CELERITY: Towards an Effective Programming Interface for GPU Clusters

Peter Thoman, Herbert Jordan,
Philipp Gschwandtner and Thomas Fahringer
Distributed and Parallel Systems Group
University of Innsbruck, 6020 Austria
Email: {petert,herbert,tf}@dps.uibk.ac.at

Biagio Cosenza and Ben Juurlink
AES Group, EECS
Technical University of Berlin
Berlin, 10623 Germany
Email: {cosenza,b.juurlink}@tu-berlin.de

I. Introduction & Motivation

The complexity of today's HPC systems is growing along with their computational power. The TOP 500 list [1] shows that the most powerful current HPC systems are highly parallel and heterogeneous, consisting of a combination of multi-core CPUs, GPUs and accelerators in clusters of interconnected nodes. Writing efficient applications for such systems is challenging, as it requires the use of specific low-level parallel programming paradigms at node level (e.g., OpenMP [2] or OpenCL [3]), while leaving inter-node communication to libraries such as MPI [4]. Although the latter has evolved over time, its use still limits productivity as the application programmer is responsible for the complexity of task scheduling.

To deliver higher productivity for scientists and other end-users, a number of high-level, abstract programming models have been proposed, which leverage run-time systems and task-based DAG representation in order to distribute tasks among available processing devices. Most programming models require the user to locate and specify parallelism or explicitly require user-placed synchronization; examples include UPC [5], Cilk [6], and Chapel [7]. Although static, user-specified schedules and partitionings are common, the increasing complexity of future systems will require automatic tuning support to dynamically optimize the utilization of resources through runtime systems; examples of such dynamic systems supporting heterogeneous distributed memory architectures are *StarPU* [8] and *OmpSs* [9].

A promising HPC programming approach leverages C++ template libraries, which hide the details of the underlying infrastructure from application experts. Implementations of this principle include Kokkos [10] from Sandia National Laboratories and the RAJA portability layer [11], which is currently under development at Lawrence Livermore National Laboratory. A lower level of abstraction, which is based on similar technology, is provided by the OCCA library [12].

Unfortunately, all of these template libraries require significant changes to existing code bases, and none of them seems likely to develop into an industry standard. *Therefore, we are currently designing a* ***programming system for accelerator and GPU clusters, CELERITY, which requires only minor adaption from applications developed for SYCL****, an industry standard supported by the Khronos group.*

II. The CELERITY Programming Interface

The high-productivity of the CELERITY environment relies on a simple programming model, which is based on the SYCL standard [13], a royalty-free, cross-platform abstraction layer that builds on the underlying concepts, portability and efficiency of OpenCL [3] and enables code for heterogeneous processors to be written in a *single-source* style using standard C++. SYCL has the advantage to clearly distinguish the parallel (*kernel*) from the sequential part (*host*)[1] of a program, while keeping the source code simple and concise. Existing SYCL implementations only target shared memory multi-core-systems and GPUs. In CELERITY, we extend the SYCL standard by introducing a transparent distributed work queue representing the whole distributed memory HPC cluster and hiding implementation issues such as task partitioning and resource management. The following code shows a vector addition as created for the CELERITY environment:

```
#include <sycl.hpp>
#include <celerity.hpp>
using namespace cl::sycl;
constexpr int SIZE = 1024; // size of vectors

int main() { // input/output host vectors
  std::vector h_a(SIZE), h_b(SIZE), h_c(SIZE);
  // fill inputs with random float values
  for(int i = 0; i < SIZE; i++) {
    h_a[i] = rand() / (float)RAND_MAX;
    h_b[i] = rand() / (float)RAND_MAX;
  }
  buffer d_a(h_a), d_b(h_b), d_c(h_c);
  celerity::distr_queue queue;
  queue.submit([&](execution_handle& cgh) {
    // data accessors
    auto a = d_a.get_access<acc::read>();
    auto b = d_b.get_access<acc::read>();
    auto c = d_c.get_access<acc::write>();
    // kernel
    cgh.parallel_for( count,
      celerity::kernel_functor(
        acc::one_to_one(a,b,c),
        [=](id<> item) {
          int i = item.get_global(0);
          c[i] = a[i] + b[i];
        }) );
  });
  // ... use result vector c ...
}
```

Listing 1. Vector addition example code.

[1]We adopt OpenCL terminology: a computing systems consists of a *host* processor (typically a CPU) and a number of *compute devices* (e.g., GPUs). A *kernel* is a program that executes on a device.

CELERITY provides an implementation of the SYCL standard with a distributed queue object, offering a transparent way to use the whole computing infrastructure with a single device queue. Lines 2, 14, 22 and 23 are the only difference between a usual SYCL code executed for a single GPU and the distributed environment provided by CELERITY; other concepts such as buffers and accessors, follow the same logic of a normal SYCL application.

III. Methodology

In order to allow the CELERITY runtime to transparently and effectively distribute tasks in a heterogeneous cluster, it needs to be aware of the fine-grained data dependencies induced by sub-ranges of a given kernel invocation. We propose a minimal set of API extensions to the base SYCL standard [13] which allows the user to supply this information. We believe that our proposed design combines a maximum of flexibility with a minimum of effort required to express common patterns.

a) CELERITY Kernel Functor.: The primary difference between a standard SYCL program and a proposed CELERITY program lies in the construction of kernel functors. While a SYCL kernel specifies the execution of individual work items, a `celerity::kernel_functor` additionally specifies a function describing the mapping from sub-ranges of execution to sub-ranges of buffer accessors, as shown in Listing 2. The parameters to this function describe an N-dimensional `offset` and an N-d `range` respectively, and any accessors – captured by reference – are adjusted to indicate the relevant sub-range. In the example, a direct one-to-one mapping is performed for `b`, while the range is extended by two elements for `a` as the kernel accesses a neighborhood.

```
1  auto a = d_a.get_access<acc::read>();
2  auto b = d_b.get_access<acc::write>();
3  auto f = celerity::kernel_functor(
4    [&](range<> offset, range<> range) {
5          // access specifier
6      a.access_range(offset-1, range+2);
7      b.access_range(offset, range);
8    },
9    [=](id<> item) {
10     b[i] = -a[i-1] + a[i]*2 - a[i+1];
11   }
12 );
```

Listing 2. CELERITY kernel functor example.

b) Access Specifier Generators.: As a major goal of CELERITY is providing a high-productivity environment for programming heterogeneous clusters, a set of *access specifier generators* will be provided. These higher-order functions generate access specifiers for common patterns, e.g. `one_to_one` for cases where each work item accesses the directly mapped element in the given buffer (of the same dimensionality), or `neighborhood<N>` to include a neighborhood of `N` elements in every dimension around each directly mapped element.

The generated access specifiers can be combined using an overloaded `&` operator, allowing a concise and intuitive formulation of Listing 2, as shown in Listing 3.

```
1  auto a = d_a.get_acc<acc::read>();
2  auto b = d_b.get_acc<acc::write>();
3  auto f = celerity::kernel_functor(
4    acc::neighborhood<1>(a) & acc::one_to_one(b),
5    [=](id<> item) {
6      b[i] = -a[i-1] + a[i]*2 - a[i+1];
7    }
8  );
```

Listing 3. Access specifier generators.

IV. Summary & Conclusion

Current approaches for programming heterogeneous clusters either rely on the user to handle partitioning and data dependencies [14], are targeted only at specific domains [15], or require a major re-engineering effort compared to industry standard approaches [16]. Conversely, the CELERITY API allows its system to manage partitioning for general programs while requiring only minimal extensions to SYCL code.

Acknowledgment

This work is supported by the D-A-CH project CELERITY, funded by DFG project CO1544/1-1 and FWF project I3388.

References

[1] "The top 500 list," November 2017. [Online]. Available: http://www.top500.org

[2] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[3] K. O. W. Group, "The opencl specification, version 2.0," Tech. Rep., 2014.

[4] MPI Forum, "Mpi: A message-passing interface standard," Knoxville, TN, USA, Tech. Rep., 1994.

[5] W. W. Carlson, J. M. Draper, and D. E. Culler, "Introduction to upc and language specification," Tech. Rep., 1999.

[6] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 1995, pp. 207–216.

[7] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the chapel language," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 291–312, Aug. 2007.

[8] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," in *Euro-Par Parallel Processing*, Aug. 2009.

[9] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: a proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, no. 2, pp. 173–193, 2011.

[10] H. C. Edwards and C. R. Trott, "Kokkos: Enabling performance portability across manycore architectures," in *Extreme Scaling Workshop (XSW)*. IEEE, 2013, pp. 18–24.

[11] R. Hornung, J. Keasler *et al.*, "The raja portability layer: overview and status," *Lawrence Livermore National Laboratory, Livermore, USA*, 2014.

[12] D. S. Medina, A. St-Cyr, and T. Warburton, "Occa: A unified approach to multi-threading languages," *arXiv preprint arXiv:1403.0968*, 2014.

[13] K. O. W. Group, "Sycl specification 1.2," Tech. Rep., 2015.

[14] J. Kim, G. Jo, J. Jung, J. Kim, and J. Lee, "A distributed opencl framework using redundant computation and data replication," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2016, pp. 553–569.

[15] Y. Zhang and F. Mueller, "Autogeneration and autotuning of 3d stencil codes on homogeneous and heterogeneous gpu clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 3, pp. 417–427, 2013.

[16] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly, "Dandelion: a compiler and runtime for heterogeneous systems," in *Proceedings of the ACM Symposium on Operating Systems Principles*, 2013, pp. 49–68.