INTELLIGENT SCHEDULING FOR SIMULTANEOUS CPU-GPU APPLICATIONS

BY

LIN CHENG

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2017

Urbana, Illinois

Adviser:

Professor Sarita V. Adve

ABSTRACT

Heterogeneous computing systems with both general purpose multicore central processing units (CPU) and specialized accelerators has emerged recently. Graphics processing unit (GPU) is the most widely used accelerator. To fully utilize such a heterogeneous system's full computing power, coordination between the two distinct devices, CPU and GPU, is necessary. Previous research has addressed this issue of partitioning the workloads between CPU and GPU from various aspects for regular applications which have high parallelism and little data dependent control flows. However, it is still not clear how irregular applications, which behave differently on different inputs, could be efficiently scheduled on such heterogeneous computing systems. Since CPUs and GPUs have different characteristics, task chunks of these irregular applications show preference, or affinity, to a particular device in heterogeneous computing systems. In this work, we show that by using the method of allocating workloads at task chunk granularity based on each chunk's device affinity, accompanied with work-stealing as the load balancing mechanism, we can achieve a performance improvement of as much as 1.5x over traditional ratio-based allocation, and up to 5x over naive GPU-only allocation on three irregular graph analytics applications.

ACKNOWLEDGMENTS

This work was supported in part by the Center for Future Architectures Research (C-FAR), one of the six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, and the National Science Foundation under grants CCF 13-02641 and CCF 16-19245.

Other members of my group have contributed to this work. Giordano Salvador developed the original work-stealing policies for GPU. Matt Sinclair helped with porting BC. Giordano Salvador and John Alsop ported PageRank and SSSP, and implemented the work queue and CPU atomics we used in this work.

I'd like to thank my advisor, Professor Sarita Adve. I really appreciate her guidance and support. I'd like to thank the members of my group for their friendship and help as well. I'd like to thank all my mentors and colleagues at National Center for Supercomputing Applications (NCSA), especially Professor Volodymyr Kindratenko and Peter Enstrom. I'd also like to thank Yisi Liu, Jiawei Zhang, Yinzhao Huang, and all my friends here at Urbana-Champaign. Finally, I'd like to thank my girlfriend who has stood by me since my freshman year, Mengqiao, for her supporting, for knowing when to set me straight, and for telling me when she thinks I'm wrong.

TABLE OF CONTENTS

CHAPTER 1. INTRODUCTION

To keep processor performance scaling, hardware designers are exploring specialization. On more power constrained platforms, such as mobile processors, it is not uncommon to see specialized accelerators that target various functions, even a particular OS library on the chip [29]. Although in desktop computers specialized hardware is not as popular as on the mobile platforms, hardware designers have started moving towards specialization by adopting CPU-GPU heterogeneous computing systems [30].

In the general purpose GPU (GPGPU) computing model [8], CPU is called the host while GPGPU is the device. Originally, the host prepares and copies the data explicitly from the system's main memory to the local memory on the device, configures the device, and offloads the work. This work offloading process is called kernel launch. The kernel is a chunk of code which executes on the device. After the device finishes executing the kernel, the host has to explicitly copy the data back from the device before using it. Since programmers have to move data back and forth between the system's main memory and GPGPU's specialized local memory explicitly, and because certain data is available to either the host or the device at a time, the collaboration between host and device is limited. The workload usually has to be fully decomposable, and synchronization across devices is only possible at kernel boundaries.

Knowing the downside of explicit data movement, the industry has moved towards unified memory. CUDA 6 [9] added unified memory that can automatically manage data movement for discrete GPGPU's; chips that integrate both CPU and GPU cores that share a cache coherent, unified memory system, such as AMD APU [10], have been produced recently. In addition to these efforts in hardware, work from Sinclair et al. [11], which promises simple and efficient fine-grained synchronization in heterogeneous systems, opens up the possibility to have applications run simultaneously on both CPU and GPU. Much previous research [1,2,3,4,5,6,7] has explored partitioning the workloads between CPU and GPU. However, to the best of our knowledge, they are mostly ratio-based. Namely, previous works usually aim for finding an ideal ratio and migrate the first certain percentage of the workload to CPU.

Multicore CPU and GPGPU each perform well on certain execution patterns. Thus task chunks of irregular applications which behave differently depending on input data, show preference, or affinity, to a particular device in heterogeneous computing systems. By allocating workload at task chunk granularity to both multicore CPU and integrated GPGPU based on each chunk's device affinity, along with work-stealing as the load balancing mechanism, we can achieve a performance improvement of up to 1.5x over ratio-based allocation, and up to 5x over naive GPU-only allocation.

CHAPTER 2. BACKGROUND

In this chapter, we first introduce background on the GPGPU execution model and the causes of divergence. Then, we provide three examples of irregular applications. We end this chapter with a brief review of prior work on how a workload is partitioned between CPU and GPU.

2.1 GPGPU, SIMT, and Divergence

GPGPU gains high performance on parallel tasks by running a large number of hardware threads simultaneously on simple cores that only have moderate single thread performance. By context switching between these threads, memory access latency is hidden [13]. Threads are divided into small groups, called warps. All threads from a single warp have to execute the same instruction. This execution model is often referred to as Single Instruction Multiple Threads (SIMT) model [14]. Because of the nature of SIMT, if there is an if-else branch, the execution of these two paths is serialized: threads in the warp which take the branch will execute instructions in the if-block while other threads are waiting, then threads that do not take the branch execute the instructions in the else-block. Such scenario is called control divergence [15]. During the execution of a particular instruction, the number of threads that are active over the number of all threads in the warp is called lane utilization [16]. Clearly, high lane utilization means better performance. Control divergence hurts lane utilization. Thus it hurts GPGPU performance. However, a general purpose multicore CPU does not suffer from control divergence. Although

CPU cores cannot run a large number of threads in parallel like a GPGPU can, CPU cores do not have to execute both paths for a branch.

Another kind of divergence that hurts GPGPU performance is memory divergence. When the threads in a warp perform a load or store, in the best scenario, all requests can be served by accessing a single cache line. However, in some cases where threads request data from different regions of the memory, their memory accesses cannot be coalesced into a single one, which causes a memory divergence. Instead of sending just one request to the memory, we could end up sending a large number of requests, and these requests have to be served one at a time. More over, memory divergence does not stall a portion of the warp, it stalls the entire warp, as all threads in the same warp have to execute the same instruction the whole warp has to wait until all accesses are finished. CPU utilizes multiple levels of caches to deal with high memory latency. Scattered memory accesses could induce high cache contention and low reuse rate. Thus, unlike control divergence, memory divergence also hurts CPU performance.

2.2 Irregular Applications

While GPGPU equipped heterogeneous computing systems evolve from standalone accelerator which requires explicit data movement towards tightly coupled CPU-GPU system with cache coherent unified memory, the applications users put on heterogeneous platforms changed as well. The very first applications have been ported to GPGPU were simple applications that have little or no synchronization requirement and high parallelism. These applications usually use GPGPUs to compute a large amount of relatively simple calculations.

Later on, researchers started to use GPGPUs to speed up more irregular applications [27]. Unlike regular ones, irregular applications usually have more data dependent control flows and more fine-grained synchronizations. Such data dependent control flows introduce control divergence, memory divergence and load imbalance, all three of which hurt GPGPU performance. Graph analytics applications, for instance, PageRank (PR), Single Source Shortest Path (SSSP), and Betweenness Centrality (BC) are good examples of irregular applications. In this study, these three applications have been chosen to study the behavior of irregular applications on a tightly coupled heterogeneous computing system. We ported the implementations of PR, SSSP, and BC from the Pannotia benchmark suite [17]. We also made minor modifications so they can run on our simulation infrastructure [28, 31, 32]. All three of them were parallelized using the vertex-centric method, in which each thread is in charge of a vertex in the graph.

SSSP from the Pannotia suite is a parallelized version of the well-known Bellman-Ford algorithm. The parallel version of SSSP, shown in Figure 2.1, was designed to use double buffering. Doing so eliminates the necessity to modify data that is potentially being used by other threads. Thus any fine-grained synchronization and atomics are not necessary. With traditional GPU-coherence, synchronization and atomic operations are very expensive [11] . The pseudocode of SSSP is straightforward with only two simple phases. In the first phase, each thread loops through the vertex's neighbor list and updates its distance accordingly. The second phase simply swaps the buffers, so that in the next iteration, threads can read the updated distance array. The execution of SSSP stops when the distances of all vertices are converged.

Algorithm 1 Bellman-Ford algorithm with double buffering

```
G(E,V)
procedure BF($G$,$source\_vertex$)
    for each vertex $v$ in $V$ that is not $source\_vertex$ do
        $distance1(v) := \infty$
        $distance2(v) := \infty$
    end for
    $distance1(source\_vertex) := 0$
    $distance2(source\_vertex) := 0$
    repeat
        for each vertex $v$ in $V$ do
            for each edge$(u, v)$ in $v$'s neighbors list with weight $w$ do
                if $distance1(u) + w < distance2(v)$ then
                    $distance2(v) := distance1(u) + w$
                end if
            end for
        end for
        swap $distance1$ and $distance2$
    until $distance1$ array converges
end procedure
```

Figure 2.1 Bellman-Ford algorithm.

PageRank [18], named after Larry Page, is also a well-known algorithm. Many parallelized PageRank algorithms exist. The version we ported from Pannotia suite is very similar to SSSP, where both of them utilize double buffering to eliminate atomic operations. The pseudocode of PageRank, shown in Figure 2.2, is also straightforward. For each vertex, a thread goes through all its neighbors and sums up its neighbors' PageRank values from the previous iteration. Then in the second phase, each vertex's PageRank value is normalized based on a fixed coefficient, namely, the damping factor.

Algorithm 2 PageRank with double buffering

```
1: G(E,V)
2: procedure PageRank($G$,$source\_vertex$)
3:     for each vertex $v$ in $V$ do
4:         $pagerank1(v) := \frac{1}{|E|}$
5:         $pagerank2(v) := 0$
6:     end for
7:     repeat
8:         for each vertex $v$ in $V$ do
9:             for each edge$(u, v)$ in $v$'s neighbor list do
10:                $pagerank2(v) := pagerank2(v) + pagerank1(u)$
11:            end for
12:        end for
13:        for each vertex $v$ in $V$ do
14:            $pagerank1(v) := \frac{0.15}{|E|} + 0.85 \times pagerank2(V)$
15:            $pagerank2(v) := 0$
16:        end for
17:    until $pagerank1$ array converges
18: end procedure
```

Figure 2.2 PageRank.

From the code of SSSP and PR, we can see that depending on the structure of the input graph, the amount of work that each thread processes differs, which introduces load imbalance from thread granularity to warp granularity. Also, at warp granularity, control divergence is potentially unavoidable.

Unlike the scenarios in PageRank and SSSP, the usage of atomics is not avoidable in BC. BC from Pannotia suite implemented Barndes' algorithm [19] which works only on unweighted graphs. Barndes algorithm for BC has two phases of execution, breadth-first search (BFS) phase, and a backtrace phase. From the pseudocode, it is not hard to see that atomic operations are necessary as an individual unvisited vertex could be neighbor to more than one vertex on the current wavefront. In terms of BFS, the wavefront is formed by newly discovered vertices. As in

BFS, a vertex is only processed once when the algorithm first encounters it. Initially, the wavefront is the source node. Then the wavefront spreads out to its neighbors, and then its neighbors' undiscovered neighbors. During the BFS phase, it is possible that two or more threads would try to set this unvisited vertex's distance to current wavefront distance plus one (line 15), which results in a race. As long as the distance of that vertex is set to the desired value in the end, this write does not determine the order of execution. However, it is an unacceptable violation of Data-Race-Free (DRF) memory model [20] . Thus, atomic operations have to be used here.

Among these three applications, PR and SSSP share the same execution pattern. Namely, neither of them forms a wavefront, every vertex is active during every iteration. There is no difference between iterations regarding amount of computation and number of memory accesses. On the other hand, in both phases of BC, a particular vertex is only active or being processed, at a certain iteration, which produces a wavefront. Based on their execution patterns, we divide irregular applications into two categories: applications that have uniform behavior across iterations, such as PR and SSSP, and applications whose behaviors change across iterations, for instance, BC.

Algorithm 3 Betweenness centrality in unweighted graphs

```
G(E,V)
$BC[v] := 0, v \in V$
procedure BC($G$)
    for each $s$ in $V$ do
        $\rho[t] := 0, t \in V$
        $\rho[s] := 0$
        $d[t] := -1, t \in V$
        $d[s] := 0$
        $dist = 0$
        //BFS phase
        repeat
            for each vertex $v$ in $V$ such that $d[v] = dist$ do
                for each edge$(v, w)$ in $v$'s neighbor list do
                    if $d[w] < 0$ then
                        $d[w] := dist + 1$
                    end if
                    if $d[w] := dist + 1$ then
                        $\rho[w] = \rho[w] + \rho[v]$
                    end if
                end for
            end for
            $dist = dist + 1$
        until no vertex $v$ has $d[v] = dist$
        $\sigma[v] := 0, v \in V$
        //Backtrace phase
        while $dist \neq 0$ do
            for each vertex $v$ in $V$ such that $d[v] = dist - 1$ do
                for each edge$(u, v)$ connects to $v$ do
                    if $d[u] = dist - 2$ then
                        $\sigma[u] := \sigma[u] + \frac{\rho[u]}{\rho[v]} \times (1 + \sigma[v])$
                    end if
                end for
                if $v \neq s$ then $BC[v] := BC[v] + \sigma[v]$
                end if
            end for
            $dist := dist - 1$
        end while
    end for
end procedure
```

Figure 2.3 Barnde algorithm for unweighted graphs.

2.3 Ratio Based CPU-GPU Collaboration

Since the dawn of CPU-GPU heterogeneous computing systems, researchers have been trying to split the workload between multicore CPU and GPGPU, though these efforts are usually limited by the necessity of explicit data movement and lack of fine-grained synchronization mechanism between host and device. To the best of our knowledge, all previous research on partitioning the workload between multicore CPU and accelerators in a heterogeneous computing system focuses on ratio based partitioning.

Early attempts, such as Qilin [2] , usually have an application-centric view. Qilin keeps a database of all applications it has seen so far. When a new application arrives, it conducts serval test runs with various partitioning ratios on both devices. Then it constructs two linear regressions, which will be used to predict each device's performance for a given input size. For a given actual workload characterized by input size, the intersection point of these two linear regressions will unveil the ideal partitioning ratio. By doing so, Qilin assumes that a certain application's performance is independent of input size, since the linear regression is constructed once for each application with a single sample input. Later works began to consider input dependent behaviors. For instance, in the work of Shen et al. [5], researchers track the running history of applications using input size as one of the parameters.

In addition to static partitioning methods which usually look at application histories or conducting test runs, dynamic partitioning based on more advanced technologies such as dynamic profiling at run time was proposed by many researchers as well [3, 4, 6, 7]. However, the splitting is still ratio-based. Kofler et al. [4] considered input induced behavior, but their mechanism partitions the workload at a very coarse granularity of 10%. Also, their study of input induced behavior focuses on the impact of the size of the input. For more regular applications, characterizing input by size might be sufficient, but for irregular ones, such as graph analytics applications, it is necessary to look at the internal structures of inputs. Farooqui et al. [6] introduced the idea of device affinity in the context of heterogeneous computing system; however, their partitioning method still aims for an ideal ratio. In addition to that, device affinity is determined by the entire input and is only used to guide load-balancing between devices. However, we believe device-affinity information at finer granularity could also lead us to a better workload partitioning mechanism.

CHAPTER 3. DEVICE AFFINITY AT CHUNK GRANULARITY

Inspired by the idea of device affinity by Farooqui et al., we decided to determine if device affinity also exists at chunk granularity (defined below). In other words, we wanted to know if different chunks in the same input would show affinity to different devices. In our experimental setup, the workload is divided into chunks. Each chunk has 256 vertices. The size 256 was determined by other students in our research group while finding optimal chunk size in the context of work-stealing. Since work-stealing will be used as the load-balancing mechanism in the later parts of this work, the optimized size 256 is used as is.

3.1 Experimental Setup

The SSSP algorithm described in chapter 2 was used to gather GPGPU runtime data with following modifications: (1) Instead of launching a grid of chunks, a single chunk of size 256 is launched at a time, and the number of cycles from kernel launch to finish is recorded. (2) Instead of letting SSSP converge, we ran a fixed 20 iterations. Since there is no wavefront in the Bellman-Ford algorithm, during each iteration, a particular thread that processes a specific vertex, accesses the same amount of data and does almost the same amount of computation. There should be little variance across iterations, so there is no need to run until convergence. We ran a serial implementation of the original Bellman-Ford algorithm with a single thread on a multicore CPU to gather CPU run time data. The starting time and finishing time for processing every 256 vertices is recorded. We ran a task chunk on a single thread on CPU due to the limitation of our simulation infrastructure.

Both CPU code and GPGPU code were run on the simulation infrastructure in our research group [11]. Specifically, the simulation infrastructure is built out of GEMS [24] and GPGPU-sim [22] , and it simulates a heterogeneous computing system with 16 general purpose cores, 15 GPU SM's, and a cache coherent, unified memory. Each core/SM has its private L1 cache, and all cores and SMs share a single banked L2 cache. More details are described in Chapter 5.

3.2 Results

In this study of determining if device affinity exists at task chunk granularity, we ran four graphs ported from the UF Sparse Matrix Collection [26]: bcsstk13, bcsstm38, rajat27, and dictionary28. Table 3.1 summarizes the graphs. Figures 3.1 to 3.4 show the speedup of processing a specific chunk in parallel with one GPGPU SM over a single thread on a CPU core. Our results show that these four graphs can be divided into three categories: (1) Device affinity is observed, but one device is always consistently better than the other with little variance; (2) though one device is always better than the other, the speedup is significantly different across different chunks, and some chunks can be only sped up moderately; (3) different chunks show strong affinity to different devices.

Graph	# Vertices	# Edges
bcsstk13	2003	42943
bcsstm38	8032	7842
rajat27	20640	99777
dictionary28	52652	89038

Table 3.1 Graphs used in device affinity study.

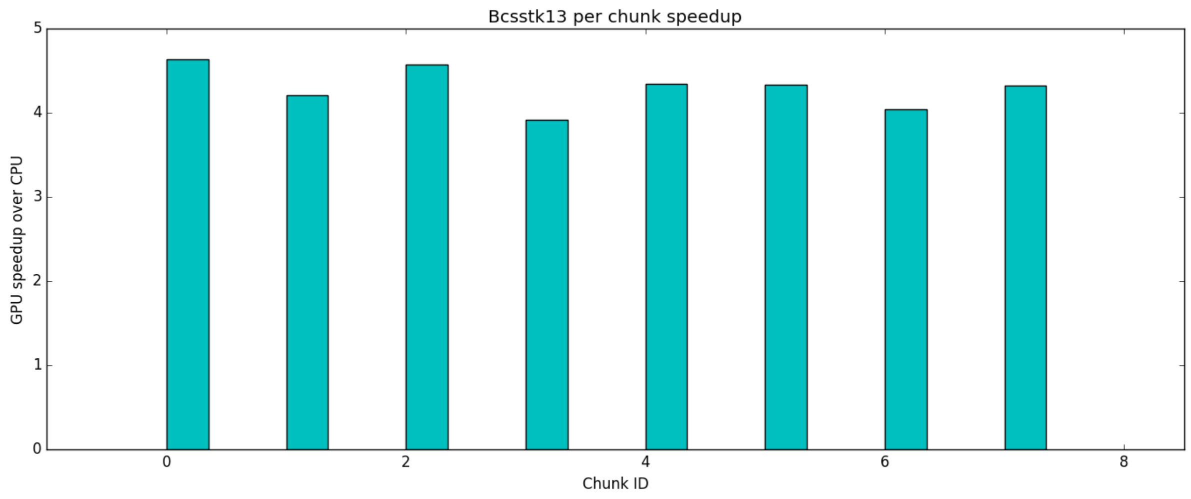


Figure 3.1 Bcsstk13 per chunk GPU speedup over CPU.

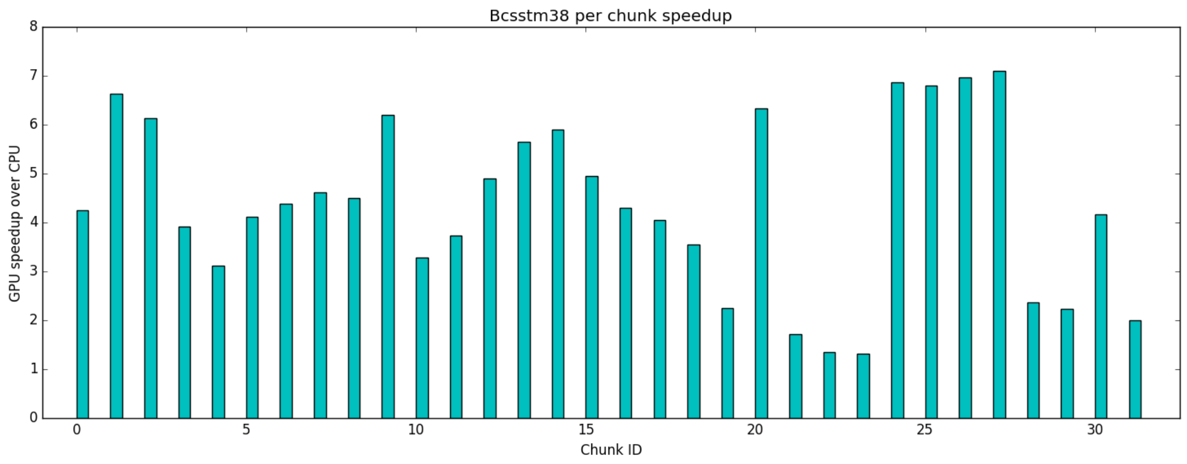


Figure 3.2 Bcsstm38 per chunk GPU speedup over CPU.

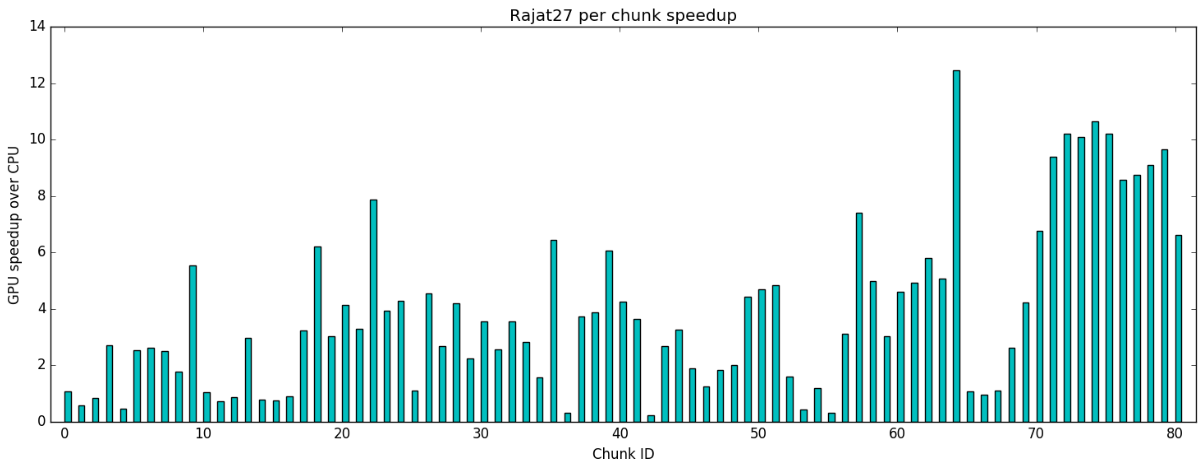


Figure 3.3 Rajat27 per chunk GPU speedup over CPU.

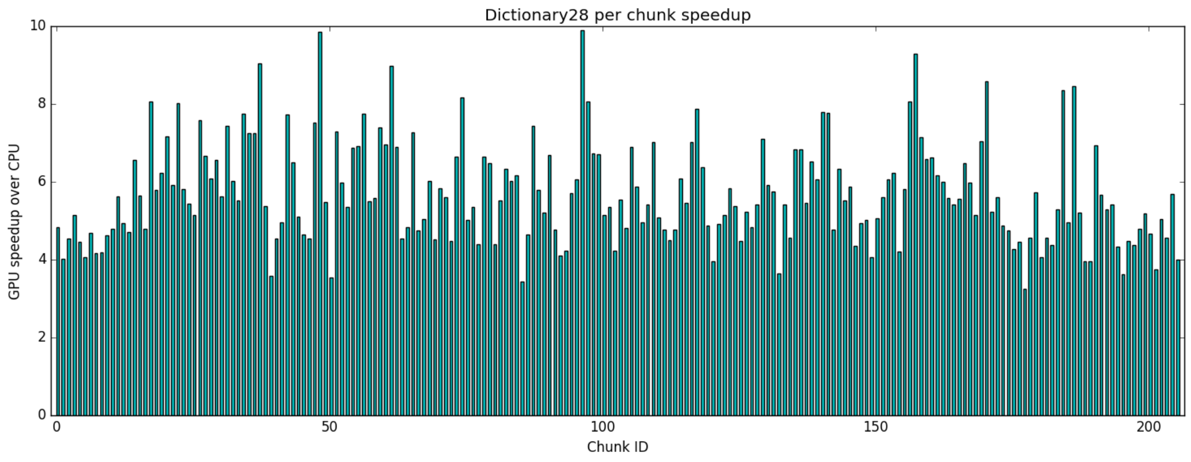


Figure 3.4 Dictionary28 per chunk GPU speedup over CPU.

For graphs in category one, where all chunks have a high affinity for a specific device, the goal of CPU-GPU collaboration is to minimize total execution time by balancing the processing time on both devices. bcsstk13, where performance on GPGPU is consistently better than on CPU with little variance, falls into this category. bcsstm38 and dictionary28 fall into category two, where though for every chunk in both graphs GPGPU achieves a better performance, certain chunks have relatively low performance degradation on the CPU. For instance, in bcsstm38, there are chunks that achieves less than 1.75x speedup, while more than half the chunks achieve a speedup of 4x. On the other hand, rajat27 is a more interesting graph, as there are both chunks that favor CPU and chunks that favor GPGPU. Note that bcsstk13, which falls into category one, is denser than the other three graphs.

Although all graphs show a certain amount of device affinity, only the ones in the latter two categories are interesting. It is because in the case where GPGPU is consistently and considerably faster than CPU, putting a portion of workload on CPU may hurt performance due to overheads like implicit data movement and communication, and vice versa. However, graphs which have chunks that have similar performance on both devices, and graphs in which some chunks strongly favor a certain device, opened up the opportunities to scheduling chunks to devices intelligently. In addition to that, these results unveiled one of the potential downsides of migrating a certain predetermined ratio of the workload onto CPU in a heterogeneous computing system. Figures 3.2 through 3.4 indicate that chunks with low GPU speedup can appear anywhere. For instance, in Bcsstm38 (figure 3.2), the chunks that are mostly worthy to migrate to CPU, are chunks around chunk ID 23. However, if a ratio based splitting mechanism has been used, the first few of chunks which have considerably better performance on GPU would be migrated instead of these chunks with low GPU speedup.

CHAPTER 4. INTELLIGENT WORK ALLOCATION

Task chunk level device affinity and the existence of category two and category three graphs, provide us the opportunity to distribute the workload among multicore CPU and GPGPU intelligently. Instead of describing a practical mechanism to determine a specific task chunk's device affinity, the goal of this work is to motivate such a study by showing the power of intelligently scheduling with an oracle scheduler. Similarly, instead of comparing our oracle scheduler to an actual ratio-based partitioning mechanism in later chapters, we used sweeping to find out the ideal splitting ratio. Methods described by previous research should ideally produce the same ratio, assuming they also migrate the first certain percentage of the workload, even though state of the art methods are more efficient and practical than naively sweeping.

4.1 Rank Based Oracle Scheduler

As we have shown in Chapter 3, for graphs in category two and three, GPU speedup over CPU varies significantly from chunk to chunk. When dealing with category three graphs, which have a portion of chunks that favor a certain device, scheduling these chunks to that device first is necessary. For category two graphs, in which all chunks have better performance on a certain device, but the amount of speedup varies, scheduling chunks with the least performance degradation after the migration is a reasonable choice. In the context of CPU-GPU heterogeneous computing systems, we implemented a simple rank based oracle scheduler: after measuring the device affinity by calculating GPU speedup over CPU like what was done in Chapter 3, chunks are ranked based on their GPU speedup in reverse order. By doing so, chunks that strongly favor CPU would have the highest rank, followed by chunks that suffer from the least performance degradation after migration from GPU to CPU. The rest are chunks that have considerably large speedup and definitely should be put on the GPU. At run time, the oracle scheduler greedily allocates the chunks with the highest ranks (until some threshold) to the multicore CPU. The chunks below this threshold are allocated to the GPU. The rank threshold for allocation is also determined in an oracular fashion. Specifically, in our study, we did a sweep over several rank thresholds to determine which would be best for each application and used that threshold.

4.2 Load Balancing Through Device Affinity Aware Work-Stealing

Although an Oracle scheduler based on the ranking described above should be able to split the workload in an optimal way, one needs to keep in mind that the data used to calculate the ranking was gathered by running a single chunk at a time using only one core or one SM. During application execution time, when all chunks are being processed by many cores and SMs, interference is unavoidable. To resolve such inaccuracy and potential load imbalance across chunks, work-stealing is applied as the load-balancing mechanism. We applied another student's work on GPU work-stealing [28], and made minor modifications to add the ability of stealing between CPU and GPU. Here Farooqui et al.'s [6] idea of device affinity aware work-stealing was adopted as well, though they introduced this idea for a slightly different purpose. Farooqui et al. measured device affinity at input graph granularity. They would determine if a certain input is more suitable for GPU or CPU. Their main concern here is that since CPU has a higher frequency than GPU, it may steal faster than GPU and steal a lot of workloads that do not favor CPU. Device affinity aware stealing is essential to this study because for category three graphs, stealing from another device would mostly likely result in lower performance, as chunks are assigned to the device they strongly favor. Thus, unless there is no work left in any core in the CPU, a CPU core is not allowed to steal from GPU, and vice versa.

In this study we examined three stealing policies: steal-successor-one, steal-successor-three and steal-all [28]. As their names indicate, under steal-successor-one and steal-successor-three policies, a core/SM attempts to steal from its neighbors, and gives up trying after falling one or three times based on the specific policy. However, in these two scenarios, a core/SM gives up before trying everyone else. Thus there is no way to determine if the device is empty, and stealing across devices is not allowed in steal-successor-one and steal-successor-three policies. On the other hand, a core/SM under the steal-all policy would first attempt to steal from all cores/SM's in the same device. If all attempts in the same device failed, it would attempt to steal from another device.

Policy	Failed attempts before give up	Allow steal across devices
Steal-successor-one	1	No
Steal-successor-three	3	No
Steal-all	(#CPU cores + #GPU cores) -1	Yes

Table 4.1 Stealing policies.

CHAPTER 5. METHODOLOGY

All studies in this work were conducted on a simulated CPU-GPU heterogeneous system with cache coherent unified memory [11, 12]. This simulator was built out of several popular architecture simulators. The Simics [21] full system simulator models a multicore CPU. Here we adopted a simple in-order core model, which assumes that all non-memory access operations take only one cycle. GPU was modeled by GPGPU-Sim [22], which is similar to an NVIDIA GTX 480. The whole system has a 16-core CPU and 15 GPU SMs. The multicore CPU and 15 SM's form a 4 x 4 mesh network, which is modeled by Garnet [23]. The memory system in this simulator is modeled by Wisconsin GEMS memory timing simulator [24]. In our simulation infrastructure, each core/SM has its private L1 cache, and all cores/SMs share a single banked L2 cache. We use DeNovo coherence protocol [25], and a data-race-free memory consistency model [20]. In this study, we only utilized 5 cores to keep synchronization overhead low.

CPU Frequency	2 GHz
# CPU Cores	16 (5 used in case studies)
GPU Frequency	700 MHz
# GPU SM’s	15
L1 Size	32 KB L1D, 32 KB L1I
L1 Hit Latency	1 cycle
Remote L1 Hit Latency	35 - 83 cycles
L2 Size	4 MB
L2 Hit Latency	29 - 61 cycles
Memory Latency	197 - 261 cycles

Table 5.1 Simulator parameters.

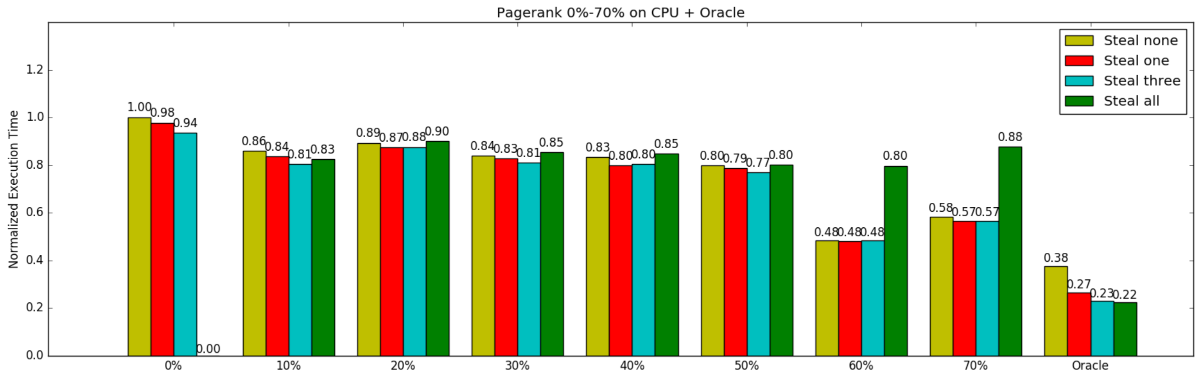
CHAPTER 6. CASE STUDIES

We implemented the oracle schedulers for three applications, PageRank (PR), Single source shortest path (SSSP), and Betweenness centrality (BC), based on the mechanism described before in Chapter 4. In Chapter 2 we divided irregular applications into two categories: applications that have uniform behavior across iterations, and applications whose behaviors change across iterations. We will examine these two categories separately.

6.1 PR and SSSP: Applications with Uniform Behavior Across Iterations

As we haven seen in Chapter 2, the algorithms for PR and SSSP have many similarities. The most noticeable difference between these two applications is that SSSP has one more branch to find the nearest neighbor while PR simply sums up a vertex's neighbors' values. Since these two applications are similar, we combined them into one case study. In this case study, dictionary28 and rajat27 were chosen to be the representatives of category two and category three graphs respectively.

We took the single chunk performance data of device affinity study in Chapter 2 and produced rankings for each graph. Then we implemented both algorithms to support simultaneous running on CPU and GPU. The program starts with the main thread, and the main thread will first perform input reading and data structure initialization. During this period, the oracle scheduler would fill each core/SM's software work queue [28] by picking chunks based on the method described in Chapter 4. After initializing all data structures, the main thread creates one worker thread on each CPU core to process the chunks allocated to that core, and performs a GPU kernel launch. All threads are synchronized through a tree barrier after finishing each iteration. Based on the same argument we made in Chapter 3, instead of running PR and SSSP until convergence, we ran a fixed number of iterations for each graph. In this particular study, the number of iterations we ran is 15. One may also notice that for both PR and SSSP, their secondary kernels (buffer switching for SSSP and scaling by damping factor for PR) are much more regular than their main kernel. Namely, both secondary kernels do not suffer from any significant control or memory divergence. Thus we implemented these two secondary kernels as simple grid computing kernels, and there is no CPU-GPU collaboration for them since they have high parallelism and minimum divergence that intuitively make them favor GPGPU regardless of input graphs. One downside of this design is that it could potentially reduce L1 cache hit rate on both CPU and GPGPU since data is moved around during this phase. In this study, we utilize 5 CPU threads (1 master thread and 4 worker threads) and 15 SM's. GPU kernel thread block size is the same as task chunk size (256). We conducted an experiment to find the optimal ratio of splitting by sweeping from 0% to 70% of the chunks on CPU with 10% steps, along with all three stealing policies: steal-successor-one, steal-successor-three, and steal-all which allows stealing between devices. In addition to finding ideal splitting ratio, we also tested our rank based oracle scheduler. Results are shown in figure 6.1 and figure 6.2.



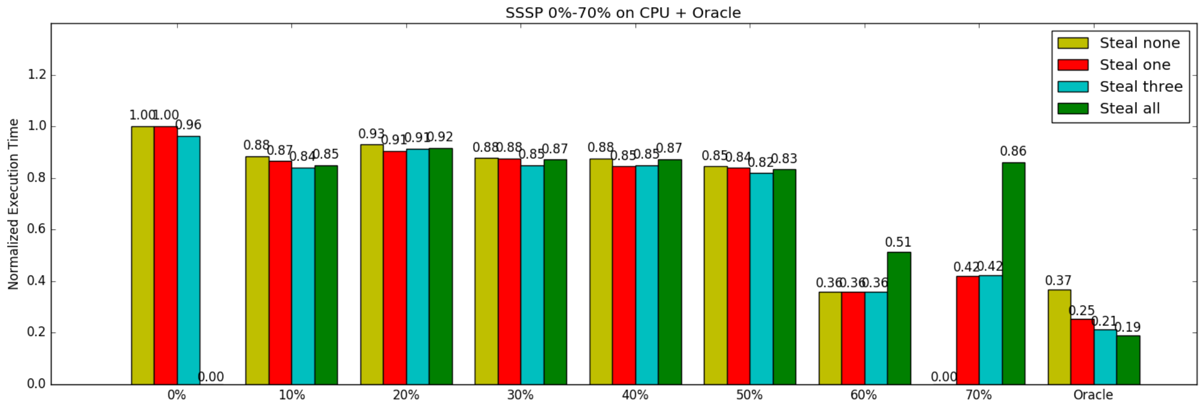
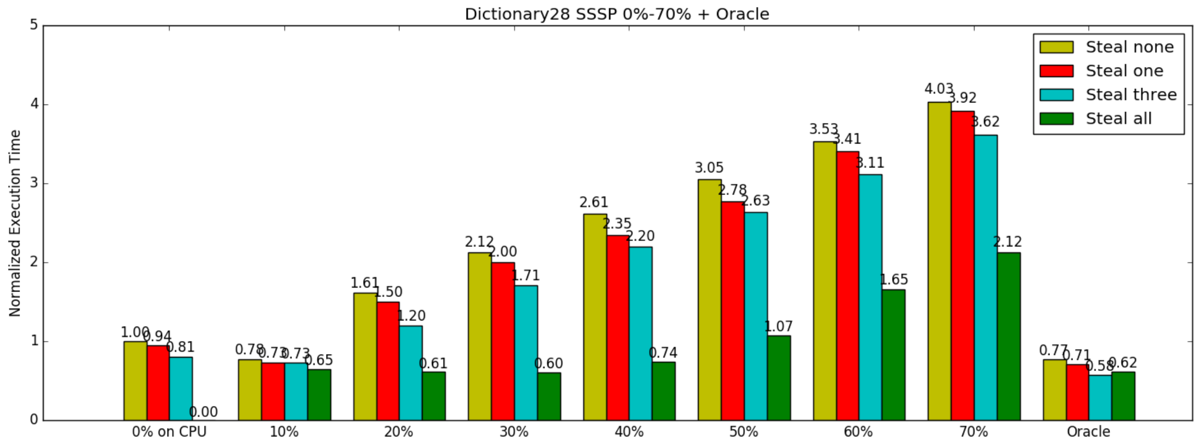


Figure 6.1 Rajat27 ratio sweep + Oracle.



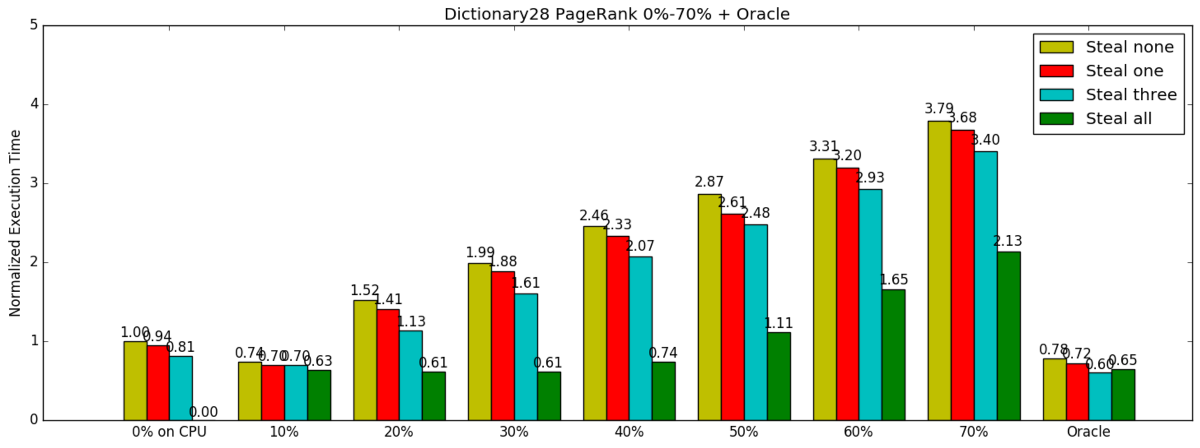
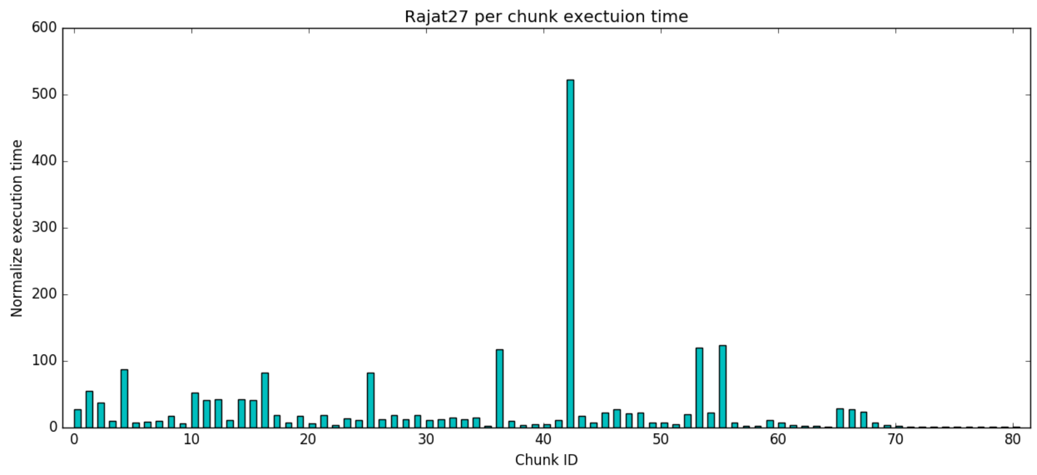


Figure 6.2 Dictionary28 ratio sweep + Oracle.

We first looked at how work-stealing helps when allocating the entire workload on GPU only, namely 0% [28]. Note that GPU-only allocation with steal-all policy is not available as we have a fixed number of four working threads and we cannot prevent a CPU thread from stealing GPU chunks. From the results, we can see that work-stealing provides considerable performance improvement in dictionary28. However, performance is only sightly improved in rajat27. To understand why this is the case, we looked into the single chunk execution time data we collected during the study of device affinity at task chunk granularity in Chapter 3. Figure 6.3 shows the normalized execution time of each task chunk on GPU in both graphs. Execution time is normalized to the shortest chunk in the graph. Through figure 6.3 we can see that load imbalance does exist in rajat27, but still, work-stealing does not help much since there are a few chunks that have much longer execution time than other chunks. Thus, the overall execution time is bounded by those few slow chunks, and the benefit of load-balancing is masked. On the other hand, dictionary28 does not have outliers as rajat27 does, which makes work-stealing more efficient. Note that in dictionary28 the slowest chunk only takes 9x longer than the fastest chunk. However, in rajat27, seven chunks are 100x slower.



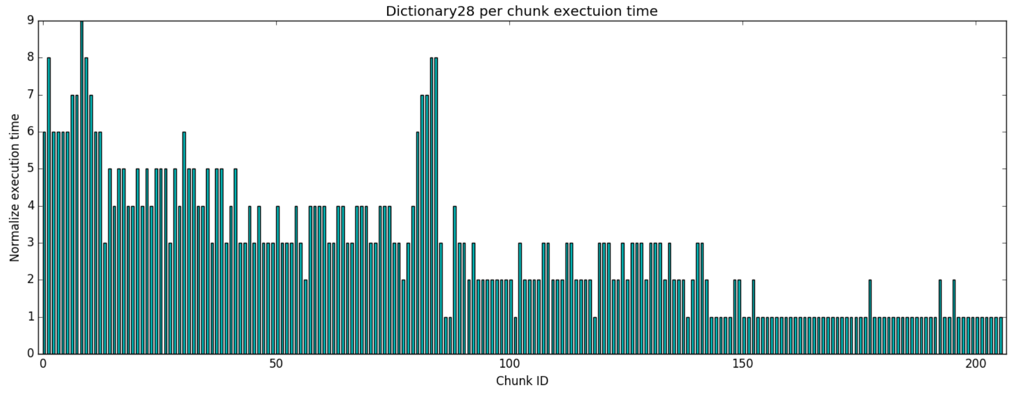


Figure 6.3 Normalized execution time.

The lack of performance improvement from work-stealing in the cases such as rajat27, and the existence of chunks that favor CPU over GPU, motivated running a single workload simultaneously on both CPU and GPU. We discuss the scenarios in rajat27 and dictionary28 separately.

In the case of rajat27 with steal-none, steal-successor-one, and steal-successor-three, from 0% to 50% of the chunks on CPU along with various stealing policies, the performance difference between configurations is relatively small. This is caused by, again, the fact that there are a few time-consuming chunks that slow down the overall process. Their performance on

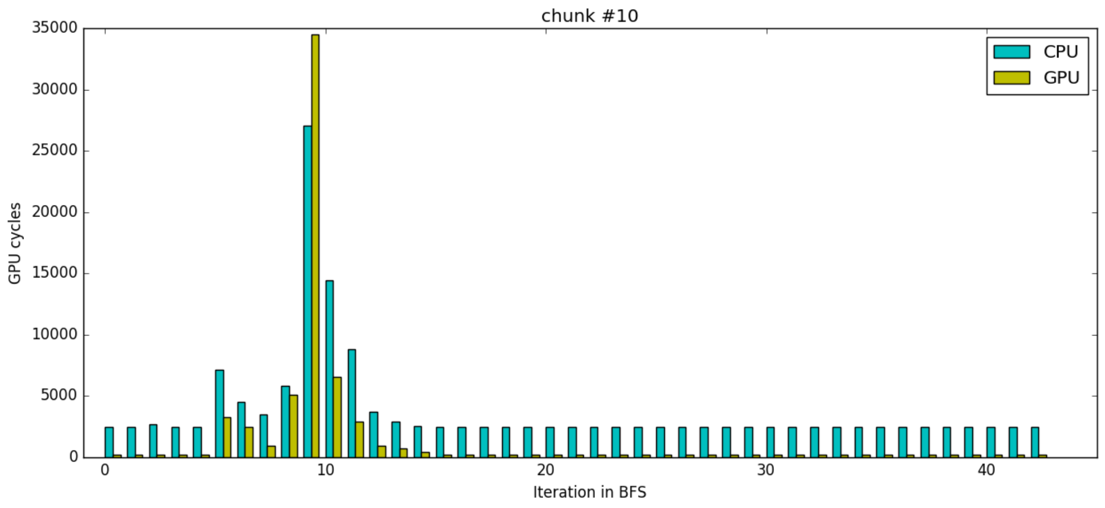
GPU is so bad that even though moving a certain portion of workload to CPU may speed up or slow down the execution of the chunks moved, the impact is masked by these few chunks' long execution time on GPU. At the mark of 60% on CPU, we see a significant drop in execution time. If we refer to figure 6.3, we can easily notice that moving from 50% to 60% of the chunks on CPU, the slowest chunk (chunk ID 42) which is more than 500x slower than the fastest chunk, is migrated to CPU. Combined with figure 3.3 from Chapter 3, we observe that chunk 42 has a GPU speedup over CPU far less than 1.0, which indicates that it strongly favors CPU. Thus, moving such a chunk to CPU results in a large performance improvement. When we move from 60% to 70% on CPU, the performance degrades again because too many chunks are allocated to the CPU, and now CPU becomes the new bottleneck. On the other hand, our rank based oracle scheduler provides a robust performance improvement. Even without work-stealing, oracle scheduling performance matches the best performance one can get from a ratio based splitting method. When combined with the work-stealing, oracle is 1.5x faster than the best configuration of ratio based splitting, and 5x better than work-stealing on GPU only. For rajat27, while steal-successor-one and steal-successor-three slightly improve performance at 60% and 70% of CPU, steal-all hurts performance badly. When 60% or 70% of the workload is allocated on CPU, GPU SMs will empty their work queues relatively fast when CPU cores are still busy. Then under the steal-all policy, SMs would begin to steal from CPU work queues and steal chunks that favor CPU back to GPU.

In the case of dictionary28, which is an example of category two graphs, our rank based Oracle scheduler still shows performance better than best ratio-based method, though the improvement is relatively small compare to what we achieved in rajat27, which is a category three graph. From the results, we see that for both applications with dictionary28 as input, as we sweep from 0% to 70%, the overall execution time slightly drops and then increases quickly. Lacking large internal variance as we have seen in rajat27, the improvement over naive GPU only allocation is not significant. One may notice that unlike the case in rajat27 where steal-all almost always hurts performance, it helps a lot in many configurations in dictionary28. If we look closer, we can observe that steal-all helps more in configurations that have a performance far worse than optimal ones. In case of 10% of chunks on CPU, which is the best ratio based solution, and oracle, steal-all only helps a little or even slightly hurts performance. However, in configurations where other stealing policies are far worse, for instance, 40% of CPU with steal-successor-one is 2.33x slower than GPU only without work-stealing, steal-all helps a lot. This is because in the ideal cases, both ratio-based and rank based, the amount of workload on both devices is balanced, while in other cases, there is room for load balancing across devices. Recall that all chunks of graphs in category two favor the same device. In this case, all chunks in dictionary28 favor GPU. As we migrate more chunks to GPU, we observe significant performance degradation without work-stealing across devices as they take longer to run on CPU, but with steal-all, the workload that GPU steals back from CPU favors GPU, which improves the performance.

6.2 BC: Applications Whose Behavior Varies Between Iterations

Unlike PR and SSSP, BC forms a wavefront when exploring the input graph, which results in different behavior between iterations. We implemented BC in a similar manner as we implemented PR and SSSP. However, instead of running a fixed number of iterations, we ran BC until convergence from a single source vertex. By doing so, we make sure each vertex, that is accessible from the source, is active during a certain iteration. Thus we can observe the correct variance of behavior between iterations. Since running until convergence takes a significant amount of time with dictionary28, this study was conducted on rajat27 only.

Before we start, we first need to understand how a certain chunk behaves over time. The execution time (in terms of GPU cycles) of two chunks over time are shown in figure 6.4. If we refer to figure 3.3 in Chapter 3, we will find that chunk 10 favors CPU while chunk 30 favors GPU. However, through figure 6.4, we can see that after iteration 14, both of them strongly favor GPU. We took a closer look and found that after iteration 14, both chunks have no vertex on the wavefront. Thus, there is no divergence, regardless of the graph structure. This shows that for applications like BC, a single prediction of device affinity is insufficient; we need to dynamically determine the affinity for the next iteration.



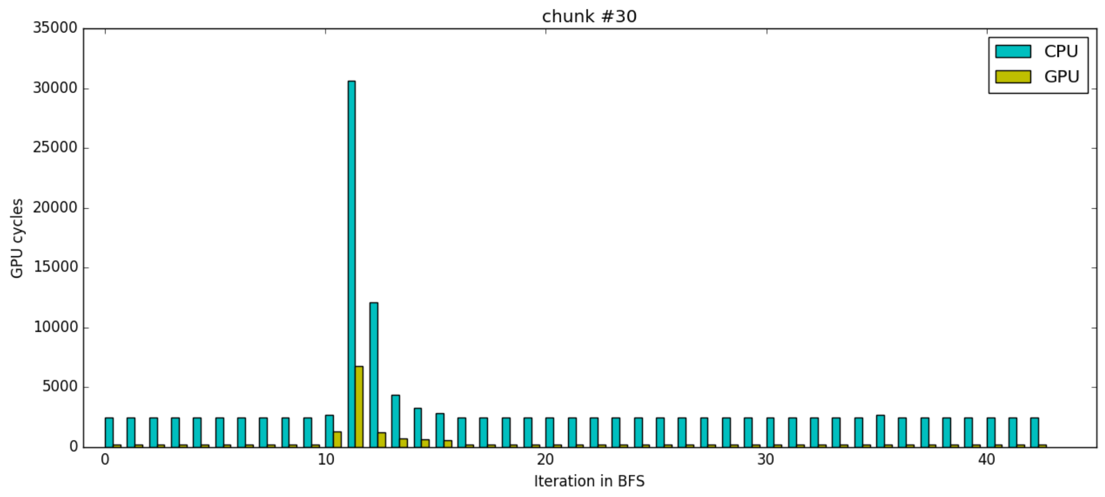


Figure 6.4 Behaviors of a particular chunk over time.

Another important observation we made before is that divergence is usually caused by the fact that vertices in a particular chunk have different numbers of neighbors. When only a portion of vertices in a chunk are on the wavefront, the problem of divergence is worse, since we can view a vertex that is not on the wavefront as a vertex with no neighbor. Thus, lane utilization degrades even more in BC.

Having made these two observations, we made minor changes to our rank based oracle scheduler. Namely, we cannot put a chunk on CPU solely based on its rank. We only migrate a chunk, that is picked by the original oracle scheduler to allocate on CPU, when the number of active edges is larger than a particular threshold. This is done by re-populating the work queues on CPU and GPU between iterations. The number of active edges is calculated by summing up the degrees of vertices on the wavefront in a particular task chunk dynamically during runtime. Same as PR and SSSP, experiments on BC utilized 5 CPU threads (1 master thread + 4 worker threads) and 15 SM's on our simulated CPU-GPU heterogeneous system. Results are shown in figure 6.5.

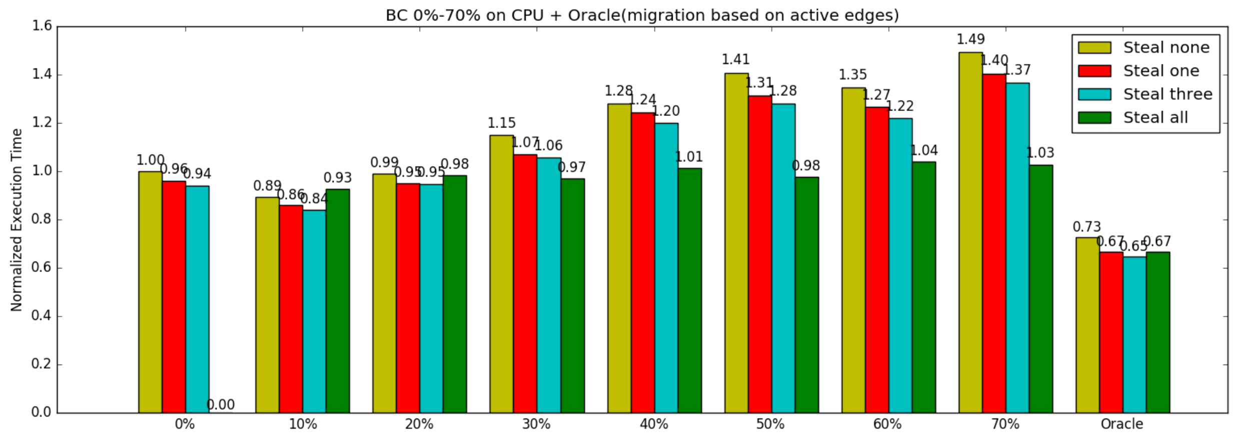


Figure 6.5 Betweenness centrality: Rajat27 ratio sweep + Oracle.

From ratio sweep part of figure 6.6, we can see the same pattern as we have seen in figure 6.4, in which as the portion of workload migrated to CPU increases, the performance increases until a certain point and then drops. Also, our modified rank based oracle schedule is consistently better than the best ratio based solution under all stealing policies.

CHAPTER 7. CONCLUSION

As tightly coupled CPU-GPU heterogeneous computing becomes increasingly popular, applications that utilize both kinds of devices at the same time have emerged. During the development of these applications, a good understanding of how to partition the workload between multicore CPU and GPGPU is necessary. Previous research has inspected this issue while aiming for an ideal partitioning ratio. However, to the best of our knowledge, no previous work has studied how device affinity at task chunk granularity, caused by the internal structures of inputs, could guide the partitioning of the irregular workloads on CPU-GPU heterogeneous systems. In this work, we first showed that device affinity exists not only at input granularity, as Farooqui et al. have shown [6], but also at task chunk granularity. Through the case studies of two categories of graph analytics applications, we have shown that for graphs which have chunks favoring different devices, our affinity aware rank based oracle scheduler could achieve performance improvement up to 1.5x over traditional ratio based splitting method, or up to 5x compared to naive GPU only allocation. For graphs where all chunks favor a certain device, our rank based Oracle scheduler can still achieve the same or slightly better performance compare to previous ratio based partitioning mechanisms. However, this work has many limitations need to be resolved in the future. For instance, our multicore CPU simulator assumes a one cycle execution time for all non-memory access operations. Also, determining the ranking of each task chunk dynamically is the key to make the proposed method practical. In this study, we assumed oracular knowledge of such a ranking. We expect that the number of edges in a vertex could be used to formulate a heuristic to drive a practical ranking algorithm. We leave this to future work.

REFERENCES

[1] Gregg, Chris, Michael Boyer, Kim Hazelwood, and Kevin Skadron. “Dynamic heterogeneous scheduling decisions using historical runtime data.” in *Workshop on Applications for Multi-and Many-Core Processors*. 2011.

[2] Luk, Chi-Keung, Sunpyo Hong, and Hyesoon Kim. “Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping.” in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE/ACM, 2009.

[3] Kaleem, Rashid, Rajkishore Barik, Tatiana Shpeisman, Brian T. Lewis, Chunling Hu, and Keshav Pingali. “Adaptive heterogeneous scheduling for integrated GPUs.” in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation.* ACM, 2014.

[4] Kofler, Klaus, Ivan Grasso, Biagio Cosenza, and Thomas Fahringer. “An automatic input-sensitive approach for heterogeneous task partitioning.” in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*. ACM, 2013.

[5] Shen, Jie, Ana Lucia Varbanescu, Yutong Lu, Peng Zou, and Henk Sips. “Workload partitioning for accelerating applications on heterogeneous platforms.” in *IEEE Transactions on Parallel and Distributed Systems* 27.9 (2016): 2766-2780.

[6] Farooqui, Naila, Rajkishore Barik, Brian T. Lewis, Tatiana Shpeisman, and Karsten Schwan. “Affinity-aware work-stealing for integrated CPU-GPU processors.” in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* ACM, 2016.

[7] Farooqui, Naila, Indrajit Roy, Yuan Chen, Vanish Talwar, and Karsten Schwan. “Accelerating graph applications on integrated GPU platforms via instrumentation-driven optimizations.” in *Proceedings of the ACM International Conference on Computing Frontiers*. ACM, 2016

[8] Owens, John D., Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. “GPU computing.” in *Proceedings of the IEEE* 96.5 (2008): 879-899.

[9] Harris, Mark. “Unified memory in CUDA 6.” *GTC On-Demand.* NVIDIA*,* 2013.

[10] Branover, Alexander, Denis Foley, and Maurice Steinman. “AMD fusion APU: Llano.” in *IEEE Micro* 32.2. IEEE, 2008.

[11] Sinclair, Matthew D., Johnathan Alsop, and Sarita V. Adve. “Efficient GPU synchronization without scopes: Saying no to complex consistency models.” in *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 2015.

[12] Komuravelli, Rakesh, Matthew D. Sinclair, Johnathan Alsop, Muhammad Huzaifa, Maria Kotsifakou, Prakalp Srivastava, Sarita V. Adve, and Vikram S. Adve. "Stash: Have your scratchpad and cache it too" in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. IEEE, 2015.

[13] Ryoo, Shane, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA." in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2008.

[14] Lindholm, Erik, John Nickolls, Stuart Oberman, and John Montrym. "NVIDIA Tesla: A unified graphics and computing architecture." in *IEEE Micro* 28.2. IEEE, 2008.

[15] Han, Tianyi David, and Tarek S. Abdelrahman. "Reducing branch divergence in GPU programs." in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*. ACM, 2011.

[16] Xu, Qiumin, Hyeran Jeon, and Murali Annavaram. "Graph processing on gpus: Where are the bottlenecks?" in *IEEE International Symposium on Workload Characterization.* IEEE, 2014.

[17] Che, Shuai, Bradford M. Beckmann, Steven K. Reinhardt, and Kevin Skadron. "Pannotia: Understanding irregular GPGPU graph applications." in *IEEE International Symposium on Workload Characterization.* IEEE, 2013.

[18] Page, Lawrence, Sergey Brin, Rajeev Motwani, and Terry Winograd. *The PageRank Citation Ranking: Bringing Order to the Web.* Stanford InfoLab, 1999.

[19] Brandes, Ulrik. "A faster algorithm for betweenness centrality." in *Journal of Mathematical Sociology* 25.2 (2001): 163-177.

[20] Adve, Sarita V., and Kourosh Gharachorloo. "Shared memory consistency models: A tutorial." in *Computer* 29.12 (1996): 66-76.

[21] Magnusson, Peter S., Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. "Simics: A full system simulation platform." in *Computer* 35.2 (2002): 50-58.

[22] Bakhoda, Ali, George L. Yuan, Wilson WL Fung, Henry Wong, and Tor M. Aamodt. "Analyzing CUDA workloads using a detailed GPU simulator." in *IEEE International Symposium on Performance Analysis of Systems and Software.* IEEE, 2009.

[23] Agarwal, Niket, Tushar Krishna, Li-Shiuan Peh, and Niraj K. Jha. “GARNET: A detailed on-chip network model inside a full-system simulator.” in *IEEE International Symposium on Performance Analysis of Systems and Software.* IEEE, 2009.

[24] Martin, Milo M.K., Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. “Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset.” in *ACM SIGARCH Computer Architecture News* 33.4 (2005): 92-99.

[25] Choi, Byn, Rakesh Komuravelli, Hyojin Sung, Robert Smolinski, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Nicholas P. Carter, and Ching-Tsun Chou. “DeNovo: Rethinking the memory hierarchy for disciplined parallelism.” in *Proceedings of the 20th International Conference on Parallel Architectures and Compilation.* IEEE, 2011.

[26] Davis, Timothy A., and Yifan Hu. “The University of Florida sparse matrix collection.” in *ACM Transactions on Mathematical Software* 38.1 (2011): 1.

[27] Burtscher, Martin, Rupesh Nasre, and Keshav Pingali. “A quantitative study of irregular programs on GPUs.” in *IEEE International Symposium on Workload Characterization*. IEEE, 2012.

[28] Salvador, Giordano, Private communication.

[29] Smith, Brad. “ARM and Intel battle over the mobile chip's future.” in *Computer* 41.5 (2008).

[30] Nickolls, John, and William J. Dally. “The GPU computing era.” in *IEEE Micro* 30.2 (2010).

[31] Sinclair, Matthew D., and Johnathan Alsop, Private communication.

[32] Sinclair, Matthew D., Johnathan Alsop, and Sarita V. Adve. “Semantics and evaluation for relaxed atomics on heterogeneous systems.” in *Proceedings of the 44th Annual International Symposium on Computer Architecture*. IEEE, 2017.