Adaptive Heterogeneous Scheduling for Integrated GPUs

Rashid Kaleem*
Dept. of Computer Science
University of Texas at Austin
rashid@cs.utexas.edu

Rajkishore Barik
Intel Labs
Santa Clara, CA
rajkishore.barik@intel.com

Tatiana Shpeisman
Intel Labs
Santa Clara, CA
tatiana.shpeisman@intel.com

Brian T. Lewis
Intel Labs
Santa Clara, CA
brian.t.lewis@intel.com

Chunling Hu
Intel Labs
Santa Clara, CA
chunling.hu@intel.com

Keshav Pingali
Dept. of Computer Science
University of Texas at Austin
pingali@cs.utexas.edu

ABSTRACT

Many processors today integrate a CPU and GPU on the same die, which allows them to share resources like physical memory and lowers the cost of CPU-GPU communication. As a consequence, programmers can effectively utilize both the CPU and GPU to execute a single application. This paper presents novel adaptive scheduling techniques for integrated CPU-GPU processors. We present two online profiling-based scheduling algorithms: *naïve* and *asymmetric*. Our asymmetric scheduling algorithm uses low-overhead online profiling to automatically partition the work of data-parallel kernels between the CPU and GPU without input from application developers. It does profiling on the CPU and GPU in a way that doesn't penalize GPU-centric workloads that run significantly faster on the GPU. It adapts to application characteristics by addressing: 1) load imbalance via irregularity caused by, e.g., data-dependent control flow, 2) different amounts of work on each kernel call, and 3) multiple kernels with different characteristics. Unlike many existing approaches primarily targeting NVIDIA discrete GPUs, our scheduling algorithm does not require offline processing.

We evaluate our asymmetric scheduling algorithm on a desktop system with an Intel 4^{th} Generation Core Processor using a set of sixteen regular and irregular workloads from diverse application areas. On average, our asymmetric scheduling algorithm performs within 3.2% of the maximum throughput with a CPU-and-GPU oracle that always chooses the best work partitioning between the CPU and GPU. These results underscore the feasibility of online profile-based heterogeneous scheduling on integrated CPU-GPU processors.

*The work presented in this paper has been supported by NSF grants CCF 1337281, CCF 1218568, ACI 1216701, and CNS 1064956.

PACT'14, August 24–27, 2014, Edmonton, AB, Canada.

http://dx.doi.org/10.1145/2628071.2628088 .

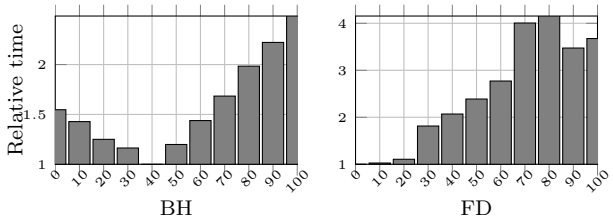

Figure 1: Relative execution time of **BH**-BarnesHut (left) and **FD**-Facedetect (right) as ratio of work offloaded to the GPU is varied from 0% to 100% in increments of 10% (*lower is better*). **BH** is optimal at 40% and **FD** is optimal at 0%.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management—*Scheduling*; D.1.3 [**Software**]: Programming—*Parallel programming*

Keywords

Heterogeneous computing; integrated GPUs; scheduling; load balancing; Irregular applications;

1. INTRODUCTION

In many modern processors such as Intel's IvyBridge and Haswell processors, and AMD's Trinity and Richland processors, the GPU is resident on the same die as the CPU and has the same address space. Such processors present new challenges to application developers, as achieving maximum performance often requires simultaneous use of both CPU and GPU. Moreover, the optimal division of work between the CPU and GPU is very application dependent. As an example, consider Fig. 1, which shows the performance of Barnes-Hut (**BH**) and Face detection (**FD**) on an Intel Haswell machine as the proportion of work assigned to the GPU is varied from 0% to 100% (the applications and the machine are described in more detail in Sec. 7). The y-axis in both graphs shows execution time normalized to the best running time for each application. For **BH**, the best running time is obtained when 40% of the work is done on the GPU. **FD** in contrast does not benefit at all from GPU execution and performs best when run on multicore CPU.

Ideally, the scheduler should partition work between CPU and GPU automatically and efficiently without any input from the application developer. The division of work between the CPU and a GPU has been the subject of a num-

ber of prior studies [4,12,18,22,24,32]. Their techniques fall into three broad categories:

- *Off-line training* [18,22,24]: Train the runtime scheduling algorithm on input data set and then use the previously obtained information on the real run. The success of this approach depends crucially on how accurately the training reflects what happens during the real run. Moreover, the training must be repeated for each new platform.
- Use a *performance model* [4,19]: Accurate performance models are notoriously difficult to construct, particularly for irregular workloads since runtime behavior is very dependent on characteristics of the input data.
- Extend standard *work-stealing* with restrictions on stealing [12, 32]: Since the GPU cannot initiate communication with the CPU, only extensions where work is pushed to GPUs can address load imbalance. Such approaches incur overheads on the CPU execution since the CPU has to act on behalf of the GPU workers.

Most of the prior work has been performed in the context of discrete GPUs, where data must be communicated between CPU memory and GPU memory over slow interconnect such as the PCIe bus. This high-latency communication limits the CPU to offload relatively coarse-grain tasks to the GPU. Integrated GPUs dramatically reduce the cost of data communication between CPU and GPU, so finer grain work sharing between the CPU and GPU than has been explored by prior work becomes possible.

In this paper, we present novel scheduling techniques for integrated CPU-GPU processors that leverage online profiling. Our techniques profile a fraction of the work-items on each device and decide how to distribute the workload between the CPU and GPU based on the measured device execution rates. Because our algorithm is fully online, it does not require any prior training and carries no additional overhead when applied to applications with new data sets or new platforms.

While seemingly simple, scheduling based on online profiling must avoid several pitfalls to produce good results. *First*, it should perform the profiling with near zero overhead, effectively utilizing all available resources, as otherwise, the profiling cost might significantly reduce the benefit of subsequent heterogeneous execution. *Second*, it should accurately measure the execution ratio of different devices, which might be non-trivial in the presence of load imbalance that often occurs in irregular applications. *Finally*, it should be able to effectively handle diverse realistic workloads including those with multiple kernels and multiple invocations of the same kernel, accounting for the fact that optimal execution might require different CPU/GPU partitioning of the different kernel invocations or different kernels.

Our approach addresses these challenges and demonstrates that, if implemented correctly, profile-based scheduling has very little overhead and produces an efficient execution strategy which competes with an offline *Oracle* model. We do note that these benefits are for integrated GPU systems where the offload cost is very small.

Specifically, our contributions are as follows:

1. We present a simple *naïve* profiling based scheduling algorithm that partitions work between CPU and GPU cores of an integrated GPU system without requiring any application-specific information from the application writer. We analyze the overhead associated with such a scheme.
2. We improve our naïve profiling scheme and devise a novel *asymmetric scheduling* algorithm that eliminates some of the overheads associated with naïve profiling. We show that asymmetric profiling introduces zero overhead in the profiling phase.
3. Furthermore, we extend our asymmetric scheduling algorithm with several adaptive schemes to address load imbalance due to irregularities within a *parallel_for* loop and also across multiple *parallel_for* invocations.
4. We present an experimental evaluation of our algorithms on an Intel 4^{th} Generation Core Processor using an extensive set of *sixteen* benchmarks which comprise of a mix of regular and irregular code. On average, our asymmetric scheduling algorithm performs within 5.8% of the maximum throughput obtained by an exhaustive search algorithm. Furthermore, our asymmetric algorithm with adaptivity can further improve the efficiency and achieves within 3.2% of the exhaustive search algorithm.

The layout of the paper is as follows. Sec. 2 outlines the problem statement for heterogeneous execution and discusses it's complexity. Sec. 3 summarizes our programming model and runtime. Sec. 4 describes our naïve profiling algorithm and analyzes it's overhead. Sec. 5 describes our asymmetric profiling scheduling algorithm. We describe the details of our implementation in Sec. 6. This is followed by the evaluation of the proposed schemes as well as a brief description of the benchmarks and hardware in Sec. 7 followed by discussion of the results. We discuss related work in Sec. 8 and conclude in Sec. 9.

2. PROBLEM STATEMENT

The heterogeneous programming framework we use provides a *parallel_for* loop construct for the data-parallel execution of independent loop iterations. Our applications have one or more data-parallel *parallel_for* loops. We would like to determine the optimal partitioning of loop iterations between the CPU and the GPU that minimizes the execution time of the *parallel_for*.

Why is this hard? Several factors make automatic heterogeneous scheduling challenging for CPU+GPU processors:

- The CPU and GPU have different device characteristics. CPU cores are typically out-of-order, have sophisticated branch predictors, and use deep cache hierarchies to reduce memory access latency. GPU cores are typically in-order, spend their transistors on a large number of ALUs, and hide memory latency by switching between hardware threads. This dissimilarity leads to significant differences in execution performance. Certain applications may execute significantly faster on one device than the other. As a result, executing even a small amount of work on the slower device may hurt performance.
- Certain *parallel_for* iterations may take more time than others. Without apriori information about the loop's behavior, it is hard to optimally divide work between the CPU and GPU automatically.
- Iterations may show execution irregularity due to, e.g., data-dependent control flow operations. Without knowl-

edge of the input data, it is hard to determine the optimal partitioning.

- There can be more than one *parallel_for* in an application and each may be invoked more than once, as in the second *parallel_for* in Fig. 2. Since one parallel loop can have the side effect of warming the cache for a second loop, understanding the interactions among the different parallel loops is important to optimally partition work. Similarly, interactions between the multiple invocations of the same parallel loop can also be important for optimal work scheduling.

As a result, devising an automatic heterogeneous scheduling algorithm to handle all these situations is a daunting task. In the remainder of this paper, we do not address interactions among two or more parallel loops, but plan to address it in future work. Furthermore, we do not specially treat *parallel_for* loops that do not have enough parallelism to keep all GPU resources busy. In the future, we plan to execute these loops entirely on the CPU.

3. BACKGROUND

This section briefly describes our heterogeneous programming model and then discusses various existing approaches to heterogeneous execution.

3.1 C++ programming model

We use the Concord [5] C++ framework to evaluate the benefits of our different scheduling algorithms. However, our scheduling techniques should apply to other programming frameworks with similar data-parallel APIs such as OpenMP, OpenACC, and C++ AMP. Concord is a heterogeneous C++ programming framework for processors with integrated GPUs, and is designed to allow general-purpose, object-oriented, data-parallel programs to take advantage of GPU execution. It provides two parallel constructs for offloading computations to the GPU: a *parallel_for* loop and a *parallel_reduce* loop. These APIs are similar to popular data-parallel multi-core C++ programming models such as TBB [1], OpenMP, Cilk [23], and Galois [28]). Both these constructs take as arguments the number of data-parallel iterations and the closure object which encompasses the parallel code. Concord does not guarantee that different loop iterations will be executed in parallel, and does not guarantee floating point determinism in reductions. Also, programmers should make no assumption about the order in which different iterations are done. For the rest of the paper, we only focus on the *parallel_for* construct – similar techniques apply to *parallel_reduce*.

Fig. 2 shows an example Concord C++ program demonstrating the use of two *parallel_for* constructs. The first *parallel_for* increments every element of an N-element array a using the data-parallel kernel in *Foo::operator* once (single invocation kernel), while the later loop decrements the elements of a using the *Bar::operator* kernel multiple times (multiple invocation kernel). The classes *Foo* and *Bar* contain the environment (e.g., a) for the parallel code specified in the *operator* functions.

3.2 Heterogeneous execution model

Supporting data parallelism in a heterogeneous system requires two things. *First*, executable versions of a kernel must be compiled for each system device. Like systems such as *Dandelion* [31] and *Lime* [16], *Concord* [5] addresses this by translating a high-level language kernel to separate native executables for the CPU and GPU. *Second*, a *scheduling* mechanism is required to determine where each *parallel_for* iteration will be executed. There are different approaches to scheduling *parallel_for* iterations across multiple devices of a heterogeneous system that aim to minimize execution time. We summarize them below.

```
// Functor class implementing one loop body
class Foo {
  int *a;
  void operator(int i) { a[i] += f(i); }
};
// Functor class implementing another loop body
class Bar {
  int *a;
  void operator(int i) { a[i] -= g(i);}
};
...
Foo *f1 = new Foo();
// Single invocation of a data-parallel loop
parallel_for(N, *f1);
...
Bar *f2 = new Bar();
// Multiple invocations of a data-parallel loop
for (int i=M; i<P; i++)
   parallel_for(i, *f2);
```

Figure 2: Example demonstrating Concord *parallel_for*

3.2.1 Offline profiling

The application is first run using a training data set and profiled. The profiling data is used to select the scheduling policy for subsequent application runs against real data. *Qilin* [24] performs an offline analysis to measure the kernel's execution rate on each device (CPU and GPU). These rates are used to decide the distribution of work for each device. Qilin uses a linear performance model to chose the scheduling policy based on the size of the input data set. In general, there are two main drawbacks of offline-profiling. *First*, the scheduling policy chosen depends on the training data, i.e., if the training data differs significantly from the actual data used in subsequent runs, the scheduling policy is likely to be suboptimal. *Second*, new execution rates and work distributions must be determined for each new target platform.

3.2.2 Online approaches

In online approaches, the decision of where to execute *parallel_for* iterations is made at runtime. The programmer does not have to train the system on representative inputs for each target platform. We discuss some common online approaches below.

- **Work-stealing** [9] is a popular way to address load imbalance in multi-core execution. However, a GPU differs from the CPU in one important aspect when addressing load imbalance. Current GPUs cannot initiate communication with the CPU or execute atomic operations visible to the CPU which means a GPU cannot request work from non-local work pools. The GPU cannot use persistent threads to steal work from a common pool because of the relaxed CPU-GPU memory consistency, which only guarantees that GPU memory updates will be visible after the GPU code terminates. Furthermore, the GPU is statically scheduled: once items are scheduled to be processed on the GPU, the order in which they are processed is undefined and hence we can make no assumptions about

which items can be stolen from the GPU's local pool by other threads.
One simple approach to addressing load imbalance that avoids these limitations is to have a dedicated *GPU proxy thread* running on the CPU to perform work-stealing on behalf of the GPU. The GPU proxy thread can steal a number of loop iterations to execute on the GPU, then wait for the GPU to complete their execution or until all iterations have been processed by either the CPU or the GPU. The time to execute a number of loop iterations on the GPU is a key performance bottleneck. If we choose many iterations in order to reduce the overhead of launching work on the GPU, the GPU might stall loop termination while the CPU threads are out of work. On the other hand, if we choose too few iterations, this will increase the number of GPU offload requests as well as underutilize the GPU hardware, as we show later in Sec. 4.2.

- **Profiling-based** is an online profiling based approach. The application kernel is dynamically profiled and used to determine the distribution of workload between the CPU and the GPU. The key to this approach is that the profiling phase should not introduce any overhead that can not be compensated by the benefits obtained by doing it. We present two online-profiling based techniques in the next two sections and analyze their overheads.

4. NAÏVE PROFILING

In this section, we propose a heterogeneous scheduling algorithm based on a simple online profiling scheme and analyze the associated overheads.

It is well-known that regular applications with little or no load imbalance are well-suited for GPUs whereas multi-core CPUs equipped with accurate branch predictors are capable of performing well across regular or irregular, balanced or imbalanced workloads. As a result, we first determine the characteristics of a workload using an online profiling run in order to determine the rate of execution on each device. This profiling information is used to decide the percentage of remaining iterations to assign to each device.

Fig. 3 shows our naïve profiling scheduling algorithm. It executes in two phases – *profiling phase* and *execution phase*. When a kernel is executed for the first time over N items, the runtime forks a proxy thread that offloads Nf_p iterations to the GPU, where f_p is a ratio between 0 and 1[1], and measures the rate G_r at which the GPU processes the kernel. Concurrently, the runtime thread itself processes another Nf_p iterations on the multicore CPU and computes the CPU rate C_r. Both threads merge on a barrier after completing the profiling phase where they compare the two rates G_r and C_r. Based on these rates, the runtime distributes the remaining iterations to the CPU and GPU in the execution phase. New rates are computed and cached for future invocations of the same kernel. For already-seen kernels, the runtime uses the old values of C_r and G_r to distribute iterations.

The naïve profiling based scheduling algorithm may introduce overheads in both the phases: (1) in the *profiling phase*, one of the devices completes execution before the other one; (2) if the iterations used in the profiling phase are not representative of the entire iteration space (applications with imbalance and irregularity fall into this category), then the *execution phase* may introduce some more overheads. We address both these overheads with an asymmetric scheduling algorithm (Sec. 5). Below, we first provide theoretical analysis of our naïve profiling algorithm and then discuss how f_p is determined.

[1]We discuss choosing f_p in Sec. 4.2.

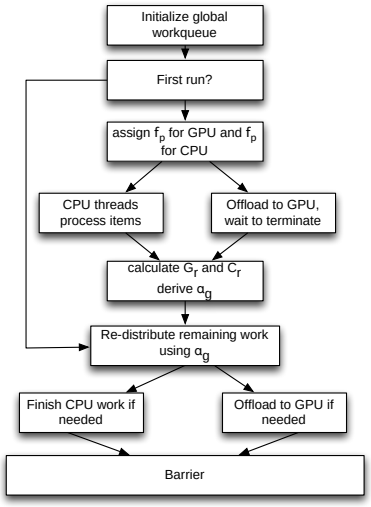

Figure 3: Naïve profiling

4.1 Analysis

Let α_g denote the ideal ratio for distribution to the GPU, and the remaining $1-\alpha_g$ to the CPU. If we denote the rate with which CPU executes work items as C_r and GPU as G_r, we would like to find the ratio α_g of work items to be offloaded to the GPU such that:

$$\frac{N\alpha_g}{G_r} = \frac{N(1-\alpha_g)}{C_r} \tag{1}$$

Where N is the total number of work items to be processed due to a call to the *parallel_for*. Using this formula, we can derive the value of α_g as:

$$\alpha_g = \frac{G_r}{G_r + C_r}$$

Intuitively, α_g specifies how fast the GPU is compared to the CPU. So if the two devices are processing items at the same rate ($C_r = G_r$), we would have $\alpha_g = 0.5$.

The ideal time to execute the N work-items can be expressed as:

$$T_{ideal} = \frac{N}{C_r + G_r} \tag{2}$$

Real execution time will consist of the T_{prof}, time to profile Nf_p work on CPU and GPU, and T_{exec}, time to execute the remaining items according to the measured distribution ratio β_g.

$$T_{real} = T_{prof} + T_{exec} \tag{3}$$

The distribution ratio β_g computed by the profiling may differ from the ideal distribution ratio α_g that allows both devices to terminate at the same time. This could happen, for example, if the workload is highly irregular or if the profiling stage is too short. How much the execution time T_{exec}

differs from the ideal time to execute $(1-2f_p)N$ remaining items depends on the difference between α_g and β_g, as stated in the theorem below. Ideally, we would like profiling to derive β_g as close to α_g as possible.

THEOREM 1. *The overhead of executing P iterations using distribution ratio β_g instead of the ideal ratio α_g is proportional to the difference between β_g and α_g.*

PROOF. *The overhead is the difference of execution times with β_g and α_g.*

$$T_{overhead} = max[\frac{N\beta_g}{G_r} - \frac{N\alpha_g}{G_r}, \frac{N(1-\beta_g)}{C_r} - \frac{N(1-\alpha_g)}{C_r}]$$

$$= max[N(\frac{\beta_g - \alpha_g}{G_r}), N(\frac{\alpha_g - \beta_g}{C_r})] \quad (4)$$

The above difference is clearly proportional to $\beta_g - \alpha_g$. □

Another source of overhead with the naïve profiling algorithm is a suboptimal distribution of work during the profiling phase. The profiling time is the maximum of time to execute Nf_p items on each of CPU and GPU.

$$T_{prof} = max[\frac{Nf_p}{G_r}, \frac{Nf_p}{C_r}] \quad (5)$$

The profiling phase can impose significant overhead if the difference between G_r and C_r is very large. For example, assume that the GPU is 10x faster than the CPU ($G_r = 10C_r$). In this case,

$$T_{prof} = max[\frac{Nf_p}{10C_r}, \frac{Nf_p}{C_r}] = \frac{Nf_p}{C_r}$$

If we assume $f_p = 5\%$, the time taken by the profiling phase becomes $\frac{0.05N}{C_r}$. As the GPU is 10x times faster than the CPU, it completes its profiling run in one tenth of the time. If the GPU would continue working during the rest of the profiling run it could complete nine times as much work, that is, another 45% of the total work. Even if the profiling is accurate, and the remaining work is divided between the CPU and GPU using the ideal distribution ratio $\alpha_g = 0.9$, the time the GPU has spent idle during the profiling stage will have significant negative effect on total execution time.

4.2 Determining profiling size

Determining the right number of iterations or work-items to use for profiling is important. Picking *too many* items for the profiling phase can increase the profiling overhead since one device may have to wait for the other device to complete execution. On the other hand, picking *too few* work items for the GPU can underestimate GPU performance since it is underutilized. Ideally, the profiling size should be large enough to utilize all the available GPU parallelism. To illustrate this point, we demonstrate the impact of different profiling sizes on two kernels: one regular kernel and the other irregular.

Regular kernel: Fig. 4 shows the impact of choosing different profiling sizes on the performance of a simple synthetic kernel running on the integrated GPU (details of our experimental platform are provided in Sec. 7). This kernel computes the sum of integers from 1 to 2048 per work-item. There are no memory accesses and no thread divergences in the kernel. We report the GPU execution rate, G_r, as we

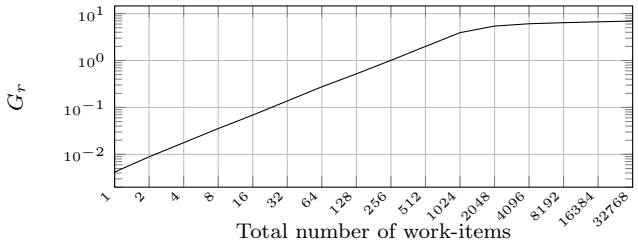

Figure 4: Plot showing the change in G_r with increasing number of work-items for a regular kernel. (Note the logarithmic scales.)

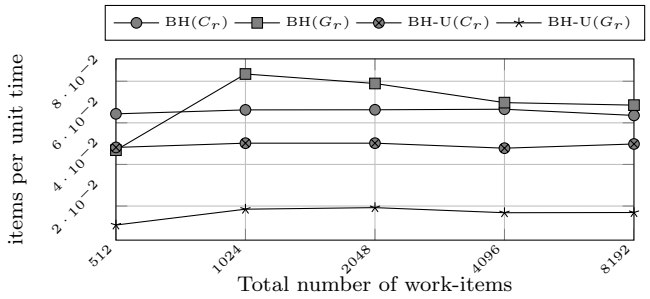

Figure 5: Plot showing average C_r and G_r for different values of Nf_p for BarnesHut (**BH**) and BarnesHut-unoptimized (**BH-U**).

increase the number of work-items offloaded to the GPU. As we increase the number of items, we see an increase in G_r as the GPU takes the same amount of time to process more items. The rate stabilizes at 2048 items, which implies that the GPU hardware was underutilized when processing less than 2048 items. Ideally, we would like to have at least 2048 items to profile accurately and not underutilize this specific GPU.

Irregular kernel: Irregular kernel code may complicate the choice of a good profiling chunk size. Fig. 5 shows the average execution rates for both **BH-U** and **BH**[2] when we process items in chunk sizes ranging from 512 to 8192. We show the execution rates separately for the CPU (C_r) and GPU(G_r). The CPU performance is relatively stable for both **BH** and **BH-U** (the difference in time per item for the two versions is due to the improved locality with **BH**), and not affected by increasing chunk size. However, for **BH-U** on the GPU, G_r increases sharply from 512 to 1024 and is relatively stable for larger values. A similar but less subtle trend is visible for **BH** on the GPU as well.

The impact of picking a small chunk size for GPU profiling is visible in Fig. 5. If we pick a chunk size of 512 for **BH-U**, we get the GPU rate $\frac{1}{92}$, and the CPU rate $\frac{1}{21}$, which leads to an offload ratio α_g of 0.17. However, if we pick 1024 as the chunk size, we get a different rate for the GPU $\frac{1}{54}$ and for the CPU $\frac{1}{19}$ leading to an GPU offload ratio of 0.27, which is close to the static optimal 0.30.

To summarize, we want to use the smallest number of items that fully utilize the GPU. The integrated GPU we use in our experimental evaluation has 20 execution units (EU), 7 threads per EU, each thread being 16-way SIMD

[2]**BH** is an optimized version of Barnes-Hut algorithm (**BH-U**), where the bodies are sorted in a breadth-first-search order from the root.

for a total of 2240 work-items that can execute at one time. This information can be obtained automatically by querying the GPU device using the OpenCL API[3]. So, we use 2048 as the profiling size for our evaluation which agrees with our empirical observation above for both regular and irregular kernels.

5. ASYMMETRIC PROFILING

In this section, we describe our asymmetric profiling algorithm that addresses the overheads of the naïve profiling algorithm and additionally, provides adaptive strategies that handle load imbalance due to irregularity and multiple invocations per kernel.

The first overhead in naïve profiling was caused by waiting on a barrier when one of the devices finishes execution before the other. We address this overhead in asymmetric profiling as follows: we initially have a shared pool of work consisting of the entire parallel iteration space, and we pick a portion f_p of items to be offloaded to the GPU using the proxy GPU thread. The CPU workers continue to pick items from the shared global pool and locally collect profiling information. When the GPU proxy thread finishes profiling, it performs the following steps in order; it (1) computes the distribution ratio by reading each worker's local profiling statistics (during this time, the CPU workers continue to work on the shared pool), (2) empties the shared pool, (3) adds the CPU portion of the work to one of the CPU worker's work-stealing queue, and (4) finally offloads GPU portion of the work to GPU. Fig. 6 illustrates the algorithm. It is important to note that while the GPU is executing, CPU workers continue to work from the shared pool, thereby eliminating the overhead seen in naïve profiling.

One of the design choices for asymmetric scheduling is to start with a shared pool in the profiling phase instead of work-stealing. This is justified since at the beginning we have no knowledge of how to partition work among CPU workers and the proxy GPU worker. For example, for applications with irregularities, it may be costly to partition the work up front into the work-stealing queues.

If both the devices are kept busy in the profiling phase by having sufficient number of parallel iterations, it can be seen that the profiling phase will introduce zero overhead in the asymmetric profiling algorithm (theoretical analysis is provided below in Sec. 5.1). Similar to the naïve profiling, the vanilla version of asymmetric profiling may still introduce overhead in the execution phase if the iterations used in profiling phase are not representative of the entire iteration space. We address this in Sec. 5.2.

5.1 Analysis

We now analyze the overhead of asymmetric profiling compared to the ideal execution scenario. We concentrate on the analysis of the profiling phase, as the behavior of the execution phase in the base algorithm is identical to that of naïve profiling. Assuming the runtime offloads f_p fraction of the original items to the GPU and allows CPU threads to execute as long as the GPU is busy, the time taken by the GPU to complete Nf_p items is $t_p = \frac{Nf_p}{G_r}$. At the same time, CPU completes an additional t_pC_r items assuming there is enough work to keep it busy. In this case, the overhead of the profiling is zero, as both GPU and CPU work all the time. On the other hand, if the amount of work offloaded to GPU is so large that there is not enough work items for CPU, there is a non-zero overhead. This observation is formally stated by the following theorem.

[3]CL_DEVICE_MAX_COMPUTE_UNITS, CL_DEVICE_MAX_WORK_GROUP_SIZE, and CL_DEVICE_NATIVE_VECTOR_WIDTH_INT.

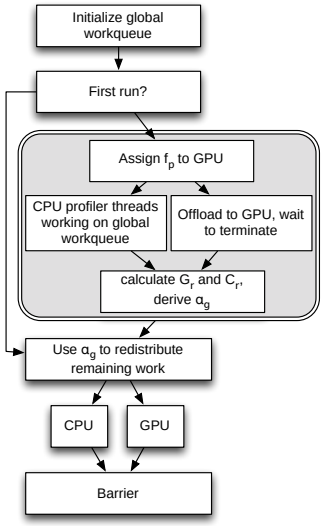

Figure 6: Asymmetric profiling

THEOREM 2. *Overhead of the profiling phase in the asymmetric profiling algorithm is zero if $f_p < \alpha_g$.*

PROOF. *The number of items executed by CPU is maximum of the amount of work it can perform in t_p time and the remaining work:*

$$N_c = max(t_pC_r, (1 - f_p)N)$$

The first argument to max is greater than the second, when $t_pC_r > (1 - f_p)N$, which simplifies to $f_p > \alpha_g$. In this case, the total amount of work executed by CPU and GPU is $N_p = Nf_p + \frac{Nf_pC_r}{G_r}$. In the ideal case, this work can be executed by CPU and GPU working together in $\frac{N_p}{C_r+G_r}$ time. This time is equal to the profiling phase time $\frac{Nf_p}{G_r}$ as can be shown by simple algebraic transformations. The overhead of the profiling phase is, thus, zero. □

In practice, f_p is likely to be greater than α_g only when CPU is significantly faster than GPU. For example, if we offload 5% of work to GPU for profiling, CPU should be at least 19x faster than GPU to trigger non-zero overhead. To handle such cases our work can be combined with static profiling techniques such as [18] to determine if the workload has a strong CPU bias and use CPU-biased asymmetric profiling instead of the GPU-biased one.

5.2 Addressing load imbalance

In order to address bias due to the initial profiling, the profiling can be repeated until a certain termination condition is reached, after which the benefits of re-profiling diminish. This ensures that we profile more than once to understand the application-level imbalance better. We consider two termination conditions: *convergence* and *size*.

Convergence-based strategy repeats profiling until β_g converges. For example, we could re-profile until computed ratios differ by no more than 0.05. The assumption with convergence strategy is that once the distribution ratio stops changing, it is close to the ideal ratio α_g. Convergence strategy works well when distribution ratio indeed stabilizes after temporary converging. On the otherhand, *size*-based strategy repeats the profiling until a certain portion of the work items has been completed. For example, we could re-profile till the number of remaining items is more than 50%. Size-based strategy works well when *i)* running CPU threads in profiling mode does not impose a lot of overhead, and *ii)* the irregularity of the workload gets amortized over fixed portion of the items.

When re-profiling, it is important to keep both CPU and GPU busy. As has been shown by Theorem 2, asymmetric profiling incurs no overhead only when there is sufficient amount of work to keep CPU busy while GPU performs its fixed portion of work. The profiling strategy, thus, should keep the total fraction of items offloaded to GPU below the ideal distribution ratio α_g, or, more practically, below its approximation, β_g, from a prior profiling run.

It is important to note that there can be scenarios where neither size-based or convergence-based strategies can determine the optimal partitioning – for example, triangular loop inside a *parallel_for* loop. We believe such scenarios are rare in practice – none of the sixteen benchmarks we studied exhibit this kind of behavior.

5.3 Multiple invocations per kernel

In programs where the kernel is executed multiple times, the programmer can assume the first invocation as a profile run to obtain the rate of execution on both devices. After the first run, whenever we execute some items on one or both devices, we observe the execution rates and update our offload ratio β_g according to the selected update strategies described below.

5.3.1 Update functions:

Given an execution of some items on the devices, we use (C_r, G_r, C_n, G_n), where C_r and G_r are rates and C_n, G_n are number of items processed by the CPU and GPU respectively. Two variants we propose are :

1. **Greedy**: Simply use the G_r, C_r to compute β_g. This is always used the first time a rate is computed since we assume no initial distribution of work-items.
2. **Sample weighted**: Compute $w = \frac{C_n+G_n}{C_n+G_n+T_n}$, where T_n is the number of items used to compute the rate so far, and $\beta_g^{new} = \frac{G_r}{G_r+C_r}$. And set the new ratio to be $\beta_g = w\beta_g^{new} + (1-w)\beta_g$. Update $T_n += C_n + G_n$.

We also evaluate a device-weighted variant, which weights the rate computed by each device by the number of items processed by the device for that profiling run. We do not observe a significant improvement in the performance of device-weighted updates. We use the *sample-weighted* updates for our experimental results.

6. IMPLEMENTATION

In this section, we briefly discuss the implementation details that is used in the evaluation in Sec. 7. We start with details on the compiler infrastructure, followed by a summary of the heterogeneous runtime describing when and where the scheduling decision takes place.

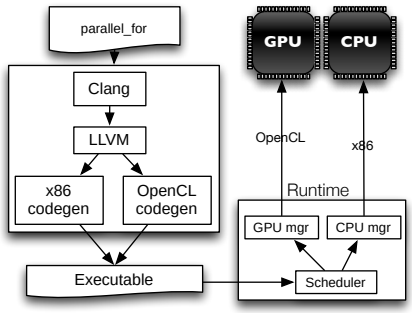

Figure 7: Compiler and runtime

Compiler: Our evaluation uses data-parallel C++ applications where the parallel regions are expressed using Concord *parallel_for* constructs as described in Sec. 3.

Fig. 7 depicts our overall compilation and runtime framework. We use the CLANG and LLVM infrastructure to compile our C++ programs. The static compiler identifies kernels and generates both native CPU and GPU OpenCL code for them. It then emits a host-side executable containing all code including the OpenCL code.

Runtime: During program execution, the runtime loads the embedded OpenCL code and just-in-time compiles it to GPU *ISA* using the vendor-specific OpenCL compiler. It also decides how to distribute the work between the CPU and GPU. It implements work-stealing on the CPU, with one of the CPU worker threads (the GPU proxy thread) offloading to the GPU as guided by online profiling. The observed distribution ratio, β_g, for our scheduling algorithms (described in Sec. 4 and Sec. 5) is computed in the GPU proxy thread which then redistributes the parallel iterations among the CPU and GPU cores.

7. EVALUATION

We now present an evaluation of our techniques. We begin with an overview of the hardware and software environment. We next describe the benchmarks used in the evaluation and their static and runtime characteristics. Finally, we present performance results and follow it up with a summary.

7.1 Environment

We evaluated our scheduling techniques on a desktop computer with a 3.4GHz Intel 4^{th} Generation Core i7-4770 Processor with four CPU cores and with hyper-threading enabled. The integrated GPU is an Intel HD Graphics 4600 with 20 execution units (EUs), each with 7 hardware threads where each thread is 16-wide SIMD, and running at a turbo-mode clock speed from $350MHz$ to $1.2GHz$. The system has 8GB system memory and is running 64-bit Windows 7.

7.2 Benchmarks

We consider a diverse set of applications (sixteen in total) to evaluate our proposed runtime system. These applications span a spectrum of application domains and exhibit different behaviors: single kernel vs. multiple kernels, single invocation vs. multiple invocations, and regular vs. irregular. Most of these were ported from existing sources: TBB [1], Rodinia [13], and Parsec [8]. We also developed some applications from scratch. The details of our benchmarks are shown in Table 1.

Name	Abbrv.	Input	# kernels	kernel-name	invocations	Oracle-time(s)	num iter per kernel	Dynamic. Instr.
BarnesHut	BH [6]	1M bodies, 1 step	1	Barneshut	1	6.852	1000000	1.96×10^{11}
BarnesHut(unoptimized)	BH-U [6]	1M bodies, 1 step	1	Barneshut	1	9.194	1000000	1.98×10^{11}
LavaMD	LMD [13]	-boxed1d 10	1	kernel	1	0.582	1000	2.02×10^{10}
Matrix Multiply	MM	2048 by 2048 float matrix	1	testc	1	1.506	4194304	7.08×10^{10}
Ray Tracer	RT	sphere=256,material=3,light=5	1	kernel	1	2.426	10240000	3.25×10^{11}
Breadth first search	BFS	W-USA ($\|V\|$=6.2M, $\|E\|$=1.5M)	1	relax	1748	14.05	6262104	5.68×10^{11}
Black Scholes	BS [8]	64K	1	mainwork	2000	0.32	65536	6.17×10^{10}
Connected Component	CC	W-USA ($\|V\|$=6.2M, $\|E\|$=1.5M)	1	merge	2147	22.37	6262104	1.54×10^{12}
Face Detect	FD [2]	3000 by2171 Solvay-1927	1	HaarDetect	132	1.884	22-1370340	6.09×10^{10}
Heartwall	HW [13]	test.avi	1	kernel	5	0.924	51	4.62×10^{10}
N-Body	NB	4096 bodies	1	Slowpar	101	4.506	20475	3.91×10^{11}
Seismic	SM [1]	1950 by 1326, 100 frames	1	UpdateStress	100	2.048	25791520	1.43×10^{11}
Shortest Path	SP	W-USA ($\|V\|$=6.2M, $\|E\|$=1.5M)	1	relax	2577	30.66	6262104	9.17×10^{11}
BTree	BT [13]	big-command	2	kernel_cpu	1	0.586	1000000	1.66×10^{11}
				kernel_cpu_2	1		1000000	
Computational fluid dynamics	CFD [13]	missile - 0.3M	4	compute-time	6000	24.072	232704	1.49×10^{12}
				compute-flux	6000		232704	
				compute-step	2000		232704	
				init	1		232704	
Particle Filter	PF [13]	(128, 128, 10), 80K particles	7	find-index	9	4.646	8000	3.50×10^{11}
				calculate-u	9		8000	
				divide-weights	9		8000	
				update-weights	9		8000	
				particle-filter-like	9		8000	
				initial-arrays	1		8000	
				initial-weights	1		8000	

Table 1: Key statistics for the kernels of benchmarks used in the evaluation. *Oracle-time* is the time taken by the best offline distribution, and is also used as the normalizing factor in Fig. 9 and Fig. 10.

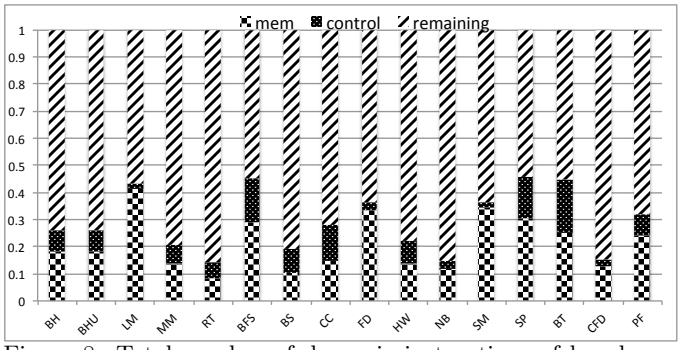

Figure 8: Total number of dynamic instructions of benchmarks divided into three categories: *memory*, *control*, and *remaining*, obtained via a serial CPU execution. The number of dynamic instructions is given in Table 1(column 9).

The compile-time and run-time characteristics of our benchmarks are tabulated in Table 1. The order of the benchmarks we use are as follows: (1) single kernel and single invocation (**BH**, **BH-U**, **LMD**, **MM**, **RT**); (2) single kernel and multiple invocations (**BFS**, **BS**, **CC**, **FD**, **HW**, **NB**, **SM**, **SP**); (3) multiple kernels and multiple invocations (**BT**, **CFD**, **PF**). The table shows the number of times a kernel is invoked in column 6. The *Oracle-time* in column 7 reports the absolute execution time for an *Oracle* approach described in Sec. 7.3 – this serves as the baseline for the runtime evaluation in Fig. 9 and Fig. 10, which are normalized to this time. Column 8 reports the number of parallel iterations per kernel invocation. Column 9 reports the number of instructions executed at runtime per benchmark.

To better understand the irregularities in our benchmarks, we also analyze the dynamic instruction traces for a single-threaded CPU execution via *Intel VTune Amplifier* to determine the run-time behavior of our applications[4]. Fig. 8 shows the division of the instructions into three categories: *memory* operations (load and store instructions), *control* instructions, and *remaining*, which can be a crude estimate for the *compute* instructions. The absolute number of dynamic instructions is given in column 9 of Table 1. Applications such as **BH**, **BH-U**, **BFS**, **CC**, **SP**, **BT**, and **PF** show considerable amount of control flow irregularities where as **FD**, and **SM** show significant amount of memory related operations. Since these applications are not hand-tuned for GPUs, the memory related operations may not necessarily be coalesced and give an indication of non-coalesced memory accesses on the GPU. Also note that benchmarks with large amount of control flow irregularities may not perform well on the GPU.

7.3 Comparison schemes

We compare the following scheduling strategies to distribute the parallel iterations between the CPU and GPU:

1. **CPU**: Multi-core CPU execution based on TBB. The *parallel_for* invocation is simply forwarded to the TBB runtime to complete the execution.
2. **GPU**: GPU-only execution, where all items are offloaded to the GPU via OpenCL.
3. **Oracle**: The best performance obtained by exhaustive search of different amount of parallel_for iterations assigned to the CPU and GPU. The GPU works on some percentage of the parallel iterations while the CPU works on the remaining ones. The percentage is varied from 0% to 100% on the GPU in increments of 10%

[4]Note that this measurement provides an upper-bound of the irregularities present in a benchmark.

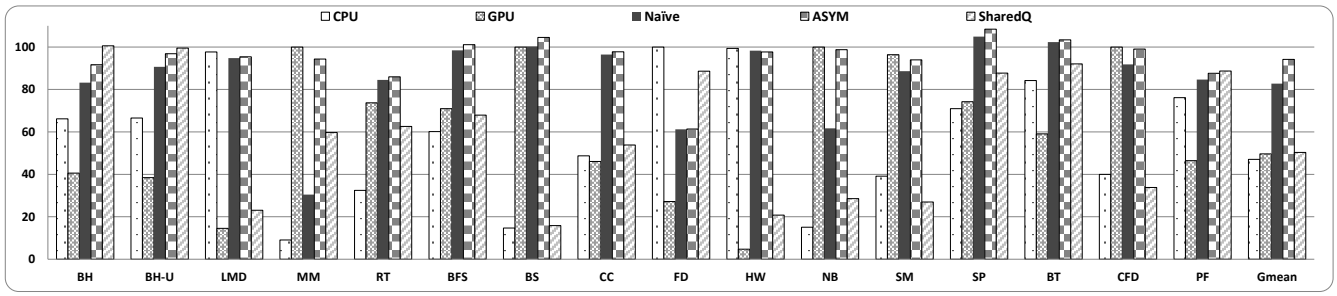

Figure 9: Relative speedup for all benchmarks compared to Oracle. Oracle is at 100% (higher is better).

and the best performance obtained is selected. The best percentage for single kernel benchmarks is given in Table 2(row 1) and the absolute runtime for those percentages is given in Table 1(column 7). We select the runtime obtained by such a technique as the baseline for all comparisons.

4. **Naïve**: The naïve profiling scheme described in Sec. 4 where we use the same profiling size for both the CPU and the GPU.
5. **ASYM**: The asymmetric profiling scheme described in Sec. 5. We use the *sample weighted* variant for multiple invocation kernels.
6. **SharedQ**: We also compare against a shared queue implementation where a GPU proxy-thread atomically grabs a fixed chunk of work (profile size) and offloads it to the GPU while CPU threads also atomically obtain work-items from the same global queue. Since there will be contention on the shared queue, this approach may suffer as the number of workers increase[5].

We also evaluate two variants of our asymmetric profiling algorithm to address load imbalance, as described in Sec. 5.2:

- **ASYM+CONV**: A convergence-based update strategy, where half of remaining items are used for reprofiling until convergence is achieved. Convergence is reached when the β_g computed on two successive profiling steps do not differ by more than 0.05.
- **ASYM+SIZE**: A size-based update strategy, where we repeatedly profile with f_p items on the GPU until at least half of the total items are left.

7.4 Results

Heterogeneous execution: First we highlight the significance of heterogeneous execution compared to single device execution. We compare the single-device execution (either on the CPU or the GPU) against the offline optimal model (Oracle). Fig. 9 shows that there is an obvious advantage to heterogeneous execution: (1) using only the multicore CPU, we can achieve 47.07% of Oracle on average; (2) using only the GPU, we can achieve 49.67% of Oracle on average. Clearly, heterogeneous execution results in 2× improvement in runtime compared to the best of executing on the CPU-alone or the GPU-alone. These results emphasize the improvement of heterogeneous execution over single device execution.

Naïve profiling: Naïve profiling as described in Sec. 4 performs online profiling on a fraction of the parallel iterations to determine the offload ratio, β_g. Although the profiling information obtained is able to outperform single-device execution as illustrated in Fig. 9, it only performs at 82.7% of the Oracle on average. There are two sources of inefficiency in naïve profiling: (1) The *overhead of profiling* can dominate the execution time for workloads that are highly biased towards a particular device. For instance **NB** and **MM** are both highly biased towards the GPU. With a fixed profiling size on both devices, the scheduler has to wait at the barrier for both devices to report the rates to determine the distribution. If the GPU profiling finishes early, the scheduler waits for the CPU to complete the profiling step resulting in a large overhead. (2) The *inaccuracy of profiling information*, although not evident, is another source of inefficiency. For instance, **FD** only obtains 61.2% of the Oracle, because the profiling information obtained is not representative of the entire iteration space. The inaccuracy can also be observed in optimal distribution percentages (as shown in Table 2) for benchmarks with single *parallel_for* kernels.

	BH	BH-U	LMD	MM	RT	BFS	BS	CC	FD	HW	NB	SM	SP
Oracle	40	40	0	100	70	70	100	70	0	0	100	70	50
Naïve	52.4	44.3	0	89.6	84.4	72.8	94.9	73.9	31.1	0	77.3	77	71
ASYM	47.9	41.7	0	92.5	84.3	72.8	96.2	73.8	32.9	0	89.6	92.1	70.6
ASYM+SIZE	39.6	40.2	0	92.2	69.7	61.1	96.0	68.8	18.4	0	88.3	92.2	59.2
ASYM+CONV	41.2	40.6	0	92.4	70.6	73.1	96	73.8	33.2	0	89	92.7	70.7

Table 2: GPU work percentages computed by different schemes for single-kernel applications.

Asymmetric profiling: As described in Sec. 5, asymmetric profiling does not suffer from the overhead of waiting on a barrier to compare the rates for both devices. Most applications benefit from using asymmetric profiling compared to naïve profiling. That is, it performs on average at 94.2% efficiency compared to the Oracle. The biggest improvement is observed in applications that show a high GPU bias such as **NB** and **MM**. This is because the distribution ratio β_g can be decided quickly, and hence the overhead of profiling is reduced significantly (close to zero[6]). We note, however, that **BH** and **FD** do not benefit from the vanilla version of asymmetric profiling. This is, as explained before, due to the inaccuracy of the profiling information as evident from the distribution computed by asymmetric profiling differing from the optimal distribution obtained by the Oracle (as shown in Table 2). Note that benchmarks such as **BS**, **BT**, **BFS**, and **SP** improve performance better than Oracle using asymmetric profiling – this is because the Oracle is obtained in increments of 10%.

ASYM+CONV: In order to address the load imbalance and the inaccuracy of profiling information, we rely on re-

[5]Note that our SharedQ implementation does not use the optimization strategies described in Sec. 5.3.

[6]This overhead in our implementation consists of : for N workers, we read a local counter from each worker and perform N additions and 3 divisions, which is negligible compared to the total execution time of an application.

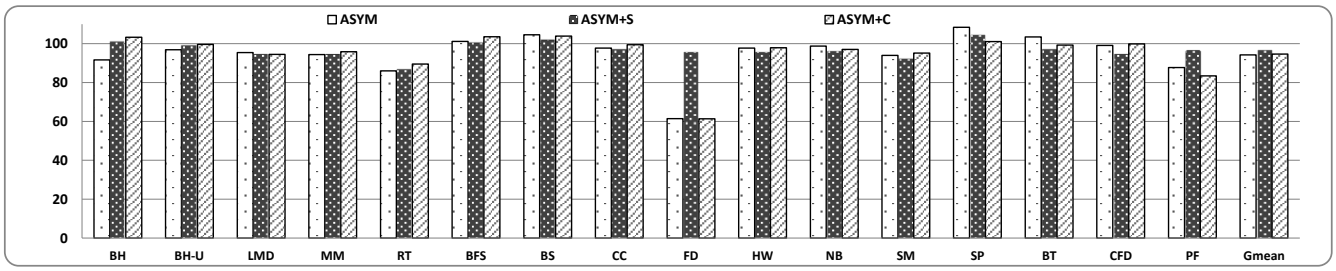

Figure 10: Comparison of different adaptive schemes. Vertical axis shows relative speedup vs. Oracle (higher is better).

peated profiling to achieve more accurate information. The convergence based approach repeatedly profiles until the offload ratio, β_g, determined by consecutive profiling phases differs by less than 0.05. This does introduce an overhead of repeated profiling, but the benefit of more accurate profiling information should offset this overhead. **ASYM+CONV** can achieve on average 94.6% of Oracle as shown in Fig. 10. The biggest improvement is observed in the applications that have some irregularity, for instance **BH**. The convergence based approach determines a more accurate distribution ratio, as evident from the percentages reported in Table 2 (47% for **ASYM** versus 41% for **ASYM+CONV**) is more closer to the optimal value 40%. Similarly, for **RT**, the ratio reported becomes 69% versus the optimal 70%.

However, we observe that this approach still does not improve **FD**, and furthermore the efficiency of **PF** goes down. Both of them are false positives as they converge fairly quickly in **ASYM+CONV** approach. The source of inefficiency for **FD** stems from the algorithm: the cascade of classifiers is grouped to discard failing face images early. The profiling phase obtained from the initial rounds converges in two iterations on these cascade of classifiers, which is not representative of later stages that search further for faces in parts of the image not discarded by earlier stages. For **PF**, an inaccurate rate due to convergence makes the performance drop from 87% for **ASYM** to 83% for **ASYM+CONV**. Clearly, in order to improve the efficiency of **FD** and **PF**, the profiling information has to be resilient to local optima. Also note that, **FD** shows a large amount of memory related irregularities and **PF** shows a large amount of control flow and memory irregularities (as shown in Fig. 8).

ASYM+SIZE: Instead of relying on convergence, we use half of the work-items to determine the profiling ratio and use this ratio to distribute the remaining items. This approach proves the best performing overall since it does not rely on an initial portion of the work-items to determine the rate and does not converge on a local maxima. There is an overhead to repeated profiling which reduces the efficiency for some applications, but not significantly. However, the biggest winner is **PF**, which performs at 96% of Oracle, followed by **FD**, which performs at 95% of Oracle. Overall, the size-based approach performs at 96.8% (geo-mean) efficiency of the Oracle, thereby clearly demonstrating that it is the best strategy across all benchmarks.

7.5 Summary

In conclusion, we show that:

- *Heterogeneous* execution improves the execution time of our applications by 2× compared to a single device execution (either the multicore CPU or the GPU).
- A simple profiling technique, such as Naïve profiling, can provide reasonably good performance, on average 82% of the Oracle.
- *Asymmetric* profiling is necessary to reduce the overhead of online profiling and improves the efficiency of heterogeneous execution to 94.2% compared to the Oracle.
- Some applications such as **FD**, **RT** and **BH** require repeated profiling to obtain optimal distribution of work. Our adaptive scheme **ASYM+SIZE** obtains the best distribution with minimal profiling overhead and achieves 96.8% efficiency compared to the Oracle.

8. RELATED WORK

Making GPUs more accessible to programmers has been an increasingly active research area [5, 14, 16, 31]. Below we discuss prior work that particularly focus on scheduling and load balancing in heterogeneous architectures.

8.1 Heterogeneous execution

There have been increasing efforts [4, 24, 27, 35] to make heterogeneous execution of applications more efficient. A load balancing scheme for discrete devices, *HDSS*, is presented in [7], and is similar to our convergence based approach. [25] describes a model-driven 2D-FFT scheduling for matrix computations across two devices. [29] describes a dynamic scheduling strategy for heterogeneous execution for generalized reductions and structured grids based on a cost model.

Qilin [24] is an API and runtime to support execution of applications across the CPU and the GPU. Programmers use the API to manage data movement and provide implementation of kernels for both devices. The system performs offline profiling to determine parameters for a linear performance model which is used to determine distributions for real runs. The *StarPU* [4] system is a data management and runtime framework similar to *Qilin* [24]. It supports user defined scheduling policies along with a set of pre-defined policies for dynamic workload scheduling. *SKMD* [22] supports execution of single kernel on multiple devices. It relies on static analysis of kernels to determine workload distribution. More recently, [35] presents a heterogeneous library for executing dense linear algebra. [27] presents a compiler and runtime to address portable performance across heterogeneous systems. They utilize evolutionary algorithms to search optimal algorithms from *PetaBricks* [3] specifications. Pandit et al. [26] balance CPU and GPU workload by restricting

CPU to executing work in coarse-grain chunks with all CPU threads synchronizing at the end of each chunk. While this approach works well for their regular PolyBench workloads, our work targets both regular and irregular applications.

Ravi et al. [30] use work-sharing to distribute work between the CPU and a discrete GPU. Their non-uniform chunk size-based work distribution resembles our baseline SharedQ approach that is 53.5% slower than our **ASYM** approach. Grewe et al. [17] use machine learning to divide work between the CPU and GPU when there is contention from other programs. In contrast, we use online profiling and do not require offline training or other prior knowledge of the applications and platform. Scogland et al. [34] present several scheduling techniques for systems with discrete devices. Their most sophisticated approach resembles our naïve profiling that we found yields only 82% of Oracle performance compared to 96.8% for **ASYM+CONV**.

Our work is different from the above work in several aspects. Data transfer is not a factor in our scenario – thus making the offloading cost to minimal. Besides the wide-variety of regular workloads studied in previous works, we handle irregular workloads as well. It is hard to get optimal partitioning for them through offline and machine model based approaches. We do not require knowledge of the underlying architectures, no offline processing, and we do not require separate program execution needed by [29] to decide the parameters in their cost model for each application.

8.2 Load balancing among multiple GPU cores

Cederman et al. [11] and Chatterjee et al. [12] address load balancing of workloads across different execution units on a GPU. In particular, they use work-stealing between tasks running on the different streaming multiprocessors (SM) of a discrete GPU. The host CPU populates the initial work-stealing queues. Each SM maintains its own work-stealing queue and steals [9] work from other SMs. This is possible due to the availability of an atomic *CAS* operation on the GPU between its SMs. However, since no current hardware supports those operations between the CPU and GPU, this approach does not extend to the general case, in particular to integrated GPUs like ours. [15] describe a fine-grained load balancing scheme by running persistent kernels which communicate with the host via task-queues. Abstracting multiple GPUs as a single virtual device [21] has also shown to be a useful way to support multiple GPUs. A performance prediction based on modeling of multiple GPUs is presented in [33].

9. CONCLUSIONS AND FUTURE WORK

Heterogeneous execution on integrated GPUs and SoCs will be a key feature of future architectures [20]. Dynamic load balancing will be essential to realize the full potential of these architectures. We presented a set of scheduling algorithms that use load balancing to minimize application execution time on integrated GPUs. To improve accuracy, these algorithms make use of low-cost online profiling. After presenting a naïve profiling algorithm, we describe asymmetric profiling scheme that mitigates some overheads in the naïve scheme. We also explored two extensions to asymmetric profiling that address load imbalance: size-based and convergence-based. We observed that these two techniques help to address irregularities in application behavior and can achieve within 3.2% of the maximum throughput obtained by an exhaustive search algorithm using a diverse set of sixteen benchmarks running on Intel's 4^{th} Generation Core Processor. In the future, we want to extend our work to consider computer systems with multiple CPUs, discrete GPUs, and discrete accelerators [10] such as the Intel Xeon Phi coprocessor. While discrete GPUs and accelerators have greater communication costs and longer latencies, they have higher power budgets and higher computational potential. A key issue to consider will be the data-transfer overhead between different devices. Finally, we would like to extend our scheduling algorithms to handle unpredictable system loads.

10. REFERENCES

[1] Intel thread building blocks.

[2] Opensource computer vision library.

[3] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. Petabricks: a language and compiler for algorithmic choice. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 38–49, NY, USA, 2009. ACM.

[4] 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.

[5] 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.

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

[7] M. E. Belviranli, L. N. Bhuyan, and R. Gupta. A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures. *ACM Trans. Archit. Code Optim.*, 9(4):57:1–57:20, Jan. 2013.

[8] 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 '08, pages 72–81, NY, USA, 2008. ACM.

[9] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, Sept. 1999.

[10] C. Cascaval, S. Chatterjee, H. Franke, K. Gildea, and P. Pattnaik. A taxonomy of accelerator architectures and their programming models. *IBM Journal of Research and Development*, 54(5):5:1–5:10, 2010.

[11] D. Cederman and P. Tsigas. On dynamic load balancing on graphics processors. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '08, pages 57–64, Aire-la-Ville, Switzerland, 2008.

[12] S. Chatterjee, M. Grossman, A. Sbirlea, and V. Sarkar. Dynamic task parallelism with a GPU work-stealing runtime system. In *Languages and Compilers for Parallel Computing*, volume 7146 of

Lecture Notes in Computer Science, pages 203–217. Springer Berlin Heidelberg, 2011.

[13] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 44–54, Washington, DC, USA, 2009. IEEE Computer Society.

[14] S. Che, J. W. Sheaffer, and K. Skadron. Dymaxion: optimizing memory access patterns for heterogeneous systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 13:1–13:11, NY, USA, 2011. ACM.

[15] L. Chen, O. Villa, S. Krishnamoorthy, and G. Gao. Dynamic load balancing on single- and multi-GPU systems. In *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–12, 2010.

[16] C. Dubach, P. Cheng, R. Rabbah, D. F. Bacon, and S. J. Fink. Compiling a high-level language for GPUs: (via language support for architectures and compilers). In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 1–12, NY, USA, 2012. ACM.

[17] D. Grewe, Z. Wang, and M. O'Boyle. OpenCL task partitioning in the presence of GPU Contention. In S. Rajopadhye and M. Mills Strout, editors, *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013.

[18] D. Grewe, Z. Wang, and M. O'Boyle. Portable mapping of data parallel programs to OpenCL for heterogeneous systems. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–10, 2013.

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

[20] S. Keckler, W. Dally, B. Khailany, M. Garland, and D. Glasco. GPUs and the future of parallel computing. *Micro, IEEE*, 31(5):7–17, 2011.

[21] J. Kim, H. Kim, J. H. Lee, and J. Lee. Achieving a single compute device image in OpenCL for multiple GPUs. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 277–288, NY, USA, 2011. ACM.

[22] 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.

[23] C. E. Leiserson. The cilk++ concurrency platform. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 522–527, NY, USA, 2009. ACM.

[24] 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 42, pages 45–55, NY, USA, 2009. ACM.

[25] Y. Ogata, T. Endo, N. Maruyama, and S. Matsuoka. An efficient, model-based CPU-GPU heterogeneous FFT library. In *IEEE International Symposium on Parallel and Distributed Processing. IPDPS.*, pages 1–10, 2008.

[26] P. Pandit and R. Govindarajan. Fluidic kernels: Cooperative execution of opencl programs on multiple heterogeneous devices. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 273:273–273:283, NY, USA, 2014. ACM.

[27] 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 '13, pages 431–444, NY, USA, 2013. ACM.

[28] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The tao of parallelism in algorithms. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 12–25, NY, USA, 2011. ACM.

[29] V. Ravi and G. Agrawal. A dynamic scheduling framework for emerging heterogeneous systems. In *High Performance Computing (HiPC), 2011 18th International Conference on*, pages 1–10, Dec 2011.

[30] V. T. Ravi, W. Ma, D. Chiu, and G. Agrawal. Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 137–146, NY, USA, 2010. ACM.

[31] 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 '13, pages 49–68, NY, USA, 2013. ACM.

[32] A. Sbîrlea, Y. Zou, Z. Budimlíc, J. Cong, and V. Sarkar. Mapping a data-flow programming model onto heterogeneous platforms. In *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, LCTES '12, pages 61–70, NY, USA, 2012. ACM.

[33] D. Schaa and D. Kaeli. Exploring the multiple-GPU design space. In *IEEE International Symposium on Parallel Distributed Processing. IPDPS.*, pages 1–12, 2009.

[34] T. Scogland, B. Rountree, W. chun Feng, and B. De Supinski. Heterogeneous task scheduling for accelerated OpenMP. In *IEEE 26th International Parallel Distributed Processing Symposium (IPDPS)*, pages 144–155, May 2012.

[35] F. Song and J. Dongarra. A scalable framework for heterogeneous GPU-based clusters. In *Proceedinbgs of the 24th ACM symposium on Parallelism in algorithms and architectures*, SPAA '12, pages 91–100, NY, USA, 2012. ACM.