Using SYCL as an Implementation Framework for HPX.Compute

Marcin Copik
RWTH Aachen University, Aachen, Germany
mcopik@gmail.com

Hartmut Kaiser
Center for Computation and Technology,
Louisiana State University, Baton Rouge, USA

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

hkaiser@cct.lsu.edu

ABSTRACT

The recent advancements in High Performance Computing and ongoing research to reach Exascale has been heavily supported by introducing dedicated massively parallel accelerators. Programmers wishing to maximize utilization of current supercomputers are required to develop software which not only involves scaling across multiple nodes but are capable of offloading data-parallel computation to dedicated hardware such as graphic processors. Introduction of new types of hardware has been followed by developing new languages, extensions, compilers and libraries. Unfortunately, none of those solutions seem to be fully portable and independent from specific vendor and type of hardware.
HPX.Compute, a programming model developed on top of HPX, a C++ standards library for concurrency and parallelism, uses existing and proposed C++ language and library capabilities to support various types of parallelism. It aims to provide a generic interface allowing for writing code which is portable between hardware architectures.
We have implemented a new backend for HPX.Compute based on SYCL, a Khronos standard for single-source programming of OpenCL devices in C++. We present how this runtime may be used to target OpenCL devices through our C++ API. We have evaluated performance of new implementation on graphic processors with STREAM benchmark and compare results with existing CUDA-based implementation.

CCS CONCEPTS

• **Computing methodologies → Parallel programming languages**; • **Computer systems organization →** *Single instruction, multiple data*; • **Software and its engineering → Language types**;

KEYWORDS

C++, HPX, SYCL, parallel programming, heterogeneous programming, GPGPU

ACM Reference format:
Marcin Copik and Hartmut Kaiser. 2017. Using SYCL as an Implementation Framework for HPX.Compute. In *Proceedings of IWOCL '17, Toronto, Canada, May 16-18, 2017,* 7 pages.
https://doi.org/http://dx.doi.org/10.1145/3078155.3078187

1 INTRODUCTION

In the last few years, HPC world has moved to more heterogeneous architectures, exploiting hardware offering different types of parallelism and enforcing specific programming models. It has become a standard practice to offload computations to dedicated accelerators and it is expected that importance of massively parallel processors is going to increase. Up to now, we have not seen a standardized approach in C++ which would allow for writing portable code for heterogeneous systems.

HPX.Compute[6] is an attempt to solve this problem basing on standard C++ language and library. Proposed extensions to parallelism in C++17 has been implemented and extended to create a system which not only allows to control data placement and execution through a standard-conforming C++ API but is also orthogonal to compilers, libraries and vendor-specific extensions. The triple *target*, *allocator* and *executor* define execution model suiting widely different types of hardware. Compute is based on HPX[9], a parallel runtime system for applications of any scale, exposing an API conforming to C++ standard.

This paper presents an initial evaluation of SYCL[16] toolchain as an implementation of HPX.Compute API. SYCL allows for single source programming of OpenCL devices in C++, including graphic processors. The increasing importance of data-parallel accelerators makes it very important to properly support GPU programming in C++.

We discuss similar attempts and competing solutions in sec 2. The mapping between Compute and SYCL concepts is presented in section 3 and we describe problems faced so far in section 4, explaining possible workarounds and suggesting future extensions. Section 5 presents performance evaluation and overheads analysis on STREAM benchmark, followed by a summarization of results and plans for future work in section 6.

2 TECHNOLOGIES

HPX.Compute implements a CUDA[12] backend for NVIDIA graphic processors with compilers nvcc and LLVM-based gpucc[19]. Although graphic cards offered by NVIDIA have gained huge popularity in HPC applications, there have been approaches to develop competing standards, with OpenCL as the most popular alternative choice. The ability to support other vendors have been evaluated so far with two novel standards: Heterogenous Computing(HC)[2], an AMD initiative based on C++AMP[11], and SYCL. Our previous work[4] verified and compared those standards on OpenCL-supported graphic cards but it is no longer possible due to removed support for OpenCL devices in HCC[3], so far the only compiler for HC.

Our work on STL algorithms may be compared with Boost's Compute[15], AMD Bolt[1] and NVIDIA Thrust[7], but none of them define their API in terms of standard C++. Data allocation and movement requires dedicated containers. In OpenCL-based approaches, such as Boost library, user-defined functions and lambda expressions require specific approach because OpenCL compiler generates device code on runtime from a textual kernel representation. The only other solution using C++17 execution policies are parallel algorithms in ParalellSTL[17], a SYCL-based project which also requires a non-standard data movement to device.

3 IMPLEMENTATION

This section gives an overview how a new backend implements HPX.Compute facilities with SYCL. Fundamental concepts, assumptions, and full interfaces have been already fully described[6]; here a short summary with simplified code samples are presented.

HPX.Compute expresses data and execution locality through *targets*. It is an abstract concept representing a place in the system for both data allocation and execution. A large and rich set of hardware which may be represented as a target does not allow to define detailed interface.

Our target for SYCL backend communicates with the device through a `sycl::queue`. An immediate consequence is implicit concurrency of separate targets referring to the same device. Device selection is currently limited to providing an index of a device. At least one SYCL device must be present in the system, otherwise an exception is thrown. SYCL host device cannot be used due to restrictions created by current implementation of future, as explained in details later.

Both CUDA and SYCL backends define two important functionalities in their target interface, presented in Listing 1. Synchronization itself is a trivial call to underlying `sycl::queue`. Target synchronization is not affected by tasks pushed to the device through other targets which happen to refer to the same SYCL device and utilize different queues; such behaviour is consistent with CUDA backend. Future implementation in SYCL target is a more complex problem and has been described in details in section 4.2.

```
//Blocks until target is ready
void synchronize();
//Future is ready when all tasks allocated on target
     have been finished
hpx::future<void> get_future() const;
```

Listing 1: A generic interface of target.

3.1 Data locality

HPX.Compute resolves the problem of standard-conforming data allocation on a device by defining a proper allocator type. Compute allocators used for CUDA or NUMA targets are quite straightforward due to already existing API representing allocated data as pointers. SYCL can not be integrated within HPX.Compute trivially because it is a runtime handling memory locality on its own. Buffers are not physically tied to any device and runtime guarantees that data is going to be present on a device before it executes a kernel accessing this buffer. To minimize problems coming from building a runtime on top of another runtime, we have decided to not interfere with SYCL buffer management and we only keep a record of a logical connection between target and data chunks allocated in their runtime.

Construction and deconstruction of objects on graphic processors can not be performed efficiently with `std::allocator_traits`. This problem has been resolved by introducing bulk versions for these two operations. When bulk versions are not available in an allocator, traits can always use the default interface. A part of extended `std::allocator_traits` interface is presented in Listing 2. First function is supposed to enqueue a kernel using placement new to construct multiple objects, destroy is responsible for calling destructor. Additional template parameter *Name* is used as a kernel name.

```
template<typename Allocator>
struct allocator_traits
 : std::allocator_traits<Allocator>
{
 template<typename Name, typename ... Ts>
 static void bulk_construct(Allocator & alloc, pointer p
      , size_type count, Ts &&... vs);

 template<typename Name>
 static void bulk_destroy(Allocator & alloc, pointer p,
      size_type count) noexcept;
}
```

Listing 2: Extensions to the C++ standard library allocator model.

3.2 Execution

The third and final part of HPX.Compute model is executor, a concept introduced and discussed for next revisions of C++ standard[8]. Executors define where and how tasks should be executed, implementing the most efficient mapping of work to a specific type of hardware. In HPX, an executor is required to implement only `async_execute` functionality and other synchronous or bulk execution are callable through `executor_traits` but this approach is not compatible with execution model of graphic processors. Listing 3 gives an example of basic SYCL executor interface. In our implementation, Shape container should embed triples consisting of iterator, offset and chunk size.

One of the problems with single source accelerator programming in C++ is restrictions on functions executed on the device. SYCL inherits restrictions from OpenCL and current specification tends to be slightly more restrictive than CUDA where recursion, function pointers or virtual functions are permitted. Those kinds of limitations are not relevant for parallel algorithms but should be taken into account for generic C++ programming of accelerators. Many constraints are perfectly justified by hardware limitations,

```
template<typename Allocator>
struct default_executor
 : hpx::parallel::executor_tag
{
 target target_;

 template <typename Parameters, typename F, typename
      Shape, typename ... Ts>
 void bulk_launch(Parameters &&, F && f, Shape const&
      shape, Ts &&... ts) const;

 template <typename Parameters, typename F, typename
      Shape, typename ... Ts>
 std::vector<hpx::future<void>> bulk_async_execute(
      Parameters &&, F && f, Shape const& shape, Ts
      &&... ts) const;
};
```

Listing 3: Basic interface of SYCL dedicated executor.

but a prime example of constraint, which may not be necessary and is very problematic, is marking functions as capable of running on a device. Other standards, such as CUDA or HC, define new keywords or use attributes to help compilers determine which function should be compiled to device code. Contrary to CUDA and HC, SYCL does not require additional marking and in our opinion it makes much easier to integrate accelerator programming into C++ projects.

3.3 Algorithms

So far we we have evaluated few algorithms which are obvious candidates for parallelization due to lack of synchronization and dependencies: `for_each`, `transform`, `copy`. In HPX.Compute the `copy` algorithm is the standard way of transferring data between host and device. Internal implementation is specialized for iterators representing data allocated with different backends to perform an actual copy to the device.

An example of an algorithm which has been evaluated and tested with CUDA, but it is not supported with SYCL backend, is `for_loop`[14], an index-based parallel loop. We believe it is a very useful addition to existing STL algorithms because it allows to easily implement code requiring random access to memory, ability to pass multiple memory objects which do not have to be iterated in a regular fashion or the knowledge of current loop index. Those requirements can be found in many applications, e.g. stencil code or linear algebra operations. The current implementation in HPX requires transferring to device a complex structure implemented with `std::tuple`, which is an obstacle as explained in section 4.4. A tuple is required to encapsulate a sequence of user-defined loop iterators. While it is technically possible to perform tuple deconstruction in command group and reconstruct the object in kernel scope, it would require a rather complicated, large and very static conversion procedure which would have to be adapted to each modification of `for_loop` implementation.

Another addition to algorithm interface are `projections`, simple unary functions performing transformations of data represented by input iterators. The idea is similar to `adaptive_iterator` implemented in Boost.Compute.

3.4 Device accessors

An important distinction between previous and new backends is the way in which data is managed on the host side and accessed on a device. Contrary to CUDA-based applications, where data allocation functions return pointers which are valid addresses on a device, SYCL introduces a concept of accessors created in command group scope, before kernel execution. The memory model with implicit movement and unspecified location of data could not be implemented without the knowledge which buffers are going to be read or modified in the kernel.

A lot of examples present device accessors in static environments where executed kernel and accessed buffers are known a priori. It is not the case for libraries such as parallel algorithms in HPX, where data objects are encapsulated and executor interface is very generic. Listing 4 presents the structure of objects representing device memory on the host side. An underlying SYCL buffer is extracted from an iterator and a device accessor to this buffer is created in the command group scope. A SYCL `global_ptr`, obtained from the device accessor, serves the role of device iterator for an algorithm execution. A specialization of `iterator_traits` for SYCL device pointer type had to be implemented.

```
template<typename T>
struct target_ptr
{
  sycl::buffer<T, 1> * ptr_;
  std::size_t pos_;
};

template<typename T>
struct compute_iterator
{
  target_ptr<T> * ptr_;
};
```

Listing 4: HPX.Compute SYCL objects representing device memory.

To complicate things even more, executor can expect more than iterator passed for execution, using `hpx::zip_iterator`. As long as the set of all possible input types is limited and rather small, it is possible to implement a static conversion algorithm inside command group scope, which transforms host side input into similar objects containing not buffers, but accessors. Another consequence is that all functions executed inside kernel have to be generic and not dependent on types used on the host. A similar problem has been observed in other applications, such as the transformation of expression tree with nodes referring to device data[13].

For dynamic problems structure of objects passed to kernel should not be encoded in the executor. Iterator, function objects and other types can expose API to perform an automatic transformation. A visitor object would give access to procedure switching from a buffer to an accessor.

4 ISSUES

This section describes features which we were not able to implement with APIs provided by SYCL, we were not able to implement in a standard-compliant way or we were not able to implement at all. We believe that those issues are not HPX specific and may appear in other programming models in HPC. Recommended changes may help to express clearly what capabilities would make the integration process as simple as possible or just possible.

4.1 Kernel naming

SYCL technical specification allows for two approaches in compiler design: an integrated two-pass compiler generating both host and device code or an offline device compiler. Our experience with other technologies suggests that latter approach does not suffer from problems such as a huge increase in compilation time and a restriction on the whole project to use only the dedicated C++ compiler. However, it creates a previously unknown problem: host compiler needs to know how to link kernel invokation in C++ application with device bytecode. So far the only known solution for kernels represented as a lambda functions is to explicitly name the kernel because tools already offered by standard C++ does not allow for automatic name generation[18]. This responsibility has to be shifted from library to the user and it becomes problematic as soon as we consider STL algorithms which do not require passing a user-define function.

To solve this problem, ParallelSTL uses *named execution policies* to pass kernel name for execution. We consider this solution problematic for two reasons. Firstly, execution policy defines how the algorithm can be executed and what dependencies may exist between executions on separate data items. There is little logical connection between execution policy and kernel name which seems to be tied not even to underlying hardware target, but to a single execution context on this hardware. Our further doubts are based on the main goal of this project which is to provide a vendor independent and standard conforming implementation. The new `named` and already known `parallel` execution policies would represent the same type of parallelized execution which would create unnecessary confusion. Thus, we have decided to find another solution in HPX.Compute.

Our proposition for this problem is based on executor parameters, an HPX extension to proposed executor concepts for next parallelism update in C++ standard. They have been used so far to control chunking in an OpenMP fashion or measure execution time. We have defined a new type of parameter `hpx::parallel::kernel_name` as presented in Listing 5. In such scenario, the executor needs to know only the type of passed parameters.

```
// uses default executor: par
hpx::parallel::for_each(
 hpx::parallel::par.with(
  hpx::parallel::kernel_name<class Name>()
 ),
 ...
);
```

Listing 5: Example presenting a parallel execution of `for_each` algorithm with a user-defined name.

Another example of kernel name appearing in standard interfaces is allocator defined in section 3.1. Allocators tend to be used multiple times in STL containers for construction and deconstruction of objects and each single action requires enqueuing a SYCL kernel.

Our experience suggests that the restriction of naming kernels needs to be lifted for development of standard-compliant solutions on top of SYCL.

4.2 Asynchronous communication

Asynchronous execution is a crucial component of HPX programming model, allowing to hide latencies. It is realized by joining events happening in the system with actions triggered by this event and their immediate continuations. Execution model on graphic processors is implicitly asynchronous and SYCL exposes that by not blocking after enqueing a command group. Although it is possible to execute asynchronously on a device, there is no standard way of notifying host about device status. Continuation is allowed only as defining a kernel executing after another kernel through data dependencies in accessors. Restricting ourselves to using functionalities provided only by SYCL would permit only one implementation of `hpx::future` which is launching a new HPX thread and waiting there until allocated tasks have finished.

We have resolved, at least partially, this issue by making use of SYCL-OpenCL interoperability. Several SYCL types function as wrappers around OpenCL objects and access methods are exposed. Among those types is the command queue which is sufficient to use OpenCL callback mechanism. Listing 6 presents the process of adding a new callback triggered after finishing all tasks which are already in a queue. Usage of `boost::intrusive_ptr` ensures that `future_data`, state of hpx::future, is not destroyed prematurely and callback can be always executed safely.

```
// future_data is a shared state of hpx::future
cl::sycl::queue queue = ...;
future_data * ptr = ...;
cl_event marker;
clEnqueueMarkerWithWaitList(queue.get(), 0, nullptr, &
     marker);
clSetEventCallback(marker, CL_COMPLETE,
 [](cl_event, cl_int, void * ptr) {
  marker_callback(
   static_cast<future_data*>(ptr)
  );
 },
 ptr);
```

Listing 6: Adding a callback to `sycl::queue`. Please note that the sample ignores error checking for OpenCL operations. Pointer is not captured, but passed as a void pointer to allow a conversion of lambda expression to function pointer.

This approach has drawbacks and we do not consider it to be a final or model solution. We are aware that it relies on the quality of OpenCL runtime and HPX may be hurt by additional overhead when an implementation of callback mechanism is far from being perfect. After all, there is no guarantee that it is not limited to launching a new thread and performing a blocking wait for completion of OpenCL event. Section 5 includes a performance comparison between synchronous and asynchronous execution in the benchmark.

A hidden assumption made here is that OpenCL queue is always available. This requirement is not fulfilled in a `sycl::queue` created on the SYCL host device. Hence SYCL backend can not operate in

systems where OpenCL devices are not available.
This limitation could be removed by extending host queues with a mock OpenCL queue which implements only critical features such as callbacks. However, we believe that in the long run, the best solution is to extend SYCL queues with callbacks or futures.

4.3 Data transfers

We have not been able to find an efficient method of copying data from a SYCL buffer to user-defined pointer in memory, given that the user may be interested in accessing only part of the buffer. Offset and size, two typical parameters of copy, can be emulated by using ranged sub-buffers. Host accessor, a standard SYCL approach for accessing a buffer or sub-buffer on the host, always creates an intermediate copy in SYCL runtime. This trouble is avoided by using `map_allocator` for buffer construction, but it simply moves a temporary placement from SYCL runtime to pointer used by the allocator which can not be changed.

Asynchronous transfers between host and device is another feature which is missing in SYCL, comparing to OpenCL. Overlapping data movement and kernel execution have been advised by both major vendors as an important optimization for applications where performance is crucial and PCIe bandwidth may be a bottleneck.

Listing 7 presents a suggestion how copy functionalities could be exposed in SYCL. Range-based access may be implemented in buffer API, as presented in listing 8. Return type indicates a desired property of asynchronous data movement and chaining transfers with other events in the system. However, we are aware that it may not possible to implement those features around OpenCL events.

```
// copy all contents of buffer
template<typename T, int N, typename OutIter>
sycl::event copy(const sycl::buffer<T,N>& src, OutIter
     dest);

// copy range [begin, end) to buffer, fully replacing
     its contents
template<typename InIter, T, int N>
sycl::event copy(InIter begin, InIter end, sycl::buffer<
     T,N> & dest);
```

Listing 7: Data movement between a generic storage in memory and SYCL buffer.

```
// write range to buffer starting at 'pos'
template<typename T, int N, typename InIter>
sycl::event sycl::buffer<T,N>::write(
 std::size_t pos, InIter begin, InIter end
);

// read 'size' elements starting at 'pos'
template<typename T, int N, typename OutIter>
sycl::event sycl::buffer<T,N>::read(
 size_t pos, size_t size, OutIter dest
);
```

Listing 8: Data movement including user-defined size and offset for accessing `sycl::buffer`.

4.4 Non-standard layout datatypes

SYCL restrictions on datatypes which may be transferred to a device are quite strict when compared to other standards. Limiting permitted types in buffers and objects captured in device lambda to C++11 standard layout prohibits from transferring `std::tuple` from host to device code. All popular tuple implementations, including ones from GNU libstdc++, libc++ or HPX, violate standard layout rules by multiple inheritance with more than one parent class containing non-static member fields. Known methods of implementing a standard-layout tuple are rather tricky and inefficient. Furthermore, introducing a special version of the tuple may require modifying existing implementation to not use `std::tuple`.

The tuple is a quite recurring problem in HPX but we can not exclude similar problems appearing in future. We are planning to solve this problem by explicitly performing a serialization before capturing objects in kernel scope. This solution is based on implementing serialization and deserialization functionalities in user-defined types. It is not an uncommon solution, current version 1.2 of HCC resolves problem of non-standard layout datatypes by adding serialization procedures during compilation. Then HC runtime can safely assume that each object transferred to device is serializable.

5 RESULTS

Our first implementation of SYCL backend has been evaluated with the STREAM benchmark[10]. A comparison of bandwidth obtained through HPX parallel algorithms and standalone SYCL implementation, provided by GPU-STREAM[5], gives a clear view whether there is an overhead when performing the same task from HPX and how significant it is.

Our testing platform was an AMD GPU R9 Nano with a memory bandwidth of 512 GB/s. Results have been gathered from 100 runs with first timing ignored. A SYCL compiler ComputeCpp in version: CommunityEdition-0.1.1 has been used for both measurements. All four functions have been enqueued with `sycl::nd_range` and local work size equal to 256. `sycl::range` has not been used due to technical problems appearing inside HPX.
We compare the slowdown to results from CUDA backend running NVIDIA Tesla K40m GPU, as presented in [6]. Direct comparison of SYCL and CUDA backends on the same device is not possible at the present time. SYCL kernels can only run on devices supporting SPIR, an intermediate language for OpenCL kernels, and all CUDA-enabled devices are lacking this feature.

Results presented on figure 1 show that SYCL backend is about 2.1% slower than native implementation, with 3.2% worst slowdown on copy functionality. New backend creates larger overhead on AMD devices than CUDA implementation showing 0.4% slowdown on NVIDIA GPUs. Taking into consideration performance concerns of asynchronous callbacks, as described in section 4.2, we include results obtained when futures are disabled and no callbacks are created in OpenCL runtime[1]. Comparing to purely synchronous execution, average slowdown drops to about 0.6% with a worst slowdown of 1.8%, again in copy function.

[1]Obviously, STREAM benchmark involves sequences of purely synchronous executions. However, HPX usually implements synchronous algorithms as a call to asynchronous implementation immediately followed by waiting on futures.

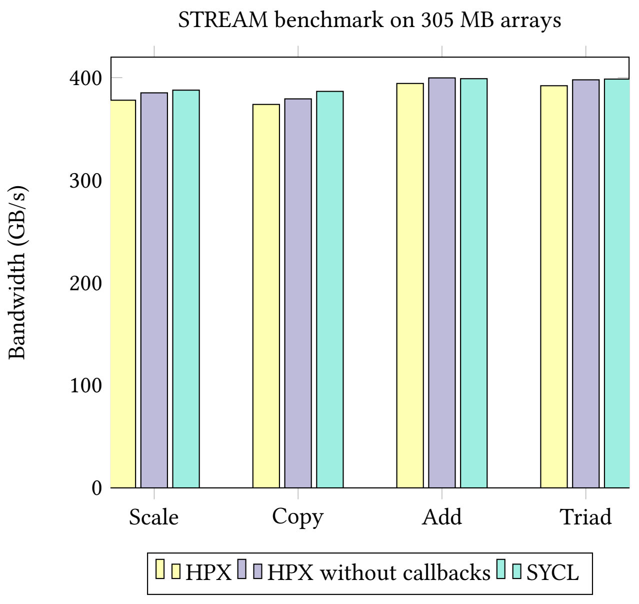

Figure 1: Results of STREAM benchmark with two different implementations based on SYCL. The benchmark run on a single AMD GPU R9 Nano. Best results obtained from 100 iterations.

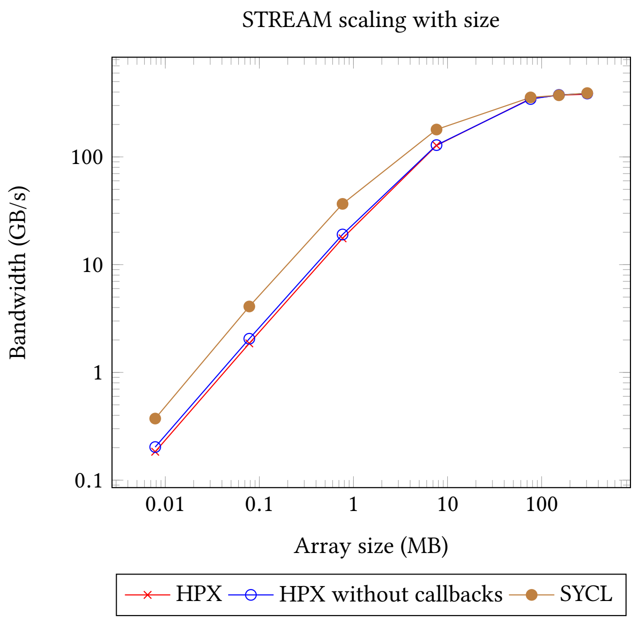

Figure 2: Results of STREAM benchmark scaling from very small to larger arrays, from less than 1 MB to 305 MB. The benchmark run on a single AMD GPU R9 Nano. Best results obtained from 100 iterations.

Figure 2 gives an insight into overheads created by HPX for different array sizes. Difference is quite significant for small array sizes, resulting in almost 55% slowdown in worst case. For larger array sizes time spent on execution becomes large enough to hide latencies introduced in HPX. The results seem to correspond with measurements obtained on CUDA backend.

6 CONCLUSION

So far we have been able to implement most of features related to Compute. Full integration requires SYCL capabilities which seem to not be supported yet in available compilers. Reaching our next goals involves solving the issue of transferring non-standard layout types to the device, such as std::tuple. An implementation of `for_loop` would vastly enhance capabilities of GPU-parallelized STL algorithms and we can not proceed with it unless C++ tuple is supported in host-device data movement. Further work may be concentrated on optimizing algorithms for GPU execution and on an interface to access on-chip memory which would be compatible across different standards. In our opinion, such feature is necessary to offer generic kernels which are not only standard-conforming but can compete with other tools in terms of performance.

Another important step is to bring support for multiple devices to SYCL backend and evaluate performance with segmented algorithms.

Comparing to other standards evaluated for HPX.Compute, SYCL has its own advantages and disadvantages. A higher-level memory management runtime may find its use in applications working with multiple devices but it may hurt runtime and libraries which want to explicitly perform data allocation and movements, such as HPX. The memory model could not be implemented without accessors which enforce manual conversion of complex types representing device data on the host side. On the other hand, SYCL has made an important step forward for better integration with C++ by doing a proper analysis of source code to determine which functions should be built with device compiler, removing a need for additional and non-standard markups.

There is no single standard for single-source C++ accelerator programming which would support all major vendors offering graphic processors for HPC systems. Neither HC nor SYCL can compete now with CUDA backend due to lack of support for OpenCL and HSA features on CUDA-enabled devices. Our results so far give an indication that SYCL may be capable of serving as a standard backend for HPX.Compute with graphic processors supporting OpenCL's intermediate representation SPIR.

ACKNOWLEDGMENTS

We would like to express our gratitude to Codeplay Software's *ComputeCpp* development team for support, suggestions, and discussions on integrating SYCL with HPX.

REFERENCES

[1] AMD. Bolt C++ Template Library, version 1.3, 2015. https://github.com/HSA-Libraries/Bolt.
[2] AMD. Heterogeneous Computing C++ API, 2016. https://scchan.github.io/hcc/.
[3] AMD. HCC: An open source C++ compiler for heterogeneous devices, 2016. https://github.com/RadeonOpenCompute/hcc/.
[4] M. Copik. HPX and GPU parallelized STL. C++Now, 2016. URL https://cppnow2016.sched.org/event/6SfU.
[5] T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith. *GPU-STREAM v2.0: Benchmarking the Achievable Memory Bandwidth of Many-Core Processors Across Diverse Parallel Programming Models*, pages 489–507. Springer International Publishing, Cham, 2016. ISBN 978-3-319-46079-6. doi: 10.1007/978-3-319-46079-6_34.

URL http://dx.doi.org/10.1007/978-3-319-46079-6_34.

[6] T. Heller, H. Kaiser, P. Diehl, D. Fey, and M. A. Schweitzer. *Closing the Performance Gap with Modern C++*, pages 18–31. Springer International Publishing, Cham, 2016. ISBN 978-3-319-46079-6. doi: 10.1007/978-3-319-46079-6_2. URL http://dx.doi.org/10.1007/978-3-319-46079-6_2.

[7] J. Hoberock and N. Bell. Thrust: A parallel template library. *Thrust: A Parallel Template Library*, 2009.

[8] J. Hoberock, M. Garland, and O. Girioux. N4406 Parallel Algorithms Need Executors. Technical report, 2015. URL http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4406.pdf.

[9] H. Kaiser, B. Adelstein-Lelbach, T. Heller, A. BergÃĺ, J. Biddiscombe, A. Bikineev, G. Mercer, A. SchÃďfer, J. Habraken, A. Serio, M. Anderson, M. Stumpf, D. Bourgeois, P. Grubel, S. R. Brandt, M. Copik, V. Amatya, K. Huck, L. Viklund, Z. Khatami, D. Bacharwar, S. Yang, E. Schnetter, Bcorde5, M. Brodowicz, Bibek, atrantan, L. Troska, Z. Byerly, and S. Upadhyay. hpx: HPX V0.9.99: A general purpose C++ runtime system for parallel and distributed applications of any scale, July 2016. URL https://doi.org/10.5281/zenodo.58027.

[10] J. D. McCalpin. Stream: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, Charlottesville, Virginia, 1991-2007. URL http://www.cs.virginia.edu/stream/. A continually updated technical report. http://www.cs.virginia.edu/stream/.

[11] Microsoft Corporation. C++ AMP Open Specification V1.2. Technical report, 2013.

[12] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, Mar. 2008. ISSN 1542-7730. doi: 10.1145/1365490.1365500. URL http://doi.acm.org/10.1145/1365490.1365500.

[13] R. Potter, P. Keir, R. J. Bradford, and A. Murray. Kernel composition in sycl. In *Proceedings of the 3rd International Workshop on OpenCL*, IWOCL '15, pages 11:1–11:7, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3484-6. doi: 10.1145/2791321.2791332. URL http://doi.acm.org/10.1145/2791321.2791332.

[14] A. D. Robison, P. Halpern, R. Geva, and C. Nelson. P0075r1 Template Library for Parallel For Loops. Technical report, 2016. URL http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0075r1.pdf.

[15] J. Szuppe. Boost.compute: A parallel computing library for c++ based on opencl. In *Proceedings of the 4th International Workshop on OpenCL*, IWOCL '16, pages 15:1–15:39, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4338-1. doi: 10.1145/2909437.2909454. URL http://doi.acm.org/10.1145/2909437.2909454.

[16] The Khronos Group. SYCL Provisional Specification Version 2.2. Technical report, 2016.

[17] A. Vilches and R. Reyes. Syclparallelstl: A parallel stl library for heterogeneous systems. 1st SYCL Programming Workshop, 2016. URL http://ppopp16.sigplan.org/event/sycl-2016-papers-syclparallelstl-a-parallel-stl-library-for-heterogeneous-systems.

[18] M. Wong, A. Richards, M. Rovatsou, and R. Reyes. P0236R0 Khronos's OpenCL SYCL to support Heterogeneous Devices for C++. Technical report, 2016. URL http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0236r0.pdf.

[19] J. Wu, A. Belevich, E. Bendersky, M. Heffernan, C. Leary, J. Pienaar, B. Roune, R. Springer, X. Weng, and R. Hundt. Gpucc: An open-source gpgpu compiler. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, pages 105–116, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3778-6. doi: 10.1145/2854038.2854041. URL http://doi.acm.org/10.1145/2854038.2854041.