Closing the Performance Gap with Modern C++

Thomas Heller[1,5](✉), Hartmut Kaiser[2,5], Patrick Diehl[3,5], Dietmar Fey[1], and Marc Alexander Schweitzer[3,4]

[1] Computer Science 3, Computer Architectures, Friedrich-Alexander-University, Erlangen, Germany
thom.heller@gmail.com

[2] Center for Computation and Technology, Louisiana State University, Baton Rouge, USA

[3] Institute for Numerical Simulation, University of Bonn, Bonn, Germany

[4] Meshfree Multiscale Methods, Fraunhofer SCAI, Schloss Birlinghoven, Sankt Augustin, Germany

[5] The STELLAR Group, Baton Rouge, USA
hpx-users@stellar.cct.lsu.edu
http://stellar-group.org

Abstract. On the way to Exascale, programmers face the increasing challenge of having to support multiple hardware architectures from the same code base. At the same time, portability of code and performance are increasingly difficult to achieve as hardware architectures are becoming more and more diverse. Today's heterogeneous systems often include two or more completely distinct and incompatible hardware execution models, such as GPGPU's, SIMD vector units, and general purpose cores which conventionally have to be programmed using separate tool chains representing non-overlapping programming models. The recent revival of interest in the industry and the wider community for the C++ language has spurred a remarkable amount of standardization proposals and technical specifications in the arena of concurrency and parallelism. This recently includes an increasing amount of discussion around the need for a uniform, higher-level abstraction and programming model for parallelism in the C++ standard targeting heterogeneous and distributed computing. Such an abstraction should perfectly blend with existing, already standardized language and library features, but should also be generic enough to support future hardware developments. In this paper, we present the results from developing such a higher-level programming abstraction for parallelism in C++ which aims at enabling code and performance portability over a wide range of architectures and for various types of parallelism. We present and compare performance data obtained from running the well-known STREAM benchmark ported to our higher level C++ abstraction with the corresponding results from running it natively. We show that our abstractions enable performance at least as good as the comparable base-line benchmarks while providing a uniform programming API on all compared target architectures.

M. Taufer et al. (Eds.): ISC High Performance Workshops 2016, LNCS 9945, pp. 18–31, 2016.
DOI: 10.1007/978-3-319-46079-6_2

1 Introduction

The massive local parallelism available on today's and tomorrow's systems poses one of the biggest challenges to programmers, especially on heterogeneous architectures, where conventional techniques require to develop and tune independent code bases for each of the separate parts of the machine. This paper focuses on how to address portability in terms of code and performance when developing applications targeting heterogeneous systems. More and more systems come online which consist of more than one hardware architecture, all of it made available to the developers through often independent and orthogonal tool-chains.

With the recently growing interest in the community in C++ and the increased activity towards making all of the available parallelism of the machine available through native C++ language features and library facilities, we see an increasing necessity in developing higher level C++ APIs which ensure a high level of portability of code while providing the best possible performance. At the same time, such APIs have to provide a sufficient amount of generality and flexibility to provide a solid foundation for a wide variety of application use cases. GPGPU vendors have started to make their C++ tool chains more conforming with the newest C++11/C++14 Standards [17], as demonstrated for instance by recent versions of NVidia's CUDA [6] or the newer HCC compiler [3] as provided by AMD. Unfortunately, there are no usable standards-conforming library solution available yet which would help in writing C++ code which is portable across heterogeneous architectures.

One of the key problems to solve while developing such higher level library abstractions is to provide facilities to control and coordinate the placement of data in conjunction with the location of the execution of the work on this data. We describe the result of our research in this direction, provide a proof of concept optimization, and present performance results gathered from comparing native implementations of the STREAM benchmark for OpenMP [16] and CUDA [9] with an equivalent application written based on our design. We show that there is essentially no performance difference between the original benchmarks and our results.

Our presented implementation of C++ algorithms is fully conforming to the specification to be published as part of the C++17 Standard [18]. It is based on HPX [13], a parallel runtime system for applications of any scale. For our comparisons with the native OpenMP and CUDA benchmarks we use the same sources demonstrating a high degree of portability of code and performance. The used parallel algorithms are conforming to the latest C++17 Standard and are designed to be generic, extensible and composable.

In the remaining part of this paper we describe related work (Sect. 2), talk about locality of data and work (Sect. 3), describe our implementations (Sect. 4), show the results (Sect. 5), and summarize our findings (Sect. 6).

2 Related Work

The existing solutions for programming accelerators mostly have in common that they are based either on OpenCL [5] or on CUDA [6] as their backends.

Table 1 shows an overview of the different approaches existing today. The most prominent in that regard are pragma based language extensions such as OpenMP and OpenACC [4]. The pragma solutions naturally don't offer good support for C++ abstractions. In order to get better C++ language integration, software has to directly rely on newer toolchains directly supporting C++, such as recent versions of CUDA, the newer HCC compiler [3], or SYCL [2].

Generic higher level abstractions are also provided by various library based solutions such as Kokkos [10], raja [12], Thrust [11], and Bolt [1]. Those attempt to offer higher level interfaces similar but not conforming to the parallel algorithms specified in the upcoming C++17-Standard [19]. One of the contributions of this paper is to provide standards-conforming implementations of those parallel algorithms combined with truly heterogeneous solutions enabling transparent locality control, a feature not available from the listed existing libraries.

In addition, we aim to provide a solution for all existing accelerator architectures, that is not limited to either OpenCL or CUDA based products providing a modern C++ programming interface.

Table 1. overview of different approaches: pragma based solutions, low level compiler and libraries to leverage different architectures.

Name	Type	Hardware support	
OpenMP	pragmas	cpu, accelerators	[5]
OpenACC	pragmas	accelerators	[4]
HCC	compiler	OpenCL, HSA	[3]
CUDA	compiler	CUDA	[6]
SYCL	compiler	OpenCL	[2]
Kokkos	library	OpenMP, CUDA	[10]
Raja	library	OpenMP, CUDA	[12]
Thrust	library	CUDA, TBB, OpenMP	[11]
Bolt	library	C++Amp, OpenCL, CPU	[1]

3 Locality of Work and Data

Modern computing architectures are composed of various different levels of processing units and memory locations. Figure 1 shows an example for such architectures that are a common in today's nodes for GPU accelerated supercomputers. Tomorrow's systems will be composed of even more complex memory architectures. In addition, when for instance looking at autonomous driving applications requiring a huge amount of processing power, the diversity of different processing units as well as different memory locations will increase.

In order to program these architectures efficiently it is important to place the data as close as possible to the site where the execution has to take place. As such, we need APIs that are able to effectively and transparently express the data placement on and data movement to concrete memory locations

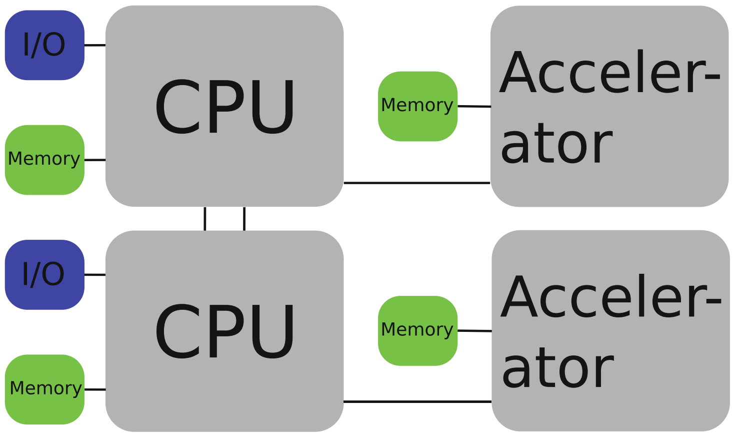

Fig. 1. This Figure shows an example of a typical heterogeneous architecture that is composed of multiple CPUs containing different physical blocks of memory as well as Accelerators and Network Interfaces with their own discrete memory locations, which are connected through a common bus.

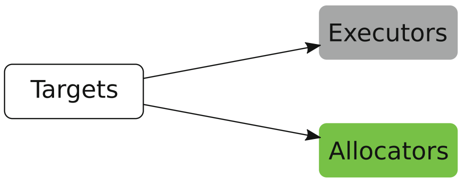

Fig. 2. This figure shows the relation between targets, memory allocation and work execution; A target is the link between co-locating memory allocation and execution of tasks close to the memory location. It is used to transparently express the notion of a 'place' in the system to both, allocation and execution.

(or places) in a system. We also need APIs allowing to coordinate the desired data placement with fine control over defining the execution site from where the code will access the data.

This paper proposes concepts and APIs that are rooted within the C++ language and Standard Library to create an expressive, performant, and extensible way to control locality of work and data by refining the *allocator* concept already defined in the C++ standard as well as using the proposed *executor* concept. These are tied together by defining *targets*, which represent places in a system, to properly co-locate placement of data and execution of work (see Fig. 2).

3.1 Defining Places in a System

In order to define a place in a system, or a *target*, we first need to evaluate the landscape of all available different targets. Examples for targets are: Sets of CPU cores, which can be used to solve NUMA related problems; Different memory areas, such as scratch pads, used to access high bandwidth or other special purpose memory; Accelerator devices, such as GPUs, which can be used to offload compute intensive tasks; Remote Processes in a distributed application; Other types of special purpose hardware, like FPGAs; etc.

Since all the examples for different targets given above have distinct use cases in terms of their ability to execute different tasks (data parallelism, code with many control structures) and different properties such as different address spaces or mechanisms to allocate memory as well as executing code, it becomes obvious that the definition of a target should be in the form of an concept that doesn't define the behavior directly, but rather is a handle to an opaque implementation defined place in the system. This does not require any additional constraints or specification for the concept itself, since an implementation is directly operating on target properties specific to a in place memory and to execute work. By not having those artificially imposed limitations, we allow for maximal flexibility by defining the required customization points wherever appropriate.

For supporting the richer functionality to dispatch memory placement and work execution, the implementation is based on the interfaces described in Sects. 3.2 and 3.3.

3.2 Controlling Memory Placement

After creating a concept for targets (see Sect. 3.1), we discuss the actual placement of memory. Placement of memory in general needs to first and foremost handle the allocation of data, but should also cover the transparent migration (movement) of data to different places of a given target family. For memory allocation purposes, we leverage the already widely used concept of the `Allocator`.

```
template <typename Allocator>
struct allocator_traits
  : std::allocator_traits<Allocator>
{
  typedef unspecified reference;
  typedef unspecified const_reference;

  typedef unspecified access_target;
  typedef typename access_target::target_type target_type;

  static target_type target(Allocator const& alloc);

  template <typename ...Ts>
  static void bulk_construct(Allocator& alloc, pointer p,
    size_type count, Ts &&... vs);

  static void bulk_destroy(Allocator& alloc, pointer p,
    size_type count) noexcept;
};
```

Listing 1.1. Extensions to `std::allocator_traits` to support efficient memory allocation and construction operations for targets as described in Sect. 3.1.

Allocators are already widely used within the C++ standard library (for example with containers or smart pointers) with the main purpose of encapsulating memory allocation. This allows for great reuse of the defined concepts in already existing code and serves our purpose of hiding memory allocation on opaque targets perfectly. For the sake of making memory allocations efficient on various targets, such as for discrete GPU memory or remote processes, we introduced backwards compatibly extensions. Listing 1.1 outlines the traits class which supports our extensions, the remaining interface follows `std::allocator_traits`. The extensions are optional, and fall back to the requirements for C++ standard allocators. The extensions introduced, serve the purpose to perform bulk construction and destruction of C++ objects. This is necessary to either avoid overheads of offloading the constructor or destructor code or to support first-touch policies (as used for ccNUMA architectures) efficiently.

The topic of transparent data migration is not covered within this paper and does not fall within the realm of memory allocation. Another mechanism would need to be created with appropriate customization points to support different target use cases. One example within the HPX runtime system is the migration of objects between different localities (where a locality is a HPX specific target).

3.3 Controlling Execution Locality

The previous sections described the mechanisms to define targets (Sect. 3.1) and memory allocation (Sect. 3.2). The missing piece, execution of work close to targets, is based on the `Executor` concept. Executors are an abstraction which define where, how, and when work should be executed, in a possibly architecture specific way (see also [7]).

```
template <typename Executor>
struct executor_traits
{
  typedef Executor executor_type;

  template <typename T>
  struct future { typedef unspecified type; };

  template <typename Executor_, typename F, typename ... Ts>
  static void apply_execute(Executor_ && exec, F && f,
    Ts &&... ts);

  template <typename Executor_, typename F, typename ... Ts>
  static auto async_execute(Executor_ && exec, F && f,
    Ts &&... ts);

  template <typename Executor_, typename F, typename ... Ts>
  static auto execute(Executor_ && exec, F && f,
    Ts &&...ts);
    template <typename Executor_, typename F, typename Shape,
        typename ... Ts>
    static auto
    bulk_async_execute(Executor_ && exec, F && f,
      Shape const& shape, Ts &&... ts);

    template <typename Executor_, typename F, typename Shape,
        typename ... Ts>
    static auto
    bulk_execute(Executor_ && exec, F && f,
      Shape const& shape, Ts &&... ts);
};
```

Listing 1.2. `std::executor_traits` to support efficient execution on targets as described in Sect. 3.1

Executors follow the same principle as `Allocators` in such that they are accessible through the trait `executor_traits` (see Listing 1.2), in a similar fashion to `allocator_traits`. It is important to note that an implementation for a given executor is not required to implement all functions as outlined but the traits are able to infer missing implementations. The only mandatory function an executor needs to be implement is `async_execute`. The remaining facilities are, if not provided otherwise, automatically deduced from that. However, it is important to note that architectures like GPGPUs benefit tremendously by implementing the bulk execution features.

Specific executor types are then specialized, architecture dependent implementations of the executor concept which use this architecture dependent knowledge to provide the target specific mechanisms necessary to launch asynchronous tasks. We introduce a selection of special purpose executors in Sect. 4.

3.4 Parallel Algorithms and Distributed Data Structures

Now that we have all the necessary ingredients to co-locate work and data, we are going to make it usable by providing a specialized implementation of a vector. This vector is exposing the same high level interface as `std::vector<T>`. This data structure encapsulates an array of elements of the same type and enables accessing the stored data element-wise, through iterators, and using other supporting facilities like resizing data, giving the user an abstraction over contiguous data using the API as described in Sect. 3.2.

The exposed iterators can be used directly with the parallel algorithms [14] already existing in HPX. Additionally, HPX's parallel algorithms allow us to pass executors (see Sect. 3.3) which will in turn execute the algorithm on the designated resources. By using compatible targets for both, the executor and the allocator, the co-location of tasks and data is guaranteed.

Listing 1.3 is providing an example that transforms the string "hello world" to all uppercase. Note that this example is omitting actual targets and specific allocators/executors which will be introduced in Sect. 4.

```
auto target = ...;

target_allocator<char> alloc(target);
vector<char, target_allocator<char>> s(
  {'h', 'e', 'l', 'l', 'o', 'w', 'o', 'r', 'l', 'd'},
  alloc);

target_executor exec(target);
transform(par.on(exec), s.begin(), s.end(),
    [](char c){ return to_upper(c); });
```

Listing 1.3. Hello world example using the introduced concepts `Target`, `Allocator` and `Executor`

4 Implementation

This Section describes specific implementations for the concepts we defined in Sect. 3 to demonstrate the feasibility of our claims. As a proof of concept we implemented special allocators and executors to support NUMA architectures as well as an allocator and various executors for CUDA devices.

4.1 Support for NUMA Aware Programming

Contemporary compute nodes nowadays usually consist of two or more sockets. Within this architecture, co-locating work and data is an important ingredient to leverage the full memory bandwidth within the whole system to avoid NUMA related bottlenecks and to reduce cross-NUMA-domain (cross-socket) memory accesses.

For this purpose of identifying cores, the target (see Sect. 3.1) is a numeric identifier for a specific CPU (see Fig. 3). To identify the various cores in the system we use hwloc [8]. This allows us to use bit-masks to define multiple CPUs as one single target, making up a convenient way to build targets for whole NUMA domains. For convenience, two functions are provided: `get_targets()` which returns a vector containing all processing units found in the system and `get_numa_domains()` which is returning a vector with targets, where each element in that vector represents the cpuset (bit-mask) identifying the respective NUMA domain. The targets returned from those functions can then easily be transformed as necssary, for instance to construct finer grained sets.

After having the targets defined to support our CPU based architecture we need a target specific allocator to support a vector of separate targets. As a proof of concept we chose to implement a block allocation scheme by dividing the number of bytes to be allocated evenly across the passed elements in the target vector.

The same scheme as described above is used to implement the executor. This ensures that work that is to be executed on data using the block allocator is

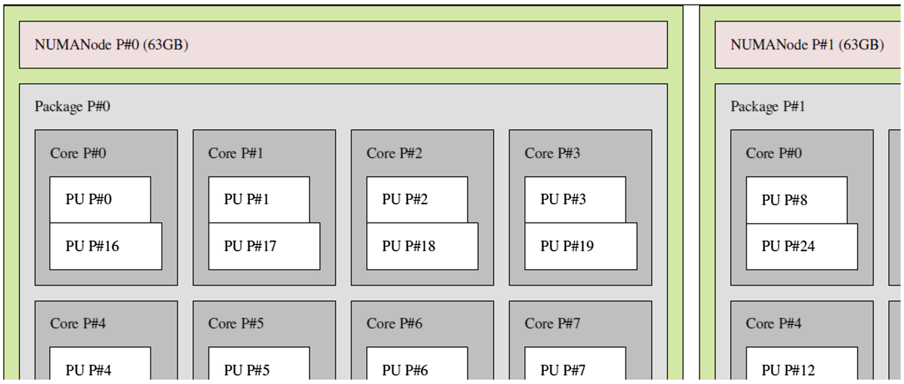

Fig. 3. Graphical output from `hwloc-ls` showing a two-socket Ivy Bridge system. The Figure shows the different identifiers for the different processing units and groups them in their respective NUMA domain.

co-located with the data. The cpusets in the associated targets are used as an affinity mask to pin the executed tasks to the cores accordingly.

4.2 Leveraging CUDA Based GPGPUs

Since more and more supercomputers are equipped with NVIDIA based GPGPUs as accelerators, this section will cover a prototypical solution based on CUDA.

Within the CUDA based implementation, the choice for how to define a target is determined by the underlying programming model. The devices are represented by numerical identifiers and in order to support asynchronous operations such as kernel invocations and memory copies, CUDA streams are used. That means a CUDA target is implemented as a wrapper for an integer representing the device and a CUDA stream attached to that device.

For memory placement, an `Allocator` (see Listing 1.1) are specialized to allocate memory on the given device and the (bulk) construct/destruct functions offload directly to the GPU. In terms of transparent memory access via references to a given object we introduce a special proxy object that allows to hide the implementation specific details on how to read and write memory and as such seamlessly supports the interfaces described in Sect. 3.4. For copying data between host and device, we extended the parallel copy algorithm to provide a internal specialization that is able to directly call the respective CUDA memcpy functions for maximum efficiency.

The executor support (see Sect. 3.3) is exploiting the dual compilation mode of CUDA's nvcc compiler and is therefore able to execute any callable that is marked with the CUDA specific `__device__` attribute. This gives great flexibility since code that is supposed to be offloaded needs to be, in theory, only marked with the `__device__` (in addition to `__host__`) attribute and can be used immediately with the new executor.

The executor itself implements all functions as outlined in Listing 1.2. For one, the implemented bulk execution facility ensures best-possible performance for executing kernels on an array of data-elements. Secondly, we implemented both, the synchronous and asynchronous versions of the executor interfaces as the synchronous versions can be implemented more efficiently than they could be generated by the traits class. The asynchronous versions additionally need to attach a callback to the stream in order to notify the returned future about the completion of the operation on the GPU.

In practice however, this way of programming leads to unforeseen problems during compilation since not all features of C++ are supported on devices and the need to mark up every function does not scale well especially for third party code. Newer compiler technologies such as hcc are more promising in this regard and a truly single source solution without additional mark up can be implemented there in the near future.

5 Results

For the sake of giving a first evaluation of our abstractions defined in the previous sections, we are using the STREAM Benchmark [16]. As a proof of concept, the presented APIs have been implemented with the HPX parallel runtime system, and have been ported to the parallel algorithms as defined in the newest C++ Standard [19] (see Listing 1.4).

```
template <typename Executor, typename Vector>
void stream(Executor& e, Vector const& as, Vector const& bs,
     Vector const& cs)
{
  double scalar = 3.0
  // Copy
  copy(e, as.begin(), as.end(), cs.begin());
  // Scale
  transform(e, cs.begin(), cs.end(), bs.begin(),
      [scalar](double c){ return c * scalar;});
  // Add
  transform(e, as.begin(), as.end(), bs.begin(), cs.begin(),
      [](double a, double b){ return a + b;});
  // Triad
  transform(e, bs.begin(), bs.end(), cs.begin(), as.begin(),
      [scalar](double b, double c){ return b + c*scalar;});
}
```

Listing 1.4. Generic implementation of the STREAM benchmark using HPX and C++ standards conforming parallel algorithms.

It is important to note that the benchmark is parameterized on the allocaor used for the arrays (vectors) and the given executor, which allows to design an portable implementation of the benchmark ensuring best possible performance across heterogeneous architecturs in plain C++. For any of the tested architectures, the used `Executor` and `Vector` is are using the same target. In our case, we use the NUMA target as defined in Sect. 4.1 and a CUDA target as described in Sect. 4.2. The test platform we use is a dual socket Intel Xeon CPU E5-2650v2 with 2.60 GHz together with a NVIDIA Tesla K40m GPU. As a reference implementation for the NUMA based benchmark, we used the original STREAM benchmark [15], The GPU version was compared with the CUDA based GPU-STREAM [9]. For the CPU based experiments, we used two sockets and 6 cores per socket, that is a total of 12 CPU Cores, which has been determined to deliver the maximal performance for the benchmark

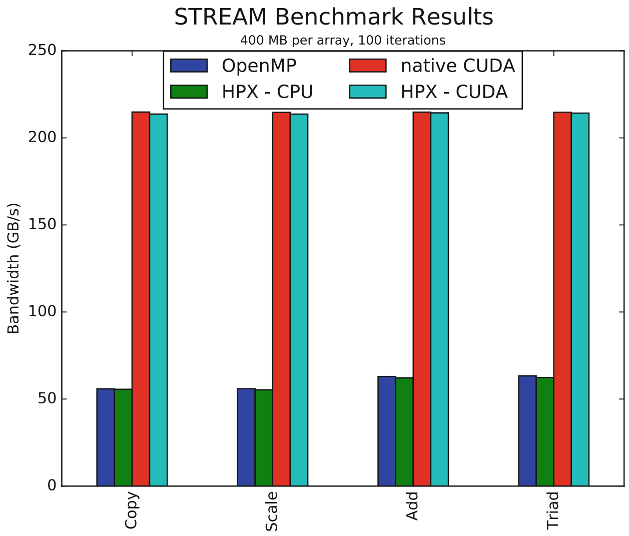

Fig. 4. Results for STREAM Benchmark on the host and GPU. The figure shows the resulting bandwidth achieved with the native implementations and the HPX port showed in Listing 1.4. The achieved performance for all tests is approximately the same for the respective architectures. The benchmark on the CPU used 2 NUMA domains with 6 cores each, the GPU version ran on a single Tesla K40m.

The results we obtained from running our benchmark show that the utilized memory bandwidth is essentially equivalent to that achieved by the native benchmarks. Figure 4 is showing results comparing to the respective reference implementations. What can be seen is that our CPU based implementation is about 1.1% slower than the reference and our CUDA implementation are about 0.4% slower. Figure 5 is giving an impression on the overheads involved with the parallel algorithms abstractions. For small array sizes, the overhead is

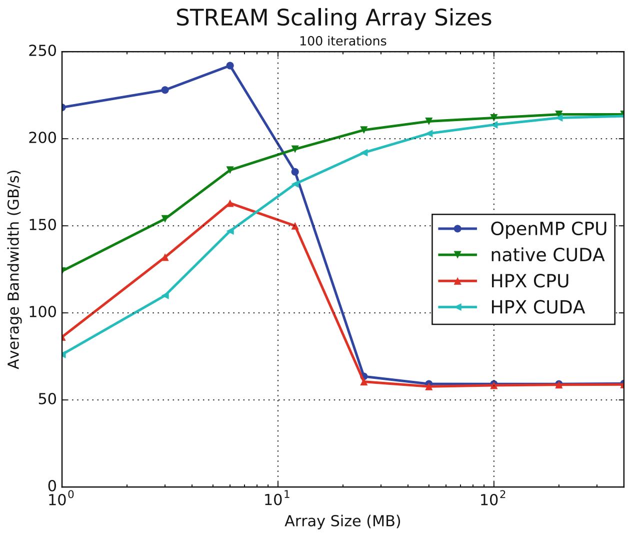

Fig. 5. Results for STREAM Benchmark on the host and GPU. This graph shows the average performance of the entire benchmark suite with varying input sizes, from 10 to 400 MB. While the native implementations provide lower overhead for small input sizes, all implementations converge to the same maximum. The benchmark on the CPU used 2 NUMA domains with 6 cores each, the GPU version ran on a single Tesla K40m.

noticeable within the HPX implementation, however, for reasonable large extents of the array, the implementations are almost similar again.

6 Conclusion

This paper presented a coherent design and implementation based on the foundation of the upcoming C++17 Standard and provided extensions to ensure locality of work and data. We showed that the performance of the introduced higher-level parallelism framework is does not significantly reduced compared to the performance of today's prevalent programming environments. The benefit of our presented solution is to provide a single source, generic, and extensible abstraction for expressing parallelism, together with no loss in performance.

For future work, we are going to extend the number of targets to include more support for different memory hierarchies (e.g. Intel Knights Landing High Bandwitdth Memory) as well as improving the support for GPGPU based solutions by implementing other back ends such as HCC and SYCL.

Acknowledgement. This work is supported by the NSF awards 1240655 (STAR), 1447831 (PXFS), and 1339782 (STORM), and the DoE award DE-SC0008714 (XPRESS) and by the European Union's Horizon 2020 research and innovation program under grant agreement No 671603.

References

1. Bolt C++ Template Library. http://developer.amd.com/tools-and-sdks/opencl-zone/bolt-c-template-library/
2. C++ Single-source Heterogeneous Programming for OpenCL. https://www.khronos.org/sycl
3. HCC: an open source C++ compiler for heterogeneous devices. https://github.com/RadeonOpenCompute/hcc
4. OpenACC (Directives for Accelerators). http://www.openacc.org/
5. OpenMP: a proposed Industry standard API for shared memory programming, October 1997. http://www.openmp.org/mp-documents/paper/paper.ps
6. CUDA (2013). http://www.nvidia.com/object/cuda_home_new.html
7. N4406: parallel algorithms need executors. Technical report (2015). http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4406.pdf
8. Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., Namyst, R.: hwloc: a generic framework for managing hardware affinities in HPC applications. In: PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing. IEEE, Pisa, Italy. https://hal.inria.fr/inria-00429889
9. Deakin, T., McIntosh-Smith, S.: GPU-STREAM: benchmarking the achievable memory bandwidth of graphics processing units. In: IEEE/ACM SuperComputing (2015)
10. Edwards, H.C., Trott, C.R., Sunderland, D.: Kokkos: enabling manycore performance portability through polymorphic memory access patterns. J. Parallel Distrib. Comput. **74**(12), 3202–3216 (2014). Domain-Specific Languages and High-Level Frameworks for High-Performance Computing
11. Hoberock, J., Bell, N.: Thrust: a parallel template library, vol. 42, p. 43 (2010). http://thrust.googlecode.com
12. Hornung, R., Keasler, J., et al.: The Raja portability layer: overview andstatus. Lawrence Livermore National Laboratory, Livermore, USA (2014)
13. Kaiser, H., Adelstein-Lelbach, B., Heller, T., Berg, A., Biddiscombe, J., Bikineev, A., Mercer, G., Schfer, A., Habraken, J., Serio, A., Anderson, M., Stumpf, M., Bourgeois, D., Grubel, P., Brandt, S.R., Copik, M., Amatya, V., Huck, K., Viklund, L., Khatami, Z., Bacharwar, D., Yang, S., Schnetter, E., Bcorde5, Brodowicz, M., Bibek, atrantan, Troska, L., Byerly, Z., Upadhyay, S.: hpx: HPX V0.9.99: a general purpose C++ runtime system for parallel and distributed applications of any scale, July 2016. http://dx.doi.org/10.5281/zenodo.58027
14. Kaiser, H., Heller, T., Bourgeois, D., Fey, D.: Higher-level parallelization for local and distributed asynchronous task-based programming. In: Proceedings of the First International Workshop on Extreme Scale Programming Models and Middleware, pp. 29–37. ACM (2015)
15. McCalpin, J.D.: Stream: sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, Charlottesville, Virginia (1991–2007), a continually updated Technical report. http://www.cs.virginia.edu/stream/
16. McCalpin, J.D.: Memory bandwidth and machine balance in current high performance computers. IEEE Comput. Soc. Tech. Committee Comput. Archit. (TCCA) Newsl. **59**, 19–25 (1995)

17. The C++ Standards Committee: ISO International Standard ISO/IEC 14882: 2014, Programming Language C++. Technical report, Geneva, Switzerland: International Organization for Standardization (ISO) (2014). http://www.open-std.org/jtc1/sc22/wg21
18. The C++ Standards Committee: N4578: Working Draft, Technical Specification for C++ Extensions for Parallelism Version 2. Technical report (2016). http://open-std.org/JTC1/SC22/WG21/docs/papers/2016/n4578.pdf
19. The C++ Standards Committee: N4594: Working Draft, Standard for Programming Language C ++. Technical report (2016). http://open-std.org/JTC1/SC22/WG21/docs/papers/2016/n4594.pdf