Portable and Transparent Software Managed Scheduling on Accelerators for Fair Resource Sharing

Portable and Transparent Software Managed Scheduling on Accelerators for Fair Resource Sharing

Christos Margiolas *
School of Informatics
University of Edinburgh
c.margiolas@ed.ac.uk

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

Abstract

Accelerators, such as Graphic Processing Units (GPUs), are popular components of modern parallel systems. Their energy-efficient performance make them attractive components for modern data center nodes. However, they lack control for fair resource sharing amongst multiple users. This paper presents a runtime and Just In Time compiler that enables resource sharing control and software managed scheduling on accelerators. It is portable and transparent, requiring no modification or recompilation of existing systems or user applications. We provide an extensive evaluation of our scheme with over 40,000 different workloads on 2 platforms and we deliver fairness improvements ranging from 6.8x to 13.66x. In addition, we also deliver system throughput speedups ranging from 1.13x to 1.31x.

Categories and Subject Descriptors D.3.4 [*Software*]: Programming Languages—Processors, Run-time environments, Optimization

General Terms Performance, Experimentation, Measurement

Keywords heterogeneous computing, GPUs, accelerators, resource management, fair resource sharing, accelerator sharing, multi-tasking, OpenCL

1. Introduction

Accelerators, such as Graphic Processing Units (GPUs), are increasingly popular components of modern parallel platforms. They deliver high computational throughput with reduced power for data parallel applications. However, this raw hardware performance comes at a software cost. Although highly parallel, the accelerators are managed as co-processors and support a limited number of concurrent kernel executions at a time.

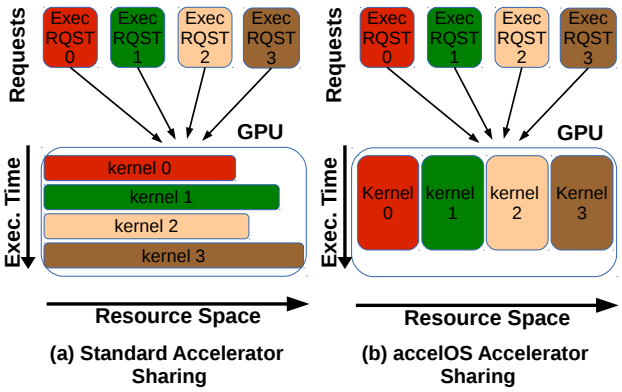


Figure 1: Sharing (a) standard OpenCL (b) accelOS

While sharing of accelerator resources is not an issue for dedicated application systems found in HPC, it is a real barrier for accelerator adoption in general purpose servers and data centers. Such systems typically host multiple users who cannot efficiently share and access accelerators. There is no fair resource sharing on accelerators for execution requests arriving concurrently from distinct users or applications.

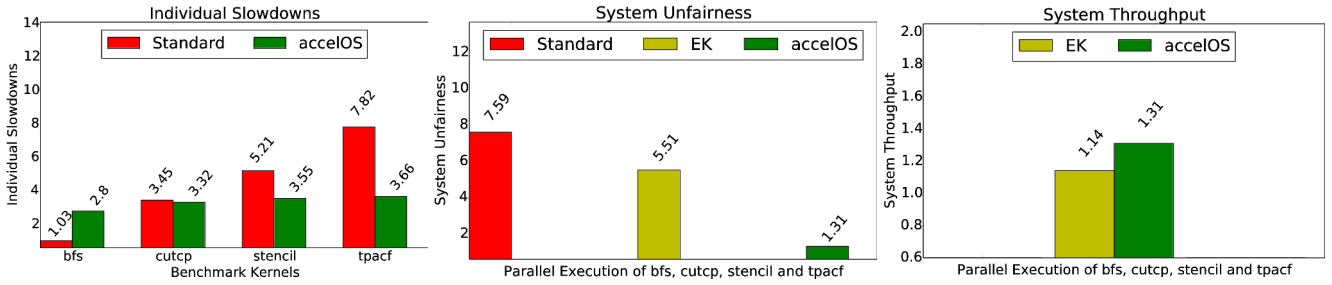
Modern computing systems need a mechanism that allows accelerators to be shared fairly among several concurrent kernel executions from distinct users. This should incur minimal overhead and support immediate deployment in existing systems with minimal disruption.

This paper develops a portable and transparent approach for accelerator sharing control. It enables concurrent space sharing of the accelerator by multiple kernels without any change to the application code, Operating System or hardware. It is immediately deployable on existing hardware and OpenCL[21] systems. Furthermore, our work does not compromise security in favor of improved accelerator sharing.

We achieve this by deploying a host runtime environment and an LLVM[23] based, Just In Time (JIT) compiler. It determines the number of work groups needed by each kernel

* Currently at Intel Labs, e-mail: christos.margiolas@intel.com

CGO'16, March 12–18, 2016, Barcelona, Spain
© 2016 ACM. 978-1-4503-3778-6/16/03...
http://dx.doi.org/10.1145/2854038.2854040



(a) Individual Slowdowns. (b) System Unfairness. (c) System Throughput Speedup.

Figure 2: Parallel execution of *bfs*, *cutcp*, *stencil*, and *tpacf*.

so that all fit and have an equal share of hardware resource. Kernel code is JIT modified so that the work performed by the original number of work-groups is dynamically assigned to the newly reduced number of work groups.

The need for concurrent space sharing of GPUs is well known; in fact GPU manufacturers have separate hardware queues specifically for this purpose. These are intended to allow efficient utilisation by different application streams and kernels. The NVIDIA architecture is a good example. In practice, however, although 2 or more kernels can be sent for execution, the hardware scheduler currently assigns all resources to which ever one arrives first. There is no notion of fair access. There have been hardware based proposals to improve performance [12][29][7][5] and memory bandwidth[33]. They do not, however, investigate multi-kernel scheduling and fair resource sharing. Furthermore, they crucially require hardware modifications that are not currently available.

There has been significant interest in software approaches to GPU sharing for performance [31][1][15] and power efficiency[18]. However, these techniques require static code merging with no dynamic control and do not investigate fair resource sharing. Furthermore, they raise security concerns because they merge kernel codes of different applications and users. There is also significant work proposing host runtime and Operating System techniques for managing accelerator resources [20][34][26]. However, they focus on allowing tasks to be easily allocated to a CPU or GPU, rather than resource sharing control on accelerators.

Prior work has investigated system resource sharing for non accelerator based systems [36][6][38][41][11] and a number of proposed metrics quantify fairness[14][17][13]. We adopt the fairness metric proposed in [9] and extend it to quantify fairness on accelerators.

This work presents accelOS, a host runtime and JIT compiler that enables resource control and scheduling on accelerators. Its operation remains transparent to the application, OS and runtime libraries as there is no requirement for code modifications or recompilation.

Our approach is evaluated extensively by using workloads consisting of multiple OpenCL kernels from the Parboil benchmark suite. We first evaluate all pairwise combinations of kernels ($25 \times 25 = 625$ in total). We then evaluate 16384, randomly selected 4-kernel combinations and 32768 randomly selected 8-kernel combinations. We compare our approach to the hardware baseline and the *Elastic Kernels* approach [31]. To show accelOS portability, we evaluate its performance on two modern heterogeneous platforms from two different manufacturers.

We dramatically improve fairness, ranging from 6.8x to 13.66x. This has the added bonus of improving system throughput on average from 1.13x to 1.31x. Our scheme incurs no overhead due to our optimizations; in fact we actually improve isolated kernel execution times due to dynamic scheduling. In fact, we deliver an average single kernel speedup of 1.07x and 1.1x on NVIDIA and AMD GPUs.

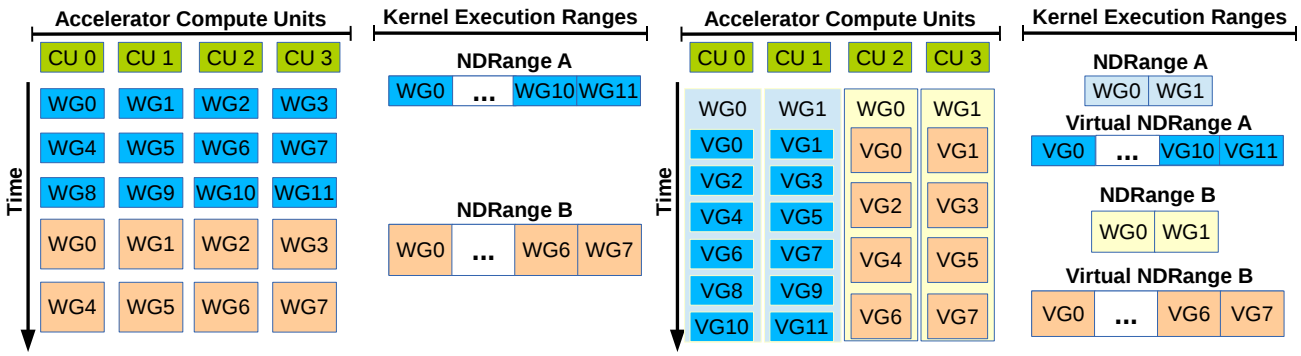
2. Motivation

Consider figure 1a which illustrates the typical execution of four kernels launched concurrently, from distinct applications, on a modern dedicated GPU. Rather than executing concurrently, each kernel is executed sequentially in turn. As each kernel is able to use the majority of the system resources, there is little space to execute the others. The first one to execute effectively excludes the rest. There is no resource sharing control and the architecture does not support preemption. This leads to unfair sharing for different applications and their users.

Figure 1b shows what happens when we use our infrastructure. It restricts resource allocation for kernels so that they have more work per thread but less concurrent threads and thus demand less system resources. The accelerator resources are now allocated equally among the four kernel executions. This new behavior leads to fair accelerator sharing, concurrent kernel executions and improved throughput.

2.1 Example

To make this concrete, let us consider the performance of 4 kernels, *bfs*, *cutcp*, *stencil*, and *tpacf* when concurrently executing on an NVIDIA platform, using first the standard software stack and then accelOS. Figure 2a shows the slowdown of each kernel when executed concurrently relative to executing in isolation. The standard scheme executes them in order and *bfs* has the least slowdown as it is executed first while program *tpacf* has the largest slowdown as it is executed last. accelOS slows each kernel more evenly giving



(a) Scheduling of work groups, WG, from kernels A and B.

(b) Fair accelOS scheduling of WG from kernels A and B.

Figure 3: (a) Standard OpenCL compared to (b) accelOS (b).

fairer access to the GPU. Using the unfairness metric [9], this means that accelOS is 5.79x times fairer (figure 2b).

As accelOS is better able to use system resources, it actually improves system throughput as well, 1.31x over the standard scheme, as shown in figure 2c. An alternative scheme, *Elastic Kernels* [31], attempts to statically merge kernels when system resources may not be fully utilised. This scheme is able to provide some improvement in system throughput, 1.14x, but only marginaly improves fairness, 5.51, as it does not allocate resources evenly.

2.2 Throughput vs Fairness

This paper targets a particular type of fairness where we aim to give each concurrent application an equal amount of hardware resources. As shown in figure 2, this means that each application is equally slowed down when sharing and has the added benefit of improved overall throughput and hence speedup. There may be occassions where it is deemed fairer to give more resources to one application over another *e.g.* if it is longer running or more important. This can easily be achieved by changing the sharing ratio. Determining the correct ratio is scenario dependent. In this paper we assume equal sharing as the default.

2.3 Standard Scheduling Approach

OpenCL does not expose any control on how accelerator resources are allocated among concurrent kernel execution requests. In practice, the execution request that *arrives* first tends to reserve all the available resources. This happens for two reasons. First, the hardware and firmware of the accelerator do not constrain the resources a kernel execution uses. Second, a kernel execution request typically represents a computational range that is large enough to occupy all the accelerator resources.

Consider figure 3 which illustrates accelerator sharing and work group scheduling for two parallel kernel execution requests (A and B). Here, the accelerator has 4 compute units (CUs). A's Kernel Execution Range (NDRange) consists of 12 work groups (WGs), while B has 8 work groups.

Figure 3a illustrates current accelerator sharing and work group scheduling. Here, the hardware/firmware scheduler

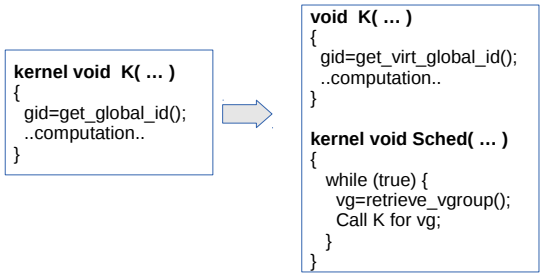


Figure 4: A high level schema of our JIT compiler transformation targeting OpenCL Kernels.

assigns work groups to compute units based on hardwired heuristics. There is no external control on how work groups are assigned to the compute units. The kernel that arrives first, kernel A in this example, allocates resources across all the compute units and the scheduler assigns work groups across the units in a round robin fashion. The work groups of kernel B start executing only after the completion of kernel A. The lack of resource sharing control leads to serialized kernel executions and unfair sharing.

2.4 accelOS: Software Scheduling & Resource Sharing Control

Figure 3b shows our approach. The number of work groups for both kernel executions is now reduced. Both kernels A and B now have just 2 work groups. The work groups of kernel A start executing on compute units 0 and 1, while the work groups of kernel B on compute units 2 and 3.

To ensure the original computation is performed, we have to compute all the original work groups of each kernel execution. First of all, each of the original 12 work groups of A (8 of B) are stored in a software queue as *virtual groups*. The software queue is stored in accelerator memory as *Virtual NDRange A (B)*. Next, our JIT compiler transparently modifies the kernel code as shown in figure 4. It now consists of a simple loop that dynamically dequeues a virtual group and executes it. This means all the original work is done but uses less physical resources. In figure 3b, work groups WG0 and WG1 dequeue virtual groups and execute them. The actual

virtual groups executed per compute unit will vary due to dynamic scheduling.

The reduction of NDrange and the software scheduling of virtual groups ensure fair sharing of accelerator resources and efficient allocation of work to compute units. The operations of accelOS take place transparently and do not require any modification of application code or changes in the existing software stack.

2.5 accelOS Design Internals

Our approach for software managed scheduling enforces resource sharing control across kernel execution requests. However, a kernel execution is bound to the resources initially allocated to it and cannot leverage additional resources that may be released if other kernel executions terminate first. The reason for this limitation is that the programming model and accelerator architectures do not provide this functionality neither expose the required level of architecture control.

Our approach does not introduce any limitation about the size and dimension of kernel execution ranges; in fact it delimits any related architecture constrains. The execution range, in our approarch, is represented at software level.

We also do not introduce any security compromise. Every kernel execution operates independently and we do not merge the code of different kernels.

3. Accelerator Resource Sharing Control

The key issue for our fair sharing scheme is determining the right number of work groups per kernel execution. We wish to determine the appropriate number of work groups for each kernel execution so that they all approximately allocate equal resources. We consider modern accelerator architectures with compute units that may host multiple work group executions at a time if their resource requirements can be satisfied. There are three resources that we need to consider for accelerator sharing: thread number, local memory usage and register usage.

Thread number We first have to constrain the number of work groups each kernel i executes so that all their threads can execute concurrently. If T is the maximum number of threads a device can execute, and w_i is the size of a work group for each kernel i then we must constrain the number of work groups x_i for each kernel: $\sum_i x_i w_i \leq T$.
To ensure that each kernel has roughly the same resources we have to allocate, as possible, equal number of hardware threads across the kernels. This is mathematically expressed as: $\min_i(\min_j(|x_i w_i - x_j w_j|))$ where we try to minimise the difference in threads number between all active kernels.

Local Memory A similar set of constraints can be built for local memory usage. Let L be the maximum local memory available, and m_i is the memory usage of a work group then the number of work groups y_i per kernel is: $\sum_i y_i m_i \leq L$.
We ensure that each kernel has roughly the same local memory usage by minimizing the difference in memory usage between all active kernels ($\min_i(\min_j(|y_i m_i - y_j m_j|))$).

Registers Again a similar set of constraints can be built for register usage. Let R be the maximum registers available, and r_i is the register usage of a work group then the number of work groups z_i per kernel is: $\sum_i z_i r_i \leq R$. We ensure that each kernel has roughly the same register usage by minimizing the difference in register file usage between all the active kernels ($min_i(min_j(|z_i r_i - z_j r_j|))$).

Determining Number of Work Groups Each of the three constraints can be approximately solved as follows: $x_i = \frac{T}{Kw_i}, y_i = \frac{L}{Km_i}, z_i = \frac{R}{Kr_i}$.
Given that all constraints must hold simultaneously the final work group size is $min(x_i, y_i, z_i)$. As these are Diophantine equations the resulting work group sizes may be conservative. If not all resources are used, we apply a simple greedy heuristic to incrementally increase the number of work-groups iteratively across the kernel executions until resource saturation.

4. Infrastructure Overview

We provide a high level overview of the accelOS infrastructure as it is shown in figure 5. This section demonstrates how our infrastructure is seamlessly integrated in existing systems without requiring code modifications or recompilation.

System Interface (level 0) It is the connection of accelOS with the existing system infrastructure. We use standard OpenCL to leverage accelerators. This way we can deploy our work on existing systems.

accelOS (level 1) It is a background system process that consists of a host runtime and a Just in Time (JIT) compiler. The host runtime manages accelerator resources and schedules kernel execution requests. The JIT compiler transforms kernel codes and links them against a scheduling library to support software work group scheduling.

Application interface (level 2) It is responsible for monitoring and interacting with OpenCL applications. This is done via a library called **ProxyCL** which replaces standard OpenCL. As we have demonstrated in prior work [26] this is done efficiently by using Interprocess Shared Memory for negligible communication overhead.

5. Host Runtime

This section presents the host runtime of accelOS. It consists of two components described below. The host runtime is built exclusively in user-space and relies on standard POSIX and OpenCL libraries.

Application Monitor This is the only component of accelOS that interacts with applications via **ProxyCL**. It monitors OpenCL requests made by applications. If these requests involve new kernel code compilation or kernel execution special actions take place. The finite state machine

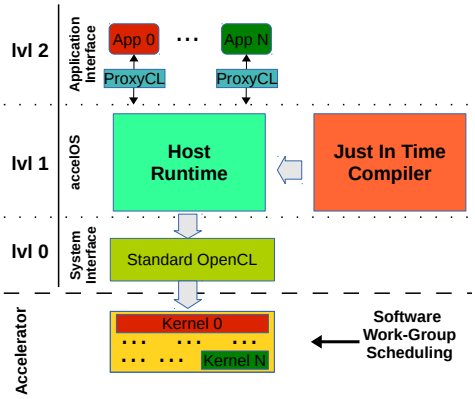


Figure 5: accelOS supports application interface (level 0), accelOS core (level 1) and a system interface (level 2). The core contains the host runtime and the JIT compiler.

(FSM) of figure 6 presents its operation. When an application performs an OpenCL request three scenarios may take place. (a) If the request creates a new OpenCL Program the JIT compiler takes control, analyzes and transforms the kernel code. The original operation is then performed with the transformed version of the code. (b) If the request is a new kernel execution, Kernel Scheduler takes control, which alters the number of work groups in order to control resource allocation and schedules its execution. (c) For any other request, the application continues its execution instantly and accelOS does not intervene.

Kernel Scheduler It centrally manages the scheduling of kernel execution requests. It leverages the resource sharing algorithm (section 3) to select the number of work groups for each kernel execution. For every request, it first constructs a *Virtual Kernel Execution Range* which is copied to the accelerator memory. It then alters the global size of the Kernel Execution Range to match the new number of work groups. It does not modify the work group size or the dimensions of the computation. Finally, it launches the kernel.

Memory Management The host runtime keeps track of the memory allocations of applications on the accelerator memory. It makes sure that all the allocations can be served safely. In case that the accelerator memory is not sufficient for serving all the applications concurrently, one or more applications may be paused.

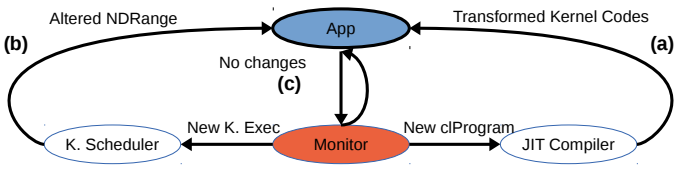


Figure 6: Application Monitor Finite State Machine.

6. Just In Time Compilation

Our JIT compiler intervenes in the standard compilation of OpenCL kernels. It transforms kernel codes and links them statically against a runtime library that enables software scheduling as we described in section 2.4. Its operations remain transparent to the application and no modifications are required. Our compiler infrastructure is based on LLVM [23] and we rely on vendor toolchains for target code generation.

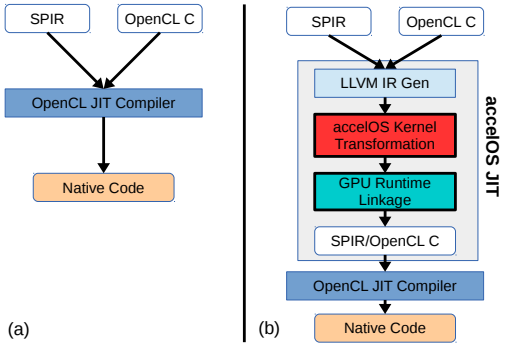


Figure 7: Kernel Compilation (a) Standard OpenCL against (b) accelOS.

6.1 Compilation Procedure

Figure 7a presents kernel compilation under standard OpenCL. The application provides the kernel code either in OpenCL C or SPIR[22]. The vendor compiler then performs a set of optimizations and generates native code for the accelerator.

Figure 7b presents our scheme. We intercept the OpenCL call that provides the kernel code. If the code is given in OpenCL C we use clang to generate LLVM IR. We instantiate an LLVM Pass Manager and load our compiler passes. We transform kernel codes and statically link them against our GPU scheduling library. Next, if the vendor compiler supports SPIR, we generate SPIR code. Otherwise, we generate OpenCL C. Finally, we use the vendor compiler for the target code generation.

6.2 Transformation Overview

Our compiler transformation enables software work group scheduling on existing OpenCL kernels transparently. For every OpenCL kernel we perform the following:

1. Convert the OpenCL kernel function to a regular computation function.
2. Extend the function interface with pointer arguments to include the data structures of the runtime library.
3. Replace built-in work item functions of OpenCL with runtime equivalents.
4. Create a scheduling kernel function. Its interface includes all the arguments of the original kernel function plus pointer arguments to the runtime data structures.
5. Generate the scheduling kernel body that atomically accesses the Virtual NDRange of the kernel execution and calls the computation function for every virtual group.

6.2.1 Kernel Transformation Example

Consider the code of figure 8a where each work item either adds or subtracts the input of two buffers depending on its

```
1  kernel void mop(global const float *ina,
2      global const float *inb, global float *out)
3  {
4    size_t gid=get_global_id(0);
5    size_t grid=get_group_id(0);
6
7    if(grid<NConstant)
8        out[gid]= ina[gid] + inb[gid];
9    else
10       out[gid]= ina[gid] - inb[gid];
11
12 }
```

(a) Kernel Code Example.

```
1  void mop(global const float *ina,
2  global const float *inb, global float *out,
3  global struct RT *rt, local struct SD *sd, int hdlr)
4  {
5    size_t gid=rt_global_id(rt, hdlr, 0);
6    size_t grid=rt_group_id(rt, hdlr, 0);
7
8    if(grid<NConstant)
9        out[gid]= ina[gid] + inb[gid];
10   else
11       out[gid]= ina[gid] - inb[gid];
12 }
13
14 kernel void dyn_sched(global const float *ina,
15     global const float *inb, global float *out,
16     global struct RT *rt, local struct SD *sd,
17     local void *lheap)
18 {
19   size_t ind;
20
21   if( rt_is_master_work_item() )
22       rt_env_init(rt,&sd);
23
24   for(;;){
25       if( rt_is_master_work_item() )
26           rt_sched_wgroup(rt,&sd);
27       barrier(CLK_LOCAL_MEM_FENCE);
28       if(sd.status==RUN_TERMINATE)
29           break;
30       for(ind=sd.wg_base; ind<sd.wg_end; ++ind)
31           mop(ina, inb, out, rt, sd, ind);
32   }
33 }
```

(b) Kernel Code After Transformation.

Figure 8: Kernel Code Transformation for software work group scheduling on accelerators. Our compiler transforms the original kernel code. It injects runtime calls and adds control flow for scheduling control and links against our GPU scheduling library.

group ID. Figure 8b shows the transformed code. We convert the kernel function to a regular function and replace the built-in, work item functions of OpenCL with runtime function calls as shown in lines 5 and 6. The first parameter **rt**, provides access to the *Virtual Virtual NDRange*, the second, **sd**, to scheduling information which is local to work group. The last, **hdlr**, is a runtime handler.

Function Calls An OpenCL kernel may call regular functions which perform work-item function calls that we should replace with our runtime functions. We do the replacement and we extend their interface to access **rt** and **sd**.

Software Scheduling Control Lines 14 to 33 of figure 8b perform the software scheduling of virtual groups. Scheduling is initialized in lines 21 and 22 by a single work item, the *master* of the work group. All the work items proceed to a loop, where the master, line 26, retrieves virtual groups for execution from the Virtual NDRange. We have an adaptive scheduling scheme that may assign more than one virtual groups for execution at a time. For every group we call the computation function, line 31.

Local Data Hoisting OpenCL standard exclusively permits declaration of data in local address space as part of a kernel function body and not regular function bodies. We convert the original kernel code to a regular function and we need to hoist its local data declarations in the scheduling kernel body.

6.3 Runtime Library

Our library performs the dynamic scheduling of virtual groups provided by Virtual NDRanges. Every kernel work group has a runtime instance performing virtual group scheduling independently. The library provides operations for environment control and scheduling. It also provides replacements for the built-in work item functions of OpenCL. The original work groups of a kernel execution are now described by Virtual NDRanges in accelerator memory and our function replacements provide the appropriate values at runtime.

6.4 Adaptive Scheduling

The runtime operations incur negligible overheads except for the scheduling operation which has atomic semantics. Scheduling of small kernels would expose significant overhead. To compensate for that we support scheduling of multiple virtual groups at a time. If the number of kernel instructions in LLVM IR is less than 10, a scheduling operation assigns 8 virtual groups to the work group at a time. Respectively, 6 groups for less than 20 instructions, 4 groups if less than 30, 2 groups if less than 40. Otherwise, the scheduling is done by 1 group at a time. We evaluate this in section 8.5.

6.5 Register Usage

Our transformation increases register usage by 3 per work-item. Nevertheless, after the function inlining, which is performed by default in GPU compilers, this overhead accounts to 0 or 1 registers per work-item. Our approach does not lead to register pressure.

7. Experimental Setup

This section describes the platforms, workloads, metrics and methodology used in our evaluation.

7.1 Evaluation Platforms

We evaluate our approach on two platforms. Both have an Intel i7 4770K CPU @ 3.50GHz and 16GB of DDR3 RAM at 1600Mhz. The first has an NVIDIA Tesla K20m[30] GPU; while the second has an AMD R9 295X2[2]. Both systems

run Linux with kernel version 3.13. We use the NVIDIA OpenCL platform, version 331.79 and the Accelerated Parallel Processing framework of AMD, version 1445.5.

7.2 Workloads

We use all the kernels from the OpenCL version of the Parboil benchmark suite [37]. The characteristics and nature of the benchmarks have been explored extensively in prior work [31]. We consider workloads consisting of **2**, **4** or **8** parallel kernel execution requests. We first evaluate all pairwise combinations of kernels. As there are 25 Parboil kernels, this gives $25 \times 25 = 625$ in total. It is impractical to evaluate all the available combinations for workloads of 4 and 8 requests and we evaluate a subset of them. There are 390265 4-kernel workload combinations from which we randomly selected 16384. There are 1.5×10^{11} 8-kernel combinations from which we randomly selected 32768. To have robust results, each workload is executed 20 times and the mean execution time is reported.

7.3 Comparison to Other Approaches

We present all results relative to the standard OpenCL provided by NVIDIA and AMD. To provide a broader evaluation, we implemented the the `Elastic Kernels` [31] approach. This work was originally aimed at CUDA and required a new implementation for OpenCL.

7.4 Metrics

We evaluate our scheme with respect to fairness and throughput using existing metrics.

Fairness Metrics for Accelerator Sharing A heterogeneous system is considered fair, if the slowdowns of kernel executions running concurrently on the accelerator resources are the same [9][28][36]. We adopt the metrics proposed in [9]. Given K kernels, the Individual Slowdown IS_i of a kernel execution i is: $IS_i = \frac{T(s)_i}{T(a)_i}$. $T(s)$ is the time taken sharing with other executions. $T(a)$ is time executing in isolation. *System unfairness*, U is defined: $U = \frac{\max(IS_0, IS_1, ..., IS_{K-1})}{\min(IS_0, IS_1, ..., IS_{K-1})}$. *Fairness improvement* over baseline for either our scheme or Elastic Kernels is a simple ratio: $\frac{U_{baseline}}{U_X}$, where $U_{baseline}$ and U_X are the system unfairness values for standard OpenCL and either our scheme or Elastic Kernels, respectively.

Kernel Execution Overlap The amount of time kernels co-execute is another measure of GPU sharing. *Execution overlap* O is defined as $O = \frac{T(c)}{T(t)}$, where $T(t)$ is the total time the accelerator is executing at least one of the kernels and $T(c)$ is the amount of time all the kernels are co-executing.

Throughput Speedup Although we focus on fairness, overall performance is also important. We report overall throughput speedup relative to the baseline i.e. $(\frac{T_{baseline}}{T_X})$ where $T_{baseline}$ is the time for *all* kernels to execute on the standard system and T_x is the time for either our approach or the Elastic Kernels to execute *all* kernels.

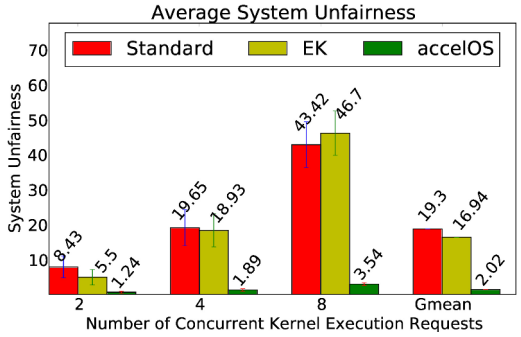
Additional metrics To evaluate the overhead of our scheme, we measure the time for a single kernel to execute using our approach vs the baseline. We also report average normalized turn around time (ANTT) and worst case ANTT to allow direct comparison with [31]. We also provide STP[10] results.

8. Results

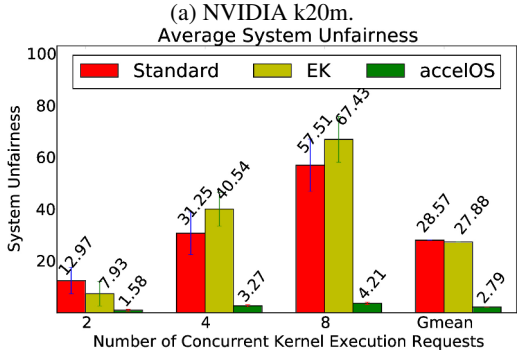
In this section, we evaluate accelOS for fairness and performance on varying multi-kernel workloads and compare against Elastic Kernels where appropriate.

8.1 Fairness in Accelerator Sharing

Here, we investigate what is the impact of accelOS on fairness. We use the metrics of *Unfairness* and *Fairness Improvement* as described in section 7. For the Unfairness metric, lower values are better while for the Fairness Improvement metric, higher values are better.



(a) NVIDIA k20m.

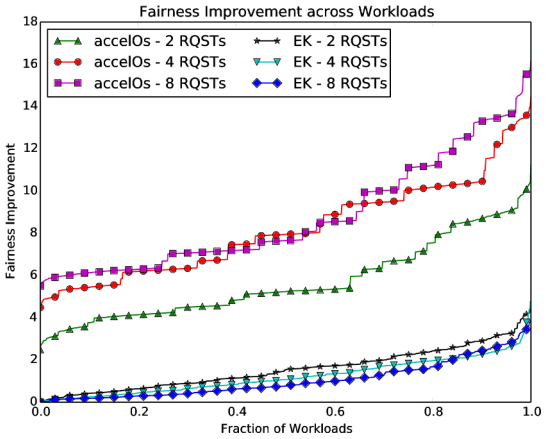


(b) AMD R9 295X2.

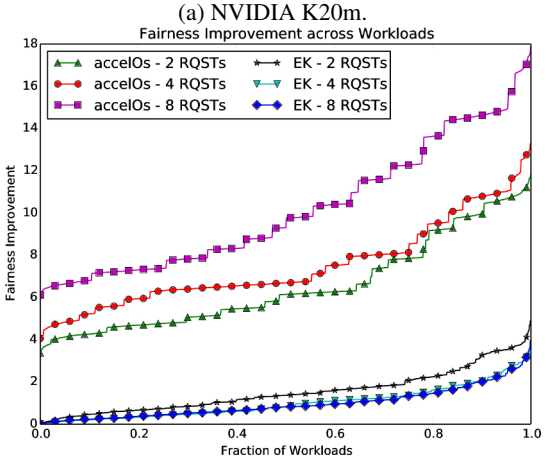
Figure 9: Average System Unfairness. Standard OpenCL, *Elastic Kernels (EK)* and accelOS. Lower is better. accelOS outperforms the other approaches.

8.1.1 Result Summary

Figure 9a shows the average results on the NVIDIA platform. accelOS reduces unfairness from 8.43 to 1.24 for 2 requests, from 19.65 to 1.89 for 4 requests and from 43.42 to 3.54 for 8 requests. It leads to fairness improvements of 6.8x, 10.4x and 12.27x, while the average improvement is 9.55x. accelOS outperforms *Elastic Kernels (EK)* approach which delivers fairness improvements of 1.53x, 1.03x and 0.93x, respectively and an average improvement of 1.13x.



(a) NVIDIA K20m.



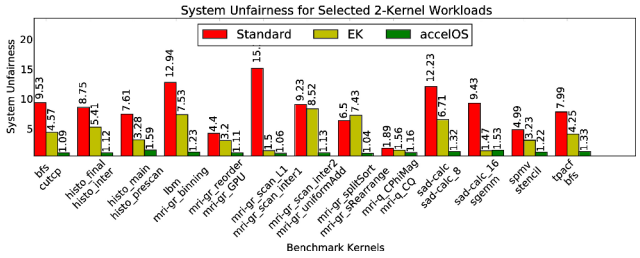
(b) AMD R9 295X2.

Figure 10: Fairness Improvements delivered by *accelOS* and *Elastic Kernels (EK)* for sets of 2, 4 and 8 kernel execution requests relative to standard OpenCL. Higher is better.

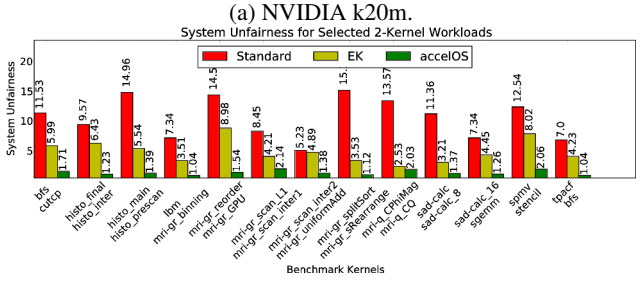
Figure 9b shows the results on the AMD platform. Here, the benefits of accelOS are similar to those of NVIDIA. accelOS delivers 8.21x improvement for 2 execution requests, 12.97 against 1.58. In the case of 4 requests, it improves fairness by 9.56x where it reduces unfairness from 31.25 to 3.27. For 8 requests, accelOS improves by 13.66x, reducing unfairness from 28.57 to 2.79. accelOS, again, outperforms *EK* which delivers fairness improvements of 1.63x, 0.77x and 0.85x with an average of 1.02x.

8.1.2 Individual Results

Figures 10a and 10b provide an overview of the fairness improvement results across all the workloads we use in our experiments. We provide individual results for workloads of 2, 4 and 8 kernel execution requests on both NVIDIA and AMD platforms for accelOS and *EK*. accelOS results range from 0.81x to 15.84x times improvement, where less than 2% of the workloads have a negative fairness result. In contrast, the EK delivers negative results for 44% of the workloads. accelOS uses dynamic resource sharing via software managed scheduling while *EK* uses static resource



(a) NVIDIA k20m.



(b) AMD R9 295X2.

Figure 11: Unfairness results for some 2-kernel workloads. The selection has been done by pairing the available OpenCL kernels by the alphabetical order of their names. We provide unfairness results for standard *OpenCL*, *Elastic Kernels (EK)* and *accelOS*. Lower is better.

allocation. accelOS successfully adapts to large number of requests and fairly assigns system resources while *EK* fails.

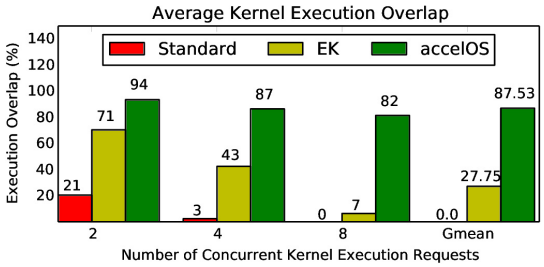
8.1.3 Pairwise Results

In order to drill down to specific workloads, we present fairness results delivered by accelOS and *EK* for a selection of 2-kernel workloads in figure 11. As there is insufficient space to show all 625 combinations, we pair each benchmark with its alphabetic neighbor, i.e. *bfs* with *cutcp*, *histo_final* with *hist_inter*. As there are 25 benchmarks, we show 13 pairs. This alphabetic pairing is arbitrary and used to prevent cherry-picking of results. accelOS steadily delivers the best results on both NVIDIA and AMD. There are two cases where *EK* and accelOS deliver nearly the same results. The sad-calc_16 - sgemm pair on NVIDIA and mri-q_computePhiMag - mri-q_ComputeQ pair on AMD suffer from performance degradations due to work group imbalances that negatively affect our software scheduling heuristics. These performance degradations in conjunction with the execution times of these kernels make accelOS less effective. However, accelOS delivers significant improvements even for these two cases.

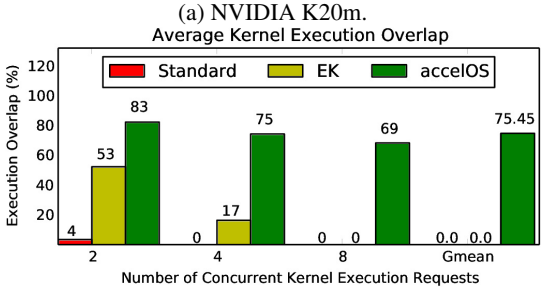
8.2 Concurrent Kernel Executions

Here, we examine how many kernels are actually executing concurrently. We use the Kernel Execution Overlap metric described in section 7. Higher is better.

Figure 12a gives the results for NVIDIA. In the case of 2 requests, we improve concurrent kernel execution from 21% to 94%. For 4 requests, standard OpenCL delivers 3% while we deliver 87%. Finally, for 8 requests, standard OpenCL

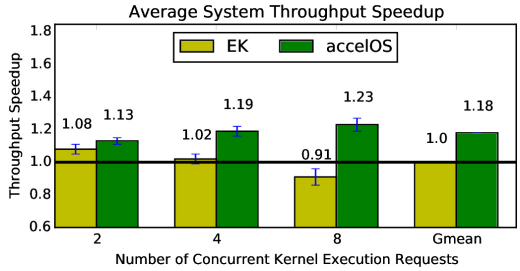


(a) NVIDIA K20m.

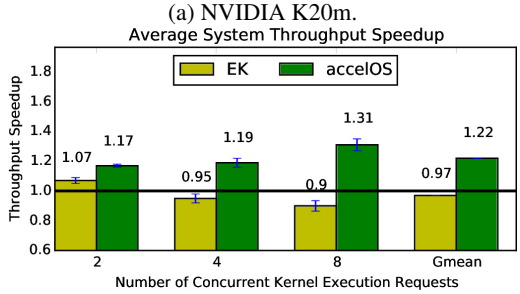


(b) AMD R9 295X2.

Figure 12: Average Kernel Execution Overlap. Comparison of the kernel execution overlap (percentage) on standard OpenCL and *Elastic Kernels (EK)* against accelOS approach. Higher is better.



(a) NVIDIA K20m.

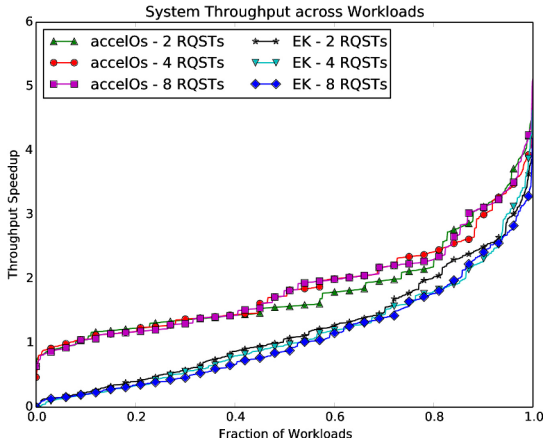


(b) AMD R9 295X2.

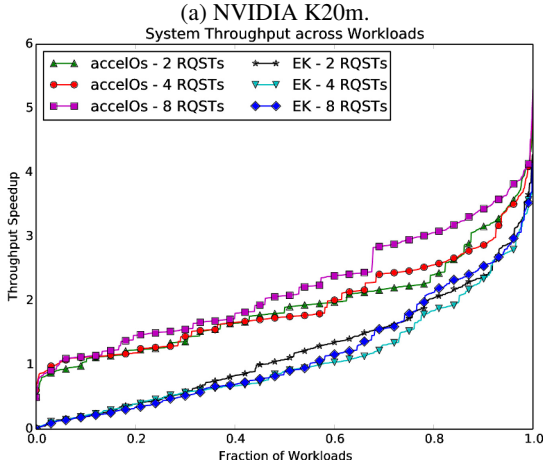
Figure 13: Average System Throughput Speedups for sets of 2, 4 and 8 kernels. The baseline is the standard OpenCL. Higher is better.

delivers 0%, while we enable 82%. accelOS outperforms *Elastic Kernels (EK)* approach which delivers 71%, 43% and 7%, respectively.

The AMD platform behaves worse than NVIDIA. As seen in 12b, standard OpenCL delivers 4%, 0% and 0% for 2, 4 and 8 requests, respectively. accelOS gives increased



(a) NVIDIA K20m.



(b) AMD R9 295X2.

Figure 14: System Throughput Speedups for sets of 2, 4 and 8 kernel. The baseline is the standard OpenCL. Higher is better.

concurrency. It delivers 83%, 75% and 69% for the 3 request sizes. accelOS again is more efficient than EK which delivers 53%, 17% and 0%, respectively.

On both NVIDIA and AMD, the execution overlap results are lower when we scale up from 2 to 8 requests. This happens because (a) the accelerator multi-tenancy leads to higher resource contention between the kernel executions and (b) the varying kernel workloads that lead to execution time imbalances.

8.3 System Throughput

We evaluate throughput speedups delivered by accelOS and compare against the *Elastic Kernels (EK)*[31]. The baseline is the standard OpenCL.

8.3.1 Result Summary

The results for NVIDIA are shown in figure 13a. We deliver an average speedup of 1.13x against 1.08x of *EK* for 2 requests and 1.19x against 1.02x of *EK* for 4 requests. Finally, we deliver a speedup of 1.23x against 0.91x of *EK* for 8 re-

quests. On average for all the request sizes, accelOS delivers 1.18x while *EK* 1.00x.

The results for AMD are shown in figure 13b. We deliver a speedup of 1.17x against 1.07x of *EK* for 2 requests, 1.19x against 0.95x of *EK* for 4 requests. Finally, we deliver a speedup of 1.31x against 0.9x for 8 requests. On average for all the request sizes, accelOS delivers 1.22x while *EK* 0.97x.

accelOS enables resource sharing control and dynamic work group scheduling. This leads to significant throughput results that increase as we scale up to larger number of requests. In contrast, *EK* relies on static heuristics and static resource allocation and fails to manage large number of requests or adapt to dynamic system changes. Due to this *EK* delivers negative results for large number of requests.

8.3.2 Individual Results

Figures 14a and 14b provide an overview of the throughput speedup results across the workloads we use in our experiments. We provide individual results for workloads of 2, 4 and 8 kernel execution requests on both NVIDIA and AMD platforms for accelOS and *EK*. The throughput speedup results range from 0.52x to 4.8x. Less than 5% of the workloads have slowdowns for accelOS while 54% of the workloads have slowdowns for *EK*.

	EK			accelOS		
RQSTs	**STP**	**ANTT**	**W. ANTT**	**STP**	**ANTT**	**W. ANTT**
2	1.13	3.57	56.7	1.15	1.12	8.2
4	0.99	4.33	72.2	1.18	1.32	14.2
8	0.93	7.57	87.59	1.25	1.78	23.1

Table 1: Additional metrics for accelOS and *EK* on NVIDIA. Higher values are better for STP, while lower values are better for ANTT. W.ANTT is the worst ANTT value reported.

	EK			accelOS		
RQSTs	**STP**	**ANTT**	**W.ANTT**	**STP**	**ANTT**	**W. ANTT**
2	1.04	4.2	64.6	1.18	1.35	13.4
4	0.97	6.83	84.6	1.18	2.12	19.5
8	0.92	7.98	98.54	1.28	3.26	31.34

Table 2: Additional metrics for accelOS and *EK* on AMD. Higher values are better for STP, while lower values are better for ANTT. W. ANTT is the worst ANTT value reported.

8.4 Additional Evaluation Metrics

Prior research work has considered some additional metrics which are STP[31][10] for system throughput evaluation and ANTT[31] as an indirect metric for quantifying system fairness. We provide a brief summary of the average results for Elastic Kernels (EK) and accelOS on NVIDIA, in table 1 and on AMD, in table 2. accelOS delivers better results on both platforms. It improves system throughput as the number of kernels increases, while EK can only manage this for 2-kernel workloads. Its average turnaround time and worst case time turn around time is significantly better than EK.

8.5 accelOS Performance Impact

AccelOS adds a software layer that may compromise performance. We therefore examine single kernel execution times delivered by accelOS against standard OpenCL. We consider two versions of accelOS, the (a) *naive* and (b) *optimized* versions. *Optimized* includes adaptive scheduling described in section 6. Figures 15a and 15b present speedups for the NVIDIA and AMD GPUs, respectively. Naive introduces a small overhead while optimized on average actually improves performance due to load balancing of work.

In the case of NVIDIA, shown in figure 15a, speedup values range from 0.92x to 1.03x for *naive* and from 0.96x to 1.14x for *optimized*. The geometric average is 0.98x for *naive* and 1.07x for *optimized*. For the *optimized* version, the one we use for our experiments, benchmark kernels sgemm and uniformAdd of mri-gridding have the lowest values, 0.96x and 0.97x, while splitSort of mri-gridding and GPU of mri-gridding have the highest values of 1.13x and 1.14x, respectively.

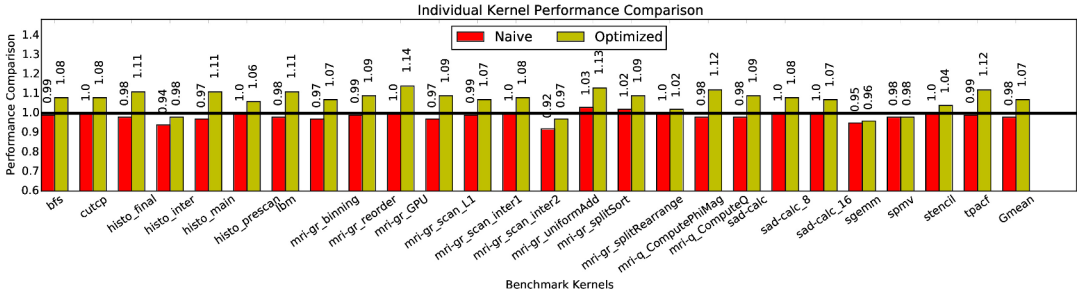
In the case of AMD, shown in figure 15b, speedups range from 0.91x to 1.04x for *naive* and from 0.95x to 1.19x for *optimized*. For the *optimized* version, the one we use for our experiments, kernels such as ComputePhiMag of mri-q and calc_16 of sad have the lowest values, 0.95x and 0.96x, while kernels lbm and splitSort of mri-gridding have the highest values of 1.18x and 1.19x. The geometric average is 0.99x for *naive* and 1.10x for *optimized*.

Our naive implementation leads to average slowdowns of 2% (NVIDIA) and 1% (AMD). However our optimized version does not just compensate the overhead but it leads to significant performance gains. This is due to the software scheduling which is dynamic and leads to well balanced scheduling. As we describe in section 6.4, we consider the overhead of our runtime for short kernels where we use adaptive scheduling to minimize that overhead. However, we still suffer small slowdowns for few kernels on both platforms.

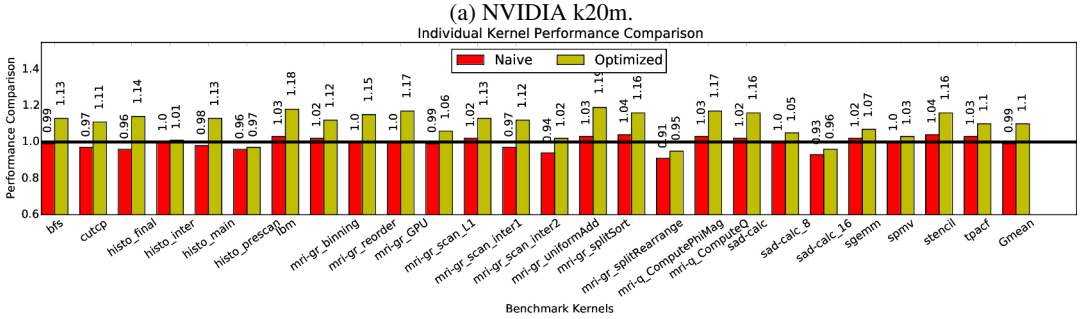
To quantify the performance of accelOS for small kernel executions, which cannot effectively utilize all the accelerator resources, we generated artificial small datasets and modified versions of bfs, spmv and tpacf. We run kernel executions with 2, 4 and 8 work groups. Our measurements show that for such small kernels the execution times differ by less than 3% for standard OpenCL against accelOS and that there is significant system noise due to the driver overheads.

9. Related Work

We presented a runtime and compiler infrastructure for fair accelerator sharing and efficient concurrent kernel executions. Our work seamlessly integrates with existing systems while it does not require any modification of the application or software stack. Here, we discuss prior work and we clarify our contributions.



(a) NVIDIA k20m.



(b) AMD R9 295X2.

Figure 15: accelOS Performance Impact. We compare accelOS against the standard OpenCL environment. We consider two versions of accelOS, the *naive* and *optimized*. The *naive* leads to small average slowdowns while the *optimized* significantly boosts performance. By default, we use the *optimized* version.

In [31][1][15], the authors propose techniques that increase accelerator utilization by statically merging multiple kernel executions. These techniques are static, require application modifications and raise security concerns. Furthermore, they do not investigate real multi-tasking scenarios where different number and types of applications may join or leave a system dynamically. In contrast, our work is transparent to applications and software stack, adaptive to varying workloads and guarantees safety.

Kernelet[39] reduces GPU underutilization in shared systems. It performs kernel slicing by dividing a kernel into multiple sub-kernels. Each kernel slice has tunable occupancy to allow concurrent execution of multiple slices.

TimeGraph [20] is a real-time scheduler that enables GPU sharing across applications and mimics preemption behavior in the absence of hardware support. It is implemented as part of the kernel driver, it is vendor specific and its evaluation is limited to OpenGL graphics benchmarks. In [27], the authors have modified the Linux kernel and intervene in the driver operations to enable fair sharing of NVIDIA GPUs between multiple applications. This work aims on new driver features and it is orthogonal to our work. PTask[34] provides a set of OS abstractions for GPUs and enables a dataflow programming model with enhanced programmability. However, it does not consider fair sharing, a problem solved by our work. GPES[40] combines user-level and driver-level techniques to deliver scheduling guarantees in real time systems.

In [12], the authors propose architecture modifications for efficient SIMD branch execution on GPUs. In [29], the authors present the large warp micro-architecture and two-level warp scheduling for effective resource utilization. Paper [7] presents a warp scheduling algorithm with flexible round robin policies that delivers efficient resource utilization. A cache-conscious scheme for warp scheduling is proposed in [33]. Paper [16] provides the design and evaluation of various algorithms that manage thread divergence encountered in recursive programs. OWL[19] presents wrap scheduling techniques for reducing memory access latencies. In [25], the authors provide improved GPU utilization via alternative thread block scheduling algorithms.

rCUDA[8] and [4] enable access to GPUs located on remote cluster nodes. Paper [35] presents the design of a GPU resource abstraction for balancing CUDA requests across cluster nodes and their accelerators. The SKMD framework[24] performs collaborative execution of data parallel kernels across multiple asymmetric CPUs and GPUs. An auto-tuning framework and runtime collaborative scheme of different accelerator types is presented in [32]. In both cases, however, only single applications are considered. Concord [3] is a compiler and runtime framework that enables efficient execution of kernels on integrated GPUs. It enables seamless sharing of pointer containing data structures but again it targets just a single application. These papers present infrastructures and techniques for sharing and exploiting multiple accelerators in parallel, however they do not investigate accelerator resource sharing and dynamic scheduling for multi-tenancy. Our work provides an efficient, portable and transparent solution for this.

10. Conclusion

In this paper we presented accelOS, a runtime and compiler infrastructure that enables software work group scheduling on accelerators. It enables fair accelerator sharing, efficient multi-kernel executions and throughput speedups. accelOS integrates seamlessly with the existing software stack and it does not require any modification or recompilation of applications, libraries or drivers. We delivered fairness improvements ranging from 6.8x to 13.66x for multi-kernel workloads of various sizes. Furthermore, we deliver system throughput speedups ranging from 1.13x to 1.31x. Future work will investigate additional techniques for software managed scheduling on accelerators.

References

[1] J. Adriaens et al. The case for gpgpu spatial multitasking. HPCA '12.

[2] AMD. Accelerated parallel processing: Opencl programming guide revision 2.7, 2013.

[3] R. Barik, R. Kaleem, et al. Efficient mapping of irregular c++ applications to integrated gpus. CGO '14.

[4] M. Becchi et al. A virtual memory based runtime to support multi-tenancy in clusters with gpus. HPDC '12.

[5] N. Brunie et al. Simultaneous branch and warp interweaving for sustained gpu performance. ISCA '12.

[6] F. J. Cazorla et al. Qos for high performance smt processors for embedded systems. *IEEE MICRO*, 24(4):24–31, 2004.

[7] J. Chen et al. Guided region-based gpu scheduling: Utilizing multi-thread parallelism to hide memory latency. IPDPS '13.

[8] J. Duato et al. rcuda: Reducing the number of gpu-based accelerators in high performance clusters. HPCS '10.

[9] E. Ebrahimi et al. Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems. ASPLOS '10.

[10] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *Micro, IEEE*, 28(3):42–53, 2008.

[11] S. Eyerman et al. Probabilistic job symbiosis modeling for smt processor scheduling. ASPLOS '10.

[12] W. W. L. Fung et al. Dynamic warp formation and scheduling for efficient gpu control flow. MICRO '07.

[13] R. Gabor et al. Fairness and throughput in switch on event multithreading. MICRO '06.

[14] A. Ghodsi et al. Dominant resource fairness: Fair allocation of multiple resource types. NSDI '11.

[15] M. Guevara et al. Enabling task parallelism in the cuda scheduler. In *Workshop on Programming Models for Emerging Architectures '09*.

[16] X. Huo et al. Efficient scheduling of recursive control flow on gpus. ICS '13.

[17] R. Jain et al. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems.

[18] Q. Jiao et al. Improving gpgpu energy-efficiency through concurrent kernel execution and dvfs. CGO '15.

[19] A. Jog et al. Owl: Cooperative thread array aware scheduling techniques for improving gpgpu performance. ASPLOS '13.

[20] S. Kato et al. Timegraph: Gpu scheduling for real-time multi-tasking environments. USENIX ATC '11.

[21] Khronos Group. The opencl specification, version 1.2, '11.

[22] Khronos Group. Spir (standard portable intermediate representation), version 1.2, '12.

[23] C. Lattner et al. Llvm: A compilation framework for lifelong program analysis & transformation. CGO '04.

[24] J. Lee et al. Transparent cpu-gpu collaboration for data-parallel kernels on heterogeneous systems. PACT '13.

[25] M. Lee et al. Improving gpgpu resource utilization through alternative thread block scheduling. HPCA '14.

[26] C. Margiolas et al. Palmos: A transparent, multi-tasking acceleration layer for parallel heterogeneous systems. ICS '15.

[27] K. Menychtas et al. Disengaged scheduling for fair, protected access to fast computational accelerators. ASPLOS '14.

[28] O. Mutlu et al. Stall-time fair memory access scheduling for chip multiprocessors. MICRO '07.

[29] V. Narasiman et al. Improving gpu performance via large warps and two-level warp scheduling. MICRO '11.

[30] NVIDIA. Nvidia kepler gk110 architecture, 2012.

[31] S. Pai et al. Improving gpgpu concurrency with elastic kernels. ASPLOS '13.

[32] P. M. Phothilimthana et al. Portable performance on heterogeneous architectures. ASPLOS '13.

[33] T. G. Rogers et al. Cache-conscious wavefront scheduling. MICRO '12.

[34] C. J. Rossbach et al. Ptask: Operating system abstractions to manage gpus as compute devices. SOSP '11.

[35] D. Sengupta et al. Multi-tenancy on gpgpu-based servers. VTDC '13.

[36] A. Snavely et al. Symbiotic jobscheduling for a simultaneous mutlithreading processor. *SIGPLAN Not.*, 35, Nov. 2000.

[37] J. A. Stratton et al. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 2012.

[38] A. Tucker et al. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *ACM SIGOPS Operating Systems Review*, volume 23, pages 159–166. ACM, 1989.

[39] J. Zhong et al. Kernelet: High-throughput gpu kernel executions with dynamic slicing and scheduling. *Parallel and Distributed Systems, IEEE Transactions on*, 25(6), 2014.

[40] H. Zhou et al. Gpes: a preemptive execution system for gpgpu computing. RTAS '15.

[41] S. Zhuravlev et al. Addressing shared resource contention in multicore processors via scheduling. ASPLOS '10.