2018 IEEE International Parallel and Distributed Processing Symposium

An Energy-Efficient Single-Source Shortest Path Algorithm

Sara Karamati
School of Computational Science and Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332–0250
Email: s.karamati@gatech.edu

Jeffrey Young
School of Computational Science and Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332–0250
Email: jyoung9@gatech.edu

Richard Vuduc
School of Computational Science and Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332–0250
Email: richie@cc.gatech.edu

***Abstract*—We present a novel strategy to control the energy-efficiency of an algorithm from software, which is to make the degree of parallelism dynamically and automatically tunable. The specific algorithm is a variation of delta-stepping for computing a single-source shortest path (SSSP); its available parallelism is highly irregular and strongly input-dependent. Informed by an analysis of these runtime characteristics, we propose a software-based controller that uses online learning techniques to tune parallelism to meet a given target, thereby improving the average available parallelism while reducing its variability. We show experimentally the efficacy of our self-tuning algorithm in managing tradeoffs among performance and power. Our experimental apparatus is based on the SSSP implementation available in the Gunrock GPU library running on an embedded CPU+GPU, whose hardware has GPU core and memory frequency knobs.**

1. Introduction

The performance and power of a parallel algorithm correlate with its parallelism. One major source of inefficiency is wasted idle power, when the algorithm cannot expose enough parallelism and, therefore, does not utilize all available compute units. But when the parallelism required by the algorithm outstrips the power limit of the underlying hardware, most existing techniques reduce power by slowing down the computation. These methods use either hardware techniques, like dynamic frequency and voltage scaling (DVFS), or low-level software mechanisms, such as schedulers in the operating system or a runtime library.

By contrast, this study seeks complementary techniques, or *algorithmic* knobs, to take control of the parallelism of the algorithm. That is, can one build into an algorithm itself parameters that can be used to control parallelism deliberately? Also, can one tune these algorithmic knobs automatically to meet a given parallelism limit?

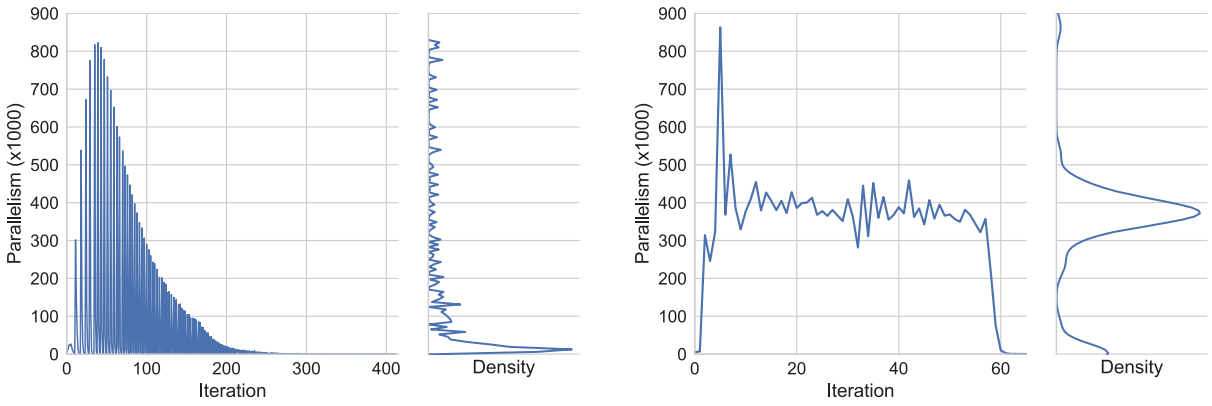
We present a proof-of-concept example, in the context of single-source shortest path (SSSP) computations for weighted graphs (either directed or undirected). We are motivated by SSSP because it is compact enough to study in detail while also having challenging characteristics: its concurrency and resulting performance and power behavior depend on the input graph and can be highly irregular.

The problem posed by irregularity is illustrated by the "parallelism profiles" of Figure 1. Start with Figure 1(a), which shows how the amount of available parallelism within a high-performance baseline SSSP implementation varies as it executes on a scale-free network. During each iteration of the algorithm, shown along the x-axis, the available parallel work varies, shown along the y-axis, where higher means more parallelism. The variability in available parallelism is high. To see that more precisely, consider the adjacent inset, which shows the amount of parallelism as a distribution, labeled "Density" and rotated on its side so the y-axis is the same. This distribution confirms that the amount of parallelism is usually low but also has a long tail; thus, parallelism varies over a large dynamic range. Why is that a problem? First, large amounts of parallelism are associated with large instantaneous power requirements since all computational units are busy. Secondly, high variability wastes time when threads are underutilized, and wastes energy (Joules) as idle cores consume their base power [1]. From this figure, one expects overall performance to be low when the amount of parallelism is low, with large bursts in instantaneous power when parallelism is high.

We propose a new method that changes this kind of runtime behavior. It derives from the baseline method, which is the *near-far SSSP algorithm*, a variation on delta-stepping that was developed and first implemented in the Gunrock library for graph analysis on graphics co-processors (GPU) systems [2]. The near-far method has a natural tuning parameter (Section 3), which a user must select manually and remains fixed throughout the execution. We instead turn this parameter into a knob that can be adjusted *dynamically* and even possibly altered in every iteration. Moreover, we develop a fully automatic controller to set the knob to meet a given concurrency target and therefore a mechanism for controlling dynamic power in the algorithm.

Our method yields profiles like Figure 1(b), which is for the same input graph but using our scheme. The variability is lower, producing a higher and more consistent average over a smaller dynamic range, especially after the initial convergence phase has passed.

1530-2075/18/$31.00 ©2018 IEEE
DOI 10.1109/IPDPS.2018.00117



(a) Baseline Gunrock SSSP

(b) Tunable SSSP

Figure 1. Concurrency profiles for the baseline Gunrock SSSP versus our proposed SSSP algorithm. The latter can reduce the variability in the amount of parallelism, which can reduce power demand and improve performance.

We evaluate our method experimentally against the Gunrock baseline on NVIDIA's TK1 and TX1 platforms. We choose Gunrock because it has one of the best performing SSSP implementations on GPU platforms. Our key results may be summarized as follows:

- We investigate how algorithmic parameters affect variations in both parallelism and power usage; from this study, we determine that delta, a parameter related to how many vertices are processed in each phase, can be varied directly to either increase or decrease parallelism in a particular iteration of SSSP and to maintain a narrow band of average parallelism.
- Building on this link between delta and parallelism, we design and demonstrate a fully automatic controller that can be used to optimize SSSP to affect power and performance tradeoffs.
- We evaluate the efficiency of our new controller operating in tandem with DVFS techniques and show that power savings can be increased by up to 10 % over DVFS, performance can be boosted to 13 % above baseline techniques, or power and performance can both be optimized to achieve a 40 % speedup and 10 % power savings over the baseline algorithm.

2. Related Work

There is a large body of energy and power management techniques from the hardware design, scheduling, and application literature. These techniques work primarily by turning off unneeded hardware resources, making better use of existing hardware, or executing an application in a "race-to-halt" fashion that saves energy by finishing as fast as possible and then turning components off [3]. For GPUs, Mittal and Vetter survey many of these techniques [4]; and for control-theoretic methods (as compared to our algorithmic approach), see the surveys of Abdelzaher et al. and Macías-Escrivá et al. [5], [6]. From these areas, there are at least two notable observations. First, techniques that rely on DVFS or hardware features are often not portable across different architectures, such as GPU accelerators, and can respond slower than algorithmic techniques. This observation further motivates our algorithmic approach. Secondly, control-theoretic and reinforcement learning approaches can offer gains in managing system power, but there is little prior work in mapping control feedback to specific application kernels to affect system power and performance management. Our proposed method helps to address this gap.

Hardware-assisted power management

Hardware-based power management consists largely of DVFS or other techniques for power capping. For GPUs, previous work by Price et al. has shown that application-aware DVFS [7] can provide substantial benefits in energy efficiency while Jiao et al. demonstrate that memory or GPU core frequencies can be matched to memory- or compute-intensive HPC applications to achieve power savings [8].

Hoffmann et al. provide two relevant power capping approaches [9], [10]. The authors argue for the concept of "dynamic knobs" where static input parameters are used as inputs to tune the configuration of the application and to satisfy desired power constraints. Likewise, Zhang and Hoffman advocate a combined hardware and software approach to power capping that combines DVFS with a binary search mechanism to maximize performance while staying under the desired cap [10].

Scheduling and work reduction

Most recent related work on energy-efficient computing has focused on efficient task scheduling to provide average power savings across multiple compute devices, whether they be CPU and GPU or a cluster of processors. The most complete methods are part of Charm++, which includes an integrated runtime system for power-aware runtime management (PARM) [11] and Garzon et al. [12], which proposes adaptive scheduling and load balancing heuristics for CPU-GPU systems.

A separate subset of scheduling research involves using power-down scheduling to turn off unused resources when

they are idle by power-gating during idle periods [13], selecting the optimal set of cores with semi-analytical models [14], and offline profiling to estimate the optimal number of powered-down cores [15]. It should be noted that until the most recent generations of hardware, most GPUs have had limited low power states [16].

Application and algorithm-specific optimizations

For GPU-based applications, application-based power savings are generally gained by making substantial changes to an application to reduce data movement and work imbalances between the CPU host threads and GPU acceleration threads. In this vein, kernel fusion allows for combining similar kernels to improve GPU utilization [17], but it can also require complex analysis to combine kernels without oversubscribing avaiable GPU hardware (e.g., shared memory).

At the compiler level, recent work describes a tool that enables programmers to select a variety of throughput optimizations on an intermediate representation of CUDA-based graph algorithms including SSSP, with speedups from $1\times$ to $15\times$ over Gunrock, depending on the type of input graph (RMAT, road networks) [18]. Our method complements compiler techniques.

Other application-based modifications look at commonalities between GPU codes and software level optimizations. These techniques suggest code practices like tweaking kernel dimensions and memory reuse patterns to reduce hardware underutilization [19] and correlate software optimizations like loop unrolling and data transformations with their effects on power consumption [20]. However, these types of changes require a good understanding of the underlying type of GPU hardware to achieve the best results.

The k-level asynchronous (KLA) model provides a tunable software knob, k, that is billed as a generalization of delta-stepping for combining level synchronous, parallel graph algorithms with asynchronous execution in a multi-node environment [21]. The method is adaptive, determining a constant optimal value of k across all iterations to provide scalable performance. KLA complements our proposed technique, and, moreover, does not focus on combined energy conservation and performance. Furthermore, it assumes a single optimal and universal value of k, in contrast to our iteration-by-iteration tuning of our analogous parameter (delta).

More recent work has looked at the concept of "self-aware systems" or self-tuning algorithms [6]. Currently, a limited set of work at the algorithmic level focuses on improving performance via self-aware tuning [22] with some limited additional work in managing system-level power using reinforcement learning techniques [23]. Notably, Das et al. extend a Q-learning-based DVFS technique with heuristic based thread selection to create a two-level runtime that can reduce energy usage by $15\,\%$ over the baseline on an NVIDIA Jetson board [24].

The work in this paper differs from these previous control-based techniques in that it explicitly focuses on a specific application kernel rather than on system power or higher-level aspects of the application like thread scheduling. It may be possible to combine prior work and our methods synergistically.

3. Baseline: Near+Far SSSP

We take as our baseline the work-efficient *near+far SSSP algorithm* of Davidson et al., implemented in the Gunrock graph processing library for GPUs [2], [25]. Gunrock is a state-of-the-art high-performance graph library and supports a wide range of computational primitives for traversals (e.g., breadth-first search), node-ranking (e.g., HITS, SALSA, PageRank), and global properties analysis (e.g., connected components, minimum spanning tree). In Gunrock, all primitives maintain a frontier of nodes or edges consisting of parallel work; each primitive then performs on the frontier a sequence of basic operations, such as expansion, exploration, and filtering. Our approach to controlling parallelism will rely on an analysis and direct manipulation of this frontier.

We picked Gunrock in part to have a concrete baseline for our experimental results although there are other approaches, including compiler-based frameworks [18]. In any case, our specific modeling and controlling techniques for this baseline should be relevant to other graph primitives, libraries, and implementations.

We focus on SSSP because it is a fundamental graph primitive and is also known to be one for which it is hard to develop a fast and practical work-efficient parallel algorithm. And per Figure 1, SSSP has irregular parallelism, which varies by input graph. These traits make SSSP a challenging case study for developing a concurrency control technique.

3.1. Algorithm overview and notation

The baseline near+far SSSP algorithm derives from the delta-stepping method of Meyer and Sanders [26]. In particular, it is parameterized by a "delta value" (δ), which is a parameter for prioritizing which vertices to visit next. The near+far method maintains a frontier of vertices to visit, initialized with the source vertex. It also maintains a current estimate of the shortest path length for each vertex. The value of δ serves a threshold for partitioning the vertex frontier into a *near-queue*, which becomes the new frontier in the next iteration, and a *far-queue*, which holds candidate frontier vertices that are heuristically being "postponed" for visiting in future iterations once the near-queue is exhausted. The algorithm is iterative; we use k throughout this paper to denote the current iteration. Each iteration executes a sequence of four stages named **advance**, **filter**, **bisect-frontier**, and **bisect-far-queue**. The advance and filter stages update distances of each frontier vertex from the source. The bisect-frontier and bisect-far-queue stages then update the frontier, prioritizing which vertices to visit next based on their current distance estimates and the value of δ. More specifically, each stage operates as follows:

1) **advance**: The advance stage explores all of the outgoing edges of the current frontier, updating the distances from the source to each endpoint if the corresponding edge leads to a shorter path than the current estimate. After advance, the new frontier consists only of vertices with updated distances.

2) **filter**: Since advance processes nodes in the frontier in parallel, it is possible that different nodes in the frontier adjacent to the same node will redundantly add that node to the new frontier. The filter stage removes such duplicates from the frontier and ensures correct (shortest) distances are recorded. The Gunrock implementation of this stage uses an atomic-min operation and a bitmap array associated with the frontier to find the minimal distance value and to remove redundant vertices from the frontier.
3) **bisect-frontier**: The algorithm maintains a current phase, denoted by i, and only vertices with distance between $i\delta$ and $(i+1)\delta$ from the source vertex will remain in the frontier to be processed in the next iteration. Specifically, if the updated distance is greater than $(i+1)\delta$, then bisect-frontier appends the corresponding vertices to a far-queue to be processed later.
4) **bisect-far-queue**: When there are vertices with distances between $i\delta$ and $(i+1)\delta$ in the frontier, the algorithm returns to stage one, advance. Otherwise, the bisect-far-queue stage adds all the vertices in the far queue with distances between $(i+1)\delta$ and $(i+2)\delta$ to the frontier.

This sequence repeats until no vertices remain.

Our proposed method monitors and attempts to control the frontier size using dynamic analysis and modeling of the inputs to each of the four stages. As such, we summarize how the frontier changes and introduce some notation, which we use in Section 4 in the description of our scheme.

The advance stage receives as input the current frontier, whose size (number of vertices) we denote by $X_k^{(1)}$. This stage distributes these vertices among GPU threads to produce in parallel an updated frontier as output. It visits all edges adjacent to the frontier vertices; these edges are load balanced by dividing a frontier vertex's neighbor list among GPU thread blocks such that each block receives approximately equal number of edges. In fact, the number of vertices in the neighbor list determines the load to be processed in each iteration by each block and is closely related to the available parallelism in each iteration. As its output, advance creates the updated frontier, which is roughly the same as the neighbor list of the processed vertices. Let $X_k^{(2)}$ denote the number of vertices in the updated frontier after advance. This number has a direct impact on the performance of this kernel and subsequent kernels.

Unlike advance, the filter, bisect-frontier, and bisect-far-queue stages balance the load by distributing the input frontier vertices among GPU threads. Consequently, the parallelism of these stages is proportional to their input frontier size. The input to filter is the output of advance, so the amount of parallelism within filter depends on $X_k^{(2)}$. Since the main task of filter is to remove redundant vertices, its output will be smaller than its input. Let $X_k^{(3)}$ denote the number of vertices in the output of filter; then $X_k^{(3)} \leq X_k^{(2)}$. Thus, controlling $X_k^{(2)}$ is critical to controlling the parallelism in the first three stages. The inputs to the fourth stage, bisect-far-queue, are the current frontier, whose size is $X_k^{(4)}$, and the far queue. Thus, the size of the far queue also affects the amount of parallelism in this stage.

TABLE 1. DATA SET CHARACTERISTICS

Input graph	Nodes	Edges	Max degree
Wiki	1 634 989	19 735 890	4970
Cal	1 890 815	4 630 444	7

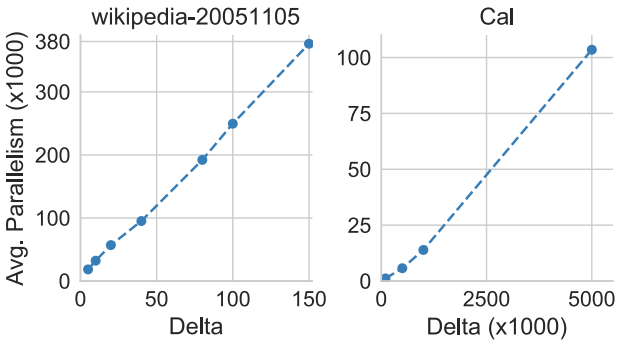


Figure 2. Delta versus Parallelism

3.2. Experimental analysis of the baseline

The choice of δ, affects the available parallelism and thus the overall execution time and power consumption. The following experiments demonstrate this claim.

Experimental setup

We evaluate the algorithms on two of NVIDIA's embedded CPU-GPU platforms, namely Jetson TK1 and TX1. The older TK1 system contains an NVIDIA Kepler GPU with 192 CUDA Cores and an ARM Cortex™-A15 CPU while the newer TX1 contains a Maxwell GPU with 256 CUDA cores and a quad-core ARM Cortex™-A57 CPU. The Gunrock library code is written in CUDA C++, and the NVCC compiler with `-O3` flag is used to compile the code. The data sets used in our experiments are a road network, Cal [27], and a scale-free network, wikipedia-20051105 (hereafter, Wiki), taken from the UF sparse matrix collection [28]. We chose these because they differ considerably in their characteristics. The Cal, which is part of the DIMACS Shortest Path Challenge, has a high diameter and low degree; by contrast, Wiki, is a hyperlink network with high-degree and low-diameter. For Wiki, the edge weights are uniform random integers between 1 and 99. A few salient properties of these two graphs also appears in Table 1.

To measure power, we use the PowerMon device [29]. It measures system-level power for the entire board. PowerMon works by measuring DC current across a resistor and transmits data to a host via USB at a rate of up to 1 kHz per channel. The TK1 system uses one 12 V input, and PowerMon is placed in the input path by splitting a wall adapter and connecting the leads to the PowerMon pins.

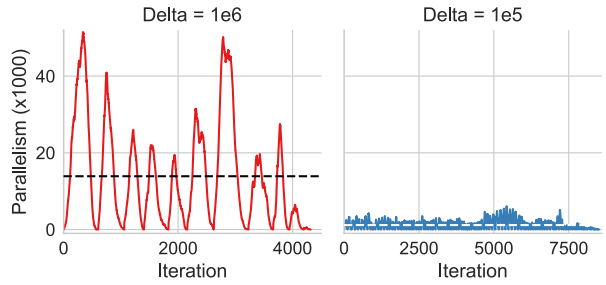


Figure 3. Cal (road network) performance versus delta.

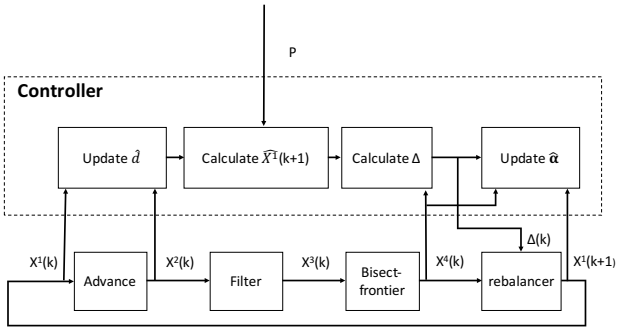


Figure 4. A dynamically adaptive (or "tunable") near+far algorithm

Observations

As part of our algorithmic tuning work, we investigate the effects of variations in δ on parallelism and consequently on algorithm performance and power consumption with the standard Gunrock SSSP algorithm. Figure 2 shows the amount of parallelism as delta is varied for each data set. Average parallelism is defined as the average frontier size ($X_k^{(2)}$) over all iterations. For small values of δ, the number of vertices to process in each phase is small, which limits the amount of potential parallelism. As δ increases, the parallelism increases. As such, δ is an important factor for balancing the parallelism.

Figure 3 shows the effect of δ on the number of nodes in the frontier. A small δ results in sub-par parallelism, and consequently, longer running time. As δ increases, the peak parallelism, as dictated by the size of the frontier, grows proportionally, resulting in a reduced number of iterations.

4. A Self-Tuning Near+Far Algorithm

Our proposed scheme improves on the baseline near+far SSSP algorithm in two ways. First, we treat δ not as a static parameter that is fixed for the duration of the algorithm but rather one that can be tuned dynamically in every iteration. This change is significant as it gives us a mechanism to throttle fine-grained parallelism. Secondly, the baseline method requires a user to select δ *a priori*. But as far as we are aware, there is little guidance on precisely how to do so. By contrast, our controller algorithm can make this choice fully automatically.

The input to our scheme is a *parallelism set-point*, which is the desired amount of parallelism. We denote the set-point by P: by choosing its value, a user says she wishes for the amount of parallelism to converge to values at or below P. Choosing P is in principle easier and more intuitive than selecting δ. In particular, an optimal value of δ is complicated to determine, as it depends on the input graph and its edge weights. By contrast, P is a function primarily of available hardware resources, so it is possible to create an input-independent "menu" of P values beforehand. These might be based on, for instance, the number of processing elements or the power required per processing element.

4.1. Overview of our approach

In short, our proposed method is an adaptive near+far SSSP algorithm with a feedback loop that dynamically adjusts δ so that the amount of available parallelism converges to values around the set-point P. This controller is software-based and executes on the CPU. We summarize the controller schematically in Figure 4. Compared to the baseline, our controller directly monitors $X_k^{(1)}$, $X_k^{(2)}$, and $X_k^{(4)}$; it then uses these to update internal models, or estimators, which appear Figure 4 as boxes within the controller. These models are used to eventually compute δ_k, a dynamic version of the δ parameter in the baseline algorithm (Section 3). Lastly, the bisect-far-queue stage of the baseline is replaced with a rebalancer stage that uses δ_k to heuristically update the frontier and far queue.

The structure of the controller is as follows. First, the controller maintains two dynamic models, which we refer to as the ADVANCE-MODEL and BISECT-MODEL. The ADVANCE-MODEL learns to estimate $X_k^{(2)}$ from $X_k^{(1)}$, and BISECT-MODEL learns to estimate $X_k^{(1)}$ from $X_{k-1}^{(4)}$ and δ_{k-1}. Together, the models may be used to estimate values for δ_k to target a desired value for $X_{k+1}^{(1)}$, given the set-point P. We use an online learning algorithm based on stochastic gradient descent (SGD) to iteratively improve the parameters embedded within these models.

The new rebalancer adds vertices to or removes them from the frontier based on δ_k. If the model estimates determine that δ_k should increase, then the rebalancer inspects the far queue for vertices with distances in the target range of the new delta-value and moves them to the frontier. If instead δ_k should decrease then frontier vertices are moved to the far queue. Otherwise, δ_k has not changed and no additional action is needed.

4.2. ADVANCE-MODEL

Recall that in iteration k the advance stage visits all neighbors of nodes in the frontier, the size of which is $X_k^{(1)}$. Intuitively, the expected amount of parallelism during advance would be the average node degree times the frontier size; therefore, letting $\hat{X}_k^{(2)}$ denote an *estimate* of $X_k^{(2)}$, we might posit a model for $\hat{X}_k^{(2)}$ of the form,

$$\hat{X}_k^{(2)} = d \cdot X_k^{(1)}, \tag{1}$$

where d specifies how the cardinality of the output of the advance stage scales with the input frontier size $X_k^{(1)}$. In other words, the model parameter d becomes an estimate of the average degree of nodes in the frontier.

We may ask for a d that minimizes the deviation of $\hat{X}_k^{(2)}$ from $X_k^{(2)}$ by, say, the sum of squared errors,

$$\min_d \sum_k \left(X_k^{(2)} - \hat{X}_k^{(2)}\right)^2 = \min_d \sum_k \left(X_k^{(2)} - d \cdot X_k^{(1)}\right)^2. \tag{2}$$

We use the online method of SGD [30] to update d after observing the true values of $X_k^{(1)}$ and $X_k^{(2)}$ in each iteration k. Algorithm 1 shows this update precisely. Lines 1 and 2 are the first and second order derivatives of the contribution of current iteration to the objective function, i.e., the summand in Equation (2). Line 8 is the main SGD update formula. Lines 3–6 update μ, the learning rate. The $\overline{g}$ and $\overline{h}$ variables are the exponential moving average (EMA) estimates of the first and second order derivatives, respectively; and $\overline{v}$ is the EMA of the uncentered variance of the first order derivative. The EMAs are parameterized with the time constant τ, which line 7 updates to adapt the EMA decay rate according to the size of the steps taken in each iteration [30].

Algorithm 1 stochastic gradient descent (SGD) with an adaptive learning rate [30]. Initially, when $k = 0$, this algorithm takes $X_0^{(1)} = 1$ for the initial source vertex, $\tau = (1+\epsilon) \times 2$, and $\overline{g} = 0, \overline{h} = 1, \overline{v} = \epsilon$.

1: $\nabla_d = -2\left(X_k^{(2)} - d \cdot X_k^{(1)}\right) X_k^{(1)}$
2: $\nabla_d^2 = 2\left(X_k^{(1)}\right)^2$
3: $\overline{g} \leftarrow \left(1-\tau^{-1}\right) \cdot \overline{g} + \tau^{-1} \cdot \nabla_d$
4: $\overline{v} \leftarrow \left(1-\tau^{-1}\right) \cdot \overline{v} + \tau^{-1} \cdot (\nabla_d)^2$
5: $\overline{h} \leftarrow \left(1-\tau^{-1}\right) \cdot \overline{h} + \tau^{-1} \cdot \nabla_d^2$
6: $\mu \leftarrow \frac{(\overline{g})^2}{\overline{h} \cdot \overline{v}}$
7: $\tau \leftarrow \left(1 - \frac{(\overline{g})^2}{\overline{v}}\right) \cdot \tau + 1$
8: $d \leftarrow d - \mu \nabla_d$

4.3. Constraining parallelism

Recall from Section 3 that the available parallelism in the next iteration, $k+1$, will be at most $X_{k+1}^{(2)}$. Thus, we would like $X_{k+1}^{(2)} \leq P$, the target parallelism set-point. From the ADVANCE-MODEL , we can thus estimate that it would be desirable to have an input frontier size of

$$\hat{X}_{k+1}^{(1)} \leftarrow \frac{P}{d}. \tag{3}$$

This target constraint on the input frontier size becomes the goal for the BISECT-MODEL .

4.4. BISECT-MODEL

The BISECT-MODEL is the estimator associated with the rebalancer stage. This stage consumes the frontier, of size $X_k^{(4)}$, and may move vertices from the far queue to the frontier if it is determined that the delta-value needs to change. The updated frontier becomes the input to the next iteration, and its size will be $X_{k+1}^{(1)}$.

Suppose we determine that a change of size $\Delta\delta_k$ to the current δ_k is necessary. A value of $\Delta\delta_k = 0$ means no change is necessary, so that the updated frontier of size $X_k^{(4)}$ is exactly the size of the input frontier in the next iteration, $X_{k+1}^{(1)}$. Otherwise, we estimate $X_{k+1}^{(1)}$ by a linear model of the form,

$$\hat{X}_{k+1}^{(1)} = X_k^{(4)} + \alpha \cdot \Delta\delta_k, \tag{4}$$

where $\hat{X}_{k+1}^{(1)}$ denotes the estimate of $X_{k+1}^{(1)}$ and α is a parameter to be learned. Similar to the ADVANCE-MODEL , we formulate the problem of choosing α as one of minimizing the deviation between $\hat{X}_{k+1}^{(1)}$ and $X_{k+1}^{(1)}$:

$$\min_\alpha \sum_k \left(X_{k+1}^{(1)} - X_k^{(4)} + \alpha \cdot \Delta\delta_k\right)^2. \tag{5}$$

The update formulas are analogous to Algorithm 1 with derivatives taken with respect to α instead of d.

4.5. Updating the delta-value, δ_k

Given d and α, the new δ_{k+1} at iteration $k+1$ may be derived by combining Equations (3) and (4):

$$\delta_{k+1} \leftarrow \delta_k + \frac{\hat{X}_{k+1}^{(1)} - X_k^{(4)}}{\alpha} = \delta_k + \frac{\frac{P}{d} - X_k^{(4)}}{\alpha}. \tag{6}$$

4.6. Rebalancing

To control the parallelism imposed by the size of the far queue, our controller recursively partitions the far queue, based on the vertex distance from the source, to keep the size of each partition below the parallelism set-point P. This is accomplished by periodically updating partition boundaries. Once these boundaries are updated, in each iteration, the bisect-frontier and rebalancer stages place each new vertex into partition i if $B_{i-1} < d \leq B_i$, where d, B_{i-1}, and B_i are vertex distance, lower bound of partition i, and upper bound of partition i, respectively. Note that the upper bound of partition $i-1$ is always equal to the lower bound of partition i; that is, both are equal to B_{i-1}.

The far queue partitioning algorithm starts with two partitions with their upper bounds initialized to *average edge weight* and `MAX_INT`, respectively. In each iteration, the upper bound of each partition is updated. If the size of the current partition is zero, the next partition becomes the current partition. If the updated upper bound belongs to the last remaining partition, a new partition with its upper bound equal to `MAX_INT` is appended to the sequence of existing partitions. Critically, updates to the boundaries only impact the subsequent iterations. However, to preserve the algorithm's correctness, we have to impose monotonic boundary shifts. In other words, the updates to boundaries will always decrease their value. Accordingly, the following modifications are made to bisect-frontier and and bisect-far-queue stages: in bisect-frontier, when the algorithm adds vertices to the far queue, a vertex is assigned to a partition

based on its distance d. In bisect-far-queue, instead of searching all vertices to find ones with the desired distance, only the partitions with the desired boundaries are searched.

The method for selecting boundaries is similar to the model used to find the optimal δ in Section 4.4. The BISECT-MODEL in Equation (4) formulates the relation between $\Delta\delta_k$ and the number of vertices with their distance in the range $[\delta_k, \delta_{k+1}]$. Here, the desired number of vertices in the range dictated by partition boundaries is P. With the lower bound anchored to the upper bound of the previous partition, we need to update the upper bound for the current partition. Therefore, the update formula is:

$$B_i \leftarrow B_{i-1} + \frac{P}{\alpha}. \tag{7}$$

In our experiments, the algorithm described above converged to an acceptable value of α after about 5 iterations. However, an improper parameter estimation, as would occur before the algorithm converges, can make the algorithm unstable during initial iterations. During the initial iterations and before the algorithm stabilizes, to reduce overshoots and undershoots in our estimates of α, the α is estimated based on the current δ and the frontier and far queue size:

$$\alpha = \begin{cases} \frac{X_k^{(4)}}{\delta_k} & \text{if } X_k^{(4)} \geq \hat{X}_{k+1}^{(1)} \\ \\ \frac{S_i}{B_i - \delta_k} & \text{otherwise} \end{cases}, \tag{8}$$

where i and S_i are the index and size of the current far queue partition, respectively.

5. Experimental Results

We evaluated our self-tuning algorithm on the same evaluation platforms and input graphs of Section 3.2.

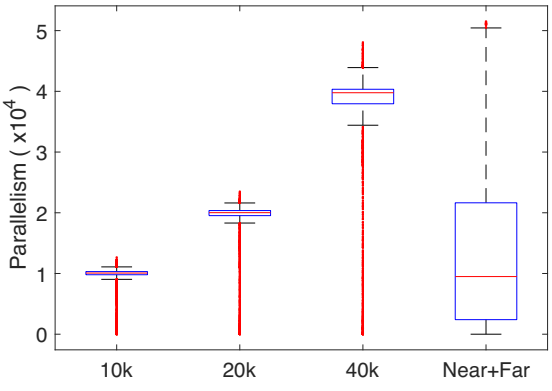


Figure 5. *Efficacy of parallelism control:* The distributions of available parallelism for the self-tuning controller-based algorithm at three values of the set-point (P) compared to a baseline that runs at a time-minimizing δ.

5.1. Efficacy of parallelism control

We assess whether the controller algorithm maintains the available parallelism at a given set-point P. The results for the Cal road network appear in Figure 5. Results for the Wiki network are similar but omitted due to space constraints.

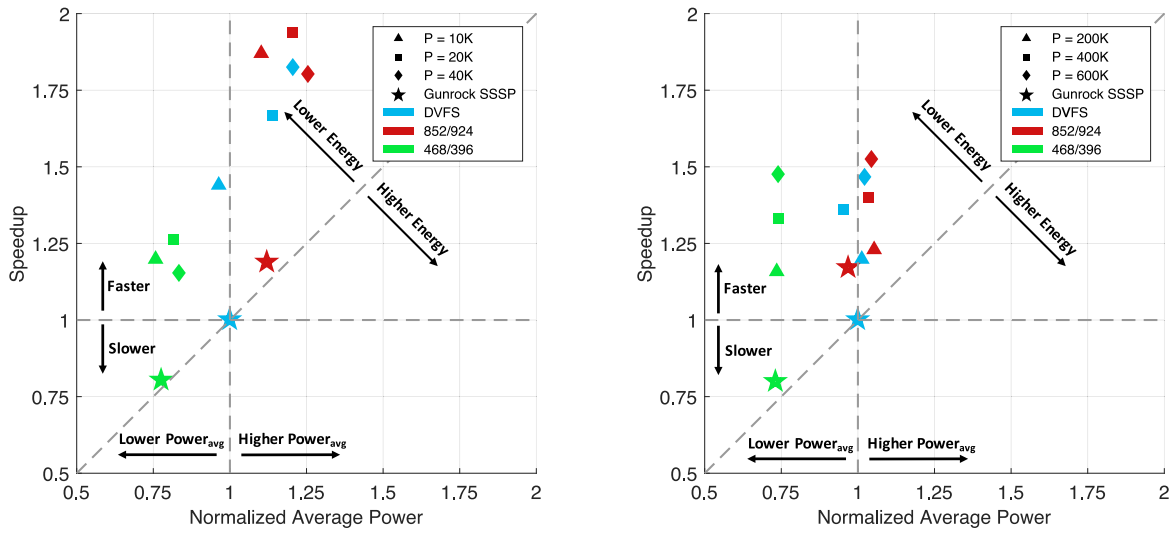
For each $P \in \{10\,000; 20\,000; 40\,000\}$, we measured the available parallelism in every iteration. The distribution appears in Figure 5, comparing against the baseline algorithm ("Near+Far"). The baseline uses δ to minimize execution time while our algorithm will adjust its δ_k to maintain P. At each set-point, the controller is able to maintain a median available parallelism close to P, with most of the distribution's mass confined to a region near that median. By contrast, the baseline method has a much lower median value of available parallelism as well as a much higher variance.

5.2. Performance and power

The next question is whether the proposed method has good performance and power behavior compared to the baseline. In addition, we consider the interaction between the controller algorithm and hardware-based DVFS. The results are summarized in Figure 6 and Figure 7, where we compare the speedup (y-axis) and relative power (x-axis) of the following implementations:

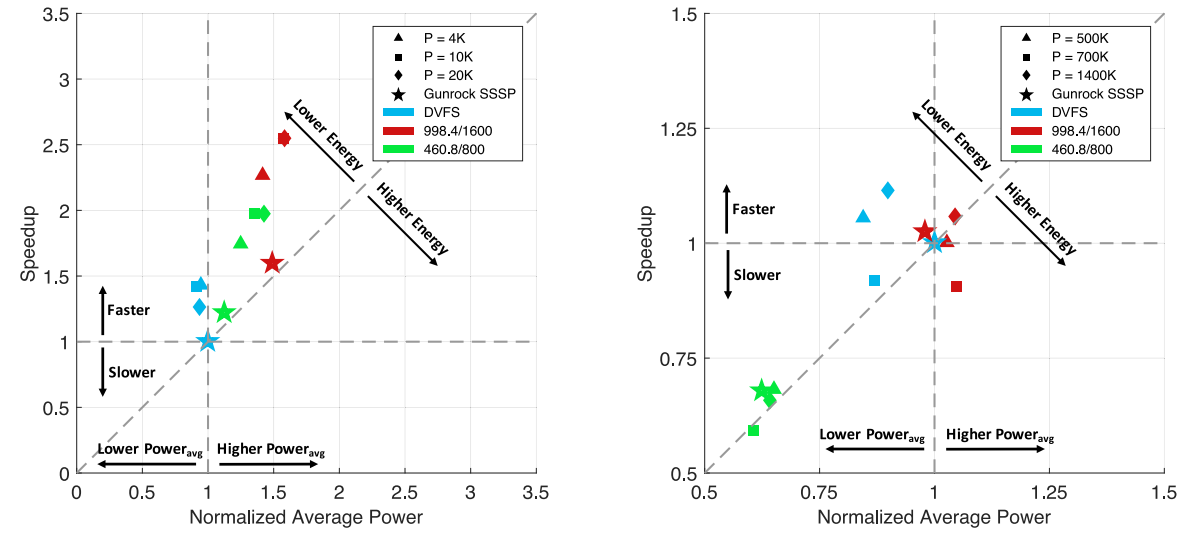
- Baseline "Gunrock SSSP" [blue star markers at the (1, 1) position]: We take the baseline to be the near+far SSSP algorithm of Section 3 using a δ that maximizes performance (minimizes execution time). In addition, we impose *no additional explicit control* on the hardware's DVFS settings, i.e., the hardware uses its own automatic policy.
- Gunrock SSSP at different frequency settings [red and green star markers]: We also run the baseline code at specific combinations of memory bus and GPU core frequencies. These are denoted by c/m values, where c is the GPU core frequency setting and m is the memory bus frequency setting, both in units of MHz. For example, "852/924" means a 852 MHz GPU core frequency and an 924 MHz memory bus frequency.
- Self-tuning at specific set-points, P [triangle, square, and circle markers]: We run our self-tuning method at three specific set-points. As above, we use colors to denote different DVFS settings. Blue markers and lines indicate unconstrained (hardware or system-managed) settings; by contrast, other colors correspond to different core/memory frequency combinations.

First, consider the results on the Cal road network shown in Figure 6(a) and Figure 7(a). At different frequency settings, controlled parallelism improves the performance and power consumption over the baseline. For all the frequency levels, the controller actually improves the performance when tuning to meet the parallelism set-point. In fact, most of the self-tuning points are both faster (higher speedups) and more energy-efficient (above the diagonal line $x = y$) than the baseline algorithm. Moreover, whereas reducing the frequency settings of the baseline reduces the average power



(a) Cal (road network) (b) Wiki (scale-free network)

Figure 6. *Performance versus power (TK1):* Comparisons between the baseline near+far algorithm against the self-tuning method for three different set-points (P), both with and without explicit DVFS control. Using self-tuning, simultaneous speedups and reductions in power are possible on Cal; on Wiki, a smooth tradeoff between speedup and power is possible.



(a) Cal (road network) (b) Wiki (scale-free network)

Figure 7. *Performance versus power (TX1):* Comparisons on the TX1 between the baseline near+far algorithm against the self-tuning method for three different set-points (P), with and without explicit DVFS control. Self-tuning provides similar speedups and power reductions as on TK1 with Cal but more closely follows DVFS for Wiki.

usage, as Figure 6(a) shows, it also reduces performance. By contrast, the self-tuning algorithm can preserve performance.

The Cal results show contrasting behavior as P varies with different frequency combinations. When core and memory frequencies are either both high (red symbols) or both low (green symbols), there is a peak speedup at the middle P setting ($P = 20\text{k}$ and $P = 10\text{k}$) for both platforms. Too much parallelism (high P) leads to redundant work, thereby reducing energy efficiency [25]. However, the results from Figure 7(a) demonstrate that self-tuning does not necessarily improve power usage over the baseline with DVFS on newer platforms, although it does enable additional speedup at the same system power. This result points to continued improvements in DVFS set points on the TX1 versus the TK1. However, it also reveals the variability in DVFS between hardware platforms, which can make "power-portable" code difficult to implement without self-tuning.

The results on the Wiki scale-free network, Figures 6(b) and 7(b), have some qualitative similarities but also a few differences. Among the similarities, at different frequency settings the results show that changing available parallelism is effective in exposing a power and performance tradeoff. For all the frequency levels, performance can be improved using the self-tuning algorithm, although to less of an extent with networks like Cal. In particular, speedups usually require as much or slightly more power than the baseline. That is because Wiki has a broader distribution of available parallelism with more bursts of high parallelism and higher overall medians. The controller can mitigate some of this behavior but cannot entirely eliminate smooth peaks in parallelism. Also, as with the Cal results, results from the TX1 platform are more clustered as P varies, in part due to improved DVFS and potentially due to lower utilization of the GPU overall. Moreover, the optimal P depends on both the hardware platform and the input graph.

Even more importantly, the results in Figure 6(b) show the limits of using only hardware-based DVFS as opposed to (also) using DVFS with finer-grained algorithmic knobs. For the Wiki network, DVFS can provide either a 25 % power savings or a 20 % speedup, but it cannot by itself achieve the maximum combined power savings and speedups seen with algorithmic knobs, which yield a 50 % speedup with a 25 % power savings for $P = 600\text{k}$. While this work focuses on using algorithmic tuning with specific frequency set-points, algorithmic knobs could also provide a powerful supplemental tool to existing DVFS techniques for either power capping or overclocking.

We make the additional claim that P can be directly related to power. In principle, a user might specify a power limit instead of P, and the controller could then adjust itself in response to direct power observations. While that is not possible on the Jetson evaluation platforms, Figure 8 shows that there is some correlation between average power and P and that power tuning could be done using algorithmic knobs. This figure shows that average power does vary with average parallelism when the hardware runs in its default DVFS mode. Exploring this capability should be possible in future work as direct power monitoring becomes more widely available.

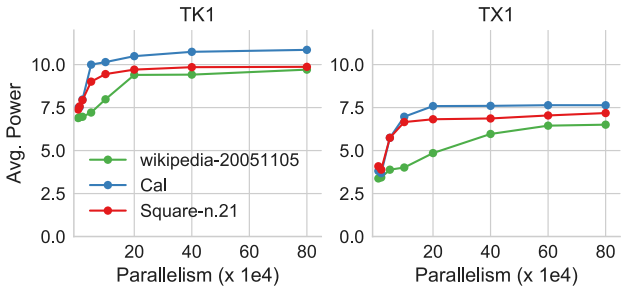


Figure 8. Variation in average power as the parallelism set-point, P, varies.

The controller's runtime overheads are, for every second of the Wiki and Cal running times, approximately 50 µs and 200 µs, respectively, on both platforms. Thus, this overhead is a small fraction of the overall runtime varying from 0.005 % to 0.02 % of runtime. (All reported speedup numbers include this overhead.)

6. Conclusion

One high-level finding of this case study is that it is possible to design an automatic controller scheme to manage irregular parallelism. This control may, in turn, be used to improve energy-efficiency or limit power consumption as part of increasing performance or managing a power-performance tradeoff. Designing such a technique at the level of the algorithm complements power-performance management at other levels, such as at the hardware level through DVFS mechanisms or within the system runtime environment.

Though we argue that the set-point P is easier to pick than δ, the exact choice of P one should use to meet a specific power limit is not known precisely. To overcome this issue, measured power would need to be part of the feedback control system, which was not possible in our experimental setup. Nevertheless, Figure 8 shows it should be possible to do so as fine-grained dynamic power monitoring becomes more widely available.

While we only demonstrated a concrete implementation and results of our controller idea for SSSP, we believe the same ideas are relevant to other graph implementations. For instance, in the Gunrock library, many of the other graph computations have a similar structure to SSSP: they are expressed as sequences or banks of "frontier filters" that manipulate a frontier work-queue. Similarly, recent work to generalize delta-stepping to other graph problems, like k-core decomposition or PageRank, suggest our controller might be adapted for other settings and problems [21]. Extending and testing the controller model in all such cases will be part of our future work.

Lastly, there is a growing literature on the use of control theoretic and online learning ideas in performance tuning (Section 2). Such methods can help make this kind of algorithmic tuning more systematic, data-driven, and fully automatic, and we expect such ideas to continue to have a prominent role in future work.

Acknowledgements

We thank the anonymous reviewers for their thoughtful comments. This material is based upon work supported by the U.S. National Science Foundation (NSF) Award Number 1422935. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of NSF.

References

[1] J. W. Choi, D. Bedard, R. Fowler, and R. Vuduc, "A roofline model of energy," in Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on. IEEE, 2013, pp. 661–672.

[2] Y. Wang, A. A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the GPU," CoRR, vol. abs/1501.05387, 2015. [Online]. Available: http://arxiv.org/abs/1501.05387

[3] J. W. Choi and R. W. Vuduc, "How much (execution) time and energy does my algorithm cost?" XRDS, vol. 19, no. 3, pp. 49–51, Mar. 2013. [Online]. Available: http://doi.acm.org/10.1145/2425676.2425691

[4] S. Mittal and J. S. Vetter, "A survey of methods for analyzing and improving gpu energy efficiency," ACM Comput. Surv., vol. 47, no. 2, pp. 19:1–19:23, Aug. 2014. [Online]. Available: http://doi.acm.org/10.1145/2636342

[5] T. Abdelzaher, Y. Diao, J. L. Hellerstein, C. Lu, and X. Zhu, Introduction to Control Theory And Its Application to Computing Systems. Boston, MA: Springer US, 2008, pp. 185–215. [Online]. Available: http://dx.doi.org/10.1007/978-0-387-79361-0_7

[6] F. D. Macías-Escrivá, R. Haber, R. del Toro, and V. Hernandez, "Self-adaptive systems: A survey of current approaches, research challenges and applications," Expert Systems with Applications, vol. 40, no. 18, pp. 7267 – 7279, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0957417413005125

[7] D. C. Price, M. A. Clark, B. R. Barsdell, R. Babich, and L. J. Greenhill, "Optimizing performance-per-watt on gpus in high performance computing," Computer Science - Research and Development, vol. 31, no. 4, pp. 185–193, 2016. [Online]. Available: http://dx.doi.org/10.1007/s00450-015-0300-5

[8] Y. Jiao, H. Lin, P. Balaji, and W. Feng, "Power and performance characterization of computational kernels on the gpu," in Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int'l Conference on Int'l Conference on Cyber, Physical and Social Computing (CPSCom), Dec 2010, pp. 221–228.

[9] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard, "Dynamic knobs for responsive power-aware computing," SIGPLAN Not., vol. 46, no. 3, pp. 199–212, Mar. 2011. [Online]. Available: http://doi.acm.org/10.1145/1961296.1950390

[10] H. Zhang and H. Hoffmann, "Maximizing performance under a power cap: A comparison of hardware, software, and hybrid techniques," SIGPLAN Not., vol. 51, no. 4, pp. 545–559, Mar. 2016. [Online]. Available: http://doi.acm.org/10.1145/2954679.2872375

[11] B. Acun, A. Langer, E. Meneses, H. Menon, O. Sarood, E. Totoni, and L. V. Kalé, "Power, reliability, performance: One system to rule them all," in Special Issue of IEEE Computer on Energy-Efficient Computing. IEEE, 2016.

[12] E. M. Garzón, J. J. Moreno, and J. A. Martínez, "An approach to optimise the energy efficiency of iterative computation on integrated gpu–cpu systems," The Journal of Supercomputing, pp. 1–12, 2016. [Online]. Available: http://dx.doi.org/10.1007/s11227-016-1643-9

[13] P.-H. Wang, C.-L. Yang, Y.-M. Chen, and Y.-J. Cheng, "Power gating strategies on gpus," ACM Trans. Archit. Code Optim., vol. 8, no. 3, pp. 13:1–13:25, Oct. 2011. [Online]. Available: http://doi.acm.org/10.1145/2019608.2019612

[14] S. Hong and H. Kim, "An integrated gpu power and performance model," SIGARCH Comput. Archit. News, vol. 38, no. 3, pp. 280–289, Jun. 2010. [Online]. Available: http://doi.acm.org/10.1145/1816038.1815998

[15] Y. Wang and N. Ranganathan, "An instruction-level energy estimation and optimization methodology for gpu," in 2011 IEEE 11th International Conference on Computer and Information Technology, Aug 2011, pp. 621–628.

[16] J. Kraus, "Increase performance with gpu boost and k80 autoboost," NVIDIA ParallelForAll blog, 2014. [Online]. Available: https://devblogs.nvidia.com/parallelforall/increase-performance-gpu-boost-k80-autoboost/

[17] G. Wang, Y. Lin, and W. Yi, "Kernel fusion: An effective method for better power efficiency on multithreaded gpu," in Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int'l Conference on Int'l Conference on Cyber, Physical and Social Computing (CPSCom), Dec 2010, pp. 344–350.

[18] S. Pai and K. Pingali, "A compiler for throughput optimization of graph algorithms on gpus," in Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, ser. OOPSLA 2016. New York, NY, USA: ACM, 2016, pp. 1–19. [Online]. Available: http://doi.acm.org/10.1145/2983990.2984015

[19] Y. Yang, P. Xiang, M. Mantor, and H. Zhou, "Fixing performance bugs: An empirical study of open-source gpgpu programs," in 2012 41st International Conference on Parallel Processing, Sept 2012, pp. 329–339.

[20] Y. Ukidave and D. R. Kaeli, "Analyzing optimization techniques for power efficiency on heterogeneous platforms," in 2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum, May 2013, pp. 1040–1049.

[21] Harshvardhan, A. Fidel, N. M. Amato, and L. Rauchwerger, "Kla: A new algorithmic paradigm for parallel graph computations," in Proceedings of the 23rd International Conference on Parallel Architectures and Compilation, ser. PACT '14. New York, NY, USA: ACM, 2014, pp. 27–38. [Online]. Available: http://doi.acm.org/10.1145/2628071.2628091

[22] T. Kumar, "Characterizing and controlling program behavior using execution-time variance (phd thesis)," 2016.

[23] H. Shen, Y. Tan, J. Lu, Q. Wu, and Q. Qiu, "Achieving autonomous power management using reinforcement learning," ACM Trans. Des. Autom. Electron. Syst., vol. 18, no. 2, pp. 24:1–24:32, Apr. 2013. [Online]. Available: http://doi.acm.org/10.1145/2442087.2442095

[24] A. Das, B. M. Al-Hashimi, and G. V. Merrett, "Adaptive and hierarchical runtime manager for energy-aware thermal management of embedded systems," ACM Trans. Embed. Comput. Syst., vol. 15, no. 2, pp. 24:1–24:25, Jan. 2016. [Online]. Available: http://doi.acm.org/10.1145/2834120

[25] A. Davidson, S. Baxter, M. Garland, and J. D. Owens, "Work-efficient parallel gpu methods for single-source shortest paths," in 2014 IEEE 28th International Parallel and Distributed Processing Symposium, May 2014, pp. 349–359.

[26] U. Meyer and P. Sanders, "δ-stepping: a parallelizable shortest path algorithm," Journal of Algorithms, vol. 49, no. 1, pp. 114–152, October 2003.

[27] C. Demetrescu, A. V. Goldberg, and D. S. Johnson, The Shortest Path Problem: Ninth DIMACS Implementation Challenge. American Mathematical Soc., vol. 74.

[28] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," ACM Trans. Math. Softw., vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011. [Online]. Available: http://doi.acm.org/10.1145/2049662.2049663

[29] D. Bedard, M. Y. Lim, R. Fowler, and A. Porterfield, "Powermon: Fine-grained and integrated power monitoring for commodity computer systems," in Proceedings of the IEEE SoutheastCon 2010 (SoutheastCon), March 2010, pp. 479–484.

[30] T. Schaul, S. Zhang, and Y. LeCun, "No More Pesky Learning Rates," ArXiv e-prints, Jun. 2012.