GRAPE: Minimizing Energy for GPU Applications with Performance Requirements

Muhammad Husni Santriaji
Surya University & University of Chicago
Email: muhammad.santriaji@surya.ac.id
santriaji@uchicago.edu

Henry Hoffmann
University of Chicago
Email: hankhoffmann@uchicago.edu

***Abstract*—Many applications have performance requirements (*e.g.,* real-time deadlines or quality-of-service goals) and we can save tremendous energy by tailoring resource usage so the application just meets its performance using the minimal resources. This problem is a classic constrained optimization: the performance goal is the constraint and energy consumption is the objective to be optimized. While several existing hardware approaches solve unconstrained optimizations (*i.e.,* maximizing performance or minimizing energy), we are not aware of a hardware approach that minimizes GPU energy under an externally defined performance constraint. Therefore, we propose GRAPE, a hardware control system for GPUs that coordinates core usage, wavefront/warp action, core speed, and memory speed to deliver user-specified performance while minimizing energy. We implement GRAPE in VHDL (to demonstrate feasibility) and as an extension to GPGPU-Sim (for performance and power measurement). We find that GRAPE can be implemented with very low hardware overhead; however, compared to the no-overhead approach of race-to-idle, GRAPE reduces energy by 9-26% (depending on the performance goal), while meeting performance goals with an average error of 0.75%.**

I. Introduction

Energy consumption is a first order concern for computing systems, from mobile devices (where it defines battery life) to supercomputers (where it determines operating costs). At the same time, ever-increasing performance demands have led to the adoption of GPU-based acceleration in a wide range of computing platforms. While GPUs deliver tremendous computational throughput, they consume a significant portion of total system energy. Therefore, this paper studies hardware support for GPU energy reduction when executing applications with performance requirements.

Such applications are not required to achieve the best possible performance, but deliver results with predictable timing — often expressed as a latency or quality-of-service goal. Examples exist in mobile platforms — including video, media capture, and display where the system interacts with a user [14, 26, 45]. At the other end of the computing spectrum, future supercomputer workloads will include interactive simulations and data analysis applications [37, 42, 43, 52]. In these cases, applications should not run as fast as possible, but meet their performance requirements with minimal energy.

Many prior approaches manage resources for energy reduction. Some consider a single resource only, *e.g.,* CPU [27] or memory [11] frequency. Others coordinate multiple resources for greater energy savings [12, 18, 20, 32, 44]. Finally, some approaches incorporate application-level knowledge (e.g., frame rates) to tailor resource usage to an application's frame-based performance requirements [35, 57]. While these approaches combine multi-component management with domain knowledge, they do so in software. We are not aware of a hardware approach that manages multiple components to meet performance requirements while minimizing energy.

A hardware resource management system has the potential to both remove the optimization burden from software and react more quickly than software can. Of course, providing a hardware solution presents several challenges:

- **Overhead:** Significant area, time, or power overhead will diminish any potential gains.
- **Unknown Applications:** To support different applications; hardware management should (1) rapidly detect applications' response to different resources and (2) react when this response changes (*e.g.,* it transitions from memory bound to compute bound).

To provide hardware resource management, we introduce GRAPE (GPU Resource Adaptation for Performance and Energy). GRAPE overcomes the above challenges to meet user-specified performance goals by managing: (1) the number of streaming multiprocessors (SMs), (2) the number of warps/wavefronts, (3) the SM frequency, and (4) the DRAM frequency. GRAPE's control theoretic approach meets the performance goal and dynamically tailors response to the application's behavior – including phases in an application. Compared to prior work, GRAPE provides three innovations:

- The domain knowledge comes at runtime in the form of a performance requirement. For this paper, the desired performance is expressed as instructions per second. GRAPE's design, however, is independent of any one

This project is funded by the U.S. Government under the DARPA BRASS program, by the Dept. of Energy under DOE DE-AC02-06CH11357, by the NSF under CCF 1439156, and by a DOE Early Career Award.

978-1-5090-3508-3/16/$31.00

metric; *e.g.,* it could be trivially modified to support floating point rate. GRAPE's general interface supports frame-based applications as well as potential future interactive applications.
- All management is performed in hardware. Whereas prior approaches for constrained optimization in GPUs require software support, GRAPE's hardware solution meets performance goals with minimal energy.
- GRAPE's control theoretic design provides some formal guarantees about its dynamic behavior, including guaranteed convergence to the desired performance and bounded convergence time. These guarantees make are appropriate for meeting soft real-time requirements.

We integrate GRAPE into GPGPU-Sim v3.2.2 [4] and GPUWattch [28] and then it using 17 benchmarks drawn from Rodinia [8] and Parboil [49]. We compare to the strategy of *racing-to-idle*; *i.e.,* allocating all resources and transitioning to a low-power idle state when a task completes. We also implement GRAPE in VHDL to demonstrate its feasibility. The evaluation shows that GRAPE provides:

- **Low Overhead:** We synthesize the VHDL for an FPGA using Quartus to demonstrate feasibility and provide a rough estimate of overhead. The PowerPlay Early Power Estimator shows that GRAPE needs 0.434 Watts to operate. (See Section III-D.)
- **Performance Predictability:** Across a range of different targets (from 25% to 100% of maximum achievable performance), GRAPE meets the goal with only 0.75% average error. (See Section V-A.)
- **Energy Efficiency:** At low performance targets, GRAPE consumes only 74% of the energy of race-to-idle. At higher performance targets, the energy savings diminishes; however, even at maximum performance GRAPE reduces energy consumption by 9.02% compared to allocating all resources. (See Section V-B.)
- **Peak Power Reduction:** At low performance targets, peak power is only 40.29% of race-to-idle. At maximum performance, peak power is 87.48% of race-to-idle. (See Section V-C.)

GRAPE is for applications with performance constraints; however, its low overhead allows it to be incorporated into many GPU designs. Overall this paper makes the following contributions:

- Developing a hardware control framework that adapts resource usage to meet application performance requirements with minimal energy.
- Evaluating the approach empirically.
- Releasing the code (both simulation and VHDL) as open source so others can expand or evaluate it [1].

To the best of our knowledge, GRAPE is the first approach to propose a hardware solution for reducing GPU energy while meeting user-defined performance goals.

[1] Available at: https://github.com/grapemicro/GRAPE.git

II. Motivational Example

We motivate GRAPE by considering two separate GPU applications: cfd and hotspot to show that carefully tailoring resource usage to the application and its required performance can save energy while delivering predictable timing.

Of course, performing resource allocation in hardware incurs some cost. In the following, we argue that the GRAPE approach is low overhead. In this section, however, we compare against a zero overhead approach which is commonly used for general purpose platforms: *race-to-idle* [16, 34]. In the race-to-idle approach, an application uses all resources to complete work as fast as possible and then transitions to the idle state until the next interactive job arrives. This race-to-idle approach: 1) is easy to implement, 2) never misses deadlines, and 3) requires no hardware support.

We implement both GRAPE and race-to-idle using GPGPUSim. The GRAPE design is described in detail in Section III. At a high-level, software sets a performance goal (currently desired instructions per second) in a hardware register. GRAPE reads the goal, executes the application, and measures its performance. GRAPE's internal control system dynamically determines how to speed up or slow down the application. As the application executes, the controller's coefficients are adjusted in response to application phases and changing resource needs. Specifically, GRAPE will dynamically adjust SM count, wavefront action, memory speed and SM speed. Race-to-idle, in contrast, does not adapt to the application, but uses all resources at their maximum setting until the job is complete.

Figure 1 shows the comparison of the race-to-idle and GRAPE approaches. For each application we launch with a performance goal that requires only 50% of maximum resource usage (we test a wide range of different goals in the full evaluation). The left column of charts shows CFD, while the right column shows hotspot. The first row shows performance, the second shows power, and the third shows resource usage (as a %) – all as a function of time.

The charts show several interesting behaviors. First, for both applications, GRAPE's performance fluctuates before settling; however, the average performance over the life of the application is always above the goal. The performance of race-to-idle also averages to the desired performance, but it starts by running as fast as possible and then transitioning to a low-power idle state once the work is done. For these examples, both approaches meet the required performance. Looking at power consumption, however, shows a difference. For these examples, race-to-idle has a larger power consumption than GRAPE — these power numbers include the power overhead of the GRAPE system itself. In this example, GRAPE reduces energy consumption by over 20% for each application. Peak power consumption is also considerably reduced. Results will vary with different applications and performance targets, but this shows the potential of using hardware to tailor resource usage to performance goals.

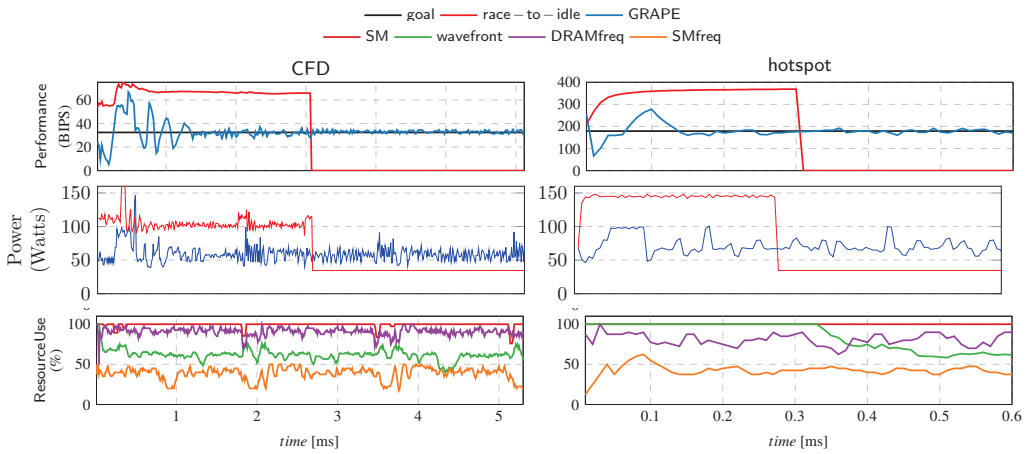

Fig. 1: Performance (top row), power (middle) and resource usage (bottom) for CFD (left) and hotspot (right) applications using both race-to-idle and GRAPE.

The final chart shows GRAPE's resource usage over time. For each resource the chart lists the percentage in use, so a value of 50% means that resource is used at half its capacity. CFD is a memory limited application [8], and GRAPE captures this behavior, quickly reducing wavefront action and SM speed, while keeping memory speed high. In contrast, hotspot tends to need more compute resources [8]. Indeed, GRAPE begins hotspot with high wavefront action and lower memory frequency. Interestingly, just over halfway through hotspot's execution, it changes behavior. Inspecting the output of GPGPUSim, we see that at that point, hotspot hits a shared memory bank conflict that causes serialized memory access. While GRAPE does not explicitly detect that stall, it does detect a drop in performance and it reduces wavefront action accordingly, as that no longer benefits performance, but costs power.

These results demonstrate the potential benefits of tailoring application resource usage with GRAPE. GRAPE's feedback models are 1) robust, 2) computationally inexpensive, and 3) implementable in hardware. These models are also provably convergent to the desired behavior. When an application is running, the GRAPE control system detects differences between the application's desired performance and the actual measured performance. Rather than trying to diagnose the cause of that difference, GRAPE simply adjusts resource usage until the difference goes away. GRAPE is technically an *adaptive* control system in that internally monitors the quality of its control and adjusts the controller while it executes. This adaptive property allows GRAPE to handle a wide variety of applications.

Intuitively, GRAPE's control system works like the cruise control in a car. Car drivers set a speed and cruise control adjusts fuel flow to ensure that the speed is met. In principle, a huge number of variables affect the relationship between fuel flow and a car's speed such as wind velocity, incline, surface condition, and tire pressure. Modeling this huge set of parameters is quite difficult and produces complicated models that are not useful in practice. Control engineers, however, have found that simple feedback models produce robust cruise controls that deliver desired speed even when the operating environment is unknown ahead of time [29]. Motivated by prior successes in control theory, we work to apply these techniques to the problem of maintaining a GPU performance goal while minimizing energy consumption.

III. GRAPE System Design

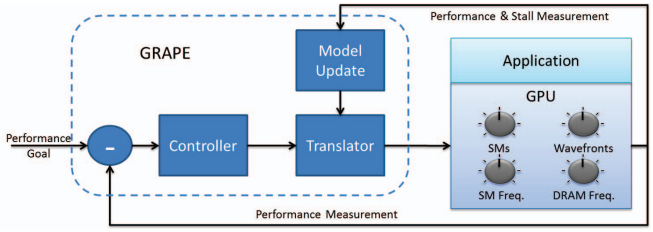

Fig. 2: GRAPE Control Diagram

GRAPE is a hardware control system for GPUs designed to meet user-specified performance while minimizing energy through management of (1) streaming multiprocessors (SMs), (2) wavefronts/warps, (3) SM speed, and (4) DRAM speed, while keeping overhead low. Figure 2 illustrates the GRAPE controller's block diagram. The application provides a performance goal in the form of a target computation rate. This target is compared to the current performance and the difference is passed to a *controller*. The controller computes a signal indicating how much to speedup the application. This speedup signal is passed to a *translator*, which converts speedup into specific allocations of SMs, wavefront/warps, SM speed, and DRAM speed that deliver the controller-specified speedup with minimal energy. Each

TABLE I: Notation used in the paper.

Symbol	Meaning
Controller	
g_{user}	performance goal set by the user
t	time index
e	performance error
g	internal performance goal set by controller
α	control correction constant
$\hat{w}$	current workload
x	inverse workload
$\hat{x}$	a posteriori estimate of x
$\hat{x}^-$	a priori estimate of x
p	a posteriori performance variance estimate
p^-	a priori performance variance estimate
k	Kalman filter gain
h	current performance
s	general control speedup signal
Translator	
$cost$	power cost
sm	status of SM in GPU
nSM	number of SMs
nW	number of wavefronts
$DRAM_{stall}$	stall from SM to DRAM
$Cache_{stall}$	stall in SM pipeline due to memory request
SM_{stall}	stall from DRAM to SM
$DRAM_{threshold}$	stall threshold for DRAM
$Cache_{threshold}$	stall threshold for SM
M	ordered set of memory configurations
$MEMindex$	index in M; *i.e.,* a specific configuration
$s_{MEMindex}$	speedup value of memory configuration
$c_{MEMindex}$	power cost of memory configuration
$f_{MEMindex}$	memory frequency of configuration
$SMindex$	SM configuration
$s_{SMindex}$	speedup of SM configurations
$c_{SMindex}$	power cost of SM configurations
$f_{SMindex}$	SM frequency of configuration
f_{max}	maximum SM frequency available
$cost_{temp}$	temporary cost for finding selection
$cost_{min}$	least cost for finding selection
Model Update	
$highbound_{MEMindex}$	upper bound on speedup for mem. config.
$lowbound_{MEMindex}$	lower bound on speedup for mem. config.
β	learning rate

resource is adjusted and the application executes with the new resource configuration. GRAPE then observes both the stall behavior and the new performance of the application. The stall behavior will be used to update translation on the next iteration, while the performance is fed back into the controller and the process begins again.

We detail each GRAPE module in turn. For each of the three modules we give an intuitive overview and then formally specify its behavior in the form of equations and algorithms. The final subsection discusses GRAPE's hardware implementation. Table I summarizes the notation used throughout this section.

A. The Controller

The controller determines how much to speed up the application at time t. It does so by computing the error between the desired behavior and the measured behavior. The controller accounts for both immediate behavior — *e.g.,* application performance is too low at this iteration — and long-term behavior — *e.g.,* the application initially ran too slowly and now needs extra speed to meet the overall performance target. Additionally, the controller tailors response to individual applications — or phases in applications — by continually estimating the application workload; *i.e.,* the application's instruction latency with minimal resources. GRAPE's controller has several strengths: (1) it uses a simple feedback model, which is easy to calculate in hardware at runtime, (2) it is provably convergent to the desired behavior, and (3) it is robust in the face of noise and model errors [29].

It is assumed that the controller executes at discrete time steps t, and the controller executes Algorithm 1 at each step. The algorithm has four inputs: (1) the user-specified performance goal g_{user}, (2) the current measured performance $h(t)$, (3) the number of instructions completed so far I, (4) and the elapsed time executing the application ℓ. The controller first sets an internal goal $g(t)$ allowing GRAPE to correct for any errors it may have made previously — if GRAPE is initially too slow, it will speed up its own internal goals. GRAPE then computes the difference between its internal goal and the measured performance at the current time (Algorithm 1, line 2).

Phase Estimation. Next, GRAPE estimates the application workload at the current time $\hat{w}(t)$. The workload is a key parameter that tunes control response to the current application; it represents the number of instructions that the application would retire in a time step if allocated the minimal resources. As the application goes through phases, this value might change, so it is continually updated as part of the control action. Lines 4-9 of Algorithm 1 estimate workload using a standard one-dimensional Kalman filter formulation [53]. GRAPE uses a Kalman filter because it is specifically designed to provide accurate estimations in noise and it is exponentially convergent, meaning that the time it takes to converge to the correct estimation is proportional to the logarithm of the error between the initial estimate and the true value [6].

The last step in each control iteration is to use the error (from Algorithm 1 line 2) and the workload estimate (from line 9) to compute the speedup (line 10-11). The speedup is computed according to the Proportional Integral (PI) control law using standard techniques [15]. This speedup signal is then passed to the translator.

Algorithm 1 is constant time and a small number of instructions. It can easily be implemented in fixed-point arithmetic for hardware. Despite this simplicity, control adapts in several ways. First, by keeping an internal goal, separate from the externally specified goal, the controller can adapt to both errors it generates and to phases in application behavior that radically change the performance. Second, the Kalman filter provides fine grain customization of control by adapting to the current workload. Note that as workload increases, the controller's output speedup will also increase (line 10),

Algorithm 1 The Controller

Require: g_{user} ▷ application specified performance goal
Require: $h(t)$ ▷ instructions per second at time t
Require: I ▷ completed instructions
Require: ℓ ▷ elapsed time

procedure THE CONTROLLER
▷ Update local goal based on global progress
$g(t) = g_{user} - \alpha(I/\ell - g_{user})$
▷ Error between new goal and current performance
$e(t) = g(t) - h(t)$
▷ Estimate application workload (i.e., phases)
$\hat{x}^-(t) = \hat{x}^-(t-1)$
$p^-(t) = p(t-1) + q(t)$
$k(t) = \frac{p^-(t)s(t-1)}{[s(t)]^2 p^-(t)+o}$
$\hat{x}(t) = \hat{x}^-(t) + k(t)[h(t) - s(t-1)\hat{x}^-(t)]$
$p(t) = [1 - k(t)s(t-1)]p^-(t)$
$\hat{w}(t) = \frac{1}{\hat{x}(t)}$
▷ Compute speedup
$s(t) = s(t-1) + \hat{w}(t) \cdot e(t)$
return $s(t)$ ▷ speedup to apply at current time
end procedure

which is consistent with intuition. Similarly, if the application suddenly entered a phase where it performed less work, then the controller would reduce speedup appropriately.

One GRAPE's advantages is that the control theoretic techniques presented here emit formal analysis, which heuristic techniques do not. While a rigorous mathematical analysis is beyond the scope of this paper (and also straightforward as GRAPE's controller is built on top of several standard mechanisms), we sketch the outline of such formal analysis in Appendix A. The two major advantages of GRAPE's controller are: 1) it will converge to the desired performance (if achievable) and 2) the convergence time is bounded by the logarithm of the workload error estimate produced by the Kalman filter. Intuitively, GRAPE will hit the performance target and do so in a small number of steps (*i.e.,* invocations of Algorithm 1).

B. The Translator

The translator takes the generic speedup signal and produces a specific setting for the number of SMs, wavefronts, SM frequency, and memory frequency. Ideally, the translator would guarantee the desired speedup is achieved and minimize energy usage, which is properly an integer programming problem, and thus expensive to solve in hardware exactly even for small numbers of configurable resources. GRAPE, therefore, relies on a heuristic solution based on empirical observations.

At a high-level, the heuristic solution first finds the fastest combination of SMs and wavefronts for this application, it then selects the appropriate memory frequency based on the observed number of memory stalls, and finally reduces the SM frequency as much as possible while still achieving the speedup signal. Thus this heuristic still achieves the required speedup, but may sacrifice optimality to produce a simple implementation. Said another way, the heuristic will meet the required performance, but may use more energy than a true optimal solution.

Algorithm 2 The Translator

Require: $s(t)$ ▷ speedup provided by controller

procedure THE TRANSLATOR
▷ Compute the number of SMs
$nSM = 0$
for all sm in the GPU **do**
if $sm ==$ active **then**
$nSM = nSM + 1$
end if
end for
▷ Compute the number of wavefronts
if $Cache_{stall} \geq Cache_{threshold}$ **then**
if $nW > 32$ **then**
$nW = 32$
end if
$nW = nW - 4$
else
$nW = nW + 4$
end if
▷ Compute the SM and memory frequencies
if $DRAM_{stall} \geqslant DRAM_{threshold}$ **then**
$M = \{7, 8\}$
else
$M = \{1, 2, 3, 4, 5, 6, 7\}$
end if
$cost_{min} = \infty$
for $MEMindex \in M$ **do**
if $s_i \geq s(t)$ **then**
$f_{SMindex} = \lceil f_{max} \cdot \frac{s(t)}{s_{MEMindex}} \rceil$
$cost_{temp} = c_{SMindex} \cdot c_{MEMindex}$
if $cost_{temp} \leq cost_{min}$ **then**
$cost_{min} = cost_{temp}$
$f_{SM} \leftarrow f_{SMindex}$
$f_{MEM} \leftarrow f_{MEMindex}$
end if
end if
end for
return nSM ▷ number of SMs to use
return nW ▷ number of wavefronts/warps
return f_{SM} ▷ SM frequency
return f_{MEM} ▷ DRAM frequency

end procedure

Algorithm 2 details GRAPE's translation stage. It is broken into three distinct phases labeled with comments in the

algorithm.

Number of SM. The first phase (lines 2-7) simply counts how many SMs are *active*, meaning they have a CTA(Cooperative Thread Array) running. Empirically, it is always better to run as many SMs as possible. If the SM is not active, then GRAPE will set it to its lowest frequency setting.

Wavefront Scheduler. The next phase (in lines 8-15) determines the number of wavefronts to use. The key to determining the number of SMs is looking at the time the SM pipeline is spent stalling. If the stalls are above a threshold, then GRAPE limits wavefronts to 32 — $2/3$ of its maximum capacity. We move wavefronts in steps of 4 per control action as we find it gives the best results empirically.

SM and DRAM DVFS. The final phase of translation (lines 16-32) set the SM and DRAM frequency. This phase uses two small tables that are stored in hardware and each indexed by an *id*. Each *id* has its own frequency, estimated speedups and costs (in power).

In this final phase, GRAPE is heuristically determining whether the application is compute or memory bound. It first checks if memory stalls are above a certain threshold (line 16). If they are, the application is considered memory bound, and GRAPE will only consider actuator settings that provide high memory frequency (line 17). Otherwise, GRAPE classifies the application as compute bound and only considers slower memory settings. At this stage, these settings are simply indexes into the tables mentioned above.

Once GRAPE has determined the settings to consider, it walks through those memory settings trying to find the slowest frequency setting that will meet or exceed the specified speedup (lines 21-31). Each step of the for loop matches a memory frequency setting to a corresponding SM frequency, determines whether or not that is above the required speedup, and then determines whether the cost (in power) is less than the lowest cost found so far. If the cost is lower, then GRAPE saves this new cost and the settings it found. After completing this final phase, the translator returns the number of SMs, number of wavefronts, the SM frequency, and the DRAM frequency.

Translation takes time proportional to the number of indices in the tables. In practice this tends to be a small number as hardware supports only a few frequency settings.

C. Actuator Model Update

The translator is reliant on models of frequency costs and speedups to produce good results. These values are not universal, however, and may differ for different applications. For example, a slight decrease in memory frequency may not have significant effect on a compute-bound benchmark, but it will for a memory-bound one. To account for these differences GRAPE updates these models on the fly. Initial actuator models assume that cost and performance between actions is linear, then after several decision period GRAPE will update the models to reflect actual behavior of the application under control.

We note that there is a complicated, non-linear relationship between the resources used, application workload, measured behavior and system noise. Rather than build a computationally expensive model, GRAPE adopts the approach of continually estimating this non-linear behavior with a series of linear models, including the Kalman filter presented above and the model update presented here. This approach is analogous to the way scientific applications model complicated, non-linear physical systems with iterative application of linear equations.

Algorithm 3 Update Model

Require: $h(t)$ ▷ measured throughput
Require: $\hat{w}(t)$ ▷ estimated workload
Require: $cost(t)$ ▷ measured cost (power)

1: **procedure** UPDATE MODEL
2: $s_{MEMindex} = \beta \cdot \frac{f_{max}}{f_{SMindex}} w(t) \cdot h(t) + (1-\beta) \cdot s_{MEMindex}$
3: **if** $s_{MEMindex} \geq highBound_{MEMindex}$ **then**
4: $s_{MEMindex} = highBound_{MEMindex}$
5: **end if**
6: **if** $s_{MEMindex} \leq lowBound_{MEMindex}$ **then**
7: $s_{MEMindex} = lowBound_{MEMindex}$
8: **end if**
9: $c_{MEMindex} = \beta \frac{cost(t)}{c_{SMindex}} + (1-\beta) c_{MEMindex}$
10: **end procedure**

Every time GRAPE computes a new control action, it updates the table that stores its models of speedup and cost using Algorithm 3. This is a simple algorithm that updates the model as a function of its current value and the measured behavior. Line 2 computes a new estimate for the memory speedup in the last memory configuration used. The constant f_{max} represents the maximum frequency for SMs which is 800 MHz. β the learning rate which affects how fast the value changes. For example, $\beta = 1$ would always use the last measured value and ignore history. We set this value to 0.85 in our implementation. Lines 3-8 clamp the new value such that not overlap the higher and lower ID value to prevent overflow in the fixed-point hardware implementation. Line 9 updates the cost model, or power consumption, of applying this system resource configuration.

GPU applications generally exhibit three distinct kinds of behavior depending on the resource that bottlenecks performance: compute, memory, or cache. Despite, this difference, however, we find that it is only necessary to update the model of memory response — the SM frequency response tends to keep linear for all types of benchmarks. Thus, we find that updating the SM frequency model is unnecessary. Therefore, to save area overhead, we only update the memory frequency model (using Algorithm 3) and keep the SM frequency model constant.

D. Hardware Implementation

GRAPE is designed to be implementable in hardware. While we do not have the resources to synthesize a GPU that includes GRAPE, we believe it is important to demonstrate that GRAPE can be implemented in hardware. We therefore implement (and release as open source) a VHDL implementation of GRAPE.

To get some specific numbers, we synthesize GRAPE for an FPGA using Quartus II software. The target FPGA device we use is DE2-115. We implement a fixed point package to perform multiplication and division in VHDL [5]. We synthesize the design and find that GRAPE requires 18,426 logic elements, 311 registers and 63 embedded multipliers. TimeQuest timing analyzer shows that GRAPE's f_{max} is 1.45 MHz or 482 cycles overhead in GPU SM frequency. PowerPlay Early Power Estimator shows that GRAPE needs 0.134 Watts to operate. We share this implementation on the link below [2].

We implemented the dynamic frequency and SM actuator by masking the clock in GPGPU-Sim. Wavefront actuator implemented by swl scheduler. We edit the GPUWattch to count the dynamic voltage and leakage static power [7]. We use GTX480 model provided by those simulators. We model the DVFS overhead as 512 cycles [25]. These actuators increase the GPU power consumption by 0.3 W. One decision period for GRAPE is 4096 cycles. We include all of this overhead during simulation in GPGPU-Sim.

GRAPE samples the sensors every 4096 cycles. We assume there is no overhead in sampling the data. Control calculation is called early at 550 cycles earlier to reduce the error in calculation. Frequency overhead is 512 cycles, during this overhead period the simulator runs the application in previous frequency action.

Overall, we find that these results show GRAPE to be low overhead and easily implemented in hardware. The area, power, and timing would probably all improve if GRAPE was synthesized in ASIC or custom VLSI and added to a real GPU implementation. For the purposes of our evaluation we use these numbers from the FPGA for all experiments.

IV. Methodology

A. Baseline Architecture

GRAPE is evaluated on GPGPU-Sim v3.2.2, a cycle-accurate GPGPU Simulator [4] and GPUWattch [28] to get the energy consumption. Our baseline implementation in GPGPU-Sim is based on the configuration in Table II.

In this implementation, we have a set of SM actuators, SM frequency actuators, memory frequency actuators and wavefront actuators. SM actuators have 16 members covering operation from idle to 15 SMs. We assume that SM frequency actuator has 8 P-states, ranging from a peak of 800 MHz to a minimum of 100 MHz, with step size of 100 MHz.

[2]Available at: https://github.com/grapemicro/GRAPE.git

TABLE II: GPGPU-Sim Configuration.

Field	Value
Shader Core	Fermi 15 Shader Cores, 5-Stage Pipeline,
Resources / Core	Max 1536 Threads, 32 kB Shared Memory, 32768 Registers 48 warps
L1 Caches / Core	32kB 8-way L1 Data Cache, 8 kB 4-way texture, 8kB 4-way constant cache, 64B Line Size
L2 Unified Cache	786 kB, 128B Line size, 8-way associative
Scheduling	GTO Scheduling, Load Balance scheduling
Interconnect	1 crossbar, 16B Channel Width,
DRAM Model	FR-FCFS (128 RQS/MC), 4B Bus Width, 4 DRAM-banks/MC, 2 kB page size, 4 burst size, 8 MCs

These settings align with the GTX 480's existing DVFS settings. DRAM frequency actuators have 8 P-states, ranging from a peak of 1056 MHz to 152MHz. We use the 45 nm predictive technology models [39] to scale the voltage with frequency (from 1.14 V to 0.55 V). We use the scheduler's wavefront limiting (swl) [41] to manipulate the number of active wavefronts.

B. Benchmarks

We evaluate GRAPE using the benchmarks in Table III. The table lists each benchmark name, an abbreviation used in this paper, and the resource on which the benchmark is most dependent as characterized in prior work by Sethia and Mahlke [44]. For every benchmark we measure the timing data from the simulator and get the power data from GPUWattch.

C. Points of Comparison

We compare GRAPE to the following:

Race-to-Idle: Runs each job in an application as fast as possible and then the idles the system until the next job is to be processed [16, 34]. According to Smith the GTX480 idles at 47 W[46].

Race-to-Sleep: Runs each job as fast as possible and then puts the GPU to sleep until the next job is ready for processing. We optimistically assume that the sleep state consumes minimal power and it takes no overhead to enter and exit the sleep state. Thus, the numbers unrealistically favor race-to-sleep, but it demonstrates the most effective savings a sleep state could possibly have.

Equalizer-to-idle: Equalizer is a system that maintains near maximum performance while minimizing energy con-

TABLE III: Benchmarks Used.

	Applications	Abbr.	Resource Need	Source
1	AES Cryptography	AES	Memory	[30]
2	Backpropagation	BP	Memory	[8]
3	FFT Algorithm	FFT	Memory	[49]
4	Needelman-Wunch	NW	Unsaturated	[8]
5	Breadth First Search	BFS	Cache	[49]
6	B+ Tree	BPT	Memory	[8]
7	CFD Solver	CFD	Memory	[8]
8	Coulombic Potential	CUTCP	Compute	[49]
9	Gaussian Elimination	GE	Unsaturated	[8]
10	Hotspot	HOT	Compute	[8]
11	KMeans	KM	Cache	[8]
12	LU Decomposition	LUD	Unsaturated	[8]
13	Particle Filter Float	PFF	Memory	[8]
14	Pathfinder	PF	Compute	[8]
15	Sum of Absolute Diff.	SAD	Unsaturated	[8]
16	SRAD	SRAD	Memory	[8]
17	Streamcluster	SC	Unsaturated	[8]

sumption [44]. Equalizer, as originally conceived, is not designed to meet performance targets, but we add this capability by using the state produced by Equalizer and then idling once the job is complete. Equalizer is not available as open source, so we derive these numbers from Equalizer's published results and only compare the applications GRAPE has in common with Equalizer.

V. Experimental Evaluation

This section presents our empirical evaluation of GRAPE, using the experimental setup described in the previous section. We first measure GRAPE's ability to meet performance requirements. We then evaluate GRAPE's energy savings and peak power reduction. The section concludes by studying the impact of idle power on energy savings.

A. Performance Impact

For each benchmark we evaluate several performance goals. Specifically, we set a performance goal corresponding to $X\%$ of maximum performance where $X \in \{25, 50, 75, 100\}$. For example, FFT has a maximum performance of 408,346 MIPS (millions of instructions per second). The 25% goal for FFT means we set the performance at 102,086 MIPS.

We quantify error as the relative error expressed as a percentage. Relative error is the difference between the target and achieved performance divided by the target. We only count error if GRAPE runs the application below goal and count the error as zero if GRAPE runs the benchmark above the goal. Of course, running above the goal will incur additional energy costs, but we evaluate energy in the next section.

Figure 3 shows the relative error for each benchmark and performance target. GRAPE successfully maintains the performance goal achieving, on average, 99.25% of the desired performance; *i.e.,* 0.75% average error across all performance targets. We note that the race-to-idle strategy will never miss a target – the major advantage of this strategy – because it always completes all work in the default configuration and idles.

While GRAPE's accuracy is, in general, quite good, the results demonstrate two areas where GRAPE struggles. First, error increases as the performance target increases. Second, the errors are highest for the LUD and SAD benchmarks. As the performance target increases, GRAPE's margin for error decreases. While GRAPE's controller is self-correcting, at high performance targets, there may simply not be enough time to correct an error before the benchmark completes. This lack of time for the self-correction mechanism to take effect is the same issue that affects the LUD and SAD benchmarks. Both of these benchmarks consist of multiple kernels where the first kernel has very high parallelism and subsequent kernels have lower parallelism. GRAPE reduces resource usage for the high parallelism kernel, but then it is not physically possible to meet the performance target when the lower-parallelism kernels start to execute.

This pattern – highly parallel kernels followed by low-parallelism kernels – represents the worst case for GRAPE. Despite this worst case behavior, the results for SAD and LUD are still fairly good. We note that the results would improve if we ran these kernels in a loop, as that would allow GRAPE's self-correction mechanism to work over repeated application invocations. In addition, in future work, we could address this issue by combining GRAPE with static program analysis to provide GRAPE with the foreknowledge necessary to address this pattern.

B. Energy Impact

Figure 4 shows GRAPE's energy consumption and figure 5 shows its energy efficiency (performance/Watt). All numbers are normalized to the race-to-idle strategy. By geometric mean, targeting performance goals of 25%, 50%, 75% and 100% results in energy reductions of 25.76%, 24.66%, 19.25% and 9.02% compared to race-to-idle (higher is better). Meanwhile, targeting performance goal as 25%, 50%, 75% and 100% from default gives us 1.35×, 1.34×, 1.25× and 1.11× energy efficiency (MIPS/Watt, higher is better) compared to race-to-idle.

At the 100% performance target, race-to-idle is not actually idling the system at all. However, GRAPE's intelligent resource allocation strategies provide relative energy savings even when performance is not reduced. This is because GRAPE can reduce the energy consumed by unnecessary resources even when running at maximum performance. For example, GRAPE will reduce memory energy for compute bound benchmarks and reduce compute energy for memory bound benchmarks, compared to race-to-idle.

The kmeans benchmark gets the biggest benefit, as GRAPE increases its energy efficiency up to 1.93× and achieves energy saving of 48.18% compared to race-to-idle. GRAPE's wavefront scheduling successfully configures the

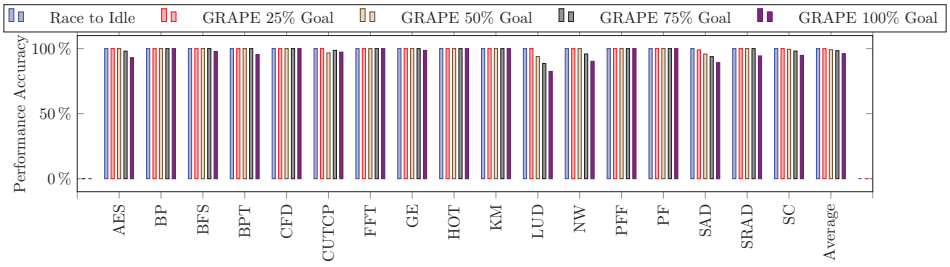

Fig. 3: GRAPE Performance Accuracy

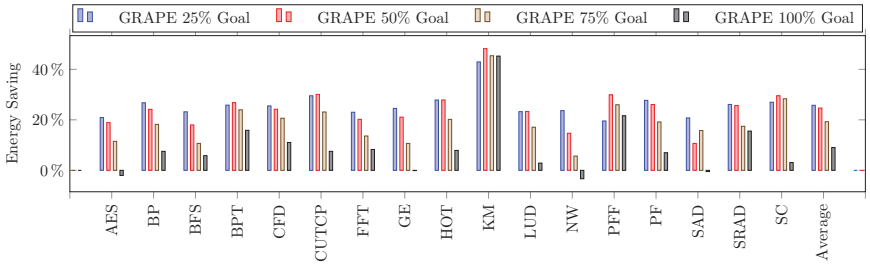

Fig. 4: GRAPE energy savings compared to race-to-idle.

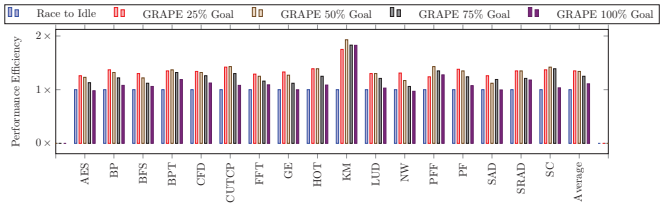

Fig. 5: GRAPE energy efficiency (performance/Watt) compared to race-to-idle.

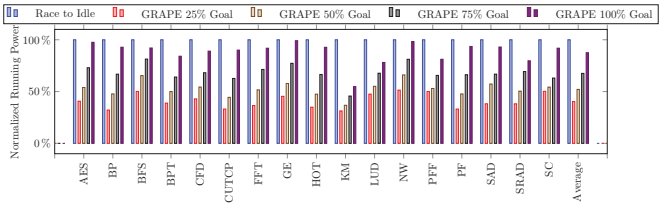

Fig. 6: Average Power Consumption

most effective wavefront available. This increasing performance then turns into a reduction in resource usage. Streamcluster also benefits from this scenario, increasing energy efficiency up to 1.42× and achieves energy saving of 29.55%. GRAPE's generality also benefits computational benchmarks like Hotspot – increasing its performance efficiency up to 1.39× and achieves energy saving of 27.84%.

These results demonstrate the claim from the introduction: that careful tailoring of resource usage can greatly reduce energy consumption compared to strategies like racing-to-idle. Furthermore, these results demonstrate that it is possible to build a resource management strategy into hardware and achieve good results.

C. Power Impact

GRAPE not only decreases the energy consumption, it also decreases peak power consumption significantly compared to racing to idle. As we see in figure 6 GRAPE successfully manages the performance goal for 25%, 50%, 75% and 100% to give us 40.29%, 52.08%, 67.54% and 87.84% power reductions respectively.

D. Comparison with Prior Work

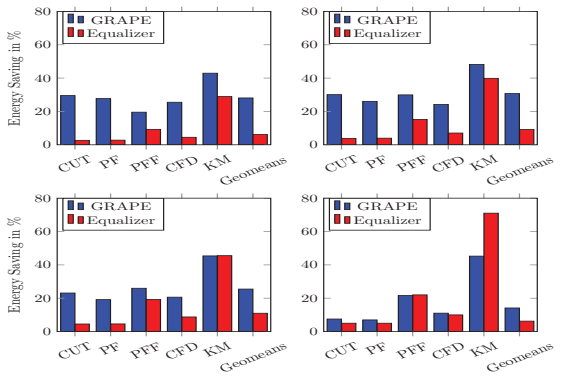

Fig. 7: GRAPE comparison with Equalizer-to-idle in 25% goal (top-left chart), 50% goal (top-right chart), 75% goal (bottom-left chart) and unconstrained performance (bottom-right chart).

Figure 7 compares the energy savings of GRAPE to that of Equalizer [44], a comprehensive dynamic system

which coordinates SM frequency, DRAM frequency, interconnect frequency, L2 frequency and number of CTA. While GRAPE performs constrained optimization (meeting performance with minimum energy), Equalizer is an unconstrained optimizer, it provides no performance guarantees, but generally tries to reduce energy without impacting performance. In this section, we compare GRAPE's constrained optimization approach to Equalizer-to-idle.

The results show that GRAPE's incorporation of performance requirements allows it to save substantial energy compared to Equalizer for all the targets less than 100%. For the 100% target, GRAPE is similar to Equalizer. We emphasize that GRAPE is not designed to improve on Equalizer, instead it solves a different problem: constrained optimization. These results, however, demonstrate that GRAPE can provide competitive behavior on unconstrained performance (bottom-right chart). Thus, GRAPE provides a new capability without diminishing existing capabilities.

E. Sensitivity to Idle Power

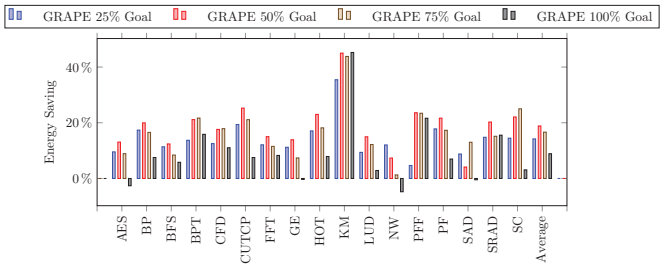

Fig. 8: GRAPE energy saving compared to race-to-sleep.

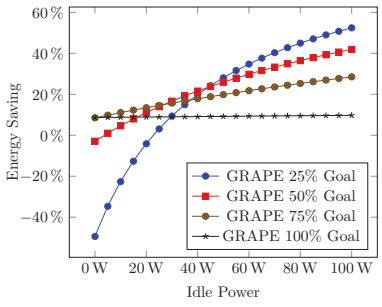

Fig. 9: Energy Reduction in Varying Idle Power

GRAPE's energy reduction is clearly sensitive to both the idle power consumption and the performance goal. We have already explored sensitivity to various performance goals in the above results. In this section we explore sensitivity to idle power.

Figure 8, compares GRAPE's energy savings to a race-to-sleep strategy. We assume sleeping GPU GTX480 power is 34.3265W, which all the SMs are in idle state and the voltage is minimum [28] and that the sleep state can be entered and exited with no overhead (likely an optimistic assumption). Compared to race-to-sleep, GRAPE can still increase the energy efficiency to 1.18 and decrease the energy consumption to 0.86×.

GRAPE saves energy by finding the best configuration and avoiding the high idle power in the GPU. If the idle power is very low then the energy saving and energy efficiency also become lower and if the idle energy is higher then the energy saving will be increasing too. We show the relation between different idle power and energy savings in Figure 9. However, due limitations in technology scaling, future processors are expected to decrease the dynamic power while the leakage power – and thus, idle power – increase [22, 24]. Therefore, we believe GRAPE will continue to be suitable and applicable to reduce future GPU energy consumption while maintaining performance goals.

VI. Related Works

We discuss related work in GPU resource management, especially for power and energy savings.

Throughput Control: A method to control computation throughtput already presented in [50], [9], [2] and [10]. While those method try to control the system's throughput, however those method is not implemented on GPGPU. Some potential energy saving is missed because some actuators that can only found in GPGPU is not controlled.

Core Clock Gating in GPU: Hong et al. propose an offline calculation to predict the optimal cores configuration to reduce energy consumption in GPUs [19]. GRAPE finds optimal resources configuration online without any prior knowledge before about GPGPU kernel applications. Suleman et al. proposes Feedback-Directed Pipelining (FDP) [51], a software framework that chooses the core-to-stage allocation at run-time. The FDP strategy maximizes the performance of the workload and then saves power by reducing the number of active cores. GRAPE is implemented in hardware, considers more than just core usage, and reduces overall energy consumption. Song et al. proposes Throttle CTA Scheduling (TCS)[47] to improve energy efficiency for memory intensive GPU workloads. GRAPE works not only in memory-intensive applications but in all interactive GPU applications.

DVFS in GPU: Mei et al., [31] Jiao et al., [21] and Abe et al. [1] use DVFS to reduce power consumption and energy in GPUs, however there are some unpredictable performance losses. GRAPE maintains predictable, user-specified performance goals while reducing energy consumption. Rong et al. increase energy efficiency by increasing the GPU frequency. [13] in GPU K20. However, increasing the frequency leads to increasing power. GRAPE not only decreases the energy but the power. GRAPE could easily be extended to include these increased frequency settings. Amur et al. predict performance degradation due to DVFS[3]. GRAPE adapts the DVFS

performance degradation online and trade it to reduce energy consumption while maintain performance goal.

Dynamic Resource Allocation: Sethia et al. propose Equalizer [44] to dynamically tune GPU resources. Equalizer focuses on increasing performance so the energy will be reduced due reduced running time. Equalizer does not count the impact of idle energy consumption after the application is finished and allows increasing power due increasing resource usage. In that sense, Equalizer is better suited for non-interactive applications that benefit from completing as fast as possible. GRAPE, however, focuses on minimizing energy consumption while maintaining interactive performance. GRAPE not only reduces energy, but also peak power and it accounts for the impact of idle power after the application is finished. Our empirical results show that GRAPE can achieve similar results to Eqaulizer for unconstrained optimization while allowing a new capability – meeting performance goals. Wu et al. propose a machine learning method to estimate GPU performance and power [54]. While their implementation is still offline, GRAPE already implemented an autonomous system that predicts performance and power online without prior knowledge of applications. Pothukuchi et al propose a general technique for synthesizing multiple input, multiple output controllers for architectures [38].

Managing number of threads for cache locality: Narasiman et al. proposed two level warp scheduling to reduce memory stall [36]. Kayiran et al. proposed DynCTA to manage CTA number in GPU core [23]. Rogers et al. proposed cache concious wavefront scheduling to manage wavefront [41]. However, GRAPE uses dynamic wavefront scheduling to reduce the stall between core and memory.

Application-aware Approaches: Zhu and Reddi use models of webpage resource needs to schedule rendering on large or small cores in a heterogeneous processor [57]. Nachiappan et al. use application-level knowledge to tune many different system-level resources, including GPU, CPU, and DRAM to meet frame rates for frame-based mobile applications [35]. Mishra et al. use probabilistic graphical models to predict the most efficient resource usage for interactive applications running on a server [33]. The JouleGuard system uses control theory and high-level application feedback to meet energy guarantees by tuning both resources and application configuration [17]. The CASH architecture and runtime uses application-level feedback to minimize the cost of renting compute resources while meeting a performance constraint [56]. These approaches are all similar in their high-level goals: deliver some guarantee to the user while optimizing some objective. The JouleGuard and CASH even incorporate control systems, like GRAPE. GRAPE is unique in that it solves this problem in hardware.

A number of libraries and middle-ware frameworks support real-time applications through resources management [20, 40, 48, 55]. Most of these frameworks allow applications to express their high-level timing requirements and then allocate resources to ensure that those requirements are met efficiently. In that sense, GRAPE shares the same high-level goal. The major difference is that GRAPE supports this goal in hardware and requires no software changes (other than specification of the performance requirement). We believe GRAPE is the first hardware approach to meet user-specified performance goals while minimizing energy for GPU applications. GRAPE thus removes a significant optimization burden from software.

VII. Conclusion

GRAPE is a resource management system for interactive GPU applications. GRAPE takes a performance goal and then determines how to allocate resources to an application such that the performance goal is met and energy is minimized. GRAPE uses a computationally inexpensive control system which is easily realizable in hardware with low overhead. GRAPE is highly accurate in delivering performance yet it provides significant energy savings for applications with different performance goals. In addition, our results indicate that GRAPE is competitive with prior approaches for unconstrained optimization – meaning that GRAPE can have a positive benefit even for non-interactive applications. The combination of low-overhead and competitiveness with prior techniques means that GRAPE could be integrated into GPUs with almost no downside while providing significant energy savings for interactive applications.

Appendix

We briefly outline the formal guarantees provided by the GRAPE controller. These guarantees are provided by the controller design itself, so we only outline them here. A full formal theoretic analysis is beyond the scope of this paper. We therefore sketch the analysis of two key claims:

1) The controller converges to the desired performance.
2) The convergence time is bounded.

Convergence: The controller consists of three sets of equations: 1) the equations that set the internal goal (Algorithm 1, line 2-3), 2) the Kalman filter (lines 4-9, and 3) the PI controller that sets speedup (line 10). Prior work shows this Kalman filter formulation to be exponentially convergent; *i.e.,* the number of time steps required to converge is proportional to the logarithm of the difference between the true workload and the estimated workload [6]. Once the Kalman filter has converged, the estimated workload used in line 10 is stable. Using standard control analysis, it is trivial to show that the PI controller converges to the desired performance after the Kalman filter is stable (that is, in fact, a major reason to use the PI control law) [15]. In fact, after the Kalman filter stabilizes it takes one additional time step for the controller to stabilize at the desired performance (based on Z-domain analysis of a PI controller with no pole [15]). The only thing remaining is to show that the internal goal stabilizes, which is trivial at the point where the Kalman filter and controller have

converged. Intuitively, the time spent waiting for these two systems to converge produces error, but once they converge that error becomes a fixed constant. Once that error is fixed then the difference between the external goal and internal goal is fixed and the entire system is convergent.

Bounded Convergence Time: The time for the controller and internal goal to stabilize is constant. The time for the Kalman filter to converge is the log of the error. Of course, if error was known a priori, we would not need a Kalman filter in the first place, but a logarithmic convergence is fast in practice. Furthermore, straightforward eigenvalue analysis of the control system shows that the path to convergence will be smooth (*i.e.,* not oscillate), if the workload is not overestimated. Therefore, in practice we seed the initial workload estimate with the smallest value seen in our empirical study. Thus the convergence time will be proportional to the logarithm of the difference between the minimal workload and the application's actual workload.

We note three limitations of this formal analysis. First, it does not provide hard real-time guarantees. There is no guarantee that the system will be able to fully recover from the original error. Instead this guarantee says that the system will converge to the desired performance after time proportional to the difference between true and estimated workload. In practice, this produces good results for all but some extreme cases. Second, a degenerate or adversarial application could create a very bad scenario for the controller by changing workload rapidly (every control period) by at least an order of magnitude so that the controller never has a chance to converge. We do not believe real applications will exhibit this behavior, but if they do they should not be used in conjunction with GRAPE. Furthermore, large changes in behavior are fine as long as they are spread out so that the controller can converge between these changes. Third, this analysis only guarantees that the controller converges to the desired performance, it does not guarantee convergence to the minimal energy solution. In fact, to make the optimization practical, we use a heuristic solution which may be suboptimal, but the empirical results are strong.

REFERENCES

[1] Y. Abe et al. "Power and Performance Analysis of GPU-accelerated Systems". In: *Proceedings of the 2012 USENIX Conference on Power-Aware Computing and Systems*. HotPower'12. Hollywood, CA: USENIX Association, 2012, pp. 10–10.

[2] N. Almoosa et al. "Throughput regulation in multicore processors via IPA". In: *2012 IEEE 51st IEEE Conference on Decision and Control (CDC)*. 2012, pp. 7267–7272.

[3] H. Amur et al. "Towards Optimal Power Management: Estimation of Performance Degradation due to DVFS on Modern Processors". In: (2010).

[4] A. Bakhoda et al. "Analyzing CUDA workloads using a detailed GPU simulator". In: *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE. 2009, pp. 163–174.

[5] D. Bishop. "Fixed point package users guide". In: *Packages and bodies for the IEEE* (2006), pp. 1076–2008.

[6] L. Cao and H. M. Schwartz. "Analysis of the Kalman filter based estimation algorithm: an orthogonal decomposition approach". In: *Automatica* 40.1 (2004), pp. 5–19.

[7] Y. Cao. *Predictive Technology Model for Robust Nanoelectronic Design*. Springer Science & Business Media, 2011.

[8] S. Che et al. "Rodinia: A benchmark suite for heterogeneous computing". In: *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. IEEE. 2009, pp. 44–54.

[9] T. Chen et al. "Execution Time Prediction for Energy-efficient Hardware Accelerators". In: *Proceedings of the 48th International Symposium on Microarchitecture*. MICRO-48. Waikiki, Hawaii: ACM, 2015, pp. 457–469.

[10] X. Chen et al. "Throughput Regulation in Shared Memory Multicore Processors". In: *2015 IEEE International Conference on High Performance Computing*. 2015.

[11] Q. Deng et al. "MemScale: Active Low-power Modes for Main Memory". In: *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVI. Newport Beach, California, USA: ACM, 2011, pp. 225–238.

[12] C. Dubach et al. "A Predictive Model for Dynamic Microarchitectural Adaptivity Control". In: *MICRO*. 2010.

[13] R. Ge et al. "Effects of Dynamic Voltage and Frequency Scaling on a K20 GPU". In: *Parallel Processing (ICPP), 2013 42nd International Conference on*. 2013, pp. 826–833.

[14] Google. *The Google gospel of speed*.

[15] J. L. Hellerstein et al. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[16] H. Hoffmann. "Racing and Pacing to Idle: An Evaluation of Heuristics for Energy-aware Resource Allocation". In: *Proceedings of the Workshop on Power-Aware Computing and Systems*. HotPower '13. Farmington, Pennsylvania: ACM, 2013, 13:1–13:5.

[17] H. Hoffmann. "JouleGuard: energy guarantees for approximate applications". In: *SOSP*. 2015.

[18] H. Hoffmann et al. "Self-aware computing in the Angstrom processor". In: *DAC*. 2012.

[19] S. Hong and H. Kim. "An Integrated GPU Power and Performance Model". In: *Proceedings of the 37th Annual International Symposium on Computer Architecture*. ISCA '10. Saint-Malo, France: ACM, 2010, pp. 280–289.

[20] C. Imes et al. "POET: A Portable Approach to Minimizing Energy Under Soft Real-time Constraints". In: *RTAS*. 2015.

[21] Y. Jiao et al. "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)*. 2010, pp. 221–228.

[22] A. Kahng. "The ITRS design technology and system drivers roadmap: Process and status". In: *Design Automation Conference (DAC), 2013 50th ACM / EDAC / IEEE*. 2013, pp. 1–6.

[23] O. Kayiran et al. "Neither more nor less: Optimizing thread-level parallelism for GPGPUs". In: *Parallel Architectures and Compilation Techniques (PACT), 2013 22nd International Conference on*. 2013, pp. 157–166.

[24] N. S. Kim et al. "Leakage current: Moore's law meets static power". In: *computer* 36.12 (2003), pp. 68–75.

[25] W. Kim et al. "System level analysis of fast, per-core DVFS using on-chip switching regulators". In: *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*. 2008, pp. 123–134.

[26] Kissmetrics. *How loading time affects your bottom line*.

[27] C. Lefurgy et al. "Power capping: a prelude to power shifting". In: *Cluster Computing* 11.2 (2008), pp. 183–195.

[28] J. Leng et al. "GPUWattch: enabling energy optimizations in GPGPUs". In: *ACM SIGARCH Computer Architecture News* 41.3 (2013), pp. 487–498.

[29] W. Levine. *The control handbook*. Ed. by W. Levine. CRC Press, 2005.

[30] S. A. Manavski. "CUDA Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography". In: *Signal Processing and Communications, 2007. ICSPC 2007. IEEE International Conference on*. 2007, pp. 65–68.

[31] X. Mei et al. "A Measurement Study of GPU DVFS on Energy Conservation". In: *Proceedings of the Workshop on Power-Aware Computing and Systems*. HotPower '13. Farmington, Pennsylvania: ACM, 2013, 10:1–10:5.

[32] D. Meisner et al. "Power management of online data-intensive services". In: *ISCA* (2011).

[33] N. Mishra et al. "A Bayesian Approach to Minimizing Energy Under Performance Constraints". In: *ASPLOS*. 2015.

[34] A. Miyoshi et al. "Critical Power Slope: Understanding the Runtime Effects of Frequency Scaling". In: *ICS*. 2002.

[35] N. Nachiappan et al. "Domain knowledge based energy management in handhelds". In: *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. 2015, pp. 150–160.

[36] V. Narasiman et al. "Improving GPU Performance via Large Warps and Two-level Warp Scheduling". In: *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-44. Porto Alegre, Brazil: ACM, 2011, pp. 308–317.

[37] T. Patki et al. "Practical Resource Management in Power-Constrained, High Performance Computing". In: *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. HPDC '15. Portland, Oregon, USA: ACM, 2015, pp. 121–132.

[38] R. Pothukuchi et al. "Using Multiple Input, Multiple Output Formal Control to Maximize Resource Efficiency in Architectures". In: *ISCA*. 2016.

[39] *Predictive Technology Model*. http://ptm.asu.edu/.

[40] R. Rajkumar et al. "A resource allocation model for QoS management". In: *RTSS*. 1997.

[41] T. G. Rogers et al. "Cache-Conscious Wavefront Scheduling". In: *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-45. Vancouver, B.C., CANADA: IEEE Computer Society, 2012, pp. 72–83.

[42] V. Sarkar et al. "Software challenges in extreme scale systems". In: *Journal of Physics: Conference Series* 180.1 (2009), p. 012045.

[43] *Scientific discovery at exascale: Report from the doe ascr 2011 workshop on exascale data management, analysis,and visualization*. 2011.

[44] A. Sethia and S. Mahlke. "Equalizer: Dynamic Tuning of GPU Resources for Efficient Execution". In: *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*. 2014, pp. 647–658.

[45] A. Shye et al. "Power to the people: Leveraging human physiological traits to control microprocessor frequency". In: *MICRO*. 2008.

[46] R. Smith. *NVIDIAs GeForce GTX 480 and GTX 470: 6 Months Late, Was It Worth the Wait?* http://www.anandtech.com/show/2977/nvidia-s-geforce-gtx-480-and-gtx-470-6-months-late-was-it-worth-the-wait-/19. Accessed: 2015-05-01.

[47] S. Song et al. "Energy-efficient Scheduling for Memory-intensive GPGPU Workloads". In: *Proceedings of the Conference on Design, Automation & Test in Europe*. DATE '14. Dresden, Germany: European Design and Automation Association, 2014, 19:1–19:6.

[48] D. C. Steere et al. "A Feedback-driven Proportion Allocator for Real-rate Scheduling". In: *OSDI*. 1999.

[49] J. A. Stratton et al. "Parboil: A revised benchmark suite for scientific and commercial throughput computing". In: *Center for Reliable and High-Performance Computing* (2012).

[50] J. Suh and M. Dubois. "Dynamic MIPS Rate Stabilization in Out-of-order Processors". In: *Proceedings of the 36th Annual International Symposium on Computer Architecture*. ISCA '09. Austin, TX, USA: ACM, 2009, pp. 46–56.

[51] M. A. Suleman et al. "Feedback-directed Pipeline Parallelism". In: *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. PACT '10. Vienna, Austria: ACM, 2010, pp. 147–156.

[52] *Synergistic Challenges in data-intensive science and exascale computing*. 2013.

[53] G. Welch and G. Bishop. *An Introduction to the Kalman Filter*. Tech. rep. TR 95-041. UNC Chapel Hill, Department of Computer Science.

[54] G. Wu et al. "GPGPU performance and power estimation using machine learning". In: *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. 2015, pp. 564–576.

[55] R. Zhang et al. "ControlWare: A middleware architecture for Feedback Control of Software Performance". In: *ICDCS*. 2002.

[56] Y. Zhou et al. "CASH: Supporting IaaS customers with a sub-core configurable architecture". In: *ISCA*. 2016.

[57] Y. Zhu and V. Reddi. "High-performance and energy-efficient mobile web browsing on big/little systems". In: *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*. 2013, pp. 13–24.