2017

A Work-Stealing For Dynamic Workload Balancing On Cpu-Gpu Heterogeneous Computing Platforms

Esraa Abdelkareem Gad Abdelmageed
University of Mississippi

A WORK-STEALING FOR DYNAMIC WORKLOAD BALANCING ON CPU-GPU HETEROGENEOUS COMPUTING PLATFORMS

A Thesis
presented in partial fulfillment of requirements
for the degree of Masters of Science
in the Department of Computer and Information Science
The University of Mississippi

by

Esraa A. Gad

August 2017

ABSTRACT

Although many general purpose workloads have been accelerated on Graphical Processing Units (GPUs) over the last decade, other applications whose runtime behaviors are dynamic and irregular such as ones based on trees and graphs have suffered from serious workload imbalance problem caused by architectural differences between CPU and GPU processors. In this thesis, we propose a work-stealing framework to overcome such problems. Our proposed framework allows CPU and GPU threads to steal tasks from each other as well as within the same device by leveraging fine-grained data sharing and thread communication feature available on modern CPU-GPU heterogeneous systems. The implementation of BFS application on the top of our framework achieves a minimum of 8.5% performance improvement over the one with coarse-grained task partitioning scheme. It also achieves 16% performance improvement on average over its non-stealing counterpart.

ACKNOWLEDGEMENTS

My first and foremost acknowledgment goes to my family. I'd like to thank my parents who have supported me till I got here. I would like to start with my mother, my role model, who always pursued education and hard work to the last day of her life. I thank her for planting in me the love of learning and believing in my abilities. I would like to thank my father for the full freedom and trust he granted me to navigate through my self-awareness journey the way I choose. My deepest thanks goes to my sisters Summer and Ghada, and brother Mohamed for the endless support and unconditional love. I would like to thank my roommate and best friend Khadija for being my home away from home. I wouldn't have done it without her. Many thanks to my friends back home for staying close regardless of the physical distance.

I would like to express my gratitude to my research adviser, Dr. Byunghyun Jang, for encouraging me to think independently and opening for me new doors of knowledge. I am grateful for all the guidance, help and honest advice he has offered me during my Masters journey. I have been fortunate to work with him and gain professional skills that are valuable for my future career.

I am grateful to all my current lab mates David Troendle, Mason Zhao and Hossein Pourmeidani, and previous lab members Tuan Ta, Kyoshin Choo, Xiaoqi Hu and Ajay Sharma for their collaboration. I would like to especially thank Tuan Ta for all the insightful discussions and the foundation on which this thesis was built.

Last but not least, I would like to thank all the professors in the department of computer science for all the help they are always willing to offer. Special thanks to Dr. Philip J. Rhodes and Dr. Feng Wang for the valuable discussion and the constructive critique of my thesis.

TABLE OF CONTENTS

ABSTRACT ii

ACKNOWLEDGEMENTS iii

LIST OF FIGURES vi

INTRODUCTION 1

BACKGROUND 3

2.1 Fine-grained Data Sharing and Thread Communication on CPU-GPU Heterogeneous Platforms 3

2.2 Irregular GPGPU Graph Applications 6

2.3 Related Work 8

DESIGN AND IMPLEMENTATION 10

3.1 Structure of Work-stealing 10

3.2 Overall Flow 11

3.3 Communication 12

3.4 Task Distribution 13

3.5 Design Decisions 16

EXPERIMENT RESULTS 19

4.1 Experiments Setup 19

4.2 Analysis 19

4.3 Performance Evaluation 20

CONCLUSION 29

BIBLIOGRAPHY 30

VITA 34

LIST OF FIGURES

2.1 Separate memory vs. Shared memory [16]. 5
2.2 Number of tasks change over iterations 6
2.3 Example of task imbalance in BFS. 7
3.1 Illustration of tasks buffer. 11
3.2 Overall flow of the proposed work-stealing framework. 12
3.3 Popping tasks different scenarios 14
3.4 Types of dynamic task creation. 17
4.1 The number of tasks consumed as stolen or originally assigned throughout the BFS traversal of CAL. 20
4.2 Performance of GPU-only version with and without stealing normalized to CPU performance. 21
4.3 Number of tasks per iteration for different inputs and threshold guides (red for 1C_8G, yellow for 1C_16G, green for 2C_16G and blue for 2C_32G). 23
4.4 Execution time of different number of workers with and without thresholds. . 24
4.5 Performance of different number of workers with and without stealing normalized to CPU performance 25
4.6 Performance of different number of workers with CPU-GPU stealing vs. CPU-only stealing normalized to CPU performance. 26
4.7 Execution time of fine-grained work-stealing and coarse-grained partitioning. 27
4.8 Execution time of appropriate and sequentially consistent memory orders. . . 28

CHAPTER 1

INTRODUCTION

Graphical Processing Units (GPUs) have been used to accelerate general purpose applications in diverse domains. One problem that programmers face when working with GPUs is workload partitioning between CPU and GPU. In order to balance the workloads, the programmers should know the amount of work that can be done on each processor before launching the kernel. Thanks to APUs that have SVM feature, we present a framework that dynamically distributes the tasks to processors based on their availability. Our framework aims to balance workloads between CPU and GPU to maximize the hardware utilization. In this context, a balance means allowing GPU to work more when GPU-friendly tasks are present, and CPU to work more when CPU-friendly tasks are present. As a demonstration of the problem, Che et al. [7] observed that in irregular GPGPU graph applications the number of active threads varies a lot over the application runtime. This is due to the change in the amount of work (number of tasks) in each stage of the program. Which in turn leads to the underutilization of GPU SIMD hardware resources. Our proposed framework is able to detect when the SIMD is underutilized and hence allows CPU to finish the remaining work in such situation.

Work-stealing is a well known technique for dynamic workload balancing. It aims at dynamically balancing work across different processors. Although the work-stealing is very well studied in multi-threaded environments, it has not been sufficiently studied in CPU-GPU heterogeneous environments [8]. In traditional GPU-powered computing, effi-

cient collaboration between CPU and GPU has been very challenging due to the need for data transfer from one unit to the other. In addition, concurrent access of shared data was impossible. Recent features in heterogeneous systems such as a unified Shared Virtual Memory (SVM) and system wide atomics made it possible for CPU and GPU to collaborate efficiently, easily and cheaply through shared data object. SVM allows seamless data sharing between CPU and GPU. This grants both CPU and GPU fine-grained accesses and modifications of data within SVM without the need of copying data back and forth. It also allows effective communication between CPU and GPU through atomic operations on shared data. This enables CPU and GPU to work side by side without blocking each other.

One type of applications that can benefit from the proposed work-stealing framework is a graph-traversal application. Due to the dynamic, irregular and imbalanced nature of their behavior, static load balancing is not satisfactory. Different graphs have different properties such as depth, density and vertex degree. All these properties affect the dynamic behavior of workloads. Thus, this challenge makes work-stealing a good solution. This thesis makes the following contributions.

- We leverage the recently introduced fine-grained data sharing and thread communication feature to implement a work-stealing and solve dynamic load imbalance problem on CPU-GPU heterogeneous systems.
- We develop a work-stealing framework for GPGPU programmers to use for their applications.
- We study the impacts of work-stealing on the behavior of graph applications in depth, using BFS as a case study.

CHAPTER 2

BACKGROUND

This chapter presents an overview of the features in the current state-of-the-art heterogeneous platforms that are important to create a work-stealing framework. We also discuss our target benchmark application that we use to demonstrate the benefits of the proposed work-stealing framework. Lastly, we briefly discuss the related works in the literature.

2.1 Fine-grained Data Sharing and Thread Communication on CPU-GPU Heterogeneous Platforms

In traditional GPU computing (i.e., OpenCL 1.x versions), explicit data transfer between host and device is required due to their separate memory spaces. Also, concurrent modification to data was not possible during kernel execution. In contrast, the fine-grained data sharing feature introduced in recent GPU hardware and software (i.e., OpenCL 2.x) allows for effective and smooth collaboration between CPU and GPU devices; Host CPU and device GPU can share the same virtual memory address space and work on pointer-based data-structures without relaunching the kernel. Moreover, concurrent accesses to shared data enables host and device to communicate easily and cheaply during kernel execution.

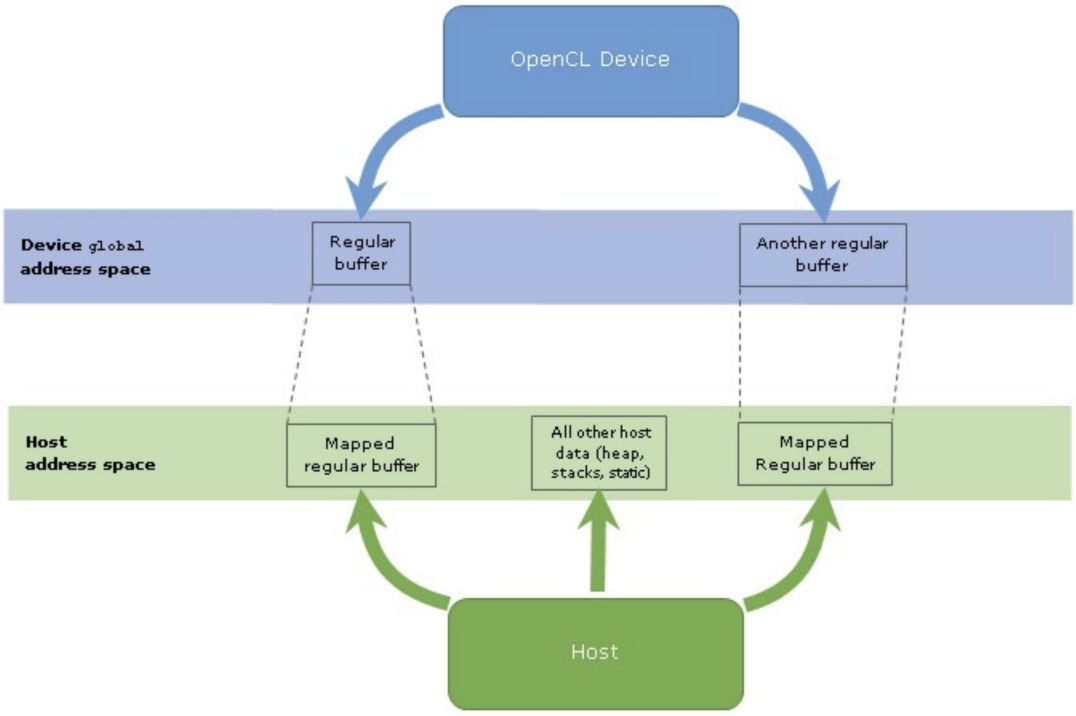
The fine-grained sharing and thread communication is supported by SVM, system-wide atomics and fence operations introduced in recent OpenCL 2.x platforms. Figure 2.1 compares the difference between OpenCL 1.x and OpenCL 2.x memory models. In the separate memory, in order for the host to read or write to data in a buffer that is in device

memory, host needs to map the whole buffer into host address space and unmap it when the read or the write is complete. In contrast, SVM allows seamless fine-grained data access and modification. In a separate address space, data synchronization can not happen during kernel execution, but in SVM, data can be synchronized during kernel execution using atomic and fence operations. All of these make it possible for CPU and GPU to co-operate efficiently and achieve best performance of both worlds.

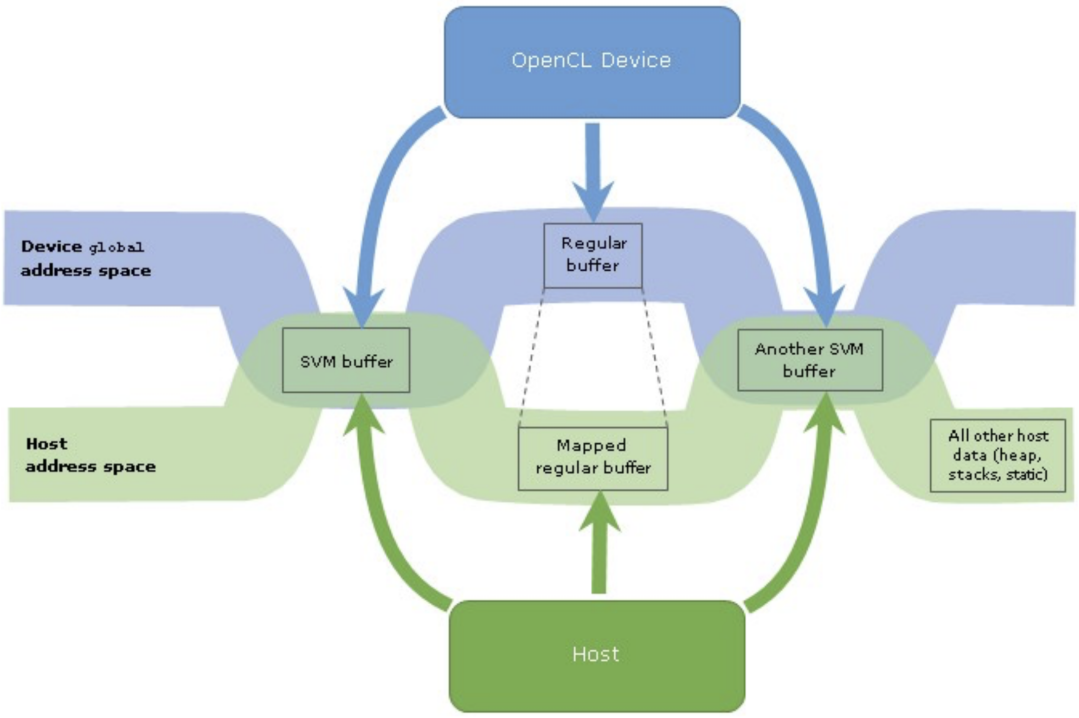
2.1.1 Atomics and Memory Consistency

Memory consistency is guaranteed for SVM allocations [16]. System-wide atomic operations are present to avoid data races during concurrent accesses. Data consistency and visibility across different units depend on memory ordering and memory scope specified with the atomic access or memory fence.

Memory ordering: Programmer can select from different memory orderings (relaxed, acquire, release, acquire_release or sequentially consistent) to achieve the desired consistency for the data accessed with an atomic operation. Atomic operations with sequential consistency force their effects to be viewed in the same order by all units within the specified memory scope. By default sequential consistency applies to atomic operations, unless another memory ordering is not explicitly specified by the programmer. This strict ordering in some cases incurs a big overhead, as it imposes inflexibility on instruction scheduling. In contrary, relaxed consistency does not force any ordering constraint, this is mainly used with counters to be concurrently incremented. Acquire consistency is used for loads and release is used for stores. Acquire and release are often used together to synchronize communication through atomic variables; one unit updates a variable and uses release to make it visible, other units need to use acquire in order to view the updated value. Aquire_release consistency does both at the same time, which is used in read-modify-write operations. The acquire consistency is for getting the most recent value of the variable, after modifying the value, the release is for making the new value visible for any unit that wants to acquire it



(a) Separate memory of CPU and GPU.



(b) Shared virtual memory.

Figure 2.1. Separate memory vs. Shared memory [16].

later.

Memory scope defines which units can view the side effect of the atomic operation. To allow the visibility of side effects within work item, work group, device or across all devices, *memory_scope_work_item*, *memory_scope_work_group*, *memory_scope_device* or *memory_scope_all_svm_devices* should be used respectively.

2.2 Irregular GPGPU Graph Applications

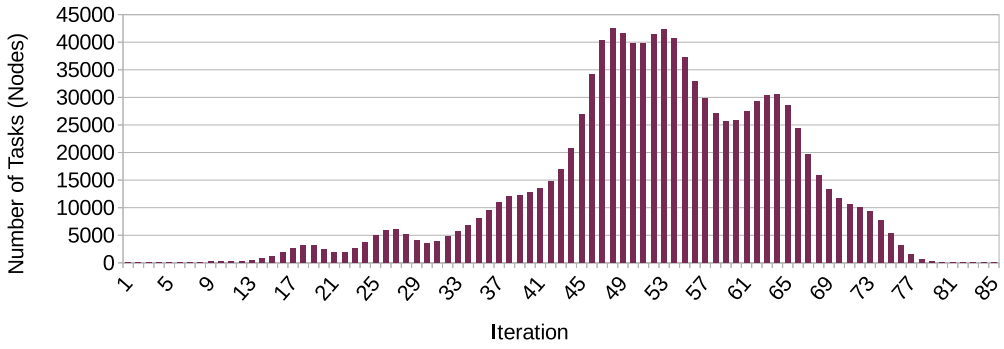
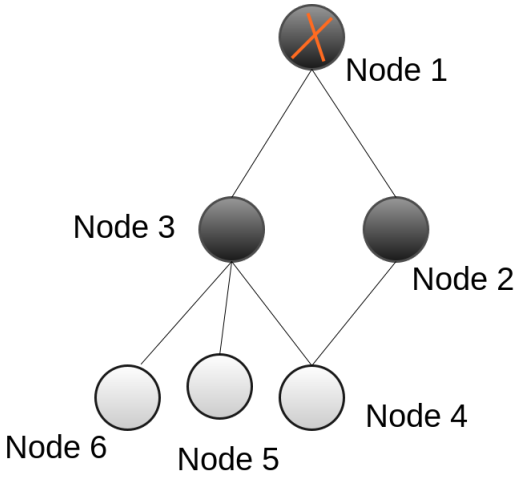


Figure 2.2. Number of tasks change over iterations

The irregularity and variation of graph structures make the utilization of GPGPU for graph algorithms very challenging. Different graph properties such as number of nodes and edges, graph density, vertex degree divergence and even the order at which the nodes are traversed can affect the overall performance of the algorithm.

There are common problems among diverse graph algorithms that hurt their performance on GPU [7]. One is load imbalance; work is unequally distributed among different threads. For example, some vertices have higher degrees (i.e., number of adjacent nodes) than others. This makes node exploration time variant and dependent on the node being traversed. If one thread is assigned all the nodes with the highest degrees, and a second thread is assigned all the ones with the lowest degrees, the second thread will complete its work first and stay idle waiting for the first thread to finish. There is no way to know the vertex degree prior to exploring it.

Figure 2.3. Example of task imbalance in BFS.



Another common problem is that the amount of parallelism in one algorithm varies dynamically across different iterations. The number of nodes to be traversed in each level of a graph is unpredictable. Figure 2.2 shows the number of nodes traversed in every iteration during BFS traversal. This makes it challenging to statically distribute work evenly between and across devices.

We aim to tackle these problems using a work-stealing mechanism. By allowing threads to steal tasks from each other, it offers an automatic way to dynamically balance the workloads. In this thesis we implement Breadth First Search (BFS) graph traversal on the top of our work-stealing framework to demonstrate the benefits.

2.2.1 Breadth First Search Algorithm

BFS is a well-known algorithm for graph traversal. In this thesis we implement the frontier based top-down version of BFS [5]. It starts by putting a source node on the frontier queue, then it explores adjacent nodes of each node on the frontier queue, and pushes the unvisited nodes to another frontier for the next iteration. After the iteration is done, it swaps the next iteration frontier with the current frontier and repeats the process. It stops only when no new nodes are pushed to the next iteration frontier, indicating that all the nodes are explored. It is illustrated in Algorithm 1.

Algorithm 1 BFS

Input: $G = (V, E)$, source s
Output: Distance from s to other vertices
Data: CF: current iteration frontier, NF: next iteration frontier

```
1: for each vertex v in V do
2:     dist[v] = ∞, color[v] = WHITE
3: end for
4: dist[s] = 0, color[s] = BLACK, iteration = 1, CF = φ, NF = φ
5: CF.enqueue(s)
6: while CF ≠ φ do
7:     for each u in CF do
8:         for each vertex v adjacent to u do
9:             if color[v] == WHITE then
10:                color[v] = BLACK,
11:                cost[v] = iteration, NF.enqueue(v)
12:            end if
13:        end for
14:    end for
15:    swap(CF, NF), NF = φ, iteration = iteration + 1
16: end while
```

Workload Imbalance in BFS: Figure 2.3 shows how the order the vertices are processed can affect the overall performance. Let's say we are at level 2, where node 2 and node 3 need to be explored by two separate workers, W1 and W2 respectively. Both of the nodes have an edge connecting to node 4. If W2 visits node 4 first before W1 visits it, W2 must push node 4 as well as 5 and 6 to the next iteration frontier. W1 checks node 4 and finds it was already visited and hence do not push anything. In contrast, if W1 reaches node 4 first, it will bear the cost of pushing it, while the W1 will only push nodes 5 and 6. This is an example of dynamic imbalance that happens due to graph structure, which can not be identified before execution.

2.3 Related Work

There are several efforts to solve the workload imbalance issue on GPUs [4], [6], [10], [26]. Chen *et al.* [10] introduced a dynamic-load balancing on GPU systems. At the time they published the paper, there did not exist any of the current features that enables fine-grained

collaboration between CPU and GPU. They invented a scheme using command events to mimic locks and synchronization communication. Their queue based solution treats host as master and device as slave, where host enqueues heterogeneous tasks concurrently to single or multiple devices. In contrast, our scheme allows for CPU and GPU to collaborate, and workload gets automatically balanced through work-stealing, instead of having an intermediate manager to balance the tasks. Arafat *et al.* [4] implemented a framework for work-stealing between different CPU-GPU clusters in a distributed system. Although the concept is similar, task stealing happens on a coarse grained level. In order to execute a task on GPU, a new kernel is launched and data need to be transfered from the host to device memory.

Work-stealing between CPU and GPU in a fine-grained sharing environment has not been sufficiently studied in the literature. Che *et al.* in [8] are the first to study a work-stealing in a heterogeneous environment with betweenness centrality as a case study.

CHAPTER 3

DESIGN AND IMPLEMENTATION

In the proposed framework where we aim to balance workloads between CPU and GPU, we treat a whole workgroup (i.e., 64 threads) as a GPU worker and each CPU parallel thread as a CPU worker. Work is divided into tasks and assigned to workers in chunks. When a GPU worker pops a chunk of tasks, every thread within that work group is mapped to one task in the chunk.

3.1 Structure of Work-stealing

We created a work-stealing structure where all workers are initially assigned an equal amount of work. Assigning work happens through queues; every worker has its own queue. Every worker has access to its queue as well as other workers' queues in a non-blocking fashion. The queues are allocated in a buffer in the shared memory, as illustrated in Figure 3.1. This buffer is divided by pointers, each of which points to the beginning of each queue. Throughout an iteration, every worker starts by consuming the tasks in its queue. Once a worker completes its tasks, it searches for other queues to see if they have any work left that it can steal. If a worker finds a chunk of tasks available, it acquires this chunk and starts working on it. A worker can return only when it has checked all other queues and made sure there is no task left.

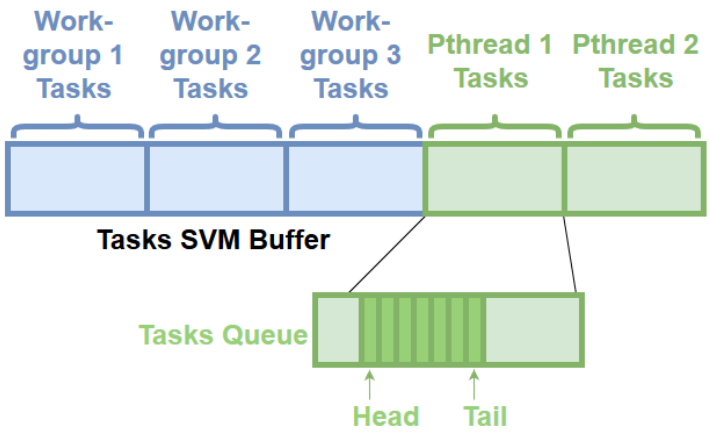


Figure 3.1. Illustration of tasks buffer.

3.2 Overall Flow

Figure 3.2 shows the overall flow of the proposed framework. The type of applications we are interested in is the one whose work is executed in iterations (discussed further in Subsection 3.5.2). Executing tasks results in the creation of new tasks, and the program reaches the final iteration when no more tasks are created. The main CPU thread is responsible for management, synchronization, as well as task distribution on workers at the beginning of every iteration. The main thread starts at step ① by setting up the work, it then chunks the work in step ② and distributes it over different queues. At step ③ it launches both GPU and CPU kernels, with the number of workers specified by the user. At this point, the workers start executing the work. The main thread does not involve with work execution or stealing. It waits at step ④ until all the workers finishes all the work in the queues. Once the workers are done, they signal back to the main thread, which in turn checks if any new tasks were created. The workers at this point would wait for a signal from the main thread to announce either the beginning of another iteration or termination of the kernel. If new tasks are created, the main thread will distribute the new tasks, and take care of any buffers that need to be re-initialized. Otherwise, the main thread will skip to the step number ⑦

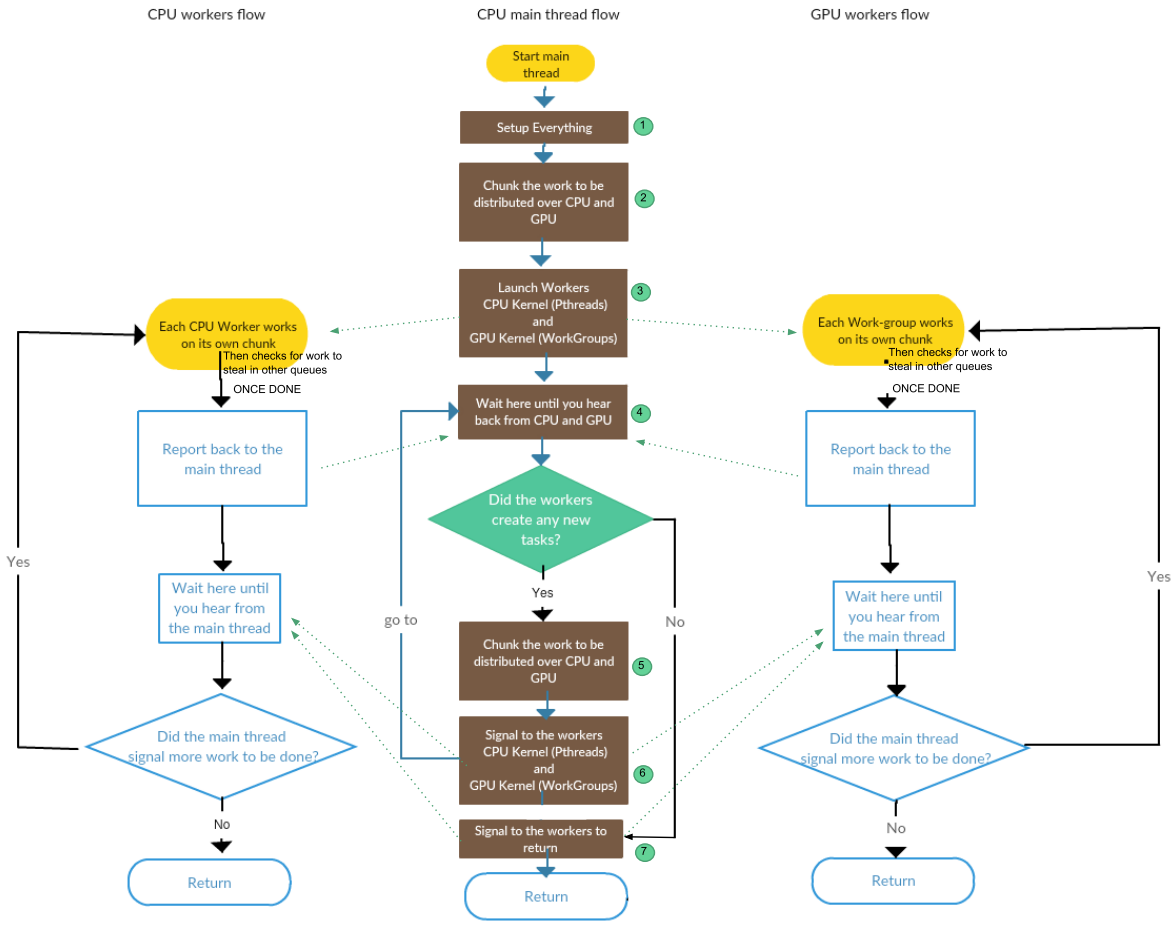


Figure 3.2. Overall flow of the proposed work-stealing framework.

to tell workers to return, and then return the results.

3.3 Communication

Communication between the main thread and workers is done through a SVM array buffer called *doneFlags*, where every worker has a corresponding element in the array and signals are sent and received through it. Since sending and receiving signals happen concurrently, atomic operations are used to ensure correctness for loading and storing buffer values. The values of *doneFlags* are initialized with 0s. When a worker is done, it updates its corresponding *doneFlag* value to 1. The main thread knows that all the workers are done when all the

doneFlags values are updated to 1. When the main thread wants to signal back to workers to continue work, it resets the *doneFlags* to 0. And when it wants to announce termination, it updates the *doneFlags* to 10.

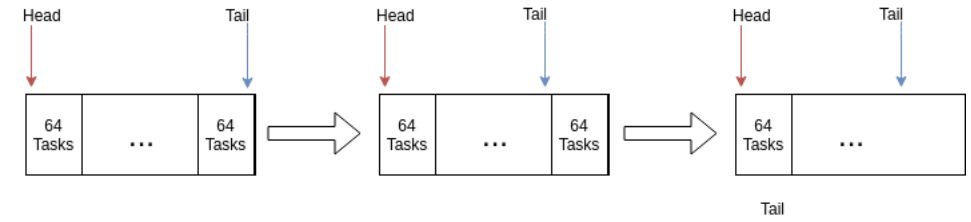
GPU workgroup as a worker: In every workgroup there is only one thread (we call it a master thread) that is responsible for communication with the main thread and managing tasks from and into the queue.

3.4 Task Distribution

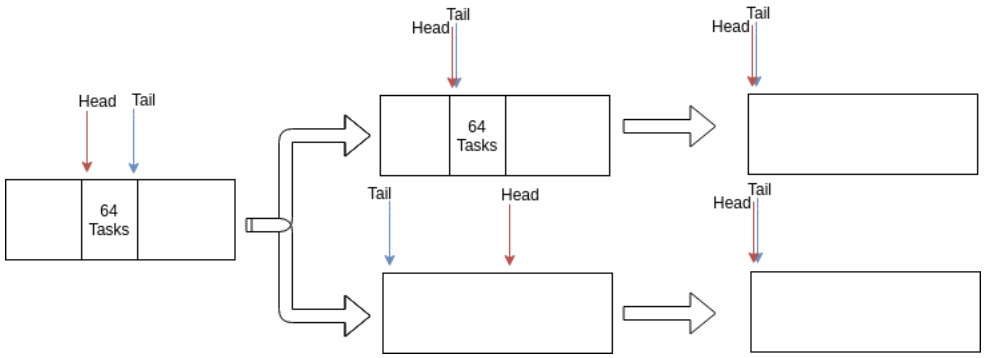
There is a single output queue where all workers push the new tasks to be executed in the next iteration at the end of every iteration. This queue is allocated in SVM to be accessible to all the workers. It is a lock-free queue, and tasks can only be pushed through its tail. During synchronization between iterations, this queue becomes the input queue that the main thread divides for workers to consume from. The main thread distributes the tasks by dividing that queue into smaller sub-queues that act as workers' queues. The queue is divided by indices that point to the beginning of each worker's queue. Before each iteration, the main queue calculates the number of tasks to be assigned to every worker. It distributes task chunks equally on the workers and assigns any remaining chunks to GPU workers first, while any remaining number of tasks whose size is less than a chunk size is assigned to a CPU worker. Task distribution happens by updating values of the indices named (*queueStartingIndex*), which denotes the location of each individual queue in the shared buffer, as well as updating *head* and *tail* values for every queue. This technique saves the cost of copying data from the big queue to individual queues. That is illustrated in Figure 3.1.

3.4.1 Workers, Queues and Operations

Workers' individual queues are implemented as lock-free Double-Ended-Queue (DEQ) [13]. Each of those queues has a head and a tail to keep track of the tasks in the queue. Head and tail variables reflects their offsets within the queue. The queue elements (i.e., tasks) can be



(a) Case 1



(b) Case 2

Figure 3.3. Popping tasks different scenarios

accessed through both head and tail using atomic operations. In our implementation, there are two main operations a worker can apply on a queue:

- **Popping** When a worker reserves tasks from its own queue, to process them.
- **Stealing** When a worker reserves tasks from another worker's queue, to process them.

One reason we chose the DEQ is to enable "pop" from one end "steal" from the other end of the queue. Task popping can only be done from the head while stealing can be done from the tail. This means the tail can only be manipulated by the queue owner, while the head can be manipulated by stealers as well as the owner. To reserve tasks, the head and tail can only be accessed and modified through atomic operations. The worker who is trying to reserve work (stealer/popper) needs to make sure that the work it is trying to reserve hasn't already been reserved by another worker during the process. In the upcoming subsection, we will discuss how it is guaranteed with a lock free algorithm.

3.4.1.1 How popping and stealing work

Popping: When popping the tasks, the master thread in a WG does the following: It starts by atomically reading the values of the head and tail. If the tail is at zero position, or head and tail are at the same position, it means the queue is empty so it returns a failure. If the head and tail are apart by a number that is not a multiple of chunk size, that means there is a fraction of task chunk in the queue. Since stealing is only allowed by a whole chunk, it is safe to reserve the chunk fraction by moving the tail backward by the number of tasks in this fraction, while the tail value is updated with an atomic store. Since this is not a chunk size, the worker should know how many tasks was it able to reserve. This is done through the pointer *nTasksToGrab*, where its value gets updated with the number of tasks.

Otherwise, the worker attempts to reserve the work for its group by atomically moving the tail's position backward by a chunk size. After doing that, there are two possible cases for the status of head and tail of the queue, illustrated in Figure 3.3.

Case 1: There is more than one available chunk of work in the queue. In this case, the master thread will directly reserve the work for its group.

Case 2: There is only one chunk of work left in the queue. The master thread needs to check whether a stealer has grabbed it during the process of moving the tail. It does that by checking whether the tail position changed or not. If it was changed, then it means that the work got stolen already. If it didn't change, it means that the work is still available so it goes to reserve it for its group. In either case, it needs to set back the head and the tail back to the zero position.

Stealing: Stealing is similar to popping except that it happens from the head. When a worker tries to steal work from another queue, it first reads the values of head and tail of that queue. If the head and tail are at the same position or the tail's value is less than the head value, or the difference between the head and the tail is less than a chunk size, this means there is no work to steal in this queue, so it returns a failure.

Otherwise, it attempts to move the head forward by an amount of chunk size. The

attempt is done using atomic compare exchange to make sure the head value wasn't updated in the middle by the queue owner or any other worker. If the compare exchange succeeds, it returns a success. Otherwise, the worker can return a failure, or it can keep retrying.

3.4.1.2 How task pushing works

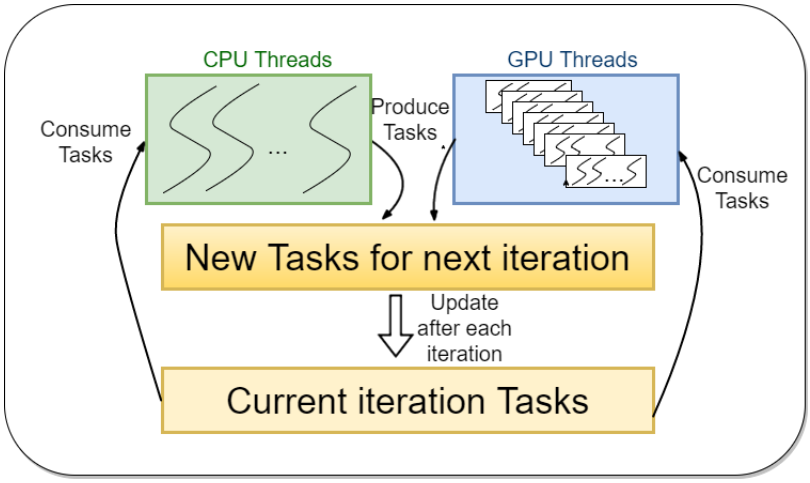
When workers push new tasks to be consumed in the next iteration, they push it to the output queue. In a GPU worker, rather than having all the threads push to the output queue immediately, master thread handles pushing for its group. We use hierarchical queues similar to the implementation in [18]. By having every thread in a workgroup count the number of new nodes it needs to push, and performing prefix sum, leveraging the optimized workgroup function *work_group_scan_exclusive_add*. After that, master thread reserves a corresponding on the queue, and then every thread pushes the tasks it created to the designated location. This helps to avoid contention on the output queue and hence serialize the accesses. During the iteration, the threads in the work group push the tasks into a local array, then at the end of the iteration, the master thread reserves space on the output queue, so that threads in its group put the tasks on that queue.

3.5 Design Decisions

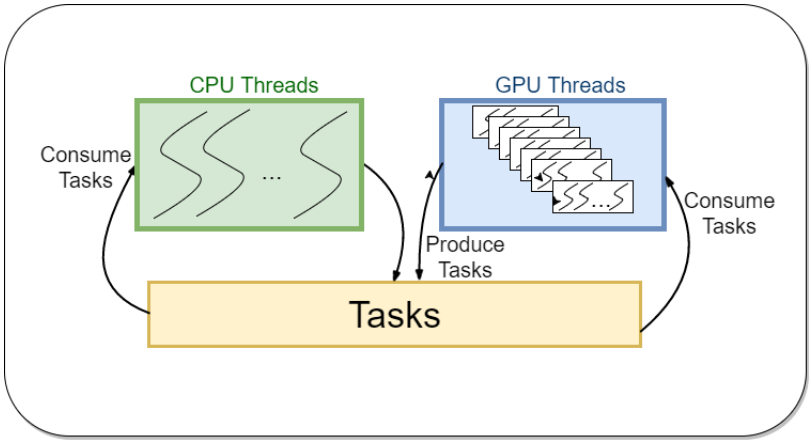
In this section, we discuss different design factors we took into consideration through the design process of work-stealing.

3.5.1 Number of Workgroups and Workgroup Size

We launch the kernel with workgroup size of a wave-front (64 threads). Since all the threads within a workgroup have to synchronize with the master thread, having more than one wave-front in a group can lead to performance loss as it imposes less freedom in thread switching. In Section 4.3, we vary the number of workgroups in our experiments and study their effects.



(a) Tasks pushed to the next iteration queue.



(b) Tasks pushed to the same queue.

Figure 3.4. Types of dynamic task creation.

3.5.2 Types of Dynamic Task Creation

The graph applications we target require the creation of new tasks during the kernel run, which means, threads are creating more tasks to be executed. There are two types of tasks as illustrated in Figure 3.4:

- Tasks that need to be executed in separate iterations: New tasks created by tasks in this iteration will be executed in the next iteration after all the current tasks are done (example: BFS nodes to be explored in the next level).
- Tasks that can be immediately executed.

The first type is of our interest in this thesis. The tasks should be pushed to a global queue that is managed by CPU. After all the work in the current work queue is done, CPU starts distributing the work on different workers, making that global queue as the new work queue and the old work queue to be the new global queue. This allows for global synchronization without the need for stopping and relaunching the kernel. We leave the second type for future work. In both cases, pushing is handled by the master thread. The master thread is responsible for computing prefix sum and passing back the location where data needs to be pushed.

CHAPTER 4

EXPERIMENT RESULTS

4.1 Experiments Setup

Environment

All the experiments are performed on real hardware using AMD Kaveri A10-7850K APU. It has four CPU cores running at the frequency of 3.7GHz and 8 GPU Compute Units running at 720 MHz. The test system runs Ubuntu 15.04 64-bit Operating System with main memory of 16 GB. The code is written and compiled using AMD SDK 3.0. we run each benchmark program 100 times after 10 warm-up runs.

Input Graphs

As input graphs for BFS traversal, we use real world maps from 9th DIMACS implementation challenge [15] in addition to seven input graphs listed in Table 4.1.

4.2 Analysis

We started out our experiments by investigating whether or not stealing happens at all. We tracked the number of tasks that are executed by the owners that they were originally assigned, and the number of tasks that are stolen from both CPU and GPU sides. Figure 4.1 illustrates those numbers for BFS traversal of CAL input graph using one CPU worker and 8 GPU workers. As expected, a non-trivial amount of stealing happened throughout the

Graph Name	#Nodes	#Edges
NYR	264,346	733,846
BAY	321,270	800,172
COL	432,893	1,048,570
FLA	1,070,376	2,712,798
NW	1,207,945	2,840,208
NE	1,524,453	3,897,636
CAL	1,890,815	4,657,742

Table 4.1. A list of different maps used as input.

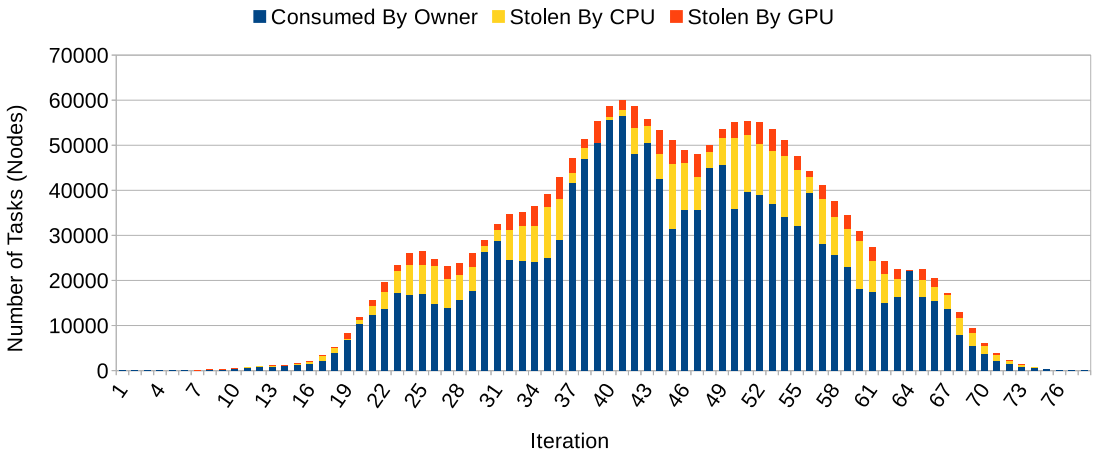


Figure 4.1. The number of tasks consumed as stolen or originally assigned throughout the BFS traversal of CAL.

computation; 24% of the tasks were consumed by stealing. That demonstrates the imbalance in workload as well as the existing opportunity to balance these loads. We also noticed that the percentage of stealing from CPU side is 70% while steals from GPU side accounts for only 30% of the total tasks stolen. This implies that a CPU worker is faster than a GPU worker in this particular application.

4.3 Performance Evaluation

We test our framework with different setups to see their effect on the performance.

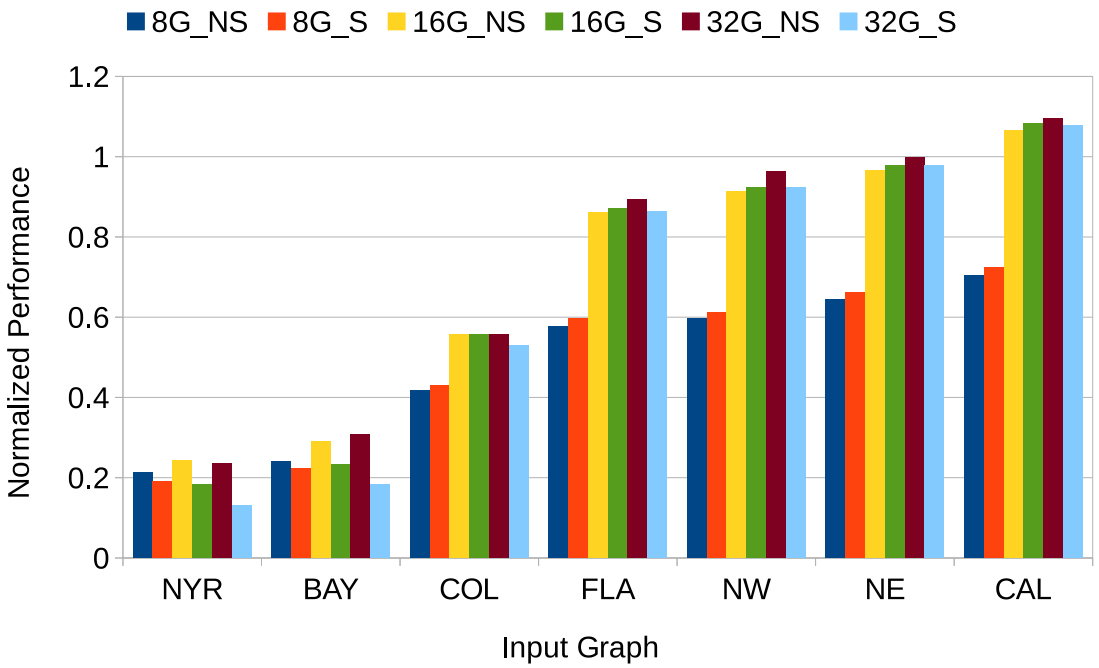


Figure 4.2. Performance of GPU-only version with and without stealing normalized to CPU performance.

4.3.1 GPU-only (Stealing vs. No stealing)

In order to understand the effect of work-stealing within GPU, we first run BFS with a GPU-only version. In this version all the tasks are assigned to GPU workers only, and a CPU worker is only allowed to process a number tasks that is less than a chunk size every iteration. We run it two times, one with stealing enabled between GPU workers, and the other without stealing. Figure 4.2 shows the normalized performance of the GPU only versions when stealing within GPU is allowed (*XG_S*), and when it is not allowed (*XG_NS*), where X is the number of GPU workers. The performance is normalized to CPU-only version performance. Overall, the use of *8G* only underutilized the GPU and hence it shows the lowest performance among all cases. We observe that in the smallest two input graphs, stealing degrades the performance, specially in case of using *32G*. However, even without stealing, GPU performance is 0.3x of the CPU-only performance. This is expected because the opportunity for stealing is very low in those graphs, since the number of available tasks

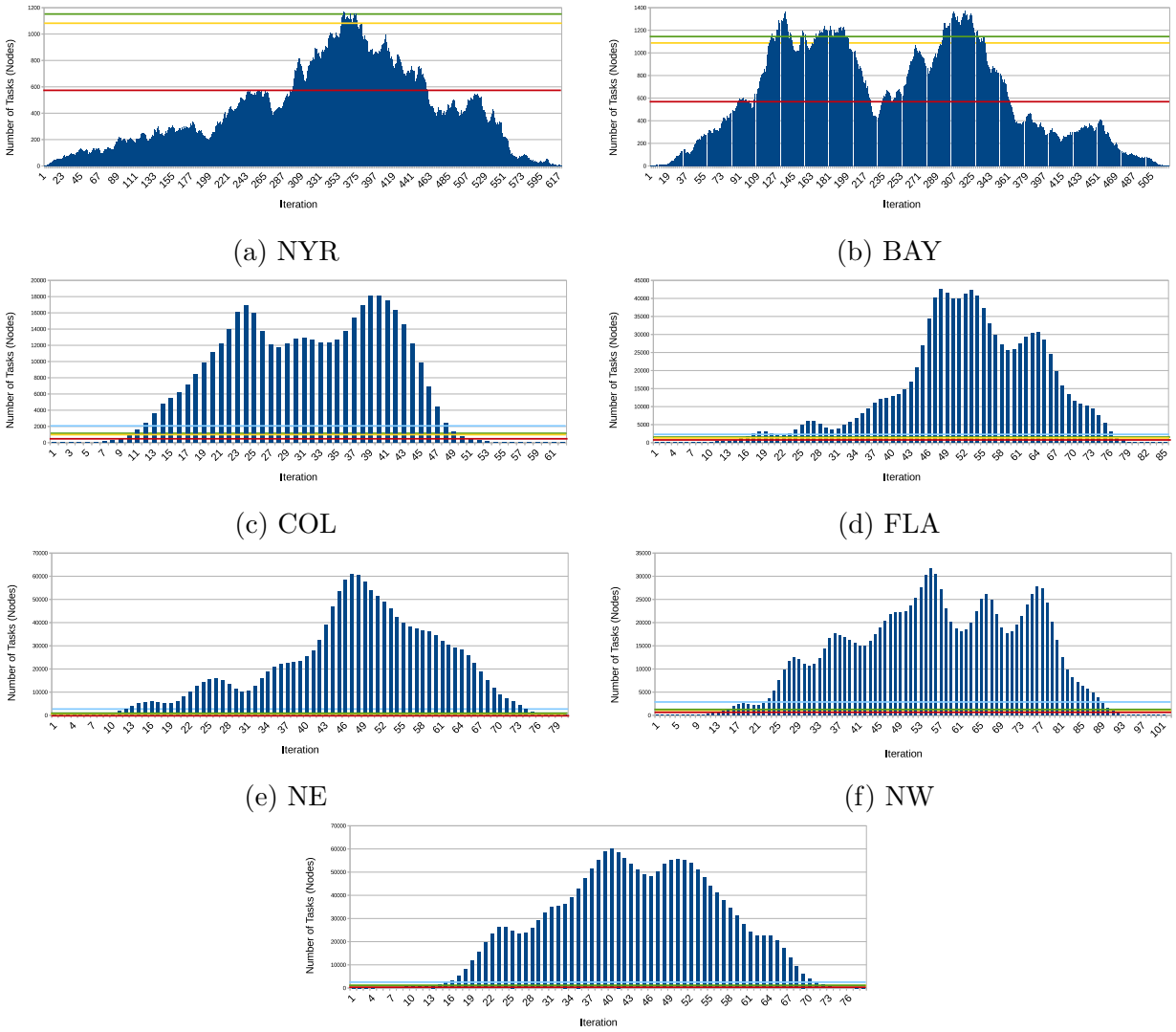
every iteration is barely equal, less than or slightly more than number tasks needed to occupy all the workers (see Figure 4.3.) When stealing is enabled, workers don't know when all the queues are empty unless they check all the queues. When this overhead cost is not compensated by the balance resulted from stealing, performance gets hurt. In bigger sized graphs, stealing yields negligible improvement in *8G* and *16G* cases, while it yields trivial degradation in *32G*. This is also expected for this type of application with low arithmetic intensity, as discussed in Section 4.2. The divergence in performance between GPU workers is not as significant as the divergence in performance between CPU and GPU workers. Hence the amount of stealing between GPU workers is not remarkable. Another evidence of the divergence between CPU and GPU in performance is that the GPU-only version outperforms the CPU-implementation only when the input graph is as big as CAL.

4.3.2 Threshold for Allowing GPU

As we observed in Section 4.3.1 when the number of tasks available is low, allowing GPU to steal hurts the performance. In this experiment we put a threshold for allowing GPU workers to consume tasks in an iteration. When the number of tasks in the iteration is at least enough for each worker to have one task chunk in its queue, GPU is enabled. The threshold is calculated by the equation:

$$threshold = NumberOfWorkers * chunkSize$$

Figure 4.3 shows the number of tasks per iteration for all seven input graphs. It also has guides for the threshold at which GPU will be enabled. The red guide is for *1C_8G* while the yellow is for *1C_16G*, the green is for *2C_16G* and the blue is for *2C_32G*. The difference in runtime before and after adding a threshold for different configurations is shown in Figure 4.4. In *1C_8G* we used one CPU worker and eight GPU workers, while in *2C_16G* we used two CPU workers and 16 GPU workers. *1C_8G_Thr* and *2C_16G_Thr* are similar to *1C_8G*



(a) NYR

(b) BAY

(c) COL

(d) FLA

(e) NE

(f) NW

(g) CAL

Figure 4.3. Number of tasks per iteration for different inputs and threshold guides (red for 1C_8G, yellow for 1C_16G, green for 2C_16G and blue for 2C_32G).

and *2C_16G* respectively, the only difference is that GPU workers get tasks only when the number of tasks is bigger than the threshold calculated by the equation.

The figure shows that putting a threshold results in a large performance gain in graphs with small inputs (NYR and BAY). This large impact is because the number of iterations that has tasks less than the threshold represent more than 50% of the total number of iterations. In addition, the total number of tasks in those iterations is non-negligible. On the other hand, applying the threshold in medium sized graphs (COL and FLA) results in

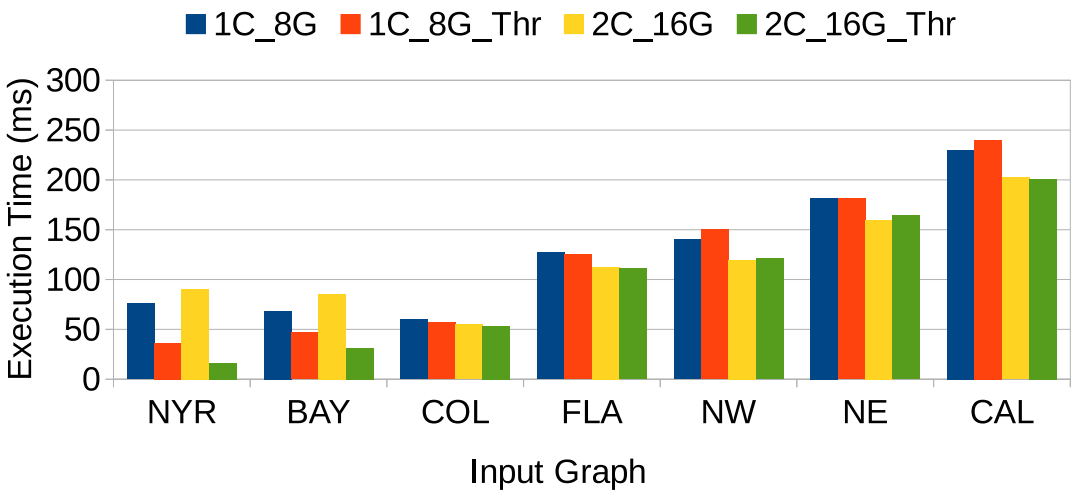


Figure 4.4. Execution time of different number of workers with and without thresholds.

a slight performance improvement, due to the low ratio between the number of iterations where the tasks were less than the threshold and the iterations where the number of tasks were higher than the threshold. For larger sized graph where the iterations that has tasks less than the threshold are very few, applying the threshold results in similar performance to the one without it.

4.3.3 CPU-GPU (Stealing vs. No stealing)

In this experiment, we compare the performance of CPU-GPU work-stealing against no stealing. Performance is normalized to CPU-only version performance. We eliminate the performance numbers for *32G* with NYR and BAY as their frontier size is always smaller than the threshold in this case, and hence GPU is never used. Overall, using two CPU workers and 16 GPU workers yields the best performance with stealing among other stealing configurations and without stealing among other configurations. This might change when we experiment on bigger sized graphs. Another observation is that, stealing always results in either a better performance or similar performance in the worst case. The only case where performance is degraded is in NYR graph is when using 1 CPU worker and 16 GPU workers (degraded by 3%). As can be seen in Figure 4.3a, the number of iterations where GPU is

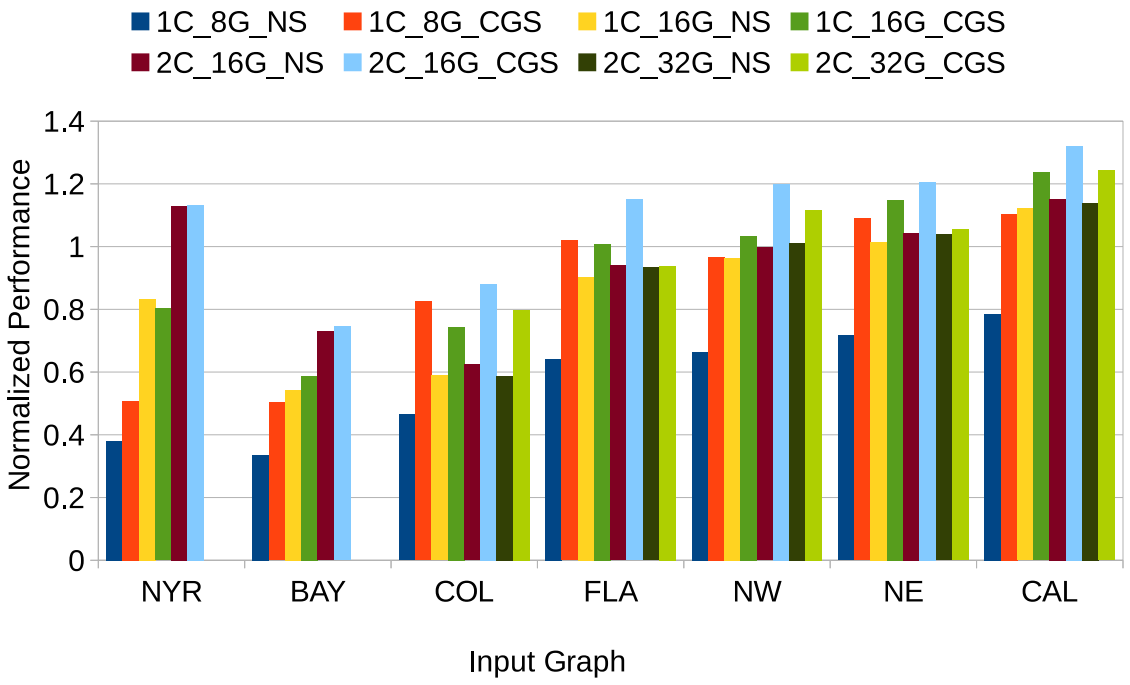


Figure 4.5. Performance of different number of workers with and without stealing normalized to CPU performance

enabled in NYR is very low, also the opportunity for stealing is lower as the number of tasks in those iterations are barely enough for assigning one chunk per worker.

One observation from this graph is that for the non-stealing cases, the performance increases as the number of workers increases until it flats out. For the smallest sized graphs (NYR and BAY) performance doubles up by using 16 GPU workers instead of 8, it improves to 1.5x again by using 2 CPU workers instead of only one. This big improvement can be explained as increasing the number of workers results in better utilization of the resources. As can be seen, stealing also improves the performance in those two graphs, but bigger sized graphs shows a different behavior in terms of improvement. In bigger sized graphs, stealing with less number of workers can perform better than or at least similar to more number of workers but without stealing. On average, stealing achieves performance improvement of 50%, 10%, 16% and 11% with *1C_8G*, *1C_16G*, *2C_16G* and *2C_32G* respectively.

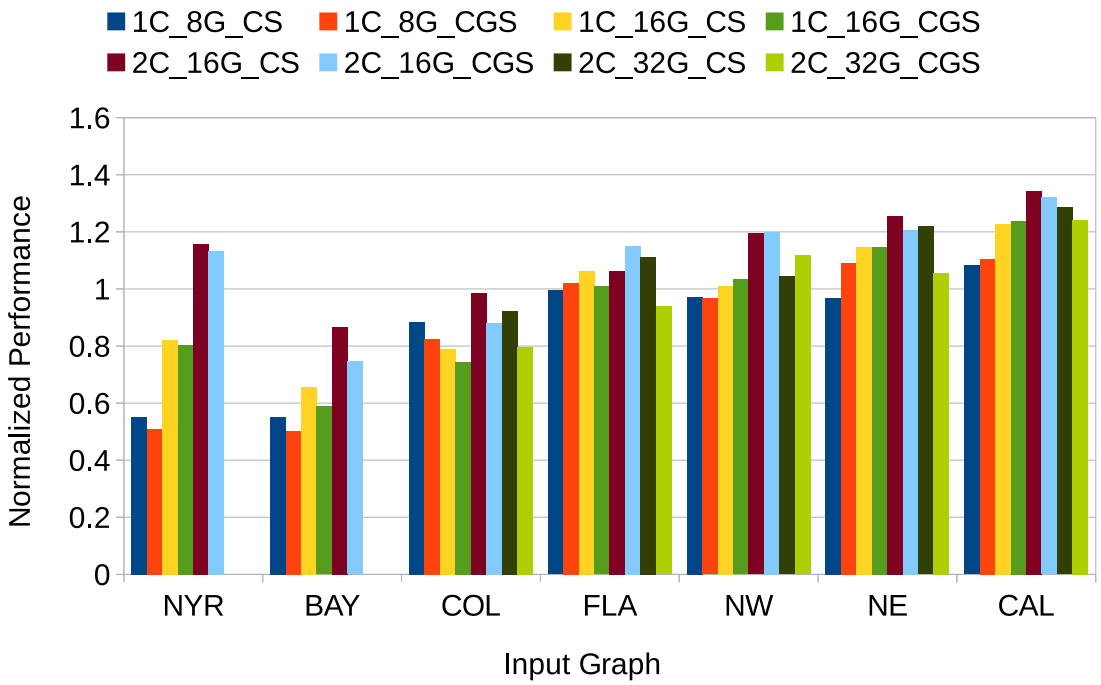


Figure 4.6. Performance of different number of workers with CPU-GPU stealing vs. CPU-only stealing normalized to CPU performance.

4.3.4 CPU-side Stealing Only

Another approach to eliminate the overhead encountered when GPU workers attempt to steal, is to disable GPU side stealing and allow only CPU side stealing. This is based on the observation that tasks stolen by CPU side are much more than tasks from CPU side. This might hold true for BFS application, but not necessarily for other graph based applications. Figure 4.6 shows normalized performance of CPU side enabled stealing as well as CPU and GPU enabled stealing while varying number of workers. Performance is normalized to CPU-only version performance. All configurations has threshold for enabling GPU as described in Section 4.3.2. We eliminate the performance numbers for *32G* with NYR and BAY for the same reason mentioned in subsection 4.3.3. In NYR, BAY and COL, GPU-stealing consistently hurts the performance. In the other four graphs, it shows a mixed performance behavior that varies between slightly better, similar and slightly worse.

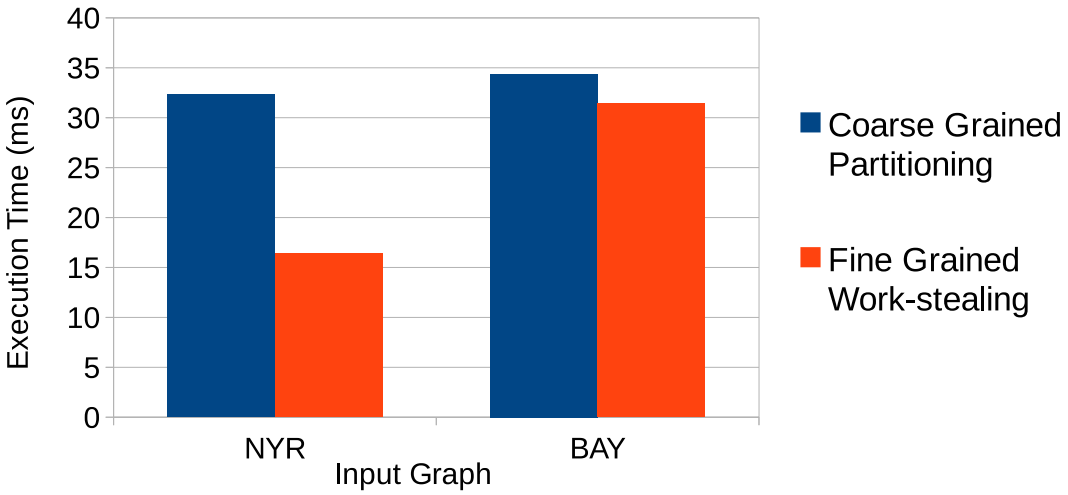


Figure 4.7. Execution time of fine-grained work-stealing and coarse-grained partitioning.

4.3.5 Work-stealing vs. Coarse-grained Task Partitioning

We compare the execution time of BFS on the top of our frame-work against the coarse-grained task partitioning version. In this version implemented in CHAI benchmark [12] partitioning happens at the beginning of every iteration, where either CPU or GPU is selected to perform the traversal. The decision is based on a predefined threshold, if the number of tasks is lower than that threshold, CPU is selected, otherwise, GPU is selected. Our version shows performance improvement of 49% and 8.5% in NYR and BAY graphs respectively. See Figure 4.7. This is because the coarse-grained method only aims at solving the variation in parallelism issue, but it does not address the imbalance within each iteration.

4.3.6 Memory Ordering

As we described earlier in the background, by default, sequential consistency applies to atomic operations unless the memory order is explicitly specified by the programmer. Sequential consistency might incur an overhead, as it leaves less freedom for instruction reordering. In order to investigate the impact of memory ordering on our work-stealing performance, we specify the appropriate memory ordering for each atomic operation accordingly. For shared data that is concurrently accessed, we use acquire for atomic loads, release for

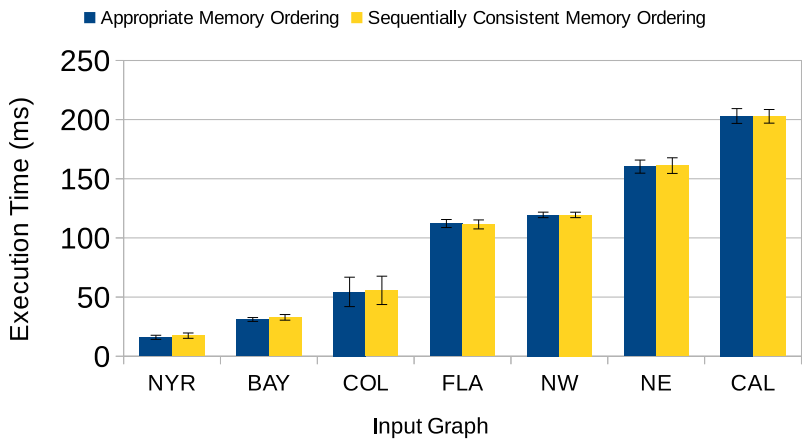


Figure 4.8. Execution time of appropriate and sequentially consistent memory orders.

stores and acquire_release ordering for compare_exchange operations. This maintains synchronization but without imposing strict memory ordering on those atomic operation. We run the traversal of all seven graphs with BFS on our framework, once with the appropriate memory ordering and once with sequential consistency for all atomic operations. Results of this experiment show execution time difference in NYR and BAY graphs by 8% and 5.5% respectively. For the other five graphs, the experiment shows a difference that falls within the margin of errors. Our explanation is that NYR and BAY have a higher number of iterations than the other five graphs by at least 5x. When the number of iterations increases, the total number of stealing attempts increases for the overall execution. This, in turn, increases the number of atomic operations, and that is why having a more strict memory order can affect the performance in those two graphs. (See Figure 4.8.)

CHAPTER 5

CONCLUSION

We have demonstrated the design of work-stealing in a CPU-GPU heterogeneous environment to achieve dynamic load balancing for irregular applications. We motivated the need for a work-stealing scheme by showing irregularity as a common characteristic of graph based applications. We have shown the feasibility of work-stealing on the current fine-grained sharing enabled heterogeneous CPU-GPU systems and the impossibility of such scheme on traditional discrete CPU-GPU systems. We used BFS as a case study to demonstrate the benefits and effects of work-stealing. We compared our scheme's performance against traditional coarse-grained task partitioning. Our scheme achieves on the minimum 8.5% performance improvement over the traditional task partitioning scheme. We anticipate that our proposed scheme can achieve beneficial results on other irregular algorithms as well. We leave that as a future work.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] AMD. Get started with codexl, 2012.

[2] AMD. Amd opencl programming user guide, 2015.

[3] AMD. Opencl optimization guide, 2015.

[4] Humayun Arafat, James Dinan, Sriram Krishnamoorthy, Pavan Balaji, and P. Sadayappan. Work stealing for gpu-accelerated parallel programs in a global address space framework. *Concurr. Comput. : Pract. Exper.*, 28(13):3637–3654, September 2016.

[5] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 12:1–12:10, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[6] Daniel Cederman and Philippas Tsigas. Dynamic load balancing using work-stealing. *GPU Computing Gems Jade Edition*, pages 485–499, 2011.

[7] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron. Pannotia: Understanding irregular gpgpu graph applications. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*, pages 185–195, Sept 2013.

[8] Shuai Che, Marc Orr, and Jonathan Gallmeier. Work stealing in a shared virtual-memory heterogeneous environment: A case study with betweenness centrality. In *Proceedings of the Computing Frontiers Conference*, CF'17, pages 164–173, New York, NY, USA, 2017. ACM.

[9] Shuai Che, Marc Orr, Gregory Rodgers, and Jonathan Gallmeier. Betweenness centrality in an hsa-enabled system. In *Proceedings of the ACM Workshop on High Performance Graph Processing*, HPGP '16, pages 35–38, New York, NY, USA, 2016. ACM.

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

[11] Benedict Gaster, Lee Howes, David R Kaeli, Perhaad Mistry, and Dana Schaa. *Heterogeneous Computing with OpenCL: Revised OpenCL 1.* Newnes, 2012.

[12] Juan Gómez-Luna, Izzat El Hajj, Victor Chang, Li-Wen Garcia-Flores, Simon Garcia de Gonzalo, Thomas Jablin, Antonio J Pena, and Wen-mei Hwu. Chai: Collaborative

heterogeneous applications for integrated-architectures. In *Performance Analysis of Systems and Software (ISPASS), 2017 IEEE International Symposium on*. IEEE, 2017.

[13] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2012.

[14] Lee Howes and Aaftab Munshi. The opencl specifications, version 2.0, 2015.

[15] Intel. 9th dimacs implementation challenge.

[16] Intel. Opencl 2.0 shared virtual memory overview, 2015.

[17] Stephen Junkins. Memory sharing and the compute architecture of intel processor graphics gen8, 2015.

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

[19] Saoni Mukherjee, Xiang Gong, Leiming Yu, Carter McCardwell, Yash Ukidave, Tuan Dao, Fanny Nina Paravecino, and David Kaeli. Exploring the features of opencl 2.0. In *Proceedings of the 3rd International Workshop on OpenCL*, page 5. ACM, 2015.

[20] Saoni Mukherjee, Yifan Sun, Paul Blinzer, Amir Kavyan Ziabari, and David Kaeli. A comprehensive performance analysis of hsa and opencl 2.0. In *Performance Analysis of Systems and Software (ISPASS), 2016 IEEE International Symposium on*, pages 183–193. IEEE, 2016.

[21] Aaftab Munshi. The opencl specification, version 1. In *Hot Chips 21 Symposium (HCS), 2009 IEEE*, pages 1–314. IEEE, 2009.

[22] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. Data-driven versus topology-driven irregular computations on gpus. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS '13, pages 463–474, Washington, DC, USA, 2013. IEEE Computer Society.

[23] L. Remis, M. J. Garzaran, R. Asenjo, and A. Navarro. Breadth-first search on heterogeneous platforms: A case of study on social networks. In *2016 28th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 118–125, Oct 2016.

[24] Michael Stoner Robert Ioffe, Sonal Sharma. Achieving performance with opencl 2.0 on intel processor graphics.

[25] Tuan Ta, David Troendle, Xiaoqi Hu, and Byunghyun Jang. Understanding the impact of fine-grained data sharing and thread communication on heterogeneous workload development. In *The 16th International Symposium on Parallel and Distributed Computing*, 2017.

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

[27] Jan Vesely, Arkaprava Basu, Mark Oskin, Gabriel H Loh, and Abhishek Bhattacharjee. Observations and opportunities in architecting shared virtual memory for heterogeneous systems. In *Performance Analysis of Systems and Software (ISPASS), 2016 IEEE International Symposium on*, pages 161–171. IEEE, 2016.

VITA

ESRAA A. GAD

esraa.abdelkreem@gmail.com

https://www.linkedin.com/in/esraa-gad

- **Education**
  - ∗ **Master's of Science**, Computer Science, University of Mississippi, August 2017
  - ∗ **Bachelor's of Science**, Computer Science, University of Mississippi, May 2015
- **Teaching Experience**
  - ∗ **Instructor** 01/2017 –05/2017

    Course: Programming for Engineering and Sciences Using MATLAB
  - ∗ **Teaching Assisstant** 08/2015 –12/2016

    Courses: Algorithm and Data Structure Analysis and Computer Science III
- **Research Experience**
  - ∗ **Graduate Student Assistant** 08/2015 –08/2017

Worked on GPU cache optimization, through cache placement and eviction policies to improve GPGPU applications performance. Ran architectural simulations and modified GPGPU-sim to accommodate a new proposed GPU cache architecture. Developed a CPU-GPU Work-Stealing framework for GPGPU applications, leveraging OpenCL 2.0 feature.

- **Skills**
    - Languages: C/C++, OpenCL 2.0, Java, Python, Shell
    - Op. Systems: Linux, Windows
    - Tools: GPGPU-sim, Git, gcc/gdb, Eclipse, Vim
- **Honors**
    - Member of Phi Kappa Phi Honor Society, 2015
    - Member of Upsilon Pi Epsilon Honor Society, 2015
    - Cultural Ambassador at NESA UGRAD Exchange Program, USA, 2014
    - Winner of INJAZ company competition for young entrepreneurs., 2011