MaxPair: Enhance OpenCL Concurrent Kernel Execution by Weighted Maximum Matching

Yuan Wen
School of Computer Science and Statistics
Trinity College Dublin
Dublin, Ireland
weny@tcd.ie

Michael F.P. O'Boyle
School of Informatics
University of Edinburgh
Edinburgh, United Kingdom
mob@inf.ed.ac.uk

Christian Fensch
MACS - Computer Science
Heriot-Watt University
Edinburgh, United Kingdom
c.fensch@hw.ac.uk

Abstract

Executing multiple OpenCL kernels on the same GPU concurrently is a promising method for improving hardware utilisation and system performance. Schemes of scheduling impact the resulting performance significantly by selecting different kernels to run together on the same GPU. Existing approaches use either execution time or relative speedup of kernels as a guide to group and map them to the device. However, these simple methods work on the cost of providing suboptimal performance.

In this paper, we propose a graph-based algorithm to schedule co-run kernel in pairs to optimise the system performance. Target workloads are represented by a graph, in which vertices stand for distinct kernels while edges between two vertices represent the corresponding two kernels co-execution can deliver a better performance than run them one after another. Edges are weighted to provide information of performance gain from co-execution. Our algorithm works in the way of finding out the maximum weighted matching of the graph. By maximising the accumulated weights, our algorithm improves performance significantly comparing to other approaches.

CCS Concepts • **Software and its engineering** → **Software performance**; **Compilers**; • **Theory of computation** → *Scheduling algorithms*;

Keywords Concurrent Kernels, Scheduling, GPGPU

ACM Reference Format:
Yuan Wen, Michael F.P. O'Boyle, and Christian Fensch. 2018. MaxPair: Enhance OpenCL Concurrent Kernel Execution by Weighted Maximum Matching. In *GPGPU-11: General Purpose GPUs, February 24–28, 2018, Vienna, Austria.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3180270.3180272

1 Introduction

Modern Graphics Processing Units (GPUs) are widely used in most of the mainstream systems because of their high performance and low energy consumption. The increasing computing demands require that the GPU moves from an exclusive accelerator serving one task at a time to a device that can be shared by multiple applications. With mainstream GPU vendors keep upgrading the fabrication process, more hardware resources such as physical cores, registers, bandwidth, shared memory, and so on, have been integrated into the same GPU device. Using these resources properly is not only desirable but also necessary, as utilisation in many cases is critical to performance.

GPGPU-11, February 24–28, 2018, Vienna, Austria

Improving GPU utilisation has been concerned by both industry and academia. As pointed out by state-of-the-art research, programs usually exhaust one kind of resource while leaving others underutilised [1, 20]. Since the intrinsic unbalanced requirement by programs causes wastage within the device, the solution of upgrading resource usage is to increase the diversity and make programs share the same GPU properly.

Concurrent execution of multiple tasks on the same GPU is the most widely accepted method to improve performance and device utilisation. Such approach in many territories is also referred as concurrent kernels because the program running on GPU is a kernel, in both Nvidia and OpenCL terminology. Boosted performance has been reported by running kernels concurrently in many prior works [13, 20, 27]. However, to the best of our knowledge, all of those works either naïvely mix compute and memory intensive workload together or use greedy algorithms which are in favor of co-run kernels that have the most promising performance and then launch them to the same GPU at the same time.

Schedulers using greedy algorithms are easy to write and simple to explain. Instead of examining all possibilities to acquire the best, greedy algorithms frequently assume that maximizing every step's profit is the best approximation for solving problems, particularly for those with high complexities. Though every step leads to an optimum solution, the overall outcome delivered by a greedy method is usually not the best. State-of-the-art schedulers are designed greedily due to simplicity.

In most cases, the guide for scheduling decisions is relative speedup (RS). Here, the relative speedup stands for the quotient of the aggregate time of programs sequential execution divided by the execution time if running them all together at the same time instead. The RS is either calculated from execution time provided by programmers or directly estimated from target workloads statically. Though RS-based greedy scheduling works well in practice, it experiences the same limitation all greedy algorithm has, which is not able to find the globally optimal solution because it does not consider all options.

In this paper, we propose a graph-based scheduling method to maximize system throughput via launching concurrent tasks to the same GPU. As the percentage of performance improvement decreases with the number of concurrent works, in this paper, we focus on selecting and allocating workload in pairs [19]. We first model the problem by using a graph, in which the nodes represent individual tasks and the edges link nodes for which co-execution of the corresponding tasks can result in enhanced performance from space sharing the same GPU. The weights on edges indicate the gain we can expect. Then, we schedule tasks in pairs from the generated weighted graph by a max matching algorithm.

We make the following contributions in this paper:

- We identify the inhibitors of GPGPU concurrent task execution comes from the greedy scheduling algorithm. Failing at considering other options makes such methods suboptimal. To overcome this shortage, we propose a graph-based scheduling solution.
- We model the concurrent execution problem as a graph. Once relationships among distinct tasks are described in the graph, various optimization can take place, such as max matching which is used in graph theory to find an independent edge set without repeated vertices.
- We propose weighted max matching algorithm to optimise system throughput. Different types of weights have been examined in this paper towards improving performance from separate aspects. For instance, we can maximize the number of concurrent pairs, accumulate RS, or aggregate execution time. As shown by our experiment, using execution time as a guide to weight edges constantly provides a better performance over the others.
- Our graph-based scheduling method constantly outperforms the state of the art once given detailed knowledge about the workloads.

2 Background

In General Purpose computing on GPU (GPGPU), graphics cards are usually connected to the central processing unit (CPU) via PCIe (Peripheral Component Interconnect Express). To perform computation, GPUs usually work as accelerators who receive tasks from the CPU and send back the results after calculation. As the host device, the CPU manages the whole process, including initializing the GPU, allocating data and functionalities to the GPU, activating the computation, and acquiring the results back. The GPU, on the other side, works as a slave device which performs the execution on data in parallel. Modern GPGPU programming models, such as CUDA and OpenCL, are designed to structure application upon this architectural prototype.

2.1 GPU Programming Model

Modern GPU programming standards, such as CUDA and OpenCL, view a computing system as a platform consisting the CPU and a number of GPUs. The software programmed following such standards constitute of two parts accordingly, which are the host code and the kernel.

Kernel The function that is executed on the GPU is called a kernel in both CUDA and OpenCL terminology and written in a C-like language. In practice, there are usually hundreds or thousands of independent threads performing the same kernel function, but on different data, namely, the kernel part works in the way of single data multiple threads (SIMT). Limited by the number of computing units on GPU, threads are grouped into sets, with each has a fixed number of threads. These sets are called workgroups in OpenCL (or blocks CUDA terminology). The GPU hardware scheduler allocates workgroups to available compute units to perform the computation and piles the others up behind. Threads within the same workgroup are scheduled to separate processing element and run in parallel.

Table 1. Execution time of each individual kernel

Kernel	Full Name	Time	Benchmark
corr	**correlation_std_kernel**	1.61ms	Pollybench
atax	**atax_kernel2**	2.54ms	Pollybench
fdtd	**fdtd2d_kernel1**	0.84ms	Pollybench
mriQ	**mriQ_ComputeQ_GPU**	3.23ms	Parboil
mm3	**mm3_kernel3**	2.1ms	Pollybench

Table 2. Speedup or slowdown of concurrent execution.

	corr	atax	fdtd	mriQ	mm3
corr	——	1.59x	0.37x	1.32x	0.61x
atax		——	1.19x	1.70x	1.60x
fdtd			——	0.58x	0.20x
mriQ				——	0.65x
mm3					——

Host Code The host code (HC) runs on the CPU to organise the software workflow. It consists of a series calls to the application programming interfaces (APIs) to manage devices and data. Classicly, the host code first checks and initializes the device. Then, it loads and compiles the kernel. Before a kernel can be launched, the programmer has to create input/output buffers which are used to store input data and the computation results. After buffers initialisation and data movement, the HC sets the arguments of the kernel function and launches it to the GPU. Once the GPU has finished kernel execution, the corresponding API in the HC will be informed to retrieve the results back and then carry on with the rest of the functions until the program reaches the end.

2.2 GPU Sharing

The SIMT model intrinsically requires the kernel to have a simple control flow. Divergence within the kernel can drastically decrease threads parallelism, which in turn degrades the overall performance. Though simple functionality is good for parallelism, it comes at the cost of lower resource utilisation. The maximum number of threads that can be accommodated by a GPU is determined by various hardware resource limitations, such as register file size, amount of shared/local memory, and the upper bound of workgroups and threads. In most cases, a given kernel exhausts only one kind of above resources and leaves the others underutilized. By squeezing multiple kernels onto the same GPU, programmers have a chance to balance the device requirements once separate kernels are bounded by different resources, thereby, improve the overall throughput.

3 Motivation

Concurrent kernel execution is an effective method to improve hardware utilization. However, to the best of our acknowledge, all prior works choose the use of a greedy algorithm to select and launch candidate kernels. This can easily result in suboptimal system performance.

We have selected five kernels from two different benchmark suites (PolyBench and Parboil) to illustrate why greedy algorithms are limited in their performance. Table 1 lists the kernels we used in this section, along with their sequential execution time.

In this work, we focus on two kernels concurrent execution, because the performance improvement decreases with the number

Table 3. Execution time of co-execution

	corr	atax	fdtd	mriQ	mm3
corr	——	2.6ms	6.55ms	3.69ms	8.71ms
atax		——	2.82ms	3.4ms	2.89ms
fdtd			——	6.97ms	14.85ms
mriQ				——	8.22ms
mm3					——

of co-run kernels. Table 2 and 3 show the relative speedup and the execution time of two kernels co-execution, respectively. Table 2 shows that some kernel pairs, such as `atax+fdtd` and `mriQ+corr`, experience an enhanced performance, but others, such as `fdtd+mm3` and `fdtd+corr`, suffer a slowdown.

Figure 1 shows the result of total execution time for all five kernels using different scheduling strategies.

In sequential scheduling, all five tasks are enqueued back to back. As the GPU is exclusively occupied by each kernel, unused resources cannot be utilized by another kernel. As a consequence, this strategy provides the poorest performance.

The performance of two state of the art greedy approaches is visualised by the second and third bar. Speedup Greedy (SG) picks the task pair that offers the highest speedup from the set of pending tasks (Table 2), while Time Greedy (TG) selects the pair that offers the highest improved execution time (Table 3). Once the greedy scheduler has exhausted the pool of task pairs, it executes the remaining tasks using sequential scheduling.

Finally, we performed an exhaustive search to find the best schedule. It performs 7% better than the SG and TG counterparts. Please note, although SG and TG perform identically in this example, in practice their performance differs frequently.

In order to understand why the SG and TG approach are suboptimal, we create a task co-execution graph shown in Figure 2. It presents the relationship of concurrent kernels. Each node represents an individual kernel. An edge indicates that the co-execution of these two nodes can experience performance gain via co-execution. Each edge is also labelled with the relative speedup of co-execution.

For SG-based scheduling, the algorithm selects `atax-mriQ` first, as it supplies the the highest speedup (1.70). After `atax` and `mriQ` are removed from the task set, the remaining tasks (`corr`, `fdtd`, and `mm3`) have to be executed sequentially, as there are no edges between them. The TG greedy algorithm selects the pair that shortens the execution time the most. In our example, it makes the same decision as SG because the co-run of `atax-mriQ` provides the largest reduction. As a consequence, the remaining kernels have to executed sequentially.

In our example, the best scheduling sequence is {`atax-mm3`, `corr-mriQ`, `fdtf`}. Rather than selecting the pair with the highest speedup or execution time improvement, this schedule selects the second best choice in order to acquire a second pair and overall improves performance. In this paper, we propose a graph-based algorithm to provide enhanced performance over greedy-based methods.

4 Concurrent Kernels

There are multiple ways to construct concurrent kernels. Each method has its pros and cons. Assigning kernels to separate command queues indicates the absense of dependencies between them enabling the potential for concurrent execution. The alternative is to merge the kernels before sending them to a command queue.

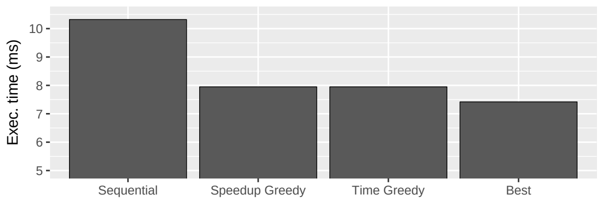


Figure 1. Execution time to run the five kernels from Table 1 using different scheduling strategies.

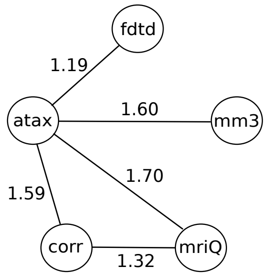


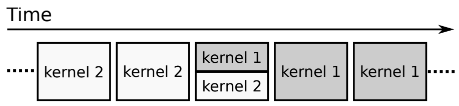
Figure 2. The task co-execution graph $G = (V, E, w)$ describes the benefits we can expect from co-execution. A node V represents a kernel; an edge E links two nodes if their co-execution performs better than sequential run; the weight w of each edge quatifies the benefit.

Using separate queues Figure 3a shows the most basic model of concurrent kernels. In this model, kernels are executed from different queues. For most of the time, the GPU is used exclusively by a given kernel. Resources are only shared between two kernels, once a kernels lacks enough active threads to fully utilize the processing elements of a GPU. At this point, some thread workgroups from the next kernel can co-run with the current one in order to fully utilise all processing elements of the GPU. This back-to-back sharing is easy to implement but provide limited benefits.

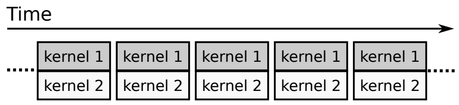
Figure 3b presents an improvement over Figure 3a. Instead of issuing all workgroups of one kernel altogether, workgroups are launched alternatively from two kernels via separate queues within a loop. By changing the number of workgroups started from each kernel, the kernel mixing ratio can be adjusted. This model is also easy to implement. However, its functionality depends on the behaviour of the driver and hardware scheduler. At the moment, there are no guarantees from either the driver or the hardware scheduler that threads from different kernels will be executed concurrently with an optimal mixing ratio to maximize utilization.

Merging kernels Figure 4 presents three approaches of kernel merging. Kernel merging is an ahead of runtime method that constructs a super kernel from two candidate kernels. Kernel merging is also referred to as kernel-fusion.

The merging can happen intra-thread, intra-thread workgroup, or inter-thread workgroup [24]. Intra-thread merging shows in Figure 4a. The super kernel contains two code from two candidates, with one glued behind the other. Intra-workgroup merging (see

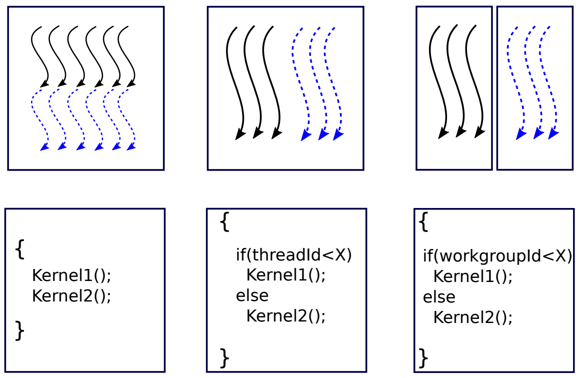


(a) Back-to-back execution: concurrent execution only happens at the transition between two kernel's execution.



(b) Scheduler alternates thread fetching from different kernels.

Figure 3. Examples of concurrent kernel execution using seperate queues.



```
{
  Kernel1();
  Kernel2();
}
```

```
{
  if(threadId<X)
    Kernel1();
  else
    Kernel2();
}
```

```
{
 if(workgroupId<X)
   Kernel1();
 else
   Kernel2();
}
```

(a) Intra-thread **(b)** Intra-workgroup **(c)** Inter-workgroup

Figure 4. Examples of kernel merging strategies [24].

Figure 4b) uses the thread index within a workgroup to decide which code is executed. Therefore, each workgroup contains a proportion of threads that belong to one kernel, and the rest belongs the other. A potetial drawback of this method is that it guarantees thread divergence. Finally, inter-workgroup merging (see Figure 4c), uses the workgroup index to select which kernel to run.

In this paper, we use the inter-workgroup method to construct concurrent kernels as it outperforms the others. However, the scheduling method proposed in this paper can be used with all above concurrent kernel implementations.

5 MaxPair Scheme

We use a graph-based algorithm to optimize the scheduling of concurrent kernels. As introduced in Section3, co-execution of kernels can be described by a graph, in which nodes represent tasks while edges represent the co-run of two tasks.

5.1 Matching in a graph

In graph theory, a matching is a set of edges in which none of the edges share a common node. The output of a matching algorithm is of a set of edges and a set of individual nodes which are not linked by an edge. By substituting the nodes with kernels, the result of the matching is a schedule, in which task are either dispatched pairs or seperately.

For any given graph, multiple matchings exists all representing possible schedules. However, we are looking for the best matching

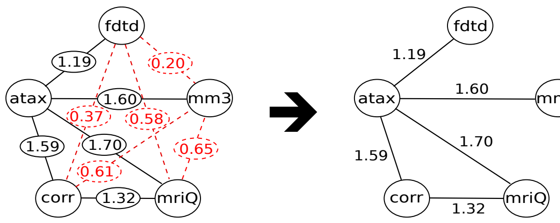


Figure 5. We simplify the graph by removing edges representing non beneficial co-execution.

representing the best schedule. Now the matching problem has become a maximization problem characterized by weights that we assign the edges. These weights can be selected in ways to represent different aspects of co-execution and are explained in the following sections. In an initial optimization, we remove edges with negative weights as those represent a performance degredation from co-execution. This process is illustrated in Figure 5, reducing the number of edges to consider by 40%.

5.2 Weights

Different aspects of co-execution are represented by different weights. We aim to create a set of edges that maximizes the total weight. In our work, we considered three weights: unified weight, speedup weight, and time weight.

Unified weight assigns an identical weight to all edges. The matching represents a schedule that contains the most kernel pairs with beneficial execution behaviour.

Speedup weight assigns the speed up to each edge expressed as percentage improvement. Expressing the speedup as a factor could result in a situation where multiple mediocre speedup pairs (e.g. 1.01x + 1.01x) are choosen over a strong pair (e.g. 1.50x). By using percentages, we avoid this problem (e.g. 1% + 1% <50%). This strategy delivers the maximum overall speed up.

Time weight assigns the time saved by concurrent execution over executing the tasks sequentially. This strategy delivers the maximum overall time saved.

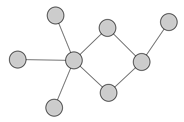
5.3 Why Schedule Task in Pairs

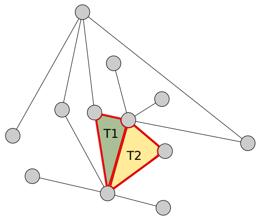
In order for tasks to benefit from co-execution they need to be connected by an edge in the co-execution graph. To execute more than two tasks concurrently, the nodes of these tasks need to form fully connected sub graphs. For example, to concurenly execute three tasks we are looking for triangles in the graph.

Figure 6 shows two examples of other co-execution graphs that we obtained as part of our experiments (for simplicity we do not label the nodes and edges). We notice that the occurance of triangles is rather limited. Only the 12-node graph contains two triangles T1 and T2. However as they share an edge, only one of these triangles could be part of a schedule. Due to this lack of of fully connected subgraphs, we decided to limit our technique to scheduling tasks in pairs or individually.

5.4 Algorithm

The graph matching problem has been studied for decades. In our work, we use maximum weight matching to schedule pairwise


(a) 8 nodes.



(b) 12 nodes.

Figure 6. Examples of larger co-execution graphs. Only the 12 nodes graph has groups of 3 kernels that could be executed concurrently.

concurrent kernels. Due to space constraints, we only present a concise description here.

The input of the maximum matching algorithm is an undirected graph, $G = (V, E, w)$, with $|V| = n$ and $|E| = m$. The nodes represent tasks, and each edge represents a beneficial co-execution of two tasks. The weights w quantify the gain. First, we introduce how Maximum Cardinality Matching works in general graphs, then we present the modifications for Maximum Weighted Matching.

The maximum cardinality matching algorithm finds a matching with larges number of edges. For any given graph, a node can either be single or matched. In Figure 7, the single and matched nodes are highlight by different colors. Matched nodes are colored in orange while the single nodes are colored in grey. An edge between two matched nodes is called matched edge while all the other edges are called unmatched edges. In the example, we have two matched edges (1-2 and 3-4) drawn with a solid red line. All unmatched edges are shown as a dashed black line.

An alternating path is a path that visits each node at most once and in which every other edge is a matched edge. In our example, we find several alternating paths: 1-2-3, 3-4-5, 1-2-3-4-8, etc. An augmenting path is an alternating path that starts and ends on a single node, for example 6-1-2-7 (highlighted in our example).

According to a theorem by Berge [6], a maximum cardinality matching can be found by iteratively searching for an augmenting path for each unmatched point. If an augmenting path found, then the matching set M is updated by a process called Symmetric Difference, which is the disjunctive union of two sets. It contains elements from either of the set but not their intersection. For instance, the Symmetric Difference of set $A = \{1, 2, 3, 4, 5\}$ and $B = \{3, 4, 5, 6, 7\}$ is $A \oplus B = \{1, 2, 6, 7\}$. If no augmenting path can be found for a given node, we can safely remove it as it is not in the maximum matching.

Theorem. *The matching* M *has maximum cardinality if and only if there is no augmenting path with respect to* M.

Circles in graphs require special attention. We employ a method developed by Edmonds [8] that replaces the circles (called a “blossom”) with a super-node. The details are beyond the scope of this concise summary. Algorithm 1 shows an overview of the final algorithm.

For a weighted graph, we assign values to nodes. By making the sum of two nodes greater and equal than that their edge weight, the problem of maximizing the edge weights has been transformed to minimizing the node values. Once the sum of node values in the augmenting path is equal to the edge weights, we extend the matching. Otherwise, we adjust the node values and keep looking for an augmenting path. More details about the algorithm are discussed in the work proposed by Berge *et al.* [6].

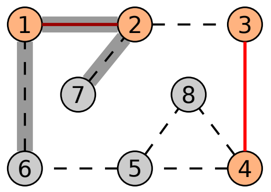


Figure 7. Augument path.

```
INPUT: Graph G, matching M on G;
OUTPUT: A maximum matching M*x on G;
Initialization: All nodes in G are unmatched nodes;
for each unmatched node in G do
    instructions;
    if the vertex is an unvisited node then
        if this vertex has been matched then
            extend augmenting path P;
        else
            found an augmenting path P;
            update M by P⊕ M;
        end
    else
        if found a blossom then
            replace the blossom by a super node;
        end
    end
end
```

Algorithm 1: The MaxPair algorithm.

6 Runtime Deployment

To test MaxPair scheduling, we develop a runtime framework that consists the scheduler and a Just-In-Time (JIT) compiler which is used to construct concurrent kernel. Figure 8 illustrates how the system works.

For all candidate kernels, the scheduler builds a task graph according to their associativities. The edge weights of the graph are assigned based on the knowledge of the candidate kernels. If we have only very limited information (e.g. only know if two kernels benefit from executing concurrently), then we allocate the same weight to all edges. In this case, the matching algorithm will find the maximum number of kernel pairs. When more knowledge about the kernels provided, such as relative speedup or precise execution time, the edge weights are updated accordingly. Therefore, the matching algorithm will find a pairing-up scheme that can maximize the accumulated speedup or time-saving.

Our JIT-compiler merges kernels utilsing a source-to-source transformation technique [27]. The fused kernel then replaces the original kernel pair and is launched to the GPU instead. It is worth noting, that our scheduling technique is independent of the mechanism used for concurrent execution.

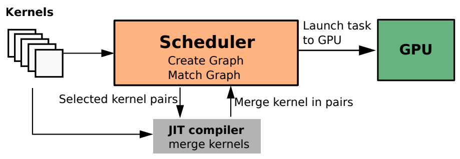


Figure 8. The runtime system consists of a graph-based scheduler and a Just-In-Time compiler that create concurrent kernels by source-to-source transformation.

Table 4. Hardware platform

	Intel CPU	NVIDIA GPU	AMD GPU
Model	Core i7 4770K	GTX 780	HD7970
Architecture	Haswell-DT	Kepler GK110	Tahiti XT
Core Clock	3.4 GHz	1.2 GHz	1.0 GHz
Core Count	4 (8 w/ HT)	2,304	2,048
Memory	16 GB	3 GB	3 GB
Bandwidth	21GB/s	288 GB/s	264 GB/s

The runtime allocates kernels sequentially to the GPU. Merged kernel first, and then separate kernels. When all merged kernels are finished, the individual kernels, which are better to run alone, are issued to the GPU one after another. As this paper focuses on maximizing concurrent kernels performance on the GPU, we didn't use the multi-core CPU as a candidate device in our experiment. However, our method can cooperate with heterogeneous scheduling approaches [27, 28] with very trivial effort.

7 Experiment Setup

7.1 Platform

We evaluate our scheduling scheme using an AMD and NVIDIA GPU. Both systems use an identical host system. The details of the hard setting are shown in Table 4. We use Nvidia driver 375.20 and AMD Catalyst driver 14.9. Programs are compiled by GCC 4.7.2 with the -O3 flag. We repeat each experiment 100x, in order to eliminate noise.

7.2 Benchmarks

We use 38 kernels from 20 applications in our experiment (see Table 5). We have selected kernels from all programs in Polybench. However, as our source-to-source compiler focusses on 1D and 2D kernels transformation, we were only able to use five applications from Parboil. Due to space constraints, we use the index in Table 5 to refer to a specific kernel.

We use our kernel fusion mechanism (see Section 6) to create concurrently executing kernels for our evaluation.

7.3 Alternative Algorithms

We use a random scheduling algorithm as our baseline. This algorithm selects a pair of tasks at random and executes them concurrently. The choosen pairs are always identical in all repeats of an experiment. We compare our scheduling methods against two existing state of the art, greedy-based algorithms: speedup greedy (SG) and time greedy (TG). These algorithms are explained in more detail in Section 3.

Table 5. List of all kernels.

Index	Kernels	Benchmark
1	mvt_kernel2	Polybench
2	bicg_Kernel1	Polybench
3	Convolution3D_kernel	Polybench
4	bicg_Kernel2	Polybench
5	covariance_reduce_kernel	Polybench
6	Convolution2D_kernel	Polybench
7	gramschmidt_kernel1	Polybench
8	gramschmidt_kernel2	Polybench
9	gramschmidt_kernel3	Polybench
10	syr2k_kernel	Polybench
11	mriQ_ComputeQ_GPU	Parboil
12	covariance_mean_kernel	Polybench
13	mriQ_ComputePhiMag_GPU	Parboil
14	correlation_corr_kernel	Polybench
15	sad_calc_8	Parboil
16	gesummv_kernel	Polybench
17	mm2_kernel2	Polybench
18	mm2_kernel1	Polybench
19	segmm_NT	Polybench
20	sad_calc_16	Parboil
21	atax_kernel1	Polybench
22	atax_kernel2	Polybench
23	bfs_kernel	Parboil
24	syrk_kernel	Polybench
25	spmv_jds_naive	Parboil
26	gemm_kernel	Polybench
27	fdtd2d_kernel2	Polybench
28	fdtd2d_kernel3	Polybench
29	fdtd2d_kernel1	Polybench
30	covariance_covar_kernel	Polybench
31	mvt_kernel1	Polybench
32	correlation_std_kernel	Polybench
33	sad_calc	Parboil
34	mm3_kernel2	Polybench
35	mm3_kernel3	Polybench
36	mm3_kernel1	Polybench
37	correlation_reduce_kernel	Polybench
38	correlation_mean_kernel	Polybench

8 Results

This section presents the result of MaxPair comparing to state of the art greedy-based scheduling methods. In addition, we show that the runtime overhead introduced by MaxPair is trivial.

8.1 Performance of Concurrent Execution

Selecting the optimal set of pairwise kernels is essential for maximizing performance. For a task scheduler, forming the task set and determining the issue order needs to be guided. Speedup and time saving of co-run kernels are two primary guides that are widely advocated by most of the task schedulers [7, 13]. Figures 9 and 10 present the results of kernel co-execution in these forms, respectively.

Figure 9 shows the speedup of concurrent execution. Distinct kernels (identified by their index according to Table 5) are listed along both axises. The resulting heatmap is based on the speedup of their co-execution over executing them sequentially. A darker grey means a higher speedup. A white tile means there is no performance gain from the co-execution. Figure 10 uses the same method,

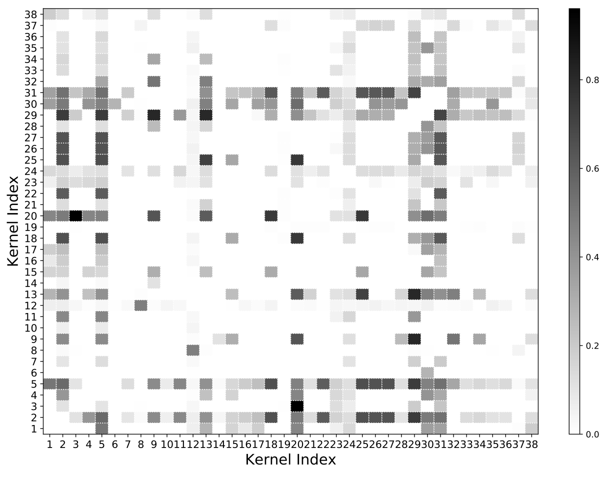


Figure 9. Relative speedup of concurrent kernels. Kernels are aligned along with x- and y-axis. The index represents each corresponding kernel that is listed in Table 5. The colour scale bar shows the range of speedup which is a form the darker, the better.

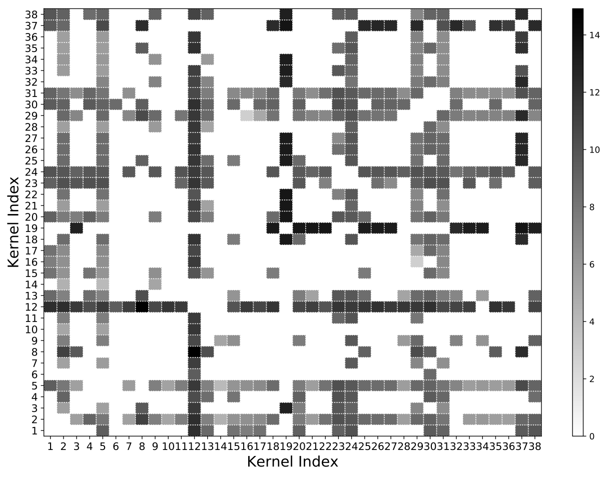


Figure 10. Time-saving from concurrent kernels. Kernels are aligned along with x- and y-axis. The index represents each corresponding kernel that is listed in Table 5. The colour scale bar shows the base 10 logarithms of saved time in microseconds. It is the form of the darker, the better.

but presents the results of time-saving from concurrent kernel execution.

As we can see, Figures 9 and 10 emphasise different aspects of performance. Take kernel 12 for example: it has a good associativity with other kernels. In most of the cases, running kernel 12 with another one enhances performance compared to running them sequentially. However as we can see from Figures 9 and 10, for some cases the relative speedup improvement is small while the time saving is large. This is because the execution time of the 12th kernel is much longer than the others. Hence, a small proportion speedup delivers a significant time-saving. On the other hand, concurrent execution of kernel 2 and 29 has a high speedup but low time improvement.

8.2 Performance Evaluation

Figures 11 and 12 present the performance improvement by our graph-based scheduling method comparing to other alternatives on AMD and Nvidia platforms, respectively. We consider three levels of information that we possess about concurrent workloads. At the first level, we only have binary knowledge if two kernels benefit from co-execution. At the second level, we posses more detailed information in form of relative speedup. At the third level, we have full knowledge of the individual and concurrent execution times.

To fairly evaluate our method, the workloads are randomly selected for each platform. However due to space constraints, we only present the details for the AMD platform in Table 6.

8.2.1 Scheduling According to Task Associativity

Figures 11(a) and 12(a) shows the results of MaxPair over the random algorithm. Each workload configuration is prefixed by the platform, followed by the number of kernels in the workload and a unique identifier.

With only binary information available, MaxPair aims to find the maximum number of kernel pairs. The lack of a weight prohibits the use of a greedy based algorithm. MaxPair improves the performance by 0.46% over the random scheduling. If only consider the average performance, MaxPair works equally poor as the random. However, the result by MaxPair is more stable. As we can see from Figures 11(a) and 12(a), the number of task scheduling that experience an improvements outweigh the number of slowdowns.

8.2.2 Scheduling According to Relative Speedup

Relative speedup provides more information about how large the performance gain we can expect from two kernels co-execution. Figures 11(b) and 12(b) present the result of MaxPair compared to a greedy-based algorithm that uses the same guide. The results show the performance improvements of MaxPair over Speedup-Greedy. MaxPair outperforms or matches Speedup-Greedy in all configurations, except #33 on AMD. Out of the whole 36 experiments, 26 and 24 show a performance improvement of over 10% on the AMD and Nvidia platform, respectively.

On average, MaxPair is 12% and 10% better than Speedup-Greedy scheduling on those platforms, respectively.

8.2.3 Scheduling According to Precise Execution Time

Precise execution time provides more detailed information than relative speedup. Figures 11(c) and 12(c) present the results by MaxPair against the Time-Greedy algorithm.Figures 11(c) and 12(c) show that MaxPair continually outperforms the Time-Greedy method. Out of the whole 36 experiments, 14 and 12 deliver an improvement over 10% on AMD and Nvidia, respectively. On average, the MaxPair improves the system performance by 8% and 4%, respectively.

8.2.4 Summary of the Results

Figures 11(d) and 12(d) summarizes our results. In general, Time-Greedy outperforms Speedup-Greedy scheduling. Though for some special cases, MaxPair-Speedup is slightly worse than greedy-based algorithms, in majority cases, it outperforms both of the greedy methods. Given execution time details, the MaxPair constantly provides the best performance by maximizing the accumulated time-saving in all cases.

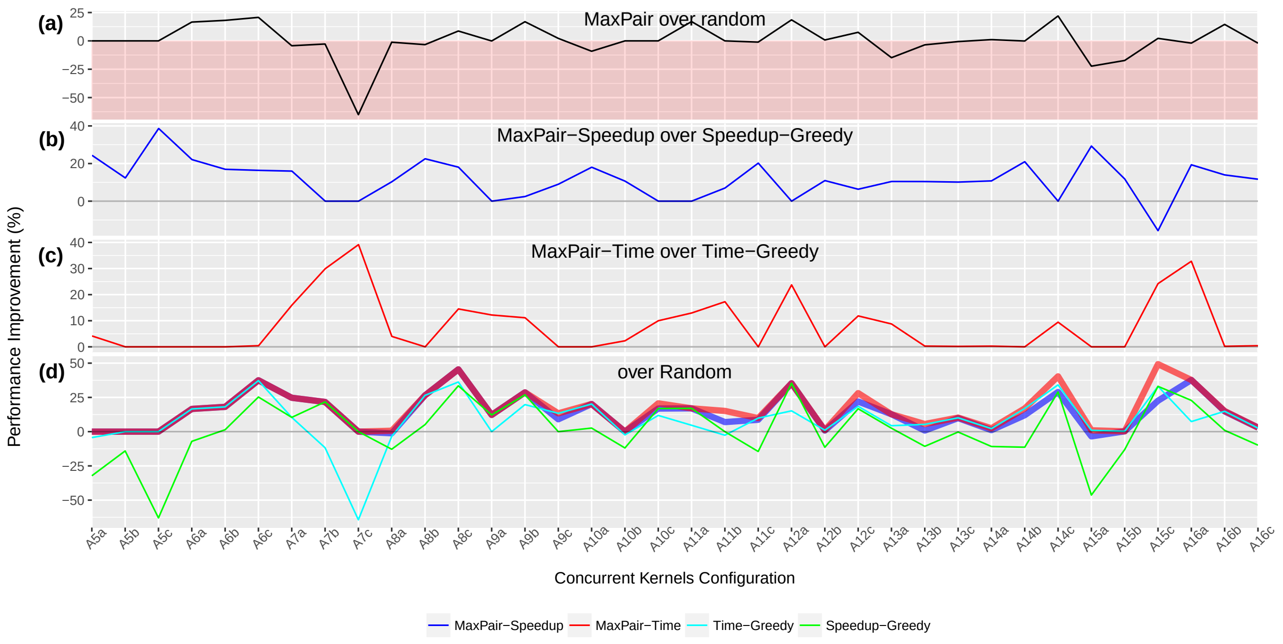


Figure 11. Performance improvement over alternative methods on the AMD system. When only kernel associativity is known, the MaxPair works as good/poor as the random concurrent scheduling on average, as shown in (a). Part (b) shows that with knowledge of concurrent kernels relative speedup, MaxPair outperform speedup-greedy algorithm in most of the case. If precise execution times are provided, MaxPair constantly works better than the time-greedy algorithm, as shown in (c). Subfigure (d) puts all methods together and normalize their results compared to random.

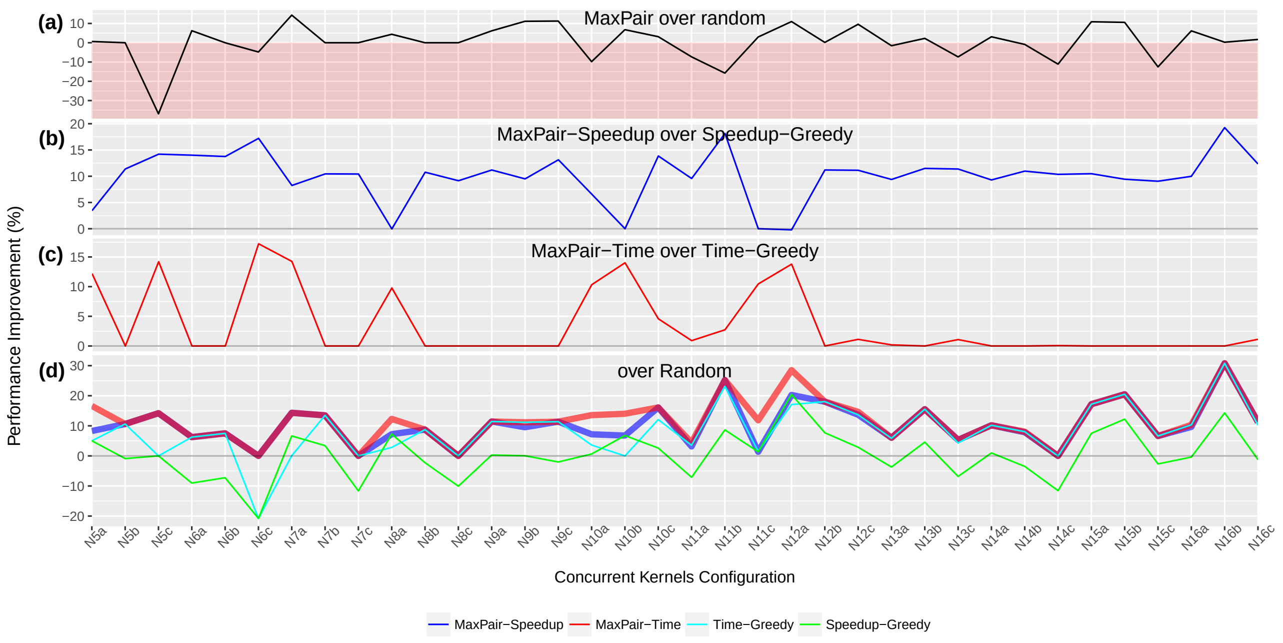


Figure 12. Performance improvement over alternative methods on Nvidia platform. MaxPair is 0.4% better than the random scheduling when only kernel associativity is given, as shown in (a). With information of relative speedup, MaxPair is 10% better than SG method, as shown in (b). With precise execution time, MaxPair is 4% better than TG method, like the line shown in (c). Figure (d) normalise MaxPair-Speedup, MaxPair-Time, Speedup-Greedy, and Time-Greedy results to the random scheduling. As can be seen from the graph, MaxPair always outperforms the other counterparts.

Table 6. Task configuration of the experiment

Index	Kernel Name	# Task
c5a	7-26-31-38-38	5
c5b	9-10-12-26-29	5
c5c	7-13-20-21-38	5
c6a	1-2-21-21-22-27	6
c6b	1-4-23-32-32-36	6
c6c	1-4-22-29-29-32	6
c7a	1-11-15-26-32-33-33	7
c7b	1-6-7-9-28-33-36	7
c7c	12-12-20-22-26-32-36	7
c8a	6-7-12-17-31-33-35-38	8
c8b	1-2-4-8-15-19-25-33	8
c8c	1-4-11-12-20-21-26-29	8
c9a	1-6-7-12-13-23-25-27-37	9
c9b	9-11-22-25-25-31-32-34-37	9
c9c	4-4-5-8-12-17-18-29-32	9
c10a	1-2-4-4-6-9-19-27-28-35	10
c10b	2-15-18-26-26-31-33-37-38-38	10
c10c	6-9-22-25-29-29-31-33-35-37	10
c11a	3-3-5-6-7-9-26-28-31-35-38	11
c11b	3-7-7-7-21-22-22-28-31-33-37	11
c11c	6-7-9-9-12-19-20-21-21-27-31	11
c12a	4-6-8-9-12-13-19-29-33-34-36-38	12
c12b	1-1-3-7-15-19-21-24-26-32-33-37	12
c12c	2-3-4-5-9-12-12-13-26-31-35-36	12
c13a	1-1-4-6-19-21-23-25-25-27-28-31-36	13
c13b	1-2-3-4-18-22-22-26-27-27-27-28-33	13
c13c	1-3-3-4-10-15-21-25-29-32-32-33-36	13
c14a	7-8-10-11-13-17-17-20-20-21-27-33-36-37	14
c14b	3-4-5-5-7-9-9-9-23-31-35-37-37-37	14
c14c	3-9-9-12-13-20-22-23-25-25-27-31-31-36	14
c15a	1-2-2-3-5-6-7-7-8-19-23-25-27-32-32	15
c15b	1-5-5-8-9-10-13-13-15-15-15-20-25-35-37	15
c15c	1-5-7-11-11-13-19-23-32-33-33-36-36-38-38	15
c16a	1-3-7-9-11-12-19-19-25-26-28-28-34-35-36-38	16
c16b	1-1-2-2-3-4-6-9-10-11-12-15-26-31-32-33	16
c16c	1-4-9-9-10-11-12-19-22-25-25-26-26-31-35-37	16

8.3 Runtime Cost

The MaxPair scheduler introduces two additional costs: generating the co-execution graph and running the matching algorithm.

Figure 13 shows the time required for generating the co-execution graph. The time spent on graph generating is linear to the number of tasks. The overhead introduced by this process is trivial. Taking a graph creation with 28 kernels for example, it needs 1.75 milliseconds on average which is far less than the sum of task execution time. One thing to note is that this overhead is not just introduced by MaxPair; greedy-based algorithm also need to create a co-execution graph.

Figure 14 presents the performance of MaxPair matching. Theoretically, the graph matching algorithm has a complexity of $O(n^3)$. In practice however, it works rather fast as the graph is sparse. For a graph with 28 nodes, the average time spent on the matching is about 2 milliseconds. As an extra overhead introduced by MaxPair, this runtime cost is neglectable.

Figures 13 and 14 also show the time proportion spent on graph creating and matching from the overall kernel execution time. They confirm that the runtime overhead is trivial and neglectable.

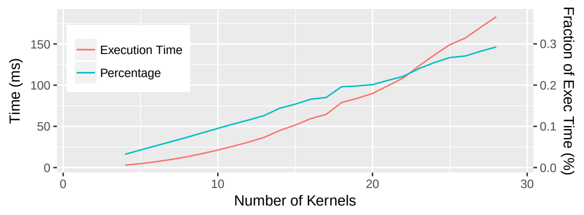


Figure 13. Overall time and percentage spent on graph creating.

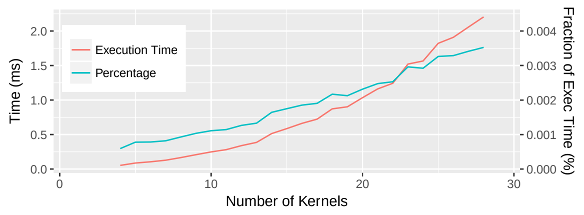


Figure 14. Overall time and percentage spent on graph matching.

9 Related Work

Both the space sharing approaches and scheduling methods for GPUs have been well studied in recent years. However, there is a lack of research on how to schedule workloads to share GPU by multiple concurrent workloads efficiently.

GPU Space Sharing At the compile and runtime, a GPU can be shared statically or dynamically [24]. Kernels can be launched through separated command queues on-the-fly to the same GPU or be merged/fused to form a new kernel which will be issued to the device as a substitute. In the former solution, typically kernels are chunked into workgroups first and then a runtime framework issues workgroups from different kernel to the same GPU via separate command queue simultaneously [13]. Slices [23, 30] with multiple workgroups are also used in fair sharing of the GPU by adjusting the kernels mixing ratio. Besides group into slices, interleaved issuing workgroups [3, 5, 11, 26, 29] to a GPU is also an option for sharing hardware among multiple kernels.

Apart from the runtime implementation, different kernels can also be physically merged to form a new kernel which performs as a substitute for the original ones. Multiple kernels are merged at the compile-time to either optimise data accessing [9, 18] or improve resource utilization [27]. Comparing to runtime solutions, static methods work more accuratly as the overhead of kernel fusion is not impacted by the GPU drivers.

Besides software methods, architectural solutions have been studied as well. The fairness of concurrent GPU application can be improved by augmenting memory scheduling scheme [10, 24], or by providing virtual memory system [4] or making the GPU device preemptible [17, 21, 22]. Jog *et al.* [14, 15, 25] have also been looking at concurrent kernel execution but with the aim to make the memory system of the GPU aware of it as opposed to our approach which focuses on finding beneficial kernel pairs. This work is orthogonal to our work proposed in this paper.

The combination of software and hardware optimisation improves both the efficiency and the utilisation of the GPU devices via running multiple workloads simultaneously. However, in practice, there is a lack of effective scheduling method.

Scheduling Schemes on the GPU Scheduling algorithms are guided by some information, like workload dependencies, relative speedup or execution time. For example, the Heterogeneous Earliest Finish Time scheme (HEFT) requires the programmer or the runtime system to provide execution time for each task on all candidate processors [7]. Also, the dependencies among workload are required to optimize the communication overhead. As a substitute for precise execution, execution models are also used as approximations when scheduling workloads to devices [2].

Scheduling concurrent workloads to the same GPU can be guided by execution time, relative speedup [13], compute-and-memory intensity [12, 16], bound of hardware resources [19] or use round-robin [5] and random methods (like First-In-First-Out) [20] when there is a lack of details about the target tasks.

All these scheduling mechanisms work properly. However, as they are all greedy-based algorithm, they provide suboptimal performance in many cases. Our work, on the other hand, delivers better performance than these state-of-the-arts with the same information used by greedy-based methods by a graph-based scheme.

10 Conclusion

Though concurrent kernel execution can improve performance of GPU execution, there is a lack of effective scheduling method. In this paper, we provide a graph-based solution: MaxpPair. By mapping the concurrent task execution to a graph matching problem, we adopt maximum weight graph matching algorithm. We compae our method to two state-of-the-art schemes on two different platforms. With the same knowledge of the workloads, MaxPair outperforms these schemes by 8% and 4% on average on the AMD and Nvidia platform, respectively.

Acknowledgments

This is work is in part supported by EPSRC (grant agreement EP/P010946/1).

References

[1] Jacob Adriaens, Katherine Compton, Nam Sung Kim, and Michael J. Schulte. 2012. The case for GPGPU spatial multitasking. In *HPCA*. IEEE Computer Society, 79–90.

[2] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2011. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience* 23, 2 (2011), 187–198.

[3] Mihir Awatramani, Joseph Zambreno, and Diane T. Rover. 2013. Increasing GPU throughput using kernel interleaved thread block scheduling. In *ICCD*. IEEE Computer Society, 503–506.

[4] Michela Becchi, Kittisak Sajjapongse, Ian Graves, Adam M. Procter, Vignesh T. Ravi, and Srimat T. Chakradhar. 2012. A virtual memory based runtime to support multi-tenancy in clusters with GPUs. In *HPDC*. ACM, 97–108.

[5] Mehmet E. Belviranli, Farzad Khorasani, Laxmi N. Bhuyan, and Rajiv Gupta. 2016. CuMAS: Data Transfer Aware Multi-Application Scheduling for Shared GPUs. In *ICS*. ACM, 31:1–31:12.

[6] Claude Berge. 1957. Two Theorems in Graph Theory. In *Proceedings of the National Academy of Sciences of the United States of America*. 842 – 844.

[7] Raphaël Bleuse, Safia Kedad-Sidhoum, Florence Monna, Grégory Mounié, and Denis Trystram. 2015. Scheduling independent tasks on multi-cores with GPU accelerators. *Concurrency and Computation: Practice and Experience* 27, 6 (2015), 1625–1638.

[8] Jack Edmonds. 1987. *Paths, Trees, and Flowers.* Birkhäuser Boston, 361–379.

[9] Jiri Filipovic, Matus Madzin, Jan Fousek, and Ludek Matyska. 2015. Optimizing CUDA code by kernel fusion: application on BLAS. *The Journal of Supercomputing* 71, 10 (2015), 3934–3957.

[10] Xiang Gong, Zhongliang Chen, Amir Kavyan Ziabari, Rafael Ubal, and David R. Kaeli. 2017. TwinKernels: an execution model to improve GPU hardware scheduling at compile time. In *CGO*. ACM, 39–49.

[11] Chris Gregg, Jonathan Dorn, Kim M. Hazelwood, and Kevin Skadron. 2012. Fine-Grained Resource Sharing for Concurrent GPGPU Kernels. In *HotPar*. USENIX Association.

[12] Marisabel Guevara, Chris Gregg, Kim Hazelwood, and Kevin Skadron. 2009. Enabling task parallelism in the CUDA scheduler. (01 2009).

[13] Qing Jiao, Mian Lu, Huynh Phung Huynh, and Tulika Mitra. 2015. Improving GPGPU energy-efficiency through concurrent kernel execution and DVFS. In *CGO*. IEEE Computer Society, 1–11.

[14] Adwait Jog, Evgeny Bolotin, Zvika Guz, Mike Parker, Stephen W. Keckler, Mahmut T. Kandemir, and Chita R. Das. 2014. Application-aware Memory System for Fair and Efficient Execution of Concurrent GPGPU Applications. In *GPGPU@ASPLOS*. ACM, 1.

[15] Adwait Jog, Onur Kayiran, Tuba Kesten, Ashutosh Pattnaik, Evgeny Bolotin, Niladrish Chatterjee, Stephen W. Keckler, Mahmut T. Kandemir, and Chita R. Das. 2015. Anatomy of GPU Memory System for Multi-Application Execution. In *MEMSYS*. ACM, 223–234.

[16] Teng Li, Vikram K. Narayana, Esam El-Araby, and Tarek A. El-Ghazawi. 2011. GPU Resource Sharing and Virtualization on High Performance Computing Systems. In *ICPP*. IEEE Computer Society, 733–742.

[17] Zhen Lin, Lars Nyland, and Huiyang Zhou. 2016. Enabling efficient preemption for SIMT architectures with lightweight context switching. In *SC*. IEEE Computer Society, 898–908.

[18] Thibaut Lutz, Christian Fensch, and Murray Cole. 2015. Helium: a transparent inter-kernel optimizer for OpenCL. In *GPGPU@PPoPP*. ACM, 70–80.

[19] Christos Margiolas and Michael F. P. O'Boyle. 2016. Portable and transparent software managed scheduling on accelerators for fair resource sharing. In *CGO*. ACM, 82–93.

[20] Sreepathi Pai, Matthew J. Thazhuthaveetil, and R. Govindarajan. 2013. Improving GPGPU concurrency with elastic kernels. In *ASPLOS*. ACM, 407–418.

[21] Jason Jong Kyu Park, Yongjun Park, and Scott A. Mahlke. 2015. Chimera: Collaborative Preemption for Multitasking on a Shared GPU. In *ASPLOS*. ACM, 593–606.

[22] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramírez, Nacho Navarro, and Mateo Valero. 2014. Enabling preemptive multiprogramming on GPUs. In *ISCA*. IEEE Computer Society, 193–204.

[23] Ayman Tarakji, Alexander Gladis, Tarek Anwar, and Rainer Leupers. 2015. Enhanced GPU Resource Utilization through Fairness-aware Task Scheduling. In *TrustCom/BigDataSE/ISPA (3)*. IEEE, 45–52.

[24] Guibin Wang, Yisong Lin, and Wei Yi. 2010. Kernel Fusion: An Effective Method for Better Power Efficiency on Multithreaded GPU. In *GreenCom/CPSCom*. IEEE Computer Society, 344–350.

[25] Haonan Wang, Fan Luo, Mohamed Ibrahim, Onur Kayiran, and Adwait Jog. 2018. Efficient and Fair Multi-programming in GPUs via Effective Bandwidth Management. In *HPCA*. IEEE Computer Society.

[26] Lingyuan Wang, Miaoqing Huang, and Tarek A. El-Ghazawi. 2011. Exploiting concurrent kernel execution on graphic processing units. In *HPCS*. IEEE, 24–32.

[27] Yuan Wen and Michael F. P. O'Boyle. 2017. Merge or Separate?: Multi-job Scheduling for OpenCL Kernels on CPU/GPU Platforms. In *GPGPU@PPoPP*. ACM, 22–31.

[28] Yuan Wen, Zheng Wang, and Michael F. P. O'Boyle. 2014. Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms. In *HiPC*. IEEE Computer Society, 1–10.

[29] F. Wende, F. Cordes, and T. Steinke. 2012. On Improving the Performance of Multi-threaded CUDA Applications with Concurrent Kernel Execution by Kernel Reordering. In *Application Accelerators in High Performance Computing (SAAHPC), 2012 Symposium on.*

[30] Jianlong Zhong and Bingsheng He. 2014. Kernelet: High-Throughput GPU Kernel Executions with Dynamic Slicing and Scheduling. *IEEE Trans. Parallel Distrib. Syst.* 25, 6 (2014), 1522–1532.