Efficient Resource Scheduling for Big Data Processing on Accelerator-based Heterogeneous Systems

Ayman Tarakji, David Hebbeker and Lyubomir Georgiev

Faculty of Electrical Engineering and Information Technology, RWTH Aachen University
ayman.tarakji@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. In this paper, we examine the heterogeneity in modern computing systems, particularly, how to achieve a good level of resource utilization and fairness, when multiple tasks with different load and computation ratios are processed. First, we present OCLSched, an OpenCL-based scheduler that is designed as a multi-user computing environment to make use of the full potential of available resources in heterogeneous compute systems. Multiple tasks can be issued by means of a C++ API that relies on the OpenCL C++ wrapper. At this point, our scheduler 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. Then, we demonstrate the ability of our scheduling approach to manage the nontrivial applications, bringing the ability of OCLSched to handle complex applications to the front. Besides the general computation tasks used in our experiments, we want to present a new implementation concept of a recent stream data clustering algorithm DenStream. Based on a specially designed task concept, the clustering functionality is described as a single user process within our scheduler, achieving its defined targets asynchronous to other general purpose computations. Besides the different CPUs, a variety of modern GPU and other accelerator architectures are used in this work including: AMD’s Graphics Core Next, NVIDIA’s Kepler, and the Intel MIC (Many Integrated Core) architecture.

Keywords: GPGPU, OpenCL, Heterogeneous Parallel Systems, Scheduling, Client-server Model

1 Introduction

In the recent years, general purpose accelerators are increasing in importance in the field of high performance computing. Examples of these are the AMD’s FirePro Server series, NVIDIA’s Tesla series and Intel’s Xeon Phi. Introduced in the Top500 list for supercomputers [1], the fastest computer of the June 2014 is called tianhe-2 featuring 48000 Xeon Phi devices to generate up to 54,902.4 TFLOP/s. Other winner systems are even deploying more heterogeneous devices, for instance, QUARTETTO ranks 37th on the June 2014 Top 500 list and features a mix of GPUs and Xeon Phi 5110P. With the rise of such accelerators, new research fields open up. For once, it becomes a challenge to write fast and portable code for such platforms. To enable that, unified computing models like CUDA and OpenCL were launched the last few years. While CUDA was developed by NVIDIA to be implemented on own platforms, OpenCL is a standard developed by the Khronos group and has the advantage of being mostly platform-independent. Both programming frameworks offer a higher abstraction layer to the programmer, while writing highly parallel code for accelerators without requiring detailed knowledge about the hardware or expert training.
However, with the increased use of such devices, utilization as a main problem was indicated in the related work [2] [11] [22]. For this reason, 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 [21]. 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).

The main goal of this paper is to provide a scheduling mechanism for common computation tasks, which facilitates the access to such devices in a heterogeneous environment and manages the processing of multiple tasks in an organized fashion. A further point of interest is represented in improving the performance of a very modern data mining application, while running with other general tasks in a multi-tasking accelerator-based environment. We propose a new implementation concept to run the data stream clustering application DenStream [6] to observe the behaviour of OCLSched when dealing with such complex applications.

OCLSched 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 [30], 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 [10], 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 [17]. The in-order superscalar architecture introduces an updated version of the Intel’s microarchitecture P5 achieving architectural simplicity as an advantage over modern CPUs, and comprising significant enhancements over P5. Major enhancements can be drawn in the 64-bit instruction support, vector capabilities, the support of four-threads-per-core and power managements. Intel extensions like AVX and SSE were omitted in MIC to save space and power to the favor of similar SIMD, which makes the new accelerator rather similar to GPUs than CPUs in this regard. But contrary to GPUs, the intermixing of vectorization on the one hand and the autonomous cores on the other allow software developers to profit extensively from outsourcing computation problems to the coprocessor, without the necessity to change the programming paradigm. In this context, an extensive study about the run-time behavior of different modern accelerators using OpenCL was presented earlier [31]. The run-time analysis of several OpenCL devices helped 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. During the experimental part, we perform a few tests on the new Intel architecture, with the aim of further developing our approach to meet its special requirements in future work.

1.1 Why OpenCL?

A variety of programing frameworks have been recently released, by means of which software developers can design and implement their applications on high performance accelerators. The most well-established frameworks in this context seem to be OpenCL (Open Computing Language), OpenACC (for Open Accelerators), OpenMP (Open Multi-Processing) and CUDA (Compute Unified Device Architecture). In terms of perfor-
mance, Karimi et al. [18] argued that CUDA can be more efficient on NVIDIA devices. However, in another research [19] it was found that the performance gap between CUDA and OpenCL can be overcome by performing special purpose and hardware-specific optimizations on OpenCL programs. Furthermore, Fang et al. [8] argued that there is no reason for OpenCL to obtain worse performance than CUDA under a fair comparison. Also, a performance comparison with OpenMP has shown that higher performances could be achieved on CPUs and accelerators using OpenCL [33].

Also OpenACC, the industry standard and the directive-based accelerator programing framework, is competitive in this field. It is orthogonal to and interoperable with MPI (Message Passing Interface) and OpenMP. Exactly for this reason, we believe that OpenACC meets the criteria for long-term success. However, Wienke et al. [35] presented a direct comparison between OpenACC and OpenCL, in which the focus was on evaluating both performance and programmability, based on two real-world applications. It was shown that the OpenACC performance is only approximately 40% of the best-effort OpenCL for a complex medical program. They assumed that the loss of OpenACC performance is mainly due to the lack of the ability to leverage the local memory of the GPU.

Besides the performance issue, the advantage of OpenCL is represented in its portability [23], whereby a fair and reasonable comparison among different devices becomes realizable. OpenMP is portable as well, but it does not offer the low level access that OpenCL provides [4], which is required to achieve such an in-depth analysis tracked in this work. To be successful, an accelerator programing model must be low-level enough to express potential parallelism in existing sequential code, but high-level and portable enough to allow efficient implementation-mappings to a variety of accelerators. For these reasons, we believe that OpenCL offers a promising programing framework for future applications, when it comes to heterogeneous computing.

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 [22], in which fine-grained control was allowed over the execution of CUDA kernels on GPUs. For the same purpose and in a similar study [2], 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 [24] proposed a hybrid MPI-CUDA preemption method for scheduling applications 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 [3] 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 [11]. 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 [12, 13]. 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\(^1\), 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 [30]. Its major function is to manage the execution of multiple tasks on different OpenCL devices.

\(^1\)www.adaptivecomputing.com/products/open-source/torque
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 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, 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 presented earlier [30], which has shown that pre-emption and context funnelling 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 on the CPU and manages the execution of tasks transparently.
- We also extended the task-management functionality to support advanced features covering a wide range of processing policies.

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 [20]. 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. 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 [30]. 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).

2.1 Scheduling Strategy

One of the main concepts of OCLSched is context funneling [34]. The management of all task streams in form of subtasks’ arrangements within a single context has great advantages, these include the achievement of higher resource utilization [30] and the realization of intelligent multitasking and concurrency. 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, in which a single work-group is run according to the given execution configuration (defined by the user). This information is then used to fill the time slices with subtasks. However, we have shown in past work [30] 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.

The scheduling procedure can be sketched in an example setup as follows: Tasks issued by a user are stored in the blocking_list and completed_list; ready tasks are listed in dependency of their priority.
in one of the task_lists. In each iteration of the scheduler, a time slice is planned. Therefore, suitable subtasks will be listed in the _computeList, _H2DCopyList or _D2HCopyList respectively. These subtasks are then distributed among the corresponding queues, which exist in the number of CU partitions / DMA engines available on the hardware. Each queue is in turn mapped to a sub-device. OpenCL dispatches copy instructions to the available DMAs and kernels through the computation queue (or rather the multiple computation queues in newer GPUs) to the available CUs. However, this is currently not achievable, since OpenCL still not provides for Device Partitioning on GPUs at the time of writing this paper. Thus, the multiple queues of subtasks are all mapped to the same computation queue of the hardware.

2.2 Advanced Task Management

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 [29]. We have already conducted several experiments on well established data mining algorithms using OCLSched, running them in conjunction with other common computation tasks [15]. 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.

- 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.

- Reuse of tasks: A terminated or stopped task can be reused by means of reset().

2.3 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 ClientHandles (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). This concept implies a multi-threading environment using proper IPC
mechanisms, in which atomic operations are required to prevent race conditions within the IPC. Data races and conflicting accesses will be inhibited.

2.3.1 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. Some examples for message passing objects are pipes, FIFOs and message queues. However, message passing causes system calls, which might lead to performance degradation. Contrary to it, using shared memory to pass data to other 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 [26].

2.3.2 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 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).

2.3.3 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. The user has the possibility to dispatch a task to the scheduler and continue with its own control path, as long as it does not have to synchronize with the task’s termination.

2.3.4 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 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.

2.3.5 Accelerator Stalling

The execution order by means of OCLSched is exemplified in fig. [Fig. 3]. In general, stalling reduces utilization and has to be avoided. In each iteration the OCLSched core needs to synchronize with the enqueued operations of the accelerator time slice. The accelerator operation is exemplarily filled with subtasks to read and write buffers and to run an OpenCL kernel. Between different time slices, the device stalls. During a stalling period the device is not used, but rather it waits for instructions. The utilization $u$ of the device can be described as follows:

$$u \propto \frac{t_{\text{slice}}}{t_{\text{slice}} + t_{\text{stall}}}$$  \hspace{1cm} (1)

However, stalling can not be completely avoided, since the enqueuing of subtasks for a time slice is started only when the previous one terminates. The length of a time slice depends on the maximum execution time
Figure 4: Simplified sequences of scheduler (left) and GPU operations (right). The GPU has been taken as an example of a common accelerator. The rectangles with continuous borders represent time slices. Operations are included within each one and separated by dashed lines, which are executed from top to down. Arrows within an operation indicate that the operation lasts for a significant time. The arrows between the columns show dependencies between both threads. The left column represents the execution of the scheduler on CPU, the GPU operation is sketched on the right side.

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. 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 Density-based Stream Clustering By Means of OCLSched

With the emergence of modern stream computing architectures, the use of accelerators to enhance the performance and efficiency of parallelized data mining applications is becoming more interesting. As a further contribution of this paper, we propose a new implementation concept to run a very recent data stream clustering application DenStream [6] in a multi-tasking accelerator environment [28]. We focus on the cluster and outlier analysis as a typical data mining strategy [7]. Our approach assumes the density based cluster definition from the DBSCAN algorithm [25] (which is a part of DenStream), determining a cluster as a maximal and connected region of dense points. Within the execution model of OCLSched, the clustering functionality discussed in this paper is described as a single user process, achieving its defined targets asynchronous to other general purpose computations. In order to design and implement this strategy, the here presented OCLSched’s implementation of DenStream combines very recent techniques from both research fields: Heterogeneous parallel systems and data mining.

Our approach considers the DBSCAN as part of DenStream and focuses on solutions provided for complex computations to share the accelerator resources with other general computation tasks by means of OCLSched. The special feature of DBSCAN is represented in the way how the clusters are collected point by point, expanding the edge of each cluster and requiring layer-by-layer previews, which constitutes a limitation on parallelizing DBSCAN. Also included in the related work, two interesting publications [5, 32] handled this problem in two different ways. The first method was based on creating multiple instances of DBSCAN and summarize the results. The idea was presented in [5], using density chains defined as connected but not maximal dense regions. This method had relatively high memory consumption and used functions with divergence accelerator-program flow.

Our OCLSched’s implementation of DenStream follows the strategy from [52], running only the time-consuming part of DBSCAN on the accelerator. This part of computation includes the calculation of the distances between the data points, ensuring low divergence and excellent parallelism of the accelerator task. However, to the best of our knowledge, we present the first study that investigates a possible DenStream parallelization using GPU (as an example of a common accelerator), on the basis of a special scheduler that manages the execution of general-purpose tasks simultaneously. It has a similarity with the idea presented in a previous work [29], in combining techniques from both data mining and high performance computing fields. The focus of that work has been on a content based similarity algorithm SQFD (Signature Quadratic Form Distance), that was designed for an earlier CUDA-based version of our scheduler [30]. But now, we are considering a completely different and more recent data mining method, a more advanced version of our scheduler and also a different programming framework OpenCL, providing for more complex and heterogeneous computing environments.

3.1 DenStream

A data stream is a massive, continuous and rapid sequence of data elements without length limitation, containing \((d, t)\)-tuples where \(d\) is a data record, and \(t\) is a time-stamp. Each reading process of such a sequence is achieved through a linear scan. The data stream model is widely used for modeling tasks such as telephone records, financial transaction statistics and multimedia data lists. For the purpose of statistical data analysis, clustering is used in many fields and represents the process of grouping similar data objects into so-called clusters. Since a variety of clustering strategies exists currently [15, 28], there are different formal definitions of cluster depending on the desired clustering algorithm. DenStream is one of the data stream clustering approaches, as it was presented in [6].

3.2 Strategy of Clustering

DenStream uses micro clusters to form the final clustering of the data stream at time \(t\). Based on an exponentially fading function \(f(t) = 2^{-\lambda t}, \lambda > 0\), whereby \(\lambda\) is an input parameter and describes the density level of the clusters
(see Algorithm 1), the weight of old data records is reduced over time. When adjusting the value of $\lambda$, the importance of the historical data becomes lower for each higher $\lambda$ value. The micro clusters in DenStream (as shown in Fig. 5) build a summary representation of groups of close data points. This representation defines a spherical region in the data set with center $c$, radius $r < \epsilon$ and weight $w$. The micro clusters evolve with the time by the new data, changing $w$, $c$ and $r$. At time $t$, the weight of the micro cluster is defined as the sum over the weights of the included points. A threshold $\beta\cdot\text{minPts}$ separates the micro clusters into two classes (types):

- Micro clusters with $w \geq \beta\cdot\text{minPts}$ are potential micro clusters and are grouped to build the final clusters using DBSCAN.
- Micro clusters with $w < \beta\cdot\text{minPts}$ are outlier micro clusters, which are still not dense enough at the current point of time.

**Data:** $\text{DataStream}, \epsilon, \beta, \text{MinPts}, \lambda$

**Result:** Clustering

```latex
def merge(p):
    if $t \mod Tp == 0$ then
        Check all micro clusters and perform downgrade if necessary;
    end
    if clustering request then
        // offline phase
        Generate clusters from microclusters using DBSCAN;
    end
```

**Algorithm 1:** DenStream

A very important property of the micro clusters is that they can be maintained incrementally. Adding a new point to a micro cluster or fading another one by the fade function $f(\lambda)$ does not require re-computation of that micro cluster. Hence, the stream data is processed only once, extracting the required features without saving each point in the memory for further use.
While achieving the incrementally maintenance of the micro clusters, the weighted linear and the weighted squared sum of the points are stored in the micro cluster. For a micro cluster summarizing $n$ points $p_1, p_2, \ldots, p_n$, with time stamps $T_1, T_2, \ldots, T_n$ and considering the fade function $f(\lambda)$ for the points’ weights, the linear sum at time $t$ is $CF_1 = \sum_{i=1}^{n} f(t - T_i) p_i$. The weighted squared sum for the same micro cluster is $CF_2 = \sum_{i=1}^{n} f(t - T_i) p_i^2$.

Using $CF_1$, $CF_2$ and $w$ as features defining a micro cluster $mc = \{CF_1, CF_2, w\}$, a merge of a new point $p$ (considered in Algorithm 1) is realized modifying $mc$ to $mc = \{CF_1 + p, CF_2 + p^2, w + 1\}$. Fading out a micro cluster for an interval $\delta t$ is performed using the transformation: $mc = \{CF_1 2^{-\delta t}, CF_2 2^{-\lambda \delta t}, w 2^{-\lambda \delta t}\}$.

The radius and center of each micro cluster, which are used in the DenStream algorithm, will be computed each time from the features $CF_1$, $CF_2$ and $w$. For $mc = \{CF_1, CF_2, w\}$, it applies that $c = CF_1/w$ and $r$ is the maximal standard deviation of the data over all dimensions of the micro cluster. The outlier micro clusters save also their creation time $t_0$ as an additional feature.

The DenStream algorithm allows a micro cluster to change its type or fade completely out and disappear. The period $T_p = \lceil \frac{1}{\lambda} \log \left( \frac{\delta t}{\beta r - 1} \right) \rceil$ determines how often a micro cluster should be checked whether it has faded out being outlier. Thus, if the weight $w$ of a given outlier micro cluster is defined as: $w < \xi(t, t_0) = \frac{1}{2^{\lambda \delta t} - \lambda r^{-1}}$ at time $t \geq kT_p$ for $k \geq 1$, this micro cluster can be discarded.

### 3.3 Algorithm

The DenStream clustering algorithm basically consists of two major parts: Online part and offline part. The complete algorithm can be described in a series of steps as depicted in Fig. [1].

#### 3.3.1 Online Part

This part of the algorithm is responsible for the micro cluster maintenance. The maintenance includes: Merging new data, creating new micro clusters and fading existing micro clusters. For each new point arriving from the data stream, DenStream tries to merge to the nearest potential micro cluster without increasing its radius over the value of $\epsilon$, which is an input parameter of the algorithm (shown in Algorithm 1) and strongly associates with the density level of the micro cluster. If this is not possible, the algorithm tries to merge it to the nearest outlier micro cluster, considering the maximum radius constraint upper bounded by $\epsilon$. If this is also not possible, the new data point is considered as a center for a new outlier micro cluster. Before the algorithm can take the decision how to merge the new data to some existing micro cluster, or start a new one, the distance between the newly arriving data point and the center of each of the existing micro clusters must be computed. In the case of merge, the radius of the merged micro cluster must be re-computed. The online part is also responsible for fading out the old micro clusters, assuming that historical data is losing importance exponentially over time. During this phase, the micro clusters are checked for the minimum weight bound to keep them as potential or outlier micro clusters, which is considered as the grade down strategy of DenStream. According to this strategy, potential micro clusters are changed to outliers, and the outliers with weight that is lower than the threshold can be deleted.

#### 3.3.2 Offline Part

The offline part runs DBSCAN (Density-Based Spatial Clustering of Applications with Noise) over the currently existing micro clusters. In this work, we follow the definition of DBSCAN that was presented in [25]. DBSCAN is characterized by detecting the clusters in the data set as dense region of data points. It defines the neighborhood of some point $p$, from a data set $D$ as $Ne(p) = \{q \in D \mid dist(p, q) \leq \epsilon \}$ for a given distance function $dist$. The $\epsilon$ is the first input parameter of DBSCAN. The second, called MinPts, describes a number of points. The so-called core-object is the point $q$, where $|Ne(q)| \geq MinPts$. Hence, a point $p$ is directly density-reachable from a point $q$, if $p \in Ne(q)$ and $q$ is core-object. Furthermore, density reachability is defined as a relation of two points in $D$, connected by directly density-reachable chains of points, implicating that density reachable points belong to the same dense area, thus, building a cluster.
3.4 DenStream’s Task-Design for OCLSched

An efficient implementation of DenStream by means of OCLSched demands a well-thought deployment of its task-concept [27]. In this section, the main OCLSched tasks used for DenStream are discussed. A major goal of this work has been establishing a challenging test case of application for our scheduling strategy. Implementing such a complex application as DenStream with OCLSched allows to investigate the adaptivity of the scheduler’s task-concept to different computation patterns. For instance, since the DenStream algorithm needs a huge number of task starts, it has been necessary to improve the task-model of OCLSched during the development, completely modifying the task’s reset and reuse methods towards avoiding the recreation of tasks, in order to decrease the host-side time overhead. Further, motivated by the implementation of memory allocation by pages...
used in our DenStream implementation (as will be described later), additional offset were added to the copy tasks, allowing to copy a given data element to a specific place in the device memory buffer. Generally, the online part of DenStream is run for each new point arriving from the data stream. This implies three OCLSched tasks, which are responsible for merging the new data and actualizing the existing micro clusters. These tasks are designed in such a way that they could be alternatively combined and separately launched on different processing devices. These are represented as follows:

- **Candidate Task**: It supports the data aggregation decision. candidate collects the data related to the merge decision (potential micro cluster), which includes the distances between the new data point and the existing micro clusters’ centers. In this phase, as many instances of the distance computation kernel (threads) as available micro clusters will be issued to execution. For efficiency reasons, the functionality of this task (OpenCL kernel) is extended to compute also the new radius of a micro cluster caused by merging the new point. Thus, in this phase all the data required for the merge decision is delivered in form of two vectors of length \(n\), where \(n\) is the current number of the micro clusters.

- **Merge Task**: This task is responsible for merging the new data points to existing micro clusters and fading out the old micro clusters to give less importance to old data at an exponential rate. In practice, this kind of fading out (implicit fade out) is realized by multiplying the micro cluster’s features \(CF1\), \(CF2\) and \(w\) with the fade factor described in a previous section. For the micro cluster that will include the current point, the multiplication is executed whenever the point is merged to it. This ensures that the micro cluster fades out for the time period between the last and the current merge actions. However, only one micro cluster is updated with the currently managed point. The merge operation is characterized by a divergent program flow and a concurrent memory access. For this reason, the merge task will not be as optimally parallelized as the candidate task, as discussed later.

- **Fade Task**: This task ensures a proper fade out of each micro cluster at most after a period \(T_p\). The fade task also checks all the micro clusters for the minimum weight bound to keep them as potential or outlier micro clusters, following a special method of DenStream (called downgrade). According to this strategy, potential micro clusters change into outlier, and outliers whose weight is below the lower bound can be deleted. The fade task can be executed in parallel for all micro clusters by means of a single one dimensional OpenCL kernel (one thread for each micro cluster).

In contrast to the online part, a single OCLSched task called Intersects is responsible for the final clustering during the offline part of DenStream. This task launches a two dimensional OpenCL kernel and calculates the centres of the final micro clusters while concluding their overlapping areas. In this work, we adopt a slightly modified version of the DBSCAN implementation from [32] for the offline part of DenStream, to generate the final request output.

## 4 Implementation

GPGPU programming follows the stream computing paradigm, based on which massive sets of data are processed. Basically, there would exist three major problems in a native GPU implementation of DenStream: The high memory transfer volume that becomes a real bottleneck when it comes to GPUs [9], the high number of task starts and also the processing of only one point per iteration and time unit in the main program loop. Since only one point is merged per program loop, the application would need a high number of loops to manage the whole data stream. In particular, the achievement of several tasks for each iteration would cause high scheduling overhead. For an efficient implementation of DenStream by means of OCLSched, a completely new design of DenStream tasks is required, in order to face all the mentioned challenges.

### 4.1 Tasks Model

In the presented OCLSched’s implementation of DenStream, the data stream is divided into packages of points. One data-point package contains a number of individual data points. All points in a given data-point package
are assumed to arrive with the same time stamp. As a result, the time stamp is generated by the number of the packages being processed and not by the points’ number from the input.

We apply a kernel merge technique that is similar to the approach from [14], in order to reduce the number of task starts, joins and resets. In total, three OpenCL kernels are implemented, while no more than two of them are started in one loop iteration, processing one data-point package each time. Compared with a native GPU implementation, our strategy saves a significant number of task starts and thus a corresponding time overhead.

The online part of the OCLSched’s DenStream implementation applies a functional merge of tasks. Two tasks are prepared during the initialization phase and recreated only in case of memory buffer expansion on the device. In the first task, both the candidate and the merging parts of DenStream are combined in one OpenCL kernel, briefly called CM (Candidate and Merge). The second task is an extended version of CM, in which the fade routine for all micro clusters is also considered, briefly called FCM (Fade, Candidate and Merge). Thus, in total three tasks are deployed when considering the online and the offline part of DenStream. One of four task combinations is started in each loop:

C1 Start, join and reset the CM-task.
C2 Start, join and reset the FCM-task.
C3 Start, join and reset the CM-task, followed by start, join and reset of the offline task.
C4 Start, join and reset the FCM-task, followed by start, join and reset of the offline task.

First of all, the host program prepares a data-point package in a host buffer and decides which task combination to start. The decision about which task combination should be started is taken on the basis of the request period Rp and the size of package PpS that is defined in the configuration file. On each Rp data-point package, a request has to be started. Finally, the fade procedure is applied according to the DenStream parameter $\lambda$, thus affecting $T_p$.

The CM kernel consists of two main parts: The candidate part and the merge part. The candidate procedure executes in an optimally efficient parallel way, as can be seen from the code snippet Listing 1. However, when adding the merge part to the kernel, only one thread will perform the merge procedure. The FCM kernel works on the same way, executing also the fade for all micro clusters in parallel, before the single thread executes the sequential merge. This kernel part achieves the calculation of the distances, centres and also the fade process for each micro cluster. The other kernel part is responsible for the merge procedure, it implements the sequential program flow described in the original DenStream (see Sec. 3.1). The candidate part of the kernel should be completed for all threads, before starting with the scan and merge procedures, which are also part of the same kernel. For this purpose, a counting semaphore (depends on the number of running threads) is applied at the point between the candidate computation and the scan of results.

Listing 1: The candidate part of the FCM kernel

```c
private size_t Id = get_global_id(0);
private float distance = 0;
private float radius = 0;

for (private size_t i = 0; i < dim; i++){
    CF1[(dim)*Id+i] *= fade;
    CF2[(dim)*Id+i] *= fade;
    distance += pow(point - CF... ,2);
    radius = max(radius, sqrt(CF...));
}
```

~ 15 ~
Both CM and FCM tasks first of all transfer a set of points $P$ that are wrapped as a data-point package ($|P| = k$) to the device memory. Then, either CM or FCM is launched for $k$ iterations, each iteration processes one point of the data-point package. With OCLSched, a specialized iteration parameter can be passed to start a number of compute iterations, instead of recalling the same task many times. By means of the iteration parameter, the same kernel is launched for all threads, considering the data dimensionality $dim$ (a constant parameter) and accessing $point(\text{iteration} \times (\ast dim) + i), \forall i \in [0 \cdots \ast dim - 1]$, where the point array is filled like $point = \{ p_1^T, p_2^T \cdots p_k^T \}$.

Finally, a copy task returns the free memory space on the device after finishing the merge procedure. This information is used to decide if a memory expansion is required at this point. A possible optimization in this context would be to process all points in the data-point package in parallel. However, a parallel merge would change the logic of the DenStream algorithm and also cause concurrent access to the data structures holding the micro clusters data.

### 4.2 Memory Allocation and Transfer

The stream data is transferred to the accelerator by building data-point packages. The memory transfer from the device to host includes the micro cluster intersections matrix. In order to decrease the amount of memory transfers between the host and the device, the micro-cluster data is completely stored on the device memory. Features of a given micro cluster are stored in individual memory buffers. For $n$ micro clusters with dimensionality $d$, the scalar valued attributes $w_i$ and the timestamps $t_i$ are stored in the same order of the micro clusters. The vector features $CF_1$ and $CF_2$ follow the same logic.

On the host side, the buffers are organized in pages. All host buffers consist of $p \geq 1$ pages of size $k > 1$, where $k$ is the number of features in the page. One page set is the set of pages from different memory buffers, holding the micro cluster’s features for a given micro cluster. A page set consists of five pages from the five different memory buffers, as illustrated in Fig. 7. All host buffers are placed in the shared host memory and managed by OCLSched. Contrary, the device buffers are directly allocated with size: $p.sizeof(page)$ (the summed up size of the allocated memory on the host side). When memory expansion is necessary, the device buffers are freed and reallocated with bigger size copying the complete micro clusters’ data from the host to the device.
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).

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

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.

5.1 Experiment Part I

5.1.1 Hardware Environment

In this paper, the main part of our tests is carried out on FirePro S7000, which is based on the AMD GCN 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). However, we also deploy other devices in our experiments as listed in table 2 whereby the device-system affiliation of each test platform is included in the first column.

5.1.2 Comparison with Native OpenCL

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

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

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.
Figure 8: Performance comparison of multiple applications (as listed in table I) 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.

we consider all applications that are listed in table I. The run times of a variety of applications submitted to execution by means of OCLSched are depicted in fig. 8a. 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 I. 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, as long as its total value fits in the available memory. Concerning very large problem sizes (sizeE), there is still a room for improvement of our strategy, especially when concurrent execution becomes available on current GPUs.

5.1.3 Heterogeneous Computing Platforms

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. 8b, 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 S10000, 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.

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). Among the considered GPUs, Tesla C2050 shows relatively low performance, in particular when processing the matrix-vector multiplication (low computation-communication ratio). A better and deeper insight into the scheduling procedure of OCLSched will be given in a further experiment.

5.1.4 Procedure Phases

In this test, only one user executes the matrix-vector multiplication using OCLSched. Apart from generating input data, the preparation which is necessary to run the tasks, consists of the following: Connecting to OCLSched, creating the buffers and tasks and compiling the kernel. The total time therefore \( t'_p \) occupied 2% of the total execution time \( T \). Only in 5% of that fraction \( t'_p \) the user process was actually running (CPU-time). The remaining time the user waited for the scheduler was basically caused by the kernel compilation (85% of \( t'_p \)). By means of the AMD accelerated parallel processing (APP) profiler, We could observe that compilation time on server-side (within OCLSched’s process) was equally long. Further, the overhead of the communication between client and server through OCLSched has been measured against small requests to the server, this was restricted on 200 \( \mu s \) (0.2% of \( T \)).

Since OCLSched is not running within the user process, this enables the user to perform other tasks at the same time independently from the scheduler. Also the idle time while waiting for the kernel compilation can be masked simply by encapsulating the calls to the OCLSched-server in a separate thread. For this purpose, traditional multithreading techniques for CPUs (e.g. C++ standard thread support library STD TSL [16]) can be used in the user space. Considering the whole procedure duration, the user was idle 99.9% of the time \( T \) in total. Using Callgrind (a tool for measuring the number of instructions made by function calls) for the analysis of the computation time, we observed that only 0.5% of the whole performed instructions were required for communication between the user and OCLSched. Further, Callgrind unveiled that only 2.7% of the computation time was spent by the front end of OCLSched, which provides for the multi-user client-server environment.

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
Figure 10: The execution time measured when running a single kernel from a single user by means of OCLSched each time. The execution is performed on different platforms. Input size: matrix-vector mult. = 2048, Laplace = 1024 × 1024

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 [31]. 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.

5.2 Experiment Part II

In this section, we evaluate our proposal of DenStream on the basis of OCLSched. All the time measurements represent the total execution time, including the initialization phase, the stream processing and the request response every Rp point. Regarding the input data, we introduce a new design of DenStream performing the data stream management in point packages. This design allows input memory transfers in bigger data portions, reducing the time overhead for small copy tasks. Further, a major improvement against a supposed native GPU implementation is represented in merging multiple kernels, saving task start overhead, which might be caused if too many small tasks scheduled on the GPU device. Both design changes have a significant impact on the measured execution time as will be seen in this section, however, when enough points per package PpS are involved in the computation.

5.2.1 Hardware Environment

All tests are performed on a Fedora 17 Linux PC with a Quad-Core-CPU Intel(R) Core(TM) i5-3550 CPU @ 3.30GHz and 4 GB RAM. The test platform is equipped with an AMD Radeon HD 7870 GPU, with the following technical specifications:1000MHz Engine Clock, 2GB GDDR5 Memory, 1200MHz Memory Clock (4.8 Gbps GDDR5), 153.6 GB/s maximal memory bandwidth, 2.56 TFLOPS in Single Precision, 20 Compute Units (1280 Stream Processors), 256 – bit GDDR5 memory interface, PCI Express 3.0x16 bus interface and an OpenCL(TM) 1.2 support.
Table 3: Data Sets Used

<table>
<thead>
<tr>
<th>data set</th>
<th>number of data objects</th>
<th>type of data objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>DE8K</td>
<td>60000</td>
<td>German cites</td>
</tr>
<tr>
<td>ICML</td>
<td>720792</td>
<td>sensor data</td>
</tr>
</tbody>
</table>

Figure 11: A performance comparison between the OCLSched’s implementation and a traditional CPU version of DenStream: (a) For different Request Periods \( R_p \) and with different numbers of points per package \( PpS \) while using the DE8K data set, (b) For different number of points \( minPts \) while using the ICML data set.

5.2.2 Test Data Sets

Two real data sets (listed in Table 2) are used to test the quality and performance of the clustering algorithm with OCLSched, compared with a CPU version. The first data set DE8K contains geographical positions. The data is provided by the OpenStreetMap project under the Open Data Commons Open Database License (ODbL). The other one represents the Physiological Data Modeling Contest at ICML 2004 data set, which is a collection of activities collected by test subjects with wearable sensors over several months. Each data object exhibits 15 attributes and consists of 55 different labels for the activities and one additional label if no activity was recorded. We picked 9 numerical attributes.

5.2.3 Discussion of Results

Since the request period \( R_p \) constitutes an important parameter in DenStream, the first test focuses on the impact of this parameter on its performance, when comparing the OCLSched’s DenStream implementation with a CPU version. In general, the user can adjust the period of output requests. For example, when setting the request period of 1000, the output with the clustering will be generated every 1000 data records, revealing the changes of the data stream in the time.

In this experiment, the clustering is generated on the DE8K data set using different request periods. As can be seen from Fig. 11a, the shorter the request periods, the smaller is the performance gap between the CPU and the OCLSched’s implementation of DenStream.

During the second experiment and as depicted in Fig. 11b, the high dimensional data set used in the Physiological Data Modeling Contest at ICML 2004 shows the same tendency. Moreover, this test with \( R_p = \)
100 shows that OCLSched’s implementation of DenStream performs faster than the sequential DenStream. It can be observed, that for a big data set and a long term computation, better performance of DenStream is achieved. The offline part computation for a high dimensional data, generates a significant CPU load in the sequential version of DenStream. Using OCLSched, this computation is efficiently achieved on the GPU, while reducing the CPU load and running in parallel with other general purpose tasks.

Resource utilization also constitutes an important point of concern when programming GPUs. In our case considering the DenStream algorithm, this depends on two parameters of the data set: The dimensionality on the one hand, and the data distribution and the speed of data fading defined by \( \lambda \) on the other hand. A small number of micro clusters, caused by a high \( \lambda \) value or data distribution concentrated in small data range, can not achieve a high level of GPU resources’ utilization. For a small number of micro clusters the offline part of DenStream performs also well on the host, and no benefit of the highly parallel GPU architecture can be obtained.

In summary, we propose parallelization of both parts of DenStream, however, due to the sequential data merge logic of DenStream, further optimization of the parallel DenStream using OCLSched would be necessary to achieve better resource utilization. This would require a data merge algorithm that is completely different from the originally used method in DenStream.

The proposed OCLSched’s design of DenStream offers a solution to reduce the CPU load and computation time using the GPU co-processor. The main contribution has been to provide a sustainable development for DenStream to run on GPUs by means of our OpenCL-based scheduler, which would offer better performance, when certain limitations in OCLSched are overcome soon. Especially, when it comes to the support of concurrent execution, future work might show major potential for improvement by boosting the OCLSched’s performance, if OpenCL device partitioning [10] becomes available. Currently, this OpenCL feature is only supported by CPUs and Cell devices.

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. Message passing and shared memory are deployed to the extent needed to facilitate the interaction between users and OCLSched.

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 are planning to investigate the possibility to combine our feature-based predictor with a code-profiling method, which uses past observations to estimate execution run times. On the basis of the predictor’s results, we plan to optimize our scheduler to execute tasks on different coprocessors (including Xeon Phi) simultaneously. Further, we also plan to evaluate the power efficiency of OCLSched when deploying different accelerators in a computing system. 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. Our concept offers a great opportunity to improve the utilization and reduce energy costs in future heterogeneous systems, using a scheduler that is easily integrated in a common OS.

References


