true. In Figure 4.11, registers `e` and `f` are recognized as the context.

The framework automates CCC optimizations using this analysis as well. For context compression, the framework examines a context register's parents inside the graph. If all the parent nodes are from the set of literals, function parameters, or other context registers and the instruction corresponding to the connecting edges is high-throughput and compute-only, the context is compressed. In Figure 4.11, virtual register `f` satisfies the compression condition. To avoid memory divergence, the index of the global memory read is examined to have a reaching definition from `tid.x` and to fulfill the coalescence. To

prioritize the costliest branch, the inverse of the instruction throughput inside every branch is aggregated and compared with other branches'.

After identification of context registers, the source-to-source compiler declares appropriate resources such as shared memory buffers and inserts CCC code segments.

4.1.5 Experimental Evaluation

In this section, we first briefly describe the benchmarks, then evaluate their performance with and without CCC in various respects. Lastly, we analyze the sensitivity of CCC. These experiments were performed on a Nvidia GeForce GTX 780 GPU equipped with 12 Streaming Multiprocessor from Kepler microarchitecture. Up to 2048 threads can reside on each SM while each SM contains 64K 32-bit registers. Up to 32 32-bit registers per thread and up to 24 bytes of shared memory per thread can be requested without affecting the occupancy. All GPU programs are compiled with the highest optimization level flag (-O3) for Compute Capability 3.5 on Ubuntu 14.04 64-bit with CUDA 7.0.

Benchmarks

We selected 8 real-world benchmarks from various domains including scientific computing, visualization, and graph analytics. These programs demonstrate substantial amount of intra-warp divergence; hence, they can benefit from CCC. Below we introduce these benchmarks and identify the transformations and optimizations that were applied to each of them.

- **BFS** *Breadth-First Search* is an iterative graph traversal algorithm. As previously mentioned, CUDA implementation of BFS [30] assigns one thread to process a number of vertices and their neighbors. We used LiveJournal [4], a social network graph with approximately 4.8M vertices and 69M edges, as the input graph. We applied a grid-stride loop, the transformation for loops with variable trip-count, and nested context collection to prepare BFS CUDA kernel for CCC.

- **DQG** *Dynamical Quadrature Grids* [54] program computes the points in a quadrature grid in which the integrand can evolve in time. *Becke* kernel of this program is the subject of our experiment. In the kernel, one thread is assigned to a point which then has to iterate over atoms two-by-two. The number of atoms in the list can vary from 2 to 80, which creates a load imbalance between threads. We used a grid-stride loop to enable application of CCC. The test molecule is BPTI and floating-point formats are double-precision.

- **EMIES** *Electromagnetic Integral Equation Solvers* [52] compute the electromagnetic field using Nonuniform Grid Interpolation Method (NGIM). The potential field domain is divided into subdomains of different sizes. The type of parallelization is "one-thread-per-observer". As a result, when comparing the domains, some threads in the warp may satisfy the Near-Field criterion and calculate the volume integral equation while others may not. We applied CCC with no transformation since the kernel is in a form that readily exposes the divergence in the repetitive code block. Average number of sources per box is set to 64 for the experiments.

- **FF** *Fractal Flames* [77] belongs to the Iterated Function Systems (IFS) class of algorithms and is based on chaos game. It involves selecting and executing a function randomly from the set of available non-linear functions. Picking and executing different functions for threads inside the warp causes task serialization. In our experiments, we defined 10 function variations and rendered a 2D scene with 10M random points. We used a grid-stride loop to iterative over points and *prioritized 3 costliest branches.* Since the context for all the divergent paths is 8 bytes (4 bytes for x and 4 bytes for y), and also since the original kernel does not consume shared memory, collecting up to 3 branches does not affect the occupancy.

- **HASH** *GPU Cuckoo hashing* [3] constructs a hash table given a set of key-value pairs. Each thread is assigned to carry out insertion of a set of key-value pairs into the hash table. Threads perform insertions simultaneously via atomic exchange operation.

Some threads in the warp may need to retry insertion due to collision creating intra-warp load imbalance. We applied a grid-stride loop to prepare the application for CCC. Then used the transformation for recursive functions. We also *compressed the initial context* that reduced the context size from 16 to 12 bytes. In the experiments we used 100 M randomly generated 8-byte-long key-value pairs, and the table load factor is set to 0.9.

- **IEFA** *Inverse Error Function Approximation* [26] involves selection and execution of one out of three possible functions depending on the input. Warp lanes that are assigned to compute the inverse error function usually take different paths. This causes traversal and execution of all three functions by the warp. We prepared the application for CCC with a grid-stride loop. For the experiments, we approximated double-precision inverse error function for 100M values.

- **RT** *Ray Tracing* [72] is a simple ray tracing CUDA kernel in which threads are assigned to rays and they verify if a ray hits the objects in the scene. A ray that hits an object has to update the closest hit depth and its own color. Threads inside the warp may or may not hit an object. This creates load imbalance and warp underutilization. We defined 8.8M rays (4K resolution) and 80 sphere objects in the scene. We applied a grid-stride loop to iterative over rays and enable context collection.

- **SSSP** *Single-Source Shortest Path* finds the shortest path to every vertex reachable from a single source vertex using iterative CUDA kernels. We used Harish et. al. [30] approach for the SSSP. A thread is assigned to process a set of vertices. Similar to BFS, the pattern of SSSP load imbalance is nested. LiveJournal [4] is our input graph. Applied transformations are the same as BFS. We also applied *memory divergence avoidance optimization* for SSSP.

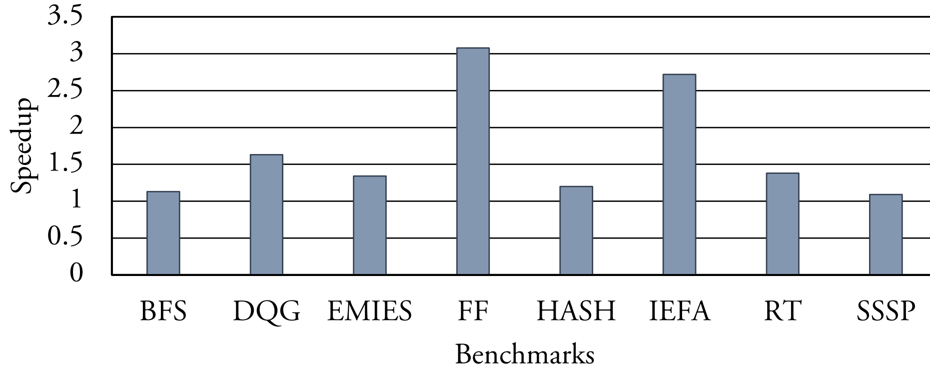


Figure 4.12: The kernel execution speedup provided by CCC. For benchmarks with iterative GPU kernel launches (BFS and SSSP) the speedup is measured based on the aggregation of kernels.

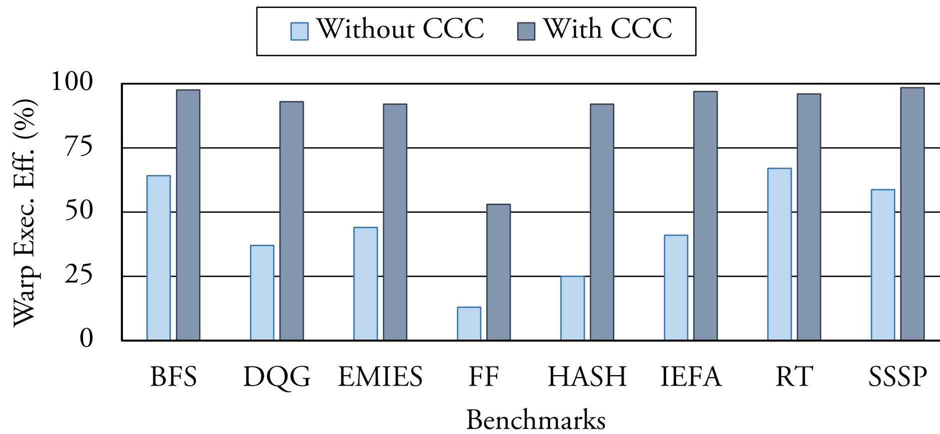


Figure 4.13: Warp execution efficiency comparison for kernels with and without CCC. For BFS and SSSP the warp execution efficiency is averaged across all the kernel launches.

CCC Performance Improvement

Figure 4.12 presents the speedups obtained by applying CCC to the eight real-world programs. These speedups are measured exclusively for CUDA kernels. On an average, the set of benchmarks experience sppedup of 1.69x due to application of CCC. We further profiled the warp execution efficiency (predicated and non-predicated averaged) of these benchmarks with and without CCC and plotted the results in Figure 4.13. On average, applying CCC increases the warp execution efficiency of benchmarks from 43.7% to 89.8%. We also measured the overhead introduced by CCC in terms of added shared

memory and register usage per thread and reported them in Table 4.1. As mentioned earlier, for our GPU, up to 32 32-bit registers per thread and 24 bytes of shared memory per thread can be requested without affecting the kernel occupancy. Now we examine the results in more detail.

CCC achieves the highest speedup of **3.08x** for the FF benchmark. For FF we collected the longest 3 divergent paths so as not to limit the occupancy by using extra shared memory. Although there are 10 divergent branches in the kernel, collecting the most expensive 3 of them enhanced the warp execution efficiency from 13% without CCC to 53% with CCC. This result demonstrates the importance of *branch prioritization* technique. Also note that the newest Nvidia GPU microarchitecture named Maxwell doubles the maximum shared memory available to the thread-block. More shared memory enables collecting more divergent path contexts without affecting the occupancy. Therefore, this allows higher speedup for benchmarks similar to Fractal Flames where diverging paths are numerous.

The benchmark for which CCC provides the next highest speedup of **2.72x** is IEFA. All three branches that can be taken by the warp lanes are collected and traversed only when full warp utilization is possible. By collecting all branches, CCC boosts the warp execution efficiency of IEFA from 41% to 97%. Similar to FF, IEFA experiences warp divergence due to dissimilar intra-warp task assignment and contains relatively long compute-only divergent task paths. These features make FF and IEFA the most benefiting benchmarks from the application of CCC.

The next benchmark for which CCC shows a relatively high speedup is DQG with speedup of **1.63x**. The different load volume assignments to each GPU thread in the original DQG results in the warp execution efficiency of 37% while CCC enhances to 93%.

The RT benchmark has the highest amount of warp execution efficiency in the original kernel with 67%. This is because warp lanes are assigned to contiguous rays which are more likely to hit an object in the scene. Nevertheless CCC provides speedup of **1.38x** for RT while increasing the warp efficiency to 96%.

Benchmark	32-bit Reg. Usage		Shared Mem. Usage (B)	
	w.o. CCC	with CCC	w.o. CCC	with CCC
BFS	15	26	0	8
DQG	25	31	8	24
EMIES	28	32	0	24
FF	21	29	0	24
HASH	22	28	0	12
IEFA	24	32	0	24
RT	21	27	0	24
SSSP	19	29	0	16

Table 4.1: The CCC overhead in terms of resource usage (per thread). Underlined entry results from spilling two excessive registers into local memory (L1 cache) via `-maxrregcount` compiler option. The maximum theoretical occupancy is 100% in all cases.

Further, CCC increased the warp execution efficiency of EMIES from 44% to 92% resulting in speedup of **1.34x**. EMIES divergent paths are long, but also containing global memory accesses. EMIES is the only benchmark where applying CCC can limit the maximum theoretical occupancy to 75% by requesting 34 registers per thread. However, for this benchmark, we pass the compiler option `-maxrregcount 32` to enforce the compiler to spill two registers into the local memory. Since the kernel asks for 24 bytes of shared memory per thread, 16 KB of shared memory in the SM is left for L1 cache, which is just enough for spilled registers.

Finally, benchmarks HASH, BFS, and SSSP are primarily memory-bound benchmarks; hence application of CCC results in smaller, yet significant, performance improvements. HASH benchmark relies heavily upon 8-byte-long atomic exchange operation on table entries. Entries accessed by threads inside the warp reside in distant memory segments. These accesses create non-coalesced and cache-unfriendly global memory requests which represent a major performance bottleneck for this kernel. As a result, although the warp execution efficiency increases from 25% to 96%, CCC provides smaller speedup of **1.20x**. The speedup of HASH without *context compression* optimization is lower – nearly **1.17x**.

Benchmarks BFS and SSSP are memory-bound graph algorithms. Both suffer from a great deal of load imbalance; however, the set of non-coalesced memory accesses to the content of neighboring vertices represents a major performance bottleneck. Therefore, applying nested CCC to BFS and SSSP provides smaller speedups of **1.13x** and **1.09x** respectively. The speedup in SSSP is the lowest due to higher amount of memory accesses; SSSP introduces additional memory accesses to the bit mask and edge value buffers. Also without *memory divergence avoidance* optimization, SSSP speedup is 1.06x. Considering the irregularity of the input graph, original BFS and SSSP kernels exhibit 58% and 64% warp execution efficiency on average. Note that this is due to early and late CUDA kernels in which most threads do not take the divergent paths. Kernels belonging to middle graph algorithm iterations carry out most of the computation and exhibit warp execution efficiency as low as 14%.

Sensitivity Analysis

In this subsection, using synthetic programs, we study the sensitivity of our technique to (a) varying amount of warp divergence over warp threads and (b) varying execution lengths of the divergent path. Original GPU kernels are in form of grid-stride loop. The loop is executed 2^{30} times (aggregated over all the CUDA kernel launched threads).

Figure 4.14 compares the execution time and the warp execution efficiency of the synthetic GPU kernel when executed normally and when CCC applied. The loop contains a divergent path with 20 FMAD instructions. We repeated the experiments each time with different number of threads inside the warp taking the divergent path (x axis). Increasing this number also increases the amount of load carried out by the CUDA kernel.

The left plot in Figure 4.14 shows that the original kernel takes an approximately constant amount of time to finish with different amount of intra-warp divergence and workload imbalance. This is a natural behavior of a SIMD device. On the other hand, when CCC is applied, we see that by increasing the workload, the execution time grows linearly. It means CCC provides work efficiency.

The right plot in Figure 4.14 presents the warp execution efficiencies. We observe that although the warp execution efficiency of the original kernel increases proportionally to the amount of intra-warp divergent tasks, the kernel with CCC always has a high warp efficiency (%96.6 on average) regardless of the amount of intra-warp divergence. *In summary, both plots demonstrate the effectiveness of CCC in form of its resistance against various amounts of load imbalance.*

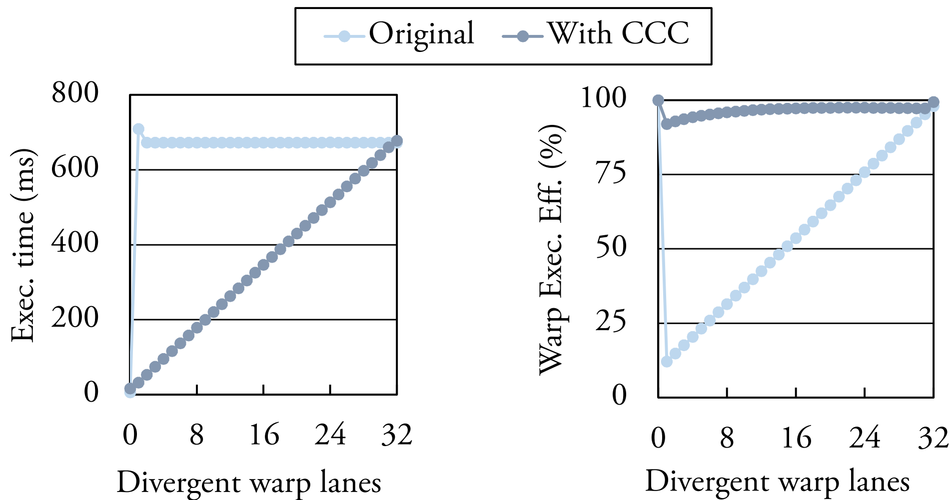


Figure 4.14: CCC performance enhancement compared to the original divergent kernel over different amount of intra-warp divergence (and hence workload imbalance). The divergent path contains 20 FMAD operations.

Figure 4.15 shows the execution times for the original kernels and kernels with CCC for different divergent path lengths. The left and right plots demonstrate it for when $\frac{1}{4}$ and $\frac{3}{4}$ of the warp lanes take the divergent path, respectively. In both plots, it is clear that as the length of the divergent path increases, the speedups approach to inverse of utilized threads in the original codes, i.e. $\frac{4}{1}$ and $\frac{4}{3}$ respectively. It is evident that CCC shows more speedup where the divergent path is longer. In addition, by comparing the two plots, we realize that to cope with the overhead of CCC, either the divergent path has to be long or the divergent ratio should be high. In both plots, the speedup is less than one only when

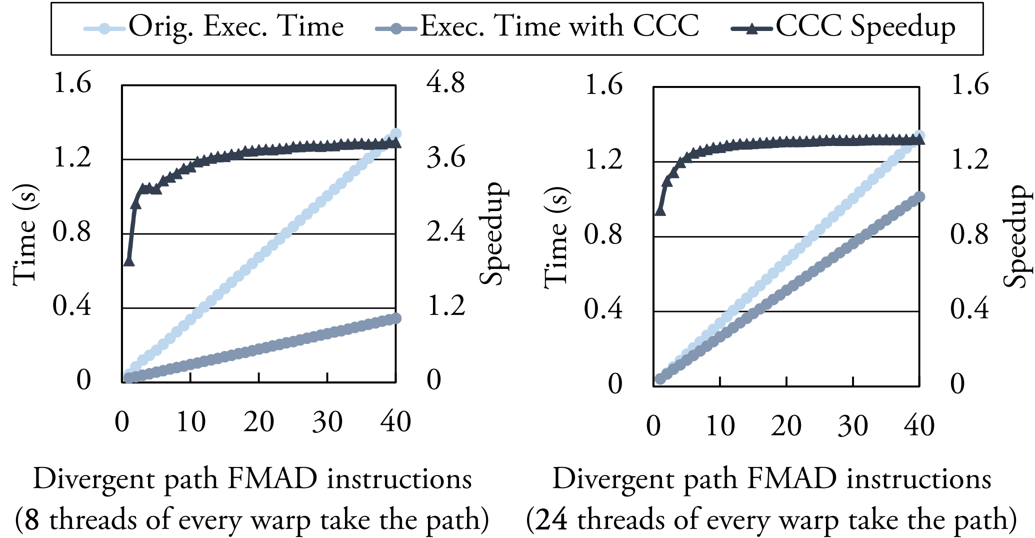


Figure 4.15: Sensitivity of CCC against different execution paths lengths plotted for two different amounts of intra-warp divergence.

the divergent path contains only one operation and at the same time 24 threads inside every warp take the path. In all other cases, the speedup provided by CCC is higher than one.

4.2 Collaborative Task Engagement

The abundance of execution units, accompanied with a high memory bandwidth, have made GPUs the primary candidates for accelerating algorithms containing data parallelism. To increase the energy efficiency, GPU threads are grouped into *warps*[1] (on current Nvidia devices, 32 threads are grouped into one warp). While the instruction fetch and decode are performed once for all the threads inside the warp, threads map into different GPU execution units (cores) to process different data. Hence, the underlying design enforces the whole warp to contain only one active PC (Program Counter) at a time. However, GPU's SIMT architecture design allows threads inside the warp to take different execution paths by masking off inactive threads. This feature has made GPU programming easier since the developers need not worry about handling diverging threads executing unwanted pieces of

[1]We use terms employed in CUDA platform to describe GPU specific architecture and programming model.

code. However, this comfort can jeopardize the performance. One such frequent scenario is of nested patterns that contain imbalanced loads, more specifically a pattern where *a set of coarse-grained tasks hold a number of fine-grained tasks with different sizes.*

Early work for handling such nested patterns on GPUs employed 1D decomposition by assigning one thread to every coarse-grained task. The thread then iterates over the fine-grained tasks and carries them out. This approach has appeared in many GPU applications including graph processing [30], Sparse Matrix Vector Multiplication (SpMV) [8], and the analysis framework in [48] where it is known as 1D mapping. While being concise and easy to reason about, 1D decomposition is highly prone to underutilization in presence of imbalanced loads since all the threads inside the warp have to wait for the thread that has been assigned the largest number of fine-grained tasks. To cope with the warp underutilization issue in irregular nested workloads, researchers suggested assigning fixed-sized sub-warps to coarse-grained tasks [34, 91]. Therefore, threads in a sub-warp carry out its assigned fine-grained tasks iteratively. We refer to this approach as sub-warp decomposition. Although providing better warp utilization compared to 1D decomposition, sub-warp decomposition lacks portable performance since every application and input combination exhibits the best performance at a specific sub-warp width. Most importantly, the same issue that hurts 1D decomposition performance still exists in sub-warp decomposition. Here, the whole warp has to wait for the sub-warp with the largest assigned task size.

We present Collaborative Task Engagement (CTE) technique to enhances the warp utilization of irregular nested tasks compared to previous approaches. It provides a portable performance across inputs and applications. Unlike aforementioned static task-to-thread assignment methods, CTE delivers dynamic decomposition via the expansion of coarse-grained tasks. In multiple rounds, each thread inside the warp gets assigned the work of mapping portion of a fine-grained task regardless of the coarse-grained task it belongs to. Later, the thread determines the coarse-grained task to which the fine-grained task belongs. This is achieved efficiently via a binary search of the buffer containing prefix sum of task sizes – the buffer is held inside the shared memory. Therefore, it can participate in the reduc-

tion for the coarse-grained task's final value, if necessary. CTE does not over-subscribe or under-subscribe warp threads in the mapping stage and yet performs parallel reduction in minimum number of steps between the fine-grained tasks of a coarse-grained task in every round. To facilitate the employment of our technique, we have prepared a CUDA C++ device-side template library. The template library abstracts away the complications of the implementation of nested pattern with CCC, allowing developers to focus on the program's algorithm and to quickly obtain the desired functionality. In addition, the template library is built with the program hence providing ultimate portability across various systems. We measured and analyzed the performance of our CTE in comparison with other static decomposition methods across different class of applications. CTE improves the warp execution efficiency of CUDA kernels by up to 37% and provides 1.51x speedup compared to the sub-warp decomposition with the best sub-warp width.

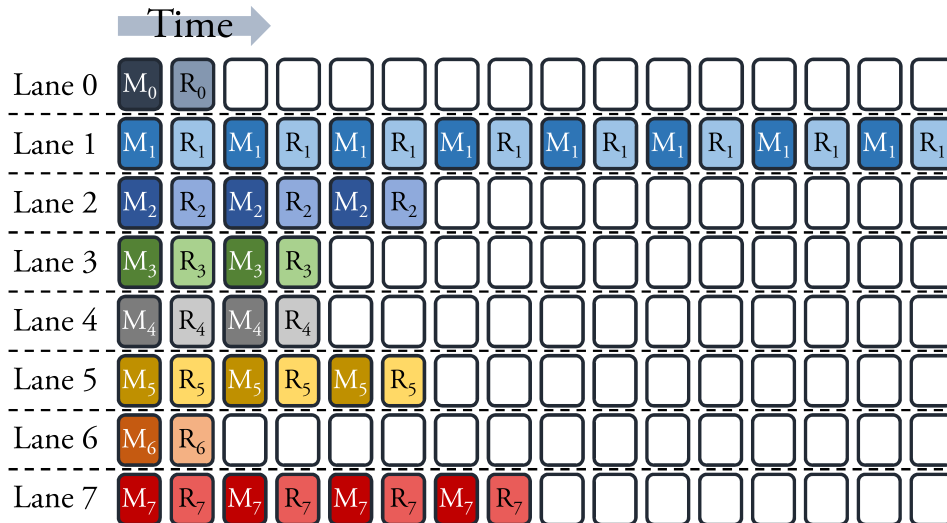
The rest of this section is organized as follows. We first explain the drawbacks of available static task-to-thread assignment approaches and motivate the necessity of a solution that copes with irregularities in the input. Then we present our solution that dynamically assigns tasks to SIMD threads and is robust against load imbalance. Finally we give experimental evaluation results.

4.2.1 Motivation: Inefficiency of Static Task Decomposition Methods

GPU's innovative SIMT architecture enables the implementation of conditional device code in which threads belonging to a warp take different execution paths. The underlying hardware keeps track of active and inactive warp threads in diverging paths and masks off irrelevant threads. While this scheme speeds up GPU software development, it can easily make GPU kernels prone to resource underutilization. If only a few threads take a divergent path, all other threads inside the warp will have to wait for those threads before they can continue. In other words, execution units are reserved for inactive threads and they perform no operations.

```
1  template<typename valT, typename idxT>
2  __global__ void spmv_CSR_1D_mapping( const valT* mat,
3   const idxT* nnzRowScan, const valT* inVec,
4   const idxT* colIdx, valT* outVec ) {
5     int rowID = threadIdx.x + blockIdx.x * blockDim.x;
6     valT sum = 0;
7     const idxT startPos = nnzRowScan[ rowID ];
8     const idxT endPos = nnzRowScan[ rowID + 1 ];
9     for( idxT i = startPos; i < endPos; ++i ) {
10       valT mapped =
11        mat[ i ] * inVec[ colIdx[ i ] ];  // MAP.
12       sum += mapped;  // REDUCE.
13    } outVec[ rowID ] = sum; }
```

(a) The SpMV CUDA C++ kernel with 1D decomposition.



(b) The visualization of a possible warp execution of the kernel in Figure 4.16a. Warp size is assumed 8.

Figure 4.16: An example — Sparse Matrix-Vector Multiplication (SpMV) CUDA kernel with a CSR matrix using 1D decomposition. Intra-warp load imbalance induces warp inefficiency and performance loss.

Aforementioned issue intensifies in GPU kernels with irregular nested patterns where there are a number of coarse-grained tasks each of which contains a different number of fine-grained tasks. Therefore, threads inside the same warp may have to iterate different number of times over the code to fully carry out their task. Figure 4.16a illustrates the above problem using an example of nested pattern that appears in the SpMV kernel with CSR (scalar) [7] decomposition. Since all the threads reconverge at the end of the loop, different amounts of load for different warp threads results in partial warp utilization. In other words, threads that finish early stay inactive until the thread with the longest number of iterations finishes. In Figure 4.16a, inside the loop, threads first compute the intermediate value (*map*) and then *reduce* it with the thread's private variable. Figure 4.16b visualizes the utilization of warp threads executing this loop. Note that in this work, we focus on intra-warp task assignment strategies.

This task assignment strategy, being very intuitive, is frequently seen in widely-used GPU applications involving nested parallel patterns; especially when the algorithm contains a set of coarse-grained tasks each of which containing a number of fine-grained tasks. Sparse Matrix-Vector Multiplication (SpMV) with Compressed Sparse-Row (CSR) format in [8] uses this task assignment strategy, and is algorithmically identical to the kernel in Figure 4.16a. This method is also employed by the analysis framework in [48] where it is called *1D mapping*. We use the term 1D decomposition throughout this paper.

To tackle the inefficiencies of 1D decomposition, CUDA provided *dynamic parallelism* to let threads spawn thread blocks for ease of expressing nested patterns; it has been shown in [91, 83] that dynamic parallelism imposes overheads such as parent thread blockage and communication via relatively slow global memory between the parent and children threads. These overheads have hindered the adoption of dynamic parallelism in GPU applications.

Other task assignment approaches aim to exploit parallelism inside a coarse-grained task— which is untouched by the 1D decomposition. The most notable among them, groups the threads belonging to the same warp and assigns the resulting coarse-

grained tasks to the warps [8]. Later works improve upon this strategy by dividing the warp into smaller sub-warps and assigning each sub-warp to process a coarse-grained task. Sub-warps have fixed width – one of 2, 4, 8, 16, or 32 – throughout the kernel computation. Threads within a sub-warp participate in carrying-out the fine-grained tasks of the coarse-grained task. As an instance of usage of this scheme, CUSP library [79] assigns a sub-warp to process a row of the CSR matrix in SpMV computation. Threads inside the sub-warp execute the mapping function for a section of the row and reduce the outcomes in parallel. This procedure is performed iteratively on all the sections of the row. CUDA-NP [91] expressed a similar approach in form of a primary thread and a few subordinate threads for nested parallel patterns. Here we refer to this approach as the *sub-warp decomposition* (such approaches have also been called *warp-based mapping* [48]). Figure 4.17 visualizes the warp execution for the example in Figure 4.16 when sub-warp decomposition with width 4 is employed.

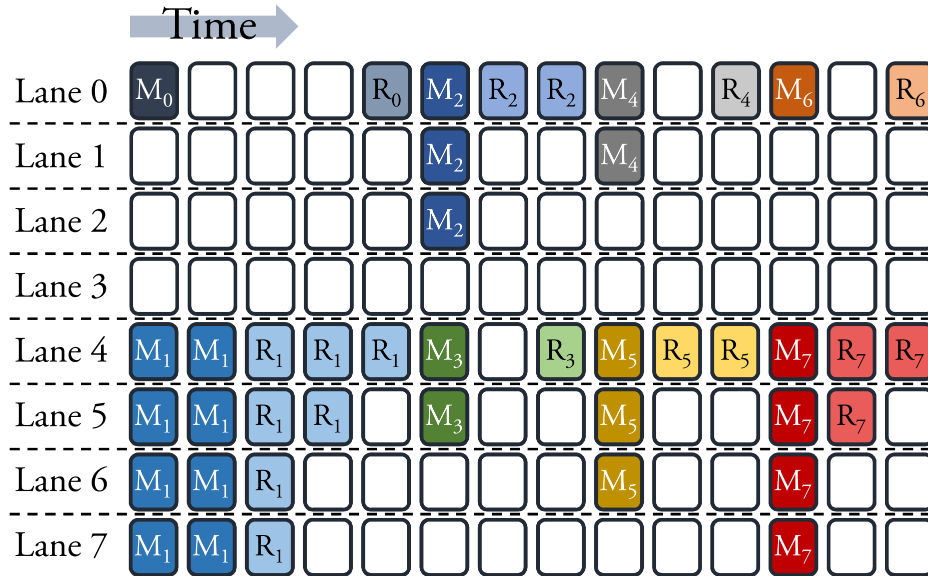


Figure 4.17: Warp execution visualization in sub-warp decomposition (with width 4) for the example in Figure 4.16. Sub-warp decomposition attempts to exploit parallelism inside coarse-grained tasks.

Benchmark	Input	1D	VW2	VW4	VW8	VW16	VW32
SpMV	Wbedu	16.6%	36.8%	41.1%	49.8%	<u>50.4%</u>	28.9%
	Delau	57.0%	63.4%	<u>66.5%</u>	61.3%	52.7%	38.7%
FMM	nEquProb	20.9%	26.8%	<u>39.7%</u>	33.2%	37.8%	39.4%
	EquProb	42.2%	44.3%	44.7%	<u>44.8%</u>	36.7%	29.1%

Table 4.2: Kernel warp execution efficiency of CUDA applications exposed to different inputs with 1D and sub-warp decomposition methods. The efficiency of kernels not only varies from one sub-warp width to another (the best in each row is underlined), it is also well below 100%.

Although sub-warp decomposition provides improved SIMD utilization, it suffers from a constraint. In order to guesstimate the best sub-warp width, for every specific GPU kernel, the developer needs to know the characteristics of the algorithm and must analyze the input in a preprocessing step. This constraint of sub-warp decomposition makes application portability unachievable. A sub-warp width that works well for one input, may not deliver a good performance for another input. In addition, sub-warp decomposition suffers from the same issue as 1D decomposition. It leaves a great portion of the warp underutilized when some coarse-grained tasks contain a large number of fine-grained tasks while others have only a few. For example, CUSP's heuristic to determine the best sub-warp width is to choose the closest equal or higher power of 2 to the average of the non-zeros per row (for averages bigger than the warp size it chooses the warp size). Relying only on the average of the coarse-grained task distribution, this method basically ignores their variance. Some rows of the input matrix may have much larger number of non-zero elements than other rows but this method assigns the same processing power to each and every row. Thus, the entire warp must wait for the sub-warp with the largest amount of fine-grained tasks.

Essentially, **in both 1D and sub-warp decomposition methods, the static thread-to-task assignment not only lacks portable performance across different inputs, but also makes the kernel highly susceptible to the warp execution inefficiency due to load imbalance between irregular coarse-grained tasks**. Ta-

ble 4.2 confirms this observation by showing profiled warp execution efficiency[2] of different benchmarks (FMM stands for Fast Multiple Method [43] for n-body approximation) and inputs for 1D decomposition and sub-warp decomposition with different sub-warp widths. We can see that different applications with different inputs demonstrate the best execution efficiency at various sub-warp widths.

Above observation motivates the need for an approach that, regardless of the input task size variance, effectively maps the irregular tasks to threads for efficient execution for nested patterns on the GPU architecture.

4.2.2 Collaborative Task Engagement

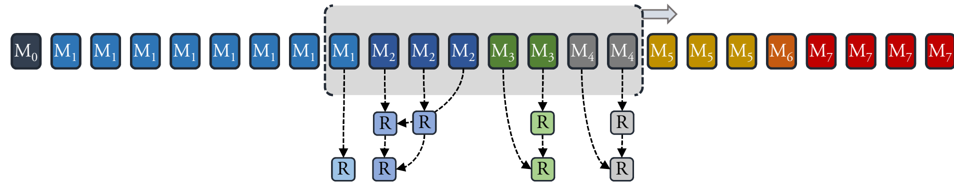
In this section, we introduce our technique, Collaborative Task Engagement (CTE), that eliminates warp inefficiencies induced by irregular nested patterns. We first describe our solution, then explain its efficient CUDA implementation, and finally, we present our template library that allows developers to use this technique with ease.

Dynamic Task Assignment in CTE

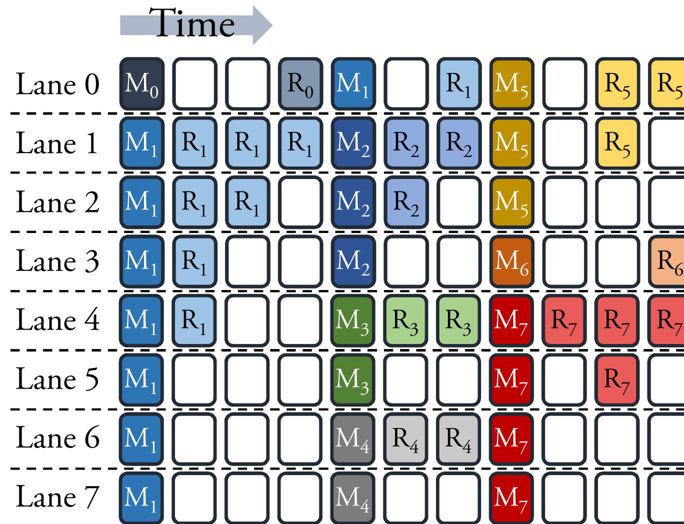
To handle irregularities in nested patterns, we propose Collaborative Task Engagement (CTE). In CTE, similar to sub-warp decomposition, a fine-grained task is defined as a combination of a *mapping function* and an *associative reduction function*. While a mapping function computes a candidate value for the fine-grained task, the reduction function does a summary operation over fine-grained tasks' candidate values for the coarse-grained task. For instance, lines 11 and 12 in Figure 4.16a show the mapping function which computes the result by multiplying the matrix element with the corresponding vector element, and line 13 shows the reduction function which accumulates the resulting values to yield the output vector element.

[2] Warp Execution Efficiency is a metric provided by Nvidia Profiler defined as the "ratio of the average active threads per warp to the maximum number of threads per warp supported on a multiprocessor." It is a measure indicating what fraction of threads of warps in a kernel have been active on an average. It can also be seen as a measure that is inversely proportional to the overall thread divergence.

In CTE, instead of assigning one coarse-grained task to one thread (1D decomposition) or a fixed number of threads inside the warp (sub-warp decomposition), **we assign a group of coarse-grained tasks to a warp, and let the threads in the warp collaborate to carry out the fine-grained tasks belonging to the coarse-grained tasks assigned to the warp**. More specifically, threads of the warp view and iterate over the list of fine-grained tasks resulting from the expansion of coarse-grained ones. Figure 4.18 demonstrates this scheme for the example in Figure 4.16.



(a) The warp acts as a sliding window executing the expanded list of fine-grained tasks in the regions with the size equal to the warp size.



(b) CCC reduces the execution time of irregular nested tasks by enhancing the warp efficiency.

Figure 4.18: Visualization of the SpMV CUDA kernel in Figure 4.16a after applying CTE.

Unlike previous static task decomposition methods, **Each thread inside the warp is assigned to execute one compute function corresponding to a fine-**

grained task; regardless of the coarse-grained task from which the fine-grained task comes from. By unbundling fine-grained tasks from their coarse-grained task, the execution of compute stage in CTE completely avoids the warp inefficiency. After the compute stage, since fine-grained tasks from a coarse-grained task are processed by consecutive threads in the warp, a thread can find its corresponding coarse-grained task and execute the reduction function over the results with its neighbors, if necessary. The parallel reduction is performed over the results of the computation for a coarse-grained task to produce its final result. Figures 4.18a and 4.18b exhibit this procedure. As can be seen, warp inefficiencies due to load imbalance in CTE can only appear during the reduction phases, however, their effect will not last longer than at most $\log warpSize$ reduction steps. The advantages of CTE include:

- It avoids under-subscribing or over-subscribing warp threads during the mapping by assigning a map function to every thread in each round regardless of their corresponding coarse-grained task; and
- It reduces the effect of load imbalance between coarse-grained tasks by performing parallel reduction over the fine-grained tasks belonging to a coarse-grained one.

Efficient CUDA Implementation of CTE

Next, we describe the details of CTE's CUDA implementation using the pseudo-code presented in Figure 4.19. We provide a step by step description of the pseudo-code.

Shared memory declaration and allocation – CTE requires shared memory buffers to exchange data between threads of a warp. To enforce sequential consistency between the shared memory accesses within a warp, `volatile` qualifier accompanies shared memory declarations (lines 1-4). Using this technique— and since in CTE the set of interactions between threads is confined to within their own warp— our procedure avoids introducing any explicit syncing or fencing primitives.

```
1  volatile shared scans[ N_CTA_WARPS ][ WARP_SIZE ];
2  volatile shared reds[ N_CTA_WARPS ][ WARP_SIZE ];
3  volatile shared mapped[ N_CTA_WARPS ][ WARP_SIZE ];
4  volatile shared taskDesc[ N_CTA_WARPS ][ WARP_SIZE ];
5  scans[ warp_id ][ lane_id ] =
    prefix_sum( ThreadCoarseTask.size );
6  NFineTasks = scans[ warp_id ][ WARP_SIZE - 1 ];
7  firstLoad = scans[ warp_id ][ 0 ];
8  reds[ warp_id ][ lane_id ] =
    ThreadCoarseTask.initRedVal;
9  taskDesc[ warp_id ][ lane_id ] =
    ThreadCoarseTask.fineTaskDescriptor;
10 for( fineTaskID = lane_id;
        fineTaskID < NFineTasks;
        fineTaskID += WARP_SIZE ) {
11    coarseTaskID =
       binary_search( fineTaskID, scans[ warp_id ] );
12    fineTask = task_descriptor( fineTaskID,
       taskDesc[ warp_id ], coarseTaskID );
13    mapped[ warp_id ][ lane_id ] = map( fineTask );
14    inSegIdx = min( lane_id, fineTaskID -
       scans[ warp_id ][ coarseTaskID ] + firstLoad );
15    segSize = min( scans[ warp_id ][ coarseTaskID ]
       - fineTaskID, WARP_SIZE - lane_id ) + inSegIdx;
16    redElemPos = ( inSegIdx != 0 ) ?
       ( mapped[ warp_id ] + lane_id - 1 ) :
       ( reds[ warp_id ] + coarseTaskID );
17    for( i = WARP_SIZE / 2; i > 0; i /= 2 )
18       if( ( inSegIdx + i ) <= segSize )
19          *redElemPos = reduce( *redElemPos,
             mapped[ warp_id ][ lane_id + i - 1 ] );
20 }
21 return reds[ warp_id ][ lane_id ];
```

Figure 4.19: GPU pseudo-code for CCC.

Coarse-grained task feature extraction – Initially, each thread corresponds to one coarse-grained task. This coarse-grained task is the input to the procedure in Figure 4.19. First, at line 5, we compute the *inclusive prefix sum* of coarse-grained task sizes that threads of the warp hold and save them into the `scan` buffer. This buffer is necessary for multiple uses in the iterative code section (lines 10-20). For a fast intra-warp prefix sum, we employed the method introduced in [86] that utilizes the *shuffle intrinsic*. After calculating prefix sums, the last element gives the total number of fine-grained tasks (line 6). Plus, we put the first element into a variable (line 7) so we can use it inside the iterative segment to get the *exclusive prefix sum* results. At line 8, each thread inserts the initial

value (given by the user-specified algorithm) for the reductions of fine-grained tasks over its initially assigned coarse-grained task. The `reds` buffer collects the reduction results in the iterative section and eventually to be output by the program (line 21). Moreover, `taskDesc` specifies the set of shared memory buffers that collect the fine-grained task descriptors for coarse-grained tasks.

Each thread, which is initially assigned to a coarse-grained task, has a number of variables that are used by the fine-grained tasks inside the coarse-grained task and vary among the coarse-grained tasks. For example, in Figure 4.16a `startPos` and `endPos` are thread-private variables that directly affect the execution of fine-grained tasks inside the loop. We call such variables *task descriptor*. Since in CTE fine-grained tasks of one coarse-grained task are executed by multiple threads, the thread saves its task descriptor variables inside the shared memory so as to make them accessible by all the threads inside the warp. Note that this is necessary for all decomposition methods that need to exploit the parallelism inside coarse-grained tasks (such as sub-warp decomposition).

Fine-grained task assignment and mapping – Lines 11-20 present the iterative segment in which every thread inside the warp is assigned to one fine-grained task identified by `fineTaskID`. First, the coarse-grained task owning the thread's assigned fine-grained task is found via a binary search on the `scans` buffer inside the shared memory at line 11. Then, at line 12, thread's assigned fine-grained task is retrieved from the `taskDesc` buffer using thread's fine-grained task index and its corresponding coarse-grained task index. The thread executes the mapping portion of the described fine-grained task in line 13 and saves the result inside the designated shared memory buffer position.

Parallel reduction of mapped fine-grained tasks – At this point, threads inside the warp performed mapping on fine-grained task, and now, need to properly reduce mapped values. Since fine-grained tasks belonging to one coarse-grained task were assigned to consecutive threads, they form segments when they are processed using the `for` loop specified in line 10. If the thread discovers its index inside the segment and also the segment size, parallel reduction inside the segment will become feasible. Thus, line 14 calculates the

intra-segment index using `scans` buffer and line 15 computes the segment size by adding the intra-segment index with the fine-grained task index inside the segment when observed from right to left. In line 16, we assign the first thread inside the segment to the segment's reduction element inside `reds` buffer and assign the rest of the threads to the mapped value of the thread before them inside the segment. This re-assignment becomes beneficial by eliminating the need for an additional reduction with the corresponding element inside the `reds` buffer at every iteration. Finally, an intra-segment parallel reduction (with unrolled loop in the actual implementation) reduces the mapped values and saves the outcome inside the corresponding `reds` buffer position. Threads keep executing the code section in lines 10-20 until all the fine-grained tasks are carried out. Finally each thread returns the reduced value for its initially-assigned coarse-grained task (line 21).

CTE as A Device-side Template Library

While CTE provides an efficient method to handle irregular nested patterns, its implementation from scratch for every GPU kernel can be time-consuming and challenging for CUDA programmers. To enable easy usage of CTE by developers, we provide our technique as a CUDA C++ device-side template library. A CUDA developer only needs to include our library header file and call the designated library function. The library function takes as its parameters the thread's fine-grained task index range, mapping and reduction functions, and initial content for thread's coarse-grained task's reduction value. While the programmer expresses the tasks as if each thread is assigned to one coarse-grained task (similar to 1D decomposition), the library manages the CTE execution behind the scenes.

Figure 4.20 shows the usage of our library for the SpMV kernel. In this example, the mapping function is defined as a lambda (lines 10 and 11) that takes the iteration index as the parameter and returns the corresponding element, i.e. the multiplication outcome. The signature of the mapping function for our library requires the first parameter to be the iteration index while the rest of the parameters can be passed by the user as the *lane*

```
1  #include <cte.cuh>  // CTE library inclusion.
2  template<uint BlockDim, typename valT, typename idxT>
3  __global__ void spmv_CSR_with_CTE( const valT* mat,
4   const idxT* nnzRowScan, const valT* inVec,
5   const idxT* colIdx, valT* outVec ) {
6     int rowID = threadIdx.x + blockIdx.x * blockDim.x;
7     valT sum = 0;
8     const idxT startPos = nnzRowScan[ rowID ];
9     const idxT endPos = nnzRowScan[ rowID + 1 ];
10    auto mapF = [&]( idxT idx ) {  // MAP.
11     return mat[ idx ] * inVec[ colIdx[ idx ] ]; };
12    auto redF = []( valT lhs, valT rhs ) {  // REDUCE.
13     return lhs + rhs; };
14    sum = cte::for_each_index<BlockDim, cte::scanned>
15     ( startPos, endPos, mapF, redF, sum );
16    outVec[ rowID ] = sum; }
```

Figure 4.20: Expressing the nested pattern in Fig. 4.16a CUDA C++ kernel in CTE form using our template library interface.

state. Lines 12 and 13 in Figure 4.20 present the reduction function— again as a lambda expression— for this example. The reduction function signature for the library accepts only two parameters and returns one value of the same type. Finally, lines 14 and 15 give the function call to execute the tasks with CTE technique.

CTA (thread-block) dimension needs to be sent to the function as the first template argument so the library would have the correct size for the static shared memory allocation. Also, the second template argument hints the library that the indices of consecutive threads are prefix summed. In other words, the ending index for thread i's region is the beginning index for thread $(i + 1)$'s. This template specialization will allow the library to avoid recalculation of the prefix sum of the fine-grained task sizes. If such relationship between indices does not exist, the user will have to pass `cte::disjoint` as the template argument. We mentioned earlier that in this example `startPos` and `endPos` act as task descriptors and need to be passed as function arguments. We specialized the CCC function calls with more template signatures so that for an arbitrary mapping function, other task descriptor variables can be passed as the last variables of the CTE function call in the order they appear as the further mapping function parameters.

Finally, since all the threads of the warp need to be present for a correct CTE execution, upon entering the execution function, the library performs a `__ballot()` operation with `true` predicate. If the result of this operation is not a variable with all bits set, it means one or a number of warp threads are absent. In this case, as a safety procedure the library falls back to the 1D decomposition method.

CTE Analysis for Comparison with Static Decomposition Methods

To analyze the CTE characteristics and compare it with static decomposition methods, we briefly provide analysis over the execution time for static decomposition methods and CTE. We show that while the execution time for 1D decomposition and the upper-bound for sub-warp decomposition execution times are a function of the *maximum*(s) of the set of coarse-grained task sizes, the upper-bound for the execution time of CTE is a function of the *average* of the workload.

Assumptions and notation. For analysis, let us assume that the warp size is W, and for simplicity, further assume that there are W coarse-grained tasks to be processed – the time for bigger coarse-grained tasks can be obtained via scaling. Let us denote the execution time of the *mapping* and *reduction* functions by T_{MAP} and T_{RED} respectively. Note that for simplicity of analysis we assume that these times are constant and not affected by memory access latencies.

1D decomposition – Given a set of coarse-grained tasks $L = \{l_1, l_2, ..., l_W\}$, the execution time of this set of loads with 1D decomposition is given by:

$$t_{1D}(L) = \max L \times (T_{MAP} + T_{RED}) \quad (4.1)$$

Equation 4.1 above can be easily understood by examining Figure 4.16b. Note that the above equation shows that t_{1D} is a function of the *maximum* of the set of loads.

Sub-warp decomposition – For sub-warp decomposition, if the width of the chosen *sub-warp* is S (note $\log_S W \in \mathbb{N}$), then the upper-bound for its execution time is

given by:

$$t_{SW}(L,S) = \sum_{s=1}^{S}(T_{RED} \times \log_2 S + (T_{MAP} \times \lceil \frac{\max\{l_i | \frac{(s-1)\times W}{S} < i \leq \frac{s\times W}{S}\}}{S} \rceil))$$
$$= T_{RED} \times S \log_2 S + T_{MAP} \times \sum_{s=1}^{S} \lceil \frac{\max\{l_i | \frac{(s-1)\times W}{S} < i \leq \frac{s\times W}{S}\}}{S} \rceil \quad (4.2)$$

Initial form of Equation 4.2 sums up the execution times in different rounds since sub-warp decomposition assigns V threads to process a coarse-grained task. While $\log_2 V$ is the maximum number of steps required for the reduction, the mapping operation is repeated within a round by the warp as long as the largest coarse-grained task's mapping functions assigned to a sub-warp are being performed. The final form of Equation 4.2 shows that t_{VW} is still a function of the largest tasks. Also, V appears at both top and the bottom of the fractions of Equation 4.2 which usually makes t_{VW} a non-monotonic function of V. The V for which t_{VW} is minimum depends on the load distribution.

CTE – On the other hand, the upper-bound of the execution time for CCC is expressed as below:

$$t_{CTE}(L) = \lceil \frac{\sum_{i=1}^{W} l_i}{W} \rceil \times (T_{MAP} + T_{RED} \times \log_2 W) = \lceil Avg(L) \rceil \times (T_{MAP} + T_{RED} \times \log_2 W) \quad (4.3)$$

In CTE, in every round the fine-grained compute portion of the tasks are assigned to the warp threads and therefore the sum of loads divided by the warp size is the coefficient of both T_{MAP} and T_{RED} in Equation 4.3. Also, the reduction will take $\log_2 W$ steps at most, therefore, this term accompanies T_{RED}. Considering the final form of Equation 4.3, we can see that the upper-bound for t_{CTE} is a function of the *average* of the loads, not their *maximum* unlike the previous two methods, i.e. 1D and sub-warp decomposition.

4.2.3 Experimental Evaluation

In this section, we evaluate the performance of CTE and compare it with 1D and sub-warp decomposition methods. We selected applications from various domains including sparse matrix operations, scientific computing, and graph analytics for this purpose. We performed the experiments on a Nvidia GeForce GTX 780 with 12 Streaming Multiprocessors from the Kepler family. We compiled and ran all the programs for CUDA Compute Capability 3.5 with -O3 and C++11 compilation flags on a system with Ubuntu 14.04 and CUDA 7.0.

Performance Analysis

Sparse matrix operations – Figure 4.21 presents the speedup of CCC over 1D decomposition and compares it with the speedup provided by sub-warp decomposition from CUSP library [79] for two application from sparse matrix operation domain. SpMV is the Sparse Matrix Vector Multiplication and DIAG is the extraction of the diagonal of the given matrix. Input graphs are from The University of Florida sparse matrix collection [17] and exhibit different structures and therefore nested load size variation. Rajat31 (Rajat) is an unsymmetric and rather regular matrix with a dimension of 4.69M and approximately 20M non-zero elements. Delauny_n24 (Delau) is a symmetric irregular matrix with 16.7M rows and columns and 100M non-zero elements. Wb-edu (Wbedu) is a more irregular unsymmetric matrix compared to Delau with 9.8M rows and columns and 57M non-zero elements. Also, to further verify the CCC performance compared to static decomposition methods, we profiled the warp execution efficiency of CCC, sub-warp and 1D decomposition kernels with Nvidia Profiler and plotted the results in Figure 4.22.

Starting with Wbedu in Figure 4.21 as the most irregular input, CTE provides 2.8x and 2.3x speedup compared to 1D decomposition for SpMV and DIAG respectively. SpMV is a more compute-intensive application compared to DIAG and can benefit more from our

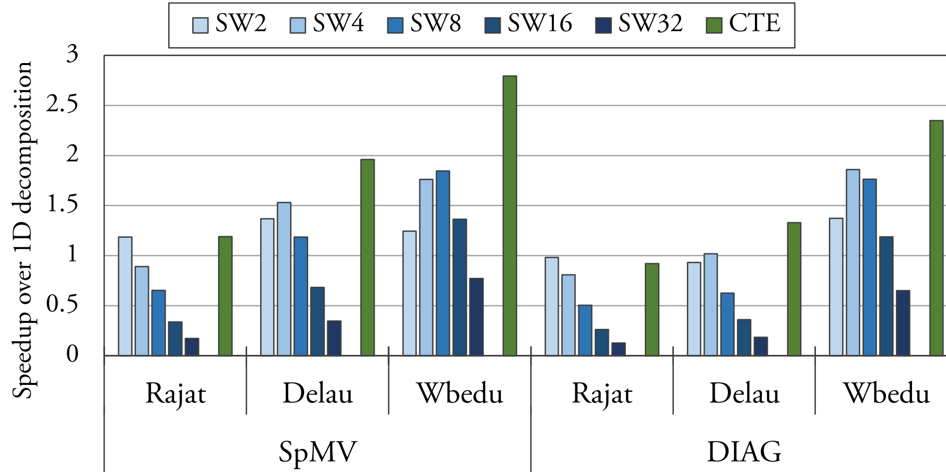


Figure 4.21: The kernel execution speedup of CTE and sub-warp decomposition over 1D decomposition for matrix operations on real-world matrices.

technique. CTE speedup becomes more compelling in the light of sub-warp decomposition speedup over 1D decomposition which can be less than 1 (for sub-warp width 32) and maximize at 1.8x. The CTE supremacy is explained via Figure 4.22 in which CTE shows 87% and 96% warp execution efficiency for SpMV and DIAG respectively while 1D decomposition warp efficiency does not exceed 20% and sub-warp decomposition warp efficiency for different sub-warp widths varies greatly from SpMV to DIAG. As we move toward more regular input matrices (Delau and Rajat), the variation in the size of coarse-grained loads reduces hence 1D and sub-warp decomposition exhibit a better warp utilization and perform better. For Delau, CTE provides 1.9x and 1.3x speedup over 1D decomposition by enhancing the warp efficiency 25% and 23% for SpMV and DIAG respectively. It also provides 1.28x and 1.3x speedup over the sub-warp decomposition method with the best sub-warp width. Finally, for Rajat, since the graph is very regular, CCC shows approximately the same speedup as the best sub-warp decomposition width (1.18x) for SpMV.

Scientific applications – In this section, we measured the performance of two scientific applications, Fast Multiple Method [43] (FMM) and Dynamical Quadrature Grids [54] (DQG) when performed using CTE and 1D and sub-warp decomposition methods. The re-

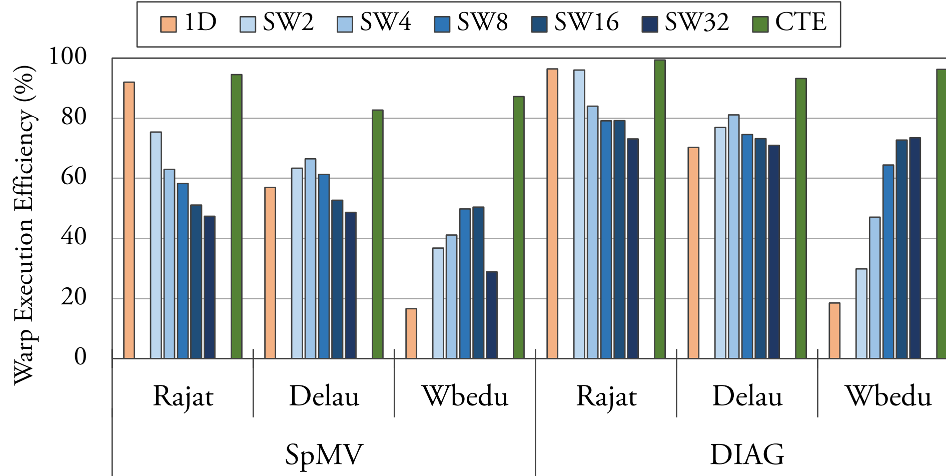


Figure 4.22: Profiled warp execution efficiency of CTE, sub-warp decomposition, and 1D decomposition kernels for experiments in Figure 4.21.

sults are depicted in Figure 4.23. For FMM, which is an n-body approximation that groups the particles in a quad-tree, we consider 10M points in 3D space as the input, and vary the maximum density (Q) of points in each leaf between 5 and 10. We calculate the U-list phase of FMM procedure and distribute the points with non-equal probability distribution (nEquProb) and equal probability distribution (EquProb). It is evident that for two irregular inputs, our technique outperforms both 1D decomposition and sub-warp decomposition by up to 1.5x and 1.15x-1.67x. However, for the regular input sub-warp decomposition with sub-warp width 16 shows slightly better performance. Moreover, DQG computes the points inside a quadrature grid. For DQG, we vary the maximum number of atoms per molecule between 20 and 40 and provide regular and irregular input sets. For the regular input, the number of atoms in molecules are randomly selected between 1 and maximum allowed. For irregular inputs, they are selected using normal distribution. Similar to FMM, irregularity in inputs manifests the CCC supremacy while for a regular input CTE performs on-par with the best sub-warp decomposition width. Also, since compared to FMM, DQG kernel has a more compute-intensive map portion, resulting speedups are slightly higher, covering CTE overhead of re-bundling fine-grained tasks with their corresponding coarse-grained ones.

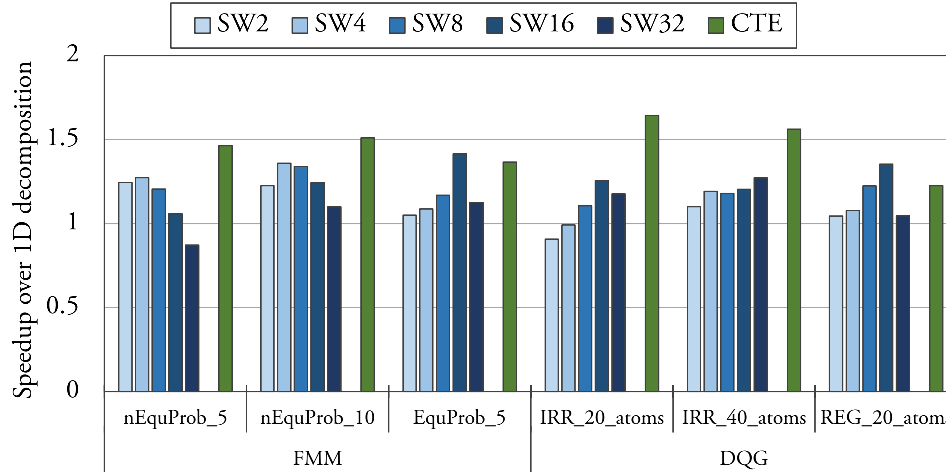


Figure 4.23: The kernel execution speedup of CTE and sub-warp decomposition over 1D decomposition for Fast Multiple Method [43] and Dynamical Quadrature Grids [54] with different inputs.

Graph Analytics – Figure 4.24 shows the kernel execution speedup of CTE and sub-warp decomposition over 1D decomposition for 3 graph applications (BFS, SSSP, and PageRank [65]) over 3 real-world graphs. These graphs have different number of nodes and edges and exhibit various degree distribution patterns. A coarse-grained task in this case processes a node which includes visiting its neighbors as the fine-grained tasks. For this section, we used the sub-warp decomposition implementation in [42] and hand-wrote 1D decomposition. First, LiveJournal [4] (LiveJ) has around 4.85M nodes and 69.0M edges and has a power-law degree distribution. CCC shows better performance for this graph in all application by being 1.30x, 1.08x, and 1.34x better than the best sub-warp decomposition option for BFS, PageRank, and SSSP respectively. Also note that the best sub-warp for different applications differ; this signifies the need for try-and-error or profiling in sub-warp decomposition for every algorithm and input combination. Second, HiggsTwitter [18] (Higgs) is even more irregular compared to LiveJournal and contains 0.46M nodes and 14.8M edges. The results for this graph demonstrate the ineffectiveness of 1D decomposition confronting heavy amount of load imbalance in nested patterns. Finally, RoadNetCA [51] (RoadN) with 1.96M nodes and 5.53M edges is an example of a regular input for the benchmarks

due to its internal connectivity. Most of the nodes in this graph have approximately 1 to 4 neighbors. Therefore, for this graph, 1D decomposition usually performs the best since coarse-grained tasks have roughly equal amounts of fine-grained loads. However, even for this regular pattern, CTE exhibits performance in-par with 1D decomposition by 0.92x, 0.96x, and 1.03x speedup for BFS, PageRank, and SSSP respectively. Also, it is clear that sub-warp decomposition performance becomes worse as we increase the sub-warp size due to over-subscription.

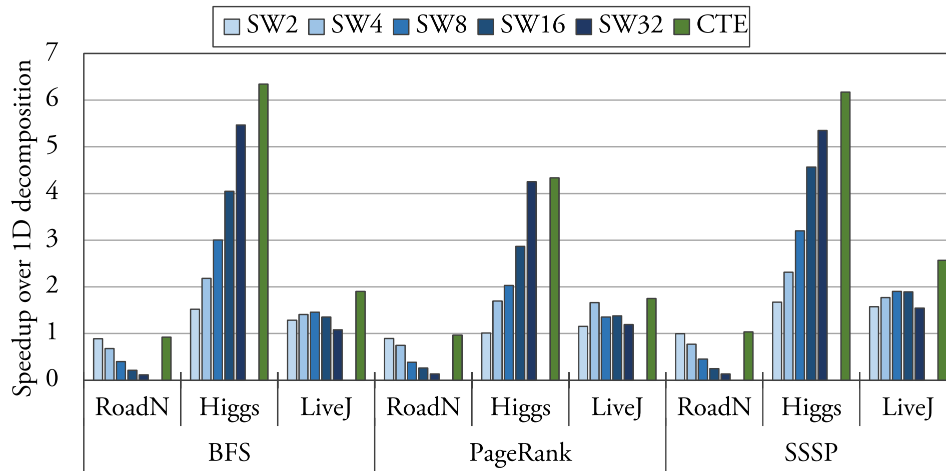


Figure 4.24: The kernel execution speedup of CTE and sub-warp decomposition over 1D decomposition for different graph applications and inputs.

4.2.4 Sensitivity Analysis: varying coarse-grained task sizes

In this section, we analyze the performance of 1D, sub-warp decomposition, and CTE with two synthetic compute-intensive scenarios. The first scenario assigns each thread inside the warp a task size linearly proportional to its $laneID$. Whereas, in the second scenario, the task sizes are proportional to $laneID^2$. These two scenarios are distinguished in Figure 4.25 with $LINE$ and $QUAD$ respectively. Note that in sub-warp decomposition threads calculate their task size using their sub-warp ID so that kernels for all decomposition

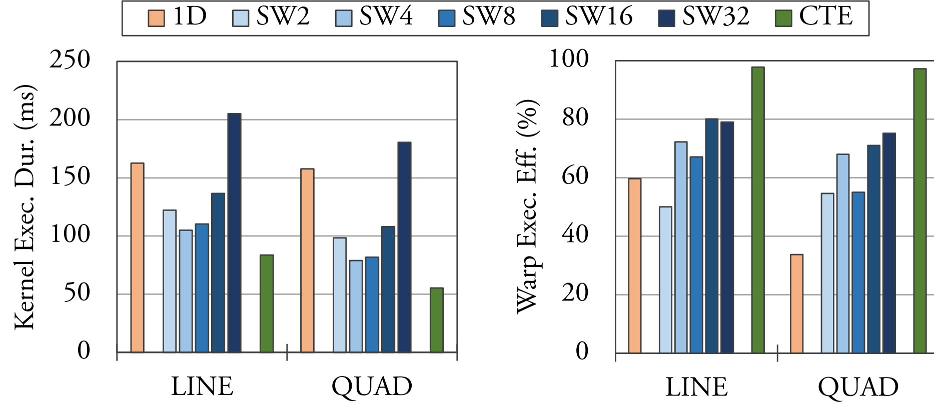


Figure 4.25: Kernel execution duration (left plot) and Warp execution efficiency (right plot) for decomposition methods when the task sizes vary linearly and quadratically proportional to the lane index. Map and reduce portion of the fine-grained tasks each contain 20 FMAD instructions. For the LINE scenario, the coarse-grained task size is calculated with $4 \times laneID$ while for the QUAD scenario it is calculated with $\frac{laneID^2}{8}$. Task sizes for the sub-warp decomposition are calculated using their sub-warp index.

methods get the same overall task sizes for a fair comparison. The coefficients for these two scenarios are selected so that they give approximately the same overall kernel execution duration for the 1D decomposition.

While 1D decomposition kernel takes the same amount of time for both LINE and QUAD scenarios to finish, as it is shown in Figure 4.25, average number of fine-grained tasks per a coarse-grained task for LINE and QUAD are approximately 60.8 and 40.5. This confirms our previous statement about the kernel duration being a function of the maximum of coarse-grained load sizes in 1D decomposition. Also, by making the loads more irregular (LINE vs QUAD), warp execution efficiency for 1D decomposition kernel halves, demonstrating its vulnerability to the intra-warp load imbalance. For sub-warp decomposition, although the kernel duration for different sub-warp sizes reduces by moving from LINE to QUAD by 12 to 24 percent, it does not reflect 33% reduction in the load size. On the other hand, CTE kernel duration drops by 33% confirming the CTE performance dependency to the average of the loads. Plus, unlike sub-warp decomposition that exhibits varying warp execution efficiency for different sub-warp widths in both scenarios, CTE exhibits excellent warp efficiency.

4.3 Summary

This chapter extended the applicability of our graph computation techniques and generalized them as a compiler solution and a template library for SIMD execution efficiency enhancement in CUDA applications. First it introduced Collaborative Context Collection (CCC) that overcomes the SIMD inefficiency of GPU kernels containing thread divergence due to intra-warp load imbalance or dissimilar task assignment. CCC collects the context of divergent threads at the stacks inside the shared memory and retrieves them such that a uniform task is performed by all the warp lanes. This chapter proposed transformations to extend the applicability of CCC to various program patterns, such as recursive functions, and also presented optimizations to enhance CCC performance and to avoid CCC potential side-effects such as occupancy limitation. Moreover, CCC's implementation as a compiler optimization relieves the user from developing the technique from the scratch.

The second section of this chapter introduced Collaborative Task Engagement (CTE) for efficient expression and execution of GPU kernels containing nested patterns. Unlike existing solutions where static assignment of threads to tasks does not provide portable application performance across multiple inputs and induces warp underutilization, CTE assigns threads inside the warp to process a group of tasks collaboratively. Consecutive threads process the consecutive fine-grained tasks resulted from the expansion of coarse-grained tasks, determine the coarse-grained task they belong to, and participate in parallel reduction with their neighbors. Packaging Collaborative Task Engagement as a CUDA C++ device-side template library facilitated its employment in arbitrary GPU application.

Chapter 5

Related Work

This chapter discusses the research in domains addressed by this thesis. First, we provide the related work on graph processing on GPUs, and then, summarize the general software and microarchitectural solutions addressing thread divergence issue.

5.1 Graph Processing on GPUs

Over the past decade there has been a great deal of interest in utilizing GPUs as general purpose accelerators to perform graph processing. The work of Harish and Narayanan [30] was the first step in this direction presenting a simple parallelization scheme that gives each CUDA thread a vertex to process. One of the paths later works took was to diminish the SIMD inefficiency due to irregularity of real-world graphs. Hong et al. [34] proposed dividing the physical warp into smaller virtual warps so as to provide finer control over the task assignment. However, this approach, which was inspired by the work of Bell and Garland for sparse matrix vector multiplication [8], can also suffer from the warp underutilization. Later works endeavored to fix this problem not only for BFS [56, 60] but also for other graph algorithms such as SSSP [16], Betweenness Centrality [57], and PageRank [88]. Nonetheless, algorithm-oriented specializations limit applicability and hence wide deployment of these methods.

For generic graph processing, CuSha [42] framework utilizes edge-centric representations namely G-Shards and Concatenated Windows (CW) to perform user-defined vertex-centric computation with coalesced memory accesses. However, not only these representations consume 2x to 2.5x more space compared to CSR format, their use inhibits work-efficiency. In this thesis we mitigated the space consumption issue by introducing a SIMD-friendly technique named Warp Segmentation [41] but work-inefficiency issue still exists in its framework. Both of the above solutions perform redundant computations on inactive vertices and their belonging edges. As we elaborated in the thesis, we overcame this issue by accompanying the CSR representations with auxiliary data structures and CSC representation, and provided a dynamic thread assignment for intra-warp underutilization avoidance. GunRock [84] is a recent work-efficient data-centric model for which the user has to provide algorithms by focusing on the operations on the frontiers. As an approach conforming to the push-based model, it heavily relies on atomics to indicate the computation, which lowers the expressiveness and hinders scaling to multiple GPUs. In contrast, the solutions in this thesis choose a pull-based model which avoids these issues.

The importance of graph computation has also lead to creation of Domain-Specific Languages. Green-Marl [33] is an example of a DSL that provides instructions and primitives to accelerate the graph processing on multi-core machines. Falcon [10] is another DSL that allows utilizing GPUs for this purpose as well. Falcon is an example of a hybrid solution in which CPU and GPU cooperate to offload the graph processing task. In [35], authors propose a scheme for heterogeneous BFS graph traversal that utilizes CPUs for early iterations with low activation and then GPUs for later iterations where most of the vertices need visitation. TOTEM [24] views the host as another accelerator and statically distributes the graph between the devices. Recently, GraphReduce [78] proposed utilizing host memory for scaling the computation to graphs exceeding the size of GPU's DRAM. Similarly, GTS [45] exploits PCIe-connected SSDs to *stream* very large graphs into GPUs in order to harness its massive parallel processing power. This is an interesting aspect which is orthogonal to our work and can be employed simultaneously with our approach.

[58, 64] propose solutions for processing the graphs that mutate during the computation. In contrast, the focus of this dissertation is algorithms that do not change the structure of the graph.

From a data structure point-of-view, *vertex grouping*, discussed in Section 3.1.2, can be viewed as a form of input reduction which has recently become popular among researchers [46]. In addition, *vertex grouping* is the equivalent of constructing the CSC representation for the reduced adjacency matrix of the graph, and therefore, resembles the idea of mipmaps [85] that is widely used in Computer Graphics. Here, instead of an image and its pixels, the graph's adjacency matrix and the vertices inside it are reduced. Furthermore, the idea of using a bitmask and atomically updating it has been used in multi-core BFS graph exploration [1, 75] as well as GPU hashing [38]. The bitmasks in these solutions are used to imitate the behavior of a bloom filter without false positives. Various other graph representation formats have been proposed that are typically beneficial for targeted applications. For instance, [58] introduces a novel idea of using sparse bit vectors, a structure similar to linked list. However, this representation is highly space inefficient and is only beneficial for morph algorithms when data accesses patterns exhibit spatial locality. [88] tries to balance the load in graphs represented in CSR format by reorganizing the vertices and putting them in three bins. Based on the size of these bins, appropriate number of GPU threads are assigned to process these bins, hence providing a balanced workload distribution.

Dymaxion [12] is an API to improve memory access patterns on GPUs. It uses two fundamental techniques to leverage high memory coalescing:

- **Data restructuring:** Although this method is effective and quite common [70], its use in Dymaxion is limited to predictable data patterns, such as transformation of two-dimensional matrices from row-major order to column-major order or vice versa.

- **Memory remapping:** Allows efficient accessing of data elements via an intermediate mapping function.

Zhang et al. [93] present *data reordering* and *job swapping* techniques to remove GPU memory access irregularities. *Data reordering*, similar to *data restructuring*, repositions elements of an array to minimize required global memory transactions. In *job swapping*, threads exchange work in order to achieve more coalesced memory accesses. It is usually done using reference redirection, which is similar to *memory remapping*. Despite their benefits for applications with regular chunkable input data, irregular and unpredictable dependency between real-world graph elements makes it costly to employ these techniques for graph applications. Recently, Wu et al. [87] classified and analyzed few fundamental methods such as *duplication*, *reordering*, and *sharing* to minimize non-coalesced memory accesses.

5.2 SIMD Thread Divergence

Next we discuss microarchitectural and software solutions to address the SIMD thread divergence problem on GPUs, and compare them with CCC and CTE techniques.

Microarchitectural Solutions

Although these techniques cannot be exploited on available hardware, clever solutions can guide future designs. *Dynamic Warp Formation* (DWF) [22] is the basis for many microarchitectural solutions. DWF merges threads from different warps but with the same PC to form new warps with no thread divergence. To enhance DWF performance, Meng et. al. propose *Dynamic Warp Subdivision* (DWS) [59] and Rhu et. al. [68] suggest *SIMD lane permutation* (SLP). Furthermore, Narasiman et. el [62] suggest *Large Warp Microarchitecture* (LWM) in which fewer but wider warps can create sub-warps that match SIMD width size when facing branch divergence.

Compaction techniques have also been proposed to remedy the SIMD divergence problem. Fung et. al. [23] offer *Thread Block Compaction* (TBC) to exploit control flow locality between threads of a block for divergent paths. Unlike our solution CCC, TBC

makes the warps synchronize at divergent branches to provide homogeneous tasks for warp lanes. To avoid the overhead of unnecessary compaction on non-diverging branches or workloads, Rhu et. al. [66] propose CAPRI, a *compaction-adequacy predictor* influenced by branch predictors. Moreover, Vaidya et. al [82] attempted to harvest dead execution cycles and position SIMD channels in order to group enabled channels together.

Other microarchitectural techniques focus on efficient scheduling for thread divergence and are complementary to CCC. Kim and Batten [44] propose a fine-grained hardware worklist that acts as a distributed queue to provide load balance in data-driven computations. Doubling stage resources in processing pipelines has also been popular [9]. Rhu and Erez [67] examine a dual-path execution model provided by two PC reconvergence stacks and two register scoreboards in order to expose the warp scheduler to more parallelism when facing divergent execution paths. To extend this solution, [20] replaces the reconvergence stack with two warp split and warp reconvergence tables. Rogers et. al. [69] propose *Divergence-Aware Warp Scheduling* (DAWS) for a cache-conscious warp scheduling upon divergence. Also, [19] and [36] suggest reconvergence methods for GPU kernels with unstructured and recursive control flow.

Software Solutions

Not requiring hardware modifications, software solutions for thread divergence are of great importance; however, existing strategies introduce limitations that restrict their usage. Branch and data herding [74] eliminates branch divergence by guiding all the threads in the SIMD group to take the path with the majority vote. In return herding expects and accepts errors in the output. Similarly, [28], [13], and [21] steer the warp lanes to take one execution path. The problem with such techniques is the lack of systematic reliability and applicability. Unlike CCC, these approaches do not take methodical measures to cope with the divergence problem, do not guarantee utilizing all the warp lanes by relying upon warp lanes majority voting, do not devise task accumulation strategies, and need information from the program and the input to schedule the traversal of divergent paths.

These issues prevent wide employment of these solutions. On the other hand, CCC and its transformation and optimization techniques offer methodical approaches to guarantee warp execution enhancement in divergent GPU kernels and can be completely realized and implemented in compile time.

Zhang et. al. [92] try to eliminate thread divergence via thread-data remapping. Unfortunately, this solution not only needs global memory accesses to realize the redirected position of the appropriate data for the warp lanes, it does not preserve GPU kernel autonomy by involving the CPU. Bauer et. al. [6] suggest an intra-CTA producer-consumer model based on which warps can have unique tasks for their threads. As opposed to CCC, this model does not support irregular data-dependent tasks; in other words, the quantity of each task has to be known at compile-time. Tzeng et.al. [81] propose a task management mechanism for irregular parallel workloads based on task donation and stealing. Nonetheless, this technique suffers from GPU underutilization and heavy use of global queues and associated global locks. Merrill et al. [60] enhance the warp execution efficiency of BFS graph traversal via efficient expansion of unequal adjacency lists. Our work is different from [60] in two major ways. First, thread divergence problem in [60] appears as a form of load imbalance on the tasks while it is assumed all the tasks will be carried out. However, CCC allows the existence of conditionals on the tasks in addition to imbalance loads. Second, CCC does not require additional storage to collect the frontiers; instead, it defers processing of tasks via stacking. Our work can also be viewed as a form of in-place stream compaction and consumption.

The load imbalance and consequently thread divergence caused by nested parallelism in GPUs have also been the subject of recent work. Han et. al. [29] introduce loop merging to reorder the code blocks inside a loop with varying trip-count and improve the performance ; however, unlike CCC, the solution does not guarantee full warp execution efficiency. Yang and Zhou created CUDA-NP [91], a source-to-source compiler that transforms GPU codes with parallel sections using the idea of master and slave threads. However, fixed number of slave threads for a master thread can hurt the performance in

irregular workloads. In [48] Lee et. al. also propose a framework supporting a number of widely-used parallel patterns for efficient nested parallelism. [37] introduces warp-aware trace scheduling for GPUs based on speculating loads and arithmetic instructions upon divergence in order to exploit ILP. Recently, Schaub et. al. [76] evaluated compiler techniques that aim to mitigate divergence against larger SIMD widths. [14] and [71] offer static code analyzers helping GPU developers to optimize the code manually. Also, [15] and [73] provide profile-guided approaches to recognize and then optimize code regions exhibiting divergence. These works complement our work.

In addition to the methods and works mentioned in Section 4, queue-based approaches are also implemented to handle irregularities of task loads. [81] gives a dynamic decomposition scheme based on task stealing and donation using queues. However, the implementation of such queues involves heavy contention over the atomic variable and also over global locks that have to be passed around spawned CTAs. Design that works around a central lock, such as [56] for BFS graph traversal, imposes inefficiencies due to latencies. Early works of graph computation on CUDA platform [30] employ 1D decomposition and assign every coarse-grained task (e.g., processing of a graph node) to one GPU thread. The thread then iterates over fine-grained tasks (e.g., visiting the node's neighbors). Later, [34] presented sub-warp decomposition for graph algorithms and named each sub-warp a virtual warp. Similarly, virtual warps have a fixed power-of-2 size throughout the kernel computation. While only one thread inside the virtual warp performs the SISD (Single Instruction Single Data) phase of the kernel, all the threads inside the virtual warp participate in the SIMD phases. Merril et.al. [60] realized the significance of load balancing with parallel scan [61] in sparse graph processing. However, their solution is limited to BFS graph traversal and does not consider the interaction between fine-grained tasks of a coarse-grained task. In comparison to Warp Segmentation [41], CTE [39] breaks the associativity between fine-grained and coarse-grained tasks, formulates the idea of expansion of fine-grained tasks, and generalizes the solution by introducing an efficiently implemented template library.

Xiang et. al. examined the effect of inter-warp load imbalance in [89], in which they referred to it as *warp-level divergence*. This solution compliments intra-warp decomposition methods (1D, sub-warp, CTE); even though inter-warp load imbalance effect is insignificant especially for GPU kernels with high occupancy. Similar to CTE library, Thrust [32] is a CUDA C++ template library that provides interfaces such as `thrust::for_each` for the expression of iterative code segments. However, underlying scheme to carry out the fine-grained tasks in nested patterns is 1D decomposition.

Load imbalance, as a problem for SIMT architecture, is generally a variation of thread divergence; thus, offered solutions for divergence are of importance for irregular nested parallel patterns. Collaborative Lanes [38] is a method to overcome intra-warp underutilization during batched insertions in GPU Hashing. To mitigate thread divergence, [21] schedules the path in the program that most threads take using `__all()` and `__any()` CUDA primitives; however, it fails to provide full warp utilization. Other solutions that rely on majority voting, [28, 29, 74], attempt to eliminate thread divergence similarly by enforcing all or none of the threads to take the divergent path. While [28, 29] require information from the program to schedule the execution of divergent path, [74] approximates the final outcome and accepts errors in the output. Moreover, CCC [40] implements an efficient all-or-none discipline for repetitive divergent tasks. Warp specialization [6] is another method to overcome thread divergence but only when there are tasks for threads within a warp that are of differing nature. Furthermore, data remapping techniques [92, 93] may reduce divergence at the expense of static analysis or disrupting GPU kernel autonomy. [73, 15, 71] aim to mitigate the effect of divergence by profiling the GPU application. Profile-guided approaches are orthogonal to our technique and can be applied contemporaneously.

Chapter 6

Conclusions and Future Work

6.1 Contributions

This dissertation enables high performance vertex-centric graph analytics on GPUs by addressing the challenges of SIMD-efficiency, scalability, and work-efficiency. *Warp Segmentation* provides dynamic thread assignment when visiting neighbors of vertices and *Vertex Refinement* eliminates the unnecessary inter-GPU data transfer during iterations. Moreover, this dissertation presented data structures for enabling work-efficiency and extended *Warp Segmentation* to sustain the high warp execution efficiency in presence of disjointed adjacency lists. It also suggested *Vertex Grouping* and *Permissive Partitioning* for memory overhead reduction and dynamic inter-GPU load balancing respectively. Finally, this dissertation discussed the generalization of graph processing techniques by proposing the CCC compiler technique and the CTE template library.

Addressing Warp Efficiency and Scalability

This thesis addressed the problems of SIMD efficiency and scalability for vertex-centric graph processing on GPUs. Due to the irregularity of real-world graphs, effective utilization of GPU's SIMD environment is challenging. We proposed *Warp Segmentation* (WS), a novel technique for compact graph representations that overcomes SIMD underuti-

lization during graph processing. In addition, we introduced *Vertex Refinement* that enables effective scaling of the graph processing procedure to multiple GPUs. It efficiently filters updated vertices of a GPU on-the-fly via high performance CUDA primitives, and therefore, unlike previous approaches that waste a great deal of PCIe bandwidth, it maximizes inter-device communication efficiency.

Enabling Work Efficiency

We enabled *work-efficiency* in iterative vertex-centric graph processing on one or multiple GPUs, thus conserving processing power and spending it only on vertices whose values are subject to change in the current iteration. We equipped the GPU kernels with a dynamic task assignment technique that efficiently maps warp threads to the elements of disjoint adjacency lists of active vertices and achieves a high warp execution efficiency. We offered *vertex grouping* that enables a trade-off between required global memory consumption for directed graphs and the work-efficiency during the computation, essentially allowing processing of larger directed graphs on GPUs. For multi-GPU computations, we presented *permissive partitioning* that allows overlap between graph segments stored in GPUs enabling dynamic inter-GPU load balancing. We packaged the above techniques in KiTES, the first GPU graph analytics framework implemented as a CUDA C++ template library. This design greatly facilitates the employment and integration of our methods with the user's code while providing options through the library API to manage the graph computation.

Extending Techniques to Other GPU Applications

We generalized two of our graph processing techniques as a compiler optimization and a template library and make them applicable to other GPU applications exposing irregularities. We proposed *CCC*, a compiler technique for CUDA programs that boosts the warp execution efficiency upon divergence. CCC collects tasks at warp granularity and remedies inefficiency due to intra-warp load imbalance or dissimilar task assignment.

To enhance the applicability of CCC, we present transformations to make common code patterns accessible to CCC and develop optimizations to increase the CCC performance.

We also proposed *Collaborative Task Engagement* (*CTE*), a novel task decomposition technique to efficiently process irregular nested parallel patterns in GPUs. Unlike previous methods, warp threads in CTE pass over the expanded list of fine-grained tasks, making it resilient against input irregularities. We developed a CUDA C++ device-side template library for easy-expression of nested patterns with CTE.

6.2 Future Directions

The youth of GPGPU computing and the constant evolvement of GPU microarchitecture accompanied with the emergence of big data analytics will cause graph computation on GPUs to remain an interesting topic to explore.

Graph Processing Challenges in Upcoming Technologies

The recent introduction of High Bandwidth Memory (HBM) technology on GPUs has opened up a great room for improving graph processing applications. Memory access bandwidth enhancement will cause the improved SIMD efficiency to further boost the overall performance of processing even larger graphs. Studying this improvement could give us insights that will lead to better design of the data structure and the approach for graph computation. In addition, recently introduced NVLink technology improves the inter-GPU and host-GPU communication bandwidth up to 16 times. Investigating the simultaneous effect of HBM and NVLink in multi-GPU graph processing and its comparison with the traditional approaches would provide a better understanding of the characteristics of the computation. Also, the inter-connectivity pattern of GPUs and the host with NVLink and modifying it based on the graph structure is an interesting topic to look at in the future.

Processing Graphs In-Memory

Contemporary advancements in 3D-stacked memory technology has revived the Processing-In-Memory (PIM) approach which reduces the off-chip data communication and hence increases performance and power efficiency. However, the graph data structure placement for an effective computation on PIM processors is challenging since the memory available on each stack is limited and inter-PIM communication is costly. These problems are similar to ones introduced in this thesis and we wish to explore the application of our proposed solutions to graph computation on PIM in the future.

Dynamically Evolving Graphs

The solutions introduced in this thesis focus on the algorithms that do not change the structure of the graph, hence, extending them to graph algorithms that change the structure of the graph is an interesting future direction. One of the main challenges for computing evolving graphs with GPUs is having an efficient representation that not only incurs minimal foorprint but also allows fast deletion and addition of graph components. This requires exploring the ways for redefining our dynamic load assignment and graph distribution schemes for the new representation.

Bibliography

[1] Virat Agarwal, Fabrizio Petrini, Davide Pasetto, and David A. Bader. Scalable graph exploration on multicore processors. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11. IEEE Computer Society, 2010.

[2] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on gpus. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 145–149. ACM, 2009.

[3] Dan A. Alcantara, Vasily Volkov, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, and Nina Amenta. Chapter 4 - building an efficient hash table on the {GPU}. In Wen-mei W. Hwu, editor, *{GPU} Computing Gems Jade Edition*, Applications of GPU Computing Series, pages 39 – 53. Morgan Kaufmann, 2012.

[4] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. Group formation in large social networks: Membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pages 44–54. ACM, 2006.

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

[6] Michael Bauer, Sean Treichler, and Alex Aiken. Singe: Leveraging warp specialization for high performance on gpus. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 119–130. ACM, 2014.

[7] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on cuda. Technical report, Nvidia Technical Report NVR-2008-004, Nvidia Corporation, 2008.

[8] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 18:1–18:11. ACM, 2009.

[9] Nicolas Brunie, Sylvain Collange, and Gregory Diamos. Simultaneous branch and warp interweaving for sustained gpu performance. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 49–60. IEEE Computer Society, 2012.

[10] Unnikrishnan C, Rupesh Nasre, and Y. N. Srikant. Falcon: A graph manipulation language for heterogeneous systems. *ACM Trans. Archit. Code Optim.*, 12(4):54:1–54:27, December 2015.

[11] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. *R-MAT: A Recursive Model for Graph Mining*, chapter 43, pages 442–446.

[12] Shuai Che, J.W. Sheaffer, and K. Skadron. Dymaxion: Optimizing memory access patterns for heterogeneous systems. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–11, 2011.

[13] Dan Connors, Skyler Saleh, Tejas Joshi, and Ryan Bueter. Data-driven techniques to overcome workload disparity. In *Proceedings of the Fourth Workshop on Irregular Applications: Architectures and Algorithms*, IA3 '14, pages 41–48. IEEE Press, 2014.

[14] B. Coutinho, D. Sampaio, F.M.Q. Pereira, and W. Meira. Divergence analysis and optimizations. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 320–329, Oct 2011.

[15] Bruno Coutinho, Diogo Sampaio, Fernando M. Q. Pereira, and Wagner Meira. Profiling divergences in gpu applications. *Concurrency and Computation: Practice and Experience*, 25(6):775–789, 2013.

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

[17] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011.

[18] M. De Domenico, A. Lima, P. Mougel, and M. Musolesi. The anatomy of a scientific rumor. *Sci. Rep.*, 3, 2013.

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

[20] A. ElTantawy, J.W. Ma, M. O'Connor, and T.M. Aamodt. A scalable multi-path microarchitecture for efficient gpu control flow. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 248–259, Feb 2014.

[21] S. Frey, G. Reina, and T. Ertl. Simt microscheduling: Reducing thread stalling in divergent iterative algorithms. In *Parallel, Distributed and Network-Based Processing (PDP), 2012 20th Euromicro International Conference on*, pages 399–406, Feb 2012.

[22] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic warp formation and scheduling for efficient gpu control flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 407–420. IEEE Computer Society, 2007.

[23] W.W.L. Fung and T.M. Aamodt. Thread block compaction for efficient simt control flow. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 25–36, Feb 2011.

[24] Abdullah Gharaibeh, Lauro Beltrão Costa, Elizeu Santos-Neto, and Matei Ripeanu. A yoke of oxen and a thousand chickens for heavy lifting graph processing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 345–354. ACM, 2012.

[25] Abdullah Gharaibeh, Elizeu Santos-Neto, Lauro Beltrão Costa, and Matei Ripeanu. Efficient large-scale graph processing on hybrid CPU and GPU systems. *CoRR*, abs/1312.3018, 2013.

[26] Mike Giles. Chapter 10 - approximating the erfinv function. In Wen-mei W. Hwu, editor, *{GPU} Computing Gems Jade Edition*, Applications of GPU Computing Series, pages 109 – 116. Morgan Kaufmann, 2012.

[27] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, Hollywood, CA, 2012. USENIX.

[28] Tianyi David Han and Tarek S. Abdelrahman. Reducing branch divergence in gpu programs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pages 3:1–3:8. ACM, 2011.

[29] Tianyi David Han and Tarek S. Abdelrahman. Reducing divergence in gpgpu programs with loop merging. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, pages 12–23. ACM, 2013.

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

[31] Mark Harris and Michael Garland. Chapter 3 - optimizing parallel prefix operations for the fermi architecture. In Wen-mei W. Hwu, editor, *{GPU} Computing Gems Jade Edition*, Applications of GPU Computing Series, pages 29 – 38. Morgan Kaufmann, Boston, 2012.

[32] J Hoberock and N Bell. Thrust: A parallel template library, 2015. Version 1.8.1.

[33] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-marl: A dsl for easy and efficient graph analysis. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 349–362. ACM, 2012.

[34] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating cuda graph algorithms at maximum warp. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 267–276. ACM, 2011.

[35] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. Efficient parallel graph exploration on multi-core cpu and gpu. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 78–88. IEEE Computer Society, 2011.

[36] Xin Huo, Sriram Krishnamoorthy, and Gagan Agrawal. Efficient scheduling of recursive control flow on gpus. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 409–420. ACM, 2013.

[37] James A. Jablin, Thomas B. Jablin, Onur Mutlu, and Maurice Herlihy. Warp-aware trace scheduling for gpus. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 163–174. ACM, 2014.

[38] F. Khorasani, M. E. Belviranli, R. Gupta, and L. N. Bhuyan. Stadium hashing: Scalable and flexible hashing on gpus. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 63–74, Oct 2015.

[39] F. Khorasani, B. Rowe, R. Gupta, and L. N. Bhuyan. Eliminating intra-warp load imbalance in irregular nested patterns via collaborative task engagement. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 524–533, May 2016.

[40] Farzad Khorasani, Rajiv Gupta, and Laxmi N. Bhuyan. Efficient warp execution in presence of divergence with collaborative context collection. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 204–215. ACM, 2015.

[41] Farzad Khorasani, Rajiv Gupta, and Laxmi N. Bhuyan. Scalable simd-efficient graph processing on gpus. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 39–50, Oct 2015.

[42] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. Cusha: Vertex-centric graph processing on gpus. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 239–252. ACM, 2014.

[43] Hyesoon Kim, Richard Vuduc, Sara Baghsorkhi, Jee Choi, and Wen-mei Hwu. *Performance Analysis and Tuning for General Purpose Graphics Processing Units*. Morgan & Claypool Publishers, 1st edition, 2012.

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

[45] Min-Soo Kim, Kyuhyeon An, Himchan Park, Hyunseok Seo, and Jinwook Kim. Gts: A fast and scalable graph processing method based on streaming topology to gpus. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 447–461. ACM, 2016.

[46] Amlan Kusum, Keval Vora, Rajiv Gupta, and Iulian Neamtiu. Efficient processing of large graphs via input reduction. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '16, pages 245–257. ACM, 2016.

[47] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 31–46, Hollywood, CA, 2012. USENIX.

[48] HyoukJoong Lee, Kevin J. Brown, Arvind K. Sujeeth, Tiark Rompf, and Kunle Olukotun. Locality-aware mapping of nested parallel patterns on gpus. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 63–74. IEEE Computer Society, 2014.

[49] Jure Leskovec, Lada A. Adamic, and Bernardo A. Huberman. The dynamics of viral marketing. *ACM Trans. Web*, 1(1), May 2007.

[50] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[51] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[52] Shaojing Li, Ruinan Chang, A. Boag, and V. Lomakin. Fast electromagnetic integral-equation solvers on graphics processing units. *Antennas and Propagation Magazine, IEEE*, 54(5):71–87, Oct 2012.

[53] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J.M. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. In *Proceedings of the VLDB Endowment*, pages 716–727, April 2012.

[54] Nathan Luehr, Ivan Ufimtsev, and Todd Martinez. Chapter 3 - dynamical quadrature grids: Applications in density functional calculations. In Wen-mei W. Hwu, editor, {*GPU*} *Computing Gems Emerald Edition*, Applications of GPU Computing Series, pages 35–42. Morgan Kaufmann, 2011.

[55] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(01):5–20, 2007.

[56] Lijuan Luo, Martin Wong, and Wen-mei Hwu. An effective gpu implementation of breadth-first search. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 52–55. ACM, 2010.

[57] Adam McLaughlin and David A. Bader. Scalable and high performance betweenness centrality on the gpu. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 572–583. IEEE Press, 2014.

[58] Mario Mendez-Lojo, Martin Burtscher, and Keshav Pingali. A gpu implementation of inclusion-based points-to analysis. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 107–116. ACM, 2012.

[59] Jiayuan Meng, David Tarjan, and Kevin Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 235–246. ACM, 2010.

[60] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable gpu graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 117–128. ACM, 2012.

[61] Duane Merrill and Andrew Grimshaw. Parallel scan for stream architectures. *University of Virginia, Department of Computer Science, Charlottesville, VA, USA, Technical Report CS2009-14*, 2009.

[62] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt. Improving gpu performance via large warps and two-level warp scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 308–317. ACM, 2011.

[63] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. Atomic-free irregular computations on gpus. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, pages 96–107. ACM, 2013.

[64] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. Morph algorithms on gpus. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 147–156. ACM, 2013.

[65] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: bringing order to the web. 1999.

[66] Minsoo Rhu and Mattan Erez. Capri: Prediction of compaction-adequacy for handling control-divergence in gpgpu architectures. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 61–71, 2012.

[67] Minsoo Rhu and Mattan Erez. The dual-path execution model for efficient gpu control flow. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, HPCA '13, pages 591–602. IEEE Computer Society, 2013.

[68] Minsoo Rhu and Mattan Erez. Maximizing simd resource utilization in gpgpus with simd lane permutation. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 356–367. ACM, 2013.

[69] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. Divergence-aware warp scheduling. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 99–110. ACM, 2013.

[70] Mehrzad Samadi, Amir Hormati, Mojtaba Mehrara, Janghaeng Lee, and Scott Mahlke. Adaptive input-aware compilation for graphics engines. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 13–22. ACM, 2012.

[71] Diogo Sampaio, Rafael Martins de Souza, Sylvain Collange, and Fernando Magno Quintão Pereira. Divergence analysis. *ACM Trans. Program. Lang. Syst.*, 35(4):13:1–13:36, January 2014.

[72] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming.* Addison-Wesley Professional, 1st edition, 2010.

[73] Santonu Sarkar and Sayantan Mitra. A profile guided approach to optimize branch divergence while transforming applications for gpus. In *Proceedings of the 8th India Software Engineering Conference*, ISEC '15, pages 176–185. ACM, 2015.

[74] John Sartori and Rakesh Kumar. Branch and data herding: Reducing control and memory divergence for error-tolerant gpu applications. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 427–428. ACM, 2012.

[75] Daniele Paolo Scarpazza, Oreste Villa, and Fabrizio Petrini. Efficient breadth-first search on the cell/be processor. *IEEE Trans. Parallel Distrib. Syst.*, 19(10):1381–1395, October 2008.

[76] Thomas Schaub, Simon Moll, Ralf Karrenberg, and Sebastian Hack. The impact of the simd width on control-flow and memory divergence. *ACM Trans. Archit. Code Optim.*, 11(4):54:1–54:25, January 2015.

[77] Christoph Schied, Johannes Hanika, Holger Dammertz, and HendrikP.A. Lensch. Chapter 18 - high-performance iterated function systems. In Wen-mei W. Hwu, editor, {*GPU*} *Computing Gems Emerald Edition*, Applications of GPU Computing Series, pages 263 – 273. Morgan Kaufmann, 2011.

[78] Dipanjan Sengupta, Shuaiwen Leon Song, Kapil Agarwal, and Karsten Schwan. Graphreduce: Processing large-scale graphs on accelerator-based systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 28:1–28:12. ACM, 2015.

[79] Luke Olson Steven Dalton, Nathan Bell and Michael Garland. Cusp: Generic parallel algorithms for sparse matrix and graph computations, 2014. Version 0.5.0.

[80] Lubos Takac and Michal Zabovsky. Data analysis in public social networks. In *International Scientific Conference AND International Workshop Present Day Trends of Innovations*, 2012.

[81] Stanley Tzeng, Anjul Patney, and John D. Owens. Task management for irregular-parallel workloads on the gpu. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, pages 29–37. Eurographics Association, 2010.

[82] Aniruddha S. Vaidya, Anahita Shayesteh, Dong Hyuk Woo, Roy Saharoy, and Mani Azimi. Simd divergence optimization through intra-warp compaction. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 368–379. ACM, 2013.

[83] Jin Wang and Sudhakar Yalamanchili. Characterization and analysis of dynamic parallelism in unstructured gpu applications. In *2014 IEEE International Symposium on Workload Characterization*, October 2014.

[84] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. Gunrock: A high-performance graph processing library on the gpu. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 11:1–11:12. ACM, 2016.

[85] Lance Williams. Pyramidal parametrics. In *Proceedings of the 10th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '83, pages 1–11. ACM, 1983.

[86] Nicholas Wilt. *The cuda handbook: A comprehensive guide to gpu programming*, pages 410–411. Pearson Education, 2013.

[87] Bo Wu, Zhijia Zhao, Eddy Zheng Zhang, Yunlian Jiang, and Xipeng Shen. Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on gpu. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 57–68. ACM, 2013.

[88] Tianji Wu, Bo Wang, Yi Shan, Feng Yan, Yu Wang, and Ningyi Xu. Efficient pagerank and spmv computation on amd gpus. *2014 43rd International Conference on Parallel Processing*, 0:81–89, 2010.

[89] Ping Xiang, Yi Yang, and Huiyang Zhou. Warp-level divergence in gpus: Characterization, impact, and mitigation. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 284–295, Feb 2014.

[90] Jaewon Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. In *Data Mining (ICDM), 2012 IEEE 12th International Conference on*, pages 745–754, Dec 2012.

[91] Yi Yang and Huiyang Zhou. Cuda-np: Realizing nested thread-level parallelism in gpgpu applications. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 93–106. ACM, 2014.

[92] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, and Xipeng Shen. Streamlining gpu applications on the fly: Thread divergence elimination through runtime thread-data remapping. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 115–126. ACM, 2010.

[93] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. On-the-fly elimination of dynamic irregularities for gpu computing. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 369–380. ACM, 2011.

[94] J. Zhong and B. He. Medusa: Simplified graph processing on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1543–1552, June 2014.