Dynamic Heterogeneous Scheduling Decisions Using Historical Runtime Data

Chris Gregg, Michael Boyer, Kim Hazelwood, and Kevin Skadron
University of Virginia Computer Engineering Labs

Abstract. Heterogeneous systems often employ processing units with a wide spectrum of performance capabilities. Allowing individual applications to make greedy local scheduling decisions leads to imbalance, with underutilization of some devices and excessive contention for others. If we instead allow the system to make global scheduling decisions and assign some applications to a slower device, we can both increase overall system throughput and decrease individual application runtimes.
We present a method for dynamically scheduling applications running on heterogeneous platforms in order to maximize overall throughput. The key to our approach is accurately estimating when an application would finish execution on a given device based on historical runtime information, allowing us to make scheduling decisions that are both globally and locally efficient. We evaluate our approach with a set of OpenCL applications running on a system with a multicore CPU and a discrete GPU. We show that scheduling decisions based on historical data can decrease the total runtime by 39% over GPU-only scheduling and 29% over scheduling that places each application on its preferred device.

1 Introduction

Heterogeneous computing consists of applications running on a platform that has more than one computational unit with different architectures, such as a multi-core CPU and a many-core GPU. Using language frameworks such as OpenCL and software platforms such as Twin Peaks [6], applications running on the CPU launch kernels that can run on either the CPU or the GPU. Generally these kernels perform better on the GPU as they are optimized for a GPU's highly parallel architecture and GPUs typically provide higher peak throughput. Therefore, applications preferentially schedule kernels on GPUs, leading to device contention and limiting overall throughput. In some cases, a better scheduling decision runs some kernels on the CPU, and even though they take longer than they would if run on the GPU, they still finish faster than if they were to wait for the GPU to be free. Furthermore, by utilizing all available processors for computational work, the total throughput of the system is increased over a static schedule that runs each kernel on the fastest device.

This dynamic approach to heterogeneous scheduling hinges on the solution to the problem of being able to predict how long an application will run when it is assigned[1] to a given device. We propose that by capturing and using historical runtime information, a scheduling algorithm is able to make a decision about the tradeoff of forcing an application to run its kernel on a slower device against waiting for the faster device to become available. As a database of application runtimes gets built, scheduling decisions become better. In this work, we show that the database can be as simple as keeping an average runtime for each application, along with information about the input data size. From this information we also show that a dynamic scheduler can improve overall

[1] We describe the process of specifying on which device an application will run its kernel(s) as *assigning an application to a device*. Note that in the case of a GPU, the serial phases of the application will still run on the CPU, and only the kernel(s) will execute on the GPU.

throughput considerably over a statically scheduled solution that only assigns applications to the GPU, and can also improve throughput against a scheduler that assigns an application to whichever device runs it fastest.

Application runtimes also depend on the amount of data that must be processed. Being able to predict the runtime of an application with a different data size than that which is captured in the historical database is also an important factor. We show that it is possible to look at trends in a small data set and predict runtimes for different input sizes for the same application. Because the scheduling methodology we describe analyzes a queue of applications that are ready to launch, some applications later in the queue may finish prior to those that were placed into the queue earlier. However, the algorithm we describe does not unfairly penalize individual applications, and we show that applications continue to progress in the queue (i.e., they are not starved), and they will almost always finish no later than they would have finished using other scheduling methods that launch application in the order they were placed into the queue. We investigate the fairness of the algorithm in Section 4.1.

The contributions of this paper are as follows:

- We present an algorithm that analyzes a queue of applications and assigns them to devices of a heterogeneous system such that overall computational throughput is increased over a statically scheduled solution. We demonstrate that even if all applications natively run faster when assigned to one device, there are situations where assigning an application to a slower device allows that application to complete before it would have if it had waited for the faster device. Furthermore, this solution preserves fairness for an individual's placement in the queue.
- We describe a history database structure that collects and averages runtime data for each application, providing the information to the scheduling algorithm so it can predict when devices will be free.
- We demonstrate how runtime predictions for applications with unique data inputs can be made from data input size differences.
- We implement the scheduler on a set of sixteen OpenCL benchmark applications and demonstrate the improvement of the algorithm over other scheduling decisions for a system that has a CPU and a GPU. We also show that the scheduler produces an improved schedule and a high utilization for both devices even when all applications individually run faster when assigned to the GPU.

2 Problem Definition

Scheduling computational work for heterogeneous computer systems is substantially different than scheduling for systems with homogenous processing cores, and in this work we focus on two of these differences. The first difference is that an application can have a drastically different running time when assigned to different device. Reports of GPU kernels running one hundred times faster than comparable CPU kernels are in the literature (for example, Che et al. [4] and Ryoo et al. [13]), although more recent research has shown that carefully controlled experiments demonstrate speedups that generally fall in the $2x$ to $10x$ range [9]. Because of these runtime differences, assigning an application to one of two devices necessitates knowing which device will allow the application to run faster.

A second difference is that GPU processors do not have the capability to time-slice workloads; i.e., kernels that are launched on a GPU run sequentially, one at a time. The latest GPUs have limited ability for multiple kernels to run in parallel, but there must be careful coordination to ensure that all kernels and their data fit onto the card, and that they do not have any dependencies. Without the ability to time-slice, a kernel launched behind other kernels must wait until all other kernels finish completely before starting its own work. In a time-sliced environment, a scheduler would have the ability to interleave kernels, enabling kernels with a small amount of work to finish in a shorter time period than if they had to wait for longer kernels to finish before running. The algorithm we describe in Section 4 provides a similar result, because fast applications farther back in the queue end up assigned to the device where they will finish first.

Typically, when using a language framework such as OpenCL or CUDA, an application that wishes to run a kernel on a heterogeneous platform queries the system to determine which devices are available, and it preferentially chooses the device that will run the kernel the fastest. In most cases, this is a GPU, and the kernel is optimized to run on this device. Applications therefore tend to all choose the same device, and if a number of applications attempt to launch kernels concurrently, this leads to contention on a device. Furthermore, this type of scheduling ignores devices on the system that can potentially run the kernels and finish them before they would be finished if they were launched on the faster device in queue-order. We propose that instead of letting applications determine where kernels should be launched, a scheduler instead determines the best device at a given time for each kernel by analyzing predicted runtimes of the applications. This scheduler has historical runtime information about the other applications in the queue, and knows which kernels, if any, are currently running.

Given a set of queued applications that are not queued for a specific device, the scheduling problem becomes one of judiciously assigning the applications to devices to maximize computational throughput while remaining fair to the queue order. Many factors can go into this scheduling decision for a given application, including the number of applications ahead in the queue, the application or applications that are currently assigned to the devices on the system and their runtimes, how much data must be transferred between device memory systems, and the relative speed of the application when assigned to each device in the system.

One assumption that we make in order to implement our scheduling algorithm is that kernels can be run on more than one device in a system. Our implementation utilizes OpenCL, which supports running the same kernel across both CPUs and GPUs. Kernels can be compiled for available devices prior to or at runtime, and when a kernel is launched onto a specific device, the OpenCL runtime uses the correct binary for that device. Not all frameworks support running kernels on different devices, however our scheme allows for implementations that have separate versions of a kernel for each available device (e.g., one written in CUDA for GPUs and in OpenMP for CPUs), and the runtime similarly chooses the correct binary once a device decision has been made. Indeed, if an application developer knows that a program is going to be run on a heterogeneous system, it might be appropriate to provide optimized copies of the kernels for all available computational devices. We believe that in the future more frameworks will be cross-compilable for multiple devices (as OpenCL is already), and more applications will ship with kernels that are able to run on more than one device.

To summarize, there are two primary differences between scheduling for a homogeneous set of devices and a heterogeneous set of devices: (1) on the latter, the running time for the same application can be widely different for each device; (2) GPUs do not have the ability to time slice kernels, and a FIFO queue may penalize applications that finish quickly because they have to wait for longer running kernels to complete. We address both problems by using a historical database that contains runtimes for applications on each device and determining a schedule that runs applications on the device in which they will finish first.

3 Collecting and Using Historical Runtime Data

Our method for heterogeneous scheduling relies on historical data about application runtimes. We propose a method for collecting and amalgamating this data such that it is accessible quickly and provides enough information about a given application to be useful for making predictions about how long an application will take when assigned to a device. Table 1 shows the lightweight and extensible data structure we use to store the data. The data structure presented in Table 1 is not exhaustive, but we believe that in this form it is both robust enough to provide worthwhile data and lightweight enough to be useful for fast access.

An application is uniquely defined by a combination of `appName`, `device`, and `appVars`. The `count` value shows how many times the application with a given set of inputs has been assigned to the device, and is used to calculate a running average (`appAvg`) and standard deviation (`appStdev`) for the historical runtimes. In order to calculate the running standard deviation, the sum of squares of differences from the mean (`appM2`) is kept as well. The minimum and maximum runtimes are kept in order to maintain a high and low bound for the running times. The `dataRecvTime` and `dataSentTime` values denote how long it takes an application to transfer data to and from the device where the kernel is run. The `dataRecv` and `dataSent` values denote the size of the data (in MB) that is transferred between devices. The `appVars` value denotes the size of each data input to the application.

Key	Value	Example	Bits
appName	hash	"MatrixMul"	64
count	int	171	32
device	hash	"gpu"	64
appAvg	float	0.04097	32
appMax	float	0.04854	32
appMin	float	0.03749	32
appM2	float	0.00104	32
appStdev	float	0.00248	32
dataRecv	float	8.0000	32
dataRecvTime	float	0.02735	32
dataSent	float	4.0000	32
dataSentTime	float	0.01841	32
appVars	dict.	{x:2048, y:2048}	128

Table 1: Data Structure used to store historical data. Data is kept serialized in a set of dictionary key/value pairs. Each time an application is run the data structure is read and updated.

Application	GPU (ms)	CPU (ms)	GPU Spdup
BinarySearch	60	4	0.07
BitonicSort	933	9122	9.8
FastWalshTrans	77	236	3.1
DCT	49	503	10.3
DwtHaar1D	842	924	1.1
EigenValue	712	1448	2.0
FFT	1.6	0.2	0.13
FloydWarshall	45	527	11.7
MatrixMult	41	4475	109
MatrixTranspose	999	3736	3.7
PrefixSum	112	3	0.03
QuasiRandSeq	18	441	25
Reduction	26	340	13
ScanLargeArrays	17	154	9
SimpleConv	287	481	1.7
SobelFilter	8	13	1.63

Table 2: Applications

4 The Scheduling Algorithm

In this section we describe our dynamic scheduling algorithm that uses the history database described in Section 3. We assume that applications are placed into a first-in-first-out queue and each kernel can run on any of the available devices. For clarity, we also assume that there are two devices available, a CPU and a GPU, although the algorithm could easily be extended to include an arbitrary number of devices. We also assume that most applications will run faster when assigned to the GPU.

4.1 Overview of the Algorithm

In essence, the scheduler we describe implements a greedy algorithm that assigns applications to devices based on a comparison between the predicted times for the application to finish on all available devices. Even if an application would run faster assigned to a particular device, if there are enough applications ahead of it in the queue for that device, it may finish faster assigned to the slower device because that device is free.

Our scheduling algorithm is laid out as follows. We create a sub-queue for each device, and place applications in those sub-queues from the main queue according to the following rules:

1. If the main queue contains applications, attempt to keep all devices busy. If a sub-queue has applications and the device for that sub-queue becomes free, assign the next application in the sub-queue on that device.
2. If one device is busy but the other device is free and the next application in the main queue has not been assigned to that device before, assign it to that device in order to build the database. This is a one-time penalty for applications that run slowly assigned to a particular device, but it is necessary to build the database. Assume that most applications will run faster assigned to the GPU, so if an application only has GPU runtime data but the GPU is free, assign the application to the GPU.
3. If only one device is free and the next application in the main queue runs slower assigned to that device, estimate how long the other device will be busy using the history database and include other application also scheduled to be assigned to that device in its sub-queue. If the next application in the main queue will finish faster by being assigned to the slower device, assign it to that device. If not, put it into the sub-queue for the busy device.
4. Continue through the main queue until both devices are busy running applications. As an application finishes, update the historical database with the runtime information, calculating the average runtime and the standard deviation, and repeat the algorithm from the beginning.

The scheduling algorithm described above continues to improve as more data is entered into the historical database, and each application is penalized at most once when it is assigned to a slower device in order to build the database. Because application runtimes are averaged into the previous runtime for a device, outlying points that could be caused by factors unknown to the scheduler (e.g., GPU contention due to video processing associated with the display) are smoothed out over time.

Because our scheduler assigns applications to devices that are not necessary optimal for each individual application, we must discuss scheduling fairness. We define a schedule to be *fair* if each application finishes no later than it would have finished if it were allowed to execute its kernel on its preferred device. In other words, a fair schedule does not penalize an application even if the application is assigned to its non-preferred device. Accordingly, no starvation occurs, and applications are scheduled for a device in queue order. The algorithm presented earlier generates fair schedules except in two cases: if the predicted runtimes are significantly incorrect or when an application is first encountered and is assigned to a slower device.

4.2 Application Runtime Prediction

If an application has been assigned to a device at least once with a given set of inputs, and the same application is subsequently run with the same input size, the scheduler

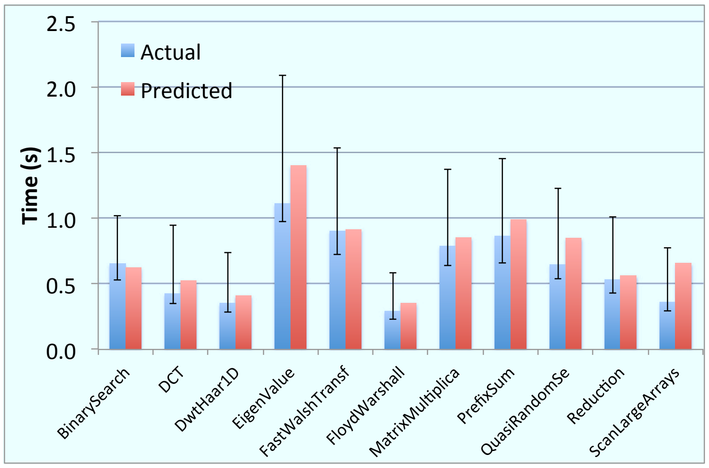


Fig. 1: Actual kernel times versus predicted kernel times. The error-bars indicate the minimum and maximum measured and the height of the bar is the average. The right-hand bars indicate the predicted kernel times after the scheduler has been trained.

uses the average application time as a runtime prediction for that device. When an application has been assigned at least twice to a given device with different inputs, and is again run with another different input, the scheduler makes the prediction based on a linear least-squares calculation, using the input sizes as one parameter, and the previous runtimes as another. Figure 1 shows the actual runtimes versus the predicted runtimes. The height of the left-side bars denote the average runtimes for the applications over 150 trials, and the error bars represent the minimum and maximum times. The right-side bars denote the predicted time after the scheduler has been trained. In our experiments, the predicted runtimes fell within the actual minimum and maximum runtimes, which was sufficient to use for the scheduling decision.

4.3 Scheduling Overhead

Although we discuss kernels running on both the CPU and the GPU as being independent from each other, this is not strictly the case. In current CPU/GPU architectures, the applications that launch kernels run themselves on the CPU, and the scheduler that runs also uses a CPU thread to coordinate the scheduling. When scheduling kernels to run on the CPU, the kernels utilize standard operating system threads, and therefore contribute to overall CPU usage. Any scheduler that launches kernels on the CPU inflicts a penalty on any applications running on the CPU, including other applications that are or will be running kernels. When we present our results in Section 5, we point out cases where this overhead is noticeable, but we found that our implementation still produces a better schedule than a GPU-only solution.

Any scheduling decision incurs an overhead simply because of the time necessary to run through the scheduling algorithm. It would be imprudent if the overhead of making a dynamic scheduling decision decreased computational throughput relative to a statically scheduled solution. The algorithm we have presented is very lightweight, especially compared to the application runtimes we have investigated. With the application database loaded into memory, our results show that the time to make a scheduling decision averages 0.7ms with a standard deviation of 0.2ms on our test platform, described in Section 5. GPU applications average 0.4 seconds, making the scheduling decision much less than 1% overhead. Furthermore, our test scheduler is written in Python and is not optimized for speed. Increases in database size and improvements to the scheduling algorithm will add to this overhead, but because the dynamic scheduling improves throughput significantly, even an overhead of one or two percent would still make the scheduler worthwhile.

4.4 Scalability

Because historical data is collected and updated each time an application is run on a device, it scales well as more applications are run on the system, or as different devices are added or replaced. The history database is kept to a manageable size by keeping runtime averages per application on each device. As application queue sizes increase, there is an increased overhead to analyzing the finish time for each application at the head of the queue, but the current finish time of all items in a device queue can be kept as a running total, so this overhead is minimal.

One concern with adding additional applications is that there is an inherent training period for each added application that does not have any associated history information. Because the database continually improves as applications are run, we make the assumption that over time the scheduler has seen most applications multiple times. Although we limited our tests to a single CPU / single GPU system, it would not be difficult to amend the scheduler to accept new devices in the system. As new devices are added to the platform, the scheduler would collect runtime data steadily and update the database accordingly. There would be a training period associated with these updates, but overall throughput would be increased as soon as a device was added to the system in most cases simply by virtue of the work being spread across more devices.

5 Experimental Results

We first describe the workload we used to test our scheduling algorithm, and then present the experimental results, compared to other scheduling schemes.

5.1 Workload and test environment

In order to test our algorithm, we used sixteen OpenCL benchmark applications with one kernel each, and ran the set of applications sequentially, for a total of 16 kernel launches. The applications we used in our experiments represent a number of computational algorithms that are commonly used in scientific computing. Table 2 shows the applications and the absolute and relative runtimes for the data sets that we tested with. As expected, most applications had kernels that ran faster on the GPU, and therefore the entire applications ran faster on the GPU. In order to demonstrate the scheduler when some application were faster assigned to the CPU, we set the data size small enough for three applications such that this was the case (Binary Search, FFT, and Prefix Sum). In Section 5.4 we remove these applications and demonstrate the scheduler when all applications run faster assigned to the GPU.

Our test environment was comprised of a 6-core, 3.7GHz AMD Phenom II 1090T CPU with 4GB of main memory and an AMD 5870 GPU with 2GB of memory. All tests were run under Ubuntu 10.04. In order to run and test the OpenCL applications, we wrote a Python application that simulated a runtime where all sixteen applications were placed into a scheduling queue and scheduled according to our dynamic algorithm.

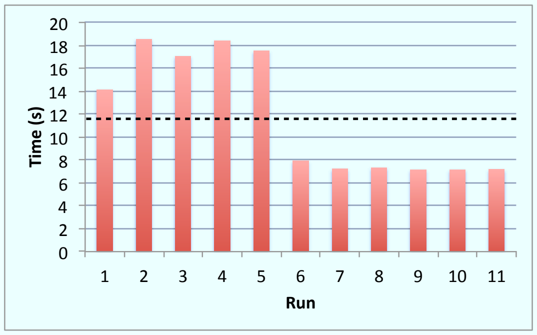


Fig. 2: Scheduler training. The scheduler was run multiple times against the queue of sixteen applications, starting with no initial historical data. The dashed line denotes the time for all applications to run on the GPU.

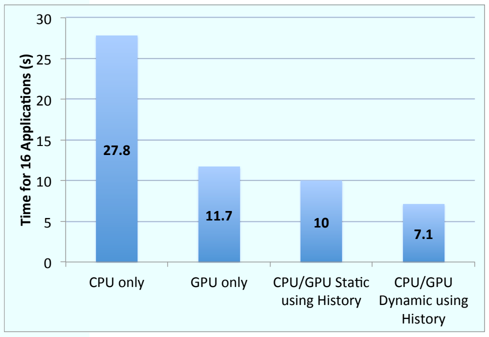


Fig. 3: Overall results for dynamic scheduling compared against assigning all application to the CPU, all applications to the GPU, and static scheduling based on the fastest device per application.

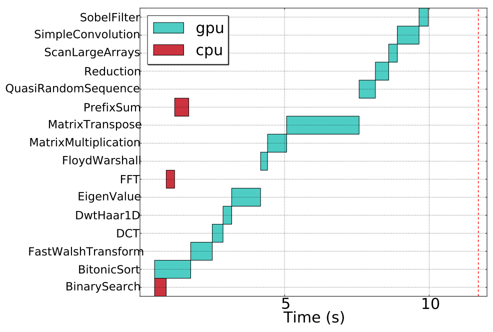
5.2 Training the scheduler

Figure 2 shows the improvement of our dynamic scheduler as the historical database improves. For the purpose of our experiments, we trained the scheduler by running the same applications in our benchmark, but with a random set of input sizes. We also chose a sample fixed set of input sizes to test with the database after each training run, in order to normalize the results for Figure 2. Initially, with limited or no historical information, applications are assigned to the GPU if it is free and to the CPU otherwise. Compared to a GPU-only scheduling solution, the scheduler performs worse on the first few runs, but it quickly improves its performance. The database gets continually cross-trained as it collects data on all the applications that it sees, and the scheduler benefits any time it sees an application more than once, regardless of the data set size. The scheduler always performs better than a CPU-only solution for this set of applications. Results in the following section reflect data taken after five training runs.

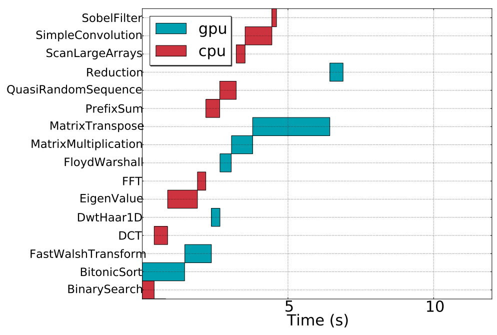
5.3 Results

Figure 3 shows the time needed to run the set of 16 applications for four different scheduling algorithms. In the CPU only and GPU only cases, all applications were assigned to each respective device. For the "CPU/GPU Static using history" case, each application is assigned to its preferred device. The final column shows the results using the dynamic scheduler we have described. As the figure indicates, the dynamic scheduler that uses historical information is 29% faster than the static scheduler that knows which device provides each application its best performance, and 39% faster than a scheduler that assigns all applications to the GPU.

Figure 4 shows the results of two schedules that perform better than a GPU-only solution. Figure 4(a) a static schedule based solely on running application on the fastest device per application. Because only three short application ran faster assigned to the CPU, the utilization of the CPU is only 17%, and most of the applications were assigned to the GPU. Figure 4(b) shows the results from our algorithm, demonstrating 67% CPU utilization, and 39% faster than the GPU-only solution (shown in both Figures 4(a)



(a) Fastest device per application, static determination. CPU utilization: 17%, GPU utilization: 100%. Time: 10.0 seconds. The dashed line is the time to run all applications on the GPU.



(b) Historical data, dynamic decision, based on algorithm from Section 4. CPU utilization: 67%, GPU utilization: 100%. Time: 7.1 seconds. The dashed line is the time to run all applications on the GPU.

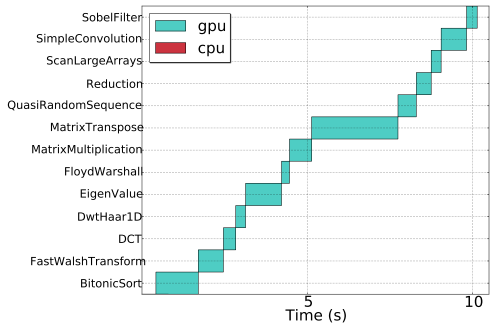
Fig. 4: Execution flow for the set of sixteen applications. In (a), all applications were assigned to the GPU. In (a), the static scheduler utilizes the CPU, but only for applications that are faster on that device. In (b), the dynamic scheduler balances the application well across both devices, leading to a much higher throughput.

and 4(b) with a dotted line). Our scheduler out performs the static scheduler based on the fastest device per application by 29%.

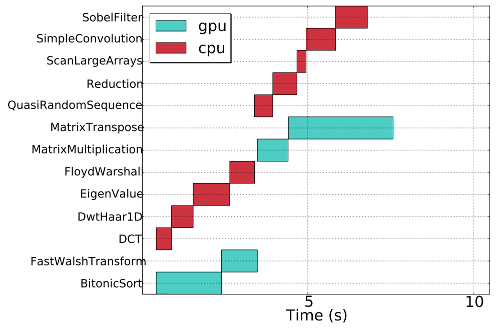
5.4 Running the scheduler when all applications are faster on the GPU

So far, we have described scheduling results from a mixed set of applications that includes some applications that run faster assigned to the CPU rather than the GPU. In practice, it is rare to find applications that run faster assigned to the CPU for anything other than very small data sets, as application developers generally target and optimize kernels to run on GPUs. Our dynamic scheduler is still able to produce worthwhile scheduling results even when each individual application is faster when assigned to one device, especially as the application queue sizes increase. As an illustration, consider a queue of ten identical applications that each run $5x$ faster when assigned to the GPU instead of the CPU. As soon as the sixth application gets assigned to the GPU, it is beneficial to assign the seventh to the CPU, even though it will take five times as long, because it will finish before it would have if it waited for the previous six applications to finish. Furthermore, instead of 0% CPU utilization, it is very high.

Figure 5 shows the results of running the dynamic scheduler on a set of thirteen applications where all applications run faster assigned to the GPU. The scheduler is able to decrease the runtime for all application by assigning a number of applications to the CPU, and this schedule produces a 90% CPU utilization and is 33% faster than the GPU-only schedule. As can be seen by comparing the time for `BitonicSort` and `Matrix Transpose` between (a) and (b), both applications take longer assigned to the GPU when scheduled alongside a CPU application, demonstrating the overhead mentioned in Section 4.3. This overhead does increase the time to run some individual applications, but the overall results demonstrate that the benefit from running the dynamic schedule over a GPU-only schedule.



(a) GPU only. CPU utilization: 0%, GPU utilization: 100%. Time: 10.1 seconds.



(b) Fastest device per application, dynamic decision. CPU utilization: 90%, GPU utilization: 100%. Time: 7.6 seconds.

Fig. 5: Execution when all applications inherently run faster when assigned to the GPU. The dynamic scheduling finishes all applications 33% faster than the statically scheduled GPU-only solution.

6 Related Work

The idea of using historical data for heterogeneous scheduling decisions has been discussed in other work, although very few have targeted GPU computing. Ali et al. show that using a historical prediction database is worthwhile for grid computing [1], and Siegel et al. [14] discuss automated heterogeneous scheduling where one stage is to profile tasks and to perform analytical benchmarking of individual tasks. This information is then used in a later stage to predict runtimes for applications based on current processor loads. Similarly, our approach profiles applications as they run, but also extrapolates runtimes for applications with different input sizes from an analytical assessment of the collected data. Topcuoglu et al. describe the "Heterogeneous Earliest-Finish-Time" (HEFT) algorithm [15], which, like our approach, attempts to minimize when individual applications finish. Maheswaran et al. [11] describe a set of heuristics that inform a dynamic heterogeneous scheduler. Their *Min-min* heuristic calculates which device will provide the earliest expected completion time across a set of tasks on a system. The task in the queue that will complete first is scheduled next. Both Topcuoglu et al. and Maheswaran et al. were written prior to the advent of GPU computing, and they simulated their algorithms. Our approach differs from both Topcuoglu et al. and Maheswaran et al. by considering fairness, ensuring that applications do not get pre-empted by other applications, and we also tested our scheduler on CPU/GPU heterogeneous hardware.

Jiménez et al. [8] demonstrate two predictive algorithms that use performance history to schedule a queue of applications between a CPU and a GPU: `history-gpu`, that schedules work on the first available device that can run an application, and `estimate-hist`, that estimates the waiting time for each device and schedules an application to the device that will be free the soonest. Our proposed scheduler expands on their work by also predicting when the application that is next in the queue will complete, and scheduling it on whichever device will allow it to complete the fastest. We also describe methods for looking at runtime trends to predict runtimes for applications with unique data size inputs.

Luk et al. [10] use a historical database for *Qilin* that holds runtime data for applications it has seen before, although *Qilin* focuses on breaking a single application across multiple devices instead of running multiple applications across multiple devices, as we do. Augonnet et al. [2] use performance models to provide scheduling hints for their StarPU scheduler, and programmers who write applications for StarPU can provide a "cost model" for each application that enables the scheduler to predict relative runtimes. Our approach does not require programmers to modify their code. Becci et al. [3] use performance and data transfer overhead history to inform a dynamic heterogeneous scheduler for legacy kernels, focusing on postponing data transfer between devices until it is actually needed.

Harmony [5] also uses performance estimates in order to schedule applications across a heterogeneous system. They propose an online monitoring of kernels and describe a dependence-driven scheduling that analyzes how applications share data and decides on processor allocation based on which applications can run without blocking. Our approach considers applications to be independent, and schedules applications from multiple applications concurrently.

Several researchers have proposed using performance models to predict runtime (e.g., Meng and Skadron [12] and Hong et al. [7]), but performance models have high overhead and are generally not portable between hardware generations. We believe that using historical runtime data provides a better prediction for kernel runtimes.

7 Conclusions

In this paper, we described and demonstrated a dynamic scheduling algorithm for heterogeneous computers that schedules application based on a historical database of runtime values. We showed that by storing and using the historical information, a scheduler can determine how to assign applications to heterogenous processors that utilize all devices available and allow for a greater computational throughput than simply assigning all applications to one device, or by other static scheduling means. The resulting schedule fairly schedules applications according to their order in the queue, and if the runtime prediction is relatively accurate, applications will finish running prior to when they would have if they had all been statically scheduled onto the GPU, or if they had been scheduled to run on the device on which they run fastest. Furthermore, we demonstrated that a scheduler that utilizes all available devices even when each application runs faster assigned to one particular device can still provide better overall throughput even though some applications get assigned to slower devices.

We presented a robust yet compact historical database data structure that contains enough data to make predictions for application runtimes, and we described how the database gets populated and trained. We demonstrated that our dynamic scheduling algorithm can increase the computational throughput for a set of sixteen applications by over 30% over a GPU-only static scheduling solution, and we showed that the algorithm also provides device utilization that is over 80% for all devices in the system.

For future work, we will take a look at how a historical scheduler could be used in a cluster of CPU/GPU machines, and for other heterogeneous machines including Cell/B.E. or embedded systems. Additionally, we will investigate how historical prediction for dynamic scheduling can be used for applications with interdependencies.

Acknowledgments

This work was supported in part by NSF grants CNS-0747273, CSR-0916908, and CNS-0964627, SRC GRC task number 1790.001, and a grant from Microsoft. We would also like to thank the anonymous reviewers for their helpful comments and suggestions.

References

1. A. Ali, A. Anjum, J. Bunn, R. Cavanaugh, F. van Lingen, R. McClatchey, M. A. Mehmood, H. Newman, C. Steenberg, M. Thomas, and I. Willers. Predicting the resource requirements of a job submission. In *Computing in High Energy Physics*, pages 130–134, Interlaken, Switzerland, September 2004.
2. C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. In *Euro-Par Conference on Parallel Processing*, pages 863–874, 2009.
3. M. Becchi, S. Byna, S. Cadambi, and S. Chakradhar. Data-aware scheduling of legacy kernels on heterogeneous platforms with distributed memory. In *ACM Symposium on Parallelism in Algorithms and Architectures*, pages 82–91, 2010.
4. S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, and K. Skadron. A performance study of general-purpose applications on graphics processors using CUDA. *Journal of Parallel and Distributed Computing*, 68(10):1370–1380, 2008.
5. G. F. Diamos and S. Yalamanchili. Harmony: an execution model and runtime for heterogeneous many core systems. In *17th International Symposium on High Performance Distributed Computing*, pages 197–200, Boston, MA, 2008.
6. J. Gummaraju, L. Morichetti, M. Houston, B. Sander, B. R. Gaster, and B. Zheng. Twin peaks: a software platform for heterogeneous computing on general-purpose and graphics processors. In *19th Conference on Parallel Architectures and Compilation Techniques*, pages 205–216, Vienna, Austria, 2010.
7. S. Hong and H. Kim. An integrated gpu power and performance model. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 280–289, New York, NY, USA, 2010.
8. V. J. Jiménez, L. Vilanova, I. Gelado, M. Gil, G. Fursin, and N. Navarro. Predictive runtime code scheduling for heterogeneous architectures. In *4th Conference on High Performance Embedded Architectures and Compilers*, pages 19–33, 2009.
9. V. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, et al. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *37th International Symposium on Computer Architecture*, pages 451–460, 2010.
10. C.-K. Luk, S. Hong, and H. Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 45 –55, 2009.
11. M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. page 30, Los Alamitos, CA, 1999.
12. J. Meng and K. Skadron. Performance modeling and automatic ghost zone optimization for iterative stencil loops on gpus. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 256–265, 2009.
13. S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, Salt Lake City, UT, 2008.
14. H. J. Siegel, H. G. Dietz, and J. K. Antonio. Software support for heterogeneous computing. *ACM Computing Surveys*, 28(1):237–239, 1996.
15. H. Topcuoglu, S. Hariri, and M.-Y. Wu. Task scheduling algorithms for heterogeneous processors. In *8th Heterogeneous Computing Workshop*, pages 3–14, 1999.