A Black-Box Approach to Energy-Aware Scheduling on Integrated CPU-GPU Systems

Rajkishore Barik
Intel Labs, USA
rajkishore.barik@intel.com

Naila Farooqui
Georgia Institute of Technology, USA
naila@cc.gatech.edu

Brian T. Lewis
Intel Labs, USA
brian.t.lewis@intel.com

Chunling Hu
Intel Labs, USA
chunling.hu@intel.com

Tatiana Shpeisman
Intel Labs, USA
tatiana.shpeisman@intel.com

Abstract

Energy efficiency is now a top design goal for all computing systems, from fitness trackers and tablets, where it affects battery life, to cloud computing centers, where it directly impacts operational cost, maintainability, and environmental impact. Today's widespread integrated CPU-GPU processors combine a CPU and a GPU compute device with different power-performance characteristics. For these integrated processors, hardware vendors implement automatic power management policies that are typically not exposed to the end-user. Furthermore, these policies often vary between different processor generations and SKUs. As a result, it is challenging to design a generally-applicable energy-aware runtime to schedule work onto both the CPU and GPU of such integrated CPU-GPU processors to optimize energy consumption.

We propose a new black-box scheduling technique to reduce energy use by effectively partitioning work across the CPU and GPU cores of integrated CPU-GPU processors. Our energy-aware scheduler combines a power model with information about the runtime behavior of a specific workload. This power model is computed once for each processor to characterize its power consumption for different kinds of workloads. On two widely different platforms, a high-end desktop system and a low-power tablet, our energy-aware runtime yields an energy-delay product that is 96% and 93%, respectively, of the near-ideal Oracle energy-delay product on a diverse set of workloads.

CGO'16, March 12–18, 2016, Barcelona, Spain
© 2016 ACM. 978-1-4503-3778-6/16/03...$15.00
http://dx.doi.org/10.1145/2854038.2854052

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers, Run-time environments

General Terms Algorithms, Performance, Experimentation

Keywords Energy efficiency, Heterogeneous CPU-GPU scheduling, Power characterization

1. Introduction

Improving energy efficiency has become an important concern for all computing systems. In mobile devices, reduced power consumption translates into improved battery life, smaller batteries, and a wider range of possible form factors. In large-scale data centers, the energy efficiency of computer systems affects many factors including density, cooling and power costs, reliability, scalability, and flexibility.

Many computing systems today include accelerators such as GPUs that, in addition to providing significant gains in performance, provide substantial energy savings for data-parallel, throughput-oriented workloads. Much recent research has focused on using dynamic voltage and frequency scaling (DVFS) settings to increase energy efficiency of GPGPU execution [11, 17, 27]. That work, however, primarily targets discrete GPUs. In integrated CPU-GPU processors, the CPU and GPU are located on the same die. While the tighter integration of CPUs and GPUs enables lower-cost compute offloading and memory sharing, it also forces the processors to share the chip-level power budget and thermal capacity. As a result, in most integrated CPU-GPU processors, hardware vendors such as Intel and AMD do not provide user-level DVFS control for the GPU, and instead use automatic hardware power management techniques such as Intel's TurboBoost and AMD's TurboCORE to achieve good performance and reduced energy consumption within a thermal budget.

The performance and energy consumption of integrated CPU-GPU processors depend on many factors. First, the performance and power use of both the CPU and GPU depend on

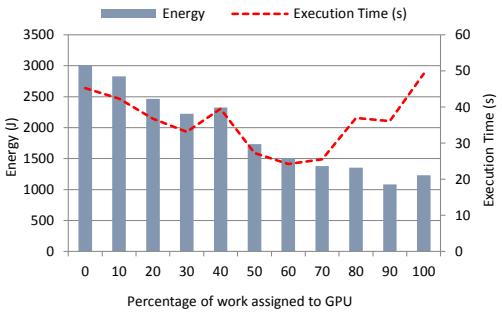

Figure 1: Energy use and performance for the Connected Components application on an Intel Haswell desktop with varying percentage of work assigned to GPU.

workload characteristics. GPUs, for example, are especially effective at executing tasks with large amounts of uniform data parallelism and high compute-to-memory access ratios. For applications whose work can be executed on both the CPU and GPU, performance and energy use also depend on how the work is distributed between the CPU and GPU. In Figure 1, we show performance and energy consumption for the Connected Components (CC) application on an Intel Haswell desktop machine. Both vary significantly as more and more work is assigned to the GPU. Minimum energy is achieved when 90% of work is offloaded to the GPU, while best performance is achieved at 60% GPU offload. This illustrates that–while GPU execution is usually beneficial for energy and performance–the lowest energy use or best performance may require both the CPU and GPU.

Automatically characterizing the behavior of power management techniques on a given platform is non-trivial. Power management techniques such as TurboBoost and TurboCORE are complex and include sophisticated heuristics for controlling the frequencies of the CPU, GPU, and other processor components to maintain a power budget. The processor package control unit (PCU) governing power management techniques adjusts CPU and GPU frequencies based on several factors including expected usage, number of OS requests, processor temperatures, and cache miss rates. Additionally, power management policies for a processor vary from one specific SKU (Stock Keeping Unit) to another, and sometimes even from die to die. For example, the power management policies in Intel's Haswell Core processors are different from those in its Atom-based processors. As an example of the complex power management policies in today's integrated processors, Figure 2 shows package power consumption over time for a memory-bound application on two processors: a Bay Trail tablet (left) and an Intel Haswell desktop (right). Package power consists of power consumption by the CPU cores, GPU cores, the ring interconnect, and the last-level cache. This application executes 90% of the work on the GPU and 10% on the CPU. On the Bay Trail, power consumption drops significantly during intervals when only the CPU is active (and the GPU has finished execution), while on the Haswell, package power consumption increases with CPU execution. Note also that the more powerful GPU on the Haswell completes work much faster than the CPU, while on the Bay Trail, the processors have similar performance so the GPU execution time is longer. In general, an application's energy consumption varies with its execution characteristics, CPU-GPU work distribution, as well as the platform's power-performance characteristics and PCU power management policies.

There are several energy-related metrics. Which metric is important depends on how the computer system is used; for example, users of mobile devices often want to reduce total energy consumption while for most other users, it is more important to optimize the energy-delay product (ED). The energy-delay product is defined as E*T, where E is energy and T is time; this metric factors in how long the system takes to execute an application. And for some data-center and HPC applications, execution time is so important that another energy metric is used, the energy delay-squared product (ED^2), defined as E*T*T. The best workload distribution will depend on the metric used. For instance, a workload distribution that improves the energy use for a given workload is not necessarily the same as one that improves its energy-delay product.

In this paper, we investigate the problem of optimizing energy efficiency on integrated CPU-GPU processors. Since most integrated processors don't allow the user to control DVFS, we use a scheduling runtime that automatically distributes work between the CPU and GPU to optimize a given energy metric. Our scheduling approach works as follows. We first do a one-time characterization of the processor's power use when executing different kinds of workloads (e.g., compute-intensive or memory-intensive) and measuring the power each uses. Afterwards, at the start of application execution, our scheduling runtime profiles its execution to understand its runtime behavior. Our scheduler then combines the application's runtime behavior with the processor's power characterization to distribute the application's remaining work between the CPU and GPU cores based on the energy metric being optimized. Our scheduler is user-level and automatic.

To the best of our knowledge, our paper is the first paper that does CPU-GPU work partitioning at runtime to optimize an energy-related objective using a black-box approach on an integrated CPU-GPU platform. Our scheduler doesn't require detailed knowledge about the processor or its power management, instead characterizes it using a set of micro-benchmarks. It is suitable for the majority of processors produced today: ones that include a PCU and don't allow the programmer to set component frequencies. It doesn't require offline profiling to perform work distribution.

Our contributions are the following:

- We present a new approach to optimizing application power use on integrated CPU-GPU systems that is based on distributing application work between the CPU and

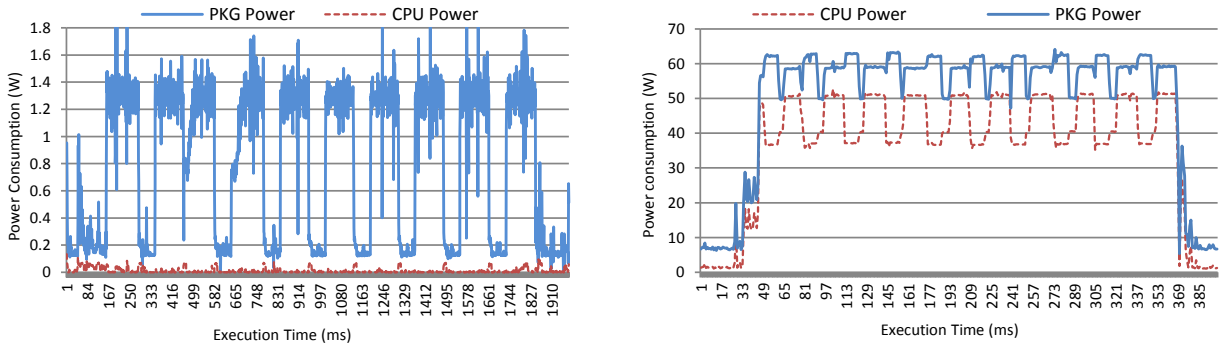

Figure 2: Package and CPU power over time for a memory-bound application with a 90-10% GPU-CPU workload distribution, on a Bay Trail tablet (left) and an Intel Haswell desktop (right).

GPU using information about the processor's power use characteristics and the workload's runtime behavior.

- We present an energy-aware scheduler (*EAS*) that partitions data-parallel work between a CPU and GPU to optimize for any user-defined energy-related metric that can be expressed as a function of power consumption and program execution time.
- We present an experimental evaluation of our EAS technique on two platforms, a high-end desktop system and a low power tablet. Our evaluation uses a set of twelve widely-varying benchmarks that include both regular and irregular workloads. On average, our EAS algorithm yields an energy-delay product that is 96% and 93% of the near-ideal Oracle energy-delay product on the desktop and tablet, respectively.

The layout of the paper is as follows. Section 2 describes our power characterization for integrated CPU-GPU platforms. We describe our energy-aware runtime in Section 3, followed by our implementation details in Section 4. Our empirical evaluation is presented in Section 5. We discuss related work in Section 6 and conclude in Section 7.

2. Power Characterization

In a processor with an integrated GPU, total package (chip) power consumption depends on the work performed by the CPU, the GPU, the LLC (Last-Level Cache), as well as the processor uncore. The uncore includes non-core functions such as memory coherence management and QPI controllers. If a workload is run using both the CPU and GPU simultaneously, package power will often be different than if all work is executed using just the CPU alone or the GPU alone. Moreover, package power typically varies during the execution of a single workload.

The processor's PCU firmware usually controls power by monitoring processor temperature and other readings in order to optimize performance for given power and thermal constraints. As pointed out in the introduction, the PCU's operation is complex and generally varies between SKUs. To model its effects on power consumption, we probe the processor's power use under various workload characteristics. We summarize our key observations below.

First, in order to demonstrate the effect of workload characteristics on power consumption, Figure 3 to the left shows the power consumption of a compute-bound micro-benchmark over time while the chart to the right is for a memory-bound micro-benchmark. Both micro-benchmarks were run on a Haswell desktop system. We expect memory-bound applications to consume different power than compute-bound ones. This behavior is shown by the power charts: the memory-bound benchmark consumes an average of ˜63W while the compute-bound benchmark consumes ˜55W for the duration in which CPU and GPU simultaneously execute. Although this behavior is not PCU specific, a power characterization step needs to capture this in order to optimize power consumption.

Second, in order to demonstrate the subtleties of PCU, we execute the above memory-bound micro-benchmark ten times with 5% of the work on the GPU and the remaining 95% on the CPU on a Haswell desktop system. The power consumption for this setting is shown in Figure 4. As can be seen in the chart, the ten brief GPU executions lower the package power from ˜60W to <˜40W while the GPU executes. Unlike the long GPU execution in Figure 3 (Right) where the package power is not altered during GPU execution, the short-burst of kernel execution affects the package power. This behavior is largely due to the processor's specific power management policy and demonstrates how the length of the kernel execution on a device affects the power consumption.

These examples show how the power consumption typically varies based on the workload's characteristics and the CPU-GPU work partitioning. In particular, a memory-bound application has a different power characteristics than a compute-bound application. Similarly, a short burst of very intense execution on a device affects the PCU's sampling, resulting in different power decisions.

Based on these observations, we capture the subtleties of this processor power variation using a set of *power characterization functions* that predict the average package power

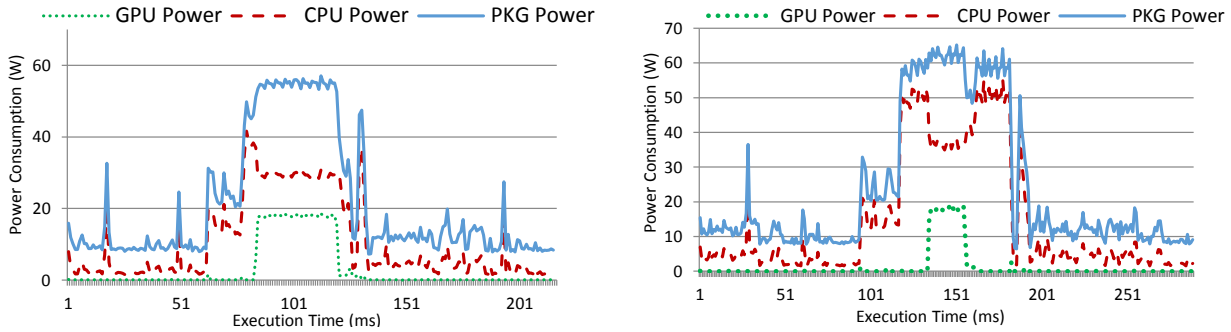

Figure 3: Power consumption over execution time for two long-running micro-benchmarks on an Intel Haswell Desktop. Left is compute-bound and right is memory-bound.

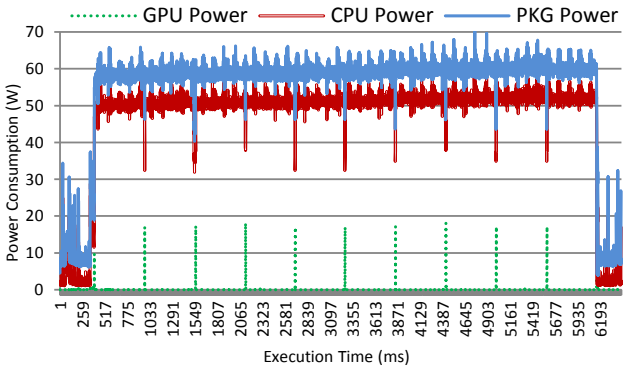

Figure 4: Power consumption over time for a memory-bound micro-benchmark executed 10-times on an Intel Haswell Desktop. At the intervals the GPU is executing, package power drops from ˜60W to $<$ ˜40W.

consumed by the execution of kernels as a function of the CPU-GPU workload distribution. There are eight offload power characterization functions, which are computed using eight different micro-benchmarks. For each micro-benchmark, we measure its energy use by sampling the machine-specific register MSR_PKG_ENERGY_STATUS[1]. From this energy consumption and the execution time, we compute the micro-benchmark's power consumption and then fit a smooth curve to derive a polynomial approximation.

The eight micro-benchmarks represent a cross-product of the following execution characteristics: 1) memory- or compute-bound, 2) short or long execution on the CPU alone, and 3) short or long execution on the GPU alone. The memory-bound micro-benchmark randomly updates memory locations in an array using indices computed at start of execution using a random number generator. Similarly, our compute-bound micro-benchmark repeatedly performs floating point multiply-and-add operations. We probe the power behavior of micro-benchmarks with short- and long-running executions on each device to explore how the processor does with CPU- and GPU-intensive applications. In particular, the *(CPU short, GPU long)* category corresponds to CPU-biased workloads that perform much faster on the CPU than the GPU. Similarly, *(CPU long, GPU short)* corresponds to GPU-biased workloads. The *(CPU long, GPU long)* category captures balanced workloads with a long-running kernel, while *(CPU short, GPU short)* corresponds to balanced short running workloads. To distinguish between short- and long-running executions, we found empirically that a 100 millisecond threshold works well on both of our platforms, and we used this threshold for all our experiments. Ideally, this threshold would be chosen based on the PCU's sampling frequency.

Fig. 5 and Fig. 6 show power characterization functions for the eight micro-benchmarks on an Intel Haswell desktop and an Intel Bay Trail Atom-based tablet, respectively. We found empirically that a sixth-order polynomial was a good fit for the power results we obtained. If we concentrate on the desktop results in Fig. 5, we observe that a short CPU execution results in a convex shape as opposed to a concave shape for long CPU execution. This is expected since on this system CPU alone execution consumes ˜45W whereas GPU alone uses ˜30W for a compute-intensive benchmark, so shorter CPU execution will eventually drop from ˜45W as we increase the GPU offload percentage, thus showing a convex shape. As we increase GPU offload percentage and are within the threshold for short CPU execution, the remaining curve is expected to be relatively flat.

On the other hand, in the Bay Trail tablet results shown in Fig. 6, a compute-bound benchmark consumes 1.5W on the CPU alone as opposed to roughly 2W on the GPU alone. Surprisingly, on this processor having a less powerful GPU, a memory-bound benchmark consumes less power than a compute-bound benchmark: it requires 0.7W on the CPU alone and 1.3W on the GPU alone. Most of the power characterization functions are concave-shaped because the GPU on this platform consumes more power than the CPU.

In our experience, this simple classification into eight categories works surprisingly well to capture subtleties in the power behavior of different workloads. As a result, for a given platform, we use a single power characterization function per category to model the power consumption of all workloads

[1] All our experiments were performed on Windows systems, where administrator privilege is required to read the MSR_PKG_ENERGY_STATUS register.

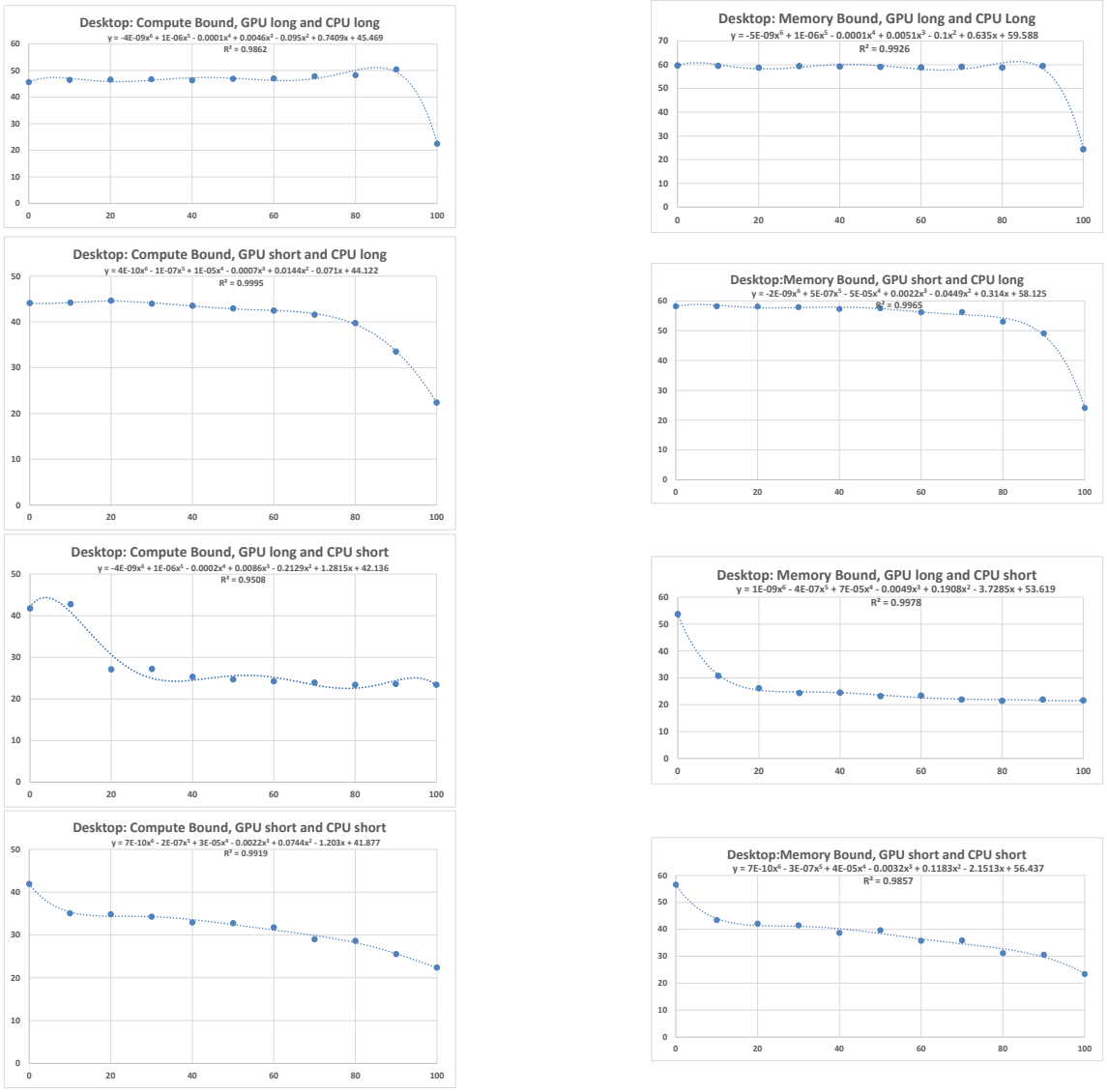

Figure 5: Desktop power characterization on eight categories: X-axis represents GPU offload percent from 0% to 100% and Y-axis represents power consumption in watts. The y-equations show the corresponding sixth-order polynomial.

in that category. Later, when executing an application on the platform, we do an initial online profiling step to monitor the application's execution. This tells us which category the application falls into (i.e., memory- or compute-bounded, short- or long-burst), and so the power characterization function we should use to schedule the application's execution.

3. Energy-Aware Runtime

Given a energy-related metric like the energy-delay product, our energy-aware runtime combines offline power characterization with online profiling to partition work between the CPU and GPU. The online profiling step estimates CPU and GPU execution times, L3 cache miss rate, and total instructions retired to derive its workload classification. We then combine this execution-time measured performance with the appropriate power characterization function (described in Section 2) to determine the work distribution for that energy metric.

3.1 Lightweight Online Profiling

Energy-aware scheduling depends on accurately predicting the throughput of each device of the integrated CPU-GPU processor. We implement the adaptive profiling algorithm described in [12] for this purpose. That is, we use a proxy GPU thread running on a CPU core to control the GPU's operation. We start initially with a shared global pool of work for a kernel, and pick a portion of the work to offload to the GPU that is enough to keep the GPU cores busy. The CPU worker threads continue to pick and execute items from this shared global pool and locally collect profiling information. This profiling

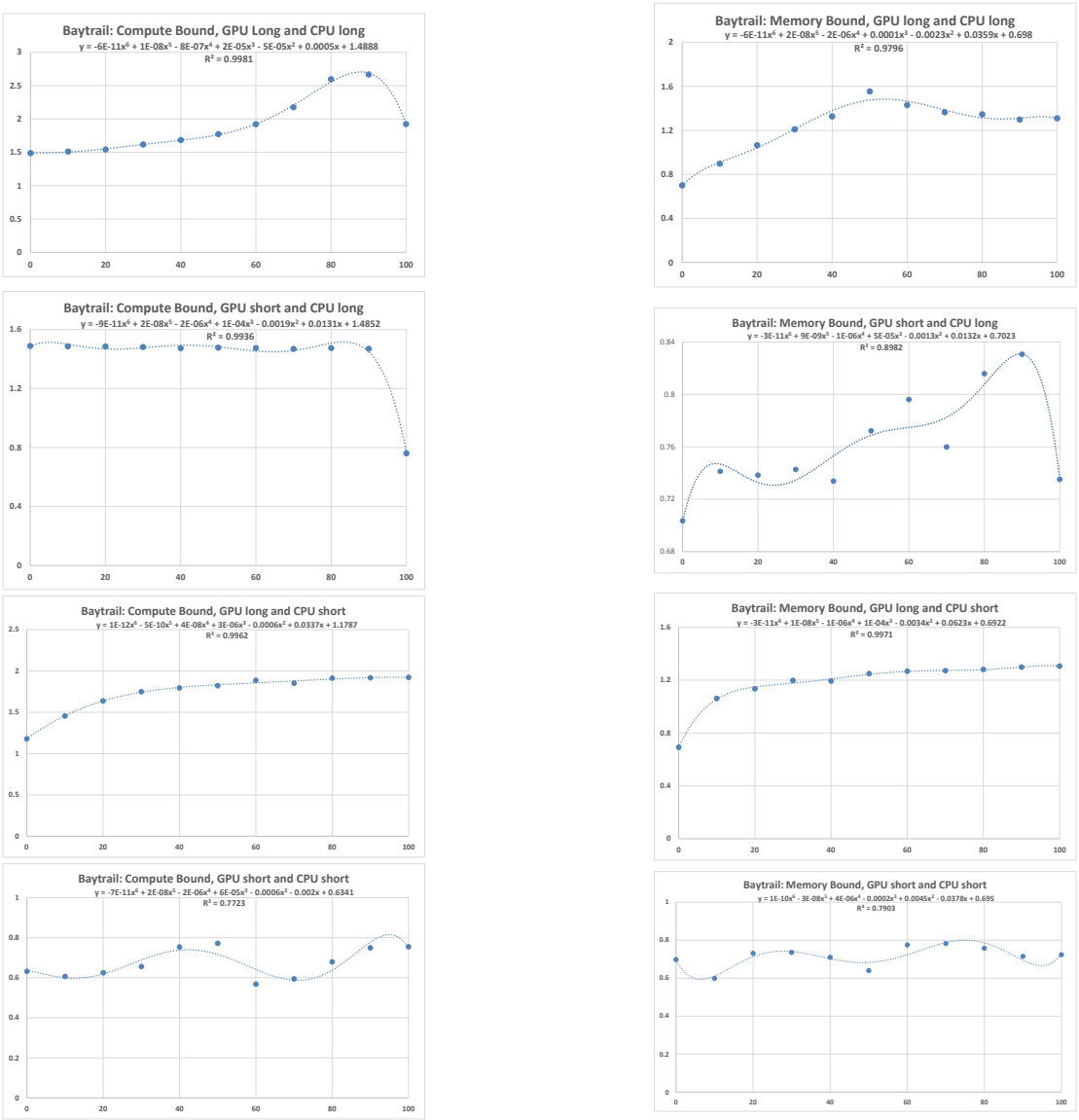

Figure 6: Bay Trail tablet power characterization on eight categories: X-axis represents GPU offload percent from 0% to 100% and Y-axis represents power consumption in watts. The y-equations show the corresponding sixth-order polynomial.

information includes the number of items processed, and amount of time taken. Based on this profiling information, we compute the throughput of each device. We repeat profiling to tackle irregularities in workloads.

We have extended [12]'s adaptive profiling algorithm to measure L3 cache misses and total instructions retired, collected via hardware counters. We compute the ratio of L3 cache misses to the total retired instructions to determine the *memory intensity* of a workload, and classify it as either memory-bound or compute-bound. Similarly, we use the observed execution time in the profiling phase to estimate the total execution time of the workload, and classify the workload as either short-running or long-running. By using these classifications, we determine the appropriate power characterization function for a given workload.

As soon as the GPU proxy thread finishes profiling on the GPU, it gathers profiling information from the CPU workers and computes the GPU offload fraction that minimizes the target energy metric (e.g., energy-use or energy-delay product) for the remaining iterations. Our runtime also records this GPU offload fraction for subsequent kernel executions. If both the devices are kept busy in the profiling phase by having a sufficient number of parallel iterations, the profiling phase is shown to introduce near-zero overhead in [12]. Additionally, for workloads where the same kernel behaves differently over time, we repeat profiling step since our online profiling has low overhead.

3.2 Energy-Aware Algorithm

The online profiling step yields CPU throughput, R_C, and GPU throughput, R_G, when executing in the CPU+GPU configuration, that is, when computation is run on both CPU and GPU at the same time.

Given an energy metric of interest—say total energy use or the energy-delay product—we now explain how we compute the best value for the GPU offload ratio to minimize that metric. The overall strategy is the following: We first use R_C and R_G to derive the execution time function $T(\alpha)$ for a given GPU offload ratio α. $P(\alpha)$ is a sixth-order polynomial chosen based on the characteristics of the workload (as described in Section 2). Once we know $P(\alpha)$ and $T(\alpha)$, we use it to compute the value for the energy metric of interest, e.g., the energy-delay product is $EDP(\alpha) = P(\alpha) * T(\alpha)^2$.

Assume that after finishing the profiling step there remain N iterations to be partitioned between CPU and GPU using a GPU offload ratio α. α will be in the range [0,1], with 0 indicating CPU-alone execution and 1 indicating GPU-alone execution. When $0 < \alpha < 1$, execution starts on both CPU and GPU and then continues on a single device, unless α is such that both devices finish their assigned iterations at the same time. When α is such that the CPU finishes its work first, the CPU's execution time can be computed as $(1-\alpha)N/R_C$, as the CPU processes $(1-\alpha)N$ iterations with a combined throughput R_C. Similarly, when the GPU finishes first, its execution time can be computed as $\alpha N/R_G$. Thus, the total time T_{CG} spent in the combined mode can be computed as

$$T_{CG}(\alpha) = min(\frac{(1-\alpha)N}{R_C}, \frac{\alpha N}{R_G}) \quad (1)$$

When the arguments to the *min* function are equal, the CPU and GPU finish their work at the same time. This observation leads to the following equation for the corresponding α value:

$$\alpha_{PERF} = \frac{R_G}{R_C + R_G} \quad (2)$$

In this equation, α_{PERF} corresponds to the performance-oriented scheduling strategy that aims to minimize energy consumption by maximizing program performance – choosing α_{PERF} as an offload ratio minimizes program execution time.

In the combined mode, the CPU and GPU work together with a combined throughput of $R_C + R_G$. Thus, they process $T_{CG} * (R_C + R_G)$ iterations. This leaves N_{rem} iterations to be processed by a single device:

$$N_{rem}(\alpha) = N - T_{CG}(\alpha) * (R_C + R_G) \quad (3)$$

These remaining iterations take $N_{rem}(\alpha)/R_C$ or $N_{rem}(\alpha)/R_G$ time to process depending on whether they are executed on the CPU or GPU. Putting this all together, we get the following formula for the total execution time to process N iterations:

$$T(\alpha) = \begin{cases} T_{CG}(\alpha) + \frac{N_{rem}(\alpha)}{R_G}, & \text{if } \alpha \geq \alpha_{PERF} \\ T_{CG}(\alpha) + \frac{N_{rem}(\alpha)}{R_C}, & \text{if } \alpha \leq \alpha_{PERF} \end{cases} \quad (4)$$

Once we know $T(\alpha)$ we can use it to compute the total energy $E(\alpha) = P(\alpha) * T(\alpha)$, the energy delay product $EDP(\alpha) = P(\alpha) * T(\alpha)^2$, or any other metric based on the combination of package power and execution time. At runtime, we find the α that minimizes the chosen target function, $E(\alpha)$ or $EDP(\alpha)$, by simply evaluating the target function on a range of values between 0 and 1 in certain increments (e.g., 0.1 or 0.05 for 10% or 5% increments of the offload ratio, respectively). In our experience, this evaluation takes negligible time compared to total program execution.

```
1  function EAS ()
     Input  : f: Function pointer to CPU code; ocl: OpenCL kernel for
              f; N: number of parallel iterations; G: global table to store f to α
              mappings; OBJ: objective function in terms of energy-use or EDP
2    if a mapping for f exists in G then
       //multiple invocations of f
3      Determine α for f from G;
4      Set N_rem to N;
5    else
       //first run for f
6      if N < GPU_PROFILE_SIZE then
7        Set α to 0;
8        Execute f on multi-core CPU alone;
9        Save α for f in G;
10       return;
11     Set shared_counter to N;
12     Set N_rem to N;
       //Repeat profiling for half of the iterations
13     while N_rem > N/2 do
14       Invoke OnlineProfile ();
15       Obtain CPU time, GPU
         time, CPU processed iterations, and GPU processed iterations;
16       Compute R_C, and R_G;
17       Obtain hardware counters for L3
         & LLC Cache Miss, and Total Instructions Retired from profiling;
18       Characterize
         workload: memory-bound or compute-bound, Long or Short;
19       Determine corresponding power curve;
20       Find α by
         computing the target function OBJ with α set to values in [0..1]
         in increments of 0.1, and determine the minimum OBJ value;
21       Record α in G for f;
22       Update N_rem to remaining items in shared_counter;
23   Offload α * N_rem items to the GPU;
24   Execute (1−α) * N_rem items on the CPU using work-stealing;
25   Wait for CPU and GPU completion;
26   Accumulate
     α across multiple invocations via sample-weighted technique from [12];
27   return;
28   function OnlineProfile ()
     Input  : f: Function pointer to CPU code;
     Output: CPU time, GPU time, CPU
             items, GPU items, L3 & LLC Cache Miss, and Total Inst Retired
29     Set up profiling counters for timers and cache misses;
30     Execute f on multi-core
       CPU asynchronously by atomically grabbing work from shared_counter;
31     Offload GPU_PROFILE_SIZE items from shared_counter to GPU;
32     Wait for GPU completion;
33     Terminate CPU workers;
34     Stop profiling counters for timers and cache misses;
35     return
```

Figure 7: Energy-aware scheduling (EAS) algorithm

The details of the energy-aware scheduling algorithm are shown in Fig. 7. The argument G to the EAS algorithm is a global runtime table that stores a mapping from a CPU func-

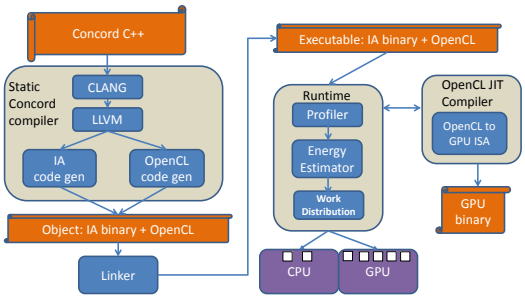

Figure 8: Compiler and runtime

tion pointer f to it's corresponding computed GPU offload ratio α. For multiple invocation of the same kernel, we reuse the α value from G (shown in Steps 2-4). When N is too small, we execute all the work on the multi-core CPU (shown in Steps 7-10). For first-time seen kernels, we perform repeated profiling using the OnlineProfile function (shown in Steps 28-35) for half of the iterations. This is the best strategy reported in [12], called *size-based* strategy, and this strategy works well in practice as it deals with application load imbalance, reflects CPU-GPU memory contention, and because profiling overhead is low, has negligible impact on performance. The *GPU_PROFILE_SIZE* parameter must be chosen carefully based on the available GPU parallelism on a given platform. For example, in our desktop platform it is set to 2048 as it has 20 execution units (EUs), 7 threads per EU, with each thread being 16-way SIMD, resulting in 2240-way parallelism. Step 20 executes a loop that determines α by iterating over the range $[0,1]$ in steps of 0.1 to determine the minimum value of the target energy function. Based on this α, we execute $\alpha * N_{rem}$ iterations on the GPU and $(1-\alpha) * N_{rem}$ on the CPU. Finally, the sample-weighted technique described in [12] is used to accumulate α across multiple invocations of the same kernel.

4. Implementation

To perform fine-grained work distribution for energy efficiency, we leverage the heterogeneous C++ programming framework Concord [2], and implement automatic partitioning of work across CPU and GPU cores. Concord provides a data-parallel *parallel_for* loop construct. Each loop might execute once or multiple times. The loop iterations are independent of each other and can be executed in arbitrary order on either the CPU or the GPU. Note that our energy-aware scheduling technique is not tied to C/C++, and could be deployed transparently for other languages including managed languages like Java.

Figure 8 shows our integration of energy-aware scheduling framework to Concord. Our runtime implements work-stealing on the CPU, with one CPU worker thread (the GPU proxy thread) offloading work to the GPU. The GPU offload fraction α that minimizes the target energy metric is computed after profiling by the GPU proxy thread, which then distributes the remaining parallel iterations among the CPU and GPU cores.

5. Experimental Evaluation

We now present an evaluation of our energy-aware scheduling algorithm on two platforms. We compare our results with the CPU-alone, GPU-alone, best-performance, and Oracle approaches.

Environment: The two platforms we used are:

1. A desktop computer with a 3.4GHz Intel 4^{th} Generation Core i7-4770 Processor with four CPU cores and hyper-threading enabled. The integrated GPU, an Intel HD Graphics 4600, has 20 execution units (EUs), each with 7 16-wide SIMD hardware threads, running at a turbo-mode clock speed from $350MHz$ to $1.2GHz$. The system has 8GB memory and is running 64-bit Windows 7.
2. A tablet with a 1.33GHz Intel Atom CPU Z3740 Processor with four CPU cores. The integrated GPU has 4 execution units (EUs), each with seven 16-wide SIMD hardware threads, running at a turbo-mode clock speed from $331MHz$ to $667MHz$. The system has 2GB memory and is running 32-bit Windows 8.1.

Benchmarks: We used a diverse set of applications spanning a spectrum of application domains and runtime characteristics: single kernel invocation vs. multiple invocations, and regular vs. irregular execution. We classify workloads as irregular if they exhibit input-dependent control flow, e.g., graph algorithms. Most of these applications were ported from existing sources: TBB [10] and Parsec [4]. We also developed some applications from scratch.

The compile-time and runtime characteristics for our workloads are provided in Table 1. The benchmarks we use include seven irregular workloads (**BH**, **BFS**, **CC**, **FD**, **MB**, **SL**, **SP**) and five regular workloads (**BS**, **MM**, **NB**, **RT**, **SM**). All the workloads have one data-parallel kernel. Benchmarks such as **BH**, **MB**, **SL**, **MM**, and **RT** invoke the kernel just once whereas the other benchmarks invoke the kernel multiple times. The number of kernel invocations is shown in Column 5. Columns 7-9 show workload characterization in terms of memory-bound/compute-bound, CPU Short/Long, and GPU Short/Long.

We classified workloads as Short/Long based on their execution. If the estimated execution time for the remaining iterations (N_{rem} in Algorithm 7) after profiling is less than 100 ms, we classify the workload as Short. Furthermore, we classify a workload as memory-bound if its ratio of L3 cache misses to total load/store instructions retired is greater than 0.33. Note that, both these thresholds were sufficient for both platforms and for the twelve regular and irregular workloads we studied. Although more accurate prediction of these thresholds can be done via hardware performance counter readings (such as [19]), we leave this for future work.

Currently, our 32-bit Bay Trail tablet can only execute seven of our workloads[2]. Additionally, the OpenCL driver on

[2] The remaining workloads do not successfully compile on 32-bit mingw and CLANG due to missing definitions in *alloca.h* and *xmmintrinsics.h*.

Name	Abbrv.	Input (Desktop)	Input (Tablet)	Num. invocations	Reg. (R)/Irreg. (IR)	Compute (C)/Mem (M)	CPU Short (S)/Long (L)	GPU Short (S)/Long (L)
BarnesHut	BH [3]	1M bodies, 1 step	N/A	1	IR	M	L	L
Breadth first search	BFS	W-USA ($\|V\|$=6.2M, $\|E\|$=1.5M)	N/A	1748	IR	M	S	S
Connected Component	CC	W-USA ($\|V\|$=6.2M, $\|E\|$=1.5M)	N/A	2147	IR	M	S	S
Face Detect	FD [18]	3000 by2171 Solvay-1927	N/A	132	IR	C	S	S
Mandelbrot	MB	image 7680x6144	image 7680x6144	1	IR	M	L	L
SkipList	SL	500M keys	45M keys	1	IR	M	L	L
Shortest Path	SP	W-USA ($\|V\|$=6.2M, $\|E\|$=1.5M)	N/A	2577	IR	M	S	S
Blackscholes	BS [4]	64K	2621440	2000	R	C	S	S
Matrix Multiply	MM	2048 by 2048	1024x1024	1	R	C	L	L
N-Body	NB	4096 bodies	1024 bodies	101	R	C	L	S
Ray Tracer	RT	sphere=256,material=3,light=5	sphere=225,material=3,light=5	1	R	C	L	L
Seismic	SM [10]	1950 by 1326, 100 frames	1950 by 1326, 100 frames	100	R	M	S	S

Table 1: Compile-time and runtime statistics of benchmarks.

this device restricts the memory region we can share between the CPU and GPU to only 250MB, so we execute smaller input sizes for these seven workloads, as shown in Column 4 of Table 1.

In our runtime, we test GPU performance counter A26 on both platforms to check if it is busy. In that case, we execute the application entirely on the CPU. Also, our runtime uses the Intel Performance Counter Monitor tool [9] to measure L3 cache misses and total instructions retired during profiling.

Comparison schemes: We compare the following energy-aware scheduling strategies to distribute the parallel iterations between the CPU and GPU:

1. **CPU**: Multi-core CPU execution based on Intel Thread Building Blocks (TBB).
2. **GPU**: GPU-alone execution using the vendor OpenCL driver.
3. **Oracle**: The best energy metric (either energy-use or energy-delay product) we found through exhaustive search of possible GPU offload ratios α in the range $[0,1]$ with 0.1 increment. This is the baseline in our evaluation.
4. **PERF**: The best-performance strategy corresponds to the workload distribution which yields the best execution time by using both CPU and GPU simultaneously.
5. **EAS**: Our energy-aware scheduling algorithm using the analytical power modeling and online profiling (described in Sec. 3).

Online profiling overhead: Our online profiling along with the sample-weighted accumulation strategy (described in Section 3.2) incurs very little overhead, i.e., on average 1-2 microseconds on both the platforms.

Desktop Results: Energy-delay product is an important metric as it balances performance with energy reduction. We report the energy-delay product efficiency results for our benchmarks in Figure 9. On this system, GPU execution is

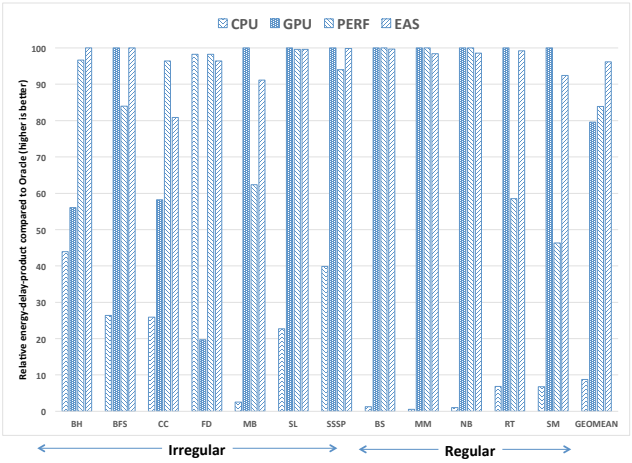

Figure 9: Relative energy-delay product efficiency compared to Oracle on the desktop. Oracle is at 100% (higher is better).

generally more energy-efficient that CPU execution, so it is not surprising to see that that **GPU** yields 79.6% of **Oracle**. The performance-oriented scheduling algorithm, **PERF**, improves upon both GPU-alone and CPU-alone execution and is at 83.9% of **Oracle**. This is because hybrid CPU+GPU execution on this system generally yields much better runtime performance than single device execution. Our **EAS** scheduler performs the best, at 96.2% of **Oracle**.

Compared to **PERF**, **EAS** performs comparably or better for most workloads except for **CC**. For this workload, we observe that **EAS** determines a GPU offload ratio of 1.0 as opposed to Oracle's 0.9. Our online profiling does not accurately predict the best GPU offload ratio, primarily due to this workload's high irregularity, even though we are able to do well for other irregular graph workloads such as **BFS** and **SSSP**. A possible solution is to increase the profiling sampling

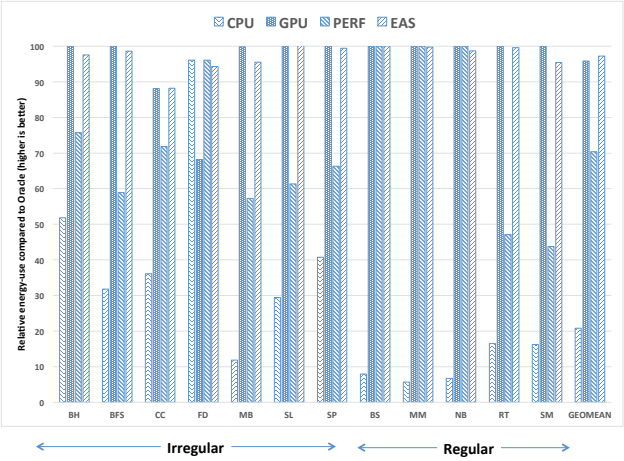

Figure 10: Relative energy-use efficiency compared to Oracle on the desktop. Oracle is at 100% (higher is better).

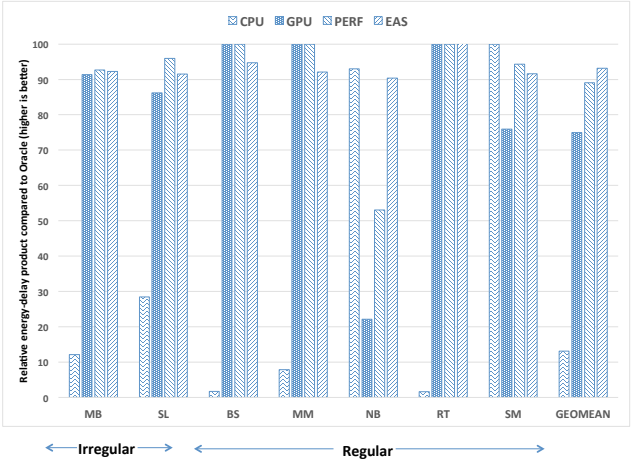

Figure 11: Relative energy-delay product efficiency compared to Oracle on the Bay Trail. Oracle is at 100% (higher is better).

rate to improve the accuracy for this workload. We intend to investigate this as part of our future work.

We present the desktop's total energy-use efficiency results in Figure 10. Since the GPU is 2-3× more power efficient than the CPU on this system, we observe that GPU-alone execution is very close to **Oracle**, on average 95.8%, while the best-performance strategy, **PERF**, is only able to achieve 70.4% of **Oracle**. By dividing work between the CPU and GPU, **PERF** achieves better execution performance but consumes enough extra power during that execution to increase overall energy use. **EAS**, which aims to optimize energy rather than runtime performance, is able to perform slightly better than GPU-alone execution with 97.2% of **Oracle**. For the CPU-biased **FD** workload, our **EAS** algorithm is able to determine 100% CPU execution, while GPU-alone execution suffers significantly.

Bay Trail Results: Earlier, we observed that using the GPU alone on the desktop consumes lower package power than using just the multicore CPU. However, using the GPU alone on the Bay Trail usually increases package power. That is, the power chart to the top-left of Fig. 6 shows that the CPU consumes about 1.5W and the GPU consumes about 2W power.

As observed in Figure 11, our **EAS** algorithm performs very well on this platform by achieving an average energy-delay product of 93.2% of **Oracle**. This is 4.4% better than **PERF**, 19.6% better than GPU-alone execution, and 85.9% better than CPU-alone execution. Similarly for the total energy-use metric, **EAS** yields on average 96.4% of **Oracle** (as shown in Figure 12). This result is 7.5% better than **PERF**, 10.1% better than GPU-alone execution, and 57.2% better than CPU-alone execution. Using the GPU on the Bay Trail usually speeds up the workload enough to overcome the greater power consumed by its GPU.

Summary: In conclusion, our results show that:

- GPU-alone execution may not be the best strategy for energy optimization on all platforms. On the desktop, although GPU execution results in total energy-use that is 95.8% of **Oracle** (best), it yields only 79.5% of Oracle when the energy-delay product metric is used. On the Bay Trail, GPU-alone execution yields worse results for both energy-use and energy-delay product.

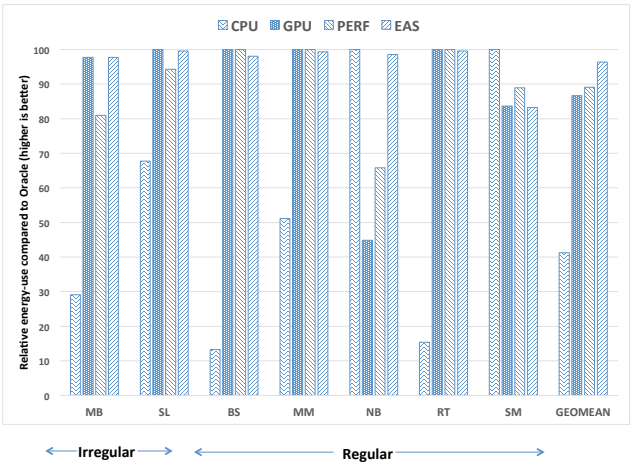

Figure 12: Relative energy-use efficiency compared to Oracle on the Bay Trail. Oracle is at 100% (higher is better).

- Performance-aware work partitioning, **PERF**, is not the best strategy for energy optimization on either platform. In particular, it does not perform well on the desktop when energy-use is the metric: its performance is 70.4% of **Oracle**. For energy-delay product metric, **PERF** is about 83.9% of Oracle, leaving an opportunity for another 16.1% of energy improvement.
- However, **EAS** is the best strategy for both the energy-use and energy-delay product metrics. It results in an energy-delay product that is 96.2% and 93.2% of the near-ideal

Oracle result on the desktop and tablet, respectively. The benefits of the **EAS** algorithm are primarily due to its use of offline platform power characterization combined with online profile-based workload characterization. Given the complexities in modern PCU operation, our results demonstrate that black-box platform power characterization can indeed be very useful.

6. Related Work

There have been many efforts to make heterogeneous execution of applications more efficient. As energy and power consumption became first-order constraints in mobile system and large-scale heterogeneous system design, more and more research work on power and energy optimizations for heterogeneous systems has emerged.

Heterogeneous Execution for Performance: *Qilin* [15] and *StarPU* [1] support adaptive mapping to run a function across CPU and GPU. [14] relies on static analysis of kernels to determine workload distribution across multiple devices. [20] presents a compiler and runtime to address portable performance across heterogeneous systems. [12] shows several adaptive heterogeneous scheduling algorithms for integrated CPU-GPU systems. Dandelion [21] provides a unified programming model for heterogeneous systems.

Heterogeneous System Energy Modeling: [8] achieves high accuracy in modeling power consumption and temperature in a GPU through using micro-benchmarks to determine power consumption of different components. [6] models the energy consumption of a CPU and discrete GPU system by counting the energy consumption of different components. In [11], the authors build a neural network model to estimate optimal energy efficiency for concurrent kernel execution, while in [26], the authors estimate GPU performance and power using machine learning. All of the above works explore power consumption and energy efficiency in the context of discrete GPUs.

Heterogeneous Energy-Aware Management: A price theory based power management framework for heterogeneous multi-cores is presented in [23]. [5] explores mapping of data-parallel programs to heterogeneous multi-cores using different criteria such as performance, power and energy, and develops an approach to select the best partitioning for a given program based on its profiling result on different cores. [16] proposes a two-tier holistic energy management framework for CPU-GPU heterogeneous architectures. Hoffman [7] describes several heuristics to solve the problem of assigning computing resources in order to meet a deadline with the least energy consumption. Kim et al [13] characterize the performance and energy-efficiency impact of using processors with integrated GPUs to run server workloads. [22] provides a runtime system to enable offloading machine learning algorithms to DSPs for energy efficiency. [24] studies the effect of different mapping techniques on energy consumption for heterogeneous CPU-GPU systems and shows that using software pipelining can improve energy efficiency. [17] shows the effectiveness of DVFS on energy savings for discrete GPUs, while [19] implements a coordinated energy management algorithm for integrated CPU-GPU systems, which relies on coordinating DVFS states for both the CPU and GPU. Similarly, [25] requires programmer-exposed DVFS knobs to control runtime power management for GPGPU applications.

In contrast to the efforts above, we propose a black-box, user-level approach for improving energy efficiency on integrated CPU-GPU platforms that combines a offline power model with online workload characterization to partition work across devices. Most existing integrated processors from Intel and AMD don't allow the programmer to control DVFS settings. Thus, we focus on characterizing the power behavior of workloads on these platforms without relying on internal knowledge of the PCU's power management techniques and strategies. Afterwards, we use the power characterization functions and online profiling at runtime to improve the energy efficiency of a GPGPU application.

7. Conclusion

Energy efficiency will continue to be a key design goal for all computing systems. In this paper, we investigated the problem of improving the energy efficiency of processors with integrated GPUs by finding an appropriate distribution of work between the CPU and GPU. We presented a methodology for characterizing the integrated CPU-GPU processor power behavior by relating it to the workload distribution between the CPU and GPU components for a set of representative micro-benchmarks. This characterization only needs to be done once for each processor. We use this power use characterization to optimize the energy use of applications. Our runtime scheduler leverages this characterization, as well as results from lightweight online profiling, to determine the work distribution for an application that optimizes a user-specified energy-related metric such as the energy-delay product or total energy-use.

We presented an experimental evaluation of our scheduling technique on two widely-different platforms, a high-end desktop system and a low power tablet, each executing a diverse set of twelve benchmarks that contain a mix of regular and irregular code. On average, our algorithm yields an energy-delay product that is 96% and 93% of the near-ideal Oracle energy-delay product on the desktop and tablet, respectively. Given the complexities of modern processor hardware power management, our results demonstrate that a black-box, user-level approach to optimizing energy efficiency can be surprisingly effective. In future, we would like to incorporate feedback from our user-level runtime in power management techniques.

References

[1] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous

multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[2] R. Barik, R. Kaleem, D. Majeti, B. Lewis, T. Shpeisman, C. Hu, Y. Ni, and A.-R. Adl-Tabatabai. Efficient mapping of irregular C++ applications to integrated GPUs. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2014.

[3] J. Barnes and P. Hut. A hierarchical $O(N \log N)$ force calculation algorithm. *Nature*, 324:446–449, 1986.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques (PACT)*, pages 72–81, NY, USA, 2008.

[5] K. Chandramohan and M. F. O'Boyle. Partitioning data-parallel programs for heterogeneous mpsocs: Time and energy design space exploration. In *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES)*, pages 73–82, 2014.

[6] R. Ge, X. Feng, M. Burtscher, and Z. Zong. PEACH: A Model for Performance and Energy Aware Cooperative Hybrid Computing. In *Proceedings of the 11th ACM Conference on Computing Frontiers (CF)*, pages 24:1–24:2, 2014.

[7] H. Hoffmann. Racing and pacing to idle: An evaluation of heuristics for energy-aware resource allocation. In *Proceedings of the Workshop on Power-Aware Computing and Systems (HotPower)*, pages 13:1–13:5, 2013.

[8] S. Hong and H. Kim. An integrated GPU power and performance model. *SIGARCH Comput. Archit. News*, 38(3): 280–289, June 2010.

[9] Intel Performance Counter Monitor. URL `https://software.intel.com/en-us/articles/intel-performance-counter-monitor`.

[10] Intel Thread Building Blocks. URL `https://www.threadingbuildingblocks.org/`.

[11] Q. Jiao, M. Lu, H. P. Huynh, and T. Mitra. Improving GPGPU Energy-efficiency Through Concurrent Kernel Execution and DVFS. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–11, 2015.

[12] R. Kaleem, R. Barik, T. Shpeisman, B. Lewis, C. Hu, and K. Pingali. Adaptive Heterogeneous Scheduling on Integrated GPUs. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2014.

[13] S. Kim, I. Roy, and V. Talwar. Evaluating integrated graphics processors for data center workloads. In *Proceedings of the Workshop on Power-Aware Computing and Systems (HotPower)*, pages 8:1–8:5, 2013.

[14] J. Lee, M. Samadi, Y. Park, and S. Mahlke. Transparent CPU-GPU collaboration for data-parallel kernels on heterogeneous systems. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques (PACT)*, 2013.

[15] C.-K. Luk, S. Hong, and H. Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 45–55, 2009.

[16] K. Ma, X. Li, W. Chen, C. Zhang, and X. Wang. GreenGPU: A Holistic Approach to Energy Efficiency in GPU-CPU Heterogeneous Architectures. In *Proceedings of the 2012 41st International Conference on Parallel Processing (ICPP)*, pages 48–57, 2012.

[17] X. Mei, L. S. Yung, K. Zhao, and X. Chu. A Measurement Study of GPU DVFS on Energy Conservation. In *Proceedings of the Workshop on Power-Aware Computing and Systems*, HotPower '13, pages 10:1–10:5, 2013.

[18] OpenSource Computer Vision Library. URL `http://sourceforge.net/projects/opencvlibrary/`.

[19] I. Paul, V. Ravi, S. Manne, M. Arora, and S. Yalamanchili. Coordinated energy management in heterogeneous processors. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 59:1–59:12, 2013.

[20] P. M. Phothilimthana, J. Ansel, J. Ragan-Kelley, and S. Amarasinghe. Portable performance on heterogeneous architectures. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems (ASPLOS)*, pages 431–444, 2013.

[21] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly. Dandelion: a compiler and runtime for heterogeneous systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, pages 49–68, NY, USA, 2013.

[22] C. Shen, S. Chakraborty, K. R. Raghavan, H. Choi, and M. B. Srivastava. Exploiting processor heterogeneity for energy efficient context inference on mobile phones. In *Proceedings of the Workshop on Power-Aware Computing and Systems (HotPower)*, pages 9:1–9:5, 2013.

[23] T. Somu Muthukaruppan, A. Pathania, and T. Mitra. Price theory based power management for heterogeneous multi-cores. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 161–176, 2014.

[24] E. Totoni, M. Dikmen, and M. J. Garzarán. Easy, fast, and energy-efficient object detection on heterogeneous on-chip architectures. *ACM Trans. Archit. Code Optim.*, 10(4): 45:1–45:25, Dec. 2013.

[25] H. Wang, V. Sathish, R. Singh, M. J. Schulte, and N. S. Kim. Workload and power budget partitioning for single-chip heterogeneous processors. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 401–410, 2012.

[26] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou. GPGPU performance and power estimation using machine learning. In *IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 564–576, Feb 2015.

[27] Q. Wu, M. Martonosi, D. W. Clark, V. J. Reddi, D. Connors, Y. Wu, J. Lee, and D. Brooks. A dynamic compilation framework for controlling microprocessor energy and performance. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 271–282, 2005.