POWER AND PERFORMANCE MODELING FOR HIGH–PERFORMANCE COMPUTING ALGORITHMS

A Thesis
Presented to
The Academic Faculty

by

Jee Whan Choi

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
May 2015

POWER AND PERFORMANCE MODELING FOR HIGH–PERFORMANCE COMPUTING ALGORITHMS

To my parents, for their sacrifice, dedication, and patience.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor, Dr. Richard Vuduc, for his support, encouragement and wisdom. He taught me the joys of research and I will solely miss the hours of discussions – which we never felt was enough – we had in his office. I can only hope to one day match his enthusiasm for research and his dedication to his students when I pursue a career in academia. His wisdom and patience that seem well beyond his years and his infectious enthusiasm in both the classroom and the research lab make him truly one–of–a–kind. I would also like to thank Agata for her support; she is most definitely an integral (and fun) part of being Rich's student.

I would also like to thank Dr. Scott Wills and Dr. Linda Wills. Scott and Linda were my Master's thesis advisors and they helped me through a very difficult time when I was making no progress on my research and was beginning to lose interest in it altogether; their patience, encouragement, and wisdom shaped me into the researcher that I am today, and for that I will forever be grateful. I greatly miss Scott's child–like energy and enthusiasm for research, and it is a great loss for the research community that he was taken away from us so prematurely.

I would like to thank the members of my Ph.D. thesis committee – Dr. George Riley, Dr. Sudhakar Yalamanchili, Dr. David Bader, and Dr. Richard Fujimoto – for their time and effort. I greatly appreciate all the valuable feedback I received during my defense, and hope to one day collaborate with and learn from them again. Among other faculty members at Georgia Tech, I would like to thank Dr. Edmond Chow and Dr. Hyesoon Kim in particular for their advice, encouragement, and feedback.

My time at Georgia Tech would not have been the same without other talented graduate students in our lab, especially Dr. Aparna Chandramowlishwaran, (soon to be Dr.) Kenneth Czechowski, Dr. Xing Liu, Karl Jiang, Marat Dukhan, Piyush Sao, and David Noble. Without them, graduate school would not have been as enjoyable and memorable, and I thank them for all that I have learned from them, about both research and life in general. I will greatly miss the time I spent with them as a graduate student.

A special and loving thanks goes to my girlfriend Jeongmin Yun for waiting patiently for me in Korea for 8 years. This thesis would have have been possible without her emotional support and love.

Most of all, I want to thank my parents for their love and support throughout my unusually long graduate study. They have sacrificed greatly – not only during my time as a graduate student, but since the day I was born – to make my life better and I can never pay them back for all that they have done for me. Thank you so much.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The overarching goal of this thesis is to provide an algorithm-centric approach to analyzing the relationship between time, energy, and power. This research is aimed at algorithm designers and performance tuners so that they may be able to make decisions on how algorithms should be designed and tuned depending on whether the goal is to minimize time or to minimize energy on current and future systems.

First, we present a simple analytical cost model for energy and power. Assuming a simple von Neumann architecture with a two-level memory hierarchy, this model predicts energy and power for algorithms using just a few simple parameters, such as the number of floating point operations (FLOPs or flops) and the amount of data moved (bytes or words). Using highly optimized microbenchmarks and a small number of test platforms, we show that although this model uses only a few simple parameters, it is, nevertheless, accurate.

We can also visualize this model using energy "arch lines," analogous to the "rooflines" in time. These "rooflines in energy" allow users to easily assess and compare different algorithms' intensities in energy and time to various target systems' balances in energy and time. This visualization of our model gives us many interesting insights, and as such, we refer to our analytical model as the *energy roofline model*.

Second, we present the results of our microbenchmarking study of time, energy, and power costs of computation and memory access of several candidate compute-node building blocks of future high–performance computing (HPC) systems. Over a dozen server-, desktop-, and mobile-class platforms that span a range of compute and power characteristics were evaluated, including x86 (both conventional and Xeon Phi accelerator), ARM, graphics processing units (GPU), and hybrid (AMD accelerated processing units (APU) and other system–on–chip (SoC)) processors.

The purpose of this study was twofold; first, it was to extend the validation of the energy roofline model to a more comprehensive set of target systems to show that the model works well independent of system hardware and microarchitecture; second, it was to improve the model by uncovering and remedying potential shortcomings, such as incorporating the effects of power "capping," multi–level memory hierarchy, and different implementation strategies on power and performance.

Third, we incorporate dynamic voltage and frequency scaling (DVFS) into the energy roofline model to explore its potential for saving energy. Rather than the more traditional approach of using DVFS to reduce energy, whereby a "slack" in computation is used as an opportunity to dynamically cycle down the processor clock, the energy roofline model can be used to determine *precisely* how the time and energy costs of different operations, both compute and memory, change with respect to frequency and voltage settings. This information can be used to target a specific optimization goal, whether that be time, energy, or a combination of both.

In the final chapter of this thesis, we use our model to predict the energy dissipation of a real application running on a real system. The fast multipole method (FMM) kernel was executed on the GPU component of the Tegra K1 SoC under various frequency and voltage settings and a breakdown of instructions and data access pattern was collected via performance counters. The total energy dissipation of FMM was then calculated as a weighted sum of these instructions and the associated costs in energy. On eight different voltage and frequency settings and eight different algorithm–specific input parameters per setting, for a total of 64 total test cases, the accuracy of the energy roofline model for predicting total energy dissipation was within 6.2%, with a standard deviation of 4.7%, when compared to actual energy measurements.

Despite its simplicity and its foundation on the first principles of algorithm analysis, the energy roofline model has proven to be both practical and accurate for real applications running on a real system. And as such, it can be an invaluable tool for algorithm designers and performance tuners with which they can more precisely analyze the impact of their design decisions on both performance and energy efficiency.

CHAPTER I

INTRODUCTION

Contents

The overarching goal of this thesis is to develop a simple explanation, aimed at algorithm designers and performance tuners, about the relationship between time, energy, and power. For that audience, a useful model would directly connect properties of an algorithm – such as concurrency and locality – with architectural time and energy costs. It would explain whether there is any difference in optimizing an algorithm for time versus optimizing for energy, why such difference exist, and what properties of the architecture might lead to non–trivial time–energy trade–offs.

1.1 A Roofline Model of Energy

To address this challenge, we start from a simple analytical model grounded in the first principles of algorithm analysis. Our analysis is inspired by a similar set of thought experiments based on "Amdahl" analysis written by and for architects [59, 128, 133]. Such analyses, based on Amdahl's Law [3], offer architectural insights, but abstract away essential properties of an algorithm. By contrast, our analysis more explicitly connects algorithmic and architectural parameters. However, for clarity we pose and study an intentionally simple – but not overly so – model, with some initial experimental tests to confirm its basic form. Our analytical model is known as the *energy roofline model*, and below, we summarize what our model implies. These claims both reflect familiar intuition and also yield new or alternative explanations about time, energy, and power relationships.

First, when analyzing time, the usual first–order analytic tool is to assess the *balance* of the processing system [80, 60, 61, 12, 89, 126], where balance is the ratio of work (e.g., flops) the system can perform per unit of data transfer (e.g., bytes). To this notion of time–balance, we define an *energy–balance* analogue, which measures the ratio of flops and bytes *per unit of energy* (e.g., Joules). We compare balancing computations in time against balancing in energy.

Second, we use energy–balance to develop an energy–based analogue of the time–based roofline model [126], which can be used to visualize our model. Because time can be overlapped, while energy cannot, the energy–based "roofline" (hence, the name *energy roofline model*) is actually a smooth "arch line." Interestingly, if time–balance differs from energy–balance, then there are distinct notions of being "compute bound" versus "memory bound," depending on whether the optimization goal is to minimize time or to minimize energy. We can measure this difference as a time–energy *balance gap*. We also posit an analogous "powerline" model for power.

Third, when a balance gap exists and energy–balance *exceeds* time–balance, the arch line predicts that optimizing for energy may be fundamentally *more difficult* than optimizing for time. It further suggest that high algorithmic energy efficiency may imply time efficiency, while the converse – that time efficiency implies energy efficiency – is *not* true.

Fourth, we test the basic form of the model using experiments on real CPU and GPU platforms. Using our model and these data, we show that the hypothetical balance gap described above does not yet really exist, which consequently explains why on today's platforms race–to–halt is likely to work well [6]. This raises the question for architects and hardware designers about what the fundamental *trends* in the balance gap will be: if energy–balance will eventually overtake time–balance, race–to–halt could break. We further use the experiments to highlight both the strengths and the limitations of our model and analysis.

Lastly, we ask under what general conditions we should expect an *algorithmic* time–energy trade–off. "Algorithmic" here stands in contrast to "architectural." Architecturally, for instance, increasing frequency reduces time, but increases energy, due to the non–linear relationship between frequency and power. What is the story for algorithms? We consider one scenario. Suppose it is possible algorithmically to trade more compute operations for less communication. One may derive a general necessary condition under which such a trade–off will improve energy efficiency. Furthermore, we show what improvements in energy efficiency may or may not require a slowdown, and by how much. Again, these conditions depend fundamentally on how time–balance compares to energy–balance.

Taken together, these theoretical analyses, supported by experiments on real platforms, can be used to improve the collective understanding of the relationship among algorithm properties and their costs in time, energy, and power.

1.2 *Algorithmic Time, Energy, and Power on Candidate High Performance Computing Building Blocks*

In this section, we extend and further validate the energy roofline model to increase its accuracy and usability. We claim three key contributions.

First, we add several important components to our model. These components include explicit modeling of a power cap and the energy cost of accessing different levels of the memory hierarchy. The power cap is especially significant, as it implies a way to predict power–throttling requirements. That is, if we wish to keep average power below some threshold, the model predicts by how much compute and memory operations should slow down. These additions come as a direct result of validating the model on real systems, where we discovered that certain systems operate under a strict power budget, and a power "cap" prevented them from achieving the expected peak performance. We also discovered that without individually accounting for the cost of accessing different levels of the memory hierarchy, it was difficult to predict energy consumption accurately for a real application.

Second, we compare the model to measurements on 12 platforms. These include x86 (both conventional, and Xeon Phi accelerators), several flavors of ARM, desktop and mobile GPUs and accelerated processing units (or APUs, such as those from AMD and other system–on–chip (SoC) manufacturers). These experiments validate the basic form of the *extended* model and yield empirical estimates of the effective energy required to perform flops and to move data. Breaking down the energy costs to these different components allows us to consider energy efficiency at different arithmetic intensity points, allowing more flexibility and analytical precision than simply dividing, say, peak performance by thermal design power (TDP). Moreover, these basic estimated constants may in and of themselves be useful reference values.

The microbenchmarks used in these measurements have been carefully tuned with the necessary architectural details in mind, in order to properly stress the targeted operations. These benchmarks, including the ones used in the previous section, have been written in an array of programming environments, including assembly [66], C (with SIMD intrinsics), CUDA [99], and OpenCL [2]. We have made these microbenchmarks available for download in a public repository as part of this thesis [1].

We claim our analysis and conclusion, drawn from the extensive experimental data collected from a dozen systems, as our final contribution. It includes comparing platforms on how power is allocated between memory and processing and the potential for reconfiguring power between memory and processing to improve or adapt energy efficiency to computation. It also includes comparing platforms for power throttling scenarios, where the "usable" power (i.e., power dedicated to computation and data movement, not including overhead) is reduced to satisfy a power budget and how such action impacts performance.

Beyond specific findings and data, we emphasize the methodological aspect of this section. In particular, architects may find our high–level approach to be a useful additional way to assess systems across computations; our analysis technique aims to provide more insight than a collection of blackbox benchmarks provides but without having to know too much detail about the specific computation. Similarly, we hope algorithm designers may find ways to reason about algorithmic techniques for managing energy and power, and trade–offs, if any, against time.

1.3 Adapting the Energy Roofline Model for Dynamic Voltage and Frequency Scaling

So far, we have assumed a fixed cost for all operations, in both time, and energy. This allowed us to keep our model and analysis simple and easy to understand; however, in more practical settings, DVFS can have a strong impact on energy efficiency by taking advantage of superlinear reduction in power with reduced frequency (and therefore

[1] http://hpcgarage.org/archline

voltage). We incorporate DVFS into the energy roofline model and validate it on a unique new SoC from NVIDIA.

We consider DVFS differently from how other have done in the past in that, rather than focusing on predicting "slack" in computation as an opportunity to arbitrarily lower frequency and voltage, we attempt to determine precisely by how much performance and energy of different operations change. Our new model can predict what the optimal frequency and voltage settings are for different computations, and tell us which algorithm would be better suited for a particular application under different constraints, such as a power cap. We show that our model is accurate to within 6.56% using 16–fold cross validation of our experimental data and to within 2.87% using the holdout method (2–fold cross validation).

We also evaluate and compare our new test platform, Jetson TK1, based on the Tegra K1 SoC which combines a single CUDA–capable SMX with a ARM Cortex A15 CPU, to high–end GPUs. Jetson TK1 is unique in that it is one of the first low–power platforms that can run scientific kernels and applications written in CUDA, and also in that it can vary frequency and voltage of both its processors and memory over a wide range, making it an ideal platform for DVFS studies.

We first compare Jetson TK1 to a GTX Titan, the highest–end desktop GPU from NVIDIA. In an iso–energy comparison, where we scale up the number of Jetson TK1 systems so that the aggregate TDP matches that of a single GTX Titan, we show that a 22× Jetson TK1 "cluster" outperforms GTX Titan over all arithmetic regimes in both performance and energy efficiency. When compared to a Tesla K40c GPU, the latest server–grade GPU, a cluster of 15× Jetson TK1 systems closely matches K40c in both performance and energy efficiency. Our study suggests that despite efforts in pushing mobile SoC for HPC [108, 106, 52], there is nothing fundamentally more, or less, energy efficient about low–power devices; that is, you get (in performance) what you pay for (in power).

1.4 Analysis of the Fast Multiple Method Using the Energy Roofline Model

As the final chapter of this thesis, we apply the energy roofline model to predicting the energy consumption of a real application on a real system. We target the fast multipole method (FMM) running on the CUDA–capable GPU of a Jetson TK1 system.

Given a system of N *source* particles, with positions given by $\{y_1, \ldots, y_N\}$, and N *targets* with positions $\{x_1, \ldots, x_N\}$, we wish to compute the N sums,

$$f(x_i) = \sum_{j=1}^{N} K(x_i, y_i) \cdot s(y_j), \quad i = 1, \ldots, N$$

where $f(x)$ is the desired *potential* at target point x; $s(y)$ is the *density* at source point y; and $K(x, y)$ is an *interaction kernel* that specifies "the physics" of the problem. For instance, the single-layer Laplace kernel, $K(x, y) = \frac{1}{4\pi} \frac{1}{||x-y||}$, might model electrostatic or gravitational interactions.

Evaluating these sums appears to require $O(N^2)$ operations. The FMM instead computes *approximations* of all of these sums in optimal $O(N)$ time with a guaranteed user-specified accuracy ϵ, where the desired accuracy changes the complexity constant [54]. As such, FMM is used in a variety of scientific simulations including electromagnetic, fluid, and gravitational phenomena [14], and has been hypothesized to be of increasing importance on future exascale systems [131].

We begin our analysis by using *nvprof*, a command–line performance counter monitor provided by NVIDIA, to break down the FMM kernel to its basic operations. These operations include various floating point instructions, such as fused–multiply–add (FMA), multiply, and add, as well as integer operations, which are typically used for loops and address calculation. We also break down data access to different levels of the memory hierarchy; for GPUs these are accesses to shared memory, L1 cache, L2 cache, and main memory.

Once we have this breakdown, we simply take a weighted sum of these operations with the derived energy costs (§ 5.1–5.3) to get the final energy consumption estimate. To get the energy dissipated by constant power, π_0, we also need the execution time of the kernel. Although we use measured execution time in our validation, we could use the modeling and analysis technique developed in our previous work [20, 23] to get a time estimate as well. The FMM implementation used in this section is also from our previous work [20, 23], where we developed a highly tuned CPU–GPU hybrid implementation of FMM.

For validation, we take eight different voltage and frequency settings and run eight different configurations of FMM for each setting, and then compare our energy estimates with physical measurements. We observed a mean error of 6.17% and a standard deviation of 4.65%. We found that although we use very basic operations to represent our algorithm, our model can, nevertheless, yield an accurate estimate of energy consumption under various DVFS settings.

CHAPTER II

PREVIOUS WORK

Contents

2.1 History

This is not the first time that power and cooling concerns have been raised in the history of computing. As early as 1947, ENIAC, which is considered to be the first digital computer, dissipated 174 KW using 17,468 vacuum tubes [11], a considerable amount of power considering its capability (or lack thereof). Although it consumed a large amount of power, cooling was not such a difficult problem because of the large physical size of a vacuum tube (i.e., areal power density, or watts per unit area, was relatively low).

Transition to relatively lower-power bipolar junction transistor (BJT) kept power dissipation in check. For example, the Intel 4004, the first commercially available microprocessor, was produced in 1971 and had a similar compute capability as the ENIAC while only dissipating a few watts. However, during the 1980s increasingly denser transistor integration led to a rapid rise in power dissipation and density. Power delivery and cooling once again became a concern and companies such as IBM and Cray produced liquid-cooled supercomputers in order to meet high performance targets [78].

In the early 1990s transition from bipolar to complementary metal-oxide-semiconductor (CMOS) transistors once again brought a temporary relief from power challenges. CMOS transistors had the appealing behavior of dissipating power only during switching transitions. The complementary gate structure meant that early gates drew little or no current between transition points because in a stable state the gate has no clear path to ground. Although some initially considered CMOS too slow for widespread use in high–performance computing, when power concerns became too great to deal with, widespread adoption of CMOS brought technology improvements that significantly improved their performance.

2.2 The Problem

Initially, most of the power dissipated by CMOS transistors came from switching (this is known as *dynamic* power). Due to the imperfect nature of transistors, there was also some *leakage* of current even when the transistor was not switching, or “turned off” (i.e. the supply voltage was below some threshold voltage). At first, this contributed only an insignificant amount of power (this is known as *leakage* or *static* power).

For decades, supply voltage dropped steadily with each technology generation [65, 64, 67] and because of its cubic influence on dynamic power, processor designers were able to keep the overall areal power density in check. However, when supply voltage is lowered, gate delay increases, and therefore the threshold voltage has to be lowered as well in order to maintain performance. Unfortunately, lowering threshold voltage has an *exponential* influence on leakage power; since no exponential can last forever, what was initially insignificant now makes up a significant proportion of the total power budget. Moreover, there is a limit to how low the threshold voltage can be set and once this limit is reached, changing supply voltage can only trade off power for performance, and scaling performance and maintaining constant areal power density will no longer be feasible. That is, more performance will necessarily mean more power consumption.

For future supercomputers, operational costs of powering and cooling the system are expected to exceed the cost of building them. Currently, the most "green" supercomputer in the world is the L-CSC supercomputer at the GSI Helmholtz Center for Heavy Ion Research which consists of Intel Xeon CPUs and AMD FirePro GPUs and is capable of processing 5.3 GFLOPs for every watt of power [53]. However, the next generation of supercomputers, expected to come into service around 2018 and perform at Exa-FLOPS levels, has been capped at 20 MW in power [42]. This is approximately equivalent to performing at 50GFLOPS/W. So, in order to build the next generation of supercomputers, the research community must come up with a way to improve power efficiency by a factor of 10 within the next few years! Considering that we have already hit a wall, this is a daunting task.

2.3 Current Research

The perspective of this work is algorithms, rather than architecture, systems, or embedded software, where time, power, and energy are traditionally studied (see the survey of Kaxiras et al. [72].) Our work is perhaps most similar to a recent work by Demmel et al. [34, 33, 49]. However, our model is more parsimonious and, as such, clarifies a number of issues such as the notion of a balance gap or why race-to-halt works on current systems. At a more technical level, we also differ in that we assume computation-communication overlap and have furthermore tried to validate the basic form of our model with experiments.

Additional algorithmic theory work The algorithms community has also considered the impact of energy constraints, particularly with respect to exploiting scheduling slack and there have been numerous other attempts to directly explore the impact of energy constraints on algorithms. These include new complexity models, including new energy-aware Turing machine models [121, 88, 68, 10]; this body of work addresses fundamental theoretical issues but is hard to operationalize for practical algorithm design and tuning. Other algorithmic work takes up issues of frequency scaling and scheduling [79, 5, 4]. Such models are particularly useful for exploiting slack to reduce energy by, for instance, reducing frequency of non-critical path nodes.

Systems-focused frequency scaling In more practical software-hardware settings, the emphasis is usually on reducing energy usage through Dynamic Voltage and Frequency Scaling (DVFS). DVFS attempts to minimize energy consumption with little or no impact in performance by scaling down the frequency (and therefore the voltage) when processor speed does not limit performance [47, 44, 71, 46, 114, 84, 1]. This work suggests a different flavor of time–energy trade–off, which comes from the superlinear scaling of power and energy with frequency, than what we consider in this paper. Among these, the work by Lively et al. uses principle component analysis (PCA) method to model execution time and power consumption using a small set of performance counters, which is then used to determine the appropriate DVFS and dynamic concurrency throttling (DCT) settings. It is, however, not explicit about algorithmic features such as intensity.

Dynamic concurrency throttling is a technique that limits the number of active cores to reduce energy consumption [27, 19, 28, 74], and is particularly effective when employing an additional core brings a limited performance improvement but a significant increase in power. This technique is, like DVFS, system–centric and does not provide algorithmic insight into energy efficiency.

Microbenchmarking studies Kestor et al. [75] and Molka et al. [93] have developed a microbenchmarking methodology similar to our own. In particular, Kestor et al. focus on measuring energy costs due to the memory hierarchy and communication. Our modeling approach differs, with power caps being its most notable distinction.

Diop et al. present a microbenchmark–based modeling technique for chip multiprocessors (CMP) and AMD APUs [35]. However, unlike ours, the microbenchmarks used in their work are not explicit about algorithmic properties and their methodology requires additional performance information from the Multi2Sim simulator [122] and performance monitoring counters (PMC).

Czechowski et al. also use microbenchmarks to evaluate a set of Intel CPUs that span a range of microarchitecture and process technology generations [31]. The authors demonstrate that various microarchitectural improvements in recent generations of Intel processors, and subsequent increase in the complexity of these "big cores," have contributed to improving both performance and energy efficiency. Their work is more focused on processor comparison than modeling, and is specific to only Intel processors.

Profiling and observation-based studies There are a number of empirical studies of time, power, and energy in a variety of computational contexts, such as linear algebra and signal processing [36, 43, 16, 48, 86, 41, 37, 13, 56]. One notable example is the work of Dongarra et al., which observes the energy benefits of mixed precision [36]; another is the study by Blem et al. [13] that debunks the myth that complex instruction set computing (CISC) instructions are inherently less energy efficient than reduced instruction set computing (RISC) instructions.

Esmaeilzadeh et al. measure chip power for a variety of applications, with a key high-level finding being the highly application-dependent behavior of power consumption [41]. They create abstract profiles to capture the differing characteristics of these applications. However, they do not ascribe specific properties of a computation in a way that programmers or algorithm designers can use to understand and change time–energy behavior.

Tools Although we adopted PowerMon 2 [8] as our measurement infrastructure, there are numerous other possibilities. Perhaps the most sophisticated alternative is PowerPack [48], a hardware-software "kit" for power and energy profiling. However, the infrastructure is relatively elaborate and expensive to acquire, in contrast to PowerMon 2. In future studies, we expect to be able to use even simpler measurement methods based on vendor-provided hardware support. These include Intel hardware counters for power [109] and NVIDIA's Management Library for power measurement [100]. However, unlike Intel's tool, NVML only provides power consumption monitoring for the entire GPU, including the memory. Also, NVML on certain GPU models show a number of anomalous behaviors that require correction [17]

In terms of tools that help with energy efficiency, there is eprof [111] that can pinpoint energy–hungry sections of code, and other methods that attempt to reduce energy consumption through various hardware features [90, 91].

Other modeling approaches The direct inspiration for this paper comes from studies of architecture-cognizant extensions to Amdahl's Law, balance, and the time-based roofline [32, 29, 59, 128, 133, 80, 60, 61, 12, 89, 126, 70, 73].

However, there are numerous other approaches. For instance, numerous recent studies have developed detailed GPU-specific models [62, 7, 132]; though these models are capable of directly predicting time, they require very detailed characterizations of the input program and/or intimate knowledge of the GPU microarchitecture. As such, it is nontrivial to translate the output of these models into actionable algorithmic or software changes. There are also numerous models that try to incorporate power and energy [46, 85, 119, 63, 114, 118, 115, 96, 57]. However, like the time-based models, much of this work is systems centric, abstracting away algorithmic properties.

Simulators for power and energy Other efforts to model power dissipation include CACTI [127], McPAT [82], and GPUWattch [81] simulators. The underlying models derive from device–level estimates of power dissipation for caches and processor cores. The tools enable design–space exploration by quantifying the cost of new features and materials over different process technology generations. However, CACTI and McPAT are only validated against other simulators or against a breakdown of TDP published by the vendors. GPUWattch validates against real measurements but only on NVIDIA GPUs. Also, it was noted by Nowitzki et. al [97] that all of these simulators are often overfitted to certain benchmarks or configurations for validation purposes, and can therefore have significant modeling errors. An interesting question may be to what extent these tools corroborate, complement or contradict our experimental data and modeling approach.

Low-power system for HPC Interest in embedded/mobile SoC for HPC has grown rapidly in recent years. The most notable and perhaps the largest work is the Mont Blanc project [108] which attempts to build a supercomputer from low–power, low–performance ARM SoCs. However, there are numerous others that also employ mobile SoC for HPC [45, 69, 117, 52, 87, 106, 95, 116, 107], as well as those that use them for more general–purpose applications such as computer vision and speech processing [55, 125, 124, 113, 22]

Notably, Fürlinger et al. [45] assembled a cluster of ARM Cortex-A8 based systems (Apple TV) to test the viability of an ARM-based system for HPC using the LINPACK benchmark. In the work by Reddi et. al [69], the authors consider low-power atom processor as an alternative to conventional server processors in the context of search engines. A study by Stanley-Marbell et. al [117] compares ARM, Atom, and Freescale systems in the context of dense linear algebra, graphs, and MapReduce. These studies are typically limited to specific systems or applications and cannot easily be extended to a wider range of architectures and workloads.

Architectural studies In the work by Hameed et al. [58], the authors use H.264 decoder as a case study to compare general purpose processors to ASICs in terms of energy efficiency. The authors show that ASICs achieve orders of magnitude higher energy efficiency by reducing instruction overhead that typically make up over 90% of the energy cost associated with each computation.

Others have found ways to *circumvent* the energy problem altogether by designing general purpose processors that use a different paradigm. Esmaeilzadeh et al. [38] propose a new architecture and programming framework for off-loading portions of annotated code that does not require exact computation to accelerators that approximates the result using neural networks. In [104, 105], the authors propose processors that are designed specifically for the sole purpose of computing matrix multiplication.

These solutions are unique and innovative, but less practical in the foreseeable future. One distinction of our study is its analysis of off-the-shelf components, which may be better suited to near-term design space exploration and experimentation.

Design space exploration for energy efficiency Hardware–software co–designing where traditional architecture design space exploration is tightly coupled with application tuning can lead to energy–efficient systems [92, 83]. In particular, Libuschewski et al. formulate this challenge as an optimization problem and utilize an evolutionary algorithm to automatically reach the optimal solution.

Others have proposed "overprovisioning" as a viable solution to tackling limited power budget for supercomputers [103, 40, 39]. Patki et al. [103] predict that in the future supercomputing centers will buy more compute capacity than can be used, and, instead of running them simultaneously as before, allow users to customize the system to the application to achieve better performance. The idea of "dark silicon" proposed by Esmaeilzadeh et al. [40, 39] suggests that we will not be able to power all transistors simultaneously (hence, there will be “dark” or unpowered transistors) in the future. In that case, these unpowered transistors may become specialized units and will only be turned on when the application can benefit from them, which is similar to the idea behind overprovisioning.

Metrics Our models reason directly about the basic measures of time, energy, and power. When considering trade–offs and multiobjective optimization, other metrics may be better suited. These include the energy delay product (EDP) and generalizations [51, 9], FLOP/s per Watt (i.e., flops per Joule) [112], and The Green Index [120].

CHAPTER III

ENERGY ROOFLINE MODEL

Contents

In this section, we describe the energy roofline model and demonstrate how this model can help guide analysis and tuning for energy efficiency. In § 3.1, we derive the basic model and define new terms, such as the energy–balance and the time–energy balance gap of a system. We also show how our model can be visualized using rooflines in energy (or arch lines), analogous to the rooflines in time [126], to aid in algorithmic energy–efficiency analysis. In § 3.2 we show how the balance gap impacts average power and discuss our experimental setup and microbenchmarks in § 3.3. In § 3.4, we discuss our fitted parameters and experimental result. We also apply our model to a real application to see how closely the model can predict energy consumption, and discuss why our predictions are not accurate and what can be

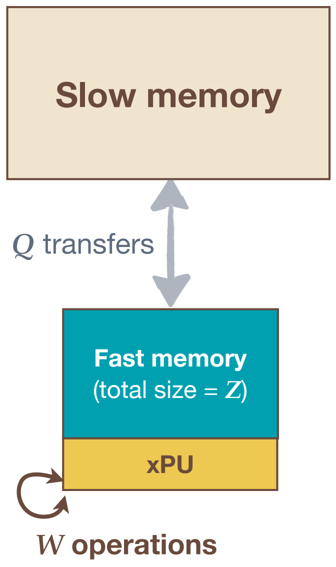


Figure 1: A simple von Neumann architecture with a two-level memory hierarchy. In our first analysis, suppose that an algorithm performs W arithmetic operations and Q memory operations, or "mops," between slow and fast memories.

done to remedy it. We conduct a hypothetical study on work–communication trade–off, where we reduce communication in exchange for extra computation, to test its viability for saving energy §3.5. Finally, we give our conclusion in §3.6.

3.1 A Basic Model and its Interpretation

Assume the simple architecture shown in fig. 1. This architecture has a processing element, labeled "xPU", as well as two levels of memory, namely, an infinite slow memory and a fast memory of finite capacity. This system roughly captures everything from a single functional unit (xPU) attached to registers (fast memory), to a manycore processor (xPU) attached to a large shared cache (fast memory). Further assume that the xPU may only perform operations on data present in the fast memory. As such, an algorithm for this architecture must explicitly move data between slow and fast memories.

3.1.1 Algorithm characterization

Let W be the total number of "useful" compute operations that the algorithm performs and let Q be the total amount of data it transfers between the slow and fast memories. (Table 1 summarizes all of the parameters of our model.) By *useful*, we mean in an algorithmic sense; for example, we might only count flops when analyzing matrix multiply, or comparisons for sorting, or edges traversed for a graph traversal algorithm. For simplicity, we will assume W is measured in units of scalar flops. (That is, a 4-way SIMD add is 4 scalar flops; a FMA is two scalar flops.) We will also refer to W as the total *work* of the algorithm. Regarding Q, we will for simplicity not distinguish between loads and stores, though one could do so in principle. We will refer to Q as mops measured in some convenient storage unit, such as a word or a byte.

In a typical algorithm analysis, both W and Q will of course depend on characteristics of the input, such as its size n;[1] in addition, Q will depend on the size of the fast memory. We discuss these dependences momentarily.

For performance analysis and tuning, we may measure the algorithm's *computational intensity*, which is defined as $I \equiv W/Q$. Intensity has units of operations per unit storage, such as flops per word or flops per byte. Generally speaking, a higher value of I implies a more "scalable" algorithm. That is, it will have more work than mops; therefore, it is more likely to improve as the architecture's compute throughput increases, which happens as cores increase or SIMD lanes widen.

What should we expect about the value of I? Recall that Q depends on fast memory capacity, which we denote by Z units of storage (words or bytes), as shown in fig. 1. Therefore, intensity will also depend on Z. A well-known result among

[1]That is, imagine a $W(n) = \mathcal{O}(n)$ style of analysis. However, unlike the traditional forms of such analysis, we will also want to characterize constants and costs much more precisely whenever possible.

Table 1: Summary of model parameters

Variable	Description
W	# of useful compute operations, e.g., # of flops
Q	# of main memory operations ("mops")
I	Intensity, or W/Q (e.g., flops per byte)
τ_{flop}	Time per work (arithmetic) operation, e.g., time per flop
τ_{mem}	Time per mop
B_τ	Balance in time, $\tau_{\text{mem}}/\tau_{\text{flop}}$ (e.g., flops per byte)
ϵ_{flop}	Energy per arithmetic operation
ϵ_{mem}	Energy per mop
B_ϵ	Balance in energy, $\epsilon_{\text{mem}}/\epsilon_{\text{flop}}$ (e.g., flops per Joule)
ϵ_0	Constant energy per flop
$\hat{\epsilon}_{\text{flop}} \equiv \epsilon_{\text{flop}} + \epsilon_0$	Minimum energy to execute one flop
η_{flop}	Constant–flop energy efficiency, $\frac{\epsilon_{\text{flop}}}{\epsilon_{\text{flop}}+\epsilon_0}$
π_0	Constant power, e.g., Joule per second = Watts
π_{flop}	Baseline power per flop excluding constant power, $\frac{\epsilon_{\text{flop}}}{\tau_{\text{flop}}}$
$\hat{B}_\epsilon(I)$	Effective energy-balance ($\pi_0 \geq 0$)
T_{flops}	Total time to perform arithmetic
T_{mem}	Total time to perform mops
T	Total time
E_{flops}	Total energy of arithmetic
E_{mem}	Total energy of mops
E_0	Total "constant" energy
E	Total energy
P	Average power
Z	Fast memory size (e.g., words, bytes)

algorithm designers is that no algorithm for $n \times n$ matrix multiply can have an intensity exceeding $I = \mathcal{O}\left(\sqrt{Z}\right)$ [70]. Consequently, if we improve an architecture by doubling Z , we will improve the inherent algorithmic intensity of a matrix multiply algorithm by no more than $\sqrt{2}$. Contrast this scenario to that of just summing all of the elements of an array. Intuitively, we expect this computation to be memory bandwidth bound if the array is very large. Indeed, it has an intensity of $I = \mathcal{O}(1)$, that is, a constant independent of problem size or Z . Thus, increasing Z has no effect on the intensity of this kind of reduction. In short, the concept of intensity measures the inherent locality of an algorithm.

3.1.2 Time and energy costs

Next, we translate the abstract W and Q into concrete time and energy costs. We will distinguish between the costs of performing work versus that of data transfer. Furthermore, our model of energy cost will have two significant differences from our model of time cost, namely, (i) time costs may be overlapped whereas energy may not; and (ii) we must burn *constant energy*, which is an additional minimum baseline energy on top of operation and data movement costs. These distinctions are critical, and together determine whether or not one should expect an algorithmic time–energy trade–off (see § 3.5).

More formally, suppose T_{flops} and T_{mem} are the total time (seconds) to execute all work operations and all mops, respectively. Further assume, *optimistically*, that overlap is possible. Then, the total time T is

$$T \equiv \max\left(T_{\text{flops}}, T_{\text{mem}}\right). \tag{1}$$

Similarly, suppose that E_{flops} and E_{mem} are the total energy (Joules) for work and mops. In addition, let $E_0(T)$ be the *constant energy* of the computation. Constant energy is the energy that must be expended for the duration of the computation, which we will further assume is a fixed cost independent of the type of operations

Table 2: Sample values for model parameters, based on best case (peak) capabilities of currently available systems. See table 1 for a summary of the definitions of these parameters.

	Representative values
Variable	NVIDIA "Fermi" GPU [123]
τ_{flop}	$(515\text{ Gflop/s})^{-1} \approx 1.9$ ps per flop[a]
τ_{mem}	$(144\text{ GB/s})^{-1} \approx 6.9$ ps per byte[b]
B_τ	$6.9/1.9 \approx 3.6$ flops per byte
ϵ_{flop}	≈ 25 pJ per flop[c]
ϵ_{mem}	≈ 360 pJ per byte
B_ϵ	$360/25 \approx 14.4$ flops per Joule

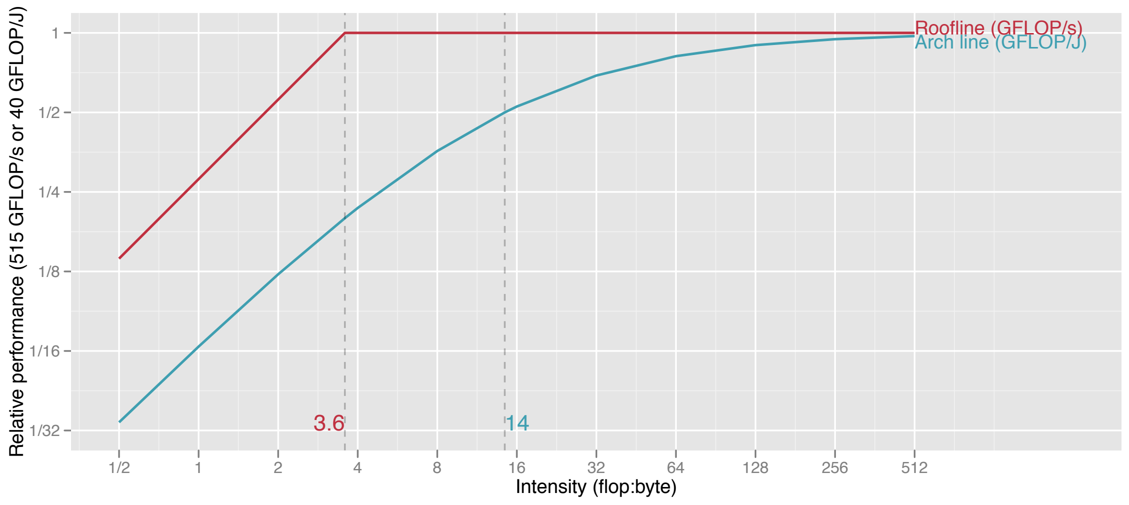
[a] Based on peak double-precision floating-point throughput.
[b] Based on peak memory bandwidth.
[c] Based on 50 pJ per double-precision fused multiply-add.

being performed. (see § 3.2 for a more detailed description of the constant term). Then, our model of energy cost is

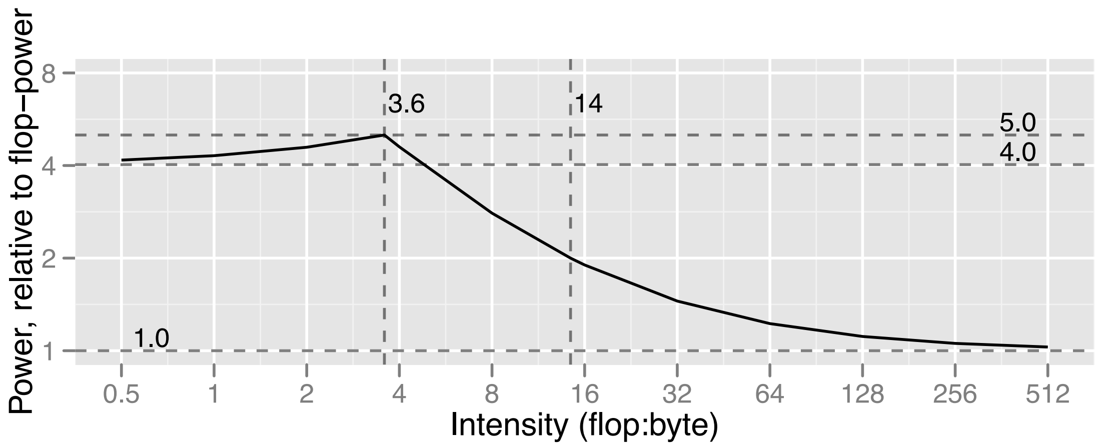
$$E \equiv E_{\text{flops}} + E_{\text{mem}} + E_0(T). \tag{2}$$

Consider the component costs, beginning with time. Suppose each operation has a fixed time cost. That is, let τ_{flop} be the time per work operation and τ_{mem} be the time per mop. We will for the moment tacitly assume *throughput*-based values for these constants, rather than latency-based values. (See table 2 for sample parameters.) This assumption will yield a best-case analysis,[2] which is only valid when an algorithm has a sufficient degree of concurrency; we discuss a more refined model based on work-depth in prior work [32]. From these basic costs, we then define the component times

[2]Additionally, assuming throughput values for τ_{mem} implies that a memory-bound computation is really memory *bandwidth* bound.



(a) Rooflines versus arch lines. The red line with the sharp inflection shows the roofline for speed; the smooth blue line shows the "arch line" for energy efficiency. The time- and energy-balance points (3.6 and 14 FLOP:Byte, respectively) appear as vertical lines and visually demarcate the balance gap.



(b) A "power-line" chart shows how average power (y-axis, normalized to π_{flop}) varies with intensity (x-axis, flop:byte). Going from bottom to top, the horizontal dashed lines indicate the flop power (y=1), the memory-bound lower limit (y=$\frac{B_\epsilon}{B_\tau}$=4.0), and the maximum power (y=$1 + \frac{B_\epsilon}{B_\tau}$).

Figure 2: Rooflines in time, arch lines in energy, and power lines. Machine parameters appear in table 2, for an NVIDIA Fermi-class GPU assuming constant power is 0. Dashed vertical lines show time- and energy-balance points.

as $T_{\text{flops}} \equiv W\tau_{\text{flop}}$ and $T_{\text{mem}} \equiv Q\tau_{\text{mem}}$. Then, under the optimistic assumption of perfect overlap, the algorithm's running time becomes

$$\begin{aligned} T &= \max\left(W\tau_{\text{flop}}, Q\tau_{\text{mem}}\right) \\ &= W\tau_{\text{flop}} \cdot \max\left(1, \frac{B_\tau}{I}\right), \end{aligned} \tag{3}$$

where we have defined $B_\tau \equiv \tau_{\text{mem}}/\tau_{\text{flop}}$. This quantity is the classical *time-balance point*, or simply *time-balance* [80, 60, 61, 12, 89, 126]. Time-balance is the architectural analogue of algorithmic intensity and has the same units thereof, e.g., flops per byte. Furthermore, if we regard $W\tau_{\text{flop}}$ as the ideal running time in the absence of any communication, then we may interpret $\frac{B_\tau}{I}$ as the *communication penalty* when it exceeds 1. We refer to this condition, $I > B_\tau$, as a *balance principle* [32]. Our algorithmic design goal is to create algorithms that minimize time and have high intensity relative to machine's time-balance.

We might hypothesize that a reasonable cost model of energy is similar to that of time. Suppose each work operation has a fixed energy cost, ϵ_{flop}, and for each mop a fixed cost, ϵ_{mem}. Additionally, suppose the constant energy cost is linear in T, with a fixed *constant power* of π_0 units of energy per unit time. (This notion of constant power differs from leakage power; see § 3.1.5.)

Then,

$$\begin{aligned} E &= W\epsilon_{\text{flop}} + Q\epsilon_{\text{mem}} + \pi_0 T \\ &= W\epsilon_{\text{flop}} \cdot \left(1 + \frac{B_\epsilon}{I} + \frac{\pi_0}{\epsilon_{\text{flop}}}\frac{T}{W}\right), \end{aligned} \tag{4}$$

where $B_\epsilon \equiv \epsilon_{\text{mem}}/\epsilon_{\text{flop}}$ is the *energy-balance point*, by direct analogy to time-balance. When π_0 is zero, B_ϵ is the intensity value at which flops and mops consume equal amounts of energy.

Let us refine eq. (4) so that its structure more closely parallels eq. (3), which in turn will simplify its interpretation. Let $\epsilon_0 \equiv \pi_0 \cdot \tau_{\text{flop}}$ be the *constant energy per flop*, that is, the energy due to constant power that burns in the time it takes to perform one flop. Moreover, $\hat{\epsilon}_{\text{flop}} \equiv \epsilon_{\text{flop}} + \epsilon_0$ becomes the actual amount of energy required to execute one flop, given non-zero constant power. Next, let $\eta_{\text{flop}} \equiv \epsilon_{\text{flop}}/\hat{\epsilon}_{\text{flop}}$ be the *constant flop energy fefficiency.* This machine parameter equals one in the best case, when the machine requires no constant power ($\pi_0 = 0$). Then, substituting eq. (3) into eq. (4) yields

$$E = W \cdot \hat{\epsilon}_{\text{flop}} \cdot \left(1 + \frac{\hat{B}_\epsilon(I)}{I}\right), \tag{5}$$

where $\hat{B}_\epsilon(I)$ is the *effective energy-balance*,

$$\hat{B}_\epsilon(I) \equiv \eta_{\text{flop}} B_\epsilon + (1 - \eta_{\text{flop}}) \max(0, B_\tau - I). \tag{6}$$

The ideal energy is that of just the flops, $W\hat{\epsilon}_{\text{flop}}$. On top of this ideal, we must pay an effective energy communication penalty, which $\hat{B}_\epsilon(I)/I$ captures. Therefore, similar to the case of execution time, our algorithmic design goal with respect to energy is to create work-optimal algorithms with an intensity that is high relative to machine's effective energy-balance. That is, just like time,‘ there is a balance principle for energy, $\hat{B}_\epsilon(I) \ll I$.

When $B_\tau = \hat{B}_\epsilon(I)$, optimizing for time and energy are most likely the same process. The interesting scenario is when they are unequal.

3.1.3 Rooflines in time and energy

We can visualize the balance principles of eqs. (3) and (5) using a roofline diagram [61, 126]. A roofline diagram is a line plot that shows how performance on some system varies with intensity. Figure 2a depicts the simplest form of the roofline, using the values for τ_{flop}, τ_{mem}, ϵ_{flop}, and ϵ_{mem} shown in table 2. Keckler et al. presented these values for an NVIDIA Fermi-class GPU [73]; but since they did not provide estimates for constant power, for now assume $\pi_0 = 0$. The x-axis shows intensity I. The y-axis shows performance, normalized either by the maximum possible speed (flops per unit time) or by the maximum possible energy efficiency (flops per unit energy). That is, the roofline with respect to time is the curve given by $W\tau_{\text{flop}}/T = \min(1, I/B_\tau)$ plotted against I. Similarly, the curve for energy is given by $W\hat{\epsilon}_{\text{flop}}/E = 1/(1 + \hat{B}_\epsilon(I)/I)$. In both cases, the best possible performance is the time or energy required by the flops alone.

The roofline for speed is the red line of fig. 2a. Since the component times may overlap, the roofline has a sharp inflection at the critical point of $I = B_\tau$. When $I < B_\tau$, the computation is memory bound in time, whereas $I \geq B_\tau$ means the computation is compute bound in time. Assuming that all possible implementations have the same number of operations, the algorithm designer or code tuner should minimize time by maximizing intensity according to the balance principle.

There is also a "roofline" for energy efficiency, shown by the smooth blue curve in fig. 2a. It is smooth since we cannot hide memory energy and since $\pi_0 = 0$. As such, we may more appropriately refer to it as an "arch line." The energy–balance point $I = B_\epsilon$ is the intensity at which energy efficiency is half of its best possible value.[3] Put another way, suppose W is fixed and we increase I by reducing Q. Then, B_ϵ is the point at which mops no longer dominate the total energy. In this sense, an algorithm may be compute bound or memory bound in *energy*, which will differ from time when $B_\tau \neq B_\epsilon$.

3.1.4 The balance gap

The aim of rooflines and arches is to guide optimization. Roughly speaking, an algorithm or code designer starts with some baseline having a particular intensity (x-axis value). A roofline or arch line provides two pieces of information: (a) it suggests the target performance tuning goal, which is the corresponding y-axis value; and (b) it suggests by how much intensity must increase to improve performance by a desired

[3]This relationship follows from $a + b \leq 2\max(a, b)$.

amount. Furthermore, it also suggests that the optimization strategies may differ depending on whether the goal is to minimize time or minimize energy.

The balance points tell part of the story. Ignoring constant power, we expect $B_\tau < B_\epsilon$. The reason is that wire length is not expected to scale with feature size [73]. An algorithm with $B_\tau < I < B_\epsilon$ is *simultaneously* compute bound in time while being memory bound in energy. Furthermore, assume that increasing intensity is the hard part about designing new algorithms or tuning code. Then, $B_\tau < B_\epsilon$ suggests that energy efficiency is even harder to achieve than time efficiency. The *balance gap*, or ratio B_ϵ/B_τ, measures the difficulty.

Having said that, a nice corollary is that energy efficiency may imply time efficiency. That is, $I > B_\epsilon$ implies that $I > B_\tau$ as well. However, the converse—that time efficiency implies energy efficiency—would not in general hold. Of course, roofs and arches are only bounds, so these high-level claims are only guidelines, rather than guaranteed relationships. Nevertheless, it may suggest that if we were to choose one metric for optimization, energy is the nobler goal.

If, on the other hand, $B_\tau > B_\epsilon$, then time efficiency would tend to imply energy efficiency. Under this condition, so-called *race-to-halt* strategies for saving energy will tend to be effective [6].[4]

Lastly, the analogous conditions hold when $\pi_0 > 0$, but with $\hat{B}_\epsilon(I)$ in place of B_ϵ. Higher constant power means lower η_{flop}; consequently, referring to eq. (6), it would cause $\hat{B}_\epsilon(I)$ to be lower than B_ϵ.

3.1.5 Interpreting constant power

Constant power in our model differs from conventional notions of static (or leakage) power and dynamic power [76]. Static power is power dissipated when current leaks through transistors, even when the transistors are switched off; dynamic power is

[4]The race-to-halt strategy says that the best way to save energy is to run as fast as possible and then turn everything off.

power due to switching (charging and discharging gate capacitance). These terms refer primarily to the device physics behind processor hardware, whereas constant power in our model represents a more abstract concept that depends on hardware *and* the algorithm or software that runs atop it.

With respect to hardware, constant power includes everything that is required to operate the device on top of leakage power. For example, constant power on a GPU will also include chips and circuitry on the printed circuit board, cooling fan, and other parts of the microarchitecture. These components may or may not be directly involved in computing or fetching data but would need to be on for the GPU to run at all.

Our constant power model does not explictly express the concept of dynamic power, which may lead to measurable inaccuracy. However, hardware designers are aggressively implementing techniques that can, for instance, turn off unused cores or aggressively gate clocks. These and other techniques tend to significantly reduce the impact of dynamic power.

With respect to software, our model of constant power can also capture inefficiencies in the algorithm or code. If a program is running inefficiently due to not having enough threads to saturate the processors or the memory units, there will be unused cores and/or longer instruction latencies due to stalls. The model includes such inefficiencies by charging a constant energy cost that depends on constant power and the total running time T.

3.2 What the Basic Model Implies About Power

Assuming our time and energy models are reasonable, we can also make analytic statements about the *average power* of a computation, $P \equiv E/T$.

Let $\pi_{\text{flop}} \equiv \epsilon_{\text{flop}}/\tau_{\text{flop}}$ be the *power per flop*. This definition excludes constant power.

Then, dividing eq. (5) by eq. (3) yields

$$P = \frac{\pi_{\text{flop}}}{\eta_{\text{flop}}} \left[\frac{\min(I, B_\tau)}{B_\tau} + \frac{\hat{B}_\epsilon(I)}{\max(I, B_\tau)} \right]. \tag{7}$$

The "power-line" diagram of fig. 2b depicts the most interesting features of eq. (7), again using the parameters of table 2 with $\pi_0 = 0$ ($\eta_{\text{flop}} = 1$). If the computation is severely memory bound ($I \to 0$), then $P \geq \pi_{\text{flop}} \frac{B_\epsilon}{B_\tau}$. If it is instead very compute bound ($I \to \infty$), P decreases to its lower limit of π_{flop}. Power P achieves its maximum value when $I = B_\tau$. From these limits, we conclude that

$$\pi_{\text{flop}} \frac{B_\epsilon}{B_\tau} \leq \quad P \quad \leq \pi_{\text{flop}} \left(1 + \frac{B_\epsilon}{B_\tau} \right). \tag{8}$$

Relative to π_{flop}, we pay an extra factor related to the balance gap. The larger this gap, the larger average power will be.

3.3 An experiment

The model of § 3.1 is an hypothesis about the relationships among intensity, time, and energy. This section tests our model on real systems.

3.3.1 Experimental Setup

Hardware Table 6 shows our experimental platforms, which include an Intel quad-core Nehalem CPU and two high-end consumer-class GPUs (Fermi and Kepler). We use two tools to measure power. The first is PowerMon 2, a fine-grained integrated power measurement device for measuring CPU and host component power [8]. The second is a custom in-house PCIe interposer for measuring GPU power. At the time of this writing, our consumer-grade NVIDIA GPU hardware did not support fine-grained power measurement via NVML, NVIDIA's API [100].

Figure 10 shows how the measurement equipment connects to the system. PowerMon 2 sits between the power supply unit and various devices in the system. It measures direct current and voltage on up to eight individual channels using digital

Table 3: Platforms – TDP is Thermal design power.

Device	Model	Peak performance Single (Double) GFLOP/s	Peak memory bandwidth GB/s	TDP Watts
CPU	Intel Core i7-950 (Nehalem)	106.56 (53.28)	25.6	130
GPU 1	NVIDIA GeForce GTX 580 (Fermi)	1581.06 (197.63)	192.4	244
GPU 2	NVIDIA GeForce GTX 680 (Kepler)	3532.8 (147.2)	192.2	190

power monitor integrated circuits. It can sample at 1024 Hz per channel, with an aggregate frequency of up to 3072 Hz. PowerMon 2 reports formatted and time-stamped measurements without the need for additional software, and fits in a 3.5 inch internal hard drive bay.

Modern high-performance GPUs have high power requirements. Typically, they draw power from multiple sources, including the motherboard via the PCIe connector. In order to measure the power coming from the motherboard we use a PCIe interposer that sits between the GPU and the motherboard's PCIe connector. The interposer intercepts the signals coming from the pins that provide power to the GPU.

Measurement method The GPU used in our study draws power from two 12 Volt connectors (8-pin and 6-pin) that come directly from the ATX PSU, and from the motherboard via the PCIe interface, which supply 12 V and 3.3 V connectors. When benchmarking the GPU, PowerMon 2 measures the current and voltage from these four sources at a regular interval. For each sample, we compute the instantaneous power by multiplying the measured current and voltage at each source and then sum over all sources. We can then compute the average power by averaging the instantaneous power over all samples. Finally, we compute the total energy by multiplying average power by total time. In this setup, we are able to largely isolate

GPU power from the rest of the system (e.g., host CPU).

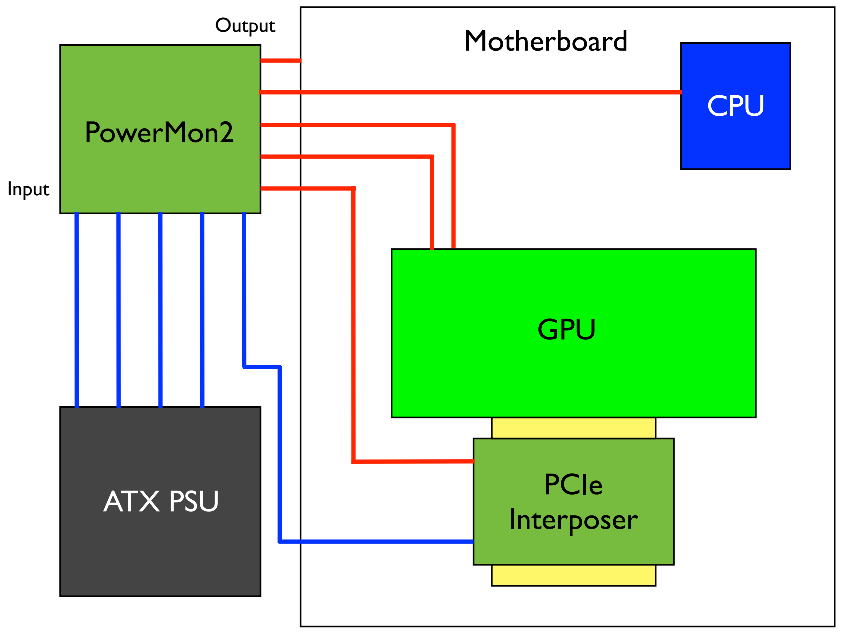


Figure 3: Placement of the measurement probes, PowerMon 2 [8] and our custom PCIe interposer

The PSU provides power to our CPU system using a 20-pin connector that provides 3.3 V, 5 V and 12 V sources and a 4-pin 12 V connector. As with the GPU, PowerMon 2 measures current and voltage from these four sources; we compute the average power and total energy in the same manner as above. For our CPU measurements, we physically remove the GPU and other unnecessary peripherals so as to minimize variability in power measurements.

In the experiments below, we executed the benchmarks 100 times each and took power samples every 7.8125 ms (128 Hz) on each channel.

3.3.2 Intensity microbenchmarks

We created microbenchmarks that allow us to vary intensity, and tuned them to achieve very high fractions of the peak FLOP/s or bandwidth that the roofline predicts. We then compared measured time and power against our model. The results for double precision and single precision appear in fig. 4 and fig. 5 respectively, with

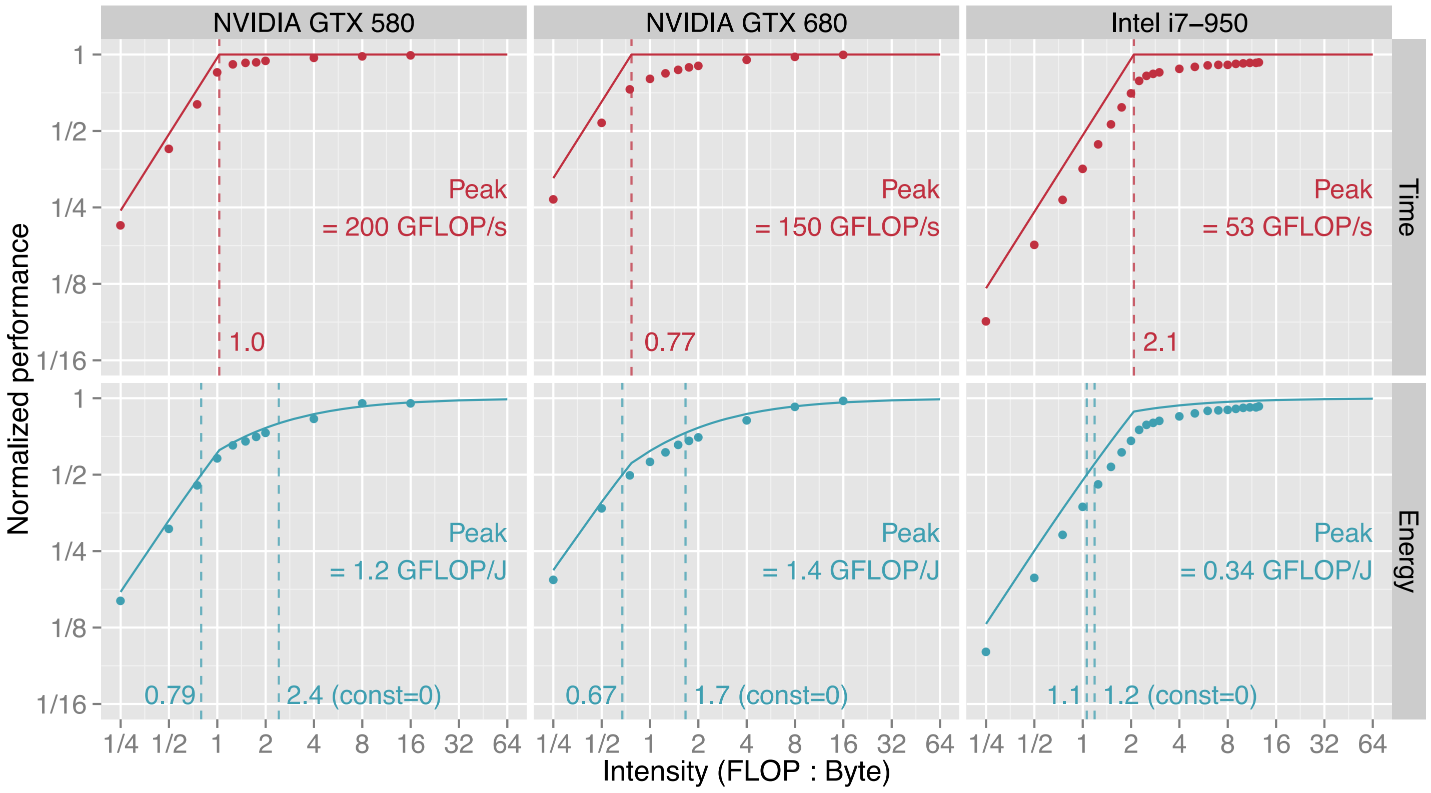


Figure 4: Measured time and energy for a double-precision synthetic benchmark corroborates eqs. (3) and (5). The impact of constant energy can be profound: GPUs have $\hat{B}_\epsilon < B_\tau < B_\epsilon$ (see vertical dashed lines). In other words, time efficiency implies energy efficiency *because of constant power*, which further suggests that "race-to-halt" is a reasonable energy-saving strategy; were $\pi_0 \to 0$, the situation could reverse.

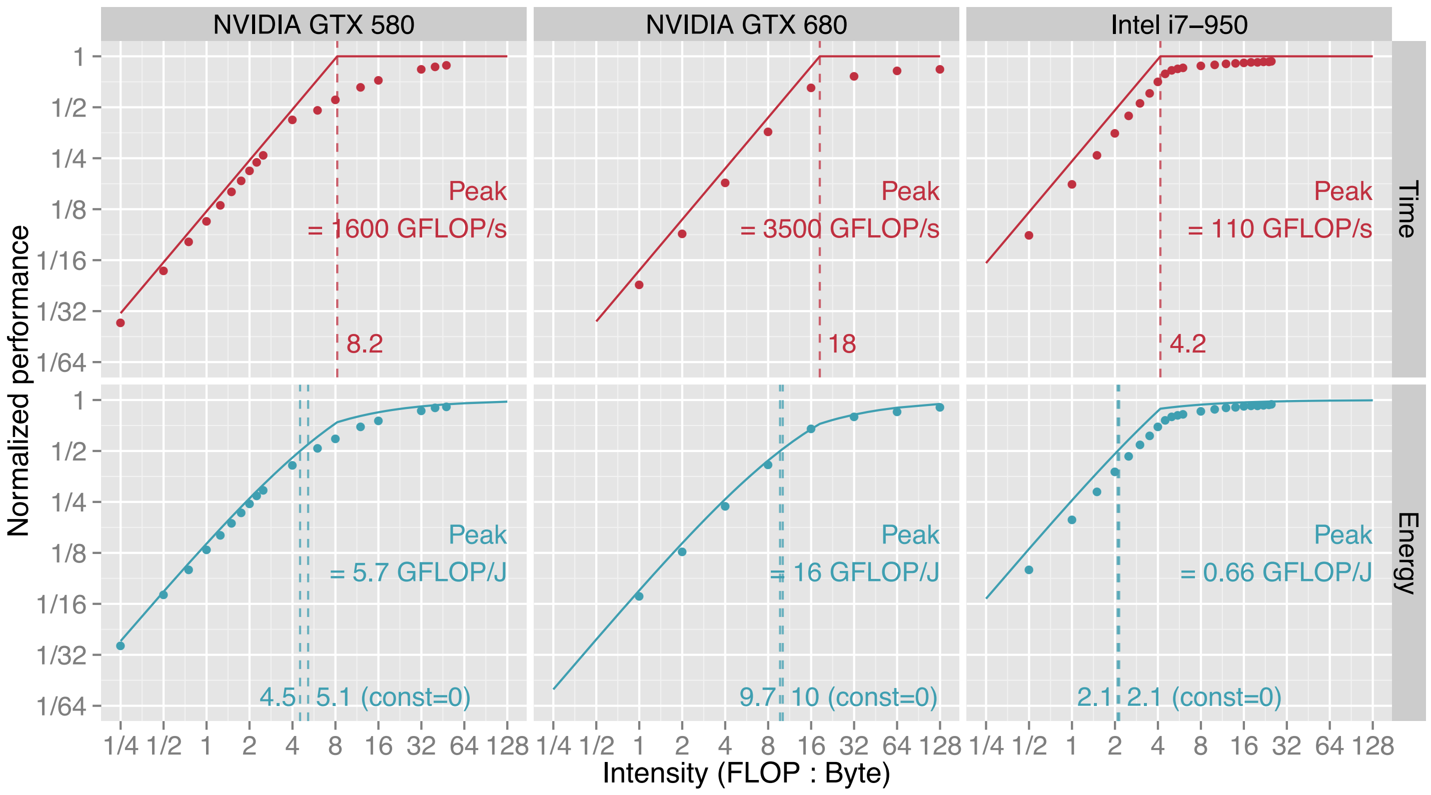


Figure 5: Same as fig. 4, but for single precision. In this case, all platforms have $\hat{B}_\epsilon \leq B_\epsilon < B_\tau$.

measured data (shown as dots) compared to our model (shown as a solid line). We describe these benchmarks and how we instantiated the model below[5].

The GPU microbenchmark streams a multi-gigabyte array and, for each element, executes a mix of independent FMA operations. We autotuned this microbenchmark to maximize performance on the GPU by tuning kernel parameters such as number of threads, thread block size, and number of memory requests per thread. The GPU kernel is fully unrolled. To verify the implementation, we inspected the PTX code and compared the computed results against an equivalent CPU kernel. The CPU microbenchmark evaluates a polynomial and is written in assembly, tuned specifically to maximize instruction throughput on a Nehalem core. Changing the degree of the polynomial effectively varies the computation's intensity. The kernel is parallelized using OpenMP to run on all four cores. Although the CPU and GPU benchmarks differ, their intent is simply to permit varying of intensity and achieving performance as close to the roofline as possible. As such, what they compute is not as important as being highly tuned and having controllable intensity.

Figures 4 and 5 show that both microbenchmarks perform close to the roofline in most cases. Refer specifically to the "Time" subplots, where the roofline is drawn using peak GFLOP/s and bandwidth numbers from table 6. For instance, on the GTX 580 the double-precision version of the GPU benchmark achieves up to 170 GB/s, or 88.3% of system peak when it is bandwidth bound, and as much as 196 GFLOP/s, or 99.3% of system peak when it is compute bound. For single precision, the kernel performs up to 168 GB/s and 1.4 TFLOP/s respectively. However, performance does not always match the roofline. Again on the NVIDIA GTX 580, we see the largest gap near the time-balance point; this gap is much smaller on the GTX 680. We revisit this phenomenon in light of our model in § 3.4.2.

Percentage of system peak performance observed on GTX 680 was somewhat

[5]The microbenchmarks are available for download at `http://hpcgarage.org/archline`

Table 4: Fitted energy coefficients. Note that ϵ_{mem} is given in units of picoJoules per *Byte*. As it happens, the π_0 coefficients turned out to be identical to three digits on GTX 580 and i7-950 which are built on 40 nm and 45 nm technologies respectively, whereas GTX 680 is built on a significantly lower technology of 28 nm.

	NVIDIA GTX 680	NVIDIA GTX 580	Intel Core i7-950
ϵ_{s}	43.2 pJ /FLOP	99.7 pJ / FLOP	371 pJ / FLOP
ϵ_{d}	262.9 pJ / FLOP	212 pJ / FLOP	670 pJ / FLOP
ϵ_{mem}	437.5 pJ / Byte	513 pJ / Byte	795 pJ / Byte
π_0	66.37 Watts	122 Watts	122 Watts

lower than that of GTX 580 and it took slightly more effort and tuning to achieve it. The maximum observed bandwidth was 147 GB/s, or 76.6% of system peak in both single precision and double precision. The maximum observed performance when the benchmark was compute bound was 148 GFLOP/s, or 101% of system peak in double-precision and 3 TFLOP/s, or 85.6% of system peak in single precision. These percentages are based on the 1150 MHz "Boost Clock" specification. We speculate that these variations (even exceeding 100%) are due to the aggressive DVFS employed on Kepler GPUs which allows over-clocking from the "Base Clock" of 1006 MHz as long as the GPU remains under its TDP, even exceeding the Boost Clock.[6]

The CPU microbenchmark achieves up to 18.7 GB/s and 99.4 GFLOP/s, or 73.1% and 93.3% of peak in single precision performance. The achieved bandwidth is similar to that of the STREAM benchmark[7] and the lower fraction of peak bandwidth observed is typical for CPU systems. Double-precision performance is 18.9 GB/s (73.8%) and 49.7 GFLOP/s (93.3%), respectively.

Model instantiation To instantiate eq. (3), we estimate time per flop and time per mop using the inverse of the peak manufacturer's claimed throughput values as shown in table 6. For the energy costs in eq. (5), such specifications do not exist.

[6] `http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf`

[7] `streambench.org`

Therefore, we estimated them using linear regression on our experimental data.[8] In particular, the data points are a series of 4-tuples (W, Q, T, R), where we choose W and Q when running the microbenchmark, T is the *measured* execution time, and R is a binary variable set to 0 for single precision and 1 for double precision. We use linear regression to find the coefficients of the model,

$$\frac{E}{W} = \epsilon_{\text{s}} + \epsilon_{\text{mem}}\frac{Q}{W} + \pi_0\frac{T}{W} + \Delta\epsilon_{\text{d}}R, \tag{9}$$

which yields the energy per single-precision flop, ϵ_{s}; energy per single-precision word, ϵ_{mem}; constant power, π_0; and $\Delta\epsilon_{\text{d}}$, which is the *additional* energy required for a double-precision flop over a single-precision flop.[9] That is, the energy per double-precision flop is $\epsilon_{\text{d}} \equiv \epsilon_{\text{s}} + \Delta\epsilon_{\text{d}}$. We summarize the fitted parameters in table 4. We then plug these coefficients into eq. (5) to produce the model energy curves shown in fig. 5 and fig. 4. These curves visually confirm that the fitted model captures the general trend in the data. We analyze these curves in § 3.4.

3.3.3 Cache microbenchmarks

Exploiting data locality is the main algorithmic tool for controlling I, though so far we have ignored the cost of explicit cache access. Let Q_{cache} be the number of cache accesses (measured in words) that our computation incurs, which are distinct from the Q that we may assume all go to main memory. We may modify eq. (4) to account for Q_{cache} as follows, assuming a per-cache access energy cost of ϵ_{cache}:

$$E = W\epsilon_{\text{flop}} + Q\epsilon_{\text{mem}} + Q_{\text{cache}}\epsilon_{\text{cache}} + \pi_0 T. \tag{10}$$

This equation assumes the two-level memory hierarchy of fig. 1. It would be straightforward to add terms for a memory hierarchy with more than two levels.

[8] We use the standard regression routine in R, `r-project.org`.

[9] Normalizing the regressors by W produces high-quality fits, with R^2 (residual) coefficient near unity at p-values below 10^{-14}.

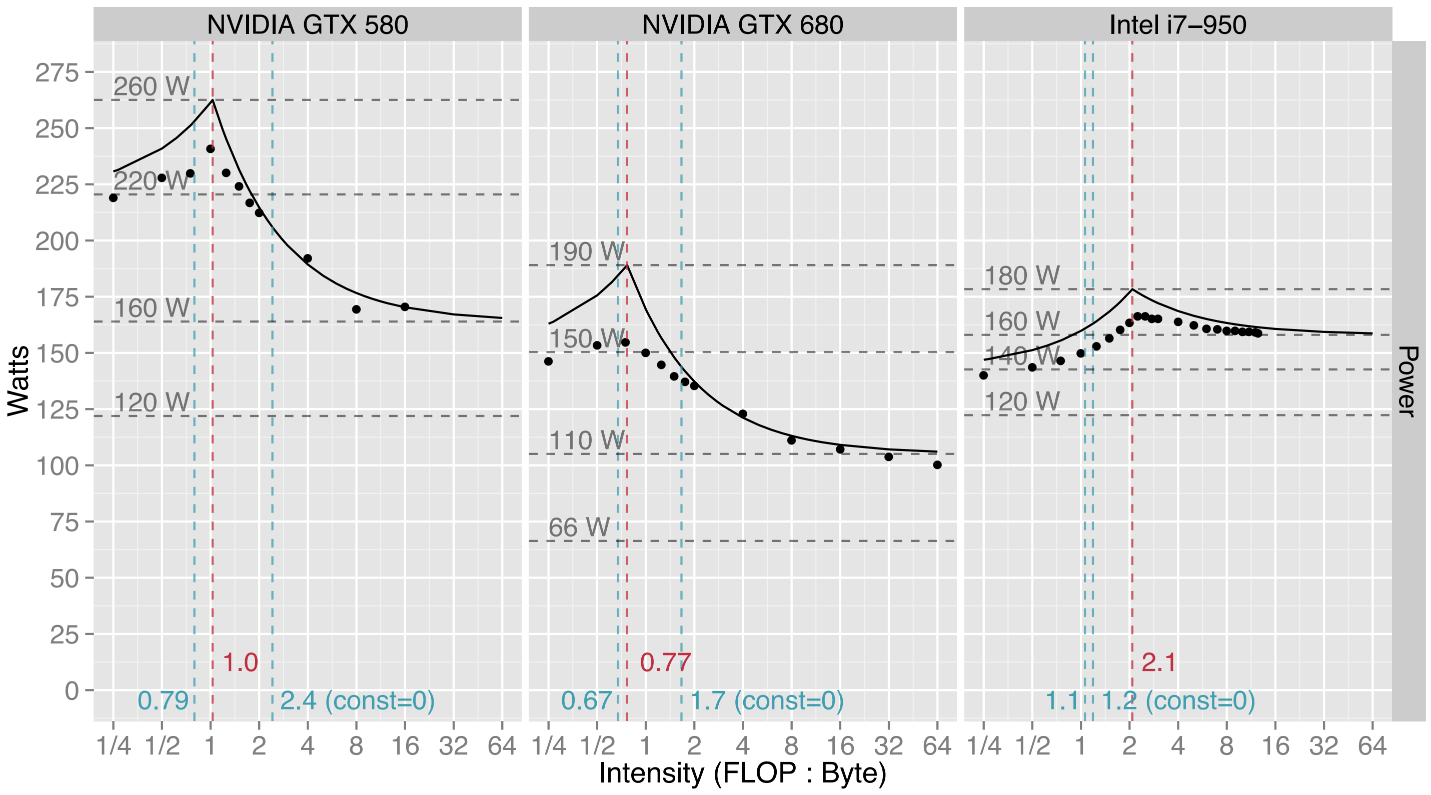


Figure 6: Measured power for the double-precision microbenchmark corroborates the "powerline" model. On the GTX 580 platform, NVIDIA reports a limit of 244 Watts, which explains the discrepancy between the observed data and the predicted powerline in the single-precision GTX 580 case.

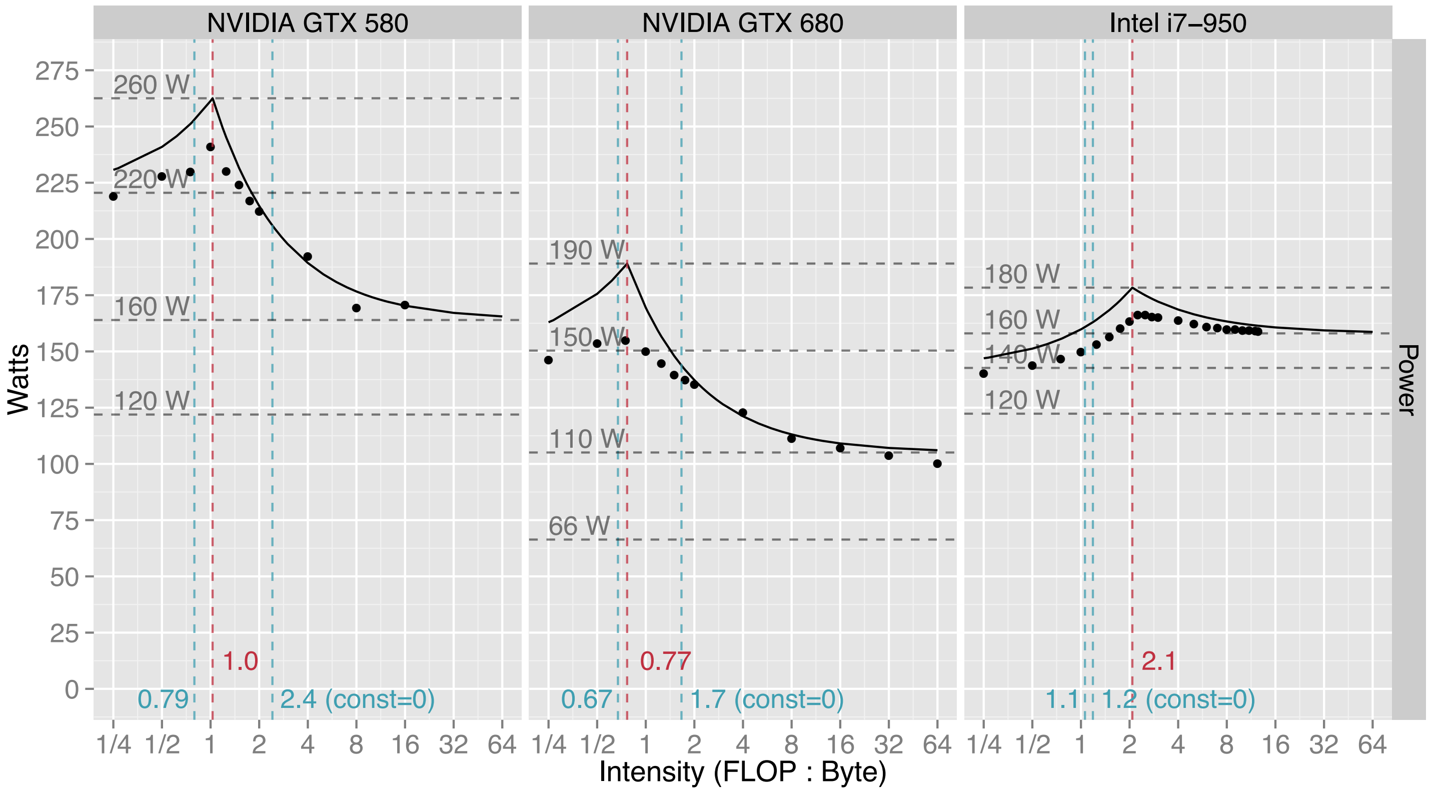


Figure 7: Measured power for the single-precision microbenchmark .

We created a pointer-chasing microbenchmark for the GPU to help measure cache access energy. First, consider the last-level L2 cache on Fermi- and Kepler-class GPUs. Our benchmark computes `k = A[k]`, where initially `k == A[k] ==` the local thread ID. As such, the first thread block to access the array will fetch data from main memory into cache; threads from other thread blocks with the same local thread ID effectively fetch that same element over and over from cache. The number of threads is limited so that the entire array fits in cache. We compile the microbenchmark with `-Xptxas -dlcm=cg`, which forces the compiler to generate code that caches data in the L2 cache only. Let ϵ_{L_2} be the L2 cache energy access cost; using our previously fitted values of ϵ_{mem} and π_0, we can compute ϵ_{L_2} via

$$\epsilon_{\mathrm{L}_2} = \frac{E - W\epsilon_{\mathrm{flop}} - Q\epsilon_{\mathrm{mem}} - \pi_0 T}{Q_{\mathrm{L}_2}} \tag{11}$$

where we use performance counters to estimate the number of L2 accesses, Q_{L_2}.

We can adjust then reuse this microbenchmark to estimate L1 cache energy cost. First, recall that the GPU L1 caches are private to each multiprocessor. When L1 caching is enabled, the very first thread block to fetch a particular array element will load the corresponding cache line into both the L2 cache and the L1 cache of the multiprocessor in which it resides. Then, all *subsequent and first* thread blocks in each multiprocessor will fetch the array from the L2 cache into their respective L1 caches. After that, subsequent thread blocks will then access this array from their own L1 caches. Since we have already calculated the cost of fetching data from the L2 cache (ϵ_{L_2}), the energy cost ϵ_{L_1} of fetching data from the L1 cache can be computed using

$$\epsilon_{\mathrm{L}_1} = \frac{E - W\epsilon_{\mathrm{flop}} - Q\epsilon_{\mathrm{mem}} - \pi_0 T - Q_{\mathrm{L}_2}\epsilon_{\mathrm{L}_2}}{Q_{\mathrm{L}_1}}. \tag{12}$$

This scheme works for NVIDIA Fermi-class GPUs, but we must modify it for those based on Kepler. There, the L1 cache is reserved exclusively for spilling local

Table 5: Estimated energy costs of explicitly fetching data from L1 and L2 caches on GTX 580, and from shared memory and L2 on the GTX 680.

	NVIDIA GTX 680	NVIDIA GTX 580
ϵ_{L_1}	51 pJ /Byte (shared memory)	149 pJ / Byte
ϵ_{L_2}	195 pJ / Byte	257 pJ / Byte

data.[10] Therefore, we instead estimate the energy cost of fetching data from the explicitly-programmed scratchpad memory, or "shared memory" in NVIDIA CUDA parlance. However, doing so will have two consequences. First, we may observe more L2 cache accesses than we would if we used L1. The reason is that a copy of the array is required *per thread block*, rather than *per multiprocessor*. Secondly, we may observe different effective energy costs, since the shared memory and L1 cache hardware implementations differ. The reader should, therefore, interpret "L1" energy cost estimates on Kepler accordingly.

The estimated costs of fetching data from L1 and L2 caches on Fermi and Kepler-class GPUs appear in table 5, where ϵ_{L_1} is the L1 (or shared memory) access cost, and ϵ_{L_2} is the L2 cost. The median relative residuals between the fitted model and the measured microbenchmark data were less than 4.5% on the GTX 680, and less than 4% on the GTX 580.

As expected, the costs of fetching data from L1 and L2 caches are much smaller on the Kepler-class GPU. Note that Kepler-based GPUs use a better process technology than Fermi (28 nm vs. 40 nm on Fermi). We will analyze these estimates in § 3.4.1.

3.4 Discussion, Application, and Refinement

3.4.1 The fitted parameters

We may compare the fitted parameters of table 4 against those that Keckler et al. provide [73]. Since they only discuss Fermi-class GPUs, giving estimates for only 45 nm and 10 nm process technologies, we will limit the following discussions of our

[10]See: `http://docs.nvidia.com/cuda/kepler-tuning-guide/index.html`

energy cost estimates to the GTX 580.

First, Keckler et al. state that the energy cost of the floating-point unit that performs one double-precision FMA is about 50 pJ, or 25 pJ *per flop*; our estimate in table 4 is about eight times larger. This discrepancy arises because the 50 pJ FMA cost excludes various instruction issue and microarchitectural overheads (e.g., registers, component interconnects), which our measurement implicitly includes. Based on our estimates, these overheads account for roughly 187 pJ/flop.

Second, the discussion of Keckler et al. on memory access costs suggests a baseline memory–energy cost of 253–389 pJ per Byte. This cost includes DRAM access costs, interface costs, and wire transfer. However, this estimate ignores instruction overheads and possible overheads due to cache. Recall that we estimated the instruction overhead for a floating point instruction to be roughly 187 pJ, or approximately 47 pJ/Byte in single precision. Adding this number to the baseline produces an estimate of 300-436 pJ/Byte. We also have to account for the costs of storing and reading the data from the L1 and L2 caches as it travels up the memory hierarchy. From Keckler et al.'s paper, we can estimate this cost to be approximately 1.75 pJ/Byte per read/write for both L1 and L2 (assuming they are both implemented using SRAM), or a total of 7 pJ/Byte for both L1 and L2 read and write traffic. This brings the total cost estimate to 307–443 pJ/Byte. Our estimate of ϵ_{mem} is larger, which may reflect additional overheads for cache management, such as tag matching.

Another point of note is the cost of fetching data from the L1 and L2 caches. Since the cost of reading the data from SRAM is only 1.75 pJ/Byte and the cost of transferring the data over the wire is roughly 10–20 pJ/Byte, instruction overhead accounts for most of the cost. Streamlining the microarchitecture to reduce this overhead is an opportunity for future work.

There is no information provided to check our constant power estimate. For reference, we measured true GPU idle power—when the GPU is on but not running anything—to be approximately 40 Watts. Thus, what we estimate as constant power is not the same as idle power.

The estimates of CPU energy costs for both flops and memory are higher than their GPU counterparts. This observation is not surprising since a CPU processing core is widely regarded as being more complex than its GPU counterpart. Similarly, memory energy costs are higher in the CPU system than the GPU system. A likely explanation is that GPU memory sits closer to the GPU processor than CPU memory does to the CPU processor. All of these characteristics have a profound impact on the balance gap, discussed next.

3.4.2 Balance gaps and power caps

Consider the rooflines and arch lines of fig. 4 and fig. 5. In all cases, the time–balance point exceeds the y=1/2 energy–balance point, which means that time efficiency will tend to imply energy efficiency. That is, once the microbenchmark is compute bound in time ($I > B_\tau$), it is also within a factor of two of the optimal energy efficiency. We believe this observation explains why race-to-halt can be such an effective energy-saving strategy in practice on today's systems [6].

If instead it were possible to drive $\pi_0 \to 0$, then the situation could reverse. In the two bottom-left subplots of fig. 4, we show this scenario using the hypothetical energy-balance lines labeled, "const=0." However, also observe that having $\pi_0 = 0$ does not invert the balance gap on the Intel platform. As table 4 suggests, ϵ_{flop} and ϵ_{mem} on the Intel platform are much closer than they are on the NVIDIA platform. Reflecting on these two types of systems, one question is to what extent π_0 will go toward 0 and to what extent microarchitectural inefficiences will reduce.

As noted previously, the single-precision NVIDIA GTX 580 performance in fig. 5 does not track the roofline closely in the neighborhood of B_τ. The reason is that our model does not include explicit power caps. To see this effect, refer to the powerlines of fig. 6 and fig. 7 and the theory of fig. 2b. Our model demands that power increase sharply as I approaches B_τ (see § 3.2). For instance, on the GPU in single-precision, our model says we will need 387 Watts on the GPU as shown in fig. 7. This demand would in reality cause excessive thermal issues. Indeed, the GTX 580 has a maximum power rating of 244 Watts, which our microbenchmark already begins to exceed at high intensities. Thus, incorporating power caps will be an important extension for future work.

3.4.3 Applying and refining the model: FMM_U on the GPU

To see how well we can estimate time and energy using our model, we apply it to the FMM. The FMM is an $O(n)$ approximation method for n-body computations that would otherwise scale as $O(n^2)$ [54]. We specifically consider the most expensive phase of the FMM, called the *U-list* phase (FMM_U). For this exercise, we consider just a GPU version of FMM_U.

Algorithm 3.4.1 The FMM_U algorithm

```
1: for each target leaf node, $B$ do
2:     for each target point $t \in B$ do
3:         for each neighboring source node, $S \in U(B)$ do
4:             for each source point $s \in S$ do
5:                 $(\delta_x, \delta_y, \delta_z) = (t_x - s_x, t_y - s_y, t_z - s_z)$
6:                 $r = \delta_x^2 + \delta_y^2 + \delta_z^2$
7:                 $w = \text{rsqrtf}(r)$ {Reciprocal square-root}
8:                 $\phi_t += d_s * w$ {$d_s$ and $\phi_t$ are scalars}
```

Algorithm sketch The FMM_U phase appears as pseudocode in Algorithm 3.4.1. The n points are arranged into a spatial tree, with leaf nodes of the tree containing a subset of the points. For every leaf node B, FMM_U iterates over its neighboring leaf nodes. The list of neighbors is called the "U-list," denoted as $U(B)$. The node B is the *target* node, and each neighbor $S \in U(B)$ is a *source* node. For each pair (B, S), FMM_U iterates over all pairs of points $(t \in B, s \in S)$ and updates ϕ_t, a value associated with the target point t. According to lines 5-8, each pair of points involves 11 scalar flops, where we count "reciprocal square-root" $(1/\sqrt{r})$ as one flop. Furthermore, each leaf contains $O(q)$ points for some user-selected q; the number of flops is therefore $O(q^2)$ for every $O(q)$ points of data, with q typically on the order of hundreds or thousands. Thus, the FMM_U phase is compute bound.

In prior work, we generated approximately 390 different code implementations of this benchmark [20, 23]. These variants use a variety of performance optimization techniques and tuning parameter values.

Fitting We use the values in table 4 and table 5 to estimate the total energy cost of each FMM_U implementation on the NVIDIA GTX 680. All implementations are in single precision. We derive the number of flops from the input data and the number of bytes read from the DRAM and caches using hardware counters provided by NVIDIA's Compute Visual Profiler. The average error for estimated energy consumption as compared to measured was 32%, which is much worse than our cache microbenchmark.

There are two culprits. First, we count only flops, thereby ignoring the overhead of integer instructions, branches, and other non-flop operations. Secondly, our method of estimating flop energy does not distinguish between types of flops. Recall that our flop energy estimate is half the cost of a FMA instruction. Our simple estimate will usually underestimate the true energy consumption of multiplies, divisions, and (reciprocal) square roots, for instance. For the 390 FMM_U implementations, we always underestimated total energy.

3.4.4 Refining the performance estimate

Though compute bound, the FMM_U instruction mix is more complex than a series of FMAs. As such, we should evaluate time and energy with respect to a refined notion of peak that accounts for the instruction mix. Below, we show how to do so for the

GTX 580 (Fermi). A similar line of reasoning is possible for the GTX 680 (Kepler).

The intrinsic operations that the FMM_U must perform require 11 scalar flops, where reciprocal square root counts as a single flop. These flops occur over eight operations: one transcendental (reciprocal square root), three FMAs, and four individual floating–point add and subtract operations.

To achieve peak on the GTX 580 (Fermi), consider its microarchitecture. The GPU processor has 512 clusters of functional units, called "CUDA cores," spread among 16 SM units. Each of these cores runs at 1.54 GHz. Thus, the processor can perform (512 scalar instructions) times (1.54 GHz) = 788 billion instructions per second. If each instruction is a scalar FMA, the peak performance is 1.58 TFLOP/s. However, there is a restriction owing to the dual-warp schedulers in each SM. A warp is a group of 32 threads that execute instructions in lock-step.[11] An SM may select one instruction each from two independent warps in a cycle. That is, to achieve peak there must be at least two independent warps each with a ready FMA in every cycle. Otherwise, achieving peak is impossible.

For transcendental instructions like reciprocal square root, there are more limits. Only one of the two warp schedulers in an SM may issue a transcendental operation each cycle. Furthermore, there are only four special function units (SFUs) capable of executing such operations on each SM. Therefore, to issue a transcendental operation for a warp requires (32 threads) divided by (4 SFUs per cycle) = 8 cycles.

The key to deriving a refined estimate of peak, given the architecture and instruction mix, lies with the instruction issue rate and the dual-warp scheduler. The FMM_U instruction mix requires at least 3 cycles (for the FMAs) + 4 cycles (adds and subtracts) + 8 cycles (reciprocal square root) = 15 cycles. However, the issue rate varies during these 15 cycles: when issuing FMAs and other floating point instructions, if there is at least one other warp available to issue the same type of

[11]This style of execution is sometimes called SIMT execution.

instructions, the SM will issue a total of 32 instructions. However, when issuing reciprocal square roots, no more than 4+16=20 instructions may issue if there is at least one other warp available that can issue FMAs or floating point instructions. Therefore, the average issue rate is $(7 \times 32 + 8 \times 20)/15 = 25.6$ scalar instructions per cycle. However, this does not differentiate FMAs and floating point instructions. As such, we can redefine issue rate in terms of flops/cycle. In this case, the average issue rate is $(3 \times 64 + 4 \times 32 + 8 \times 20)/15 = 32$ flops/cycle. Then the maximum performance is (32 flops/cycle/SM) $\times$ (16 SMs) $\times$ (1.54 GHz) = 788 GFLOP/s.

To verify this estimate, we wrote a synthetic benchmark where each thread executes many independent copies of this particular floating-point instruction mix in a fully unrolled loop. The performance levels off at approximately 785 GFLOP/s as we increase the size of the loop. Unfortunately, we cannot completely unroll our loops in FMM_U since we do not have any prior knowledge of the number of neighbors in the U-list, or the number of points in each neighbor. When we put our instruction mix in a loop (i.e., *not* unrolled), the performance drops to approximately 512 GFLOP/s. We believe this performance is the true peak that we should expect from the ideal FMM_U kernel. In fact, the best performance observed among our 390 kernels was 467 GFLOP/s, or 91% of this refined peak.

3.4.5 Constant power

The constant power and energy terms exhibit a large impact on energy consumption. When we ran our microbenchmark in compute-only mode, it spent 55% of its energy on flops on the GTX 580 and 69% on the GTX 680. The flop energy for FMM_U, as a result, was even worse. The FMM_U spent just 23% to 50% of its energy on flops when running on the GTX 580, and 40% to 70.27% on the GTX 680. These relatively low fractions suggest that significant energy savings are still possible.

Additionally, the wide range of energy for flops observed for the FMM_U stress the importance of algorithmic and software–level techniques. Otherwise, we could end up *wasting* as much as 80% of the total energy judging by the 23% expenditure of energy for flops observed on the GTX 580. Beyond algorithms and software, for FMM_U specifically there is also a strong case for even more specialized function units.

3.5 Algorithmic Trade–offs

With the background of § 3.1, we may now ask what the consequences for algorithm design may be. One interesting case is that of a family of algorithms that exhibit a *work–communication trade–off.* Such a trade–off is one in which we can reduce memory communication at the cost of increasing computation. Below, we state when a work–communication trade–off improves time or energy, or both or neither.

The most interesting scenario will be when there exists a balance gap. As such, we will in this section assume $\pi_0 = 0$. The same analysis for $\pi_0 > 0$ appears in § 3.5.5

3.5.1 Notation and key definitions

Denote an abstract algorithm that executes W flops and Q mops by the pair, (W, Q). A "new" algorithm $(fW, \frac{Q}{m})$ exhibits a *work–communication trade–off* with respect to the baseline (W, Q) if $f > 1$ and $m > 1$. From § 3.1, the time and energy of this algorithm are

$$T_{f,m} = W\tau_{\text{flop}} \max\left(f, \frac{1}{m}\frac{B_\tau}{I}\right) \tag{13}$$

$$\text{and} \qquad E_{f,m} = W\epsilon_{\text{flop}} \left(f + \frac{1}{m}\frac{B_\epsilon}{I}\right), \tag{14}$$

respectively. The intensity $I \equiv W/Q$ is that of the *baseline* algorithm.

Higher intensity does *not* mean less time. The new algorithm's intensity is fmI by definition. Though this value is higher than that of the baseline, the new algorithm takes less time only if $1 < f < \frac{B_\tau}{I}$, i.e., the baseline was initially memory bound.

Instead, the best way to compare times is to do so directly. As such, our analysis will also use the usual measure of *speedup*, denoted by ΔT and measured relative to the baseline:

$$\Delta T \equiv \frac{T_{1,1}}{T_{f,m}} = \frac{\max\left(1, \frac{B_\tau}{I}\right)}{\max\left(f, \frac{1}{m}\frac{B_\tau}{I}\right)}. \tag{15}$$

The algorithm exhibits a speedup if $\Delta T > 1$ and a slowdown if $\Delta T < 1$.

Similarly, we may ask whether the algorithm uses less energy than the baseline. In the same way that eq. (15) measures time efficiency, we may measure energy efficiency, or "greenup" ΔE, as

$$\Delta E \equiv \frac{E_{1,1}}{E_{f,m}} = \frac{1 + \frac{B_\epsilon}{I}}{f + \frac{1}{m}\frac{B_\epsilon}{I}}. \tag{16}$$

The new algorithm is more energy efficient than the baseline if $\Delta E > 1$.

3.5.2 A general "greenup" condition

Equation (16) implies that a $(fW, \frac{Q}{m})$ algorithm decreases energy relative to the baseline when $\Delta E > 1$, or (after algebraic rearrangement)

$$f < 1 + \frac{m-1}{m}\frac{B_\epsilon}{I}. \tag{17}$$

Equation (17) is a general necessary condition for a work–communication trade–off to reduce energy.

Acccording to eq. (17), a work–communication trade–off may increase intensity but only up to a limit. Reducing communication (increasing m) increases the slack that permits extra work (higher f) to still pay off. But the energy communication penalty imposes a hard upper limit. In particular, even if we can eliminate communication entirely ($m \to \infty$), the amount of extra work is bounded by $f < 1 + \frac{B_\epsilon}{I}$.

To get a feeling for what this might mean, suppose we have a *baseline* computation that is compute bound in time, and furthermore is maximally tuned. In the language of the roofline of fig. 2a, "compute bound in time" means $I \geq B_\tau$, lying at or to the right of the sharp inflection; and "maximally tuned" means minimum time (or, equivalently, maximum performance), which occurs at y-values that are on the roofline itself. Equation (17) says that a new algorithm exhibiting a work-communication trade–off may reduce energy even if it requires increasing flops. However, there is a limit; even if we eliminate communication, we must have $f < 1 + \frac{B_\epsilon}{I} \leq 1 + \frac{B_\epsilon}{B_\tau}$. For the NVIDIA Fermi GPU architecture of table 2, this upper bound is about five, meaning we should not increase the flops by more than a factor of five.[12] The relative gap between B_τ and B_ϵ determines the slack for extra work.

3.5.3 Time and energy relationships

Equation (15) suggests that we consider three cases in relating speedup ΔT and greenup ΔE:

1. Both the baseline algorithm and the new algorithm are memory bound in time.
2. The baseline algorithm is memory bound in time but the new algorithm is compute bound in time.
3. Both the baseline algorithm and the new algorithm are compute bound in time.

(The fourth logical case, that the baseline is compute bound in time while the new algorithm is memory bound in time, is impossible since we only consider $f, m > 1$.) In each of these cases, we will calculate ΔT and then ask what the resulting lower and upper bounds on ΔE may be. From this analysis, we will see when a speedup, a greenup, or both may be possible.

Case 1: Baseline and new algorithms are memory bound in time If the baseline and new algorithms are memory bound in time, then $I < B_\tau$ and $f < \frac{1}{m}\frac{B_\tau}{I}$. We may conclude from eq. (15) that $\Delta T = m > 1$, where the inequality follows from assumption. In this case, we should always expect a speedup.

[12]This factor does *not* depend asymptotically on the input size, n.

We can get a lower bound on ΔE using eq. (16) and upper bounds on f and $\frac{1}{m}$. The only upper bound on f is $\frac{1}{m}\frac{B_\tau}{I}$. The only upper bound on $\frac{1}{m}$ is 1. Substituting these inequalities into eq. (16), we have

$$\Delta E > \frac{1 + \frac{B_\epsilon}{I}}{\frac{B_\tau}{I} + \frac{B_\epsilon}{I}} = \frac{1 + \frac{I}{B_\epsilon}}{1 + \frac{B_\tau}{B_\epsilon}}. \tag{18}$$

Since both the numerator and denominator are greater than one but $I < B_\tau$, this lower-bound will generally be slightly less than 1, meaning a small loss in energy efficiency can occur.

To get an upper bound on ΔE, we need lower bounds on f or $\frac{1}{m}$ or both. By definition, we have $f > 1$. By memory boundedness, we also have $\frac{1}{m} > f\frac{I}{B_\tau}$. Thus,

$$\Delta E < \frac{1 + \frac{B_\epsilon}{I}}{1 + \frac{B_\epsilon}{B_\tau}}. \tag{19}$$

By memory boundedness of the baseline algorithm, $I < B_\tau$ and so the value of the upper bound of eq. (19) will be greater than 1, but limited by the balance gap.

Equations (18) and (19) exhibit a natural symmetry and, furthermore, are independent of m and f.

Case 2: Baseline is memory bound and new algorithm is compute bound
If the baseline is memory bound in time but the new algorithm is compute bound, then $I < B_\tau$, $\frac{1}{m}\frac{B_\tau}{I} < f$, and $\Delta T = \frac{1}{f}\frac{B_\tau}{I}$. The expression for speedup says that we only expect a speedup if the increase in flops is not too large relative to the communication time penalty.

To get a lower bound on greenup using eq. (16), we need upper bounds on f, $\frac{1}{m}$, or both. The only such bounds are $\frac{1}{m} < 1$, by definition, and $\frac{1}{m} < f\frac{I}{B_\tau}$, by compute boundedness of the new algorithm. Which of these is smaller depends on specific values of the various parameters; however, since both must hold we may conservatively assume either. Choosing the latter, we find

$$\Delta E \;>\; \frac{1}{f} \cdot \frac{1 + \frac{B_\epsilon}{I}}{1 + \frac{B_\epsilon}{B_\tau}} = \Delta T \cdot \frac{1 + \frac{I}{B_\epsilon}}{1 + \frac{B_\tau}{B_\epsilon}}. \tag{20}$$

The rightmost factor is less than 1 since $I < B_\tau$, by assumption. Thus, eq. (20) makes it clear that whatever speedup (or slowdown) we achieve by exploiting the work-communication trade–off, the energy efficiency can be slightly lower.

To get an upper bound on ΔE, we need lower bounds on f, $\frac{1}{m}$, or both. For f, we have two: $f > 1$ and $f > \frac{1}{m}\frac{B_\tau}{I}$. The latter is more general, though it will be pessimistic when $m \gg \frac{B_\tau}{I}$. With that caveat, we obtain

$$\Delta E \;<\; m \cdot \frac{1+\frac{I}{B_\epsilon}}{1+\frac{B_\tau}{B_\epsilon}}. \tag{21}$$

Equation (21) emphasizes the critical role that reducing communication plays in increasing energy efficiency.

Case 3: Baseline and new algorithms are compute bound in time If both the baseline and new algorithms are compute bound in time, then we may deduce that $I > B_\tau$ and $\Delta T = \frac{1}{f} < 1$. That is, we should generally expect a slowdown because flops already dominate; increasing flops only increases work without the possibility of saving any time.

We can obtain a lower bound on ΔE by invoking upper bounds on $\frac{1}{m}$ or on f. Our choices are $\frac{1}{m} < 1$, by definition; and $\frac{1}{m} < f\frac{I}{B_\tau}$, which follows from the new algorithm being compute bound. The tightest of these is first; thus,

$$\Delta E \;>\; \frac{1+\frac{B_\epsilon}{I}}{f+\frac{B_\epsilon}{I}} \;=\; \Delta T\frac{1+\frac{B_\epsilon}{I}}{1+\frac{1}{f}\frac{B_\epsilon}{I}} \;>\; \Delta T. \tag{22}$$

Substituting ΔT for $\frac{1}{f}$ leads to the final equality and inequality. Equation (22) says that the greenup will be no worse than the speedup. However, in this case $\Delta T < 1$, which means that a loss in energy efficiency is possible.

We can get an upper bound on ΔE using eq. (16) with a lower bound on f. The two choices are $f > 1$ and $f > \frac{1}{m}\frac{B_\tau}{I}$. Since the baseline was also compute bound in time, meaning $\frac{B_\tau}{I} \leq 1$, the latter lower bound is strictly less than the trivial lower bound. Thus, the tighter upper bound is

$$\Delta E = \frac{1+\frac{B_\epsilon}{I}}{f+\frac{1}{m}\frac{B_\epsilon}{I}} \;<\; \frac{1+\frac{B_\epsilon}{I}}{1+\frac{1}{m}\frac{B_\epsilon}{I}} \;<\; 1+\frac{B_\epsilon}{I}, \tag{23}$$

where the last inequality shows the limit of the new algorithm having no communication ($m \to \infty$). We may conclude that even though in this case we will always slowdown in time, a greenup may be possible, albeit bounded by roughly the energy communication penalty.

3.5.4 Summary of the three cases

We summarize the three cases as follows.

Case 1 Baseline and new algorithms are memory bound in time:

$$I < B_\tau$$

$$1 < \Delta T = m$$

$$\frac{1+\frac{I}{B_\epsilon}}{1+\frac{B_\tau}{B_\epsilon}} < \Delta E < \frac{1+\frac{B_\epsilon}{I}}{1+\frac{B_\epsilon}{B_\tau}}.$$

Case 2 Baseline is memory bound in time while the new algorithm is compute bound:

$$I < B_\tau$$

$$\Delta T = \frac{1}{f}\frac{B_\tau}{I}$$

$$\Delta T \cdot \frac{1+\frac{I}{B_\epsilon}}{1+\frac{B_\tau}{B_\epsilon}} < \Delta E < m \cdot \frac{1+\frac{I}{B_\epsilon}}{1+\frac{B_\tau}{B_\epsilon}}.$$

Case 3 Baseline and the new algorithm are compute bound in time:

$$B_\tau < I$$

$$\frac{1}{f} = \Delta T < 1$$

$$\Delta T \cdot \frac{1+\frac{B_\epsilon}{I}}{1+\frac{1}{f}\frac{B_\epsilon}{I}} < \Delta E < \frac{1+\frac{B_\epsilon}{I}}{1+\frac{1}{m}\frac{B_\epsilon}{I}}.$$

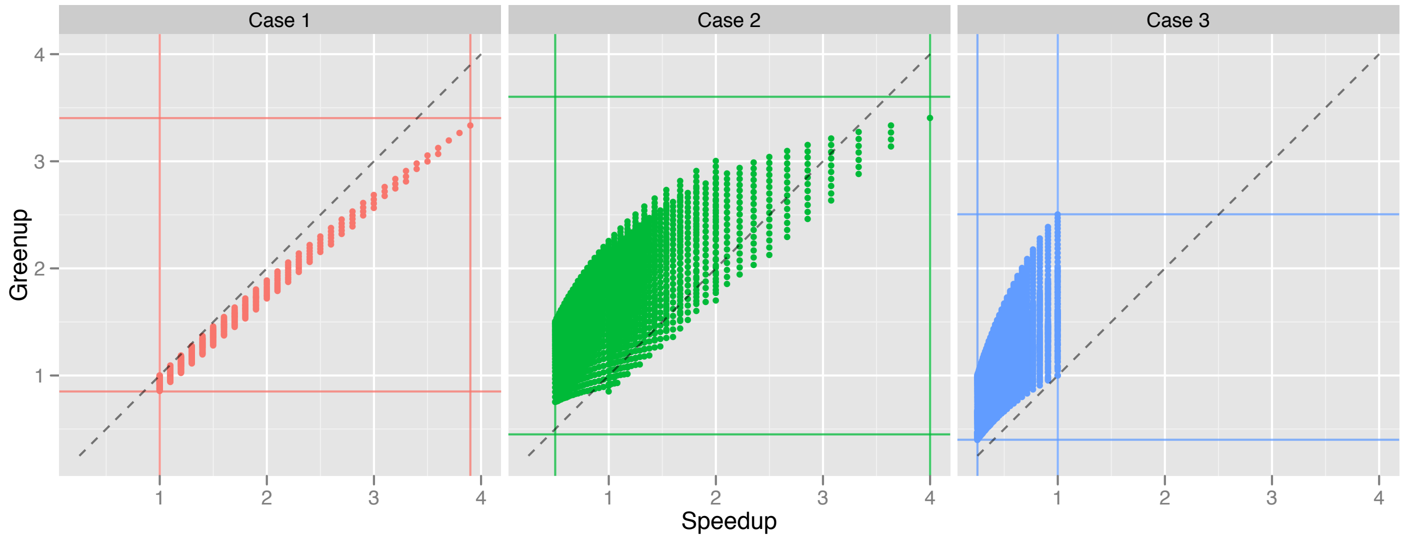


Figure 8: Illustration of the speedup and greenup bounds summarized in § 3.5.4. Points correspond to $(\Delta T, \Delta E)$ pairs at particular values of I, f, and m; horizontal and vertical lines indicate the corresponding minimum lower bounds and maximum upper bounds on speedup and greenup, taken over all values of I, f, and m in each case.

Illustration To illustrate the preceding bounds, consider fig. 8. There, we compute the speedups, greenups, and their lower and upper bounds, for a variety of intensity values $I \in [B_\tau/4, B_\epsilon]$. Figure 8 shows modeled speedups and greenups as points; the minimum and maximum bounds are shown as horizontal and vertical lines. Figure 8 clarifies how it is unlikely that one will improve time ($\Delta T > 1$) while incurring a loss in energy efficiency ($\Delta E < 1$). However, there are many opportunities for the converse, particularly in either Case 2 or Case 3.

3.5.5 Time and Energy Trade-offs under Constant Power

This section generalizes the work-communication trade–off of § 3.5 to the case of $\pi_0 > 0$.

3.5.6 Total energy, $E_{f,m}$

Equation (5) expresses total energy $E(W, I)$ as a function of W and I. Therefore, we may write $E_{f,m}$ as

$$E_{f,m} = E(fW, fmI) \tag{24}$$

$$= fW\hat{\epsilon}_{\text{flop}} \cdot \left(1 + \frac{\hat{B}_\epsilon(fmI)}{fmI}\right) = W\hat{\epsilon}_{\text{flop}} \cdot \left(f + \frac{1}{m}\frac{\hat{B}_\epsilon(fmI)}{I}\right). \tag{25}$$

3.5.7 Effective energy balance, $\hat{B}_\epsilon(I)$

When analyzing work–communication trade–offs, we will make use of the following relationship between $\hat{B}_\epsilon(I)$ and $\hat{B}_\epsilon(fmI)$.

Lemma 3.5.1. *Let $f, m \geq 1$. Then, $\hat{B}_\epsilon(I) \geq \hat{B}_\epsilon(fmI)$.*

Proof. This fact follows simply from the definition of $\hat{B}_\epsilon(I)$.

$$\begin{aligned}\hat{B}_\epsilon(I) &\equiv \eta_{\text{flop}}B_\epsilon + (1-\eta_{\text{flop}})\max(0, B_\tau - I)\\ &\geq \eta_{\text{flop}}B_\epsilon + (1-\eta_{\text{flop}})\max(0, B_\tau - fmI)\\ &= \hat{B}_\epsilon(fmI).\end{aligned}$$

□

3.5.8 Greenup, ΔE

By the definition of greenup, eq. (16),

$$\Delta E = \frac{1 + \frac{\hat{B}_\epsilon(I)}{I}}{f\left(1 + \frac{\hat{B}_\epsilon(fmI)}{fmI}\right)} = \frac{1 + \frac{\hat{B}_\epsilon(I)}{I}}{f + \frac{\hat{B}_\epsilon(fmI)}{mI}}. \tag{26}$$

3.5.9 General necessary condition for greenup

We derive a general necessary condition for an actual greenup, or $\Delta E > 1$. Let I be the intensity of the baseline algorithm. Suppose the new algorithm increases flops by $f > 1$ while reducing memory operations by a factor of $m > 1$. By eq. (26), the condition $\Delta E > 1$ implies

$$f < 1 + \frac{\hat{B}_\epsilon(I)}{I} - \frac{1}{m}\frac{\hat{B}_\epsilon(fmI)}{I} < 1 + \frac{\hat{B}_\epsilon(I)}{I}, \tag{27}$$

where the last inequality follows from lemma 3.5.1. This upper bound on f also holds in the limit that we eliminate communication entirely ($m \to \infty$). From here, there are two interesting cases to consider.

Compute bound limit. Suppose the baseline algorithm is compute bound, so that $I \geq B_\tau$. Then,

$$f < 1 + \frac{\hat{B}_\epsilon(I)}{I} = 1 + \eta_{\text{flop}}\frac{B_\epsilon}{I} \leq 1 + \eta_{\text{flop}}\frac{B_\epsilon}{B_\tau}. \tag{28}$$

In the limit that $\pi_0 \to 0$, $\eta_{\text{flop}} \to 1$ and the bound matches that of § 3.5.2.

Memory-bound limit. Suppose instead that the baseline is memory bound, so that $I < B_\tau$. Then,

$$\begin{aligned} f &< 1 + \frac{\hat{B}_\epsilon(I)}{I} \\ &= 1 + \eta_{\text{flop}}\frac{B_\epsilon}{I} + (1 - \eta_{\text{flop}})\left(\frac{B_\tau}{I} - 1\right) \\ &= \eta_{\text{flop}} + \frac{\eta_{\text{flop}}B_\epsilon + (1 - \eta_{\text{flop}})B_\tau}{I}. \end{aligned} \tag{29}$$

That is, the upper limit on the extra flop factor f is related to a weighted average of the time- and energy-balance constants.

3.5.10 ΔE bounds for case 1: New algorithm is memory bound in time

Lower Bound Suppose $fmI < B_\tau$ and $\frac{1}{m} < 1$. Then,

$$
\begin{aligned}
\Delta E &= \frac{1 + \frac{\hat{B}_\epsilon(I)}{I}}{f + \frac{1}{m}\frac{\hat{B}_\epsilon(fmI)}{I}} \\
&= \frac{1 + \frac{\eta_{\text{flop}} B_\epsilon + \left(1-\eta_{\text{flop}}\right)\cdot \max(0, B_\tau - I)}{I}}{f + \frac{1}{m}\frac{\eta_{\text{flop}} B_\epsilon + \left(1-\eta_{\text{flop}}\right)\cdot \max(0, B_\tau - fmI)}{I}} \\
&= \frac{1 + \frac{\eta_{\text{flop}} B_\epsilon + \left(1-\eta_{\text{flop}}\right)\cdot (B_\tau - I)}{I}}{f + \frac{1}{m}\frac{\eta_{\text{flop}} B_\epsilon + \left(1-\eta_{\text{flop}}\right)\cdot (B_\tau - fmI)}{I}}. \qquad (30)
\end{aligned}
$$

Applying $f < \frac{1}{m}\frac{B_\tau}{I}$ and $\frac{1}{m} < 1$ yields

$$
\begin{aligned}
\Delta E &> \frac{1 + \frac{\eta_{\text{flop}} B_\epsilon + \left(1-\eta_{\text{flop}}\right)\cdot (B_\tau - I)}{I}}{\frac{B_\tau}{I} + \frac{\eta_{\text{flop}} B_\epsilon + \left(1-\eta_{\text{flop}}\right)\cdot (B_\tau - B_\tau)}{I}} \\
&= \frac{\frac{I + \eta_{\text{flop}} B_\epsilon + \left(1-\eta_{\text{flop}}\right)\cdot (B_\tau - I)}{I}}{\frac{B_\tau + \eta_{\text{flop}} B_\epsilon}{I}} \\
&= \frac{I + \eta_{\text{flop}} B_\epsilon + (1-\eta_{\text{flop}})\cdot (B_\tau - I)}{B_\tau + \eta_{\text{flop}} B_\epsilon} \\
&= \frac{B_\tau \cdot (1-\eta_{\text{flop}}) + \eta_{\text{flop}} \cdot (B_\epsilon + I)}{B_\tau + \eta_{\text{flop}} B_\epsilon}. \qquad (31)
\end{aligned}
$$

When $\pi_0 = 0$, we recover eq. (18). Otherwise, whether there is a greenup depends on η_{flop}, since $B_\tau \cdot (1-\eta_{\text{flop}}) < B_\tau$ but $\eta_{\text{flop}} \cdot (B_\epsilon + I) > \eta_{\text{flop}} B_\epsilon$.

Upper Bound To derive an upper bound on ΔE, consider the conditions $1 < m < \frac{B_\tau}{fI}$ or $f\frac{I}{B_\tau} < \frac{1}{m} < 1$ and $f > 1$. From these,

$$
\Delta E = \frac{1 + \frac{\eta_{\text{flop}} B_\epsilon + \left(1-\eta_{\text{flop}}\right)\cdot (B_\tau - I)}{I}}{f + \frac{1}{m}\frac{\eta_{\text{flop}} B_\epsilon + \left(1-\eta_{\text{flop}}\right)\cdot (B_\tau - fmI)}{I}}.
$$

Applying the lower bounds on $f, \frac{1}{m}, m$, we get

$$
\begin{aligned}
\Delta E &< \frac{1 + \frac{\eta_{\text{flop}} B_\epsilon + \left(1-\eta_{\text{flop}}\right)\cdot (B_\tau - I)}{I}}{1 + \frac{I}{B_\tau}\frac{\eta_{\text{flop}} B_\epsilon + \left(1-\eta_{\text{flop}}\right)\cdot (B_\tau - I)}{I}}, \\
&= \frac{1 + \frac{\eta_{\text{flop}} B_\epsilon + \left(1-\eta_{\text{flop}}\right)\cdot (B_\tau - I)}{I}}{1 + \frac{\eta_{\text{flop}} B_\epsilon + \left(1-\eta_{\text{flop}}\right)\cdot (B_\tau - I)}{B_\tau}}.
\end{aligned}
$$

Again, $\pi_0 = 0$ recovers eq. (19). Since $B_\tau > I$, the right hand side will always be greater than 1, meaning that a greenup is always possible. However, the balance gap and the intensity of the original algorithm will limit it.

3.5.11 ΔE bounds for case 2: baseline is memory bound but new algorithm is compute bound in time

Lower Bound To derive a lower bound, consider the conditions $I < B_\tau$, $fmI > B_\tau$, and $\Delta T = \frac{1}{f}\frac{B_\tau}{I}$. From these,

$$\begin{aligned}
\Delta E &= \frac{1 + \frac{\hat{B}_\epsilon(I)}{I}}{f + \frac{1}{m}\frac{\hat{B}_\epsilon(fmI)}{I}} \\
&= \frac{1 + \frac{\eta_{\text{flop}} B_\epsilon + \left(1-\eta_{\text{flop}}\right)\cdot\max(0, B_\tau - I)}{I}}{f + \frac{1}{m}\frac{\eta_{\text{flop}} B_\epsilon + \left(1-\eta_{\text{flop}}\right)\cdot\max(0, B_\tau - fmI)}{I}} \\
&= \frac{1 + \frac{\eta_{\text{flop}} B_\epsilon + \left(1-\eta_{\text{flop}}\right)\cdot(B_\tau - I)}{I}}{f + \frac{1}{m}\frac{\eta_{\text{flop}} B_\epsilon + \left(1-\eta_{\text{flop}}\right)\cdot(0)}{I}} \\
&= \frac{1 + \frac{\eta_{\text{flop}} B_\epsilon + \left(1-\eta_{\text{flop}}\right)\cdot(B_\tau - I)}{I}}{f + \frac{1}{m}\frac{\eta_{\text{flop}} B_\epsilon}{I}}.
\end{aligned}$$

Applying the upper bound $\frac{1}{m} < f\frac{I}{B_\tau}$ yields,

$$\begin{aligned}
\Delta E &> \frac{1 + \frac{\eta_{\text{flop}} B_\epsilon + \left(1-\eta_{\text{flop}}\right)\cdot(B_\tau - I)}{I}}{f + f\frac{I}{B_\tau}\frac{\eta_{\text{flop}} B_\epsilon}{I}} \\
&= \frac{1 + \frac{\eta_{\text{flop}} B_\epsilon + \left(1-\eta_{\text{flop}}\right)\cdot(B_\tau - I)}{I}}{f + f\frac{\eta_{\text{flop}} B_\epsilon}{B_\tau}} \\
&= \frac{1}{f}\cdot\frac{1 + \frac{\eta_{\text{flop}} B_\epsilon + \left(1-\eta_{\text{flop}}\right)\cdot(B_\tau - I)}{I}}{1 + \frac{\eta_{\text{flop}} B_\epsilon}{B_\tau}} \\
&= \Delta T \cdot \frac{I + \eta_{\text{flop}} B_\epsilon + (1-\eta_{\text{flop}})\cdot(B_\tau - I)}{I}\frac{B_\tau}{B_\tau + \eta_{\text{flop}} B_\epsilon}\frac{I}{B_\tau} \\
&= \Delta T \cdot \frac{I + \eta_{\text{flop}} B_\epsilon + (1-\eta_{\text{flop}})\cdot(B_\tau - I)}{B_\tau + \eta_{\text{flop}} B_\epsilon} \\
&= \Delta T \cdot \frac{\eta_{\text{flop}} B_\epsilon + B_\tau - \eta_{\text{flop}} B_\tau + \eta_{\text{flop}} I}{B_\tau + \eta_{\text{flop}} B_\epsilon} \\
&= \Delta T \cdot \frac{B_\tau\left(1-\eta_{\text{flop}}\right) + \eta_{\text{flop}}\left(B_\epsilon + I\right)}{B_\tau + \eta_{\text{flop}} B_\epsilon}.
\end{aligned}$$

If $\pi_0 = 0$, this equation yields eq. (20).

Since $B_\tau(1-\eta_{\text{flop}}) < B_\tau$ and $\eta_{\text{flop}}(B_\epsilon + I) > \eta_{\text{flop}} B_\epsilon$, we cannot determine if this lower-bound on ΔE will be greater or less than the speedup.

Upper Bound To derive an upper bound, consider the conditions $I < B_\tau$ and $fmI > B_\tau$. From these,

$$\Delta E = \frac{1 + \frac{\eta_{\text{flop}} B_\epsilon + \left(1-\eta_{\text{flop}}\right)\cdot(B_\tau - I)}{I}}{f + \frac{1}{m}\frac{\eta_{\text{flop}} B_\epsilon}{I}}.$$

Applying the lower bound $f > \frac{1}{m}\frac{B_\tau}{I}$,

$$\begin{aligned}
\Delta E &< \frac{1 + \frac{\eta_{\text{flop}} B_\epsilon + \left(1-\eta_{\text{flop}}\right)\cdot(B_\tau - I)}{I}}{\frac{1}{m}\frac{B_\tau}{I} + \frac{1}{m}\frac{\eta_{\text{flop}} B_\epsilon}{I}}, \\
&= m \cdot \frac{1 + \frac{\eta_{\text{flop}} B_\epsilon + \left(1-\eta_{\text{flop}}\right)\cdot(B_\tau - I)}{I}}{\frac{B_\tau}{I} + \frac{\eta_{\text{flop}} B_\epsilon}{I}}, \\
&= m \cdot \frac{\frac{I + \eta_{\text{flop}} B_\epsilon + \left(1-\eta_{\text{flop}}\right)\cdot(B_\tau - I)}{I}}{\frac{B_\tau + \eta_{\text{flop}} B_\epsilon}{I}}, \\
&= m \cdot \frac{I + \eta_{\text{flop}} B_\epsilon + (1-\eta_{\text{flop}})\cdot(B_\tau - I)}{B_\tau + \eta_{\text{flop}} B_\epsilon}, \\
&= m \cdot \frac{B_\tau - \eta_{\text{flop}} B_\tau + \eta_{\text{flop}} B_\epsilon + \eta_{\text{flop}} I}{B_\tau + \eta_{\text{flop}} B_\epsilon}, \\
&= m \cdot \frac{B_\tau(1-\eta_{\text{flop}}) + \eta_{\text{flop}}(B_\epsilon + I)}{B_\tau + \eta_{\text{flop}} B_\epsilon}.
\end{aligned}$$

When $\pi_0 = 0$, this inequality recovers eq. (21). As with the lower bound, the equation cannot tell us if the right hand side will be greater than or less than m, but does stress its importance.

3.5.12 ΔE bounds for case 3: baseline is compute bound in time

Lower Bound To derive a lower bound on ΔE, consider the conditions $I > B_\tau$, $fmI > B_\tau$, $\frac{1}{m} < 1$, and $\Delta T = \frac{1}{f}$. From these,

$$\begin{aligned}\Delta E &= \frac{1 + \frac{\hat{B}_\epsilon(I)}{I}}{f + \frac{1}{m}\frac{\hat{B}_\epsilon(fmI)}{I}} \\ &= \frac{1 + \frac{\eta_{\text{flop}} B_\epsilon + \left(1-\eta_{\text{flop}}\right)\cdot 0}{I}}{f + \frac{1}{m}\frac{\eta_{\text{flop}} B_\epsilon + \left(1-\eta_{\text{flop}}\right)\cdot 0}{I}} \\ &= \frac{1 + \frac{\eta_{\text{flop}} B_\epsilon}{I}}{f + \frac{1}{m}\frac{\eta_{\text{flop}} B_\epsilon}{I}}.\end{aligned}$$

Applying the tighter upper bound of $\frac{1}{m} < 1$ (rather than $fmI > B_\tau$) yields,

$$\begin{aligned}\Delta E &> \frac{1 + \frac{\eta_{\text{flop}} B_\epsilon}{I}}{f + \frac{\eta_{\text{flop}} B_\epsilon}{I}} \\ &= \frac{1}{f} \cdot \frac{1 + \frac{\eta_{\text{flop}} B_\epsilon}{I}}{1 + \frac{1}{f}\frac{\eta_{\text{flop}} B_\epsilon}{I}} \\ &= \Delta T \cdot \frac{1 + \frac{\eta_{\text{flop}} B_\epsilon}{I}}{1 + \frac{1}{f}\frac{\eta_{\text{flop}} B_\epsilon}{I}} \\ &> \Delta T.\end{aligned}$$

In this case, greenup will be at least as good as the speedup, at least in theory. In reality, performance will tend to decrease so that a loss in energy efficiency is also likely.

Upper Bound To derive an upper bound on ΔE, consider the conditions $I > B_\tau$, $fmI > B_\tau$, $\frac{1}{m} < 1$, and $\Delta T = \frac{1}{f}$. From these,

$$\Delta E = \frac{1 + \frac{\eta_{\text{flop}} B_\epsilon}{I}}{f + \frac{1}{m}\frac{\eta_{\text{flop}} B_\epsilon}{I}}.$$

Using the trivial lower-bound $f > 1$ (rather than $f > \frac{1}{m}\frac{B_\tau}{I}$),

$$\Delta E < \frac{1 + \frac{\eta_{\text{flop}} B_\epsilon}{I}}{1 + \frac{1}{m}\frac{\eta_{\text{flop}} B_\epsilon}{I}},$$

In the limit of eliminating communication, $(m \to \infty)$,

$$\Delta E < \frac{1 + \frac{\eta_{\text{flop}} B_\epsilon}{I}}{1 + \frac{1}{m}\frac{\eta_{\text{flop}} B_\epsilon}{I}} < 1 + \frac{\eta_{\text{flop}} B_\epsilon}{I}.$$

Recall that the condition $\Delta T = \frac{1}{f}$ implies there will always be a slowdown in time. However, eq. (32) implies that a greenup may nevertheless be possible. The upper limit on any such greenup will be roughly the energy communication penalty.

3.5.13 Summary

Relative to § 3.5.2, the fact of constant power $(\pi_0 > 0)$ removes any guarantee on whether we will see a greenup. However, greenups are nevertheless possible, albeit at a reduced amount related to the efficiency factor η_{flop}. It then becomes critical to understand the architectural and hardware trends most likely to affect π_0 (and therefore also η_{flop}).

3.6 Conclusion

In our view, the most interesting outcome of our analysis is the balance gap, or the difference between the classical notion of time–balance, B_τ, and its energy analogue, B_ϵ. We believe balance gaps have important consequences for algorithms as we have shown in § 3.5. Today, $B_\tau > B_\epsilon$, due largely to idle power and other microarchitectural inefficiencies; consequently, race-to-halt strategies will be the most reasonable first-order technique to save energy. Will this conclusion change significantly in the future?

Our study reinforces three relevant trends that suggest the balance gap will shift. First, we observed that the newer Kepler-class GPU, based on a 28 nm process technology, had a significantly lower constant power compared to the 40 nm Fermi-class GPU: 66 W for the Kepler-based GTX 680 vs. 122 W for the Fermi-based GTX 580. Secondly, GPUs in general exemplify simpler core microarchitecture design when compared to general-purpose CPU platforms. The balance gap did not appear possible on the CPU system in our study, even with no constant power; however, it did emerge on the GPU platforms in a hypothetical $\pi_0 = 0$ scenario. Thirdly, although the cost ratio between reading data from memory and computing on data is expected to remain constant [73], wire capacitance will not scale with time. Consequently, the cost of moving data will stay the same. Unless the distance between off-chip memory and the processing cores decreases significantly (e.g., via die stacking), the balance gap will increase.

Limitations Our model is just a first cut at bridging algorithm and architecture analysis. Regarding its limitations, these are the most important in our view.

First, we have suppressed latency costs, under the assumption of sufficient concurrency; we have done so in prior work [32] and plan to extend it for energy.

Second, we consider the best-case scenario of flops-centric (and on GPUs, FMA-centric) computation. In order for our model to estimate energy consumption more accurately, a more nuanced accounting of the instruction mix is necessary.

Third, we ignored power caps, which can cause our analysis to overestimate power consumption and performance (§ 3.4). Having said that, at least the predictions appear empirically to give upper bounds on power and lower bounds on time.

In spite of these limitations, we hope algorithm designers, performance tuners, and architects will find our basic model an interesting starting point for identifying potential new directions lying at the intersection of algorithms and architecture.

CHAPTER IV

ALGORITHMIC TIME, ENERGY, AND POWER ON CANDIDATE HPC COMPUTE BUILDING BLOCKS

Contents

In the previous chapter [§ 3], we described the basic model and presented its implications for energy efficiency of algorithms. We also validated the model using a small number of microbenchmarks and test platforms [§ 3.3.1,§ 3.3.2], as well as attempted to accurately predict the energy consumption of a real application [§ 3.4.3]. From our experience on real systems, we learned that many high–performance systems are overprovisioned (i.e., their capability exceeds the TDP [§ 3.4.2]), and that caches and shared memory play an important role in improving energy efficiency, and therefore, must be properly accounted for when trying to predict energy consumption [§ 3.4.3].

In this chapter, we focus on improving our model to account for these shortcomings, and also demonstrate how our model and experimental data can be used to precisely analyze and assess candidate compute–node building block systems across computations.

We start by giving a simple demonstration in § 4.1 to give the readers a taste of what our model can accomplish. In § 4.2, we incorporate power cap into our model, and in § 4.3, we describe our extended experimental setup and microbenchmark suite. In § 4.4 we discuss our model fitted parameters, and consider example scenarios, such as power throttling and power bonding. We give our conclusion in § 4.5.

4.1 A Demonstration

As a quick preview, suppose we wish to know whether overall time, energy, and power to compute would be better if the system building block is a high-end desktop GPU or a low-end low-power mobile GPU. Specifically, consider the desktop-class NVIDIA GTX Titan against the "Arndale GPU," the on-chip GPU component of the Samsung Exynos 5 mobile processor. The GTX Titan has a peak performance of 5 trillion floating-point operations per second (5 Tflop/s) in single precision and 250 Watts TDP for the whole card; the Arndale GPU has a 72 Gflop/s peak and its standard developer board uses less than 10 Watts. These specifications suggest GTX Titan is better, based on its considerably higher flop/s per Watt. Yet, there are also active efforts to build systems from close equivalents to the latter.[1] Which is "correct?"

While a natural response is, "it depends," our model offers a more precise analysis, which fig. 9 summarizes. It compares time efficiency (performance, or operations per unit time), energy efficiency (operations per unit energy), and power (energy per unit time) of the two platforms. The y-axis measures this performance on a normalized scale. (The power and energy costs include the entire board, including memory and on-board peripherals, but *excluding* any host system.) The x-axis abstracts away

[1]E.g., The Mont Blanc Project: `http://www.montblanc-project.eu/` [108].

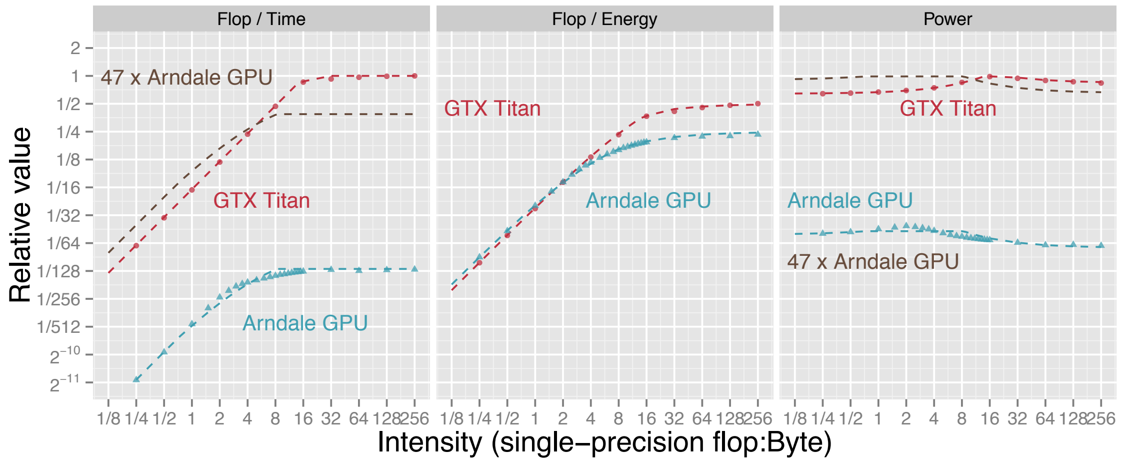


Figure 9: Comparison of the time efficiency (performance), energy efficiency, and power required by a mobile GPU (from an "Arndale" Samsung Exynos 5 developer board) versus high-end gaming-grade desktop GPU (NVIDIA GTX Titan), over a range of synthetic computations with varying computational intensities (flop:Byte). Combining 47 of the mobile GPUs to match on peak power can lead to a system that outperforms the desktop GPU by up to 1.6× for relatively bandwidth bound codes (flop:Byte less than 4), but at the cost of sacrificing peak performance (less than ½) for compute–bound codes.

a possible computation by its operational intensity, or the ratio of computation-to-communication (flop:Byte ratio). Decreasing values of intensity indicate increasing memory bandwidth boundedness. The dots are measured values from a synthetic microbenchmark (§ 4.3); the dashed lines indicate our model's predictions. The model and measurements correspond well.

While the GTX Titan is much faster, in energy efficiency the Arndale GPU compares well to it over a range of intensities; the two systems match in flops per Joule (flop/J) for intensities as high as 4 flop:Byte. For reference, a large sparse matrix-vector multiply is roughly 0.25–0.5 flop:Byte in single precision and a large fast Fourier transform (FFT) is 2–4 flop:Byte [126]. And even at more compute bound intensities, the Arndale is within a factor of two of the GTX Titan in energy efficiency despite its much lower peak. It therefore appears to be an attractive candidate.

From these data, what is the best case scenario for an Arndale GPU-based "supercomputer," assembled from enough Arndale GPUs to match the GTX Titan in peak *power*? Matching on power may require up to 47 Arndale GPUs, yielding the hypothetical system shown by a dashed brown line. This system would have less than half of the GTX Titan's peak, but would also have an aggregate memory bandwidth that is up to 1.6× higher for intensities up to about 4 flop:Byte, which could include, for instance, a large multidimensional FFT. However, this best-case ignores the significant costs of an interconnection network, or further potential improvements to the Arndale GPU system by better integration. As such, the 47 Arndale GPUs are more likely to improve upon GTX Titan only marginally or not at all across the full range of intensities. Regardless of one's interpretation, this type of analysis offers an analytical way to compare these as building blocks.

4.2 Modeling Power Cap

As discussed in § 3.1, our model of time, energy, and power assumes the abstract von Neumann architecture of fig. 1. The system comprises a processor attached to a fast memory of finite capacity (Z words), which is then attached to an infinite slow memory. The fast memory is effectively a last-level cache and may be generalized in the presence of a memory hierarchy. An abstract algorithm running on this machine executes $W = W(n)$ flops and transfers $Q = Q(n; Z)$ bytes of data between slow and fast memory, given an input of size n. (In what follows, we suppress the arguments n and Z unless needed explicitly.[2]) Below, we describe how we estimate time, energy, and power for the abstract algorithm running on this abstract machine. The model derives partly from our earlier work [25, 26] from § 3.1; here, we highlight the additions we have made to our model.

[2]If flops are not the natural unit of work, one could imagine substituting, for instance, "comparisons" for sorting or "edges traversed" in a graph traversal computation.

Cost model Let the abstract machine be described by four fundamental time and energy costs: the time per flop, τ_{flop}; the time per byte, τ_{mem}; the energy per flop, ϵ_{flop}; and the energy per byte, ϵ_{mem}. Time and energy have units of, for instance, seconds (s) and Joules (J), respectively. In our abstract model, we do not take τ_{flop} and τ_{mem} to be latency costs; rather, we will use throughput values based on peak flop/s and peak memory bandwidth, respectively. That is, these costs are optimistic.

They also imply *power* costs, in units of energy per unit time. These are the peak power per flop, $\pi_{\text{flop}} \equiv \epsilon_{\text{flop}}/\tau_{\text{flop}}$, and the peak power per byte, $\pi_{\text{mem}} \equiv \epsilon_{\text{mem}}/\tau_{\text{mem}}$.

In addition, our abstract machine will require a minimum amount of *constant power*, π_0. This power is what the machine requires independent of what operations are executing. In contrast to other notions of "static power" in the literature, constant power in our model may *include* the power of other system components and peripherals, taken together.

Modeling execution energy To estimate an algorithm's energy cost, we tally the total energy to execute all flops, to move the full volume of data, and to run given the cost of constant power. That is, the total energy E is

$$E = E(W, Q) \equiv W\epsilon_{\text{flop}} + Q\epsilon_{\text{mem}} + \pi_0 T(W, Q), \tag{32}$$

where $T(W, Q)$ is the total running time of the computation, estimated below. The basic form of eq. (32) is identical to our earlier energy model [25, 26]from § 3.1.

We will sometimes consider another form of eq. (32), parameterized by *intensity*, $I \equiv W/Q$, and *energy balance*, $B_\epsilon \equiv \epsilon_{\text{mem}}/\epsilon_{\text{flop}}$. Both these quantities have units of flops per Byte, with I expressing an intrinsic property of the algorithm and B_ϵ expressing an intrinsic property of the machine with respect to energy. From these definitions, eq. (32) becomes

$$E = E(W, I) = W\epsilon_{\text{flop}}\left(1 + \frac{B_\epsilon}{I}\right) + \pi_0 T\left(W, \frac{W}{I}\right). \tag{33}$$

Equation (33) clarifies that the total energy, relative to the minimum energy $W\epsilon_{\text{flop}}$ to execute the flops alone, increases with increasing energy balance, decreases with increasing intensity, and increases with relative increases in constant power or other time inefficiencies.

Modeling execution time Executing W flops takes $W\tau_{\text{flop}}$ time and moving Q bytes takes $Q\tau_{\text{mem}}$ time. In the best case, we may maximally overlap flops and memory movement, in which case T will be the maximum of these two values. Indeed, this definition was exactly our previous model [26] from § 3.1.

Here, we extend our model of T to include *power caps.* Our previous model sometimes overpredicted performance and average power [26]. For some GPU and low-power systems we consider in this section, which go beyond the original work, ignoring a potential power cap can have severe consequences(§ 4.4).

We model a power cap as follows. Suppose that on top of the π_0 constant power, the system has $\Delta\pi$ additional units of *usable power* to perform any operations. The parameter $\Delta\pi$ is now a new fundamental parameter of the system. Thus, an algorithm limited only by $\Delta\pi$ will require no more than $(W\epsilon_{\text{flop}} + Q\epsilon_{\text{mem}})/\Delta\pi$ time to execute. Then, the best-case execution time is

$$T = T(W,Q) \equiv \max\left(W\tau_{\text{flop}}, Q\tau_{\text{mem}}, \frac{W\epsilon_{\text{flop}} + Q\epsilon_{\text{mem}}}{\Delta\pi}\right). \tag{34}$$

That is, if there is enough usable power to run at peak operational or memory performance, we do so assuming maximal overlap; otherwise, we must throttle all operations, which the third term of max captures. We may also rewrite eq. (34) as,

$$T = T(W,I) = W\tau_{\text{flop}} \max\left\{1, \frac{B_\tau}{I}, \frac{\pi_{\text{flop}}}{\Delta\pi}\left(1 + \frac{B_\epsilon}{I}\right)\right\}, \tag{35}$$

where $B_\tau \equiv \tau_{\text{mem}}/\tau_{\text{flop}}$ is the *time balance* of the system. This value is more commonly referred to as the intrinsic flop-to-Byte ratio of the machine, and defines the intensity at which the time to execute flops and time to execute memory operations are equal.

Modeling power Given models of E and T, we may model average instantaneous power as $P \equiv E/T$.

There are many ways to expand this definition, one of which we present here. Let

$$\begin{aligned} B_\tau^+ &\equiv B_\tau \max\left(1, \frac{\pi_{\text{mem}}}{\Delta\pi - \pi_{\text{flop}}}\right) && (36) \\ \text{and } B_\tau^- &\equiv B_\tau \min\left(1, \frac{\Delta\pi - \pi_{\text{mem}}}{\pi_{\text{flop}}}\right). && (37) \end{aligned}$$

Observe that $B_\tau^- \leq B_\tau \leq B_\tau^+$. When $\Delta\pi \geq \pi_{\text{flop}} + \pi_{\text{mem}}$, there is enough usable power to run flops and move data at their maximum rates, in which case $B_\tau^+ = B_\tau^- = B_\tau$. Otherwise, $[B_\tau^-, B_\tau^+]$ defines an interval containing B_τ. From these definitions, one can show that P becomes

$$P = P(I) = \pi_0 + \begin{cases} \pi_{\text{flop}} + \pi_{\text{mem}} \frac{B_\tau}{I} & \text{if } I \geq B_\tau^+ \\ \pi_{\text{flop}} \frac{I}{B_\tau} + \pi_{\text{mem}} & \text{if } I \leq B_\tau^- \\ \Delta\pi & \text{otherwise} \end{cases} . \qquad (38)$$

Equation (38) reflects what we might expect. As I increases beyond B_τ^+ toward infinity, P decreases toward flop-only power, π_{flop}. Similarly, as I decreases away from B_τ^- toward 0, P decreases toward memory–only power, π_{mem}. The peak power occurs when $B_\tau^- \leq I \leq B_\tau^+$. In particular, when there is enough usable power, meaning $\Delta\pi \geq \pi_{\text{flop}} + \pi_{\text{mem}}$, then $P(I)$ peaks at the value, $\pi_0 + \pi_{\text{flop}} + \pi_{\text{mem}}$, at $I = B_\tau$; otherwise, the power cap dominates and $P(I) = \pi_0 + \Delta\pi$ for all $B_\tau^- \leq I \leq B_\tau^+$.

4.3 Experimental Setup and Microbenchmarks

We benchmarked and assessed the nine systems shown in table 6. For the four discrete coprocessors—NVIDIA GTX 580, 680, Titan, and Intel Phi—we consider just the card itself and ignore host power and host-to-coprocessor transfer costs. Additionally, three of the systems—labelled "NUC[3]," "Arndale," and "APU"—have hybrid CPU+GPU processors. We considered their CPU and GPU components separately, i.e., running

[3]Next Unit of Computing

no or only a minimal load on the other component. As such, we claim to evaluate twelve “platforms.”

For each platform, we wrote an architecture-specific hand–tuned microbenchmark that gets as close to the vendor’s claimed peak as we could manage. These microbenchmarks measure “sustainable peak” flop/s, streaming bandwidth (from main memory, L1, and L2 caches where applicable), and random main memory access. The measured values appear parenthetically in columns 8–10 of table 6, and should be compared against the theoretical peaks shown in columns 3–5. For cache bandwidth, we were not able to determine theoretical peaks on all platforms; therefore, we report only measured values in columns 11 and 12. Lastly, we fitted our model to the microbenchmark data. The model parameters appear as columns 6–13 of table 6.

Intensity microbenchmark The intensity microbenchmark, identical to the one used in § 3.3.2, varies intensity nearly continuously, by varying the number of floating point operations (single or double) on each word of data loaded from main memory. We hand tuned these microbenchmarks for each platform. Examples of specific tuning techniques we used include unrolling, to eliminate non-flop and non-load/store overheads that might otherwise distort our energy estimates; use of fused-multiply adds where available; tuning the instruction selection and instruction mix, carefully considering pipeline and issue port conflicts; prefetching; and resorting to assembly where needed; to name a few. We test single- and double-precision operations separately; their energy costs appear as ϵ_{s} and ϵ_{d}, respectively.

Random access microbenchmark Our random access microbenchmark implements pointer chasing, as might appear in a sparse matrix or other graph computation. It fetches data from random places in the memory rather than streaming the data, reporting sustainable accesses per unit time. By its nature, it cannot fully use the memory interface width or the prefetching units. Therefore, we expect poor

Table 6: Platforms summary, with 9 systems and 12 distinct "platforms." We distinguish between manufacturer's peak (columns 3–5) and "sustained peak" using our microbenchmarks, shown parenthetically in columns 8–10. *Note 1*: In four cases, denoted by an asterisk ("*"), our fitted constant power is less than observed idle power. *Note 2*: Some data are missing; double precision support is not available on all platforms and deficiences in the OpenCL driver prevented some microbenchmarks from running.

Column 1	2	3	4	5	6	7	8	9	10	11	12	13
		Vendor's claimed peak			Power (empirical)		Energy (and empirical throughput)					Random access
Platform	Processor	single Gflop/s	double Gflop/s	mem. bw. GB/s	π_0 Watts (idle)	$\Delta\pi$ Watts	ϵ_s pJ/flop (Gflop/s)	ϵ_d pJ/flop (Gflop/s)	ϵ_{mem} pJ/B (GB/s)	ϵ_{L1} pJ/B (GB/s)	ϵ_{L2} pJ/B (GB/s)	ϵ_{rand} nJ/access (Macc/s)
Desktop CPU "Nehalem"	Intel Core i7-950 (45 nm)	107	53.3	25.6	122 (79.9)	44.2	371 (99.4)	670 (49.7)	795 (19.1)	135 (201)	168 (120)	108 (149)
NUC CPU "Ivy Bridge"	Intel Core i3-3217U (22 nm)	57.6	28.8	25.6	16.5 (13.2)	7.37	14.7 (55.6)	24.3 (27.9)	418 (17.9)	8.75 (201)	14.3 (103)	54.6 (55.3)
NUC GPU	HD 4000	269	—	25.6	10.1 (13.2)*	17.7	76.1 (268)	—	837 (15.4)	—	—	—
APU CPU "Bobcat"	AMD E2-1800 (40 nm)	13.6	5.10	10.7	20.1 (11.8)	1.39	33.5 (13.4)	119 (5.05)	435 (3.32)	84.0 (25.8)	138 (11.6)	75.6 (8.03)
APU GPU "Zacate"	HD 7340	109	—	10.7	15.6 (11.8)	3.23	5.82 (104)	—	333 (8.70)	6.47 (46.0)	—	45.8 (115)
GTX 580 "Fermi"	NVIDIA GF100 (40 nm)	1580	198	192	122 (148)*	146	99.7 (1400)	213 (196)	513 (171)	149 (761)	257 (284)	112 (977)
GTX 680 "Kepler"	NVIDIA GK104 (28 nm)	3530	147	192	66.4 (100)*	145	43.2 (3030)	263 (147)	437 (158)	51 (1150)	195 (297)	184 (1420)
GTX Titan "Kepler"	NVIDIA GK110 (28 nm)	4990	1660	288	123 (72.9)	164	30.4 (4020)	93.9 (1600)	267 (239)	24.4 (1610)	195 (297)	48.0 (968)
Xeon Phi "KNC"	Intel 5110P (22 nm)	2020	1010	320	180 (90)	36.1	6.05 (2020)	12.4 (1010)	136 (181)	2.19 (2890)	8.65 (591)	5.11 (706)
PandaBoard ES "Cortex-A9"	TI OMAP 4460 (45 nm)	9.60	3.60	3.20	3.48 (2.74)	1.19	37.2 (9.47)	302 (3.02)	810 (1.28)	79.5 (18.4)	134 (4.12)	60.9 (12.1)
Arndale CPU "Cortex-A15"	Samsung Exynos 5 (32 nm)	27.2	6.80	12.8	5.50 (1.72)	2.01	107 (15.8)	275 (3.97)	386 (3.94)	76.3 (50.8)	248 (15.2)	138 (14.8)
Arndale GPU "Mali T-604"		72.0	—	12.8	1.28 (1.72)*	4.83	84.2 (33.0)	—	518 (8.39)	71.4 (33.4)	—	125 (33.6)

performance compared to the system's bandwidth.

Cache microbenchmarks Our cache microbenchmarks assess the performance and energy cost of accessing the different levels of the cache. These are similar to the ones described in § 3.3.3, but refined to stress the memory system further; on CPU systems, these can be either the pointer chasing or the intensity benchmark, depending on which gives better performance. We need only ensure the data set size is small enough to fit into the target cache level.

GPUs have different memory hierarchy designs, which requires platform-dependent coding and tuning. On NVIDIA Fermi GPUs, we assess L1, L2 caches; on Kepler systems, we test L2 and shared memory, since the L1 cache is no longer used to store data and all data reuse has to be done manually via the shared memory. On AMD HD 7340 and ARM Mali-T604, we use the software-managed scratchpad memory.

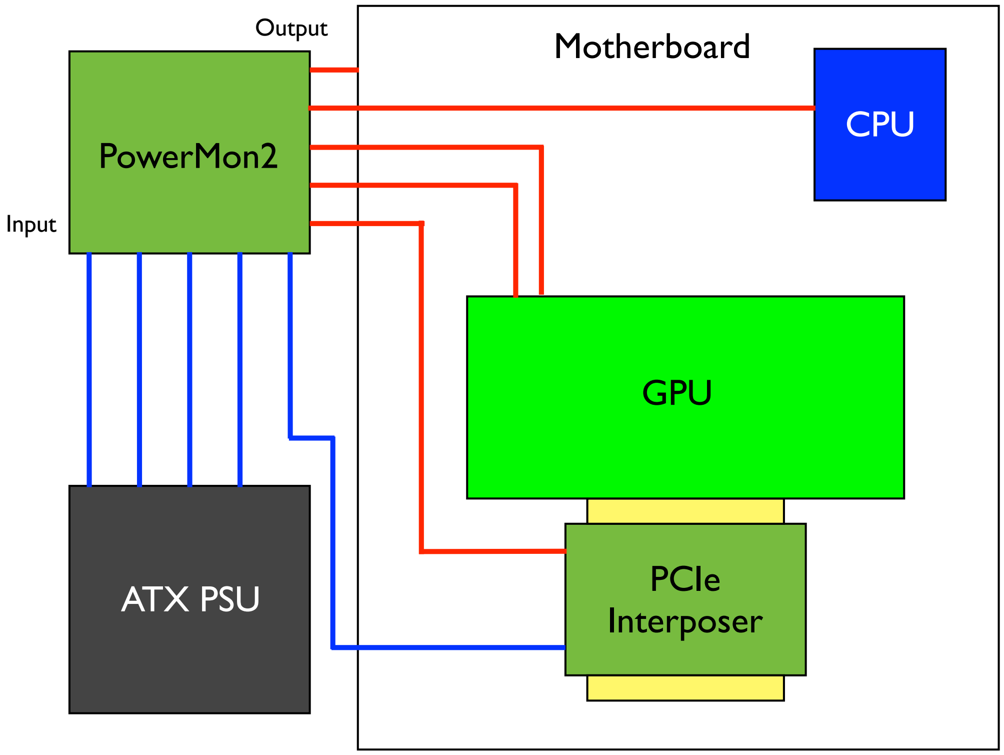


Figure 10: Placement of the measurement probes, PowerMon 2 and our custom PCIe interposer

Power measurement infrastructure Our extended power measurement setup appears in fig. 10. As before, we use two tools to measure power. The first is Powermon 2, a fine-grained integrated power measurement device for measuring direct current (DC) voltage and current [8]. The second measurement tool is a custom-made PCIe interposer that sits between the motherboard and the target PCIe device (i.e., GPU) to measure the power provided by the motherboard. For the mobile systems that we have newly added in this section, we measure system-level power, which includes CPU, GPU, DRAM, and peripherals. Readers may refer to § 3.3.1 for more details on how these tools work.

4.4 Results and Discussion

We divide the analysis and discussion of our experiments into four parts. First, we compare the power-capped model of this paper with our prior model [25, 26], showing both qualitative and quantitative improvements (§ 4.4.1). Second, we explain our memory hierarchy measurement results (§ 4.4.2). Third, we analyze the platforms in detail (§ 4.4.3), to show what one might conclude about their relative time efficiency, energy efficiency, and power characteristics. Lastly, we use the model to consider a variety of "what-if" scenarios, such as what we expect to happen under power throttling or power bounding (§ 4.4.4).

We focus on single-precision results, since full support for double is incomplete on several of our evaluation platforms. However, the interested reader can still find the main summary estimate of double-precision flop energy cost in table 6.

4.4.1 Model fitting and accuracy

For each of the twelve platforms shown in table 6, we ran our microbenchmark suite (§ 4.3) at varying W and Q values, and measured total execution time and energy. These include runs in which the total data accessed only fits in a given level of the memory hierarchy. We then used (nonlinear) regression parameter fitting techniques

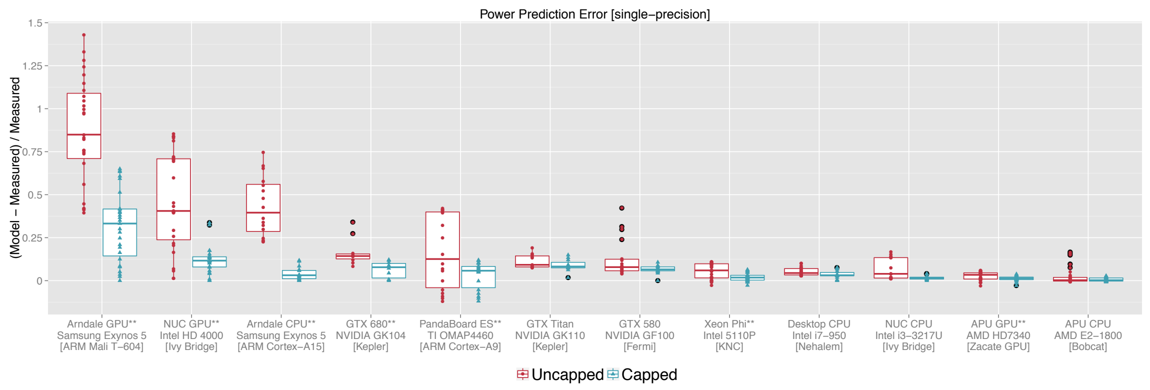


Figure 11: Summary of modeling errors with respect to performance (FLOP/s). We compare the prediction errors of our prior model ("free" or uncapped; [25, 26]; § 3.1) against our new model ("capped"), which notably includes an explicit power cap. Qualitatively, the distribution of errors on all platforms improves, becoming either lower in median value or more tightly grouped. On platforms labeled by double-asterisks ("**"), the free and capped distributions differ statistically (at $p < .05$) by the Kolmogorov-Smirnov test [77].

to obtain statistically significant estimates of the values τ_{flop}, τ_{mem}, ϵ_{flop}, ϵ_{mem}, π_0, and $\Delta\pi$, as well as the corresponding parameters for each cache level where applicable.[4] The resulting set of parameters appears in columns 6–13 of table 6.

To assess the model fits, we evaluated both our prior "uncapped" model [25] and our new "capped" model, eqs. (32)–(38). We compared them against the measured values by first calculating the relative error, (model − measured)/measured, at each intensity value. Given a platform and model, we regard the set of errors over all intensity values as the error distribution. For each platform, we compared our prior and new models by comparing their distributions.

Figure 11 summarizes these error distributions. (We have similar data for time and energy, omitted for space.) Each platform appears on the x-axis, and the y-axis measures relative error. Each observed error value appears as a dot; the boxplots reflect the median, 25%, and 75% quantiles of the distribution of those errors. Platforms are sorted in descending order of median uncapped model's relative error. That

[4]The precise fitting procedure follows our prior work [25] and is part of our publicly released source code: `http://hpcgarage.org/archline`

is, our original model does relatively better as we move from platforms on the left toward platforms on the right of fig. 11.

Qualitatively, fig. 11 shows that the new capped model tends to reduce the magnitude and spread of relative error compared to the previous model. The bias is to overpredict, i.e., most errors greater than zero. To facilitate a more rigorous quantitative comparison, we also performed the Kolmogorov-Smirnov non-parametric test of whether two empirical distributions differ, against a null hypothesis that the distributions come from the same underlying distribution. (The K-S test makes no assumptions about the distributions, such as normality, and so may be pessimistic.) Any platform for which the null hypothesis may be rejected (at a p-value less than 0.05) is marked with two asterisks ("**") in fig. 11. In this statistical sense, seven of the 12 platforms–Arndale GPU, NUC GPU, Arndale CPU, GTX 680, PandaBoard ES, Xeon Phi, and APU GPU—the uncapped and capped error samples likely come from different distributions.

4.4.2 Interpreting memory hierarchy energy costs

Some care is needed to correctly interpret the memory hierarchy parameter estimates of table 6. The key principle is that our energy cost estimates reflect *inclusive* costs. Alternatively, one should regard the energy cost of a memory hierarchy operation in our model as the *additional* energy required to complete *one additional* instance of that operation. The following examples clarify how this interpretation works.

The cost of loading a byte from DRAM (ϵ_{mem}) includes not only the costs of reading the byte from the DRAM cells and driving the wires, but also the energy spent by the memory controller as well as the cost of going through the memory hierarchy (e.g., the L1 and L2 caches). The rationale is that these costs are unavoidable whenever data moves between DRAM and registers. Note that we currently do not differentiate reads and writes, so consider ϵ_{mem} as the average of these costs. Also, in order to prevent the prefetcher from loading unused data, we have designed our microbenchmark to "direct" the prefetcher into prefetching only the data that will be used.

We define ϵ_{L1}, ϵ_{L2}, and ϵ_{rand} in a similar manner as ϵ_{mem}. The energy cost of loading data from the L2 cache (ϵ_{L2}) includes the energy consumed by reading data from the L2 memory cells as well as those consumed by reading from and writing data to the L1 cache as the data moves up the cache hierarchy. It also includes instruction overheads, as well as any other costs that might be involved such as the energy consumed by the cache coherency protocol. Naturally, we expect ϵ_{L1} to be smaller than ϵ_{L2} on the same system as they would most likely incur similar overheads, but fetching data from the L1 cache will not involve going through the L1 cache itself. This also serves as a way of sanity checking our model and as it can be seen in table 6, $\epsilon_{\mathrm{L1}} \leq \epsilon_{\mathrm{L2}}$ for every system.

The energy cost of accessing memory at random locations (ϵ_{rand}) will include the cost of reading an entire cache line from the memory, as well as the usual overheads such as instruction, memory hierarchy, and associated protocols. As such, we expect this cost to be at least an order of magnitude higher than ϵ_{mem}, as table 6 reflects.

4.4.3 Constant power and power caps across platforms

There is a wide range of power behaviors across platforms, but also a narrow relative range within each platform. Refer to fig. 12, which shows the power of the twelve platforms from table 6. It compares model (solid lines) to measurements (dots), and facilitates cross-platform comparisons across a full range of intensities (x-axis). The platforms are ordered from top-left to bottom right in decreasing order of *peak energy efficiency*, with the GTX Titan in the top left at 16 Gflop/J and the Desktop CPU (Nehalem) at the bottom right at just 620 Mflop/J.

Across platforms, power allocation between memory and processing differs. Flops on the GTX Titan consume more of the power budget than memory operations, while

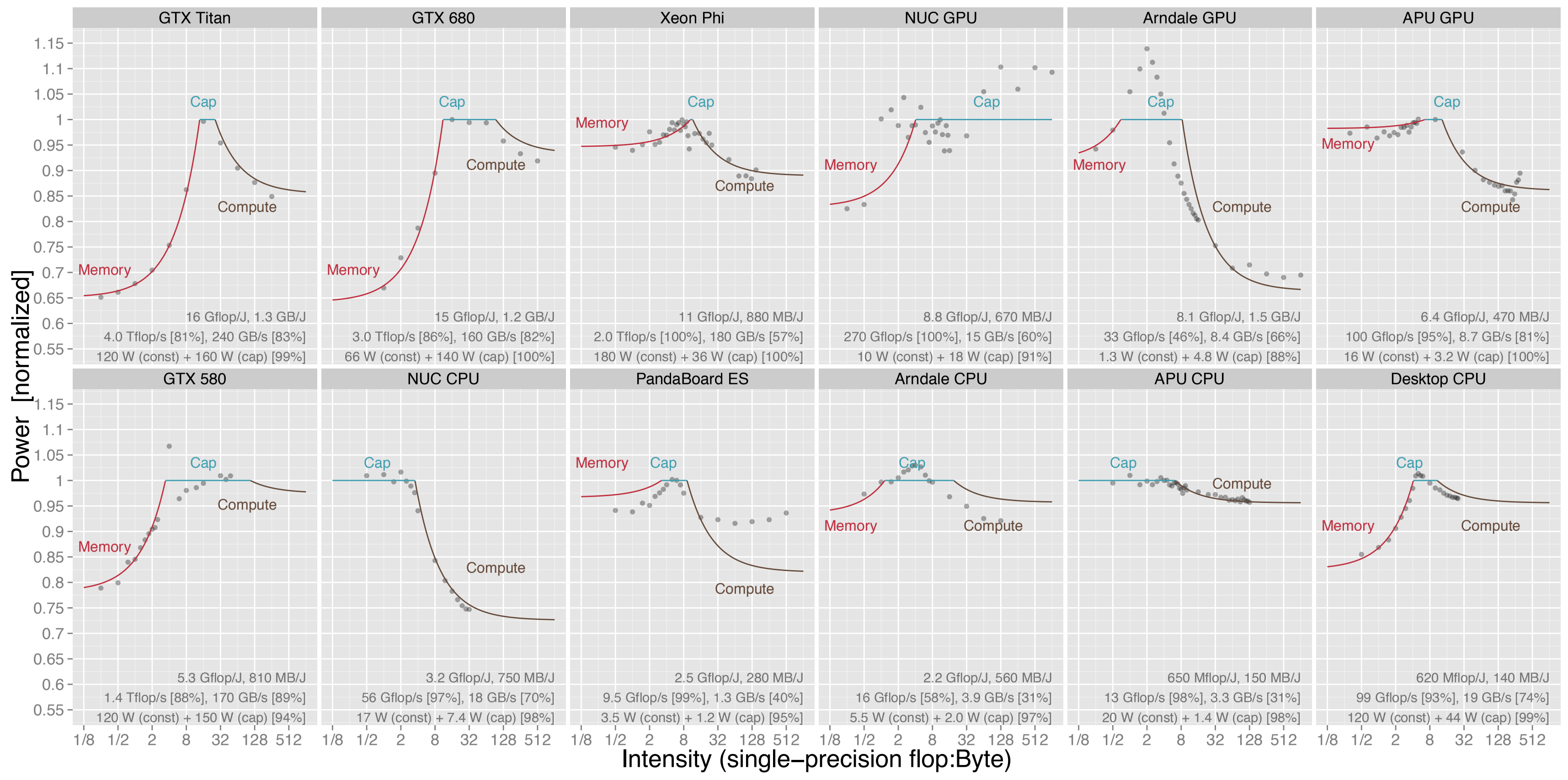


Figure 12: Power (normalized). The y-axis is linear but the x-axis is logarithmic, base 2. The model appears as solid lines in up to three segments, indicating the three possible regimes: memory bandwidth bound, power bound due to capping, and compute bound. Measurements appear as solid grey dots. Platforms appear from left–to–right and top–to–bottom in *decreasing* order of *peak energy efficiency*, i.e., the most energy efficient platform is the GTX Titan, whose peak is 16 Gflop/J.

the Arndale GPU exhibits just the opposite behavior. However, there are no strongly discernable patterns in the architectural power allocations as energy efficiency varies. The Arndale GPU, for example, has a peak energy efficiency within a factor of two of the GTX Titan (8.1 Gflop/J vs. 16 Gflop/J), while putting much more of its power into the memory system. Consequently, its design yields on peak flop energy efficiency to boost memory energy efficiency, e.g., 1.5 Gflop/J on the Arndale GPU vs. 1.3 Gflop/J on GTX Titan.

But within a platform, the measurements vary only between the range of 0.65 to 1.15, or less than 2×. A narrow range means there is little room to reconfigure power to improve or adapt energy efficiency to a computation. The main obstacle is the relatively large value of constant power, π_0, shown in table 6. Indeed, the fraction of maximum power that π_0 consumes, or $\pi_0/(\pi_0 + \Delta\pi)$, is more than 50% for seven of the 12 platforms in table 6. As it happens, this fraction correlates with overall peak energy efficiency, with a correlation coefficient of about -0.6 (not shown). Thus, driving down π_0 would be the key factor for improving overall system power reconfigurability.

Indeed, the mere fact of π_0 can invert our expectations. Consider a hypothetical workload that simply streams data from memory. How much energy does this computation use? Referring to table 6, the Xeon Phi has the lowest ϵ_{mem} (136 pJ/B). It is lower than, for instance, the higher bandwidth GTX Titan (267 pJ/B) and lower-power Arndale GPU (518 pJ/B). However, we must also pay a constant energy charge of $\tau_{\text{mem}} \times \pi_0$, which adds 994 pJ/B to Xeon Phi, 515 pJ/B to GTX Titan, and just 153 pJ/B to the Arndale GPU. Then, the total energy per byte is 671 pJ/B for the Arndale GPU, 782 pJ/B for the GTX Titan, and 1.13 nJ/B to the Xeon Phi. This example underscores the critical role that π_0 plays.

The extent to which power capping, as we model it, affects the power characteristics of a given platform varies widely. Caps apply over a wider range of intensities on the hybrid NUC CPU+GPU and AMD APU platforms than they do on the GTX Titan and Xeon Phi. However, our approach to capping appears inaccurate on the NUC GPU and Arndale GPU. Although these mispredictions are always less than 15%, they raise questions about what mechanisms are operating. On the NUC GPU, as it happens measurement variability owes to OS interference.[5] However, on the Arndale GPU, the mismatch at mid-range intensities suggests we would need a different model of capping, perhaps one that does *not* assume constant time and energy costs per operation. That is, even with a fixed clock frequency, there may be active energy–efficiency scaling with respect to processor and memory utilization.

4.4.4 Power throttling scenarios

Using the model, we may consider a variety of "what-if" scenarios related to power.

Power throttling Suppose we lower $\Delta\pi$. What could the impact on maximum system power, performance, and energy efficiency be, assuming all other model parameters, including π_0, remain equal?

Figure 13 shows power when the power cap is set to $\Delta\pi/k$, where $\Delta\pi$ is the original power cap (see table 6) and $k \in [1, 8]$ is the reduction factor.

First, consider the extent to which reducing $\Delta\pi$ reduces overall system power. Figure 13 confirms that, owing to constant power $\pi_0 > 0$, reducing $\Delta\pi$ by k reduces overall power by less than k. It further shows that the Arndale GPU has the most potential to reduce system power by reducing $\Delta\pi$, whereas the Xeon Phi, APU CPU, and APU GPU platforms have the least. More node-level headroom—that is, low π_0 compared to $\Delta\pi$—may be very important, since it leaves more relative power for other power overheads, including the network and cooling.

Next, consider the extent to which reducing $\Delta\pi$ reduces performance. Figure 14a

[5]OpenCL drivers for NUC are available only when running Windows, which lacks easy user-level power management support.

shows the variability across platforms and computational intensities. Highly memory-bound, low intensity computations on the GTX Titan degrade the least as $\Delta\pi$ decreases, since its design overprovisions power for compute. By contrast, for highly compute–bound computations, the NUC CPU degrades the least, since its design overprovisions power for memory. A similar observation holds for energy efficiency, fig. 14b.

Power bounding The preceding scenarios may have further implications for the idea of dynamic *power bounding*, which Rountree et al. have suggested will be a required mechanism of future systems [110].

Recall fig. 9 from § 4.1, which compared GTX Titan and Arndale GPU building blocks. That analysis suggested that, as configured, an Arndale GPU building block would, even in the best case, offer only marginal improvements over GTX Titan, and would likely be worse.

However, in a power bounding scenario, it might be necessary to reduce node power to a particular level. Suppose that, in a system based on GTX Titan nodes, it is necessary to reduce per-node power by half, to 140 Watts per node. This corresponds to a power cap setting of $\Delta\pi/8$ in fig. 13, which in turn will imply a performance of approximately $0.31\times$ at $I = 0.25$ relative to the default $\Delta\pi$. One can determine that, in the best case, assembling 23 Arndale GPUs will match 140 Watts but will be approximately $2.8\times$ faster at $I = 0.25$, which is better than the $1.6\times$ scenario from fig. 9. Essentially, a lower power grainsize, *combined* with a compute building block having a lower π_0, may lead to more graceful degradation under a system power bound.

4.5 Conclusion

Our study adopts an algorithmic first-principles approach to the modeling and analysis of systems with respect to time, energy, and power. This approach can offer high-level analytical insights for current debates about the form of future HPC platforms. Our central example is the ability to consider—even in the model's relatively simple form—a variety of "what-if" scenarios, including what would happen if it became necessary to impose a power cap on any of the 12 evaluation platforms.

Constant power, π_0, is a critical limiting factor. It accounts for more than 50% of observed power on seven of the 12 evaluation platforms. Its impact is to reduce the degree of power reconfigurability, and invert our expectations relative to the time and energy costs of primitive operations. These observations raise a natural question for device designers, architects, and system integrators: To what extent can π_0 be reduced, perhaps by more tightly integrating non-processor and non-memory components?

Beyond these observations, we hope the microbenchmarks and modeling methodology, as well as the parameters of table 6, will prove useful to others. Indeed, table 6 is full of interesting data points, such as the fact that random memory access is on the Xeon Phi requires at least one order of magnitude less energy per access than any other platform, suggesting its utility on highly irregular data processing workloads.

The main limitations of this work are its many simplifying assumptions and its microbenchmark-only evaluation. Consequently, there may be a considerable gap between the best-case abstract analysis of this paper and actual applications. In the next two sections, we will consider a more complex application and incorporate DVFS into our model to further enhance its utility.

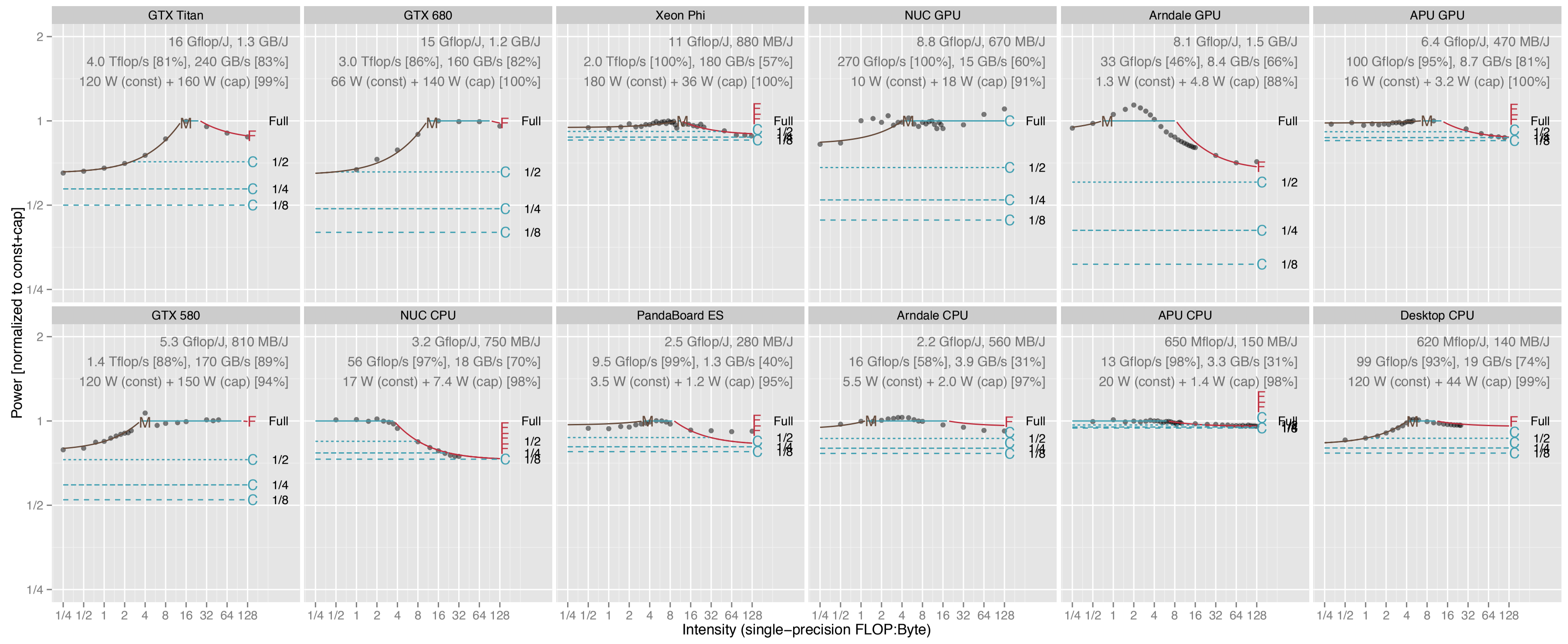
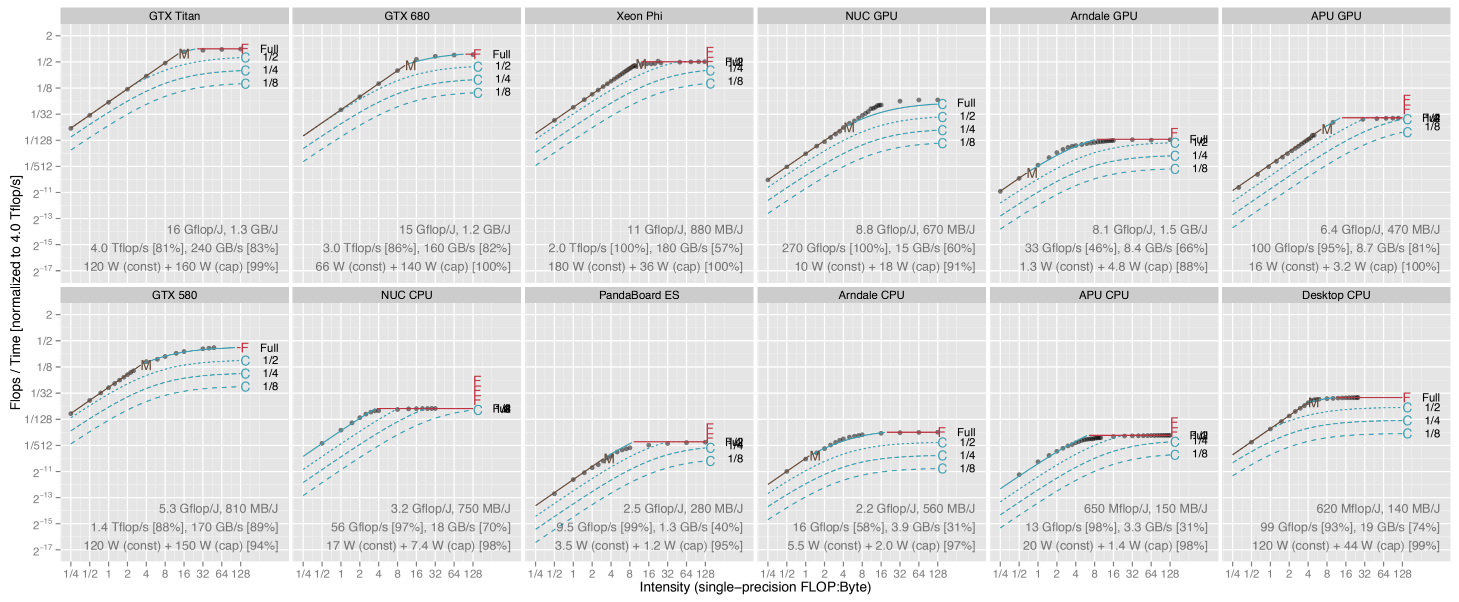


Figure 13: Hypothetical power, performance, and energy efficiency as the usable power cap ($\Delta\pi$) decreases. Note the log-log scales, base 2. Each curve represents a power cap setting: full refers to $\Delta\pi$ from table 6 and $^1/_k$ refers to a power cap setting of $\Delta\pi/k$, for $k \in \{2, 4, 8\}$. The curve color annotations, "F" for flop bound (compute bound), "C" for power cap bound, and "M" for memory bound.

(a) Performance (Gflop/s)



(b) Energy-efficiency (Gflop/J)

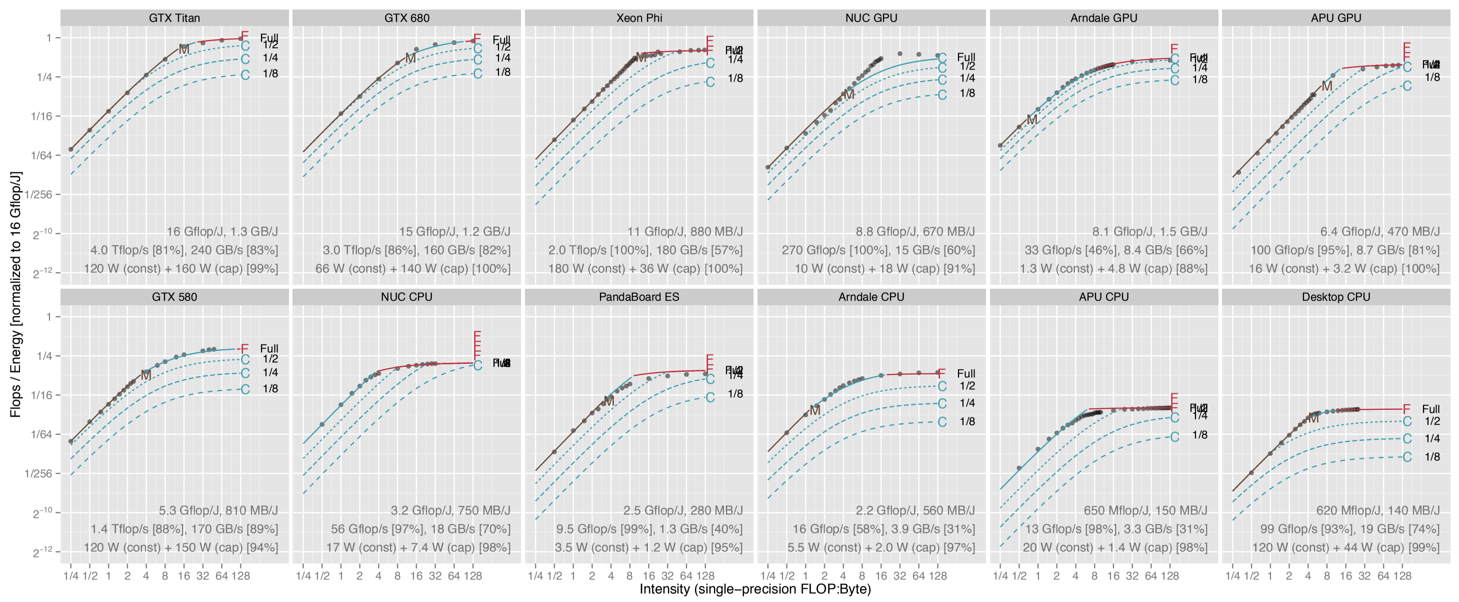


Figure 14: Hypothetical performance and energy efficiency as the usable power cap ($\Delta\pi$) decreases. This figure uses same plotting and notation conventions as fig. 13.

CHAPTER V

INCORPORATING THE EFFECT OF DYNAMIC VOLTAGE AND FREQUENCY SCALING

Contents

To keep our model and analysis simple, we have, so far, assumed fixed costs for operations (both time and energy) and constant power by disabling or circumventing voltage and frequency scaling mechanisms. In this section, we consider the impact that DVFS can have on energy efficiency by incorporating its effect into our model.

We consider DVFS differently from how others have done (§ 2.3) in that, rather than focusing primarily on predicting "slack" in computation as an opportunity to arbitrarily lower frequency and voltage, we attempt to determine *precisely* by how much performance and energy of different operations change. Our model can predict what the optimal frequency voltage settings are for different computations, and tell us which algorithm would be better suited for a particular application under different constraints (e.g., a power cap).

In section § 5.1, we describe the new model which combines our basic model (§ 3.1) with equations that describe the dynamic and leakage power of CMOS transistors [72]. In section § 5.2, we describe our experimental setup that includes the evaluation of a new and unique SoC from NVIDIA that is well suited for our DVFS study. We present our results and discuss their implications in section § 5.3 and give our conclusion in § 5.4.

5.1 Modeling Dynamic Voltage and Frequency Scaling

We adapt our previous work in energy roofline modeling [26, 24] (§ 3.1,§ 4.2) to incorporate processor and memory frequency scaling.

The original model assumed fixed frequency (and therefore performance) for both processor and memory which allowed us to assume fixed time and energy costs for operations, as well as a fixed *constant power*, π_0 [26]. This allowed us to keep our model and analysis simpler and easier to understand; however, in more practical settings, DVFS can have a strong impact on energy efficiency. In order to account for changes in processor and memory frequencies, we must assume that these costs are now variable with respect to the frequency (or rather, the voltage).

The basic equations for dynamic power and static (leakage) power for CMOS transistors [72] are given by eqs. (39) and (40), respectively.

$$P_{dyn} \propto CV^2Af \tag{39}$$

$$P_{leak} \propto V\left(ke^{-qV_{th}/(ak_aT)}\right) \tag{40}$$

For eq. (39), C is the load capacitance, V is the supply voltage, A is the activity factor, and f is the clock frequency. For eq. (40), V_{th} is the threshold voltage, T is temperature, and the parameters q, a, k_a are related to the logic design and fabrication characteristics (which we assume is constant for a given system).

To start, we assumed that activity factor (how often the transistorsx are switching, or "working") and capacitance (physical property of the material and its quantity) remain constant, and reduce eq. (39) to eq. (41).

$$P_{dyn} = C_0 V^2 f \tag{41}$$

where C_0 is some constant that reflects the system's load capacitance and activity factor.

Similarly, we assume that the system operates at a relatively constant temperature when fully utilized for an extended period of time (which is typical for HPC applications), and reduce eq. (40) to eq. (42).

$$P_{leak} = C_1 V \tag{42}$$

where C_1 is some constant. Although the threshold voltage V_{th} may reduce as a result of decreasing the supply voltage, we choose to ignore this effect for two reasons; threshold voltage is strongly dependent on process technology details (which we do not have access to) and because the range of allowed values for the supply voltage is relatively narrow, it may not need to change the threshold voltage to operate at their respective frequencies.

As a reminder, the basic form of our original model [26] (§ 3.1) is shown again in eq. (43).

$$E = W\epsilon_{\text{flop}} + Q\epsilon_{\text{mem}} + \pi_0 T \tag{43}$$

Here, W and Q are the number of floating point operations computed and the number of bytes moved (from slow to fast memory), respectively. ϵ_{flop} and ϵ_{mem} are the *energy* costs associated with floating point and memory operations respectively, π_0 is the constant power (we give the new form for constant power below), and T is the total execution time of our code.

Due to changing the frequency and voltage settings, ϵ_{flop}, ϵ_{mem}, and π_0 are no longer constants, but variables with respect to these settings, as shown in equations 44, 45, and 46.

$$\begin{aligned}\epsilon'_{\text{flop}} &= P_{\text{dyn,flop}}\tau_{\text{flop}} \\ &= C_{0,\text{core}}V_{\text{core}}^2 f_{\text{core}}\tau_{\text{flop}} \\ &= C''_{0,\text{core}}V_{\text{core}}^2 \end{aligned} \tag{44}$$

$$\begin{aligned}\epsilon'_{\text{mem}} &= P_{\text{dyn,mem}}\tau_{\text{mem}} \\ &= C_{0,\text{mem}}V_{\text{mem}}^2 f_{\text{mem}}\tau_{\text{mem}} \\ &= C''_{0,\text{mem}}V_{\text{mem}}^2 \end{aligned} \tag{45}$$

$$\begin{aligned}\pi'_0 &= \pi_{\text{core}} + \pi_{\text{mem}} + \pi_{\text{misc}} \\ &= P_{\text{leak,core}} + P_{\text{leak,mem}} + \pi_{\text{misc}} \\ &= C_{1,\text{core}}V_{\text{core}} + C_{1,\text{mem}}V_{\text{mem}} + \pi_{\text{misc}} \end{aligned} \tag{46}$$

where V_{core}, f_{core}, and V_{mem}, f_{mem} are the processor and memory supply voltages and clock frequencies, respectively, and τ_{flop} and τ_{mem} are the time costs of floating point operations and data movement. Note that some constants appear when factors cancel out in eqs. (44) and (45), but these ultimately become parts of constants $C''_{0,\text{core}}$, $C''_{1,\text{core}}$ and $C''_{0,\text{mem}}$, $C''_{1,\text{mem}}$. For example, in eq. (44) $f_{\text{core}}\ \tau_{\text{flop}} = f_{\text{core}} \cdot 1/\left(2N_{cores}f_{\text{core}}\right) = 1/\left(2N_{cores}\right)$ (assuming the system has N_{core} cores, each capable of executing a single fused–multiply–add (FMA) instructions per cycle), so $C''_{0,\text{core}} = C_{0,\text{core}} \cdot 1/\left(2N_{cores}\right)$.

We define constant power to be the power that is *not directly* involved in computation or data movemement (§ 3.1.5). Such power includes static power and power consumed by peripherals. In equation 46, we separate these different components since *some* of these components will change with respect to the supply voltage and clock frequency; π_{core} is the constant power dissipated by the processing cores; π_{mem}

is the constant power dissipated by the memory units (DRAM, memory controller, etc.); and π_{misc} is the power dissipated by peripherals. In the case of SoCs that have both a CPU and a GPU on the same chip, π_{misc} for the GPU component could also include the idle power dissipated by the CPU cores, and vice versa.

The final equation for total energy dissipated is shown in eq. (47).

$$\begin{aligned} E = {} & WC''_{0,\text{core}}V^2_{\text{core}} + QC''_{0,\text{mem}}V^2_{\text{mem}} \\ & + C_{1,\text{core}}V_{\text{core}}T + C_{1,\text{mem}}V_{\text{mem}}T \\ & + \pi_{\text{misc}}T \end{aligned} \tag{47}$$

5.2 *Experimental Setup*

In this section we describe the hardware and the software stack behind our test platform, NVIDIA's Jetson TK1 mobile SoC development board. One unique aspect of this device it that it incorporates a single CUDA–capable SMX multiprocessor (with 192 CUDA cores) that is typically found in high–performance GPUs with a quad–core ARM CPU on the same chip, essentially making it one of the first complete high–performance computing–capable, yet *low–power*, platforms. Another interesting feature of this system is its ability to easily change both its core and memory frequencies (and voltage) over a wide range of values. This allows for a large number of systems settings with which to experiment, both in terms of performance and power, making it an ideal platform for DVFS studies.

5.2.1 Hardware

The main processing engine behind this development board is the Tegra K1 SoC which consists of a 4–plus–1 quad–core ARM Cortex A15 CPU and a single Kepler GPU multiprocessor (SMX) with 192 CUDA cores. The processor is built on TSMC's 28nm HPM process and also includes other components typically found on mobile processors such as audio and video encoder/decoder, and image signal processor (ISP)

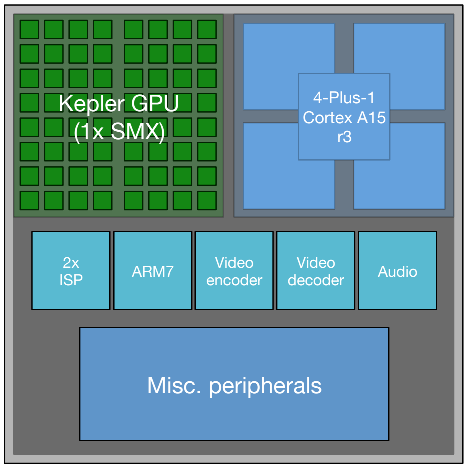


Figure 15: Schematic of the NVIDIA Tegra K1 mobile processor

for processing media content. The multiprocessor used in the Tegra K1 SoC is based on the Kepler microarchitecture [98] and is capable of delivering 1 single–precision fused multiply–add (FMA) per cycle per core via four warp schedulers. A schematic of the Tegra K1 SoC is shown in fig. 15.

When compared to a traditional discrete GPU, one distinct feature is the use of 64–bit wide DDR3L memory rather than 384–bit wide GDDR5 memory which can deliver significantly higher memory bandwidth. The total memory capacity is limited to 2 GB and is also *shared* between the CPU and the GPU.

One downside of the Tegra K1 SoC for use in HPC is that the SMX on Tegra K1 is like those found on desktop–grade GPUs, rather than those found on Tesla–grade GPUs; its double precision computation capability has been severely limited (1/24× of single precision). As it stands, the Jetson TK1 development board is ill–suited for HPC. However, we assume that it would be trivial to simply activate the double–precision cores (if it's already present but simply deactivated) or to replace the SMX with a Tesla–grade version.

5.2.2 Software

In terms of the software stack, the Jetson TK1 system runs Linux for Tegra (L4T), a modified Ubuntu 14.04 Linux distribution which uses Linux kernel 3.10. The software stack also includes CUDA 6 Toolkit, OpenGL 4.4 drivers, and the NVIDIA VisionWorks Toolkit.

Jetson TK1 allows users to change both the core and memory clock frequencies from the OS without requiring a reboot. There are 15 and seven different frequency settings for the core and the memory respectively, allowing for a total of $15 \times 7 = 105$ unique system settings. The list of available frequencies and voltage settings can be accessed through

```
/sys/sys/kernel/debug/clock/dvfs_table
```

The frequency of the SMX can be set by writing the desired frequency (in Hz) to

```
/sys/kernel/debug/clock/override.gbus/rate
```

followed by writing "1" to

```
/sys/kernel/debug/clock/override.gbus/state
```

Similarly, the memory frequency can be changed by writing the desired frequency and "1" respectively to

```
/sys/kernel/debug/clock/override.emc/rate
```

and

```
/sys/kernel/debug/clock/override.emc/state
```

For more details on the hardware and software specification of Jetson TK1, readers may refer to its technical brief [101].

5.2.3 HPC candidate building block bake–off

Now that we have analyzed the Jetson TK1 system using our energy roofline model, we can now add it to our ever–growing database of HPC building blocks [24] and see how it fares against other candidate systems. Here, we limit our comparison to single–precision performance only as Jetson TK1 has severely limited double–precision performance.

In the context of HPC systems, an interesting question that one may ask is this; given a limited power budget, which systems would you use to build your new supercomputer? Would you build a cluster of HPC nodes made up of traditional server–grade CPUs and high–end GPU/accelerators? If so, why, and why not an even *larger* cluster of platforms based on low–power SoCs? While a natural response is "it depends," our energy roofline analysis provides one *method* of answering this question in a precise and analytical manner.

5.2.3.1 Jetson TK1 vs. GTX Titan

First, let us perform an iso–power comparison of Jetson TK1 to GTX Titan, a high–end consumer–grade GPU with peak computation and memory bandwidth performances of 4.7 TFLOP/s and 288 GB/s respectively. We first calculate the number of Jetson TK1 systems that are required to match the power dissipation of a *single* GTX Titan by taking the maximum observed power dissipation of GTX Titan and dividing it by the maximum power dissipation of Jetson TK1; our experimental results show that it would take approximately 22 Jetson TK1 development boards to match a single GTX Titan. Then, we generate roofline plots for time, energy, and power for this hypothetical "cluster" to see how it fares against GTX Titan over a range of arithmetic intensities. We use arithmetic intensities as a means of parameterizing target applications using a single, simple metric. These plots are shown in fig. 16.

Power As it can be see from the right–most powerline plot in fig. 16, 22× Jetson TK1 cluster's power dissipation signature follows that of the GTX Titan almost exactly. This is not entirely surprising given that both of these systems have the same microarchitecture. However, given that 22× Jetson TK1 systems would provide a much wider surface area and volume with which to dissipate heat, it would incur a lower cooling cost and perhaps more stability as a result. On the other hand, the larger number of units would most likely incur a higher infrastructure set up cost, as well as a larger floor space to house it. In a real world setting, the actual amount would depend on the configuration (number of boards per node and number of processors per board) and the network type (number of required routers and network cards).

Energy efficiency In terms of energy *efficiency*, which is illustrated by the energy "arch line" in the center plot, Jetson TK1 is slightly more efficient. Energy efficiency as defined by the energy roofline model is a metric of how much energy is used for "useful" work (i.e., floating point operations) as compared to those that are *not* useful (i.e., moving data). Analogous to the time roofline, energy arch line has what we call an *energy balance* [26] which is the arithmetic intensity point at which equal amounts of energy are consumed by computation and data movement (similar to how time balance point indicates the point where equal *time* is spent on computation and data movement). Typically, we desire a low energy balance point, as that allows more algorithms to be energy–efficient. For example, for GTX Titan, time balance occurs at arithmetic intensity of approximately 16, whereas the energy balance occurs at approximately 8.

Our energy archline plot shows that when the system is memory bound, Jetson TK1 is only *slightly* more efficient, whereas when it's compute bound, Jetson TK1 is significantly more energy efficient. This means that energy cost of data movement, ϵ_{mem}, is only slightly lower for Jetson TK1, while the cost of computation ϵ_{flop} is *significantly* lower. There is no plot for a 22× Jetson TK1 cluster as scaling the number of cores up does not change the energy efficiency of the system, as we are ignoring network interconnect cost.

Performance In terms of performance, the left–most plot (time roofline) shows that the aggregate bandwidth of 22 Jetson TK1 systems is slightly higher, while the aggregate peak computation is at least 50% better. This means that for applications that are not network–bound, the 22× Jetson TK1 cluster would perform better.

Dollar cost In terms of dollar cost, a cluster based on Jetson TK1 would be more expensive; the dollar cost per CUDA core is exactly $1, whereas the Titan retails for around $1,000 for 2688 CUDA cores. However, if we compare it to a typical supercomputing node made up of server–class CPU, motherboard, and power supply, the GTX Titan based system would incur at least a similar cost (we do not include DRAM cost since the Jetson TK1 is currently limited to only 2 GB of memory). The story would change yet again if we include the cost of the interconnection network.

One last consideration is the cost of cooling; since a Jetson TK1 cluster would consist of many more nodes, the infrastructure for cooling would be higher (e.g., larger housing space, number of heat sinks, etc.), but the running cost of cooling would be lower due to lower areal density.

Summary It would appear that in a power constrained scenario, Jetson TK1 clusters would make a better building block for a supercomputer in terms of performance, energy efficiency, and cooling. Although not quite as apparent, in terms of dollar cost, Jetson TK1 cluster *could* also be cheaper in some instances, depending on the the choice of interconnection network and cooling infrastructure.

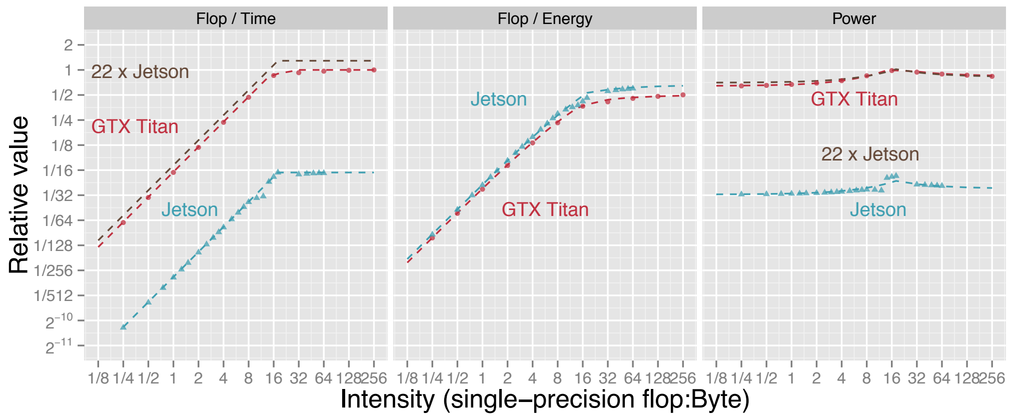


Figure 16: 22× Jetson TK1 vs. GTX Titan

5.2.3.2 Jetson TK1 vs. Tesla K40c

Tesla K40c is the latest generation Tesla–grade GPU from NVIDIA, also based on the Kepler microarchitecture. It is capable of delivering 4.29 GFLOP/s in single–precision performance and 288 GB/s in memory bandwidth. When compared to the GTX Titan, it is significantly more power efficient – it can deliver similar performance at much lower power levels. For Tesla K40c, it takes approximately 15 Jetson TK1 systems to match its peak power. Figure 17 shows the comparison between 15× Jetson TK1 cluster and Tesla K40c.

Power In terms of power, the power signature of 15× Jetson TK1 cluster again closely matches that of Tesla K40c in the lower arithmetic intensity regimes. However, for compute–bound intensities, Tesla K40c actually dissipates slightly *less* power. Although the 15× Jetson TK1 cluster dissipates slightly more power, overall, it would still maintain most of the advantages of using SoCs, as discussed in section 5.2.3.1.

Energy efficiency As with power, energy efficiency is identical at memory–bound regimes, while for compute–bound regimes, Tesla K40c is slightly more efficient. This seems to indicate that K40c would make for a better HPC building block overall if energy efficiency was paramount.

Performance In terms of performance, the two systems are almost identical, both in terms of aggregate bandwidth and aggregate floating point capability.

Dollar cost Dollar cost is difficult to compare in this case, as significant percentage the cost of Tesla–grade GPU comes from using ECC memory. However, if we ignore this fact, 15 Jetson TK1 systems would cost approximately the same as a single K40c. When we compare these two systems node–to–node (including the cost of the CPU, motherboard, etc.), then a cluster of Jetson TK1 would be a much cheaper alternative.

Summary Comparing a cluster of Jetson TK1 systems to GTX Titan and Tesla K40c shows that, unfortunately, there isn't a clear winner. However, this could be considered good news for proponents of supercomputers based on mobile SoCs. Current trend in HPC point towards increasingly application–specific hardware systems [94, 18, 102] and severe limitations on power [42]. Therefore, depending on the target application and the power constraint, a supercomputer based on Jetson TK1 could show tangible benefits in terms, performance, energy, power, cooling and perhaps even more. For example, one such use case could be a distributed application written in CUDA that requires high network bi–section bandwidth, such as fast Fourier transform (FFT) [30] and certain sorting algorithms.

Jetson TK1 in its current form still has several disadvantages when considering HPC; it lacks support for larger memory capacity due to its having a 32–bit processor. It also does not have any support for interfacing to high–performance routing

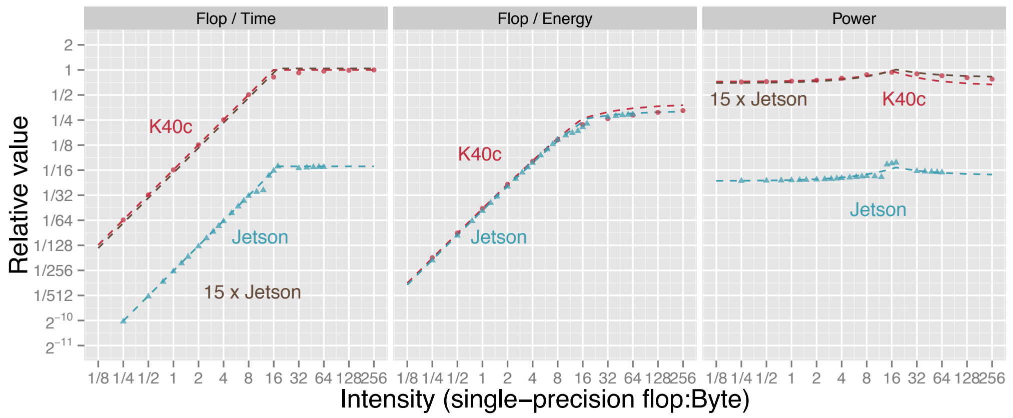


Figure 17: 15× Jetson TK1 vs. Tesla K40c

networks that are essential for supercomputers. However, we believe that once there is enough interest in the HPC community for such systems, these problems will be easily overcome. In fact, the problem of limited memory capacity will most likely be solved in the near future when the 64–bit Denver processor [15] is integrated into Tegra K1.

5.3 *Results and Discussion*

We derive the various constants defined in § 5.1 using linear regression on experimental data as before (§ 3.3.2) and validate our results.

5.3.1 Fitting and Validation

Linear regression We collected data under 16 different frequency and voltage settings on Jetson TK1 using three separate microbenchmarks – intensity (single and double precision), integer, and cache (both shared memory and L2) – for a total of 1856 samples (16 × 116). For each setting, we measured 25 samples from single–precision intensity benchmark, 36 from double–precision intensity benchmark, 23 from integer benchmark, 16 from shared memory benchmark, and 16 from L2 cache benchmark $(25 + 36 + 23 + 16 + 16 = 116)$. The 16 settings are summarized in table 7.

Non–negative least square (nnls) [21] function included with R[1] was used for the fitting, with the fitting equation shown in eq. (48).

$$\begin{aligned}
\frac{E}{W} &= C''_{0,\text{core,single}} V^2_{\text{core}} + \frac{Q}{W} C''_{0,\text{mem}} V^2_{\text{mem}} \\
&+ C_{1,\text{core}} V_{\text{core}} \frac{T}{W} + C_{1,\text{mem}} V_{\text{mem}} \frac{T}{W} \\
&+ \frac{W_{int}}{W} C''_{0,\text{core,integer}} V^2_{\text{core}} + \frac{Q_{SM}}{W} C''_{0,\text{core,SM}} V^2_{\text{core}} + \frac{Q_{L2}}{W} C''_{0,\text{core,L2}} V^2_{\text{core}} \\
&+ \pi_{\text{misc}} \frac{T}{W} + \Delta C''_{0,\text{core,double}} V^2_{\text{core}} R
\end{aligned} \tag{48}$$

where $C''_{0,\text{core,single}}$ is the single precision floating equivalent for $C''_{0,\text{core}}$ (eq. (44)), and $C''_{0,\text{core,single}} + \Delta C''_{0,\text{core,double}}$ is the double precision equivalent The binary variable R is set to 0 for single precision and 1 for double precision.

(W_{int}, $C''_{0,\text{core,integer}}$), ($Q_{SM}$, $C''_{0,\text{core,SM}}$), and (Q_{L2}, $C''_{0,\text{core,L2}}$) are the integer, shared memory, and L2 equivalents for W/Q, and $C''_{0,\text{core}}$, respectively. Notice that access to shared memory and L2 cache depends on the core frequency (V_{core}) rather than the memory frequency (V_{mem}), even though they are accessing data.

Once we obtain the coefficient values, we can then compute ϵ_{flop} (for both single and double precision), ϵ_{mem}, and π_0 using equations eqs. (44)–(46). The cost of integer computation and access to shared memory and L2 cache can be computed similarly using eq. (44).

Cross Validation We first used cross validation [50] on our collected data to assess how well it will generalize to an independent data set when used to make predictions of energy consumption for other kernels and applications. We chose k–fold cross validation, where the data is divided into k equal size *subsamples*, and of the k

[1]`http://r-project.org`

subsamples, a single subsample is retained as the validation data for testing the model, and the remaining $k-1$ subsamples are used as training data. This process is repeated k times, with each of the k subsamples used exactly once as the validation data. The k results from the subsamples are then averaged to produce a single estimation.

The result of 16–fold cross validation of our data set of 1856 samples shows a mean error of 6.56%, with a standard deviation of 3.80%. The minimum and maximum error are 1.60% and 15.22%, respectively.

Prediction trend We also validated our data using 2–fold cross validation (also known as holdout method) to see how the error rates change with different benchmarks and frequency settings when the derived energy costs from one set is applied to the other. The derived energy costs are summarized in table 7. Data type "T" was used for training, and type "V" was used for validation.

When compared to measured energy, the mean error for the validation set was 2.87% with a standard deviation of 2.47%, and the minimum and maximum error were 0.00% and 11.94%, respectively. When we compared error for single precision data to error for double precision data, we saw that single precision fared much worse at 4.25% than double precision at 2.67%; we believe that this is caused by the data overfitting double precision data due to their having more samples (25 vs.36 per setting). When comparing error rates across arithmetic intensities, we found that the error rate was much lower for the memory–bound region. This is again likely caused by having more data points in the memory bound region; although the cost of computation is fundamentally different for single and double precision computation, cost of moving data is the same for both benchmarks, which results in curve fitting being more biased towards data movement.

Table 7: List of settings and derived energy costs.

Type	Core freq. (MHz)	Core volt. (mV)	Memory freq. (MHz)	Memory volt. (mV)	Energy SP (pJ)	Energy DP (pJ)	Energy Integer (pJ)	Energy SM (pJ)	Energy L2 (pJ)	Energy Mem (pJ)	Const. power (W)
T	852	1030	924	1010	29.0	139.1	60.0	35.4	90.2	377.0	6.8
T	396	770	924	1010	16.2	77.7	33.5	19.8	50.4	377.0	6.1
T	852	1030	528	880	29.0	139.1	60.0	35.4	90.2	286.2	6.3
T	648	890	528	880	21.7	103.8	44.8	26.4	67.3	286.2	5.9
T	396	770	528	880	16.2	77.7	33.5	19.8	50.4	286.2	5.6
T	852	1030	204	800	29.0	139.1	60.0	35.4	90.2	236.5	6.0
T	648	890	204	800	21.7	103.8	44.8	26.4	67.3	236.5	5.6
T	396	770	204	800	16.2	77.7	33.5	19.8	50.4	236.5	5.2
V	756	950	924	1010	24.7	118.3	51.0	30.1	76.7	377.0	6.6
V	180	760	528	880	15.8	75.7	32.7	19.3	49.1	286.2	5.5
V	540	840	528	880	19.3	92.5	39.9	23.5	59.9	286.2	5.8
V	540	840	204	800	19.3	92.5	39.9	23.5	59.9	236.5	5.4
V	756	950	204	800	24.7	118.3	51.0	30.1	76.7	236.5	5.8
V	72	760	68	800	15.8	75.7	32.7	19.3	49.1	236.5	5.2
V	756	950	68	800	24.7	118.3	51.0	30.1	76.7	236.5	5.8
V	180	760	924	1010	15.8	75.7	32.7	19.3	49.1	377.0	6.0

Table 8: Summary of autotuning results. "Energy lost" shows how much *more* energy the configuration chosen by our model or the "time oracle" dissipated when compared to experimentally measured minimum.

			Energy lost (%)		
		Mispredictions	Mean	Minimum	Maximum
Single	Model	0 (out of 25)	0	0	0
	Time Oracle	20 (out of 25)	18.52	7.21	26.52
Double	Model	10 (out of 36)	3.11	0.34	7.30
	Time Oracle	23 (out of 36)	3.95	0.23	13.90
Integer	Model	6 (out of 23)	2.37	0.32	5.12
	Time Oracle	23 (out of 23)	3.56	0.44	9.72
Shared memory	Model	7 (out of 10)	3.31	2.92	3.99
	Time Oracle	10 (out of 10)	10.64	7.07	12.75
L2	Model	0 (out of 9)	0	0	0
	Time Oracle	0 (out of 9)	10.71	10.49	11.28

5.3.2 Autotuning for energy

In the context of "autotuning," we compare using our model to predict the "best" (i.e., minimizes energy) system configuration ($[f_{core}, f_{mem}]$ pair) to using the execution time to choose the best configuration instead for a given arithmetic intensity on our intensity microbenchmark. That is, we assume that we have a "time oracle" that can tell us a priori which configuration yields the best performance for a given arithmetic intensity. The "best" configuration for each case was chosen from one of the 16 settings shown in table 7, and the resulting energy consumption (chosen by our two competitors) was compared to the measured energy consumption to see how much energy was "lost" in each case.

If "race–to–halt[2]" is true, we should expect the configuration based on execution time should do just as well or better than our model. However, we found this to be untrue. Table 8 summarizes our findings.

As it can be seen from table 8, blindly choosing the configuration that yields the best performance (or the smallest execution time) did not always result in the most

[2]The race–to–halt strategy says that the best way to save energy is to run as fast as possible and then turn everything off

energy efficient configuration. In the case of the single precision microbenchmark, adopting the "race–to–halt" strategy resulted in getting energy *inefficient* configuration 20 out of 25 cases (arithmetic intensity values), whereas our model never made incorrect predictions. Surprisingly, despite having predicted energy consumption more accurately for double precision (§ 5.3.1), our model made incorrect predictions 10 out of 36 cases. It is still much better than using the "race–to–halt" strategy that made 23 incorrect predictions.

For the shared memory and L2 benchmarks, the number of total tests were 10 and nine respectively. This is lower than the number of samples taken (16 for both cases (§ 5.3.1)) because the total number of unique intensities for these benchmarks are lower than the number of samples; some of the samples were taken with the same number of bytes loaded but using different thread configurations. Although predictions for shared memory are bad for both cases (seven mispredictions for our model vs. 10 for the time oracle), the configuration chosen by our model dissipates at most only 3.99% more, whereas the configuration chosen by the time oracle could dissipate as much as 12.75% more.

Granted, the energy "lost" by either strategy is relatively small (at most 7.3% vs. 26.52% more than experimentally determined minimum energy). However, unlike some other strategies in DVFS, where energy is "gained" only in the presence of application "slack," we use time–energy trade–off to find the most energy–efficient setting even for a very uniform computation like our microbenchmarks. When we compare our strategy to other system–wide approaches that try to take advantage of time–energy trade–off of frequency scaling, we have the advantage of providing a more precise and analytical approach which can be expressed using algorithm–friendly metrics.

5.4 Conclusion

Our experience with the Jetson TK1 development board was overall a positive one. The ability to change both the core and memory frequency manually allowed us to validate our new DVFS–aware model, which can predict energy consumption of our microbenchmarks under different DVFS settings to within 2.87%. It also allowed us to conduct interesting time–energy trade–off experiments using DVFS, where we found that there could be non–trivial energy *loss* if we assumed "race–to–halt" to be the best strategy for achieving energy efficiency. In that regard, an autotuning framework based on our energy roofline model could help in choosing better system settings for energy efficiency.

Our energy and time roofline analysis of the Jetson TK1 and its comparison against other high–end GPUs have shown that Jetson TK1 can be a competitive alternative with which to build supercomputers. In terms of performance and energy efficiency, 22× Jetson TK1 cluster outperforms the GTX Titan, especially in the compute–bound regime where the aggregate compute performance of the Jetson TK1 cluster is as much as 1.5× better. The 15× Jetson TK1 cluster closely matches the latest high–end Tesla GPU in terms of both memory and compute performance. This suggests that low–power systems are neither better nor worse than high–end systems such as Tesla GPUs; in the end, you get (in performance) what you pay for (in power).

Limitations Althgough the Jetson TK1 development board shows potential, it is not without problems. In the context of HPC, the biggest problems are the lack of high capacity memory, double–precision performance, and interface to high–speed internconnection network. However, we believe that these problems could be easily solved if there was enough interest from the HPC community.

Our validation and analysis efforts are also not without limitations. Although useful, our studies are limited to microbenchmarking studies; the more interesting question is whether our model can be applied to real applications running on real systems. We will address this issue in the next section.

Lastly, we ignore the costs of the network. We do so for two main reasons. First, as far as we are aware, there have been no studies on designing energy efficient network components, making it difficult (and perhaps unfair) to incorporate them in our analysis. Second, we do not know what kind of network topology would be suitable for a cluster of embedded SoCs; it may be that a traditional topology with multiple SoCs per node to mimic current designs is ideal; however, it may just as likely be that a completely different type of topology is required for such large number of nodes (15$\times$ more). We believe that this could be an interesting research topic for the future.

CHAPTER VI

EVALUATING THE FAST MULTIPOLE METHOD USING THE ENERGY ROOFLINE MODEL

Contents

In our last chapter, we apply our model and analysis to predict the energy consumption of a real and important application, the fast multipole method (FMM), running on the GPU core of the Jetson K1 platform. We take eight randomly chosen processor and memory frequency settings and run FMM using eight different input sets for a total of 64 separate test cases. We use our previously developed and highly–tuned FMM implementation [23] for the tests.

We first break execute FMM on our test platform to gather relevant performance counter measurements to breakdown the FMM kernel into its individual components. Then, we apply the derived costs from § 5.3 as a weighted sum of these components to get the final energy consumption estimate and compare it against measured energy.

We start this chapter by giving a brief overview of FMM in § 6.1. Then in § 6.2, we describe the process of deconstructing the FMM kernel into its various components, such as floating point and integer operations, and accessing data from various levels of the memory hierarchy. We present our validation results and observations in § 6.3 and conclude in § 6.4.

6.1 Introduction to the Fast Multipole Method

Given a system of N *source* particles, with positions given by $\{y_1, \ldots, y_N\}$, and N *targets* with positions $\{x_1, \ldots, x_N\}$, we wish to compute the N sums,

$$f(x_i) = \sum_{j=1}^{N} K(x_i, y_i) \cdot s(y_j), \quad i = 1, \ldots, N \tag{49}$$

where $f(x)$ is the desired *potential* at target point x; $s(y)$ is the *density* at source point y; and $K(x, y)$ is an *interaction kernel* that specifies "the physics" of the problem. For instance, the single-layer Laplace kernel, $K(x, y) = \frac{1}{4\pi} \frac{1}{||x-y||}$, might model electrostatic or gravitational interactions.

Evaluating these sums appears to require $O(N^2)$ operations. The FMM instead computes *approximations* of all of these sums in optimal $O(N)$ time with a guaranteed user-specified accuracy ϵ, where the desired accuracy changes the complexity constant [54]. The FMM is based on two key ideas:

- organizing the points in a *spatial tree*; and
- using *fast approximate evaluations*, in which we compute summaries at each node using a constant number of tree traversals with constant work per node.

We model and implement the *kernel-independent* variant of the FMM, or KIFMM [130]. KIFMM has the same structure as the classical FMM [54]. Its main advantage is that it avoids the mathematically challenging analytic expansion of the kernel, instead requiring only the ability to evaluate the kernel. This feature of the KIFMM allows us to leverage our optimizations and techniques and apply them to new kernels and problems.

Tree construction Given the input points and a user-defined parameter Q, we construct an octree T (or quad-tree in 2D) by starting with a single box representing all the points and recursively subdividing each box if it contains more than Q points. Each box (octant in 3D or quadrant in 2D) becomes a tree node whose children are its immediate sub boxes. During construction, we associate with each node one or more neighbor *lists*. Each list has bounded constant length and contains (logical) pointers to a subset of other tree nodes. These are canonically known as the U, V, W, and X lists. For example, every leaf box B has a U *list*, $U(B)$, which is the list of all leaves adjacent to B. Figure 18 shows a quad-tree example, where neighborhood list nodes for B are labeled accordingly.

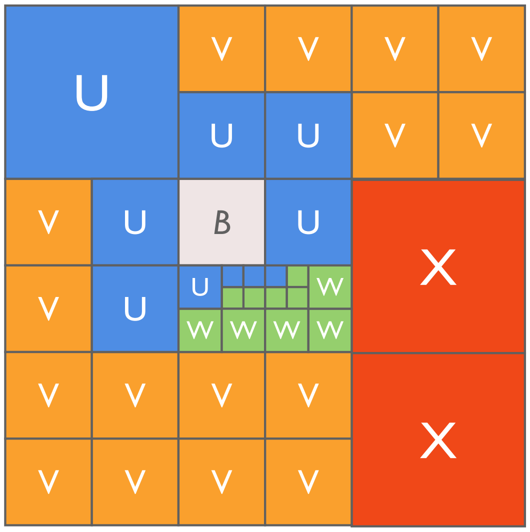


Figure 18: U, V, W, and X lists of a tree node B for an adaptive quadtree.

Tree construction has $O(N \log N)$ complexity, and the $O(N)$ optimality of FMM refers to the evaluation phase (below). However, tree construction is typically a small fraction of the total time; moreover, many applications build the tree periodically, thereby enabling amortization of this cost over several evaluations.

Evaluation Given the tree T, evaluating the sums consists of six distinct computational phases: there is one phase for each of the U, V, W, and X lists, as well as *upward* (up) and *downward* (down) phases. These phases involve traversals of T or subsets of T. Rather than describe each phase in detail, we refer the readers to the following publications [54, 129, 130].

There are multiple levels of concurrency during evaluation: across phases (e.g., the upward and U-list phases can be executed independently), within a phase (e.g., each leaf box can be evaluated independently during the U-list phase), and within the per-octant computation (e.g., vectorizing each direct evaluation).

Another important property of FMM is the variability of its arithmetic intensity. The two most expensive phases of FMM are U list and V list computations; U list computation is highly compute bound as it calculates interactions with its nearest neighbors directly ($O(q^2)$); the V list computation, on the other hand, is highly memory bound, as it approximates interactions with far neighbors through fast fourier transform (FFT) and vector addition kernels. By changing the input Q, the maximum number of points per box, we can change the workload of these two phases so that its overall ratio of computation to data movement can be tailored to a particular platform to maximize performance.

6.2 Breaking Down FMM

We use *nvprof* performance counter monitor (PCM) to gather the necessary performance counter measurements to create a breakdown of our FMM implementation. A summary of these counters are given in table 9. Type "E" in the table is a counter event, which corresponds to a single hardware counter value, and type "M" is a counter metric, which is a *characteristic* of the running application and is calculated from one or more events.

For the number of different instructions, we use the readings given by counter events directly. However, calculating the number of bytes coming from different levels of the memory is a little more complicated, due to the complex nature of the memory hierarchy.

- First of all, we can calculate the amount of data coming from L1 cache by multiplying *l1_global_load_hit* by 128 bytes, the cache line size for L1. Total access to shared memory can be calculated similarly using the event *l1_shared_load_transactions*
- The event *l2_subp0_total_read_sector_queries* multiplied 32 bytes gives the total *request* to the L2 cache, or those that missed in the L1 cache. This data will come from either L2 cache or main memory, described below.
- We can use events *l2_subp0_read_l1_hit_sectors* – *l2_subp3_read_l1_hit_sectors* to calculate data that hit in the L2 cache. These counters are summed and multiplied by 32 bytes to get the total amount.
- We can use the events *fb_subp0_read_sectors* and *fb_subp1_read_sectors* to calculate data read from main memory, also by summing the counters and multiplying it by 32 bytes to get the total amount.
- Similarly, we can calculate the total amount of data written back to memory using counters *gst_request*, *2_subp0_total_write_sector_queries*, and *l1_shared_store_transactions*.

The breakdown of FMM for different input parameters is shown in fig. 19a and fig. 19b. The input parameters shown in the x–axis represent different combinations of N, the total number of points, and Q, the maximum number of points per box (§ 6.1).

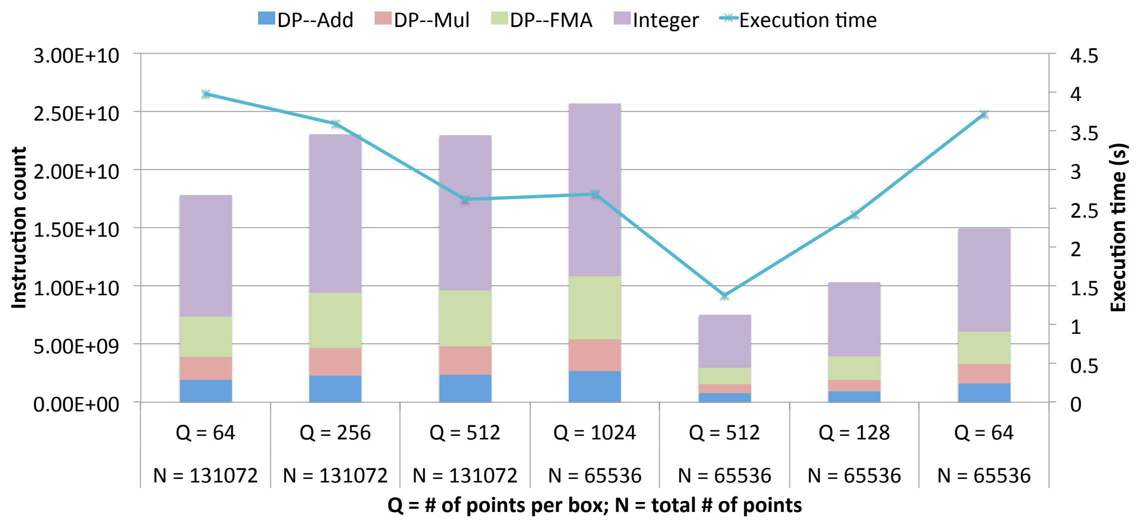
6.3 Results and Observations

Using the instruction and data breakdown of the FMM kernel (§ 6.2), and the predicted cost of operations under different DVFS settings (§ 5.3), we can now predict the total energy consumption of FMM. We validate our predictions against real energy measurements of eight DVFS settings and eight different sets of inputs to our FMM kernel, for a total of 64 test cases. The DVFS settings and the input parameters are summarized in table 10

Table 9: Summary of counter events and metrics used to create a breakdown of the FMM kernel. Type “E” are events and type “M” are metrics.

Type	Name	Description
M	flops_dp_fma	# of double–precision floating point multiply–accumulate operations
M	flops_dp_add	# of double–precision floating point add operations
M	flops_dp_mul	# of double–precision floating point multiply operations
M	inst_integer	# of integer instructions
E	l1_global_load_hit	# of cache lines that hit in L1 cache
E	l2_subp0_total_read_sector_queries	Total read request for slice 0 of L2 cache
E	gld_request	# of load instructions
E	l1_shared_load_transactions	# of shared load transactions
E	fb_subp0_read_sectors	# of DRAM read request to sub partition 0
E	fb_subp1_read_sectors	# of DRAM read request to sub partition 1
E	l2_subp0_read_l1_hit_sectors	# of read requests from L1 that hit in slice 0 of L2 cache
E	l2_subp1_read_l1_hit_sectors	# of read requests from L1 that hit in slice 1 of L2 cache
E	l2_subp2_read_l1_hit_sectors	# of read requests from L1 that hit in slice 2 of L2 cache
E	l2_subp3_read_l1_hit_sectors	# of read requests from L1 that hit in slice 3 of L2 cache
E	gst_request	# of store instructions
E	l2_subp0_total_write_sector_queries	Total write request to slice 0 of L2 cache
E	l1_shared_store_transactions	# of shared store transactions

(a) Instruction breakdown



(b) Data breakdown

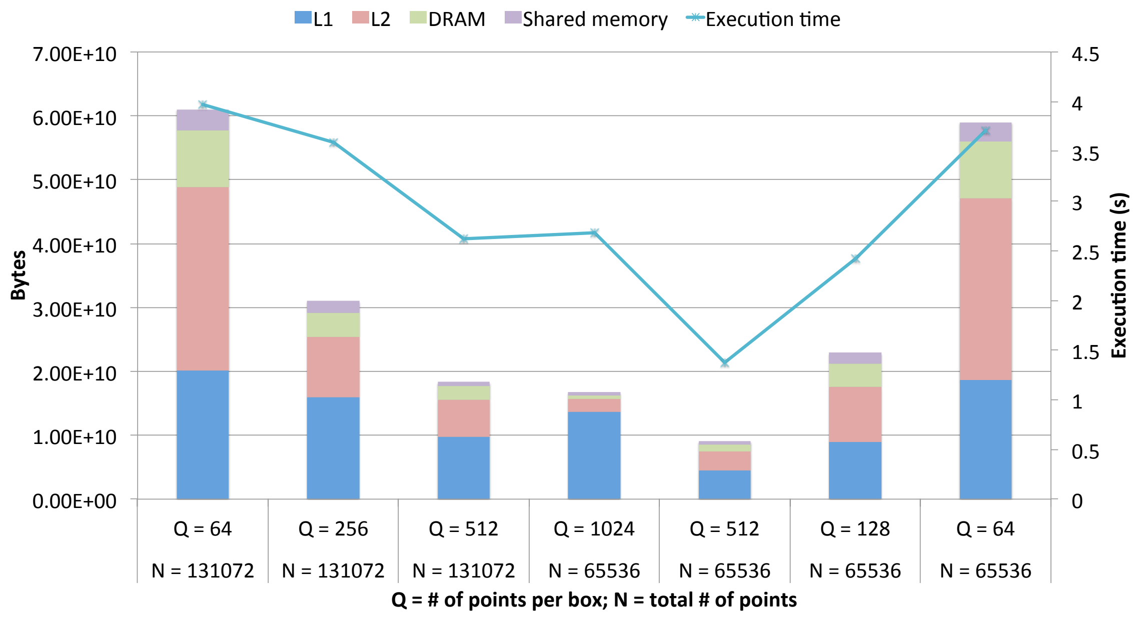


Figure 19: Breakdown of the FMM kernel to component instructions and data access to different levels of the memory hierarchy

Table 10: Summary of DVFS settings and input parameters used for validation

System Setting			FMM Input		
ID	Core Frequency	Memory Frequency	ID	N	Q
S1	852 MHz	924 MHz	F1	262144	128
S2	756 MHz	924 MHz	F2	131072	64
S3	180 MHz	924 MHz	F3	131072	256
S4	852 MHz	792 MHz	F4	131072	512
S5	612 MHz	528 MHz	F5	65536	1024
S6	540 MHz	528 MHz	F6	65536	512
S7	612 MHz	396 MHz	F7	65536	128
S8	852 MHz	204 MHz	F8	65536	64

Although we have derived the cost of FMA and integer operations, we have yet to write microbenchmarks to derive add and multiply instructions separately; in algorithm analysis, add, multiply, and FMA instructions are not differentiated and treated as floating point operations. However, since a large fraction of energy consumed by operations are due to instruction overheads [58], we assume that multiply, add and FMA instructions incur similar cost as they all take similar execution paths.

Figure 20 shows the result of our validation. Over all 64 test cases, we observed a mean error of 6.17% when comparing our model estimates against measured energy, with a standard deviation of 4.65%. However, as can be seen from the figure, the error rate ranges from as low as 0.09% to as high as 14.89%.

Observations Generally speaking, the error rate tends to be higher for inputs with small values for the number of points per box, Q, regardless of the DVFS setting. We attribute this to the fact that reducing the number of points per box increases the portion of the total time spent on the U list phase, which consists of FFT and vector addition kernels; these kernels, especially FFT, exhibit complicated data access patterns which, together with the complex memory hierarchy of modern architectures, may utilize additional hardware whose usage is not captured by our microbenchmarks.

When we break down energy consumption by type – computation, data movement,

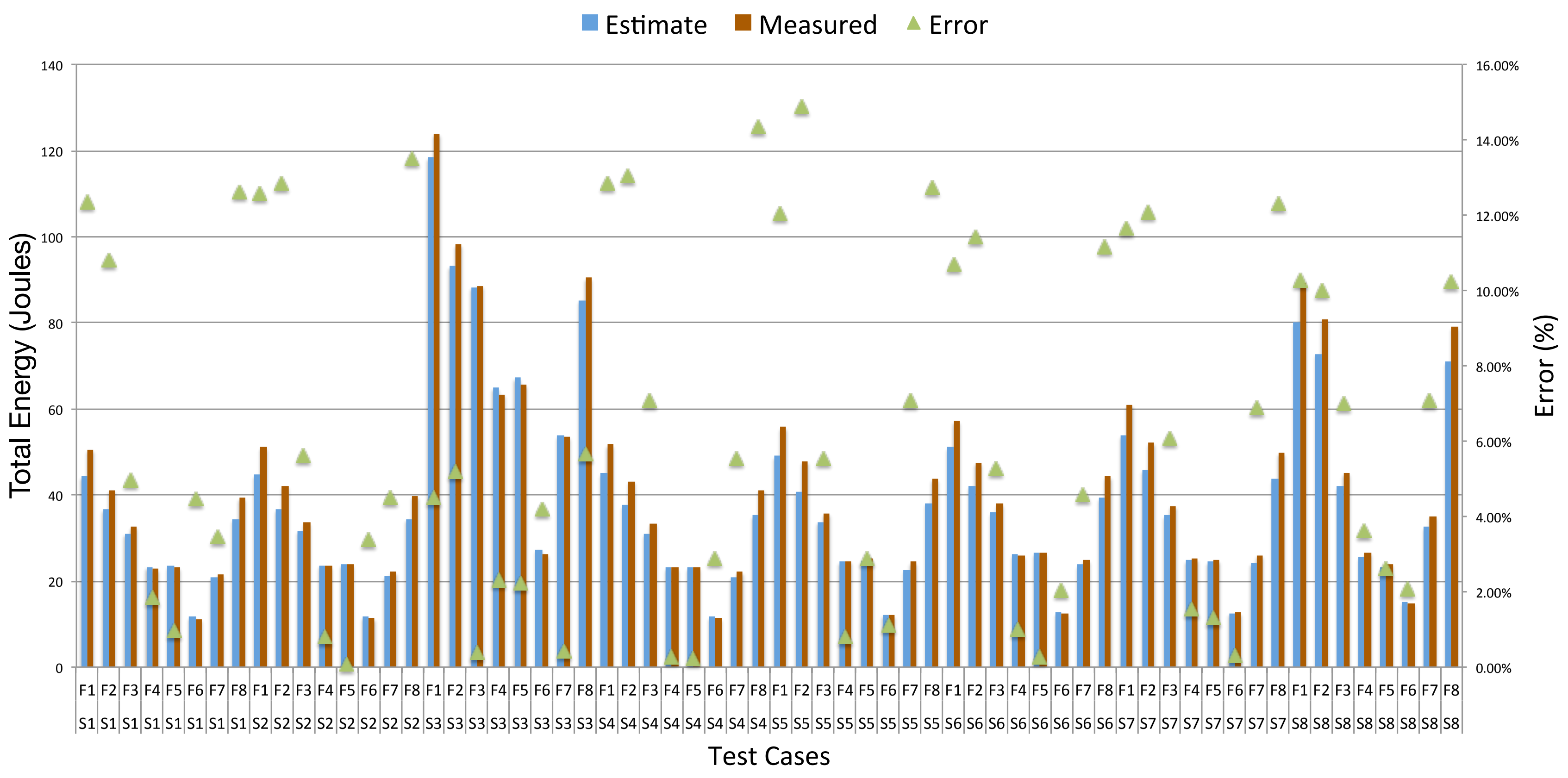


Figure 20: Comparison of estimate vs. measured energy for various test cases

and constant power – we see an interesting trend. As it can be seen in fig. 21, we see that a large portion of energy is "wasted" on constant power; it accounts for at least 74.39% of the total energy, and as much as 95.98%. Although we see no correlation between execution time and amount of energy wasted on constant power within the same DVFS setting, we see that it is particularly high for setting *S3* (table 10). At this setting, the core frequency is set to $180MHz$, and the memory frequency is set to the highest possible at $924MHz$. Although, in theory, the performance of FMM settings that favor data movement (low values of Q) should not be severely affected, this is not the case; compared to the best setting (*S1*), the speeddown is approximately 3 – 3.9× with no particular correlation to Q. We believe that this is because at such a low core frequency of $180MHz$, instructions can't issue quickly enough to saturate the memory system, and as a result, both compute and memory performance suffer. This teaches us that we need to be aware of more than just the impact of DVFS on energy costs, but also on the overall performance at the architecture level.

Figure 21 and fig. 19a also highlight two potential problems for our quest for algorithm redesign for energy efficiency. First, integer instructions make up approximately half of the total executed instructions. Integer operations are typically used for loops and address calculation and while necessary, they do not contribute to *useful* work. Unfortunately, there aren't any practical methods for reducing this integer overhead, as almost every application uses loops to iterate over data, and address calculation is fundamental to how modern architectures operate. On the slightly more optimistic side, in work–communication trade–off settings (§ 3.5), if the extra computation required to reduce data movement can be done locally (e.g., data decompression), it may reduce the overall impact of integer overhead.

While the FMM kernel used in our experiments is optimized close to the achievable peak [23], more than three quarters of the total power is being spent on constant power, and not on doing useful work (i.e., computation and data movement). As

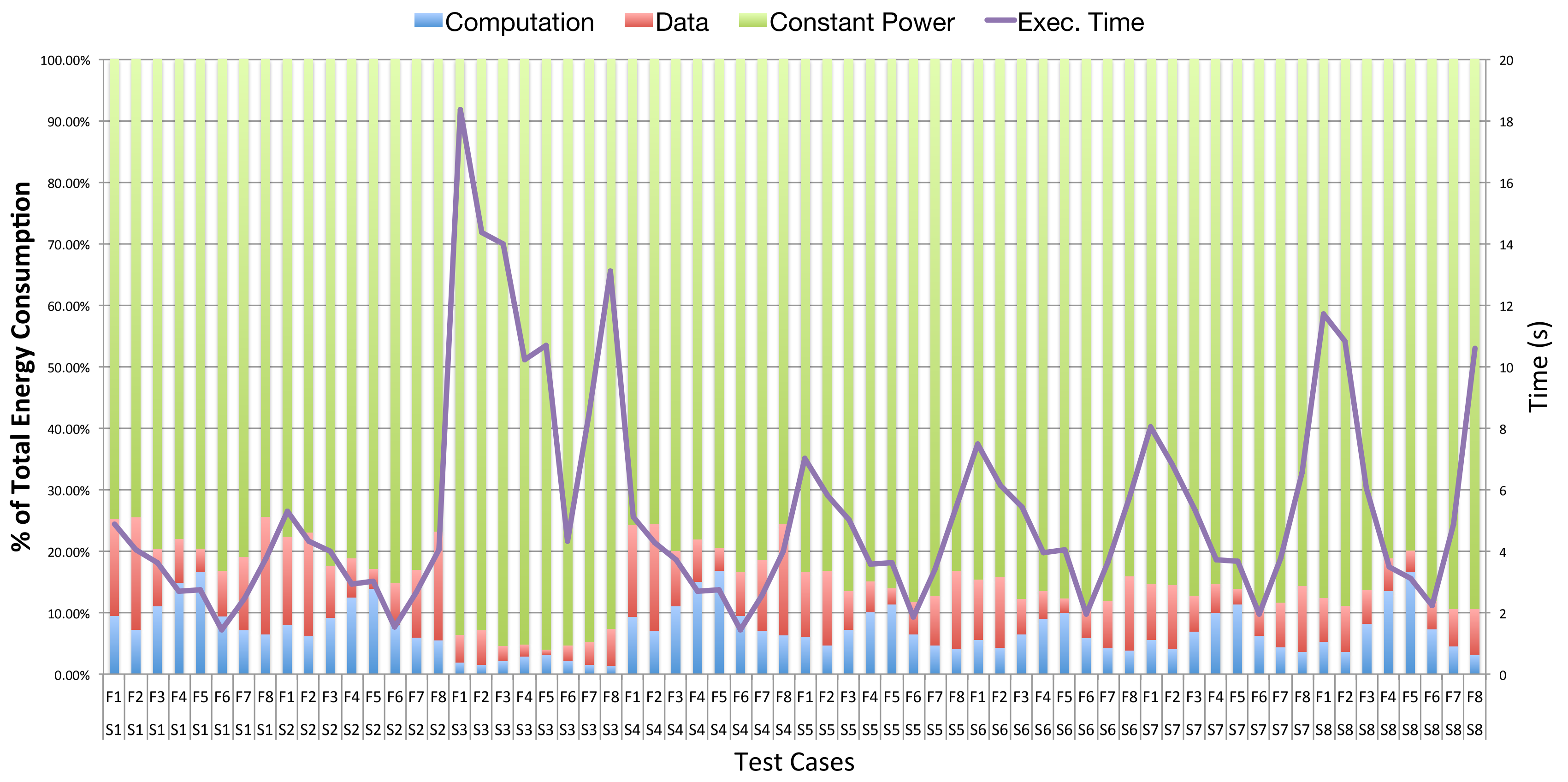


Figure 21: Breakdown of energy consumption by three types: energy spent on computation (incl. integer operations), data movement (from all levels of the memory hierarchy), and constant power (or overhead)

in the case of Amdahl's Law, where speedup is bound by the serial portion of the code, the amount of greenup (§ 3.5) we can achieve is severely limited by constant power. This issue is strongly correlated to system utilization. As demonstrated by our microbenchmarks, achieving full system utilization reduces the portion of energy spent on constant power to less than half, but requires issuing FMA instructions (or instructions that yields the highest FLOP/s performance), every cycle. Unfortunately, most applications have an instruction mix (and dependences) that do not conform to this behavior, so achieving full system utilization is *impossible* on most systems.

While this conclusion suggests that current systems are poor platforms for designing algorithms for energy efficiency, we believe that it also points us to the "right" direction. That is, future systems should focus on providing support for adapting performance to specific applications or kernels. For example, "dark silicon" [39, 40] could be used to implement specific functions or tasks; these could be turned on for applications that require them, while everything else that is redundant could be turned off. This way, every application could achieve full system utilization, as components that are turned off would not contribute to the overhead. An example of such a function or task could be the commonly used linear algebra operations [104].

6.4 Conclusion

We have demonstrated the utility of the energy roofline model by using it to accurately predict the energy consumption of a real and important scientific application running on a state–of–the–art mobile platform under a variety of voltage and frequency settings. Unlike other ideas that are focused more modeling the hardware to predict energy, or heuristics to reduce energy consumption at the system level, our model directly connects properties of an algorithm with architectural time and energy costs. This provides algorithm designers and performance tuners with information that they can use to consider and design algorithms for energy efficiency.

Unfortunately, our analysis of FMM on the Jetson K1 SoC has shown that reducing energy through algorithmic means, such as work–communication trade–off, for real applications may be fundamentally difficult due to the limitations imposed by the overhead of constant power. Only certain simple kernels, such as matrix multiplication or stencil calculation which consist mostly of FMA operations and large amounts of parallelism, can achieve high percentage of peak throughput on the Jetson K1's Kepler GPU core (and other GPU–like systems), and consequently, reduce the relative overhead of constant power by large enough amount to consider algorithmic trade–off scenarios. Others will be limited by the amount of available parallelism, instruction mix, and overall system utilization.

On the other hand, our analysis has suggested a potentially rewarding path for energy–efficiency research, in terms of both architecture and algorithm design. We believe that specialized units will become increasingly important for energy efficiency, especially when dark silicon becomes a reality. Utilizing dark silicon to implement highly–efficient special functional units will allow the system to "reconfigure" itself to better fit the application by dynamically enabling and disabling various units. This, we believe, will lead to better utilization and lower relative overhead in power, which will in turn allow algorithm designers to make changes to increase energy efficiency using our model, analysis, and ideas.

CHAPTER VII

CONCLUSION

The idea behind this thesis started from a simple question: what can we do to improve the energy efficiency of our algorithms and applications?

We address this challenge by starting with a simple model that expresses time and energy in terms of operations, concurrency, and data movement of algorithms. Our analysis of this model lead us to new ideas that relate energy with time, such as the notion of an energy–balance, analogous to the traditional time–balance of a system, and their difference – the time–energy balance gap.

The time–energy balance gap has important consequences for algorithms; today time–balance is greater than energy–balance, due largely to constant power, and consequently, race–to–halt strategies are the most reasonable first–order techniques to save energy. However, there is a strong possibility that this will change in the future; as was the case when industry transitioned from BJ to CMOS transistors – driven by our need to reduce wasted power – much effort is currently being invested in reducing this overhead, or constant power. These efforts include power gating techniques, drowsy caches, and new transistor technologies (e.g., Tri–gate, FinFET). When this happens, race–to–halt will most likely break, and open up new possibilities in algorithmic energy–efficiency research, such as our idea of work–communication trade–offs.

After establishing the basis for our new model, we validate it on real systems to see whether our theory and ideas translate to real world settings. We first create a highly–tuned and architecture–cognizant microbenchmark that can target a particular arithmetic intensity and stress the system to its maximum throughput performance.

Using a fine–grained power measurement tool, we collect energy measurements from our microbenchmark and use linear regression to derive the values of our model parameters – the cost of floating point operations and data movement. We found that the measured performance, energy and power closely matches the basic form of our model.

However, we also encountered a number of limitations; first, under certain arithmetic regimes, our microbenchmark underperformed on high–performance GPUs, which we hypothesized as being caused by power limitations; second, for a real application, our energy estimates were always significantly lower than measured energy, which lead us to the conclusion that energy is fundamentally different from time in that all actions must be accounted for – there are no overlaps in energy. To account for these shortcomings we extended our model to include the idea of a power "cap," and extended our microbenchmark to include other operations, such as integer and accessing data from different levels of the memory hierarchy.

Using our new model and benchmark suite, we extend our evaluation and analysis to server–, min–, and mobile–class systems that span a wide range of compute and power characteristics. We demonstrate how our model can be used to precisely and analytically compare different systems using concrete examples. For example, we show which systems are better than others in certain arithmetic regimes, show how performance may scale under power throttling or bounding scenarios.

As a first cut at bridging algorithm and architecture analysis for time and energy, we kept our model intentionally simple for reasons of clarity; however, dynamic voltage and frequency scaling is an important feature for energy–efficient computing in most modern system. Using a state–of–the–art CUDA–capable mobile SoC, the Jetson K1, as our testing platform, we incorporate and validate the impact of DVFS on our model. We show that our new model is accurate in predicting the change of energy cost of operations with frequency and voltage, and demonstrate that race–to–halt is not necessarily the best strategy for reducing energy consumption, at least in the context of highly–tuned microbenchmarks; for different arithmetic intensity values, our model was better at predicting the best DVFS setting for minimizing energy when compared to a strategy based on finding the DVFS setting that minimizes time.

Finally, we bring together all the work thus far to predict the energy consumption of a real, and important, scientific application, the fast multipole method (FMM), on the Jetson K1 SoC. We show that even with our simple model, we can accurately predict the energy consumption of FMM under different DVFS settings. Perhaps more importantly, we highlight several important issues regarding algorithmic energy efficiency. On modern systems and applications, there are a number of large overheads that hinder our efforts to improve energy efficiency; first, more than half of total computation operations are devoted to integer overhead of loops and address calculation; second, overhead, or constant power, accounts for more than three quarters of the total power, likely due to system *underutilization*, even for highly–tuned kernels like our FMM implementation.

While the outlook may be bleak for current systems and applications in terms of algorithmic energy efficiency, our research also highlights a path for the future that may help overcome these issues. We believe that focusing on specialized hardware functional units, such as those emphasized by proponents of dark silicon and linear algebra cores, will allow future processors to adapt to specific applications by enabling and disabling various units on the chip, and alleviate the problems of power overheads and underutilization.

REFERENCES

[1] Abe, Y., Sasaki, H., Kato, S., Inoue, K., Edahiro, M., and Peres, M., "Power and performance characterization and modeling of gpu-accelerated systems," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pp. 113–122, May 2014.

[2] AMD, *AMD Accelerated Parallel Processing OpenCL Programming Guide*. AMD, 2013.

[3] Amdahl, G. M., "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), (New York, NY, USA), pp. 483–485, ACM, 1967.

[4] Assayad, I., Girault, A., and Kalla, H., "Tradeoff exploration between reliability, power consumption, and execution time," in *Proceedings of the 30th International Conference on Computer Safety, Reliability, and Security*, SAFE-COMP'11, (Berlin, Heidelberg), pp. 437–451, Springer-Verlag, 2011.

[5] Aupy, G., Benoit, A., and Robert, Y., "Energy-aware scheduling under reliability and makespan constraints," Tech. Rep. October, INRIA, Grenoble, France, 2011.

[6] Awan, M. A. and Petters, S. M., "Enhanced Race-To-Halt: A Leakage-Aware Energy Management Approach for Dynamic Priority Systems," in *2011 23rd Euromicro Conference on Real-Time Systems*, pp. 92–101, IEEE, July 2011.

[7] Baghsorkhi, S. S., Delahaye, M., Patel, S. J., Gropp, W. D., and mei W. Hwu, W., "An adaptive performance modeling tool for GPU architectures," *SIGPLAN Not.*, vol. 45, pp. 105–114, Jan. 2010.

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

[9] Bekas, C. and Curioni, A., "A new energy-aware performance metric," in *Proceedings of the International Conference on Energy-Aware High-Performance Computing (EnA-HPC)*, (Hamburg, Germany), Sept. 2010.

[10] Bingham, B. D. and Greenstreet, M. R., "Computation with Energy-Time Trade-Offs: Models, Algorithms and Lower-Bounds," *2008 IEEE International Symposium on Parallel and Distributed Processing with Applications*, pp. 143–152, Dec. 2008.

[11] Birnbaum, J. and Williams, R. S., "Physics and the information revolution," *Physics Today*, vol. 53, no. 1, pp. 38–42, 2000.

[12] Blelloch, G. E., Maggs, B. M., and Miller, G. L., "The hidden cost of low bandwidth communication," in *Developing a Computer Science Agenda for High-Performance Computing* (Vishkin, U., ed.), pp. 22–25, New York, NY, USA: ACM, 1994.

[13] Blem, E., Menon, J., and Sankaralingam, K., "Power struggles: Revisiting the risc vs. cisc debate on contemporary arm and x86 architectures," in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pp. 1–12, Feb 2013.

[14] Board, J. and Schulten, K., "The fast multipole algorithm," *Computing in Science and Engineering*, vol. 2, pp. 76–79, January/February 2000.

[15] Boggs, D., Brown, G., Rozas, B., Tuck, N., and Venkatraman, K. S., "Nvidia's denver processor," *HotChips'14*, 2014.

[16] Bosilca, G., Ltaief, H., and Dongarra, J., "Power profiling of Cholesky and QR factorizations on distributed memory systems," *Computer Science - Research and Development*, pp. 1–9, 2012.

[17] Burtscher, M., Zecena, I., and Zong, Z., "Measuring gpu power with the k20 built-in sensor," in *Proceedings of Workshop on General Purpose Processing Using GPUs*, GPGPU-7, (New York, NY, USA), pp. 28:28–28:36, ACM, 2014.

[18] Butts, J. A. and Shaw, D. E., "Anton 2: A 2nd-generation asic for molecular dynamics simulation," *HotChips'14*, 2014.

[19] Chakraborty, K., "A case for an over-provisioned multicore system: Energy efficient processing of multithreaded programs," tech. rep., University of Wisconsin Madison, 2007.

[20] Chandramowlishwaran, A., Choi, J. W., Madduri, K., and Vuduc, R., "Towards a communication optimal fast multipole method and its implications for exascale," in *Proc. ACM Symp. Parallel Algorithms and Architectures (SPAA)*, (Pittsburgh, PA, USA), June 2012. Brief announcement.

[21] Chen, D. and Plemmons, R. J., "Nonnegativity constraints in numerical analysis," in *in A. Bultheel and R. Cools (Eds.), Symposium on the Birth of Numerical Analysis, World Scientific*, Press, 2009.

[22] Cheng, K.-T. and Wang, Y.-C., "Using mobile gpu for general-purpose computing; a case study of face recognition on smartphones," in *VLSI Design, Automation and Test (VLSI-DAT), 2011 International Symposium on*, pp. 1–4, April 2011.

[23] Choi, J., Chandramowlishwaran, A., Madduri, K., and Vuduc, R., "A cpu: Gpu hybrid implementation and model-driven scheduling of the fast multipole method," in *Proceedings of Workshop on General Purpose Processing Using GPUs*, GPGPU-7, (New York, NY, USA), pp. 64:64–64:71, ACM, 2014.

[24] Choi, J., Dukhan, M., Liu, X., and Vuduc, R., "Algorithmic time, energy, and power on candidate hpc compute building blocks," in *In Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium (IPDPS 14)*, 2014.

[25] Choi, J. and Vuduc, R., "A roofline model of energy," Tech. Rep. GT-CSE-2012-01, Georgia Institute of Technology, December 2012.

[26] Choi, J., Vuduc, R., Fowler, R., and Bendard, D., "A roofline model of energy," in *In Proceedings of the 27th IEEE International Parallel and Distributed Processing Symposium (IPDPS 13)*, 2013.

[27] Curtis-Maury, M., Dzierwa, J., Antonopoulos, C. D., and Nikolopoulos, D. S., "Online power-performance adaptation of multi-threaded programs using hardware event-based prediction," in *Proceedings of the 20th Annual International Conference on Supercomputing*, ICS '06, (New York, NY, USA), pp. 157–166, ACM, 2006.

[28] Curtis-Maury, M., Shah, A., Blagojevic, F., Nikolopoulos, D. S., de Supinski, B. R., and Schulz, M., "Prediction models for multi-dimensional power-performance optimization on many cores," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, (New York, NY, USA), pp. 250–259, ACM, 2008.

[29] Czechowski, K. and Vuduc, R., "A theoretical framework for algorithm-architecture co-design," in *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pp. 791–802, 2013.

[30] Czechowski, K., Battaglino, C., McClanahan, C., Iyer, K., Yeung, P.-K., and Vuduc, R., "On the communication complexity of 3D FFTs and its implications for Exascale," in *Proceedings of the 26th ACM international conference on Supercomputing - ICS '12*, (San Servolo Island, Venice, Italy), p. 205, ACM Press, 2012.

[31] Czechowski, K., Lee, V. W., Grochowski, E., Ronen, R., Singhal, R., Dubey, P., and Vuduc, R., "Improving the energy efficiency of big cores," in *Proc. ACM/IEEE Int'l. Symp. on Computer Architecture (ISCA)*, (Minneapolis, MN, USA), June 2014. (extitaccepted).

[32] Czechowski, K., Battaglino, C., Mcclanahan, C., Chandramowlishwaran, A., and Vuduc, R., "Balance principles for algorithm-architecture co-design," in *USENIX Wkshp. Hot Topics in Parallelism (HotPar)*, (Berkeley, CA, USA), pp. 1–5, Usenix Association, 2011.

[33] Demmel, J., Gearhart, A., Lipshitz, B., and Schwartz, O., "Perfect strong scaling using no additional energy," in *In Proceedings of the 27th IEEE International Parallel and Distributed Processing Symposium (IPDPS 13)*, 2013.

[34] Demmel, J., Gearhart, A., Schwartz, O., and Lipshitz, B., "Perfect strong scaling using no additional energy," Tech. Rep. UCB/EECS-2012-126, EECS Department, University of California, Berkeley, May 2012.

[35] Diop, T., Jerger, N. E., and Anderson, J., "Power modeling for heterogeneous processors," in *Proceedings of Workshop on General Purpose Processing Using GPUs*, GPGPU-7, (New York, NY, USA), pp. 90:90–90:98, ACM, 2014.

[36] Dongarra, J., Ltaief, H., Luszczek, P., and Weaver, V. M., "Energy footprint of advanced dense numerical linear algebra using tile algorithms on multicore architecture," in *The 2nd International Conference on Cloud and Green Computing*, Nov. 2012.

[37] Dressler, S. and Steinke, T., "Energy consumption of cuda kernels with varying thread topology," *Computer Science - Research and Development*, vol. 29, no. 2, pp. 113–121, 2014.

[38] Esmaeilzadeh, H., Sampson, A., Ceze, L., and Burger, D., "Neural acceleration for general-purpose approximate programs," in *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*, pp. 449–460, Dec 2012.

[39] Esmaeilzadeh, H., Blem, E., St. Amant, R., Sankaralingam, K., and Burger, D., "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, (New York, NY, USA), pp. 365–376, ACM, 2011.

[40] Esmaeilzadeh, H., Blem, E., St. Amant, R., Sankaralingam, K., and Burger, D., "Power limitations and dark silicon challenge the future of multicore," *ACM Trans. Comput. Syst.*, vol. 30, pp. 11:1–11:27, Aug. 2012.

[41] Esmaeilzadeh, H., Cao, T., Yang, X., Blackburn, S. M., and McKinley, K. S., "Looking back and looking forward: power, performance, and upheaval," *Commun. ACM*, vol. 55, pp. 105–114, July 2012.

[42] et al., J. D., "The international exascale software project roadmap 1."

[43] Feng, X., Ge, R., and Cameron, K., "Power and energy profiling of scientific applications on distributed systems," in *Proceedings of the 19th IEEE*

International Parallel and Distributed Processing Symposium (IPDPS), p. 34, april 2005.

[44] Freeh, V. W., Lowenthal, D. K., Pan, F., Kappiah, N., Springer, R., Rountree, B. L., and Femal, M. E., "Analyzing the energy-time trade-off in high-performance computing applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, pp. 835–848, June 2007.

[45] Fürlinger, K., Klausecker, C., and Kranzlmüller, D., "Towards energy efficient parallel computing on consumer electronic devices," in *Proceedings of the First international conference on Information and communication on technology for the fight against global warming*, ICT-GLOW'11, (Berlin, Heidelberg), pp. 1–9, Springer-Verlag, 2011.

[46] Ge, R. and Cameron, K., "Power-aware speedup," in *In Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS 07)*, 2007.

[47] Ge, R., Feng, X., and Cameron, K. W., "Performance-constrained distributed dvs scheduling for scientific applications on power-aware clusters," in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, SC '05, (Washington, DC, USA), pp. 34–, IEEE Computer Society, 2005.

[48] Ge, R., Feng, X., Song, S., Chang, H.-C., Li, D., and Cameron, K., "Powerpack: Energy profiling and analysis of high-performance systems and applications," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 21, pp. 658–671, May 2010.

[49] Gearhart, A., *Bounds on the Energy Consumption of Computational Kernels*. PhD thesis, EECS Department, University of California, Berkeley, Oct 2014.

[50] Geisser, S., *Predictive inference*, vol. 55. CRC Press, 1993.

[51] Gonzalez, R. and Horowitz, M., "Energy dissipation in general purpose microprocessors," *IEEE J. Solid-State Circuits*, vol. 31, pp. 1277–1284, sep 1996.

[52] Grasso, I., Radojkovic, P., Rajovic, N., Gelado, I., and Ramirez, A., "Energy efficient hpc on embedded socs: Optimization techniques for mali gpu," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pp. 123–132, May 2014.

[53] Green500, "Green500 list," 2012.

[54] Greengard, L. and Rokhlin, V., "A fast algorithm for particle simulations," *J. Comp. Phys.*, vol. 73, pp. 325–348, 1987.

[55] Gupta, K. and Owens, J. D., “Compute & memory optimizations for high-quality speech recognition on low-end gpu processors,” in *Proceedings of the 2011 18th International Conference on High Performance Computing*, HIPC ’11, (Washington, DC, USA), pp. 1–10, IEEE Computer Society, 2011.

[56] Hackenberg, D., Schöne, R., Molka, D., Müller, M., and Knüpfer, A., “Quantifying power consumption variations of hpc systems using spec mpi benchmarks,” *Computer Science - Research and Development*, vol. 25, no. 3-4, pp. 155–163, 2010.

[57] Hager, G., Treibig, J., Habich, J., and Wellein, G., “Exploring performance and power properties of modern multicore chips via simple machine models,” *CoRR*, vol. abs/1208.2908, 2012.

[58] Hameed, R., Qadeer, W., Wachs, M., Azizi, O., Solomatnikov, A., Lee, B. C., Richardson, S., Kozyrakis, C., and Horowitz, M., “Understanding sources of inefficiency in general-purpose chips,” in *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA ’10, (New York, NY, USA), pp. 37–47, ACM, 2010.

[59] Hill, M. D. and Marty, M. R., “Amdahl’s Law in the Multicore Era,” *Computer*, vol. 41, pp. 33–38, July 2008.

[60] Hillis, W. D., “Balancing a Design,” *IEEE Spectrum*, 1987.

[61] Hockney, R. W. and Curington, I. J., “f1/2: A parameter to characterize memory and communication bottlenecks,” *Parallel Computing*, vol. 10, pp. 277–286, May 1989.

[62] Hong, S. and Kim, H., “An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness,” in *Proceedings of the 36th annual International Symposium on Computer Architecture (ISCA)*, (New York, NY, USA), pp. 152–163, ACM, 2009.

[63] Hong, S. and Kim, H., “An integrated GPU power and performance model,” *SIGARCH Comput. Archit. News*, vol. 38, pp. 280–289, June 2010.

[64] Horowitz, M., Alon, E., Patil, D., Naffziger, S., Kumar, R., and Bernstein, K., “Scaling, power, and the future of cmos,” in *Electron Devices Meeting, 2005. IEDM Technical Digest. IEEE International*, pp. 7 pp. –15, dec. 2005.

[65] Huang, W., Rajamani, K., Stan, M., and Skadron, K., “Scaling with design constraints: Predicting the future of big chips,” *Micro, IEEE*, vol. 31, pp. 16 –29, july-aug. 2011.

[66] Intel, *Intel 64 and IA-32 Architectures Software Developer’s Manual.* Intel, 2015.

[67] ITRS, "International technology roadmap for semiconductors 2005 edition," tech. rep., ITRS, 2005.

[68] Jain, R., Molnar, D., and Ramzan, Z., "Towards a model of energy complexity for algorithms," in *IEEE Wireless Communications and Networking Conference, 2005*, pp. 1884–1890, IEEE, 2005.

[69] Janapa Reddi, V., Lee, B. C., Chilimbi, T., and Vaid, K., "Web search using mobile cores: quantifying and mitigating the price of efficiency," *SIGARCH Comput. Archit. News*, vol. 38, pp. 314–325, June 2010.

[70] Jia-Wei, H. and Kung, H. T., "I/O complexity: The red-blue pebble game," in *Proceedings of the thirteenth annual ACM symposium on Theory of computing - STOC '81*, (New York, New York, USA), pp. 326–333, ACM Press, May 1981.

[71] Jiao, Y., Lin, H., Balaji, P., and Feng, W., "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)*, pp. 221 –228, dec. 2010.

[72] Kaxiras, S. and Martonosi, M., *Computer Architecture Techniques for Power-Efficiency.* Morgan and Claypool Publishers, 1st ed., 2008.

[73] Keckler, S., Dally, W., Khailany, B., Garland, M., and Glasco, D., "Gpus and the future of parallel computing," *Micro, IEEE*, vol. 31, pp. 7 –17, sept.-oct. 2011.

[74] Kerbyson, D., Vishnu, A., and Barker, K., "Energy templates: Exploiting application information to save energy," in *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pp. 225 –233, sept. 2011.

[75] Kestor, G., Gioiosa, R., Kerbyson, D., and Hoisie, A., "Quantifying the energy cost of data movement in scientific applications," in *Workload Characterization (IISWC), 2013 IEEE International Symposium on*, pp. 56–65, Sept 2013.

[76] Kim, N., Austin, T., Baauw, D., Mudge, T., Flautner, K., Hu, J., Irwin, M., Kandemir, M., and Narayanan, V., "Leakage current: Moore's law meets static power," *Computer*, vol. 36, pp. 68–75, Dec 2003.

[77] Kolmogorov, A., "Sulla determinazione empirica di una legge di distribuzione," *Giornale dell'Instituto Italiano degli Attuari*, vol. 4, pp. 83–91, 1933.

[78] Kolodzey, J., "Cray-1 computer technology," *Components, Hybrids, and Manufacturing Technology, IEEE Transactions on*, vol. 4, pp. 181 – 186, jun 1981.

[79] Korthikanti, V. A. and Agha, G., "Analysis of Parallel Algorithms for Energy Conservation in Scalable Multicore Architectures," in *2009 International Conference on Parallel Processing*, (Vienna, Austria), pp. 212–219, IEEE, Sept. 2009.

[80] Kung, H. T., "Memory requirements for balanced computer architectures," in *Proceedings of the ACM Int'l. Symp. Computer Architecture (ISCA)*, (Tokyo, Japan), 1986.

[81] Leng, J., Hetherington, T., ElTantawy, A., Gilani, S., Kim, N. S., Aamodt, T. M., and Reddi, V. J., "Gpuwattch: Enabling energy optimizations in gpgpus," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, (New York, NY, USA), pp. 487–498, ACM, 2013.

[82] Li, S., Ahn, J. H., Strong, R. D., Brockman, J. B., Tullsen, D. M., and Jouppi, N. P., "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, (New York, NY, USA), pp. 469–480, ACM, 2009.

[83] Libuschewski, P., Siedhoff, D., and Weichert, F., "Energy-aware design space exploration for gpgpus," *Computer Science - Research and Development*, vol. 29, no. 3-4, pp. 171–176, 2014.

[84] Lively, C., Taylor, V., Wu, X., Chang, H.-C., Su, C.-Y., Cameron, K., Moore, S., and Terpstra, D., "E-amom: an energy-aware modeling and optimization methodology for scientific applications," *Computer Science - Research and Development*, vol. 29, no. 3-4, pp. 197–210, 2014.

[85] Lively, C., Wu, X., Taylor, V., Moore, S., Chang, H.-C., Su, C.-Y., and Cameron, K., "Power-aware predictive models of hybrid (MPI/OpenMP) scientific applications on multicore systems," *Computer Science - Research and Development*, pp. 1–9, 2011. 10.1007/s00450-011-0190-0.

[86] Ltaief, H., Luszczek, P., and Dongarra, J., "Profiling high performance dense linear algebra algorithms on multicore architectures for power and energy efficiency," *Computer Science - Research and Development*, vol. 27, no. 4, pp. 277–287, 2012.

[87] Maghazeh, A., Bordoloi, U., Eles, P., and Peng, Z., "General purpose computing on low-power embedded gpus: Has it come of age?," in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), 2013 International Conference on*, pp. 1–10, July 2013.

[88] Martin, A. J., "Towards an energy complexity of computation," *Information Processing Letters*, vol. 77, pp. 181–187, Feb. 2001.

[89] McCalpin, J., “Memory Bandwidth and Machine Balance in High Performance Computers,” *IEEE Technical Committee on Computer Architecture (TCCA) Newsletter*, Dec. 1995.

[90] Minartz, T., Ludwig, T., Knobloch, M., and Mohr, B., “Managing hardware power saving modes for high performance computing,” in *Green Computing Conference and Workshops (IGCC), 2011 International*, pp. 1–8, July 2011.

[91] Minartz, T., Kunkel, J., and Ludwig, T., “Simulation of power consumption of energy efficient cluster hardware,” *Computer Science - Research and Development*, vol. 25, no. 3-4, pp. 165–175, 2010.

[92] Mohiyuddin, M., Murphy, M., Oliker, L., Shalf, J., Wawrzynek, J., and Williams, S., “A design methodology for domain-optimized power-efficient supercomputing,” in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC ’09, (New York, NY, USA), pp. 12:1–12:12, ACM, 2009.

[93] Molka, D., Hackenberg, D., Schone, R., and Muller, M., “Characterizing the energy consumption of data transfers and arithmetic operations on x86-64 processors,” in *Green Computing Conference, 2010 International*, pp. 123–133, Aug 2010.

[94] Momose, S., “Sx-ace processor: Nec’s brand-new vector processor,” *HotChips’14*, 2014.

[95] Mu, S., Wang, C., Liu, M., Li, D., Zhu, M., Chen, X., Xie, X., and Deng, Y., “Evaluating the potential of graphics processors for high performance embedded computing,” in *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pp. 1–6, March 2011.

[96] Nagasaka, H., Maruyama, N., Nukada, A., Endo, T., and Matsuoka, S., “Statistical power modeling of gpu kernels using performance counters,” in *In Proceedings of International Green Computing Conference*, 2010.

[97] Nowatzki, T., Menon, J., Ho, C.-H., and Sankaralingam, K., “gem5, gpgpusim, mcpat, gpuwattch, ¡your favorite simulator here¿ considered harmful,” in *Proceedings of 11th Annual Workshop on Duplicating, Deconstructing and Debunking*, 2014.

[98] NVIDIA, *Kepler Compute Architecture Whitepaper.*

[99] NVIDIA, *NVIDIA CUDA C Programming Guide.* NVIDIA, 2012.

[100] NVIDIA, “NVML API Reference Manual,” 2012.

[101] NVIDIA, *Jetson TK1 Technical Brief*, 2014.

[102] O'Connor, J., Hanke, T., Barkatullah, J., and Lewelling, R., "Goldstrike 1: A 1st generation cryptocurrency processor for bitcoin mining," *HotChips'14*, 2014.

[103] Patki, T., Lowenthal, D. K., Rountree, B., Schulz, M., and de Supinski, B. R., "Exploring hardware overprovisioning in power-constrained, high performance computing," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, (New York, NY, USA), pp. 173–182, ACM, 2013.

[104] Pedram, A., van de Geijn, R., and Gerstlauer, A., "Codesign tradeoffs for high-performance, low-power linear algebra architectures," *Computers, IEEE Transactions on*, vol. 61, no. 12, pp. 1724–1736, 2012.

[105] Pedram, A., Gilani, S. Z., Kim, N. S., van de Geijn, R. A., Schulte, M. J., and Gerstlauer, A., "A linear algebra core design for efficient level-3 blas," in *ASAP*, pp. 149–152, 2012.

[106] Rajovic, N., Carpenter, P. M., Gelado, I., Puzovic, N., Ramirez, A., and Valero, M., "Supercomputing with commodity cpus: Are mobile socs ready for hpc?," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, (New York, NY, USA), pp. 40:1–40:12, ACM, 2013.

[107] Rajovic, N., Rico, A., Vipond, J., Gelado, I., Puzovic, N., and Ramirez, A., "Experiences with mobile processors for energy efficient hpc," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pp. 464–468, March 2013.

[108] Rajovic, N., Vilanova, L., Villavieja, C., Puzovic, N., and Ramirez, A., "The low power architecture approach towards exascale computing," *Journal of Computational Science*, no. 0, pp. –, 2013.

[109] Rotem, E., Naveh, A., Rajwan, D., Ananthakrishnan, A., and Weissmann, E., "Power-management architecture of the intel microarchitecture code-named sandy bridge," *IEEE Micro*, vol. 32, pp. 20–27, March-April 2012.

[110] Rountree, B., Ahn, D. H., de Supinski, B. R., Lowenthal, D. K., and Schulz, M., "Beyond DVFS: A First Look at Performance under a Hardware-Enforced Power Bound," *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pp. 947–953, May 2012.

[111] Schubert, S., Kostic, D., Zwaenepoel, W., and Shin, K., "Profiling software for energy consumption," in *Green Computing and Communications (GreenCom), 2012 IEEE International Conference on*, pp. 515–522, 2012.

[112] Sharma, S., Hsu, C.-H., and Feng, W.-C., "Making a case for a Green500 list," in *20th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, April 2006.

[113] Singhal, N., Park, I. K., and Cho, S., "Implementation and optimization of image processing algorithms on handheld gpu," in *Image Processing (ICIP), 2010 17th IEEE International Conference on*, pp. 4481–4484, Sept 2010.

[114] Song, S., Grove, M., and Cameron, K. W., "An iso-energy-efficient approach to scalable system power-performance optimization," in *Proceedings of the 2011 IEEE International Conference on Cluster Computing*, CLUSTER '11, (Washington, DC, USA), pp. 262–271, IEEE Computer Society, 2011.

[115] Song, S., Su, C.-y., Rountree, B., and Cameron, K. W., "A simplified and accurate model of power-performance efficiency on emergent gpu architectures," in *In Proceedings of the 27th IEEE International Parallel and Distributed Processing Symposium (IPDPS 13)*, 2013.

[116] Stanisic, L., Videau, B., Cronsioe, J., Degomme, A., Marangozova-Martin, V., Legrand, A., and Mehaut, J.-F., "Performance analysis of hpc applications on low-power embedded platforms," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pp. 475–480, March 2013.

[117] Stanley-Marbell, P. and Cabezas, V., "Performance, power, and thermal analysis of low-power processors for scale-out systems," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pp. 863–870, 2011.

[118] Su, C., Li, D., Nikolopoulos, D., Cameron, K., de Supinski, B., and Leon, E., "Model-based, memory-centric performance and power optimization on numa multiprocessors," in *IEEE International Symposium on Workload Characterization*, Nov. 2012.

[119] Subramaniam, B. and Feng, W.-C., "Statistical power and performance modeling for optimizing the energy efficiency of scientific computing," in *Proceedings of the 2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, GREENCOM-CPSCOM '10, (Washington, DC, USA), pp. 139–146, IEEE Computer Society, 2010.

[120] Subramaniam, B. and Feng, W.-C., "The Green Index: A metric for evaluating system-wide energy efficiency in HPC systems," in *8th IEEE Workshop on High-Performance, Power-Aware Computing (HPPAC)*, (Shanghai, China), May 2012.

[121] Tyagi, A., "Energy-Time Trade-offs in VLSI Computations," in *Foundations of Software Technology and Theoretical Computer Science*, vol. LNCS 405, pp. 301–311, 1989.

[122] Ubal, R., Jang, B., Mistry, P., Schaa, D., and Kaeli, D., "Multi2sim: A simulation framework for cpu-gpu computing," in *Proceedings of the 21st*

International Conference on Parallel Architectures and Compilation Techniques, PACT '12, (New York, NY, USA), pp. 335–344, ACM, 2012.

[123] Vuduc, R. and Czechowski, K., "What GPU Computing Means for High-End Systems," *IEEE Micro*, vol. 31, pp. 74–78, July 2011.

[124] Wang, G., Xiong, Y., Yun, J., and Cavallaro, J., "Accelerating computer vision algorithms using opencl framework on the mobile gpu - a case study," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pp. 2629–2633, May 2013.

[125] Wang, Y.-C. and Cheng, K.-T., "Energy-optimized mapping of application to smartphone platform; a case study of mobile face recognition," in *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*, pp. 84–89, June 2011.

[126] Williams, S., Waterman, A., and Patterson, D., "Roofline: an insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, pp. 65–76, Apr. 2009.

[127] Wilton, S. J. E. and Jouppi, N. P., "Cacti: An enhanced cache access and cycle time model," *IEEE Journal of Solid-State Circuits*, vol. 31, pp. 677–688, 1996.

[128] Woo, D. H. and Lee, H.-H. S., "Extending Amdahl's Law for energy-efficient computing in the many-core era," *IEEE Computer*, vol. 41, pp. 24–31, Dec. 2008.

[129] Ying, L., Biros, G., Zorin, D., and Langston, H., "A new parallel kernel-independent fast multipole method," in *Proc. ACM/IEEE Conf. Supercomputing (SC)*, (Phoenix, AZ, USA), November 2003.

[130] Ying, L., Zorin, D., and Biros, G., "A kernel-independent adaptive fast multipole method in two and three dimensions," *J. Comp. Phys.*, vol. 196, pp. 591–626, May 2004.

[131] Yokota, R. and Barba, L. A., "A tuned and scalable fast multipole method as a preeminent algorithm for exascale systems," *Int. J. High-perf. Comput.*, 2011.

[132] Zhang, Y. and Owens, J. D., "A quantitative performance analysis model for gpu architectures," in *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, (Washington, DC, USA), pp. 382–393, IEEE Computer Society, 2011.

[133] Zidenberg, T., Keslassy, I., and Weiser, U., "Multi-Amdahl: How Should I Divide My Heterogeneous Chip?," *IEEE Computer Architecture Letters*, pp. 1–4, 2012.