Procedia Computer Science

Volume 29, 2014, Pages 231–245

ICCS 2014. 14th International Conference on Computational Science

OS Support for Load Scheduling on Accelerator-based Heterogeneous Systems

Ayman Tarakji, Niels Ole Salscheider, David Hebbeker

Faculty of Electrical Engineering and Information Technology, RWTH Aachen University
Aachen, Germany
{tarakji, salscheider, hebbeker}@lfbs.rwth-aachen.de

Abstract

The involvement of accelerators is becoming widespread in the field of heterogeneous processing, performing computation tasks through a wide range of applications. With the advent of the various computing architectures existing currently, the need for a system-wide multitasking environment is increasing. Therefore, we present an OpenCL-based scheduler that is designed as a multi-user computing environment to make use of the full potential of available resources while running as a daemon. Multiple tasks can be issued by means of a `C++` API that relies on the OpenCL `C++` wrapper. At this point, the daemon takes over the control immediately and performs load scheduling. Due to its implementation, our approach can be easily applicable to a common OS. We validate our method through extensive experiments deploying a set of applications, which show that the low scheduling costs remain constant in total over a wide range of input size. Besides the different CPUs, a variety of modern GPU and other accelerator architectures are used in the experiments.

Keywords:

Contents

Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2014

doi:10.1016/j.procs.2014.05.021

1 Introduction

In high performance computing, today's supercomputers[1] are clusters of machines consisting of CPUs, GPUs and other modern accelerators. These accelerators are either connected as PCIe devices to a host system, or on-board integrated in modern general purpose platforms [3]. Thus, an efficient and autonomous administration unit is more and more needed to facilitate the access to the accelerators as shared devices in a multi-user environment. In particular, when considering the utilization problem indicated in past work [1, 5, 12], an autonomous scheduling unit might be required to exploit the computation power of accelerator resources. OpenCL with its key features including the portability and low level access, establishes a foundation for such a unit. Besides being an open standard for parallel programming, OpenCL provides a unique benefit due to its ability to target a variety of devices. Each OpenCL capable device (e. g. CPUs, GPUs and generic accelerators) in a computing system interacts with the host through a unified model [11]. In other words, OpenCL is tailored to operate on heterogeneous systems, abstracting different computing architectures. There exist OpenCL runtime libraries for a variety of hardware platforms including: GPUs, CPUs, *Intel* Xeon Phi and digital signals processors (DSPs).

This paper presents a scheduling mechanism *OCLSched* for common computation tasks on heterogeneous computing systems. Our scheduler provides for a multi-user functionality by means of the well established server-client model. It runs in background and manages the distribution and execution of tasks centrally, exhausting the available computing units. Its core functionality is inherited from *GPUSched* [16], a shared CUDA-based library for load scheduling on *NVIDIA* GPUs. CUDA was originally chosen as it was and still is the only computing architecture that supports concurrent execution on GPUs. In theory, concurrency is also supported by OpenCL through the Device Fission Extension from version OpenCL 1.1 as well as the Device Partitioning feature introduced in version 1.2 of the standard[2], but in fact, only OpenCL implementations for CPUs support it up to this point in time. In this regard, we patched the open source graphics driver *RadeonSI Gallium* with the intention to provide this standard's feature in our scheduler. This is the reason why we discuss both closed source and open source drivers. While we use the closed source *AMD* graphics driver *fglrx* in the experimental part of our approach to evaluate the scheduling model, the open source driver *Mesa* is used in the context of improving the OpenCL support.

A further possible candidate for an accelerator-based heterogeneous cluster is the *IntelXeon Phi* accelerator with its *Many Integrated Cores* (MIC) architecture. Compared to modern multi-core CPUs, MIC provides a large number of x86-CPU cores supporting the concurrent execution of multiple threads on the basis of common programming models [9]. *Intel* extensions like AVX and SSE were omitted in MIC to save space and power to the favor of similar SIMD,

[1] http://www.top500.org/lists/2013/11/

[2] `www.khronos.org/assets/uploads/developers/library/2011_GDC_OpenCL/AMD-OpenCL-Device-Fission_GDC-Mar11.pdf`

which makes the new accelerator rather similar to GPUs than CPUs in this regard. In the context of this study, we perform a few tests on the new *Intel* architecture, in accordance with the aim of further developing our approach to meet its special requirements in future work.

1.1 Motivation

This is the first study to our knowledge, that investigates the possibility to integrate a scheduling process for different accelerators in a common operating system. Analogous to currently existing multithreading mechanisms for multi-core processors in traditional operating systems, similar mechanisms are required to perform complex tasks on available accelerators and treat with high levels of structural heterogeneity. A variety of multithreading applications are nowadays routinely used for achieving certain goals, however, system-wide automated mechanisms for processing such applications on accelerator-based systems are still lacking. We introduce an OpenCL-based scheduling framework, with which only basic skills with the OpenCL programming model would be sufficient to submit any given application for execution, sharing OpenCL processing devices with other users.

In a relevant context [17], we also introduced a static technique for predicting the suitability of a given computation task to be run on a selective device in a heterogeneous computing environment, based on code features extracted at compile time. Its major function is to achieve task-device matching in a system-wide view. By means of a machine-learning technique that uses a statistical model, performance predictions are systematically created. A combination of such a feature-based predictor with the load scheduler presented in this paper, would provide a comprehensive unit that runs in background and manages the distribution and execution of tasks centrally, exhausting the available computing units in any accelerator-based heterogeneous system. Further, due to the technical details defined in such a combination of procedures, simultaneous involvement of multiple OpenCL devices would be valuable for a more effective use of computing resources. Thus, this issue will represent a major part of the intended functionality of *OCLSched* in future developments.

A potential scenario will be the use of *OCLSched* (in connection with the static predictor) by several users over network accessing a workstation that includes GPUs and other coprocessors. Deploying *OCLSched* in such a scenario can be beneficial for programmatic access provider in insuring enhanced resource utilization and managing the execution of many tasks on accelerators-based machines simultaneously. Further, the use of such an autonomous scheduling unit that takes the responsibility of matching and distributing computation load among multiple devices might be advantageous for high performance machines' user, where a previous knowledge and likely understanding of the heterogeneity of accelerator architectures is not necessarily required.

1.2 Related Work

Many scheduling algorithms targeting a variety of GPU architectures have been proposed recently. To increase resource utilization, the so-called elastic kernels were presented in a scheduling mechanism [12], in which fine-grained control was allowed over the execution of CUDA kernels on GPUs. For the same purpose and in a similar study [1], the spatial partitioning of GPU applications was also suggested as an alternative through multitasking on GPUs. However, both approaches focused on concurrent execution on *NVIDIA*'s devices, and moreover, numerous restrictions were placed on written CUDA kernels in most cases. Similarly, a more recent work [13] proposed a hybrid MPI-CUDA preemption method for scheduling applications

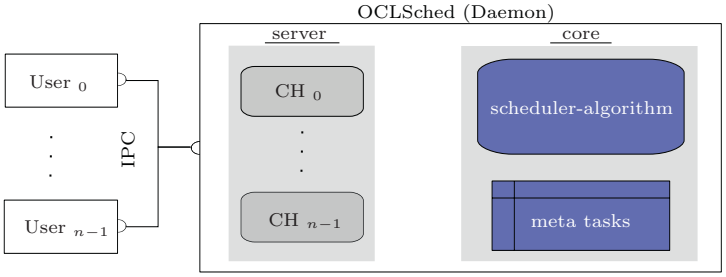


Figure 1: User-server interaction flowchart within the *OCLSched*'s multi-user environment.

on GPUs. Its goal was to allow an efficient scheduling of *entire* tasks on CUDA-supporting devices, using CPU-threads that perform GPU-related calls.

Also in the CPU-GPU heterogeneous computing era, many efforts have been taken to provide robust and efficient computing environments. For instance, *StarPu* [2] presented an execution model that unifies all computing units in a heterogeneous system. Using a co-scheduling strategy, it models operations and functions at run time to choose the execution on CPU, GPU or both of them. Similarly, resource sharing among OpenCL kernels was investigated through merging two kernels and running them by a special scheduler on a GPU [5]. In the developed scheduler, the focus was on kernel concurrency to improve the GPU throughput. In the same way, other studies have shown that under-utilization of GPU resources can be avoided by merging tasks statically before execution [6, 7]. But, launching big kernels suffers a lot on resources, especially registers and shared memory, as they need to be allocated for all (virtual) sub-kernels. Also barriers, which are permanently required by that concept are unfavorable. As a result, the concept did not scale well as the device was not designed for such kernels.

To the best of our knowledge, we present the first study that investigates the possibility to integrate a scheduling process for different accelerators in a common OS. In contrast to batch scheduling used by most of well-known cluster scheduler such as TORQUE[3], our preemptive scheduling method can utilize the computing resources of any existing OpenCL device as soon as they become available, even in case of tasks' load-imbalance [16]. Its major function is to manage the execution of multiple tasks on different OpenCL devices in a system-wide view. Further, it provides for a multi-user functionality by means of the well established server-client model. For achieving these goals, our study covers some kind of a combination of traditional CPU multithreading (POSIX threads and OpenMP) and GPGPU programming methods.

2 Methodology and Implementation

2.1 Design of *OCLSched*

Our approach provides an autonomous multi-tasking and multiprocessing environment for a system-wide view in a heterogeneous system. The computation workload of each newly issued task is scheduled on the execution units of the used device at run time. This is handled autonomously by a daemon, which can receive and manage tasks from multiple processes (users). The scheduling procedure is hidden from the user and all calls to the scheduler are executed transparently. Further, the user does not need to establish a context on an OpenCL device,

[3] www.adaptivecomputing.com/products/open-source/torque

since it is accomplished by the scheduler implicitly. *OCLSched* primarily manages the operative parts of each computing device. In the case of a GPU, these parts are the shader processors, the DMA engines and the execution queues, whose number varies according to the device model.

The core functionality of *OCLSched* is inherited from the CUDA-based scheduler *GPUSched* [16], which has shown that preemption and context funneling can increase the utilization and thus the performance of GPUs and hide the idle time of their resources. However, in contrast to the original work:

- First, the parallel computing architecture of the originally developed scheduler is changed from CUDA to OpenCL in order to exceed *NVIDIA*'s GPUs and support different OpenCL capable devices.
- Then, we redesigned the scheduler to meet the requirements of a client-server model (see fig. 1), supporting multiple users in a multitasking heterogeneous thread-safe environment.
- In the current state, *OCLSched* is implemented as a daemon that runs in background and manages the execution of tasks transparently, while the user can perform other common computations at the same time.
- We also extended the task-management functionality to support advanced features covering a wide range of processing policies.

In due consideration of the conceptual issues involved in achieving all objectives mentioned above, *OCLSched* can be easily applicable to a common OS. During the design stage, a number of disciplinary aspects that are necessary to enable the implementation of a daemon into the OS have been considered. Further, since *OCLSched* must run autonomously, it has been necessary during the implementation stage to lay down specific rules concerning system calls, input and output operations and process- and session management. One of the many advantages offered by this implementation is that, the user is not required to establish a context on any OpenCL device, as this is autonomously achieved by the scheduler. This will reduce the time to dispatch multiple kernels to the accelerator.

Towards a better integration of modern accelerators into the operating system, *OCLSched* allows the programmer to formulate different tasks, but then it takes over and manages the execution in a multi-tasking environment. In the sense of sharing the coprocessor efficiently, our scheduler pursues a higher utilization of processing resources inherently, manages the operative parts of the coprocessor, and improves the task parallelism by deploying a special task farm model [16]. Further, multiple users are able to submit their computations to the scheduler simultaneously. Each user can continue his main thread performing other tasks while the scheduler runs transparently.

OCLSched provides a `C++` API that relies on the OpenCL `C++` wrapper, through which the control flow can be described as follows:

1. The user encapsulates the assignments consisting of an unmodified OpenCL kernel and copy operations within a task.
2. The task is enqueued within the internal structure of *OCLSched*, in which lists of subtasks are generated for optimal and less costly scheduling.
3. *OCLSched* enqueues the prepared assignments through the accelerator's API on the device. The issuing order is such that the device is forced to adopt it.
4. The native device scheduler dispatches the operations on the actual compute units (CUs).

2.2 Scheduling Strategy

One of the main concepts of *OCLSched* is context funneling [19]. Managing all streams of tasks in form of subtasks' arrangements within one context has great advantages, these include basically the achievement of higher utilization of resources [16] and the realization of an intelligent multitasking and multiprocessing model. The complete execution time is divided into time slots, whose length may change in each execution cycle. The subtasks to be dispatched in the next time slice are planned in advance, whose length is determined by the duration of the subtasks. The next time slice is launched as soon as the previous operations have been terminated. In order to execute kernels' subtasks concurrently, the execution time of each kernel has to be known a priori. Therefore, they are estimated by the measurement of a dedicated execution of each kernel. This information is then used to fill the time slices with subtasks. However, we have shown in past work [16] that in the case of the unpredictability in terms of the execution time, the issue order in the **single** compute queue (in older GPU architectures) will be impaired. With modern accelerators, this problem is resolved by providing multiple compute queues, thus, the runtime unpredictability of some algorithms has less impact on the overall resource utilization. *OCLSched* also allows prioritization, thus, while planning the next time slice, the subtasks will be considered for scheduling in respect to their priorities. By means of a special combination of various lists, in which the tasks are stored depending on the current scheduling stage, the scheduler can exploit pipelining and reduce the costs.

Besides the innovative technical solutions used by *OCLSched* for achieving its objectives, it also provides for a high level of user convenience. Advanced task-management functionality is provided by our approach in its current state, this comprises among others:

- Memory allocation by chunks: Besides the number of bytes to be transferred to the device, an additional offset can be specified allowing to modify a specific part of the device memory buffer after each execution. Especially in the field of data mining, several applications can gain profit from this feature, when modifications of calculations' inputs are continuously required [15]. We have already conducted several experiments on well established data mining algorithms using *OCLSched*, running them in conjunction with other common computation tasks [8]. Such tests ensure the adaptivity of our approach to a wide range of real-world applications.

- Asynchronous operation: The user may issue several tasks to be processed on the accelerator by means of *OCLSched* and return immediately continuing the main context to perform other tasks. This point will be discussed in more details in section 3.

- It is possible to stop a running task: Use cases are for example, if the user aborts a running application, or if a heuristic for prefetch processing failed and the scheduled task became obsolete. In such cases, stopping the task before its termination allows the scheduler to free resources for other tasks in the queue. In contrast to native GPGPU programming, wasting resources might be avoided by means of our strategy in such use cases. The execution of a task can be paused and then pursued by means of special functions; `block()` and `unblock()`.

- Revoking resources allocated by tasks: This feature might be an advantageous for iterative algorithms as well as for cases, in which subsequent kernel invocations access the same data. For this purpose, the most *OCLSched* components including $cl :: programs$, $OCLSchedTasks$, device- and host buffers have been made reusable. Special remote calls are used to access such objects by the client, however, a more thorough explanation of this important design issue is provided in the coming sections.

2.3 Daemon

OCLSched is implemented as a daemon, approaching the objective of a standalone system-wide scheduling. Users communicate with the daemon in order to post requests and receive answers. In general, a daemon process performs routine tasks in background and listens for requests transmitted via an interface. Its autonomous design requires to obey specific rules and methods to detach its execution from its parent process [10]. In our case, the *OCLSched* daemon schedules user defined tasks to a device, which could be a CPU, a GPU, *Xeon Phi* or any OpenCL device. The requests of the users are passed on to *OCLSched* via shared memory.

2.4 Multi-User Environment

OCLSched should be accessible by independent *users* (processes or threads) at the same time. For this purpose, a facility to provide for different users' accesses is established in the scheduler. A local client-server model is applied as illustrated in fig. 1. When intending to submit a task for execution, the user deploys a client to communicate with the server, which is a part of *OCLSched*. The inter-process communication in our client-server model is realized via a combination of two different *IPC objects*; Message passing and shared memory.

Each user communicates via IPC with the daemon. The communication to each user is handled by private `ClientHandle`s (CH) within the server part of *OCLSched*. For every of n clients a `ClientHandle` serves as a communicator. The core functionality of *OCLSched* consists of the scheduling algorithm and the encapsulation of meta-data for each task. From the user's point of view, the execution of tasks submitted to the scheduler is performed transparently. While user's computations are running on the OpenCL device, the user can continue with its control thread (non-blocking).

2.4.1 IPC Objects

Inter-process communication can be implemented by deploying *IPC objects*, which are classified in the following categories: Message passing, shared memory and synchronization means. Message passing, in particular, employs operating system calls, these require to copy the message into a dedicated buffer. Some examples for message passing objects are pipes, FIFOs and message queues. Contrary to message passing, deploying shared memory for passing data between different processes does not imply system calls . Instead, a memory space is mapped page-wise by all affiliated processes into their own address space. In order to avoid conflicting accesses, explicit synchronization is required in this case.

OCLSched's multi-user model deploys *shared memory* and *message queues* merging the best of both worlds. On the one hand, tasks following the stream computing programming paradigm often have arguments and result structures, which are so large that the performance matters. Shared memory offers the best performance to share such large data sets between client and server. Message queues, on the other hand, are used to transfer small messages between client and server and to implicitly coordinate the accesses to the shared memory objects. In our model, we deploy shared memory for large data sets and message queues for small data sets and synchronization. In order to implement both IPC-objects, we use the platform-independent *Boost* inter-process library with its `C++` API [14].

2.4.2 Communication Modules

The communication interface is designed in hierarchically arranged modules. The symmetrical layout of these modules is illustrated in fig. 2. Remote accesses to tasks and the scheduler

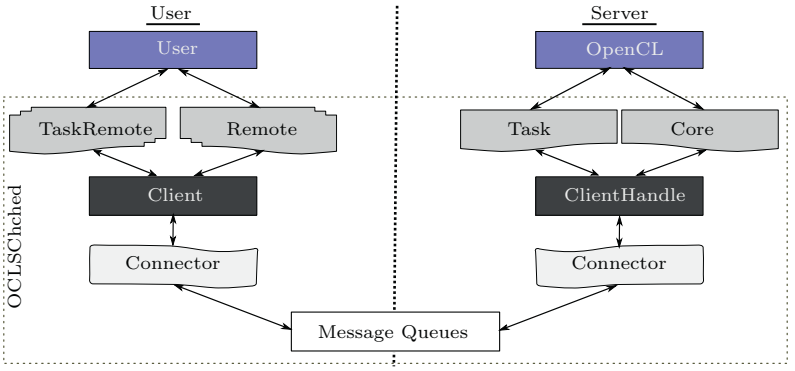


Figure 2: *OCLSched*'s communication-interface based on a client-server model. Remote procedure calls are based on message queues.

are encapsulated within the user process. The methods deployed by these modules provide for creating tasks and manipulating their scheduling. These modules in turn deploy a `Client` that is connected to its respective `ClientHandle` on the server side, whereby the connection takes place via message queues. The application of such structured modules supports the interchangeability of the deployed IPC objects (i. e. the communication protocol).

2.4.3 Communication Protocol

In *OCLSched*'s multi-user environment, conflicting accesses are prevented by strictly applying a communication protocol, when shared memory is used for the communication (in the case of large data sets). The used protocol defines the sequential order of messages between the user and *OCLSched*, and hence the messages coordinate the access to the shared memory. An example setup is shown in fig. 3.

In addition to the message queues, inter-process mutexes are used by the clients to wait for the termination of tasks. The task-scheduling procedure takes place asynchronously to the user, which results in the advantage for the user-thread of being not blocked. In order to minimize the probability of stalling, one client at most is processed in each iteration.

To sum up, tasks for the accelerator are passed on to the *OCLSched* server through message queues in a non-blocking fashion (from a user perspective). Potentially larger data volumes (e. g. kernel arguments) are put in shared memory for optimization purposes. Synchronization with the termination of tasks is done via special inter-process mutexes.

2.4.4 Temporal Execution

The task definitions and meta information are stored by *OCLSched* in so called *management data*. These data are used by the scheduling algorithm, the `ClientHandles` and the server. Running such procedures in separate threads would cause conflicting access attempts, which would result in stalling all but one accessor. Stalling the dispatch of tasks to the OpenCL device by *OCLSched* directly impairs the device utilization and thus the performance. To avoid that, the scheduling thread is given a permanent access to the management data. In addition, no other thread is created, this prevents locking the data for a unique or shared access by other threads. Instead, *OCLSched*'s only thread processes the server execution in the duration

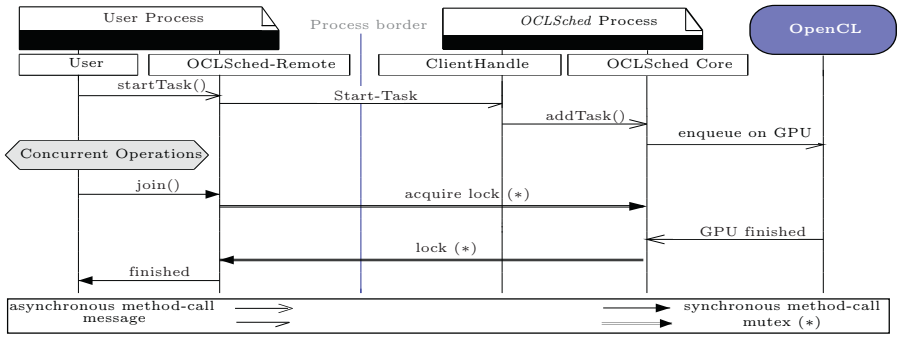


Figure 3: Inter-process communication model in *OCLSched* runtime.

between preparing the tasks for the next time slice and the termination of the previous one. As a result, the fixed sequence of procedures is of advantage for minimizing stalling periods.

The client-server interaction in our approach can be summarized as follows: A user employs a client to create and manage tasks for the OpenCL device, this might be encapsulated in a single thread using *OpenMP* (Open Multi-Processing) as an explicit programming model (as will be seen in section 3). Then, the client establishes a connection to the server and forwards the user requests via IPC. The server interprets and executes the requests as *remote procedure calls* to the *OCLSched* core. In analogy, the client-side representations of *OCLSched*'s core entities will be referenced as `remote`. In this way, it is possible to schedule multiple tasks within one accelerator context autonomously and without user's intervention. Several clients can be distributed on an arbitrary number of processes, all accessing one server and therefore sharing a common accelerator context. Thus, with at least the server and one or more user processes and threads, this model depicts a multitasking environment.

3 CPU-Assisted GPGPU

In case the user's calculations involve divisible CPU and GPU load, computations should be split between both devices, in particular, when code fractions with low compute memory ratio exist. Another reason would be the control flow divergence. Due to the SIMD character of GPUs, programs that diverge a lot gain more advantage on CPUs. In such scenarios, the user can exploit the benefits of multiple core machines by internally multithreading. A simple and flexible interface like *OpenMP* can be used to facilitate the encapsulation of different calculation parts within multiple threads. *OpenMP* is a well established API for parallelizing fractions of code with preprocessor directives [4].

Within the parts parallelized by means of *OpenMP*, the user can issue tasks with high compute-memory ratio to the accelerator using *OCLSched*. Listing 1 gives an example on how *OCLSched* and OpenMP can work together, in which the Euclidean distance is calculated between two n-dimensional vectors. In this example, the number of CPU cores is `maxT`=4 and every thread calculates a `part` of size $\frac{n}{\texttt{maxT}}$ (where $n = 1024$). Both vectors are filled with random values and the calculation is started asynchronously by means of *OCLSched*. While the calculation is running on the accelerator, the same calculation will be executed concurrently on

the CPU to serve as a comparative value. At the end of the calculation, OpenMP will reduce the partial sums, this is achieved by including a reduction inside the OpenMP `pragma` directive. In order to obtain the Euclidean distance, the square root has to be taken from the sum, which is simply a scalar operation omitted in this example.

```
# pragma omp parallel for reduction(+:deviceRes,hostRes)
for(cl_ushort i=0; i < maxT; ++i)
{
  EuclideanDistance::fillVector(&(vecA[i*part]),part);
  EuclideanDistance::fillVector(&(vecB[i*part]),part);
  OCLSchedRemote remote_scheduler;
  EuclideanDistance partialDist(&remote_scheduler,part,vecA,vecB);
  partialDist.startGPUTask(); //asynchronous
  hostRes += partialDist.hostResult(); //concurrent
  deviceRes += partialDist.deviceResult(); //blocks
}
```

Listing 1: Example of an OpenMP application using *OCLSched*.

Our initial results concerning the use of OpenMP with *OCLsched* are promising, and further possibilities in this regard are currently in development. This will be the focus of future work.

4 Improvements of OpenCL Support

The goals outlined in the design of *OCLSched* include: Ensuring an efficient deployment of all computing units in a versatile system, supporting a broad set of processing devices, and considering the characteristics of different real-world computation problems. Different vendors supply and maintain their own OpenCL drivers, which are mostly closed source. However, there exist an open source driver *Mesa* that supports the main GPU architecture used in this work with the *RadeonSI* driver. A special designed software stack is provided in its infrastructure providing developers with a complete suite of software tools which includes the OpenCL compiler, OpenCL run time and performance libraries for optimized algorithms. The open source graphics stack consists of multiple levels defining the interaction between the user and the kernel space in Linux operating systems. The *clover state tracker* that implements the OpenCL API represents an important core component in *Mesa* with regard to GPGPU computing. We patched the *clover state tracker* in order to support special features of *OCLSched* and fulfill the demands of many real-world applications. We also modified LLVM's R600 back-end[4] in order to support the missing features in the used open source driver. Patches that accomplished their goal (Double-precision Floating Point Support, Profiling Support) have been merged upstream in the corresponding implementation.

4.1 Double-precision Floating Point and Profiling Support

Double-precision instructions that are required by scientific applications were missing in the R600 back-end. We developed a patch[5] that was merged into LLVM supporting additional double-precision instructions for the *RadeonSI* (e. g. `V_SQRT_F64`, `V_SUB_F64`). Further, we

[4] lists.cs.uiuc.edu/pipermail/llvmdev/2012-March/048404.html
[5] lists.freedesktop.org/archives/mesa-dev/2013-July/041341.html

added the support for 64 bit floating point kernel arguments and constants, by splitting them into 32 bit floats.

Profiling is also an important component of the OpenCL API for evaluation purposes (e. g. `Event::getProfilingInfo<T>()`). Originally, its implementation was missing in *Mesa*'s Clover, therefore, we added the support[6] for time-stamp queries in the *clover state tracker* of *Mesa*.

4.2 Multiprocessing on GPUs

Overlapping computation and memory operations when using GPUs lead to better utilization of resources. Additionally, overlapping computation operations provide further benefits enabling thread-level parallelism. This multiprocessing feature is implemented using OpenCL's *Device Partitioning*, which is only supported by CPUs and Cell devices currently. OpenCL's *Device Partitioning* feature allows to divide one computing device into multiple sub-devices. Each of the sub-devices is responsible for the execution of a sequence of commands and runs asynchronously to other sub-devices. In order to enable the multiprocessing feature in our scheduler also for GPUs, we modified the driver infrastructure so that sub-devices could be created and used independently for the execution.

In the well established design of the *AMD*'s GCN (Graphics Core Next) architecture (Southern Islands), two *Asynchronous Compute Engine*s (ACEs) are controlling the access to the hardware, scheduling tasks and generating compute task graphs. Our implementation sets registers to configure the mapping of compute units (CUs) to an ACE. The hereby created sub-devices are then filled by *OCLSched*, whereas the tasks are distributed dynamically. The newer version (Sea Islands) provides up to 8 ACEs supporting better scheduling possibilities and thus a higher level of multi-processing on GPUs. Currently however, all compute commands are submitted to the graphics ring[7]. The reason for this workaround is that when the clover state tracker flushes its queues, everything is submitted to the graphics ring. The desired functionality will be achievable as soon as this problem has been fixed, we let this for future work.

5 Experimental Evaluation

A major goal of our approach has been the integration of an intelligent scheduler in a common OS, with the purpose of managing the execution of common applications on existing accelerators autonomously. To demonstrate this capability in purposeful tests, we use a variety of well established OpenCL applications, which implement common algorithms from mathematics and physics (as listed in table 1).

In this paper, the main part of our tests–the first experiment–is carried out on FirePro S7000, which is based on the *AMD* GCN[8] architecture and consists of 20 compute units (64 processing elements for each) and 4 GB global memory. The card is installed and connected through a PCI Express 3.0 to a quad-core *Intel*CPU (i5-3550). The other devices used in the second experiment are all listed in table 2, whereby the device-system affiliation of each test platform is included in the first column.

Unavoidable costs of the scheduling method must be expected, especially when considering the unavailability of concurrency on GPUs. Thus, we focus in the fist experiment on measuring the overhead caused by our scheduling scheme and comparing it against the native OpenCL

[6] lists.freedesktop.org/archives/mesa-dev/2013-August/043003.html
[7] lists.freedesktop.org/archives/mesa-commit/2013-August/044780.html
[8] `developer.amd.com/wordpress/media/2013/06/2620_final.pdf`

Application	Dim	*sizeA*	*sizeB*	*sizeC*	*sizeD*	*sizeE*
Matrix-Matrix mult.	2	256	512	1024	2048	4096
Matrix-Vector mult.	2	512	1024	2048	4096	8192
Mandelbrot Set	2	256	512	1024	2048	4096
Laplace	2	64	128	256	512	1024
Convolution	1	1280	2560	5120	10240	20480
Electrical Field	3	4	8	16	32	64
MergeSort	1	4096	8192	16384	32768	65536
Euclidean Distance	2	512	1024	2048	4096	8192
N-body	1	10	100	1000	10000	100000

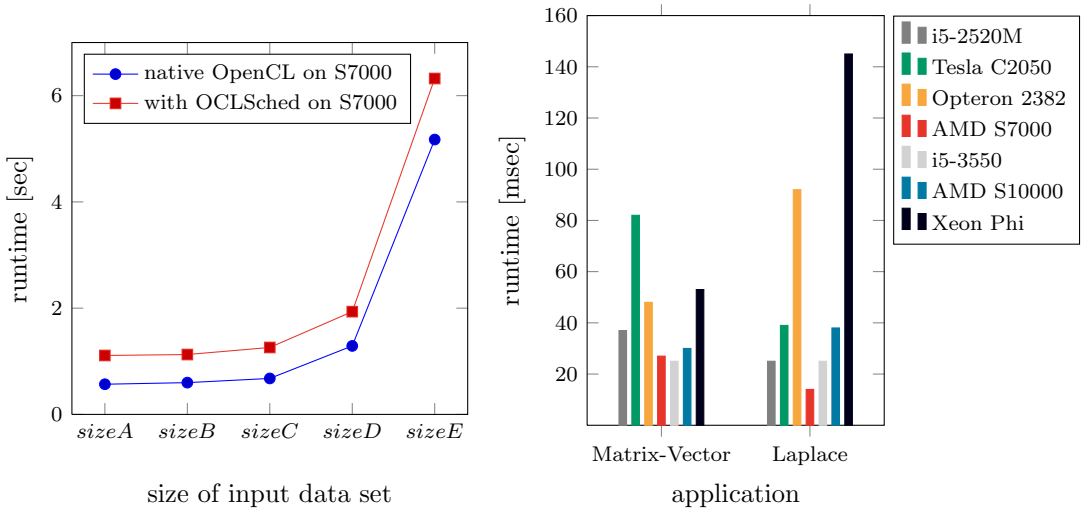
Table 1: *List of OpenCL benchmarks used for experiments on* OCLSched. *The input size of each application is represented as a number of double floating point elements in each dimension.*

Computer	**Device / Host**	**CUs**	**Device**	**CUs**
System 1	*Intel* Core i5-2520M CPU	4	–	–
System 2	*AMD* Opteron 2382	8	*NVIDIA* Tesla C2050	14
System 3	*Intel* Core i5-3550 CPU	4	*AMD* FirePro S7000	20
System 4	*Intel* Xeon E5-2650	32	*Intel* Xeon Phi 5110P	236
System 5	*Intel* Xeon E5-2650	32	*AMD* FirePro S10000	28

Table 2: *Device equipment of test platforms. The scheduler was hosted on each computer's CPU listed here during the experiments. CU: number of compute units.*

implementation. In this experiment, we consider all applications that are listed in table 1. The run times of a variety of applications submitted to execution by means of *OCLSched* are depicted in fig. 4a. Multiple users were created during this experiment in order to verify the *OCLSched*'s capability to handle computations by different users sharing a single GPU. The x-axis marks the different sizes of applications used in this experiment. Since the input size has a great influence on the run time behavior in general, we vary the input size in each step of the evaluation. The used input sizes of the different test computations are also illustrated in table 1. With just a single queue (no kernel concurrency), the scheduler enables only multitasking and asynchronous memory transfers. As shown in the run-time diagram, the overhead of our scheduler remains constant independent of the input size.

During the second experiment carried out on a variety of devices, we want to proof the general concept of our approach on a selection of devices. We are measuring the turnaround time within *OCLSched*, when executing a single kernel submitted by a single user each time. Due to its objective, two simple applications are used in this experiment: Matrix-Vector multiplication and Laplace. As depicted in fig. 4b, this experiment clearly shows that our approach is applicable to the most processing devices that support OpenCL. However, a performance comparison of the different platforms has not been an objective of this experiment, due to many factors including the exclusive access to the platforms and the inconsistencies of the different OpenCL drivers. Especially when considering the performance of *AMD S*10000, an exclusive access to the machine would be required if a fair and reliable comparison with other platforms is desired. However, it must be noted that except system 5, the test platforms were reserved exclusively for our tests.



(a) multiple applications (b) different devices

Figure 4: Performance comparison of multiple applications (as listed in table 1) when using *OCLSched*. All computations except Electrical Field are performed on double precision floats. (*a*) is performed on the AMD GPU S7000, while applications sets have been issued by multiple users. This depicts a comparison of native OpenCL versus *OCLSched*. (*b*) is performed on different platforms and shows the execution time measured when running a single kernel from a single user by means of *OCLSched* each time, whereby the input size: Matrix-Vector= 2048, Laplace = 1024×1024. (*b*) **verifies** the applicability of *OCLSched* to the multiple OpenCL processing devices.

Since Opteron 2382 belongs to a relatively older CPU generation, its relatively long run times in comparison to the other devices is not surprising. Also, *Xeon Phi* exhibits relatively long execution times, particularly in the case of Laplace. The cause is assumed to be the OpenCL driver which is still in the early stages of development and not specially designed for this architecture. Instead, *Intel* propagates its language extension for offload (LEO)[9]. Among the considered GPUs, Tesla C2050 shows relatively low performance, in particular when processing the matrix-vector multiplication (low computation-communication ratio).

In course of our experiments, inconsistencies in the behavior of different OpenCL drivers and devices caused troubles. This and other developments with *OCLSched* have experienced that the particularities of the manufacturers' drivers need to be studied in-depth. Not only to achieve a good performance but also in order to prevent some drivers to crash. Especially when it comes to the support of device partitioning, future work may analyze the architectures in order to boost *OCLSched*'s performance on those devices. An extensive study about the run-time behavior of different modern accelerators using OpenCL is presented in a related context [18]. The run-time analysis of several OpenCL devices might help us to extend our knowledge concerning the different devices' characteristics and to further develop our scheduling approach, in accordance with the different particularities of modern accelerators and coprocessors.

[9]software.intel.com/sites/default/files/article/326701/heterogeneous-programming-model.pdf

6 Conclusion and Future Work

Due to the currently increasing heterogeneity in modern systems, the support of multiple computing devices is becoming more attractive for many researchers and programmers in the field of high performance computing. For this reason, we developed *OCLSched*, an OpenCL-based scheduler for heterogeneous parallel systems. By means of *OCLSched*, computation tasks generated by different users can be executed on different devices in a multi-threaded client-server processing environment. The benefits of such a system-wide scheduling process that is easily integrated in a common OS, can reduce power consumption and leverage the combined capabilities of multi-core CPUs, many-core GPUs and other accelerators.

OCLSched's functionality has been successfully verified on CPUs and GPUs and evaluated by numerous tests when multiple users dispatched OpenCL applications for execution at the same time. Also, initial tests have been introduced comparing the *Intel*Xeon Phi accelerator with the other devices when deploying *OCLSched*. In future work, we plan to develop and optimize our scheduler to execute tasks on different coprocessors (including *Xeon Phi*) simultaneously. This would open a new horizon in involving accelerators and co-processors in high performance machines, as well as modern desktop computers. We believe that building a base for executing programs on heterogeneous devices autonomously could create an evolutionary path for the deployment of accelerators in the field of general purpose computing.

6.1 Acknowledgments

We thank Tom Stellard from AMD Corporation for constructive feedback on the extension of the open source AMD driver's functionality.

*Bibliography

[1] Jacob T. Adriaens, Katherine Compton, Nam Sung Kim, and Michael J. Schulte. The case for GPGPU spatial multitasking. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1–12, 2012.

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

[3] Nathan Brookwood. AMD Fusion Family of APUs: Enabling a Superior, Immersive PC Experience. Technical report, AMD Corp., March 2010.

[4] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, 2007.

[5] Chris Gregg, Jonathan Dorn, Kim Hazelwood, and Kevin Skadron. Fine-grained resource sharing for concurrent GPGPU kernels. In *4th USENIX conference on Hot Topics in Parallelism*, pages 10–10. USENIX Association, 2012.

[6] Dominik Grewe and Michael F.P. O'Boyle. A static task partitioning approach for heterogeneous systems using OpenCL. In *20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software*, number 20 in CC'11/ETAPS'11, pages 286–305, 2011.

[7] Marisabel Guevara, Chris Gregg, Kim Hazelwood, and Kevin Skadron. Enabling Task Parallelism in the CUDA Scheduler. In *Workshop on Programming Models for Emerging Architectures*, PMEA'09, pages 69–76, 2009.

[8] Marwan Hassani, Ayman Tarakji, Lyubomir Georgiev, and Thomas Seidl. Parallel implementation of a density-based stream clustering algorithm over a gpu scheduling system. In *18th Pacific-Asia Conference, PAKDD 2014, Tainan, Taiwan, May 13-16, 2014*, 2014. (to appear).

[9] Jim Jeffers and James Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. Elsevier / Morgan Kaufmann, 2013.

[10] Dave Lennert. How To Write a UNIX Daemon. Technical report, Hewlett-Packard Company. Available: `http://cjh.polyplex.org/software/daemon.pdf`.

[11] Aaftab Munshi, Benedict R. Gaster, Timothy G. Mattson, James Fung, and Dan Ginsburg. *OpenCL Programming Guide*. Addison-Wesley Pearson Education, 2011.

[12] Sreepathi Pai, Matthew J. Thazhuthaveetil, and R. Govindarajan. Improving GPGPU Concurrency with Elastic Kernels. In *Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 407–418. ACM, 2013.

[13] Kittisak Sajjapongse, Xiang Wang, and Michela Becchi. A Preemption-based Runtime to Efficiently Schedule Multi-process Applications on Heterogeneous Clusters with GPUs. In *22Nd International Symposium on High-performance Parallel and Distributed Computing*, number 12 in HPDC '13, pages 179–190. ACM, 2013.

[14] Boris Schaeling. The Boost C++ Libraries. *XML Press*, 2011.

[15] Ayman Tarakji, Marwan Hassani, Stefan Lankes, and Thomas Seidl. Using a Multitasking GPU Environment for Content-Based Similarity Measures of Big Data. In *Computational Science and Its Applications*, ICCSA'13, pages 181–196. Springer Berlin Heidelberg, 2013.

[16] Ayman Tarakji, Maximilian Marx, and Stefan Lankes. The Development of a Scheduling System *GPUSched* for Graphics Processing Units. In *International Conference on High Performance Computing Simulation*, (HPCS'13), pages 566–57. ACM / IEEE, 2013.

[17] Ayman Tarakji, Niels Ole Salsceider, Stephan Alt, and Jan Heiducoff. Feature-based Device Selection in Heterogeneous Systems. In *Proceedings of the ACM International Conference on Computing Frontiers*, (CF '14). ACM, 2014. (to appear).

[18] Ayman Tarakji and Niels Ole Salscheider. Runtime Behavior Comparison of Modern Accelerators and Coprocessors. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium Workshops*, HCW 2014. IEEE Computer Society Press, 2014. (to appear).

[19] Lingyuan Wang. Towards efficient gpu sharing on multicore processors. *PMBS11/The 2nd International Workshop on Performance Modeling*, 6:1, November 2011.