Toward OpenCL Automatic Multi-Device Support

Sylvain Henry[1], Alexandre Denis[2], Denis Barthou[3], Marie-Christine Counilh[3], and Raymond Namyst[3]

[1] Exascale Computing Research Laboratory, France
sylvain.henry@exascale-computing.eu
[2] Inria Bordeaux – Sud-Ouest, France
alexandre.denis@inria.fr
[3] Univ. of Bordeaux, France
denis.barthou@inria.fr, {counilh,raymond.namyst}@labri.fr

Abstract. To fully tap into the potential of today heterogeneous machines, offloading parts of an application on accelerators is no longer 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 paper 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. We demonstrate the relevance of our approach by experimenting with several OpenCL applications on a range of heterogeneous architectures. We show that performance portability is enhanced by using SOCL extensions.

1 Introduction

Heterogeneous architectures are becoming ubiquitous in high-performance computing centers as well as in embedded systems [1]. The number of top supercomputers using accelerators such as GPU or Xeon Phi keeps growing. 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) [2] 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 the different processing devices. In OpenCL, programmers explicitly define code fragments (*kernels*) to be executed on particular devices. Kernel executions, synchronizations, and data transfers are then explicitly triggered by the host and dependencies are enforced by user-defined events. OpenCL applications are portable over a wide range of supported platforms. However, performance

F. Silva et al. (Eds.): Euro-Par 2014, LNCS 8632, pp. 776–787, 2014.

portability is still difficult to achieve because high performance kernels have to be adapted in term of (i) parallelism, (ii) granularity and (iii) memory working set to the target device architecture. Adapting parallelism requires that the implicit kernel dependence graph built by the programmer exposes enough parallelism to feed all computing devices. This effort has to be achieved by the user. As each task has to be mapped to a particular device in OpenCL, load-balancing strategies for heterogeneous architectures have also to be hand-tuned. Load-balancing issues for heterogeneous platforms are clearly a limiting performance factor for OpenCL codes. Likewise, adapting granularity is a strong scalability requirement, and since different devices may have very different memory hierarchies, granularity and working sets have also a high impact on performance. While OpenCL kernels are compiled at load-time, their granularity are determined by the user. Adapting granularity thus results in writing as many kernels as there are different devices. Performance comes therefore at the expense of portability, reducing the competitive edge of OpenCL compared to other parallel languages.

Our contribution lies in the design, implementation and validation of new OpenCL mechanisms that tackle load-balancing issues on heterogeneous devices. Kernels submitted by users are automatically scheduled on devices by our OpenCL runtime system. It handles load-balancing issues and maintains the coherency of data across all devices by performing appropriate data transfers between them. These mechanisms have been implemented in our unified OpenCL platform, named SOCL. 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. With little impact on the code, making OpenCL codes use SOCL implementation is a way to adapt transparently to multi-device architectures.

The remainder of this paper is organized as follows: we present SOCL, our unified OpenCL platform in Sect. 2, and its implementation in Sect. 3; in Sect. 4, we evaluate the performance of SOCL; in Sect. 5, we compare our work with existing related works; finally we draw conclusions in the last section.

2 Dynamic Adaptation of Parallelism to Heterogeneous Architectures

We aim at bringing dynamic architecture adaptation features into an OpenCL framework called SOCL. In this section we show how OpenCL applications can benefit from the following advantages: (1) a unified OpenCL platform, (2) an automatic memory management over all devices, and (3) an automatic command scheduler.

2.1 SOCL: A Unified OpenCL Platform

The OpenCL specification defines a programming interface (API) for the *host* to submit commands to *computing devices*, and a programming language called OpenCL C language for writing the *kernels* — the tasks to execute on the

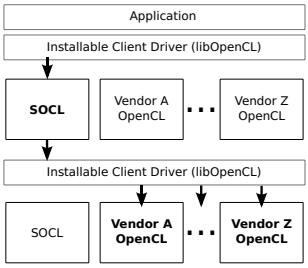


Fig. 1. SOCL unified platform uses OpenCL implementations from other vendors and can be used as any other implementation using the ICD. Thus, SOCL is both an OpenCL implementation and an OpenCL (client) application.

devices. Kernels can be dynamically compiled during the execution of the application for any available accelerator device that supports OpenCL.

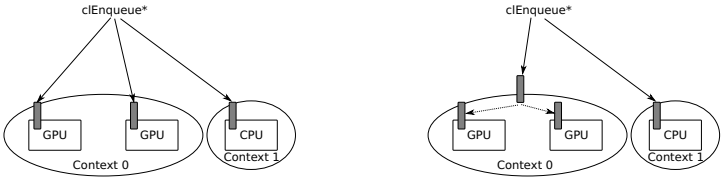
To handle the case where multiple OpenCL devices from different vendors are available on a given machine, each vendor provides an implementation of the OpenCL specification, called a *platform*. The Installable Client Driver (ICD) exposes all platforms available for an application. Devices that need to be synchronized, to share or exchange data can be grouped into *context* entities. However OpenCL is restrictive about interaction between devices: devices in a context must all belong to the same platform. Thus it prevents synchronization commands between devices from different vendors — as is often found on heterogeneous architectures.

As an answer to this issue we propose a *unified platform* provided by SOCL. It can be used like any other OpenCL implementation with the OpenCL host API. As the 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, as depicted in Fig. 1.

A distinctive feature of SOCL is that it wraps all entities of the other platforms into entities of its own unified platform. SOCL implements everything needed to make this unified platform support every OpenCL mechanism defined in the specification. Hence, applications using SOCL unified platform can create contexts mixing devices that were initially in different platforms. In particular, it is possible to use command queues, context and events for tasks to schedule on different devices.

2.2 Automatic Memory Management

SOCL provides a global virtual memory encompassing every device memory and part of the host memory with a relaxed consistency model. Every buffer can be accessed by any command (kernel execution, transfer, etc.) on any device because the runtime system ensures that a valid copy of the buffer is present in the device memory before executing the command, performing appropriate data transfers beforehand if required. Moreover it ensures that two commands accessing the



(a) Explicit mapping with OpenCL contexts

(b) Explicit or automatic mapping with SOCL scheduling contexts

Fig. 2. (a) Command queues are attached to single devices (b) Context queues are attached to contexts and SOCL automatically schedules commands on devices in the context

same buffer are not executed simultaneously if one of them is writing, while a buffer can be concurrently accessed for reading. Finally, as commands can be enqueued in advance (into command queues), SOCL can anticipate some data transfers for commands whose dependencies have not yet completed.

In addition to device memories, SOCL uses host memory space to store buffers that have to be evicted from a device memory to make some room for other buffers as well as to perform indirect data transfers between device memories. When buffers are created using host memory mapping (`CL_MEM_USE_HOST_PTR` flag), the host memory space is aggregated to the SOCL managed host memory and must not be used directly anymore (i.e., without using OpenCL API) by the application. Temporarily direct access to a buffer in the managed host memory can be obtained using OpenCL buffer mapping in host address space facilities (`clEnqueueMapBuffer`).

Using the `CL_MEM_USE_HOST_PTR` flag is the preferred way to create initialized buffers with SOCL as it avoids any superfluous data transfer. Nevertheless other mechanisms such as explicitly writing into a buffer (i.e., `WriteBuffer` command) or writing into a buffer mapped in host address space are fully supported.

2.3 Automatic Command Scheduler

In OpenCL applications, commands such as kernel executions, memory transfers or synchronizations are submitted to a command queue attached to a single device. Synchronization between commands from different queues is possible using *event* entities. Events give a fine control of the dependencies between commands. As such they subsume command queue ordering and barriers. Each command can trigger an event when the command completes, and depends on a list of events triggered by other commands. Events can only be defined and used *within the same context*.

We propose to attach command queues to contexts, independently of any particular device, as illustrated on Fig. 2. It enables the runtime system to schedule commands submitted to these queues onto any device of the context. This extends the notion of context to what we call *scheduling contexts*, and these

Listing 1.1. 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 1.2. 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);
```

command queues are named *context queues*. Thanks to context queues, programmer may rely on automatic task placement by the runtime system, rather than to decide placement manually. On another hand, it does not forbid manual placement if programmer wants so for optimization purposes. Several contexts may be created to ease application development, e.g., programmer may create a context queue with all accelerators for data parallel tasks (GPU, Xeon Phi) and another context queue for code with more control (CPU, Xeon Phi). Listing 1.1 shows an example of code with two scheduling contexts and two context queues. Note that command queues are created with a `NULL` device since they are *context queues*.

The runtime scheduling strategy has to take into account various device properties such as: *memory capacity*, so as not to saturate a device memory; *affinity between tasks*, i.e., schedule tasks on the same device as their input buffer already is; *performance* of devices, i.e., schedule tasks on the most efficient device for the task. A predefined set of scheduling strategies assigning commands to devices is available for SOCL, brought by StarPU. They can be selected through context properties. For instance, the code in Listing 1.2 selects the *heft* scheduling policy, implementing the Heterogeneous Earliest Finish Time heuristic [3], based on estimated tasks and transfers durations. Other heuristics are available, such as *eager* where every device picks a task in a shared queue when it becomes idle, and additional strategies can be user-defined if need be.

3 SOCL Implementation

SOCL currently implements the whole OpenCL 1.0 specification (except imaging), and parts of newer specifications. SOCL relies on StarPU [4] runtime system. Namely, SOCL is an OpenCL frontend for StarPU with unified platform, and is distributed as open-source software together with StarPU. StarPU uses a task-based programming model with explicit dependencies; SOCL

extends StarPU memory management and event mechanism in order to handle all OpenCL specification.

When a kernel is created using OpenCL, the SOCL implementation automatically handles the allocation and configuration of a StarPU kernel. When an OpenCL kernel is enqueued for execution, a StarPU task is created and configured to be executed on appropriate devices (all the devices of the target context or a selected device). OpenCL provides two mechanisms to order task executions: events, i.e., explicit dependencies, and synchronization on command queues. Implicit dependencies between kernels placed in an in-order command queue are converted by SOCL into explicit dependencies for StarPU. Similarly, barriers are also translated into explicit dependencies between tasks separated by these synchronizations.

To implement OpenCL buffer allocation, SOCL triggers the allocation of StarPU data, of the "variable" flavor, following StarPU terminology. To implement OpenCL buffer initialization mechanisms, SOCL circumvents StarPU limitations. Indeed, StarPU only provides a registering mechanism similar to OpenCL buffer allocation when the `CL_USE_HOST_PTR` is set. All the other allocation modes in OpenCL have been implemented within SOCL. Moreover, data transfers between host memory and buffers are not supported directly by StarPU, where transfers are the consequence of data dependence between tasks. In SOCL, the implementation of `ReadBuffer` and `WriteBuffer` commands for instance resorts to StarPU tasks with no computational part but dependent on the data to transfer.

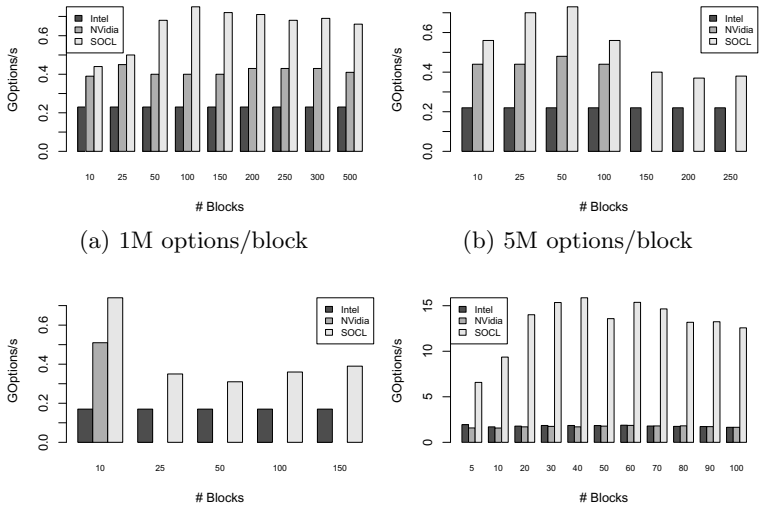
4 Performance Evaluation

In this section, we present performance figures to show the benefits of our approach. Three OpenCL benchmarks are considered: Black-Scholes, LuxRender and HDR Tone Mapping. Experiments are conducted on 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; `averell` — 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.

4.1 Black-Scholes

The Black-Scholes model is used by some option market participants to estimate option prices. Given three arrays of n values, it computes two new arrays of n values. The code is easily parallelized in any number of blocks of any size. We use the kernel provided by Nvidia OpenCL SDK, using float values for each array.

Figures 3a, 3b and 3c present performance obtained on `hannibal` with blocks of fixed size of 1 million, 5 millions and 25 millions options, comparing Intel OpenCL, Nvidia OpenCL, and SOCL. Intel and Nvidia OpenCL tests have been performed using a static round-robin distribution of the blocks on devices. Since



(a) 1M options/block (b) 5M options/block

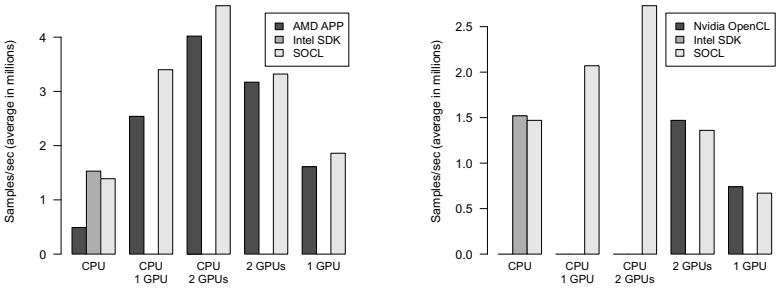
(c) 25M options/block (d) total 25M options, 10 iterations

Fig. 3. Performance of Black-Scholes algorithm with blocks containing 1M (a), 5M (b) and 25M options (c). Performance of 10 iterations with a total of 25M options (d).

Nvidia OpenCL implementation is restricted to GPU memory, it fails in case the problem does not fit graphic card memory, which explains why some results are missing for Nvidia. SOCL tests were obtained with automatic scheduling mode, able to schedule tasks on any device (GPU or CPU). On this example, SOCL automatic scheduling always reaches better performance than 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, while Nvidia and Intel OpenCL implementations use only one type of device.

Figure 3d shows results obtained on 10 iterations on the same data set using the same kernel, with a total option count of 25 millions. It illustrates the benefits of automatic memory management associated with scheduling, when there is some temporal locality. The test was conducted on `averell`. 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 for the context, and to remove device information in the definition of command queues.

Overall, this example shows the benefits of a unifying platform, able to use both CPU and GPUs with an efficient dynamic memory management allowing large computations to be performed on GPUs, contrary to Nvidia OpenCL implementation. It exhibits a performance gain up to 85% without data reuse and way higher in case of data reuse.



(a) Results on `alaric` (b) Results on `averell`

Fig. 4. LuxRender benchmark results (average number of samples rendered per second)

4.2 LuxRender

LuxRender [5] 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 use the existing SLG2 OpenCL code unmodified, and run on Nvidia, AMD, Intel OpenCL and SOCL with the example "luxball" scene with default parameters. We use batch mode and run rendering for 120 seconds. We disable CPU compute threads to avoid conflicts with OpenCL CPU devices.

The average amount of samples computed per second for each OpenCL platform is shown in Fig. 4. When a single device is used (CPU or GPU), SOCL introduces only a small overhead compared to the direct use of the vendor OpenCL. However in the case of a single AMD GPU, SOCL outperforms the vendor implementation, presumably thanks to a better data pre-fetching strategy.

On `alaric`, CPU is better handled with the Intel OpenCL implementation than with AMD OpenCL. The best performance is obtained with the SOCL platform using 2 GPUs and the CPU, combining the use of the AMD implementation for the GPUs and the Intel for the CPU. On `averell`, the best performance is also obtained with SOCL when it uses both Nvidia and Intel implementations.

This test shows that an OpenCL application designed for using a single OpenCL platform can directly benefit from using the SOCL unified platform without any change in its code.

4.3 HDR Tone Mapping

HDR Tone Mapping is an image processing technique to render a high dynamic range image on a device with a limited dynamic range. Intel has implemented [6]

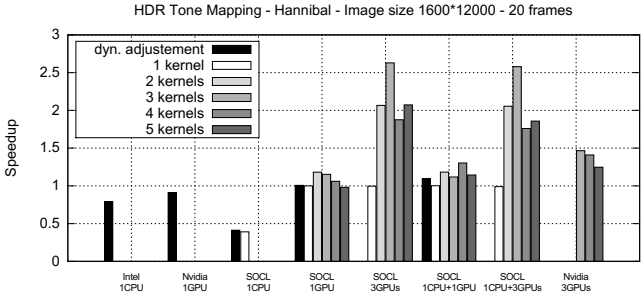


Fig. 5. Speed-ups for HDR Tone Mapping on `hannibal`, relative to SOCL / 1 GPU.

this technique in OpenCL. Their code features multi-device support, both CPU and GPU, given that both devices are in the same platform (Intel GPU and CPU). Each frame is split to balance load between CPU and GPU, the splitting ratio being dynamically computed based on processing times measured for previous frames.

We have modified the code to run more than two kernels, with an equal amount of data between kernels, in order to let SOCL perform kernel scheduling using the *heft* scheduler. Kernels are submitted to SOCL command queue attached to context with out-of-order execution. The number of kernels can be greater than the number of devices.

Our benchmark consists in rendering 20 frames for an image of size 1600×12000. The results are shown in Fig. 5. Original code with dynamic load balancing is referred to as "dynamic adjustment". Since our test machine has an Nvidia GPU, not Intel, the original dynamic adjustment code runs on CPU (Intel platform) or GPU only (Nvidia platform). It can use CPU+GPU through our SOCL platform which unifies both devices on a single virtual platform, and gives only a small performance boost compared to CPU or GPU only. The modified code running on SOCL gets similar performance to the original code on GPU and CPU+GPU. It is slower on CPU alone, which is explained by the fact that SOCL considers CPU as a regular OpenCL device and performs memory transfers that could be optimized out in a future version. When all 3 GPUs are used by SOCL, we get a speed-up of 2.6 with the modified code, and Nvidia OpenCL gets a speed-up of 1.5; the difference may be explained by StarPU managing data transfers better than the application code. For CPU+3 GPUs, SOCL performance is roughly the same since CPU is slow on this example; thanks to SOCL scheduler, adding a slow CPU to device set does not degrade performance.

This benchmark demonstrates that SOCL is able to efficiently aggregate performance of multiple OpenCL devices. Contrary to original Intel code, it is able to aggregate performance of devices from multiple platforms, and more than two devices. Moreover, it handles kernel scheduling and load balancing in a generic fashion in the runtime system, rather than hard-coded in the application. The speed-ups we obtain are convincing.

5 Related Works

About unifying OpenCL devices. IBM OpenCL Common Runtime [7] provides a unified OpenCL platform consisting of all devices provided by other available implementations, like SOCL does. However OpenCL Common Runtime does not provide automatic scheduling. Multicoreware GMAC (Global Memory for Accelerator) [8] 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. Kim *et al.* [9] propose an OpenCL framework that considers all available GPUs as a single GPU. It 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, nor 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. De La Lama *et al.*[10] propose a compound OpenCL device in order to statically divide the work of one kernel among the different devices. Maestro[11] 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 applies dynamic scheduling strategy at runtime which is more flexible. SnuCL [12] is an OpenCL framework for clusters of CPUs and GPUs. The SnuCL runtime does not offer automatic scheduling between CPUs and GPUs, on contrary to SOCL and the scheduling is performed 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).

About automatic scheduling on heterogeneous architectures. Grewe and O'Boyle [13] propose a static approach to load partitioning and scheduling. At runtime, the decision to schedule code 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 [14] and PGI [15]). Grewe *et al.* [16] 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 [4] 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 [17,18,19] 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.*[20] 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 applications made of multiple kernels, scheduled using dependencies.

6 Conclusion and Future Work

The OpenCL language is a *rosetta stone* to program heterogeneous parallel computing platforms. It is portable across a range of different devices and make them usable through a unified interface. However it lacks mechanisms to make multiple devices usable seamlessly.

In this paper we have presented several extensions to OpenCL to simplify programming of applications on heterogeneous architectures. We have proposed the SOCL platform, able to make OpenCL mechanisms usable equally with all devices regardless of their initial platform. In addition, SOCL offers a mechanism to automatically schedule commands on devices belonging to a context.

The unified platform proposed in SOCL means that OpenCL applications do not have to worry about data transfers and kernel scheduling. These operations are automatically performed. It requires only minor changes to existing OpenCL code to use context queues rather than explicit device queues. This brings significant performance gain on multi-GPU, multi-core machines, compared to solutions using only the GPUs. Moreover, we have shown that automatic memory management in SOCL enabled large computations to be performed on GPUs, on contrary to other OpenCL implementations.

As future work, we currently study a preliminary strategy to adapt dynamically the granularity of kernels, in order to adapt to the heterogeneity. The user explicitly gives a function to divide work; the runtime calls it whenever it needs more parallelism to feed devices. Preliminary results are promising but need further exploration about strategies to choose the best suited granularity for given devices.

References

1. HSA Foundation: Heterogeneous System Architecture (2012), `http://hsafoundation.com`
2. Khronos OpenCL Working Group: The OpenCL Specification, Version 1.2 (2011)
3. Topcuoglu, H., Hariri, S., Wu, M.Y.: Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE Transactions on Parallel and Distributed Systems 13(3), 260–274 (2002)
4. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.-A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 863–874. Springer, Heidelberg (2009)

5. LuxRender: GPL physically based renderer (2013), http://www.luxrender.net
6. Intel: Hybrid HDR tone mapping for post processing multi-device version (2013), http://software.intel.com/en-us/vcsource/samples/hdr-tone-mapping-multi-device
7. IBM: OpenCL Common Runtime for Linux on x86 Architecture (version 0.1) (2011)
8. Multicoreware, Inc.: GMAC: Global Memory for Accelerator, TM: Task Manager (2011), http://www.multicorewareinc.com
9. Kim, J., Kim, H., Lee, J.H., Lee, J.: 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 2011, pp. 277–288. ACM, New York (2011)
10. de La Lama, C., Toharia, P., Bosque, J., Robles, O.: Static multi-device load balancing for opencl. In: 2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications (ISPA), pp. 675–682 (2012)
11. Spafford, K., Meredith, J., Vetter, J.: Maestro: data orchestration and tuning for OpenCL devices. In: D'Ambra, P., Guarracino, M., Talia, D. (eds.) Euro-Par 2010, Part II. LNCS, vol. 6272, pp. 275–286. Springer, Heidelberg (2010)
12. Kim, J., Seo, S., Lee, J., Nah, J., Jo, G., Lee, J.: SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters. In: Proceedings of the 26th ACM International Conference on Supercomputing, ICS 2012, pp. 341–352. ACM, New York (2012)
13. Grewe, D., O'Boyle, M.F.P.: A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL. In: Knoop, J. (ed.) CC 2011. LNCS, vol. 6601, pp. 286–305. Springer, Heidelberg (2011)
14. Dolbeau, R., Bihan, S., Bodin, F.: HMPP: A hybrid Multi-core Parallel Programming Environment (2007)
15. Wolfe, M.: Implementing the PGI accelerator model. In: GPGPU (2010)
16. Grewe, D., Wang, Z., O'Boyle, M.F.: 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)
17. Luk, C.K., Hong, S., Kim, H.: Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 42, pp. 45–55. ACM, New York (2009)
18. Ayguadé, E., Badia, R.M., Igual, F.D., Labarta, J., Mayo, R., Quintana-Ortí, E.S.: An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 851–862. Springer, Heidelberg (2009)
19. Gautier, T., Besseron, X., Pigeon, L.: 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 2007, pp. 15–23. ACM, New York (2007)
20. Boyer, M., Skadron, K., Che, S., Jayasena, N.: Load balancing in a changing world: Dealing with heterogeneity and performance variability. In: IEEE Computing Frontiers Conference (2013)