Multi-Stage Programming for GPUs in Modern C++ using PACXX

Michael Haidl* Michel Steuwer‡ Tim Humernbrum* Sergei Gorlatch*
m.haidl@uni-muenster.de michel.steuwer@ed.ac.uk t.hume@uni-muenster.de gorlatch@uni-muenster.de

*University of Muenster, Germany ‡University of Edinburgh, United Kingdom

Abstract

Writing and optimizing programs for high performance on systems with GPUs remains a challenging task even for expert programmers. One promising optimization technique is to evaluate parts of the program upfront on the CPU and embed the computed results in the GPU code allowing for more aggressive compiler optimizations. This technique is known as *multi-stage programming* and has proven to allow for significant performance benefits. Unfortunately, to achieve such optimizations in current GPU programming models like OpenCL, programmers are forced to manipulate the GPU source code as plain strings, which is error-prone and type-unsafe.

In this paper we describe PACXX - a GPU programming approach using modern C++ standards, with the convenient features like type deduction, lambda expressions, and algorithms from the standard template library (STL). Using PACXX, a GPU program is written as a single C++ program, rather than two distinct host and kernel programs. We extend PACXX with an easy-to-use and type-safe API for multi-stage programming avoiding the pitfalls of string manipulation. Using just-in-time compilation techniques, PACXX generates efficient GPU code at runtime.

Our evaluation shows that using PACXX allows for writing multi-stage code easier and safer than currently possible. Using two detailed application studies we show that multi-stage programming can significantly outperform equivalent non-staged programs. Furthermore, we show that PACXX generates code with performance comparable to industrial-strength OpenCL compilers.

Categories and Subject Descriptors D.3.4 [*Processors*]: Code generation, Compilers, Optimization

Keywords Multi-Stage Programming, GPUs, Modern C++, Runtime Optimization, Runtime Code Generation

1. Introduction

Graphics Processing Units (GPUs) are increasingly used in today's computer systems. However, writing high-performance programs for such systems remains a complicated task. Special programming models like CUDA [1] or OpenCL [2] must be mastered for writing the parts of an application program (so-called *kernels*) which are to be executed on a GPU. The kernels are written in a limited subset of C or C++ and are distinct from the rest of the program, the so-called *host program* which runs on a CPU. Crucially, OpenCL – the only programming approach which is portable across GPUs of different vendors – represents kernels as strings in the host program, making sharing of code between kernels and the host program a non-trivial engineering issue.

Multi-stage programming (MSP) is a programming technique for generating optimized programs at runtime. In MSP, parts of the program are evaluated at code generation time and the computed results are embedded in the generated program. This has proven to allow for significant performance benefits [3] enabling the compiler of the generated program to apply aggressive optimizations.

OpenCL programmers can exploit the idea of multi-staging by manually embedding values known in the host program into a string representing the OpenCL kernel before kernel compilation. Examples of such values might be the size of arrays or the number of threads executing the kernel. Projects like PyCUDA and PyOpenCL [4] go a step further and generate OpenCL kernels at runtime by carefully concatenating predefined string building blocks. However, these multi-staging techniques have an inherent weakness: the string embedding is not safe, e. g. there are no guarantees that a syntactically correct OpenCL kernel is produced or that type safety is maintained.

To support the development of optimized programs for GPUs, we propose our PACXX programming approach [5]. PACXX is a unified programming approach for programming GPUs using the newest C++ standard. Our approach allows programmers to write their applications as a single C++ program, rather than two distinct host and kernel programs, and to make use of modern C++ features like type deduction, lambda expressions, and algorithms from the standard template library (STL). The PACXX runtime automatically manages the program execution on the GPU including data transfers to and from the GPU. For supporting runtime optimizations, we extend PACXX with MSP capabilities where syntactical correctness and type safety are ensured automatically. Internally, PACXX uses just-in-time (JIT) compilation techniques to generate GPU code at runtime.

We demonstrate our approach on two well-known algorithms in parallel computing – Parallel Reduction and N-Body simulation. Using the Parallel Reduction example we show that MSP in PACXX is easy to use and safe, in contrast to MSP in OpenCL. Us-

ing the N-Body Simulation, we demonstrate how modern C++ and MSP can be used together in PACXX to abstract and implement hardware-specific optimizations of programs for GPUs.

In detail we make three main contributions:

1. We present our PACXX programming model for simplified, C++-based GPU programming and its implementation.
2. We integrate Multi-Stage Programming (MSP) into PACXX and describe our compiler-based implementation which ensures correctness of the staging and maintains type safety.
3. We evaluate the ease of use and performance of our approach using two detailed application studies.

The remainder of the paper is structured as follows. Section 2 briefly presents the PACXX approach to GPU programming. Section 3 discusses multi-stage programming for GPUs in OpenCL and describes how PACXX overcomes the shortcomings of OpenCL. Section 4 presents a case study of using PACXX and multi-stage programming. Section 5 discusses the PACXX implementation. We experimentally evaluate our approach in Section 6. Finally, Section 7 discusses related work and we conclude in Section 8.

2. GPU Programming in PACXX

In this section, we briefly introduce our PACXX (Programming Accelerators with C++) programming approach [5] and compare its programming style with OpenCL. In PACXX, systems with accelerators such as GPUs are programmed using pure C++, without specific language constructs like in OpenCL or CUDA. Throughout this and the next section we will use a common example for discussing GPU programming and multi-staging: parallel reduction.

GPU Programming in OpenCL Listing 1 shows the parallel reduction implemented in OpenCL. This implementation is provided by Nvidia as part of their SDK [6]. The listing shows all important steps required in OpenCL to execute a computation on a GPU, but the code is simplified due to space constrains. The program is divided into two distinct parts: 1) the *kernel program* written in OpenCL C which is represented as a string in OpenCL (lines 2–13); 2) the management program (called *host program* in OpenCL) which is usually written in C and makes calls to the OpenCL API for managing the kernel execution (lines 16–45).

In the kernel program (line 8), every thread copies an element from the input array to the fast local memory (`sm`). Then, a tree-based reduction is performed in the for loop in line 10 and then the final result is written to the output array in line 13.

The host program consists of nine steps highlighted with comments in Listing 1. These steps include: creating and compiling the kernel program (lines 18 and 20); creating a kernel from it (line 22); allocating OpenCL buffers for the input and output memory (lines 24–25); manually copying the input data to the GPU (lines 27–28); preparing and launching the kernel (lines 30–35); and copying the computed result back to the host program (lines 37–40). As OpenCL is by default an asynchronous API, we have to explicitly wait for all operations to finish (line 42). In an OpenCL kernel program it is only possible to synchronize inside a group of threads, called *workgroup*, using barriers (see lines 9 and 12) but it is not possible to synchronize across workgroups. Therefore, the shown kernel performs one reduction per workgroup and we finish the reduction on the host in line 44.

```
1  // Kernel program written in OpenCL C
2  char* code = "kernel                                \
3  void partReduce(global float *input,                \
4                  global float *ouput,                \
5                  int n, local float* sm) {           \
6   unsigned int lid = get_local_id(0);                \
7   unsigned int gid = get_global_id(0);               \
8   sm[lid] = (gid < n) ? input[gid] : 0;              \
9   barrier(CLK_LOCAL_MEM_FENCE);                      \
10  for(int s=get_local_size(0)/2; s>0; s>>=1){        \
11   if (lid < s) { sm[lid] += sm[lid + s]; }          \
12   barrier(CLK_LOCAL_MEM_FENCE); }                   \
13  if(lid==0) ouput[get_group_id(0)] = sm[0];}";
14
15 // Host program written in C using OpenCL API
16 float reduce(const float* inputPtr, int n) {
17  // 1. Creating program
18  program = clCreateProgramWithSource(code);
19  // 2. Compiling kernel
20  clBuildProgram(program)
21  // 3. Create kernel
22  kernel = clCreateKernel(program);
23  // 4. Alocate OpenCL buffers
24  inputBfr = clCreateBuffer(n*sizeof(float));
25  outputBfr = clCreateBuffer(n*sizeof(float));
26  // 5. Copying input data to the GPU
27  clEnqueueWriteBuffer(inputBfr, inputPtr,
28                       n*sizeof(float));
29  // 6. Preparing and launching kernel
30  clSetKernelArg(inputBuffer);
31  clSetKernelArg(outputBuffer);
32  clSetKernelArg(n);
33  clSetKernelArg(amountOfLocalMemory);
34  clEnqueueNDRangeKenel(glbSize, lclSize,
35                        kernel);
36  // 7. Copying result from the GPU
37  float* outputPtr =
38   malloc(glbSize/lclSize*sizeof(float));
39  clEnqueueReadBuffer(outputBfr, outputPtr,
40                      n*sizeof(float));
41  // 8. Wait for the operation to finish
42  clFinish();
43  // 9. Finish reduction on the host
44  return reduceOnHost(outputPtr);
45  }
```

Listing 1: Reduction example from the Nvidia OpenCL SDK [6].

GPU Programming in PACXX Listing 2 shows the same reduction example written in PACXX. While in OpenCL the program is divided into two parts, in PACXX we write a single C++ program. We use the PACXX provided `kernel` function (line 5) to specify the code to be executed in parallel on the GPU which is written as a C++ lambda expression (lines 6–17). We can see the same operations as in OpenCL of copying the data into the local memory (`sm`) in line 12, the for loop for the tree-based reduction (line 14), and the barriers in line 13 and 16. PACXX provides its own C++ API for accessing the thread identifiers (lines 8–10) and it uses a slightly different notation for the barriers which makes explicit that they work only for a group of threads (called *block* in PACXX in analogy to CUDA).

For launching the computation we call the `std::async` function (line 20) defined in the C++ standard, and we finish the computation on the CPU by using the C++ standard `std::accumulate` function (line 22).

```
1  // Single, unified C++ program
2  float reduce(const std::vector<float>& input){
3    std::vector<float> output(glbSize/lclSize);
4
5    auto redKernel = kernel(
6     [](const auto& input, auto& output){
7     shared_memory<float> sm;
8     auto block = Block::get();
9     auto lid = Thread::get().index.x;
10    auto gid = Thread::get().global.x;
11    auto n = input.size();
12    sm[tid] = (gid < n) ? input[gid] : 0;
13    block.synchronize();
14    for(int s=block.size.x/2; s>0; s>>=1) {
15      if (tid < s) { sm[tid] += sm[tid + s]; }
16      block.synchronize(); }
17    if (tid==0) ouput[block.index.x] = sm[0];
18   }, glbSize, lclSize, amountOfLocalMemory);
19
20   std::async(launch::kernel, redKernel,
21              input, output).wait();
22   return std::accumulate(output); }
```

Listing 2: Reduction example as a unified program in PACXX.

Comparison By comparing the listings 1 and 2 we can clearly see the advantages of PACXX over OpenCL.

PACXX provides a unified programming experience in a single C++ program whereas OpenCL separates the implementation into the distinct kernel and host programs. Data type definitions and functions can be easily shared and reused in PACXX across kernel and host code whereas they have to be written twice in OpenCL.

The management of the GPU is implicit in PACXX, it is significantly more detailed and cumbersome in OpenCL. PACXX is implemented as a compiler, as described in Section 5, and, therefore, compiles the kernel code automatically into an executable, whereas in OpenCL the compilation of the kernel code must be arranged manually (Steps 1–3). The memory management is performed automatically by PACXX using the standard C++ vector container and data is transferred automatically before a kernel is executed, while OpenCL requires explicit memory management with their custom OpenCL buffers (Steps 4, 5, and 7). The launching of a kernel in PACXX uses the C++ standard `async` function whereas a kernel launch in OpenCL is quite verbose (Step 6) as every kernel argument is set explicitly using a function call.

3. Multi-Stage Programming for GPUs

In this section we study how multi-stage programming can be used to optimize programs for GPUs. Using the reduction example we will show the potential benefits of multi-staging and how this is currently achieved in OpenCL. We will then identify inherent weaknesses of the existing multi-staging solutions and present how our extended approach based on extended PACXX overcomes them.

Multi-stage programming with OpenCL Listing 3 shows an optimized implementation of reduction in OpenCL by Nvidia [7] which applies, among other optimizations, also multi-staging. Again, the implementation is split into the kernel program (lines 1–20) and the host program (lines 22–31). We omit the steps 2–9 from the host program (line 30) as they are the same as in Listing 1.

The kernel program has been optimized as compared to Listing 1: a larger number of elements is reduced by a single thread without barrier synchronizations in the while loop in line 8. The for loop of the tree-based reduction has been unrolled (lines 15–19) and replaced by individual if statements.

```
1  char* code = "kernel                                    \
2  void partReduce(global float *input,                    \
3                  global float *ouput, int n,             \
4                  local volatile float* sm) {             \
5   int tid = get_local_id(0);                             \
6   int i = get_group_id(0)*(LOCAL_SIZE*2)+tid;            \
7   float sum = 0.0f;                                      \
8   while (i < n) {                                        \
9    sum += input[i];                                      \
10   if (N_IS_POW2 || i + LOCAL_SIZE < n) {                \
11     sum += input[i+LOCAL_SIZE]; }                       \
12   i += LOCAL_SIZE*2*get_num_groups(0); }                \
13  sm[tid] = sum;                                         \
14  barrier(CLK_LOCAL_MEM_FENCE);                          \
15  if (LOCAL_SIZE >= 512) {                               \
16    if (tid < 256) { sm[tid] += sm[tid+256];}            \
17    barrier(CLK_LOCAL_MEM_FENCE); }                      \
18  // ...                                                 \
19  if (LOCAL_SIZE >=  2)  { /*...*/ };                    \
20  if (tid==0) output[get_group_id(0)]=sm[0];}";
21
22 float reduce(const float* inputPtr, int n) {
23  char* define_lcl_size =
24   "#define LOCAL_SIZE " + lclSize + "\n";
25  char* define_n_is_pow2 =
26   "#define N_IS_POW2 " + isPow2(n) + "\n";
27  // 1. Creating program
28  program = clCreateProgramWithSource(
29    define_lcl_size + define_n_is_pow2 + code);
30  // Steps 2. - 9. as in Listing 1
31 }
```

Listing 3: Reduction using multi-stage programming from the Nvidia OpenCL SDK [6].

The multi-stage programming in Listing 3 is split across the host and kernel program. In the first part of multi-staging in the kernel program, the two identifiers `LOCAL_SIZE` and `N_IS_POW2` are used as constants, but they are not declared anywhere in the kernel program. These identifiers are defined in the host program (lines 23–26) as C macros represented as strings which are then combined with the kernel program in line 29. Therefore, `lclSize` and `isPow2(n)` are evaluated in the host program before the values are embedded in the kernel program. This allows the OpenCL compiler to statically evaluate some of the if statements in the kernel program, such as the one in line 15, and avoid generating a branching instruction at kernel runtime. As we will see in Section 6, removing these branches brings a significant performance benefit, improving the performance by up to 2× on some GPU architectures.

Similarly, by evaluating whether the input size `n` is a power of 2 and embedding this information statically in the kernel code, the kernel compiler can sometimes avoid the if statement in line 10, which is significant since it is called multiple times in a loop.

Problems of multi-staging in OpenCL Listing 3 demonstrates some inherent weaknesses of multi-stage programming in OpenCL. The staging is achieved by evaluating expressions in the host program and then embedding them in the kernel program by concatenating plain strings. In the example, macros are used for propagating the staged values in the kernel program. Working with a string representing the source code is potentially dangerous, as it is very easy to make mistakes which can only be detected at runtime of the application and not when the host program is compiled. There is no guarantee that the string manipulations result in an OpenCL kernel which is syntactically valid and in which the type safety between host and kernel program is maintained.

```
1  float reduce(const std::vector<float>& input){
2  std::vector<float> output(glbSize/lclSize);
3
4  auto redKernel = kernel(
5   [=](const auto& input, auto& output){
6   shared_memory<float> sm;
7
8   auto localSize = stage(lclSize);
9   auto n         = stage(input.size());
10  auto nIsPow2   = stage([](auto x) {
11                  return ((x&(x-1))==0); }, n);
12
13  auto block = Block::get();
14  auto tid = Thread::get().index.x;
15  auto i = block.index.x * localSize * 2 + t;
16  float sum = 0.0f;
17  while (i < n) {
18    sum += input[i];
19    if (nIsPow2 || i + localSize < n) {
20      sum += input[i + localSize]; }
21    i += localSize *2* Grid::get().range.x; }
22  sm[tid] = sum;
23  block.synchronize();
24  if (localSize >= 512) {
25    if (tid < 256) { sm[tid] += sm[tid+256];}
26    block.synchronize(); }
27  // ...
28  if (localSize >=  2) { /* ... */ }
29  if (tid == 0) output[block.index.x] = sum;
30 }, glbSize, lclSize, amountOfLocalMemory);
31
32 std::async(launch::kernel, redKernel,
33            input, output).wait();
34 return std::accumulate(output); }
```

Listing 4: Reduction in PACXX using multi-stage programming.

Multi-stage programming with PACXX We extended our PACXX programming model with multi-staging capabilities to provide an easy-to-use and safe API for multi-stage programming that overcomes the mentioned weaknesses of OpenCL.

Listing 4 shows the reduction example with multi-stage programming implemented in PACXX. The code implements the same optimizations as in the OpenCL version in Listing 3. The key difference is the handling of the multi-staging. PACXX provides a special function named `stage` which can be called from the code executed on a GPU. The expression or function wrapped in a `stage` call are evaluated on the CPU *prior* to the execution on the GPU and the computed result is embedded into the kernel code.

For our example in Listing 4, this means that the expressions in lines 8, 9 and 10 are evaluated before the kernel is launched. In line 8, the value of `localSize`, which is passed as parameter lSize to the kernel, becomes a constant known to the kernel compiler. In line 9, the size of the input `n` is obtained from the `size` function invoked on the input vector. Finally, in line 10, the `nIsPow2` value is computed by evaluating the lambda expression in lines 10 and 11 on the CPU. This instance of the stage function takes `n` as an argument, which is only valid because `n` itself is a staged value. The PACXX implementation ensures that no expressions which are evaluated in the kernel code are passed as arguments to a stage function as these expressions will only be available after the kernel executes on the GPU. Due to the PACXX implementation of the `stage` function, which we will discuss in the following section, all type information is preserved and the usual C++ type safety guarantees are maintained.

Comparison By comparing the two listings 3 and 4, we can see the clear advantages of the PACXX multi-staging API as compared to the string handling in OpenCL.

First, the PACXX API is easy to use: a single `stage` function is used at the point where the code should be embedded into the kernel program. In contrast, OpenCL splits the staging across host and kernel program and involves cumbersome string handling.

Second, PACXX guarantees by design a syntactically correct program, whereas this is not always the case in OpenCL. Especially, errors in the string handling in OpenCL are only detected at runtime, whereas in PACXX errors are detected at compile time.

Finally, PACXX guarantees type safety thanks to its implementation built on top of the Clang and LLVM compiler frameworks. In OpenCL it is easy to introduce type errors where the host and kernel program disagree on the type of a certain value. Such errors might not even be caught at runtime: each part of the application would interpret the underlying bits differently which can lead to subtle and hard to find bugs in the program.

4. Application Study: N-Body Simulation

In this section we describe an application study implemented with PACXX. We will show how the power of C++ available in PACXX together with multi-staging enables programmers to conveniently express and efficiently implement GPU-specific optimizations.

N-Body simulations are an important class of physical applications. For a number of particles (also called *bodies*), each with a position and a velocity, the interaction between all particles is computed in an iterative process which updates the position and velocity for each particle in every step.

N-Body implementation in PACXX Listing 5 shows the code of the computation kernel performing one iteration step where one thread computes a new position and velocity of a particle. Each thread loads a particle `p` in line 10 and then computes the interaction with all other particles by iterating over them using the `for_each` function in line 13. After loading the corresponding velocity `v` in line 25, the new particle position and velocity are computed and written to memory in lines 33 and 34.

Multi-staging is used in line 7 for making the number of particles available to the compiler. PACXX implicitly stages the launch configuration, thus, the highest global id becomes known to the compiler. Therefore, the compiler can statically evaluate the comparison in line 7 and, if not too many threads are launched, the compiler will remove the branch instructions in lines 9, 24, and 32 which depend on the `disabled` value.

Abstraction of loop tiling with PACXX The `for_each` function used in Listing 5 is not provided by PACXX; it is rather an example of how application developers can implement their own abstractions using the power of C++ available in PACXX.

Listing 6 shows the implementation of the `for_each` function which is part of the N-Body program. The function iterates over a vector `v` and calls the function `func` on every element. The implementation is optimized for GPUs: it makes use of the fast local memory by applying *loop tiling* which is useful when multiple threads iterate over the same memory area. The actual iteration is split into two for loops (line 9 and line 12). In the outer loop, multiple threads iterate simultaneously over chunks of memory, called *tiles*. In the inner loop, each thread iterates over all elements of a single tile. This is advantageous, as a tile is loaded into the fast local memory in line 10 and, therefore, each element is only accessed once in the slow global memory and multiple times in the fast local memory.

```
1  #define sq(x) ((x)*(x))
2  #define cu(x) ((x)*(x)*(x))
3
4  auto nbody = [eps2 = 0.00125f]
5   (const auto &pos, auto &npos, auto &vel) {
6    auto idx = Thread::get().global.x;
7    bool disabled = idx >= stage(pos.size());
8    data_t p, v;
9    if (!disabled) {
10      p = pos[idx]; }
11   data_t a = {0.0f, 0.0f, 0.0f, 0.0f};
12   data_t r = {0.0f, 0.0f, 0.0f, 0.0f};
13   for_each(pos, [&](auto elem) {
14     r.x = p.x - elem.x;
15     r.y = p.y - elem.y;
16     r.z = p.z - elem.z;
17     r.w = 1.0f / std::sqrt(sq(r.x) + sq(r.y)
18                            + sq(r.z) + eps2);
19     a.w = G * elem.w * cu(r.w);
20     a.x += a.w * r.x;
21     a.y += a.w * r.y;
22     a.z += a.w * r.z;
23   });
24   if (!disabled) {
25     v = vel[idx]; }
26   p.x += v.x * dt + a.x * 0.5f * sq(dt);
27   p.y += v.y * dt + a.y * 0.5f * sq(dt);
28   p.z += v.z * dt + a.z * 0.5f * sq(dt);
29   v.x += a.x * dt;
30   v.y += a.y * dt;
31   v.z += a.z * dt;
32   if (!disabled) {
33     vel[idx]  = v;
34     npos[idx] = p; } };
```

Listing 5: N-Body kernel using the `for_each` function.

Multi-staging is used here for the length of the input vector (line 3). Together with the implicitly staged launch configuration, the trip counts for both loops become known at compile time which increases the likelihood for unrolling them. In addition, the case when the input size is not evenly divisible by the size of a tile can be handled (line 16) without deteriorating performance as compared to the case when the input size is evenly divisible and the branch can be removed statically.

5. The PACXX Implementation

This section discusses the PACXX implementation. We start with the overall design of the PACXX framework, then we discuss challenges for the implementation of multi-staging in C++ and how the PACXX implementation addresses these.

5.1 Overview of the PACXX Implementation

PACXX transforms C++ code using a combination of offline and online compilation to a representation executable on a GPU: PTX on Nvidia GPUs, and SPIR on AMD GPUs.

Figure 1 gives an overview of the PACXX implementation which comprises two main components:

1) The *PACXX Offline Compiler* is based on Clang 3.7 [8] – an open source compiler front-end for the C language family with feature-complete C++14 support, and
2) The *PACXX Runtime* library is statically linked into the executable; it consists of a just-in-time compiler implemented using the LLVM library [9], and specific GPU back-ends which use the CUDA and OpenCL runtime libraries.

```
1  template <typename T, typename F>
2  void for_each(T &v, F func) {
3    auto s = stage([&]{return v.size();});
4    shared_memory<typename T::value_type> sm;
5    auto block = Block::get();
6    auto bSize = block.range.x;
7    auto tidx = Thread::get().index.x;
8    int trips = s  / bSize;
9    for (int i = 0; i < trips; ++i) {
10     sm[tidx] = v[i * bSize + tidx];
11     block.synchronize();
12     for (int j = 0; j < bSize; ++j)
13       func(sm[j]);
14     block.synchronize();
15   }
16   if (s % bSize != 0)
17     for (int j = trips * bSize; j < s; ++j)
18       func(v[j]);
19 }
```

Listing 6: Abstraction of loop tiling using multi staging.

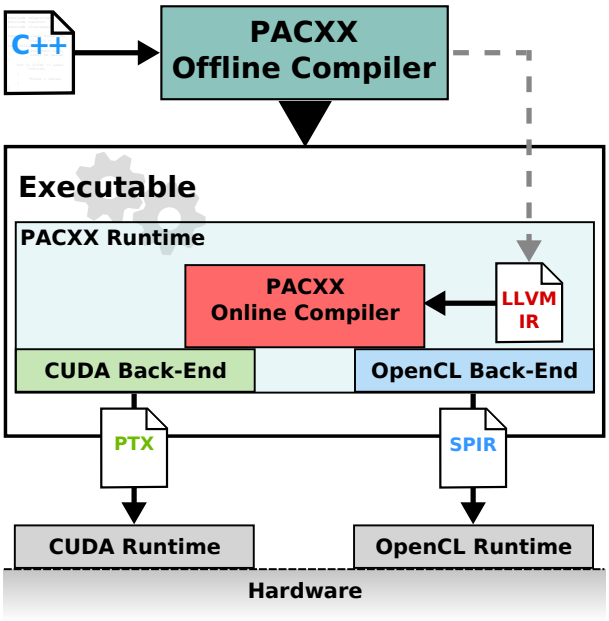

Figure 1: Key components of PACXX.

Correspondingly, C++ code is compiled by PACXX in two stages: 1) the offline compilation stage separates the GPU code from the CPU code and prepares the executable for the PACXX runtime, 2) the online compilation stage during program execution just-in-time compiles the code for the GPU using our LLVM-based online compiler contained in the PACXX runtime library.

PACXX offline compiler In PACXX, code executed on a GPU is wrapped in the `kernel` function provided by PACXX. Internally, lambda expressions and functions passed to this function are annotated with the [[pacxx::kernel]] generalized attribute, a feature introduced in the C++11 standard. The PACXX offline compiler automatically identifies all the code which should run on the GPU by annotating every function called from inside the original kernel function with the same attribute. Using generalized attributes has the advantage that the code remains valid C++, and other compil-

ers have the freedom to ignore PACXX custom annotations. In the PACXX compiler (as in Clang), attributes are part of the abstract syntax tree (AST) built from the C++ source code.

After the annotations are added, the PACXX offline compiler performs two separate passes: the first pass for preparing the GPU code generation at runtime and the second pass for compiling the CPU program.

In the first pass, *the kernel compilation pass*, the entire AST is lowered to the LLVM intermediate representation (IR) and functions with the `[[pacxx::kernel]]` attribute are enriched with special metadata to identify them as kernel code in the IR. The enriched IR is then transformed in the following steps:

1) aggressive dead code elimination removes everything from the IR besides the kernel and functions called from the kernel;
2) a custom inliner tries to inline as many function calls as possible into the kernel;
3) the kernel functions are optimized with standard optimizations (equal to `O3` optimizations);
4) the final IR is wrapped in an object file and passed to the linker.

The PACXX runtime library then loads this prepared IR and compiles it at runtime for a particular GPU.

In the second pass, *the host compilation pass*, the PACXX offline compiler lowers the AST to LLVM IR a second time, but this time the calls to functions with the `[[pacxx::kernel]]` attribute are replaced with calls to the PACXX runtime library for managing data transfers and launching the corresponding kernel. Finally, the generated IR is lowered for the specific host architecture and object files are generated as usually for C++ programs. The PACXX runtime library is statically linked into the final executable, as shown in the bottom half of Figure 1.

PACXX runtime During program execution, the PACXX runtime loads the integrated IR from inside the executable. Additional optimization passes perform GPU-specific optimizations, such as loop-unrolling and rearranging of load instructions. Finally, the IR is lowered to GPU code using the most appropriate LLVM compiler back-end: we use PTX [10] together with the CUDA runtime library when targeting Nvidia GPUs, and SPIR [11] for GPUs with an OpenCL implementation (e.g., from AMD and Intel).

5.2 Implementation of Multi-Stage Programming in the PACXX Compiler

As described in Section 2, the PACXX programmer uses the `stage` function for multi-stage programming: expressions are evaluated prior to the kernel execution and their computed values are embedded into the kernel program. We saw two variations of the `stage` function: 1) where a single expression is provided (e.g., the variable `lclSize` in Listing 4 line 8), and 2) where a function and corresponding arguments are provided (e.g., evaluating whether `n` is a power of 2 in Listing 4 line 10). Internally, PACXX unifies these two variants: if an expression is provided, it is wrapped in a lambda expression, therefore, making the first variant a special case of the more general second variant.

Overview When performing the kernel compilation pass, the PACXX offline compiler handles calls to the `stage` function in a special manner, because we want to separate the code wrapped by the `stage` function from the rest of the kernel program. To prevent optimization passes from inlining and, therefore, combining the staged code with the kernel program, the function passed as an argument to `stage` is annotated with the `noinline` attribute provided by Clang. Then, the steps 1) dead code elimination, 2) inlining, and 3) optimizations, are performed as described in the previous subsection. Before the last step of wrapping the final IR in the object file, code for calling staged functions as well for the staged functions themselves is generated.

For each staged function, a corresponding new function is generated which will eventually be evaluated at runtime on the host prior to executing the kernel program. The code for the staged function is removed from the kernel program and call instructions to the function are replaced by call instructions to a proxy function `pacxx_eval`. These calls will be replaced at runtime with the values obtained by evaluating the staged function on the host.

Generating Staged Functions For every staged function, a corresponding function is generated and its IR is embedded into the executable. The IR for staged functions is separated from the IR for the kernel program. In the next section, we will see how the IR of the staged functions is loaded at runtime, just-in-time compiled, evaluated on the host and the computed results are embedded into the kernel program. Here we describe how for a staged function its corresponding new function is generated.

From the examples presented so far, it might seem straightforward to identify the code which should be staged and executed on the host. But that is not always the case. Consider the following example.

```
1 auto n = stage(inputSize);
2      n = n / 2;
3 auto b = stage([](auto x){return x<1024;}, n);
```

Listing 7: Staging using a value modified in the kernel program.

In line 1 of Listing 7, the value `inputSize` is staged and then modified in the kernel program in line 2. The second `stage` call in line 3 now depends on a value which is computed in the kernel program. However, this is still safe, as the computation in line 2 only depends on a staged value (`n`) and a constant value (`2`) available at compile time. We could forbid this behavior and require that each `stage` call only directly depends on constants or staged values. Instead, we allow staged functions to depend *indirectly* on constants and staged values, as in the example.

We achieve this by implementing an LLVM IR transformation pass which performs the following four steps:

1. The call instructions to the `stage` function are identified in the kernel code.
2. For each call, all instructions before the call instruction itself are cloned into a new function named `pacxx_staged_eval#` (where `#` is a unique identifier). A new return instruction is added returning the value computed by the staged function and the function's return type is changed appropriately.
3. Branches not leading to the staged function call are removed.
4. A second function `pacxx_wrapped_eval#` is generated which provides a unified interface to be called by the PACXX runtime, as we will discuss in the next subsection.

5.3 Implementation of Multi-Stage Programming in the PACXX Runtime

We will now describe how the PACXX runtime evaluates the `pacxx_staged_eval#` functions on the host at runtime and how the computed values are embedded into the kernel program prior to its execution on the GPU.

The PACXX 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 kernel IR is modified.
4. The kernel is just-in-time compiled and launched.

The PACXX offline compiler generates code for calling the PACXX runtime to perform these four steps. The first two steps are straightforward. We will describe the two last steps in the following.

Staged Function Evaluation To evaluate the staged functions in step 3, their IR is loaded from the executable and just-in-time compiled for the host architecture. For every staged function, the corresponding `pacxx_wrapped_eval#` function is called using the unified C++ interface: `void(void*, void*)`. The first argument is a pointer to an array holding all input arguments and the second argument points to a memory location for the output value. The PACXX runtime copies all arguments to the heap and allocates memory for the output value prior to calling the function.

The kernel program's IR is then modified by replacing the calls to the proxy function `pacxx_eval`, inserted by the PACXX offline compiler, with a constant expression of the evaluated value.

Kernel Compilation and Launch After the staged values have been embedded into the kernel program, the PACXX runtime performs some additional optimizations on the code: the information of the launch configuration, i. e., the global and local size, is always embedded into the kernel program. This can be viewed as an implicit staging of the launch configuration. A special pass optimizes the control flow graph to remove branches that are never entered by a thread on the GPU.

The kernel program is lowered to the machine code representation by one of two backends currently implemented in the PACXX runtime: PTX [10] for CUDA and SPIR [11] for OpenCL. Finally, the generated GPU code is linked by the corresponding CUDA or OpenCL runtime.

To minimize the overhead of the just-in-time compilation process, the compiled kernel code is cached by PACXX. If the kernel is launched again, all staged functions are evaluated again and their results are checked against the previous results stored internally by PACXX. If all results are equal then the cached kernel code is still valid and can be launched straight away. If staged values have changed, the kernel compilation process is repeated to generate a new version of the kernel code which is then launched as usual.

6. Experimental Evaluation

In this section, we evaluate two case studies which make use of multi-stage programming, in PACXX and OpenCL: the reduction example and the N-Body simulation.

Experimental Setup We used three GPUs for our evaluation: 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.

Kernel runtimes are reported as median of 1000 runs measured using the OpenCL profiling API and the Nvidie profiler.

6.1 Parallel Reduction

The reduction OpenCL implementation is taken from the Nvidia's SDK [6]. We discussed the parallel reduction implementations in Section 2 and Section 3. We evaluate the programs shown in Listing 3 (OpenCL) and Listing 4 (PACXX). To observe the impact of multi-staging, we also created two corresponding programs which do not use the multi-stage optimization.

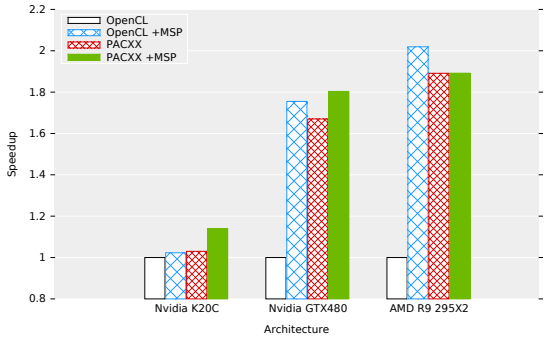

Figure 2: Speedup of PACXX over OpenCL with and without multi-staging for parallel reduction, input off 2^{27} integers.

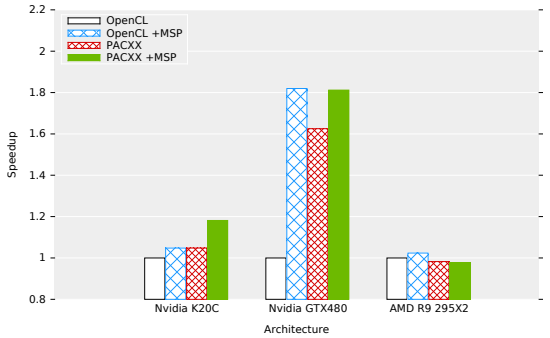

Figure 3: Speedup of PACXX over OpenCL with and without multi-staging for parallel reduction, input of $2^{27} + 172$ integers.

We evaluate OpenCL and PACXX with and without multi-staging using two input sizes: 2^{27} and $2^{27} + 172$ (as one of the staged functions decides whether the input is a power of two, we are interested on the effect of the input size on the performance). Figure 2 shows the results for the power of two input size and Figure 3 for the other input size.

Depending on the architecture, we observe speedups of up to $2\times$ due to multi-staging. We observe that using multi-staging for removing branch instructions is beneficial on the Fermi architecture (GTX 480) and the AMD GPU. The Kepler architecture has reduced the cost of branch instructions and, therefore, benefits less in this use case.

Interestingly, from comparing the results for AMD across the two figures, we obverse a significant impact of the input size on performance; therefore, removing the branch which depends on the input size being a power of two is crucial for performance on the AMD GPU. This is not the case for the Nvidia GTX 480 where removing bra nches using multi-staging is beneficial independently of the input size.

The speedup of the PACXX implementation without multi-staging as compared to OpenCL on the GTX 480 GPU and the AMD GPU results from the implicit staging of the launch configuration by PACXX online compiler. This implicit staging results in nearly the same kernel code as in the PACXX version using multi-staging, removing dead branches in the tree-based reduction.

On the Kepler architecture, branches are not as costly, and the performance improvements in the PACXX version using MSP results from the aggressive loop unrolling by the online compiler when compiling for Nvidia GPUs. For the AMD GPU, the loop unrolling is done more conservatively and shows no performance improvements at all.

6.2 N-Body Simulation

We compared the runtime of the N-Body simulation presented in Section 4 against an equivalent OpenCL implementation applying the same loop tiling optimization. The N-Body OpenCL implementation is a manually extended version of the implementation provided by Nvidia [12]. Our extended version is capable of handling arbitrary number of particles while Nvidia's original version only handles certain input sizes. As with the parallel reduction, we also implemented versions without multi-staging to observe the performance implications. We evaluated with ten different numbers of particles ranging from 2^{10} to 2^{19}.

Nvidia K20c Results The experimental results for the Nvidia K20c are shown in Figure 4 as speedups vs. the OpenCL implementation not using multi-staging. The PACXX implementation with multi-staging has a clear performance advantage over all other implementations. Performance improves by up to 1.4 times as compared with OpenCL. The performance advantage of PACXX results from the following two main reasons.

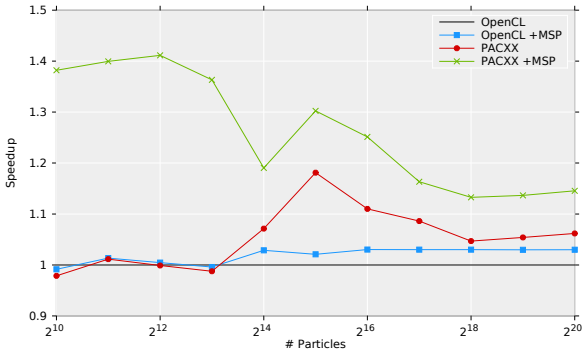

Figure 4: Speedup of the N-Body simulation on a Nvidia K20c as compared to OpenCL without multi-staging.

The first reason is the aggressive loop unrolling performed by the PACXX online compiler. For the PACXX version with multi-staging, both loops are unrolled, due to the information available through multi-staging. The PACXX online compiler performs a more aggressive loop unrolling than the Nvidia OpenCL compiler, ignoring possible performance losses through cache misses in the instruction cache. Our study of the PTX binaries generated by the Nvidia OpenCL compiler shows that the outer loop is not unrolled even though the number of iterations is known statically. For the version without multi-staging PACXX decides not to unroll the outer loop as it is done by the OpenCL compiler, because without knowing the loop condition exactly, branching inside of the loop would be necessary and would introduce negative effects on the kernels performance. The performance benefit of unrolling the outer loop is more significant for smaller input sizes than it is for larger ones, explaining the decreasing speedup of the multi-staging version of PACXX compared to the version without multi-staging.

The second reason originates from different register usage. PACXX lowers the LLVM IR to PTX without performing specific optimizations for reducing the number of registers. For this application, Nvidia's compiler generates PTX code using 35 registers, while the PACXX versions use 37 registers.

These two reasons result in the observed speedup which decreases for larger input sizes, because the memory transfers on the global memory start to dominate the performance and the advantages of our generated PTX code are mitigated.

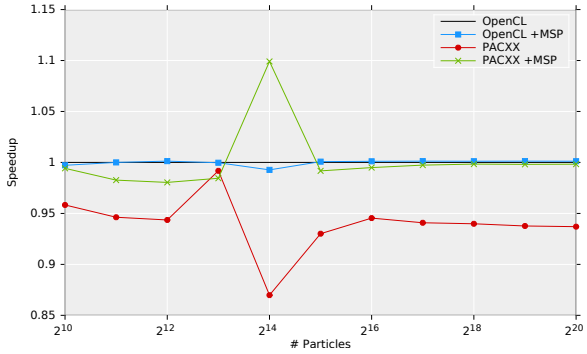

Figure 5: Speedup of the N-Body simulation on an Nvidia GTX 480 GPU as compared to OpenCL without multi-staging.

Nvidia GTX 480 Results Figure 5 shows the performance results for the Nvidia GTX 480. The PACXX version without multi-staging is about 5% slower than the OpenCL implementations. This results from the CUDA Toolkit version (6.5) and the OpenCL 1.1 driver shipped with it, which is used by the OpenCL implementations but not the PACXX implementations. The use of OpenCL 1.1 driver results in faster code for the OpenCL implementations because non IEEE 754 compliant floating point optimizations are performed by the OpenCL compiler. However, PACXX uses the proper floating point operations. The PACXX version with multi-staging compensates this disadvantage and is on-par with the OpenCL implementations without loosing floating point accuracy. As described in the previews paragraph, the Nvidia OpenCL compiler does not unroll the outer loop and performance of the multi-staging version is equal to the version without multi-staging.

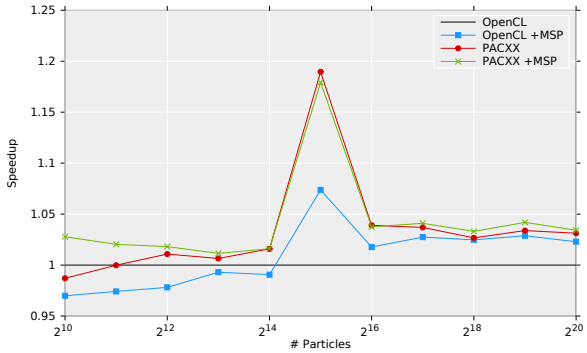

Figure 6: Speedup of the N-Body simulation on an AMD R295X2 GPU as compared to OpenCL without multi-staging.

AMD R295X2 Results Figure 6 shows the speedup of PACXX over the OpenCL version without multi-staging for the AMD R295X2 GPU. On the AMD architecture ,all implementations are close for most input sizes. Using multi-staging on the AMD GPU does not provides as much advantage as on the Nvidia GPUs for this particular application. The optimizations performed by the PACXX online compiler, such as aggressive loop unrolling, is currently better tuned for Nvidia GPUs and not as effective on AMD architectures. Interestingly, the spike for 2^{15} particles is a result of a performance drop in the baseline OpenCL implementation. We executed the same kernel across all GPUs and inputs sizes, and we did not observe this behavior elsewhere, therefore, we believe this to be an architecture-specific behavior related to the kernel launch configuration.

7. Related Work

There exist projects related to ours in the area of GPU programming and compilation as well as multi-stage programming.

GPU programming CUDA [1] and OpenCL [2] are the two main programming approaches used for GPU programming today. As discussed earlier, OpenCL separates an application into host and kernel program. In CUDA, programs are implemented in a single but not standard-conform C++ program and the functions executed on the GPU still have to be explicitly annotated. Multi-stage programming, as discussed in this paper, is not possible in CUDA as the kernel program is not compiled at runtime and, therefore, no runtime values can be embedded in the kernel program. PACXX offers a unified C++ programming approach with a safe and easy-to-use interface for multi-stage programming.

SYCL [13] is a recently developed high-level interface that integrates the OpenCL programming model into C++. However, SYCL still requires explicit memory management using provided `Buffers` in the host and kernel code, while in PACXX the memory management happens implicitly for the programmer. Furthermore, multi-stage programming is not possible in SYCL as the kernel and host program are compiled together as in CUDA.

Concord [14] is another approach for integrating GPU programming into C++. As PACXX, Concord is built on top of LLVM, however, Concord compiles the C++ code to OpenCL C code while PACXX generates LLVM intermediate representation directly. Concord uses the advanced shared virtual memory (SVM) features from OpenCL 2.0 to provide a transparent memory handling, especially suitable for pointer-intense data structures. In PACXX, these irregular data structures must be manually maintained by the programmer. The portability of Concord programs is limited to GPUs from hardware vendors providing an OpenCL 2.0 implementation supporting SVM which, e.g., currently excludes Nvidia GPUs. The implicit memory handling in PACXX does not rely on the SVM features of OpenCL to provide better portability.

To simplify GPU programming, projects like Thrust [15], Bolt [16], and SkelCL [17] provide generic patterns of parallel programming which are customized by application programmers. While these abstractions simplify GPU programming, it is often hard to implement application-specific optimizations, like the loop tiling optimization implemented with PACXX and applied in this paper for the N-body simulation.

Just-in-time GPU Compilation LambdaJIT [18] is a JIT approach similar to PACXX: GPU code is compiled at runtime from C++ lambda expressions used in algorithms from the C++ standard library. A limited form of multi-staging is supported by LambdaJIT: variables captured by the lambda expression by-value are embedded as constants in the GPU code enabling similar optimizations as available in PACXX. By providing the `stage` function, PACXX allows a general purpose usage of multi-staging in the kernel code compared to LambdaJIT.

Multi-Stage Programming MSP was pioneered by Taha [19] and first introduced into the functional programming languages MetaML [20] and MetaOcaML [21]. Multiple efforts have been made to make MSP available in other languages, including Mint [22] for Java, LMS [23] for Scala, and Terra [24] for Lua [25]. None of these specifically target GPU programming like PACXX does.

The Lightweight Modular Staging [23] framework implemented in Scala builds the foundation of the Delite [26] project which simplifies the development of domain-specific languages (DSL) for parallel processors including GPUs. Multi-staging can be used in DSLs to generate more efficient GPU code with similar optimizations as presented in this paper for C++ and STL.

8. Conclusion

In this paper we present PACXX – a unified programming approach for GPU using modern C++ with support for multi-stage programming. We show that our programming model provides a unified programming experience for application developers and does not split the program into separate parts as OpenCL does. This results in shorter programs as compared to OpenCL as type declarations and commonly used functions can be reused and do not have to be reimplemented as in OpenCL.

PACXX offers support for multi-stage programming, such that values computed on the CPU at runtime are embedded into the GPU program enabling the just-in-time compiler to generate more efficient GPU code. We demonstrate that, depending on the architecture, multi-stage programming can provide significant speedups of up to $2\times$ as compared to code not using this optimization technique. Multi-stage programming is not possible in CUDA, as the GPU code is not accessible at runtime. In OpenCL multi-staging can be used, but is cumbersome and error-prone as plain strings have to be manipulated explicitly. PACXX provides a type-safe and easy-to-use interface for multi-staging.

Acknowledgments

The authors would like to thank NVIDIA Corp. for their generous hardware donations supporting this work.

References

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

[2] Khronos OpenCL Working Group. *The OpenCL Specification*, 2012.

[3] Tiark Rompf, Kevin J. Brown, HyoukJoong Lee, Arvind K. Sujeeth, et al. Go meta! A case for generative programming and DSLs in performance critical systems. In *1st Summit on Advances in Programming Languages, SNAPL 2015*, volume 32 of *LIPIcs*, pages 238–261. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[4] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan C. Catanzaro, Paul Ivanov, and Ahmed Fasih. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. *Parallel Computing*, 38(3):157–174, 2012.

[5] 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.

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

[7] Mark Harris. *Optimizing Parallel Reduction in CUDA*. Nvidia, 2007.

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

[9] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO 2004*, pages 75–86. IEEE, 2004.

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

[11] Khronos OpenCL Working Group. *The SPIR Specification*, 2014.

[12] Lars Nyland, Mark Harris, and Jan Prins. Fast N-Body Simulation with CUDA. *GPU Gems*, 3(1):677–696, 2007.

[13] Khronos OpenCL Working Group. *SYCL Specifcation*, 2015.

[14] Rajkishore Barik, Rashid Kaleem, Deepak Majeti, Brian T Lewis, Tatiana Shpeisman, Chunling Hu, Yang Ni, and Ali-Reza Adl-Tabatabai. Efficient Mapping of Irregular C++ Applications to Integrated GPUs. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 33. ACM, 2014.

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

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

[17] Michel Steuwer, Philipp Kegel, and Sergei Gorlatch. SkelCL - A Portable Skeleton Library for High-Level GPU Programming. In

Workshop on High-Level Parallel Programming Models and Supportive Environments at IPDPS 2011, pages 1176–1182. IEEE, 2011.

[18] Thibaut Lutz and Vinod Grover. LambdaJIT: A Dynamic Compiler for Heterogeneous Optimizations of STL Algorithms. In *Workshop on Functional High-Performance Computing at ICFP*, pages 99–108. ACM, 2014.

[19] Walid Taha. A Gentle Introduction to Multi-Stage Programming. In *Domain-Specific Program Generation*, pages 30–50. Springer, 2004.

[20] Walid Taha and Tim Sheard. Multi-Stage Programming with Explicit Annotations. In *ACM SIGPLAN Notices*, volume 32, pages 203–217. ACM, 1997.

[21] Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. Implementing Multi-Stage Languages Using ASTs, Gensym, and Reflection. In *Generative Programming and Component Engineering*, pages 57–76. Springer, 2003.

[22] Edwin Westbrook, Mathias Ricken, Jun Inoue, Yilong Yao, Tamer Abdelatif, and Walid Taha. Mint: Java Multi-Stage Programming Using Weak Separability. *ACM SIGPLAN Notices*, 45(6):400–411, 2010.

[23] Tiark Rompf and Martin Odersky. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *ACM SIGPLAN Notices*, volume 46, pages 127–136. ACM, 2010.

[24] Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. Terra: A Multi-Stage Language for High-Performance Computing. In *ACM SIGPLAN Notices*, volume 48, pages 105–116. ACM, 2013.

[25] Roberto Ierusalimschy, Luiz Henrique De Figueiredo, and Waldemar Celes Filho. Lua - An Extensible Extension Language. *Software: Practice and Experience*, 26(6):635–652, 1996.

[26] Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, et al. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Trans. Embedded Comput. Syst.*, 13(4s):134:1–134:25, 2014.