A Framework for Transparent Execution of Massively-Parallel Applications on CUDA and OpenCL

Jörn Teuber[1], Rene Weller[1], Gabriel Zachmann[1]

[1]University of Bremen, Germany; {jteuber,weller,zach}@cs.uni-bremen.de

Abstract

We present a novel framework for the simultaneous development for different massively parallel platforms. Currently, our framework supports CUDA and OpenCL but it can be easily adapted to other programming languages. The main idea is to provide an easy-to-use abstraction layer that encapsulates the calls of own parallel device code as well as library functions. With our framework the code has to be written only once and can then be used transparently for CUDA and OpenCL. The output is a single binary file and the application can decide during run-time which particular GPU-method it will use. This enables us to support new features of specific platforms while maintaining compatibility. We have applied our framework to a typical project using CUDA and ported it easily to OpenCL. Furthermore we present a comparison of the running times of the ported library on the different supported platforms.

Categories and Subject Descriptors (according to ACM CCS): I.3.6 [Computer Graphics]: Methodology and Techniques—Device independence

1. Introduction

The single core performance of CPUs stagnates widely since many years. On the other hand, the demand for larger scenes with increasing details, accurate physics simulation, marker less tracking devices and many other expectations on modern VR applications requires more and more computational power. Obviously, the solution is an increasing degree of parallelization. While the parallelization of CPUs progresses relatively slowly, current GPUs offer an exhaustive number of parallel processing units. Unfortunately, it is often not trivial to parallelize tasks or algorithms. Hence, a lot of research has been spent in the development of new algorithms that are better suited for parallel execution.

Especially in the academic world, such algorithms are often implemented only as a proof-of-concept. To do this, researches often choose NVIDIA's CUDA programming language [NBGS08] because it is easy to use, it offers sophisticated tools for debugging and profiling and it has great community support. Moreover, a lot of libraries exist that allow rapid application development and the features of new GPU generations are integrated immediately. Unfortunately, programs developed in CUDA are restricted to work only on NVIDIA GPUs [LNOM08]. Obviously, software companies that sell their products to a broader audience need a different API.

Here, OpenCL [SGS10], a platform-independent standard for parallel programming, defined by the Khronos Group, is usually the method of choice. However, porting code from CUDA to OpenCL requires the replacement of standard libraries, for instance for parallel sorting or image processing, and the re-implementation of large parts of the code.

We present a novel method to overcome these, often time consuming, limitations. Our new wrapper framework for CUDA/OpenCL supports an easy work-flow for porting CUDA to OpenCL code and supports platform independent development from the start. To do that we implemented an easy to use common interface that encapsulates the calls of parallel device code as well as library functions. The resulting single binary includes both CUDA as well as OpenCL code, and the application can decide during run-time which particular GPU-method it will use. This enables us to support new features that are supported much earlier by

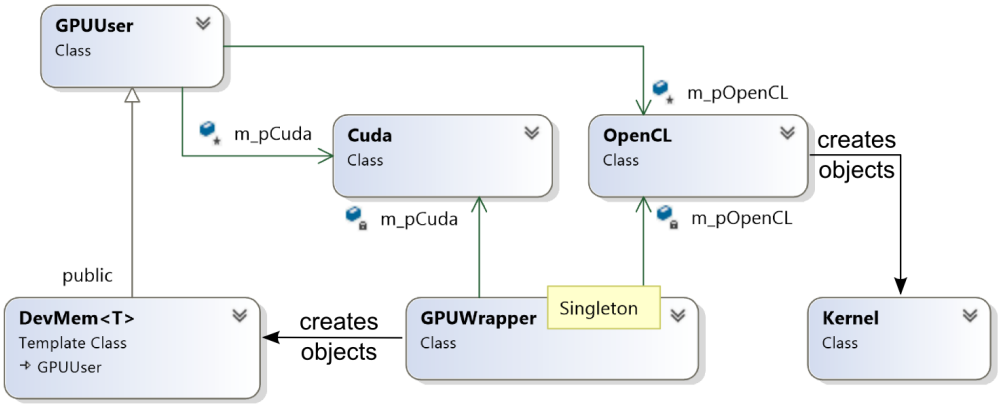

Figure 1: Class diagram of our wrapper. The `GPUWrapper` class is a singleton and the central wrapper for all host-code-calls while `DevMem<T>` wraps the memory access. Objects of DevMem are created by the `GPUWrapper`. The `GPUUser` class provides access to the used GPGPU-Implementation to all instances of the `DevMem` template.

CUDA while maintaining compatibility. In addition, it allows us to perform comparisons between CUDA and OpenCL, and between execution on the GPU and on the CPU regarding their respective performance very easily. We have applied our framework to an extensive CUDA library for the parallel computation of sphere packings [WZ10].

2. Related Work

In the past there were several projects to enable developers to port CUDA to OpenCL. The most recent one, CU2CL [GSFM13], is an attempt at an automated translator for CUDA code to OpenCL code. It translates given .cu files, the .cpp file equivalent for CUDA code, into several C++ and OpenCL files to use with OpenCL. This automatic conversion requires all CUDA API calls to be encapsulated into functions in .cu files, which is not the common way of using CUDA. Also the intended result of CU2CL is just the code for OpenCL, CUDA can not directly be used side-by-side. Hence, the programmers have to avoid the most recent CUDA features that are not yet supported by OpenCL and moreover, it does not support external library calls.

A project more similar to ours was Swan [HDF11], which combined a translator for CUDA kernels to OpenCL kernel code with a common API encapsulating the CUDA and OpenCL APIs. Unfortunately this project was discontinued in 2010 and not updated since then. To our knowledge there are no other current projects providing transparent access to CUDA and OpenCL.

There are several papers comparing the performance of CUDA and OpenCL, mostly for very specific use cases. [KDH10] for instance investigated a Monte Carlo simulation of a quantum spin system while [MS14] considered text encryption using the GPU, to name a few. Both papers observed that OpenCL is slightly slower than CUDA. [LCMH14] on the other hand used CUDA, OpenCL and GLSL to accelerate skeletal animations and found that all three methods yielded a comparable performance.

3. Our Approach

The basic idea of our approach is to maintain a single wrapper class that encapsulates all CUDA/OpenCL related functions. This class has a simple and general interface that allows the handling of both parallel programming languages simultaneously. The output is a single binary for both CUDA and OpenCL. The application can decide at runtime, which library to use. Also it is reusable and may enable other programs to provide both OpenCL and CUDA support.

We start with a short recap of the GPU programming terminology. **GPGPU** is the abbreviation for general purpose computation on graphics processing units. It is used to refer to any kind of computing on the GPU outside of computer graphics, from re-purposed shaders to CUDA and OpenCL. The processor on which the program is started, i.e. the CPU, is called the **Host** while the processor that is used for the massively parallel algorithm is called **device**. This can be a GPU, but also a CPU or an accelerator card in the case of OpenCL. **Kernels** are the programs that are being run in parallel on the device.

```
void main()
{
  // initialize with CUDA as GPGPU method if available, or OpenCL if not
  if( GPUWrapper::getCUDADeviceCount() > 0 )
    GPUWrapper::init( GPCUDA );
  else
    GPUWrapper::init( GPOpenCL );
  GPUWrapper* gpu = GPUWrapper::getSingletonPtr();

  // generate random numbers on the host
  float host_array[1000000];
  std::generate(host_array, host_array+1000000, rand);

  // create a new array on the device and copy the array to the device
  DevMem<float>* device_array = gpu->copyToDev( host_array, 1000000 );

  // sort the array and download it from the device
  gpu->sort( device_array );
  device_array->copyToHost( host_array );

  return 0;
}
```

Figure 2: Simple example program using our wrapper for sorting a random array.

Our approach consists of three main components (See Figure 1).

- `GPUWrapper` class: This class builds the core of our approach. It realizes the main interface for the developers, including initialization and platform-independent access to the massively parallel functions and libraries.
- `DevMem` class: A template class to handle the massively parallel data structures.
- `GPUUser` class: This class implements the real mapping to the particular platform. Actually, this class is used only internally and not directly accessed by the developer.

In the following, we will describe the functionality of all these components in more details. The `GPUWrapper` class is the central wrapper for both OpenCL and CUDA, the functionality of which are implemented in the same-named classes. The `GPUWrapper` class is a singleton which has to be initialized by the user with the information about the GPGPU method and device to use. We use the singleton pattern here to provide universal access to the initialized `GPUWrapper`. Actually, all CUDA API functions can be accessed everywhere in the code but the access to OpenCL devices is restricted to only a single device at the same time.

The `DevMem` template encapsulates the different models of pointer to device memory. It also provides the basic member functions like `memSet` and `copyToHost`. Other important methods for GPGPU like scan, sort or reduce are provided by `GPUWrapper` and than delegated to the *Thrust* library for CUDA or *boost::compute* for OpenCL.

The `GPUUser` class consists simply of static pointers to objects of the `CUDA` and the `OpenCL` classes, which are set by `GPUWrapper` during initialization. This provides all instances of `DevMem` access to `CUDA` and `OpenCL`. A template inheriting from a class with static variables is a common pattern to provide all instances of the template with access to those static member variables.

Figure 2 shows a typical example how to use our approach: We start with the initialization of an `GPUUser` object, define some device data using our `DevMem` template class and finally, do our massively parallel computations on this data.

So far, our wrapper can automatically handle only code on the host side, but not the kernel code. This means, the kernel code has to be converted manually. Luckily, the porting of the kernels from CUDA to OpenCL is mostly straight-forward text replacement as long as no advanced CUDA specific methods are used. In our case, we just had to make minor adjustments, which were mostly due to the different memory addressing models of CUDA and OpenCL and the fact that OpenCL uses plain C while CUDA allows more C++-like code.

The method of kernel calls differs significantly be-

tween CUDA and OpenCL. NVIDIA uses a compiler extension to make CUDA kernel calls to be as similar to normal function calls as possible. OpenCL kernels on the other hand are runtime-compiled for every device (with the option to save and load compiled kernels at runtime) and called by supplying an OpenCL function with the handle of the kernel and other information. This dissimilarity leads to a necessary deviation from good wrapper design, which is the *Kernel* class, objects of which can only be created by the OpenCL class. Fortunately, NVIDIA suggests to use small wrapper functions for the kernel calls anyway, so by just extending those, the main host code does not change. You can see one of those wrapper functions in Figure 3.

4. Application

We applied our method to a library that computes space filling sphere packings for arbitrary 3D objects [WZ10]. Originally, the library was developed in CUDA but industrial partners have the demand to run it also on systems without NVIDIA GPUs and even without dedicated graphics cards at all. The library relies on typical library calls for sorting and scanning but it additionally contains different individual kernels and different data structures.

We will provide a short recap on the basic idea of the sphere packing algorithm to enable a better judgment of its complexity: The main idea is to generate a greedy sphere packing, i.e., to insert successively the largest possible sphere into the object considering the already inserted spheres. The major challenge is to find the position of this largest spheres. Here, the authors provided a simple heuristic that is similar to techniques that are often used in machine-learning algorithms: A so-called prototype is initially inserted at an arbitrary position inside the object and then this prototype is iteratively moved, depending on its minimum distance to the surface (see Fig. 4). In order to avoid local minima, the authors do not use a single prototype but many of them that move independently and hence, can be easily parallelized. In order to accelerate the distance computations the library contains different implementations of discrete distance fields (see [TWZG13] for more details). Algorithm 1 provides a short overview on the main steps in pseudo code.

5. Results

We have applied our semi-automatic CUDA/OpenCL version to the above mentioned sphere packing library. The porting of the complete library took only a half day for the adjustment of the CUDA kernels code. Additionally, we tested the performance of the compiled dual-executable on an PC with Intel I7 CPU and

Algorithm 1: computeSpherePacking(3D object O)

while *O not densly filled* **do**
  while *Not converged* **do**
    In parallel Insert new prototypes
    In parallel Move protoypes
  In parallel Sort prototypes with respect to distance
  In parallel Insert new spheres
  In parallel Update discrete distance field

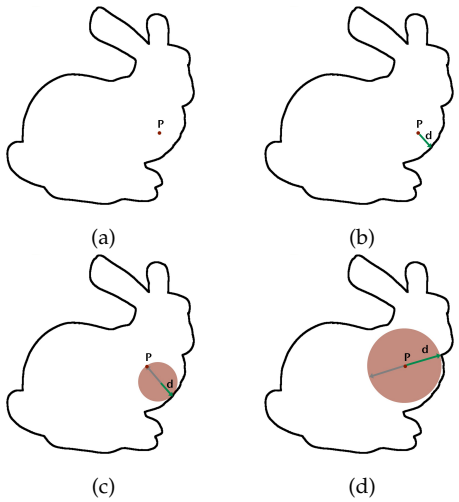

Figure 4: Visualization of the prototype convergence: (a) Place the prototype P randomly inside the object, (b) calculate the closest point on the surface and the distance d, (c) move P away from the closest point, and (d) repeat this until the prototype converges.

a NVIDA GTX 680 GPU with 8 GByte of memory. The CUDA code was executed on the GPU exclusively whereas the OpenCL code was compiled for the CPU as well as for the NVIDIA GPU and the built in Intel graphics adapter. We tested different 3D objects and filled them with different numbers of spheres.

Figure 5 shows the results of our tests. In all our test cases the CUDA and the OpenCL-GPU version have a very similar performance. In our first test case, the OpenCL-CPU version is slower than all GPU versions, which is expected. We recognized a speed-up of about an order of magnitude with the GPU version for the NVIDIA GPU. Even the built-in Intel GPU runs up to a factor of two faster than the CPU version.

Surprisingly, in our second scenario (see Fig. 5, right), the CPU outperforms both GPU versions at the beginning of the algorithm, when the number of spheres is small. In this scenario we changed the reso-

```
void countMemory(int nrTriangles, ObjectOnDevice *object,
                    DevMem<unsigned int>* cellsPerTriangle )
{
  if( GPUWrapper::getSingletonPtr()->getType() == GPCuda )
  {
    uint numThreadsPerBlock, numBlocks;
    computeGridSize( nrTriangles, blockSize, &numBlocks, &numThreadsPerBlock );

    countMemoryKernel<<< numBlocks, numThreads >>>( object->m_uiNumTriangles,
        object->m_dVertices->getCUDA(), object->m_dVertexIndices->getCUDA() );
  }
  else if( GPUWrapper::getSingletonPtr()->getType() == GPOpenCL )
  {
    static Kernel* spKernel = NULL;
    if( spKernel == NULL )
      spKernel = GPUWrapper::getSingletonPtr()->getRawOpenCL()->createKernel(
          "countMemoryKernel", "ExplicitGrid_kernels.cl" );

    spKernel->execute( nrTriangles, blockSize, object->m_uiNumTriangles, getGridConfigOCL(),
        object->m_dVertices->getOCL(), object->m_dVertexIndices->getOCL() );
  }
}
```

Figure 3: Example of a user-defined wrapper function for custom kernel calls. This has to be implemented by the application's programmer for every custom kernel. If CUDA is used, the kernel is called with the respective CUDA syntax. In the case of OpenCL, the kernel is loaded from a file into a static pointer and executed using a variadic method of the kernel class.

lution of the discrete distance map while maintaining the number of prototypes. Basically, in the convergence step of Algorithm 1, each prototype checks all grid cells in the discrete distance field that could possibly contain the closest point on the surface. If we increase the resolution of the discrete distance field, this results in a higher number of cells that has to be visited and consequently, to an increasing workload per prototype, i.e. per thread. On the GPU, the threads on a warp are executed in lock-step, whereas the CPU can schedule new threads at any time. Consequently, we have a worse degree of parallelization on the GPU and therefore a worse core utilization than on the CPU.

Actually, we were already aware of this problem but we did not anticipate the magnitude of the impact on the timings. So, the porting of the sphere packing to OpenCL provided us also new insights about opportunities to further optimize the algorithm.

6. Conclusion and Future Work

We presented a new wrapper approach which enables programmers to develop massively parallel algorithms for CUDA and OpenCL simultaneously or to port their CUDA algorithms to OpenCL or vice versa very easily. Our semi-automatic wrapper is easy to use and it supports platform specific libraries as well as kernel code written by the application programmers. We used our wrapper to port a complex library from CUDA to OpenCL. In addition, we present the running times of that library that allow a comparison of the respective performances of massively-parallel implementations on CUDA, OpenCL/GPU, and OpenCL/CPU. Our measurements show a very similar running time of CUDA vs. OpenCL on the GPU with OpenCL on the CPU being slower by about an order of magnitude in case of optimized parallelization.

In conclusion, the additional work of porting a program from CUDA to OpenCL can be held low. In this paper, we presented such a software architecture that is easy to implement. Having thus the opportunity to run your code on different platforms and parallelization APIs at any time during the development process helps a lot to gain insights into how your code performs on those vastly different different platforms.

In the future we would like to extend our wrapper to support more external libraries and we will release it under an open-source license as we believe that this might be useful to other people. Furthermore, some parts of the kernel code still have to be ported manually. A complete automatic conversion between CUDA and OpenCL kernel code would be an interesting project for the future.

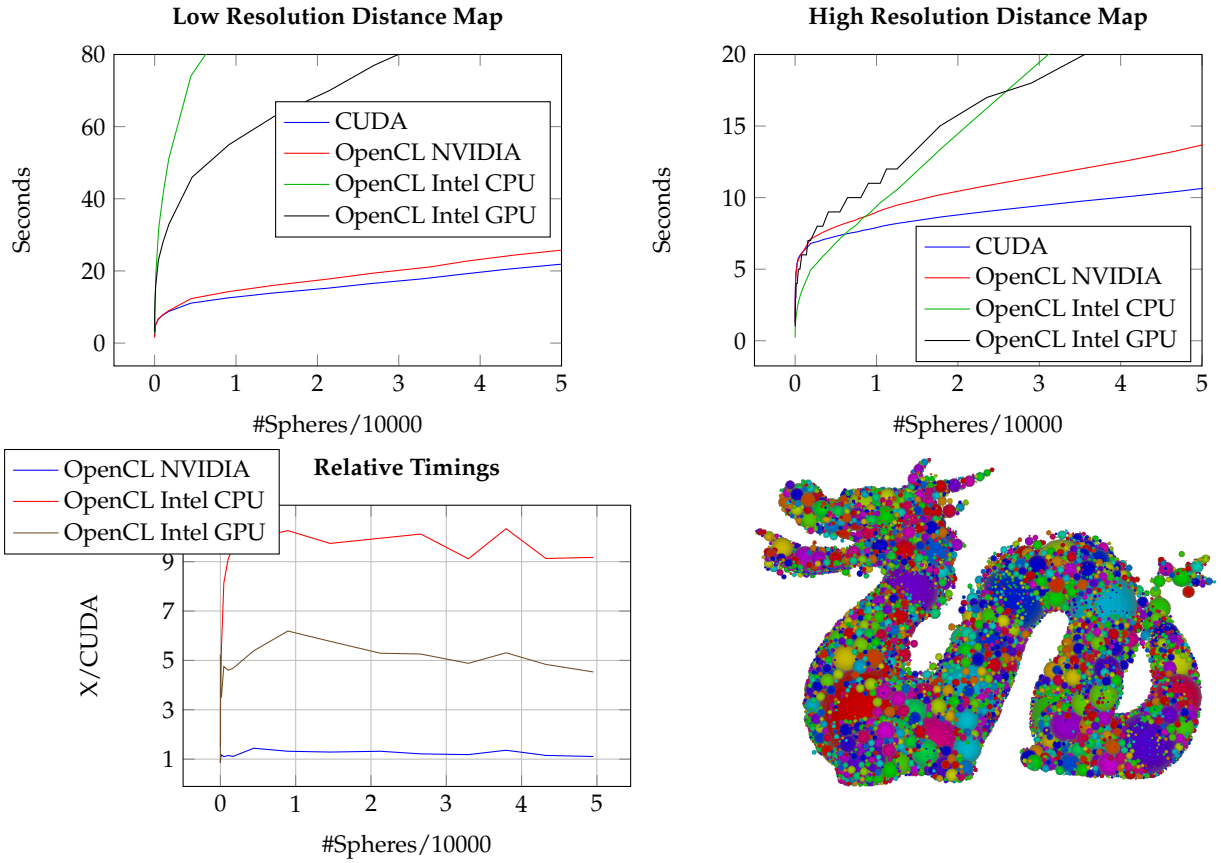

Figure 5: Performance of sphere packing algorithm with different resolutions of the discrete distance map for the dragon model (bottom right). In case of a low resolution we have a high degree of parallelization and both GPU versions outperform the CPU version by a factor of 5 (upper left). In case of a high resolution, the degree of parallelization decreases and the CPU version is faster, at least in early stages of the algorithm (upper right).

References

[GSFM13] GARDNER M., SATHRE P., FENG W.-C., MARTINEZ G.: Characterizing the Challenges and Evaluating the Efficacy of a CUDA-to-OpenCL Translator. *Parallel Computing* (October 2013). 2

[HDF11] HARVEY M. J., DE FABRITIIS G.: Swan: A tool for porting cuda programs to opencl. *Computer Physics Communications 182*, 4 (2011), 1093–1099. 2

[KDH10] KARIMI K., DICKSON N. G., HAMZE F.: A performance comparison of cuda and opencl. *arXiv preprint arXiv:1005.2581* (2010). 2

[LCMH14] LIU S., CHEN G., MA C., HAN Y.: Gpgpu acceleration for skeletal animation-comparing opencl with cuda and glsl. *Journal of Computational Information Systems 10*, 16 (2014), 7043–7051. 2

[LNOM08] LINDHOLM E., NICKOLLS J., OBERMAN S., MONTRYM J.: Nvidia tesla: A unified graphics and computing architecture. *IEEE micro*, 2 (2008), 39–55. 1

[MS14] MAHAJAN S., SINGH M.: Performance analysis of efficient rsa text encryption using nvidia cuda-c and opencl. In *Proceedings of the 2014 International Conference on Interdisciplinary Advances in Applied Computing* (New York, NY, USA, 2014), ICONIAAC '14, ACM, pp. 31:1–31:6. URL: http://doi.acm.org/10.1145/2660859.2660941, doi:10.1145/2660859.2660941. 2

[NBGS08] NICKOLLS J., BUCK I., GARLAND M., SKADRON K.: Scalable parallel programming with cuda. *Queue 6*, 2 (2008), 40–53. 1

[SGS10] STONE J. E., GOHARA D., SHI G.: Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering 12*, 1-3 (2010), 66–73. 1

[TWZG13] TEUBER J., WELLER R., ZACHMANN G., GUTHE S.: Fast sphere packings with adaptive grids on the gpu. In *In GI AR/VRWorkshop* (Würzburg, Germany, September 2013). 4

[WZ10] WELLER R., ZACHMANN G.: Protosphere: A gpu-assisted prototype guided sphere packing algorithm for arbitrary objects. In *ACM SIGGRAPH ASIA 2010 Sketches* (New York, NY, USA, 2010), SA '10, ACM, pp. 8:1–8:2. URL: http://cg.in.tu-clausthal.de/research/protosphere, doi:http://doi.acm.org/10.1145/1899950.1899958. 2, 4