2018 IEEE/ACM 4th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)

Integration of CUDA Processing within the C++ library for parallelism and concurrency (HPX)

Patrick Diehl, Madhavan Seshadri, Thomas Heller, Hartmut Kaiser

Abstract—Experience shows that on today's high performance systems the utilization of different acceleration cards in conjunction with a high utilization of all other parts of the system is difficult. Future architectures, like exascale clusters, are expected to aggravate this issue as the number of cores are expected to increase and memory hierarchies are expected to become deeper. One big aspect for distributed applications is to guarantee high utilization of all available resources, including local or remote acceleration cards on a cluster while fully using all the available CPU resources and the integration of the GPU work into the overall programming model.
For the integration of CUDA code we extended HPX, a general purpose C++ run time system for parallel and distributed applications of any scale, and enabled asynchronous data transfers from and to the GPU device and the asynchronous invocation of CUDA kernels on this data. Both operations are well integrated into the general programming model of HPX which allows to seamlessly overlap any GPU operation with work on the main cores. Any user defined CUDA kernel can be launched on any (local or remote) GPU device available to the distributed application.
We present asynchronous implementations for the data transfers and kernel launches for CUDA code as part of a HPX asynchronous execution graph. Using this approach we can combine all remotely and locally available acceleration cards on a cluster to utilize its full performance capabilities. Overhead measurements show, that the integration of the asynchronous operations (data transfer + launches of the kernels) as part of the HPX execution graph imposes no additional computational overhead and significantly eases orchestrating coordinated and concurrent work on the main cores and the used GPU devices.

Index Terms—Asynchronous many task systems (ATM), CUDA, parallelism, concurrency, HPX

1 INTRODUCTION

The biggest disruption in the path to exascale will occur at the intra-node level, due to severe memory and power constraints per core, many-fold increase in the degree of intra-node parallelism, and to the vast degrees of performance and functional heterogeneity across cores. The significant increase in complexity of new platforms due to energy constraints, increasing parallelism and major changes to processor and memory architecture, requires advanced programming techniques that are portable across multiple future generations of machines [1]. This trend has already manifested itself for some time in the domain of accelerator and co-processor boards.

It is well known that a large part of the available compute power of a machine (in terms of FLOPS) today is provided by various accelerators and co-processors. Especially general purpose GPUs however require special programming languages and techniques. Unfortunately, those devices are architecturally not too well integrated with the main cores. This requires special effort from the programmer in terms of managing a heterogeneous code-base, having to explicitly manage the data transfer to and from the devices and the execution of special kernels. In order for this scheme to be scaleable, special care is required to a) keep the main cores busy while kernels are being executed on a GPU, and b) hide the latencies and overheads of data transfers behind useful work. In short, the currently available solutions make it very hard to achieve *scalability*, *programmability*, and *performance portability* for applications running on heterogeneous resources.

In this paper we will focus on a technique and programming environment, which overcomes part of the above mentioned problems by transparently enabling asynchronous data transfer and kernel execution for CUDA, while still being able to concurrently execute tasks on the main cores in a seamless way. All GPU operations are represented as asynchronous tasks similar to any other parallel task run on a main core. This facilitates an easy way to express dependencies which is a critical precondition for managing parallel execution graphs in the HPX framework [2].

The presented solution not only transparently facilitates the hiding of latencies of data transfers to and from accelerators and the asynchronous execution of compute kernels, it also provides a framework for load balancing work across the system over large amount of (possibly remote) accelerator cards. For the kernels themselves, the solution still relies on the proven CUDA technology of the existing and widely used programming environments for GPUs. We do however expose the events generated by CUDA as C++ *future* objects to the user (see Section 3.1) which enables to integrate the data transfer and the execution of the kernels with the overall parallel execution flow on the main cores.

This paper makes the following contributions to the C++ library for parallelism and concurrency:

- *P. Diehl, T. Heller, and H. Kaiser were with the Center for Computation and Technology, Louisiana State University, LA, US. E-mail: P. Diehl see https://orcid.org/0000-0003-3922-8419*
- *M. Seshadri was with the Nanyang Technological University, Singapore*
- *T. Heller was with the Department of Computer Science, Friedrich-Alexander-University of Erlangen-Nürnberg, Germany*
- *H. Kaiser was with the Department of Computer Science, Louisiana State University, LA. USA.*
- *P. Diehl, T. Heller, and H. Kaiser were at the Ste||ar group*

978-1-7281-0178-1/18/$31.00 ©2018 IEEE
DOI 10.1109/ESPM2.2018.00006

- HPXCL provides an API for transparently enabling asynchronous data transfer and kernel execution for CUDA,
- all API functions return a C++ *future* object, which can be used within the synchronization mechanics provided by HPX for the integration in its asynchronous execution graph,
- the CUDA specific elements, *e.g.* blockSize, threadSize, and the kernel code, are not hidden from the user which allows the easy integration of existing CUDA code into the HPX framework.

The remainder of this paper is structured as follows: In Section 2 the related work is presented. In Section 3 HPX's details, which are used for the integration, are briefly introduced. In Section 4 examples for accessing all remote and local CUDA devices and the complete work flow for data transfers and launching a kernel are shown. Section 5 shows the overhead measurements compared to a native CUDA implementation. Finally, Section 6 concludes this presented approach.

2 Related work

This section provides a brief overview of related approaches for the integration of CUDA processing. For unifying the data transfer between different compute nodes via the Message Passing Interface (MPI), Nvidia provides CUDA-aware MPI [3]. Here, Unified Virtual Addressing (UVA) is utilized to combine the host memory and device memory of a single node into one virtual address space. Thus, pointers to data on the device can be handled by MPI directly and can be integrated in the MPI message transfer pipeline. However, the synchronization of kernel launches and data transfer is not addressed here.

The Chapel programming language [4] provides *parallel* and *distributed* language features for parallel and distributed computing. Thus, there is a distinct separation of parallelism and locality in this programming model. In addition, the specific functionality of the acceleration card is hidden from the user trough parallel feature of the language.

HPX.Compute [5] is fully compatible to the C++ 17 standard N4578 [6] and implements the triple define execution model: *targets*, *allocator* for memory allocation purposes, and *executors* for specifying when, where, and how the work is executed. HPX.Compute supports CUDA devices.

HPX.Compute SYCL [7] provides a new back end for HPX.Compute utilizing SYCL, a Khronos standard for single-source programming of OpenCL devices.

Kokkos [8] provides abstractions for parallel code execution and data management for CUDA, OpenMP, and Pthreads via a C++ library. Similar to CUDA-aware MPI, it does support a specific memory space (CudaVMSpace) and allocation within this space are accessible from host and device. The computational bodies (kernels) are passed to Kokkos via function objects or lambda function to the parallel executor.

The Phalanx programming model [9] provides a unified programming model for heterogeneous machines on a single node. For multiple nodes of a distributed-memory cluster the GASNet [10] run time is utilized. This model provides its interface of generic and templated functions via C++ template library. The implementation of the kernel function is identical for the CUDA and OpenMP backend by providing an abstraction level and the asynchronous launches of the kernels are synchronized using events.

TABLE 1
Summary of the different approaches for the integration of CUDA. The second column lists the technologies provided by each approach, the third column indicates the type of the approach, and the last column provides the reference.

Name	Technology	Type	Ref
Chapel	CUDA,Xeon Phi	Language	[4]
CUDA-aware MPI	MPI,CUDA	Lib	[3]
HPX.Compute	HPX,CUDA	C++ Lib	[5]
HPX.Compute SYCL	HPX,OpenCL	C++ Lib	[7]
Kokkos	CUDA,OpenMP	C++ Lib	[8]
Phalanx	CUDA,OpenMP, GASNet	C++ Lib	[9]
RAJA	OpenMP,CUDA	C++ Lib	[11]
Thrust	CUDA,OpenMP	C++ Lib	[12]
X10	CUDA	Language	[13]

RAJA [11] introduces its fundamental concept for separating the loop body from the loop transversal by introducing the *forall* feature where a sequential code block is defined. By providing the execution policy and a *indexset* the separation is modeled. Its concept is used in physics code, where plenty of for loops are used to do multi physics simulations.

Thrust [12] is based on the Standard Template Library (STL) and CUDA::thrust is an extension for providing CUDA devices and provides a set of common parallel algorithms. CUDA::thrust tries to hide CUDA specific language features by providing a similar API to the C++ API.

The programming language X10 [13], [14] provides an open-source tool chain, which translates the X10 code to C++. As the underlying programming model APGAS is used on the GPU and the CPU. Thus, the CUDA threads are defined as APGAS activities by specifying a first loop over the CUDA blocks and a second loop over the CUDA threads. Within the second for loop the sequential code for each CUDA thread is defined. The philosophy here was to use existing X10 constructs to hide the CUDA specific semantics.

Table 1 summarizes all these different approaches. Two different type of approaches can be seen in the related work. First, the approaches of providing a programming language are different from the presented approach, since the C++ library for parallelism (HPX) is extended for the integration of CUDA. Second, the library-based approaches are more similar to HPXCL, but all these approaches try to hide the CUDA specific language features as much as possible from the user. HPXCL instead allows the user to define CUDA specific feature, *e.g.* provide kernel functions and specify block and thread sizes. Therefore, HPXCL focuses on the approach to integrate existing CUDA kernels in the asynchronous execution graph of HPX. For hiding CUDA specific language features the HPX.Compute framework is more suitable.

3 HPX's basics

The HPX compute language (HPXCL) [15] is an extension for the Open Source C++ library for parallelism and con-

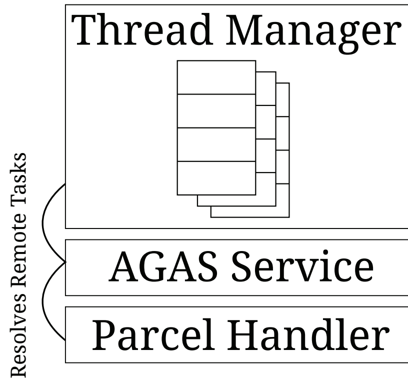

Fig. 1. Run time components of HPX (Thread manager,Active Global Address Space (AGAS) Service, and Parcel Handler), which resolve the remote tasks. The thread manager deals with the light-weight user level threads and provides a high-level API.The Active Global Address Space (AGAS) Service provides Global IDs to each allocated objects for hiding the explicit message passing. The parcel handler provides the communication between different nodes via remote procedure call / Active Messaging. Adapted from [2].

currency (HPX) [2]. The asynchronous many tasks (AMT) programming paradigm provided by HPX is extended with the asynchronously data transfer from the host to device (and vice versa) and asynchronously kernel launches. The synchronization between the tasks on a CUDA devices and CPUs is realized by the concept of futurization. We briefly review the main components of HPX, which are utilized for the integration of asynchronous tasks and synchronization. For more details about HPX we refer to [2], [16], [17]. Figure 1 shows the three main components of HPX, which resolve the remote tasks.

– The *Thread Manager* [18] deals with the light-weight user level threads and provides a high level API. Within HPX pre-defined scheduling policies are defined: 1) *static* means that one queue is attached to one core, 2) *thread local* which is HPX's default scheduling policy and means one queue is attached to one core, but in addition to the static scheduling policy, task stealing from neighboring cores on the same node is enabled and 3) *hierarchical* means that there is a tree of queues and new tasks are attached to the root of the tree and move down when a core fetches new work. There is always the possibility to define application specific scheduling policies, but HPXCL uses the *static* one.

– *Active Global Address Space (AGAS) Service* [17] supports the distributed computing. Each object within AGAS is represented by its Global ID (GID) to hide explicit message passing. Thus, its address is not bound to a specific locality on the system and its remote or local access is unified. This feature allows us to provide the same API for a local or remote CUDA device in our environment.

– *Parcel Service* [18], [19] For the communication between different nodes in the cluster environment, HPX is utilization remote procedure call (RPC) / Active Messaging. In the terminology of HPX an active message is a so-called Parcel which provides calls to remote nodes in a C++ fashion. For the communication between nodes either the tcp protocol or the Message passing Interface (MPI) is used.

3.1 Futurization

The API exposed by HPXCL is fully asynchronous and returns a `hpx::future`. The *future* is an important building block in the HPX infrastructure. By providing an uniform asynchronous return value, percolation as discussed in this paper is tightly integrated with any other application written in HPX and allows writing *futurized* code by employing the standard-conforming API functions such as `hpx::future<T>::then` and `hpx::when_all<T>` for composition, and `hpx::dataflow` to build implicit parallel execution flow graphs [20]. This allows for a unified continuation based programming style for both, regular host code and device code, and is an excellent tool to close the architectural gap between classic GPUs and CPUs.

4 Design of the implementation

Figure 2 shows the basic structure and relation between the classes. The user facing API is represented by client side objects referencing the object in AGAS via its Global ID (GID). This approach offers two benefits: a) Both copying and passing the object to different localities are now transparent in regard to the object pointing to either a remote or local object. b) They are also transparent in regard to where the actual representation of the object resides, i.e. is the data on the locality where it is needed.

For example, if a buffer or kernel is created for a specific device, the client side object only references the actual memory. Once those objects are used, the operation is completely local to the accelerator and the associated kernel is executed where the data lives. The device code to be executed is compiled just-in-time, that is, each accelerator action is available in source code or compatible binary form and can be sent and executed on the respective device, after it has been compiled for a specific device.

This represents an implementation of *percolation*, which allows data and code to be freely moved around in the (possibly) distributed system using the parcel service. The data is moved between the node using either tcp or MPI. The functions exposed by this API are inherently asynchronous (*i.e.* return a *future* object representing its return value) and therefore allows to naturally overlap unavoidable latencies when compiling kernels, copying data, or executing device functions. Each of these functions is attached to an light-weight user level thread using the *static* scheduling policy.

– A *device* is the logical representation of an accelerator and defines the functionality to execute kernels, create memory buffers, and to perform synchronization. HPXCL exposes functionality to discover local and remote devices within a system. Each device is equipped with its own, platform dependent asynchronous work queue, while still allowing cross-device synchronization through HPX's *future* API.

– A *buffer* represents memory which is allocated on a specific *device*. The operations defined on a buffer are related to copying data from and to the host system and between different devices. While the content itself is not directly addressable through AGAS, the asynchronous copy functionality is which allows effective memory exchange between different entities of a possible distributed memory system. The copy functions return futures which can be

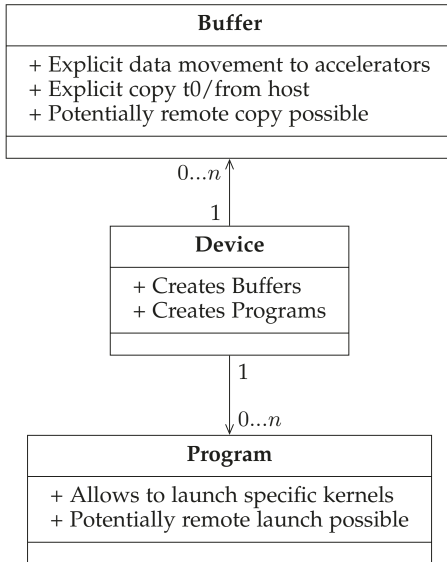

Fig. 2. Class diagram of Buffer, Device and Program and the functionality provided by each class. The device represents a logical device on a remote or local locality. A program which handles the compilation and the potential remote launches of kernels. The memory of a device is represented by a buffer. Adapted from [21].

used as dependencies to either kernel calls and additionally allows for naturally overlapping of communication with computation.

– A *program* serves as the compiled form of code that is to be executed on a specific device. By executing kernels, buffers need to be supplied as arguments. Executing a kernel returns a *future* object as well. Those futures can also be used to express data flow dependencies from memory copy operations or other previous calls to kernels.

4.1 Access of local and remote devices

Listing 1 shows the one line of source code to discover local and remote devices within a system. The method `get_all_devices` takes two arguments (the major and minor compute capability) and returns a `std::vector` with all available devices `hpx::cuda::device` having at least a compute capability as specified. The method returns a future and therefore `.get()` has to be called to receive the content of this future. Note that all device objects have the same API independent of the device is a remote or a local device.

4.2 Workflow of HPXCL

Listing 2 shows the work flow for running a CUDA kernel for computing the sum of n elements and stores the result in one variable. In Line 2 all available devices within the cluster environment are collected. In Line 4–12 the data on the host is allocated. In Line 14 the first device in the list which can be either a remote or local device is selected to run the computation. In Line 17–22 three buffers are generated which means that internally a `cudaMalloc` is called. After the creation of each buffer the data is copied into this buffer which means internally a `cudaMemcpyAsync` is

Listing 1. Gathering all remote and local CUDA devices on the cluster having CUDA compute capability of at least 1.0. Note that the function call returns a *future* and calling the `.get()` function return the content of this future.

```
int hpx_main(int argc, char* argv[]) {

//Get list of available CUDA Devices.
std::vector<hpx::cuda::device> devices =
        hpx::cuda::get_all_devices(1,0).get();

return hpx::finalize();
}
```

done and the future of this function calls are stored in a vector `futures` for later synchronization. In Line 25 the CUDA kernel is loaded from the file *kernel.cu*, the run time compilation of the kernel is started using NVRTC - CUDA Runtime Compilation, and the future is added to the vector of futures. In Line 27–31 the configuration of the kernel launch is defined. In Line 33–36 the arguments of the CUDA kernel are defined. In Line 38 the compilation of kernel and the copy from the data to the CUDA device have to be finished to execute the kernel properly. Therefore, a barrier with `hpx::wait_all` is introduced to assure that all these dependencies are finished. In Line 40 the kernel is executed and finally in Line 42 the result of the execution is copied to the host using `cudaMemcpyAsync`.

Note that we use native CUDA functionality to synchronize the asynchronous CUDA function calls, but we hide this from the user by returning a future object. This allows the users to combine these tasks with tasks on the CPU within the cluster environment in a unified fashion. In addition, the usage of remote or local devices has the same unified API and HPXCL internally copy the data to the node where the data is needed.

5 Overhead measurements

The unified API and utilizing HPX as an additional layer introduces some overhead. To measure the which is introduced by additional layer of HPX, the same benchmark is done by using native CUDA. Therefore, the native CUDA functions call used in each method and there synchronization were analyzed and were re-implemented without using the additional layer of HPX. For all benchmarks the same kernel, block size, and thread size were used. Note that for these measurements, the focus is on the overhead introduced by HPXCL and not on the optimization of the kernels for obtaining the optimal performance. The authors are aware that several highly optimized benchmark suites are available. However, comparing against these suites would not measure the overhead introduced by HPXCL, since the memory bandwidth and the computational throughput are measure for a kernel [22]. Since we use the same kernel, the device performance would not change significantly and instead the overall end to end performance can be compared.

In the Appendix: the software, the hardware, the operating system and drivers, and compilers are listed in detail, to enhance the reader’s ability to easily reproduce these

Listing 2. Workflow of a simple application.

```
1         // Get list of available Cuda Devices.
2         std::vector<device> devices = get_all_devices(2, 0).get();
3         // Allocate the host data
4         unsigned int* input;
5         unsigned int* n;
6         unsigned int* res;
7         cudaMallocHost((void**)&input, sizeof(unsigned int)* 1000);
8         cudaMallocHost((void**)&result,sizeof(unsigned int));
9         cudaMallocHost((void**)&n,sizeof(unsigned int));
10        memset (input,1,1000);
11        result[0] = 0;
12        n[0] = 1000;
13        // Create a device component from the first device found
14        device cudaDevice = devices[0];
15        // Create a buffers and copy data into them
16        std::vector<hpx::lcos::future<void>> futures;
17        buffer outbuffer = cudaDevice.create_buffer(SIZE * sizeof(unsigned int)).get();
18        futures.push_back(outbuffer.enqueue_write(0, SIZE * sizeof(unsigned int), inputData));
19        buffer resbuffer = cudaDevice.create_buffer(sizeof(unsigned int)).get();
20        futures.push_back(resbuffer.enqueue_write(0,sizeof(unsigned int), result));
21        buffer lengthbuffer = cudaDevice.create_buffer(sizeof(unsigned int)).get();
22        futures.push_back(lengthbuffer.enqueue_write(0,sizeof(unsigned int), n));
23        // Compile the CUDA Kernel
24        program prog = cudaDevice.create_program_with_file("kernel.cu").get();
25        futures.push_back(prog.build("sum"));
26        // Prepare the configuration of the kernel
27        hpx::cuda::server::program::Dim3 grid;
28        hpx::cuda::server::program::Dim3 block;
29        grid.x = grid.y = grid.z = 1;
30        block.x = 32;
31        block.y = block.z = 1;
32        // Set the arguments of the kernel
33        std::vector<hpx::cuda::buffer>args;
34        args.push_back(outbuffer);
35        args.push_back(resbuffer);
36        args.push_back(lengthbuffer);
37        // Synchronize the copy of data to the device and the compilation of the kernel
38        hpx::wait_all(data_futures);
39        //Run the kernel at the default stream
40        prog.run(args,"sum",grid,block).get();
41        //Copy the result back
42        unsigned int* res = resbuffer.enqueue_read_sync<unsigned int>(0,sizeof(unsigned int));
```

benchmarks on their own cluster environment. In order to alleviate start-up times, we repeated the algorithm for 11 iterations and took the mean execution time out of the last ten iterations, the first iteration was considered to be the warm-up (meant to be ignored). For the corresponding details, we refer to the appendix.

5.1 Single device

For the single devices overhead measurements, a Nvidia Tesla K40 and Nvida Tesla K80 in to different compute nodes (bahram,reno) were utilized. For each benchmark the identical kernel was executed using the native CUDA implementation and the equivalent HPX implementation.

5.1.1 *Stencil kernel (Intel Parallel Research Kernels)*

This benchmark, where a 3-point stencil $s(x_i) := 0.5x_{i-1} + x_i + 0.5x_{i+1}$ is computed for a vector $X := \{x_i, \ldots, X_n | x_i \in \mathbb{R}\}$ has been defined within the Intel Parallel Research Kernels (PRK) [23] developed by Intel labs. The block size was one and the thread size was 32 for all benchmarks. The aim of this benchmark is to test the computation and synchronization of the application. Figure 3 shows the execution time vs. length of the vector for the K40 (black lines) and for the K80 (blue lines). In both cases, the HPX implementation is $\approx$ 28% faster than the native CUDA implementation and the trend of the two lines is nearly linear. Thus, overlapping computation and data transfer utilizing futures has helped reduce the overall compute time. Note, that in this benchmark the CUDA code was executed

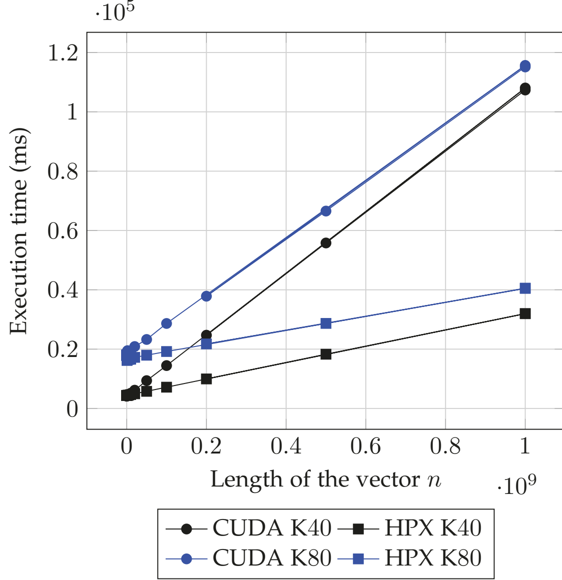

Fig. 3. Comparison for the overhead introduced by the additional layer of HPXCL for the stencil benchmark. The measurements were performed on a single Tesla K40 and a single K80. The native CUDA implementation is compared against the HPX implementation using one CPU.

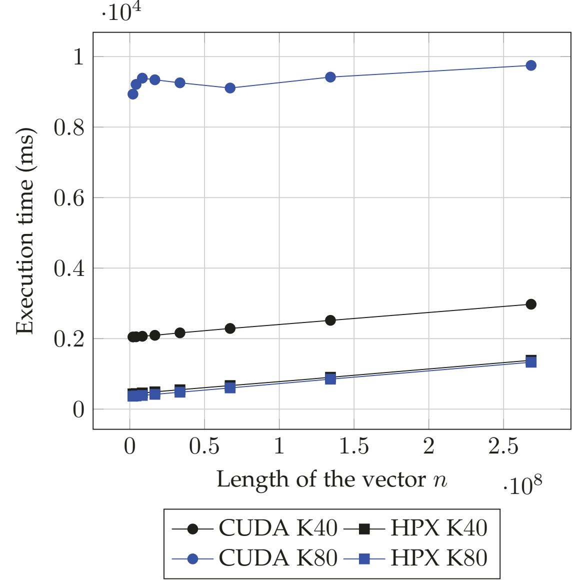

Fig. 4. Comparison for the overhead introduced by the additional layer of HPXCL for the partition benchmark. The measurements were performed on a single Tesla K40 and a single K80. The native CUDA implementation using asynchronous function calls and synchronization utilizing streams is compared against the HPX implementation using one CPU.

sequentially and the HPX code was using the asynchronous functionality within the CUDA SDK. In the next benchmark, the asynchronous functionality will be used in the native CUDA implementation as well.

5.1.2 Partition example

This benchmark focuses on the asynchronous data transfer and the efficient overlapping between communication and computation. The native CUDA implementation was adopted from [24] and asynchronous function calls are used in both implementations. In this benchmark, a kernel $k(x_i) := \sqrt{\sin^2 i + \cos^2(i)}$ is computed for a vector $X := \{x_i, \ldots, X_n | x_i \in \mathbb{R}\}$. The length of the vector is given by $n = 2^m * 1024 * blockSize * p$, where $m = \{1, 2, \ldots, 7, 8\}$, $blockSize = 256$, and $p = 4$ is the amount of partitions. The vector is divided in p partitions and each partition is asynchronously copied to the CUDA device, the kernel k is executed, and the result is asynchronously copied back to the host, see Algorithm 1. CUDA streams are used for the synchronization for the native CUDA implementation and the HPX implementation. Figure 4 shows the execution

Algorithm 1 Multiple Partitions Benchmark

Init X
for $i = 0\,; i < p\,; {+}{+}i$ **do**
  cudaMemcpyAsync(X_i,cudaMemcpyHostToDevice)
end for
for $i = 0\,; i < p\,; {+}{+}i$ **do**
  Apply kernel k to partition X_i
end for
for $i = 0\,; i < p\,; {+}{+}i$ **do**
  cudaMemcpyAsync(R_i,cudaMemcpyDeviceToHost)
end for

time vs. length of the vector for the K40 (black lines) and for the K80 (blue lines) using four partitions. In both cases, the HPX implementation is $\approx 4\%$ faster than the native CUDA implementation and the trend of the two lines is nearly linear. It is clearly shown that the speed-up of HPX is reduced by a factor of ≈ 4 when the native CUDA implementation utilizes asynchronous function calls. This benchmarks shows that the overhead introduced by HPXCL is negligible, even when the CUDA kernel is compiled at run time, for large enough vector sizes.

5.1.3 Mandelbrot example using concurrency with CPUS

The Mandelbrot set is a set of complex numbers for which c does not diverge from 0 for the function $f_c(z) = z^2 + c$ when integrated from zero. The pixels are then colored based on how rapidly the function value diverges from zero. In this example the Mandelbrot set for increasing images sizes is computed on a $K80$ GPU using HPXCL and saved to the file system as a PNG image. Figure 5 shows the computation time (blue line) when the image is synchronously written after the computation. The black line shows the computational time when the concurrency with the CPU of the HPXCL framework is used and the image is written asynchronously using `hpx::async` to the file system. Using the concurrency with the CPU decreases the computational time and this feature is beneficial for example to write results to the file system and asynchronously compute the next iteration.

5.2 Multiple devices

For the multiple device partition benchmark two Nvidia Tesla K80 cards containing two sub cards in one node (bahram) were utilized. Note, that each K80 has a dual-GPU design and therefore, 2×2 GPUs are available.

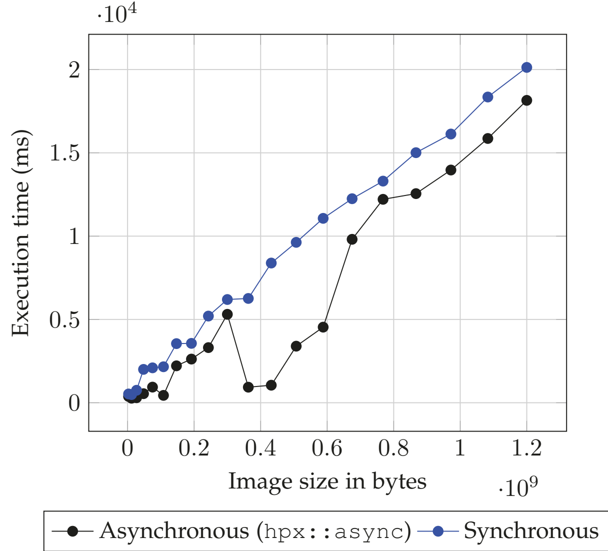

Fig. 5. Comparison of synchronous and asynchronous writing the resulting image of the Mandelbrot set to the file system. Both measurements were done on one K80 card using HPXCL.

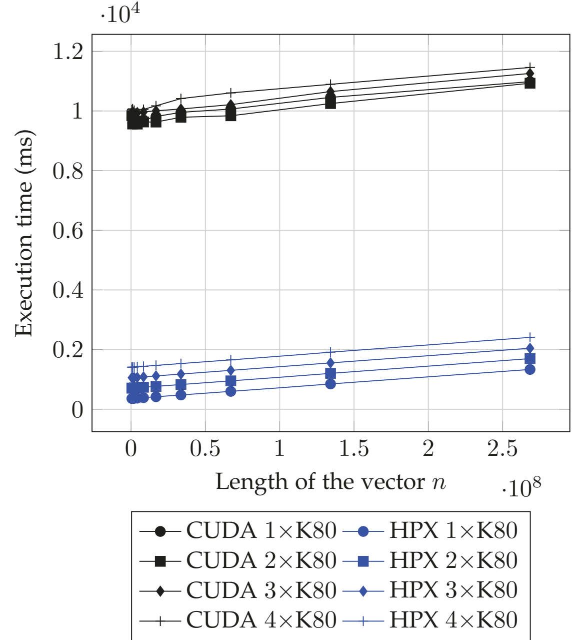

Fig. 6. Comparison for the overhead introduced by the additional layer of HPXCL for the partition benchmark on multiple devices. The vector is sliced in $1, 2, 3$, and 4 partitions and each partition is handled by one of the K80 cards.

5.2.1 *Partition example*

The benchmark describes in Section 5.1.2 is modified, such that each partition of the vector is handled by one of the Nvidia Tesla K80 cards in the bahram node. The vector with the length $n = 2^m * 1024 * blockSize$, where $m = \{1, 2, \ldots, 7, 8\}$ and $blockSize$ is 256, is sliced in $1, 2, 3$, and 4 partitions and each partition is handled by one of the K80 cards. The black lines show the execution time for the native CUDA implementation and the blue lines for 1 up to 4 K80 devices, see Figure 6. For the native CUDA implementation a increase of computation time is seen when going from one physical card to two physical cards. Once the dual-GPU architecture is used, a increase in computational time is seen. For the HPXCL implementation the same behavior for the computational time (blue lines) is seen. The difference between the computational times is one order of magnitude. First, the exact behavior for the dual-GPU case was obtained. This could be improved by using new CUDA features, but since the focus is on the overhead and not the performance this is not relevant for this paper. Second, also for the Multiple GPUs case, the overhead introduced by HPXCL is small and the execution time is faster.

6 Conclusion and Future Work

In this paper we present an abstraction over CUDA – HPXCL – which is tightly integrated into the HPX general purpose parallel run time system. This allows seamless integration into a fully heterogeneous application accessing local and remote devices in a unified fashion. Note that within this implementation the CUDA specific functionality, *e.g.* blockSize, threadSize, and kernel code, is not hidden from the application developer. Therefore, HPXCL is suitable to integrate existing CUDA kernel into the asynchronous execution of HPX.

Our overhead evaluation showed that the performance cost of using such an abstraction is minimal and our implementation could outperform the native CUDA implementation. For the sake of fairness, the same features, *e.g.* CUDA streams, asynchronous memory functionality (`cudaMemcpyAsync`), and `cudaStreamSynchronize` for synchronization, were used in both implementations. However, HPX uses light-weighted threads and even with the usage of one CPU, more than one light-weighted thread is generated. Thus, HPX has the benefit of using more threads which results in the faster computational times. Also, the CUDA kernel is compiled at run time using NVRTC - CUDA Runtime Compilation for HPXCL and compiled at compile time for the native CUDA application. Within this set up a fair comparison of the execution times is hard to archive.

Nevertheless, the presented abstraction can be considered to improve programmability and maintainability of heterogeneous, distributed workloads. The data transfers and the launch of the kernel can be easily integrated in the asynchronous workload on the CPU, like the writing of the image in the Mandelbrot benchmark while the next image size is computed.

This work showed the proof of concept for an unified API for the integration CUDA to HPX by introducing a negligible overhead. A next step would be to do some performance benchmarks against existing benchmark suits which would require more optimization in the naive CUDA kernels. Two possible future directions for HPXCL are PeridynamicHPX [25] and a simple computational fluid dynamics (CFD)[1] solver for in compressible Navier-Stokes equations [26].

1. https://github.com/ltroska/nast_hpx

For most CFD problems, e.q. driven cavity, natural convection and the Kármán vortex street, the matrix-vector operation is the bottle neck for the overall computational time. Here, the benefits of GPUs for solving such problems is easily integrated in the existing HPX code with the HPXCL library. For PeridynamicHPX, a non-local fracture mechanics code, one large portion of the computational costs is the neighbor search. Here, this task could be done on GPUs, where fast algorithms for this task are available.

ACKNOWLEDGEMENTS

This material is based upon work supported by the NSF Award 1737785 and a Google Summer of Code stipend.

REFERENCES

[1] "X-Stack: Programming Challenges, Runtime Systems, and Tools, DoE-FOA-0000619," 2012. [Online]. Available: http://science.energy.gov/~/media/grants/pdf/foas/2012/SC_FOA_0000619.pdf

[2] T. Heller, P. Diehl, Z. Byerly, J. Biddiscombe, and H. Kaiser, "HPX – An open source C++ Standard Library for Parallelism and Concurrency," in *Proceedings of OpenSuCo 2017, Denver, Colorado USA, November 2017 (OpenSuCo'17)*, 2017, p. 5.

[3] J. Kraus, "An introduction to cuda-aware mpi," https://devblogs.nvidia.com/parallelforall/introduction-cuda-aware-mpi/, 2013, accessed: 2017-11-24.

[4] 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. [Online]. Available: http://dx.doi.org/10.1177/1094342007078442

[5] T. Heller, H. Kaiser, P. Diehl, D. Fey, and M. A. Schweitzer, "Closing the performance gap with modern c++," in *International Conference on High Performance Computing*. Springer, 2016, pp. 18–31.

[6] T. C. S. Committee, "N4578: Working draft, technical specification for c++ extensions for parallelism version 2." Tech. Rep. [Online]. Available: http://open-std.org/JTC1/SC22/WG21/docs/papers/2016/n4578.pdf

[7] M. Copik and H. Kaiser, "Using sycl as an implementation framework for hpx.compute," in *Proceedings of the 5th International Workshop on OpenCL*, ser. IWOCL 2017. New York, NY, USA: ACM, 2017, pp. 30:1–30:7. [Online]. Available: http://doi.acm.org/10.1145/3078155.3078187

[8] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014, domain-Specific Languages and High-Level Frameworks for High-Performance Computing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731514001257

[9] M. Garland, M. Kudlur, and Y. Zheng, "Designing a unified programming model for heterogeneous machines," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 67:1–67:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389087

[10] D. Bonachea and P. Hargrove, "Gasnet specification, v1. 8.1," 2017.

[11] R. D. Hornung and J. A. Keasler, "The raja portability layer: Overview and status," 9 2014.

[12] N. Bell and J. Hoberock, "Chapter 26 - thrust: A productivity-oriented library for cuda," in *GPU Computing Gems Jade Edition*, ser. Applications of GPU Computing Series, W. mei W. Hwu, Ed. Boston: Morgan Kaufmann, 2012, pp. 359 – 371. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780123859631000265

[13] K. Ebcioglu, V. Saraswat, and V. Sarkar, "X10: Programming for hierarchical parallelism and non-uniform data access," in *Proceedings of the International Workshop on Language Runtimes, OOPSLA*, vol. 30, 2004.

[14] ——, "X10: an experimental language for high productivity programming of scalable systems," in *Proceedings of the Second Workshop on Productivity and Performance in High-End Computing (PPHEC-05)*, 2005.

[15] P. Diehl, M. Stumpf, T. Heller, M. Seshadri, and H. Kaiser, "HPXCL v0.1," September 2018. [Online]. Available: https://doi.org/10.5281/zenodo.1409043

[16] A. Tabbal, M. Anderson, M. Brodowicz, H. Kaiser, and T. Sterling, "Preliminary design examination of the parallex system from a software and hardware perspective," *ACM SIGMETRICS Performance Evaluation Review*, vol. 38, no. 4, pp. 81–87, 2011.

[17] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "Hpx: A task based programming model in a global address space," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. ACM, 2014, p. 6.

[18] H. Kaiser, M. Brodowicz, and T. Sterling, "Parallex an advanced parallel execution model for scaling-impaired applications," in *Parallel Processing Workshops, 2009. ICPPW'09. International Conference on*. IEEE, 2009, pp. 394–401.

[19] J. Biddiscombe, T. Heller, A. Bikineev, and H. Kaiser, "Zero Copy Serialization using RMA in the Distributed Task-Based HPX runtime," in *14th International Conference on Applied Computing*. IADIS, International Association for Development of the Information Society, 2017.

[20] H. Kaiser, T. Heller, D. Bourgeois, and D. Fey, "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*, ser. ESPM '15. New York, NY, USA: ACM, 2015, pp. 29–37. [Online]. Available: http://doi.acm.org/10.1145/2832241.2832244

[21] P. Diehl, "Modeling and simulation of cracks and fractures with peridynamics in brittle materials." Dissertation, Institut für Numerische Simulation, Universität Bonn, 2017.

[22] M. Harris, "How to implement performance metrics in cuda c/c++," https://devblogs.nvidia.com/how-implement-performance-metrics-cuda-cc/, 2012, accessed: 2018-06-24.

[23] R. F. Van der Wijngaart and T. G. Mattson, "The parallel research kernels," in *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*. IEEE, 2014, pp. 1–6.

[24] M. Harris, "How to Overlap Data Transfers in CUDA C/C++," https://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc/, 2012, accessed: 2017-11-24.

[25] P. Diehl, P. K. Jha, H. Kaiser, R. Lipton, and M. Levesque, "Implementation of Peridynamics utilizing HPX – the C++ standard library for parallelism and concurrency," *ArXiv e-prints*, Jun. 2018.

[26] L. Troska, "A HPX-based parallelization of a Navier-Stokes-solver," Master's thesis, Universität Bonn, August 2016.

APPENDIX A
ARTIFACT DESCRIPTION APPENDIX: INTEGRATION OF CUDA PROCESSING WITHIN THE C++ LIBRARY FOR PARALLELISM AND CONCURRENCY (HPX)

Patrick Diehl and Hartmut Kaiser
Center for Computation & Technology
Louisiana State University

A. Abstract

The appendix contains the information to launch the examples presented in the SC18 paper " Integration of CUDA Processing within the C++ library for parallelism and concurrency (HPX)" in Section 5. We provide the used compilers and libraries, which were used to compile HPXCL. In addition, we provide the shell scripts and sbatch scripts, we used to run the examples on the rostam[1] cluster. HPX and HPXCL are available on github.

B. Description

1) Check-list (artifact meta information):

- **Algorithm:** Stencil kernel (Intel Parallel Research Kernels), Mandelbrot set
- **Program:** C++ binary, C, and C++ libraries
- **Compilation:** gcc (20160615) 5.4.0, nvcc 8.0, V8.0.61, CMake 3.9, and OpenMPI (OpenRTE) 1.10.7
- **Run-time environment:** Cent OS 7 running on kernel 3.11 using Nvidia SMI 396.37 and Nvidia Driver 396.37
- **Hardware:**
  1) Bahram: Intel E5-2660 v3 (2.60GHz), 128GB Memory, 2 Nvidia K80 with 25GB memory
  2) Reno: Intel CPU E5-2670 v2 (2.50GHz), 1 Nvidia K40 with 12GB memory
- **Execution:** Jobs were sent to the nodes using slurm 17.02.10
- **Output:** execution time,length of vector or execution time, image size in bytes
- **Experiment workflow:** Variation of input sizes, amount of GPUs, and amount of nodes
- **Publicly available?:** yes

2) How software can be obtained: HPX(7f3e67c)[2] and HPXCL[3] can be obtained via github.

3) Hardware dependencies: Nvidia CUDA cards

4) Software dependencies: For this paper following libraries were utilized in the version showed in the table. For a minimal required version for HPX we refer to[4] and for HPXCL to[5]. The compilers used for this paper are listed above.

Libraries	
hwloc	1.11.2
boost	1.62
libpng	1.5.13

[1]https://github.com/STEllAR-GROUP/hpx/wiki/Running-HPX-on-Rostam
[2]https://github.com/STEllAR-GROUP/hpx
[3]https://github.com/STEllAR-GROUP/hpxcl
[4]http://stellar.cct.lsu.edu/files/hpx-1.1.0/html/hpx/manual/build_system/prerequisites.html
[5]https://github.com/STEllAR-GROUP/hpxcl

C. Installation

We assume that the prerequisites, boost, hwloc, libpng, gcc, cmake, CUDA SDK, are already installed on the system. We assume that boost was compiled with the same compiler, which was used to compile HPX.

Listing 1. Compile HPX

```
module load gcc hwloc boost
git clone \
 https://github.com/STEllAR-GROUP/hpx.git
cd hpx && mkdir build
cmake \
 -DCMAKE_BUILD_TYPE=Release \
 -DCMAKE_INSTALL_PREFIX=path_to_hpx \
 ..
make core -j
make install
```

Listing 2. Compile HPXCL

```
module load gcc cuda libpng
git clone \
 https://github.com/STEllAR-GROUP/hpxcl.git
cd hpxcl && mkdir build
cmake \
 -DCMAKE_BUILD_TYPE=Release \
 -DHPX_ROOT=path_to_hpx \
 -DHPXCL_WITH_CUDA=ON \
 ..
make -j
```

D. Experiment workflow

For each input size the program was executed 11 times on the same node within the same slurm job. The first execution time was ignored and the average out of the 10 reaming times was computed and used as the execution time in each benchmark. For all benchmarks, everything was measured, including allocation, data transfer, and deallocation of memory. Expect the validation of the copied data from the device was excluded from the overall computation time. For the Mandelbrot example in Section 5.1.3 only one measurement due to the writing of the images to the file system was recorded.

Listing 3 shows the sbatch script, which was used to send all jobs to the cluster nodes. Listing 4 shows the shell script, which was used to obtain the measurements in Section 5.1.1. Listing 5 shows the shell script, which was used to obtain the measurements in Section 5.1.2 and Section 5.2.1. Listing 6 shows the shell script for the measurements obtained in Section 5.1.3.

Listing 3. Slurm script for sending the jobs to the nodes reno (K40) and bahram (K80).

```
#!/usr/bin/env bash
#SBATCH -o application_%j.out
#SBATCH -t 1:00:00
#SBATCH -p {bahram/reno}
#SBATCH -N 1
#SBATCH -D path_to_application
module load gcc/5.4.0 cuda/8.0.61 \
 boost/1.62.0-gcc5.4.0
srun script or executable
```

Listing 4. Bash script to run the stencil example.

```
#!/bin/bash
for j in 100 1000 10000 50000 100000 250000
500000 750000 1000000 1500000 2000000
5000000 10000000 20000000 50000000 100000000
1000000000 200000000 500000000 1000000000
2000000000 5000000000 10000000000 20000000000
50000000000
do
  for i in {0..10..1}
  do
    executable ${j}
  done
done
```

Listing 5. Bash script to run the partition example.

```
#!/bin/bash
for j in {1..8..1}
do
  for i in {0..10..1}
  do
  executable ${j}
  done
done
```

Listing 6. Bash script to run the Mandelbrot example.

```
#!/bin/bash
./mandelbrot_cuda h w iterations devices
```

E. Evaluation and expected result

For the benchmarks in Section 5.1.1, Section 5.1.2, and Section 5.2.1 the CUDA kernel was implemented in C++ and the result of the CUDA kernel launch was compared with the result of the C++ implemented for each input vector. All resulting images for the Mandelbrot examples in Section 5.1.3 and Section **??** were validated by eye for correctness and Figure 1 shows one resulting image. For all examples the length of the input vector or the image size in byes were stored together with the execution time in second as comma separated values (CSV) files, see Listing 7.

Listing 7. Example output for the Mandelbrot example.

```
cat imagesync.dat
3000000,527.689
12000000,490.021
27000000,749.036
48000000,2002.65
75000000,2096.07
108000000,2160.49
147000000,3549
192000000,3564.16
```

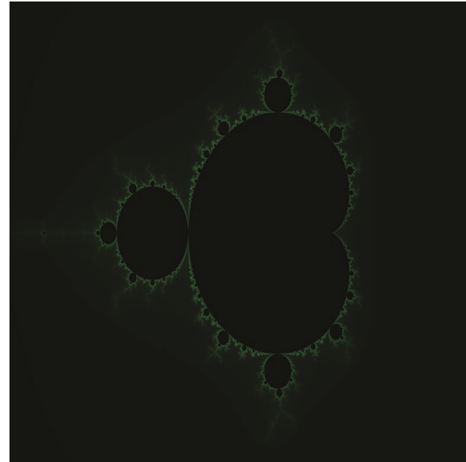

Fig. 1. Resulting Mandelbrot set image from the example in Section **??**.

F. Experiment customization

For all experiments the same executable were used and the experiment customization is realized via command line arguments. For the examples in Section 5.1.1 and Section 5.1.2 the input size of the vector varies. In Section 5.2.1 in addition to the input size of the vector, we varied the amount of GPUs. In Section 5.1.3 the size of the image in bytes is varied and the resulting image is asynchronously and synchronously written to the file system.

G. Notes

The aim of this paper was to measure the overhead introduced by the additional layer of the HPXCL API. The authors are aware that several highly optimized benchmark suites are available for measurement of performance of CUDA kernels.We did not use these and tried to mimic the same behavior as in the HPXCL layer by re-implementing it in native CUDA, using the same kernels and the same CUDA features, *e.g.* CUDA streams, asynchronous memory functionality (`cudaMemcpyAsync`), and `cudaStreamSynchronize` for synchronization.