A Power Characterization and Management of GPU Graph Traversal

Adam McLaughlin[†] Indrani Paul[‡] Joseph L. Greathouse[‡] Srilatha Manne[‡]
Sudhakar Yalamanchili[†]

[†]Georgia Institute of Technology
Adam27X@gatech.edu, sudha@ece.gatech.edu

[‡]AMD Research
{indrani.paul, joseph.greathouse, srilatha.manne}@amd.com

ABSTRACT

Graph analysis is a fundamental building block in numerous computing domains. Recent research has looked into harnessing GPUs to achieve necessary throughput goals. However, comparatively little attention has been paid to improving the power-constrained performance of these applications.

Through firmware changes on a state-of-the-art commodity GPU, we characterize the power consumption of Breadth-First Search (BFS) as a function of the structural properties of the graph. We choose to study this algorithm since graph traversals are used as a building block for many other graph analysis applications. Based on this characterization, we propose and evaluate a power management algorithm to maximize *power cap efficiency*, or the performance under a fixed power cap. Across a range of benchmark graphs, we demonstrate power cap efficiency improvements averaging 15.56% on a state-of-the-art GPU.

Categories and Subject Descriptors

G.2.2 [**Discrete Mathematics**]: Graph Theory—*Graph algorithms*; B.0 [**Hardware**]: General—*Power management*

General Terms

Power Management, GPUs, Graphs

Keywords

Breadth-First Search, DVFS, Power Capping

1. INTRODUCTION

Graph analysis is a fundamental building block in numerous computing domains. Areas as diverse as electronic design automation [6], optimizing compilers [12], scientific computing [5], and social networking [7] rely on graph analyses. The explosive growth of graph sizes towards billions of nodes has pushed graph analysis to the forefront of data intensive applications seeking effective high performance computing solutions. Consequently, recent research has attempted to harness the throughput of general purpose graphics processing units (GPUs) for high performance graph analysis.

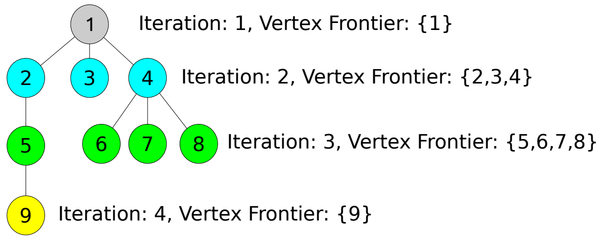
Algorithms for graph analysis possess different computational characteristics than traditional scientific workloads. They exhibit poor memory access locality, and their data structure accesses are pointer intensive, irregular, and difficult to predict [10]. They have low compute densities (the ratio of arithmetic to memory operations) and data dependent control flow that is also difficult to predict. Further, graph applications exhibit significantly time-varying compute and memory system behaviors [6, 9, 11, 13]. Considerable efforts have been devoted to developing fast GPU graph analyses, but little work have been reported for understanding and effectively managing the power consumption of such applications. This paper addresses this gap.

Modern processors are thermally and power constrained. They frequently reconfigure their voltage/frequency state in order to maximize performance without exceeding a power level - the *power cap* - that would cause dangerous temperatures [15]. Commercial processors dynamically allocate power caps to individual SoC components in an attempt to prevent damaging hotspots while also maintaining chip-wide power and thermal limits [16]. As such, it is important to understand how to dynamically reconfigure these components to maximize performance under these power caps.

Using measurements on an AMD A10-5800K Accelerated Processing Unit (APU), we explore the power and performance implications of fine grained parallelism in a GPU implementation of breadth first search (BFS). In particular, we are interested in how the properties of the graph being traversed influence power consumption using two power management techniques - i) Dynamic Voltage and Frequency Scaling (DVFS) and ii) core scaling - changing the number of active GPU compute units (CUs). Our study exposes a fundamental trade-off between frequency and parallelism when maximizing performance under a fixed power cap. We chose BFS since graph traversals are a primitive found in many graph analysis applications, and BFS is the kernel for the Graph 500 benchmark suite [14]. It is representative of a class of emergent, data intensive, high performance computing applications.

This paper seeks to make the following contributions:

- Through firmware changes on a state-of-the-art GPU, we measure the power consumption of BFS over a range of graphs as a function of frequency and number of active GPU cores.
- We present a characterization of the impact of two power management techniques on the performance of



Iteration: 1, Vertex Frontier: {1}
Iteration: 2, Vertex Frontier: {2,3,4}
Iteration: 3, Vertex Frontier: {5,6,7,8}
Iteration: 4, Vertex Frontier: {9}

Figure 1: A simple graph traversed in breadth-first order

BFS - specifically dynamic voltage frequency scaling (DVFS) of the GPU and scaling the number of active GPU cores. We believe this is the first such reported analysis.

- Using the above-mentioned measurement data, we relate the power consumption behavior of BFS to the structural properties of a graph.

- Based on the preceding, we propose and evaluate a power management algorithm to maximize *power cap efficiency*, which we define as performance under a fixed power cap. Across a range of benchmark graphs, we demonstrate that in comparison to employing only frequency scaling or only GPU core scaling, we can provide additional execution time improvements by employing both - an average of 15.6% and 13.6% over each, respectively.

2. BACKGROUND

This section provides relevant background on the specific choice of traversal algorithm (BFS), the target heterogeneous processor utilized in this work, and details regarding our experimental setup used throughout the paper.

2.1 Breadth-First Search

Graph traversal represents an important class of algorithms used in a wide variety of fields. For example, betweenness centrality, a popular graph analytic used to determine influential people in social networks, requires a graph traversal starting from every node in the graph, making graph traversal a dominant part of the algorithm's overall execution time. Breath-First Search (BFS) is an important general graph traversal algorithm that often serves as the framework for the design of customized traversal techniques.

Figure 1 illustrates the progression of BFS for a simple graph. This example demonstrates the time-varying parallelism exhibited by BFS across iterations and its dependency 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 also varies 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 work heavy thread as the *critical thread*. We later show that these behaviors can have a significant impact on power cap efficiency.

DVFS State	Frequency	Voltage
High	800 MHz	1.275 V
Medium	633 MHz	1.2 V
Low	304 MHz	0.9375 V

Table 1: GPU DVFS states for the AMD A10-5800K APU

In this paper, we utilize the parallel algorithm presented by Luo et al. [11] from the Scalable Heterogeneous Computing (SHOC) benchmark suite [4]. To the best of our knowledge, it is the fastest available OpenCL BFS implementation, with asymptotically optimal $O(m + n)$ linear time complexity. Graph data is stored in compressed sparse row (CSR) format [2]. In this implementation, atomic operations are used to prevent data races and prevent duplicates from being inserted into the next frontier.

2.2 The GPU Hardware Platform

The AMD A-Series Accelerated Processing Unit (codenamed "Trinity") used in this study contains two out-of-order dual-core CPU Compute Units (CUs), a graphics processing unit (GPU), and shared logic such as the memory controller. The GPU consists of 384 AMD Radeon$^{\text{TM}}$ cores, each capable of one single-precision fused multiply-add computation per GPU cycle. The GPU is organized as six SIMD units (also known as compute units or CUs) each containing 16 processing units that are each four-way VLIW. More details regarding this processor can be found in [3].

The GPU is on a separate power plane from the CPUs, which allows its voltage and frequency to be independently controlled. Table 1 shows the DVFS states for the GPU in the AMD A10-5800K APU; we will refer to these states throughout the rest of this paper [15]. Although all CUs in a GPU share the same voltage plane, individual GPU CUs can be power gated through software accessible registers.

The current state-of-the-art in GPU power management is the application of DVFS. In this paper we also consider scaling the *number of active GPU CUs* via power gating. We analyze the value of scaling *both* frequency and the number of active CUs in a GPU. Our power management goal is to maximize performance subject to the power cap using both management techniques. Finally, we refer to a *power configuration* as the combination of a DVFS state and the number of active CUs in the GPU. There are six SIMD CUs and three GPU frequencies in the AMD A-series APUs, resulting in a total of 18 power configurations.

2.3 Experimental Methodology

Changing the DVFS state on the 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, which is notably smaller than the time required to execute a typical search iteration.

Digital estimates provided by the power-management firmware are used to measure the power of the GPU [15]. Since the hardware does not directly support power capping, our analysis methodology involves performing exhaustive measurements on the entire search space in real hardware and post-processing the data to evaluate the consequences of the 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

Name	Vertices	Edges	Significance
af_shell10	1,508,065	25,582,130	Sheet Metal Forming
asia.osm	11,950,757	12,711,603	Street Network
coPapersCiteseer	434,102	16,036,720	Social Network
delaunay_n23	8,388,608	25,165,784	Random Triangulation
G3_circuit	1,585,478	3,037,674	Circuit Simulation
hugebubbles-00020	21,198,119	31,790,179	2D Dynamic Simulation
in-2004	1,382,908	13,591,473	Web Crawl
kkt_power	2,063,494	6,482,320	Nonlinear Optimization
ldoor	952,203	22,785,136	Sparse Matrix
packing_500 x100x100-b050	2,145,852	17,488,243	Fluid Mechanics
rgg_n_2_22_s0	4,194,304	30,259,198	Random Geometric Graph

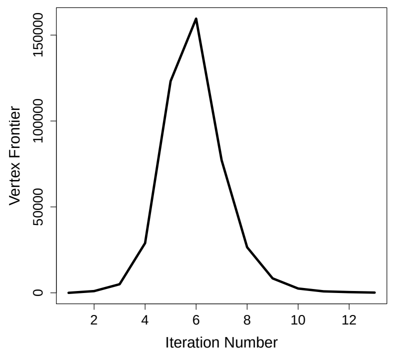
Table 2: Suite of benchmark graphs

iteration on each input graph. The power measured is for the GPU only and not the CPU nor the rest of the system. CPU power consumption was relatively constant regardless of input since the graph traversal is entirely offloaded to the GPU. 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 oracle looks at the execution times of the graph traversal for all eligible configurations and chooses the single configuration for all iterations that yields the fastest time.

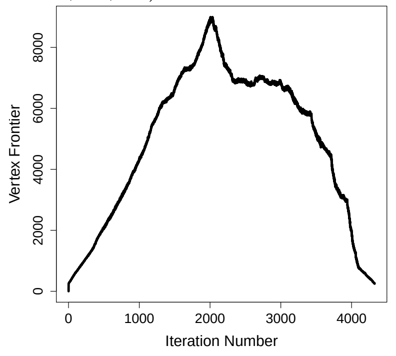
We choose this offline methodology because we cannot change existing hardware management algorithms. In the future, the algorithms we describe could run in the same power management firmware that currently assigns power caps to the components of an SoC. A GPU with a particular hardware configuration would use some amount of power to run BFS kernel iterations. As the firmware observes this power usage, it could use the algorithms we will describe to dynamically change the GPU's hardware configuration in order to maximize performance while meeting the power limit. This future hardware could not use our oracle, but we will later describe an algorithm that could be used online.

We focus our analyses on a power cap that is approximately 62% of the maximum GPU power dissipated on this processor, i.e. the thermal design power (TDP). The methodology is essentially to emulate processors with different TDP values (or components with different thermally-driven dynamic power caps) using the AMD A10-5800K APU. The choice of the actual power cap value was motivated by a design point with enough power for useful throughput while being low enough to be challenging and thus resulting in useful levels of power efficiency. Note that the absolute value of this power cap is not artificial: it is within the range of GPU power limits in mid-range laptop chips.

We evaluate performance relative to two baseline configurations that never exceed this power cap. The *throughput baseline* is defined as the 304 MHz DVFS state with four active CUs. This baseline configuration emphasizes exploiting parallelism over frequency for performance. The *latency baseline* is defined as the 633 MHz DVFS state with two active CUs. This baseline instead emphasizes exploiting frequency over parallelism. 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



(a) *coPapersCiteseer* (n=434,102, m=16,036,720)



(b) *hugebubbles-00020* (n=21,198,119, m=31,790,179)

Figure 2: Workload over time for two input graphs

the instantaneous power throughout an iteration is approximately constant. Experiments are run under CentOS release 6.4 with AMD Catalyst^{TM} 13.6 Beta drivers. A fixed-time cool-down period is applied before each run to eliminate any variations in leakage power resulting from different initial temperatures. Many iterations of each benchmark graph are run and averaged to eliminate run-to-run variance in our hardware measurements.

Table 2 shows the set of benchmark graphs used for this study. These graphs are taken from the 10th DIMACs Implementation Challenge [1].

3. WORKLOAD CHARACTERIZATION

This section provides a characterization of the power consumption of graph traversal. We focus on how properties of the graph being traversed influence power consumption using the two power management techniques - i) Dynamic Voltage and Frequency Scaling (DVFS), and ii) CU scaling, or adjusting the number of active CUs.

3.1 Sensitivity to Graph Structure

The workload experienced by graph traversal algorithms is highly input-dependent. Consider Figure 2, which illustrates variation in the size of the vertex frontier for two example graphs as a function of the iteration number. These graphs are representative of two broad categories of graphs that we study in this work. The first category, represented by *coPapersCiteseer*, corresponds to graphs where 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

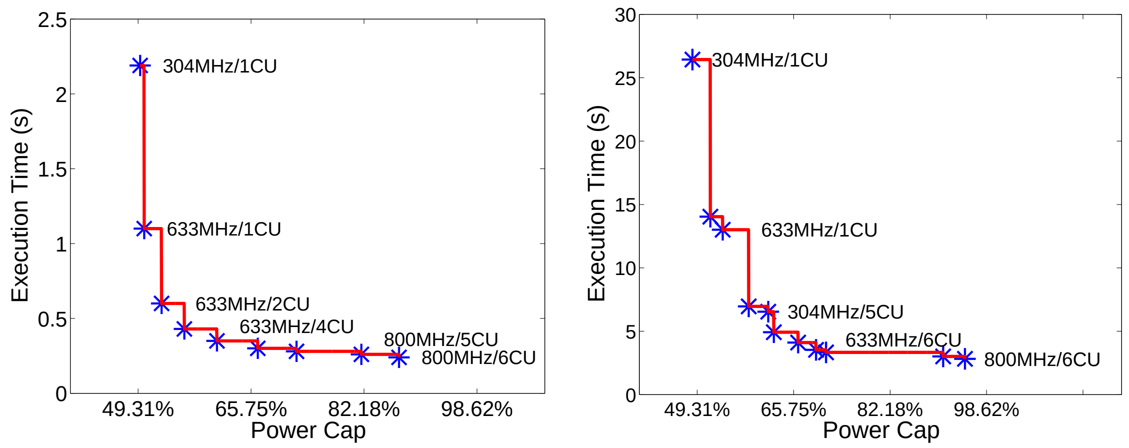


Figure 3: Optimal power configuration for *G3_Circuit* and *delaunay_n23* as a function of power cap

to a large number of nodes). Power law graphs, such as web crawls of the Internet, tend to fall into this category [8]. 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 simulation of physical phenomena or road networks tend to fall into this category [1].

To characterize the power efficiency of BFS, we exhaustively measured the BFS execution times for each of the 18 possible power configurations on the AMD A10-5800K APU, recording the average power consumption as well as the elapsed time. These measurements were repeated for a range of power caps that were fixed as a percentage of TDP. We define the best power configuration for each power cap to be that with the smallest execution time that has an average power consumption that is less than the power cap.

Figure 3 shows the behavior and makeup of the best power configuration as a function of the power cap. Consider the power cap of 82% of the maximum GPU power; for the *G3_Circuit* graph input, the best configuration under this power cap is the high-frequency DVFS state with five active CUs. In contrast, it is the mid-frequency DVFS state with six active CUs that is optimal for the *delaunay_n22* graph input. This implies that the execution behavior of BFS on the former graph is more sensitive to frequency while on the latter graph, it is more sensitive to number of active CUs.

3.2 Graph Classification

In this section, we analyze the issues encountered in selecting a single power configuration that provides the best performance for a given power cap for the BFS traversal of a specific input graph. Our goal is to understand the effectiveness of frequency scaling or CU scaling in improving the power cap efficiency of various types of graphs.

Figure 4 shows the execution time for a power cap set to 62% of the maximum observed GPU power. The execution time is normalized to that of the throughput baseline, which is 4 CUs running at low frequency (see Section 2.3). For each graph, with the number of active CUs held at the baseline, frequency is scaled until either the highest DVFS state is reached or the next DVFS state would, at some point, cause the GPU to go over its power cap. Similarly, we also examine varying the number of active CUs while maintaining the DVFS state at the baseline 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.

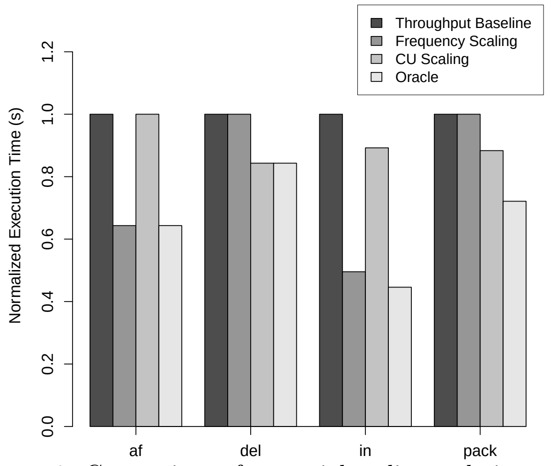


Figure 4: Comparison of potential scaling techniques at a 62% power cap

Results in Figure 4 show that the execution of *delaunay_n23* (del) is more sensitive to CU scaling than frequency scaling; CU scaling alone can match the performance of the oracle. In contrast, the execution of *af_shell10* (af) is more sensitive to frequency scaling rather than CU scaling, since frequency scaling can match 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. 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 waiting at a barrier. This consumes power in a GPU, since all threads in a wavefront must go through the pipeline if any thread still has work to do. Speeding up the critical thread with frequency scaling improves performance, similar to speeding up the serial fraction of parallel programs.

Conversely, for graphs such as *delaunay_n23*, 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

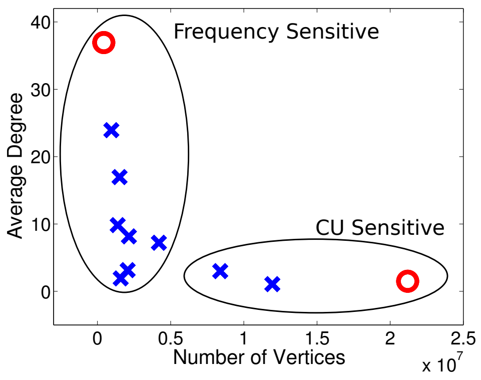


Figure 5: Graph classification as frequency- or CU-sensitive

across nodes in the vertex frontier. Frequency scaling may increase the algorithm's performance for these graphs, but at a disproportionate increase in power (since power usage increases faster than frequency). As such, when operating under a power cap, these type of graphs are better served by added parallelism.

The remaining two cases are especially interesting. For *packing_500x100x100-b050* (pack), increasing CU count provides limited benefit. Reducing the number of CUs slightly from the 4-CU baseline 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* (in) web crawl graph also exemplifies this result; 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.

We found that the number of vertices and the average degree of each node are good indicators of the sensitivity of power cap efficiency to frequency or CU scaling. While complicated metrics such as the diameter of a graph or the distribution of its edges tend to provide additional insight regarding the graph's structure, they rarely offer better results. Intuitively, the higher the node degree, the higher the probability of load imbalance in a workgroup. Additionally, this information tends to be available with the graph data itself such that no preprocessing of graph input is required for our scheme.

We classify graphs using the k-means clustering algorithm to create two groups of graphs: frequency-sensitive and CU-sensitive. Figure 5 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 (and shown in Figure 2). 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.

4. POWER MANAGEMENT ALGORITHM

We propose and evaluate a power management algorithm for maximizing power cap efficiency - recall that power cap efficiency is defined as performance under a fixed power cap. The algorithm statically chooses a single power configuration for the duration of execution of the traversal.

Algorithm 1: Algorithm for Power Management of GPU Graph Traversal. Run Once per Graph.

```
   Input: Total number of nodes and the average degree of each
          node in the input graph
1  $Class \leftarrow classify\_graph(total\_nodes, average\_degree)$
2  $Setting \leftarrow P_{min}$ #304 MHz, 1 CU
3  if $Class = frequency\_sensitive$ then
4      $Setting \leftarrow scale\_frequency()$
5      if $Remaining\_Headroom$ then
6          $Setting \leftarrow scale\_CUs()$
7  else if $Class = CU\_sensitive$ then
8      $Setting \leftarrow scale\_CUs()$
9      if $Remaining\_Headroom$ then
10         $Setting \leftarrow scale\_frequency()$
11 return $Setting$
```

Our power management algorithm, shown in Algorithm 1, 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 previously measured power (see the experimental methodology in Section 2.3). Therefore, this analysis (and algorithm) is intended to reflect the best that any power configuration algorithm can achieve without changing power configurations at runtime, as the algorithm already knows or can estimate how much power an iteration will use. A runtime or firmware power manager may use this static decision to set the CU count and frequency, per iteration or periodically. In this case, the control algorithm could back away from decisions that are too aggressive.

Operationally, if it is determined that a particular input graph is frequency-sensitive, our algorithm will attempt to maximize frequency first (using the minimum number of active CUs) and then increase the number of active CUs if there is additional power headroom left. The converse is true for CU-sensitive graphs.

5. EXPERIMENTAL RESULTS

Figure 6 compares the impact of various scaling and power management techniques applied to the throughput baseline for a power cap of 62% of maximum observed GPU power. The execution times are normalized to that of the throughput baseline. We omit results for *coPapersCiteseer* and *hugebubbles-00020* because they were used to train the k-means classification. 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). While each baseline configuration can execute BFS for all input graphs while staying under the power caps, the scaling techniques fully utilize any remaining power before reaching the cap. We also repeat this analysis relative to the latency baseline but omit results due to space constraints. Lastly, we compare our power management algorithm with an oracle scheme that chooses the best power configuration based on offline profiling.

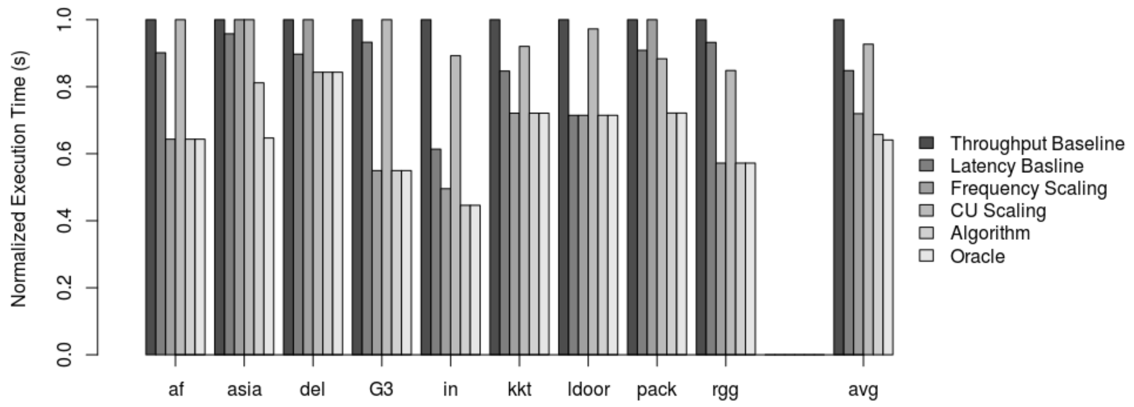


Figure 6: Comparison of power management techniques

The performance of our algorithm matches that of the 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, our algorithm on average leads to execution times that are 15.56% faster than achieved using 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.

6. CONCLUDING REMARKS

In this paper we have addressed the power-constrained performance optimization of an important class of irregular applications - graph traversal - on GPUs. We found that substantive improvements are indeed feasible - averaging 15.56% reduction in execution time for a given power cap. In summary, the most important lessons are:

- 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.

- For scale-free 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 and idling.

- When the graphs are more structured, i.e., have a larger diameter, exploiting parallelism is more power efficient than higher frequency operation.

- For the graphs we studied, there appears to be more diversity in the workload than in the number of available power states. To effectively exploit workload variations, we argue that irregular applications will benefit from more power states and fine-grained power management capabilities.

In general, the diversity of compute and memory behaviors seems to demand more diversity in power management capabilities to improve power efficient operation.

7. REFERENCES

[1] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, editors. *Graph Partitioning and Graph Clustering - 10th DIMACS Implementation Challenge*, volume 588 of *Contemporary Mathematics*, 2013.

[2] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC*, 2009.

[3] A. Branover, D. Foley, and M. Steinman. AMD Fusion APU: Llano. *IEEE Micro*, 32(2):28–37, Mar. 2012.

[4] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The scalable heterogeneous computing (SHOC) benchmark suite. In *GPGPU*, 2010.

[5] L. Dematté and D. Prandi. GPU computing for systems biology. *Briefings in Bioinformatics*, 11(3):323–333, 2010.

[6] Y. Deng, B. Wang, and S. Mu. Taming irregular EDA applications on gpus. In *ICCAD*, 2009.

[7] D. Ediger, K. Jiang, J. Riedy, D. A. Bader, C. Corley, R. Farber, and W. N. Reynolds. Massive social network analysis: Mining twitter for social good. In *ICPP*, 2010.

[8] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. In *SIGCOMM*, volume 29, pages 251–262, 1999.

[9] P. Harish and P. J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *HiPC*, 2007.

[10] B. Hendrickson. Graphs and hpc: Lessons for future architectures. Technical report, Sandia National Labs, 2008.

[11] L. Luo, M. Wong, and W.-m. Hwu. An effective gpu implementation of breadth-first search. In *DAC*, 2010.

[12] M. Mendez-Lojo, M. Burtscher, and K. Pingali. A GPU implementation of inclusion-based points-to analysis. In *PPoPP*, 2012.

[13] D. Merrill, M. Garland, and A. Grimshaw. Scalable gpu graph traversal. In *PPoPP*, 2012.

[14] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang. Introducing the graph 500. 2010.

[15] S. Nussbaum. AMD "Trinity" APU. In *Hot Chips*, 2012.

[16] I. Paul, V. Ravi, S. Manne, M. Arora, and S. Yalamanchili. Coordinated energy management in heterogeneous processors. In *SC*, 2013.