OpenCL Task Partitioning in the Presence of GPU Contention

Dominik Grewe, Zheng Wang, and Michael F.P. O'Boyle

School of Informatics, The University of Edinburgh, UK
dominik.grewe@ed.ac.uk,zh.wang@ed.ac.uk,mob@inf.ed.ac.uk

Abstract. Heterogeneous multi- and many-core systems are increasingly prevalent in the desktop and mobile domains. On these systems it is common for programs to compete with co-running programs for resources. While multi-task scheduling for CPUs is a well-studied area, how to partitioning and map computing tasks onto the hetergeneous system in the presence of GPU contention (i.e. multiple programs compete for the GPU) remains an outstanding problem.
In this paper we consider the problem of partitioning OpenCL kernels on a CPU-GPU based system in the presence of contention on the GPU. We propose a machine learning-based approach that predicts the optimal partitioning of OpenCL kernels, explicitly taking GPU contention into account. Our predictive model achieves a speed-up of 1.92 over a scheme that always uses the GPU. When compared to two state-of-the-art dynamic approaches our model achieves speed-ups of 1.54 and 2.56 respectively.

1 Introduction

Integrated GPUs are becoming ubiquitous for desktop PCs and mobiles. The integrated GPU utilizes a portion of the system's RAM rather than dedicated graphics memory, which shares the same memory space with the CPU. They are less costly when compared to the dedicated GPUs, while still providing parallel computing resources for certain classes of applications. There is an increasing number of applications that make use of the integrated GPU on PCs and mobiles. On these systems it is typical to have multiple programs running at the same time, competing for the shared resources including the GPU. Under such settings, decisions about which computing device (the CPU or the GPU) to use to run the program and how the work should be partitioned across different devices have significant impact on the application's performance.

While multi-task scheduling on the general purpose CPU is a well-studied area, how to partitioning and map tasks onto the underlying platform in the presences of GPU contention remains an outstanding problem. Currently the use of the computing device is statically determine and hard-coded when the application is built. Given that the availability of resources and the behaviours of the workload programs vary in a multi-programmed environment and have a dramatic impact on the correct mapping and scheduling of work, entirely static approaches are likely to fail. What is needed is an approach that can adjust the mapping decision according to the dynamic computing environment by taking into consideration the target program behavior.

Several methods for automatically mapping tasks to devices in a heterogeneous system have been proposed. Luk et al. [14] use offline profiling to determine the best partitioning between the CPU and GPU while Grewe and O'Boyle [6] apply machine learning techniques to predict the optimal partitioning. Both approaches deliver good results but only under the assumption that no other programs are running on the system. Ravi et al. [16] use a dynamic, "task farm" approach for task mapping. They divide the task into a fixed number of chunks and send one chunk to each device. When a device finishes processing it requests a new chunk. As we will show in this paper, this dynamic approach delivers poor performance in the presence of GPU contention.

In this paper a new task partitioning approach is introduced that explicitly takes GPU contention (i.e., multiple programs are competing for the GPU) into account. Unlike most dynamic approaches which require an online searching phase to determine the best partition of work, it uses a machine learning-based predictive model to directly predict the best work partition using code features of the program and runtime information. Unlike previously dynamic approaches, our scheme avoids the potential expensive online searching overhead by directly predicting the best partitioning scheme. The other advantage is that our model is automatically generated off-line at the factory. This avoids the pitfalls of using a hard-wired heuristic that requires human modification whenever the hardware changes.

Across a set of 22 benchmarks and 10 different contention scenarios the predictive model achieves a speed-up of 1.92 over using only the GPU and 1.23 over using only the CPU. When compared to two dynamic approaches our approach achieves speed-ups of 1.54 and 2.56 respectively.

2 Motivation

This section demonstrates the importance of explicitly taking GPU contention into account when mapping programs to heterogeneous systems.

Figure 1 shows the running time of the `nbody` benchmark in three GPU contention scenarios: no contention, and medium and heavy contention. The medium and heavy contention scenarios are created by running a separate application which uses the GPU alongside the target program to be optimized. More details on which applications were used is given in section 5.1. Along the x axis different static partitioning configurations (represented as the percentage of work mapped to the CPU) are explored. On an idle system (i.e. the no contention scenario) the running time of using only the GPU ($x = 0$) is shorter than the running time of using only the CPU ($x = 100$). This changes, however, when the GPU contention is introduced. Similarly, the optimal partitioning between the devices changes in different contention scenarios. When the system is idle, the best performance is achieved by a $30-70$ split but already in medium contention more work should be assigned to the CPU, namely 50%. In a heavy contention

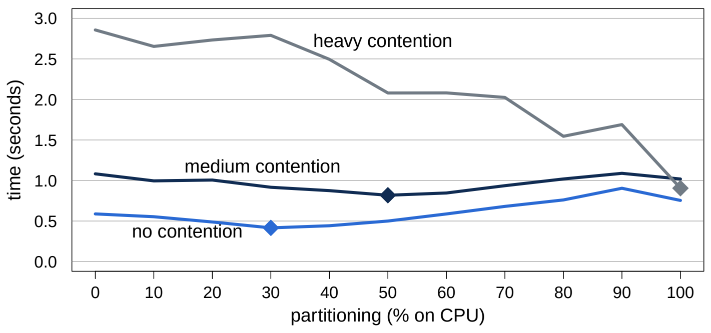


Fig. 1: Running time of the `nbody` benchmark in multiple GPU contention scenarios for different partitionings. The diamonds indicate the optimal partitioning for the corresponding contention.

scenario the runtime of the application increases significantly when some of the work is mapped to the GPU. The best performance is thus achieved when only the CPU is used. The partitioning that is optimal on an idle system ($x = 30$) leads to a $3x$ *slow-down* over the optimal partitioning.

This example demonstrates the need for partitioning techniques that can adapt to contention on the GPU. As the contention information can only be obtained at runtime, any static schemes will fail to achieve good performance on systems being shared by multiple co-running applications. What we need is a dynamic scheme that can adapt to various GPU contention scenarios. The next section will discuss the challenges to be tackled for designing such a dynamic scheme in the presence of GPU contention.

3 Challenges in the Presence of GPU Contention

This work targets OpenCL applications because OpenCL is emerging as a standard for heterogenous computing, which allows the same code to be executed on different computing devices including CPUs and GPUs.

In OpenCL, kernels are launched through command queues. Each time a kernel is being executed it passes through multiple phases, i.e. queueing, ready and execution, as depicted in figure 2. Unlike the general purposed CPU where multiple programs can run simultaneously by sharing the CPU time, only one kernel is allowed to execute at a time[1] on the GPU. An OpenCL kernel may therefore have to spend an significant amount of time in the ready phase waiting for the GPU to become available if the GPU is being used by another program. Furthermore, GPU tasks are non-preemptive, i. e. once a task gains access to the GPU no other task can use the GPU until the current task has finished

[1] NVIDIA GPUs allow concurrent executions of kernels from the same application but not from different applications.

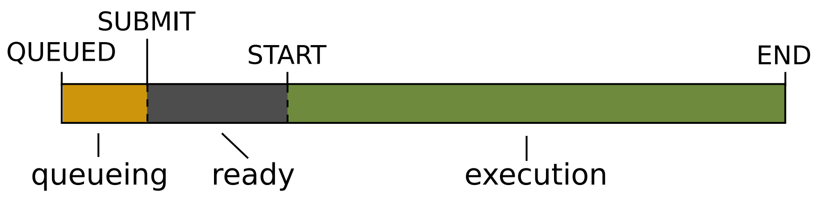


Fig. 2: The three pases of launching and executing an OpenCL kernel. The labels on top correspond to the CL_PROFILING_COMMAND_* parameters passed to the clGetEventProfilingInfo function of the OpenCL API.

execution. This means that a kernel scheduled to the GPU may have spent a long time in the ready phase if a different applications has previously launched a long-running kernel. Therefore, non-careful mappings of kernels onto the GPU at the presence of other GPU workload can lead to longer waiting time and result in poor performance. The goal of this work is to determine how the work of OpenCL kernels should be partitioned across the CPU and the GPU to give the best performance by taking the GPU contention into consideration.

The OpenCL API provides an interface for querying the starting and finishing time of each of the phases as indicated by the labels in figure 2. This allows one to get insights into the behavior of OpenCL applications in the presence of GPU contention. Figure 3 shows the behavior of the `nbody` benchmark in the presence of heavy contention. Each bar represents the running time of a kernel launch, divided into the times spent in each of the three phases. During the first few kernel executions the system is idle. Only halfway through the contention is introduced on the respective device. This highlights the difference in behavior on an idle system and one with resource contention.

When the system is idle the overall running time of each kernel is dominated by the execution phase which is stable across these runs. Time spent in the queueing and ready phases is minimal (they are not even visible in the graphs) because no other applications compete for resources. The behaviour of the application is the same and it takes around 19ms to run on the GPU. When a competing application is launched, however, the behavior is different. Under such a setting, time spent in the execution phase on the GPU remains the same at 19ms because the kernel will always have exclusive access to all GPU resources. However, time spent waiting for the GPU to became available (the ready phase) shows dramatic variation. Sometimes it is as low as on the idle system but it can be as high as 120ms (more than 6 times of the time spent in the execution phase). The variation is due to other applications blocking the GPU. If a long-running kernel has been launched before a kernel is submitted it has to wait for the long-running kernel to finish. If, however, the long-running kernel is just about to finish when the kernel is launched the wait time is small. The total execution time of a kernel launch therefore varies significantly even if the contention is constant, i. e. a fixed co-running applications. In such heavy GPU contention

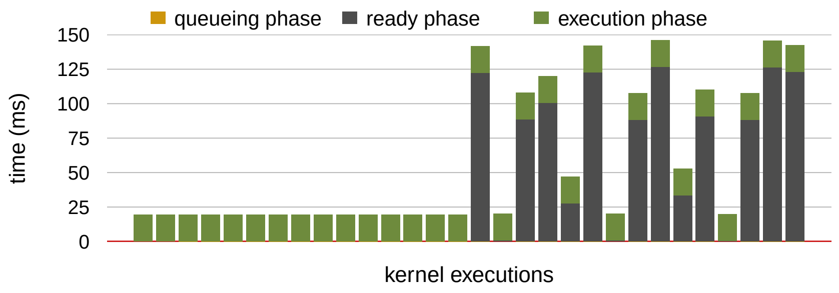


Fig. 3: Profiling of kernel launches in the presence of GPU contention. Each bar shows the total running time of a kernel launch; broken down into queueing, ready and execution times. During the first half the system is idle, then another application is launched competing for resources of the GPU.

it is thus better to use the CPU which provides a much quicker running time (25ms v.s. 120ms). This unpredictable behavior of OpenCL applications in the presence of GPU contention provides a big challenge to schedulers trying to find the best partitioning of a kernel across devices. The next section will describe how we build a dynamic scheme that can adapt to the GPU contention using predictive modeling.

4 Predictive Modeling

This section describes how a machine learning-based model can be built for determining a good partitioning for OpenCL tasks in the presence of GPU contention. The input to the predictive model contains information on both the OpenCL kernel and the GPU contention. Its output is a ratio describing the amount of work to map to the CPU and the GPU.

4.1 The Features of the Model

The inputs of model are two sets of numerical values, namely *feature sets*, that represent the input program and the runtime contention. The first set of features describe the OpenCL kernel itself. They are extracted using a static analysis tool based on clang [12]. The second set of features characterize the contention on the GPU device. These features are constructed from information readily available via the OpenCL API.

Program Features The program features contain information about the number and types of instructions in a kernel. Additionally, the number of coalesced memory accesses is determined. The benchmarks used for this study, as described in section 5.1, often contain vector data types because they were targeted towards both CPUs and GPUs. Another feature is thus the number of vector operations in the code. The full list of program features is shown in table 1.

Program Features
1: # global memory accesses
2: # compute operations
3: # conditionals and loops
4: communication-to-computation ratio
5: percentage of vector operations
6: percentage of coalesced memory accesses
7: # work-items

Table 1: List of program features used by the predictive model.

Contention Features Contention on the GPU is experienced by increased delays in the ready phase waiting for the device to become available (see section 3). The specific GPU kernel causing the contention does not have any other influence on the remaining programs because access to the GPU is exclusive. The best way to characterize GPU contention is thus to quantify this delay. Since the delay exhibits a significant variance a single observation does not carry much information. Therefore, to characterize the contention the average delay (using the arithmetic mean) is computed over time.

4.2 Building the Model

Machine learning models are built by fitting a mathematical model to *training data*. Training data are observations where both the features (input) and target (output) are known. In our case the training data comprise a set of benchmarks and contention scenarios together with the optimal partitioning in each case. The process of obtaining this data is described in the next section. Once the model has been built predictions for new programs and contention scenarios can be made. This process is depicted in figure 4.

In order to model the problem of task partitioning a multi-class classification model is used. Specifically, the model is based on support vectors machines (SVMs) [3]. SVMs try to find hyperplanes in the feature space that separate data points from two different classes. By combining multiple SVMs multi-class problems can be modeled. In order to better find hyperplanes separating the data, kernel functions are used to map the input data into a high-dimensional space. More information on SVMs can be found in [2].

4.3 Collecting Training Data

A set of training programs and “workload” programs are used to collect training data for the predictive model. The workload programs introduce contention on the GPU by either using it for graphics or by executing OpenCL kernels on it. A detailed discussion of which programs were used is given in section 5.1.

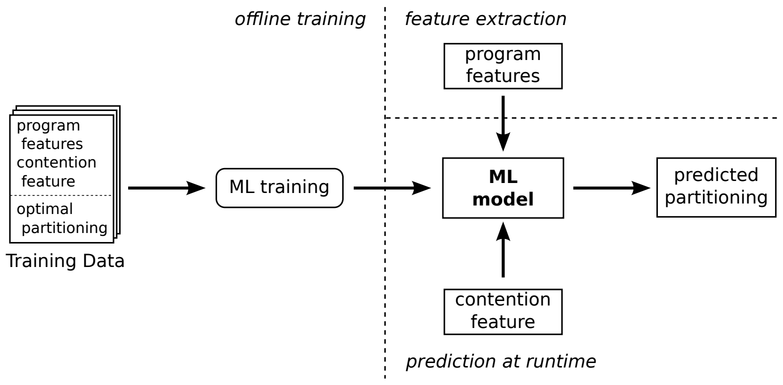


Fig. 4: Building the machine learning (ML) model.

Each combination of training and workload program is executed with different partitionings of the training program. In total eleven different partitionings are evaluated, ranging from CPU-only to GPU-only execution in steps of 10% as shown in figure 1. To ensure a constant degree of contention the workload program is started a few seconds ahead of the training program and input parameters are chosen so that it only finishes after the training program has finished execution.

The best partitioning for each scenario is computed by finding the one with the shortest overall running time. Since some benchmarks contain multiple kernels this computation is done on a per-kernel basis using OpenCL profiling information. The running time of a kernel that is partitioned across the CPU and GPU is defined as the maximum running time across the two devices.The optimal partitions form the targets of the predictive model. The features are collected by performing static analysis on the training program and by recording the incurred delay using the profiling functions provided by OpenCL.

4.4 Deployment of the Model

The model can only be evaluated at run-time because it partially relies on runtime information. On the one hand, the number of work-items needs to be known to compute the program features. This is often only known at run-time. Furthermore, the current GPU contention can obviously only be determined when the program is actually running.

Computing the contention features involves monitoring the waiting times on the GPU and computing the average waiting time as described in section 4.1. Only a window of waiting times of the last few kernel executions should be used, however, to be able to adapt to changes in the contention. This information can either be obtained through previous kernel executions of the program (assuming it launches a sequence of kernels) or by sharing this information across programs.

	CPU	**GPU**
Model	Intel Core i5-3570K	Intel HD Graphics 4000
Core Clock	3.40 GHz	1.15 GHz
Core Count	4	16
Peak Performance	108.8 GFLOPS	147.2 GFLOPS
System Memory	8 GB	
Operating System	Windows 7 Professional SP 1	
OpenCL SDK	Intel SDK for OpenCL Applications 2013	

Table 2: Experimental Setup.

5 Experimental Methodology

This section describes the setup used for evaluating the approach presented in this paper. It details how and which aspects of the model were evaluated and describes which other methods it was compared against.

5.1 Experimental Setup

Platform All experiments were carried out on an Intel IvyBridge platform with a quad-core CPU and an integrated GPU. Full details are shown in table 2. The aim of this work is to find the best partitionings of OpenCL kernels between the CPU and the integrated GPU. The system was running on Windows 7 and the Intel SDK for OpenCL Applications 2013 [10] was used. Each measured run was repeated 10 times and the average execution time was recorded.

Integrated platforms such as the Intel IvyBridge chip are increasingly common in the desktop and mobile computing space. The trend is to further integrate the CPU and GPU in order to allow close cooperation between the two types of processors.

Benchmarks We used 22 different benchmarks from the Intel SDK [10] and the AMD SDK [1] to evaluate our approach. These benchmarks were chosen because they are not specifically tuned for GPUs but for use on both CPUs and GPUs, e. g. by using vector data types. They thus provide for more interesting partitioning scenarios. In order to increase the set of training points each benchmark was used with multiple input sizes.

The main computational parts of the benchmarks were executed repeatedly to ensure a minimum running time of around 500ms. This was done to expose each benchmark to the fluctuations of GPU contention as shown in section 3. It further allows the online search method to find a good partitioning.

Contention Scenarios To introduce contention to the system a range of applications using GPUs was used. These mainly include OpenCL benchmarks targeting the GPU but also a video player application (VLC). Additionally, the scenario of the idle system, i. e. without any contention, was evaluated. A list of all contention scenarios is given in table 3. The entries are ordered by how disruptive they are in terms of the average waiting caused as described in section 4.1.

Name	Type	Waiting Time (μs)
none	no contention	65
vlc	video player	68
sobel-512	OpenCL application	759
monte_carlo	OpenCL application	1,306
sobel-1024	OpenCL application	3,228
aes-512	OpenCL application	8,471
sobel-2048	OpenCL application	12,584
aes-1024	OpenCL application	16,259
aes-2048	OpenCL application	21,944
sort	OpenCL application	36,585

Table 3: Contention scenarios. Ordered from lowest to highest contention.

5.2 Comparison

Our approach is compared to three approaches: "oracle", "task farm" and "online search". Unless stated otherwise, performance is shown as the speedup over CPU-only execution.

Oracle. This is a theoretical scheduler that always picks the best static partitioning. It thus provides an estimate of the upper bound performance available in each scenario.

Task farm. This is a dynamic approach which splits each task into a fixed number of chunks. Initially, one chunk is sent to each device and devices request more work after they have finished processing their chunk. For this evaluation we specified the number of chunks to be 8, which leads to the best average performance on the platform we used.

Online search. This dynamic approach finds a good partition over time. For each kernel the scheme keeps track of what the partitioning between the CPU and GPU is. The partition is represented by a *split value* which is the percentage of work mapped to the CPU. The split value is set to 50% initially, which will be adjusted over time to balance the running times on the CPU and the GPU.

5.3 Evaluation Methodology

We evaluated our approach using the standard *leave-one-out cross-validation* technique. When predicting for a certain benchmark and contention scenario no data from that benchmark, including data from runs with different input sizes, or that contention scenario were used in building the model. It was assumed that information about the GPU contention is available. Section 4.4 provides a brief discussion on how this information can be obtained.

Each benchmark run contains multiple iterations of the main computational phase of the program. This is a common scenario in, for example, linear algebra applications or video processing. Having multiple iterations allows the online

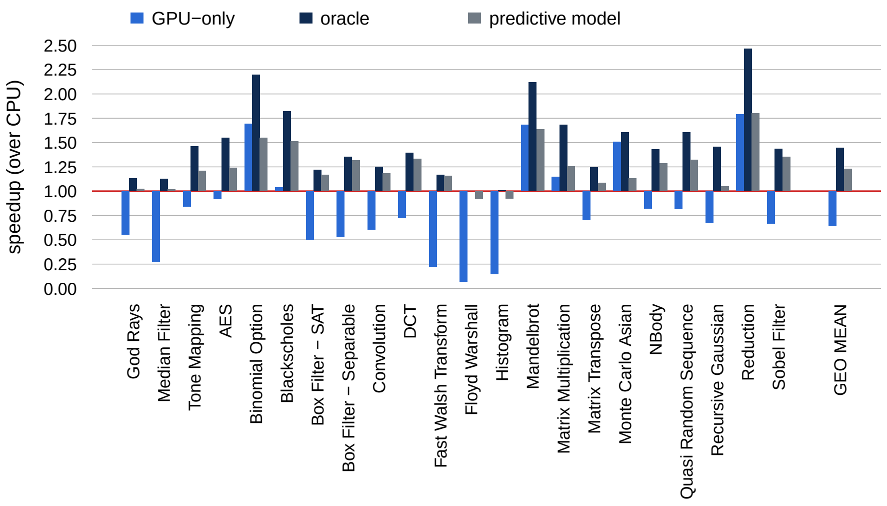


Fig. 5: Speed-up over CPU-only execution averaged across all ten contention scenarios. The GPU-only, oracle and predictive model achieve average speed-ups of 0.61, 1.48 and 1.24 respectively.

search method to find a good partitioning between the CPU and GPU. Furthermore, due to the variable waiting time shown in section 3 it is needed to achieve consistent results.

6 Results

This section evaluates and analyzes the performance of the proposed approach. Firstly, we compare our model and a scheme that uses only the GPU to an oracle scheduler. Then, the performance of the two dynamic schemes is evaluated and compared to our model.

6.1 Comparison to Oracle

Overall Figure 5 shows the performance of the oracle and the predictive model. Additionally, the performance of a GPU-only approach, which executes all kernels on the GPU, is shown because OpenCL applications typically target the GPU. The performance on each benchmark is shown, averaged across all ten contention scenarios. The numbers are normalized to (parallel) CPU-only execution.

With the exception of five benchmarks, the GPU-only approach leads to on average 1.5x slow downs over CPU-only execution. As shown in section 2 GPU execution can be severely delayed in contention leading to increased running times. The oracle results demonstrate, however, that when using the GPU in the right way significant speed-ups can be achieved. An averaged speed-up of 1.43 (up to 2.47) can be observed across all benchmarks *and* contention scenarios.

Our predictive modeling approach achieves performance close to the oracle for a number of benchmarks, e. g. `DCT` or `Sobel Filter`. For all but three benchmarks it outperforms the GPU-only approach and in only two cases can slow-downs over the CPU be observed. On average, the predictive model achieves a speed-up of 1.23 which translates to 86% of the oracle performance, compared to 45% and 70% for the GPU-only and CPU-only approaches, respectively. These results demonstrate that the predictive model manages to adjust well to different contention scenarios.

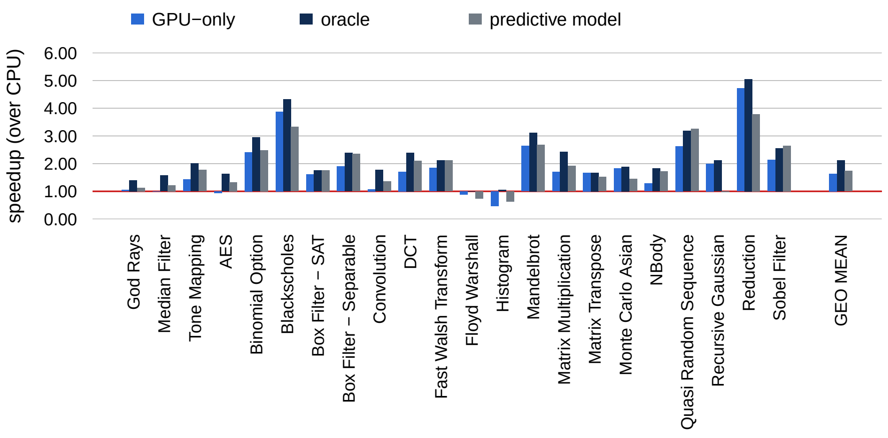
The overall accuracy of the model is 47.8%, i. e. in almost half of all cases the model picks the correct partitioning out of the 11 possibilities. A wrong prediction does not necessarily lead to bad performance though. If the prediction is only slightly off, the performance is often close to the optimum (figure 1).

Case studies To gain a better understanding of the results figure 6 shows performance results for 3 of the 10 contention scenarios, namely `none`, `monte_carlo` and `sort`, representing no, medium and heavy contention respectively.

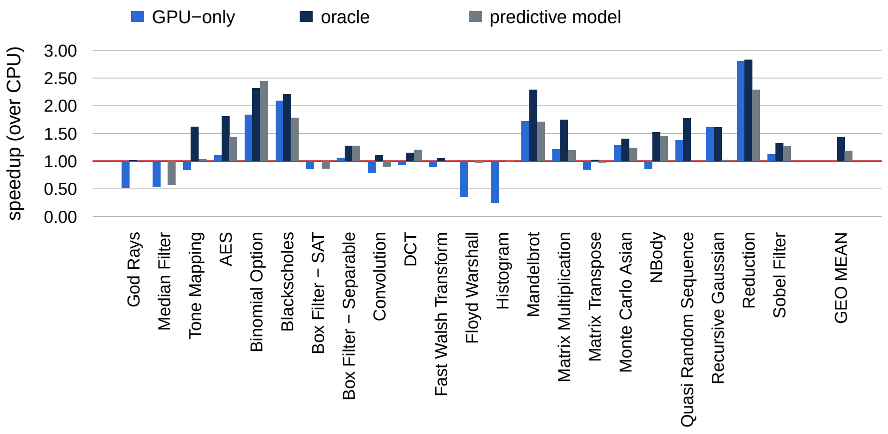
No contention. The average performance of the oracle in no contention is a speed-up of 2.11. The oracle results show, however, that partitioning the work between the CPU and GPU improves performance for all but one benchmark (`Floyd Warshall`) over using only a single device. When there is no contention on the GPU, the GPU-only approach leads to good performance because the GPU generally outperforms the CPU. On average it achieves a speed-up of 1.64. The predictive model outperforms the GPU-only scheme with an average speed-up of 1.74.

Medium contention. When introducing medium contention on the GPU (figure 6b) performance of the GPU-only method suffers significantly for some benchmarks, e. g. `God Rays` and `Median Filter`, while staying strong for others, e. g. `Binomial Option` or `Reduction`. On average, it slows the program down to 0.98 over the CPU execution. By contrast, the predictive modeling approach leads to an average 1.20 speed-up across the benchmarks, which is not far from the 1.42 speed-up of the oracle performance.

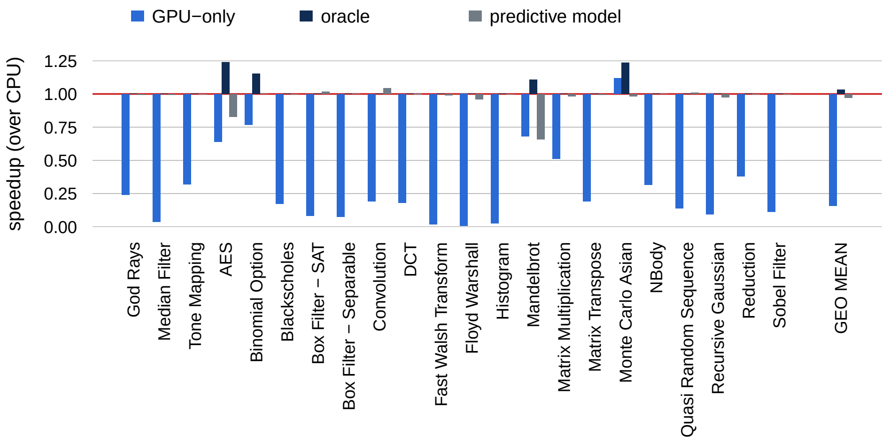
Heavy contention. In a heavy contention, the waiting time on the GPU increases significantly. Therefore, in generally, we should avoid to map the program on the GPU. The oracle approach is only able to achieve speedups on 3 out of 22 benchmarks, with an averaged speedup of 1.03 over the CPU-only scheme. It is not supervised that in such a scenario the GPU-only approach performs poorly (figure 6c). For only one benchmark, `Monte Carlo Asian`, an improvement over CPU execution can be observed with some benchmarks have 100x slowdown. On average, the GPU-only method leads to a slow-down of 6 times. Unlike the massive slow-down performance delivered by the GPU-only scheme, the predictive model leads to only minor slow-down over the CPU execution, i.e. 3%.



(a) none (no contention)



(b) `monte_carlo` (medium contention)



(c) `sort` (heavy contention)

Fig. 6: Speed-up over CPU-only execution in three different contention scenarios: no contention (a), medium contention with `monte_carlo` as the workload program (b) and heavy contention with `sort` as the workload program (c).

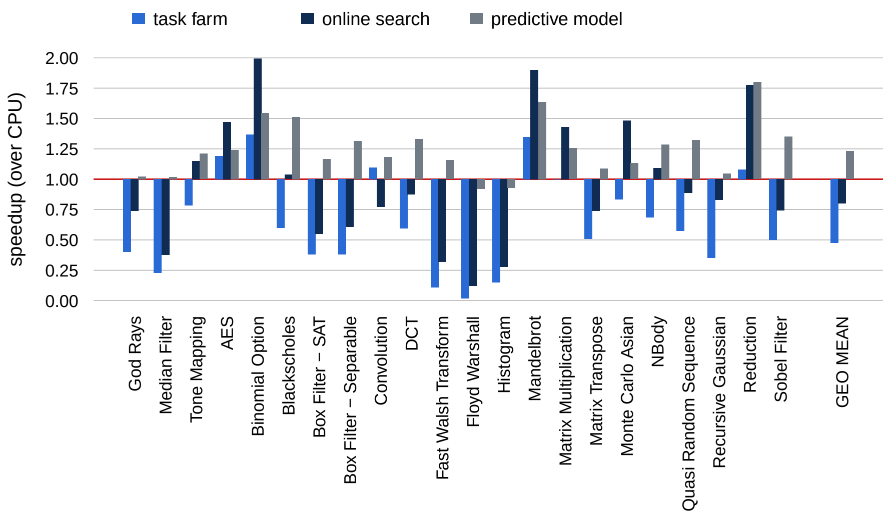


Fig. 7: Speed-up over CPU-only execution averaged across all ten contention scenarios. The task farm, online search and predictive modeling approaches achieve average speed-ups of 0.45, 0.73 and 1.24 respectively.

Summary. The predictive model is able to adapt to contention on the GPU and outperforms single-device approaches. When there is no contention the GPU typically outperforms the CPU but this is reversed when contention is introduced. In all scenarios the predictive model is able to at least match the performance of the fastest single-device strategy. The next section investigates the performance compared to two dynamic mapping approaches.

6.2 Comparison to State-of-the-arts

Figure 5 shows the performance of the two dynamic approaches, task farm and online search, as well as that of the predictive model. The performance of each benchmark is shown, averaged across all ten contention scenarios. The numbers are normalized to (parallel) CPU-only execution.

It can be seen immediately that both dynamic approaches fail to achieve good performance in the presence of GPU contention. With a few exceptions, e. g. `Binomial Option` or `Mandelbrot`, both approaches are not able to outperform the CPU-only approach. Especially the task farm mapper leads to slowdowns in most cases. For all but one benchmark, namely `Convolution`, the online search approach beats the task farm mapper. In only 5 of the 22 benchmarks does either of the dynamic approaches outperform the predictive modeling approach.

On average, the task farm method leads to 2.2x slow-down and the online search approach leads to 1.36x slow-down. In other words, both dynamic schemes fail to achieve speedups when there is contention on the GPU. The predictive model, on the other hand, achieves a speed-up of 1.24, demonstrating that using the GPU in the right way can be very beneficial.

The benchmarks where the dynamic approaches, especially the online search method, do well are the ones where GPU execution performs strongly even in heavy contention scenarios, e. g. `Binomial Option` or `Monte Carlo Asian`, as can be seen in figure 6c. Conversely, benchmarks where a GPU-only approach performs poorly in heavy contention, e. g. `Fast Walsh Transform` or `Floyd Warshall`, also show huge slow-downs on the dynamic approaches.

7 Related Work

Programming Frameworks for GPUs As GPUs become ubiquitous for computing, many programming models [8, 9, 11] have been proposed for GPU programming. These approaches provide APIs to develop GPU applications. All these approaches implicitly assume the GPU gives the best performance.
Program Mapping for GPUs A number of approaches have been proposed to partitioning a GPU program kernels across the CPU and the GPU [14, 6]. However, those approaches assume the program runs in isolation and do not consider the GPU contention.
Dynamic Task Scheduling Previous work investigates hardware and operating system based approaches to schedule tasks on CPUs. For examples, symbiotic job scheduling tries to find the best mix of jobs [17, 5] on SMT processors; an Parcae is a dynamic tuning framework [15] for CPU execution. Ravi et al. [16] develop a dynamic approach to make task for heterogeneous systems. Their approach searches for the best partition at runtime. However, the searching can lead to significant runtime overhead. Our approach, by contrast, avoids this overhead by directly predicting the portioning setting.
Predictive Modeling In addition to optimizing sequential programs [4, 13], recent studies have shown that predictive modeling is effective in optimizing parallel programs [20, 21, 18] or scheduling multiple programs on the CPU [19, 7]. However, none of the previous research addresses the problem of task mapping in the presence of workload contention on a heterogeneous platform with different computing devices.

8 Conclusion

This paper has investigated the impact of contention for GPU resources on mapping OpenCL programs to CPU-GPU systems. Standard mapping techniques fail to adapt to this type of contention because, unlike on the CPU, kernels have exclusive access to the GPU and cannot be preempted. It is possible, however, to adapt mapping decisions to GPU contention by explicitly taking it into account. We have proposed a machine learning-based approach that uses information of the contention as well as program characteristics to decide how to partition an OpenCL kernel across the CPU and GPU. Across a set of 22 benchmarks and 10 different contention scenarios this method achieved a speed-up of 1.23 over

CPU-only execution. This corresponds to 86% of the performance of an oracle approach. Two dynamic mappers, task farm and online search, only achieve speed-ups of 0.48 and 0.80 respectively, thus actually slowing down the execution time compared to the CPU-only method.

References

1. AMD. Accelerated parallel processing (APP) sdk, 2013.
2. Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.
3. Bernhard E. Boser, Isabelle Guyon, and Vladimir Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the 5th Annual ACM Conference on Computational Learning Theory*, pages 144–152, 1992.
4. Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. In *LCTES '99*, pages 1–9, 1999.
5. Stijn Eyerman and Lieven Eeckhout. Probabilistic job symbiosis modeling for smt processor scheduling. In *ASPLOS '10*, pages 91–102.
6. Dominik Grewe and Michael F.P. O'Boyle. A static task partitioning approach for heterogeneous systems using opencl. In *CC*, 2011.
7. Dominik Grewe, Zheng Wang, and Michael F. P. O'Boyle. A workload-aware mapping approach for data-parallel programs. In *HiPEAC '11*, 2011.
8. Tianyi David Han and Tarek S. Abdelrahman. hiCUDA: a high-level directive-based language for GPU programming. In *GPGPU '09*.
9. Amir Hormati, Mehrzad Samadi, Mark Woh, Trevor Mudge, and Scott Mahlke. Sponge: portable stream programming on graphics engines. In *ASPLOS '11*.
10. Intel. Intel SDK for OpenCL applications 2013 — intel developer zone, 2013.
11. Jungwon Kim, Honggyu Kim, Joo Hwan Lee, and Jaejin Lee. Achieving a single compute device image in opencl for multiple GPUs. In *PPoPP '11*.
12. LLVM. Clang: a C language family frontend for LLVM. `http://clang.llvm.org/`.
13. Shun Long and Michael FP O'Boyle. Adaptive java optimisation using instance-based learning. In *ICS '04*.
14. Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *MICRO 42*, 2009.
15. Arun Raman, Ayal Zaks, Jae W. Lee, and David I. August. Parcae: a system for flexible parallel execution. In *PLDI '12*, pages 133–144.
16. Vignesh T. Ravi, Wenjing Ma, David Chiu, and Gagan Agrawal. Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations. In *SC*, pages 137–146, 2010.
17. Allan Snavely and Dean M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *ASPLOS-IX*, pages 234–244, 2000.
18. Zheng Wang and Michael F. P. O'Boyle. Using machine learning to partition streaming programs. *ACM Trans. Archit. Code Optim.*, 10(3), 2013.
19. Zheng Wang, Michael F. P. O'Boyle, and Murali Krishna Emani. Smart, adaptive mapping of parallelism in the presence of external workload. In *CGO '13*, 2013.
20. Zheng Wang and Michael F.P. O'Boyle. Mapping parallelism to multi-cores: a machine learning based approach. In *PPoPP '09*, 2008.
21. Zheng Wang and Michael F.P. O'Boyle. Partitioning streaming parallelism for multi-cores: a machine learning based approach. In *PACT '10*, 2010.