SOCL: An OpenCL Implementation with Automatic Multi-Device Adaptation Support

Sylvain Henry, Denis Barthou, Alexandre Denis, Raymond Namyst, Marie-Christine Counilh

RESEARCH REPORT
N° 8346
August 2013
Project-Team Runtime

ISSN 0249-6399   ISRN INRIA/RR--8346--FR+ENG

SOCL: An OpenCL Implementation with Automatic Multi-Device Adaptation Support

Sylvain Henry [*], Denis Barthou[*], Alexandre Denis[†], Raymond Namyst[*], Marie-Christine Counilh[*]

Project-Team Runtime

Research Report n° 8346 — August 2013 — 18 pages

Abstract: To fully tap into the potential of today's heterogeneous machines, offloading parts of an application on accelerators is not sufficient. The real challenge is to build systems where the application would permanently spread across the entire machine, that is, where parallel tasks would be dynamically scheduled over the full set of available processing units. In this report we present SOCL, an OpenCL implementation that improves and simplifies the programming experience on heterogeneous architectures. SOCL enables applications to dynamically dispatch computation kernels over processing devices so as to maximize their utilization. OpenCL applications can incrementally make use of light extensions to automatically schedule kernels in a controlled manner on multi-device architectures. A preliminary automatic granularity adaptation extension is also provided. We demonstrate the relevance of our approach by experimenting with several OpenCL applications on a range of representative heterogeneous architectures. We show that performance portability is enhanced by using SOCL extensions.

Key-words: OpenCL, Heterogeneous Architectures, Runtime Systems

[*] University of Bordeaux, 351 cours de la Libération, 33405 Talence, France. Email: first.last@labri.fr
[†] Inria, 200 avenue de la Vieille Tour, 33405 Tour, France. Email: first.last@inria.fr

SOCL: une implémentation OpenCL adaptée aux architectures multi-accélérateurs

Résumé : Pour exploiter au mieux les architectures hétérogènes actuelles, il n'est pas suffisant de déléguer aux accélérateurs seulement quelques portions de codes bien déterminées. Le véritable défi consiste à délivrer des applications qui exploitent de façon continue la totalité de l'architecture, c'est-à-dire dont l'ensemble des tâches parallèles les composant sont dynamiquement ordonnancées sur les unités d'exécution disponibles. Dans ce document, nous présentons SOCL, une implémentation de la spécification OpenCL étendue de sorte qu'elle soit plus simple d'utilisation et plus efficace sur les architectures hétérogènes. Cette implémentation peut ordonnancer automatiquement les noyaux de calcul sur les accélérateurs disponibles de façon à maximiser leur utilisation. Les applications utilisant déjà OpenCL peuvent être migrées de façon incrémentale et contrôlée vers SOCL car les extensions fournies sont non intrusives et requièrent très peu de modifications dans les codes. En plus de l'ordonnancement automatique de noyaux de calcul, une extension préliminaire permettant l'adaptation automatique de la granularité est mise à disposition. Nous démontrons la pertinence de cette approche et l'efficacité des extensions fournies à travers plusieurs expérimentations sur diverses architectures hétérogènes représentatives.

Mots-clés : OpenCL, Archtitectures hétérogènes, supports exécutifs

Contents

1 Introduction

Heterogeneous architectures are becoming ubiquitous, especially in high-performance computing centers and in systems on chips for embedded machines [10]. The number of top supercomputers using accelerators such as Graphical Processing Units (GPU) keeps increasing, as highlighted by the Titan supercomputer which uses a mix of AMD Opteron CPUs and NVIDIA Tesla K20 GPUs. With a sustained peak performance measured at 17.5 PFlop/s, it is currently the fastest system worldwide [21]. As a result, for an increasing part of the HPC community, the challenge has shifted from exploiting hierarchical multicore machines to exploiting heterogeneous multicore architectures.

The Open Computing Language (OpenCL) [13] is part of this effort. It is a specification for heterogeneous parallel programming, providing a portable programming language together with a unified interface to interact with various processing units and accelerator devices. With OpenCL, applications explicitly offload computations (*kernels*) on accelerator devices. Synchronizations between computations and data transfer commands are achieved by commands that can be connected by events, enforcing dependencies. OpenCL is available on a wide range of architectures and OpenCL codes are portable on supported platforms. However, performance portability is still difficult to achieve because high performance kernels have to be adapted in term of granularity, parallelism, memory working set to the underlying platform. This performance tuning is so far performed by hand and leads to kernel specialization, depending also on the platform implementation. Besides, OpenCL does not provide support for load balancing strategies over different computing devices. Designing and hand-coding an OpenCL application that maintains load balance between different heterogeneous devices, managing memory coherency is a complex task. This clearly hinders the development of multi-device OpenCL codes and limits their performance.

In this report, we present a unified OpenCL platform, named SOCL, that transparently handles multiple devices and offers to the user a way to use multi-device, heterogeneous architectures with a minimal development cost:

- SOCL is capable of scheduling kernels across multiple platforms and devices, applying dynamic and adaptive load balancing strategies. OpenCL is extended in order to support command queues attached not only to one particular device, but to a group of devices (or all devices).
- SOCL manages transparently memory transfers and coherency accross the devices. The automatic memory management mechanism proposed is similar to caches, removing the need to manually micro-manage device memory.
- If the user defines kernels with different granularities, through a mechanism we propose, SOCL is able to dynamically adapt the granularity of the kernels to the architecture.

Moreover, we show that existing OpenCL codes, where devices and memory transfers are managed manually can be migrated incrementally to automatic scheduling and memory management with SOCL. On the one hand, with little impact on the code, SOCL brings the benefits of a unifying platform, merging different implementations of the installable driver client (ICD). On the other hand, SOCL relies on recent runtime developments for heterogeneous architectures (such as Harmony [5], KAAPI [7], Qilin [16], StarPU [1] or StarSs [2]). Several automatic scheduling strategies are proposed within SOCL and it is possible to customize application dependent strategies if needed.

Three applications are used to evaluate the performance of our approach: an implementation of the Black-Scholes algorithm, a ray-tracing application and a simulation of a dynamical system of particles (N-body).

The remainder of this report is organized as follows: in Section 2, we introduce parts of the OpenCL specification needed to understand our work; we present SOCL, our unified OpenCL platform in Section 3, and its implementation in Section 4; in Section 5, we evaluate the performance of SOCL; in Section 6, we compare our work with existing related works; finally we draw conclusions in the last Section.

2 The OpenCL Platform Model

The OpenCL specification defines both a programming interface (API) for the *host* to submit commands to one or several *computing devices*, and a programming language called OpenCL C Language for writing the tasks to execute on the devices. These tasks, or *kernels*, can be dynamically compiled during the execution of the application for any available accelerator device that supports OpenCL. In this section, we only present the OpenCL programming interface.

Several OpenCL devices from different vendors can be available on a given architecture. Each vendor provides an implementation of the OpenCL specification, called a *platform*, to support its devices. The mechanism used to expose all platforms available for an application is called the Installable Client Driver (ICD).

Devices that need to be synchronized, to share or exchange data can be grouped into *context* entities. *Only devices from the same platform can belong to the same context.* The data buffers required for the execution of computing kernels are created and associated to a context, and are lazily allocated to a device memory when needed.

Commands are executed by computing devices and are of three different types: kernel execution, memory transfer and synchronization. Commands have to be submitted into *command* *queues* to be executed and *each command queue is associated to a single device*, thus command scheduling is performed explicitly by the host program. Commands are issued in-order or out-of-order, depending on the queue type and barriers can be submitted into out-of-order command queues to enforce some ordering.

Additionally, synchronization between commands submitted into different queues (e.g. queues associated to different devices) is possible using *event* entities *as long as command queues belong to the same context*, hence to the same platform. Events give applications a finer control of the dependencies between commands. As such they subsume command queue ordering and barriers. Each command can be associated to an event – triggered when then command completes – and to a list of events that must have been triggered before the command is executed (i.e. dependences). These events can only be defined and used *within the same context*.

3 SOCL Extensions to OpenCL

Our goal with SOCL is to bring dynamic architecture adaptation features into an OpenCL framework. From a programmer perspective, our framework only diverges in minor ways from the OpenCL specification and is thus straightforward to use. Moreover, we strive to extend existing concepts (e.g. contexts become scheduling contexts) instead of introducing new ones so that some of these extensions could be included in a future OpenCL specification revision.

Unlike most other implementations, SOCL is not dedicated to a specific kind of device nor to a specific hardware vendor: it simply sits on top of other OpenCL implementations. SOCL is both an OpenCL implementation and an OpenCL client application at the same time: the former because it can be used through the OpenCL API and the latter because it uses other implementations through the OpenCL API.

SOCL can be considered as a wrapper between applications and OpenCL implementations. Applications using it instead of directly using the other implementations are expected to benefit from the following advantages: (1) a unified OpenCL platform, (2) an automatic command scheduler, (3) some kernel adaptation capabilities.

3.1 A Unified OpenCL Platform

SOCL is an implementation of the OpenCL specification. As such, it can be used like any other OpenCL implementation with the OpenCL host API. As the Installable Client Driver (ICD) extension is supported, it can be installed side-by-side with other OpenCL implementations and applications can dynamically choose to use it or not among available OpenCL platforms (cf Figure 1).

Platform and device entities are the entry points to use any OpenCL implementation (cf Section 2). The OpenCL ICD can be used to multiplex several platforms. On architectures where several platforms are available, applications need to deal with them explicitly. This offers very limited integration because entities from one platform cannot be mixed with entities of another. This implies that for two devices that do not belong to the same platform, it is not allowed:

- to create a context containing both devices;
- to synchronize commands executed on one device with commands executed on the other;
- to share buffers between both devices;
- to copy data from a buffer on the first device to another buffer in the other.

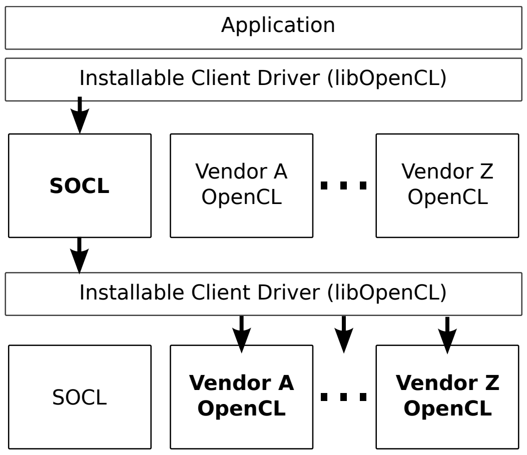


Figure 1: SOCL unified platform uses OpenCL implementations from other vendors and can be used as any other implementation using the ICD. Thus, SOCL is both an OpenCL implementation and an OpenCL (client) application.

SOCL fixes all these issues by wrapping all entities of other platforms into entities of its own unified platform. Hence, applications using SOCL unified platform can create contexts mixing devices that were initially in different platforms. SOCL implements everything needed to make this unified platform support every OpenCL mechanism defined in the specification. In particular, while it was not allowed to combine command queues and events from different platforms, it is now possible to use a single dependency graph for commands submitted on every device of the SOCL Platform.

3.2 Automatic Command Scheduler

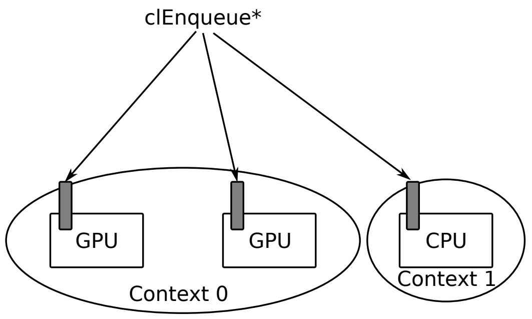
SOCL proposes to extend OpenCL capabilities with automatic task scheduling across different OpenCL devices, while preserving the OpenCL API. We describe in this section how to do this extension without impacting OpenCL programming model.

As stated in Section 2, the current OpenCL definition compels users to decide in which context and on which device each computing kernel has to be executed. SOCL allows applications to define command queues only associated to a context, independently of any particular device. This extends the notion of context to what we call *scheduling contexts*, and these command queues are named *context queues*. Figure 2. (a) and (b) show the two different cases, where command queues are attached to devices or to contexts.

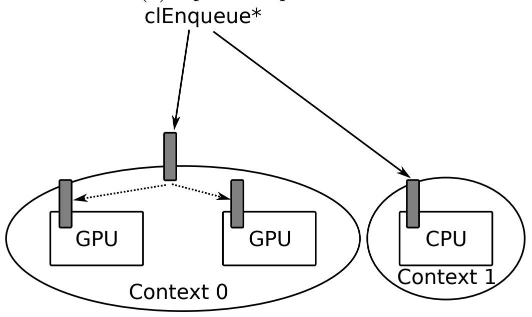
SOCL automatically schedules commands that are submitted in context queues on one of the context devices. Context queues relieve programmers from having to assign a target device to a task, but does not prevent from doing so if they want to, for optimization purposes.

Thanks to the unified platform, applications are free to create as many contexts as they want and to organize them the way it suits them the most. For instance, devices having a common characteristic may be grouped together in the same context. One context containing GPUs and accelerators such as Intel MIC could be dedicated to heavy data-parallel kernels and another containing CPUs and MICs would be used to compute kernels that are known to be more sequential or to perform a lot of control (jumps, loops...). The code in Listing 1 shows how this example would look like.

A predefined set of scheduling strategies assigning commands to devices is built in SOCL.



(a) OpenCL Specification



(b) SOCL Scheduling Contexts

Figure 2: (a) Command queues can only be attached to a device (b) Command queues can be attached to contexts and SOCL automatically schedules commands on devices in the context

They can be selected through context properties. For instance, the code in Listing 2 selects the "heft" scheduling policy, corresponding to the Heterogeneous Earliest Finish Time heuristic [22]. Other available heuristics are referenced in [12], and additional strategies can be user-defined if need be.

3.3 Kernel Adaptation Capabilities

With automatic command scheduling, applications slightly lose the control on where commands are executed. For kernel execution commands, it can be problematic as applications may not want to execute exactly the same kernel depending on the targeted device. Generic OpenCL kernels – that do not use device specific capabilities – may not have good performance compared to kernels that have been tuned for a specific architecture, taking into account specificities such as cache sizes, number of cores. . . The code of the kernels, as well as the size of their dataset, have to be adapted to the device architecture. We propose in this section an approach to tackle this issue.

3.3.1 Conditional Kernel Compilation

SOCL provides additional compilation constants specific to the targeted device. They may be used in kernel source code to help the user to define device-dependent codes. For instance, the constant SOCL_DEVICE_TYPE_GPU is defined for every GPU device.

As kernels can be compiled per device at runtime, this specialization can be done by the user. This enables code adaptation to the devices, however this cannot change the granularity of a

Listing 1: Context queue creation example. Scheduling and load-balancing of commands submitted in these queues are automatically handled by SOCL.

```
cl_context ctx1 = clCreateContextFromType(NULL,
    CL_DEVICE_TYPE_GPU | CL_DEVICE_TYPE_ACCELERATOR, NULL, NULL, NULL);
cl_context ctx2 = clCreateContextFromType(NULL,
    CL_DEVICE_TYPE_CPU | CL_DEVICE_TYPE_ACCELERATOR, NULL, NULL, NULL);

cl_command_queue cq1 = clCreateCommandQueue(ctx1, NULL, 0,
 CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, NULL);
cl_command_queue cq2 = clCreateCommandQueue(ctx2, NULL, 0,
 CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, NULL);
```

Listing 2: Context properties are used to select the scheduling policy.

```
cl_context_properties properties[] = {CL_CONTEXT_SCHEDULER_SOCL, "heft", 0};

cl_context ctx = clCreateContextFromType(properties, CL_DEVICE_TYPE_CPU,
    NULL, NULL, NULL);
```

task. Automatic approaches for kernel optimization, such as proposed by Grewe *et al.* [8], are out of the scope of this document.

3.3.2 Granularity Adaptation

Adapting granularity is a common issue for applications that aim to be portable. Given an architecture composed of several heterogeneous computing devices, and computation that could be split into smaller parts, it is often difficult to estimate how the partition scheme will impact performance. In order to deliver high performance, we believe task granularity adaptation should be done dynamically at runtime by the runtime system.

SOCL supports a preliminary granularity auto-adaptation feature. The developer can associate a *divide function* to each kernel. On kernel submission, the runtime system may decide either to execute this function in place of the kernel, or the kernel. The divide function, executed by the host, takes a parameter called the *partitioning factor* and issues several smaller kernels equivalent to the initial kernel. In general, these kernels define a task graph that correspond to a decomposition of the initial kernel computation. The partitioning factor drives how this initial computation is divided into smaller ones. The range of values for this factor is associated to the kernel in the application and SOCL selects the value dynamically from this range. This means that each time a command is issued for the execution of a kernel, the runtime can decide to run instead the divide function associated to this kernel (on the host), that will itself issue new commands.

SOCL uses a granularity adaptation strategy to explore the partitioning factor range each time a kernel with an associated divide function is submitted, with the same input sizes. In a first phase, it saves statistics about execution time for each partitioning factor. In a second phase it uses and updates these statistics to choose appropriate partitioning factors. These statistics are still updated in the second phase to dynamically adapt to the execution environment (e.g. device load, etc.).

In the future, several granularity adaptation strategies could be implemented, just like several

Listing 3: Divide function prototype

```
cl_int (*)(cl_command_queue cq, cl_uint partitioning_factor,
           const cl_event before, cl_event *after)
```

Listing 4: Definition of a divide function for a kernel

```
#define FUNC  -1
#define RANGE -2

cl_uint sz = sizeof(void *);
cl_uint range= 5;

clSetKernelArg(kernel, FUNC,  sz, part_func);
clSetKernelArg(kernel, RANGE, sz, &range);
```

scheduling strategies are available. Each kernel could use the most appropriate strategy.

The divide function has the prototype presented in Listing 3.

Parameters are respectively the command queue that should be used to enqueue the kernels, the partitioning factor, an event that must be set as dependency to sub-kernels. The last parameter allows the divide function to return an event indicating that every sub-kernel has completed (typically an event associated to a marker command). The return value indicates if an error occurred or not.

Listing 4 shows how an application can associate the divide function (`part_func`) and the partitioning factor range ($[1, 5]$). For this preliminary version, we choose to use the OpenCL API function `clSetKernelArg` with a specific argument index.

4 Implementation

SOCL currently implements the whole OpenCL 1.0 specification (with no imaging support) and parts of newer specifications. It relies on an existing runtime system, namely StarPU to schedule tasks on devices.

The StarPU runtime system [1] is a C library that provides programmers with a portable interface for scheduling dynamic graphs of tasks onto a heterogeneous set of processing units (i.e. CPUs and GPUs). The two basic principles of StarPU are firstly that tasks can have several implementations, for some or each of the various heterogeneous processing units available in the machine, and secondly that necessary data transfers to these processing units are handled transparently by StarPU. StarPU tasks are defined as multi-version kernels, gathering the different implementations available for CPUs and GPUs, associated to a set of input/output data. To avoid unnecessary data transfers, StarPU allows multiple copies of the same registered data to reside at the same time on several processing units as long as it is not modified. Asynchronous data prefetching is also used to hide memory latencies.

StarPU manages a virtual memory just like the one SOCL needs: it handles multiple device memories and performs data transfers as required when kernels are scheduled. Each kernel submitted through SOCL is converted into a StarPU task. For a kernel to be automatically scheduled using a scheduling context, the application must compile it for every device contained in the context beforehand. When StarPU schedules a task on a device, SOCL selects the appropriate compiled kernel.

StarPU only provides a single way to create and initialize a data in the global virtual memory, namely registering. It consists in extending the host managed memory to include a memory area that is already initialized. It is a very cheap operation and the only constraint is that applications do not access the memory area directly thereafter. SOCL mimics this behavior when a buffer is created with the `CL_USE_HOST_PTR` flag. Thus, it is the recommended way to create and initialize a buffer with SOCL.

Nevertheless, SOCL also supports data transfers provided by the `ReadBuffer` and `WriteBuffer` commands. The latter may also be used to initialize a buffer but with the risk of an overhead due to the data transfer to the selected (manually or automatically) OpenCL device. Finally, another way to create and initialize a buffer is to use the `CL_MEM_COPY_HOST_PTR` flag on buffer creation. In this case, SOCL duplicates initializing data in host memory before using StarPU's data registering mechanism.

5 Evaluation

In this Section, we present performance figures to show the benefits of our unified approach. Three OpenCL benchmarks are considered: Black-Scholes, LuxRender and N-body simulation. In our experiments, we have used the following hardware platforms:

- Hannibal: Intel Xeon X5550 2.67GHz with 24GB, 3 NVidia Quadro FX 5800
- Alaric: Intel Xeon E5-2650 2.00GHz with 32GB, 2 AMD Radeon HD 7900
- Averell1: Intel Xeon E5-2650 2.00GHz with 64GB, 2 NVidia Tesla M2075

The software comprises Linux 3.2, AMD APP 2.7, Intel OpenCL SDK 1.5 and Nvidia CUDA 4.0.1.

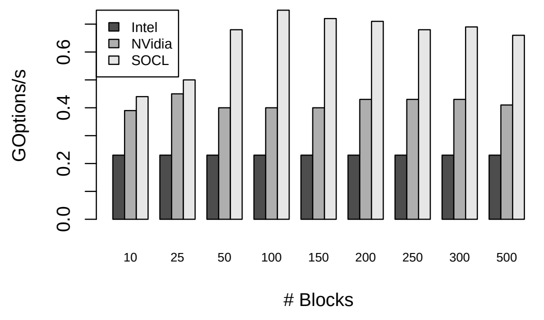
5.1 Black-Scholes

The Black Scholes model is used by some options market participants to estimate option prices. Given three arrays of n values, it computes two new arrays of n values. It can be easily parallelized in any number of blocks of any size. We use the kernel provided in NVidia OpenCL SDK that uses float values in each array. Figures 3a, 3b and 3c present performance obtained on Hannibal with blocks of fixed size of 1 million, 5 million and 25 million options.

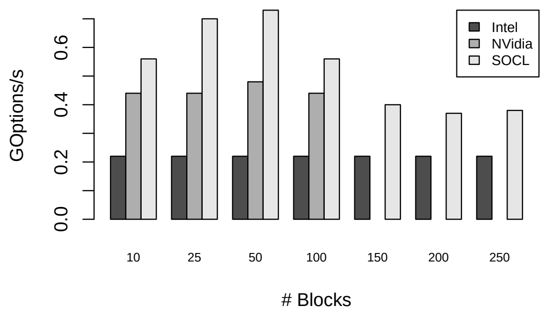
The first two tests have been performed using a round-robin distribution of the blocks with Intel and NVidia OpenCL implementations. When the size and the number of blocks are too high, the graphic cards do not have enough memory to allocate required buffers. Since NVidia's OpenCL implementation is not able to automatically manage memory on the GPU, it fails in these cases, which explains why some results are missing. In contrast, when it uses graphic cards, SOCL resorts to swap into host memory.

The last test presents performance obtained with the automatic scheduling mode of SOCL. The blocks are scheduled on any device (GPU or CPU). We observe that on this example, automatic scheduling always gives better performance than the naive round-robin approach, nearly doubling performance in the case of 1M options (for 100 blocks). This is due to the fact that both computing devices (CPU and GPU) are used, which neither NVidia nor Intel implementation of OpenCL is able to achieve.

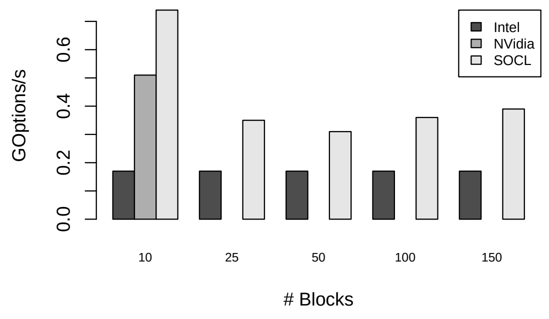
Figure 3d presents the results we obtained by performing 10 iterations on the same data using the same kernel. This test illustrates the benefits of automatic memory management associated



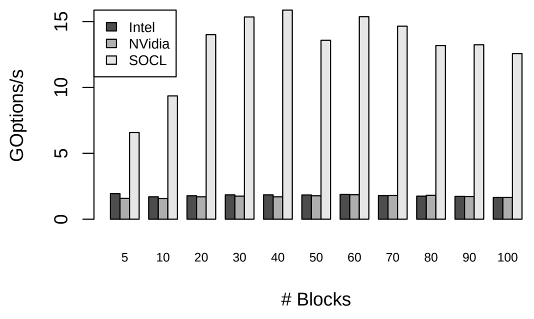
(a) 1M Options



(b) 5M Options



(c) 25M Options



(d) 25M Options / 10 iterations

Figure 3: Performance of Black Scholes algorithm on Hannibal with blocks containing 1 million (a), 5 millions (b) and 25 millions of options (c). Performance of 10 iterations (i.e. with data reuse) of the Black Scholes algorithm on Averell1 with a total option count of 25 millions (d).

with the scheduling, when there is some temporal locality. The test was conducted on Averell1 with a total option count of 25 millions. This time we used two different scheduling algorithms – eager and heft – to show the impact of choosing the appropriate one. We can see that the heft algorithm clearly outperforms the other approaches in this case, and avoids unnecessary memory transfers. Indeed, this algorithm takes into account memories into which data are stored to schedule tasks. This advantage comes with very little impact on the original OpenCL code, since it only requires to define a scheduling strategy to the context, and to remove device information in the definition of command queues.

Overall, this example shows the following properties of the SOCL platform:

- The swapping mechanism allowing large computations to be performed on GPUs, on contrary to NVidia OpenCL implementation
- An efficient scheduling strategy in case of data reuse with the heft scheduler
- A performance gain up to 85% without data reuse and way higher in case of data reuse

5.2 LuxRender

LuxRender [17] is a rendering engine that simulates the flow of light using physical equations and produces realistic images. LuxRays is a part of LuxRender that deals with ray intersection using OpenCL to benefit from accelerator devices. SLG2 (SmallLuxGPU2) is an application that performs rendering using LuxRays and returns some performance metrics. SLG2 can only use a single OpenCL platform at a time. As such, it is a good example of an application that could benefit from SOCL property of grouping every available device in a single OpenCL platform.

For this experiment, we did not write any OpenCL code, we used the existing SLG2 OpenCL code unmodified, once for each OpenCL platform. We used SLG2 batch mode to avoid latencies that would have been introduced if the rendered frame was displayed and we arbitrarily fixed the execution time to 120 seconds. Longer the execution time is, better the rendered image quality is as the number of computed samples is higher. SLG2 also supports usual CPU compute threads but we disabled those because they would have interfered with OpenCL CPU devices. SLG2 has been configured to render the provided "luxball" scene with the default parameters.

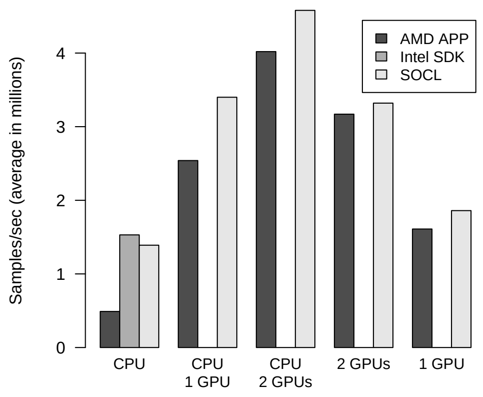
The average amount of samples computed per second for each OpenCL platform on two architectures is shown in Figure 4. When a single device is used (CPU or GPU), SOCL generally introduces only a small overhead compared to the direct use of the vendor OpenCL implementation. However it could even perform better in some cases (e.g. with a single AMD GPU) presumably thanks to a better data pre-fetching strategy.

On Alaric architecture, we can see that the CPU is better handled with the Intel OpenCL implementation than with the AMD one. Best performance is obtained with the SOCL platform using only GPUs through the AMD implementation and the CPU through Intel's one. On Averell1 architecture, best performance is also obtained with SOCL when it commingles Nvidia and Intel implementations.

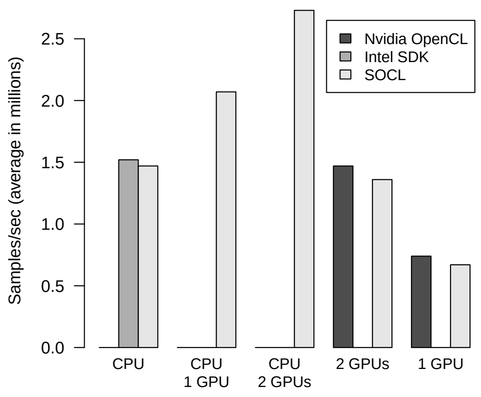
In conclusion, this shows that an OpenCL application designed for using a single OpenCL platform could directly benefit from using the SOCL unified platform without any change in its code.

5.3 N-body

The OTOO simulation code [19] is an simulation of a dynamical system of particles, for astrophysical simulations. The OpenCL code is written for multiple devices. We have adapted it to



(a) Results on Alaric



(b) Results on Averell1

Figure 4: LuxRender rendering benchmark indicates the average number of samples rendered per second (higher is better)

be able to launch more kernels than the number of devices and to use command queues with out-of-order execution mode. We assess the code performance on 4 heterogeneous devices (1 CPU and 3 GPU) for Hannibal configuration. The benchmark computes 20 iterations over 4000K particles, and for each iteration, i.e. each time step, the same computation is performed (a force calculation for each particle of the system using a tree method). In the OTOO simulation code, the force calculations for different particles are completely independent of each other. Therefore, the computation can be simply cut into smaller computations by distributing the particles to kernels. The kernels use shared read-only buffers describing all the particles and the tree, and each kernel uses its own write-only buffer to store the resulting force of each particle.

The difficulty for this benchmark and for this heterogeneous architecture is to balance the load between the multi-core CPU and the 3 GPUs. This is a strong scalability issue: the same amount of computation has to be divided among all devices. As GPUs are faster on this kernel than the CPU, too few kernels creates either load imbalance between CPU and GPU, or leads to use only GPUs. Too many kernels creates a high level of parallelism, but reduces the computation efficiency (and the parallelism inside kernels) for each device.

The number of kernels launched for each iteration can be along 1 to 64. Figure 5 shows performance gains when the number of kernels ranges from 2 to 64 (for all 20 iterations), compared to the version running one kernel per iteration. The last experiment (Adaptive) shows the speed-up when the number of kernels launched is adapted dynamically by SOCL. All speed-ups are evaluated for two scheduling heuristics, Eager and Heft.

The figure shows that the heuristics do not propose a perfect load-balancing strategy for all numbers of kernels. The efficiency starts to reduce after 32 kernels (resp. 8 kernels) for the Eager scheduling (resp. the Heft scheduling), for this size of input.

For the adaptive execution, the partitioning factor range is [1,5] (see Listing 4). An overview of a way to write the code of the *divide function* `part_func` is given in Listing 5. The real partitioning factor set used here is {4, 8, 16, 32, 64}. So, during the first 5 iterations of the 20 iterations, the number of kernels launched in the `part_func` function, will be successively a power of 2, from 4. During this first phase of the execution, statistics about the execution time of the `part_func` function are saved. For the remaining iterations, the `part_func` function is called with a partitioning factor in the range [1,5], determined by the SOCL run time support.

Overall, this shows that an adaptive strategy is an efficient way to find automatically an

Listing 5: Divide function

```
cl_int part_func(cl_command_queue cq, cl_uint partitioning_factor,
                 const cl_event before, cl_event *after) {

// Enqueue commands transferring relevant data in input buffers (read-only).
// These commands depend on the "before" event. (not shown)

// Compute the real partitioning factor from the  given factor that belongs
// to the partitioning range
factor = pow(2, partitioning_factor+1);

// Compute the number of particles per block
size = number_of_particles / factor;

// Kernel executions must happen after transfers
clEnqueueBarrierWithWaitList(cq,0,NULL,NULL);

// Launch kernels
for (i=0; i < factor; i++) {
   offset = size * i;

   // Set kernel arguments: buffers and offset
   // in the input. (not shown)

   // Enqueue kernel execution command
   clEnqueueNDRangeKernel(cq, kernel, 1, NULL,
                          global, local,
                          0, NULL, &evs[i]);

   // Enqueue a command to transfer output data
   // that depends on evs[i] (not shown)
 }

 // Set "after" marker event
 clEnqueueBarrierWithWaitList(cq,0,NULL,after);

}
```

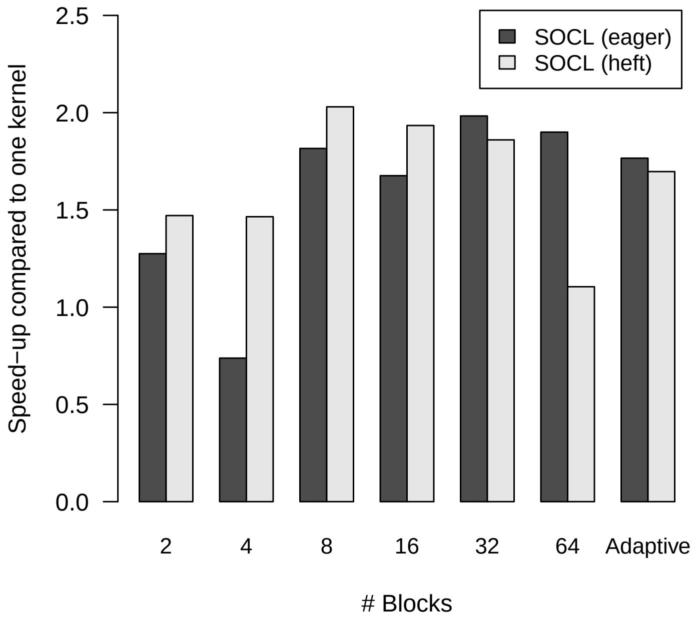


Figure 5: Performance speed-ups for NBody benchmark when the number of kernels is changed from 2 to 64 (by power of 2) and when using the adaptive granularity approach, on Hannibal. Speed-ups are relative to the performance of one kernel.

appropriate level of decomposition for the computation so as to avoid load-imbalance. Besides, this is also a way to overcome the shortcomings of scheduling heuristics.

6 Related Works

Unifying OpenCL devices IBM's OpenCL Common Runtime [11] also provides a unified OpenCL platform consisting of all devices provided by other available implementations. This feature is also proposed by SOCL. However the OpenCL Common Runtime does not provide automatic scheduling. Multicoreware's GMAC (Global Memory for Accelerator) [18] allows OpenCL applications to use a single address space for both GPU and CPU kernels. However, it defines its own API on contrary to SOCL. Used with TM (Task Manager), it also supports automatic task scheduling on heterogeneous architectures using a custom programming interface. Kim *et al.* [14] proposes an OpenCL framework that considers all available GPUs as a single GPU. Their framework expresses code for a single GPU and partitions the work-groups among the different devices, so that all devices have the same amount of work. Their approach does not handle heterogeneity among GPUs, not a hybrid architecture with CPUs and GPUs, and the workload distribution is static. Besides, data dependences between tasks are not considered since work-groups are all independent. However, the method they propose to split the computation into different independent smaller computations could be used in our adaptive granularity framework. This would relieve the programmer from defining a divide function. De La Lama *et al.*[4] propose a compound OpenCL device in order to statically divide the work of one kernel among the different devices. Maestro[20] is a unifying framework for OpenCL, providing scheduling strategies to hide communication latencies with computation. Maestro proposes one unifying device for heterogeneous hardware. Automatic load balance is achieved thanks to an autotuned performance model, obtained through benchmarking at install-time. This mechanism also help to adapt the size of the data chunks given as parameters to kernels. On contrary to SOCL, Maestro assumes the kernels can be tuned at compile-time, while SOCL performs a runtime adaptation. Moreover, the mechanism proposed in SOCL for varying task granularity in more general than changing the size of data chunks.

SnuCL [15] is an OpenCL framework for clusters of CPUs and GPUs. The OpenCL API is extended so as to include functions similar to MPI. Besides, the OpenCL code is either translated into C for CPU, or Cuda for GPUs. The SnuCL runtime does not offer automatic scheduling between CPUs and GPUs, on contrary to SOCL and the scheduling has to be achieved by the programmer. Moreover, SnuCL does not handle multi-device on the same node. The approach of SnuCL (multiple nodes, one device per node) is complementary to SOCL (single node, mutliple devices).

Automatic scheduling on heterogeneous architectures A static approach to load partitioning and scheduling is presented by Grewe and O'Boyle [8]. At runtime, the decision to schedule the code on CPU or GPU uses a predictive model, based on decision trees built at compile time from microbenchmarks. However, the case of multiple GPU is not directly handled, and the decision to schedule a code to a device does not take into account memory affinity considerations.

Besides, some recent works use OpenCL as the target language for other high-level languages (for instance, CAPS HMPP [6] and PGI [23]). Grewe *et al.* [9] propose to use OpenMP parallel programs to program heterogeneous CPU/GPU architectures, based on their previous work on static predictive model. The work proposed here for SOCL could be used in these contexts.

Finally, several previous works have proposed dedicated API and runtimes for the automatic scheduling on heterogeneous architectures. StarPU [1] is a runtime system that provides both a global virtual memory and automatic kernel scheduling on heterogeneous architectures. SOCL currently relies on it internally and provides the additional OpenCL implementation layer that was not available initially in StarPU which only supports its own programming interface. Qilin, StarSS and Kaapi [16, 2, 7] are other examples of runtimes for heterogeneous architectures, that do not rely on the standard OpenCL programming interface but on special APIs or code annotations.

Boyer *et al.*[3] propose a dynamic load balancing approach, based on an adaptive chunking of data over the heterogeneous devices and the scheduling of the kernels. The technique proposed focuses on how to adapt the execution of one kernel on multiple devices. SOCL offers a wider range of application, with multiple scheduling strategies, more elaborate task division and dynamic granularity adaptation.

7 Conclusions and Perspectives

In this report we presented SOCL, an OpenCL implementation that provides a *unified OpenCL platform* that simplifies programming of applications on heterogeneous architectures: OpenCL mechanisms can be used equally with all devices regardless of their initial platform when they are used through the SOCL platform. In addition, SOCL offers a mechanism to automatically schedule commands on devices belonging to a context. Finally, a preliminary automatic granularity adaptation scheme is proposed: the user has to propose how to divide its tasks, and SOCL applies this method to create more parallelism among devices, following a simple strategy.

The first two contributions require only to change a few lines of code in existing OpenCL applications, as shown for instance on the LuxRender code. This brings performance gain up to 85% on multi-GPU, multi-core machines, compared to solutions using only the GPUs (example for LuxRender and Black-Scholes).

The unified platform proposed in SOCL means that OpenCL applications do not have to worry about data transfers among devices anymore. These are automatically performed using prefetching and eviction strategies: buffers allocated in device memory are automatically swapped out in host memory if necessary; indirect data transfers from devices to devices through host memory are handled internally; when a kernel is scheduled on a device, the buffers it uses are automatically transfered in device memory if necessary and as soon as possible considering available memory. We have shown on Black-Scholes benchmark that automatic memory management in SOCL enabled large computations to be performed on GPUs, on contrary to NVidia OpenCL implementation.

We have proposed in SOCL to extend contexts of OpenCL to be *scheduling contexts*, where command queues do not need to be attached to a single device anymore. Commands submitted in these command queues are automatically scheduled on any device of the context. This functionality could be used to create groups of devices with the same capabilities (e.g. a group for data intensive tasks and another for control intensive tasks respectively corresponding to a context composed of GPU devices and a context composed of CPU devices).

Finally, we proposed a preliminary strategy to adapt dynamically the granularity of the kernels scheduled on the devices, in order to adapt to the heterogeneity. While this requires some extension of the OpenCL API and requires some additional work from the user (specifying how to divide work), the results on N-Body benchmark are promising. This will be further explored in future works.

References

[1] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 2010.

[2] Eduard Ayguadé, Rosa M. Badia, Francisco D. Igual, Jesús Labarta, Rafael Mayo, and Enrique S. Quintana-Ortí. An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 851–862, Berlin, Heidelberg, 2009. Springer-Verlag.

[3] Michael Boyer, Kevin Skadron, Shuai Che, and Nuwan Jayasena. Load balancing in a changing world: Dealing with heterogeneity and performance variability. In *IEEE Computing Frontiers conference*, 2013.

[4] C.S. de La Lama, P. Toharia, J.L. Bosque, and O.D. Robles. Static multi-device load balancing for opencl. In *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, pages 675–682, 2012.

[5] Gregory F. Diamos and Sudhakar Yalamanchili. Harmony: an execution model and runtime for heterogeneous many core systems. In *Proceedings of the 17th international symposium on High performance distributed computing*, HPDC '08, pages 197–200, New York, NY, USA, 2008. ACM.

[6] Romain Dolbeau, Stéphane Bihan, and François Bodin. HMPP: A hybrid Multi-core Parallel Programming Environment. 2007.

[7] Thierry Gautier, Xavier Besseron, and Laurent Pigeon. Kaapi: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In *Proceedings of the 2007 international workshop on Parallel symbolic computation*, PASCO '07, pages 15–23, New York, NY, USA, 2007. ACM.

[8] Dominik Grewe and Michael F.P. O'Boyle. A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL. In *CC '11: Proceedings of the 20th International Conference on Compiler Construction*. Springer, 2011.

[9] Dominik Grewe, Zheng Wang, and Michael F.P. O'Boyle. Portable mapping of data parallel programs to opencl for heterogeneous systems. In *ACM/IEEE International Symposium on Code Generation and Optimization*, Shenzen, China, February 2013.

[10] Heterogeneous System Architecture. `http://hsafoundation.com`, 2012.

[11] IBM. OpenCL Common Runtime for Linux on x86 Architecture (version 0.1), 2011.

[12] Inria. StarPU Handbook, 2013. `http://runtime.bordeaux.inria.fr/StarPU/`.

[13] Khronos OpenCL Working Group. The OpenCL Specification, Version 1.2, 2011.

[14] Jungwon Kim, Honggyu Kim, Joo Hwan Lee, and Jaejin Lee. Achieving a single compute device image in opencl for multiple gpus. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 277–288, New York, NY, USA, 2011. ACM.

[15] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. Snucl: an opencl framework for heterogeneous cpu/gpu clusters. In *Proceedings of the 26th ACM international conference on Supercomputing*, ICS '12, pages 341–352, New York, NY, USA, 2012. ACM.

[16] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 45–55, New York, NY, USA, 2009. ACM.

[17] LuxRender. Gpl physically based renderer. `http://www.luxrender.net`, 2013.

[18] Multicoreware, Inc. GMAC: Global Memory for Accelerator, TM: Task Manager, 2011. `http://www.multicorewareinc.com`.

[19] Naohito Nakasato, Go Ogiya, Yohei Miki, Masao Mori, and Ken'ichi Nomoto. Astrophysical particle simulations on heterogeneous cpu-gpu systems. *CoRR*, abs/1206.1199, 2012.

[20] Kyle Spafford, Jeremy Meredith, and Jeffrey Vetter. Maestro: data orchestration and tuning for opencl devices. In *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part II*, Euro-Par'10, pages 275–286, Berlin, Heidelberg, 2010. Springer-Verlag.

[21] The Top 500. TOP500 List. `http://www.top500.org/lists/2012/11`, 2012.

[22] H. Topcuoglu, S. Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *Parallel and Distributed Systems, IEEE Transactions on*, 13(3):260 –274, mar 2002.

[23] Michael Wolfe. Implementing the pgi accelerator model. In GPGPU, 2010.

RESEARCH CENTRE
BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour
33405 Talence Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399