Int J Parallel Prog (2018) 46:23–41
https://doi.org/10.1007/s10766-017-0497-y

High-Level Programming for Many-Cores Using C++14 and the STL

Michael Haidl[1] · Sergei Gorlatch[1]

Received: 3 October 2016 / Accepted: 28 February 2017 / Published online: 13 March 2017
© Springer Science+Business Media New York 2017

Abstract Programming many-core systems with accelerators (e.g., GPUs) remains a challenging task, even for expert programmers. In the current, low-level approaches—OpenCL and CUDA—two distinct programming models are employed: the host code for the CPU is written in C/C++ with a restricted memory model, while the device code for the accelerator is written using a device-dependent model of CUDA or OpenCL. The programmer is responsible for explicitly specifying parallelism, memory transfers, and synchronization, and also for configuring the program and optimizing its performance for a particular many-core system. This leads to long, poorly structured and error-prone codes, often with a suboptimal performance. We present PACXX—an alternative, unified programming approach for accelerators. In PACXX, both host and device programs are written in the same programming language—the newest C++14 standard with the Standard Template Library (STL), including all modern features: type inference (auto), variadic templates, generic lambda expressions, and the newly proposed parallel extensions of the STL. PACXX includes an easy-to-use and type-safe API for multi-stage programming which allows for aggressive runtime compiler optimizations. We implement PACXX by developing a custom compiler (based on the Clang and LLVM frameworks) and a runtime system, that together perform memory management and synchronization automatically and transparently for the programmer. We evaluate our approach by comparing it to OpenCL regarding program size and target performance.

✉ Michael Haidl
m.haidl@uni-muenster.de

Sergei Gorlatch
gorlatch@uni-muenster.de

[1] Department of Computer Science, University of Muenster, Münster, Germany

Keywords Parallel programming · High-level programming · Accelerators · C++ · Compiler optimizations

1 Motivation

Various many-core accelerators, e.g., Graphics Processing Units (GPUs) are increasingly used in today's computer systems. However, writing efficient parallel programs for systems with accelerators remains a complicated task. Special low-level programming models like CUDA [19] or OpenCL [12] must be mastered for writing the parts of an application program (so-called *kernels*) which are executed on an accelerator.

The main problems of the programming approaches based on CUDA and OpenCL are as follows:

- Kernels are written in an extended dialect of C or C++ and are distinct from the *host program* which runs on a CPU. In OpenCL—the only programming approach which is portable across accelerators of different vendors—kernels are represented as strings in the host program which brings non-trivial engineering issues. In particular, type definitions must be duplicated and attention must be paid to the alignment of the data structures in memory which may be different between the host compiler and the OpenCL compiler even for equal types and thus leads to subtle and hard- to-track errors. Besides the cumbersome development, the kernel code in a string is readable by a non-authorized party from the compiled code, which is often undesired, especially in commercial applications.
- Transfers between the host memory and the accelerator's memory must be managed manually, which leads to long and error-prone code.
- Kernels are launched asynchronously; this implies additional programming effort to synchronize the execution on the CPU with the accelerator, which often results in performance penalties.
- Threads must be explicitly managed by the developer which requires detailed knowledge of parallel programming to avoid numerous possible pitfalls (e.g., race conditions); work must be efficiently partitioned across threads, and threads must be properly synchronized with each other.

Managing all these cumbersome details in low-level programming approaches is time-consuming, but high performance is still not guaranteed: kernel optimizations require a deep understanding of the target many-core architecture.

To overcome the weaknesses of current many-core programming models, we develop the *PACXX (Programming Accelerators with C++)* approach, with the following novel contributions:

- PACXX provides a unified programming model, i.e., host and kernel code are written in the same widely-used programming language: C++. PACXX fully supports the latest C++14 standard [8] with all advanced language features (e.g., variadic templates and generic lambda expressions), as well as modern features from the Standard Template Library (STL) [8]. Through the compilation approach used by PACXX, parallel programs are portable across hardware architectures without the disadvantages of OpenCL: (1) the kernel source code is not readable

by unauthorized parties, and (2) parts of the code can be shared between the host program and the kernels without duplications; problems with memory alignment are avoided by design.
- STL containers (e.g., `std::vector`) are used in PACXX to liberate the programmer from tedious memory management tasks and perform memory allocations and transfers transparently for the developer.
- Synchronization of the host with kernels is achieved with `std::future` objects from the STL, which allows for a fine-grained synchronization as used for standard C++ threads.
- STL algorithms relieve the developer from the error-prone tasks of work partitioning across threads and thread synchronization which could result in race conditions or even deadlocks. The work distribution is optimized for each STL algorithm in its PACXX implementation.

The structure of this paper is as follows: in Sect. 2, we present the unified programming model of PACXX and compare it with CUDA using a simple example. In Sect. 3, we demonstrate how the newest C++14 standard together with the parallel extension for STL [11] can be conveniently used for high-level programming of many-core systems and we advocate using ranges [9]—a powerful new concept in C++. In Section 4, we describe the architecture of PACXX and its implementation. In Sect. 5, we show how the advanced technique of multi-stage programming is supported by PACXX for many-core-specific optimizations. In Sect. 6, we evaluate the performance and the portability of PACXX on a variety of benchmark applications. We discuss related work in Sect. 7, and we conclude in Sect. 8.

2 The PACXX Programming Model

To explain our C++-based approach to many-core programming, we consider a popular example in parallel computing: vector addition. We start with a C++ sequential program and then we compare how this program is parallelized for GPU using CUDA versus our PACXX approach.

```
1 std::vector<int> a(N), b(N), c(N);
2 for (int i = 0; i < N; ++i)
3   c[i] = a[i] + b[i];
```

Listing 1 Sequential vector addition in C++.

Listing 1 shows the sequential C++ source code for the vector addition. Here, the memory is implicitly allocated and initialized during the construction of the three STL containers of type `std::vector` in line 1 according to the *RAII (Resource Acquisition Is Initialization)* idiom of C++. The calculation is performed straightforwardly by the for-loop in line 2.

Listing 2 shows how the vector addition is parallelized for GPU using the popular CUDA approach. The CUDA kernel (lines 1–4) replaces the sequential for-loop of the C++ version with a data-parallel version of the vector addition. The `vadd` kernel is annotated with the `global` keyword (line 1) and is, according to the CUDA

standard, a free function with `void` as return type. In CUDA, functions called by a kernel have to be annotated with the `device` keyword to explicitly mark them as GPU functions and instruct the CUDA compiler to generate GPU code for these functions. This restriction restricts code sharing between host and GPU. In particular, STL functions, e.g., algorithms or member functions of `std::vector`, are not annotated with `device` and thus not usable in a kernel.

```
1  _ _global_ _ void vadd (int* a, int* b, int* c, size_t size){
2    auto i = threadIdx.x + blockIdx.x * blockDim.x;
3    if (i >= size) return;
4    c[i] = a[i] + b{i]; }
5
6  std::vector<int> a(N), b(N), c(N);
7  int* da, db, dc;
8  size_t size = N * sizeof(int);
9  cudaMalloc(&da, size);
10 cudaMemcpy(da, &a[0], size, cudaMemcpyHostToDevice);
11 ... // three additional lines: two for b and one for c
12 vadd<<<N/1024 + 1, 1024>>>(da, db, dc, N);
13 cudaDeviceSynchronize();
14 cudaFree(da);  // free db and dc
```

Listing 2 Parallel vector addition in CUDA

Memory accessed by the kernel must be managed by the developer explicitly in the host part of the CUDA program (lines 9–14) and requires double memory allocation: the first allocation happens with the construction of the `std::vector` in line 6 on the host and the second allocation is performed by `cudaMalloc` in line 9 on the GPU, thus making the developer responsible for ensuring consistency between both memories by copying data explicitly (line 10) and, since C arrays without RAII property are used, also for initializing memory. On the host side, three instances of `std::vector` are used to store the data. Since the STL cannot be used in CUDA kernels, three raw integer pointers are defined in line 7 to access the data in the GPU's memory; thereby, the convenient features of `std::vector` are lost and the programmer has to use pointers or build own wrappers to get an easier and safer memory access than using raw pointers. Memory is allocated using the CUDA Runtime API; for brevity, the calls to this API are only shown here in line 9 and 10 for one vector. For each memory allocation and transfer, an additional line of code is necessary.

To launch the kernel, a launch configuration is specified within «< »> in each kernel call (line 12), i.e., a CUDA-specific, non-C++ syntax is used. CUDA threads are organized as a grid of blocks with up to three dimensions: in our example, 1024 threads are the maximal number of threads in one block, and $N/1024 + 1$ blocks (N is the size of the vectors) form the so-called launch grid. All threads execute the same kernel code, and the work is partitioned among them using indices: in our example, each thread computes a single addition depending on the thread's index in the x-dimension (line 2). The indices of a thread within the grid and the block are obtained using special variables `threadIdx`, `blockIdx` and `blockDim` (Listing 2). To prevent an out-of-bounds access, an if-statement guards the access to the memory (line 3). The size of the input vector has to be known in the kernel code and is passed as additional parameter to the kernel (line 1). The GPU works asynchronously to

the host, therefore, the host execution must be explicitly synchronized with the GPU using the CUDA Runtime API (line 13).

Summarizing, the CUDA code in Listing 2 is very different from the original C++ program in Listing 1: a complete program restructuring and new syntax are necessary, and the size of code increases by more than three times up to 17 lines of code (due to the omitted code in line 11).

```
1 std::vector<int> a(N), b(N), c(n);
2 auto vadd = kernel([](const auto& a, const auto& b auto& c){
3   auto i = Thread::get().global;
4   if (i >= a.size()) return;
5   c[i.x] = a[i.x] + b[i.x];
6 }, {{N/1024 + 1}, {1024}};);
7 std::future<void> F = std::async(vadd, a, b);
8 F.wait();
```

Listing 3 Parallel vector addition in PACXX

Listing 3 shows how our PACXX approach works for the same example: the PACXX program is a pure C++14 code without any extensions (e.g., new keywords or special syntax), with the kernel code inside of the host (main) code. PACXX provides the C++ template class `kernel` to identify kernels: instances of `kernel` will be executed in parallel on the GPU. In PACXX, there are no restrictions on the functions which can be called from the kernel code, however, their source code must be available at runtime, i.e., functions from pre-compiled libraries cannot be used. The listing demonstrates how code is shared between the host part and a kernel: `std::vector` is used in the host code and passed to the kernel by reference as common in C++. Like in CUDA, a PACXX kernel is implicitly data-parallel: it is defined by a C++ lambda function (lines 2–6). The launch configuration (the second parameter in line 6) is defined similar to CUDA, and threads are identified with up to three-dimensional indices which are used to partition the work among the GPU's cores. The thread's index can be obtained through the `Thread` class (line 3). We use `std::async` and `std::future` from the newest version of the STL concurrency library [8] to express parallelism: the kernel instance created in line 2 is passed to the STL-function `std::async` (line 7) that invokes the kernel's execution asynchronously on the GPU. The `kernel` function in line 2 decorates the lambda expression with a special type which is recognized by our modified implementation of `std::async`. According to the type of the first parameter of `std::async` in line 7, our implementation decides whether the execution happens on the GPU or CPU. The additional parameters of `std::async` are forwarded to the kernel.

By comparing Listings 3 to 2, we observe that PACXX provides a much more convenient, implicit memory management than CUDA, using the C++ STL containers `std::vector` (for dynamic arrays) and `std::array` (for static arrays): we avoid double memory allocations and explicit memory transfers. To ensure data consistency between the host and the GPU, a lazy copying is supported in the PACXX implementation of the STL, as discussed in Sect. 4. A PACXX kernel also requires guarding the memory access by an if-statement to prevent out-of-bounds access, but there is no need to pass the size of the vector to the kernel, because each instance

of std::vector can obtain the number of elements by using the size function (line 4). The std::async function used to execute the kernel returns an std::future object associated with the kernel launch (line 7); this object is used to synchronize the host and kernel execution more flexibly than in CUDA: since the std::future instance is a C++ object, synchronization with the kernel can happen anytime during the program execution. The wait function (line 8) from the std::future class blocks the host execution if the associated kernel has not yet terminated.

Summarizing, due to the exclusive usage of the modern C++14 and STL features and implicit memory management of STL containers, programming using PACXX is easier and yields shorter code than when using CUDA.

3 High-Level Programming Using C++ and STL

In this section, we demonstrate how C++ and STL implemented in PACXX are used for programming a case study—the N-Body simulation—which we also use for the performance evaluation in Sect. 6. N-Body simulations [22] are an important class of physical applications. For a number of particles, the interaction between all particles is computed in an iterative process which updates the position and velocity of each particle in every step. Our PACXX-based implementation of the N-Body simulation on a GPU follows the ideas described in the CUDA Toolkit [20] as example code. More sophisticated algorithms (e.g., Barnes-Hut [4]) require a lot more development effort, making them impractical as illustrative examples.

STL Containers and Iterators We start our N-Body case study using PACXX by defining the memory to hold the properties (i.e., position and velocity) of particles. In Listing 4 (line 1), we declare three instances of the STL container std::vector. Since all STL containers are template classes, a type (in this case a custom type Float4) is provided which defines the type of vector's elements.

```
1 std::vector<Float4> pos1(N), pos2(N), velo(N);
2 for (auto I = pos1.begin(), E = pos1.end(); I != E; ++I)
3   *I = get_random_position();
```

Listing 4 Using containers and iterators for N-Body.

The std::vector provides all properties we need for efficient N-Body simulation: (1) contiguous memory with constant access time, and (2) dynamic size to simulate an arbitrary number of particles. Listing 4 defines N (i.e., an arbitrary number of) Float4 instances allocated by vector's constructor, and each vector element is initialized with the default constructor. In our case, Float4 defines a data type holding four 32-bit floating point numbers, and the constructor initializes them with 0. In contrast to dynamic memory handling in C++ (using new and delete), an STL container releases all its resources when the lifetime of the container ends. Together with RAII, this reduces possible sources of errors and memory leaks.

In Listing 4, we use iterators offered by containers to initialize the pos1 vector with random positions of particles: in the for-loop (line 2), the *begin* (I) and the *end* (E) iterator are returned by begin() and end(). The first iterator (I)

is used to iterate over the elements of a container while the second iterator (`E`) defines the position where the iteration must stop. We use the type inference feature of C++ (`auto`) to obtain the type of iterator `I` (i.e., `auto` resolves to `std::vector<Float4>::iterator` in line 2). According to the C++ standard, each iterator must be dereferenceable and incrementable, i.e., iterator's type implements the `*` operator (line 3) to access the data an iterator points at, and it implements the prefix ++ operator (line 2) to iterate over the elements of a container. Iterators of `std::vector` satisfy the *RandomAccessIterator* concept of the C++ standard, which allows us to access each element of the vector in constant time by simply adding an integer to an iterator. In terms of [9], all STL containers are *iterables*, i.e., they offer a `begin()` and `end()` function to obtain a *pair of iterators*.
STL Algorithms STL algorithms are very popular in the C++ community, and they are one of the main programming means in our PACXX approach.

```
1 std::vector<Float4> pos1(N), pos2(N), velo(N);
2 std::generate(pos1.begin(), pos1.end(), get_random_position);
```

Listing 5 Using an STL algorithm to initialize data.

In Listing 5, we use the `std::generate` algorithm in line 2 to abstract from the for-loop of Listing 4, such that the code becomes shorter.
Ranges To provide an easier interface to STL algorithms and to allow more efficient implementation of data structures representing sequences of data, *ranges* [9] are proposed for the upcoming C++17 standard. Ranges are iterables and provide the same interface as containers, i.e., the `begin()` and `end()` functions. However, the types returned by `begin()` (the *iterator type*) and `end()` (the *sentinel type*) may be different while they are equality comparable, i.e., they implement the operators == and !=. Allowing different types for the iterator and the sentinel type brings two advantages: (1) data structures can be implemented more efficiently, e.g., iterating over a null-terminated string does not require to search for the null-byte in advance in order to find the end; (2) dynamic conditions can be implemented in the sentinel type to terminate an STL algorithm early, which is impossible with iterators of the same type. Compared to containers, ranges are lightweight objects without own memory which can be copied or moved very efficiently. They can be used to provide different views on the actual data, e.g., one can combine multiple ranges (and containers) to one range, as we show later in Listing 8.

```
1 std::vector<Float4> pos1(N), pos2(N), velo(N);
2 std::generate(pos1, get_random_position);
```

Listing 6 Using an STL algorithm with iterables (e.g., a container or range).

Listing 6 illustrates the interface to STL algorithms with iterables: the instance `pos1` of `std::vector` is passed directly to the `std::generate` algorithm, which expects an iterable (e.g., a container or range) as input (line 2); retrieving the begin and the end of the range or container is now performed inside the STL algorithm.

Computations in N-Body Listing 7 shows our implementation of the N-Body computation part (we use the same formulas as in [22]), using a C++ lambda expression (`nbody`) in lines 2–10. We use the type inference for the arguments of the lambda expression (to keep it independent of the used data types) and for the expression itself, since a lambda expression has an anonymous type. Its first two arguments (`p` and `v`) are the position and the velocity of the particle to update, the third (`np`) is the new position after updating, and the last argument (`part`) is a vector of all particles' positions. The velocity can be updated in-place (line 9); for the position of a particle, however, dependencies between computations exist, so we use `np` (line 8).

```
1  constexpr float G = -6.673e-11f, dt = 3600.f, eps2 = 0.00125f;
2  auto nbody = [=](auto p, auto &v, auto &np, const auto &part) {
3    auto a = std::accumulate(part, Float4{},[&](auto F, auto q} {
4      Float4 r = p - q;
5      r.w = rsqrt(sq(r.x) + sq(r.y) + sq(r.z) + eps2);
6      F.w = G * q.w * cu(r.w);
7      return F + F.w * r; });
8    np = p + v * dt + a * 0.5f * sq(dt);
9    v += a * dt;
10 };
```

Listing 7 Defining the computations of N-Body.

Our user-defined `Float4` type provides overloads for the `*`, `+` and `-` operators to allow convenient multiplication with scalar floating point values and element-wise addition and subtraction of two `Float4` objects. Only the `x`, `y` and `z` values participate in these operations while `w` is used for padding the data structure to the 128 bit width. This ensures coalesced memory loads on Nvidia GPUs whose hardware supports coalesced loading of 32, 64 and 128 bit. Without padding, 2 loads (64 and 32 bit) would be required and the second load would prevent coalesced loading because consecutive threads do not access consecutive memory. For the position vector, we use the padding space to store the mass of each particle together with its position, because the mass of each particle is needed to calculate the attracting force.

In C++, lambda expressions provide the possibility to capture variables from the surrounding scope and use them in the body of the lambda expression. PACXX allows to use this C++ feature, however, currently it is only safe to capture variables by value because a value captured by reference remains in the address space of the host which is (currently) unaccessable by the majority of many-core architectures. The constants `G` and `dt` are captured by value which is indicated by the capture clause `[=]` in line 2. The `eps2` constant taken from [22] is used during the computation of the distance between two particles: it prevents that the square root in line 5 becomes 0 when the distance of a particle to itself is computed which would result in a division by 0. Using the `eps2` value makes it unnecessary to add an if-statement which would prevent the division by 0 but would have a negative effect on the performance on GPU [22].

In line 3 of Listing 7, we apply the STL algorithm `std::accumulate` which uses the interface for ranges explained in Listing 6. The `std::accumulate` algorithm is designed as a higher-order function; it takes in this case a binary lambda expression as argument (lines 3–7) which is used as operator in a sequential summation. To customize the `std::accumulate` algorithm, we use again a lambda

expression. This time it is safe to capture values by reference (`[&]`), because the lambda expression and all variables from the surrounding scope are in the address space of the accelerator.

The lambda expression in lines 3–7 in Listing 7 computes the gravitational forces from all other particles that influence the current particle. Listing 8 demonstrates how these forces are computed for all particles in a vector. We use the `std::for_each` algorithm in line 5, which accepts a unary operator as customizing function. To prepare the input for the `std::for_each` algorithm we are using ranges (our ranges are implemented in the `range` namespace): in line 3, we use the `repeat` range on the `pos1` vector; `repeat` is an infinite range whose iterators return a reference to the vector when dereferenced. Iterators from infinite ranges never reach the sentinel of the range, i.e., comparing an iterator with the sentinel always evaluates to false.

```
1 // initialize data
2 auto nbody = [=] ... // the lambda expression from Listing 7
3 auto ref = range::repeat(pos1);
4 auto data = range::zip(pos1, velo, pos2, ref);
5 std::for_each(data, [=](auto&& t){std::apply(nbody, t);});
```

Listing 8 The N-Body computation using `std::for_each` and ranges.

To meet the requirement of `std::for_each` that the customizing function is unary, we merge in Listing 8 all input ranges together using the `zip` range which stores references to the underlying ranges and returns an instance of `std::tuple`—a fixed-size collection of heterogeneous values. In our case, we use an `std::tuple` with references to the values obtained by dereferencing all iterators from the zipped ranges. The equality operator of the `zip`'s sentinel type, which is called inside the `std::for_each` algorithm, will stop the execution of the algorithm when the end of one of the merged ranges is reached; this is necessary because `repeat` in line 3 is an infinite range. However, this is all hidden from the developer inside the STL implementation. To make the parameters of the `nbody` lambda expression which expects four arguments compatible with the type resulting from dereferencing an iterator of our `zip` range, we unwrap the values stored in the `std::tuple` in line 5. We provide an anonymous lambda expression as the customizing function to `std::for_each` expecting an `std::tuple` from our zipped ranges as a single argument. Inside the lambda expression, `std::apply` is called which is also proposed for the C++17 standard [10]: it unwraps an `std::tuple` (here `t`) and forwards the unwrapped values to `nbody`.

Parallel STL Algorithms Until the proposed C++17 standard, all algorithms in the STL have been implemented sequentially. The proposed parallelism in the STL [11] extends the interface of the STL algorithms by a so-called *execution policy* which defines how the algorithm should be executed: the STL implementation is responsible for executing the algorithm in parallel, without breaking backward compatibility with the sequential version. Three standard execution policies are proposed for C++17: sequential execution (`std::seq`), parallel execution (`std::par`) and parallel_vectorized execution (`std::par_vec`); it is also possible to specify additional (implementation-defined) execution policies.

```
1 // initialize data
2 auto nbody = [=] ... // the lambda expression from Listing 7
3 auto ref = range::repeat(pos1);
4 auto data  = range::zip(pos1, velo, pos2, ref);
5 std::execution_policy pacxx = pacxx_execution_policy{};
6 std::for_each(pacxx, data,[=](auto&& t){std::apply(nbody, t);});
```

Listing 9 Computing N-Body in parallel using an execution policy.

In Listing 9 we show the use of a parallel STL algorithm with a new execution policy provided by PACXX, named `pacxx_execution_policy` (line 5). Calling `std::for_each` with the PACXX execution policy in line 6 will execute the algorithm in parallel on an available GPU. If no many-core accelerator is available, a fallback (e.g., sequential) implementation is used.

The complete code for the N-Body simulation comprises the previously described codes of Listings 6, 7 and 9.

4 Architecture and Implementation of PACXX

The PACXX framework transforms C++ code using a combination of offline and online compilation to a representation executable on a many-core architecture. Currently, PACXX supports two target representations: PTX [21] for Nvidia GPUs, and SPIR [13] for AMD GPUs and Intel Xeon Phi.

Figure 1 shows an overview of our PACXX implementation which comprises two main components:

1) the *PACXX Offline Compiler* is based on Clang 3.8 [15]—an open-source compiler front-end with feature-complete C++14 support;
2) the *PACXX Runtime* library is statically linked into the executable; it consists of the online just-in-time (JIT) compiler implemented using the LLVM library [15], and specific GPU back-ends which use the CUDA and OpenCL runtime libraries.

Correspondingly, a C++ code is compiled by PACXX in two stages: (1) the offline compilation stage (blue shaded in Fig. 1) separates the many-core code (kernel) from the CPU code (host) and prepares an executable for the PACXX runtime, (2) the online compilation stage (red shaded in Fig. 1) JIT compiles the code for the GPU during program execution using our LLVM-based online compiler contained in the PACXX runtime library.

Our offline compiler performs two steps: (1) preparing the GPU code generation at runtime, and (2) compiling the CPU program. In the first step, the Clang front-end builds an abstract syntax tree (AST) which is lowered to the LLVM intermediate representation (IR), and kernel functions are enriched with metadata to identify them as GPU code in the IR. The enriched IR is then transformed using LLVM as follows: (a) aggressive dead code elimination removes from the IR all code not reachable from a kernel; (b) the PACXX inliner inlines as many function calls as possible; (c) standard optimizations (equal to Clang's `O3` optimizations) are performed on the IR; and (d) the resulting IR is wrapped in an object file and passed to the linker. In the second step, the offline compiler lowers the AST to LLVM IR a second time: the

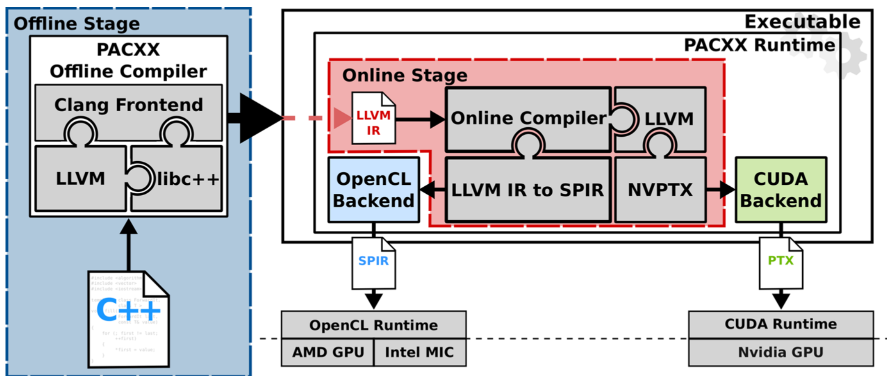

Fig. 1 Key components of the PACXX framework

calls to kernel functions are replaced with calls to the PACXX runtime library for managing data transfers and launching the GPU code. Finally, the generated IR is lowered for the specific host architecture, and object files are generated as usual for C++ programs. As shown in Fig. 1, our runtime library is statically linked into the final executable.

During program execution, our runtime library loads the integrated IR from inside the executable. Additional optimizer passes perform architecture-specific optimizations, such as loop-unrolling and rearranging of load instructions. Finally, the IR is lowered to executable code for the available many-core architecture using the most appropriate LLVM compiler back-end: we use PTX [21] together with the CUDA runtime library when targeting Nvidia GPUs, and SPIR [13] for GPUs and other accelerators with an OpenCL implementation (e.g., from AMD and Intel).

PACXX implements an automatic lazy copying for STL containers from CPU to GPU that is completely hidden from the developer. It is the lazy copying that allows the developer to write an application for accelerators in the same way as for sequential execution on a CPU. To avoid unnecessary data transfers of output containers, we use a compiler-based approach: the lazy copying is implemented in the STL and in the PACXX runtime library, but it is guided by the compiler. To steer the data movement, each container carries a flag indicating which copy of the data is the most recent one. The PACXX offline compiler classifies containers passed to a kernel in three categories:

– *input* if only load instructions are performed on the parameter, then a data transfer occurs prior to a kernel launch, the flag is not raised for the container, i.e., the container is not modified by the kernel and the data in the CPU's main memory stay valid;
– *output* if only store instructions are performed on the parameter, then the data is not transferred from host to the accelerator. The flag is raised which indicates that the copy on the accelerator is the most recent one and has to be transferred back to CPU's main memory prior to the next access to this container;
– *random access* if both load and store instructions are performed, then a data transfer occurs prior to a kernel launch and the flag is raised.

For the instances of `std::vector`, used in our N-Body implementation as data containers, the lazy coping ensures data consistency between the host and accelerator's memory. The vector for position (`pos1`) is categorized as input while the container storing the velocity (`velo`) is categorized as random access. The vector for the new position (`pos2`) is marked as output. When executing the N-Body code, two data transfers to the accelerator and one back to the host are performed transparently by the PACXX runtime.

5 Support for Multi-Stage Programming

Multi-Stage Programming (MSP) [27] is an optimization technique based on runtime code generation: parts of the source code are compiled not at the offline compilation stage, but rather during the runtime of the program. This allows the values only known at runtime to be included into the generated code. Branch conditions and stop conditions of loops are good candidates to be optimized with MSP since performance on accelerators, e.g., GPUs, suffers from a diverging control flow between threads. Moreover, MSP can be used as a meta-programming technique allowing for specialized kernel generation, e.g., the size of input data can be included into the generated code, which results in kernels optimized for particular input sizes.

```
1 auto vadd = kernel([](const auto& a, const auto& b auto& c){
2   auto i =Thread::get().global;
3   if (i >= stage([&]{return a.size()})) return;
4   c[i.x] = a[i.x] + b[i.x];
5 }, {{N/1024 + 1}, {1024}};);
```

Listing 10 Using MSP to optimize vector addition.

Listing 10 shows how the kernel from Listing 3 is optimized with multi-stage programming for a particular input size. PACXX provides an easy-to-use and safe API for MSP: the `stage` function is used to specify which expressions should be evaluated prior to the execution of a kernel. The value of such an expression is incorporated into the kernel's IR by the PACXX runtime and the kernel is then optimized by the online compiler introduced in Sect. 4. The code in the listing differs from the original kernel only in line 3: the call to the `size` function is wrapped into an anonymous lambda expression which is the input to the `stage` function. Since the vector size becomes a constant value, the entire branch in line 3 is removed by the optimizer in the online stage from the kernel code, because the online compiler knows the number of threads executing the kernel.

The implementation of multi-stage programming in the PACXX framework is as follows. When performing the kernel compilation pass, our offline compiler separates the code wrapped by the `stage` function from the rest of the kernel code in order to prevent inlining and, therefore, combining the staged code with the kernel code, which would lead to the same kernel code as the first version of the vector addition in Listing 3. The `stage` function is a variadic template function annotated with the `[[staged]]` generalized attribute known to the PACXX offline compiler. As the first parameter, a lambda expression is required; parameters will be forwarded to the

lambda. After the separation of the MSP-related code, the usual steps of dead code elimination, inlining, and optimizations are performed. Before wrapping the final IR in the object file, the code for calling all instantiations of `stage` and the instantiations of the `stage` functions themselves are generated. For each instantiation, a corresponding new function is generated which will eventually be evaluated on the host at runtime prior to executing the GPU code. The code for `stage` is removed from the kernel program, and calls to this function are replaced by calls to a proxy function `pacxx_eval`. They will be replaced at runtime with the values obtained by evaluating the staged function on the host.

Let us now describe how the PACXX runtime evaluates the staged functions on the host at runtime and how the computed values are embedded into the kernel program prior to its execution on the accelerator. The offline compiler integrates the kernel's IR and the IR for the staged functions into the executable. For executing a kernel, four steps are performed: (1) the kernel's parameters are set; (2) the kernel's launch configuration is set; (3) the staged functions are just-in-time compiled, evaluated, and the results are incorporated into the IR; and (4) the IR is just-in-time compiled and launched. Our offline compiler generates the code for calling the PACXX runtime to perform these four steps. After the staged values have been embedded into the kernel program, our runtime performs additional optimizations on the code using the information of the launch configuration (i.e., the numbers of threads and blocks). This can be viewed as an implicit staging of the launch configuration. A special pass optimizes the control flow graph by removing branches that are never entered. For branches like line 3 in Listing 10, the pass calculates the lower and the upper bound of the thread id: if the comparison evaluates to false for the lower and upper bound the branch can be removed safely. We use the multi-staging API to optimize the STL algorithms for different platforms: the optimized version is created at runtime when the target accelerator becomes known.

6 Evaluation

In this section, we evaluate the performance of PACXX, first using popular parallel benchmarks, and then using our N-Body simulation which makes use of C++, parallel STL, and multi-stage programming.

Benchmarking PACXX To evaluate the performance of PACXX, we implement seven benchmark applications and we compare their runtime with implementations using Nvidia's highly-tuned Thrust [3] library. Thrust provides an STL-like interface and a set of containers and iterator types designed specifically for GPU programming. We evaluate all benchmarks on an Nvidia Tesla K20c GPU with CUDA 7.5. All benchmarks make use of either the `transform` or the `reduce` STL algorithm.

We study simple benchmarks—vector addition (vadd), saxpy, vector sum (sum) and dot product of two vectors (dot)—to evaluate the performance of our implementation of the STL algorithms, which are currently optimized for the Nvidia architecture. The fifth benchmark is a Monte Carlo simulation which estimates the number Pi through pseudo-random number generation. This benchmark shows the ability of PACXX to compile any C++ code: PACXX uses the random number generator from

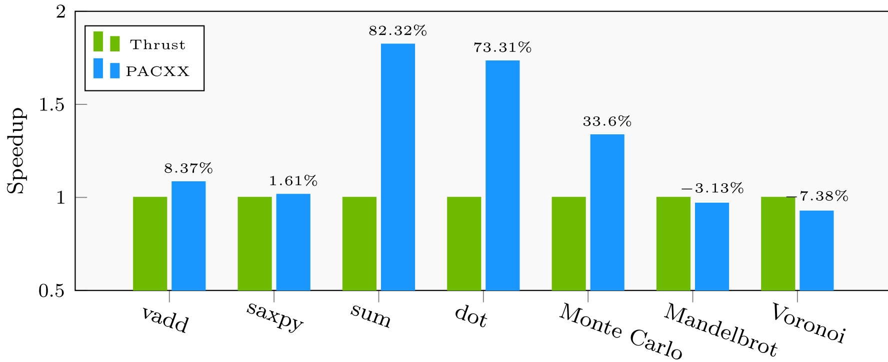

Fig. 2 Speedup of PACXX compared to Nvidia's Thrust library

the Thrust library without modifying Thrust's source code. The final two benchmarks are the Mandelbrot-set computation and the computation of the Voronoi diagram [23] which is an iterative stencil computation.

Figure 2 shows the speedup of our implementations compared to Thrust. We measured the execution time of each benchmark in 1000 runs for different input sizes and report the median speedup.

For the vadd, saxpy and Mandelbrot-set benchmarks that use the STL `transform` algorithm, the slightly better performance of our implementation results from a different thread configuration for the executed kernel. Our implementation of the transform algorithm starts 128 concurrent threads per block while Thrust always starts the maximum number of threads per block allowed by the GPU architecture. For these benchmarks, starting fewer threads per block results in a better block scheduling on the GPU and thus better performance for small input sizes while the effect becomes weaker for larger input.

The results for the STL reduction algorithm used in the sum, dot and Monte Carlo benchmarks show an average speedup for PACXX up to 1.82. This results from the different implementation strategies for the reduction: while Thrust executes two reductions on the GPU and copies the final result back to the CPU, the PACXX implementation performs the second reduction phase on the CPU. Thus we avoid the overhead of starting a second kernel on the GPU and we exploit the fact that transferring a single value (commonly 4–8 bytes) is as expensive as transferring 128 values (512–1024 bytes) over the PCIExpress bus. Moreover, after transferring data from the GPU memory back to the main memory the data is already in the CPU's cache hierarchy, allowing for a fast reduction on the CPU. This strategy results in an average speedup for small input sizes ($2^{15} - 2^{18}$ 32 bit values) of up to 3.2 while both implementations are on par for larger input sizes.

The Voronoi diagram computation is the only benchmark dominated by Thrust. Again the STL transform algorithm is used, however, in this benchmark the input to the algorithm is a composition of positions in the input array. In the Thrust implementation, the neighborhood relations are defined through different offsets (positive and negative) added to the begin iterator of the input vector which is a raw pointer for

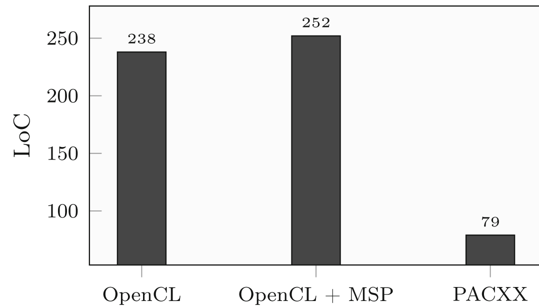

Fig. 3 Size in LoC of implementations for N-Body simulation

Thrust's `device_vector` container. In our implementation, we have to wrap raw pointers first into a range to use our proposed STL algorithms with ranges. This adds an overhead of retrieving the values required for the computation, which results in an average advantage of Thrust over our implementation of 7.4%.

For all benchmarks Thrust and PACXX are on par regarding code length because both approaches provide a similar level of abstraction.

Evaluation: N-Body Simulation with MSP optimization Besides the presented PACXX version of the N-Body simulation, we implemented an identical version using OpenCL for the comparison purpose. We evaluate two versions of each implementation: one with runtime values incorporated through multi-staging and one without multi-staging. In the OpenCL implementation with multi-staging the runtime values are incorporated into the kernel code using string replacement.

Code Size In Fig. 3, we compare the lines of code (LoC) needed to implement all-pairs N-Body simulation with our PACXX approach using the C++ STL versus using OpenCL. We observe that using C++ and STL in PACXX significantly reduces the development effort, which results from a higher level of abstraction: whereas OpenCL requires manual management of the GPU, the PACXX runtime performs all necessary steps and initializes the GPU transparently. Moreover, the lazy copying feature of PACXX further reduces the length of code. As shown in our case study on N-Body simulation, no additional MSP code is used inside the PACXX program: In this benchmark, we used an MSP optimized version of the STL `std::accumulate` algorithm shown in Listing 7, thus there is no difference in the code size for a PACXX implementation with and without multi-staging.

Performance To demonstrate the portability and performance of PACXX, we use three different GPUs and one Intel Xeon Phi co-processor for the evaluation of the presented N-Body simulation: (1) an Nvidia Tesla K20c GPU (Kepler architecture) with OpenCL 1.2 and CUDA 7.5; (2) an Nvidia GTX 480 GPU (Fermi architecture) with OpenCL 1.1 and CUDA 6.5; (3) an AMD R9 295X2 GPU (Hawaii architecture) with OpenCL 2.0; (4) an Intel Xeon Phi 5110p (Knights Corner architecture) with OpenCL 2.0. Kernel runtimes are measured with the OpenCL profiling API and the Nvidia profiler. The median of 1000 measurements is reported.

Figure 4 shows the measured speedup over the OpenCL reference implementation without MSP. We compute 2^{21} particles on the Nvidia K20c and 2^{20} particles on the other architectures.

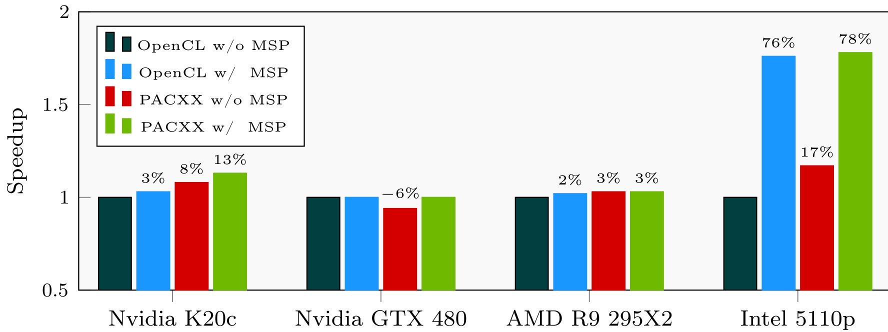

Fig. 4 Speedup of PACXX compared to the OpenCL implementation

We observe that, on the Nvidia K20c, MSP improves the performance in all cases by 3% for the OpenCL code and by 5% for the PACXX code. However, our PACXX compiler achieves additional performance through its aggressive optimizations: the PACXX version with MSP outperforms the OpenCL reference implementation by 13%.

On the Nvidia GTX 480, the results are different. The OpenCL implementation with MSP only achieves 0.1% higher performance compared to the version without MSP. For the PACXX code without MSP we observe a slowdown of 6% compared to the OpenCL reference implementation. We explain this by the fact that the Nvidia GTX 480 is programmed with an OpenCL 1.1 implementation, which allows the compiler to use non-IEEE 754 conform floating point optimizations, i.e., the performance benefit is achieved due to a decreased precision. Our PACXX compiler, on the other hand, respects the IEEE 754 standard. At the same time, PACXX can make up for this performance loss with MSP and is on par with OpenCL.

On the AMD R9 295X2, the PACXX code is 3% faster than its OpenCL equivalent. However, MSP increases performance for PACXX by only 0.3% while it brings 2.2% for OpenCL. On the AMD GPU the performance benefit of MSP is not so high compared to the GPUs from Nvidia. In contrast to the Nvidia GPUs which have a specialized optimizer pipeline, our online compiler uses a generic optimizer pipeline not tailored especially for AMD GPUs.

On the Intel Xeon Phi 5110p, MSP significantly improves the performance for OpenCL and PACXX: by 76 and 78% correspondingly. The PACXX code without MSP is 17% faster than the OpenCL reference implementation. The performance increase of the MSP-optimized kernels results from an automatic vectorization performed by Intel's OpenCL implementation. The auto-vectorization benefits from the additional information about the input size, because no additional control flow (similar to the if-statement shown in Listing 3) is required to prevent out-of-bounds access.

7 Related Work

The C++ AMP standard [17] extends C++ by an explicit data-parallel construct (`parallel_for_each`), and so-called `array_views` provide functions for

memory transfers. The developer still needs to use a wrapper (i.e., has to write an additional line of code) for each allocation and has to use the C++ AMP views instead of the original C++ data types in order to handle the memory synchronization by C++ AMP. SYCL [14]—a high-level interface to the OpenCL infrastructure—integrates the OpenCL programming model into C++ by using the lambda features of C++11. However, SYCL still requires multiple memory allocations for so-called `Buffers` in the host code and the kernel code. Our PACXX approach avoids these restrictions of C++ AMP and SYCL by design.

Thrust [3] and Bolt [2] are parallel libraries implementing STL algorithms on GPUs. Thrust uses CUDA while Bolt uses OpenCL. Both libraries suffer from the limitations of the corresponding low-level programming model, especially when specifying a user-function to customize an algorithm. In PACXX, the developer can customize the STL algorithms with standard C++ features. Portability is another issue of these libraries: for Thrust only a CUDA implementation exists, Bolt is only supported on AMD architectures.

First implementations of the parallel STL [11]—by Microsoft [18] and ParallelSTL [16]—use multi-threading on CPUs for parallel execution. Our PACXX implementation is portable across architectures and can be executed on GPUs of different vendors, on Intel Xeon Phi, and on multi-core CPUs.

High-level libraries FastFlow [1], Muesli [6], SkelCL [25] and SkePU [5] provide higher-order functions (a.k.a. algorithmic skeletons [7]) comparable with the STL algorithms for a broad class of applications. None of these libraries provide data types similar to C++ ranges used in PACXX, so the skeletons in these libraries are not as easily customized as STL algorithms, e.g., in SkelCL customizing functions must be expressed as strings and additional data can only be used inside a customizing function by exploiting side effects. Lazy copying strategies implemented in high-level libraries [5,25] require additional information from the developer to decide when data is write-only: this is achieved either using type decorators (as in SkelCL) or by contract, i.e., output parameters must be the first parameter of a function (as in SkePU). In contrast, the compiler-supported lazy copying in PACXX does not require any additional information from the developer.

Multi-stage programming [27] in the Lightweight Modular Staging [24] framework implemented in Scala builds the foundation of the Delite [26] project which simplifies the development of domain-specific languages (DSL) for parallel processors. While Delite provides a framework for DSL development and execution on GPUs only, we support a general-purpose use of multi-staging across different many-core architectures.

8 Conclusion

We presented PACXX—a unified and portable programming model, providing the state-of-the-art C++14 standard with all modern features as a programming language for a diversity of many-cores. The main advantages of using exclusively C++ and the STL for programming accelerators are as follows:

- With the parallel extension for the STL enriched with ranges, PACXX provides a new way to express parallelism within C++ and offers a modern interface familiar to C++ developers. Cumbersome tasks of low-level accelerator programming (e.g., partitioning of work across threads) are hidden inside the implementation of the STL algorithms.
- With lazy copying for STL containers (e.g., `std::vector`) managed by the PACXX runtime, programs become shorter and less error-prone. The necessity of allocating memory twice is removed completely, and parallel programs for many-cores are written just like sequential C++ programs.
- Exploiting `std::async` and `std::future` allows for a fine-grained and flexible synchronization between the host and the accelerator execution.

The multi-stage programming API provided by PACXX enables program optimizations beneficial on a GPU without performance penalties on other architectures (e.g., Intel Xeon Phi). We show that PACXX programs provide competitive and in some cases even better performance than low-level OpenCL implementations while being significantly shorter.

Acknowledgements We would like to thank Michel Steuwer for many fruitful discussions and Nvidia Corp. for their generous hardware donation.

References

1. Aldinucci, M., Campa, S., Danelutto, M., Kilpatrick, P., Torquati, M.: Targeting distributed systems in fastflow. In: Euro-Par 2012: Parallel Processing Workshops, pp. 47–56, Springer (2012)
2. AMD: Bolt C++ Template Library. Version 1.2 (2014)
3. Bell, N., Hoberock, J.: Thrust: a parallel template library. GPU Computing Gems Jade Edition. pp. 359–372 (2011)
4. Bischof, H., Gorlatch, S., Leshchinskiy, R., Müller, J.: Data parallelism in C++ template programs: a Barnes-Hut case study. Parallel Process. Lett. **15**(03), 257–272 (2005)
5. Enmyren, J., Kessler, C.: SkePU: A multi-backend skeleton programming library for multi-GPU Systems. In: Proceedings of the Fourth International Workshop on High-Level Parallel Programming and Applications, ACM, pp 5–14 (2010)
6. Ernsting, S., Kuchen, H.: Algorithmic skeletons for multi-core, multi-GPU systems and clusters. Int. J. High Perform. Comput. Netw. **7**(2), 129–138 (2012)
7. Gorlatch, S., Cole, M.: Parallel skeletons. In: Encyclopedia of Parallel Computing. pp. 1417–1422, Springer (2011)
8. isocpp (2014a) Programming languages - C++ (committee draft)
9. isocpp (2014b) Working draft, C++ extensions for ranges [N4569]
10. isocpp (2015a) Programming languages—C++ extensions for library fundamentals [N4480]
11. isocpp (2015b) Technical specification for C++ extensions for parallelism [N4578]
12. Khronos Group: the OpenCL specification. Version 1.2 (2012)
13. Khronos Group: the SPIR specification. Version 1.2 (2014)
14. Khronos Group: SYCL specifcation. Version 1.2 (2015)
15. Lattner, C.: LLVM and Clang: next generation compiler technology. In: Proceedings of the BSD Conference, pp 1–2 (2008)
16. Lutz, T.: ParallelSTL. https://github.com/t-lutz/ParallelSTL. Accessed 30 Apr 2016
17. Microsoft: C++ AMP: language and programming model. Version 1.0 (2012)
18. Microsoft: Parallel STL. https://parallelstl.codeplex.com/ Accessed 30 Apr 2016
19. Nvidia: CUDA programming guide. Version 7.5 (2015a)
20. Nvidia: CUDA Toolkit 7.5 (2015b)
21. Nvidia: Parallel thread execution ISA. Version 4.3 (2015c)

22. Nyland, L., Harris, M., Prins, J.: Fast N-body simulation with CUDA. GPU Gems **3**(1), 677–696 (2007)
23. Okabe, A., Boots, B., Sugihara, K., Chiu, S.N.: Spatial Tessellations: Concepts and Applications of Voronoi Diagrams, vol. 501. Wiley, Hoboken (2009)
24. Rompf, T., Odersky, M.: Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. ACM SIGPLAN Notices, vol. 46, pp 127–136, ACM (2010)
25. Steuwer, M., Kegel, P., Gorlatch, S.: SkelCL—a portable skeleton library for high-level GPU programming. In: Workshop on High-Level Parallel Programming Models and Supportive Environments at IPDPS 2011, IEEE, pp 1176–1182 (2011)
26. Sujeeth, A.K., Brown, K.J., Lee, H., et al.: Delite: a compiler architecture for performance-oriented embedded domain-specific languages. ACM Trans. Embed. Comput. Syst. **13**(4s), 134:1–134:25 (2014)
27. Taha, W.: A gentle introduction to multi-stage programming. In: Domain-Specific Program Generation, pp 30–50, Springer (2004)