

**POWER-CONSTRAINED PERFORMANCE OPTIMIZATION OF
GPU GRAPH TRAVERSAL**

A Thesis
Presented to
The Academic Faculty

by

Adam McLaughlin

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
December 2013

Copyright © 2013 by Adam McLaughlin

POWER-CONSTRAINED PERFORMANCE OPTIMIZATION OF GPU GRAPH TRAVERSAL

Approved by:

Sudhakar Yalamanchili, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Hyesoon Kim
School of Electrical and Computer Engineering
Georgia Institute of Technology

Saibal Mukhopadhyay
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: 13 December 2013

To Jennifer, for listening to my most inconsequential concerns and showing me that there's a lot more to experience in life than I had once thought.

ACKNOWLEDGEMENTS

I'd like to begin by thanking Professor Sudhakar Yalamanchili, not only for his advice with regard to this thesis but also for his insight on life as a graduate student in general. I was tremendously fortunate to have the opportunity to work with him as well as the other members of CASL and hope to continue to collaborate with them in the future.

Additionally, I would like to thank Professors Hyesoon Kim and Saibal Mukhopadhyay for taking the time to review this thesis and showing interest in my work.

Next, I'd like to thank all of my collaborators at AMD Research. Special thanks go to Indrani Paul, Joseph Greathouse, Bobbie Manne, Peter Bailey, and Bill Brantley for guiding me through my weekly updates. I am truly grateful for all of their contributions to this work and my development as a researcher.

Finally, I'd like to thank my parents Tom and Diana for their continued support and pride for what I've done academically and otherwise. Words cannot describe my appreciation for their trust and encouragement.

TABLE OF CONTENTS

DEDICATION	iii
ACKNOWLEDGEMENTS	iv
LIST OF TABLES	vi
LIST OF FIGURES	vii
SUMMARY	viii
I INTRODUCTION	1
II BACKGROUND	5
2.1 The OpenCL Programming Model	5
2.2 Breadth-First Search	7
2.3 The GPU Hardware Platform	9
III WORKLOAD CHARACTERIZATION	12
3.1 Graph Properties	12
3.2 Analysis of Static Properties	14
3.3 Analysis of Dynamic Properties	18
IV POWER MANAGEMENT ALGORITHMS	22
4.1 Static Power Management Algorithm	22
4.2 Dynamic Power Management Algorithm	23
V PERFORMANCE EVALUATION	26
5.1 Experimental Methodology	26
5.2 Experimental Results	28
5.3 Lessons Learned	33
VI RELATED WORK	34
VII CONCLUDING REMARKS	36
REFERENCES	38

LIST OF TABLES

1	GPU Computing Vocabulary	5
2	GPU DVFS states for the A10-5800K APU	10
3	Suite of benchmark graphs	26
4	GPU counter statistics for a few benchmark graphs	31

LIST OF FIGURES

1	Workload over time for <i>coPapersCiteseer</i> ($n = 434,102$, $m = 16,036,720$) . .	2
2	Workload over time for <i>hugebubbles-00020</i> ($n = 21,198,119$, $m = 31,790,179$)	2
3	Two-dimensional OpenCL work-item layout [36]	6
4	A simple graph traversed in breadth-first order	7
5	Kernel for one iteration of BFS	9
6	Die shot of AMD A-Series APU	10
7	Optimal power configuration for <i>G3_Circuit</i> as a function of power cap . . .	13
8	Optimal power configuration for <i>delanay-n23</i> as a function of power cap .	13
9	Comparison of potential scaling Technologies	15
10	Classification of graph input as frequency- or CU-sensitive	17
11	Change in most advantageous power state with search iteration for <i>ldoor</i> . .	19
12	Change in most advantageous power state with search iteration for <i>asia.osm</i>	20
13	Algorithm for Static Power Management of GPU Graph Traversal	23
14	Algorithm for Dynamic Power Management of GPU Graph Traversal	24
15	Comparison of static and dynamic power management techniques	29
16	Histogram of search iteration by most advantageous DVFS state for <i>asia.osm</i>	30
17	Histogram of search iteration by the most advantageous number of CUs for <i>asia.osm</i>	31

SUMMARY

Graph traversal represents an important class of graph algorithms that is the nucleus of many large scale graph analytics applications. While improving the performance of such algorithms using GPUs has received attention, understanding and managing performance under power constraints has not yet received similar attention.

This thesis first explores the power and performance characteristics of breadth first search (BFS) via measurements on a commodity GPU. We utilize this analysis to address the problem of minimizing execution time below a predefined power limit or power cap exposing key relationships between graph properties and power consumption. We modify the firmware on a commodity GPU to measure power usage and use the GPU as an experimental system to evaluate future architectural enhancements for the optimization of graph algorithms. Specifically, we propose and evaluate power management algorithms that scale i) the GPU frequency or ii) the number of active GPU compute units for a diverse set of real-world and synthetic graphs. Compared to scaling either frequency or compute units individually, our proposed schemes reduce execution time by an average of 18.64% by adjusting the configuration based on the inter- and intra-graph characteristics.

CHAPTER I

INTRODUCTION

Graph analysis is a fundamental building block in numerous computing domains. Areas as diverse as electronic design automation [11], compilers [26], scientific computing [10], and social networking [12] rely on these algorithms for many of their important functions. In addition, graph algorithms are an integral part of an emerging class of data intensive supercomputing applications [35].

However, graph algorithms present a unique set of architectural challenges compared to traditional supercomputing physics applications. Their runtime is dominated by latency, there is relatively little computation to hide memory costs, the access patterns are irregular and highly data dependent, and there is little apparent locality at all levels of memory [17]. At the same time the size and scope of graphs in these emergent applications are stressing memory and compute throughput of architectures for the foreseeable future and consequently have received much attention [11] [16] [25] [27].

Graph traversal in particular is a primitive found in many graph analysis applications. Figures 1 and 2 illustrate the workload characteristics for a breadth-first search (BFS) algorithm across two different input graphs. The X-axis indicates the iteration count and the Y-axis indicates the amount of parallelism for that iteration (size of the vertex frontier as shown in Figure 4 and described in Figure 5). These data show that parallelism varies significantly within the traversal of a graph and across graphs. Within an iteration, the massive parallelism in a GPU can be exploited to concurrently traverse nodes. However, there is a synchronization barrier at the end of each iteration. Thus, load imbalances across threads within an iteration can cause a thread (a *critical thread*) to significantly reduce power efficiency by reducing hardware utilization. In this case, an investment in frequency scaling can provide performance improvement analogous to the manner in which high frequency operation can speed up serial sections of parallel code, although with a

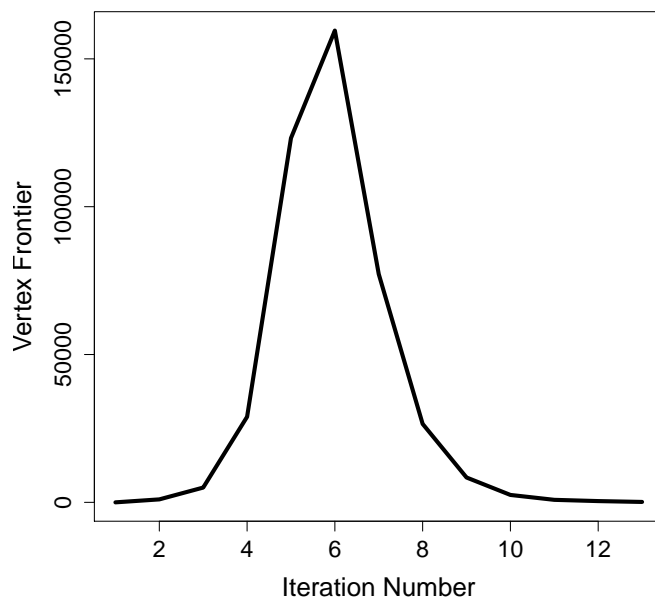


Figure 1: Workload over time for *coPapersCiteseer* ($n = 434,102$, $m = 16,036,720$)

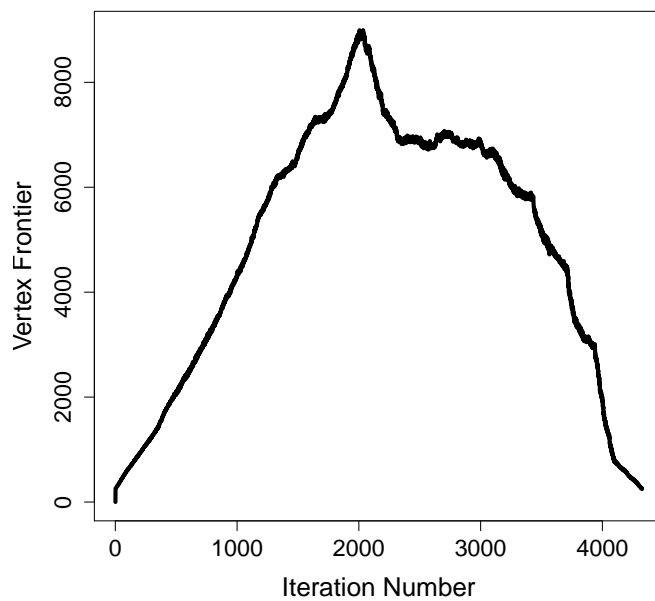


Figure 2: Workload over time for *hugebubbles-00020* ($n = 21,198,119$, $m = 31,790,179$)

commensurate increase in power. Conversely if parallelism is low, GPU compute units are idle wasting energy/power. In this case scaling the number of active compute units by power gating can improve power efficiency.

The performance challenge is accompanied by a power challenge as modern and emerging processors are power constrained. As silicon technology nodes integrate more transistors, the lack of concordant power reduction results in increased power density [5]. As such, it may not be possible to power all of the transistors in a chip at their optimal frequency due to thermal effects [13]. Commercial CPUs and GPUs cannot operate at high frequencies unless some of their cores are disabled or throttled [6] [30] [33]. Power may also be constrained due to requirements external to the processor. For example, power is a first-class design constraint in large data centers, where it can cost nearly as much to power an installation as it costs to purchase the equipment [15]. With installations consuming tens of megawatts [38], power companies often request that data centers limit their power consumption during parts of the day. Going over this contractual power limit can invoke penalty rates, which greatly increase power costs and therefore lead to power caps imposed on processor level operation. Consequently, it is important to understand and implement performance optimizations under a power cap.

This thesis first seeks to characterize the power behavior and the evolution of fine-grained parallelism in breadth first search (BFS) using measurements on a Trinity A10-5800K APU. There is particular interest in how the properties of the graph being traversed influence power consumption using two power management techniques - dynamic voltage and frequency scaling and scaling the number of active GPU compute units. This characterization exposes a fundamental trade-off between frequency and parallelism when maximizing power efficiency. This trade-off is closely related to the structural properties of graphs which in turn determine the available amount of fine-grained parallelism and the relative impact of critical threads. This tradeoff is exploited by developing adaptive power management schemes that seek to minimize execution times while remaining under a power cap. In particular it was observed that the most effective choice of power management scheme - i) scaling frequency, ii) scaling number of compute units, or iii) both - evolves over time with

the workload.

This thesis seeks to make the following contributions:

- We characterize the power consumption of breadth first search (BFS) for large graphs executing on GPUs using measurements on an AMD Trinity APU. We explain how the power consumption is related to the evolution of fine grained parallelism and load imbalances within thread groups as determined by the graph’s structural properties.
- This characterization is used to identify and explain the tradeoffs between scaling frequency vs. number of active GPU compute units to improve power efficiency. These tradeoffs form the foundation of two adaptive power management algorithms - i) a static algorithm that fixes the GPU power configuration for each input graph, and ii) a dynamic algorithm that adapts the GPU power configuration for each BFS iteration.
- Through firmware changes on state-of-the-art GPU we enable and evaluate scaling the number of active GPU compute units as a power management strategy in addition to traditional dynamic voltage frequency scaling.
- Across a range of benchmark graphs, we show that the static algorithm provides on average 15.6% improvement in execution time over that achievable by frequency scaling alone or 13.6% over that achievable by scaling compute units alone. Dynamic adaptation of the power configuration at finer grain intervals during traversal improves execution time by an additional 3%.

Section 2 provides background information on OpenCL, BFS, and the GPU hardware used in this analysis. Section 3 provides a characterization of the power behavior of BFS on a GPU across a wide range of graphs. This characterization is the basis of the static and dynamic power management schemes described in Section 4.1 and Section 4.2 respectively. A comparative evaluation of these algorithms with an oracle is elaborated on in Section 5 and the thesis concludes with a discussion of related work and concluding remarks.

CHAPTER II

BACKGROUND

This section provides relevant background on the OpenCL programming model in the context of GPU computing, the specific choice of traversal algorithm (BFS), and the target heterogeneous processor utilized in this work. We point out that from a power constrained optimization perspective, we can view other algorithms for graph traversal as simply differing in how workload (i.e., parallelism) evolves. Consequently we argue that the principles learned in this study are relevant to the design and implementation of power constrained graph traversal algorithms for GPUs in general.

2.1 The OpenCL Programming Model

In this section we provide a brief description of the OpenCL programming model. Further information regarding OpenCL can be found in [28].

The OpenCL programming model was designed to support general-purpose computing on heterogeneous architectures. It differs from models such as CUDA because code written in OpenCL is general enough to execute on devices such as traditional CPUs, AMD and Nvidia GPUs, and DSPs, among other architectures, assuming that no architecture-specific optimizations have been applied. For readers who are more familiar with CUDA, a comparison of OpenCL and CUDA terminology is presented in Table 1. Previous work has shown that this application portability tends to come at the cost of performance [39]. Hence, if performance of an application is crucial, the development of OpenCL code that is

Table 1: GPU Computing Vocabulary

CUDA Terminology	OpenCL Terminology
Thread	Work-item
Thread Block	Work-group
Warp	Wavefront
Shared Memory	Local Memory

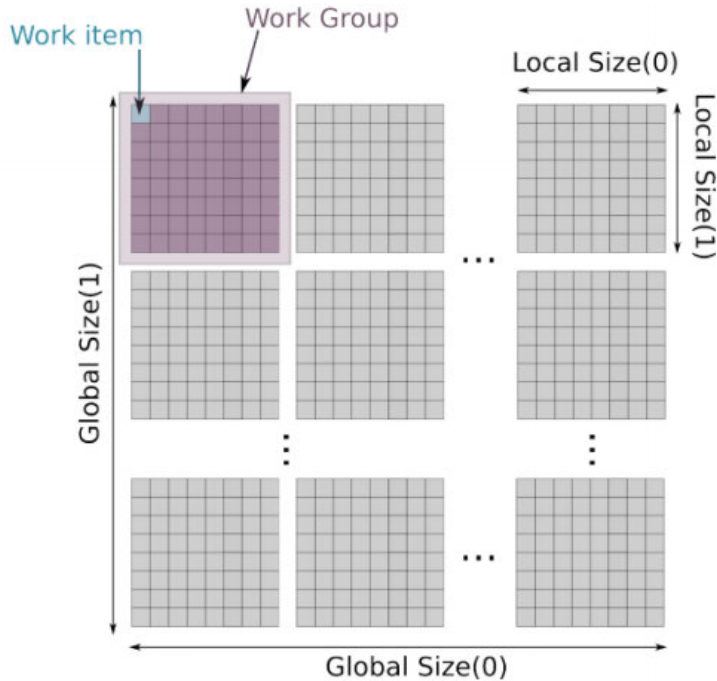


Figure 3: Two-dimensional OpenCL work-item layout [36]

specialized to a target architecture is required.

The execution of an OpenCL kernel spawns a programmer-specified number of *work-groups*, each of which contains a programmer-specified number of *work-items*, which can be thought of as fine-grained threads, as shown in Figure 3. This hierarchical model allows for different granularities of parallelism. Each work-item executes the code within the kernel, using their specific registers to operate on their respective chunk of global data. An implicit barrier among all work-items of all work-groups occurs between kernel launches. Explicit barriers within a kernel for the work-items within each work-group are also supported [28]. Since the parallel Breadth-First Search of a graph requires a barrier among all work-items between each search iteration and since work-items can have varying workloads, situations where many work-items are stalled at a barrier waiting for the critical thread can occur. Work-groups can be one, two, or three-dimensional layouts of work-items, depending on how the programmer chooses to decompose the task at hand.

In the context of GPU computing, data must be transferred from the *host* (typically a CPU) to the *device* (the GPU). This process transfers data to the GPU’s global memory.

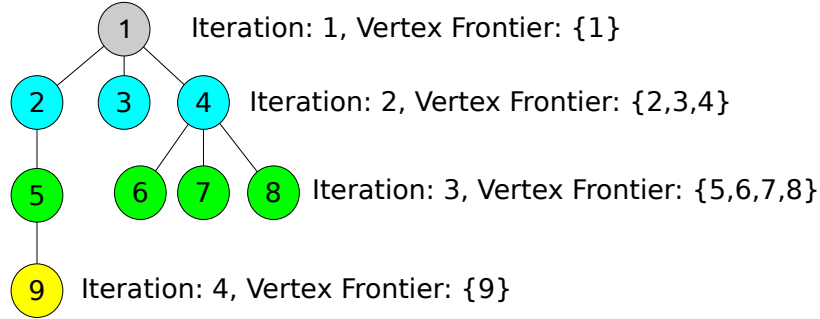


Figure 4: A simple graph traversed in breadth-first order

A limited amount of programmer accessible *local memory* (scratchpad) can also be utilized (shared) by the work-items among a work-group [28]. Local memory is a low-latency and high-bandwidth memory that is conceptually similar to an L1 cache. From a hardware context, code is executed in a group of threads, which is known as a *wavefront*. The target architecture used in this study has a wavefront size of 64 threads. Ideally, all threads within a wavefront execute the same instruction. In the case that a branch statement causes different groups of threads within the same wavefront to execute different instructions, the execution of these instructions will be serialized, which can lead to significant performance losses. This phenomenon is known as *branch divergence* [37]. Similarly, performance degradations can occur depending on how the work-items within a wavefront access global memory. In the case that all work-items in a wavefront access consecutive locations in global memory, all of the memory accesses can be *coalesced* into one memory access [37]. Otherwise, these accesses are serialized. Avoiding branch divergence and maximizing coalesced memory accesses is particularly difficult for graph algorithms since the structure of the graph, which isn't known until run time, plays such a monumental role in the distribution of work among work-items.

2.2 Breadth-First Search

Graph traversal represents an important class of graph algorithms that is the nucleus of many graph analytics applications used within fields such as social networking [12], electronic design automation [11], and compiler optimization [26]. Breadth-first search (BFS) is an important general graph traversal algorithm that often serves as the framework for the design of customized traversal techniques and therefore has broad impact. Consequently,

we study BFS as an exemplar of general graph traversal algorithms, from which power and time optimization principles can be derived for use in general graph traversal algorithms (and thereby graph analytics applications).

BFS can be summarized as follows: given a graph $G = (V, E)$ with n vertices and m edges, BFS starts at a source node s and in one iteration traverses (discovers) all of its neighbors. These neighbors form the *vertex frontier*. In the next iteration, all of the neighbors of nodes in the current vertex frontier are traversed to produce the next vertex frontier. Iteration continues until all nodes reachable from s have been visited. By definition, the vertex frontier of the first iteration is s . All nodes in the vertex frontier of a particular iteration can be processed in parallel.

Figure 4 illustrates the progression of the algorithm for a simple graph and demonstrates that the parallelism of BFS varies across iterations and is dependent on the structure of the graph. The first iteration has just one node to process, while the third iteration has four nodes that can each be assigned to parallel threads. Furthermore, the amount of work done by each thread can also vary and is dependent on the structure of the graph. Using the second iteration as an example, node 2 has two edges to traverse, node 3 has one edge to traverse, and node 4 has four edges to traverse. In bulk synchronous parallel (BSP) implementations where nodes in a frontier are distributed amongst parallel threads, this workload imbalance can leave many threads stalled at a barrier, waiting for the thread with the most work. We refer to this thread in a group as the *critical thread* of the group. We later show that these variations can have a significant impact on power management.

In this thesis we utilize the parallel algorithm presented by Luo et al. [25] from the Scalable Heterogeneous Computing (SHOC) benchmark suite [8]. To the best of our knowledge, it is the fastest available OpenCL BFS implementation, with asymptotically optimal $O(m+n)$ linear time complexity. Pseudocode for this implementation of BFS is provided in Figure 5. This kernel receives the vertex frontier to be processed for the current iteration, traverses the edges of the frontier, and adds the unexplored neighbors to the frontier for the next iteration while simultaneously updating their cost, or distance from the source node.

```

1: function BFS_MULTI_BLOCK_KERNEL(frontier, frontier_length, visited, cost, edgeArray,
   edgeArrayAux, next_frontier)
2:   tid ← get_global_id(0) #Obtain Thread ID
3:   if (tid < frontier_length) then
4:     node ← frontier[tid]
5:     visited[node] ← 0
6:     offset ← edgeArray[node]
7:     next ← edgeArray[node + 1]
8:     while (offset < next) do
9:       nid ← edgeArrayAux[offset]
10:      old_cost = cost[nid]
11:      cost[nid] ← min(old_cost, cost[node] + 1)
12:      if (old_cost > cost[node] + 1) then
13:        old_visited = visited[nid]
14:        visited[nid] = 1
15:        if (old_visited == 0) then
16:          next_frontier.append(nid)
17:        end if
18:      end if
19:      offset = offset + 1
20:    end while
21:  end if
22: end function

```

Figure 5: Kernel for one iteration of BFS

Graph data is stored in the Compressed Sparse Row (CSR) format [3]. In the implementation of Figure 5, atomic operations are used to prevent data races, and the cost array is initialized to infinity for all nodes.

2.3 The GPU Hardware Platform

Figure 6 shows the floor plan of an AMD A-Series heterogeneous accelerated processing unit (APU), code-named “Trinity”, which will be used in this study. It contains two out-of-order dual-core CPU Compute Units (CUs), a graphics processing unit (GPU) accelerator, and shared logic such as the memory controller. The GPU consists of 384 AMD Radeon cores, each capable of one single-precision fused multiply-add computation operation per cycle. The GPU is organized as six SIMD units (also known as CUs) each containing 16 processing units that are each four-way VLIW. More details regarding this processor can be found in [6].

The GPU is on a separate power plane, allowing its voltage and frequency to be controlled independently. Table 2 shows the DVFS states for the GPU in the AMD A10-5800K;

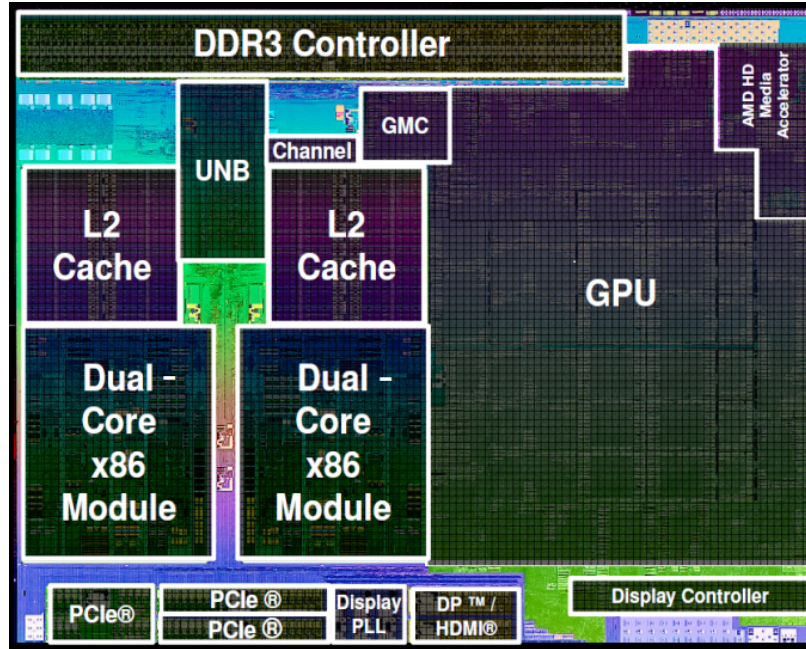


Figure 6: Die shot of AMD A-Series APU

Table 2: GPU DVFS states for the A10-5800K APU

DVFS State	Frequency
High	800 MHz
Medium	633 MHz
Low	304 MHz

we will refer to these states throughout the rest of this thesis [30]. All CUs in a GPU share the same voltage plane but can be power gated on or off individually through software registers.

The AMD A-Series APU uses a sophisticated power monitoring and management technology called AMD Turbo CORE [6] to optimize performance for a given power and thermal constraint. The power-management firmware on the AMD A-Series APU uses approximated power and temperature values as inputs to the Bidirectional Application Power Management (BAPM) algorithm, which controls the power allocated to each compute entity on the chip [30].

In this thesis, we do not rely on the BAPM infrastructure for power management. Rather, we use the GPU as an experimental system to evaluate future architectural modifications for the optimization of graph algorithms. The current state-of-the-art in GPU power management algorithms only consider frequency scaling. They do not perform techniques such as increasing or decreasing the number of active compute units (CUs) to optimize performance within power constraints. Our results argue that CU scaling should be considered for future architectures in the era of dark silicon, when all of the components of a chip cannot be simultaneously powered [13]. Alternately, a power cap imposed by the chip’s controller may require the GPU to lower power usage - providing alternative control options can be more efficient. This thesis analyzes the value of scaling *both* frequency and the number of active CUs in a GPU. There are six SIMD CUs and three GPU frequencies in the AMD A-series APUs, resulting in a total of 18 power configurations evaluated for this study.

CHAPTER III

WORKLOAD CHARACTERIZATION

This section provides an analysis of graph traversal from the perspective of power management. In particular we are interested in maximizing power efficiency which we define as minimizing execution time under a fixed power cap. We first focus on how properties of the graph being traversed influence power consumption using two power management techniques - i) dynamic voltage frequency scaling (DVFS) or simply frequency scaling, and ii) CU scaling, or scaling the number of active CUs (equivalently power gating CUs). Details regarding the experimental setup used to make these measurements can be found in Section 5.1.

3.1 Graph Properties

The workload experienced by graph traversal algorithms is highly input-dependent - the structure of input graphs can vary significantly, challenging the design of power management algorithms. Consider Figures 1 and 2, which illustrate variation in the size of the vertex frontier for two example graphs as a function of the iteration number. These graphs are representative of the two broad categories of graphs that we study in this work. The first category, represented by *coPapersCiteseer*, corresponds to graphs wherein BFS exhibits a small number of iterations. However, some of these iterations process a large number of nodes because some of the nodes in the graph have a very high degree, i.e., are connected to a large number of nodes. Power law graphs, such as web crawls of the Internet, tend to fall into this category [14]. The second category, demonstrated by *hugebubbles-00020*, corresponds to graphs whose nodes have a smaller, more consistent node degree. BFS over these graphs takes significantly more iterations to traverse with a smaller number of nodes searched per iteration. Graphs that represent meshes for dynamic simulations or road networks tend to fall into this category [2].

We refer to a *power configuration* as the combination of a DVFS state and the number

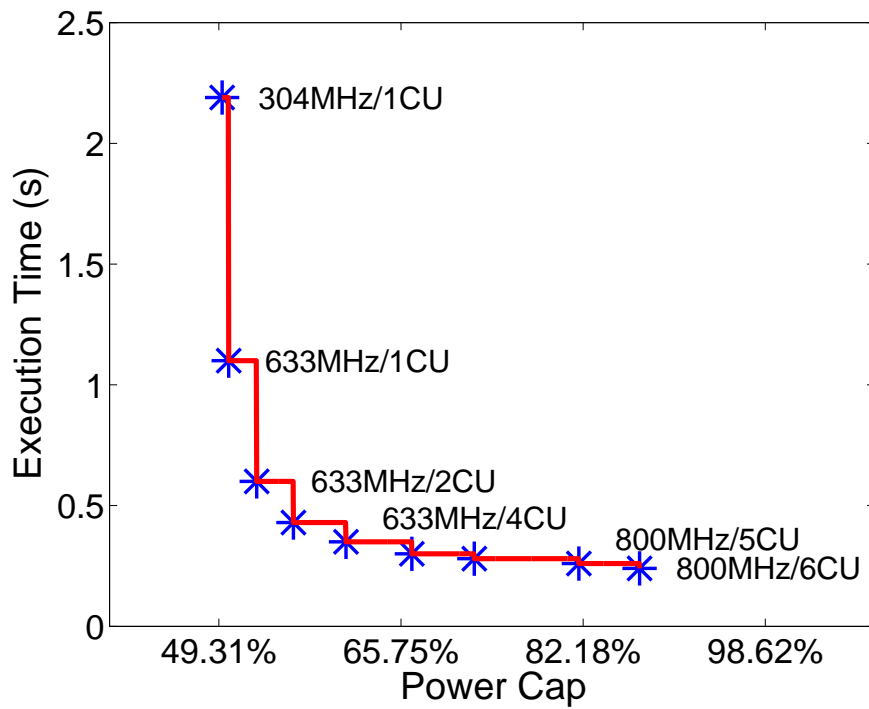


Figure 7: Optimal power configuration for *G3_Circuit* as a function of power cap

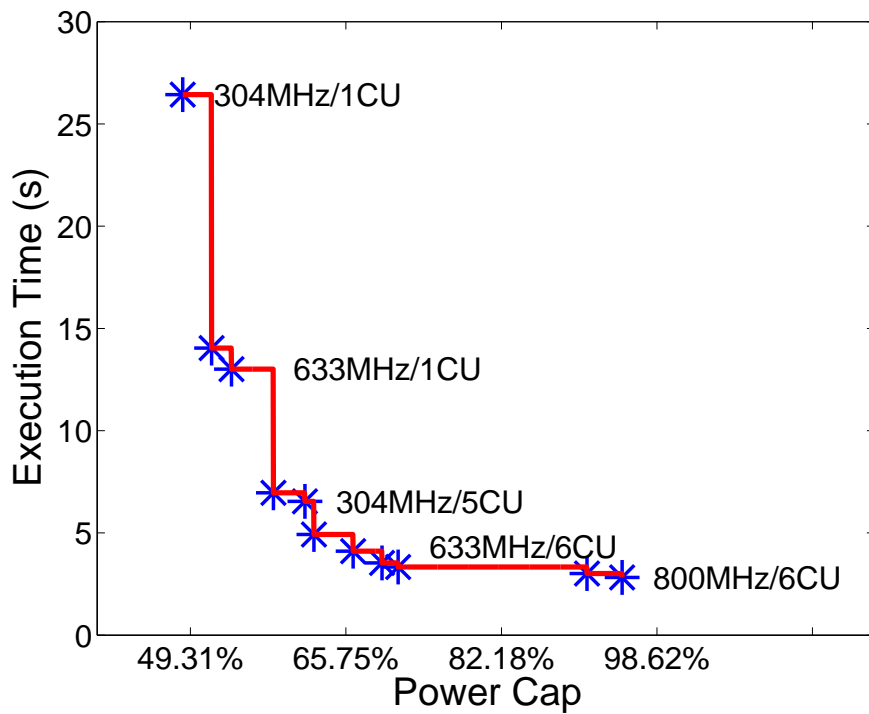


Figure 8: Optimal power configuration for *delaunay_n23* as a function of power cap

of active CUs (recall all CUs share a voltage plane) in the GPU. We exhaustively explored the BFS execution times for all 18 possible power configurations (three DVFS states * six CUs) on the A10-5800K, recording the average power consumption throughout the search as well as the elapsed time of the search. We define the “best” configuration to be that with the smallest execution time of all of the configurations that have an average power consumption that is less than the power cap for all search iterations. Figures 7 and 8 show the behavior of the best power configuration as a function of the power cap.

Consider a power cap of 82.18% of the maximum GPU power observed for any graph in our benchmark suite (see Section 5.1 for more details); for the *G3-Circuit* graph input, the best configuration under this power cap is the high-frequency DVFS state with five active CUs, but it is the medium-frequency DVFS state with six active CUs that is optimal for the *delaunay_n22* graph input. This result implies that the execution behavior of BFS with the former graph is more sensitive to frequency while with the latter graph is more sensitive to number of active CUs. We further explore causes of this behavior in the next section.

This analysis shows that the most power efficient operation for BFS depends on the evolution of the vertex frontier in terms of both parallelism (number of nodes in the frontier) and speed (rate at which the frontier grows). These properties are correspondingly addressed by scaling the number of CUs (parallelism) and frequency (speed) and can require exercising both techniques to realize the most power efficient operation.

3.2 Analysis of Static Properties

In this section we analyze the issues encountered in selecting a single power configuration that is the most power efficient for the BFS traversal of a specific input graph. In practice, this analysis would be used to statically configure the power state of the processor prior to execution, and this configuration would remain fixed for the duration of the traversal. Selection of the power configuration for a given power cap is achieved by selecting i) the frequency of the CUs, and/or ii) the number of active CUs. Our goal is to understand the effectiveness of scaling the frequency or number of active CUs for various types of graphs.

Figure 9 shows the execution time for a power cap set to 62% of the maximum power

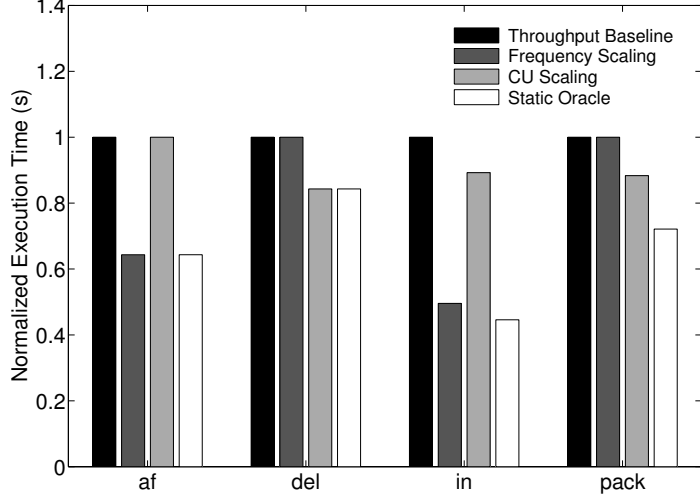


Figure 9: Comparison of potential scaling Technologies

observed for any graph application in the benchmark suite. The execution time is normalized to that of the throughput baseline (see Section 5.1). With this baseline power configuration, we were able to fit all graphs analyzed in this thesis under the power cap with possible power headroom to scale frequency or CUs (or both) for some graphs. We examine the following data to understand the sensitivity to frequency and CU scaling. For each graph, with a constant number of active CUs, frequency is scaled until either the highest power DVFS state or the power cap is reached. Similarly, we maintain the DVFS state of and vary the number of active CUs until either the maximum number of CUs is reached or the power cap is exceeded. Finally, we compare these execution times with that of an oracle with knowledge of the best power configuration, i.e., the one that minimizes execution time while staying under the power cap.

Results in Figure 9 show that the execution of the *delaunay_n23* random triangulation graph is more sensitive to CU scaling than frequency scaling, matching the performance of the oracle. In contrast, the execution of the *af_shell10* sheet metal forming graph, is more sensitive to frequency scaling rather than CU scaling, again matching the performance of the oracle. The *af_shell10* graph has many nodes of high degree and a large variance in node degree across the vertex frontier. When nodes in the vertex frontier are assigned to threads in a work group, the probability of load imbalance across threads is high. The thread processing the node with the largest degree (the *critical thread*) will have many edges

to traverse while other threads in the workgroup are idle waiting at a barrier and consuming power. Frequency scaling speeds up the execution of the critical thread, and thereby the workgroup and program improving power efficiency. Speeding up the critical thread in this manner is analogous to speeding up the serial fraction in parallel programs.

Conversely, for graphs such as *delanay_n22*, the node degrees are more balanced. Thus, when nodes in a vertex frontier are distributed across threads the workload tends to be more balanced across all threads and the effect of critical threads is less pronounced. Such graphs are more sensitive to CU scaling which effectively exploits parallelism across nodes in the vertex frontier. Thus the choice of static power management strategy is a tradeoff between exploiting parallelism and speeding up critical threads which form a bottleneck to parallel computation.

The remaining two cases are especially interesting. For the *packing_500x100x100-b050* fluid mechanics graph, CU scaling provides limited benefit. In actuality, reducing the number of CUs slightly so that frequency scaling is possible provides the configuration chosen by the oracle. This result demonstrates that, while frequency scaling and CU scaling are independently useful, combining them can be better.

The *in-2004* web crawl graph also exemplifies this result because sufficient room exists under the power cap for both frequency and CU scaling, though a combination of both is needed to achieve the best performance. The issue now is to classify graphs into categories for which the graph traversal is more or less sensitive to frequency scaling vs. CU scaling.

The metadata we chose to classify benchmark graphs is simple. We found that the number of vertices and the average degree of each node are good indicators of the sensitivity of BFS to frequency or CU scaling. This information tends to be available with the graph data itself so no preprocessing of graph input is required for our scheme. For cases in which graph metadata is limited, the dynamic scheme that we discuss in Section 4.2 will have more utility. Intuitively, the higher the node degree, the higher the probability of load imbalance in a workgroup, resulting in performance being dominated by critical threads. Graphs with high variance in node degree tend to have higher average node degree. Conversely, nodes with smaller average node degree also tend to have low variance in node degree and thus

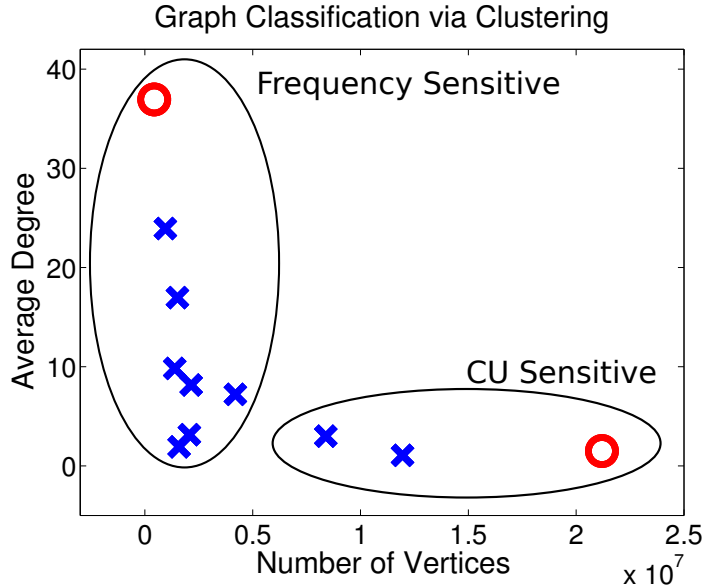


Figure 10: Classification of graph input as frequency- or CU-sensitive

greater amenability to parallelism and less impact from critical threads.

We classify graphs using the k-means clustering algorithm to create two groups of graphs: frequency-sensitive and CU-sensitive. To choose the initial classification, we consider two graphs with vastly different characteristics: *coPapersCiteseer*, which has a small number of nodes and a high average degree, and *hugebubbles-00020*, which has a large number of nodes and a small average degree. There are also the graphs used in Figures 1 and 2. We initially classify *coPapersCiteseer* as frequency-sensitive and *hugebubbles-00020* as CU-sensitive. The low average degree of *hugebubbles-00020* implies that a critical path scenario is less likely to occur and that the threads are more likely to have roughly equal amounts of work. For this case we want to take full advantage of the available parallelism by maximizing the number of active CUs.

Figure 10 depicts the classification of our particular set of input graphs. The circles indicate *coPapersCiteseer* and *hugebubbles-00020*, the graphs we originally classified to initialize the algorithm. The x's indicate the remaining graphs that were classified by k-means clustering. The set of analyzed graphs come from a diverse set of industrial and scientific domains and are dominated by graphs for which BFS is frequency sensitive. This observation agrees well with the power-law findings of Faloutsos et al.: node out degree tends to

have a skewed distribution [14]. Although none of our benchmark graphs had both high degree and large number of vertices due to memory constraints, one might expect that such graphs would benefit from both frequency and CU scaling.

The lesson here is that simple graph properties can provide effective guidance on selecting power efficient configurations for the GPU.

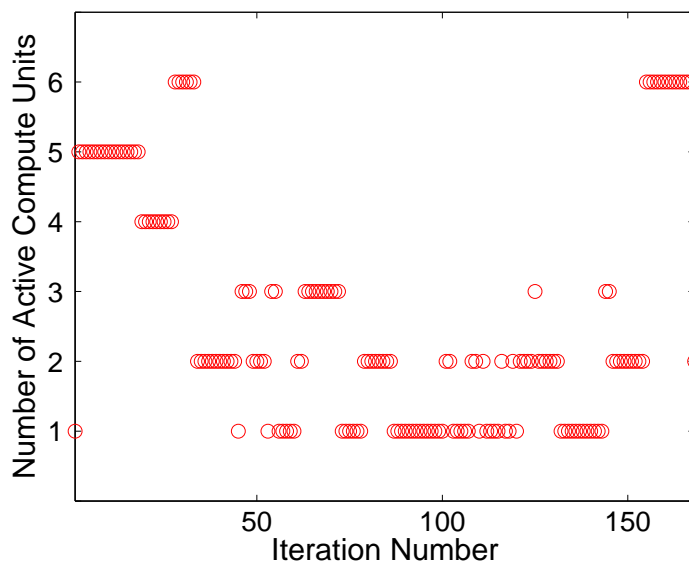
3.3 Analysis of Dynamic Properties

In contrast to setting a single power configuration for the duration of the traversal for each graph, in this section we address the potential of dynamically changing the power configuration for each iteration of BFS. For each BFS iteration for each graph, we record the average power consumption and execution time for each of the 18 DVFS and CU settings on the Trinity APU. Using this data we can explore the utility of dynamically setting the power configuration for each iteration.

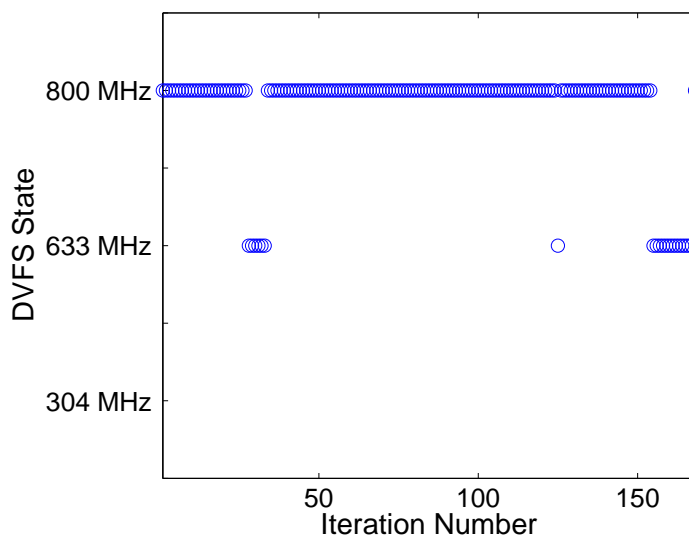
Figures 11 and 12 show how the best CU and frequency settings change over time for the traversal of *ldoor* and *asia.osm*. Note the much larger number of search iterations on the X-axis of Figure 12 for *asia.osm*. These settings are with respect to a power cap of 76% of the maximum power observed for any graph application in the benchmark suite.

For a majority of the iterations in *ldoor*, the 800 MHz setting provides the best performance, implying that *ldoor* is frequency sensitive. Furthermore, the power cap is high enough such that the lowest frequency setting is not needed. In contrast, *asia.osm* utilizes all six CUs for a large majority of iterations as shown on top of Figure 12. In addition, the bottom of Figure 12 shows that maximizing frequency is not necessary for most search iterations for *asia.osm*, implying that this benchmark is CU-sensitive.

Taking both the CU and frequency plots into account simultaneously is even more illuminating. For both graph inputs, there are distinct power-management phases. For example, the iterations that favor 633 MHz on the bottom of Figure 11 correspond to the iterations that favor a larger number of CUs on the top of Figure 11. Similarly for *asia.osm*, the iterations that favor 800 MHz on the bottom of Figure 12 correspond to the iterations that favor a smaller number of CUs on the top of Figure 12. While we classify each of these

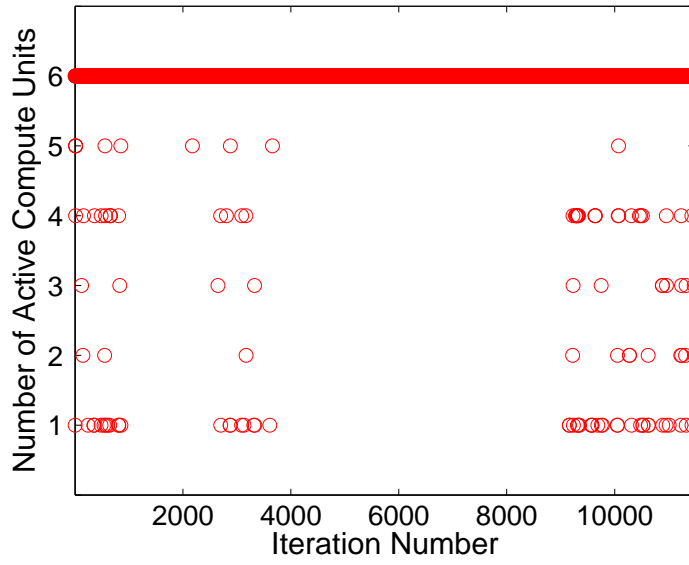


(a) Change in CUs for *ldoor*

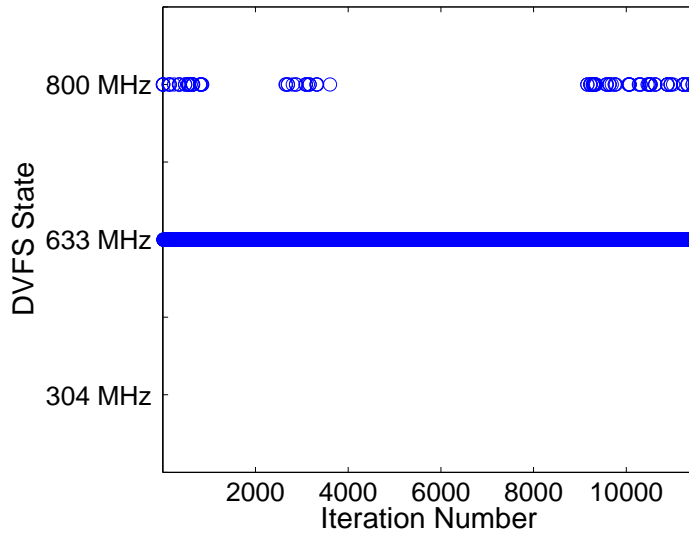


(b) Change in frequency for *ldoor*

Figure 11: Change in most advantageous power state with search iteration for *ldoor*



(a) Change in CUs for *asia.osm*



(b) Change in frequency for *asia.osm*

Figure 12: Change in most advantageous power state with search iteration for *asia.osm*

graphs as *mostly* having a preference for CU scaling or frequency scaling, this preference can change on an iteration basis depending on which portion of the graph is being traversed.

The lesson learned here is that the power behavior can vary in phases across iterations. However, we observe execution distributed between a limited number of power states. It appears that the diversity in the structures of the graph outweighs the diversity in available power states, suggesting that irregular applications can benefit more fine-grained power-management techniques, e.g., a larger number of DVFS states and power configurations.

CHAPTER IV

POWER MANAGEMENT ALGORITHMS

4.1 Static Power Management Algorithm

Given a particular input graph and power cap we would like to develop a systematic methodology for choosing a single hardware power configuration, i.e., one of the 18 possible configurations, that provides the fastest program execution time while remaining under the power cap.

Our static power management algorithm, shown in figure 13, uses the graph classification described in Section 3.2 to determine whether BFS across an input graph is frequency or CU-sensitive based on simple graph metadata. Once this input classification is made, the algorithm then chooses to maximize either the frequency or number of CUs accordingly, using any additional remaining power headroom to maximize the other metric. Note that the *scale_frequency()* and *scale_CUs()* functions operate based on measured power (recalling the experimental methodology in Section 5). Therefore, this analysis (and algorithm) is intended to reflect the best that any static power configuration algorithm can achieve.

Operationally, if it is determined that a particular input graph is frequency-sensitive, the algorithm will attempt to maximize frequency (with the minimum number of active CUs) until either the highest DVFS setting has been reached or the next DVFS state would cross the power cap. If frequency setting is maximized and there remains headroom under the power cap, the algorithm will scale the number of active CUs until all CUs are active or until CUs can no longer be activated due to the power cap. Conversely, if a particular input graph is determined to be CU-sensitive, the algorithm will first maximize the number of active CUs (at the lowest DVFS setting) and then use any remaining headroom to scale the frequency. Note that we can determine if remaining power headroom exists by using digital estimates of power provided by the Trinity APU's power management firmware.

As summarized in Section 5.2, this algorithm provides favorable results for most of our

```

1: function STATIC_BFS_POWER_MANAGEMENT( total_nodes, average_degree)
2:   Class = classify_graph(total_nodes, average_degree)
3:   Setting =  $P_{min}$  #304 MHz, 1 CU
4:   if (Class == frequency_sensitive) then
5:     Setting = scale_frequency()
6:     if (Remaining_Headroom) then
7:       Setting = scale_CUs()
8:     end if
9:   else if (Class == CU_sensitive) then
10:    Setting = scale_CUs()
11:    if (Remaining_Headroom) then
12:      Setting = scale_frequency()
13:    end if
14:  end if
15:  return Setting
16: end function

```

Figure 13: Algorithm for Static Power Management of GPU Graph Traversal

benchmark graphs but is inefficient for one or two graphs, depending on the chosen power cap. In each of the cases we inspected, the algorithm was inefficient whenever it chose to use only one CU. This situation typically occurs when an input is classified as frequency-sensitive and can just barely fit under the power cap at a certain DVFS state, allowing no further headroom for increasing the number of CUs. We consider this affect to be a result of the Trinity architecture as we observed some cases where scaling from two active CUs to one resulted in worse than a 2x performance loss. As such we show results for this technique with the additional constraint of having two active CUs at all times to avoid this scenario. on at all times.

4.2 *Dynamic Power Management Algorithm*

In the previous section we presented a methodology for choosing the best power configuration given a particular input graph and power cap. We now focus on the more general problem of determining the best power configuration at a finer granularity: each iteration of the graph traversal.

If we again consider Figures 1 and 2, we can see that, even for two graphs with significantly different sparsity, the search tends to begin with a small number of nodes per iteration. After this, the graph peaks at some maximum number of nodes towards the middle of the search and finally falls back to a small number of nodes toward the end of the

```

1: function DYNAMIC_BFS_POWER_MANAGEMENT( vertex_frontier, nodes_processed, total_nodes)
2:   Phase = get_progress(nodes_processed, total_nodes)
3:   progress = progress + vertex_frontier
4:   if (Phase == 1) || (Phase == 3) then
5:     Setting = scale_CUs()
6:     if (Remaining_Headroom) then
7:       Setting = scale_frequency()
8:     end if
9:   else if (Phase == 2) then
10:    Setting = scale_frequency()
11:    if (Remaining_Headroom) then
12:      Setting = scale_CUs()
13:    end if
14:  end if
15:  return Setting
16: end function

```

Figure 14: Algorithm for Dynamic Power Management of GPU Graph Traversal

search. If we consider Figure 11, we can estimate that this behavior corresponds to favoring frequency scaling for the middle iterations and CU scaling during initial and final stages. While the bottom of Figure 11 shows that the 800 MHz DVFS state is favorable even at the beginning of the search, this portion of the search is also able to use a larger number of CUs without violating the power cap, as seen in the top of Figure 11. In this section, fewer nodes are being processed simultaneously, meaning that fewer memory references and ALU instructions are executed. As such, less dynamic power is consumed, allowing the activation of more CUs at a higher frequency than is possible in the middle section. Additionally, we conjecture that static power consumption slightly increases from the beginning of the search to the end of the search due to thermal coupling between temperature and leakage power [31].

Given this analysis, we design a simple methodology for dynamically managing power, shown in Figure 14. We distribute the search into three phases: beginning ($Phase = 1$), middle ($Phase = 2$), and end ($Phase = 3$). The phase of the particular iteration is determined by the percentage of nodes that have already been visited (line 2). Once one-third of all nodes have been visited, we transition from the beginning phase to the middle phase. Once two-thirds of all nodes have been visited, we transition from the middle phase to the end phase. We chose uniform phase sizes for simplicity. In our experiments,

changing the transition points to occur slightly sooner or later had a negligible effect. For the beginning and end of the search, when parallelism tends to be limited, we maximize the number of CUs. For the middle of the search, where workload imbalances due to critical paths are more likely to occur, we maximize the frequency.

CHAPTER V

PERFORMANCE EVALUATION

This section presents a quantitative evaluation of static and dynamic power management schemes relative to an oracle across a range of benchmark graphs.

Table 3: Suite of benchmark graphs

Name	Vertices	Edges	Significance
<i>af_shell10</i>	1,508,065	25,582,130	Sheet Metal Forming
<i>asia.osm</i>	11,950,757	12,711,603	Street Network
<i>coPapersCiteseer</i>	434,102	16,036,720	Social Network
<i>delaunay_n23</i>	8,388,608	25,165,784	Random Triangulation
<i>G3_circuit</i>	1,585,478	3,037,674	Circuit Simulation
<i>hugebubbles-00020</i>	21,198,119	31,790,179	2D Dynamic Simulation
<i>in-2004</i>	1,382,908	13,591,473	Web Crawl
<i>kkt_power</i>	2,063,494	6,482,320	Nonlinear Optimization
<i>ldoor</i>	952,203	22,785,136	Sparse Matrix
<i>packing_500x100x100-b050</i>	2,145,852	17,488,243	Fluid Mechanics
<i>rgg_n_2_22_s0</i>	4,194,304	30,259,198	Random Geometric Graph

5.1 Experimental Methodology

All measurements are performed on the integrated GPU of an AMD A10-5800K (“Trinity”) desktop APU. Detailed information about this device can be found in Section 2.3. Changing the DVFS state on this GPU requires sending memory-mapped messages through the GPU driver to the chip’s power-management firmware. The overhead for changing the GPU DVFS state is on the order of a few microseconds. The number of active CUs can also be changed in software, though on the Trinity architecture this change simply *disables* the CUs, meaning that they cannot execute instructions but still consume power. To overcome this obstacle we use a fusing method to power gate the CUs so that their static power is actually reduced. It is noteworthy that even having one active CU activates a significant amount of additional circuitry on the Trinity architecture. In the best case scenario that we observed having one CU active consumed around two-thirds of power that having all

six CUs active consumed. Hence, we presume that CU-scaling on future architectures can have a much more profound effect in saving power.

Digital estimates provided by the power-management firmware are used to measure the power of the GPU [30]. The TDP of the A10-5800K is 100 W, which is significantly higher than the power consumption of even the worst-case BFS input graph for the highest DVFS state with all CUs active. The dynamic nature of parallelism demands and critical paths found within graph algorithms often leads to poor resource utilization and low activity. The hardware does not support power capping. Therefore our analysis methodology involves performing exhaustive measurements on the entire search space in real hardware and post-processing the data for evaluation of power management schemes.

We measure and record for offline analysis the power and execution time at each of the 18 possible power configurations (three DVFS states, up to six active CUs) for each BFS iteration on each input graph. The power measured is for the GPU only, and not for the CPU or the rest of the system. When analyzing this data with respect to a power cap, a configuration is marked ineligible for a given iteration if the average power for the iteration exceeds the power cap. The static oracle looks at the aggregate times of the graph traversal for all eligible configurations and chooses the fastest. The dynamic oracle instead looks at the times of each iteration of the graph traversal for all eligible configurations, choosing the fastest for each iteration.

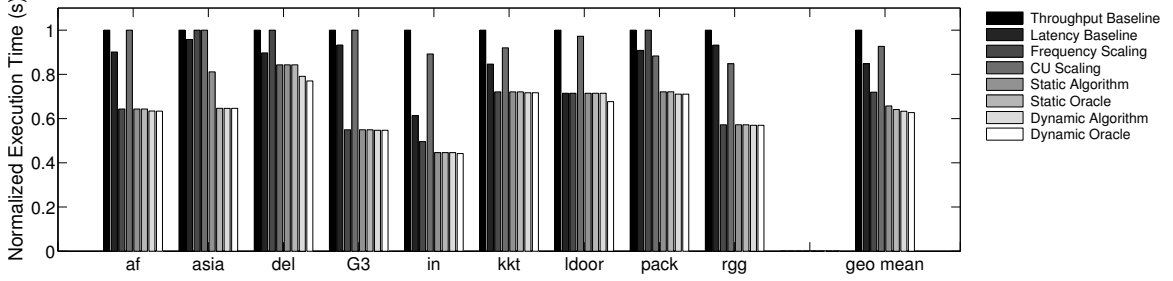
We choose to focus our analyses on power caps that are 62% and 76% of the maximum GPU power that we measured for any of our graph inputs with the highest power settings. These power caps correspond to absolute values of 19W and 23W, which were chosen because they led to the most interesting frequency/CU tradeoffs. Note that these specific values of power are not contrived: the Trinity A8-4555M has a TDP of 19W [31]. We evaluate performance relative to two baseline configurations. The *throughput baseline* configuration is defined by the 304 MHz DVFS state with four active CUs. This baseline configuration emphasizes exploiting parallelism over frequency for performance. The *latency baseline* configuration is defined by the 633 MHz DVFS state with two active CUs. This baseline instead emphasizes exploiting frequency over parallelism. Each baseline configuration can

execute BFS for all input graphs while staying well under the power cap. Power is measured on real hardware at a sampling rate of one millisecond. Typical BFS iteration times observed are on the order of a few milliseconds, so the instantaneous power throughout an iteration is approximately constant. Experiments are run under CentOS Linux 6.4 with AMD Catalyst 13.6 Beta drivers. A fixed-time cool-down period is applied before each run to eliminate any variations in start temperature resulting from variations in initial leakage power. Many iterations of each benchmark graph are run and averaged to eliminate run-to-run variance in our hardware measurements. We omit results for *coPapersCiteseer* and *hugebubbles-00020* because these two graphs were used to train the k-means classification.

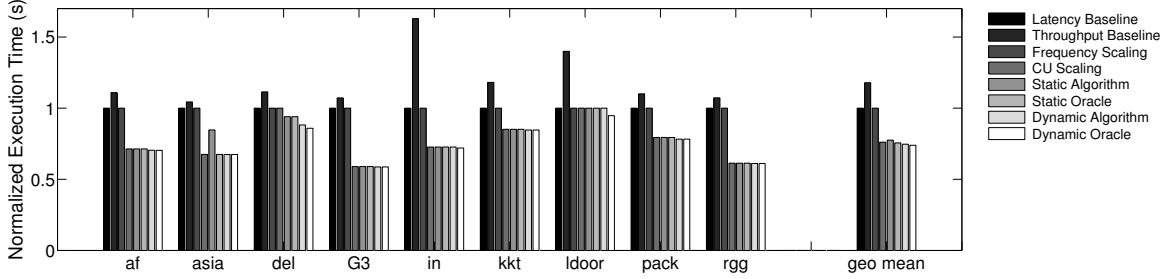
Table 3 shows the set of benchmark graphs used for this study. These graphs are taken from the 10th DIMACS Implementation Challenge [2] and the University of Florida Sparse Matrix Collection [9]. The major limitation in choosing these benchmarks is available GPU memory that can be allocated to a single OpenCL buffer. To overcome this constraint as much as possible, we set the *GPU_MAX_ALLOC_PERCENT* environment variable to the value 100, which allows for nearly all of the GPU memory to be allocated to a single buffer. The benchmarks are chosen to vary in size and connectivity and come from a diverse set of both real-world applications and random graph generators.

5.2 *Experimental Results*

Figure 15 (a) compares the impact of various scaling and power management techniques applied to the throughput baseline for the power cap of 62% of maximum observed GPU power. The execution times are normalized to that of the throughput baseline. The frequency scaling technique attempts to increase the frequency of the throughput baseline when sufficient power headroom exists (keeping the number of CUs constant). Similarly, CU scaling is applied to the throughput baseline if power headroom exists (keeping the frequency constant). Lastly, we compare our power management algorithms with two oracle schemes that choose the best power configuration based on offline profiling either for the entire graph input (static oracle) or for per-iteration of the graph input (dynamic oracle). We also repeat this analysis relative to the latency baseline, as shown in Figure 15 (b).



(a) Throughput baseline



(b) Latency baseline

Figure 15: Comparison of static and dynamic power management techniques

Note that frequency scaling and CU scaling have different interpretations in these two sub-figures. The throughput baseline favors CUs over frequency and as shown in Figure 15 (a), CU scaling has little benefit in comparison to frequency scaling because further scaling CUs requires significantly more power. On the other hand, the latency baseline favors frequency over CUs and as shown in Figure 15 (b), frequency scaling from the medium DVFS state to the high DVFS state cannot be done at all because it requires too much additional power.

The performance of our static algorithm matches that of the static oracle for all benchmark graphs except for *asia.osm*. While the algorithm is able to leverage available power headroom, it chose to maximize the number of CUs at the expense of frequency due to *asia.osm* being classified as CU-sensitive. In doing so, the throughput baseline of 304 MHz with four active CUs is chosen. All other configurations that had four or more CUs violate the power cap. The best configuration for this graph and power cap is 633 MHz with three active CUs, which could have been achieved by applying CU scaling to the latency baseline. The initial classification overlooks the case where scaling both frequency and CUs is preferable over one or the other in isolation.

Considering both the latency and throughput baselines, the static algorithm on average

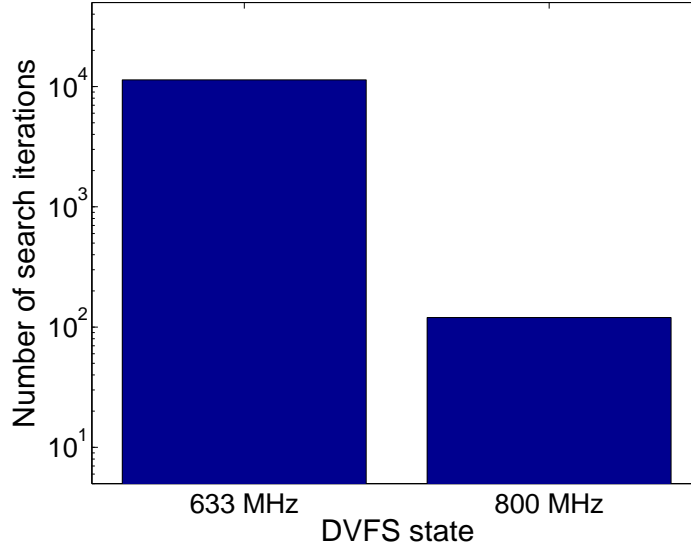


Figure 16: Histogram of search iteration by most advantageous DVFS state for *asia.osm*

leads to execution times that are 15.56% faster than with only frequency scaling and 13.61% faster than using only CU scaling. The key lesson here is that statically determined power configurations that leverage both frequency and the number of CUs enable one to be used at the expense of the other towards better overall configurations, e.g., reducing the number of CUs and then scaling frequency.

Figure 15 also illustrates the advantage of dynamic power management compared to static power management. Recall that the static techniques choose one frequency and CU configuration for the entire graph traversal, while the dynamic techniques choose one such configuration for every iteration of the graph traversal. The dynamic oracle, selects power configurations that improves execution times by 4.62% over that realized with statically determined configurations.

Figures 16 and 17 show histograms of the data presented in Figure 12. Note the logarithmic scale of the y-axes in these figures. Taking the histograms into account with the change in best power configuration over time, it's clear that the setting of 633 MHz and six active CUs is best for an overwhelming majority of search iterations (by two orders of magnitude in both cases). This result implies that a dynamic scheme will be of limited benefit for this particular input graph.

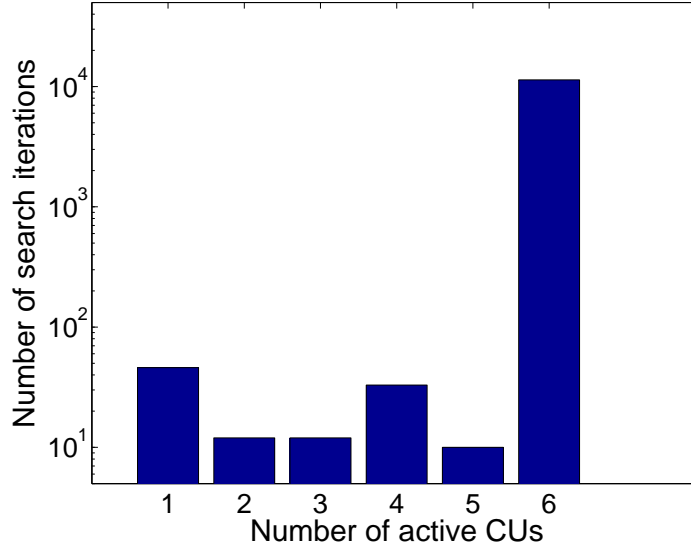


Figure 17: Histogram of search iteration by the most advantageous number of CUs for *asia.osm*

We see two cases where a dynamic approach exhibits significant performance gains compared to a static approach. The first case is when a static algorithm chooses an incorrect configuration, as seen for *asia.osm*. For this graph, both the dynamic algorithm and oracle perform much better than the static algorithm. The second case occurs when the structure of the graph exhibits multiple distinguishable power phases and each phase represents a significant amount of execution time. For the set of benchmarks that we consider, *delau-nay-n23* and *ldoor* fit this description.

Table 4: GPU counter statistics for a few benchmark graphs

Name	Branch Divergence	Std. Dev. Branch Divergence	Load Inst.	Store Inst.
<i>asia.osm</i>	1.00	1.00	0.39	0.42
<i>delau-nay-n23</i>	0.71	0.32	0.32	0.34
<i>kkt-power</i>	0.23	0.73	1.00	1.00
<i>ldoor</i>	0.10	0.14	0.24	0.22

To understand why these particular graphs match this description, we use hardware performance counters to obtain statistics regarding the traversals, as recorded in Table 4. The values in Table 4 are normalized to the graph with the largest values for each column.

The raw values are recorded for each iteration (at the same power setting) for each graph and then were averaged before being normalized.

While both *asia.osm* and *delaunay_n23* have relatively high branch divergence, the branch divergence of *delaunay_n23* is more consistent due to its lower standard deviation (shown in the third column). This consistency in branch divergence implies that threads are constantly executing different instruction streams throughout the traversal. This variation in instruction streams means there is high variety in the number of threads on the critical path and the absolute length of the critical path, both of which present variations in how power should be managed.

While *ldoor* does not exhibit significant branch divergence, it does have a relatively small number of load and store instructions compared to the other benchmark graphs. This result implies that the traversal of *ldoor* spends less time in memory than the other benchmark graphs. Since our techniques for power management do not directly affect the speed of memory transactions, we can expect *ldoor* to have slightly better potential for dynamic power management schemes.

In general, the middle iterations of the search tend to process the largest number of nodes. As explained previously, these iterations are more frequency-sensitive due to potentially large workload imbalances among threads. Hence, our dynamic scheme chooses to maximize frequency for the middle iterations and maximize CUs for the beginning and end of the search. This requires keeping track of how many nodes have been visited in comparison to the total number of nodes in the graph. Overall, our dynamic scheme leads to execution times that are 18.64% faster than using frequency scaling alone and 16.77% faster than using CU scaling alone.

Based on the preceding analysis, we conjecture that dynamic schemes will perform increasingly better on future architectures that support more DVFS states, a larger number of CUs, and more fine-grained methods of power management.

5.3 *Lessons Learned*

This thesis presented a workload characterization of the power implications of GPU graph traversal and used this characterization to develop and analyze static and dynamic power-management algorithms. The following represents a summary of the most important lessons:

- Power management requires making tradeoffs between accelerating critical path computations and exploiting node-level parallelism. Each technique consumes power in different ways. Thus the most effective mix of frequency scaling and CU scaling depends on the structural properties of the graph.
- From the perspectives of parallelism (and therefore power management), the evolution of the graph workloads occurs in three phases - beginning, middle, and end. Each exhibits different amounts of parallelism and sensitivity to frequency and CUs.
- The middle iterations of the traversal tend to process the most nodes and traverse the most edges. For power law graphs in particular, workload imbalances among threads are likely to occur. The performance loss seen from this workload imbalance can be alleviated by increasing the frequency of the GPU, allowing the thread on the critical path to finish more quickly and preventing the other threads from being stalled at synchronization points, idling, and consuming power.
- For the set of graphs we studied, there appears to be more diversity in the workload than in the number of available power states to exploit them. To effectively exploit variations in workload we argue that graph applications in particular, and irregular applications as a whole, will benefit from more power states and fine-grained power management capabilities.

CHAPTER VI

RELATED WORK

There has been a fair amount of work that focuses on power-management algorithms and power analysis for both CPUs and GPUs. However, a significant portion of this research uses simulated environments or was focuses on general power management. Our results, in contrast, are based on hardware measurements and center around GPU performance optimization under a given power cap for graph algorithms.

Lee et al. [23] examine the general-purpose power management of discrete GPUs using GPGPU-sim. Lee and Kim use power-management capabilities similar to ours for optimizing throughput of multicore processors [22]. Isci et al. use a cycle-accurate simulator to show how to use DVFS for global multicore power management in [20]. Leng et al. integrate a configurable and cycle-level power model with GPGPU-sim and study the effect of DVFS and clock-gating in [24]. Keramidas et al. develop performance and power models for superscalar processors with the intention of leveraging DVFS in [21]. Anzt et al. provide an in-depth analysis of power and performance of iterative solvers for sparse linear systems on heterogeneous systems in [1]. Hong and Kim propose an integrated power and performance model for GPUs in [19]. These works establish many basic principles of power management but do not address the unique challenges of irregular applications on GPU processors.

There has also been recent work on GPU implementations of various graph algorithms. Mendez-Lojo et al. develop a GPU implementation based on Andersen’s points-to analysis, which ensures safety for compiler optimizations in [26]. Burtscher et al. provide a workload characterization of irregular GPU programs in [7]. Nasre et al. examine a more difficult subset of graph algorithms that alter the initial structure of the graph by adding or removing nodes and edges in [29]. Hong et al. focus on performance optimization of workload imbalances presented by graph applications on GPUs in [18]. All this work primarily examines performance implications of GPUs and does not examine performance under power

constraints as is done in this thesis.

CHAPTER VII

CONCLUDING REMARKS

In this thesis we have addressed the power-constrained performance optimization of an important class of irregular applications - graph traversal - on GPUs. We first characterized the power behavior of breadth first search, exposing graph characteristics that are sensitive to power consumption expended by scaling frequency or scaling the number of active GPU compute units. These insights were used to propose and study static and dynamic power management strategies for range of benchmark graphs via measurements on a commodity GPU. We found that substantive improvements are indeed feasible - averaging up to $\tilde{18}\%$ reduction in execution time for a given power cap. In addition, the characterizations relate algorithmic properties of graphs to power constrained execution. Utilizing this insight to make power sensitive algorithmic improvements is the subject of ongoing and future work.

Previous work regarding the power management of distributed systems exists [34], though the power characteristics of GPU clusters are not yet well understood. There has also been some recent work on designing distributed algorithms for BFS on clusters of GPUs [4]. Combining these concepts to design power-management algorithms for distributed BFS on clusters of GPUs, especially clusters with heterogeneous nodes, would provide significant insight towards energy utilization for exascale-era and data center class computing systems.

Power-constrained performance analysis of other high performance computing algorithms that also leverage application-specific information to manage power would be another useful area of future work. These studies could be combined with this work and more general-purpose power-management techniques such as the one presented in [23] to gain a quantitative estimate of the advantage that application-specific information gives in terms of energy conservation.

Finally, another potential area for future work is the scheduling issue of determining

whether or not a particular kernel should be executed on the CPU or GPU in heterogeneous architectures, particularly from the perspective of power efficiency. Depending on the performance characteristics of the CPU and GPU in a particular heterogeneous architecture, we surmise that the iterations toward the beginning and end of the search could be a better fit for the CPU than the GPU. This concept is especially interesting in the context of the Heterogeneous System Architecture (HSA), and Uniform Virtual Addressing (UVA) which allows sharing of address spaces between the CPU and GPU. While there has been some research regarding the scheduling of kernels in heterogeneous systems [32], to our knowledge this problem has not been considered in power-constrained environments.

REFERENCES

- [1] ANZT, H., HEUVELINE, V., ALIAGA, J., CASTILLO, M., FERNANDEZ, J., MAYO, R., and QUINTANA-ORTI, E., “Analysis and optimization of power consumption in the iterative solution of sparse linear systems on multi-core and many-core platforms,” in *Proc. of the 2011 Int’l Green Computing Conf. and Workshops (IGCC)*, 2011.
- [2] BADER, D. A., MEYERHENKE, H., SANDERS, P., and WAGNER, D., eds., *Graph Partitioning and Graph Clustering - 10th DIMACS Implementation Challenge Workshop, Georgia Institute of Technology, Atlanta, GA, USA, February 13-14, 2012. Proc.*, vol. 588 of *Contemporary Mathematics*, American Mathematical Society, 2013.
- [3] BELL, N. and GARLAND, M., “Implementing sparse matrix-vector multiplication on throughput-oriented processors,” in *Proc. of the Conf. on High Performance Computing Networking, Storage and Analysis (SC)*, 2009.
- [4] BERNASCHI, M., BISSON, M., MASTROSTEFANO, E., and ROSSETTI, D., “Breadth first search on apenet+,” in *Proc. of the Conf. on High Performance Computing, Networking, Storage and Analysis (SC), 2012 SC Companion.*, 2012.
- [5] BORKAR, S. and CHIEN, A. A., “The future of microprocessors,” *Commun. ACM*, vol. 54, pp. 67–77, May 2011.
- [6] BRANOVER, A., FOLEY, D., and STEINMAN, M., “AMD fusion APU: Llano,” *IEEE Micro*, vol. 32, pp. 28–37, Mar. 2012.
- [7] BURTSCHER, M., NASRE, R., and PINGALI, K., “A quantitative study of irregular programs on GPUs,” in *2012 IEEE Int’l Symp. on Workload Characterization (IISWC)*, 2012.
- [8] DANALIS, A., MARIN, G., MCCURDY, C., MEREDITH, J. S., ROTH, P. C., SPAFFORD, K., TIPPARAJU, V., and VETTER, J. S., “The scalable heterogeneous computing (SHOC) benchmark suite,” in *Proc. of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU)*, 2010.
- [9] DAVIS, T. A. and HU, Y., “The university of florida sparse matrix collection,” *ACM Trans. Math. Softw.*, vol. 38, pp. 1:1–1:25, Dec. 2011.
- [10] DEMATTÉ, L. and PRANDI, D., “GPU computing for systems biology,” *Briefings in Bioinformatics*, vol. 11, no. 3, pp. 323–333, 2010.
- [11] DENG, Y., WANG, B., and MU, S., “Taming irregular EDA applications on gpus,” in *Proc. of the IEEE/ACM Int’l Conf. on Computer-Aided Design (ICCAD)*, 2009.
- [12] EDIGER, D., JIANG, K., RIEDY, J., BADER, D. A., CORLEY, C., FARBER, R., and REYNOLDS, W. N., “Massive social network analysis: Mining twitter for social good,” in *Proc. of the 39th Int’l Conf. on Parallel Processing (ICPP)*, 2010.

- [13] ESMAEILZADEH, H., BLEM, E., ST. AMANT, R., SANKARALINGAM, K., and BURGER, D., “Dark silicon and the end of multicore scaling,” in *Proc. of the 38th Annual Int’l Symp. on Computer Architecture (ISCA)*, 2011.
- [14] FALOUTSOS, M., FALOUTSOS, P., and FALOUTSOS, C., “On power-law relationships of the internet topology,” in *SIGCOMM*, pp. 251–262, 1999.
- [15] GREENBERG, A., HAMILTON, J., MALTZ, D. A., and PATEL, P., “The cost of a cloud: research problems in data center networks,” *SIGCOMM Comput. Commun. Rev.*, vol. 39, pp. 68–73, Dec. 2008.
- [16] HARISH, P. and NARAYANAN, P. J., “Accelerating large graph algorithms on the gpu using cuda,” in *Proc. of the 14th Int’l Conf. on High Performance Computing (HiPC)*, 2007.
- [17] HENDRICKSON, B., “Graphs and hpc: Lessons for future architectures,” tech. rep., Sandia National Labs, 2008.
- [18] HONG, S., KIM, S. K., OGUNTEBI, T., and OLUKOTUN, K., “Accelerating cuda graph algorithms at maximum warp,” in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming, PPOPP ’11*, (New York, NY, USA), pp. 267–276, ACM, 2011.
- [19] HONG, S. and KIM, H., “An integrated gpu power and performance model,” in *Proceedings of the 37th annual international symposium on Computer architecture, ISCA ’10*, (New York, NY, USA), pp. 280–289, ACM, 2010.
- [20] ISCI, C., BUYUKTOSUNOGLU, A., CHEN, C.-Y., BOSE, P., and MARTONOSI, M., “An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget,” in *Proc. of the 39th Annual IEEE/ACM Int’l Symp. on Microarchitecture (MICRO)*, 2006.
- [21] KERAMIDAS, G., SPILIOPOULOS, V., and KAXIRAS, S., “Interval-based models for run-time dvfs orchestration in superscalar processors,” in *Proc. of the 7th ACM Int’l Conf. on Computing Frontiers (CF)*, 2010.
- [22] LEE, J. and KIM, N. S., “Optimizing throughput of power- and thermal-constrained multicore processors using dvfs and per-core power-gating,” in *Proc. of the 46th ACM/IEEE Design Automation Conf. (DAC)*, 2009.
- [23] LEE, J., SATHISHA, V., SCHULTE, M., COMPTON, K., and KIM, N. S., “Improving throughput of power-constrained gpus using dynamic voltage/frequency and core scaling,” in *Proc. of the 2011 Int’l Conf. Parallel Architectures and Compilation Techniques (PACT)*, 2011.
- [24] LENG, J., HETHERINGTON, T., ELTANTAWY, A., GILANI, S., KIM, N. S., AAMODT, T. M., and REDDI, V. J., “GPUWattch: enabling energy optimizations in gpgpus,” in *Proc. of the 40th Annual Int’l Symp. on Computer Architecture (ISCA)*, 2013.
- [25] LUO, L., WONG, M., and HWU, W.-M., “An effective gpu implementation of breadth-first search,” in *Proc. of the 47th ACM/IEEE Design Automation Conf. (DAC)*, 2010.

- [26] MENDEZ-LOJO, M., BURTSCHER, M., and PINGALI, K., “A GPU implementation of inclusion-based points-to analysis,” in *Proc. of the 17th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP)*, 2012.
- [27] MERRILL, D., GARLAND, M., and GRIMSHAW, A., “Scalable gpu graph traversal,” in *Proc. of the 17th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP)*, 2012.
- [28] MUNSHI, A., GASTER, B., MATTSON, T. G., and GINSBURG, D., *OpenCL programming guide*. Pearson Education, 2011.
- [29] NASRE, R., BURTSCHER, M., and PINGALI, K., “Morph algorithms on GPUs,” in *Proc. of the 18th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP)*, 2013.
- [30] NUSSBAUM, S., “AMD “Trinity” APU,” in *Hot Chips*, 2012.
- [31] PAUL, I., MANNE, S., ARORA, M., BIRCHER, W. L., and YALAMANCHILI, S., “Cooperative boosting: needy versus greedy power management,” in *Proc. of the 40th Annual Int’l Symp. on Computer Architecture (ISCA)*, 2013.
- [32] RAVI, V. T., BECCHI, M., JIANG, W., AGRAWAL, G., and CHAKRADHAR, S., “Scheduling concurrent applications on a cluster of cpu-gpu nodes,” in *Proc. of the 2012 12th IEEE/ACM Int’l Symp. on Cluster, Cloud and Grid Computing (CCGrid)*, 2012.
- [33] ROTEM, E., NAVEH, A., RAJWAN, D., ANANTHAKRISHNAN, A., and WEISSMANN, E., “Power-management architecture of the intel microarchitecture code-named sandy bridge,” *IEEE Micro*, vol. 32, no. 2, pp. 20–27, 2012.
- [34] ROUNTREE, B., LOWNENTHAL, D. K., DE SUPINSKI, B. R., SCHULZ, M., FREEH, V. W., and BLETSCH, T., “Adagio: making dvs practical for complex hpc applications,” in *Proc. of the 23rd Int’l Conf. on Supercomputing (ICS)*, 2009.
- [35] SUZUMURA, T., UENO, K., SATO, H., FUJISAWA, K., and MATSUOKA, S., “Performance characteristics of graph500 on large-scale distributed environment,” in *Proc. of the 2011 IEEE Int’l Symp. on Workload Characterization (IISWC)*, 2011.
- [36] TOMPSON, J. and SCHLACHTER, K., “An introduction to the opencl programming model,”
- [37] TSUCHIYAMA, R., NAKAMURA, T., IIZUKA, T., ASAHARA, A., MIKI, S., and TAGAWA, S., “The opencl programming book,” *Fixstars Corporation*, vol. 63, 2010.
- [38] U.S. ENVIRONMENTAL PROTECTION AGENCY ENERGY STAR PROGRAM, “Report to congress on server and data center energy efficiency,” tech. rep., USEPA, 2007.
- [39] WEBER, R., GOTHANDARAMAN, A., HINDE, R., and PETERSON, G., “Comparing hardware accelerators in scientific applications: A case study,” *Parallel and Distributed Systems, IEEE Transactions on*, vol. 22, no. 1, pp. 58–68, 2011.