# Efficient Scheduling and High-Performance Graph Partitioning on Heterogeneous CPU-GPU Systems

Bahareh Goodarzi

A Thesis
in the Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy (Computer Science) at
Concordia University
Montréal, Québec, Canada

July 2018

# Concordia University

Engineering and Computer Science

This is to certify that the thesis prepared

By: **Bahareh Goodarzi**

Entitled: **Efficient Scheduling and High-Performance Graph Partitioning on Heterogeneous CPU-GPU Systems**

and submitted in partial fulfillment of the requirements for the degree of

**Doctor of Philosophy (Computer Science)**

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

_____ Chair

*Dr. Nizar Bouguila*

_____ External Examiner

*Dr. Michel Dagenais*

_____ External to Program

*Dr. Anjali Agarwal*

_____ Examiner

*Dr. Brigitte Jaumard*

_____ Examiner

*Dr. Hovhannes Harutyunyan*

_____ Supervisor

*Dr. Dhrubajyoti Goswami*

Approved by _____

Dr. Volker Haarslev, Graduate Program Director

Tuesday, August 21, 2018 _____

Dr. Amir Asif, Dean

Faculty of Engineering and Computer Science

# Abstract

## Efficient Scheduling and High-Performance Graph Partitioning on Heterogeneous CPU-GPU Systems

**Bahareh Goodarzi, Ph.D.**

**Concordia University, 2018**

Heterogeneous CPU-GPU systems have emerged as a power-efficient platform for high performance parallelization of the applications. However, effectively exploiting these architectures faces a number of challenges including differences in the programming models of the CPU (MIMD) and the GPU (SIMD), GPU memory constraints, and comparatively low communication bandwidth between the CPU and GPU. As a consequence, high performance execution of applications on these platforms requires designing new adaptive parallelizing methods. In this thesis, first we explore embarrassingly parallel applications where tasks have no inter-dependencies. Although the massive processing power of GPUs provides an attractive opportunity for high-performance execution of embarrassingly parallel tasks on CPU-GPU systems, minimized execution time can only be obtained by optimally distributing the tasks between the processors. In contemporary CPU-GPU systems, the scheduler cannot decide about the appropriate rate distribution. Hence it requires high programming effort to manually divide the tasks among the processors. Herein, we design and implement a new dynamic scheduling heuristic to minimize the execution time of embarrassingly parallel applications on a heterogeneous CPU-GPU system. The scheduler is integrated into a scheduling framework that provides pre-implemented automated scheduling modules, liberating the user from the complexities of scheduling details. The experimental results show that our scheduling approach achieves better to similar performance compared to some of the scheduling algorithms proposed for CPU-GPU systems. We then investigate task dependent applications, where the

tasks have data dependencies. The computational tasks and their communication patterns are expressed by a task interaction graph. Scheduling of the task interaction graph on a cluster can be done by first partitioning the graph into a set of computationally balanced partitions in such a way that the communication cost among the partitions is minimized, and subsequently mapping the partitions onto physical processors. Aside from scheduling, graph partitioning is a common computation phase in many application domains, including social network analysis, data mining, and VLSI design. However, irregular and data-dependent graph partitioning sub-tasks pose multiple challenges for efficient GPU utilization, which favors regularity. We design and implement a multilevel graph partitioner on a heterogeneous CPU-GPU system that takes advantage of the high parallel processing power of GPUs by executing the computation-intensive parts of the partitioning sub-tasks on the GPU and assigning the parts with less parallelism to the CPU. Our partitioner aims to overcome some of the challenges arising due to the irregular nature of the algorithm, and memory constraints on GPUs. We present a lock-free scheme since fine-grained synchronization among thousands of GPU threads imposes too high a performance overhead. Experimental results demonstrate that our partitioner outperforms serial and parallel MPI-based partitioners. It performs similar to shared-memory CPU-based parallel graph partitioner. To optimize the graph partitioner performance, we describe an effective and methodological approach to enable a GPU-based multi-level graph partitioning that is tailored specifically for the SIMD architecture. Our solution avoids thread divergence and balances the load over GPU threads by dynamically assigning an appropriate number of threads to process the graph vertices and irregular sized neighbors. Our optimized design is autonomous as all the steps are carried out by the GPU with minimal CPU interference. We show that this design outperforms CPU-based parallel graph partitioner. Finally, we apply some of our partitioning techniques to another graph processing algorithm, minimum spanning tree (MST), that exhibits load imbalance characteristics. We show that extending these techniques helps in achieving a high performance implementation of MST on the GPU.

# Acknowledgments

First and foremost, I would like to express my special gratitude to my supervisor Dr. Dhrubajyoti Goswami for his invaluable guidance, support, patience, trust and encouragement during the course of my PhD. I am always indebted to him for inspiring me in digging the new research paths and providing me with every bit of guidance, expertise, assistance and valuable understanding. His constructive feedback and guidance, steered me in the right direction throughout my research. I also learned form him professionalism, commitment and compassion.

I would like to thank my committee members, Dr. Brigitte Jaumard, Dr. Hovhannes Harutyunyan and Dr. Anjali Agarwal, for their insightful comments and encouragement, to widen my research from various perspectives.

I am grateful to Dr. Martin Burtscher, form Texas State University and Dr. Farzad Khorasani and Dr. Vivek Sarkar from Georgia Institute of Technology for their effective collaboration during my research work.

I gratefully acknowledge the support of NVIDIA Corporation with the donation of two high performance GPUs used for this research.

I extend my appreciation to NSERC Strategic Grant and Concordia University for the support of my research.

I am greatly thankful to my precious colleagues and friends, Upama, Shayan, Shyam, Mahsa, Nazanin and Myriam for their friendship and continuous support specially when things would get a bit discouraging.

This dissertation would not have been possible without constant support, understanding, patience, love and encouragement of my husband Navid.

I would like to express my deepest appreciation to my amazing family, my mother, Zahra, my Father, Ramin and my lovely sister, Mahsa for their unconditional love and support throughout my life. They have been always my brightest stars in the darkest nights.

Last but not the least, I would like to thank my friends Shagha, Ali, Farnaz, Sya, Shadi and Farnoush who have been always there for me and supported me in the tough moments of this journey.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Overview and Objectives

Heterogeneous multi-core systems are gradually surpassing the homogeneous systems due to their high performance and flexibility. To respond to the high demands for more computational power, these parallel architectures have also been integrated with various heterogeneous technologies (e.g., powerXCell processors, DSPs or GPGPUs). Currently the top 500-leading IBM RoadRunner machines are composed of CPUs and accelerators [26].

The growing computational power of GPUs gives them significantly higher peak computing power compared to other accelerators and it makes them a compelling platform for computationally demanding tasks in a wide variety of application domains. The deployment of GPUs as general purpose accelerators, which started about a decade ago has now become mainstream. General-purpose graphics processing units (GPGPU) allow a host CPU to offload a wide variety of scientific computing applications, not just graphics, to a GPU. The high computing power provided by many-core processing units leverages this parallel architecture for non-graphic computations by achieving high speedup and providing other benefits, such as power efficiency and low cost. Modern GPU [1] processors are massively parallel, and are fully programmable.

[1] In the rest of this thesis we use the term GPU instead of GPGPU

The parallel floating point computing power of a modern GPU is orders of magnitude higher than that of a CPU [67]. Thus, intelligently combining the best features of both has positioned the integrated multi-core CPU and the many-core GPUs as a meritorious alternative to traditional heterogeneous multi-core systems in high performance parallelization of applications.

Figure 1.1 demonstrates the general structure of a heterogeneous CPU-GPU system with one CPU and two GPUs. As the figure shows, the CPU communicates with GPUs through the PCI-X buses. A GPU consists of a set of Streaming Multiprocessors (SM), each of which is comprised of a number of Streaming Processors (SPs). In CUDA programming model [1], which is the most common GPU programming model, the program on the GPU is executed by launching a set of threads across the SMs. Each set of 32 contiguous threads constitutes a warp. The GPU performs SIMD (Single instruction-multiple data) execution at the warp level and all the threads inside a warp execute the same instruction at any given time.

The GPU memory system provides on-chip and off-chip memories. The off-chip memory is generally referred to as the global memory of the GPU, and all the threads running across the SMs have the read/write access to this memory. However, in terms of access latency this memory is slow. The on-chip memory includes the shared memory and thread registers. The shared memory is located on each of the SMs and, consequently is as fast as accessing a register [2].

Due to different programming paradigms of the CPU (MIMD) and GPU (SIMD), optimizing the execution of different applications in terms of performance and efficiency requires considering the characteristics of both architectures for making workload distribution decisions. The difference in the applications characteristics and the proper modification of the existing parallel algorithm also need to be taken into account to fulfill the potential performance of heterogeneous platforms with single or multiple GPUs.

A large group of parallel applications fall into embarrassingly parallel category [21,

[2]We discuss the details of the CPU-GPU systems in Chapter 2

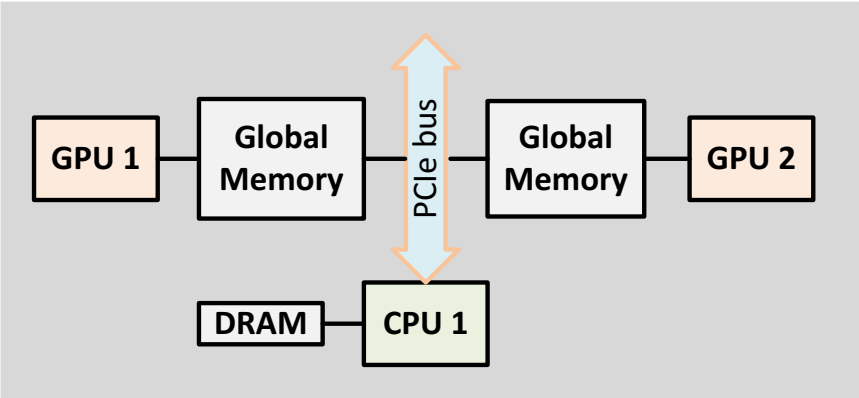



Figure 1.1: A single-CPU multi-GPU system.

66, 83] where tasks have no inter-dependencies and their operations and memory access patterns are regular. Simultaneous executing of these applications workloads on the CPU and the GPUs can result in substantial performance due to the massive computation capability of GPU in the execution of the independent parallel tasks. The minimized execution time, however, can only be obtained if we optimally distribute the tasks over the processors and avoid the idle time for all the processing units.

Several studies [45, 62, 71] showed that the appropriate load distribution over the CPU and GPU raises the efficient computation resources utilization and, consequently, archives better performance in comparison to executing the loads on either the CPU or the GPU. Nonetheless, the traditional GPU scheduler cannot automatize the distribution of workload over the CPU and GPU devices, and the programmer should manually schedule the tasks over the available devices. Furthermore, determining the best distribution rates to ensure that the makespan is minimized is a challenging problem.

The scheduling methods proposed for heterogeneous clusters [8, 13, 63, 87, 98] or multi-core environments [18, 97] are not applicable to CPU-GPU systems in a straightforward way. Unlike the traditional heterogeneous systems, where the heterogeneity comes from difference in the speed or network bandwidths of the processors, in heterogeneous CPU-GPU system, the architecture and programming model of the

processors are different. Other challenges include the transfer latency between the CPU and the GPU, GPU memory limitations and the high programming effort to distribute the workloads manually.

There are several proposed static and dynamic scheduling algorithms on CPU-GPU systems. However, these suffer either from performance or scalability limitations. The static methods [33, 62, 94] are applicable only to a system integrated with a single GPU and have a heavy and inaccurate training phase for large loads. The dynamic techniques [44, 89] underutilize the GPU processing cores, and some of them are not scalable to more than one integrated GPU [84]. The other groups require many scheduling hints from the user, and the scheduler cannot schedule the loads over the processing devices independently and adaptively [82]. As a result, designing a high performance adaptive and scalable scheduling method for embarrassingly parallel applications which minimizes the user interference as well is non-trivial.

Task dependent applications, another category of parallel applications, have data dependency limitations and irregular memory access patterns. In this category, the computation is naturally expressible in the form of a static task interaction graph with tasks of known size [55]. Each graph vertex shows a task and each edge represents a data interaction link between two incident tasks.

Scheduling the task interaction graph on a homogeneous or heterogeneous cluster can be done by partitioning the graph. Simply stated, a graph partitioner is an integrated module of the scheduler. First, the graph needs to be divided into a set of computationally balanced partitions in such a way that communication cost among the partitions is minimized. Subsequently, the partitions are scheduled to the target platform. Designing a high performance graph partitioner on the heterogeneous CPU-GPU platform is compelling since the unique features of the heterogeneous CPU-GPU system can speed up the scheduling process indirectly by performing a fast graph partitioning. Furthermore the graph partitioning, which goes beyond the scheduling domain, has extensive applications in various areas of computing such as data mining, geographical information systems, social networks and VLSI design.

Graph partitioning is a subcategory of the clustering problem with two specific objectives. The first is to balance the weights of the partitions, and the second is to minimize the communication cost among the partitions. Multilevel graph partitioning approach [14, 15, 39] is one of the most successful heuristics proposed for efficient graph partitioning. The idea is to first reduce the graph size by matching and collapsing the vertices in multiple coarsening levels until the number of vertices is below a threshold; then, the coarsened graph is partitioned; finally, the partitioning is projected back through the multiple levels onto the original graph. Although many sub-tasks of multilevel graph partitioning are serial in nature, several parallel implementation version of it have been proposed for distributed and multi-core systems. The quality of the partitions produced by parallel algorithms is lower than that produced by serial algorithms. Nonetheless, the parallel schemes deliver significant speedups compared to the serial version.

Designing a graph partitioner on a heterogeneous CPU-GPU systems is a double-edged sword. On one hand, the high processing power of GPU cores in collaboration with a CPU can achieve higher performance compared to distributed and multi-core graph partitioners. On the other hand, unlike data-parallel applications, irregular, non-uniform, and data-dependent graph partitioning sub-tasks pose multiple challenges for efficient GPU utilization. These challenges include thread divergence, load imbalance, non-coalesced memory accesses, warp execution inefficiency, and limited-size GPU memory. Consequently the existing graph partitioning parallel algorithms need to be modified for an efficient implementation.

In this thesis, we address the above challenges in the heterogeneous CPU-GPU systems. First, we develop an adaptive and automated scheduler for embarrassingly parallel applications aimed at minimizing the makespan. Then, we design and implement high performance graph partitioning methods for task dependent applications. Finally, we extend some of our developed techniques for parallelizing the other graph processing algorithms on a CPU-GPU platform.

## 1.2 Problem Statement

Modern heterogeneous systems have evolved from the traditional heterogeneous systems with CPUs of different speeds to the new generation systems equipped with accelerators. The massively data parallel computation and power efficiency of GPUs have led to the collaboration between CPUs and GPUs in achieving the high-performance parallelization of the applications. However, it is important to be mindful of the differences between the architecture and programming models of CPU (MIMD) and GPU (SIMD) in order to fully exploit the processing power of these heterogeneous platforms.Effectively parallelizing application on theses platforms may also require heavy modifications to the existing parallel algorithms.

A wide range of applications fall into embarrassingly parallel category [66] in which tasks are completely independent. In these applications, the problem is decomposed into many identical but independent tasks that can cooperatively produce the desired results in a parallel fashion. Although the high processing power of GPUs makes the CPU-GPU systems excellent candidates for parallelizing the embarrassingly parallel applications, an ideal execution time can only be achieved if the application tasks are optimally distributed over the CPU and GPUs to minimize the underutilization of the processing cores. This is challenging since determining the proper portion of workload for each device is an NP-complete [43] scheduling problem.

Another problem related to scheduling of embarrassingly parallel applications in a CPU-GPU environment is the programming effort required to distribute loads over the CPU and GPU cores. The traditional GPU scheduler cannot automatize the distribution of load over the CPU and GPUs, so the programmer must manually partition the workload. This process is tedious and does not scale well beyond solving small problems. Hence, fully automatic techniques are required to take advantage of processing strength of heterogeneous computing. While uniform programming environments like OpenCL have emerged, these do not give a programmer full control over optimal task scheduling. OpenCL offers transparency to the programmer by

hiding most of the underlying architectural differences, but this advantage comes at the cost of performance since the programmer does not have explicit control over scheduling decisions.

Meanwhile, several static scheduling algorithms have been proposed to execute embarrassingly parallel applications on CPU-GPU systems [33, 59, 94]. However, they have significant overheads due to their extensive profiling phases and are not accurate due to non-linear and black-box nature of GPU performance characteristics. Some researchers have proposed dynamic scheduling heuristics [44, 82, 84, 89] for CPU-GPU environments. A few of them have considered executing multiple applications on a CPU-GPU system and dynamically assigning each application to one of the processors without any load partitioning between the CPU and GPU [44, 89]. Some other works present solutions that are not scalable to more than one integrated GPU [84]. In [24], the authors proposed a general framework, which does not consider the distribution of tasks and schedules the entire kernel, for scheduling applications on a heterogeneous CPU-GPU system. StarPU [4] is a framework that requires some hints from the user for scheduling purposes.

In summary, majority of proposed scheduling heuristics cannot take full advantage of the computational powers of the devices in heterogeneous CPU-GPU architectures. Most of the contemporary schedulers, in fact, are application-dependent and require the user's interference for scheduling decisions and low-level programming skills for scheduling implementation. As a result, designing a high performance adaptive and scalable heuristic for embarrassingly parallel applications while hiding the scheduling complexities is essential.

A large domain of applications are non-embarrassingly parallel which includes task dependent problems with data dependency. For these applications, the computational tasks and their communication patterns can be represented by a weighted undirected graph in which the vertices represent the tasks and the edges represent the communication costs of the tasks. Many large-scale and complex real-world problems, such as social network interactions, can be expressed as task interaction graphs.

Scheduling of a task interaction graph on a heterogeneous or homogeneous cluster of processors can be done based on a primary partitioning of the graph. In other words, the graph partitioner is an integral part of the scheduler; it divides the graph into a set of equal weigh partitions in such a way that the communication cost (known as edge cut) among the partitions is minimized. Subsequently, the partitions are scheduled over the available processors. Using an efficient graph partitioner for the parallel implementation of these applications is non-trivial

Besides scheduling, graph partitioning has extensive application in many computing areas, including geographical information systems, VLSI design and data mining. Graph partitioning is also a key preprocessing step in many high performance parallel graph algorithms like Page-Rank and Breadth-First Search.

The graph partitioning problem is NP-complete. Consequently, many heuristic algorithms have been proposed [47, 70, 80]. Multilevel graph partitioning techniques [15, 39, 47, 77, 91] are generally preferred over other techniques such as spectral partitioning [80] due to higher quality of partitions at a faster computation time. Handrickson and Leland [39] validated this claim using extensive experiments. In the multilevel graph partitioning, first the graph size is reduced by matching and collapsing the vertices in multiple coarsening levels until the number of vertices is less than a certain threshold; then the coarsened graph is partitioned, and finally the partitioning is projected back iteratively onto the original graph during the un-coarsening phase [3].

The widespread applications of graph partitioning in different areas of computing have encouraged its parallel implementation on multi-core architectures [57, 88] as well on distributed systems [19, 40, 48, 49, 92]. Although serial graph partitioning and its parallel implementations on distributed and multi-core systems have been well studied, designing a graph partitioner on heterogeneous CPU-GPU systems has yet to be investigated. As a throughput-oriented device, GPU hides the memory access latency through high degrees of multi-threading. This indicates an excellent opportunity to accelerate the graph partitioning task on a heterogeneous CPU-GPU

[3] A detailed diagram of multilevel graph partitioning is shown in Chapter 2

system.

In addition, some GPU applications require graph partitioning to balance the workload among the threads and to increase the parallelism. A high performance GPU-based graph partitioner reduces host-device high data transfer costs. For example, Delaunay mesh refinement (DMR) [74] application requires graph partitioning to minimize the conflicts among the cavities processed by the GPU threads and to increase parallelism. Using a contemporary partitioning algorithm would oblige the entire graph to be transferred to the CPU, partitioned there, and moved back to the GPU. Designing a high-performance GPU graph partitioner can resolve this problem while maintaining good performance in comparison with CPU-based partitioners.

GPU allows thousands of threads to be resident on its Streaming Multiprocessors. The SIMD GPU programming paradigm demands repetitive processing patterns on regular data which is contrary to the irregular nature of real-world graphs. Therefore, an acceptable implementation of CPU-GPU graph partitioning must utilize the collective computation force of threads. This prominent difference makes proposed approaches on distributed and multi-core systems not applicable on CPU-GPU partitioner in a straightforward way. Furthermore, when processing an irregular application like graph partitioning, designing an efficient parallelization strategy becomes challenging. Particularly when dealing with large and irregular real-world graphs, non-uniform and data-dependent graph partitioning sub-tasks result in imbalanced load distribution among threads, consequently deteriorate the performance of the graph-partitioning kernels executed on the GPU.

Some of the challenges we have to overcome in our design include the following: (1) proper redesigning of the existing parallel algorithms to maximize the graph partitioner efficiency on a CPU-GPU architecture; (2) GPU memory constraints to hold large graphs; (3) the irregular nature of the graph data structure, which can result in thread divergence and poor locality in memory accesses, deteriorating the performance of the graph-partitioning code running on the GPU; (4) synchronization costs, which are much more pronounced on GPUs running tens of thousands of threads as

compared to multi-core CPUs that only run tens of threads; (5) a suitable workload distribution strategy between the CPU and the GPU; and (6) data transfer latency between the CPU and the GPU.

In summary, efficient parallelization of the multilevel graph partitioning in a heterogeneous CPU-GPU system is a challenging task. On the one hand, the highly serial and data-dependent nature of coarsening and un-coarsening phases makes it difficult to exploit the data parallelism within each phase. On the other hand, straightforward porting of existing parallelization heuristics results in inefficient GPU programs, or if the heuristic sacrifices accuracy at the expense of parallelism, it can result in poor partition qualities. These problems determine the need for a multilevel graph partitioner in a CPU-GPU platform that accelerates this task by being tailored specifically for the SIMD architecture, and at the same time, prov

iding reasonable partition qualities compared to serial and multi-core solutions.

## 1.3 Contribution

In this thesis we explore the followings: 1- design and implementation of an adaptive and automated scheduling technique for embarrassingly parallel applications on a heterogeneous CPU-GPU system; (2) design and implementation of efficient parallel multilevel graph partitioning methods for task dependent applications on heterogeneous systems; and (3) investigate the application of some of our partitioning techniques to other graph processing algorithms.

The main contribution of the thesis are as follows:

- We explore the scheduling problem for embarrassingly parallel applications on a heterogeneous environment by designing and implementing a new dynamic scheduling heuristic for embarrassingly parallel applications, where tasks have no inter-dependencies. The goal is to distribute the load among the processors adaptively so that the application makespan is minimized. Meanwhile, the

scheduler aims to take advantage of full processing powers of the GPUs' processing cores and to minimize user interference into the scheduling criteria. Our proposed dynamic scheduler is scalable to any number of GPUs integrated in the heterogeneous CPU-GPU system. We employ runtime techniques like profiling and work stealing to address the efficient load distribution between the CPU and GPUs. The scheduler is integrated into a scheduling framework that provides pre-implemented automated scheduling modules. The user is liberated from the complexities of scheduling details and from manually distributing the workload over the CPU and GPU cores. The experimental results show that our scheduling approach achieves better to similar performances compared to some of the well-known scheduling algorithms for the CPU-GPU systems.

- We design and implement multilevel graph partitioner on a heterogeneous CPU-GPU system that takes advantage of the high parallel processing power of GPUs by executing the computation-intensive parts of the partitioning sub-tasks on the GPU and assigning the parts with less parallelism to the CPU. The partitioner aims to overcome some of the challenges arising due to the irregular nature of the partitioning algorithm, load imbalance, and memory constraints on GPUs. Our design also minimizes the lock usage and does not degrade performance through fine-grained synchronization among the threads. To mitigate the global memory size limitation, we use Compressed Sparse Row (CSR) representation, which is an efficient compact format for representing large and sparse graphs inside the limited GPU memory. Our partitioner handles the aforementioned challenges through redesigning of the existing parallel multilevl partitioning algorithms, considering the heterogeneity of the architecture, and exploits special characteristics of GPUs. To the best of our knowledge, this is the first proposed multilevel graph partitioner designed for and implemented on a heterogeneous CPU-GPU system. Our CPU-GPU graph partitioner outperforms the serial and distributed multilevel graph partitioners and performs similar to mt-metis, the state-of-the-art CPU-based parallel graph partitioner .

- Next we identify the performance bottlenecks of our developed CPU-GPU graph partitioner to optimize our design accordingly so that it outperforms the CPU-based partitioner as well. We discover that our first solution is prone to load imbalance in some of the partitioning phases. The reason for this problem is that in the coarsening and un-coarsening phases of partitioning, the graph vertices are distributed among the GPU threads, and each thread serially processes the neighbor lists of its assigned vertices. Consequently, the irregular sized neighbor lists of the graph vertices results in thread divergence and in non-coalesced accesses to edge and vertex indices. To resolve this problem, instead of assigning a GPU thread to process the neighbor list of a vertex, we exploit the lock-step processing power of warps, and the warp threads process the neighbor list of a vertex in parallel. This prevents the thread divergence and underutilization of SIMD resources while balancing the load over the GPU threads.

  Furthermore, during transferring the less computational sections of the partitioning sub-tasks to the CPU, comparatively low communication bandwidth between the CPU and GPU creates performance overhead. Therefore, we design and implement a high performance multilevel partitioner that performs all the phases of partitioning on the GPU with minimal CPU interventions. We develop new coarsening and un-coarsening parallel algorithms to speedup our partitioner. Despite the irregular inter-dependency of graph partitioning sub-tasks, our approach balances load over the SIMD threads and prevents thread divergence.

  We also mitigate recurrent GPU memory allocation overhead and optimize our design by deploying a custom regional memory allocation technique, which reduces the cost for allocating data on GPU and increases the partitioning efficiency. This partitioner is autonomous as all the steps are carried out by the GPU with minimal CPU interference. Extensive experiments on our newly designed GPU-based partitioner over a set of graphs from various computing areas demonstrate better performance in terms of partitioning speed while delivering

a reasonable partition quality in comparison to multi-core graph partitioners.

- Finally we apply some of the techniques we developed specifically in the coarsening phase of our graph partitioner to another graph processing application that exhibits such characteristics as thread divergence and imbalance load distribution. Minimum Spanning Tree (MST) is a well-known graph processing algorithm that creates a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices without any cycle. One of the well-known proposed MST algorithms is that of Borůvka [12], which is known to be suitable for parallelization. This algorithm finds the minimum weighted outgoing edge at each vertex and merges the connected vertices into supervertices. Since Boroůvka's algorithm provides natural parallelism, many parallel MST algorithms are based on this approach. In the first phase of parallel Borůvka's algorithm, all the vertices find the minimum-weight crossing edge among their neighbors on the other components. This phase has a similar function to the matching process in the coarsening phase of our designed graph partitioner. Extending our developed techniques to the first phase of Boruvka's algorithm helps in achieving a high-performance implementation of MST algorithm on the GPU.

## 1.4 Thesis Outline

The rest of this thesis is organized as follows. Chapter 2 discusses the background and related work. Chapter 3 presents the design and implementation of a dynamic and automated scheduler for embarrassingly parallel applications on CPU-GPUs systems. Chapter 4 describes our solutions for enabling multilevel graph partitioning on a heterogeneous CPU-GPU system. Chapter 5 proposes our high performance multilevel GPU-based graph partitioner and discusses the extension of our techniques to parallel MST implementation on a GPU. Chapter 6 concludes the thesis by summarizing our work and provides a discussion on potential future research.

# Chapter 2

# Literature Review

Heterogeneous CPU-GPU platforms benefit from using the combined potential of both CPU and GPU computing power and features. GPUs have evolved significantly in the past decade. The new generation of GPUs has thousands of cores and multiple gigabytes of global memory. Modern GPU processors are massively parallel and are fully programmable [76]. The emergence of these heterogeneous CPU-GPU systems and the rapid programmability and capability of GPUs present a unique opportunity for speeding up parallel computations. However, deep understanding of the underlying architecture restrictions and performance challenges of the GPU is crucial to adapting the parallelization algorithms accordingly.

In this chapter, we first give a brief background of the heterogeneous CPU-GPU system architecture and CUDA programming model. Then, we provide an overview of GPU performance challenges and constraints on high-performance parallel execution of applications. Subsequently, we review some of the proposed static and dynamic scheduling heuristics for heterogeneous CPU-GPU environments. Afterward, we discuss the multilevel graph partitioning problem and cite contemporary and relevant research on serial and parallel graph-partitioning algorithms. Finally we elaborate on several parallel optimization techniques proposed for the GPUs.

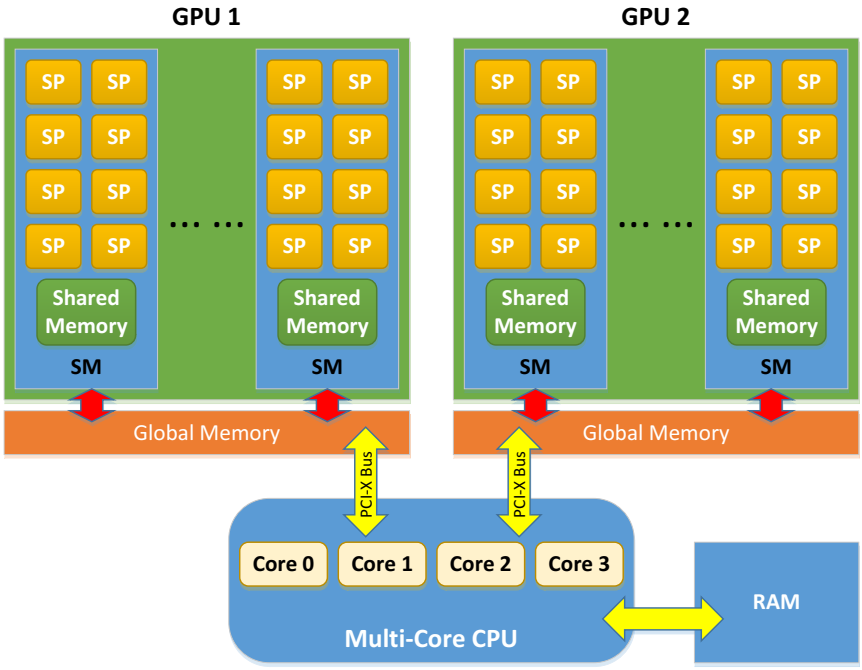



Figure 2.1: Heterogeneous CPU-GPU architecture.

## 2.1 The CPU-GPU System Architecture

Figure 2.1 illustrates the general configuration of a single CPU-multiple GPU architecture consisting of two GPUs. CPU is the dominant processor (called the host) and the GPUs are the subordinate processors (called the devices) under the control of the CPU. CPU communicates with the GPUs through the PCI-X buses. A GPU consists of a scalable number of Streaming Multiprocessors (SMs), each of which comprises a number of Streaming Processors (SPs), which are also called GPU cores.

The GPU memory system provides both on-chip and off-chip memories. The off-chip memory is generally referred as the Global Memory of the GPU, and all the threads running across the SMs have read/write access to this memory. However, in terms of access latency, this memory is slow [17]. The on-chip memory includes the shared memory and thread registers. Shared memory is located on each of the SMs and consequently is almost as fast as accessing a register. However, shared memory size is limited to less than 100 KB per SM.

## 2.2 CUDA Programming Model

The development of GPU programming models and tools has been as important as the advancement of the GPU as a general purpose processing unit. CUDA (Compute Unified Device Architecture) [1] is a common GPU programming model developed by Nvidia that provides a framework for developing parallel applications on a GPU and enables the parallel execution of thousands of threads on the GPU. Simply stated, CUDA is a software layer that gives direct access to the GPU parallel computation units. CUDA provides a set of extensions to existing languages such as C and C++ that supports useful primitives and functions to interface with the GPU. A CUDA program consists of two parts: one part is made up of the portions to be executed on the GPU, also known as the kernel; the other part is the program that is to be executed on the host.

CUDA supports a hierarchy of thread grouping for the execution of a program on the GPU. The highest level is called a grid, which encapsulates all of the threads executing an application. A grid consists of a set of thread blocks that execute a kernel function. The blocks are organized into a one-dimensional, two-dimensional, or three-dimensional grid of thread blocks. All threads within a block are executed concurrently on the GPU. Each thread within a block has a thread ID that is unique only among threads within the same block. Furthermore, each block has a block ID that identifies its position in the thread grid. These two IDs generate a unique thread ID for each thread at the grid level.

The threads within one block are grouped into a series of 32 threads that construct what is called a warp. The threads within a warp are executed in locksteps, i.e., all threads within a warp execute one common instruction at the same time. Threads within one block can share data using shared memory and can be synchronized at a barrier with very low latency. However, different blocks can communicate just through the global memory with much higher access latency. Figure 2.2 shows the CUDA thread hierarchy paradigm.

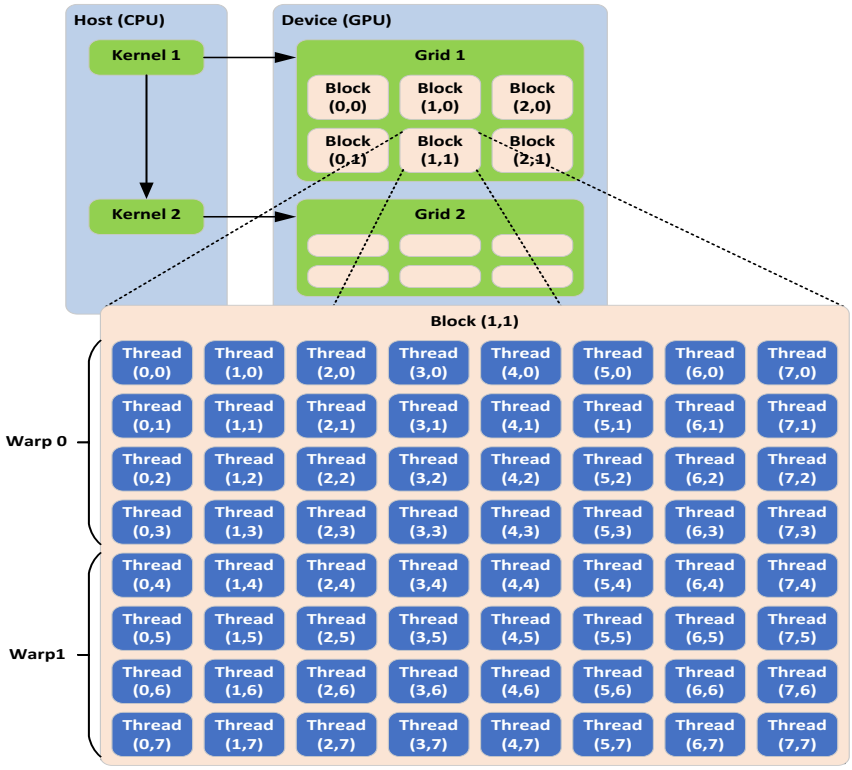



Figure 2.2: CUDA thread hierarchy.

To execute a CUDA program on a GPU the host process launches a set of kernels on GPU the selection of which depends on the application. For each kernel, the host process determines how many threads are required to execute the kernel and how many thread blocks (TB) these threads should be equally divided into. The required data for the execution should be transferred from the CPU to the GPU a priori to be able to run a kernel. This data is copied to the global memory of the GPU. The hardware schedules and distributes TBs to SMs with available execution capacity. One or multiple TBs can reside concurrently on one SM, given sufficient hardware resources such as register files and shared memory. Each thread is mapped to one SP core.

## 2.3 GPU Performance Constraints

Understanding the GPU architecture constraints and performance bottlenecks helps programmers exploit the full processing power of GPUs and enables the redesign of parallel algorithms accordingly to increase productivity. Below, we review some of the key performance constraints of the GPU architecture.

### 2.3.1 Thread Divergence

Generally, a GPU performs a SIMD (single instruction-multiple data) execution at the warp level and all the threads run in lockstep. If threads within a warp diverge, the entire warp will execute both code paths until all the threads re-converge. Simply stated, thread divergence can happen if threads of the warp take different execution paths, which results in the execution being serialized. For instance the presence of conditional statements such as if-else blocks causes thread divergence because a conditional statement may evaluate to true for some warp lanes and false for the other lanes.

Thread divergence can degrade the performance significantly, and this is a critical issue in exploiting the processing power of a GPU, especially in applications with irregular data accesses. Structuring the code to minimize divergence within warps for high-performance GPU computing is non-trivial.

### 2.3.2 Un-coalesced Memory Access

Leveraging the performance of GPU applications requires streaming memory access patterns in which threads read from and write to large consecutive blocks located in separate regions of memory. A coalesced memory access is the combination of multiple memory accesses into a single transaction. In modern CUDA-capable GPUs, sets of 32 contiguous threads constitute a warp. When all threads in a warp execute a memory instruction, the hardware checks which memory locations the threads access.

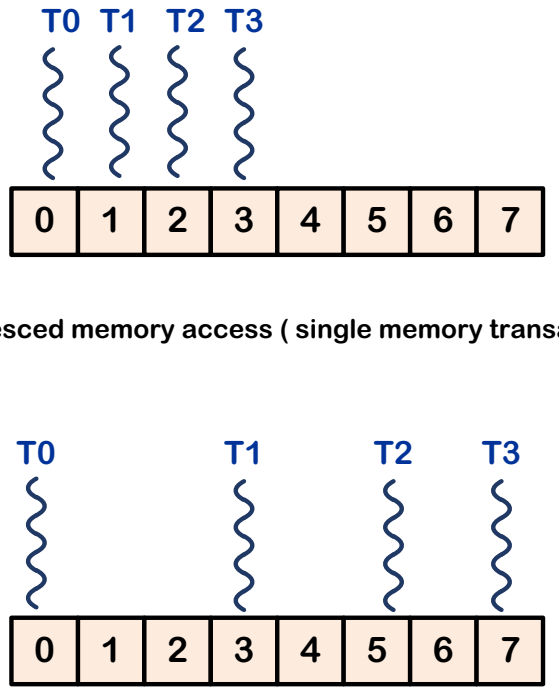



Figure 2.3: Two memory access patterns for a set of GPU threads.

Ideally, if all the memory accesses within a warp can be combined into a naturally-aligned 128-byte block in the global memory, the hardware coalesces the accesses into one transaction. Otherwise, the irregular access patterns penalize the memory performance with crossed relations between data and threads. In such a case, multiple memory transactions have to be issued, which reduces the efficiency and amortizes the throughput. Figure 2.3 shows two cases for the access pattern of a set of GPU threads.

Although regular memory access facilitates the coalescence of memory access on the GPU, coalescing will be challenging for irregular programs with the complex data structures such as graph and tree. Consequently, efficient parallelization of these programs on the GPU is more challenging and requires a mechanism to hide the latency associated with the non-coalesced memory accesses.

### 2.3.3 Limited Memory Size

Present-day high-end GPUs offer up to 16 gigabytes of global memory. This relatively limited size of the GPU memory prevents application of the GPU to process big data problems. This obliges the programmer to optimize memory allocations on the GPU to be able to handle the large-sized real-world computations on the GPU and mitigate the GPU memory bottleneck. Heterogeneous systems with multiple-GPUs are also proposed as a solution to scale to even bigger datasets. Nonetheless, they introduce additional latency and synchronization challenges.

### 2.3.4 Low-Level Programming Models

There is a trade-off between low-level access to the GPU, which accelerates the execution time, and high-level GPU programming languages, which enable productivity and flexibility for the users. Consequently, obtaining high performance on a GPU requires high programming effort to understand the low-level architecture of the GPU including the functionality of the processing cores, memory access patterns on shared and global memory, and the thread scheduling schemes.

Although researchers proposed several libraries for high-level programming approaches that make the data transfer between the CPU and the GPU more implicit [59, 76], most of the complex parallel applications achieve better performance through using lower-level programming models like CUDA, which are closer to hardware languages.

### 2.3.5 Synchronization Latency

A GPU has a relatively limited global memory size of up to 16 gigabytes in the latest generation of GPUs. While CUDA provides a barrier function to synchronize threads within a thread block, it does not support any mechanism for communications across thread blocks. Consequently, device level synchronization is possible through the global memory with large access latency.

In order to avoid data races in GPU applications that share data on the global memory, accesses from different thread blocks must be protected by locks. The huge number of GPU threads exacerbates the lock-based programming challenges.

Although locking is a contemporary practical mechanism for ensuring atomic accesses to shared data on the GPU, the improper usage of locks can degrade the performance. With the advent of atomic functions on GPUs, such as compare-and-swap (`atomicCAS()`), it is possible to perform non-blocking synchronization techniques among the threads. Recently, GPUs can resolve atomics within internal caches and spin-locks are now relatively fast. Nevertheless, due to the high large access latency of global memory, programmers consider avoiding synchronized access over the global memory or using the fast on-chip shared memory, which enables fast synchronization within the thread block.

## 2.4 Scheduling Heuristics on Heterogeneous CPU-GPU Systems

A large number of problems can be cast to the embarrassingly parallel applications, where the computation can be divided into a number of tasks with no inter-dependencies. Proper scheduling of these independent computational tasks on a given set of processors is a key factor for high performance computing. The scheduling methods proposed for heterogeneous clusters [8, 13, 63, 87, 98] or multi-core environments [18, 97] are not applicable to heterogeneous CPU-GPU systems in a straightforward way. The differences in the programming paradigm of the CPU and the GPU, requires designing efficient scheduling heuristics, to maximize the performance by optimally distributing the workload over the CPU and the GPU cores.

Although the SIMD programming paradigm of GPUs makes them excellent candidates for accelerating the embarrassingly parallel application, proper distribution of parallel tasks over both the CPU and the GPU, results in better performance in comparison to CPU-only or GPU-only execution. Figure 2.4 validate this claim by

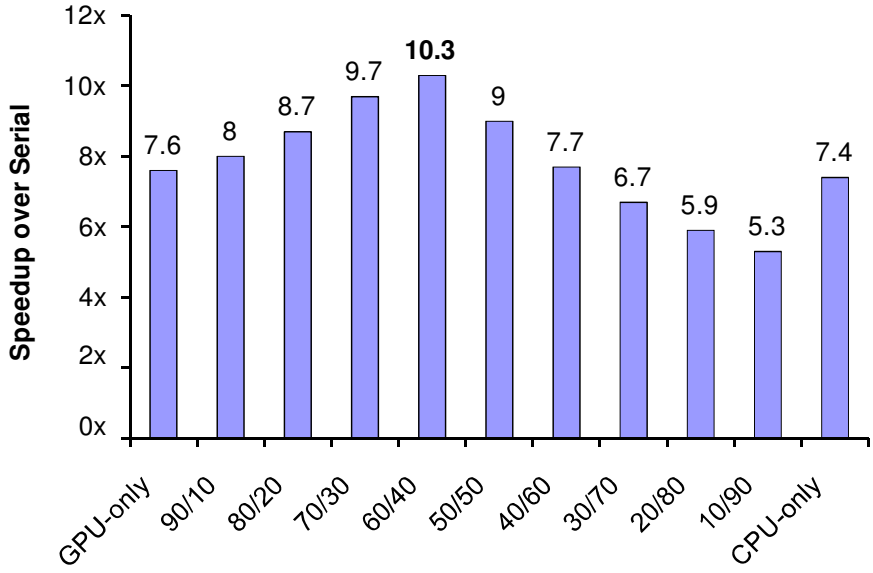



Figure 2.4: Matrix multiplication experiment in a CPU-GPU system. Matrix size is 6000. The notation "X/Y" on the x-axis means X% of work mapped to the GPU and Y% of work mapped to the CPU [62].

measuring the parallelization speedups of the matrix multiplication application in a heterogeneous system consisted of an Intel multicore CPU and an Nvidia 8800 GTX GPU [62].

Since the scheduling problem for heterogeneous CPU-GPU system is an NP-complete problem, several static and dynamic scheduling heuristics and frameworks are proposed in the literature [71]. We review some of these heuristics in the following.

### 2.4.1 Static Heuristics

Min-min is a well-known static heuristic proposed for task scheduling on a general heterogeneous environment [13]. In the min-min heuristic, the next minimum sized task is always removed from the list of tasks waiting for the execution, and it will be executed on a device, which provides the earliest expected completion time. This process repeats until all the task are mapped to the processors. The parallel GPU-based implementation of min-min heuristic has been studied as well [78].

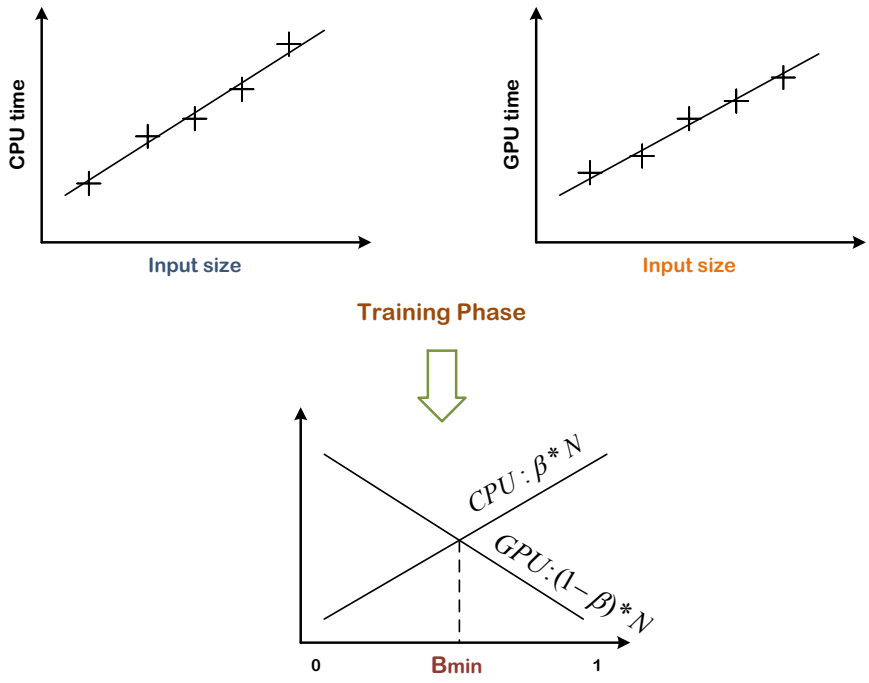



Figure 2.5: Training and partitioning phases of Qilin method [62].

There are several studies on how to statically distribute the massively parallel computations over the CPU and GPU cores on a heterogeneous environment. Qilin [62] focuses on a on a single-CPU single-GPU system. It employs an adaptive mapping scheme that involves a training phase. The training phase interpolates a system of linear equations based on empirical results from adaptive mappings. Adaptive mapping starts with a training phase in which two linear equations are built based on calculating the execution time of different sized sub-tasks of an arbiter data parallel problem size on CPU and GPU separately. Solution of these equations provides the best partitioning between the CPU and the GPU.

Figure 2.5 shows the training and partitioning phases of Quilin method. If $\beta$ shows the fraction of input load assigned to CPU, $(1 - \beta)$ shows the portion of the load assigned to GPU. The intersection of the two diagrams in Figure 2.5, determines the value of $\beta$ that minimizes the makespan. If $\beta < 0$, all the input workload is mapped to GPU. If $\beta >= 1$, all the workload is mapped to CPU and, If $0 < \beta < 1$, the distribution rate of actual problem size over CPU and GPU is determined according

to the value of $\beta$.

The most serious problem in the Qilin approach is the overhead of the training phase. Furthermore according to the empirical results, the method is efficient when the problem size in the training phase is at least 30% of the actual problem size. Consequently, when executing a very large-size problem, implementation of the training phase can impose a prohibitive overhead. Finally, this strategy is applicable in a single-CPU single-GPU system and is not scalable.

Another static task partitioning scheme for heterogeneous CPU-GPU systems [33] extracts the code features of a program at compile time by utilizing machine-learning techniques [73] and, based on some training data, the best partitioning among the processors is predicted using a two level predictor. In the first level, it filters the programs which are mapped to either CPU or GPU. The remaining programs are passed to the second level and based on their futures are classified in to 11 groups. These prediction steps are done during the run-time when the problem input size is known and it has a negligible overhead. However, the complexity related to designing the predictor and lack of suitable training data could degrade the accuracy of this approach.

Wang et al. [94] proposed a static scheduling, where task are modeled into two different computing and communicating categories using a hierarchical control data flow graph. The computing-intensive subtasks are executed by GPU, and the communication-intensive subtasks are assigned to CPU. Although the authors compared the proposed algorithm with the traditional scheduling algorithms, the details of implementation have not been well documented. Furthermore it is only applicable in a heterogeneous system equipped with a single GPU.

Boratto et al. [11] applied a static scheduling technique, to partition the workload of the matrix computation constructed for solving the landform attributes representation, over a heterogeneous CPU-GPU systems. However, the portion of workload delivered to each device is an input to the scheduler provided by the user manually.

### 2.4.2 Dynamic Heuristics

The greedy heuristic is a common dynamic heuristics for the heterogeneous system in which, once a processor becomes idle it greedily picks up a task from the task pool. Yuan et al. [95] implemented the dynamic greedy heuristic on a heterogeneous CPU-GPU system, and showed that it achieves better speedup in comparison to GPU-only and CPU-only policies. Choi et al. [20] discussed a similar scheduling heuristic which maps an incoming task on the first available device.

Ravi et al. [82] developed a compiler and a runtime scheduler for heterogeneous distributed systems, dedicated for map-reduce applications [23], which involves a generalized reduction. Each node has a multicore CPU and a GPU; it receives a number of chunks and partitions the chunks dynamically to the processors based on a master-worker paradigm. However, the optimal split size has not been determined in this work. The approach is restricted to specific application types that fit into the map-reduce model.

V. Jiménez et al. [44] explored the scheduling in multi-programmed heterogeneous systems based on a performance history aimed to fully utilize all the available processors in CPU-GPU devices. In the initiation phase, each application runs on different devices and a performance history will be created which is used in the next phase to assign the programs to the processors. Nevertheless there is not any load-partitioning algorithm in this method and the main goal is to improve the performance where several applications are concurrently scheduled in the system.

Scogland et al. [84] proposed several compiler and runtime strategies to schedule the iteration-based OpenMP [16] loads across a single-CPU single-GPU architecture. Initially a static scheduler calculates the distribution ratio of load over the devices based on the number of cores. Then a dynamic scheduler attempts to predict the portion of loads on each device in the upcoming rounds based on the execution time of the load portion in the current round. Nonetheless, this approach is not scalable to more number of GPUs and it is not accurate due to non-linear properties of the GPU for different load sizes.

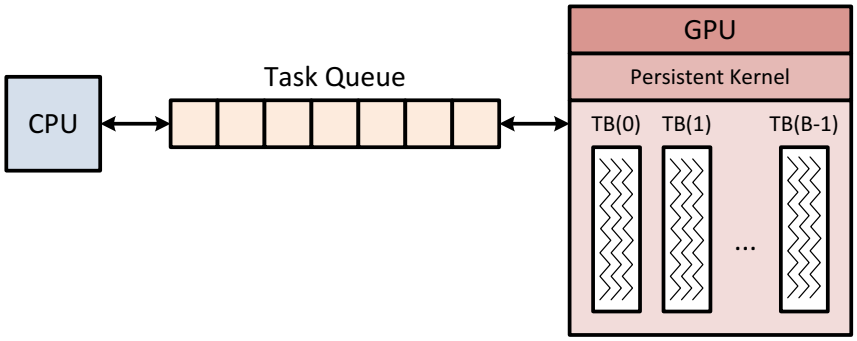



Figure 2.6: Task queue scheme for scheduling on GPU [56].

Hamano et al. [89] considered the estimated energy consumption of the CPU and GPU in the idle and busy states, as the criteria of the scheduling decision. According to their energy consumption model, each task is mapped to the device that leads to minimum energy consumption.

Some researchers have addressed the scheduling and load balancing techniques on the GPU devices. In [56], the authors proposed a dynamic task-based load balancing technique for single and multi-GPU systems. In conventional CUDA paradigm, multiple kernels are launched sequentially to execute several tasks. However, in this work, they use a task-based queue scheme and instead of launching several kernels, a persistent kernel [34] with $B$ thread blocks ($B$ is the maximum number of concurrent active blocks in a GPU Device) is launched. These thread blocks dequeue the tasks in a wait-free [41] approach and execute them concurrently according to the pre-defined tasks information. The details of this method can be seen in Figure 2.6.

Tzeng et al. [90] employed a similar technique to schedule the irregular parallel workloads dynamically on a single GPU. They implement a distributed work-queue based system, but work units are inserted to the queue in the size of warp. Also a persistent kernel model is utilized in which sufficient number of warps is launched a priori to keep all the cores busy. Each warp worker reads as many work units as possible from its dedicated queue and enqueues the possibly dynamic created work units back to the queue. To balance the load among the cores, some of the workers can steal some works from other queues or donate their work in case of queue overflow.

One shortcoming of this work is that all the applications are not fit in the warp-size work units. This degrades the generality of this scheduling method. Furthermore this technique just supports the independent work units.

In [60], the authors developed a task-based method for scheduling the loads on GPU by grouping small tasks together and executing them on a multi-kernel supporting GPU. They also proposed a methodology for executing a set of the tasks in the most efficient sequence.

### 2.4.3 Scheduling Frameworks

Harmony [24] is a general runtime model for heterogeneous multi-core systems in which each application is composed of a set of kernels with different types of dependencies. Whenever the dependencies of one kernel have been resolved it will be scheduled dynamically over the CPU and different accelerators, based on a greedy scheduling heuristic.

StarPU [4] is a run-time framework for plugging in and executing scheduling algorithms on a heterogeneous CPU-GPU system. The programming environment accesses the low-level libraries indirectly by building over the framework interfaces. Different scheduling algorithms have been proposed including greedy scheduling and performance-based scheduling. However, the programmers have to use a new API proposed by the system, because the tasks are demonstrated with codelet [25] abstraction, which consists of tasks augmented with their input and output specifications.

In [17], the authors designed a work stealing run-time on the GPU to execute irregular applications with dynamic task parallelism across the SMs on GPU and to balance the workload among them. They employ a work queue in which, the tasks are copied from the host and are executed over the SMs with the block granularity. Since some tasks may be created dynamically the work-stealing method among the SMs will balance the loads among the SMs on GPU.

## 2.5 Graph Partitioning

Graph partitioning and graph clustering have extensive applications in various areas of scientific computing. While graph clustering [3, 6] identifies the groups of the vertices in a graph that show the same behavior or similar characteristics, graph partitioning is a sub-category of the clustering problem with two specific objectives; The first is to decompose a graph into $k$ sets of partitions such that communication cost between the partitions are minimized and the second is to balance the weights of partitions.

Formally, given an undirected graph represented by a tuple $(V, E, W_V, W_E)$, where $V$ is the set of vertices, $E$ is the set of edges, $W_{v_i}$ is the weight of each vertex $v_i$, and $W_{e_i}$ is weight of each edge $e_i$, graph partitioning is to divide the graph $G$ into k partitions $\{p_1, p_2, \cdots, p_k\}$ such that:

$$p_i \cap p_j = \emptyset \text{ if } i \neq j \; and \bigcup_{i=1}^{k} p_i = V \tag{2.1}$$

A quality approach keeps the partitions as balanced as possible with respect to their accumulated vertex weights, i.e., if we show the total weight of all vertices in partition $p_i$ using $W_{p_i}$, then we expect:

$$W_{p_i(i=1,2,\cdots,k)} \simeq \frac{\sum_{i=1}^{k} W_{p_i}}{k} \tag{2.2}$$

Another common expectation is to minimize the accumulated edge cut weights (total communication cost) where an edge cut is defined as:

$$edge\ cut = \sum_{\substack{e=(a,b)\in E \\ p(a)\neq p(b)}} W_e \tag{2.3}$$

The graph partitioning problem is NP-complete. Consequently, many heuristics

have been proposed to quickly find a near-optimal solution [47, 70, 80]. Spectral partitioning methods [80] calculate the Laplacian matrix associated with the graph and, divide the vertices of the graph into the two subgraphs by using one of the eigenvectors of the Laplacian matrix. Although the spectral methods produce high quality partitioning, they are slow, sine they need expensive computations for calculating the eigenvector. Geometric graph partitioning methods [70] are applicable only when the graph vertices coordinates are available and they produce the partitions with lower quality in comparison to the spectral methods.

The most successful heuristic for partitioning large graphs used in scientific computations is the multilevel graph partitioning approach [39, 47, 77, 91]. The idea is to first reduce the graph size by matching and collapsing the vertices in multiple coarsening levels until the number of vertices is below a threshold; then the coarsened graph is partitioned, and finally the partitioning is projected back through the multiple levels onto the original graph. Multilevel graph partitioning has become the standard approach for developing high quality and computationally efficient solutions for graph partitioning.

### 2.5.1 Serial Multilevel Graph partitioning

Multilevel graph partitioning techniques [15, 39, 47, 77, 91] are generally preferred over other techniques such as spectral partitioning [80] due to higher quality of partitions at a faster computation time. Handrickson and Leland [39] validated this claim using extensive experiments. Metis [47], Scotch [77], and Jostle [91] are the well-known multilevel graph partitioning solutions.

Figure 2.7 gives an overview of the multilevel graph partitioning algorithm. The algorithm consists of three distinct phases that we describe below.

***Coarsening.*** In this phase, the graph is iteratively shrunk by matching and collapsing vertices in order to construct a compact version of the graph. Every iteration includes two steps commonly known as *matching* and *contraction*. The matching step finds a set $M$ of edges such that no pair within the set is incident on the same vertex,

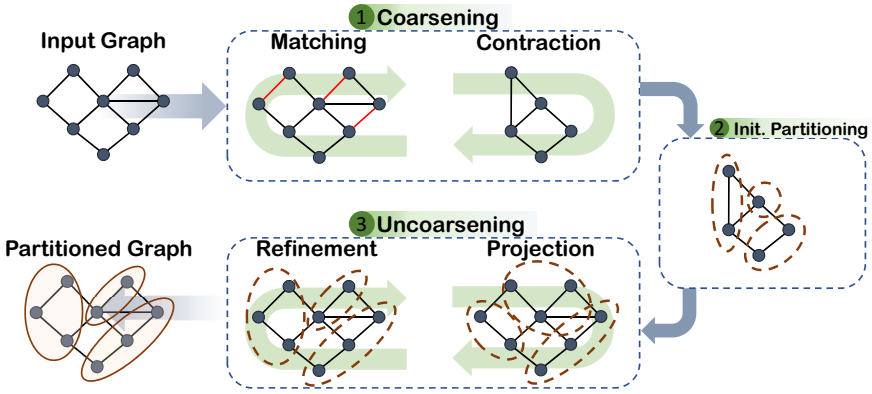



Figure 2.7: An overview of multilevel graph partitioning.

while the contraction step collapses all the matched vertex pairs together.

Serial algorithms find the maximal matching where it is not possible to add another independent edge to the set whereas parallel algorithms usually relax this assumption to avoid its overhead. Although several polynomial time algorithms have been proposed for graph matching [27, 68], they are very slow and difficult to be parallelized for large real-world graphs. Approximation algorithms such as Random Matching [39], Heavy Edge Matching [50, 81], and Light Edge Matching [47] are typically favored over polynomial time algorithms due to enabling a better trade-off between the computation time and the quality of partitions. Heavy Edge Matching (HEM) exhibits the best results where each vertex is searched for the neighbor connected with the edge having the maximum weight since iterative application of this procedure minimizes the edge weights in the coarser graph. Metis, Scotch, and Jostle all employ HEM for the matching graph vertices in the coarsening phase.

For two collapsed vertices $u$ and $v$, the weight of the newly created vertex $c$ ($W_c$) in the coarser graph is equal to $W_u + W_v$. Also, if there is one vertex $z$ that is connected to both $u$ and $v$ in the finer graph, then there will be one edge in the coarser graph from $z$ to $c$ with the weight $W_{u,z} + W_{v,z}$. The coarsening step hierarchically creates the successive coarser graphs until the number of vertices in the resulted graph is less than a threshold value or equal to the number of required partitions.

The matching and contraction steps terminate based on a specific criterion. In

Metis and Scotch, the matching ends when the number of vertices of the coarse graph is $O(p)$, where $p$ is the number of partitions, or if the difference in the number of vertices in the coarser graph $G_{i+1}$ compared to the number of vertices in the next finer graph $G_i$ is less than a threshold value. Jostle terminates the matching when the number of vertices in the coarse graph is equal to the number of required partitions.

***Initial Partitioning.*** This phase creates a preliminary partition from the coarsest graph. With vertices grouped together in larger entities, it is easier to reason about the approximate partition weights and initial edge cuts. The initial partition created in this phase drives the partitioning of the finer graphs in the next phase.

While Jostle skips initial partitioning phase by reducing the number of vertices in the coarsening phase to the exact number of required partitions $p$, Metis and Scotch apply a Greedy Graph Growing Partitioning (GGGP) algorithm to partition the coarsest graph into $p$ parts. In more details, GGGP starts from a random vertex and gradually grows a region around that vertex in a breadth-first fashion. Among the possible candidates in every step, the vertex with the largest decrease in the edge cut is chosen first for inclusion in the region. The region continues to grow until it includes approximately half of accumulated vertex weights. By repeating this recursive bisection method, the required number of partitions is obtained.

***Un-coarsening.*** Finally in the un-coarsening phase the graph is iteratively projected back and refined onto the original graph, therefore, this phase can be viewed as the reverse counterpart of coarsening phase. Every iteration of the un-coarsening phase has two steps named *projection* and *refinement*. In the projection step, partition information of the vertices in the coarser graph is projected to the vertices in the finer graph. Then in the refinement step, boundary vertices are moved among the partitions so as to reduce the edge cut. A vertex $u$ in partition $i$, that has a neighbor $v$ in partition $j$, $i! = j$, is a boundary vertex. The un-coarsening process is carried out until the original graph is formed, which includes partition information for every vertex.

Metis and Scotch utilize a modified version [28] of the Kernighan-Lin heuristic [51]

for refinement, in which the boundary vertices are sorted based on their gains. The gain of a boundary vertex is defined as the reduction in the edge cut obtained by moving that vertex from one partition to the other partition. After the sort, boundary vertices are moved between adjacent partitions if doing so reduces the edge cut while maintaining the balance among the partitions.

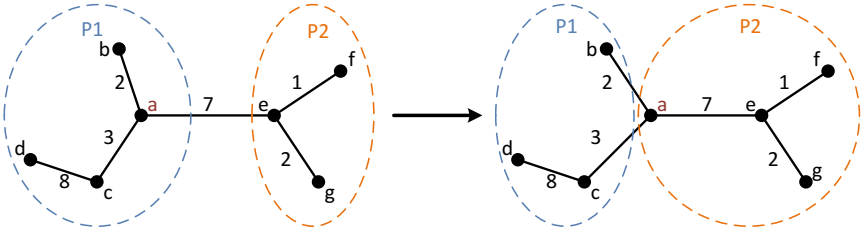



Figure 2.8: Positive gain for vertex $a$ and transferring from $P_1$ to $P_2$.

Figure 2.8 shows an example of *gain* calculation for vertex $a$ when it moves from $p_1$ to $p_2$. The sum of the weights of interior edges (edges inside the partition) connected to $a$ is (2+3), while the sum of the weight of exterior connected edges (crossing edges) is 7.

By transferring $a$ from $p_1$ to $p_2$ the achieved *gain* is calculated as:

$$gain(a) = \sum_{k \in p_2} W_{a,k} - \sum_{k \in p_1} W_{a,k} \tag{2.4}$$

Therefore if this movement does not make any of the two partitions unbalanced, it reduces the edge cut by 2.

Jostle uses a combined balancing [36] and refinement algorithm. This approach accepts a vertex movement from one partition to another even if it makes the partitions unbalanced, while in the immediately following refinement step, the vertex movement is rejected or accepted.

### 2.5.2 Parallel Multilevel Graph Partitioning on Distributed Systems

Here we discuss a few notable parallel multilevel graph partitioning solutions and how their strategies need heavy revision for GPU applicability.

Several parallel multilevel graph partitioning algorithms for distributed-memory systems have been proposed [19, 40, 48, 49, 92]. Parallelizing the coarsening and un-coarsening phases is challenging because of their highly serial and data-dependent nature.

ParMetis [48] implements a coarse-grained parallel graph partitioning, which improves performance compared to the fine-grained parallel algorithm [49]. Initially, each processor receives $n/p$ vertices, where $n$ is the number of graph vertices and $p$ is the number of processors in the cluster. The matching phase consists of two passes: in the even numbered passes, each vertex $v$ on processor $p$ sends a match request to its corresponding vertex $u$ on other processors using *HEM*, but only if $v > u$. Correspondingly, in the odd numbered passes, a vertex $v$ sends its request only if $v < u$. After a few passes, a maximal set is reached and the matching phase terminates. At the end of each iteration, a synchronization step is required in which each processor sends its match requests in one single message to the corresponding processors and subsequently receives the requests from other processors. Based on the received information, the processors decide in parallel how to collapse the vertices to create the next coarser graph.

The initial partitioning phase in ParMetis starts with an all-to-all broadcast of vertices among the processors. Each processor performs a recursive bisection algorithm [47], where the processor completes one branch of the bisection tree. At the end, each processor stores the vertices that belong to its assigned part of the $k$ partitions.

In the un-coarsening step, each processor first projects back its assigned vertices onto the finer graph. Then the same ordering method as in the coarsening step is applied in multiple passes. At the end of each pass, the requests for movement

of vertices across the partitions are communicated among the pro-cessors, and the movements that do not violate the balance constraints are committed.

PT-Scotch [19, 40] follows a Monte-Carlo approach in the matching phase. Each node sends its match request based on the *HEM* method with the probability of 0.5. The results show that, after a few iterations, a large part of the vertices are matched. To reduce the communication overhead among the processors, a folding technique is used after several coarsening levels in which the vertices of the coarser graph are duplicated and redistributed to two groups, each to $p/2$ of the processors. The two groups can continue the matching phase independently. This folding process continues recursively ($p/4$, $p/8$, ...) until each sub-graph is reduced to a single processor. Then a serial recursive bi-sectioning is performed on each processor and the best initial partitioning is chosen for the un-coarsening phase.

During the refinement phase of PT-Scotch, a banded diffusion technique [75] is utilized in which the refinement phase executes on a banded graph extracted from the initial partitioned graph. This banded graph consists of the set of vertices that are located at a specific threshold distance from the partition separators.

Parallel Jostle [92] could face high communication overhead if it continued matching until the number of vertices equals the number of required partitions. So, after reaching a threshold in the coarsening phase, an all-to-all broadcast is executed before each processor continues independently to coarsen down to a single vertex. During the un-coarsening phases, each partition creates its own set of boundary vertices with the same target partition preference, e.g., partition $p_1$ constructs a set of its boundary vertices with the preferred target partition $p_2$. At the same time, partition $p_2$ creates a similar set of vertices for partition $p_1$. Consequently, these two sets form an interface region. A serial optimization technique, e.g., KL [51], is executed independently on the different regions. This technique mitigates the communication-intensive vertex movements by isolating different regions of the graph.

It should be noted that proposed solutions for distributed systems such as ParMetis,

PT-Scotch, and Parallel Jostle, suffer from high overhead posed by inter-process communication. In the coarsening phase, each process scans its assigned vertices and sends the matching requests for the non-local vertices to the corresponding processors. Also, in the refinement step, the processors have to communicate their requests for movement of non-local vertices across the partitions. These requirements make existing parallel solutions incur high volume of communication among processors in both coarsening and un-coarsening phases.

### 2.5.3 Parallel Multilevel Graph Partitioning on Shared-Memory Systems

The shared-memory graph partitioning in multi-core systems [57, 88] allow finer specification of parallelism. However, concurrent update of memory locations obliges handling the new challenges like memory incoherency and race condition for the coarsening and un-coarsening phases.

Gmetis [88] extends a version of Metis to a multi-core platform using the Galois programming model [54], which is a sequential object-oriented programming model that supports parallel set iterators. Each Galois iterator may add new elements to the set. However, this approach is found to be not as efficient as ParMetis in terms of performance.

Mt-metis [57] is a multicore graph partitioner based on the Metis algorithm which applies the concept of ParMetis in the shared-memory system implemented using OpenMP, and achieves better performance than MPI-based distributed graph partitioners. Primarily, the $n$ graph vertices are divided among the $t$ threads and each thread finds the matches for $n/t$ vertices assigned to it. The lock-free matching step of mt-metis is split into two rounds. In the first round, all the threads process the neighbor list of their assigned vertices serially and read from and write to the matching vector freely without any synchronization. In the second round, the matching conflicts are resolved and each thread finalizes the matching of its assigned vertex.

In the initial partitioning step, each thread partitions the graph into two bisections. Then the best bi-section with the minimum edge cut is selected and half of the threads work on one of the bisection and half of them partition the other bisection recursively.

The refinement is performed in two steps as well, and the moving direction of vertices among the partitions is reversed after each round. Moreover, each thread has an assigned buffer for inserting the vertex movement requests of the other threads. At the end of each round, the threads process their buffer and confirm or undo the movements to meet the balance constraints.

GPU allows thousands of threads to be resident on its Streaming Multiprocessors. Therefore, an acceptable implementation of GPU graph partitioning must utilize the collective computation force of threads within the warps efficiently. This prominent difference makes proposed approaches on multi-core systems not applicable on GPUs in a straightforward way. An instance of where the GPU paradigm can be troublesome is the coarsening phase where each thread finds the match for its assigned vertex. Here, traversing irregular-sized neighbor lists by different threads decreases warp execution efficiency. Moreover, in the refinement step, simultaneous access of many GPU threads to the list of candidate boundary vertices for each partition not only creates a high memory contention but also unlike mt-metis it is inefficient if taking an action needs to be undone due to inter-partition imbalance. Above all, non-coalesced memory loads when accessing the neighbor list of the graph vertices impose high latency costs. Due to these GPU-specific issues, proposed parallel techniques for distributed and multi-cores environments are not directly applicable onto GPUs.

### 2.5.4 Matching Algorithms on GPU

A few studies address the graph matching on GPU [2, 3, 35, 64], exclusively and in isolation with respect to other parts of graph partitioning procedure. Nevertheless, in these solutions each thread processes the irregular sized neighbor list of vertices

serially which results in load-imbalance and thread divergence and consequently degrades the performance when employed iteratively during the graph partitioning process. Also, such solutions do not address other challenges that arise when designing a high-quality graph partitioner fully implemented on the GPU. Balancing the load among the GPU threads during the contraction and refinement steps, and efficiently keeping the intermediate graphs with minimal footprint on GPU DRAM with limited size are among these challenges.

Fagginger Auer and Biddeling [2] propose a fine-grained parallel greedy matching algorithm on the GPU. their solution randomly colors the graph vertices either blue or red and it works in two phases in each iteration. In the first phase the blue vertices propose to red neighbors. In the second phase the red vertices reply to the proposals and choose just one of the neighbors based on the matching criteria. The matching continues iteratively for the remaining vertices until the maximal matching is obtained, that is, all the vertices are either matched or they have no matching proposal.

Birn et al. [9] present another algorithm which repeatedly traverses the edge list locating dominant edges (called local max). The algorithm starts with an empty matching set. Every vertex tries in parallel to find the adjacent heaviest edge. If a vertex $v$ finds an edge $vw$ as local maximum and the other end point $w$ also finds that $wv$ as the local maximum, then the edge $vw$ is added to the matching set and its adjacent edges are removed from the graph. This process repeats until there is no more edge left to be matched.

Naim et al. [72] propose a variation of heavy edge matching matching method in which, the warp threads collaboratively process the neighbor lists of a single warp-assigned vertex. However, each warp processes its assigned vertices in serial and if the size of a vertex neighbor list is not a multiple of warp size, some of the warp threads remain idle which leads to poor warp execution efficiency.

Manne et al. [64] apply a greedy matching algorithm in computation of a stable marriage solution on GPU. In the greedy matching algorithm, first the graph edges

are sorted. In each iteration of the algorithm, the heaviest remaining edge $u, v$ is included in the matching before removing any edge incident on either $u$ or $v$.

Fagginger Auer and Bisseling [3] present a multilevel graph coarsening approach to perform agglomerative clustering using GPUs. Their matching procedure [2] can adapt to star-like structures in the graph to avoid insufficient parallelism due to small matchings. However they do not provide any local refinement mechanism. Furthermore the algorithm does not save the intermediate graphs on GPU and only keeps the first and last level graphs.

## 2.6 Optimization Techniques on GPU

Here we explain some of the proposed GPU-based optimization techniques for the fast parallel execution of *scan* and *reduce* operations, which are the core primitives of parallel computing and we employ those operations in our graph partitioning design. We also discuss the atomics instructions on GPUs.

### 2.6.1 Scan

The all-prefix-sums operation on an input sequence of values is commonly known as scan. *Scan* is a fundamental function that is applied as a base in many parallel algorithms.

The new generation of Nvidia GPUs provides specific instructions [38] when the input elements are binary, which improves the efficiency of *scan* and memory space requirements by allowing the threads of a warp to operate concurrently. One of these instructions is `__popc()`, which counts the number of bits that are set to 1 in a 32-bit integer and is compiled to one machine instruction. Each warp thread can call __popc and get the results of the sum for all of the 32 warp assigned bits. Another instruction is `__ballot()`, which collects the binary values across parallel threads, counts the number of bits set to 1, and returns this value to every thread in the warp.

Figure 2.9 shows an example of an intra-warp exclusive binary prefix sum. In

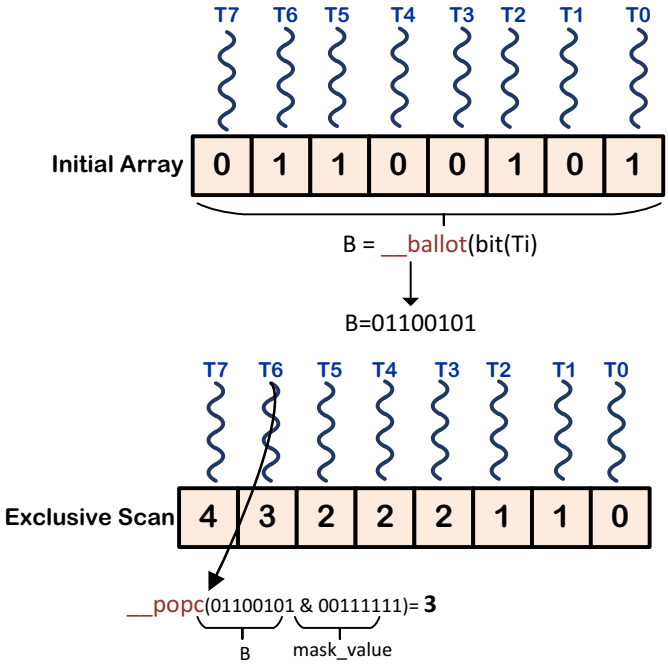



Figure 2.9: Intra-warp binary exclusive scan. Warp size is assumed 8.

the first step, all the warp threads call `__ballot()` by passing their assigned bit to this instruction. The returned value of `__ballot()` is called `B`, and it is accessible by all the threads. Then each thread calls `__popc()` on the returned value of (`B & maskvalue`) to count the number of bits set to 1. `Maskvalue` is calculated by setting the bits in the positions lower than each thread ID to 1 and 0 in the other positions. As figure 2.9 shows, for thread 6 the *maskvalue* is `00111111` since the bits 0 to 5 are set to 1, and the bits 6 and 7 are set to 0.

The scan can be extended across an entire thread block by using an intra-warp scan; first, the warp-level binary prefix sum is executed and one of the threads in each warp writes the partial results to an intermediate array of length the number of warps inside the block. Then, the first warp executes an exclusive prefix sum on the partial results. Finally, the warp threads collect the cumulative sums from the previous step and add them to the value they calculated in the intra-warp prefix sum.

### 2.6.2 Reduction

*Reduction* is an important optimization technique that is used to reduce the elements of an array into a single result. The most common reduction operation is computing the sum of a large array of values. Other operations are, for example, reducing on the maximum or minimum value across an input set of values. Since reduction is commutative and associative, the elements can be re-arranged and combined in any order. Sequential addressing reduction is an efficient intra-warp reduction method proposed for the GPU [37] in which, the second half of the warp assigned values are added pairwise to the first half by a leading set of threads. Figure 2.10 shows the

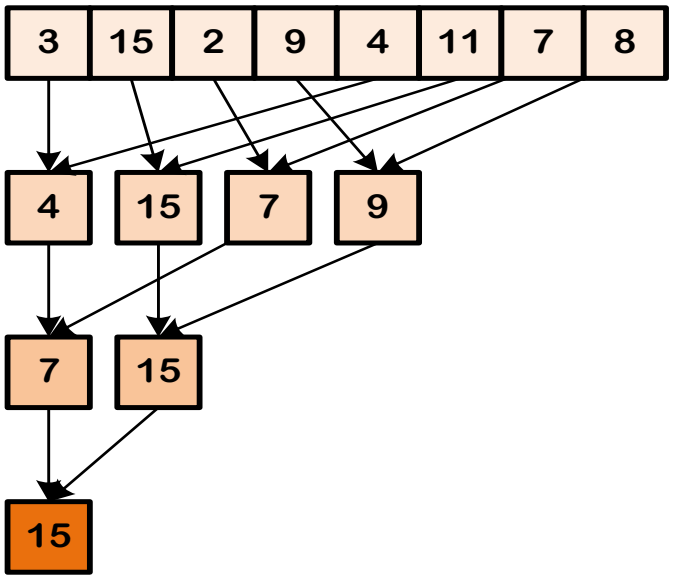



Figure 2.10: Intra-warp reduction on the maximum value. Warp size is assumed 8.

intra-warp reduction on the maximum value using the sequential addressing method that has $\log_2(warp\_size)$ steps.

The new generation of Nvidia GPUs provides a shuffle instruction (SHFL) [61], which enables the threads inside a warp to directly read a register from the other threads inside the warp. This enables the warp to collectively exchange or broadcast data without requiring access to the shared memory. SHFL is a faster mechanism for moving data between threads in the same warp. One intrinsic of SHFL is

`__shfl_xor((variable, lane_Mask)`, which calculates a source lane ID (thread id within the warp) by performing a bitwise XOR of the caller's lane ID with laneMask, and the value of *variable* held by the resulting lane ID is returned. This variant can be used to do an efficient reduction across the warps.

### 2.6.3 Atomics

GPU allows thousands of threads to be resident on its Streaming Multiprocessors. Since different blocks on GPU have to communicate through global memory, hardware atomic operations are crucial to keep memory consistency, when the threads access the shared locations on the global memory. The atomic operations are employed to avoid the race condition and to ensure the order of the write and read accesses to the memory. The atomic operations impose too high a performance overhead by serializing the memory accesses between thousands of threads on GPU [74]. However, in the new generations of GPUs such as Kepler, the architectural support for the atomic operations has evolved significantly. Consequently, the performance and efficiency of the atomic instructions (`atmocMax()`, `atmicAdd()`, etc.), have been improved substantially.

# Chapter 3

# A Dynamic Scheduling Heuristic for Embarrassingly Parallel Applications on Heterogeneous CPU-GPU Systems

In this chapter we address the problem of scheduling for embarrassingly parallel applications on a heterogeneous CPU-GPU system equipped with single or multiple GPUs. We design and implement a new, dynamic scheduling heuristic for the embarrassingly parallel applications, where tasks have no inter-dependencies. The scheduler is designed to distribute the load at a suitable rate between all the available processors in such a way that the application execution time is minimized. Meanwhile, the scheduler aims to take advantage of the full processing power of the GPU processing cores and to minimize the user interference in the scheduling criteria. Our proposed dynamic scheduler is scalable to any number of GPUs integrated in a heterogeneous CPU-GPU system.

We employ several runtime techniques such as profiling and work stealing to adapt the load distribution based on the processing power of the processors during heterogeneous execution and speed up the execution time of the embarrassingly parallel

applications.

Our developed scheduler outperforms the static min-min [78] and dynamic greedy- [95] heuristics on a system with a single CPU and multiple GPUs. It achieves similar performance in comparison to the Qilin [62] heuristic on a single CPU–single GPU system. However, unlike Qilin, our dynamic scheduler is applicable to heterogeneous CPU-GPU platforms integrated with more than one GPU, and it is also less sensitive to the size of the training data. We integrate our dynamic scheduler into a scheduling framework [99] that hides the scheduling complexities from the user and automatically distributes the loads over the processors.

## 3.1 Motivation

Parallelizing embarrassingly parallel applications comprising large groups of independent tasks on a heterogeneous CPU-GPU system is straightforward. However, to achieve high performance and take advantage of the potential processing power of all of the processors, the tasks need to be distributed over the CPU and GPUs at proper rates.

Unlike traditional heterogeneous systems, the specific features and differences in the programming model of the CPU and the GPU should be considered when designing scheduling algorithms for heterogeneous accelerator-based systems. Consequently, the scheduling methods proposed for heterogeneous clusters [8, 13, 63, 87, 98] or multicore environments [18, 97] are not applicable to CPU-GPU systems. Current GPU programming models cannot make decisions regarding the distribution of load over the CPU and GPU. Therefore, the programmer manually embeds in the CPU code and the kernel the information about the ratio of loads on each device. In Chapter 2 we discussed the proposed static and dynamic scheduling algorithms on CPU-GPU systems, which suffer either from performance or scalability issues. The static methods [33, 59, 94] are only applicable to systems integrated with a single GPU and have

a heavy and inaccurate training phase for large loads. The addressed dynamic techniques [44, 82, 84, 89] underutilize the GPU processing cores and some of them are not scalable to more than one integrated GPU. The other dynamic methods require many scheduling hints from the user and the scheduler cannot schedule the loads over the processing devices independently. As a result, it is a non-trivial task to design a dynamic and scalable scheduling heuristic for embarrassingly parallel applications that minimizes user programming effort.

## 3.2 Scheduler Architectural Model

Traditionally the software architecture of embarrassingly parallel applications can be modeled with a work pool where a *Task Generator* generates a set of parallel independent tasks to the pool, and these tasks are assigned to different workers. Technically, the parallel tasks compute the independent output data (also known as *result*) set $\{ out(in_1), out(in_2), ..., out(in_n)\}$ from the input data (load) set $\{ in_1, in_2, ..., in_n\}$ [83]. The workers all have identical tasks, i.e., they have the same code, but operate on different input loads. When the workers finish the computation, they send the results to the *Result Collector*. Figure 3.1 shows the software model of the embarrassingly parallel applications scheduler.

To schedule the tasks in the work pool, two steps must be defined: the task definition and the task assignment to the workers. The first step depends on the application, whereas the second step is more general. Task assignment can be static, which means each worker executes a fixed portion of the tasks in the pool that is known a priori. In dynamic task assignment, the work-pool tasks are assigned to the workers in a dynamic way, and workers can ask for more assignments if they finish their original assigned portion of the load.

We employ the dynamic assignment model in our designed scheduler. The underlying architecture platform is composed of a single CPU and multiple GPUs. The

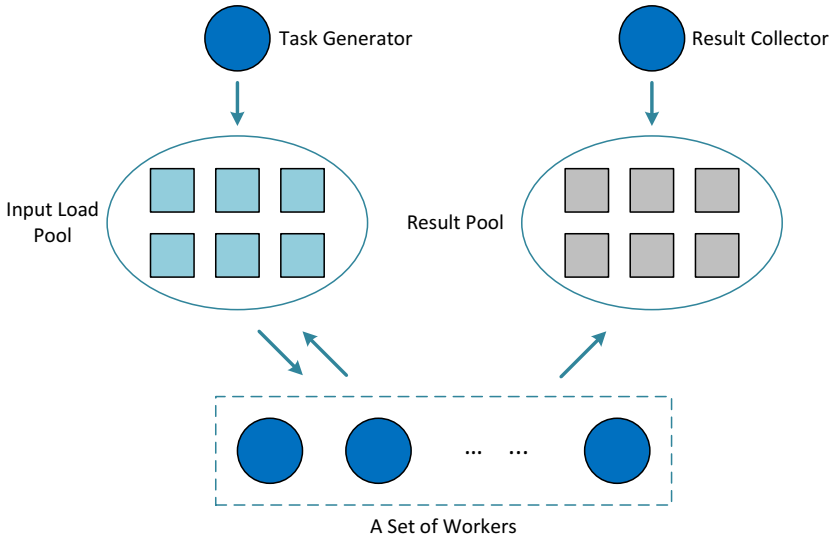



Figure 3.1: Dynamic scheduling architecture for embarrassingly parallel applications.

scheduler works as an intermediate layer sitting between the task pool and the workers and distributes the load over the CPU and GPUs adaptively to minimize the makespan.

Figure 3.2 shows a schematic representation of our scheduler architecture. In addition to the main dynamic scheduler, the architecture employs two plug-in modules as well: A partitioner module for the appropriate breaking of the input loads into smaller chunks, and a load bundling module that bundles smaller loads assigned to each device to reduce the transfer latency. We elaborate on these two modules in the following sections.

### 3.2.1 Partitioner

The partitioner, an additional plug-in module to the scheduler, divides a given load into smaller independent chunks. It is a function template filled in using an application-specific partitioning strategy, and is then supplied to the scheduler. The scheduler uses the partitioner to divide a load that has been extracted from the task pool into ideal-sized chunks that can then be scheduled by the scheduling criteria. The scheduler determines this ideal size at run time based on the application and underlying

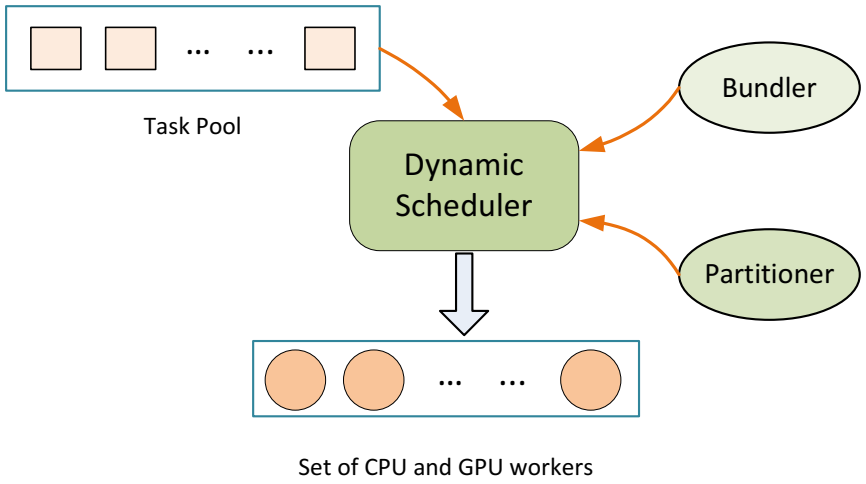



Figure 3.2: Our designed dynamic scheduler architecture.

processing system architecture.

To clarify, the partitioner function can be considered to be a matrix multiplication problem. Depending on the implementation of the application, we can assign one row of the first matrix and one column of the second matrix to each processing core or assign one element in a row and one element in a column to each process. So, the sub-load size created by the partitioner may be different.

Figure 3.3 illustrates the partitioner module. When the scheduler extracts a load from the task pool, it calls the partitioner function to break down the load into a set of independent sub-loads. Then the scheduling algorithm determines the sub-loads split ratios.

### 3.2.2 Load Bundler

The load bundler wraps multiple small loads into a single large load and assigns this one large load to the GPU devices. To design an efficient dynamic scheduler for heterogeneous CPU-GPU systems, we must consider some critical features of the CPU-GPU architecture. In contemporary CPU-GPU systems, the interconnection bandwidth between CPU and GPU is fairly high, on the order of 12 gigabytes per second or even more with newer interconnection technologies like the PCI-X bus

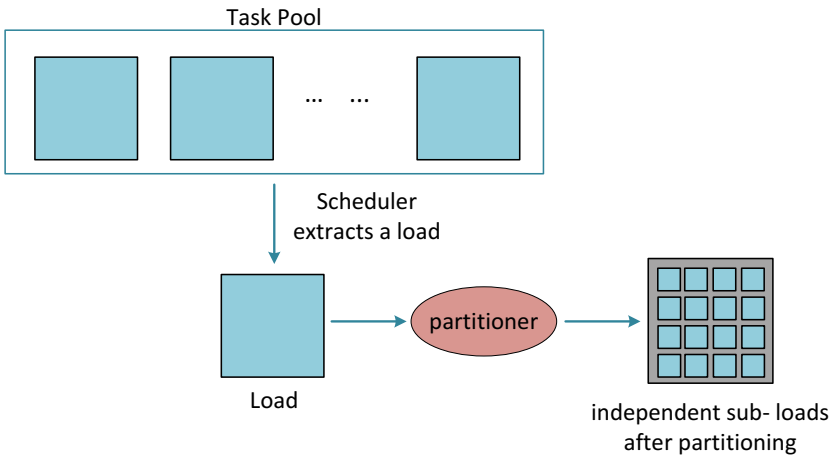



Figure 3.3: Our partitioner function.

and its extensions. Considering such high bandwidths, data transfer times between CPU and GPUs are much smaller as compared to a cluster or a grid environment. Consequently, when the data size is small, the message startup latency becomes the predominant factor in the total data transfer cost between the CPU and GPUs. Hence, load bundling can improve the data transfer performance.

In addition to data transfer performance, bundling can also reduce GPU execution latency. Since the input load can be arbitrarily partitioned into independent sub-loads, accordingly multiple independent sub-loads can also be bundled together. In other words, CPU will offload a coarser grain load to the GPU. Load bundling achieves a better match between the parallel computing of each GPU device and the workload size that has been assigned to it.

There are three reasons to implement load bundling: (1) If a load is small, it is possible that GPU will be underutilized. Considering that each GPU thread is mapped to a core's streaming processors (SP) to operate on a part of the load, a smaller load could render many of the cores idle. Hence, bundling of the loads into a single load of a suitable size will result in better utilization of the GPU. (2) Each CPU-GPU data transfer has message startup latency that is independent of the message size. Hence, bundling of multiple loads into a single load prior to transfer can minimize this transfer latency. (3) Each thread invocation on the GPU has a startup latency

(a.k.a., kernel start-up time) that is much higher than the startup latency for a function call on the CPU. Multiple loads will result in multiple kernels launching, while load bundling can minimize the number of spawning operations.

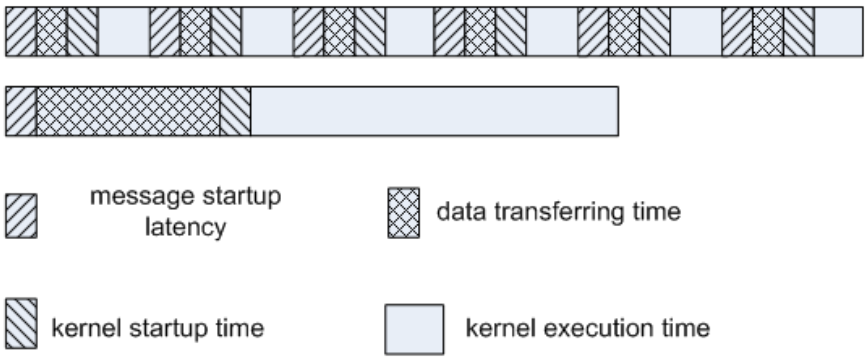



Figure 3.4: Schematic showing the reduction of processing time due to load bundling.

Figure 3.4 illustrates the impact of load bundling on the overall GPU processing time. The processing time is reduced due to the combination of reductions from the message start-up latency and kernel start-up time.

## 3.3 *HASS*: A Dynamic Scheduling Algorithm

Our proposed dynamic scheduling algorithm operates on a set of loads deposited to the task pool. The main purpose is to distribute the loads over the CPU ($p_c$) and a set of GPUs $\{ p_{g_1}, p_{g_2}, ..., p_{g_n}\}$ in such a way that minimizes the total execution time. To reach this goal, the distribution ratio sets $\{ ratio_{p_c}, ratio_{p_{g_1}}, ratio_{p_{g_2}}, ..., ratio_{p_{g_n}}\}$ ideally need be determined such a way all the processing devices complete their work at almost the same time:

$$T(ratio_{p_c} \times L) \approx T(ratio_{p_{g_1}} \times L) \approx T(ratio_{p_{g_2}} \times L)... \approx T(ratio_{p_{g_n}} \times L) \qquad (3.1)$$

In Equation 3.1, $(ratio_{p_{g_i}} \times L)$ represents the fraction of the total load size ($L$) that will be assigned to $p_{g_k}$ (the $k^{\text{th}}$) GPU), and $(ratio_{p_c} \times L)$ represents the fraction

of the load that will be assigned to the CPU.

The dynamic scheduling algorithm operates iteratively. In each round, a divisible load is extracted from the task pool and is divided into a set of independent sub-loads by calling the partitioner function. Then, a scheduling algorithm *HASS* (*Heterogeneous Architecture Scheduling Strategy*) is invoked.

The *HASS* function distributes the sub-loads to the CPU and GPUs dynamically based on the information recorded from the previous rounds. *HASS* employs a profiling process to estimate the initial fraction of loads needed to be assigned to each processing device. In other words, the profiling procedure determines the initial rate of load execution on each device based on the characteristics of the workload.

In our profiling approach, we calculate the execution time of a small portion of the load on each device and based the collected date determine an initial distribution of actual load size over the devices. The overhead of our profiling approach is much lighter than Qilin [62], and it can be performed online at run time.

After the initial distribution of the sub-loads over the available processors, if a GPU device has finished the execution of its assigned sub-loads, it applies a work stealing technique [10, 17] to ask for more work in such a way to prevent performance degradation from underutilization of the processors. At the end of each round, the final distribution ratio for all of the devices are recorded to be used in subsequent rounds. This adaptive information helps in improving the load partitioning as rounds continue. Repetition of work stealing and ratio adaptation in the subsequent rounds finally reaches the ideal distribution rates over the CPU and GPUs. The algorithm terminates when the task pool becomes empty.

Algorithm 3.1 illustrates the *HASS* which operates in three phases: *initialization phase*, *execution phase* and *adaptation phase*. In the rest of this section, we elaborate on details of each phase.

**Algorithm 3.1** HASS Algorithm.

1: **procedure** HASS
2: input = A set of equal size sub-loads $d_1, d_2, ...d_u$ with the total size of $m$. The loads are to be executed on processors $\{p_c, p_{g_1}, p_{g_2}, ..., p_{g_n}\}$
3: output = Loads executed with minimized makespan
4: $round = 0$
5: **while** loads are still generated by the task generator **do**
6: Extract a load $l_i$ from the pool and partition it into $u$ independent sub-loads according to the partitioner criteria
7: ***/*initialization phase*/***
8: **if** $round = 0$ call **profiling procedure**
9: initialize $i$ to $ratio_{p_c}$ // $i$ is the CPU split ratio: refer to equation 3.4
10: **for** $k = 1$ to $n$ **do**
11: initialize $j_k$ to $ratio_{p_{g_k}}$ // $j_k$ is $GPU_k$ split ratio: refer to equation 3.5
12: ***/*execution phase*/***
13: **for** $k = 1$ to $n$ **do**
14: put $j_k * m$ sub-loads ($PRT_{p_{g_k}}$) into $p_{g_k}$ buffer ($b_{g_k}$)
15: bundle all sub-loads in $b_{g_k}$ to a single load $D_{g_k}$
16: assign $D_{g_k}$ to $p_{g_k}$
17: **endfor**
18: put $i * m$ sub-loads ($PRT_{p_c}$) into $p_c$ buffer ($b_c$)
19: **while** $b_c$ is not empty **do**
20: **if** $p_c$ is idle **then**
21: remove a sub-load $d_i$ from $b_c$ and assign it to $p_c$
22: **for** $k = 1$ to $n$ **do**
23: **if** $p_{g_k}$ is idle **then**
24: **/*Steal work from CPU's buffer*/**
25: remove a sub-load $d_i$ from $b_c$, and assign it to $p_{g_k}$
26: ***/*adaptation phase*/***
27: update $ratio_{p_c}$
28: **foreach** $p_{g,k}$
29: update $ratio_{p_{g_k}}$
30: $round++$
31: **endwhile**

### 3.3.1 Initialization Phase

In the first round of *HASS*, for a balanced estimation of the initial load distribution, we profile a small (arbitrary) fraction of the load on each device. From this sample profile, we decide how to initially distribute the total input load over the CPU and GPUs based on the measured execution rates on each processing device. We individually execute the partial size of the load on the available set of processors, $P = \{ p_c, p_{g_1}, p_{g_2}, ..., p_{g_n}\}$, and calculate the *makespan* for each processor.

Determining the proper size of the initial profiling is a crucial factor. On one hand, if a large load volume is assigned to the GPUs, the CPU threads may finish their assigned load early and become idle. On the other hand, if the initial proportion of load that is assigned to the GPUs is too small, it may result in underestimating the processing power of the GPUs due to the lack of sufficient workload. Also, an increase in the work stealing rate requested by the GPUs to the CPU imposes overhead to the performance.

Ideally, the initial load size should be large enough to fully utilize the GPU cores. To minimize the profiling overhead, the load size can be set to the maximum number of streaming processors (SP) of the available GPUs. This load will then be executed on different devices and the makespans are measured.

The makespan of execution of an arbitrary problem size (*arbitrary_size*) is used to calculate parameter $S$, which represents the load size that a processor can execute per second (i.e., $S_{p_c}$ for the CPU, $S_{p_{g_k}}$ for the $k^{\text{th}}$ CPU)( Equations 3.2 and 3.3 in the following).

$$S_{p_{g_k}} = \frac{arbitrary\_size}{makespan_{p_{g_k}}} \tag{3.2}$$

$$S_{p_c} = \frac{arbitrary\_size}{makespan_{p_c}} \tag{3.3}$$

Then, the load split ratios are computed for each processing device according to Equations 3.4 and 3.5:

$$ratio_{p_{g_k}} = \frac{S_{p_{g_k}}}{\sum_{k=1}^{n} S_{p_{g_k}} + S_{p_c}} \tag{3.4}$$

$$ratio_{p_c} = \frac{S_{p_c}}{\sum_{k=1}^{n} S_{p_{g_k}} + S_{p_c}} \tag{3.5}$$

Finally, the above profiling information is used to determine the portions of the total load size ($PRT$) assigned to each device using Equations 3.6 and 3.7. $PRT_{p_c}$ represents the portion of the total load that will be assigned to the CPU, and $PRT_{p_{g_k}}$ represents the portion of the total load size($L$) that will be assigned to $P_{g_k}$ (the $k^{\text{th}}$ GPU).

$$PRT_{p_{g_k}} = ratio_{p_{g_k}} * realproblemsize \tag{3.6}$$

$$PRT_{p_c} = ratio_{p_c} * realproblemsize \tag{3.7}$$

### 3.3.2 Execution Phase

In the *execution phase*, loads are transferred from the processors' individual buffers (in CPU memory) to the processors. Prior to transmission to a GPU, the loads inside the GPU's buffer are bundled together by the load-bundling module and are assigned to the GPU as a whole for reasons discussed in the previous subsection.

During the *execution phase*, a work stealing mechanism [10, 17] is employed for balancing loads between the CPU and the GPUs. If one or more GPUs finish their work, they steal loads from the CPU's buffer, if available.

It should be mentioned that in the original scheduler model for embarrassingly parallel applications, an idle worker pulls work from the task pool. However, since a GPU is a subordinate processor of the CPU, its workload needs to be pushed (by the CPU) rather than transferred by a pull.

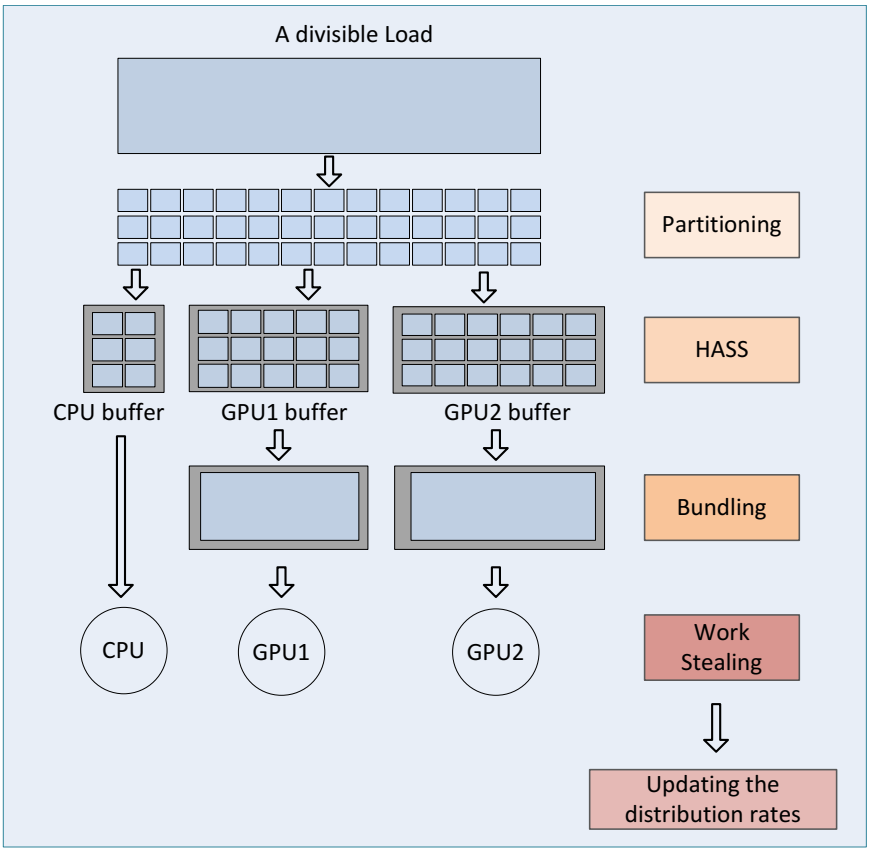



Figure 3.5: Scheduler flowchart over one round.

### 3.3.3 Adaptation Phase

The work stealing mechanism during the *execution phase* changes the initial ratio of load distribution. Therefore at the end of one round an *adaptation phase* is employed, during which the load distribution to the processors is adjusted based on the final portion of load executed by any of the processors in the *execution phase*. The use of the *adaptation phase* improves the performance in the subsequent rounds. If the load distribution ratio remains the same for two consecutive rounds, the scheduling distribution is finalized. Figure 3.5 shows the flowchart of our designed scheduler over one round.

## 3.4 Experimental Evaluation

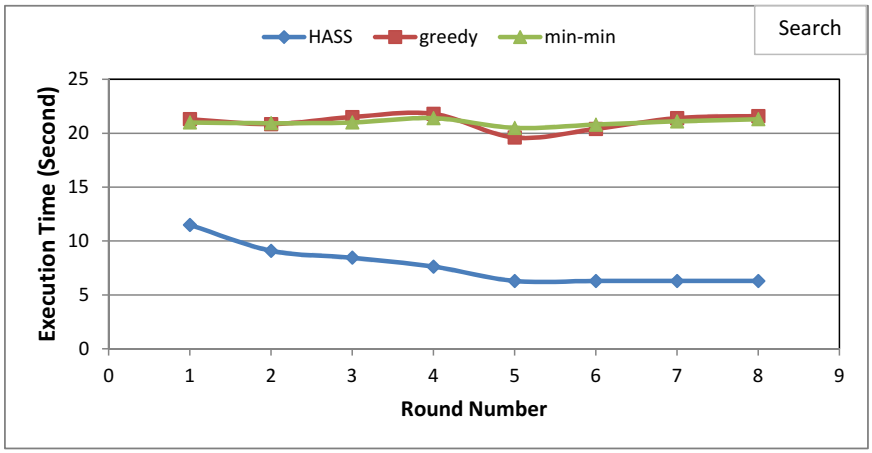
In this section, we discuss the performance results of HASS compared to three of the contemporary static and dynamic heuristics: Qilin [62] and min-min [78], which are static heuristics and greedy [95], which is a dynamic heuristic. We use two benchmarks for our experiments: a string search application that finds all strings that match certain patterns in a text file and a dense matrix multiplication application. In order to eliminate this anomaly of small loads, the loads chosen for these experiments have the size that is not less than the number of SPs of the GPU with the most number of SPs.

The system we performed experiments on is equipped with an Intel Xeon E5540 processor with 4 cores; 6 gigabytes of main memory; one Nvidia Tesla C1060 GPU, which has 30 Stream Multiprocessors (SM) with each having 8 Streaming Processors (SP) (i.e., 240 SPs in total); and one Nvidia Quadro600 GPU, which has 12 SMs each comprising 8 SPs (i.e., 96 SPs in total). We used the STL (Standard Template Library) [79] of C++ on the CPU and CUDA on the GPU.

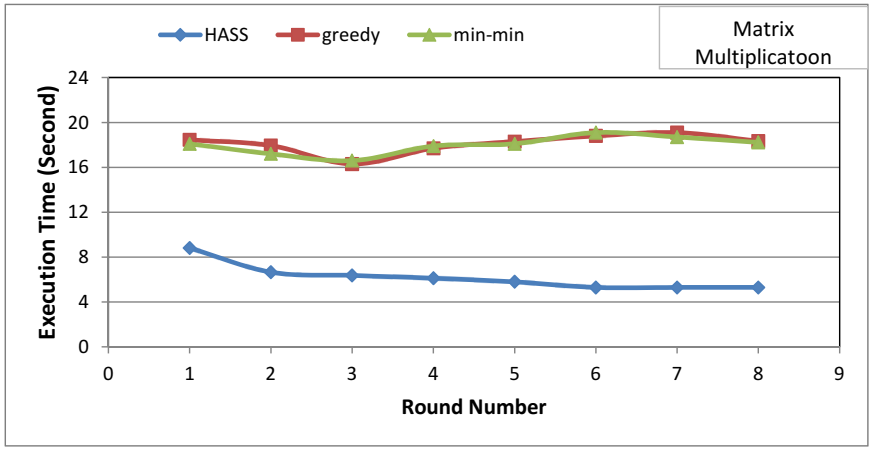
Figure 3.6 shows the execution time comparison of *HASS* with min-min and greedy heuristics for the search and matrix multiplication applications, respectively. We were not able to compare the *HASS* performance with the Qilin heuristic in a heterogeneous system with 2 GPUs, because Qilin works only on a single-CPU single-GPU system. In each round of the execution, the same amount of load is scheduled to the system and three scheduling strategies are applied.

As Figure 3.6 shows *HASS* outperforms the greedy and min-min scheduling heuristics for both applications, and its performance improves over successive rounds due to adaptive scheduling. According to Figure 3.6, we achieve the best distribution rates at round 6, and, consequently, the load distribution rates over the processors remain the same in the subsequent rounds.

We also measure the load distribution rates in each round for a one CPU and two GPU system configuration. Figure 3.7 shows the load distribution among the




(a) Search application on a text file with size of 50 MB.




(b) Matrix Multiplication application with matrix size of 6000*6000.

Figure 3.6: Execution time comparison in a single-CPU multi-GPU system.

processors for the matrix multiplication benchmark in this system configuration. As shown in the Figure, the load distribution improves over successive rounds until it reaches to an optimal rate in the $6^{th}$ round.

Next, we consider a heterogeneous system with one integrated GPU to be able to compare our scheduler performance with Qilin as well. Figure 3.8 shows a execution time comparison between *HASS*, Qilin, greedy and min-min in a single CPU-single GPU configuration (Nvidia Tesla C1060). As shown in the Figure 3.8, *HASS* has significantly better performance in comparison to the greedy and min-min heuristics. Also with increasing round number, the makespan given by *HASS* improves and

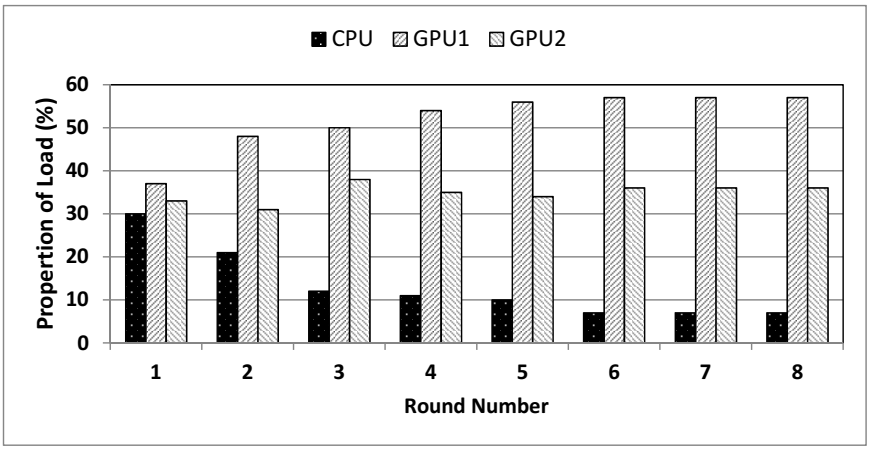



Figure 3.7: The distribution of loads over the three processors in Matrix Multiplication when using *HASS*.

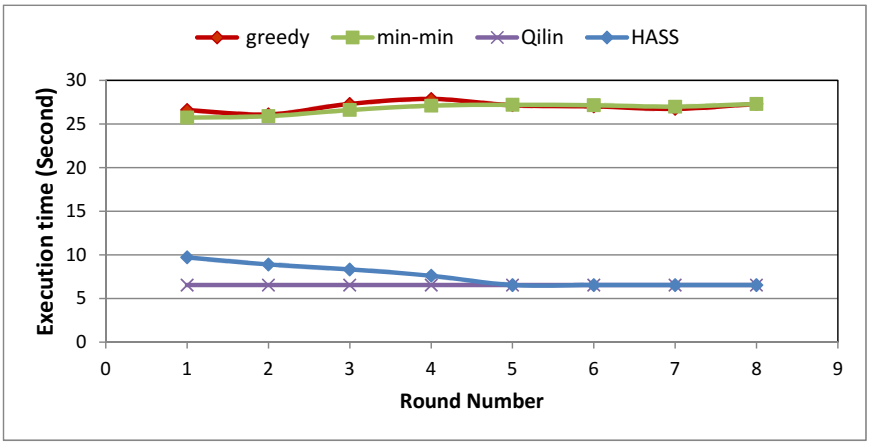
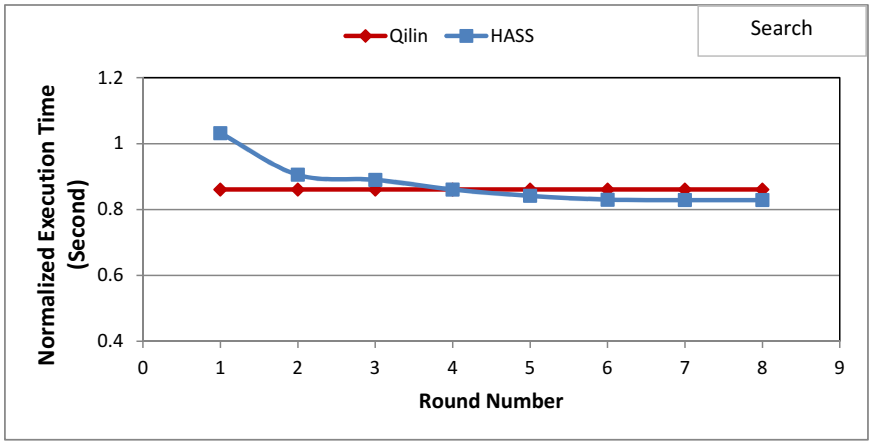


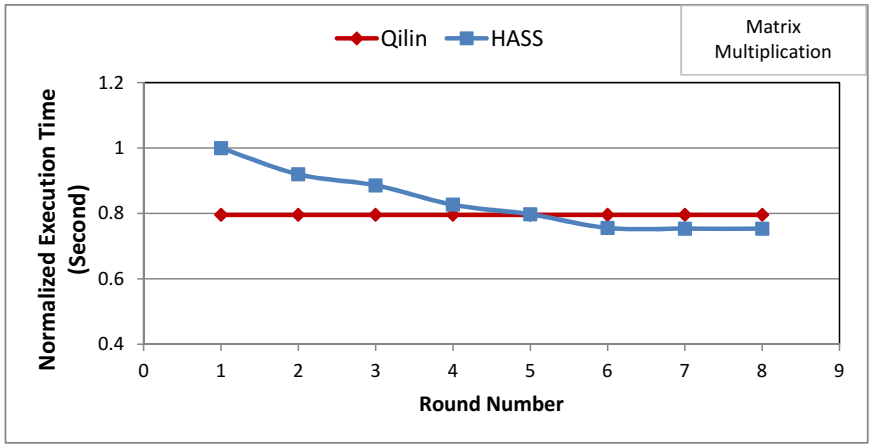

Figure 3.8: Matrix Multiplication execution time comparison of *HASS* with the three other scheduling algorithms in a single-CPU single-GPU system.

approaches that given by Qilin.

As discussed in Chapter 2, one main drawback of the Qilin method is its distribution precision dependency to the training data size. If the load size in the training phase is less than 30% of the actual problem size, the scheduling policy is likely to be sub-optimal. To compare the sensitivity of *HASS* to the problem size in the training phase, we consider the training size less than 30% of the actual input size for both *HASS* and Qilin and evaluate their makespans for the matrix multiplication application with the input matrix size of 10000*10000. Figure 3.9 demonstrates the normalized execution time comparison. *HASS* outperforms the Quilin after round 5,




(a) Search application on a text file with size of 50 MB.




(b) Matrix Multiplication application with matrix size of 10000*10000.

Figure 3.9: Execution time comparison of *HASS* and Qilin with the training size less than 30% of the real problem size.

which show it is less sensitive to the profiling problem size than is the Qilin heuristic.

In summary, for a single-CPU single-GPU environment, *HASS* gives approximately the same performance as Qilin, and it begins to perform better than Qilin if the problem size in the training phase is less than 30% of the input load. Moreover, Qilin in its current form cannot be applied to multiple-GPU environments. Consequently, *HASS* is more scalable in comparison to Qilin.

In our paper [99], a prototype framework had been implemented by the other student worked on this research, that integrates our designed dynamic scheduler for

embarrassingly parallel applications on a single-CPU multi-GPU system. The framework is implemented using C++ and CUDA and sits between the task pool and the workers as an intermediate layer that applies *HASS* to schedule the tasks over the CPU and GPUs.

This framework provides the programmer with pre-implemented building blocks and template functions that need to be filled with application-specific code. In this framework, there is a clear separation between application-specific and application-independent details abstracted by the scheduling framework. Application-specific components are interfaced with the programmer. At the same time, a programmer is liberated from the complexities of scheduling-related details, which are implemented into the framework's core.

## 3.5 Conclusion

In this chapter, we presented a dynamic scheduling heuristic for parallelizing embarrassingly parallel applications on a single-CPU multiple-GPU system. The heuristic employs (1) a learning and adaptation phase to improve load distribution over successive rounds; (2) a partitioner to divide the input workloads into a set of independent sub-loads; (3) a task-bundling technique to minimize data transfer latency between the CPU and GPUs and to prevent the underutilization of GPU cores due to idling and kernel startup latencies; and (4) a work-stealing technique for dynamic load balancing among the CPU and GPU cores. The proposed heuristic is found to perform better or similar to some of the contemporary heuristics for CPU-GPU systems. It is also scalable to any number of integrated GPUs. We embedded our scheduling algorithm into a scheduling framework in which the scheduling complexities are hidden from the user, and the distribution of the load over the available processors are performed automatically.

# Chapter 4

# A Parallel Multilevel Graph Partitioner on the CPU-GPU Architecture

In this chapter, we discuss the design and implementation of a parallel multilevel graph partitioner on a CPU-GPU system. Graph partitioning has extensive applications in the scheduling of task dependent applications as well as different areas of scientific computing such as data mining and VLSI design. Parallel development of an efficient graph partitioner on a CPU-GPU platform faces several challenges including the difficulty in the parallel sub-tasks distribution due to different programming paradigm of the CPU and GPU, irregular nature of partitioning algorithm and unpredictable memory access patterns.

We design efficient parallel partitioning methods to enable multilevel graph partitioning on a single-CPU single-GPU platform. We take advantage of the high parallel processing power of the GPU by executing the computation-intensive parts of our partitioner on the GPU and assigning the parts with less parallelism to the CPU to prevent underutilization of the GPU threads. Our partitioner aims to overcome some of the challenges arising due to the irregular nature of the algorithm, thread

divergence, and memory constraints on GPUs. Furthermore, it ameliorates the utilization of GPU threads through load-balancing schemes. We present a lock-free scheme since fine-grained synchronization among thousands of threads imposes too high a performance overhead.

To the best of our knowledge, this is the first proposed multilevel graph partitioner designed for and implemented on a heterogeneous CPU-GPU system. Experimental results on two different GPUs demonstrate that the partitioner, implemented in CUDA, outperforms serial Metis [47] and parallel MPI-based ParMetis [48]. It performs similar to the shared-memory CPU-based parallel graph partitioner mt-metis [57].

## 4.1 Motivation

Many parallel applications with sparse data structures and data-dependency patterns can be represented by a task interaction graph [55]. This graph may be regularly shaped (e.g., a mesh) or irregular (e.g., a sparse graph). The computational tasks and their communication patterns are shown by a weighted undirected graph in which the vertices represent the tasks and the edges represent the communication cost of the tasks. Scheduling of the task interaction graph on a heterogeneous or homogeneous cluster of processors can be performed based on primary partitioning of the graph. In other words, the graph partitioner is an integral part of the scheduler which partitions the graph into a set of equal weigh partitions in such a way that the total communication cost among the partitions is minimized. Subsequently, the partitions are scheduled over the available processors. Aside from scheduling, graph partitioning has numerous applications in other areas of computing, including social networks, data mining and parallel processing.

The graph partitioning problem is NP-complete. Consequently, many heuristics have been proposed to quickly find a near-optimal solution [47, 70, 80]. The most successful heuristic for partitioning large graphs used in scientific computations is the

multilevel graph partitioning approach which was elaborated in Chapter 2 .

The extensive applications of graph partitioning in different areas of computing have stirred its multilevel parallel implementation for multi-core architectures [57, 58, 88] as well as distributed systems [19, 40, 48, 49, 92]. Although serial graph partitioning and its parallel implementation in the distrusted and multi-core systems have been well studied, designing a parallel graph partitioner on heterogeneous CPU-GPU systems is yet to be investigated.

GPUs have become widely used for accelerating data-parallel applications with regular behavior because of their high computational power, energy efficiency, and low cost. As throughput-oriented devices, GPUs hide the memory access latency through high degrees of multi-threading. This indicates an excellent opportunity to accelerate the graph partitioning task using GPUs.

However, when processing an irregular application like graph partitioning, designing an efficient parallelization strategy becomes challenging. Particularly when dealing with large and irregular real-world graphs, non-uniform and data-dependent graph partitioning computation sub-tasks result in imbalanced load distribution among the threads and consequently deteriorate the performance of the graph-partitioning kernels executed on the GPU. Hence, achieving high performance requires keeping the majority of GPU threads busy, minimizing the communication between the CPU and the GPU, minimizing non-contiguous memory accesses, and preventing thread divergence.

In addition, some GPU applications require graph partitioning to balance the workload among the threads and to increase the parallelism, e.g., Delaunay mesh refinement (DMR) [74] application, requires the graph partitioning to minimize the conflicts among the cavities processed by GPU threads and to increase the parallelism. Using a contemporary partitioning algorithm would oblige the entire graph to be transferred to the CPU, partitioned there, and moved back to the GPU. Designing a high-performance graph partitioner on a heterogeneous CPU-GPU system, can

resolve this problem while maintaining good performance in comparison with CPU-based partitioners.

## 4.2 Design Challenges

We address the following four challenges to design and implement a graph partitioner on a CPU-GPU architecture:

1. The GPU threads communicate through global memory. Hence, to keep the memory consistent, we need to synchronize concurrent writes. Using atomics or locks for synchronization imposes high overheads and degrades performance. This overhead is much more pronounced on GPUs than on multicore systems because a GPU executes tens of thousands of concurrent threads as compared to a multicore CPU, which only executes tens of concurrent threads.

    - Our designed graph partitioner is lock-free and does not degrade performance through fine-grained synchronization among the threads.

2. The irregularity of the graph structure and the serial nature of the partitioning algorithm deteriorate the performance of the graph-partitioning code running on the GPU. This is because of poor locality in the memory accesses and imbalanced load distribution among the threads, which is particularly harmful to performance on GPUs due to their SIMD architecture.

    - We employ memory coalescing and GPU optimization techniques in different phases of the graph partitioning process to distribute the workload evenly among the threads.

3. Memory constraints (GPUs tend to have less memory than CPUs) in storing all the coarsening levels graphs and the transfer latency between the CPU and GPU make the implementation more challenging.

- We apply an efficient graph representation method (CSR) to mitigate the global memory overhead and reuse the allocated memory space as much as possible. We also minimize the data transfers between the CPU and the GPU.

4. Heterogeneous systems combine a SIMD and a MIMD architectural model. Thus, the partitioner needs to be decomposed properly to fully exploit the CPU and the GPU architectures at the same time.

    - Our partitioner executes the parallel computation-intensive parts on the GPU and assigns the parts with lower levels of parallelism to the CPU to prevent the underutilization of GPU cores.

## 4.3 A Multilevel Graph Partitioner for CPU-GPU Architectures

In this section, we discuss the design and CUDA implementation of our parallel multilevel graph partitioner for heterogeneous CPU-GPU environments [29]. We exploit the massive parallel processing power of the GPU cores by executing the computation-intensive sub-tasks of our partitioner on the GPU and assigning the subtasks with less parallelism to the CPU.

We start the coarsening on GPU and employ Heavy Edge Matching (HEM) technique to match the GPU threads assigned vertices. Then the matched vertices are collapsed together to construct the coarser graph. High-computational levels of coarsening are performed on the GPU and when the graph size is lower than a threshold value, the coarsening levels with less computation are transfered to the CPU. We avoid the fine-grained synchronization among the threads by developing lock-free matching and contraction methods on the GPU.

The initial partitioning has a small design space for parallelization. Therefor this phase is performed on the CPU. The un-coarsening iterative process initiates on the

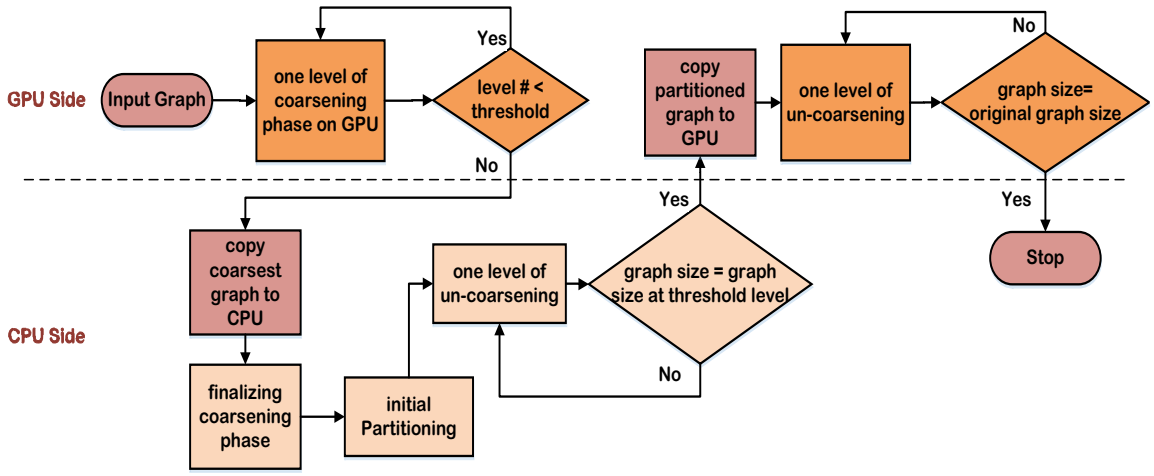



Figure 4.1: Proposed heterogeneous graph partitioning scheme.

CPU and the graph is projected back and refined during multiple levels till it reaches to the threshold level again. Subsequently the partitioned graph is transfered to the GPU and the remaining levels of un-coarsening are completed on GPU.

Figure 4.1 shows the proposed graph partitioning scheme for CPU-GPU architectures. The coarsening phase starts on GPU by distributing the graph vertices among the threads. Each thread scans its assigned vertices and finds their potential matched vertices using HEM. However, due to the possibility of the matching conflicts resulting from parallel execution of GPU threads, another GPU kernel is launched to resolve them before the contraction begins. The coarsening continues level-by-level until reaching the threshold beyond which coarsening is faster on the CPU due to the lack of sufficient parallel tasks. Thus, at the threshold level, the coarse graph is transferred to the CPU and the remaining iterations of the coarsening phase are performed on the CPU. Since the coarse graph's size is by definition small, the initial partitioning of the graph has a low level of parallelism. Hence, the initial partitioning phase is also completed on the CPU along with the initial refinement steps until the threshold level is again reached. At this point, the partitioned graph is transferred back to the GPU. The remaining iterations of the un-coarsening phase are executed on the GPU.

### 4.3.1 Data Structures for Graph Representation

In our graph partitioner, we use the Compressed Sparse Row (CSR) representation, which is a well-known compact format for representing large and sparse graphs appropriate for graph storage inside the limited GPU memory. We store the initial, intermediate, and final graphs all in the CSR form in order to minimize the memory footprint and maximize the size of the graph that the GPU DRAM can hold. CSR is preferred over the adjacency matrix representation method, since it reduces the memory occupancy significantly.

The adjacency matrix representation takes $O(|V|^2)$ memory space and wastes a big fraction of the memory specially for sparse graphs (most real-world graphs are sparse; thus $|E| << |V|^2$). CSR packs all the adjacency lists of each vertex contiguously. This format not only optimizes the memory usage but also helps coalescing the accesses in the neighbor lists of vertices on the GPU.

Figure 4.2 shows the graph data structure in one level of coarsening in CSR format which consists of 4 arrays: an *adjacency* array (*adjcny*) of length $2 \times |E|$ [1], which stores the adjacency list of the graph vertices, and an *adjacency pointer* array (*adjp*) of length $|V| + 1$, which points to the adjacency set of each vertex in the *adjacency array*. In addition, the adjacency weight *(adjwgt)* of length $2 \times |E|$ and the vertex weight *(vwgt)* of length $|V|$ contain the weights of the edges and vertices, respectively.

In all levels of coarsening, we augment the CSR data structures with two other arrays allocated in global memory: a *matching* array $M$ of length $|V|$ that includes the matched pairs to be collapsed in the coarser graph, and a *mapping* array (*Cmap*) of length $|V|$ that stores the vertex labels in the coarser graph. We construct and store all these 6 arrays permanently at each level of coarsening. This enable us to project back and retrieve the information of the finer graphs during un-coarsening levels.

[1] Since the graph is undirected and vertices at both end of an edge must easily be able to access their set of edges, an edge appears at two locations in the adjacency array.

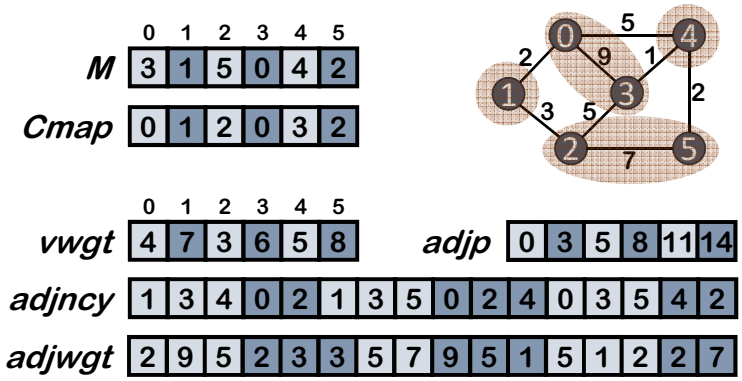



Figure 4.2: Graph data structures used in our design shown for an example. CSR format of the graph is accompanied with $M$ and $Cmap$ auxiliary arrays. These arrays are constructed for every intermediate partitioning level.

The different phases of the graph partitioning are discussed in detail in the following subsections.

### 4.3.2 Coarsening.

During the coarsening phase, the vertices of the graph are collapsed to construct the next coarser level of the graph using a lock-free approach. The coarsening phase consists of two steps: matching and contraction. As we mentioned before, at each level of coarsening, two arrays are allocated in the global memory on the GPU: a *matching* array ($M$) of length $|V|$ that includes the matched pairs to be collapsed in the finer graph ($G_i$) and a *mapping* array ($Cmap$) of length $|V|$ that stores the vertices' labels in the coarser graph ($G_{i+1}$).

***Matching.*** At the beginning of the matching step, the graph vertices are divided among the threads on the GPU. We are mindful of memory coalescing when distributing the vertices to the threads to improve the memory accesses efficiency. A coalesced memory access is the combination of multiple memory accesses into a single transaction. In modern CUDA-capable GPUs, sets of 32 contiguous threads constitute a *warp*. When all threads in a warp execute a load instruction, the hardware checks which memory locations the threads access. If all the threads in a *warp* access locations within a naturally-aligned 128-byte block in the global memory, the

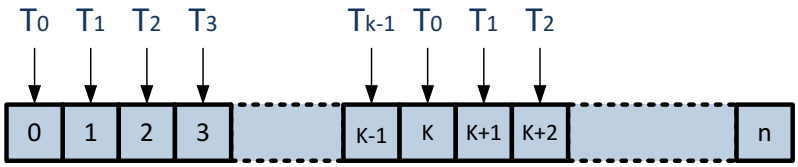



Figure 4.3: Memory coalescing.

hardware coalesces the accesses into one transaction. Otherwise, multiple memory transactions have to be issued, resulting in reduced throughput.

Figure 4.3 illustrates the memory coalescing technique. If thread 0 accesses vertex $n$, thread 1 accesses vertex $n+1$ and Thread $t_{k-1}$(the last thread) accesses vertex $n+(k-1)$, then all memory requests issued by a warp fall into the same 128-byte block. Therefore, they are coalesced, which improves the memory bandwidth significantly.

To maximize the parallelism in the matching step, we use a lock-free approach since fine-grained synchronization incurs too much overhead due to the high number of threads running on a GPU. At the beginning of the matching step, each thread goes over its assigned vertices in the graph and finds their matches using the heaviest edge matching technique ($HEM$), i.e., it searches the neighbor connected to its assigned vertex with the maximum weight edge. As a result HEM reduces the sum of weights of the edges in the coarser graph. If we assume that $E_M$ is the set of edges which are removed from graph in level $i$ of coarsening by applying a matching $M$. Then the weight of edges in the coarser graph is $W_{E_{i+1}} = W_{E_i} - W_{E_M}$. So if we increase the $W_{E_M}$ by collapsing the heavier edges, the weight of the edges in the coarser graph is reduced, which consequently decreases the edge cut by a great amount

If all the edges have the same weight, a random matching ($RM$) [39] method is used in an iterative fashion where, for each vertex, one of its unmatched neighboring vertices is chosen randomly to be collapsed with it in the coarser graph.

All the threads write to the shared matching array ($M$) in a lock-free fashion. Hence, there is a possibility that vertex $a$ assumes it has been matched with vertex $b$ while vertex $b$ finds vertex $c$ as its match. Therefore, we need to launch another kernel to resolve these conflicts. In this kernel, each thread goes over its assigned

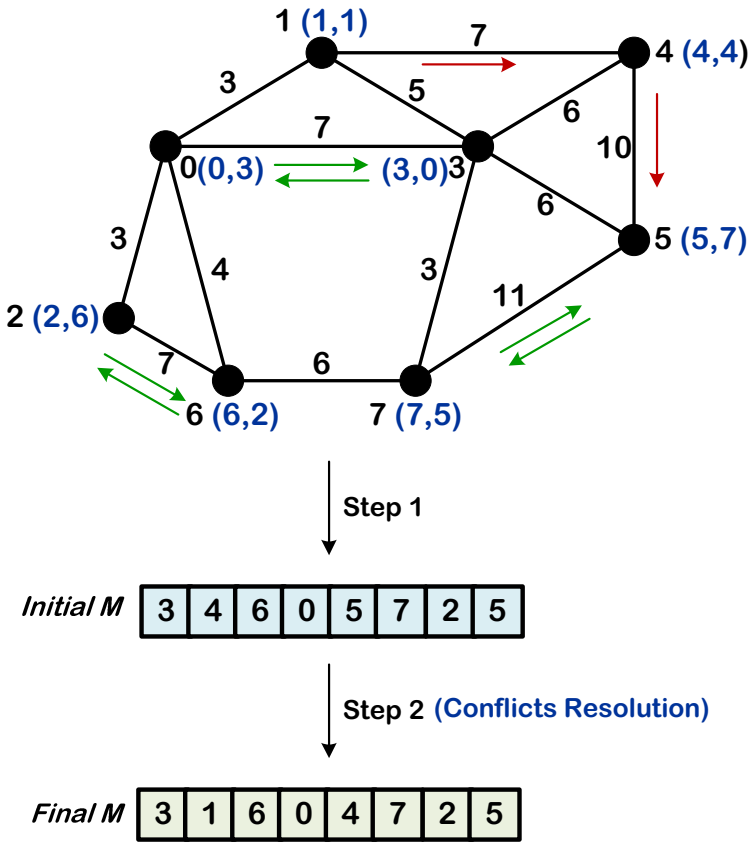



Figure 4.4: Matching array creation.

vertices again and checks the match values; if it finds any cases in which $M(a) = b$, but $M(b)! = a$, it matches vertex $a$ to itself, i.e., $(M(a) = a)$. This means that vertex $a$ has another chance to find a match in the following coarsening levels.

Figure 4.4 illustrates the steps needed to create the final matching array for a graph with eight vertices. In this example, vertices 1 and 4 are matched to themselves because $M(1) = 4$ but $M(4) = 5$. However, $M(5) = 7$ and $M(7) = 5$. Consequently, vertices 5 and 7 are matched to each other. Although such conflicts impose some overhead in the matching phase due to a decrease in the number of matched vertices and consequently an increase in the required number of matching iterations, the overhead is significantly lower than using fine-grained locks to resolve conflicts.

***Cmap Creation.*** The *matching* array $(M)$ facilitates the creation of the mapping array *Cmap*, which contains the collapsed vertices' labels in the coarser graph.

To fully parallelize the Cmap creation and to minimize the memory usage, we use a parallel prefix sum method and execute all the required computations in-place.

*Cmap* is constructed by launching the following four kernels on the GPU:

1. Creating the initial *Cmap*: Initially, the *Cmap* array of length $|V|$ is allocated in the GPU's global memory. Then, a kernel is launched in which each thread executes the following function for all the vertices assigned to it.

   ***foreach*** *(vertex $v_i$: assigned vertices to thread $T_k$)* ***do***

   ***if*** $(v_i <= M[v_i])$ ***then***

   $Cmap[v_i] = 1$

   ***else***

   $Cmap[v_i] = 0$

   In this kernel, the *Cmap* entries are initialized to zero or one depending on the vertices' labels of the matched vertices in the finer graph $G_i$.

2. Creating a *helper* array: An inclusive prefix sum is computed on the *Cmap* array to create the *helper* array *PV*. To maximize the performance, we use the parallel inclusive-scan from the *CUB* library [69], which currently is the highest-performing parallel implementation of prefix sums on GPUs. The last element in the $PV$ array indicates the number of vertices in the coarser graph, *Cgraph*.

3. Subtraction: All the threads subtract one from every entry of the $PV$ array resulting in the $SV$ array.

4. Creating the final *Cmap*: The final *Cmap* array is created through the following kernel:

   ***foreach*** *(vertex $v_i$: assigned vertices to the thread $T_k$)* ***do***

   ***if*** $(v_i > M[v_i])$ ***then***

   $Cmap[v_i] = SV[M[v_i]]$

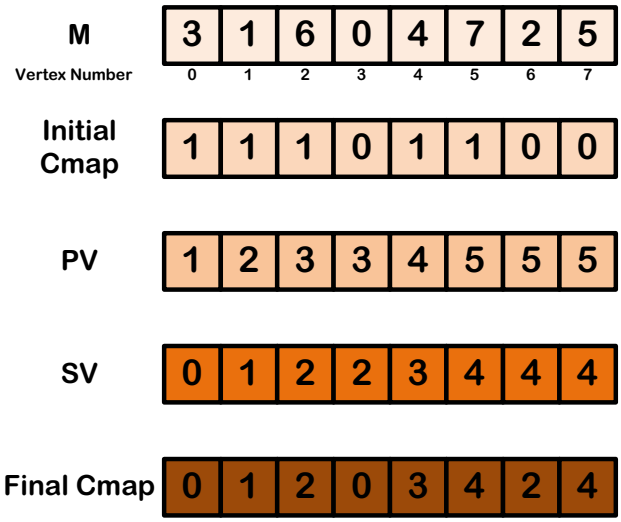



Figure 4.5: Cmap creation steps.

It should be noted that the steps needed to create the final *Cmap* array are computed in-place and we do not need any auxiliary memory space. In addition, all steps are fully parallelized. Figure 4.5 illustrates the four steps for creating the final *Cmap* array. In this example, the number of vertices in *Cgraph* is 5.

***Contraction.*** In the contraction step, with the help of the *Cmap* and *Matching* arrays, the matched vertices are collapsed to form the coarser graph. This step is more complex since distributing the contraction sub-task over the GPU threads is not straightforward; in the ideal parallelization, each thread calculates a part of the adjacency array *cadjncy* and the adjacency weight array *cadjwgt* of the coarser graph. However different sized adjacency lists of the coarser graph vertices should be created in a parallel fashion and the exact size of each segment containing the neighbors of collapsed vertices in the *cadjncy* and the *cadjwgt* array is unknown at the beginning of the contraction step.

**Algorithm 4.1** Contraction Method

1: ***kernel 1***
2: /* $V_{uw}$ is the vertex label of two collapsing vertices ***u*** and ***w*** in the *cgraph* */
3: **foreach** (vertex $V_{uw}$: assigned to thread $T_k$) in **parallel do**
4: **if** $u = w$ **then**
5: $Temp[V_{uw}] \leftarrow |adjncy(u)|$
6: **else**
7: $Temp[V_{uw}] \leftarrow |adjncy(v)| + |adjncy(w)|$
8: ***kernel 2***
9: exclusive parallel prefix sum on *Temp* to compute the initial start index of neighbor lists of each vertex in *tcadjncy* and *tcadjwgt*
10: ***kernel 3***
11: allocate two hash tables ***tb1*** and ***tb2*** for $T_k$
12: **foreach** (vertex $V_{uw}$: assigned to thread $Ti$) in **parallel do**
13: **foreach** vertex $x \in adjncy(u)$ **do**
14: $m = f(Cmap[x])$
15: search for $Cmap[x]$ in bucket $m$ of ***tb1***
16: **if** $Cmap[x]$ exists **then**
17: Add the *edge-weight (x,u)* to the edge-weight of $m$ in ***tb2***
18: **else**
19: Add $Cmap[x]$ to ***tb1***, Add the $(x, u)$ corresponding edge weight to ***tb2***
20: **Endfor**
21: **foreach** vertex $x \in adjncy(w)$ **do**
22: $m' = f(Cmap[x])$
23: search for $Cmap[x]$ in bucket $m'$ of ***tb1***
24: **if** $Cmap[x]$ exists **then**
25: Add the *edge-weight (x,w)* to the edge-weight of $m$ in ***tb2***
26: **else**
27: Add $Cmap[x]$ to ***tb1***, Add the $(x, u)$ corresponding edge weight to ***tb2***
28: **Endfor**
29: copy the values of ***tb1*** and ***tb2*** to *tcadjncy* and *tcadjwgt*
30: $Temp'[V_{uw}]$ = total number of vertices in the ***tb1***
31: **Endfor**

32: ***kernel 4***
33: exclusive prefix sum on $Temp'$
34: ***kernel 5***
35: **foreach** (vertex $V_{uw}$: assigned to thread $T_k$) in **parallel do**
36: copy the elements from *tcacadjncy* and *tcadjwgt* to *cadjncy* and *cadjwgt* according to the indices in *Temp* and *Temp'*

To parallelize the contraction procure, first, we estimate the initial size of the neighbor list for each vertex $u$ matched with vertex $w$. The number of entries in the coarser graph's adjacency array for the pair is the accumulated number of neighbors in the adjacency set of $u$ and $w$ (the maximum possible). However some of the vertices in the collapsed vertices neighbor lists may also be matched together in the coarser graph. Also there is a possibility that some of the vertices are the neighbors of both matched vertices. But they should appear just once in the merged neighbor list of new label vertex in the coarser graph. Therefore the final size of neighbor list in the coarser graph shrinks. This requires additional work to calculate the final size of the neighbor list of each vertex in the coarser graph.

To implement the contractions step on GPU, first the vertices of the coarser graph are distributed among the threads. Then, each thread finds the start and end indices of its assigned vertices neighbor list in the *cadjncy* and *cadjwgt* arrays. These indices need to be calculated beforehand and passed to the threads. To accomplish this, an auxiliary arrays (*Temp*) is allocated on the GPU, of length equal to the number of vertices in the coarser graph to hold the estimated start indices of different neighbor lists in the coarser graph adjacency array.

Algorithm 4.1 shows the designed parallel kernels for the contraction step:

***kernel 1.*** In this kernel, each thread calculates the maximum number of entries that it needs for neighbor lists of any of its assigned vertices in the cadjncy and cadjwgt arrays by scanning all vertices assigned to it. For each vertex $u$, which is matched with vertex $w$, the maximum number of entries in the coarser graph's adjacency array is the sum of the number of vertices in the adjacency set of $u$ and $w$. In other words, it is equal to the total number of neighbors of $u$ and $w$. If the vertex $u$ is matched

to itself, then the required number of entries will be equal to the size of its adjacency set. Each thread applies the same logic to each of its assigned vertices and inserts the final number of required entries in the corresponding entries of the Temp array.

***kernel 2.*** Next, an exclusive parallel prefix sum (using *CUB* library [69]) is calculated on *Temp* to compute the initial start index of neighbor lists of each vertex in the coarser graph's adjacency list and adjacency weight arrays.

***kernel 3.*** As mentioned before, the number of neighbors of two collapsed vertices is usually less than the initial calculated value, which can be due to either matched pairs having a neighbor in common or two distinct neighbors of the merged set being matched as well. This necessitates preventing the occurrence of the duplicate vertex labels in the adjacency list of the coarser graph.

To merge the neighbor lists of matched vertices, a hash table can be allocated for each thread. The length of this table is equal to the number of vertices in the coarser graph. Then, in the case of a collision, a linear scan is performed by each thread on the edge list to prevent the vertex duplication in the adjacency list of its assigned vertices in the coarser graph. However, this method poses heavy GPU memory overhead since the size of hash table allocated for each thread will be equal to number of vertices in the coarser graph. Therefore, we use a *compact-hash* table for each thread and a hash function is applied to the Cmap values of all the vertex numbers in the neighbor list of each pair of collapsing vertices. The hash function maps the neighbors to the entries in the hash table and constructs the adjacency list of the newly created vertex in the coarser graph. *Hash-compact* table reduces the memory space required for the hash tables, which ideally should be equal to the number of vertices in the coarser graph.

Since the graph edges are weighted, we require two *compact-hash* tables to store the adjacency list labels and the corresponding edges. According to *kernel 3*, in the compact-hash approach , two hash tables ($tb1$ and $tb2$) are allocated for each thread. One of the hash tables is used to create the neighbor list of the vertices in the coarser graph and the other one ($tb2$) stores the corresponding edge weights.

For each vertex $V_{uw}$ assigned to a thread, we apply a hash function "$f$" to the *Cmap* values (new label numbers of collapsed vertices) of all neighbors of collapsing vertices $u$ and $w$, which are represented by $V_{uw}$ in the coarser graph. This function maps the neighbors to the buckets of $tb1$ and $tb2$, aimed to construct the final neighbors list and its corresponding edge-weights for $V_{uw}$ in the coarser graph. Since the length of hash tables are much less than the number of vertices in the coarser graph, to avoid collisions, chaining [31] is used where each bucket of the hash tables stores multiple elements. When the Cmap value of a neighboring vertex is mapped to one of the buckets in $tb1$, we search this value in the bucket. If this value is not found, it will be added to the tb1 and the corresponding edge weight is also added to $tb2$. But if this value already exists it shows that two neighbors have been paired together. In this case, just their edge weights incident on the collapsed vertices are added together in $tb2$.

At the end of outer loop of *kernel 3* (line 29 of algorithm 4.1 ), each thread copies the values from the hash tables of the current vertex to the *tcadjncy* and *tcadjwgt* arrays allocated on GPU, respectively. These intermediate arrays facilitate the parallel execution of the threads working on different sets of vertices by determining the final size of coarser graph arrays (*cadjncy* and *cadjwgt*).

Finally each thread counts the total number of entries in the hash tables, which gives the precise size of the neighbor list of its assigned vertex in the coarser graph. This value is saved in $Temp'[V_{uw}]$ which is an auxiliary array allocated on the GPU, of length equal to the number of vertices in the coarser graph.

***kernel 4.*** This kernel performs another parallel exclusive prefix sum on $Temp'$ to calculate the start index of the neighbor list of each vertex in the final *cadjncy* and *cadjwgt* arrays.

***kernel 5.*** When *kernel 5* is launched, each thread copies the calculated elements from the *tcadjncy* and *tcadjwgt* arrays to the *cadjncy* and *cadjwgt* arrays with the help of the indices in $Temp$ and $Temp'$. Therefore the next coarse graph is constructed.

Figure 4.6 demonstrates the procedure of contraction and merging the adjacency

lists for two matched vertices $(0, 3)$) of the graph shown in Figure 4.4. According to Figure 4.6, two matched vertices 0 and 3 are collapsed to construct the new vertex 0 in the coarser graph. Vertex 0 is connected to the 4 neighbor vertices while vertex 3 has five neighbors. The hash function used in this example maps the *Cmap* values of all the neighbor lists of two matched vertices to the buckets of *compact-hash* table of size 100. Finally the adjacency list of vertex 0 is copied to the corresponding locations of cadjncy in the coarser graph.

The new neighbor list has a size of 5; Since vertices 0 and 3 are contracted, they are removed from the merged neighbor list. Furthermore, vertex 1 is the neighbor of both vertices 0 and 3. Finally some of their neighbors (e.g., 2 and 6) are also matched together in the coarser graph. Consequently, the final adjacency list of two matched vertices in the coarser graph has a smaller size than the total number of vertices in the neighborhood of each of the two vertices.

The matching and contraction process are repeated through multiple coarsening levels until the number of graph vertices is below a threshold. Then the graph is transfered to the CPU and the remaining levels of coarsening are executed on the CPU by employing the multi-core partitioner mt-metis [57].

### 4.3.3 Initial Partitioning

Initial partitioning is always performed on a small problem size since it is applied to the coarsest graph. Although the initial partitioning does not need a high level of parallelism, to maximize the performance we use mt-metis to parallelize this phase on a multi-core CPU. Mt-metis has been shown to be faster than other parallel partitioners [57]. Mt-metis employs a parallel k-sectioning approach for the initial partitioning, where each thread independently generates an initial partitioning (with $k$ partitions) of the coarsest graph using recursive bisectioning. Then the best partition with minimum edge cut is selected as the final input to the un-coarsening phase.

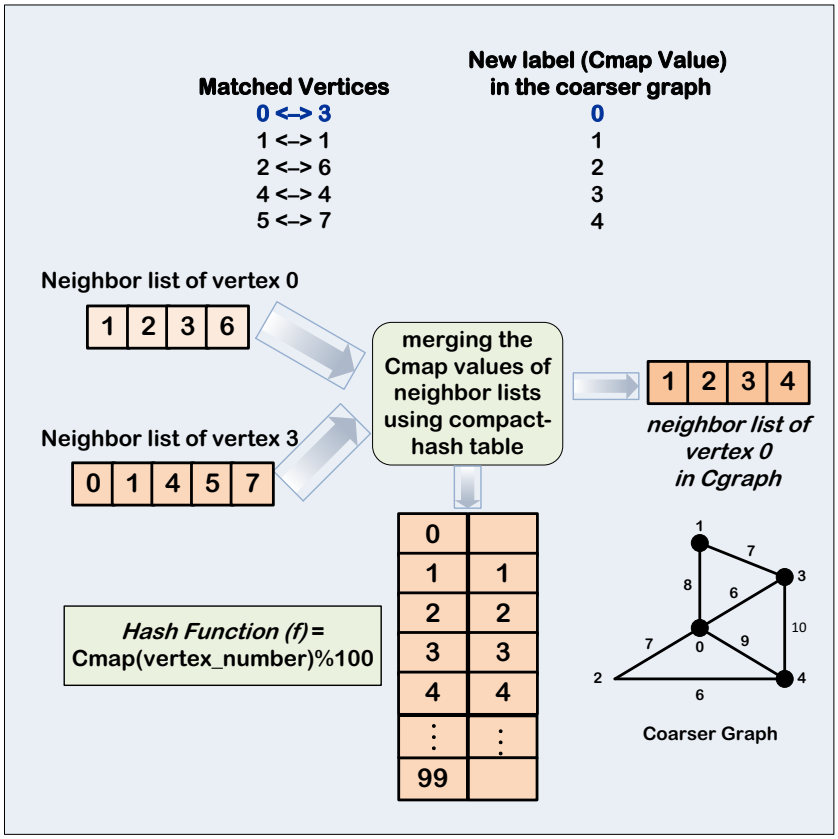



Figure 4.6: Contraction procedure.

### 4.3.4 Un-coarsening

The un-coarsening phase starts on the CPU by employing the mt-metis un-coarsening method and continues until the threshold level is again reached. At this point, the partitioned graph is transferred back to the GPU and the remaining steps of the un-coarsening phase are executed on the GPU.

Un-coarsening comprises two steps: Projection and refinement. During the projection step, the coarser graph at level *i+1* is projected back to the finer graph at level *i*. This step can easily be parallelized on the GPU by dividing the vertices of the finer graph among the threads and having each thread specify the partition labels of the projected vertices in the finer graph by considering the Cmap array and saved pointer arrays from the coarsening phase.

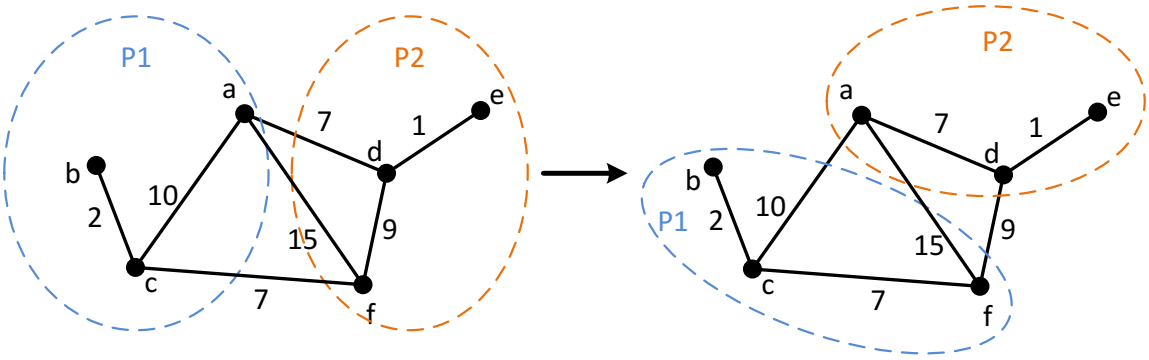



Figure 4.7: Edge cut increment by concurrent movement of boundary vertices.

The refinement step attempts to improve the graph edge cut by moving the boundary vertices between partitions. This step is more challenging because the concurrent movement of vertices among the partitions may increase the edge cut.

Figure 4.7 shows an example, where concurrent movement of vertices results in increasing the edge cut, while each individual move decreases the edge cut. Consider two vertices $a$ and $f$ belonging to partition 1 ($P1$) and partition 2 ($P2$) respectively. In a situation like this, moving vertex $a$ from $P1$ to $P2$ reduces the edge cut by 12. Similarly, moving vertex $f$ from $P1$ to $P2$ reduces the edge cut by 13. However if these two vertices were assigned to different threads, and each thread performed the move, then the overall edge cut will increase by 5.

Concurrent movement of vertices among the partitions may also violate the balance constraints and some of the partition weights become smaller than the minimum allowable partition weight, or larger than the maximum allowable partition weight.

To avoid such refinement problems, the boundary vertex movements should be ordered using locks. However with thousands of threads working concurrently on the GPU, applying lock-based methods for moving vertices between partitions would impose a high synchronization cost and degrade the performance. To overcome this challenge, we use a coarse-grain approach for refinement which consists of two kernels.

***Kernel 1.*** In the first refinement kernel, the vertices in the finer graph are distributed among the threads and each thread determines the boundary vertices among

its assigned vertices. Then it finds the best destination partition for migration of each boundary vertex, if possible. A destination partition is selected for moving a vertex if this move results in the maximum reduction of the edge cut and does not underweight the source or overweight the destination partition. In addition, an ordering method [48] is used that divides each refinement step into two iterations. During each step, vertices can move between the partitions only in one direction (decreasing or increasing). This prevents concurrent exchanges of two vertices between two neighbor partitions, which may result in increasing the edge cut.

To process the threads' concurrent requests for migrating vertices, we allocate a buffer to each graph partition where the threads insert their movement requests. A request contains the source partition's vertex labels and potential gain. Each buffer has a counter $S$ that indicates the current size of the buffer. To prevent race conditions among the threads, when one thread wants to put a request on a specific buffer, it atomically increments the counter $S$ by one. Thus, multiple threads are able to write to exclusive slots of the buffer concurrently without resorting to locks.

Figure 4.8 visualizes the parallel insertion of the boundary vertex movement requests during the refinement. If we assume the number of graph partitions is $k$, then, $k$ buffers are allocated during the refinement. As Figure 4.8 shows, vertex 3 is a boundary vertex with positive movement gain assigned to $T_0$. This vertex is currently located in partition 0 ($p0$). However if this vertex moves to partition 1, the graph edge cut is reduced by 5. Therefor, $T_0$ inserts this request including all the required information to the buffer allocated for partition 1. All the threads also insert their requests to the corresponding buffers in a parallel fashion.

***Kernel 2.*** Next an exploitive kernel is launched with a number of threads equal to the number of partitions where each thread processes the incoming requests in the buffer of its assigned partition. It initially sorts the relocation requests based on their gain. Then it accepts the moves that do not overweigh the partition's (say $p_i$) weight, e.g., for each vertex $j$, $weight(p_i) + weight(v_i)$ will be less than the maximum allowed weight for this partition. The refinement terminates when all boundary vertices have

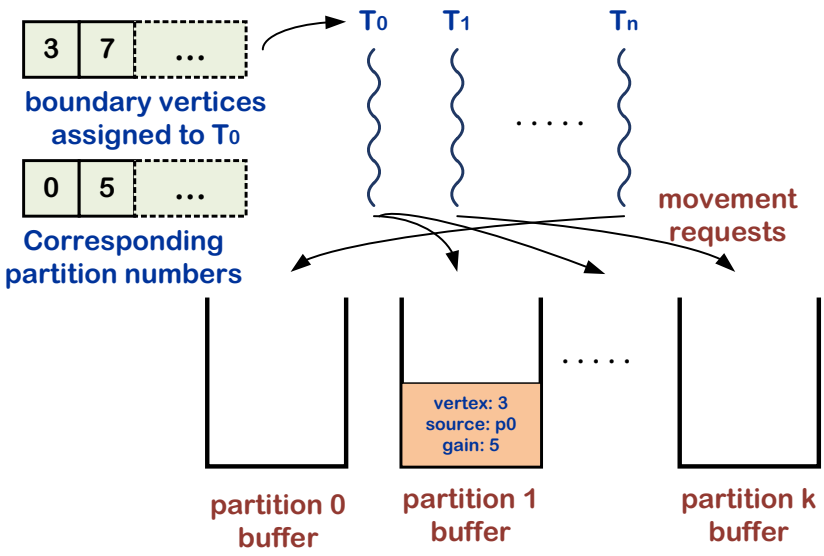



Figure 4.8: Boundary vertex movement requests insertion procedure.

been explored.

The refinement at each level repeats for a specified number of passes to improve the edge cut while keeping the partitions balanced. However, it can be terminated earlier if no move is committed in the current pass. Although the concurrent updates of a partition may unbalance it, the balance of partitions is guaranteed by continuing the refinement at the finer graph levels.

## 4.4 Comparison with mt-metis

It should be noted that CPU-GPU partitioner requires a higher amount of redundant information in the coarsening and refinement phases than mt-metis, i.e., in some of the launched kernels we need to record more information. The reason is that mt-metis employs a persistent thread paradigm, where data ownership is given to the threads at the beginning of the program and stays the same until the end of the execution. In contrast, the data ownership in CPU-GPU partitioner is not persistent throughout the execution. In particular, the kernels are launched with a variable number of threads. The rationale behind this is to balance the load among the threads as much

as possible and to maximize the performance.

## 4.5 Experimental Evaluation

In this section, we evaluate the performance of our CPU-GPU partitoner implemented using CUDA, on a CPU-GPU architecture. We compare the performance of this partitioner with *Metis* 5.1.0 (a serial partitioner), *Par-Metis* 4.0.3 (a parallel distributed-memory partitioner) and *mt-metis* 0.4.4 (a parallel shared-memory partitioner).

We ran all implementations on two different systems. The first one is equipped with an Intel Xeon E5540 processor with 8 cores and one Nvidia GeForce GTX Titan GPU and the other is equipped with the same CPU and one Nvidia Kepler K40 GPU. Experiments were performed using eight different graphs arising in various areas of computation, which were obtained from DIMACS10 [6]. Table 4.1 shows the size and specification of these graphs.

| Graph | $\|V\|$ | $\|E\|$ | Description |
|---|---|---|---|
| delaunay | 1,048,576 | 3,145,686 | Delaunay triangulation of random points |
| er-fact | 1,048,576 | 10,904,496 | Erdös-Rényi Graphs |
| co-papers | 540,486 | 15,245,729 | Co-author and Citation Networks |
| idoor | 952,204 | 22,785,143 | Sparse matrix from UFC |
| af-shell | 1,508,065 | 25,582,130 | Sparse matrix from UFC |
| USA Roads | 23,947,347 | 28,947,347 | Road network |
| Hugebubble | 21,198,119 | 31,790,179 | 2D dynamic simulation |
| nlpkkt120 | 3,542,400 | 46,651,696 | Sparse matrix from UFC |

Table 4.1: Input graphs used for the experiments.

For all implementations, we partitioned the input graph into 64 partitions and the imbalance tolerance for each partition was set to 3% (as in *Metis* [47]). The graphs edge weights added as equal numbers with value of 1 at the beginning of the partitioning process.

Figure 4.9 shows the speedup achieved by our CPU-GPU graph partitioner using a

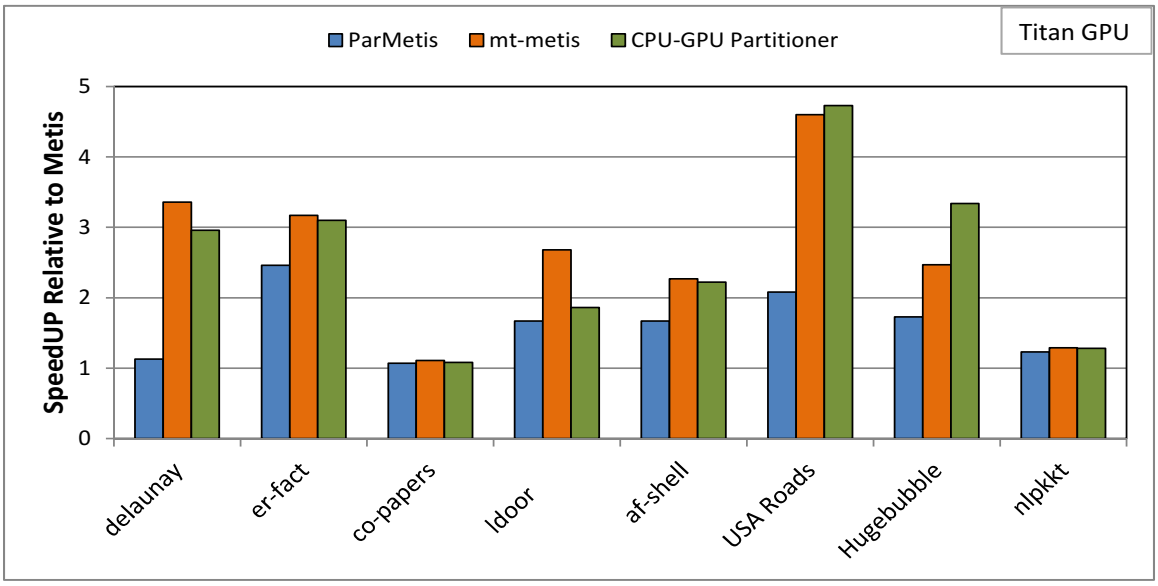



Figure 4.9: Speedup of ParMetis, mt-metis, and CPU-GPU partitioner over Metis (Titan GPU) .

Titan GPU, ParMetis and mt-metis (with 8 threads) over the serial Metis. Figure 4.10 demonstrates the comparison when we use a K40 GPU. The speedup is the runtime of the parallel graph partitioners relative to the runtime of serial Metis. In each case, we use the minimum runtime of three experiments to compute the speedup.

As Figures 4.9 and 4.10 illustrate, our CPU-GPU graph partitioner outperforms Metis and ParMetis on all tested inputs, and its performance is also quite reasonable in comparison to mt-metis (i.e., somewhat better on the larger graphs and somewhat worse on the smaller graphs). On average the CPU-GPU graph partitioner using the Titan GPU performs 2.57× and 1.52× faster than Metis and ParMetis respectively. When using K40 GPU, our partitioner performs 2.63× faster than Metis and 1.57× faster than ParMetis. The CPU-GPU partitioner achieves better speed up when we use K40 GPU du to higher parallel processing power of this GPU. The irregularity of the input graph affects the performance of CPU-GPU partitioner, since it increases the workload imbalance between the GPU threads on some of the GPU kernels, which hurts performance.

Table 4.2 show the absolute runtimes of the four parallel graph partitioners. For

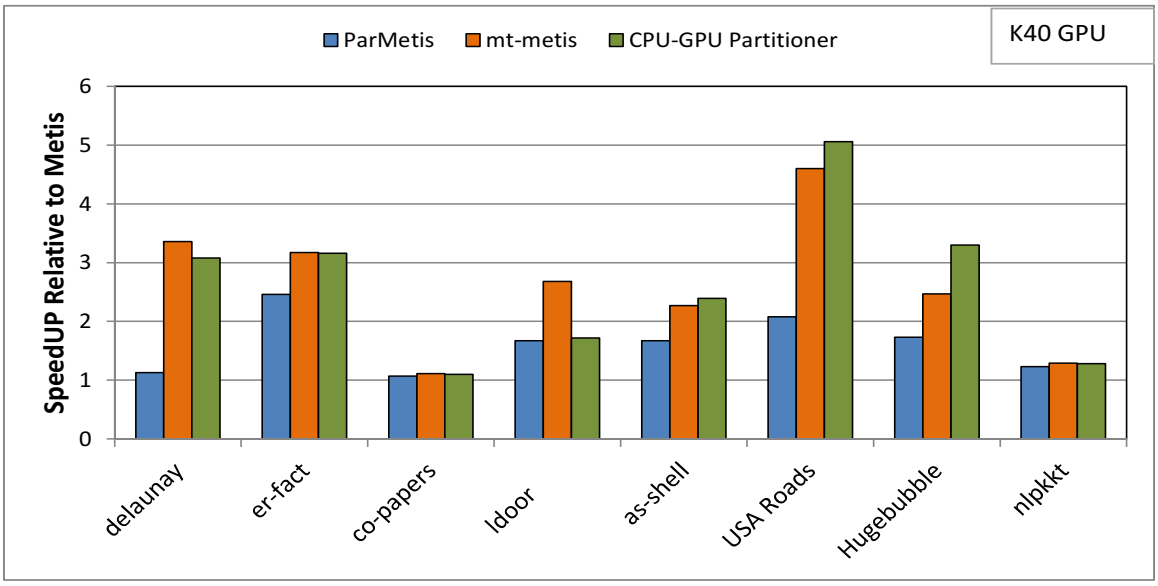



Figure 4.10: Speedup of ParMetis, mt-metis, and CPU-GPU partitioner over Metis (K40 GPU).

CPU-GPU partitioner, this time includes the time to transfer the graph between the CPU and the GPU. However, I/O times on the CPU are excluded from all timing measurements.

To validate the comparison of our partitioner with the other parallel partitioners, we also evaluate the ratio of the edge cut achieved by each parallel partitioner relative to Metis. Table shows the edge cut scaled relative to Metis for the three partitioners.

The results show that our CPU-GPU partitioner is able to produce partitions of

| Graph | ParMetis | mt-metis | CPU-GPU partitioner (Titan) | CPU-GPU partitioner (K40) |
|---|---|---|---|---|
| delaunay | 0.65 | 0.22 | 0.25 | 0.24 |
| er-fact | 28.35 | 22.05 | 22.55 | 22.12 |
| co-papers | 2.75 | 2.65 | 2.73 | 2.68 |
| idoor | 0.77 | 0.48 | 0.69 | 0.75 |
| af-shell | 2.05 | 1.51 | 1.54 | 1.43 |
| USA Roads | 11.24 | 5.09 | 4.95 | 4.63 |
| Hugebubble | 8.40 | 5.89 | 4.36 | 4.36 |
| nlpkkt120 | 8.00 | 7.63 | 7.69 | 7.67 |

Table 4.2: ParMetis, mt-metis, and CPU-GPU partitioner runtimes (in seconds).

| Graph | ParMetis | mt-metis | CPU-GPU partitioner (Titan) | CPU-GPU partitioner (K40) |
|---|---|---|---|---|
| delaunay | 1.033 | 1.027 | 1.059 | 1.045 |
| er-fact | 1.064 | 1.02 | 1.053 | 1.023 |
| co-papers | 1.181 | 1.14 | 1.157 | 1.125 |
| idoor | 1.059 | 1.057 | 1.068 | 1.053 |
| af-shell | 1.078 | 1.062 | 1.067 | 1.066 |
| USA Roads | 1.177 | 1.121 | 1.120 | 1.09 |
| Hugebubble | 1.134 | 1.103 | 1.108 | 1.105 |
| nlpkkt120 | 1.028 | 1.018 | 1.018 | 1.016 |

Table 4.3: ParMetis, mt-metis, and CPU-GPU partitioner Edge cut ratios in comparison to Metis.

comparable quality to mt-metis and ParMetis and the partitioning quality does not diverge from that of ParMetis and mt-metis. The quality degradation for some of the graphs is due to the finer-grain implementation of CPU-GPU partitioner. In the coarsening and un-coarsening phases of CPU-GPU partitioner, thousands of threads are working concurrently, making the conflict rate much higher in comparison to mt-metis, which only runs a few threads (8 threads in our experiments).

We also compare the number of coarsening levels required on GPU for the CPU-GPU partitioner, with the number of coarsening levels for mt-metis. Figure 4.11 demonstrates the comparison results over different graphs. As the figure shows, the number of coarsening iterations are higher for the CPU-GPU partitioner than that of the met-metis. This is due to higher rate of conflicts among the GPU threads in the matching step. The number of coarsening levels also increases by increasing the irregularity of the graphs since the graph size reduces with lower coarsening ratio. Although the number of coarsening levels on our CPU-GPU partitioner is higher, fast parallel processing of GPU threads in each coarsening level results in achieving the comparable results with mt-metis.

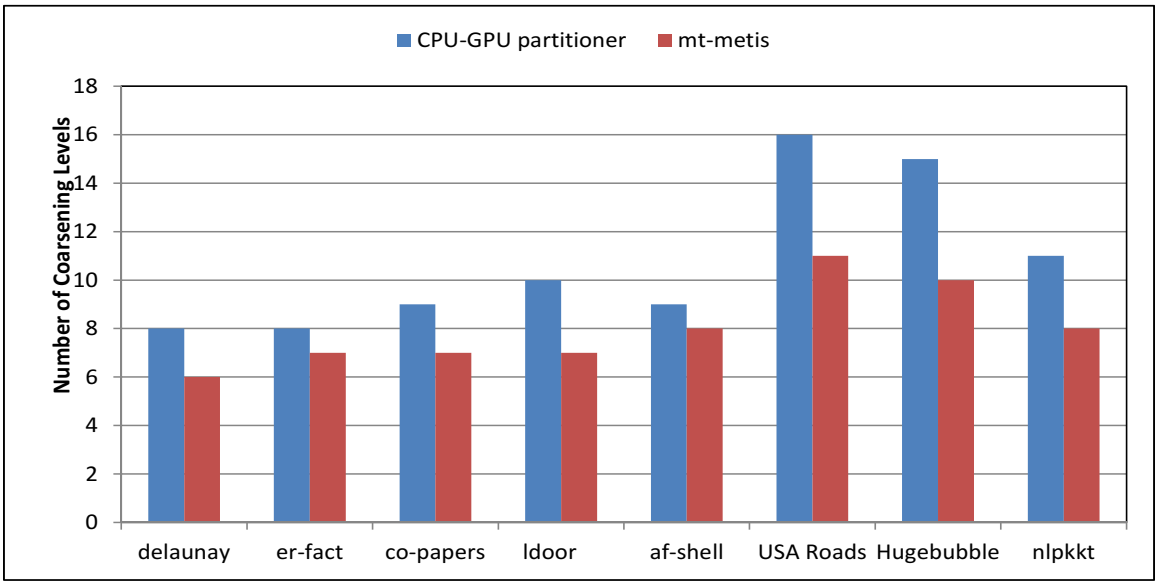



Figure 4.11: Comparison of coarsening levels for mt-metis, and CPU-GPU partitioner.

## 4.6 Conclusion

In this chapter, we described and evaluated a multilevel graph partitioner for heterogeneous CPU-GPU systems. Some of the challenges we had to overcome on the GPU are: (1) memory constraints to hold large graphs; (2) the irregular nature of the graph data structure that can degrade GPU performance; (3) synchronization costs, which are much more pronounced on GPUs running tens of thousands of threads as compared to multi-core CPUs that only run tens of threads; (4) a suitable work-load distribution strategy between the CPU and the GPU; (5) an appropriate modification to the parallel partitioning algorithm to optimally distribute the sub-tasks between the CPU and GPU; (6) data-transfer latencies between the CPU and the GPU; and (7) differences in the architectural models between CPUs and GPUs (e.g., MIMD versus SIMD).

Our designed graph partitioner assigns the sub-tasks with high parallelism to GPU and transfers the less-computational based sub-tasks to the CPU. It also avoid the heavy overhead of fine synchronization among the GPU threads by performing a

lock-free coarsening phase.

To the best of our knowledge, this is the first graph partitioner for hybrid CPU-GPU environments that efficiently takes advantage of the processing power of the GPU in the coarsening and the un-coarsening phases. The experimental results show that our implementation outperforms both Metis and ParMetis and is comparable in performance and quality of the partitions with mt-metis.

# Chapter 5

# A High Performance Multilevel GPU-based Graph Partitioner

In this chapter, we optimize our partitioning strategy and describe an effective and methodological approach to enable multi-level graph partitioning on GPUs. Our solution avoids thread divergence and balances the load over GPU threads by dynamically assigning appropriate number of threads to process the graph vertices and irregular sized neighbors. Our design is autonomous as all the steps are carried out by the GPU with minimal CPU interference. We also employ a custom regional-memory allocator which results in better performance in comparison to the contemporary GPU allocator and increases the efficiency of partitioning. We show that our design outperforms state-of-the-art CPU-based parallel graph partitioner (mt-metis) in terms of partitioning speed.

In addition, we apply some of the techniques we developed specifically in the coarsening phase of our graph partitioner to another graph processing application that exhibits such characteristics as thread divergence and imbalance load distribution. Minimum Spanning Tree (MST) is a well-known algorithm that arises in many real world applications. Extending our partitioning techniques to MST results in a high-performance parallel implementation of MST on GPU, that outperforms the serial and multi-core implementations.

## 5.1 Motivation

According to the experimental evaluation in Chapter 4, our designed CPU-GPU partitioner performs similar to the multi-core partitioner mt-metiss. We identify the performance bottlenecks in different phases of the parallel partitioning algorithm, to extract the factors that degrade the performance. This helps us optimize our parallel graph partitioning algorithm accordingly so that it outperforms the mt-metis partitioner. The main factors that deteriorate the performance of our CPU-GPU graph partitioner are as follows:

- During the matching process in the coarsening phase, the graph vertices are distributed among the GPU threads and each thread finds the match for its assigned vertex serially. Here, traversing irregular-sized neighbor lists by different threads decreases warp execution efficiency and results in thread divergence and non-coalesced accesses to the vertex indices. Furthermore, high number of conflicts among thousands of GPU threads during the matching, slows down the coarsening rate of the graph.

- In the contraction step, using *compact-hash* table for merging the neighbor lists of matched vertices leads to load imbalance among the GPU threads. The reason is that each thread is responsible for constructing the neighbor list of its assigned collapsing vertices in the coarser graph. Since the number of neighbors varies from one vertex to another, the threads in the warp cannot finish processing the neighbors of their assigned vertices at the same time. Consequently, the threads diverge due to processing non-uniformly sized lists, which deteriorate the performance.

- In the refinement step, simultaneous access of many GPU threads to the list of vertex movements requests in each partition allocated buffer, creates a high memory contention.

- Transferring the less computational sections of the partitioning sub-tasks to

the CPU in the coarsening phase and moving back the coarsened graph to the GPU in the un-coarsening phase, creates some performance overhead due to comparatively low communication bandwidth between the CPU and GPU.

Such shortcomings, motivate the need for an end-to-end high-performance multilevel GPU-based graph partiioner, that accelerates the partitioning sub-tasks by being tailored specifically for the SIMD architecture while avoid the load imbalance and thread divergence.

## 5.2 Multilevel GPU-based Graph Partitioning

In this section, we introduce the mechanics of our multilevel GPU graph partitioner [30]. In this design we balance the load across the GPU threads in the critical steps of graph partitioning. Similar to Chapter 4 , we use the Compressed Sparse Row (CSR) representation to store the graph on the GPU, which consists of 4 arrays: an *adjacency* array (*adjcny*) of length $2 \times |E|$, which stores the adjacency list of the graph vertices, and an *adjacency pointer* array (*adjp*) of length $|V| + 1$, which points to the adjacency set of each vertex in the *adjacency array*. In addition, the adjacency weight *(adjwgt)* of length $2 \times |E|$ and the vertex weight *(vwgt)* of length $|V|$ contain the weights of the edges and vertices, respectively. A *matching* array $M$ of length $|V|$ that includes the matched pairs to be collapsed in the coarser graph, and a *mapping* array (*Cmap*) of length $|V|$ that stores the vertex labels in the coarser graph, are also augmented to the CSR data structure.

In the matching step the load is balanced by dynamically assigning appropriate number of the SIMD threads to process a vertex with irregular-sized neighbors. Each warp processes a set of consecutive vertices while warp threads cooperate and explore the neighbor list of each warp-assigned vertex in order to find the matching candidates. Concurrent collaboration of all the threads in the warp in processing the neighbors of the warp-assigned vertices eliminates the intra-warp load imbalance and hence the thread divergence. We also parallelize the contraction step by simultaneous merging of

the neighbor lists of the matched vertices and applying a parallel load-balanced sorting followed by a duplicate-removal method to create the coarser graph. In addition, for the refinement step we propose a roll-back free approach where SIMD threads collaboratively find the potential boundary vertex movements among the partitions that improve the edge cut. A combination of movements which results in the highest reduction in the edge cut is selected iteratively and the weights of involving partitions are updated accordingly.

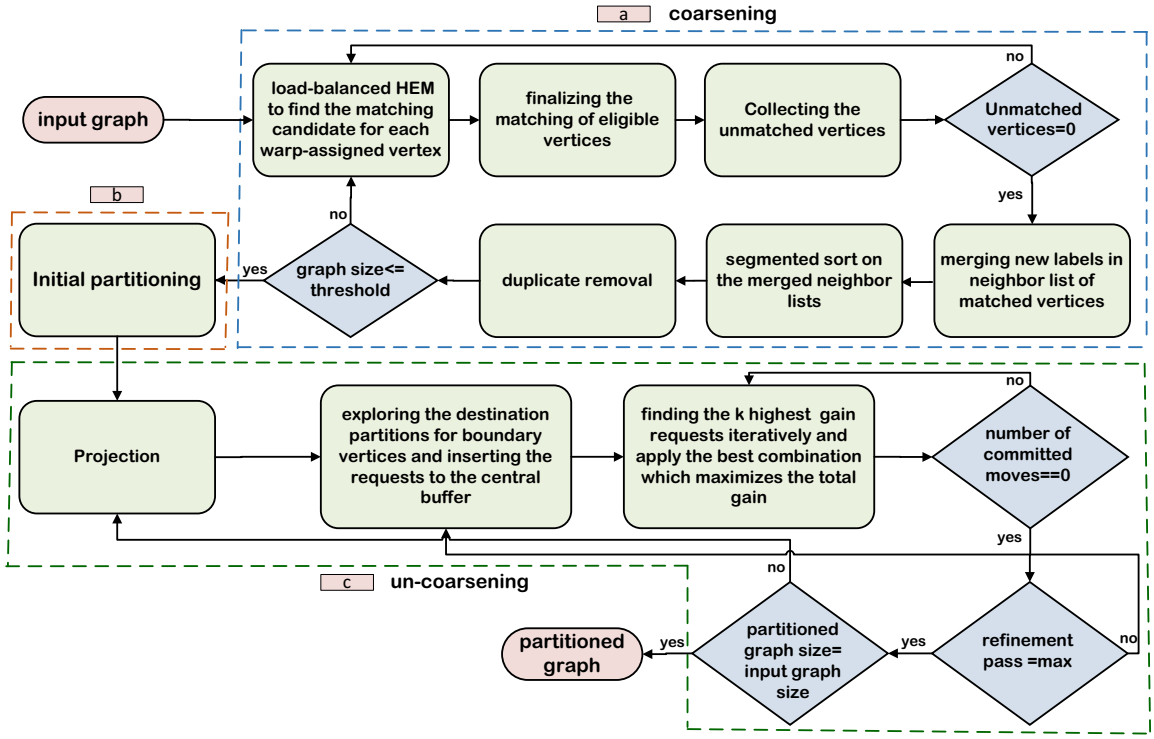



Figure 5.1: GPU graph partitioning flowchart. Green-colored boxes represent GPU operations and blue-colored boxes specify the host actions.

Figure 5.1 shows the flowchart of our proposed GPU graph partitioning scheme. Starting from the coarsening phase (box a), for a warp-efficient Heavy Edge Matching (HEM) [50] implementation we assign each warp to a group of 32 (i.e., warp width) consecutive vertices and cache associated neighbor lists inside the fast warp-specific shared memory.

Inside each warp, threads process the neighbor list of the warp-assigned vertices

collaboratively and perform a parallel reduction to find the edge with the maximum weight incident on each vertex within the group collectively. These matching candidates are written to a buffer on global memory. Then, in a uniform manner, each thread inside the warp explores the matching candidate for one of the vertices inside the group. If the candidate has a better matching suggestion, this vertex remains unmatched and it is explored in the following iterations of matching. On the other hand, if this vertex has a heavier edge incident on the candidate in comparison to the previous suggestion from another neighbor of the candidate, the thread overwrites it, but the warp will be responsible for finding the matching pair for the candidate neighbor in the following iterations. The remaining unmatched vertex are collected on the warp's designated shared memory region and warp threads continue processing the vertices iteratively until all of the warp assigned vertices are either matched or cannot find any pair.

Moreover, in the contraction step, in order to construct the new neighbor list for matched vertex pairs, new labels of the adjacency list of each pair are merged together. Then we perform an efficient parallel segmented sort [7, 93] followed by a parallel duplicate elimination routine. This is necessary since not only collapsed pairs may have common neighbors but also two neighbors of a pair may have collapsed themselves. We iteratively apply the coarsening phase until the number of vertices in the coarsest graph is less than a threshold hyper-parameter [1].

Initial partitioning (Figure 5.1 box b) is performed on the coarsest graph with a much smaller number of vertices compared to the original graph. This phase contains a small fraction of the overall partitioning time.

To implement this phase, we use a graph growing partitioning algorithm[47] by starting from a random vertex and bisecting the graph through a parallel Breadth-First Search (BFS) on GPU and refinement of the bisected partitions. We keep bisecting the created partitions until the number of required partitions is obtained.

Finally box c in Figure 5.1 demonstrates the un-coarsening phase that contains

[1] We will discuss and analyze the effect of this threshold in later sections.

the projection and refinement steps. Using vertex mapping information preserved from the coarsening phase, we parallelize the projection by dividing the vertices of the finer graph among the threads and having each thread specify the partition label for the projected vertex in the coarser graph. In the refinement step each thread finds the best possible destination partition for migration of its assigned boundary vertex if the move results in the edge cut reduction. All the threads insert their movement request in a central buffer allocated on global memory using efficient warp-aggregated atomics. Then we find the $k$ highest gain requests using a custom parallel reduction kernel. $k$ is a design hyper-parameter and obtained empirically. We assign each of $2^k$ permutations (two possible states for each move: *committed* or *rejected*) of these movement requests to $2^k$ warps, making each warp responsible for verifying one permutation. The warp threads collaboratively apply the committed moving requests based on the permutation bits and update the partition weights locally. If the warp assigned movements do not violate the balance constrains, the warp writes the total gain of that specific permutation to global memory. Finally the valid move set permutation with the highest gain is selected for committing onto partition weights. Partition labels for vertices that are affected by this move set are updated as well.The refinement process continues iteratively until the maximum number of passes has been reached or none of the move combinations can be committed.

In the rest of this section, we elaborate on details of our design.

### 5.2.1 Matching

Matching step constitutes the inner-most loop of coarsening phase (3 top most green boxes in Figure 5.1) where the goal is to pair up each vertex with its most suitable neighbor for merging. Heavy-Edge Matching (HEM) is our algorithm of choice for matching.

Algorithm 5.1 demonstrates our parallel matching which consists of two phases as follows:

***Matching Candidate Discovery.*** In order to match the vertices using HEM,

**Algorithm 5.1** Parallel Matching.

```
1: procedure MATCHING
2:     parallel_for warp w{
3:     fetch the neighbor lists of 32 vertices into the shared memory
4:     neighbor_list_length = total length of adjncy lists of 32 vertices
5:     while (there are vertices that can be matched) {
6:     /*Matching Candidate Discovery*/
7:        for(i= laneId; i< neighbor_list_length; i+ = warpsize) {
8:              Map the laneId to the proper position in the
9:              neighbor lists of collected vertices
10:              Reduce inside segment
11:        }
12:         Write the index and corresponding edge weight of the candidate
13:         neighbor on Candidate array
14:    /*Candidate Cross-examination*/
15:         Finalize the Matching on M
16:         Collect unmatched vertices
17:         Calculate the new neighbor_list_length
18:      }
19:     }
```

each warp processes the adjacency list of 32 consecutive vertices of the graph. Note that these vertices may have neighbor list segments with different lengths.

The threads of the warp also fetch 32 corresponding elements of *adjp* into a designated shared memory buffer. Using the *adjp* array's starting and ending element, warp threads can recognize the regions within *adjcnywgt* and *adjncy* arrays that are assigned to the group of vertices.

After the fetch, all threads in a warp explore the neighbor list of warp-assigned vertices collectively and find the heaviest edge candidate of matching in the neighbor list of each vertex.

For an efficient implementation of parallel matching candidate discovery, we utilized a modified version of Warp Segmentation (WS) technique [52] as follows. Warp threads perform a fast binary search over *adjp* elements for their assigned edge index to find the source vertices that are currently processed by warp threads. Then the warp threads perform an intra-warp parallel reduction over the corresponding

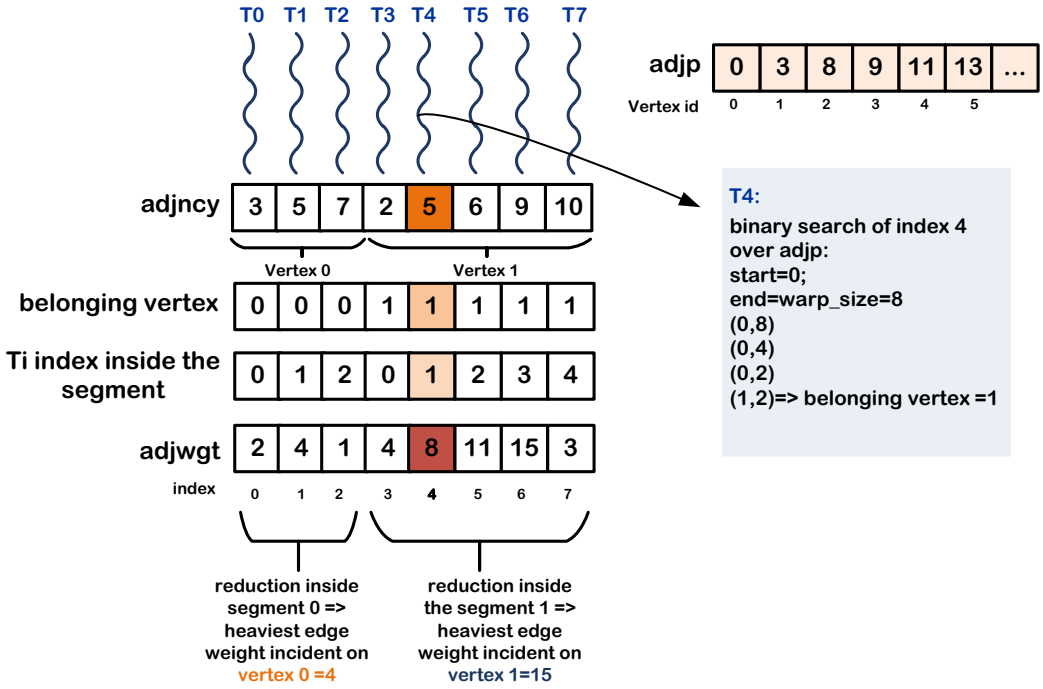



Figure 5.2: Heavy edge matching process inside a warp. Warp size is assumed 8.

edge weights of neighbor list of each warp-assigned vertex to calculate the maximum-weight. Finally, threads write the index and corresponding edge weight of the candidate neighbor on an array (called *Candidates*) in the global memory. This design keeps all warp threads busy during matching candidate discovery regardless of the irregularities of the neighbor list lengths, maximizing warp utilization.

Figure 5.2 shows an example of a heavy edge matching process using warp segmentation (the warp size is assumed 8). The threads 0 to 2 ($T_0$ to $T_2$) process the neighbor list of vertex 0 and the threads 3 to 7 ($T_3$ to $T_7$) process the neighbor list of vertex 1. Each warp thread is assigned to one edge and performs a binary search over *adjp* to find the source vertex incident on its assigned edge. For example according to Figure 5.2, $T_4$ processes the neighbor vertex 5 at index 4 with the corresponding edge weight of 8. $T_4$ performs a binary search of index 4 over *adjp* which results in calculating vertex 1 as the source vertex incident on the edge $(1, 5)$. As Figure 5.2 shows, after calculating the thread indices inside each segment, the warp threads reduce on the heaviest edge value of *adjwgt* elements in each segment. Since the warp

threads process the neighbor lists of warp assigned vertices concurrently, the load is balanced over the warp threads, and thread divergence is avoided.

***Iterative Candidate Cross-examination.*** Here, GPU threads work in parallel to examine the validity of matching candidates produced in the previous step. Threads uniformly map to vertices; for its assigned vertex, each thread retrieves the index and the incident edge of the current matching candidate from the *Candidates* array and then examines the edge weight stored in the corresponding element for this candidate in the *Matching* array $M$. If the thread's assigned vertex has heavier incident edge weight on the candidate, the thread succeeds in overwriting the old value written in $M$ and sets its assigned vertex index as the new pair for the candidate vertex. If so, the warp will be responsible to match the evicted vertex in the following iterations. For a safe replacement in presence of concurrent threads, we utilize the eight-bytes-long `atomicMax()` to ensure the selection of the neighbor with the heaviest edge. Inspecting the return of the atomic specifies the success of the operation. Conversely, if the thread is unsuccessful in this iteration, its remaining candidates in the following iterations will be verified.

The set of matching steps described above is recursive, i.e., warp threads collect all the remaining unmatched vertices using intra-warp prefix sum on the warp shared memory and perform candidate discovery and cross-examination again. The collection of unmatched vertices in our matching step is inspired by the technique used by [72]. However our technique provides a better warp efficiency. During the matching process in [72], warp threads process the collected vertices serially. On the contrary we use an augmented version of WS named CTE [53] that efficiently maps threads within the warp to the elements of the neighbor lists belonging to the collected vertices and similar to the first iteration all the warp threads collaboratively process the disjointly-located neighbor lists of the collected warp-assigned vertices in a parallel load-balanced fashion.

A warp finishes its matching process when no unmatched vertex is left for matching. This means either all of them are matched, or some are matched and some could

find no matching candidate.

### 5.2.2 Contraction

Contraction step collapses matched vertices in order to form the coarser graph. This step pertains to the 3 right most green boxes in the second row in Figure 5.1. In contraction, the main challenge for every matched pair is to discover the exact location in the *cadjcny* and the *cadjwgt* arrays in which their set of neighbors reside. This is necessary for making the CSR representation for the next coarsening iteration.

**Algorithm 5.2** Parallel Cmap Construction.

```
1:  procedure CMAP CREATION
2:      foreach (vertex Vi: assigned to thread Ti) in parallel do
3:      if Vi > M[Vi] then
4:          M.state[Vi] ← 0
5:      else
6:          M.state[Vi] ← 1
7:      PV ← Parallel_Binary_Prefix_Sum(M.state)
8:      foreach (vertex Vi: assigned to thread Ti) in parallel do
9:      if Vi > M[Vi] then
10:         Cmap[Vi] ← PV[M[Vi]]
```

***Cmap Construction.*** After finalizing the *matching* array $M$, we construct the *mapping* array *Cmap* that stores the vertex labels in the coarser graph. Algorithm 1 shows our proposed parallel process for creating *Cmap*. First, we linearly assign vertices to threads (line 2). If the assigned vertex has bigger label compared to its match, the thread resets (zeros) the vertex's corresponding *M.state* bit. However, if the vertex has smaller label compared to its match or it is matched to itself, then the thread sets the vertex corresponding bit in *M.state*. Then an exclusive binary prefix sum is computed on *M.state* to create the intermediate $PV$ array (line 7). $PV$ array essentially demonstrates the range of vertex labels in the coarser graph. Therefore, the new label of each matched pair of vertices in the coarser graph is the corresponding value of the one with the smaller vertex number in the $PV$ array. This process is demonstrated using lines 8 to 10 in Algorithm 5.3.

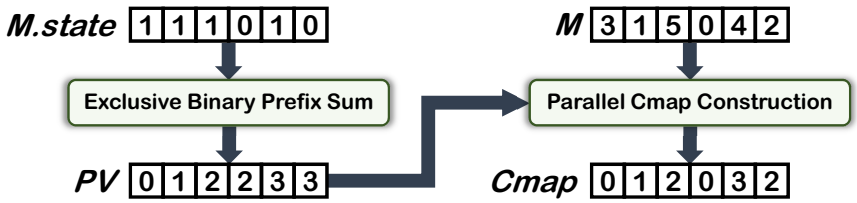


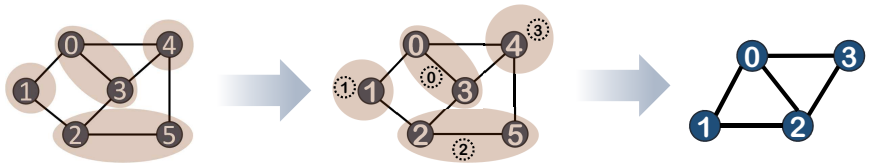

Figure 5.3: Cmap construction procedure for the graph shown in Figure 4.2. Number of vertices in the coarser graph is 4.

Similarly, Figure 5.3 illustrates an example of creating the *Cmap* based on the matching array for the graph shown in Figure 4.2.

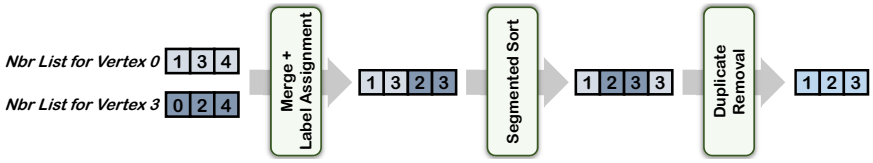
***Merging Adjacency Lists.*** At the start of the contraction step, we conservatively assume that for each vertex $u$ matched with vertex $w$, the number of entries in the coarser graph's adjacency array for the pair is the accumulated number of neighbors in the adjacency set of $u$ and $w$ (the maximum possible). Also, for a vertex $u$ matched to itself, the required number of entries will be equal to the size of its adjacency set. We allocate and utilize an auxiliary array named $T.cadjp$ with the length equal to the number of vertices in the coarser graph in order to hold the temporary length of neighbor lists for matched vertices.

Figure 5.4 shows the contraction step using an example. First, within a kernel, we uniformly map vertices in the coarser graph to kernel threads and allow threads to calculate the maximum possible number of entries in the adjacency list ($cadjcny$) and adjacency weight ($cadjwgt$) of matched pairs in parallel. The results are saved in $T.cadjp$ array. Then an exclusive parallel prefix sum on $T.cadjp$ array gives the initial start index of neighbor lists of each vertex in the coarser graph's adjacency list and adjacency weight arrays. Two temporary arrays $tcadjcny$ and $tcadjwgt$ are then allocated on GPU global memory, which store temporary adjacency lists and temporary adjacency weights respectively.

Then we distribute the vertices of the coarser graph in groups of 32 to the warps. using $Cmap$ and $M$ arrays, the warp threads copy the content of $Cmap$ (labels of vertices in the coarser graph) and edge weights into the $tcadjcny$ and $tcadjwgt$ arrays.




(a) Matched vertices in the finer graph are relabeled and represented with one vertex in the coarser graph.




(b) Contraction steps shown for two matched vertices.

Figure 5.4: Visualizing contraction using the example graph in Figure 4.2.

After merging, there can be duplicate entries in the merged neighbor list of the matched vertices, which can be due to either matched pairs having a neighbor in common or two distinct neighbors of the merged set being matched as well. This necessitates a duplicate elimination step in order to represent matched pairs or their common neighbors as only one entity in the coarse graph. To remove the duplicated values, we sort the *tcadjncy* within each neighbor group and then remove the consecutive repeated elements. In more details we apply a parallel GPU segmented-sort [7, 93] on the *tcadjncy*, which takes the unsorted array with different sized segments (each segment is the neighbor list of one vertex) and returns the array with sorted segments.

Figure 5.5 shows an example of the segmented sort. Parallel segmented sort builds on a modified merge sort algorithm for GPUs, which is load-balanced. Efficient execution of merge sort on GPU cores needs segments of equal length. But variation in the length of segments degrades the performance of merge sort. To resolve this problem the segmented sort works as follows.

In the first step, the *tcadjncy* array is divided into equal sized chunks and each chunk is sorted in parallel on the GPU. The segmented merging algorithm is based on the Merge Path approach [32], which proposes an efficient fully parallel lock-free merging algorithm for GPUs. In this method, the workload of the merge sorted arrays

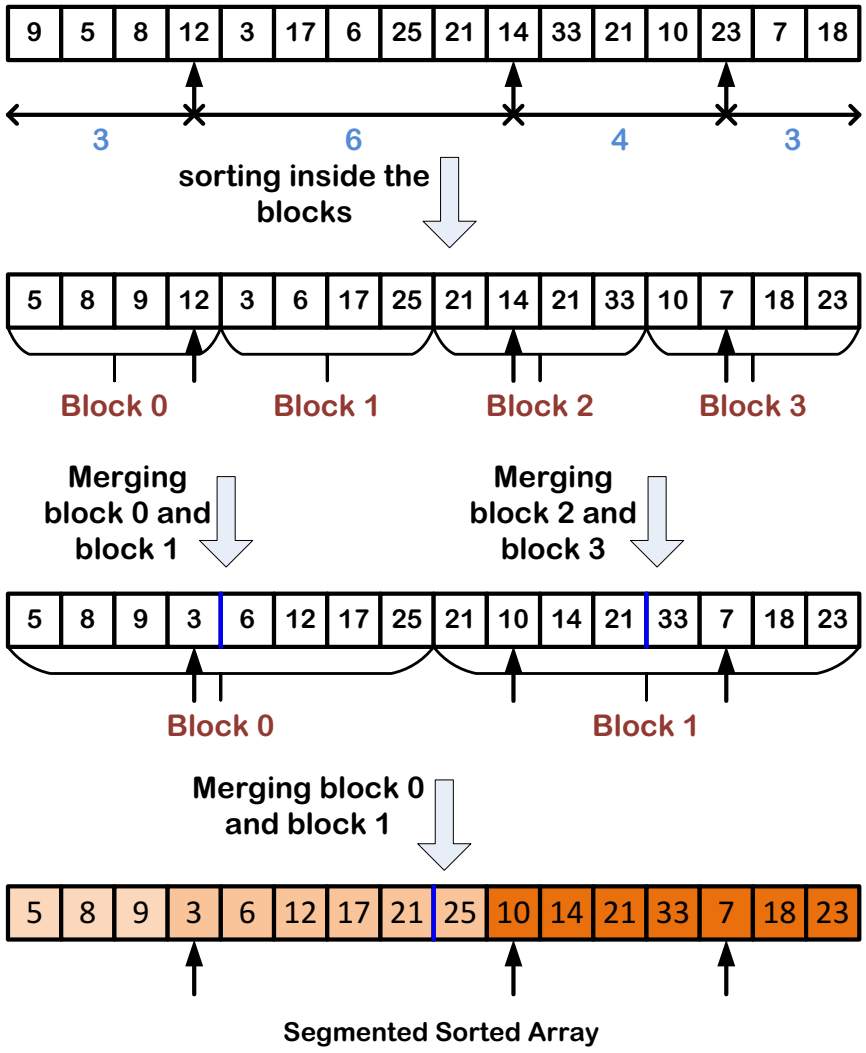



Figure 5.5: Segmented sort on an array with four different sized segments.

is partitioned among the threads evenly and the threads complete the merging step by comparing the elements in their assigned path and conduct a diagonal binary search to find the starting point in their assigned path and construct a partial part of the final sorted array independently.

To consider the segments' various lengths in the merging of two consecutive segments, the merging is just applied on the elements of the segments that cross the boundary on neighboring chunks. The other elements included in the merging are just copied to the output array unchanged. The borders that contain the spanned segments are called active boundaries. For example in Figure 5.5, there are two active

boundaries in the first round and one in the second round. According to the merge rules in each round the number of boundaries is divided by 2 and the same logic is applied for merging in each round.

Note that based on the rearrangement of the indices after applying the segmented sort, *tcadjwgt* entries are also rearranged such that weights are in accordance with their associated neighbor list indices. With *tcadjcny* array that sorted within each neighbor list, we can remove similar adjacent items in and sum their corresponding edge weights in the *tacadjwgt*. Algorithm 5.3 summarizes our parallel duplicate removal process. The algorithm starts by assigning one vertex of *tcadjcny* to each thread. A zero-initialized bit array (*Dup.state*) of length *tcadjcny.length* is allocated on GPU to specify the duplicates. Each thread compares its assigned vertex with the previous one in *tcadjcny*, and if they are different, it sets the corresponding *Dup.state* bit (lines 3 and 4 in Algorithm 5.3). Then an exclusive binary prefix-sum is performed on *Dup.state* in parallel (line 5) to find the location of non-duplicated vertices in the final adjacency lists of the coarser graph. To implement the binary prefix sum, we augmented efficient intra-warp binary prefix sum described in [38] by scaling it across multiple thread-blocks.

Figure 5.6 shows our prefix sum implementation using CUDA. In the first round (`init_prefix_sum` kernel ), each warp lane reads 32 bits from *Dup.state* and counts the number of bits that are set to 1 (using `popc()`). Calculated values are further reduced within the warp using iterative butterfly shuffle instruction [96]. Next we store the partial sums into an intermediate array on the global memory (`warp_sum` in Figure 5.6) and apply an exclusive prefix sum on this array using CUB primitive [69]. Then we perform a prefix sum across thread-blocks (1024 threads per thread-block) while threads within the block collect the cumulative sums from the previous step.

Next, using the results constructed from previous step stored in (*Par.statet*), and also using *Dup.state* bitmask, we finalize the construction of two arrays *cadjcny* and *cadjwgt*. According to lines 6 to 11 of Algorithm 5.3, for the elements with corresponding bit of 1 in *Dup.state* the vertex and accordingly the edge weight is

**Device:**

```
__device__ unsigned int  intrawarp(uint *Dup.state) {
int global_T_id=threadIdx.x+blockIdx.x*blockDim.x;
return __popc(Dup.state[global_T_id]);  }

__global__ void init_prefix_sum(uint *Dup.state, uint * partial_sum) {
const int tid = threadIdx.x;
const uint global_T_id =threadIdx.x + blockIdx.x * blockDim.x;
const uint global_W_id = global_T_id & (~(warp_size-1));
int mySum = intrawarp(Dup.state);
 mySum += __shfl_xor( mySum, 16 );
 mySum += __shfl_xor( mySum,  8 );
 mySum += __shfl_xor( mySum,  4 );
 mySum += __shfl_xor( mySum,  2 );
 mySum += __shfl_xor( mySum,  1 );
  if ( global_T_id & 31 == 0)
      partial_sum[global_W_id] = mySum; }

__global__ void dev_prefix_sum(uint *Dup.state, uint * dev_prefix_res, uint * partial_sum){
uint Intra_block_ prefix ;
uint laneidx;
uint global_T_id=threadIdx.x+blockIdx.x*blockDim.x;
const uint global_W_id = global_T_id & (~(warp_size-1));
uint pos=threadIdx.x & (warp_size-1);
uint *ptr=&Dup.state[global_W_id];
c = getBitmapAt(ptr, pos);
intra_block_ prefix = block_binary_prefix_sum (c);
dev_prefix_res[global_T_id]= intra_block_prefix + partial_sum[blockIdx.x]; }
```

(a) Device side

**Host:**

```
init_prefix-sum <<<partial_sum_size, nthreads >>>( Dup.state, partial_sum);
cub::DeviceScan::ExclusiveSum(d_temp_storage, temp_storage_bytes, partial_sum,
partial_sum, partialsum_size+1);
dev_prefix-sum <<< partial_sum_size, 1024, 1000*sizeof(int) >>> (Dup.state,
dev_prefix_result, partial_sum);
```

(b) Host side

Figure 5.6: Prefix sum code.

copied to coarser graph arrays. However the vertices with the value of 0 in the *Dup.state* are the duplicated elements and only their corresponding weight are added to weight saved in the location of their first occurrence.

### 5.2.3 Initial Partitioning

The initial partitioning is performed on the coarsest graph so as to have a much smaller problem size. This makes initial partitioning take only a small fraction of the total execution time. To create a k-way partition of the coarsest graph we employ a graph growing partitioning algorithm [47] in which we grow an area around a random selected vertex using Breadth-First Search (BFS) and expand this area with more vertices until accumulated vertex weights reaches half of the total weight of graph vertices. We repeat this process $\lceil log_2(p) \rceil$ times to create $p$ different partitions. We

**Algorithm 5.3** Parallel Duplicate Removal

```
1:  procedure DUPLICATES REMOVAL
2:      foreach (vertex Vi in tcadjcny: assigned to thread Ti) in parallel do
3:      if tcadncy[Vi]! = tcadjcny[Vi − 1] then
4:          Dup.state[Vi] ← 1
5:      Parallel exclusive prefix sum on bitarray and save the results on Par.state
6:      foreach (vertex Vi: in tacadjcny assigned to thread Ti) in parallel do
7:      if Dup.state[Vi] = 1 then
8:          cadjcny[Par.state[Vi]] ← tcadjcny[Vi]
9:          cadjwgt[Par.state[Vi]] ← tcadjwgt[Vi]
10:     else
11:         cadjwgt[Par.state[Vi − 1]] ← cadjwgt[Par.state[Vi − 1] + tcadjwgt[Vi]
```

parallelize this phase by implementing a parallel BFS on GPU executed for each bisectioning step.

### 5.2.4 Un-Coarsening

Un-coarsening consists of *projection* and *refinement*. During the projection, the coarser graph constructed at level $i + 1$ is projected back onto the finer graph at level $i$. We parallelize this step by dividing the vertices of the finer graph among the threads of a new kernel. We assign threads to realize the partition labels of the projected vertices in the finer graph ($G_i$) by visiting the Cmap array and the partition labels of vertices in the coarser graph ($G_{i+1}$). Since this step is straightforward, the rest of this section discusses the refinement step, which is more challenging from the load distribution perspective.

Refinement step attempts to improve the graph edge cut by moving some of the boundary vertices between projected partitions. As graphs get finer, the weight distribution resolution across vertices increases. Therefore, applying refinement after every level of un-coarsening provides higher partitioning precision. Figure 5.7 gives an overview of the refinement procedure. The parallel implementation of the refinement consists of 3 steps as we discuss below.

***Migration Request Insertion.*** In this step, we distribute the vertices in the

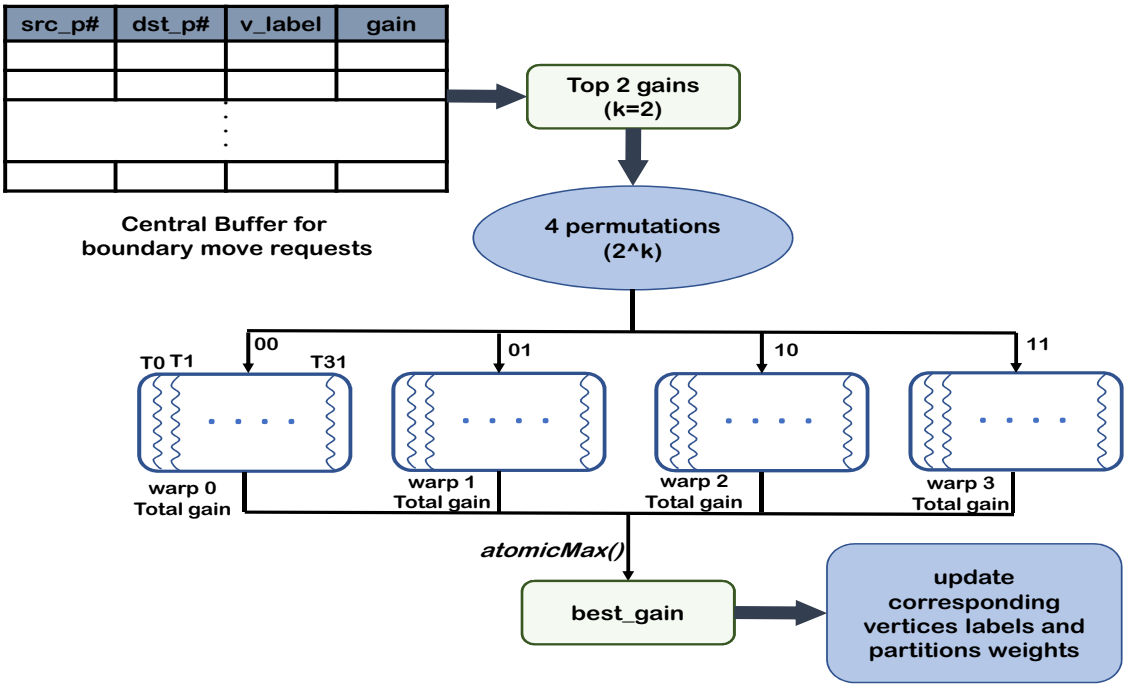



Figure 5.7: Refinement procedure with assumed k value of 2.

finer graph among the threads and each thread determines if its assigned vertex is boundary. If the vertex is boundary, the thread finds the best destination partition for migration for the vertex. A destination partition for moving a vertex is chosen between candidate partitions if this move results in the maximum reduction of the edge cut and does not underweight the source or overweight the destination partition.

A central buffer shared among the threads inside the global memory accepts move requests coming from different warps. Each entry of the central buffer contains 4 fields: source vertex label, source partition number, destination partition number and potential improved value of edge cut (gain). We utilize a variable (incremented using atomics) inside the DRAM to specify the next location inside the buffer the threads need to write. We utilized warp-aggregated atomics to reduce the contention over this variable.

***Top Gain Selection.*** This step involves finding the top k highest potential gains among the requests in the central buffer. To this aim, we perform a custom parallel

reduction kernel iteratively as follows. We execute a parallel two-pass reduction [96] with a custom reduction function. In the first round a reduction is performed within the blocks where each thread reads $k$ entries of the central buffer and sorts them. During the reduction, $k$ entries with the bigger gain values are returned (by applying the merge function on sorted arrays of threads) and the block results ($k$ for each block) are saved on an intermediate array. In the second pass, we perform the same reduction function on the intermediate array within a single block. As a result, the final outcome is the top $k$ entries with the highest gain.

***Parallel Permutation Verification.*** These top-$k$ move requests may contradict each other and therefore applying them concurrently might lead to over-weighted or under-weighted partitions. Instead of employing a role-back approach, we parallelize the conflict verification by distributing all possible move permutations across warps. We create $2^k$ move permutations from the move requests with highest gains and allow each warp to process one of these permutations. For example if we assume $k$ is equal to 8, a warp that has a global ID of `0b01000001` will verify the scenario where only the 1st and the 7th move requests are applied. Within the warp, threads read the accepted move requests from the global memory to a warp-specific shared memory region. The warp lanes collaboratively apply the committed moves (corresponding bit 1) of the warp assigned permutation value and calculate the partition weights and potential gain of the permutation accordingly.

To verify a permutation, the warp threads read a subset of top-$k$ gain entries according to the warp assigned permutation and copy the source and destination partition numbers of these entries to a buffer on the shared memory (*part*). We also utilize two other on-chip shared memory buffers to keep the sorted partition numbers (*sort_part*) and final weight changes (*w_changes*) of these partitions. Since some of the partitions may appear as both source and destination in entries being explored, warp lanes perform a radix sort on *part* to put the similar partition numbers in the consecutive locations and save the results on *sort_part*. They collaborate in removing the duplicates within *sort_part* and then read the initial partition numbers from *part*

to find each partition index in *sort_part* using a fast binary search. If the partition number read by a thread from *part* is a source partition, the thread updates the corresponding index (found by binary search on *sort_part*) of this partition in *w_change* by using `atomicSub()`. Conversely, if the thread reads a destination partitioner, it updates its corresponding index in *w_changes* using `atomicAdd()`.

In this scenario, for a particular permutation verified by the warp, some of the partitions may overweight or underweight. Therefore, the warp threads read the updated value of affected partitions and if at least one of the partitions violates the load balance, the warp assigned permutation is rejected by assigning an invalid value to the potential gain of the warp. Otherwise the first thread of the warp writes the permutation's potential gain and the warp number in a location on global memory (`best_gain`) using `atomicMax()`.

When all the warps finalized the gains, the best gain is stored on `best_gain` variable . According to the valid permutation with the best gain, we update partition labels and partition weights for the corresponding vertices at the end of the reduction kernel. We repeat finding the top-$k$ highest gain movement combinations until no move is possible.

The refinement step is performed iteratively aimed to improve the edge cut while keeping the partitions balanced. When we arrive at the maximum number of passes, or no vertices were moved in the last refinement pass, the refinement in the current level of un-coarsening ends.

### 5.2.5 Additional Optimization: Custom Memory Allocator

As we described earlier, coarsening phase hierarchically constructs coarser graphs that need to be saved inside the global memory to be used later during the un-coarsening phase. Also we need temporary arrays during the computation to hold intermediate data. If we rely on CUDA runtime for memory allocation (using `cudaMalloc()`) each time that we need a global memory buffer, since the requests have to be registered with the host, we pay a considerable penalty. To minimize this recurring delay and

avoid it in the critical path of our solution, we developed a custom memory allocator on GPU based on the region-based [2] memory allocation concept [42]. In our design, we over-provision the future memory usage and allocate a large chunk of memory, while an allocation pointer keeps track of the inserted data. When a data structure is allocated from a region, the allocation pointer is incremented by its size and the new pointer is returned.

## 5.3 Experimental Evaluation

In this section we evaluate the performance of our designed multilevel GPU graph partitioner and compare it against Metis 5.1.0 and mt-metis 0.6.0. We also demonstrate the effectiveness of our design by profiling the warp efficiency in the critical kernels of the multilevel GPU graph partitioner. The system we performed experiments on is equipped with an Intel Xeon E5540 processor with 8 cores and an Nvidia Kepler K40 GPU with 12 GB of global memory. we compiled our code with highest optimization level flag applied using CUDA version 8.0.

We use graphs listed in Table 5.1 for the experiments. These are undirected graphs from various areas of computation obtained from DIMACS10 [6] and University of Florida Sparse Matrix Collection [22]. The graphs edge weights were added as the random numbers between 0 and 100.

For all the experimental evaluations we partition the input graphs into 64 partitions and the imbalance tolerance rate for each partition is set to 3% (as in Metis and mt-metis). To have a fair comparison with mt-metis the coarsening rate is set to 0.85 and the coarsest graph size threshold variable (known as $CoarsenTo$) has been set to $\frac{Numberof Vertices}{20\times(\log_2(npartitions))}$ which both are similar to mt-metis.

[2] Region-based memory allocation is also known as arena-based memory allocation.

| Graph | $\|V\|$ | $\|E\|$ | Description |
|---|---|---|---|
| m14b | 214,765 | 1,679,018 | Walshaw's Graph Partitioning Archive |
| delaunay | 1,048,576 | 3,145,686 | Delaunay triangulation of random points |
| auto | 448,695 | 3,314,611 | Walshaw's Graph Partitioning Archive |
| AS365 | 3,799,275 | 11,368,076 | An Eurocopter AS365 Dauphin |
| NLR | 4,163,763 | 12,487,976 | Numerical simulation |
| adaptive | 6,815,744 | 13,624,320 | Numerical simulation |
| co-papers | 540,486 | 15,245,729 | Co-author and Citation Networks |
| idoor | 952,204 | 22,785,143 | Sparse matrix from UFC |
| af-shell | 1,508,065 | 25,582,130 | Sparse matrix from UFC |
| USA roads | 23,947,347 | 28,854,312 | Road network |
| Serena | 1,391,349 | 31,570,176 | gas resevoir simulation for CO2 sequestration |
| audikw1 | 943,695 | 38,354,076 | Structural Problem |
| channel-500 | 4,802,000 | 42,681,372 | Numerical simulation |
| dielfilterV3 | 1,102,824 | 44,101,598 | Electromagnetics Problem |
| nlpkkt120 | 3,542,400 | 46,651,696 | Sparse matrix from UFC |
| Flan_1565 | 1,564,794 | 57,920,625 | 2D Structural Problem |

Table 5.1: Input graphs used for the experiments.

### 5.3.1 Performance Comparison

Figure 5.8 shows the speedup achieved by our GPU graph partitioner and mt-metis (with 8 threads) over the serial Metis. In each case, we consider the minimum runtime of three experiments in calculating the speedup. As Figure 5.8 illustrates our GPU partitioner outperforms mt-metis across all the input graphs ranging from 1.45 to 2.35. This speedup is mainly due to high level of parallelism in processing of the neighbor list of the graph vertices in the matching and contraction steps. The fast boundary move combinations evaluation in the refinement step through the warp threads also ameliorate the speedup. On average our solution performs 1.93× and 5.81× faster than multi-threaded and single-threaded versions respectively.

To further compare the performance of our GPU partitioner with mt-metis's, we demonstrate the time distribution of the three phases of partitioning for mt-metis and our GPU graph partitioner in Figure 5.9. Please note that these measurements do not

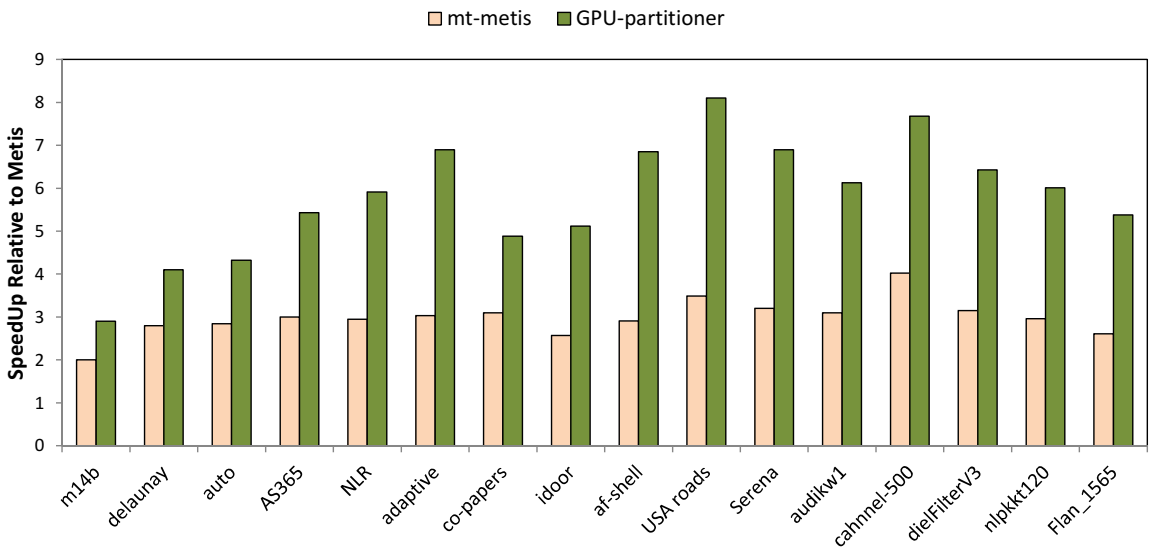



Figure 5.8: Speedup of mt-metis and GPU-partitioner relative to Metis

include I/O duration. It is evident from Figure 5.9 that our partitioner reduces the runtime significantly in the coarsening phase. The initial partitioning is almost the same as mt-metis because the design space in which The GPU implementation can be effectively parallelized is small. Finally the un-coarsening time in our partitioner is again lower than mt-metis. Using a global buffer for the boundary vertices and concurrent calculation of boundary vertices with top gains over the warps results in better performance compared to mt-metis in the refinement phase.

Next we evaluate the edge cut ratio of both methods (ours and mt-metis) relative to serial Metis. The results are listed in Table 5.2. Therein, we show that GPU graph partitioner is able to produce partitions of comparable quality to mt-metis. Serialization of the refinement step for mt-metis by distributing the boundary vertices among a few number of threads is the main reason for its supremacy in terms of edge cut.

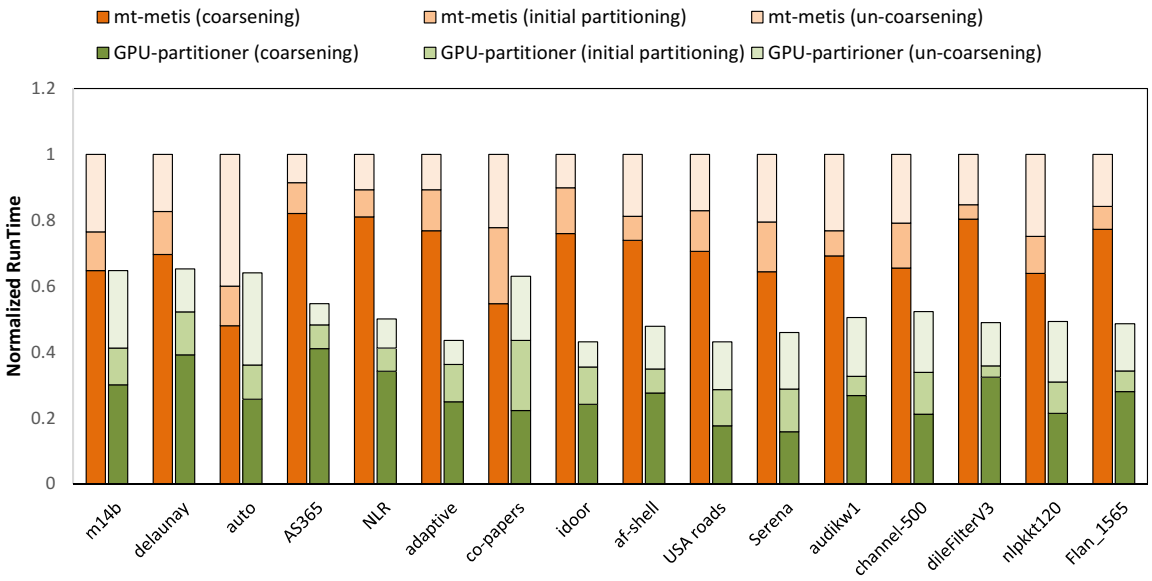



Figure 5.9: Time distribution of 3 partitioning phases of mt-metis and GPU-partitioner

| Graph | mt-metis | GPU-partitioner |
|---|---|---|
| m14b | 1.02 | 1.028 |
| delaunay | 1.01 | 1.031 |
| auto | 1.016 | 1.031 |
| AS365 | 1.024 | 1.045 |
| NLR | 1.024 | 1.05 |
| adaptive | 1.028 | 1.049 |
| co-papers | 1.12 | 1.17 |
| idoor | 1.01 | 1.05 |
| af-shell | 1.02 | 1.046 |
| USA roads | 1.14 | 1.17 |
| Serena | 1.024 | 1.038 |
| audikw1 | 1.018 | 1.0342 |
| channel-500 | 1.008 | 1.02 |
| dielFilterV3 | 1.027 | 1.051 |
| nlpkkt120 | 1.014 | 1.019 |
| Flan_1565 | 1.006 | 1.011 |

Table 5.2: Edge cut ratio in comparison to Metis.

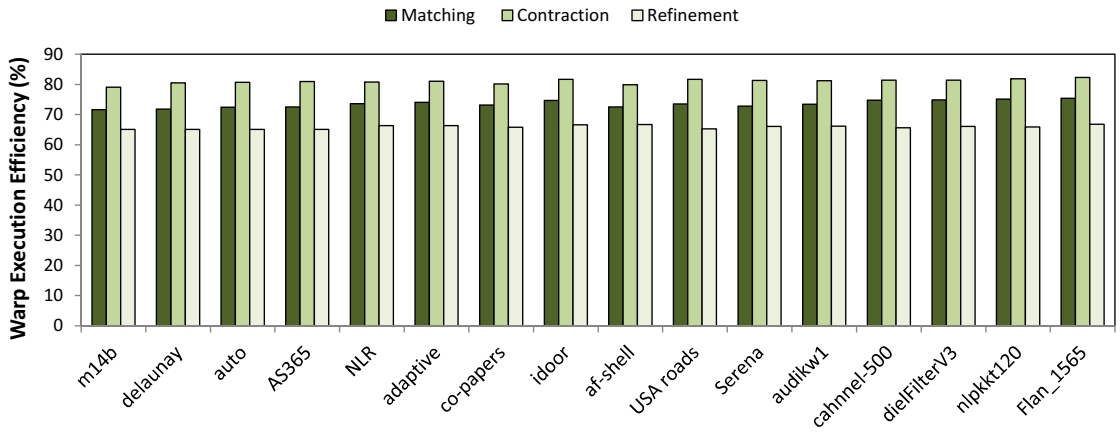



Figure 5.10: Profiled average warp execution efficiency in different kernels

### 5.3.2 Performance Analysis

To evaluate the effectiveness of our partitioner implementation on GPU, we profiled our partitioner's main kernels over different graphs for warp execution efficiency. We only measure the warp-efficiency for the main kernels within coarsening and uncoarsening phases which have been the focus of our design. Figure 5.10 shows the average warp efficiency results for 3 runs. The warp execution efficiency on average for the matching and contraction main kernels is 73% and 81%, respectively and 66% for the refinement kernel. The irregularity of refinement step reduces the warp efficiency in comparison to the matching and contraction kernels. In the matching and contraction steps the sub-tasks are more regular and resolving their dependencies and evenly distributing the loads over the SIMD threads are more straightforward. However, parallelizing the boundary vertex movements in the refinement step is more complicated as their concurrent movements may violate the balance constraints. Consequently, the refinement sub-tasks are more irregular and the load imbalance rate is higher which results in reducing the warp execution efficiency.

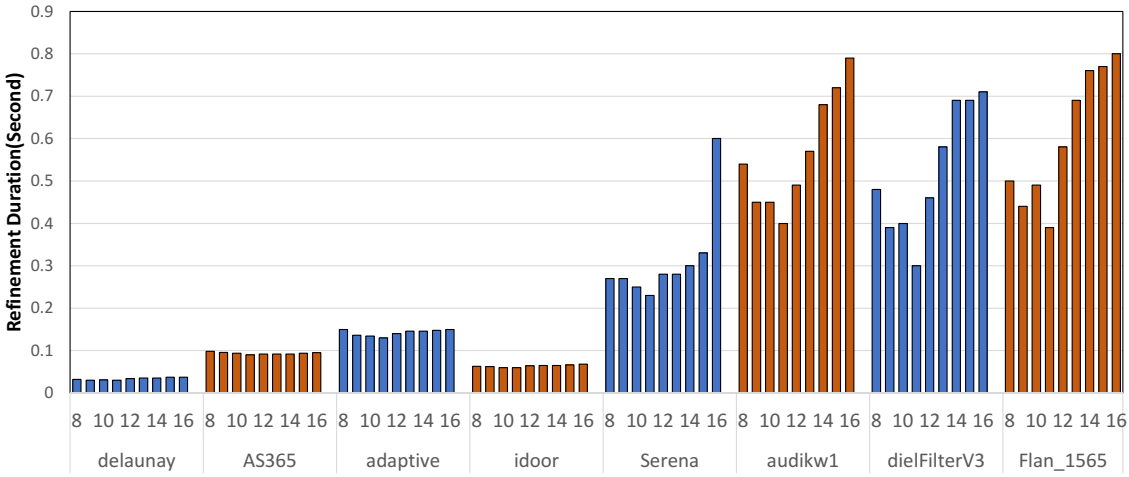



Figure 5.11: Refinement phase duration changes relative to different values of $k$ ranging from 8 to 16

### 5.3.3 Sensitivity Analysis

Now we evaluate the performance sensitivity of our GPU partitioner to 9 variants of $k$ for permutations of $k$ boundary vertex migration requests with the highest gain in the refinement phase. Figure 5.11 demonstrates the refinement phase duration changes relative to different values of $k$ ranging from 8 to 16. Based on experiments on 8 different graphs we observe that when we set $k$ to 11 and evaluate the 11 permutations of boundary vertex moves over the warps, the best refinement performance is achieved.

We also analyze the impact of coarsest graph size threshold value (`CoarsenTo`) on performance. As we mentioned before in our experiments we use the same value as mt-metis's which is set to ($\frac{Number of Vertices}{20\times(\log_2(npartitions))}$). According to Figure 5.12 we monitor the effect of reducing the value of `CoarsenTo` parameter by increasing the denominator in the formula. We change the coefficient in the denominator from 20 to 100 and measure the total partitioning time. By increasing this value, the coarsest graph size threshold reduces, therefore, more coarsening iterations are required. The best execution time is achieved when this coefficient is set to 40. However higher values cause longer delays in the coarsening and the total execution time increases subsequently. We conclude that although the value 40 gives better performance in comparison to 20, its impact

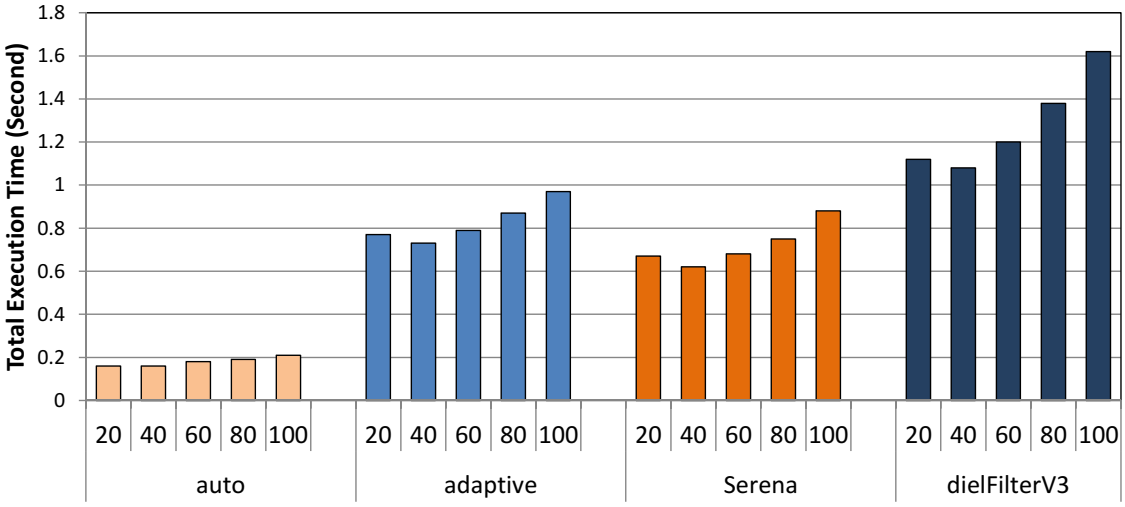



Figure 5.12: Total graph partitioning execution time changes relative to increasing the coefficient in the denominator of *CoarsenTo* formula from 20 to 100

on performance is not significant.

## 5.4 Extending the Coarsening Techniques to MST

In this section, we apply the techniques we developed in the coarsening phase of our graph partitioner to the Minimum Spanning Tree (MST) graph processing algorithm, which exhibits such characteristics as thread divergence and imbalance load distribution. The MST algorithm is a well-known graph processing algorithm that creates a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices without any cycle. MST is used in many real world applications, e.g., distributed networks, VLSI layouts, and medical imaging.

One of the well-known proposed MST algorithms is that of Borůvka [12], which is known to be suitable for parallelization. This algorithm finds the minimum weighted outgoing edge at each vertex and merges the connected vertices into supervertices (components). Since Borůvka's algorithm provides natural parallelism, many parallel MST algorithms are based on this approach.

Algorithm 5.4 shows the Borůvka's MST algorithm for an undirected graph. At

**Algorithm 5.4** Borůvka's MST algorithm.

**Input**= an undirected graph $G$
**Output**= $T$ which is a minimum spanning tree of graph $G$

```
1: procedure MST
2:     T= {C_1, C_2, C_3,..., C_k}
3:     while (number of components in T)> 1) {
4:     /*Edge Discovery Phase*/
5:     foreach component C_i in T {
6:     S = {}
7:     foreach (vertex v_j in C_i {
8:     e = minOutComponentEdge(v_j)
9:     S.add(e) }
10:    T.add(minEdge(S)) }
11:    /*Merge Phase*/
12:    mergeComponents(T) }
```

the beginning, all the vertices of graph $G$ are initialized as individual components.The algorithm iterates through the components and connects each component to another component with a minimum cost path from the component. The iterative process continues until only one component is left. During the first phase of Borůvka's algorithm (*edge discovery phase*), all the vertices find the minimum-weight crossing edge among their neighbors of the other components. This phase has a similar functionality to the heavy edge matching (HEM) process in the coarsening phase of our designed graph partitioner. We extend our load-balanced matching technique to the *edge discovery phase* of Borůvka's algorithm, to achieve a high-performance parallel implementation of MST algorithm on the GPU.

### 5.4.1 Efficient Edge Discovery

In the edge discovery phase of Borůvka's algorithm, each vertex in a graph component finds the minimum weighted edge to the minimum outgoing vertex (edge suggestion). We exploit the disjoint-set data structure to represent the components, wherein each component $C$ contains a disjoint subset of vertices; a tree data structure represents such a disjoint subset with its root vertex as the representative for $C$. Additionally,

the *union* operation merges two subsets into one by attaching the root of the tree associated with one subset into another, and the *find* operation locates the subset's representative for a constituent vertex by traversing its corresponding tree.

Each component representative maintains a shared location for storing the edge suggestions of its component vertices. A vertex's edge suggestion is compared with its representative's shared value and if such a suggestion is less expensive, the representative's content is replaced. The main task of the edge discovery phase is to determine the minimum-weight edge from each component of the graph connecting it to other components. In each component, the MST algorithm first finds the minimum-weight edge (edge suggestion) for each vertex connecting it to adjacent vertices in other components. This is followed by finding the minimum of these edge suggestions. The edge discovery phase is similar to heavy edge matching implemented in the matching phase of our GPU-based graph partitioner. The only difference is that the maximum-edge weight objective is transformed to the minimum-edge weight objective.

Hence, similar to our matching technique, in the edge discovery phase of borůvka's algorithm, each warp processes the adjacency list of 32 consecutive vertices of the graph on the fast GPU shared memory. The adjacency lists of the warp-assigned vertices are grouped in different sized segments. All the threads in a warp explore the neighbor lists of warp-assigned vertices collectively and find the minimum-edge candidate of the neighbor components in the neighbor list of each warp-assigned vertex. Then the warp threads perform an intra-warp parallel reduction over the corresponding edge weights of the neighbor list of each warp-assigned vertex to find the neighboring candidate with the minimum-weight incident edge. Following this, the first thread of each segment compares this neighboring candidate edge weight, with the current edge weight suggestion stored in the component's representative. The minimum of these two values is retained in the representative as the edge suggestion through an atomic operation (atomic "compare and swap" (`atomicCAS()`).

Figure 5.13 shows the result of edge discovery performed on a graph with three components: A, B and C. The figure also shows the MST edges discovered in each

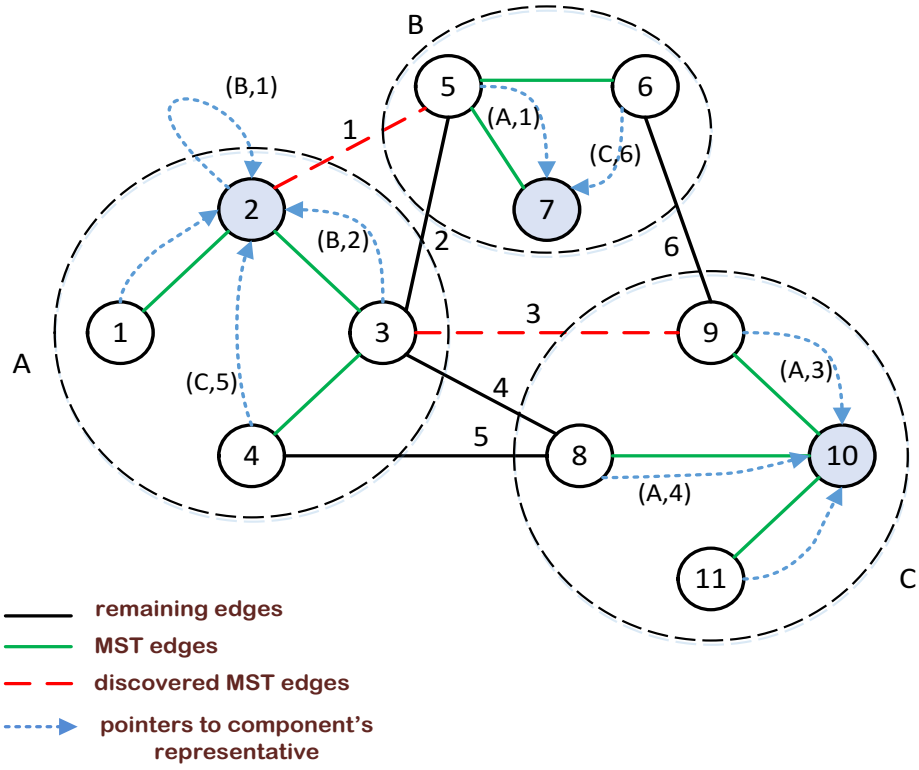



Figure 5.13: Edge discovery phase scheme.

component by retaining the least expensive edge suggestions made from the component's vertices. For example, the least expensive edge is selected among all edge suggestions made to component A's representative (i.e. vertex 2), which is illustrated by pointers from vertices 1, 2, 3 and 4 to vertex 2. The final suggestion of component A is the MST edge incident on vertex 2 (in component A) connecting to vertex 5 (in component B).

The merge phase implementation on GPU, utilizes a Software Transactional Memory (STM) [85, 86] synchronization technique to handle the race condition when the graph components are merged in a parallel way. STM offers ease of use by guaranteeing deadlock/livelock-free behavior as opposed to blocking lock-based . STM is a high abstraction level of synchronization method that simplifies the development of parallel code by allowing the programmer to identify and mark sections of the code that should be executed concurrently and atomically in an optimistic manner. The

| Graph | $\|V\|$ | $\|E\|$ | Description |
|---|---|---|---|
| USA road | 23.9M | 58M | full USA road network |
| W | 6.2M | 15M | Western USA road network |
| E | 3.5M | 8.7M | Eastern USA road network |
| FLA | 1M | 2.7M | Florida road network |
| NY | 264K | 733K | New York city road network |

Table 5.3: Input graphs used for the experiments.

underlying support for the STM replaces the memory accesses in the marked sections with transactional reads and writes. it also detects the dependence violations and inserts operations to start, commit, and retry transactions. Further details on STM-based implementation of merge phase on GPU are discussed in our paper [65].

### 5.4.2 Experimental Evaluation

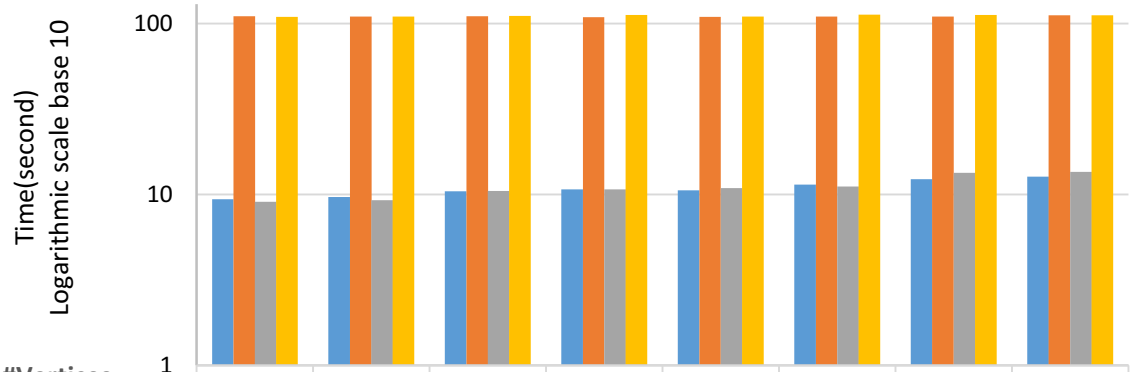
In this section, we show the results of the performance comparison of transaction-based implementation of MST on GPU against the serial and STM-based multi-core [46] implementations. We carried out the experiments using two GPUs of different strengths: a server/workstation based NVIDIA Tesla K40 with 2880 stream cores, 288GB/sec of memory bandwidth, 12GB of GDDR4 memory, and core boost clock of 875 MHz; and a desktop based Quadro K1200 with 512 steam cores, 80GB/sec of memory bandwidth, 4GB of GDDR5 memory, and core boost clock of 1124 MHz. Table 5.3 shows the sparse graphs of large and moderate sizes from USA road networks [6] used in our experiments.

Table 5.4 summarizes the execution time comparisons of our implementation on Tesla K40 GPU (CUDA) versus sequential and multi-core STM-based implementations. As the table shows, the fast edge-discovery and efficient implementation of STM-based merge phase on GPU, result in higher speedup for the larger graphs in comparison to serial and multi-core implementations. The USA graph that has the highest number of edges, shows 4.52× speedup relative to the serial MST and achieves 2.60× speedup relative to STM-based multi-core implementation.

In the next experiment, we generate random and R-MAT [5] graphs of different

| Graph | GPU-based MST Speedup vs serial | GPU-based MST Speedup vs multi-core |
|---|---|---|
| USA | 4.52 | 2.69 |
| W | 4.22 | 2.31 |
| E | 3.36 | 2.12 |
| FLA | 1.83 | 1.47 |
| NY | 1.39 | 1.28 |

Table 5.4: Speedup of STM-based GPU implementation of MST (using Tesla K40) relative to serial and multi-core STM-based implementations.




| #Vertices | 3M | 4M | 5M | 6M | 7M | 8M | 9M | 10M |
|---|---|---|---|---|---|---|---|---|
| GPU - Random | 9.39 | 9.67 | 10.46 | 10.73 | 10.57 | 11.41 | 12.29 | 12.74 |
| CPU - Random | 110.2 | 109.93 | 110.44 | 109.01 | 109.55 | 110.04 | 110.09 | 111.76 |
| GPU - R-MAT | 9.08 | 9.26 | 10.47 | 10.71 | 10.92 | 11.12 | 13.39 | 13.59 |
| CPU - R-MAT | 109.25 | 109.78 | 110.7 | 112.1 | 110 | 112.67 | 112.09 | 111.57 |

Figure 5.14: Execution time comparison of STM-based GPU implementation of MST and serial implementation over random/R-MAT graphs with 30M edges and varying number of vertices.

number of vertices ranging from 3M to 10M and with the same number of edges (30M). Figure 5.14 illustrates the execution times on CPU and GPU. Increasing the number of vertices, while keeping the number of edges fixed, increases the sparseness of the graph; however it raises the execution time mainly due to the inevitable memory latency and extra work required in the merge phases with additional vertices. However, the performance of the transaction-based GPU implementation is still significantly better than the serial implementation on CPU.

## 5.5 Conclusion

In this chapter we designed and implemented the first end-to-end high performance multilevel GPU graph partitioner. To overcome the irregularities in partitioning subtasks, we dynamically assigned them to the SIMD threads while maximizing the GPU resource utilization. In the coarsening phase we avoided the thread divergence by parallel processing of the neighbor lists of the warp assigned vertices. We also took advantage of processing power of warps in the un-coarsening phase to find the best combination of boundary vertex movements resulting in improving the partitions' quality. We employed a custom memory allocator on GPU to eliminate the allocation delay caused by the registration of allocation requests with the host during each coarsening level. The experimental results demonstrate that our partitioner is on average 1.93× faster than the the CPU-based parallel graph partitioner mt-metis. This speedup is specifically due to significant parallelism of the coarsening phase by evenly processing the graph vertices and irregular sized neighbors in the matching and contraction steps. In terms of partitioning quality, our partitioner produces comparable results with mt-metis. We also extended our efficient coarsening techniques to the edge-discovery phase of Borůvka MS algorithm, which helps in achieving a high-performance STM-based implementation of MST on GPU, that outperforms the serial and multi-core implementations.

# Chapter 6

# Conclusion and Future Work

Integrated multi-core CPU and many-core GPU systems have become mainstream. High processing potential and low cost are some of the key features that have made these environments ubiquitous. Although combining the features of both CPU and GPU is revolutionizing the future of parallel programming, fully exploiting the processing powers of all the available processors necessitates considering the difference in the programming models of CPU (MIMD) and GPU (SIMD) and modifying and redesigning the requisite parallelization methods accordingly.

In this thesis, we addressed challenges to the high performance execution of embarrassingly parallel applications on a heterogeneous CPU-GPU system. The massive parallel processing power of GPU cores makes the heterogeneous CPU-GPU platform an excellent candidate for parallelizing embarrassingly parallel applications. However, optimally distributing independent tasks over the CPU and GPUs plays an important role in minimizing execution time. Determining the ideal portion of the input tasks for each processing device is an NP-complete scheduling problem. Furthermore, the task split ratios over the processors are usually determined manually at runtime, which requires high programming effort.

We proposed an adaptive and scalable dynamic scheduling algorithm on a CPU-GPU platform. The scheduler, which operates iteratively, starts the scheduling process by an initial distribution of independent tasks based on an efficient profiling

approach. Then, during each round of execution, it employs the work stealing technique to adapt the load distribution based on the processing power of the processors. Repetition of work stealing and ratio adaptation in the subsequent rounds finally reaches the ideal distribution rates over the CPU and GPUs.

Experimental evaluation on two different embarrassingly parallel applications showed that our developed scheduler outperforms the static min-min and dynamic greedy heuristics on a system with single CPU and multiple GPUs. It also achieves similar performance in comparison to the Qilin heuristic on a single-CPU single-GPU system. Nonetheless, unlike Quilin, our dynamic scheduler is applicable to heterogeneous CPU-GPU platforms integrated with more than one GPU, and it is less sensitive to size of training data. We integrated our dynamic scheduler into a scheduling framework that hides the scheduling complexities from the user and automatically distributes the loads over the processors.

Next, we investigated task dependent applications, where the tasks and their interactions are shown by a task interaction graph. We designed and implemented a multilevel parallel partitioner for the task interaction graph on a heterogeneous CPU-GPU system. The graph partitioner implicitly accelerates the scheduling of this category of applications on either heterogeneous or homogeneous clusters by partitioning the graph into a set of computationally balanced partitions in such a way that the communication cost among the partitions is minimized. Furthermore, as the capacity to model the complex scientific problems has increased, the size and diversity of the generated graphs has also been raised. This requires designing fast and efficient graph partitioning methods. Enabling a high-performance CPU-GPU graph partitioner that outperforms the parallel distributed and multi-core methods fulfills this demand.

However, irregular and data-dependent graph partitioning sub-tasks pose multiple challenges for efficient GPU utilization, including load imbalance, non-coalesced memory accesses, and warp execution inefficiency. To overcome these challenges, we redesigned the parallel coarsening and un-coarsening methods in order to implement

a multilevel graph partitioner on a heterogeneous CPU-GPU system. We adapted our design to the parallel processing of the GPU by executing the computation-intensive parts of our partitioner on the GPU and assigning the parts with less parallelism to the CPU. We also avoided the fine-grained synchronization overhead on the GPU by designing a lock-free multilevel graph partitioner. The partitioner employs the CSR graph representation technique and reuses the GPU memory space allocated for the temporary data. Efficient employment of GPU memory enabled us to handle partitioning real-world graphs with millions number of vertices and edges.

Our experimental results showed that, on average, our CPU-GPU graph partitioner using Titan GPU performs 2.57× and 1.52× faster than serial partitioner Metis and parallel MPI-based partitioner ParMetis respectively. When using K40 GPU, our partitioner performs 2.63× faster than Metis and 1.57× faster than ParMetis. It is comparable in performance and quality of the partitions with multi-core partitioner mt-metis.

To optimize our design, we identified the performance bottlenecks of our CPU-GPU graph partitioner and discovered that the imbalanced load and thread divergence over the GPU threads in some phases of the designed graph partitioner degrades the performance. We avoided the thread divergence and balanced the load over GPU threads by dynamically assigning appropriate number of threads to process the graph vertices and irregular sized neighbors. We described an effective and methodological approach to enable a high-performance multilevel GPU-based graph partitioner. All the partitioning phases are performed on GPU with minimal CPU intervention. We also mitigated the recurrent GPU memory allocation overhead by employing a custom dynamic regional memory allocator.

The experimental results proved the superiority of our design over the multi-core partitioner and demonstrated that our partitioner is, on average, 1.93× faster than the the CPU-based parallel graph partitioner mt-metis. The high warp execution efficiency of the partitioner's main processing kernels also proved the effectiveness of our GPU-based partitioner.

Finally, we applied some of our graph partitioning techniques developed in the coarsening phase to MST, another graph processing algorithm. This resulted in a high-performance implementation of MST on a GPU, achieving 4.52× and 2.69× speedups relative to serial and multi-core implementations, respectively.

## 6.1 Future Directions

The widespread usage of heterogeneous CPU-GPU systems and the fast evolution of the GPU technology, makes efficient parallelization of various scientific applications on these platforms, an interesting topic to be more explored.

The dynamic scheduling method, that we proposed for embarrassingly parallel applications on a single-CPU multi-GPUs system, could be extended such a way it also schedules other category of applications with data-flow dependency on a heterogeneous CPU-GPU system. The scheduler can be generalized in such a way it acquires the graph with data flow dependency, resolves the dependencies, distributes the tasks over the heterogeneous devices based on an appropriate scheduling algorithm, and hides the underlying details from the user. Designing a scheduler which has "intelligence" in decision-making can be an interesting future direction of this research.

In our research, for designing the heterogeneous CPU-GPU graph partitioning methods, we assumed that the graph size is small enough to fit into the GPU's memory. However, to handle the larger graphs which do not fit in a single GPU memory, the graph partitioner could be scaled to use multiple GPUs. This provides more memory and parallel processing resources to perform the partitioning subtasks. However when the size of graph exceeds the limit of the GPU memory, arranging the communication among the GPUs during the partitioning such a way that the high throughput is maintained, is challenging. The recently introduces NVLink technology can improve the host-GPU and inter-GPU communication bandwidth up to 16 times. Employing this technology in designing a high performance graph partitioner

on multiple GPUs is an interesting topic to be investigated in the future.

In addition, the graph partitioning solutions introduced in this thesis, focus on static graphs. Therefore since the graph structure does not change, the graph partitiong process balances the load, if it only performs once. However if the graph is dynamic and its structure changes frequently in real time, new re-partitioning method is required to keep the graph balanced.

Distributed diffusion-based graph repartitioning is shown to be a promising approach for local improvement of re-partitioning of dynamic graphs. Distributed diffusion-based re-partitioning consists of initially selecting a set of seed vertices (equal to number of partitions) and iterative assignment of the remaining vertices to their closest seed vertex using a specific linear system solution. After the assignment step, each partition computes its new center for the next iteration and the graph partitions become balanced after a fix number of iterations.

Although the diffusion-based methods contain a high degree of natural parallelism and provide excellent partitioning quality in comparison to contemporary refinement methods, they are significantly slower than the popular parallel repartitioning methods. The reason is that the repeated solution of linear systems makes the repartitioning slow. This opens up a great room for exploring the high performance parallelization of diffusion-based dynamic graph partitioning on heterogeneous CPU-GPU systems.

# Bibliography


[1] Nvidia CUDA. `https://docs.nvidia.com/cuda/`. Online; accessed January-2018.

[2] B. O. F. Auer and R. H. Bisseling. A gpu algorithm for greedy graph matching, *In Facing the Multicore-Challenge II.* pages 108–119. Springer-Verlag, 2012.

[3] B. O. F. Auer and R. H. Bisseling. Graph coarsening and clustering on the GPU. *Graph Partitioning and Graph Clustering, ser. Contemporary Mathematics*, 588:223–240, 2013.

[4] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier. Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[5] D. A. Bader and K. Madduri. Gtgraph: A synthetic graph generator suite. `http://www.cse.psu.edu/~kxm85/software/GTgraph/`. Online; accessed june-2018.

[6] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, editors. *Graph Partitioning and Graph Clustering, 10th DIMACS Implementation Challenge Workshop, Georgia Institute of Technology, Atlanta, GA, USA, February 13-14, 2012. Proceedings*, volume 588 of *Contemporary Mathematics*. American Mathematical Society, 2013.

[7] S. Baxter. Moderngpu library. `https://github.com/moderngpu/moderngpu`. Online; accessed March-2018.

[8] O. Beaumont, A. Legrand, and Y. Robert. Static scheduling strategies for heterogeneous systems. In *Seventeenth International Symposium On Computer and Information Sciences*, pages 18 –22, 2002.

[9] M. Birn, V. Osipov, P. Sanders, C. Schulz, and N. Sitchinava. Efficient parallel and external matching. In *Proceedings of the 19th International Conference on Parallel Processing*, Euro-Par'13, pages 659–670, 2013.

[10] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

[11] M. Boratto, P. Alonso, C. Ramiro, and M. Barreto. Heterogeneous computational model for landform attributes representation on multicore and multi-GPU systems. *Procedia Computer Science*, 9:47–56, 2012.

[12] O. Borůvka. O jistém problému minimálním (about a certain minimal problem). *Práce Mor. Prírodoved. Spol. v Brne III*, 3, 1926.

[13] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 61(6):810–837, 2001.

[14] T. N. Bui and C. Jones. A heuristic for reducing fill-in in sparse matrix factorization. In *6th SIAM Conf.Parallel Processing for Scientific Computing*, pages 445–452, 1993.

[15] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. Recent advances in graph partitioning. *CoRR*, abs/1311.3144, 2013.

[16] B. Chapman, G. Jost, and R. Van Der Pas. *Using OpenMP: portable shared memory parallel programming.* MIT press, 2013.

[17] S. Chatterjee, M. Grossman, A. Sbîrlea, and V. Sarkar. Dynamic task parallelism with a gpu work-stealing runtime system. In *Languages and Compilers for Parallel Computing*, pages 203–217, 2013.

[18] Q. Chen, Y. Chen, Z. Huang, and M. Guo. Wats: Workload-aware task scheduling in asymmetric multi-core architectures. In *Proceeding of the 26th IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 249 –260, 2012.

[19] C. Chevalier and F. Pellegrini. Pt-Scotch: A tool for efficient parallel graph ordering. *Parallel Computing*, 34(6-8):318–331, 2008.

[20] H. J. Choi, D. H. Son, S. G. Kang, J. M. Kim, H. H. Lee, and C. H. Kim. An efficient scheduling scheme using estimated execution time for heterogeneous computing systems. *The Journal of Supercomputing*, 65(2):886–902, 2013.

[21] M. Danelutto. Adaptive task farm implementation strategies. In *proceeding of the 12th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 416 –423, 2004.

[22] T. A. Davis and Y. Hu. The university of Florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, 2011.

[23] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[24] G. F. Diamos and S. Yalamanchili. Harmony: An execution model and runtime for heterogeneous many core systems. In *Proceedings of the 17th International Symposium on High Performance Distributed Computing*, HPDC '08, pages 197–200, 2008.

[25] R. Dolbeau, S. Bihan, F. Bodin, and C. Entreprise. Hmpp$^{TM}$: A hybrid multi-core parallel programming environment.*In Proceedings of the Workshop on General Purpose Processing on Graphics Processing Units, (GPGPU)*, 2007.

[26] J. Dongarra, H. Meuer, and E. Strohmaier. Top 500 supercomputing sites. `https://www.top500.org/`. Online; accessed June-2018.

[27] J. Edmonds. Maximum matching and a polyhedron with 0, 1-vertices. *Journal of Research of the National Bureau of Standards B*, 69:125–130, 1965.

[28] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Papers on Twenty-five Years of Electronic Design Automation*, pages 241–247, 1988.

[29] B. Goodarzi, M. Burtscher, and D. Goswami. Parallel graph partitioning on a CPU-GPU architecture. In *HCW2016, as a part of 30th IEEE international Parallel and Distributed Processing Symposium (IPDPS)*, pages 58 –66, 2016.

[30] B. Goodarzi, F. Khorasani, V. Sarkar, and D. Goswami. High performance multilevel gpu graph partitioning, *submitted to Journal of Parallel and Distributed Computing (JPDC)*. 2018.

[31] M. T. Goodrich, R. Tamassia, and M. H. Goldwasser. *Data Structures and Algorithms in Java*. Wiley Publishing, 2014.

[32] R. Green, O. McColl and D. A. Bader. Gpu merge path: A GPU merging algorithm. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS)*, pages 331—-340, 2012.

[33] D. Grewe and M. F. P. O'Boyle. A static task partitioning approach for heterogeneous systems using OpenCL. In *CC*, volume 6601 of *Lecture Notes in Computer Science*, pages 286–305. Springer, 2011.

[34] K. Gupta, J. A. Stuart, and J. D. Owens. A study of persistent threads style GPU programming for GPGPU workloads. In *Proceedings of Innovative Parallel Computing*, pages 1–14, 2012.

[35] M. Halappanavar, J. Feo, O. Villa, A. Tumeo, and A. Pothen. Approximate weighted matching on emerging manycore and multithreaded architectures. *International Journal of High Performance Computing Applications*, 26(4):413–430, 2012.

[36] S. W. Hammond. Mapping unstructured grid computations to massively parallel computers. Technical report, 1992.

[37] M. Harris. Optimizing parallel reduction in CUDA (2007). `http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf`. Online; accessed January-2018.

[38] M. Harris and M. Garland. *Optimizing parallel prefix operations for the Fermi architecture*, pages 29 –38. Elsevier, 2011.

[39] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, Supercomputing '95, page 28. ACM, 1995.

[40] J. H. Her and F. Pellegrini. Efficient and scalable parallel graph partitioning. *Parallel Computing, 2010*, 2010.

[41] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[42] E. Holk, R. Newton, J. Siek, and A. Lumsdaine. Region-based memory management for gpu programming languages: Enabling rich data structures on a spartan host. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 141–155, 2014.

[43] O. H. Ibarra and C. E. Kim. Heuristic algorithms for scheduling independent tasks on nonidentical processors. *J. ACM*, 24(2):280–289, 1977.

[44] V.J Jiménez, L. Vilanoval, I. Gelado, M. Gil, G. Fursin, and N. Navarro. Predictive runtime code scheduling for heterogeneous architectures. In *High Performance Embedded Architectures and Compilers*, pages 19–33, 2009.

[45] R. Kaleem, R. Barik, T. Shpeisman, B. T. Lewis, C. Hu, and K. Pingali. Adaptive heterogeneous scheduling for integrated GPUs. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 151–162, 2014.

[46] S. Kang and D. A. Bader. An efficient transactional memory algorithm for computing minimum spanning forest of sparse graphs. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 15–24, 2009.

[47] G. Karpis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20:359–392, 1998.

[48] G. Karypis. *METIS and ParMETIS*, pages 1117–1124. Springer US, 2011.

[49] G. Karypis and V. Kumar. Parallel Multilevel K-way Partitioning Scheme for Irregular Graphs. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, 1996.

[50] G. Karypis and V. Kumar. Multilevelk-way partitioning scheme for irregular graphs. *J. Parallel Distrib. Comput.*, 48(1):96–129, 1998.

[51] B. W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell Systems Technical Journal*, 49, 1970, 1970.

[52] F. Khorasani, R. Gupta, and L. N. Bhuyan. Scalable simd-efficient graph processing on GPUs. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 39–50, 2015.

[53] 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, 2016.

[54] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 211–222, 2007.

[55] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin-Cummings Publishing Co., Inc., 1994.

[56] S. Krishnamoorthy L. Chen, O. Villa and G. R. Gao. Dynamic load balancing on single- and multi-GPU systems. In *2010 IEEE International Parallel and Distributed Processing Symposium (IPDPS10)*, pages 1–12, 2010.

[57] D. LaSalle and G. Karypis. Multi-threaded graph partitioning. In *Proceedings of 27th IEEE International Symposium on Parallel & Distributed Processing*, pages 225–236, 2013.

[58] D. Lasalle and G. Karypis. A parallel hill-climbing refinement algorithm for graph partitioning. In *Proceedings - 45th International Conference on Parallel Processing, ICPP 2016*, pages 236–241, 2016.

[59] J. Lee, M. Samadi, Y. Park, and S. Mahlke. Transparent CPU-GPU collaboration for data-parallel kernels on heterogeneous systems. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 245–256, 2013.

[60] T. Li, V. K. Narayana, and T. El-Ghazawi. A static task scheduling framework for independent tasks accelerated using a shared graphics processing unit. In

*proceeding of the 17th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, pages 88–95, 2011.

[61] J. Luitjens. Faster parallel reductions on Kepler. `https://devblogs.nvidia.com/faster-parallel-reductions-kepler/`. Online; accessed june-2018.

[62] C. K. Luk, S. Hong, and H. Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 45–55, 2009.

[63] M. Maheswaran, S. Ali, H. J Siegel, D. Hensgen, and R. F. Freund. Dynamic mapping of a class of independent tasks onto heterogeneous computing systems. *Journal of Parallel and Distributed Computing*, 59(2):107–131, 1999.

[64] F. Manne, Md Naim, H Lerring, and M Halappanavar. On stable marriages and greedy matchings. In *CSC*, pages 92–101, 2016.

[65] S. Manoochehri, B. Goodarzi, and D. Goswami. An efficient transaction-based GPU implementation of minimum spanning forest algorithm. In *2017 International Conference on High Performance Computing & Simulation (HPCS)*, pages 643 –650, 2017.

[66] T. Mattson, B. Sanders, and B. Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, 2004.

[67] C. McClanahan. History and evolution of GPU architecture. *a Survey paper*, 2010.

[68] K. Mehlhorn and G. Schäfer. Implementation of O(Nmlogn) weighted matchings in general graphs: The power of data structures. *J. Exp. Algorithmics*, 7:4, 2002.

[69] D. Merill. CUB Documentation. `http://nvlabs.github.io/cub/`. Online; accessed January-2018.

[70] G. L. Miller, S. H. Teng, and S. A. Vavasis. A unified geometric approach to graph separators. In *Proceedings of the 32Nd Annual Symposium on Foundations of Computer Science*, SFCS '91, pages 538–547, 1991.

[71] S. Mittal and J. S. Vetter. A survey of CPU-GPU heterogeneous computing techniques. *ACM Computing Survey*, 47(4):69:1–69:35, 2015.

[72] Md. Naim, F. Manne, M. Halappanavar, A. Tumeo, and J. Langguth. Optimizing approximate weighted matching on Nvidia Kepler K40. In *HiPC*, pages 105–114, 2015.

[73] S. Narof. Clang: New LLVM C front-end. `http://llvm.org/devmtg/2007-05/09-Naroff-CFE.pdf`. Online; accessed June-2018.

[74] R. Nasre, M. Burtscher, and K. 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, 2013.

[75] U. Naumann and O. Schenk. *Combinatorial Scientific Computing*. Chapman & Hall/CRC, 2012.

[76] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. GPU computing. In *Proceedings of the IEEE 96 (5)*, pages 879—899, 2008.

[77] F. Pellegrini and J. Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking*, pages 493–498, 1996.

[78] F. Pinel, B. Dorronsoro, and P. Bouvry. Solving very large instances of the scheduling of independent tasks problem on the gpu. *Journal of Parallel and Distributed Computing.*, 73(1):101–110, 2013.

[79] P. J. Plauger, M. Lee, D. Musser, and A. A. Stepanov. *C++ Standard Template Library*. Prentice Hall PTR, 2000.

[80] A. Pothen, H. D. Simon, L. Wang, and S. T. Barnard. Towards a fast implementation of spectral nested dissection. In *Proceedings Supercomputing '92*, pages 42–51, 1992.

[81] R. Preis. Linear time 1/2 -approximation algorithm for maximum weighted matching in general graphs. In *Proceedings of the 16th Annual Conference on Theoretical Aspects of Computer Science*, pages 259–269, 1999.

[82] V. T. Ravi, W. Ma, D. Chiu, and G. Agrawal. Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 137–146, 2010.

[83] J. C. Régin, M. Rezgui, and A. Malapert. Embarrassingly parallel search. In *Principles and Practice of Constraint Programming*, pages 596–610, 2013.

[84] T. R. Scogland, B. Rountree, W. Feng, and B. R. de Supinski. Heterogeneous task scheduling for accelerated OpenMP. In *proceeding of the 26th IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 144–155, 2012.

[85] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 204–213, 1995.

[86] Q. Shen, C. Sharp, W. Blewitt, G. Ushaw, and G. Morgan. Pr-stm: Priority rule based software transactions for the GPU. In *Euro-Par 2015: Parallel Processing*, pages 361–372, 2015.

[87] O. Sinnen. Task scheduling for parallel systems, *Wiley Series on Parallel and Distributed Computing*, 2007.

[88] X. Sui, D. Nguyen, M. Burtscher, and K. Pingali. Parallel graph partitioning on multicore architectures. In *Languages and Compilers for Parallel Computing*, pages 246–260, 2011.

[89] H. Tomoaki, T. Endo, and S. Matsuoka. Power-aware dynamic task scheduling for heterogeneous accelerated clusters. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–8, 2009.

[90] S. Tzeng, A. Patney, and J. D. Owens. Task management for irregular-parallel workloads on the GPU. In *Proceedings of the Conference on High Performance Graphics*, pages 29–37, 2010.

[91] C. Walshaw. A multilevel algorithm for force-directed graph drawing. In *Proceedings of the 8th International Symposium on Graph Drawing*, pages 171–182, 2001.

[92] C. Walshaw and M. Cross. *JOSTLE: parallel multilevel graph-partitioning software – an overview*, pages 27–58. Civil-Comp, 2007.

[93] L. Wang, S. Baxter, and J. D. Owens. Fast parallel suffix array on the GPU. In *Proceedings of Euro-Par 2015*, pages 573 – 587, 2015.

[94] L. Wang, Y. Z. Huang, and X. Chen. Task scheduling of parallel processing in CPU-GPU collaborative environment. In *Computer Science and Information Technology ICCSIT'08*, pages 228–232, 2008.

[95] T. Wen, Z. Wang, and M. F. P. O'Boyle. Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms. In *HiPC*, pages 1–10, 2014.

[96] N. Wilt. *CUDA Handbook: A Comprehensive Guide to GPU Programming, The*. Pearson Education, 2013.

[97] X. Yao, P. Geng, and X. Du. A task scheduling algorithm for multicore processors. In *2013 International Conference on Parallel & Distributed Computing, Applications and Technologies (PDCAT)*, pages 259 –264, 2013.

[98] D. Yu and T. G. Robertazzi. Divisible load scheduling for grid computing. In *PDCS'2003, 15th Int'l Conf. Parallel and Distributed Computing and Systems. IASTED*. Press, 2003.

[99] W. Zhang, D. Goswami, and B. Goodarzi. On the dynamic scheduling of task farm patterns on a heterogeneous CPU-GPGPU environment. In *Proceedings of the 2014 International C* Conference on Computer Science and Software Engineering*, C3S2E '14, pages 7:1–7:7, 2014.