ARCHON — A FRAMEWORK FOR DYNAMICALLY-TUNED CPU-GPU HYBRIDIZATION

By

KYLE RYAN SIEHL

A thesis submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

WASHINGTON STATE UNIVERSITY
School of Engineering and Computer Science, Vancouver

DECEMBER 2016

To the Faculty of Washington State University:

The members of the Committee appointed to examine the thesis of KYLE RYAN SIEHL find it satisfactory and recommend that it be accepted.

Dr. Xinghui Zhao, Ph.D., Chair

Dr. Scott Wallace, Ph.D.

Dr. Wayne Cochran, Ph.D.

ACKNOWLEDGMENTS

This work owes a great debt to all the faculty of WSU Vancouver. My adviser, Dr. Xinghui Zhao, was of course extremely helpful, and supplied much of the initial motivation for the topic; but every person on the thesis defense committee has been instrumental to this work's completion in at least one way. Dr. Wayne Cochran inspired much of my initial interest in GPU computation, and Dr. Scott Wallace's focus in AI is largely responsible for the selection of experimental applications. All of this is aside from the instructional value and support I have received from each of these people, which is substantial.

My family has supported me over the years, but more than that made me what I am, encouraging a five-year-old child in his perhaps ill-thought-out plan to "be a scientist". My mother in particular insisted I learn to type at an early age, the value of which is not easily overstated.

Lastly, my friends (many of them students here) have kept me sane over the years.

To all of these people, I give thanks.

ARCHON — A FRAMEWORK FOR DYNAMICALLY-TUNED CPU-GPU HYBRIDIZATION

Abstract

by Kyle Ryan Siehl, M.S.
Washington State University
May 2016

Chair: Dr. Xinghui Zhao

Graphics Processing Units (GPUs) have recently become widely used in general purpose computing, aiming for improving the performance of applications. However, this performance gain often comes with higher power consumption. In this work, we present Archon, a framework for power-aware CPU-GPU hybridization. Archon takes user's programs as input, automatically distributes the workload between the CPU and the GPU, and dynamically tunes the distribution ratio at runtime for an energy-efficient execution.

To evaluate the effectiveness of Archon, experiments have been carried out using a variety of applications. Several of these experiments involve computer vision algorithms, which often perform reasonably well on both the CPU and the GPU. We have also evaluated Archon with matrix multiplication, as a simpler, computationally-expensive example outside the field of computer vision. The results of these experiments show us that, in many cases, Archon can achieve substantial improvements in both performance and energy consumption, with little extra effort from client programmers.

TABLE OF CONTENTS

ACKNOWLEDGMENTS **iii**

ABSTRACT **iv**

LIST OF TABLES **viii**

LIST OF FIGURES **ix**

1 Introduction **1**

1.1 Background 1

1.2 Machine Architecture 3

1.2.1 CPU 4

1.2.2 GPU 4

1.3 Archon 5

1.4 Thesis Statement 6

1.5 Thesis Organization 6

2 Related Works **7**

3 Approach **11**

3.1 Architecture 12

3.2 Workload Balancing . . . 13
3.2.1 Optimizing Execution Time . . . 13
3.2.2 Instrumentation . . . 15
3.3 Energy Prediction . . . 15
3.3.1 Static Model . . . 15
3.3.2 Dynamic Model . . . 16
3.4 Implementation . . . 17
3.4.1 Configuration . . . 17
3.4.2 API . . . 18

4 Evaluation **21**
4.1 Overview . . . 21
4.1.1 Hardware Configuration . . . 21
4.1.2 Applications . . . 22
4.1.3 Experimental Design . . . 22
4.2 Image Similarity . . . 23
4.2.1 Algorithms . . . 23
4.2.2 Experimental Results . . . 26
4.3 Face Detection . . . 29
4.3.1 Algorithm . . . 30
4.3.2 Experimental Results . . . 30
4.4 Matrix Multiplication . . . 34
4.4.1 Single-core . . . 35
4.4.2 Multi-core . . . 39

5 Conclusion **43**
5.1 Future Works . . . 44

Bibliography **47**

List of Tables

4.1 PSNR — speedup and greenup 24
4.2 MSSIM — speedup and greenup 27
4.3 Viola-Jones face detection — speedup and greenup 32
4.4 Viola-Jones face detection — performance under load 34
4.5 Matrix multiplication — speedup and greenup 36
4.6 8-core matrix multiplication — speedup and greenup 40

List of Figures

1.1 CPU frequency by year 2

1.2 Heterogeneous machine architecture assumed by Archon 3

1.3 Overview of a modern multicore CPU 4

1.4 Overview of a GPU 5

3.1 Archon system diagram 12

3.2 Basic Archon workflow, with user code in red and Archon code in blue . . . 13

3.3 Interface to the Archon library 18

3.4 Example code using the Archon library 19

4.1 PSNR — variance by hybridization 25

4.2 PSNR — speedup and greenup 25

4.3 MSSIM — variance by hybridization 26

4.4 MSSIM — speedup and greenup 27

4.5 PSNR — energy model 28

4.6 MSSIM — energy model 29

4.7 Viola-Jones face detection — variance by hybridization 31

4.8 Viola-Jones face detection — speedup and greenup 31

4.9 Viola-Jones face detection — energy model 33

4.10 Viola-Jones face detection — hybridization under load 33

4.11 Matrix multiplication — variance by hybridization 35
4.12 Matrix multiplication — speedup and greenup 36
4.13 Matrix multiplication — energy breakdown — 4096×4096 37
4.14 Matrix multiplication — scalability 38
4.15 Matrix multiplication — energy model 38
4.16 8-core matrix multiplication — variance by hybridization 39
4.17 8-core matrix multiplication — speedup and greenup 40
4.18 8-core matrix multiplication — energy model 41
4.19 8-core matrix multiplication — hybridization under load 42

Chapter 1

Introduction

1.1 Background

Across the history of computing, Moore's law has governed the continual improvement of computer performance. Moore's law, roughly stated, claims that the number of transistors that can be placed in a circuit doubles every two years [1]. For much of the industry's history, Moore's law has enabled exponentially growth of computing speed.

Unfortunately, this steady drumbeat has been ground to a halt by the "power wall" [2]. The power wall is fundamentally caused by the fact that power consumption of chips increases nonlinearly with their clock frequency. The exact relationship between power consumption and clock frequency is complicated, but it is often simplified to the claim that power consumption of a chip is proportional to the cube of its clock frequency [3]. This explosion of power consumption also results in increased heat production, as heat is proportional to power usage. These factors have presented challenges in producing chips that run with clock rates greater than about 4 GHz, as can be seen in Figure 1.1 (which shows CPU frequency of available CPUs by year).

To continue improving performance in the face of the power wall, chip makers have largely

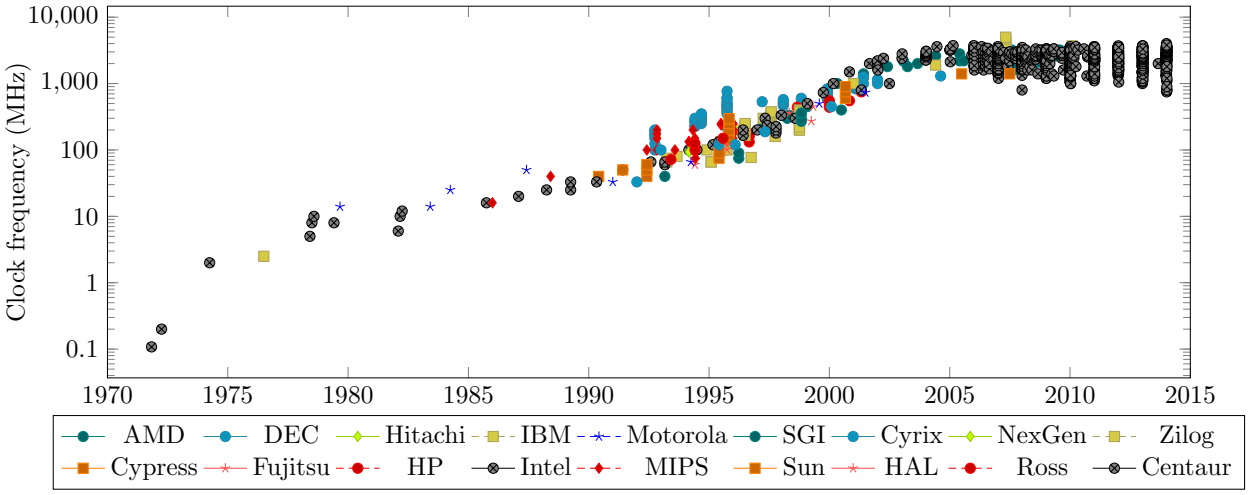


Figure 1.1: CPU frequency by year

turned to multicore chips. These can sidestep the power wall by not requiring faster clock rates to gain performance. Instead, multicore chips can improve performance by simply adding more cores. While this still requires more power, adding more cores only increases power consumption linearly, as opposed to the cubic factor obtained by increasing clock rates. This thus seems to be a more scalable method of improvement, though it complicates the design of hardware. Software is also complicated by multicore chips, as writing code to take advantage of multiple cores is commonly considered to be extremely difficult and error-prone. However, as increasing frequency does not seem to be a viable method of improving performance, these complications are inevitable in the efforts of making and utilizing faster processors.

The extreme end of this idea of adding cores, rather than increasing clock frequency, is exemplified by the graphics processing unit (GPU). While a more traditional CPU design will usually have between one and eight cores running at about 4 GHz, current consumer-facing GPUs may have as many as 2048 cores running as slowly as 1 GHz. These cores are much more limited than those of the traditional CPU, as they must share much of their execution state to keep costs down, in order to enable the kind of massive parallelism they provide.

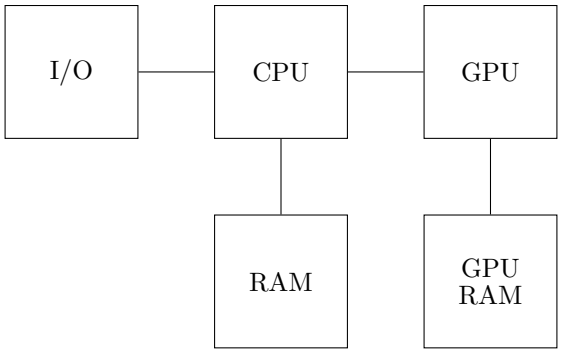


Figure 1.2: Heterogeneous machine architecture assumed by Archon

However, for many problems, these limitations are acceptable, and the performance benefits of accepting them can be enormous. An early example, for which the processors are named, is that of 3D graphics rendering; at its core, the problem consists of an extremely large number of four-by-four matrix multiplications (one for each vertex in the scene, typically) [4]. Each multiplication runs the exact same code, and so even with shared execution state, each vertex can still be processed independently.

1.2 Machine Architecture

In our work, we assume a heterogenous architecture, in which systems have both a CPU and a GPU, each with their own memory. This also represents the most common architecture for desktop systems at present. A highly simplified version of this architecture is detailed in 1.2. The most notable features of this architecture are the CPU's control of I/O and the GPU's separate memory.

The first consequence of this architecture is that the CPU's monopolization of access to IO means that the CPU must be "in charge" of the system's computation. This limitation (along with the GPU's unsuitability for general-purpose computation) relegates the GPU to being thought of as an "accelerator" onto which certain tasks are offloaded.

Additionally, because the GPU's memory is separate from the CPU's memory, data must

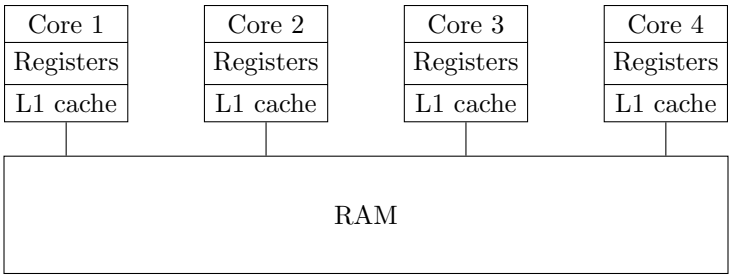


Figure 1.3: Overview of a modern multicore CPU

be transferred between the two processors in order to be acted on. This further relegates the GPU to doing large tasks to save CPU time.

1.2.1 CPU

The CPU (Figure 1.3) is the standard by which other processing methods are measured, and is the default assumed processor for most tasks. The most notable feature of the CPU is the strong independence between the different cores. Each core has its own execution state, and in most cases its own first-level cache.

This architecture has great strength in that it allows for disparate computation on each core. This can be very useful for multi-user systems, or for single-user systems with multi-tasking, as each core can handle independent applications. Additionally, single applications with separable subtasks can allocate different tasks to different cores. This is often useful in large applications, such as games, where physics calculations, game mechanics, and input processing may require completely different code to handle.

1.2.2 GPU

In contrast to the CPU, which is a more general-purpose device, the GPU (Figure 1.4) provides an architecture which is strongly suited to specific tasks. This is due to fundamental design differences. Where the cores of the CPU are independent, those of the GPU are

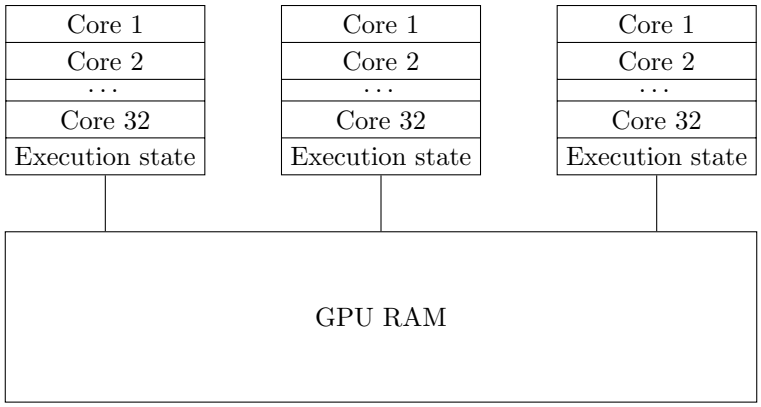


Figure 1.4: Overview of a GPU

strongly intertwined. On NVIDIA GPUs, cores are grouped into sets of 32, called "warps". These cores share much of their execution state, which means that the cores within a warp must all run the same instructions. For general-purpose computation, this will often result in low performance. However, in many cases, this limitation is acceptable, and the massive increase in core count can result in great performance benefits.

1.3 Archon

Archon is a library which enables programmers to write programs which simultaneously target CPU and GPU systems. It provides instrumentation to determine the efficiency of the disparate systems and load-balancing algorithms to split work between the two effectively.

To use Archon, the client programmer writes both CPU and GPU versions of their algorithm, and code to split the data between the two. Archon uses high-precision timers to provide instrumentation which the client programmer can wrap around their device-specific code. This instrumentation is then used to compute a split ratio, which is passed to the client program and used to split the data on the next iteration. By performing this computation, Archon can select a ratio which will result in optimal performance.

Archon is useful for client algorithms which are performant on both CPU and GPU

systems. For problems in which one of either the CPU or the GPU is vastly more suitable than the other, we will not expect hybridization to provide substantial benefit, or indeed any.

1.4 Thesis Statement

In this work, we aim to show that cooperative CPU-GPU hybridization is a viable method for improving computations' performance in some cases, and also for improving their energy efficiency. This hybridization can be instrumented at runtime with minimal effort from programmers.

1.5 Thesis Organization

In Chapter 2, we discuss some other efforts in the space of heterogeneous computing, and contrast their approaches with our own. Chapter 3 shows in detail how Archon works, both in terms of how, mathematically, Archon balances computation as well as how the client programmer interacts with Archon. Chapter 4 shows the results of our experiments, which describe both how much performance gain Archon can obtain and also how much energy savings (or losses, as the case may be) it receives in the meantime. Finally, in Chapter 5 we draw our conclusions about the viability of Archon as a method for both performance gain and energy saving.

Chapter 2

Related Works

Significant amounts of work have been carried out in analyzing, modeling and optimizing GPU energy consumption in a heterogeneous system. Hong *et al.* have proposed an integrated GPU power and performance model in [5]. This model accounts for power consumption of individual types of chips on the GPU, and even for such variables as temperature. The model is also more specific to the GPU, and would need some alteration for other types of processors. With the fine-grained modeling, the energy prediction of this approach is considerably more accurate than other approaches. However, it is not immediately clear that these predictions can be made at runtime, or without specialized hardware.

A contrasting approach to our naïve hybridization is proposed in [6], in which the authors describe a dynamical tuned push-relabel algorithm for the maximum flow problem on CPU-GPU-hybrid platforms. While this approach shows a similar ability to use both the CPU and GPU to get results faster than using either processor alone, it does it in a much more domain-specific way. Rather than splitting the work between the two processors, as we do, the authors work with only one processor at a time. However, the processors are running fundamentally different algorithms, which take advantage of the individual strengths of the two processors. The GPU performs a fairly simple massively-parallel algorithm, which is

good at performing work locally but can take a long time to find the global solution to the problem. The CPU, on the other hand, performs a more sophisticated algorithm which can perform global optimizations that the GPU cannot. By alternating between these two approaches, the authors get results that are as good or better than the individual algorithms would be. Although this approach provides good performance, it is much less general, only applying to the maximum flow problem in particular. Similar concepts could likely result in performance better than our naïve hybridization for other problems, but would require in each case a great deal of work and domain-specific knowledge.

The PEACH model [7] is a similar approach to Archon. Specifically, PEACH balances computations across CPU and GPU for optimizing performance and energy consumption. The performance model in PEACH is very similar to ours, though the energy model is slightly different, as it accounts for different factors. Additionally, this approach accounts for dynamic voltage scaling, something considered out-of-scope in our system. However, there is less discussion on how to build a framework which is usable for real-world systems, or of how to measure the variables necessary at runtime.

A more comprehensive survey on methods for analyzing and improving GPU energy efficiency can be found in [8]. Some more specific work can be found in GPUWattch [9], which attempts to model GPU energy usage at the microarchitectural level, by accounting for energy used by specific components of the GPU. Unlike the existing approaches, our work, Archon, aims for providing a generic framework in which users can run any applications. The performance and energy tuning is handled by the framework, and this process is transparent to users.

A great deal of work has been done in the area of power-aware scheduling, of which Archon is a part. An early work [10] from Xerox PARC provides a formulation of the problem, and algorithms for scheduling. Much later work focuses on such scheduling as applied to distributed systems; the survey paper [11] is a good starting point for research in this area.

Much of this work assumes an "offline scheduling" approach, in which problems arrive with known deadlines and computational costs. An example of this can be seen in [12], which discusses using both graph-theoretical and linear-programming-based methods to schedule tasks for minimal energy usage up-front (and continues to discuss some dynamic methods to help account for errors in the initial schedule). We also see more some more GPU-focused scheduling research in TimeGraph [13], though TimeGraph focuses only on performance for rendering applications, not on energy and not on general-purpose computation.

A similar approach of hybridizing a particular class of algorithms can be seen in [14]. However, the class of algorithms it applies itself to is those expressable via the MapReduce paradigm. The authors experiment with two different work-partitioning schemes; one, like ours, in which input data is split between the two processors, and another, in which the two processors perform the fundamentally different tasks of mapping and reducing data. Their schemes show speedups for most tested problems, though which scheme achieves the best speedup varies from problem to problem. The work also features notable hardware differences to ours; namely, the experiments were run on an AMD Fusion APU, a hybrid between a CPU and a GPU which shares memory between the two.

Several other works have attempted to hybridize CPU and GPU computation via an MPI-like paradigm, or sometimes via MPI itself. This can be seen initially in DCGN [15], which initiated the idea of implementing an MPI-esque API for hybridized CPU-GPU computation. Some later works in this vein [16][17] discuss MPI-ACC, which has much the same ideas as DCGN, but follows the MPI API more strictly in an effort to maintain backwards-compatibility and require less knowledge of GPU programming than DCGN.

A much more interesting approach to GPU programming can be found in GPUnet [18]. Much like our work, GPUnet attempts to subvert the master-slave paradigm of GPU programming, in which the GPU acts as an accelerator for the CPU; however, rather than attempt to make the two processors equals, GPUnet flips the situation on its head by putting

the GPU in charge of computation. By using special hardware to handle network IO, GPUnet allows programmers to use sockets to write servers that run directly on the GPU, with minimal CPU involvement (or, in the absence of such hardware, provides APIs which emulate this on the CPU). These socket-based servers can be used to implement GPU-based clusters with minimal reliance on CPUs. Sadly, work on GPUnet seems to have petered out shortly after the initial release.

The "greenup" [19] performance metric is applied extensively in our work. By analogy with speedup, greenup measures the energy consumption of an experimental algorithm versus some baseline. This notion, in combination with powerup and speedup, is used in [19] to describe behavior of algorithms in an energy-aware manner. We have elided the use of powerup as a metric, however, as it is redundant with a combination of speedup and greenup.

Chapter 3

Approach

The objective of the Archon library is to enable the workload of a computation to be intelligently divided and distributed across a CPU and a GPU. Traditional GPU computing, in which the entire computation is offloaded to the GPU and the CPU merely waits for the GPU to finish, wastes resources by leaving the CPU completely unutilized. This wastes time in many cases, and may also waste energy in some.

To address this challenge, we develop the Archon library for dynamically load-balancing CPU-GPU hybridized algorithms. The user of the Archon library writes their own code (the "client" code) which performs whatever computation they desire, on both the CPU and the GPU. The Archon library, meanwhile, performs lightweight instrumentation and computes optimal ratios with which the client code can split the work. Additionally, Archon's workload distribution function can be invoked periodically as needed, for the purpose of achieving optimal performance in both time and energy consumption. This redistribution is critical in scenarios in which multiple computations exist and the total workload of the system is unpredictable.

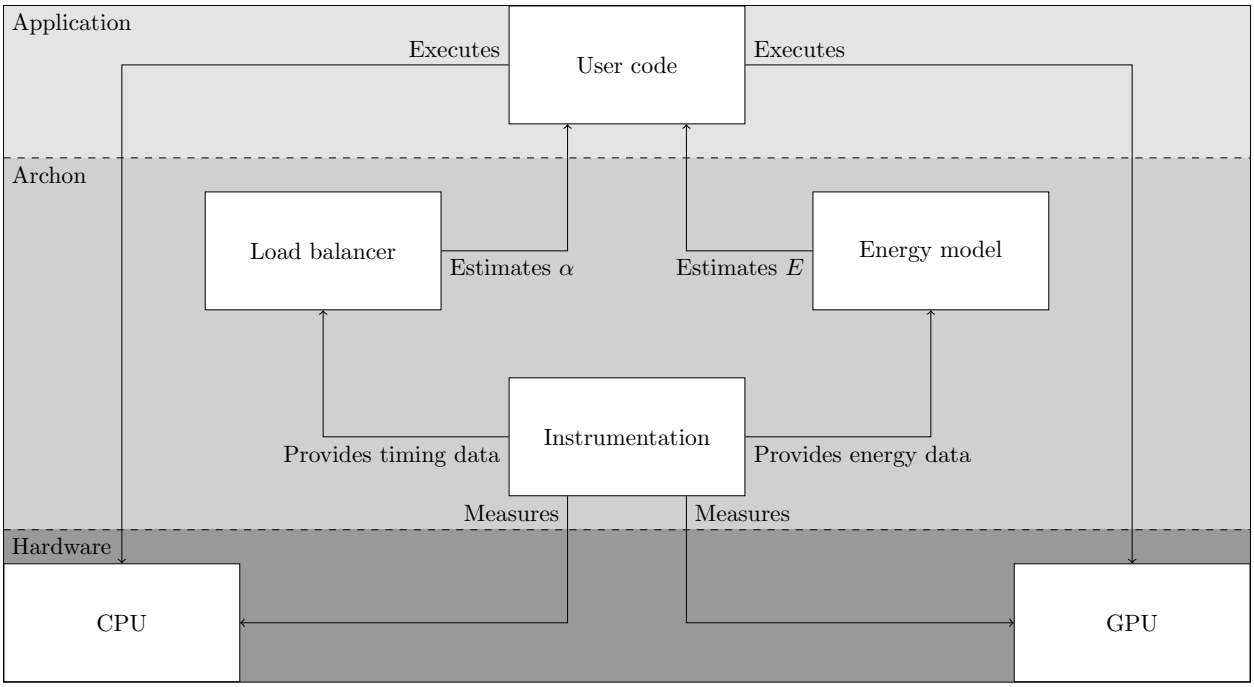


Figure 3.1: Archon system diagram

3.1 Architecture

Archon's basic workflow is shown in Figure 3.2. This depicts how the user interacts with Archon to optimize their programs. First, the user must have implementations of their algorithm for both the CPU and the GPU. The user then write a wrapper function, which calls into Archon's rebalance procedure to determine what fraction of the data each processor should get. The client program then splits some portion of the data (in whatever way is appropriate for the problem at hand) and sends the stated fractions of the data to worker threads, which perform the relevant computations while being wrapped with Archon's instrumentation code. This instrumentation code measures statistics about the execution (both time taken and, with appropriate hardware support, energy consumed), which is later used to determine optimally balanced splits. After these worker threads have both completed their task, the client code may have to combine the work of the CPU and the GPU, depending on the problem. Having done so, the client determines if it is finished with the

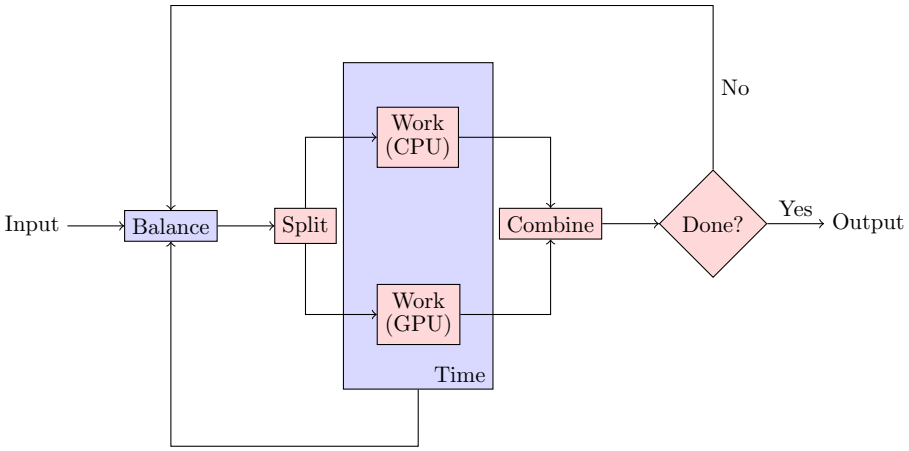


Figure 3.2: Basic Archon workflow, with user code in red and Archon code in blue

problem; if it is not finished, it goes back to the first step, in which it rebalances and allocates more workload to the worker threads. A higher-level view of this system, showing how the components interact, can be seen in Figure 3.1.

3.2 Workload Balancing

In general, our goal is to select a balancing factor $\alpha \in [0, 1]$, a fraction of the available work to offload to the GPU. The remainder, $1 - \alpha$, is left to the CPU. This factor α can conceivably be computed in different ways, depending on what we are optimizing for.

3.2.1 Optimizing Execution Time

We consider here optimizing α for time. Now, as the CPU and the GPU in our scheme do their work in parallel, the total time consumed will be the greater of runtimes of the two subroutines. That is, $T = \max(T_c, T_g)$. Since T_c increases as T_g decreases, and vice-versa, the point at which their maximum is lowest will then occur when $T = T_c = T_g$. In order to solve this equality, we need to know how T_c and T_g grow as a function of α. We can model this on the assumption that they grow linearly to α; that is, we can imagine that there

are runtimes W_g and W_c, which are the total runtimes the problem would take if offloaded entirely to the GPU or CPU, and that $T_g = \alpha W_g$ and $T_c = (1 - \alpha)W_c$. This gives us the following:

$$T_g = \alpha W_g$$
$$T_c = (1 - \alpha)\, W_c$$

Armed with these equalities, we can then find α such that $T_c = T_g$ thusly:

$$\begin{aligned} T_g &= T_c \\ \alpha W_g &= (1 - \alpha)\, W_c \\ \alpha W_g &= W_c - \alpha W_c \\ \alpha W_g + \alpha W_c &= W_c \\ \alpha &= \frac{W_c}{W_g + W_c} \end{aligned}$$

Now, we do not have access to the values of W_g and W_c at runtime; however, we can estimate them using slightly modified forms of our initial equations:

$$W_g = \frac{1}{\alpha} T_g$$
$$W_c = \frac{1}{1 - \alpha} T_c$$

This allows us to find W_g and W_c using T_g and T_c, which we can measure at runtime.

3.2.2 Instrumentation

To actually use the equations derived in Section 3.2, we need good ways to measure T_c and T_g. These measurements must have fairly high resolution (in the range of milliseconds, at worst). Additionally, they must measure time durations accurately, regardless of any network-induced shifts (as by NTP, for instance). On relatively new Linux kernels (since 2.6.28), this is available via `clock_gettime` with the `CLOCK_MONOTONIC_RAW` parameter.

For measuring the runtime cost on the GPU side, we could conceivably use CUDA's `cudaEventTimeElapsed`. However, this can be intrusive to the client program, as it may require knowledge of CUDA streams. For this reason, we choose to measure time on the GPU side in the same way as the CPU side, using `clock_gettime`.

3.3 Energy Prediction

In addition to balancing computation for optimal performance, Archon also has some abilities to predict the client program's energy consumption. This is intended to be used to provide more intelligent hybridization in the future, though at the moment it is not much used for this purpose.

3.3.1 Static Model

As a first approximation, we show here Archon's static model of energy usage. This model assumes that system power consumption is split into three components; base power consumption (P_b), CPU power consumption (P_c) and GPU power consumption (P_g). We assume that, when the CPU or GPU is active, it contributes its full P_c or P_g power consumption; and when idling, contributes nothing. Base power consumption P_b is assumed to always be active. We

then define the energy that will be consumed during a cycle as follows:

$$E = T_c P_c + T_g P_g + \max(T_c, T_g) P_b \tag{3.1}$$

In order to determine the values of these components P_c, P_g and P_b, the user can run an experiment once per machine. First, a power measurement is taken while the machien idles, to determine P_b. Next, simple programs are run, which do nothing but burn the CPU or GPU respectively, to measure total power consumption while either the CPU or GPU is running at full utilization. Subtracting P_b from the derived power numbers leaves us with only the CPU or GPU's power contribution, P_c and P_g.

3.3.2 Dynamic Model

While Archon's static model has some utility for energy prediction, we found it to fall short in some aspects. The assumption that power consumption is a binary aspect for the CPU or GPU, either drawing full power or none, turned out not to be the best. In particular on the CPU, there is some challenge in that different algorithms use the CPU in different ways. In some of our example problems, the CPU used only one core; in others two; and in others the CPU used all cores available. Each of these types of problem has a different power consumption profile, and each requires different parameters in order to apply the static model. Additionally, in the case of a program with behavior which varies over time, no single parameter will accurately describe the power consumption of the program.

To overcome these limitations, we develop Archon's dynamic model for energy consumption. The dynamic model accounts for variable power consumption by recalculating the power parameters at runtime. To do this, the model requires a way to measure energy usage at runtime; in our case using a Watts-Up! meter. Returning to Equation 3.1, we see an equation with three unknowns (P_c, P_g, and P_b); the other figures are known to us (T_c and T_g

are measured by Archon; E is provided with the assistance of our Watts-Up! meter). This means that, with data points from three "cycles", we can get a system of three equations and three unknowns, which can be solved with simple linear methods. Sadly, these solutions are often nonsensical, with such features as negative CPU power consumption and gigawatts of base power consumption. To account for this, our system throws out any results with features which are "too far" from our expectations; in particular if they include any negative power consumption. We then use for our power parameters the mean of our initial assumption and all "reasonable" solutions provided by our model.

3.4 Implementation

Archon is implemented as a C library, which is intended to be linked against programs using CUDA or OpenCL kernels. The implementation as a simple C library enables Archon to be used under a variety of circumstances and environments.

3.4.1 Configuration

The Archon library can be configured through the use of environment variables. The initially-selected ratio can be defined through the `HYBRID_GPURATIO` variable (defaulting to 50%). Additionally, the behavior of the library itself can be modified through the `HYBRID_MODE` variable. This allows for both dynamically-balanced splitting, as described in Section 3.2, as well as statically-determined splits, in which the library will always split according to the initially defined ratio. This is not likely to be useful for actual computation, but can be useful for testing purposes.

```
1  double hybrid_init(struct hybrid *h);
2  int hybrid_needs_recalculate(
3      struct hybrid *h,
4      double cycles_per_recalculate
5  );
6  double hybrid_recalculate(struct hybrid *h);
7
8  void hybrid_cpu_start(struct hybrid *h);
9  void hybrid_gpu_start(struct hybrid *h);
10
11 void hybrid_cpu_stop(struct hybrid *h);
12 void hybrid_gpu_stop(struct hybrid *h);
```

Figure 3.3: Interface to the Archon library

3.4.2 API

The Archon library is designed to allow the client application to split work in whatever manner it needs to. As such, the only responsibilities it takes for itself are to calculate how much work should be pushed to each processor. Two tasks must be performed to fulfill this responsibility.

First, instrumentation must be provided to determine how long the subprocesses are taking. This is performed with the functions `hybrid_cpu_start` and `hybrid_cpu_stop`, which must surround the CPU section of the client code. Similar functions `hybrid_gpu_start` and `hybrid_gpu_stop` are called in the GPU section of the client code. These functions are responsible for taking measurements of how the client code is performing — currently in the form of timing measurements, though other properties, such as energy consumption, could conceivably be measured as well.

Second, the library must actually use this information to compute how to optimally split the workload. This is done via the `hybrid_recalculate` function, which calculates the optimal ratio and yields it to the client program. The `hybrid_init` function, called at the start of the client program, also returns an initial ratio for the client program to use

```
1  void mat_mult_hybrid(
2          float *A, float *B, float *C, int width,
3          struct hybrid *h, double &gpu_ratio
4          ) {
5      if (hybrid_needs_recalculate(h)) {
6          gpu_ratio = hybrid_recalculate(h);
7      }
8
9      std::thread gpu([A, B, C, width, h, gpu_ratio](){
10             hybrid_gpu_start(h);
11             mat_mult_gpu(A, B, C, width, gpu_ratio);
12             hybrid_gpu_stop(h);
13     });
14
15     std::thread cpu([A, B, C, width, h, gpu_ratio](){
16             hybrid_cpu_start(h);
17             mat_mult_cpu(A, B, C, width, gpu_ratio);
18             hybrid_cpu_stop(h);
19     });
20
21     gpu.join();
22     cpu.join();
23
24     /* do combining work -
25      *    in the case of mat mult, none needed */
26
27     return;
28 }
```

Figure 3.4: Example code using the Archon library

(generally 50%, unless the user has specified otherwise). The `hybrid_needs_recalculate` function is also used by the client program to determine when enough data has been collected that a recalculation is necessary.

This information is summarized in the included header file, in Figure 3.3. An example (performing matrix multiplication) is shown in Figure 3.4. The general pattern of computation is shown; two separate threads are launched to deal with the GPU and the CPU side, and wrapped in a function which abstracts this from the calling code. The function waits until both subthreads are completed, then performs some combining work (none in this case) and returns. Each thread calls its respective `hybrid_(cpu|gpu)_(start|stop)` functions to account for its own time, and `hybrid_needs_recalculate` and `hybrid_recalculate` are used to recalculate `gpu_ratio` when necessary.

Chapter 4

Evaluation

To evaluate the effectiveness of our approach, several experiments have been carried out. In all cases, both execution time and energy consumption have been measured. Three computer vision problems have been used as test cases for our hybridization scheme, including Viola-Jones face detection [20] and both of the PSNR and MSSIM algorithms for determining image similarity. In additional, the more general problem of matrix multiplication is also used to evaluate this approach.

4.1 Overview

4.1.1 Hardware Configuration

All experiments were run on a machine with an AMD FX-8350 CPU (running at 4.0 GHz), 32 GB RAM, and an nVidia GTX 660 with 2 GB GDDR5 RAM and 930 CUDA cores running at 1033 MHz.

4.1.2 Applications

We have tested Archon with four different applications, with different performance characteristics and memory access behavior. In all cases but matrix multiplication, our implementations are built heavily upon the computer vision library OpenCV [21].

PSNR (peak signal-to-noise ratio) This is a simple algorithm for computing the similarity between two images, with minimal computational needs and easy data movement. It therefore makes a good first test for our system.

MSSIM (mean structural similarity) This is another algorithm for computing image similarity; it has roughly similar data movement (though with a little more complexity), and is quite a bit more computationally heavy.

Viola-Jones face detection Comparing to the two previous problems, which consist mostly of matrix arithmetic, Viola-Jones is a much more complex algorithm, with complicated data access patterns. It therefore serves as a more realistic test case for complex problems.

Matrix multiplication Unlike the previous problems, which all have a computer vision nature, matrix multiplication is a simply-implementable arithmetic problem, which does not require any external libraries. This enabled us to run a few extra experiments that would be made harder by external dependencies.

4.1.3 Experimental Design

For each application, a variety of experiments have been carried out.

First, we have measured both execution time and energy consumption, under the following hybridization strategies: 1) Static hybridization, in which we use Archon with fixed

pre-defined workload distribution ratios for CPU and GPU (spaced at 5% intervals); 2) Dynamic hybridization, in which Archon dynamically tunes the hybridization ratio at runtime; 3) GPU, in which only the GPU is used for execution; and 4) CPU, in which only the CPU is used. The data from these experiments is shown both as simple line graphs, and also as scatterplots using the "speedup" and "greenup" [19] metrics, in which the performance and energy consumption of the various runs are shown relative to some benchmark (the GPU variant, in our case).

Additionally, for each application we show the results of our energy model's prediction. We perform these experiments both with and without background load on the machine. In each case, we show the results of our static model, our dynamic model, and the actual energy consumed.

4.2 Image Similarity

First, we evaluate Archon with the image similarity problem. Roughly stated, the goal of this problem is to take two images, and assign a numerical score indicating how similar or different the two images are. We tested Archon using two different algorithms for solving this problem, with different complexity.

4.2.1 Algorithms

PSNR

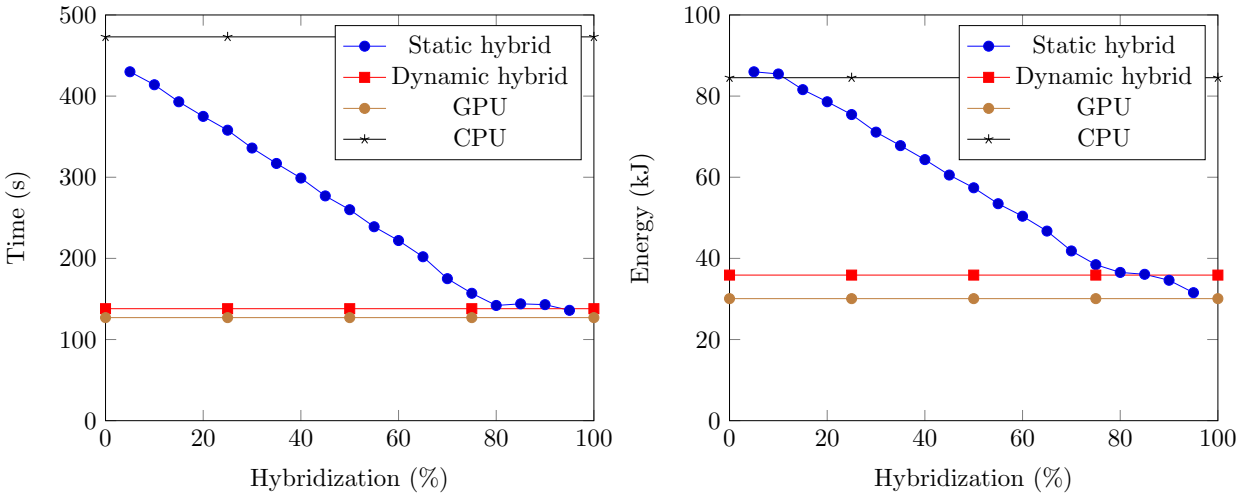
PSNR (peak signal-to-noise ratio) is the simpler of the two algorithms for solving the problem of image similarity. PSNR computes the similarity score of two images by taking the mean-squared difference between the pixels of each image, and applying logarithmic scaling to the result.

Type	Speedup	Greenup	Type	Speedup	Greenup
5%	0.295349	0.349791	60%	0.572072	0.596861
10%	0.306763	0.351987	65%	0.628713	0.64369
15%	0.323155	0.368669	70%	0.725714	0.718988
20%	0.338667	0.382667	75%	0.808917	0.782337
25%	0.354749	0.398666	80%	0.894366	0.823173
30%	0.377976	0.422935	85%	0.881944	0.833748
35%	0.400631	0.44371	90%	0.888112	0.869243
40%	0.424749	0.467468	95%	0.933824	0.953418
45%	0.458484	0.496831	GPU	1	1
50%	0.488462	0.524031	CPU	0.268499	0.355823
55%	0.531381	0.562485	Dynamic	0.92029	0.838272

Table 4.1: PSNR — speedup and greenup

Of all the experiments we run, PSNR has the simplest memory access patterns. Each output pixel's value is entirely determined by the values of the corresponding input pixels. Not only does this mean that each pixel's difference can be computed independently (and thus the computation can be trivially offloaded to either CPU or GPU), it also means that only the data used on the GPU needs to be transferred to the GPU. That is, given a 75% hybridization ratio, only 75% of the pixels of each input image will need to be sent over the bus.

MSSIM [22] (mean structural similarity) is more complex, comparing to PSNR. The two algorithms have their similarities; the most notable difference is that MSSIM incorporates a Gaussian blur to the images. This helps to account for movement within the image, while also adding complexity to the algorithm, in particular to the data transfer in our hybridized variant. Because of this Gaussian blur step, there is not a clean boundary between the data which each processor needs. Instead, some data must be present on both processors, resulting in some overlap and increased complexity.



(a) Performance (b) Energy

Figure 4.1: PSNR — variance by hybridization

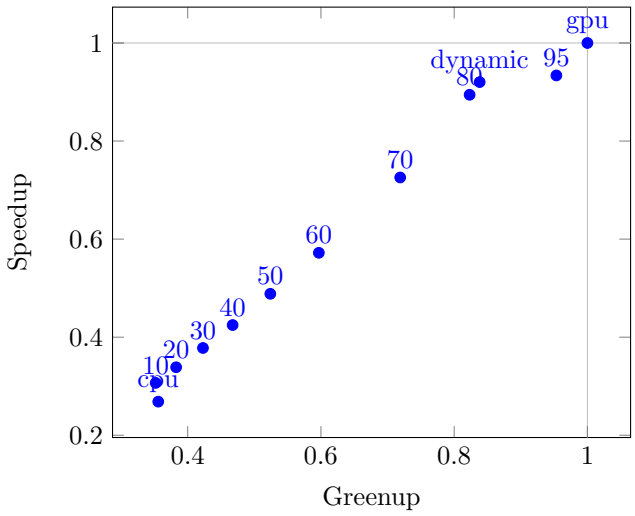
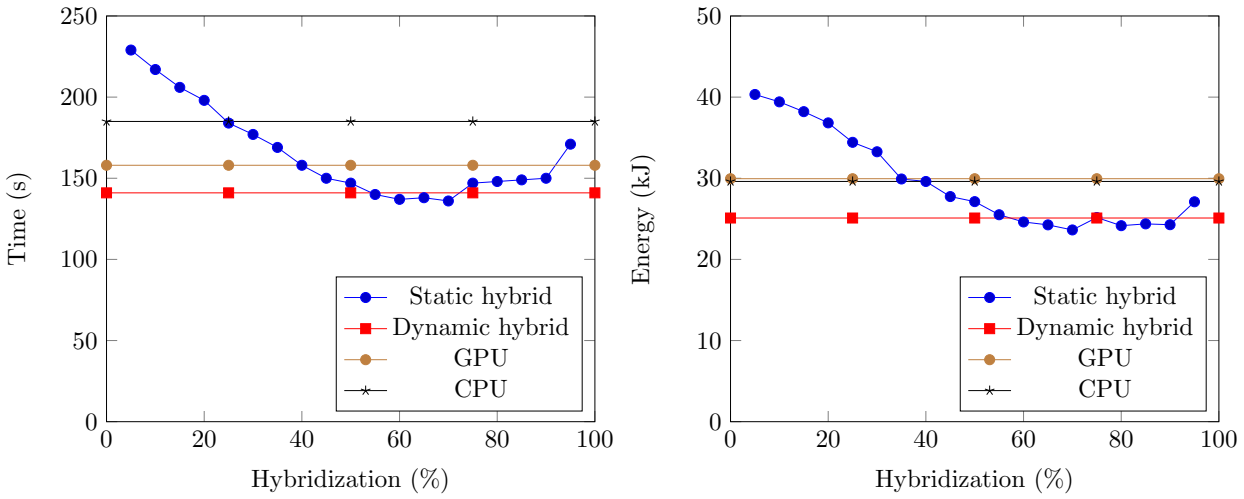


Figure 4.2: PSNR — speedup and greenup



(a) Performance

(b) Energy

Figure 4.3: MSSIM — variance by hybridization

4.2.2 Experimental Results

In the case of PSNR (Figure 4.1), we see a negative result. The GPU-only variant here outperforms all of our static hybrid systems. The dynamic hybrid performs about as well as the best static hybrids, but still falls behind the pure-GPU run. The greenup data (Figure 4.2 and Table 4.1) supports this claim in a more clear manner, showing the GPU variant to be better on both axes than any static or dynamic hybridized system.

The results of MSSIM (Figure 4.3), however, are much more promising. Our dynamically-tuned hybrid outperforms both the CPU and GPU-only variants in terms of both time (Figure 4.3a) and energy (Figure 4.3b), by fairly substantial margins. The best statically-tuned hybrids manage to, in turn, do slightly better than our dynamically-tuned hybrid, though the margin here is much smaller. The differences between the data can again be seen more clearly in Figure 4.4 and Table 4.2, which shows the dynamic hybrid scheme to perform better than all but the best of the static hybrids.

From these two similar algorithms, to the same problem, we see very different results.

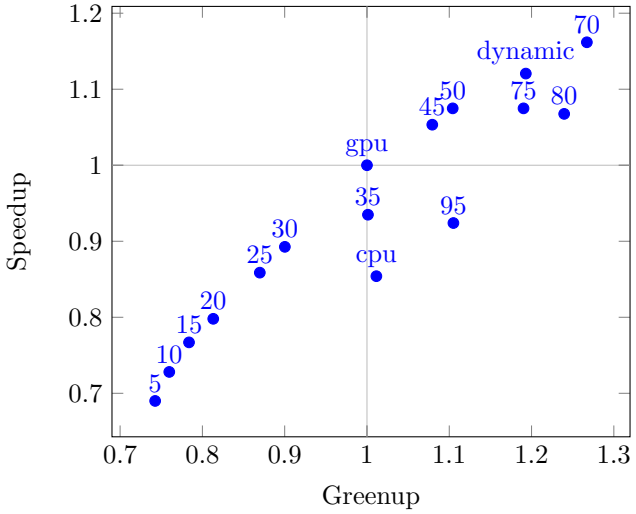
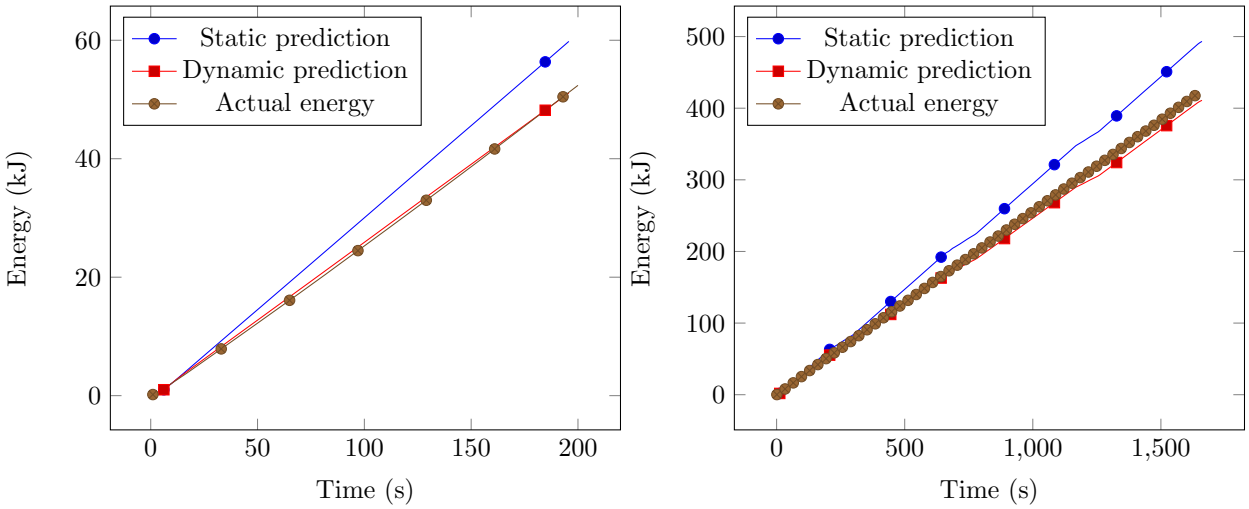


Figure 4.4: MSSIM — speedup and greenup

Type	Speedup	Greenup	Type	Speedup	Greenup
5%	0.689956	0.742596	60%	1.15328	1.21665
10%	0.728111	0.759744	65%	1.14493	1.23456
15%	0.76699	0.783742	70%	1.16176	1.26722
20%	0.79798	0.813187	75%	1.07483	1.19025
25%	0.858696	0.869776	80%	1.06757	1.23965
30%	0.892655	0.900219	85%	1.0604	1.22845
35%	0.934911	1.00119	90%	1.05333	1.23357
40%	1	1.01157	95%	0.923977	1.10504
45%	1.05333	1.07936	GPU	1	1
50%	1.07483	1.10419	CPU	0.854054	1.01142
55%	1.12857	1.17414	Dynamic	1.12057	1.19303

Table 4.2: MSSIM — speedup and greenup



(a) Without background load

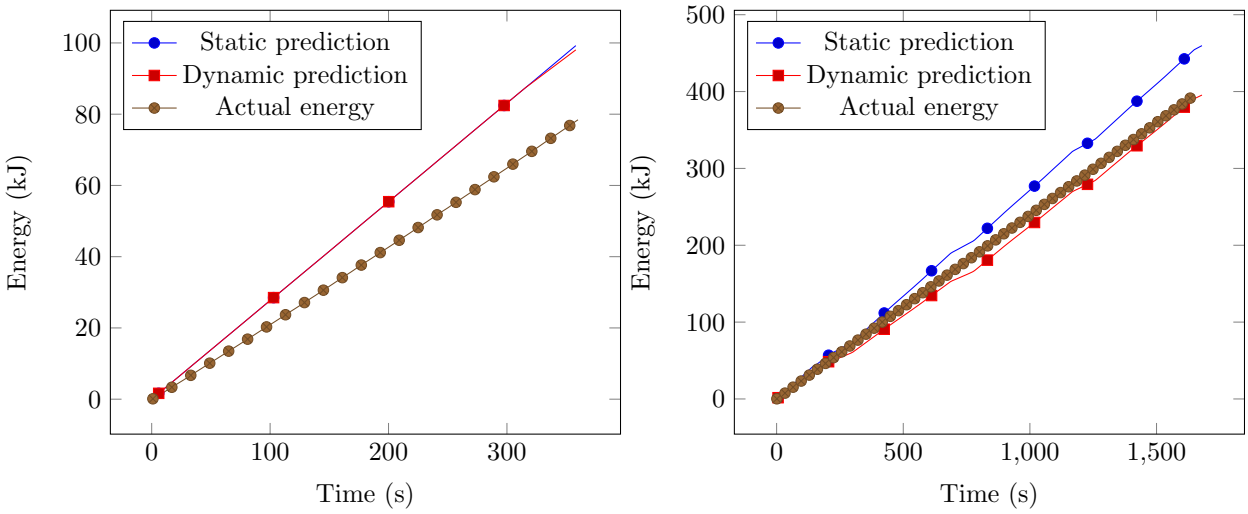
(b) With background load

Figure 4.5: PSNR — energy model

In the case of MSSIM, in which the CPU and GPU were relatively evenly matched, Archon was able to achieve substantial gains. However, in the case of PSNR, where the GPU vastly outperformed the CPU, no such gains were realized (though the losses were fairly minimal). This verifies our expectation that Archon performs better in problems in which the CPU and GPU are fairly close in performance.

Figures 4.5 and 4.6 show the results of our energy model's predictions. With the static enegy model, the results for MSSIM are fairly accurate, with an error of approximately 6%. The energy prediction for PSNR is relatively less accurate, with an error of about 14%. Interestingly, past efforts at this experiment gave much worse results; some as poor as 33%. These results were improved by the observation that our PSNR implementation was apparently multithreaded, and therefore needed to be modeled as a dual-core application. This demonstrates the necessity of understanding the implemented program in order to apply our energy model.

The dynamic energy model however, performs much better here. In each case, the dynamic energy model performs at least as well as the static model; and in all cases except



(a) Without background load

(b) With background load

Figure 4.6: MSSIM — energy model

MSSIM-without-load it performs substantially better. Notably, the dynamic model does an excellent job of tracking energy usage, in spite of its incorrect initial assumptions of the power coefficients, which assume eight-core CPU usage.

4.3 Face Detection

Next, we apply our hybridization to Viola-Jones face detection. In this problem, we receive as input a video file, which we treat as a series of still images, or "frames". For each frame, we attempt to find the bounds of any faces within the image. This algorithm is described in more detail in [20], and the GPU variant in [23].

This problem has much different work distribution characteristics, in comparison with our other problems. In our other problems, data distribution and task distribution are strongly tied. That is, if we wish to give the GPU 25% of the work, it also must receive roughly 25% of the data. With Viola-Jones face detection, this is not the case. Regardless of how work is distributed, each processor must receive the input data in full.

4.3.1 Algorithm

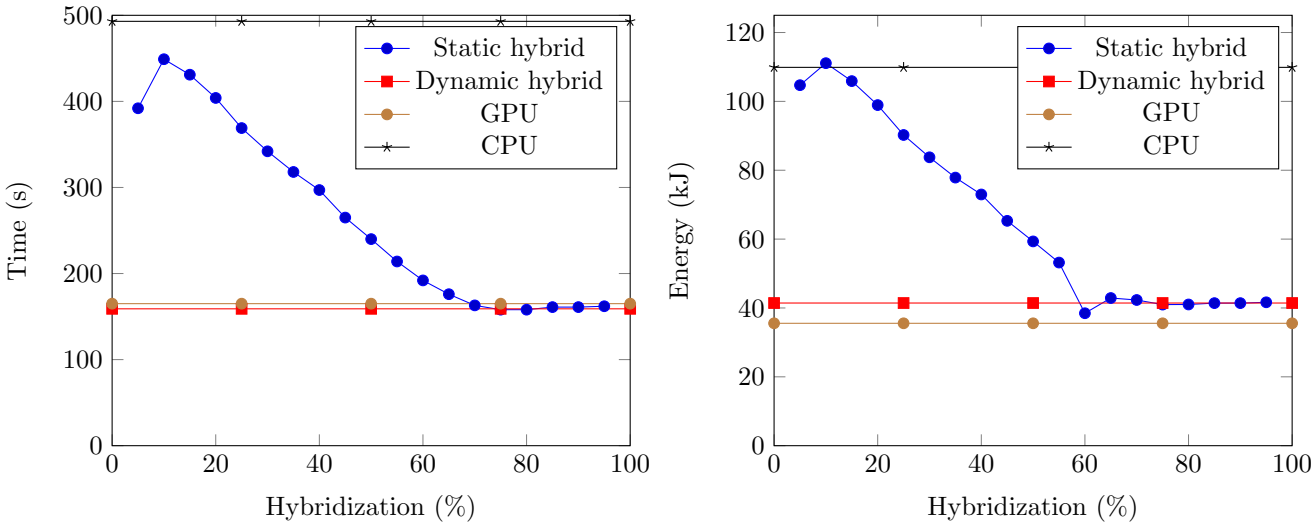
Viola-Jones face detection is built on two bedrocks: cascading classifiers and combinatorial subimages (or "windows"). First, to find where a face is, we must select some number of windows in which a face might appear. While one might imagine a more sophisticated method, the typical method is simply to try every appropriately-sized window of the original frame.

Second, we must, for each subrectangle, determine whether or not it appears to be a face. This is where the cascading classifiers come in. To determine if a subrectangle is a face, we have a series of Haar classifiers of increasing complexity. However, if we fail to match a face at any point, we can "short circuit" and skip the rest of the classifiers for that subrectangle. By arranging our classifiers such that the cheapest, and most likely to fail, occur first in our tests, we can avoid wasting time trying to apply our more complicated classifiers on subimages that are obviously not faces.

This algorithm can be parallelized on a per-window basis. That is, if we have 100 possible windows to classify, we can send some fraction of those windows to the GPU, and the remainder to the CPU. Without more intelligent splitting mechanisms, this means that we must send the entire input data to the GPU, as there is no subregion in which we guarantee the images will occur.

4.3.2 Experimental Results

As in previous cases, we compare both the time and energy consumption of our face-detection algorithm at various hybridization settings. In Figure 4.7a, we compare the performance of our dynamically-hybridized face-detection algorithm against several statically-hybridized detectors, as well as a pure-GPU variant. As with the previous example, we see that the dynamic hybrid has the best performance, and is quite similar to the best static hybrid.



(a) Performance (b) Energy

Figure 4.7: Viola-Jones face detection — variance by hybridization

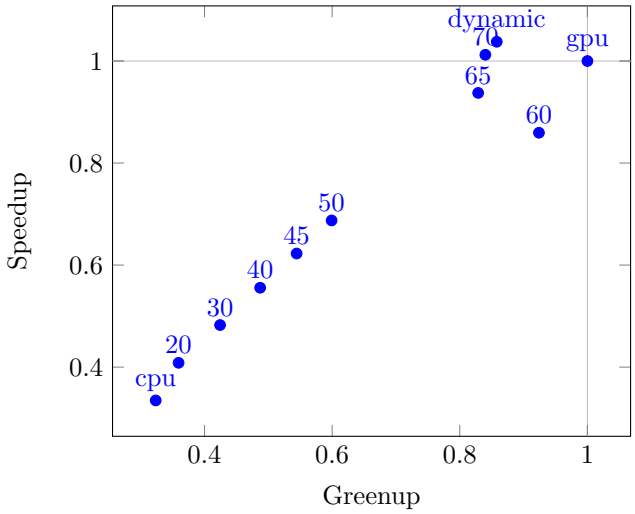


Figure 4.8: Viola-Jones face detection — speedup and greenup

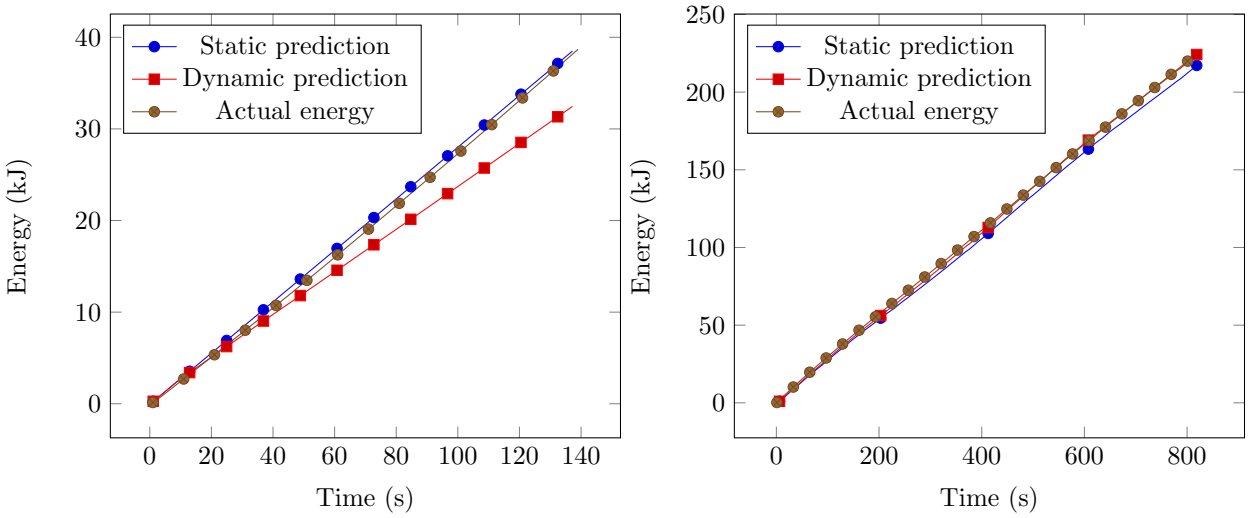
Type	Speedup	Greenup	Type	Speedup	Greenup
5%	0.420918	0.339484	60%	0.859375	0.924066
10%	0.367483	0.319954	65%	0.9375	0.828806
15%	0.382831	0.335694	70%	1.01227	0.840064
20%	0.408416	0.359332	75%	1.0443	0.867452
25%	0.447154	0.39391	80%	1.0443	0.866725
30%	0.482456	0.424386	85%	1.02484	0.85849
35%	0.518868	0.456517	90%	1.02484	0.858492
40%	0.555556	0.487197	95%	1.01852	0.853522
45%	0.622642	0.544347	GPU	1	1
50%	0.6875	0.599136	CPU	0.334686	0.323502
55%	0.771028	0.668328	Dynamic	1.03774	0.857918

Table 4.3: Viola-Jones face detection — speedup and greenup

In Figure 4.7b, we compare the energy usage of the four variants. The results show that, while we did get performance boosts, they were not strong enough to offset the increased power consumption. Therefore, the hybridized approaches have resulted in higher energy usage than merely running on the GPU alone. The relative-performance data in Figure 4.8 and Table 4.3 shows both the relatively low speedup obtained from hybridization and the more substantial energy loss associated with it.

In Figure 4.9, we take a look at our energy model, this time as applied to Viola-Jones face detection. Where our previous efforts used a model trained for single-core or dual-core power consumption, Viola-Jones, which uses all available cores on the CPU, instead uses an eight-core model (as we experiment with an eight-core CPU). Under no load, the static model again produces quite accurate results, with an error of approximately 3%. The dynamic model here turns out to underpredict a bit, however. With a background load provided, both the static and dynamic models track the actual energy consumption almost perfectly.

Finally, we examine the behavior of our Archon system under load. Unlike our previous experiments, which were run on a machine with minimal background tasks, we show here an experiment in which our machine is also busy with other tasks. To simulate a variable



(a) Without background load

(b) With background load

Figure 4.9: Viola-Jones face detection — energy model

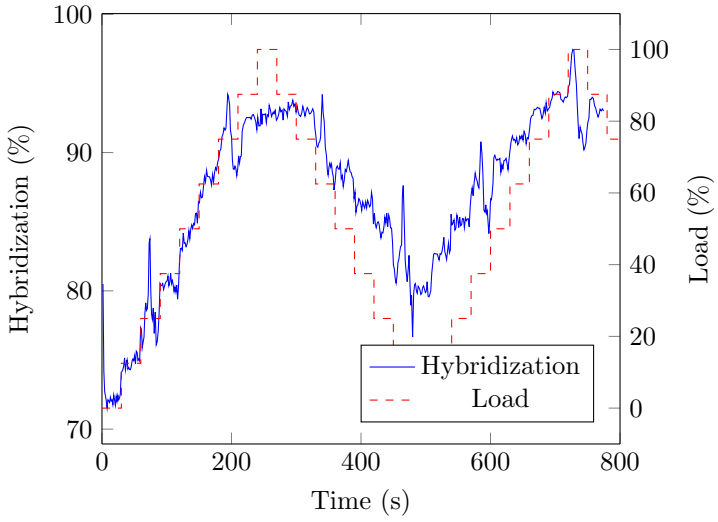


Figure 4.10: Viola-Jones face detection — hybridization under load

Type	Time (s)	Energy (kJ)
Dynamic	807	225.333
81%	1021	278.098
91%	825	226.638

Table 4.4: Viola-Jones face detection — performance under load

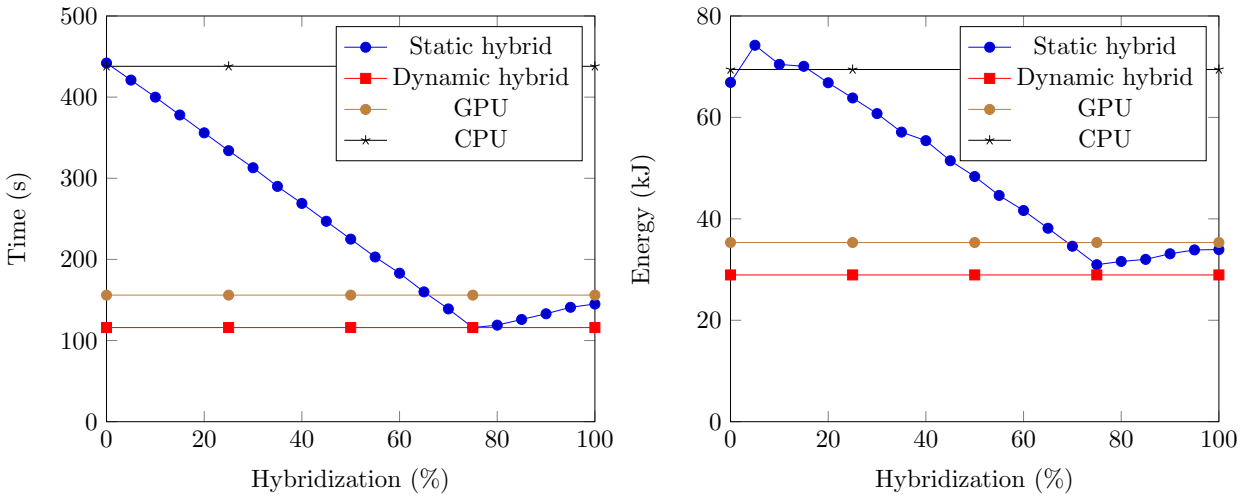
workload, a stairstep load creator was utilized. The load creator does nothing for 30 seconds, then uses one core at full capacity for 30 seconds, then two cores for 30 seconds, and so on, until eventually all eight cores are being utilized, after which the load winds down to zero in the same fashion before starting the cycle again from the beginning.

First, in Figure 4.10, we show how hybridization levels vary when load varies. As the background load on the CPU increases, Archon shifts more work off to the GPU; similarly, as the load decreases, Archon returns more work back to the CPU. Because of the nature of our workload generator (which operates on cores at a time), this experiment was not run on previous examples, as they did not exhibit 8-core parallelism on the CPU side, and so would not be affected by the load generator.

In Table 4.4, we see how our dynamic tuning performs, as opposed to a statically-tuned ratio determined from performance under no load. Our dynamic tuning system performs substantially better than a static system with 81% hybridization, which was determined to be optimal under no load. A static hybridization level of 91% is more competitive with our dynamic tuning; however, our dynamically tuned system still achieves about the same level of performance (or slightly higher), without having to specify in advance that the machine will be performing under load.

4.4 Matrix Multiplication

Finally, we test our approach on the classic problem of matrix multiplication.



(a) Performance

(b) Energy

Figure 4.11: Matrix multiplication — variance by hybridization

4.4.1 Single-core

The performance and energy consumption of a 4096×4096 matrix multiplication computation in the four sets of experiments are shown in Figure 4.11. As shown in Figure 4.11a, in terms of performance, the static hybrid outperforms the pure-GPU execution for splits between 70% and 100%. Additionally, the dynamic hybrid experiment performs about the same as the best static hybrid setting. In terms of energy (Figure 4.11b), the results are similar. These results show that the dynamic hybridization provided by Archon can effectively improve both the performance and the energy consumption without extra efforts from the users.

Another view of this data is provided in Figure 4.12 (and Table 4.5), showing the relative speedup and greenup. This shows that, for this problem, performance and energy improvement go hand-in-hand, with the best results being obtained by our dynamic solution.

A more detailed view of power consumption can be see in Figure 4.13. These results were obtained via GreenSoft [24], a project from Texas State University which enables collection of specific energy data for research purposes. Here we can see the power consumption of our

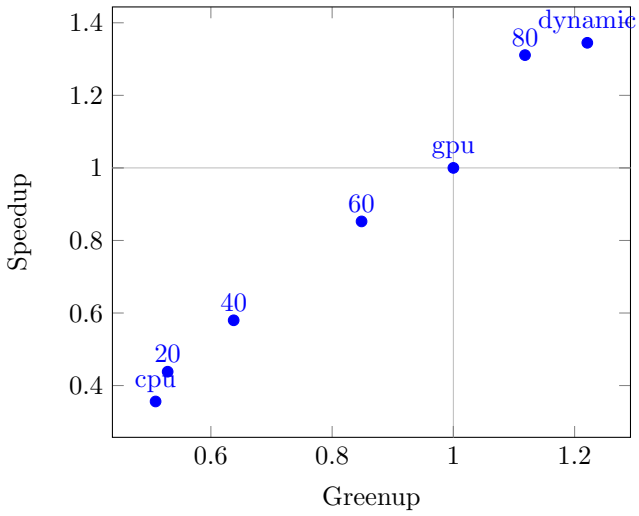


Figure 4.12: Matrix multiplication — speedup and greenup

Type	Speedup	Greenup	Type	Speedup	Greenup
5%	0.370546	0.476028	60%	0.852459	0.848534
10%	0.39	0.501535	65%	0.975	0.926259
15%	0.412698	0.504238	70%	1.1223	1.02131
20%	0.438202	0.528693	75%	1.34483	1.14137
25%	0.467066	0.553527	80%	1.31092	1.11815
30%	0.498403	0.581602	85%	1.2381	1.10392
35%	0.537931	0.61879	90%	1.17293	1.06713
40%	0.579926	0.637597	95%	1.10638	1.04308
45%	0.631579	0.68667	GPU	1	1
50%	0.693333	0.730796	CPU	0.356164	0.508803
55%	0.768473	0.791936	Dynamic	1.34483	1.2209

Table 4.5: Matrix multiplication — speedup and greenup

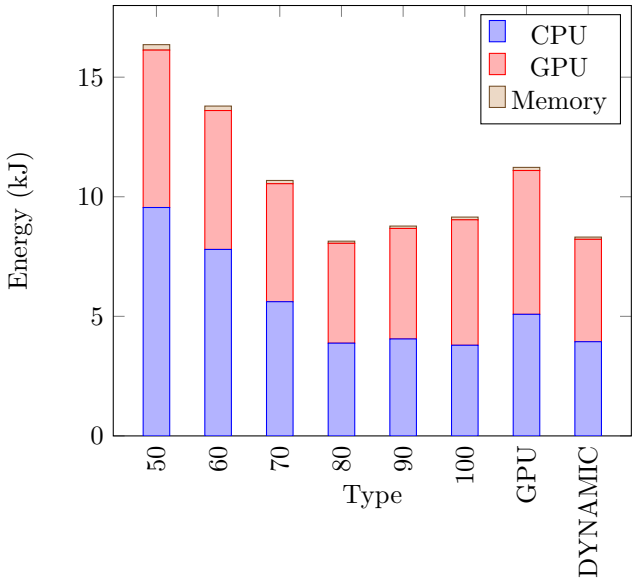


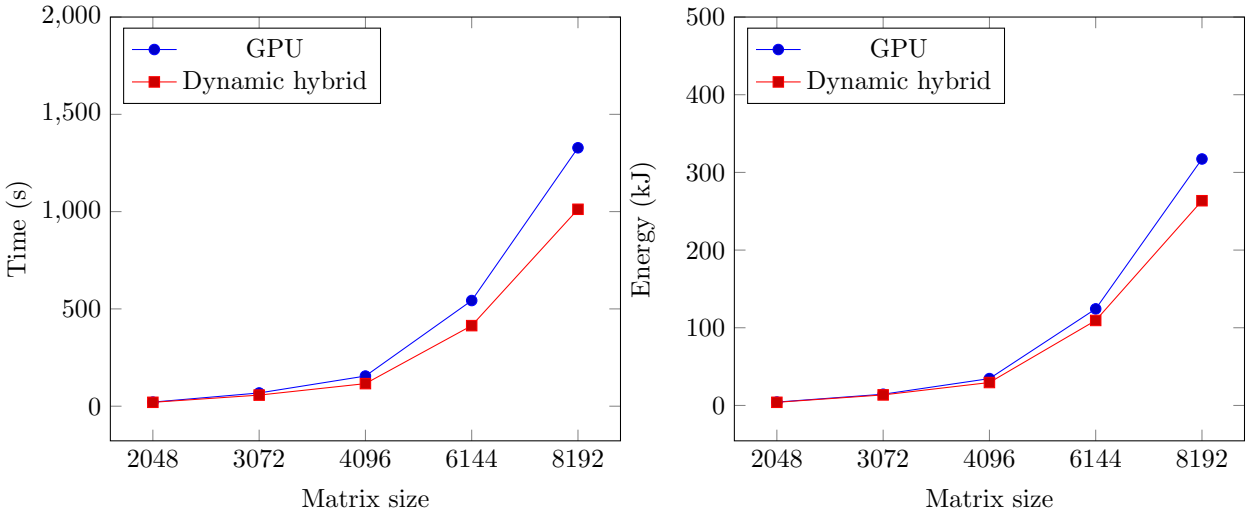
Figure 4.13: Matrix multiplication — energy breakdown — 4096×4096

algorithm, broken down by component (into CPU, GPU, and memory). This more detailed chart shows us that much of our power savings seems to come from the CPU, though we also use less energy on the GPU side of the computation.

To evaluate the scalability of Archon, we run the matrix multiplication computation with various sizes, ranging from 2048 to 8192. The performance and the energy consumption of the dynamic hybrid execution and pure GPU execution are shown in Figure 4.14. For relatively small problems, the performance and energy benefits of using Archon are limited. However, as the problem increases in size, the performance/energy gains grow as well; at the largest shown size, 8192×8192, hybridization drops us from 1328 s and 317 kJ to 1012 s and 263 kJ, for a speedup of approximately 24% and an energy savings of 17%.

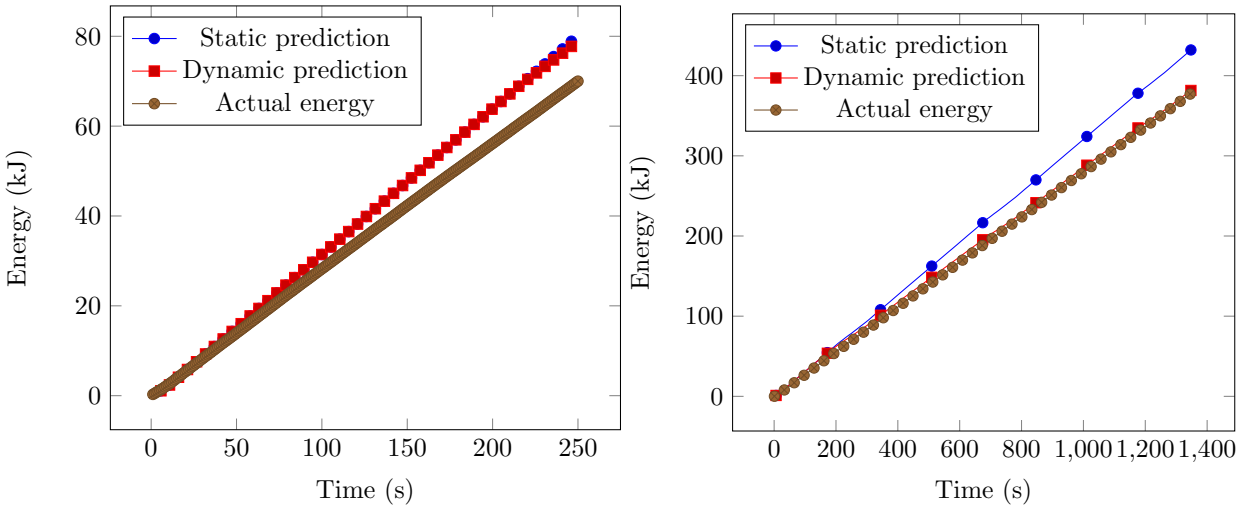
We have also evaluated the accuracy of our energy prediction model, with a 4096×4096 matrix multiplication. The results are shown in Figure 4.15. Our model accurately predicts the energy consumption at runtime; with no background load, the static model provides an error of less than 5%. The dynamic energy model performs almost identically, in this case.

In experiments with background load, the static model performs about the same as



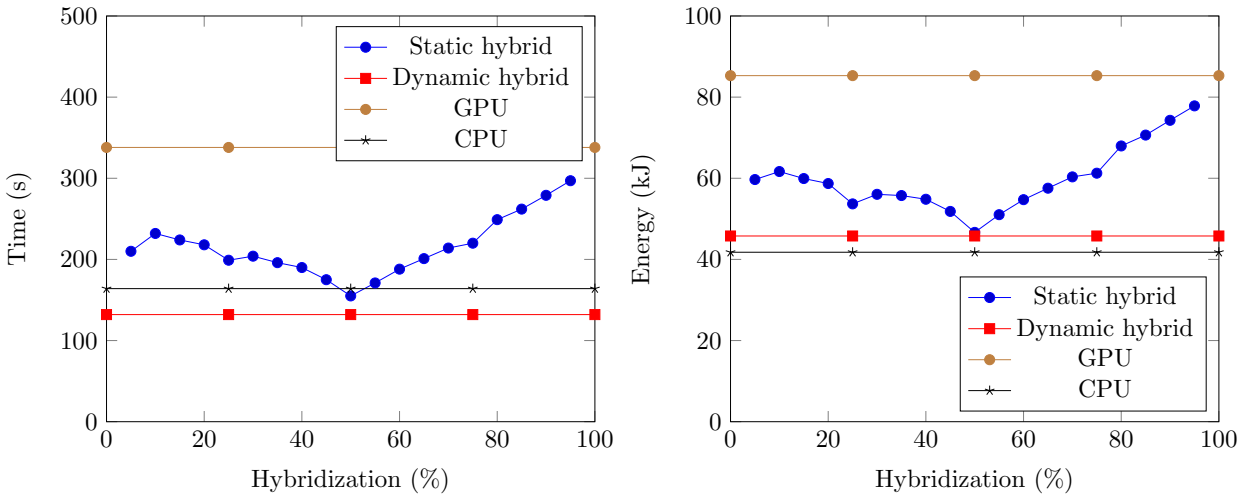
(a) Performance (b) Energy

Figure 4.14: Matrix multiplication — scalability



(a) Without background load (b) With background load

Figure 4.15: Matrix multiplication — energy model



(a) Performance

(b) Energy

Figure 4.16: 8-core matrix multiplication — variance by hybridization

before; but the dynamic prediction model does substantially better, tracking actual energy consumption almost perfectly again.

4.4.2 Multi-core

Again we show performance and energy consumption for a 4096×4096 matrix multiplication, this time with an 8-core CPU implementation in Figure 4.16. We see here that the relationship between the CPU and GPU implementations is quite reversed from the previous example; whereas before the CPU was quite a bit slower and less energy-efficient than the GPU implementation, with the 8-core variant the CPU does substantially better than the GPU. Our dynamically-tuned hybrid, however, behaves similarly; that is, it performs slightly better than the better of the two results. As one might expect, it settles on a much more CPU-biased hybridization level when the CPU implementation is more efficient. In terms of energy, the CPU implementation ends up slightly more efficient than our dynamic hybrid, however.

Again we show the relative speedup and greenup in Figure 4.17 and Table 4.6. This

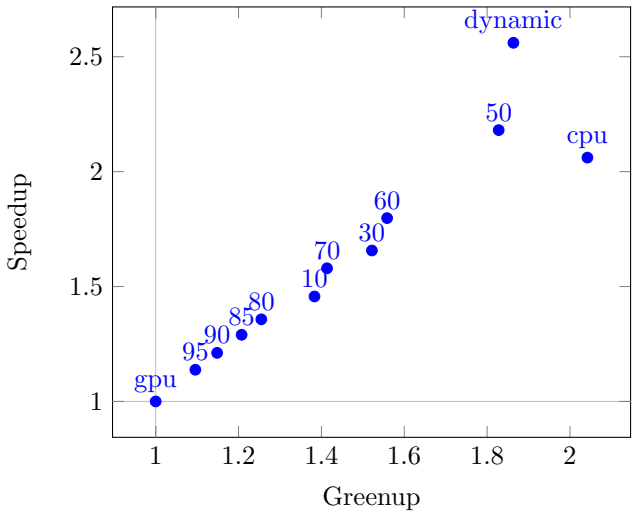
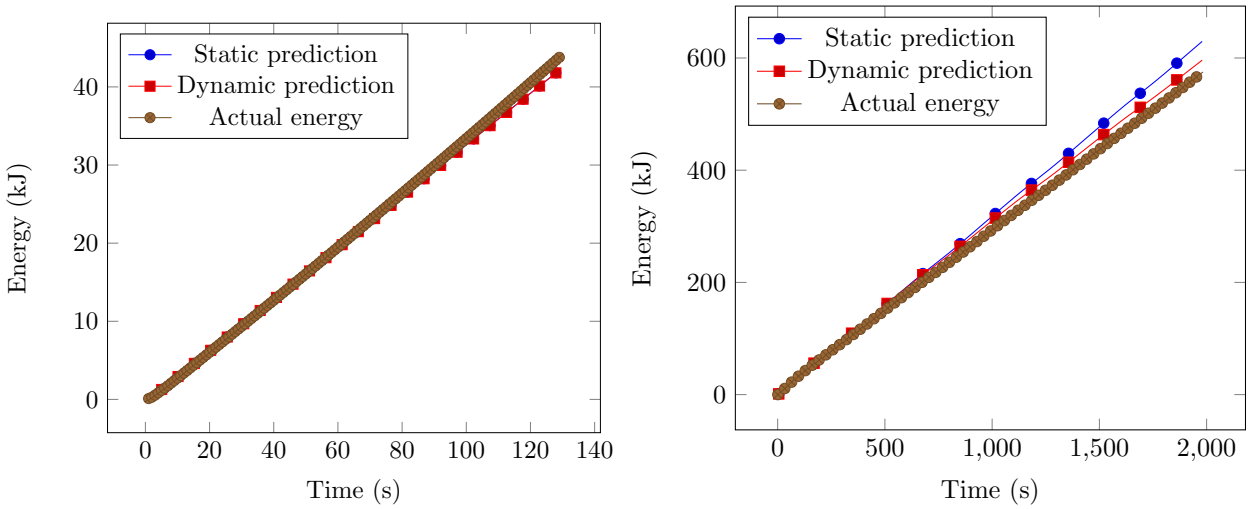


Figure 4.17: 8-core matrix multiplication — speedup and greenup

Type	Speedup	Greenup	Type	Speedup	Greenup
5%	1.60952	1.42907	60%	1.79787	1.55858
10%	1.4569	1.38329	65%	1.68159	1.48205
15%	1.50893	1.42319	70%	1.57944	1.41355
20%	1.55046	1.45303	75%	1.53636	1.39256
25%	1.69849	1.58878	80%	1.35743	1.25501
30%	1.65686	1.52194	85%	1.29008	1.20741
35%	1.72449	1.53023	90%	1.21147	1.14825
40%	1.77895	1.55576	95%	1.13805	1.09574
45%	1.93143	1.6457	GPU	1	1
50%	2.18064	1.82794	CPU	2.06098	2.04214
55%	1.97661	1.67165	Dynamic	2.5606	1.86342

Table 4.6: 8-core matrix multiplication — speedup and greenup



(a) Without background load

(b) With background load

Figure 4.18: 8-core matrix multiplication — energy model

provides another view showing that, while our dynamic hybrid was the fastest performer, and was more energy-efficient than any of the static hybrids, it was less energy efficient than the CPU-only implementation.

In Figure 4.18, we show how our energy model applies to 8-core matrix multiplication. As with the Viola-Jones experiments, we must specifically use an 8-core model to accurately model our power consumption. Having done so, we were able to predict energy consumption without load with an error less than 3%. In the case with background load, static prediction still does quite well here; however, our dynamic energy model still does slightly better.

Finally, in Figure 4.19, we again show how an 8-core application's hybridization behavior varies with varying background load. The same load generator was used as in the Viola-Jones in Section 4.3.2. As was the case with Viola-Jones, our dynamic hybridization effectively tracked the background load, moving more work to the GPU when the CPU was unavailable, and restoring work to the CPU when it was free.

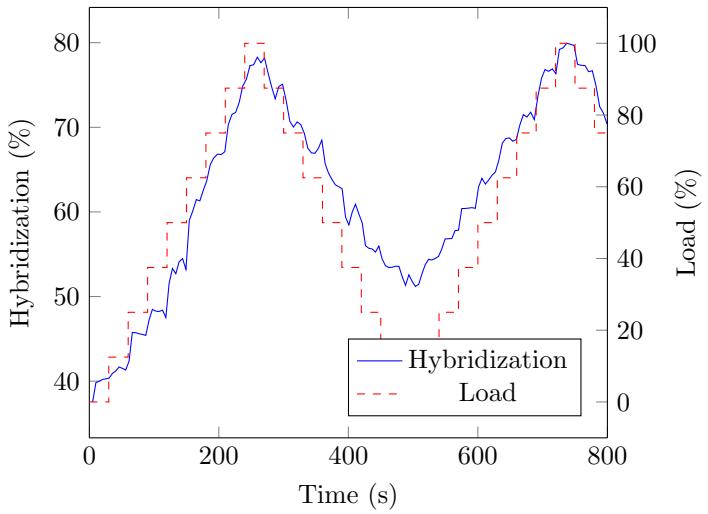


Figure 4.19: 8-core matrix multiplication — hybridization under load

Chapter 5

Conclusion

In this work, we set out to show that dynamic CPU-GPU hybridization was a viable method of optimizing performance for many applications, with minimal effort on the client programmer's part. We showed that, in problems for which a GPU and CPU are both reasonably suitable, our dynamic hybridization was quite effective at optimizing runtimes of applications, generally performing comparably with the best possible statically-selected hybridizations. In the face of variable background load, our dynamic hybridization varied and was able to take better advantage of available hardware than any of the statically-selected hybridizations. Additionally, this dynamic hybridization was achieved without requiring the programmer to determine ahead-of-time what the best hybridization ratio was, and without requiring substantial changes to the client programs.

We also set out to show that our simple static energy model was a usable method for estimating power consumption of hybridized algorithms. In cases in which we understood the performance characteristics of the application, our model was quite effective. However, in one case (PSNR), in which we did not as well understand the parallelism of our implementation, we found the predictive utility of our energy model to be somewhat worse This indicates that the static model does not apply as well to problems we do not understand, and possibly

would not apply well to problems with varying amounts of parallelism. However, our more complex dynamic energy model performed better than the static model, even in cases where we did not know the specific performance characters of the problem up-front.

5.1 Future Works

First, in the future, we would like to extend our framework to work more automatically, in cooperation with OpenCL. A major design goal of Archon is to enable optimization with minimal burden on the client programmer; however, in its current form with CUDA, Archon requires users to write two independent implementations of their algorithm (one for the CPU and one for the GPU), even if they are identical. OpenCL allows for a single piece of code to be compiled to target either the GPU or the CPU (as well as a variety of other processor types). Conceivably, our Archon system, tied to OpenCL, could be used to create a "virtual processor", which could automatically hybridize computations with even less effort on the client programmer's part. However, there are some issues with this plan that must be addressed; memory in particular is somewhat complicated, as data transfer requirements depend on the problem being solved in non-trivial ways.

Additionally, we would like to make better use of our energy model. While we have tested our memory model's predictive power in several cases in this work, we have not applied it to solve any actual problems. It may be possible to use our energy model to adaptively tune hybridization for improved energy-performance, rather than the time-performance for which we currently optimized.

Bibliography

[1] R. R. Schaller, "Moore's Law: Past, Present and Future," *Spectrum, IEEE*, vol. 34, no. 6, pp. 52–59, 1997.

[2] H. Sutter, "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software," *Dr. Dobb's Journal*, vol. 30, no. 3, March 2005.

[3] S. Cho and R. G. Melhem, "Corollaries to Amdahl's Law for Energy," *Computer Architecture Letters*, vol. 7, no. 1, pp. 25–28, 2008.

[4] J. F. Hughes, A. van Dam, M. McGuire, D. F. Sklar, J. D. Foley, S. K. Feiner, and K. Akeley, *Computer graphics: principles and practice (3rd ed.)*. Boston, MA, USA: Addison-Wesley Professional, July 2013.

[5] S. Hong and H. Kim, "An integrated gpu power and performance model," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 280–289. [Online]. Available: http://doi.acm.org/10.1145/1815961.1815998

[6] Z. He and B. Hong, "Dynamically tuned push-relabel algorithm for the maximum flow problem on cpu-gpu-hybrid platforms," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, April 2010, pp. 1–10.

[7] R. Ge, X. Feng, M. Burtscher, and Z. Zong, "Performance and energy modeling for cooperative hybrid computing," in *Networking, Architecture, and Storage (NAS), 2014 9th IEEE International Conference on*, Aug 2014, pp. 232–241.

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

[9] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "Gpuwattch: Enabling energy optimizations in gpgpus," *SIGARCH Comput. Archit. News*, vol. 41, no. 3, pp. 487–498, Jun. 2013. [Online]. Available: http://doi.acm.org/10.1145/2508148.2485964

[10] F. Yao, A. Demers, and S. Shenker, "A scheduling model for reduced cpu energy," in *Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on*, Oct 1995, pp. 374–382.

[11] H. F. Sheikh, H. Tan, I. Ahmad, S. Ranka, and P. Bv, "Energy- and performance-aware scheduling of tasks on parallel and distributed systems," *J. Emerg. Technol. Comput. Syst.*, vol. 8, no. 4, pp. 32:1–32:37, Nov. 2012. [Online]. Available: http://doi.acm.org/10.1145/2367736.2367743

[12] J. Kang and S. Ranka, "Dynamic algorithms for energy minimization on parallel machines," in *16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, Feb 2008, pp. 399–406.

[13] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa, "Timegraph: Gpu scheduling for real-time multi-tasking environments," in *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 2–2. [Online]. Available: http://dl.acm.org/citation.cfm?id=2002181.2002183

[14] L. Chen, X. Huo, and G. Agrawal, "Accelerating mapreduce on a coupled cpu-gpu architecture," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 25:1–25:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389030

[15] J. A. Stuart and J. D. Owens, "Message passing on data-parallel architectures," in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, May 2009, pp. 1–12.

[16] A. M. Aji, J. Dinan, D. Buntinas, P. Balaji, W. c. Feng, K. R. Bisset, and R. Thakur, "Mpi-acc: An integrated and extensible approach to data movement in accelerator-based systems," in *High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, June 2012, pp. 647–654.

[17] A. M. Aji, L. S. Panwar, F. Ji, M. Chabbi, K. Murthy, P. Balaji, K. R. Bisset, J. Dinan, W.-c. Feng, J. Mellor-Crummey, X. Ma, and R. Thakur, "On the efficacy of gpu-integrated mpi for scientific applications," in *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing*, ser. HPDC '13. New York, NY, USA: ACM, 2013, pp. 191–202. [Online]. Available: http://doi.acm.org/10.1145/2462902.2462915

[18] S. Kim, S. Huh, X. Zhang, Y. Hu, A. Wated, E. Witchel, and M. Silberstein, "Gpunet: Networking abstractions for gpu programs," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield,

CO: USENIX Association, 2014, pp. 201–216. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/kim

[19] S. Abdulsalam, Z. Zong, Q. Gu, and M. Qiu, "Using the greenup, powerup, and speedup metrics to evaluate software energy efficiency," in *Green Computing Conference and Sustainable Computing Conference (IGSC), 2015 Sixth International*, Dec 2015, pp. 1–8.

[20] P. Viola and M. Jones, "Robust real-time object detection," in *International Journal of Computer Vision*, 2001.

[21] "OpenCV," http://opencv.org, accessed: 2015-05-08.

[22] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: from error visibility to structural similarity," *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, April 2004.

[23] D. Hefenbrock, J. Oberg, N. T. N. Thanh, R. Kastner, and S. B. Baden, "Accelerating viola-jones face detection to fpga-level using gpus," in *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, May 2010, pp. 11–18.

[24] "GreenSoft," https://greencode.cs.txstate.edu/, accessed: 2016-11-07.