Programming GPUs with C++14 and Just-In-Time Compilation

Michael Haidl, Bastian Hagedorn, and Sergei Gorlatch

University of Muenster
{m.haidl, b.hagedorn, gorlatch}@uni-muenster.de

Abstract. Systems that comprise accelerators (e.g., GPUs) promise high performance, but their programming is still a challenge, mainly because of two reasons: 1) two distinct programming models have to be used within an application: one for the host CPU (e.g., C++), and one for the accelerator (e.g., OpenCL or CUDA); 2) using Just-In-Time (JIT) compilation and its optimization opportunities in both OpenCL and CUDA requires a cumbersome preparation of the source code. These two aspects currently lead to long, poorly structured, and error-prone GPU codes. Our PACXX programming approach addresses both aspects: 1) parallel programs are written using exclusively the C++ programming language, with modern C++14 features including variadic templates, generic lambda expressions, as well as STL containers and algorithms; 2) a simple yet powerful API (PACXX-Reflect) is offered for enabling JIT in GPU kernels; it uses lightweight runtime reflection to modify the kernel's behaviour during runtime. We show that PACXX codes using the PACXX-Reflect are about 60% shorter than their OpenCL and CUDA Toolkit equivalents and outperform them by 5% on average.

1 Introduction

Accelerators such as Graphics Processing Units (GPUs) are increasingly used in today's high-performance systems. However, programming such systems remains complicated, because it requires the use of two distinct programming models: one for the host CPU (e.g., C or C++) and one for the GPU (e.g., OpenCL or CUDA). The codes for GPUs (so-called kernels) are written using limited subsets of the C/C++ language which miss many advanced features of the current standards like C++14 [1]. Furthermore, specific language constructs of CUDA and OpenCL for parallelization and synchronization have to be additionally mastered by the GPU software developers. Last but not least, the Just-In-Time (JIT) compilation which is a proven technique of simplifying and optimizing the programming process is provided for GPUs on a very restricted scale.

This paper aims at simplifying and improving the programming process for systems with GPUs and other accelerators by making two main contributions:

1. We present and implement *PACXX (Programming Accelerators with C++)* - a unified programming model based on the newest C++14 standard that uniformly covers both host and kernel programming without any language extensions.

2. We develop *PACXX-Reflect* – a simple yet powerful API to enable lightweight JIT compilation of PACXX programs in order to optimize the kernel code during program execution by using values that become known at runtime.

We evaluate our programming approach for GPUs with C++14 and JIT compilation using two case studies – matrix multiplication and Black-Scholes model for the option market – and demonstrate that PACXX codes are about 60% shorter than their manually optimized OpenCL and CUDA Toolkit equivalents and outperform them 5% on average.

The structure of the paper is as follows. Section 2 provides an overview of the state of the art and related work in programming systems with accelerators in general and the JIT compilation approaches for such systems in particular. In Section 3, we explain the PACXX programming approach and compare it to CUDA by way of example. We present our approach to JIT compilation using the PACXX-Reflect API in Section 4.

In Section 5, we briefly describe the implementation of PACXX and we evaluate our approach on two case studies by comparing the size and performance of PACXX codes to the corresponding CUDA and OpenCL programs. Finally, we conclude in Section 6.

2 State of the Art and Related Work

In current programming approaches like CUDA and OpenCL, the code for a GPU (so-called kernel) is written using a limited subset of the C/C++ language, e.g., CUDA C++ [3] which misses many advanced features of the current standards like C++14 [1]. Memory management for the GPU memory has to be performed by the developer explicitly, because C++ has no language support for distinct memories. Memory is explicitly allocated twice - first in the host memory and then again in the GPU's memory. The developer is also responsible for performing explicit synchronization (copying) between the two distinct memories. This implies a significantly longer boilerplate (host-)code for memory management as compared to C++ where allocation and initialization are performed together through the RAII (Resource Acquisition Is Initialization) idiom [4].

Although CUDA and OpenCL (in a provisional version of the OpenCL 2.1 standard [5]) have been recently extended with static C++11 language features, these C++ extensions define new, for C++ developer unfamiliar, language elements (e.g., cast operators), while dynamic language features of C++ such as the Standard Template Library are still not provided by neither CUDA nor OpenCL.

Several recent approaches aim at integrating the accelerator programming into C++. The C++ AMP approach [6] extends C++ by an explicit data-parallel language construct (`parallel_for_each`), and so-called `array_views` provide functions for memory transfers. The developer still needs to use a wrapper (i.e., write an additional line of code) for each memory allocation and use the C++ AMP views instead of the original C++ data types in order to achieve that memory synchronization is done transparently by the system. SYCL [7] is a high-level interface that integrates the OpenCL programming model into C++ by using the lambda features of the C++11 standard, but it still demands multiple memory allocations for so-called `Buffers` both in the host and kernel

code. Nvidia Thrust [8] and AMD Bolt [9] are libraries implementing the functionality of the C++ Standard Template Library (STL) in a data-parallel way, but they are restricted to accelerators from the corresponding vendor and do not support modern C++14 language features. Annotation-based approaches OpenMP [10] and OpenACC [11] expect the user to use parallelism-specifying directives in addition to C++. STAPL [12] offers STL functionality which is executed in parallel by the underlying runtime system, but it targets distributed-memory systems rather than systems with GPUs.

None of the described programming models offers the possibility to transform and optimize the kernel code during execution, i.e., in a Just-In-Time (JIT) manner. For example, writing a kernel for a particular size of input data provides usually better performance than a generic kernel, due to additional optimizations performed by the compiler. However, writing separate kernels for different data sizes would lead to a poorly structured, hardly maintainable codes. Just-in-time compilation can be used to optimize code by taking into account values which become known during the execution of the program: thereby, compilers can additionally optimize code when performance-critical variables (e.g., exit conditions of loops) are resolved.

OpenCL and the newest releases of CUDA support JIT compilation of kernel code. However, both approaches demand that the kernel code is provided as human-readable code which has a security drawback: the source code may be disclosed to non-authorized parties. The NVRTC [13] library supports all C++ features available in CUDA for JIT compilation. Unfortunately, a complicated problem arises using NVRTC: to allow function overloading and template functions as kernels, CUDA C++ follows the C++ standard regarding function name mangling while kernel names are machine generated and unknown to the developer. Without knowing the mangled name of a kernel, the function pointer for invoking the kernel cannot be retrieved and the kernel cannot be called. Additionally, template kernels must be explicitly instantiated prior to their use by the developer. The current solution is to enforce the C naming policy using `extern "C"`, but this completely disables function overloading and templates for kernels, because function names are no longer unique and the right kernel cannot be resolved by its name. Another solution could be an additional library providing function name de-mangling, but this would introduce more development overhead and unnecessary boilerplate code, because function names would have to be combined with the actual data types used in the templates. Another recent JIT approach is LambdaJIT [14] that automatically parallelizes the lambda functions used in STL algorithms. Through different back-ends, LambdaJIT is capable of offloading computations to a GPU.

This paper describes a novel approach to programming systems with GPUs and other accelerators: our PACXX model relies exclusively on the newest C++14 standard without any extensions, and we use lightweight runtime reflection to provide JIT compilation and optimization of PACXX programs. PACXX provides the programmer with all advanced features of C++14, e.g., variadic templates, generic lambda expressions, as well as STL containers and algorithms. Reflection enables a program to modify its behaviour and is well known in languages like Java and Scala [15] but is unavailable in C++. In contrast to other approaches like Reflex [16] and XCppRefl [17] which aim to integrate full runtime reflection of the type system into C++, PACXX-Reflect follows a lightweight approach: we do not perform any type analysis or introspection.

3 The PACXX Programming Model

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

```
1 int main(){
2   vector<int> a{N}, b{N}, c{N};
3   for (int i = 0; i < N; ++i)
4     c[i] = a[i] + b[i];
5 }
```

Listing 1.1: Sequential vector addition in C++.

Listing 1.1 shows the sequential C++ source code for the vector addition. The memory is allocated and initialized with the default constructor of the vector's element type, here with 0 – following the RAII (Resource Acquisition Is Initialization) idiom – during the construction of the three STL containers of type `std::vector` in line 2. The calculation is performed by the for-loop in line 3.

```
1  __global__ void vadd (int* a,
2      int* b, int* c, size_t size){
3    auto i = threadIdx.x
4         + blockIdx.x * blockDim.x;
5    if (i < size)
6        c[i] = a[i] + b{i];
7  }
8  int main(){
9    vector<int> a{N}, b{N}, c{N};
10   int* da, db, dc;
11   cudaMalloc(da, N*sizeof(int));
12   cudaMemcpy(da,&a[0],sizeof(int)
13     * N, cudaMemcpyHostToDevice);}
14   ⋮ // 5 additional lines
15     // for vectors b and c
16   vadd<<<N/1024 + 1, 1024>>>
17               (da, db, dc, N);
18   cudaDeviceSynchronice();
19 }
```

```
int main(){
  vector<int> a{N}, b{N}, c{n};
  auto vadd = kernel([](
      const auto& a, const auto& b
      auto& c){
    auto i = Thread::get().global;
    if (i < a.size())
      c[i.x] = a[i.x] + b[i.x];
  }, {{N/1024 + 1}, {1024}});
  auto F = async(launch::kernel,
                 vadd, a, b);
  F.wait();
```

Listing 1.2: Two versions of parallel vector addition: in CUDA (left) and PACXX (right).

Listing 1.2 (left) shows how the vector addition is parallelized using CUDA. The CUDA kernel replaces the sequential for-loop of the C++ version with an implicitly 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. This restriction prohibits the use of the C++ Standard Template Library (STL) in the kernel code because the functions provided in the STL are not annotated and, therefore, callable only from the host code. The parameters of the `vadd` kernel are raw integer pointers. Memory accessed by the kernel must be managed by the developer explicitly (line 11). To use the kernel, memory is allocated on the GPU and the data is synchronized between both memories (line 12). On the host side, three instances of `std::vector` are used for the computation. Since STL features cannot be used in CUDA, three raw integer pointers are defined in line 10 to represent the memory of the GPU. Memory is allocated using the CUDA Runtime API; for brevity, the calls to this API are only shown for one vector. For each allocation and synchronization, an additional line of code is necessary (e.g., line 11). Passing arguments to a kernel by reference which is common in C++, is not possible in CUDA, so we have to use the pointers defined in line 10 and pass them by value. To launch the kernel, a launch configuration must be specified within `<<< >>>` in each kernel call (line 16), i.e., a CUDA-specific, non-C++ syntax is used. CUDA threads are organized in 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. While all threads execute the same kernel code, the work is partitioned using the index ranges; in our example, each thread computes a single addition depending on its absolute index in the x-dimension (line 3). The identifier of the thread within the grid and the block are obtained using variables `threadIdx`, `blockIdx`, `blockDim` and `gridDim` (not shown in the code). To prevent an out-of-bounds access of threads with $i >= N$, a so-called *if guard* (line 5) is used. However, the size of the vector has to be known in the kernel code and is passed as additional parameter to the kernel (line 2). The GPU works asynchronously to the host, therefore, the host execution must be explicitly synchronized with the GPU using the CUDA Runtime API (line 18).

Summarizing, the CUDA code is very different from the original C++ version. A complete restructuring and new language constructes are necessary, and the size of code increases significantly.

Listing 1.2 (right) shows the PACXX source code that performs the same computation. It is a pure C++14 code without any extensions (e.g., new keywords or special syntax), with the kernel code inside of the host code, that uses `std::async` and `std::future` from the STL concurrency library [1] to express parallelism on the GPU. In PACXX, there are no restrictions which functions can be called from kernel code, however, the code must be available at runtime, i.e., functions from pre-compiled libraries cannot be called. As in CUDA, a PACXX kernel is implicitly data parallel: it is defined with a C++ lambda function (lines 5-9). PACXX provides the C++ template class `kernel` (line 4) to identify kernels: instances of `kernel` will be executed in parallel on the GPU. The launch configuration (the second parameter in line 9) is defined analogous to CUDA. As in CUDA, threads are identified with up to three-dimensional index ranges which are used to partition the work amongst the GPU's threads. The thread's index can be obtained through the `Thread` class (line 6). To retrieve values from the thread's block, the class `Block` is available. The kernel instance created in line 3 is passed to the STL-function `std::async` (line 10) that invokes the kernel's execution

on the GPU. PACXX provides an STL implementation based on libc++ [18] where an additional launch policy (`launch::kernel`) for `std::async` is defined to identify kernel launches besides the standard policies of the C++ standard (`launch::async` and `launch::deferred`). Passing the `launch::kernel` policy to `std::async` (line 10) implies that the kernel should be executed on the GPU, rather than by another host thread. The additional parameters passed to `std::async` (line 11) are forwarded to the kernel. As a significant advantage over CUDA, parameters of a kernel in PACXX can be passed by reference, as with any other C++ function. PACXX manages the GPU memory implicitly [2], i.e., no additional API for memory management (as in CUDA and OpenCL) has to be used by the developer. Non-array parameters passed by reference are copied to the GPU prior to the kernel launch and are automatically synchronized back to the host when the kernel has finished. For arrays, the `std::vector` and `std::array` classes are used. As in the original C++ version, memory is allocated using the `std::vector` class (line 2). PACXX extends the STL containers `std::vector` and `std::array` with the lazy copying strategy: synchronization of the two distinct memories happens only when a data access really occurs, either on the GPU or on the host. This happens transparently for the developer and reduces the programming effort significantly. As compared, to the CUDA version, the PACXX code only needs 13 LoC, i.e., it is almost 50% shorter. The PACXX kernel also requires an *if guard* to prevent out-of-bounds access, but there is no need to pass the size of the vectors to the kernel as an additional parameter, because each instance of `std::vector` knows the number of contained elements which can be obtained by the vector's `size` function (line 7). The `std::async` function used to execute the kernel on the GPU returns an `std::future` object associated with the kernel launch (line 10); this object is used to synchronize the host execution with the asynchronous kernel more flexibly than in CUDA: the `wait` member function (line 12) blocks the host execution if the associated kernel has not finished yet.

Due to the exclusive usage of the C++14 syntax and implicit memory management of kernel parameters and STL containers, GPU programming using PACXX is shorter and easier for C++ developers than when using CUDA.

4 The PACXX-Reflect API

To exploit JIT compilation in CUDA or OpenCL, the source code must be prepared manually by the developer; it is commonly represented as a string, either in the executable itself or loaded from a source file.

Listing 1.3 compares the JIT-compilable version of vector addition in CUDA (left) and PACXX (right). In CUDA, a placeholder (`"#VSIZE#"`) is introduced into the kernel code. During program execution this placeholder is replaced by the string with the actual size of vector `a` (line 12– 19). However, changing the kernel code at runtime requires additional programming effort in the host code and, therefore, increases the program length. Furthermore, during the JIT compilation the kernel code passes all compiler stages including parsing, lexing and the generation of an abstract syntax tree (AST). Since the AST for the kernel code is created at runtime of the program, errors in the kernel code are found first at program execution which makes debugging more

```
vector<int> a{N}, b{N}, c{N};
string str{ R"(
extern "C"
__global__ void vadd(int* a,
                     int* b, int* c){
  auto i = threadIdx.x
       + blockIdx.x * blockDim.x;
  int size = #VSIZE#;
  if (i < size)
    c[i] = a[i] + b[i];
})" };
string v{to_string(a.size())};
string p{"#VSIZE#"};
string vadd{str};
auto npos = string::npos;
for(size_t i = 0;
  (i = vadd.find(p, i))!= npos;
   i += v.size())
  vadd.replace(i, p.size(), v);
```

```
vector<int> a{N}, b{N}, c{N};

auto vadd = [](const auto& a,
               const auto& b,
               auto& c){
  auto i = Thread::get().global.x;
  auto size =
  reflect([&]{return a.size();});
  if (i < size)
    c[i] = a[i] + b[i];
};
```

Listing 1.3: JIT-compilable vector addition in CUDA (left) and PACXX (right).

difficult. With the vector's size hard-coded into the kernel code, the additional parameter for the vector size as in Listing 1.2 becomes unnecessary, but this is only a minor optimization since kernel parameters are stored in the very fast constant memory of the GPU.

On the right-hand side of Listing 1.3, the same modification to the *if guard* is made using PACXX-Reflect: the dynamic part of the *if guard* condition is replaced by a constant value using the Reflect API. This is accomplished using the `reflect` function which is defined as a variadic template function that takes a lambda expression and an arbitrary number of additional parameters. The `reflect` function forwards the return value of the lambda expression passed to it (line 8); the expression is evaluated in the host's context, i.e., the instances of the `reflect` template function are JIT-compiled for the host architecture and executed on the host prior to the kernel's launch. Only constant values and kernel parameters are allowed as arguments or captured values by the lambda expression, because they are, from the kernel's point of view, compile-time constant values. In Listing 1.3 (right), a direct call to the `size` function of the `std::vector` requires two loads (of the `begin` and the `end` iterator) from the GPUs memory which might introduce overhead on some architectures. To avoid this, a lambda expression which wraps the call to the `size` function of vector `a` is passed to the `reflect` function; the returned value from `reflect` is considered static for the kernel execution and the function call is replaced by the computed value, such that no loads are necessary to retrieve the vector's size. Summarizing, the replacement of the function call with the constant value happens transparently for the developer.

5 Implementation and Evaluation

PACXX is implemented using LLVM [19] and uses its code generation libraries to generate PTX code [20] for Nvidia GPUs and SPIR code [21] for other architectures (e.g., Intel Xeon Phi accelerators). PACXX-Reflect operates on the LLVM IR (intermediate representation) generated by the PACXX offline compiler (implemented in Clang [22]), rather than on the source code itself, thus reducing the overhead of JIT compilation as compared to the current solutions for OpenCL and CUDA and avoids the name mangling problem by design.

For evaluation, two case studies programmed in C++14 and implemented using PACXX are compared to the reference implementations from the CUDA Toolkit [23]: 1) Black-Scholes computation [24] used in high-frequency stock trading, and 2) matrix multiplication [25].

To evaluate the performance of PACXX codes on other architectures than GPUs, the programs from the CUDA Toolkit were also manually re-written in OpenCL. The PACXX and CUDA implementations are compiled at runtime for CUDA Compute Capability 3.5 with standard optimizations enabled (no additional floating point optimizations), using the latest CUDA Toolkit (release 7.0). The CUDA PTX code is generated by the Nvidia NVRTC library; the host code is compiled by Clang 3.6 with standard `O3` optimizations.

For evaluation we use the state-of-the-art accelerator hardware: an Nvidia Tesla K20c GPU controlled by an Intel Xeon E5-1620 v2 at 3.7 GHz, and an Intel Xeon Phi 5110p hosted in a dual-socket HPC-node equipped with two Intel Xeon E5-2695 v2 CPUs at 2.4 GHz. For the Intel Xeon Phi and Intel Xeon, we used the Intel OpenCL SDK 2014. We employed the Nvidia profiler (nvprof) and the OpenCL internal facilities for performance measurements.

5.1 Black-Scholes computation

Our first example is the Black-Scholes (BS) model which describes the financial option market and its price evolution. Both programs, from CUDA Toolkit and the PACXX implementation, compute the Black-Scholes equation on randomly generated input data seeded with the same value to achieve reproducible and comparable results. Input data are generated for $81.92 \cdot 10^6$ options.

Figure 1 shows the measured runtime and the program size in Lines of Code (LoC); the latter is calculated for the source codes formatted by the Clang-format tool using the LLVM coding style. We observe that the OpenCL version (273 LoC) is about 3 times longer and the CUDA version (217 LoC) about 2 times longer than the PACXX code in pure C++ (107 LoC). The advantages of PACXX regarding the code size arise from the tasks performed transparently by the PACXX runtime: device initialization, memory management and JIT compilation, whereas in CUDA and OpenCL these tasks have to be performed explicitly.

The PACXX code has also advantages regarding runtime on the Nvidia K20c GPU (about 4.3%) and on the dual socket cluster node (about 6.7%). On the Intel Xeon Phi, the PACXX code is only about 0.1% slower than the OpenCL version. On the Nvidia Tesla K20c, since the OpenCL implementation from Nvidia is still for the OpenCL 1.1

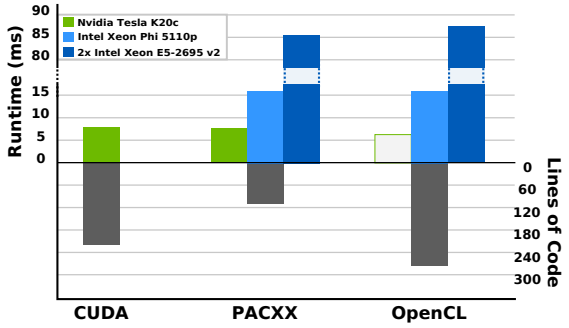

Fig. 1: Evaluation results for the Black-Scholes application.

standard where floating point division are not IEEE 754 compliant, the higher speed of the OpenCL version (16.1% faster than PACXX) is achieved for the lower accuracy.

5.2 Matrix Multiplication

Our second case study is the multiplication of dense, square matrices. The PACXX code uses the Reflect API to compute the sizes of the matrices: the `reflect` function retrieves the input vector's size and computes its square root to get the number of rows/-columns in the matrix. The `reflect` call is evaluated during the runtime compilation and the value (our matrices are of size 4096×4096) is automatically embedded into the kernel's intermediate representation. For the OpenCL and CUDA codes, the values are introduced into the kernel code by string replacement before it is JIT compiled.

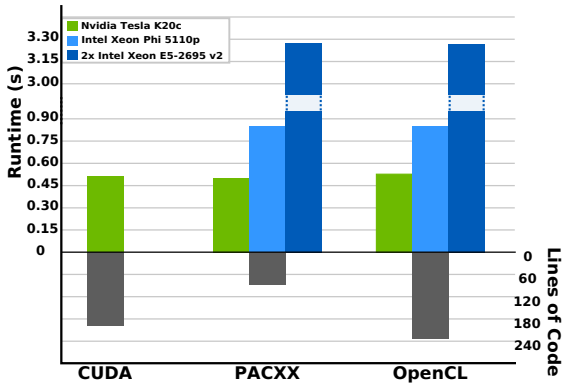

Fig. 2: Evaluation results for matrix multiplication.

Figure 2 (left) shows the runtime and size of the PACXX code as compared to the CUDA code (compiled with NVRTC) and the OpenCL code on the three evaluation platforms. The PACXX program is again much shorter than its CUDA and OpenCL

counterparts. We observe that the kernel becomes 2.7% faster on the K20c GPU when using PACXX-Reflect and its JIT capabilities. The PACXX code is about 0.2% slower on the Intel architectures as compared to the OpenCL implementation. On the Nvidia Tesla K20c, PACXX code outperforms CUDA NVRTC and OpenCL codes by 2.6% and 3.3%, correspondingly.

6 Conclusion

We presented PACXX – a programming model for GPU-based systems using C++14 and JIT compilation. We demonstrated that on modern accelerators (GPUs and Intel Xeon Phi) PACXX provides competitive performance and reduces the programming effort by more than 60% of LoCs as compared to CUDA or OpenCL. The code size reduction is achieved through JIT compilation and memory management tasks performed by PACXX implicitly in contrast to CUDA and OpenCL where they must be programmed explicitly. Additionally, PACXX enables application developers to program OpenCL and CUDA capable accelerators (e.g., Nvidia GPU and Intel Xeon Phi) in a modern object-oriented way using all advanced features of C++14 and the STL.

Bibliography

[1] isocpp.org. *Programming Languages - C++ (Committee Draft)*, 2014.

[2] Michael Haidl and Sergei Gorlatch. PACXX: Towards a Unified Programming Model for Programming Accelerators Using C++14. In *Proceedings of LLVM Compiler Infrastructure in HPC (LLVM-HPC) at Supercomputing 14*, pages 1–11. IEEE, 2014.

[3] Nvidia. *CUDA C Programming Guide*, 2015. Version 7.0.

[4] Herb Sutter and Andrei Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. Pearson Education, 2004.

[5] Khronos OpenCL Working Group. *The OpenCL C++ Specification*, 2015. Version 1.0.

[6] Microsoft. *C++ AMP : Language and Programming Model*, 2012. Version 1.0.

[7] Khronos OpenCL Working Group. *SYCL Specifcation*, 2015. Version 1.2.

[8] Nathan Bell and Jared Hoberock. Thrust: A Parallel Template Library. *GPU Computing Gems Jade Edition*, page 359, 2011.

[9] AMD. *Bolt C++ Template Library*, 2014. Version 1.2.

[10] James C. Beyer et al. OpenMP for Accelerators. In *OpenMP in the Petascale Era*, pages 108–121. Springer, 2011.

[11] openacc-standard.org. *The OpenACC Application Programming Interface*, 2013. Version 2.0a.

[12] Ping An et al. STAPL: An Adaptive, Generic Parallel C++ Library. In *Languages and Compilers for Parallel Computing*, pages 193–208. Springer, 2003.

[13] Nvidia. *NVRTC - CUDA Runtime Compilation - User Guide*, 2015.

[14] Thibaut Lutz and Vinod Grover. LambdaJIT: A Dynamic Compiler for Heterogeneous Optimizations of STL Algorithms. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-Performance Computing*, pages 99–108. ACM, 2014.

[15] Martin Odersky et al. The Scala Programming Language. *URL http://www.scala-lang.org*, 2008.

[16] S. Roiser. Reflex - Reflection in C++. In *Proceedings of Computing in High Energy and Nuclear Physics*, 2006.

[17] Tharaka Devadithya, Kenneth Chiu, and Wei Lu. C++ Reflection for High Performance Problem Solving Environments. In *Proceedings of the 2007 Spring Simulation Multiconference*, volume 2, pages 435–440. International Society for Computer Simulation, 2007.

[18] The LLVM Compiler Infrastructure. *libc++ C++ Standard Library*, 2014.

[19] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of International Symposium on Code Generation and Optimization, 2004. CGO 2004*, pages 75–86. IEEE, 2004.

[20] Nvidia. *PTX:Parallel Thread Execution ISA*, 2010. Version 4.2.

[21] Khronos OpenCL Working Group. *The SPIR Specification*, 2014. Version 1.2.

[22] Chris Lattner. LLVM and Clang: Next Generation Compiler Technology. In *Proceedings of the BSD Conference*, pages 1–2, 2008.

[23] Nvidia. *CUDA Toolkit 7.0*, 2015.

[24] Victor Podlozhnyuk. Black-Scholes Option Pricing. *CUDA Toolkit Documentation*, 2007.

[25] Vasily Volkov and James W Demmel. Benchmarking GPUs to Tune Dense Linear Algebra. In *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis, SC'08.*, pages 1–11. IEEE, 2008.