Merge or Separate? Multi-job Scheduling for OpenCL Kernels on CPU/GPU Platforms

Merge or Separate? Multi-job Scheduling for OpenCL Kernels on CPU/GPU Platforms

Yuan Wen
the University of Edinburgh
ywen@inf.ed.ac.uk

Michael F.P. O'Boyle
the University of Edinburgh
mob@inf.ed.ac.uk

ABSTRACT

Computer systems are increasingly heterogeneous with nodes consisting of CPUs and GPU accelerators. As such systems become mainstream, they move away from specialized high-performance single application platforms to a more general setting with multiple, concurrent, application jobs. Determining how jobs should be dynamically best scheduled to heterogeneous devices is non-trivial. In certain cases, performance is maximized if jobs are allocated to a single device, in others, sharing is preferable. In this paper, we present a runtime framework which schedules multi-user OpenCL tasks to their most suitable device in a CPU/GPU system. We use a machine learning-based predictive model at runtime to detect whether to merge OpenCL kernels or schedule them separately to the most appropriate devices without the need for ahead-of-time profiling. We evaluate out approach over a wide range of workloads, on two separate platforms. We consistently show significant performance and turn-around time improvement over the state-of-the-art across programs, workload, and platforms.

CCS CONCEPTS

•**Software and its engineering** →**Process management;**
•**Computing methodologies** →*Artificial intelligence;*

KEYWORDS

CPU-GPU runtime system, GPU kernel scheduling, concurrent kernel, machine learning

1 INTRODUCTION

Incorporating GPUs into multi-core parallel systems is increasingly popular. They provide the potential for high performance computing with relatively low power consumption. Users typically write part of their applications as a kernel, using CUDA or OpenCL, which is then executed on a GPU.

GPUs are normally used as dedicated accelerators for a single application. There is no overall operating system resource management, no hardware support for time sharing and very limited support for space sharing. This lack of support is a problem as GPUs become incorporated into mainstream parallel systems and used by multiple concurrent user applications.

GPGPU-10,, Austin, TX, USA
 978-1-4503-4915-4/17/02...$15.00
DOI: http://dx.doi.org/10.1145/3038228.3038235

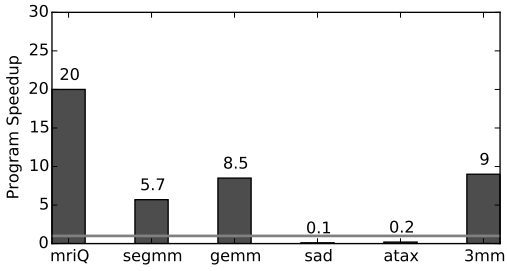
While GPUs are excellent at accelerating certain jobs, they are poorly suited to others. The research challenges are to determine (i) when to share the GPU among jobs and (ii) when to schedule jobs to the multi-core CPU. Given that this trade-off will be based on programs and the underlying platform, we want an approach that is portable and low overhead. Furthermore, as we focus on a general purpose dynamic multi-user setting, we need an approach that does not require profiling or prior knowledge of the user program.

This problem of space sharing GPU to increase utilization has been recognized by other researchers. However, such approaches are inappropriate for multi-user scheduling. Elastic Kernels [21] (EK) is the best-known work. Here, they scale the kernel code to fit multiple kernels on the same GPU. The performance improvement is poor in practice unless the best co-execute kernels are known in advance. In [14], Energy-Efficient Concurrent Kernel (EECK), they determine the best combination of kernels by profiling every candidate application ahead of time and then selecting the two kernels that are likely to improve energy and throughput. While this approach may work in single-user cases where the same combinations of kernels are executed repeatedly, it cannot be used in a dynamic multi-user setting with unknown jobs where prior profiling is not feasible.

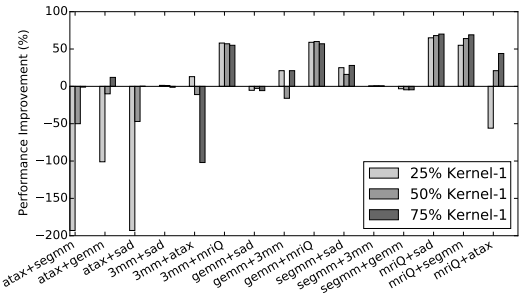
Both EK and EECK focus on sharing the GPU but do not consider the host CPU as a potential scheduling target. As all GPU systems have a host multi-core CPU, this is a wasted opportunity. In [2, 29] they determine whether to schedule OpenCL kernels to a CPU or GPU. They show performance improvement over partitioning the job between CPU and GPU but do not consider scheduling multiple jobs concurrently to the GPU in order to exploit hardware resources.

This paper develops a new scheduling approach for multiple OpenCL applications on CPU/GPU heterogeneous systems. It first determines which user jobs should be scheduled to the CPU and which to the GPU. It then determines, if appropriate, which kernels should be merged and executed concurrently on the GPU to improve performance. The merging of kernels is performed by a JIT compiler, while scheduling is performed by a thin runtime layer. Unlike previous approaches, this is totally transparent to the user and requires no profiling. To achieve this, our approach relies on offline predictive modeling. We show that merging kernels based on simple characteristics such as as memory intensity or branch divergence does not work. Instead, we build a one-off, statistical model "at the factory", based on offline experiments to determine the best merging and scheduling of kernels. It uses both *static* program features and *dynamic* runtime parameters such as data and workgroup size. This model is then cheaply deployed at runtime to predict the best scheduling decision.

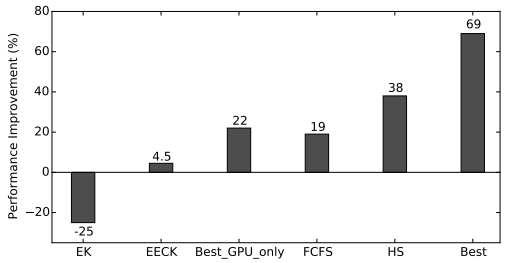
It is evaluated on a large number of workloads ranging from 2 to 64 jobs in size randomly selected from 20 benchmarks selected from the Parboil and Polybench benchmark



(a) Single kernel speedup on GPU over on CPU.



(b) Performance improvement by concurrent kernel execution.



(c) Performance improvement by different multi-task execution approaches.

Figure 1: Multi-Kernel execution on CPU+GPU platform (AMD). Each kernel runs on its default input provided by the benchmark suite. The platform hardware configuration is shown in Table 6.

suites. Our framework was evaluated on an NVIDIA and AMD platform where in each case we significantly improve performance over state-of-the-art techniques by between 11% and 36% on average over the best competitive technique.

2 MOTIVATION

This section shows that sharing a GPU and effectively using the CPU can improve performance for multi-program OpenCL workloads. We first examine six benchmarks, from the Polybench and Parboil benchmark suites and show that they have widely varying GPU performance. We then show that concurrent execution can improve performance but existing approaches perform poorly. We then show that combining concurrent GPU execution with CPU/GPU scheduling can deliver significant performance gains.

Single kernel GPU performance. Figure 1a shows the speedup of individual kernels on an AMD GPU. Kernels such as `mriQ`, `gemm` and `3mm` `sgemm`, perform well, while others e.g. `sad` and `atax`, slowdown because of data transfer overhead. Knowing which device to use clearly impacts application performance.

Concurrent kernel execution. The concurrent execution of kernels on a GPU can improve hardware utilization and performance. However, such improvement depends on the kernels and how much of the GPU's resources are allocated to each one. The proportion of GPU resource allocated to each kernel is called the *mixing ratio*. To illustrate the impact of kernels and mixing proportion on performance, we ran all pairwise combination and mixing proportions of the programs shown in Figure 1a. The kernel pairs are created by a source-to-source transformation described in Section 5.

Figure 1b shows the results. Here the x-axis denotes the pair of programs run together, and the y-axis describes the performance improvements over running the kernels non-concurrently. Each pair has three performance bars. The first corresponds to 25% of the GPU allocated to the first program with the rest assigned to the second program. The second and the third bar represent 50% and 75% respectively. As can be seen, performance of many concurrent kernels is poor and only improves in certain cases.

In Figure 1b, no matter how `atax` and `sgemm` are combined, their performance is always worse than running these two kernels sequentially. Kernel pairs, such as `3mm+mriQ`, `sgemm+mriQ`, `sgemm+sad`, and `mriQ+sgemm`, experience a higher throughput when running them concurrently. Other kernel pairs, such as `3mm+atax`, `gemm+3mm`, and `mriQ+atax`, can achieve good performance, but the correct allocation of resources is critical. Otherwise, they will slow down.

Existing Concurrent Approaches. In Figure 1c the first three bars correspond to existing concurrent scheduling approaches. The first bar corresponds to Elastic-Kernel [21] (EK) which merges kernels pairwise without regard to suitability. The second, Energy-Efficiency Concurrent Kernel [14] (EECK) uses prior profiling to determine what to run concurrently, ahead of time. To make the use of profiling realistic, we use profiling information from a smaller data set to guide merging.

The third bar to refers Best_GPU_only which represents the best performance available by choosing the right kernels to execute together by trying all combinations and represents an upper-bound on performance Figure 1c shows the speedup of each approach relative to just running the kernels sequentially on the GPU. EK suffers a 25% slowdown while there is a 4.5% improvement when using EECK.

For EK, the slowdown is caused by poor kernel pair and mixing ratio selection. EECK is sensitive to the accuracy of the profiling. More accurate profiling would certainly help but its excessive cost cannot be justified in a multi-tasking environment. The third bar in Figure 1c shows that there is a potential 22% performance improvement available when merging kernels smartly.

Separate vs Concurrent Kernel Scheduler. GPU based systems have host multi-cores which are also scheduling targets [29]. The final three bars in Figure 1c show the performance achieved when using different scheduling policies that also use the CPU relative to the performance achieved when just executing kernels sequentially on the GPU. FCFS is a simple

first-come-first-served scheduler that gives 19% improvement. The Heterogeneous-Scheduler (HS) [29] can achieve a significant improvement, 38%. If, however, we were able to correctly determine which kernels to merge and which kernels to schedule to the CPU by exhaustively trying all possbilities, we can achieve a 69% improvement as shown by last bar labelled Best.

In summary, both concurrent kernel execution and scheduling to CPU/GPU can boost performance. Furthermore, there is significant room for improvement over existing schemes. In this paper, we propose a runtime framework together with a Just-In-Time compiler to create and schedule concurrent kernels to CPU/GPU heterogeneous platforms without the use of profiling.

3 CONCURRENT KERNEL ANALYSIS

Correctly sharing resources between kernels is difficult. This section explores the impact of program characteristics on performance and shows that no single characteristic is useful in determining what kernels to execute concurrently. Similar fact has been observed by other work, such as [24]. One widely accepted view is that complementary computation and memory bandwidth intensity (or memory intensity) have a significant impact on resource sharing. To examine this, Table 1 shows the percentage of time the GPU computing and memory units are active for the kernels shown in Figure 1a. Values in bold highlight high intensity. Most kernels are memory bound except for `sgemm` and `mri-q`.

Table 1: Compute vs. Memory Intensity.

Kernels	Compute Intensity	Memory Intensity	Benchmark Suite
atax_kernel1	0.315	**88.86**	Polybench
sad_calc_8	1.7	**90.27**	Parboil
sgemm	7.58	32.43	Parboil
3mm_kernel2	20.83	**87.41**	Polybench
gemm_kernel	23.3	**88.9**	Polybench
mri-q	**35.16**	0.16	Parboil

Combining compute and memory intense kernels can improve performance in some cases. Co-running `mri-q` with `sad` improves performance. However, when co-executing `mri-q` with a different memory intense kernel, such as `atax`, the mixing ratio is critical. If `mri-q` has 25% of the resources, the system experiences a 50% slowdown.

Conversely, a memory intense kernel such as `gemm`, co-running with another memory intense kernel `3mm` can benefit from co-execution as long as one of the applications receives the majority of resources. Equal resource partition surprisingly leads to slowdown.

3.1 Concurrent execution based on kernel characteristics

We now examine the impact of kernel characteristics on concurrent performance across a larger number of programs to see if a pattern emerges. The programs are detailed in Table 7, section 7.

Impact by branches. Branches within a kernel have a significant impact on a single kernel's performance [26, 27].Figure 2 shows their impact on concurrent execution performance. Here, the kernels are sorted according to the percentage of branches executed. On the x-axis, the kernels located on the left have fewer branches, as do the kernels located on the lower part in y-axis. Red dots represent slowdown, blue dots speedup. While we would expect co-execution of kernels with few branches to be advantageous (lower left hand corner) this is not the case. In fact, there is no clear pattern.

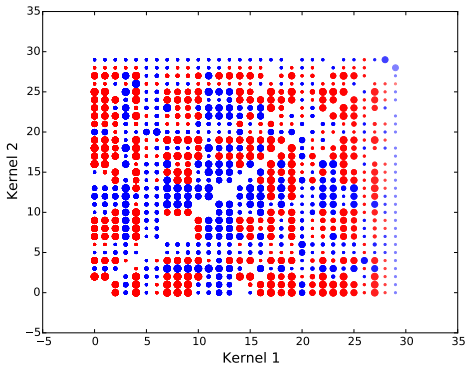


Figure 2: Performance distribution when co-executing kernels according to branch density. Kernels are ordered increasingly on branch density. Performance of the kernel pair is designated by the size of the dot. Red means a slowdown, blue means speedup.

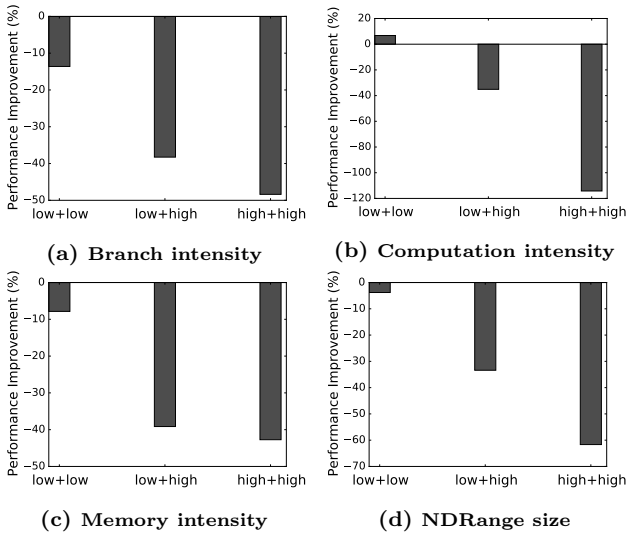
If we separate the kernels into low and high branch categories, Figure 3a shows the result of kernel co-execution in three categories: low branch with low branch, low branch with high branch and high branch with high branch averaged across mixing ratio. While it is true that increasing the branches in the workload decreases performance, from 12% to 47% to slowdown, executing kernels which have low branch rates does not improve performance either, confirming Figure 2.

Impact by compute intensity. It is often claimed that co-running a high and low compute intensity kernel will allow effective utilization of resources. After dividing kernels into low and high compute intensity categories Figure 3b shows their co-execution performance. The middle bar shows that co-running low with high compute intensity actually gives a 35% slowdown. Surprisingly if both have low compute intensity, then a small gain of 7% is possible. Co-running of high intensity workloads, however is clearly not a good idea.

Impact by memory accessing intensity. It may be the case that sharing low computation intensity kernels can be extended to memory intense kernels. Figure 3c presents the result of kernel co-running into the previous three categories. It shows that memory intensities alone as a criteria for concurrent execution is a poor policy.

Impact by NDRange. Finally, we consider the impact of the number of threads on performance. Figure 3d shows the results of kernel co-running in all three categories. While co-executing small kernels gives best performance, it is still a 4% slowdown

In summary, using program characteristics such as divergence, memory and compute intensity or number of parallel threads alone is not a useful guide to selecting which kernels to merge. In the next three sections we develop a smarter scheme that combines these and other characteristics to best exploit concurrent execution and also allows exploitation of the multi-core CPU host.



(a) Branch intensity (b) Computation intensity

(c) Memory intensity (d) NDRange size

Figure 3: Performance of co-executing tasks based on program characteristics. Low+Low denotes a Low intensity task co-executing with another Low intensity task. Low+High denotes Low intensity co-executing with High intensity etc.

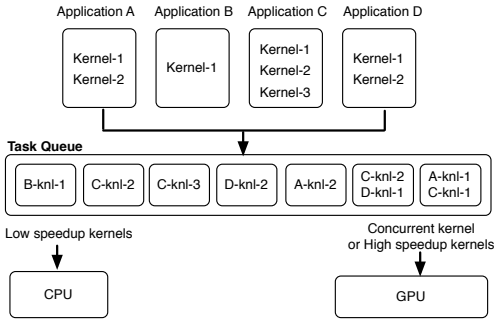


Figure 4: Kernels are scheduled to either CPU or GPU from different ends of the queue in parallel.

4 OVERALL SCHEME

Figure 4 shows a specific example of our overall scheme. It requires no modification or recompilation of user application code or system software. Initially, the framework queries all devices and sets up a global context. This context is shared with the application threads and saves considerable overhead.

Users compile and execute their applications. Each application, as usual, builds its kernel and data buffer and registers with the runtime. Our runtime then extracts both *static* and *dynamic* features from the kernel which are input into a predictive model. This model determines (i) whether to schedule it on CPU or GPU and (ii) whether to merge it with another currently available kernel. Based on these predictions, the kernels are then inserted into a double-ended sorted task queue with CPU friendly kernels at one end and (merged) GPU friendly kernels at the other. Tasks are dynamically dispatched from one end of the queue to the CPU, the other end to the GPU.

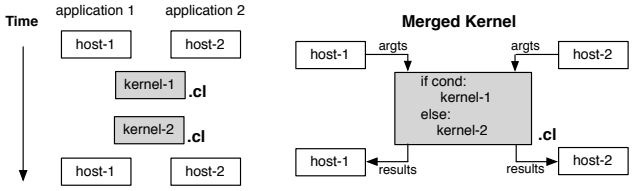


Figure 5: Merged kernel construction.

5 JIT COMPILER

Kernels are compiled at runtime, so as to be portable across OpenCL devices. As each application registers its kernel, our JIT compiler is called, extracting kernel static features and creating concurrent kernels based on a predictive model where appropriate.

5.1 Concurrent kernel construction

Concurrent kernel execution is a well-studied topic. Prior work uses either separate streams or command-queues for CUDA and OpenCL platforms respectively. However, fair sharing actually depends on driver implementation and, as shown in [19], is not guaranteed. Rather than relying on device drivers, we use a compiler based approach that gives fine-grain control over the merging of kernels and hence their co-execution.

Our approach creates a concurrent kernel by the source-code merging of two OpenCL kernels as shown in Figure 5. In this paper, we only consider the merging of two kernels, as more than two usually degrades performance, [19, 20]. We use an improved version of inter-thread-block-fusion [28] to merge kernels. We first replace the workgroup ID comparison with a modulo operation, in which the divisor represents the number of computing units and the dividend illustrates how many of these computing units would be used by one of the constituent kernels. Hence, two kernels could share the GPU concurrently with a workgroup index transformation. Secondly, we adapt the thread decoupling technique [21] to decouple physical hardware allocation from logical workgroup identities for both the merged kernel and its constituent kernels. The kernels' ND-Ranges became variable and can be adjusted.

Using a modulo operation to control kernel mixing does not introduce branch divergence, as the workgroup is a basic scheduling unit to the SIMD processor. However, it presents a problem to thread index acquisition, as such operation splits the workgroup index space into two areas, each area contains the workgroups with indices of either quotient or remainder of the division and the thread index space suffers the same problem. To tackle this problem, we transform discrete physical workgroup indices to continuous logical indices.

6 PREDICTIVE MODEL

At the heart of our scheme is a machine learning based predictive models which classify newly arriving OpenCL kernels. The first model separates kernels into two groups CPU or GPU based on estimated device affinity. The second model determines whether or not to merge two GPU kernels and their merging ratio as shown in Figure 6. These models determine where, in the double-ended sorted task queue, (merged) kernels are inserted.

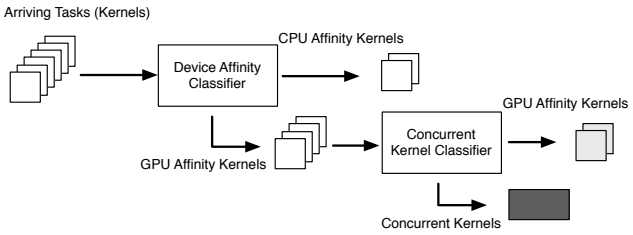


Figure 6: Tasks (OpenCL kernels) are classified by two predictive models, one is used to estimate device affinity and the other is used to predict co-executing kernels.

Table 2: Program Features

Static features	#instructions	#load/stores
	#blocks	#br/condbranches
	#mathFunctions	#vector operations
	#int operations	#float operations
	#control instruction	#logic operations
	#barriers	#atomic operations
Runtime features	local_work_size	global_work_size
	input size	output size

6.1 Model Training

Machine learning is a well known technique which uses training samples to build a model that classifies kernels into the appropriate category. Samples consisting of kernel features $\underline{f}$ and the best scheduling classification $\underline{t}$. The algorithm builds a model m that, given a new feature $\underline{f}$ predicts the target classification t i.e. $m : \underline{f} \mapsto t$. This learning is a one-off activity performed "at the factory". Once the model is learnt it can be used online to determine kernel scheduling. There are many types of models available; we use decision trees as they are easier to explain. Crucial to model accuracy are the types of features used.

6.2 Features

Determining the right features is critical. We considered all features that can be extracted from kernel functions and their runtime parameters, and only those shown Table 2 had any impact on performance. Features improtance is analysed in Section 9.1. All such features were collected from the kernel source file using our LLVM based JIT compiler and the runtime environment.

Feature Transformation. Using all these features directly leads to model overfitting [1, 3, 11]. so we normalise and transform them. Table 3 shows the seven features used by the first CPU vs GPU model. Features `F1` through `F4` represent the percentage of each each instructions in the source code Table 4 shows the features used in the concurrent kernel model. Here the features are a weighted sum of the features from the two kernels to be mixed. depending on mixing ratio. Different weighted sums of kernel features are input to this model. The first ratio that yields a positive classification *i.e.* merge the kernel determines the mixing ratio selected.

In all our experiment, we use leave-N-out cross-validation [5] to ensure we do not use the same training data when evaluating. This means whenever we have 2 programs to consider for merging, neither have been used in training.

6.3 Predicting device affinity

To illustrate how the model works, Figure 7 shows the decision tree for NVIDIA. We just show the first few layers of the tree due to space. The leaf nodes refer to the classification, gray for GPU, white for CPU. Table 5 shows the features of the programs shown in Figure 1a. As an example, consider `atax`. Its F1 value, which corresponds to compute intensity, is 0.6<=0.6795 which means that the left hand branch is taken. Its F2 value, representing memory intensity, is 0.2<=0.2341 so again the left hand branch is taken. F7, representing number of work threads, is 0.00055>0.0021 so the right branch is taken this time . Finally, after checking its compute intensity again, it is classified as being best scheduled on the CPU. A similar tree will be learnt for the AMD platform. Discussion of the trees learnt for merging are provided in the analysis section.

Table 3: Combined feature for separate kernel model

	Features	Description
F1	comptInst / (allInst)	compute instruction ratio
F2	memInst / (allInst)	memory instruction ratio
F3	br / (allInst)	branch ratio
F4	barriers / (allInst)	barriers ratio
F5	dataSize / (MaxMem)	datasize
F6	globalWorkSize	number of global threads
F7	localWorkSize	number of local threads

Table 4: Combined feature for concurrent kernel model

	Features	Description
F1	comptInst / (allInst)	compute instruction ratio
F2	memInst / (allInst)	memory instruction ratio
F3	br / (allInst)	branch ratio
F4	barriers / (allInst)	barriers ratio
F5	dataSize / (MaxMem)	datasize
F6	dataSizeRatio1	first kernel datasize ratio
F7	dataSizeRatio2	second kernel datasize ratio
F8	globalWorkSize1 / (globalWorkSize)	first kernel global threads ratio
F9	globalWorkSize2 / (globalWorkSize)	second kernel global threads ratio
F10	localWorkSize1 / (localWorkSize)	first kernel local threads ratio
F11	localWorkSize2 / (localWorkSize)	second kernel local threads ratio

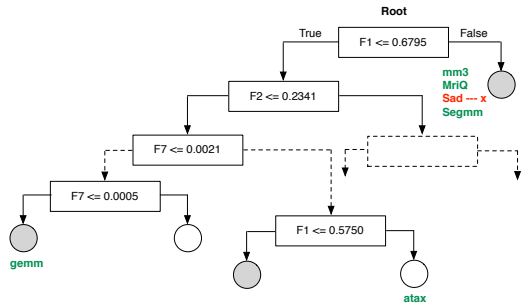


Figure 7: Single kernel classification on Intel+Nvidia. The nodes stand for tests on separate features values. Leaf nodes refer to the classification, gray for GPU, white for CPU. All input tasks are correctly classified except `Sad` which has CPU affinity but misclassified to a GPU group.

Table 5: Combined feature for separate kernel model

	F1	F2	F3	F6	F7
atax_kernel1	0.6	0.2	0.2	0.25	0.0.0055
sad_scalc_8	0.8039	0.1275	0.0686	0.125	0.0.0029
sgemm	0.7143	0.1904	0.0952	0.25	6.4e-05
3mm_kernel2	0.6923	0.1538	0.1538	0.25	0.0003
gemm_kernel	0.6551	0.1724	0.1724	0.25	0.0003
mri-q	0.7188	0.25	0.0312	0.0625	0.0002

7 EXPERIMENT SETUP

7.1 Platform and Benchmarks

We evaluate on two CPU+GPU systems. The details are shown in Table 6. Both have an Intel Core i7 4-core CPU and 16GB main memory. Both systems host OpenSUSE

Table 6: Hardware platform

	Intel CPU	NVIDIA GPU	AMD GPU
Model	Core i7 4770K	GeForce GTX 780	Radeon HD7970
Architecture	Haswell-DT	Kepler GK110	Tahiti XT
Core Clock	3.4 GHz	1215 MHz	1000 MHz
Core Count	4 (8 w/ HT)	2304	2048
Memory	16 GB	3 GB	3GB
Memory Bandwidth	21GB	288 GB	264 GB

Table 7: Benchmarks

Suite	Benchmarks				
Parboil	BFS	Cutcp	Sgemm	Spmv	Sad
Polybench	ATAX	BICG	CORRELATION	GESUMMV	SYR2K
	SYRK	2DCONV	3DCONV	GEMM	GRAMSCHMIDT
	2MM	3MM	COVAR	FDTD-2D	MVT

12.3 Linux. We use LLVM 3.4 for JIT compilation and benchmarks are compiled using GCC 4.7.2 with -O3 option.

We restrict our attention to benchmarks with 1D and 2D NDranges from two mainstream OpenCL benchmark suites: Parboil and the Polybench benchmark suite giving 20 programs in all. The selected benchmark applications are shown in Table 7.

7.2 Scheduling Approaches

We compare our approach to a number of approaches on two platforms.

Elastic Kernels (EK) This approach runs two kernels concurrently and merges the corresponding host programs. It does not have a model to determine what to merge and does not use the CPU as a scheduling target.

Energy-Efficient Concurrent Kernels (EECK) This approach is similar to EK but requires profiling of the applications beforehand to determine what to merge. We use smaller data sizes as profile training sizes as profile input. It does not use the CPU as a scheduling target

Separate or Concurrent on GPU(SoC_GPU) Our approach to concurrent execution of kernels without using the CPU as a scheduling target

First-come-first-served(FCFS) This is a simple scheme that schedules jobs to either the CPU or GPU based on availability. It does not run kernels concurrently on GPU.

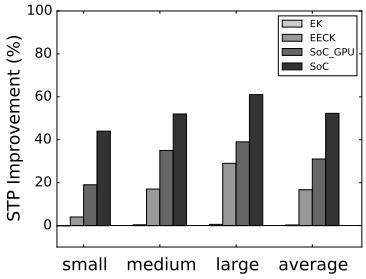
Heterogeneous Scheduling (HS) This uses a model to schedules jobs to either the CPU or GPU based on availability. It does not run kernels concurrently

Separate or Concurrent on GPU (SoC) Our approach to both concurrent execution of kernels and using the CPU as a scheduling target

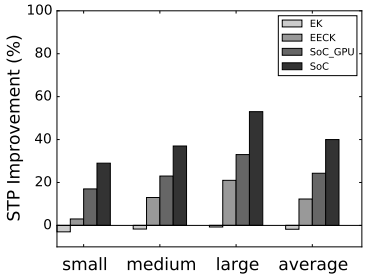
To make a fair comparison, we ignore the introduced overhead *i.e.* the cost of profiling and recompilation for `EK` and `EECK`. Such overheads always outweighed the benefits, making them hard to work in practice.

7.3 Performance Evaluation

We evaluated our schemes with 500 different task configurations. We selected 10 different task queue sizes containing between 2 and 64 kernels. For each task queue size, we randomly selected 50 different programs, to give 500 configurations. As behaviour is dynamic, we evaluated each configuration 30 times and report the median performance. This results in 15000 experiments per policy. We then summarize our results into three categories according to the number of tasks: the small group has less than 16 tasks, the medium group includes less than 32 tasks, and the large group contains less than 64 tasks. Performance is presented throughout as speedup relative to executing just on a GPU i.e. the throughput speedup STP metric and normalized turn around time - the ANTT metric [29].



(a) SoC vs. concurrent kernel on NVIDIA



(b) SoC vs. concurrent kernel on AMD

Figure 8: Summary of performance improvement when run all tasks on GPU only. Task sizes: small (2-16 kernels), medium (16-32 kernels), large(32-64 kernels).

8 RESULTS

In this section, we evaluate our approach against alternative approaches for throughput and turnaround time,

8.1 Speedup over State-of-the-art Methods

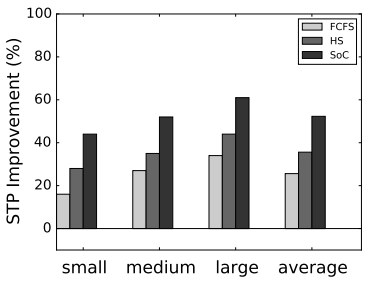
Figure 8 and 9 presents the throughput or speedup performance for each approach relative to sequential execution on the GPU. The results are presented for small, medium and large job size.

When evaluating those methods that use just target the GPU (see Figure 8), EECK and our approach (SoC_GPU) improve performance by up to 38% but EK experiences a small slowdown on both platforms. Both EECK and our approach increase performance as the number of task increases, as more pairing options are possible.

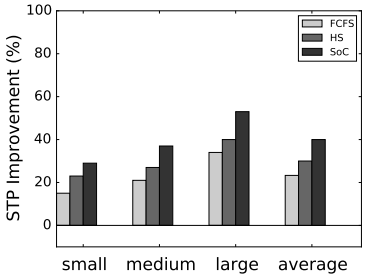
Using the multi-core CPU as another scheduling device improves throughput further. Our approach (SoC) consistently outperforms FCFS and HS, as shown in Figure 9. All three approaches increase in speedup as the number of tasks increase. Since our strategy also optimizes GPU utilization our overall performance is better and increases with the number of tasks.

8.2 ANTT Improvement over State-of-the-art Alternatives

Turnaround time describes the amount of time has been taken from a task's arrival to its computation finish. We use Average Normalized Turnaround Time (ANTT) to measure the amortized waiting time for all tasks. As ANTT is a



(a) SoC vs. separate scheduling on NVIDIA



(b) SoC vs. separate scheduling on AMD

Figure 9: Performance improvement when use both CPU and GPU.

Table 8: Improvement on Average Turnaround Time for both Nvidia and AMD Platforms

Benchmarks		Small	Medium	Large	Average
Nvidia	EK	-15	-5	7	-4.3
	EECK	5.5	15	17	12.5
	SoC_GPU	11	12	23	15.3
	FCFS	16	20	27	21
	HS	25	36	53	38
	SoC	51	65	77	64
AMD	EK	-12	-1	1	-3.3
	EECK	3	10	16	9.7
	SoC_GPU	9	21	25	18.3
	FCFS	15	31	33	26
	HS	30	52	59	47
	SoC	44	61	68	57

smaller is better metric, we transform the results into ANTT improvement; analogous to speedup.

Table 8 shows the results on NVIDIA and AMD platforms. When considering the GPU as the only device, our technique outperforms the others. Once we take multi-core CPU into consideration, our approach is significantly better than its counterparts. EK has a poor ANTT, which is worse than the baseline due to the performance degradation caused by incorrect pairing of kernels.

8.3 Overall Performance Summary

On average, on the GPU alone, our (SoC_GPU) method has 11.2% better speedup than the best alternative (EECK) on the NVIDIA platform and 11% better on the AMD platform. ANTT is 3% and 9% better than EECK on NVIDIA and AMD platform separately. When we include the CPU into consideration, throughput speedup improves to 36% and 28% respectively. ANTT is also significantly improved, widening to 52% and 47%. This shows that a smart scheduling policy has significant impact.

When comparing to the other approaches that use both CPU and GPU as their target scheduling device, our method (SoC) provides an enhanced performance which is 17% and 10% better than the best alternative (HS) on NVIDIA and AMD platform respectively. The corresponding ANTT improvement between our method and HS is 26% and 10%.

8.4 Detailed Performance in Pairwise Execution

While this paper is concerned with performance of multi-task scheduling, it is useful to explore performance in more detail by focusing on what happens when we restrict our attention to just 2 kernels to be scheduled. Figure 10 presents the performance distribution of an individual kernel when paired against every other kernel within the best mixing ratio. The horizontal line represents the mean execution time of our approach for that kernel, while the box and whiskers represent the 75% and max range of speedups.

As can be seen, most kernels are improved by our approach with a positive pairwise speedup. In some cases there are slowdowns caused by misprediction of our model. For some, `correlation_corr_kernel`, `covariance_covar_kernel`, `syr2k_kernel`, the best configuration is running it sequentially on GPU with the other candidate kernel, hence there is no speedup available. This is due to their very long execution time compared to others. Similarly, kernels with very short execution time such as `sgemm_NT` and `spmv_jds_native` have the same behaviour. One exception is `gesummv_kernel`, which has a long execution time and is best scheduled to the CPU . It is always scheduled to the CPU and dominates overall throughput. On average, our approach improves performance by 21.8% for all pairwise executions over the baseline of running sequentially on a GPU.

To dig deeper, we compare our performance against the Best scheduler. The results are shown in Figure 11. Due to space we cannot show all kernel pairs and just paired them up alphabetically. Results are grouped according to scheduling decisions. Bars in the white area represent kernels where we scheduled to the CPU/GPU while those in the gray areas means that the kernel pairs are running concurrently on the same GPU.

We have the same behavior as the Best scheduler except in 3 cases. For `atax_kernel1+bfs_kernel`, both of the kernel have CPU affinity and incur data transfer costs between CPU and GPU. Best schedules them both to the CPU. However, our model classifies `bfs_kernel` as a GPU task leading to degraded performance. For `Convolution3D_kernel+correlation_corr_kernel` and `correlation_mean_kernel+covariance_covar_kernel`, both candidate kernels are classified as having GPU affinity. As we concurrently dequeue from both ends of the task queue, one of them will be scheduled to CPU while the right choice would be to stall CPU dequeuing and execute both on the GPU and prevent CPU dequeuing. As the number of tasks increases, this is a less of an issue as there will be more CPU suitable jobs available.

9 ANALYSIS

The decision tree models trained for the Nvidia and AMD platform to predict concurrent kernel merging are shown in Figure 12 and 13. In both cases, only the first few levels are shown due to space. The percentage labels refer to the number of kernels merged down a particulate branch. The two trees have separate shapes that reflect the difference between the two models trained for Nvidia and AMD platform. In both cases F3 (branches) and F5 (datasize) are used to classify early on though with different thresholds. The number of

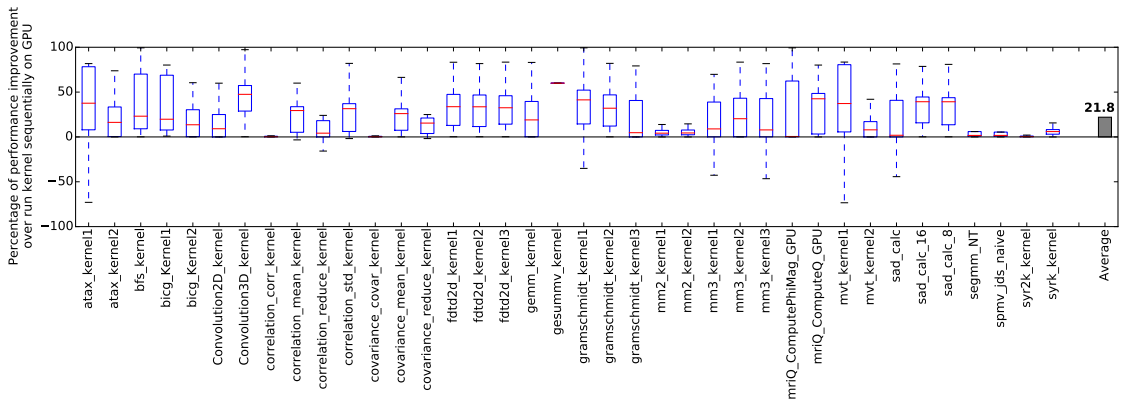


Figure 10: Performance distribution of a given kernel co-running with all other kernels in pairs on AMD GPU only with the best mixing ratio. Baseline: run both kernels on the GPU sequentially

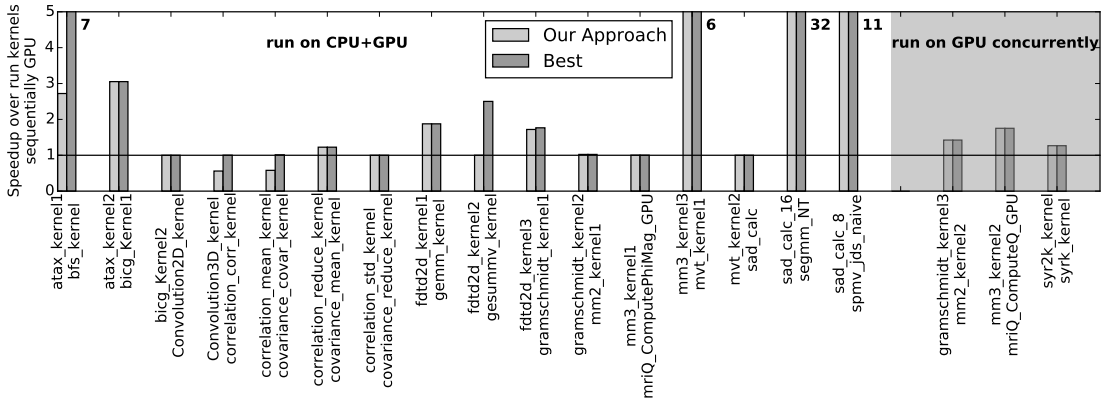


Figure 11: Performance of two kernel's co-execution by our approach vs the best achievable. Kernels are paired alphabetically. The results are shown in form of speedup over the baseline in which two kernels run on GPU sequentially.

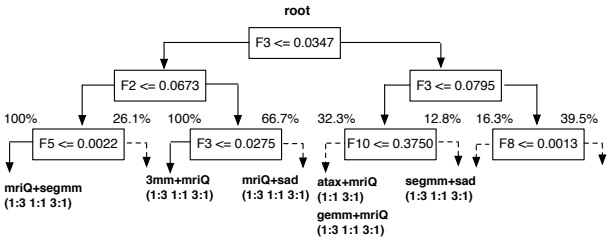


Figure 12: Decision tree to estimate kernel merging on Intel+Nvidia platform. Lables at each branch account for the percentage of how many merged kernels out of the whole can be found in that branch.

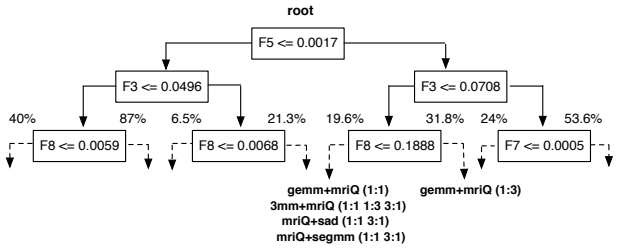


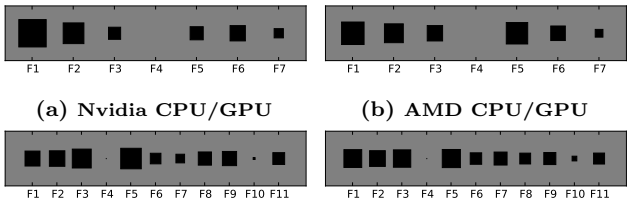
Figure 13: Decision tree to estimate kernel merging on Intel+AMD platform.

threads, F8 and F10 are also important in later classification.

Estimation Accuracy. In our system, we trained our classifier using a leave-one-out-cross-validation [5, 29], on 38 distinct kernels and 2031 concurrent kernels with different mix ratios. For the CPU/GPU classifier, we have an accuracy of 88% on NVIDIA and 90.3% on AMD platform. For concurrent kernel classifier, its accuracy is 81% and 85% on those platforms.

9.1 Feature Importance

While the decision trees precisely describe the model used, they do not give much insight into how important certain program features are. Figure 14 shows detailed Hinton diagrams for each platform and model. The size of each box corresponds to importance and measures how correlated a feature is with performance. If we first consider scheduling to either the CPU or GPU, then 2 features, F1 and F2 dominate on the Nvidia platform. These correspond to compute and memory intensity as shown in Table 3. On the AMD platfrom, F5, datasize is just as important. This appeals to intuition as programs with high computational intensity are likely to benefit from GPU scheduling.



(a) Nvidia CPU/GPU **(b) AMD CPU/GPU**

(c) Nvidia concurrent GPU **(d) AMD concurrent GPU**

Figure 14: Feature Importance

When it comes to concurrent kernel execution, F5 or data size (Table 4) is the key factor especially on NVIDIA. This suggests that whether 2 kerenls can fit in memory is highly signicant. Feature F4, the percentage of barriers has the least impact. This is because barriers are rarely found in the benchmarks. Section 3 showed that no single characteristic was useful enough for scheduling. In fact, all characteristics are important and need to be combined carefully.

9.2 Limit Study

To examine how the classifier accuracy impacts the performance, we performed a limit study. We evaluated the system throughput in three different cases. In the first case, we use our our learnt models. In the second case, we replace the models with decisions that are 100% accurate. They use prior execution runs to always correctly determines whether a kernel runs faster on a GPU or a CPU and whether or not it runs faster concurrently. This is of course, not possible in practise.

In the third case, for each sequence of tasks, we run 10,000 different scheduling orders to test the potential performance upper bound. Because of combinatorial complexity, we can not evaluate all possible kernel combinations and scheduling orders, and this approach gives only an approximation of performance achievable. It represents the Best schedule found. Figure 15b and 15a show our results on two different platforms. In general, the more accurate a classifier is the higher throughput we can expect. For a 100% accurate classifier, we can get a 5% performance improvement on NVIDIA platform and 7% on AMD platform over our classifier on average.

Classifying accurately is not enough. There may be a case where 2 kernels have large concurrent speedup but have insignificant execution time while 2 others have a small concurrent improvement but are long running and dominate throughput time. This is shown by the performance of the best scheduler which improves STP further. On NVIDIA platform, its performance is 20% better than our classifier and on AMD platform this performance improvement is 19%. This shows that while our approach has significant improvement over existing schemes, there is still further room for improvement

10 RELATED WORK

Running multiple kernels on the same GPU in parallel is a popular way to improve system performance [8, 30] and power consumption [17]. There are static and dynamic approaches to run multiple kernels concurrently. In [10], kernels are combined at compile time to execute concurrently on the GPU, which has no hardware support for concurrency. Thread blocks of different kernels run interleaved on the GPU to improve hardware utilization. Static kernel merging approach is also used to optimize data movement between kernels that have data dependencies [7]. Several kernel fusing methods are discussed and compared in [28] to reduce energy consumption and improve GPU power efficiency.

In dynamic approaches, kernels are issued via independent stream, or command queue, blocks by blocks. In [30], kernels are partitioned slices containing a fixed number of thread blocks. Kernels are co-executed on by issuing their slices simultaneously via separate CUDA stream. Combining kernel slice with DVFS, Jiao et al [14]. present an approach to improve GPU energy efficiency through concurrent kernel execution.

Though concurrent kernel execution is good for throughput by improving hardware utilization, indiscriminate running multiple kernels leads to a sub-optimal GPU performance. Jog et al. [15] propose an application-aware memory system to improve the fairness and efficiency of concurrent kernels. Chimera [23] utilizes streaming machine flushing technique to preempt GPU and keeps multiple tasks sharing GPU in a balanced way.

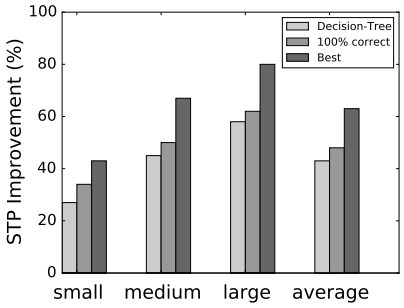
Task scheduling on CPU/GPU heterogeneous [25] often requires the estimated execution time for each task on each device. An accurate execution time is hard to get unless run the task ahead of time on the target device, which limits the range of its uses in practice. Wen et al. [29] present a machine learning based approach to schedule OpenCL kernels on CPU/GPU platform according to the kernels predicted speedup. The system throughput is improved by finding tasks their best-fitting devices, however, it cannot increase GPU hardware resource utilization.

Some prior work uses kernel partitioning to balance the workload between CPU and GPU [9, 16, 22]. In this method, a kernel is usually partitioned into two sub-kernels, one running on CPU and the other running on GPU. However, this method is only good at speedup a single kernel's performance. When there are multiple kernels running concurrently, performance delivered by kernel partitioning is usually not as good as separate kernel scheduling [29].

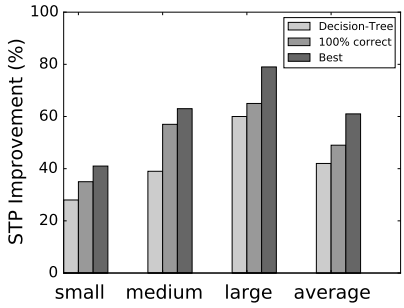
PALMOS [18] provides a runtime infrastructure to eliminate high setup overhead for OpenCL applications. DyManD [13] proposes a run-time library and a set of compiler passes to optimize the data management and communication on CPU-GPU system. StarPU [4, 6, 12] provides a runtime framework that enables mapping single application on heterogeneous with various type of devices.

11 CONCLUSION

In this paper, we have proposed a runtime system and a JIT compiler to schedule multiple OpenCL kernels on a CPU-GPU heterogeneous platform without profiling. We have trained two predictive models from OpenCL kernel's static and runtime features to determine whether the kernels would be merged and launched together, or dispatched to the best-fit device separately. To evaluate the performance, we compare our approach with a wide range of state-of-the-art methods on two different heterogeneous platforms and outperfromed tem in throughput and turn around time. Though our approach has a significant improvement over existing methods, there is still further room for improvement. Precisely predicting task execution time would improve performance further and is future work.



(a) Nvidia



(b) AMD

Figure 15: Limit study. On average, a 100% accurate classifer can improve the performance by another 5%. If we have more information about the task, such as kernel execution time, the performance can be improved by 20%.

REFERENCES

[1] W. Ahmad and A. Narayanan. Artificial immune system: An effective way to reduce model overfitting. In *Computational Collective Intelligence - 7th International Conference, ICCCI 2015, Madrid, Spain, September 21-23, 2015. Proceedings, Part I*, pages 316–327, 2015.

[2] A. M. Aji, A. J. Peña, P. Balaji, and W. Feng. Automatic command queue scheduling for task-parallel workloads in opencl. In *2015 IEEE International Conference on Cluster Computing, CLUSTER 2015, Chicago, IL, USA, September 8-11, 2015*, pages 42–51, 2015.

[3] N. I. Al-Najjar and M. M. Pai. Coarse decision making and overfitting. *J. Economic Theory*, 150:467–486, 2014.

[4] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*.

[5] G. C. Cawley and N. L. C. Talbot. Efficient approximate leave-one-out cross-validation forkernel logistic regression. *Machine Learning*, 71(2):243–264, 2008.

[6] L. Courtès. C language extensions for hybrid CPU/GPU programming with starpu. *CoRR*, abs/1304.0878, 2013.

[7] J. Filipovic, M. Madzin, J. Fousek, and L. Matyska. Optimizing CUDA code by kernel fusion: application on BLAS. *The Journal of Supercomputing*.

[8] C. Gregg, J. Dorn, K. M. Hazelwood, and K. Skadron. Fine-grained resource sharing for concurrent GPGPU kernels. In *4th USENIX Workshop on Hot Topics in Parallelism, HotPar'12, Berkeley, CA, USA, June 7-8, 2012*, 2012.

[9] D. Grewe, Z. Wang, and M. F. P. O'Boyle. Opencl task partitioning in the presence of GPU contention. In *Languages and Compilers for Parallel Computing - 26th International Workshop, LCPC 2013, San Jose, CA, USA, September 25-27, 2013. Revised Selected Papers*.

[10] M. Guevara, C. Gregg, and K. H. K. Skadron. Enabling task parallelism in the cuda scheduler. In *Workshop on Programming Models for Emerging Architectures (PMEA), 2009*.

[11] D. M. Hawkins. The problem of overfitting. *Journal of Chemical Information and Modeling*, 44(1):1–12, 2004.

[12] A. Hugo, A. Guermouche, P. Wacrenier, and R. Namyst. Composing multiple starpu applications over heterogeneous machines: A supervised approach. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum, Cambridge, MA, USA, May 20-24, 2013*, pages 1050–1059, 2013.

[13] T. B. Jablin, J. A. Jablin, P. Prabhu, F. Liu, and D. I. August. Dynamically managed data for CPU-GPU architectures. In *10th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '12, San Jose, CA, USA - March 31 - April 04, 2012*.

[14] Q. Jiao, M. Lu, H. P. Huynh, and T. Mitra. Improving GPGPU energy-efficiency through concurrent kernel execution and DVFS. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2015, San Francisco, CA, USA, February 07 - 11, 2015*, 2015.

[15] A. Jog, E. Bolotin, Z. Guz, M. Parker, S. W. Keckler, M. T. Kandemir, and C. R. Das. Application-aware memory system for fair and efficient execution of concurrent GPGPU applications. In *Seventh Workshop on General Purpose Processing Using GPUs, GPGPU-7, Salt Lake City, UT, USA, March 1, 2014*.

[16] R. Kaleem, R. Barik, T. Shpeisman, B. T. Lewis, C. Hu, and K. Pingali. Adaptive heterogeneous scheduling for integrated gpus. In *International Conference on Parallel Architectures and Compilation, PACT '14, Edmonton, AB, Canada, August 24-27, 2014*.

[17] T. Li, V. K. Narayana, and T. A. El-Ghazawi. A power-aware symbiotic scheduling algorithm for concurrent GPU kernels. In *21st IEEE International Conference on Parallel and Distributed Systems, ICPADS 2015, Melbourne, Australia, December 14-17, 2015*, pages 562–569, 2015.

[18] C. Margiolas and M. F. P. O'Boyle. PALMOS: A transparent, multi-tasking acceleration layer for parallel heterogeneous systems. In *Proceedings of the 29th ACM on International Conference on Supercomputing, ICS'15, Newport Beach/Irvine, CA, USA, June 08 - 11, 2015*.

[19] C. Margiolas and M. F. P. O'Boyle. Portable and transparent software managed scheduling on accelerators for fair resource sharing. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization, CGO 2016, Barcelona, Spain, March 12-18, 2016*, pages 82–93, 2016.

[20] B. Neelima, G. R. M. Reddy, and P. S. Raghavendra. Communication and computation optimization of concurrent kernels using kernel coalesce on a GPU. *Concurrency and Computation: Practice and Experience*, 27(1), 2015.

[21] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan. Improving GPGPU concurrency with elastic kernels. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, 2013.

[22] P. Pandit and R. Govindarajan. Fluidic kernels: Cooperative execution of opencl programs on multiple heterogeneous devices. In *12th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '14, Orlando, FL, USA, February 15-19, 2014*.

[23] J. J. K. Park, Y. Park, and S. A. Mahlke. Chimera: Collaborative preemption for multitasking on a shared GPU. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*.

[24] A. Pattnaik, X. Tang, A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, and C. R. Das. Scheduling techniques for GPU architectures with processing-in-memory capabilities. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation, PACT 2016, Haifa, Israel, September 11-15, 2016*, pages 31–44, 2016.

[25] V. T. Ravi, M. Becchi, G. Agrawal, and S. T. Chakradhar. Valuepack: value-based scheduling framework for CPU-GPU clusters. In *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12, Salt Lake City, UT, USA - November 11 - 15, 2012*.

[26] S. Sarkar and S. Mitra. A profile guided approach to optimize branch divergence while transforming applications for gpus. In *Proceedings of the 8th India Software Engineering Conference, ISEC 2015, Bangalore, India, February 18-20, 2015*, pages 176–185, 2015.

[27] J. Sartori and R. Kumar. Branch and data herding: Reducing control and memory divergence for error-tolerant GPU applications. *IEEE Trans. Multimedia*, 15(2):279–290, 2013.

[28] G. Wang, Y. Lin, and W. Yi. Kernel fusion: An effective method for better power efficiency on multithreaded GPU. In *2010 IEEE/ACM Int'l Conference on Green Computing and Communications, GreenCom 2010, & Int'l Conference on Cyber, Physical and Social Computing, CPSCom 2010, Hangzhou, China, December 18-20, 2010*.

[29] Y. Wen, Z. Wang, and M. F. P. O'Boyle. Smart multi-task scheduling for opencl programs on CPU/GPU heterogeneous platforms. In *21st International Conference on High Performance Computing, HiPC 2014, Goa, India, December 17-20, 2014*.

[30] J. Zhong and B. He. Kernelet: High-throughput GPU kernel executions with dynamic slicing and scheduling. *IEEE Trans. Parallel Distrib. Syst.*