Towards Composable GPU Programming:

Programming GPUs with Eager Actions and Lazy Views

Michael Haidl* Michel Steuwer† Hendrik Dirks* Tim Humernbrum* Sergei Gorlatch*

*University of Münster, Germany †University of Edinburgh, UK

{ michael.haidl, hendrik.dirks, t.hume, gorlatch }@wwu.de michel.steuwer@ed.ac.uk

Abstract

In this paper, we advocate a *composable* approach to programming systems with Graphics Processing Units (GPU): programs are developed as compositions of generic, reusable patterns. Current GPU programming approaches either rely on low-level, monolithic code without patterns (CUDA and OpenCL), which achieves high performance at the cost of cumbersome and error-prone programming, or they improve the programmability by using pattern-based abstractions (e.g., Thrust) but pay a performance penalty due to inefficient implementations of pattern composition.

We develop an API for GPUs based programming on C++ with STL-style patterns and its compiler-based implementation. Our API gives the application developers the native C++ means (views and actions) to specify precisely which pattern compositions should be automatically fused during code generation into a single efficient GPU kernel, thereby ensuring a high target performance. We implement our approach by extending the *range-v3* library which is currently being developed for the forthcoming C++ standards. The composable programming in our approach is done exclusively in the standard C++14, with STL algorithms used as patterns which we re-implemented in parallel for GPU. Our compiler implementation is based on the LLVM and Clang frameworks, and we use advanced multi-stage programming techniques for aggressive runtime optimizations.

We experimentally evaluate our approach using a set of benchmark applications and a real-world case study from the area of image processing. Our codes achieve performance competitive with CUDA monolithic implementations, and we outperform pattern-based codes written using Nvidia's Thrust.

1. Introduction

Graphics Processing Units (GPUs) are nowadays an inherent part of computing at small and large scale: from robotics applications accelerated with embedded GPUs up to large scientific simulations. Developing such applications is challenging as GPU code is often developed separately from the main CPU application, in a low-level programming language such as CUDA. Achieving high performance on GPUs is demanding even for experienced programmers; it is currently achieved by writing low-level, hand-optimized *kernel* functions. It would be highly desirable to use a single programming language and take advantage of convenient programming abstractions while achieving high performance.

The research community has recognized this challenge and moved towards unified CPU/GPU programming in C++ in the newest versions of CUDA and projects like C++ AMP [12] and SYCL [16]. At the same time higher-level abstractions have been proposed in form of libraries such as Thrust [2] and SkelCL [19] which provide reusable programming patterns, customizable by developers for particular applications. Unfortunately, these libraries introduce significant overhead when applications are composed of multiple patterns, each executed in a separate kernel, while in low-level code a single efficient kernel would be used. Therefore, programmers currently either benefit from the composable high-level patterns but pay a performance penalty for using these abstractions, or they achieve high efficiency for the cost of writing low-level and non-composable code.

In this paper, we present a *composable* GPU programming approach which enables developers to compose their applications from simple and reusable patterns. The main contribution of our approach is the implementation of pattern composition that guarantees that composed patterns are executed in a single efficient GPU kernel. Our implementation is a compiler-supported library that extends the `range-v3` library [13] which is currently developed as the next generation of the C++ Standard Template Library (STL). In our approach, programmers obtain full control to either a) use patterns expressed as STL *algorithms* and *actions* which are computed eagerly on the GPU or b) use *views* which our implementation guarantees to fuse during code generation and execute in a single GPU kernel.

Our LLVM-based, C++-to-GPU code generator uses *multi-staging* optimizations to embed values only known at CPU runtime into the kernel code which is then JIT compiled and executed on the GPU. This allows to specialize the kernel code for the particular input size and particular number of threads executing the kernel.

Our experimental evaluation demonstrates that composable code written using our approach achieves the same performance as low-level monolithic implementations in CUDA and outperforms composable code written using Thrust.

This paper makes the following main contributions:

- Enabling composable GPU programming by enriching the `range-v3` library with GPU-enabled algorithms;
- Giving programmers full control and better predictability about kernel fusion by choosing between eager actions and algorithms and lazy views;
- Implementing JIT code generation which takes advantage of *multi-staging* to optimize GPU code based on values known at runtime of the CPU program.

The remainder of the paper is organized as follows: In Section 2 we recap related work on GPU programming and we motivate the necessity for writing efficient composable GPU code. Section 3 introduces our API with ranges and GPU-enabled algorithms. Section 4 introduces actions and views and show how these give the programmer precise control over kernel fusion. Section 5 explains our C++-to-GPU code generation process and our enhanced compiler optimizations (in particular multi-staging). Section 6 evaluates our approach, and Section 7 concludes the paper.

2. Background and Related Work

GPU programming using the current low-level programming models such as OpenCL and CUDA is challenging. Special *kernel* functions are executed in parallel on the GPU by explicitly specifying the number of executing threads running in parallel. OpenCL and CUDA clearly separate the *host program* running on the CPU from the *GPU program* (kernel) which is written in a subset of the C/C++ programming language with GPU-specific extensions.

```
 1  __global__ void partialDotProduct(float* a,
 2                                    float* b,
 3                                    float* res){
 4   extern __shared__ float* tmp;
 5   int lid = threadIdx.x;
 6   int gid = lid + blockIdx.x * blockDim.x;
 7   tmp[lid] = a[gid] * b[gid];
 8   __syncthreads();
 9   for (int i = get_local_size(0)/2; i>0; i*=2) {
10    if (lid < i) tmp[lid] += tmp[lid + i];
11    __syncthreads(); }
12   if (lid == 0) res[blockIdx.x] = tmp[lid];}
```

Listing 1: GPU Kernel in CUDA computing a dot product.

Listing 1 shows a simple, unoptimized CUDA kernel, adopted from the Nvidia toolkit [14] that computes a dot product of vectors `a` and `b`. In line 7 each thread multiplies the corresponding elements of the two vectors with each other. A tree-based reduction is performed by a group of threads (called *block* in CUDA): in each iteration of the loop in line 9 the number of active threads which add up two elements (line 10) is halved. Finally, one thread of each block writes the computed result back into memory (line 12). The barriers in lines 8 and 11 ensure a consistent view of the memory across all threads. As synchronization across blocks is not possible, the summation of all results computed by the blocks cannot be performed in this kernel and must either be done on the host or in another CUDA kernel. The implementation in Listing 1 is not very efficient on modern GPUs and can be significantly optimized [9] which makes the code significantly more complex. But even the simple dot product code is not trivial: it works in a hierarchical CUDA address space (global and shared memory) and requires barrier synchronization, making it prone to subtle bugs like deadlocks and race conditions.

One popular way to overcome this low-level programming style is to provide reusable generic parallel patterns, also known as *algorithmic skeletons*. Thrust [2], Bolt [1], Accelerate [11], and SkelCL [19] follow this approach.

```
float dotProduct(const vector<float>& a,
                 const vector<float>& b) {
 thrust::device_vector<float> d_a = a;
 thrust::device_vector<float> d_b = b;
 return thrust::inner_product(
  d_a.begin(), d_a.end(), d_b.begin(), 0.0f); }
```

Listing 2: Optimal dot product implementation in Thrust using a domain-specific library function.

Listing 2 shows the implementation of the dot product in Thrust using a single library call. This implementation is straightforward and relies on an optimized GPU pattern written by experts and tuned for performance. Unfortunately, it uses a very domain-specific library function (`inner_product`) which limits its applicability for other applications.

```
float dotProduct(const vector<float>& a,
                 const vector<float>& b) {
 thrust::device_vector<float> d_a = a;
 thrust::device_vector<float> d_b = b;
 thrust::device_vector<float> tmp(a.size());
 thrust::transform(d_a.begin(), d_a.end(),
                   d_b.begin(), tmp.begin(),
                   thrust::multiplies<floal>());
 return thurst::reduce(tmp.begin(), tmp.end());}
```

Listing 3: Generic and composable, but non-optimal Thrust implemenation of dot product

In Listing 3, the dot product is implemented in Thrust using two smaller but more universal patterns: `transform` and `reduce`. However, here we pay a performance penalty for pattern composition: two kernels are launched and a temporary vector `tmp` is required to store the intermediate result of `transform`. This loss of efficiency is the reason why Thrust offers the specific `inner_product` function: high performance is not achievable by composing universal patterns.

In the next sections, we present our approach where higher-level programs can be expressed as a composition of universal patterns without a performance loss.

3. Programming with Ranges and GPU-Enabled Algorithms

Our suggested composable API for GPUs is inspired by the Standard Template Library (STL) which is widely used in C++ programming. The STL consists of three main components: 1) *containers*: collection data types such as `std::vector` or `std::set`, 2) *algorithms*: reusable operations on containers such as `std::accumulate` or `std::sort`, 3) *iterators*: glue containers with algorithms, such that the same algorithm is reusable for different containers. This library design has proven to be highly flexible and is one of the main reasons of the STL's success.

3.1 From Iterators to Ranges

The existing STL with iterators does not allow to compose algorithms easily. Listing 4 shows the dot product example using two STL algorithms with iterators: `transform` to apply the binary `mult` function to the corresponding elements of the input containers `a` and `b`, and `accumulate` to sum up all the values of a sequence. These two algorithms do not compose nicely, because `transform` returns only a single iterator, while `accumulate` expects a pair of iterators as its first two arguments describing the start and end of the input container. Furthermore, a temporary vector `tmp` is required, as the transform algorithm has to write the computed intermediate result into a container. The algorithm cannot allocate this temporary container itself, because the iterator abstraction hides the type of container (in this case a `vector`) from the algorithm.

```
float dotProduct(const vector<float>& a,
                 const vector<float>& b) {
 auto mult = [](auto x, auto y){return x * y;};

 vector<float> tmp(a.size());
 transform(a.begin(), a.end(), b.begin(),
           tmp.begin(), mult);
 return accumulate(tmp.begin(), tmp.end(),0.0f);
}
```

Listing 4: Dot product implementation using iterators.

Listing 5 shows how we can overcome these composability problems using the recent `range-v3` library [13] that replaces iterators with *ranges*. The `accumulate` call now takes its input as the single first argument compared to two separate iterators in Listing 4. Ranges are composable, because they carry information about the start and end points of a container, i.e.

```
1 float dotProduct(const vector<float>& a,
2                  const vector<float>& b) {
3  auto mult = [](auto p){
4     return get<0>(p) * get<1>(p); };
5
6  return
7   accumulate(
8    view::transform(view::zip(a,b),mult),0.0f); }
```

Listing 5: Dot product implementation using composable ranges.

ranges combine the information scattered across the first two arguments of `accumulate` into a single value. This change, together with the introduction of *views* — lazily computed ranges, which we will discuss in more detail in Section 4 — allows to write the dot product example in a concise and composable style in Listing 5. Here the `zip` function creates pairs of elements from vectors `a` and `b` which are then multiplied and summed up. Composability is a central feature of the `range-v3` library: it allows and encourages developers to use the pipe symbol `|` to denote composition (similar to the `bash` shell). Therefore, we can also rewrite lines 7–8 of Listing 5 as explicit composition of patterns:

```
view::zip(a,b) | view::transform(mult) | accumulate(0.0f)
```

The idea of our approach is to bring this range-based abstraction to GPU programming.

3.2 GPU-enabled containers and algorithms

We extend the `range-v3` library with a GPU-enabled container and GPU-enabled algorithms to allow programmers to write GPU applications in a composable way.

For storing data on the GPU we introduce the `gpu::vector` container which provides a range-based interface to access its elements. We use type traits to ensure statically that the GPU-enabled algorithms only operate on data stored in a `gpu::vector`. Data is transferred to the GPU by copying data into a `gpu::vector`, e.g., by using the `gpu::copy` function as in Listing 6, and transferred back by copying such a container into a regular STL `vector`.

Our algorithm implementations for the GPU using ranges currently covers the three central algorithms – `gpu::for_each`, `gpu::transform`, and `gpu::reduce` – and is currently being extended to cover all algorithms of the recently standardized parallel STL [10]. The `for_each` and `transform` algorithms apply a given function to every element of the input range in parallel. The `transform` writes the result of each function application into an output range (this algorithm is also known as *map* in functional programming). The `for_each` algorithm produces no result directly, but is executed for its side effects. This allows, for example, to write computed values to memory in a less structured fashion than using `transform`. The `reduce` algorithm performs a parallel reduction using a given binary operator which has to be associative. We will see that already using these three algorithms as patterns allows us to express many interesting applications, especially when combined with views (lazily evaluated ranges) as discussed in Section 4.

3.3 First GPU example

Listing 6 shows a code of the dot product using our API with the `gpu::reduce` algorithm. Notice how close the implementation is compared to Listing 5. In line 6 the input vectors `a` and `b` are copied to the GPU, and then their pairwise multiplication results (line 7) are summed up in line 8. Our API implementation (described in Section 5) guarantees to fuse the operations expressed as views into a single efficient GPU kernel. In the example, the zip and the vector multiplication in lines 6 and 7 are fused together with the reduction in line 8. We will discuss the implementation of views in the next section.

```
1  float dotProduct(const vector<float>& a,
2                   const vector<float>& b) {
3   auto mult = [](auto p){
4      return get<0>(p) * get<1>(p); };
5
6   return view::zip(gpu::copy(a), gpu::copy(b))
7        | view::transform(mult)
8        | gpu::reduce(0.0f);  }
```

Listing 6: GPU dot product using composable patterns.

3.4 Summary

In our API, ranges combined with GPU algorithms and GPU containers enable a natural way to program GPUs in C++ similar to the programming approach widely known from the STL. To achieve composability, programs are written by combining small and simple-to-understand patterns which greatly simplify programming as compared to traditional low-level programming approaches like CUDA. Next we discuss the key idea for achieving high performance in our composable approach: guaranteed kernel fusion using views.

4. Eager Actions and Lazy Views

The `range-v3` library introduces two new consructs to the STL — *actions* and *views* — which enhance the composability. We exploit views and actions for GPU programming: by using them programmers can control the fusion of computations expressed by patterns into a single GPU kernel.

4.1 Actions

Actions perform (potentially mutating) in-place operations on containers, i.e., actions do not require an externally provided output container, such as the temporary vector `tmp` in Listing 4. Actions are implemented with the STL algorithms and, therefore, we implemented corresponding actions for our GPU-enabled algorithms. Actions return a reference to the modified container and, therefore, compose nicely with other actions.

4.2 Views

Views are the counterpart to actions and describe non-mutating operations on ranges. Views, like actions, compose nicely and are designed to be used together with each other and with the algorithms described earlier.

An example for a view is `view::transform(mult)` in Listing 6. It applies a given function (in this case `mult`) to its input range. When executed on the CPU, this computation is not performed eagerly by writing the computed result to some (temporary) memory location, but rather an object (the *view*) is created which behaves like a range and performs the computation lazily, i.e., on-demand once an element in the range is requested. The view object holds references to its input range and the function to be called. When the view is iterated over, it evaluates the requested elements. Views are implemented as first class objects which can be stored in variables and passed to and returned from functions.

When views are composed with each other or with algorithms, they are evaluated only when the finally computed range is accessed. For example, in Listing 6 the `zip` and `transform` views are composed with the `gpu::reduce` algorithm. The implementation of `reduce` iterates over the input range to sum up all of its elements. Inside of this iteration, the pairwise multiplication expressed by the two views is performed, i.e., it is automatically fused by our API implementation into the implementation of the reduce algorithm. When we compile this code for the GPU, as described in Section 5 this API design ensures that only a single GPU kernel is emitted which performs the `zip` and `transform` computations inside of the iteration code of the `reduce` algorithm.

Together with our GPU-enabled algorithms, this guaranteed behavior of the views let programmers reason precisely about the cost of operations and the number of GPU kernels launched. Views allow to write composable and elegant code without paying a performance penalty, as we will see in our experimental evaluation.

4.3 Provided Views

The `range-v3` library offers currently over 40 views (e.g., `filter` or `generate`) which can be used together with our GPU algorithms. This greatly enhances the flexibility of our GPU programming approach. Interestingly, some views such as `repeat` represent infinite ranges; the `take` and `take_while` views can be used to limit such infinite ranges. In our approach kernel fusion is directly tied to the available views. The possibility to fuse two kernels corresponds to the ability to express a computation as a view which can be performed lazily and, therefore, folded into another computation.

The benefit of reusing the existing views from the `range-v3` library is ensured by our LLVM-based GPU code generator which compiles arbitrary C++ code for the GPU. There are only minor not supported exemptions, such as virtual functions or exceptions which are not used in the STL.

4.4 GPU evaluation of views

Some computations can be expressed in our API only using views and without ever using an action or algorithm. In such cases, the evaluation will happen implicitly when the final view is iterated over, e.g., when printing the result or copying it into a data container. By default, this evaluation happens sequentially on the CPU. To allow programmers to perform such computations on the GPU, we implemented a variation of the C++ standard `async` function. Our `gpu::async` function takes an arbitrary function whose return value is fixed to be a `gpu::vector` and, therefore, if in the function implementation a view is returned then it is implicitly evaluated on the GPU in parallel and the result is written into a `gpu::vector`.

```
1  auto saxpy(float A, const vector<float>& X,
2                      const vector<float>& Y) {
3   return gpu::async([=](auto a, const auto& x,
4                                const auto& y){
5    auto ax =  view::zip(view::repeat(a), x)
6            | view::transform(mult);
7    return  view::zip(ax,y)
8          | view::transform(plus);}, A, X, Y) ;}
```

Listing 7: *saxpy* on GPU using views and `gpu::async`.

Listing 7 shows an implementation of the *saxpy* computation which only uses views and no algorithms or actions. Our `gpu::async` function is used in line 3 to evaluate the views on the GPU. First, $a \times x$ is computed where a is a scalar and x is a vector, before the result is added to the vector y. We use four views (twice `zip` and twice `transform`) to describe the computation which is nested inside a `gpu::async`. All views are fused together during code generation and evaluated in a single GPU kernel which produces a `gpu::vector`. The `gpu::async` function ensures that all input containers are automatically copied to `gpu::vector` objects if they are not already of that type. This ensures that all data used in `gpu::async` is available on the GPU.

4.5 Summary

With the support of actions and views, our GPU extension of the `range-v3` library encourages a compositional programming style. We make use of this style for GPU programming by combining actions and views with our GPU-enabled algorithms and providing a natural way to evaluate lazy views in parallel on the GPU via `gpu::async`. The views provide a simple mechanism for the programmer to express programs compositionally while being guaranteed that the views are fused together into a single GPU kernel. We will now look at our GPU code generation and discuss optimizations applied when generating GPU kernel code.

5. Code Generation and Optimization

Our API with GPU-enabled algorithms and containers is implemented as a C++ library compiled using PACXX [8] — our LLVM-based compiler generating GPU code from C++ code. In this section, we briefly describe the general design of PACXX and then we focus on a particular feature, called multi-staging, which allows us to optimize the GPU code based on values only known at host runtime.

5.1 Overview of PACXX

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

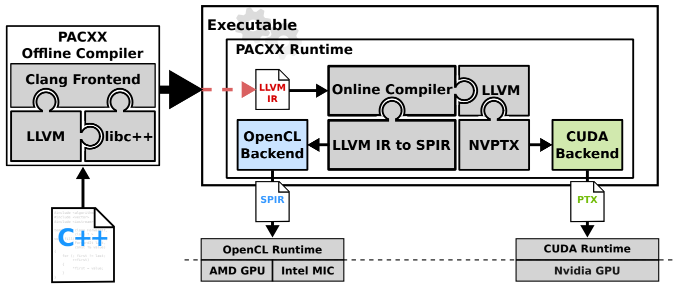


Figure 1: Key components of PACXX.

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

1) The *PACXX Offline Compiler* based on the open-source Clang compiler, and
2) The *PACXX Runtime* library consisting of a just-in-time compiler and a GPU execution runtime.

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

Stage 1: Offline Compilation In PACXX, code to be executed on a GPU is written as a `pacxx::kernel` function supported by PACXX. Based on this kernel function, the PACXX offline compiler marks the code executed on the GPU by annotating every function called from inside the kernel function with a PACXX-specific C++ 11 generalized attribute. Using generalized attributes of C++ has the advantage that the code remains valid C++ and other compilers have the freedom to ignore PACXX custom annotations.

After the annotations are added, two passes are performed: the first pass prepares the GPU kernel generation at runtime, and the second pass compiles the CPU program.

In the *kernel compilation* pass, the program's abstract syntax tree (AST) is lowered to the LLVM intermediate representation (IR), and functions with the PACXX-specific attribute are marked as kernel code in the IR. Then the following steps transform and optimize the IR:

1) dead code elimination removes all IR nodes beside the code reachable from the kernel;

2) function calls are inlined into the kernel functions;

3) `-O3` equivalent optimizations are performed;

4) the IR is embedded in the executable.

At runtime the prepared IR is loaded, then JIT-compiled and optimized for the target GPU before execution.

In the *host compilation* pass, the PACXX offline compiler lowers the AST to LLVM IR a second time, but this time replacing calls to kernel functions with PACXX runtime calls which manage data transfers and launch the corresponding kernel. The compilation of the host C++ program is performed as usual, by generating an executable which is statically linked against the PACXX runtime library.

Stage 2: Online Compilation During program execution, the PACXX runtime loads the IR previously embedded in the executable by the PACXX offline compiler, performs additional, GPU-specific optimizations, such as loop-unrolling and rearranging load instructions. Finally, the IR is compiled to GPU code using one of the two LLVM back-ends: PTX [15] together with the CUDA runtime library when targeting Nvidia GPUs, and SPIR [7] for GPUs with an OpenCL implementation (e.g., from AMD or Intel).

5.2 GPU Algorithm Implementations

We implemented three GPU-enabled STL algorithms to be used together with the `range-v3` library. We are discussing their efficient implementations for Nvidia GPUs here. This implementation is not portable, but we intend to incorporate recent related work in the future which uses a functional techniques to generate efficient GPU implementations from portable pattern-based representations [20].

Transform and For_Each The `gpu::transform` algorithm applies a given function in parallel to every element of its input range and stores the produced results in the output range. The `gpu::for_each` algorithm is a special case of transform, where its given function does not produce a result directly but is rather executed for its side effects. Therefore, we concentrate on the implementation of `gpu::transform`.

```
auto kernel = pacxx::kernel(
  [func](auto in, auto out, size_t size) {
   auto id = Thread::get().global;
   if (id.x >= size) return;
   *(out + id.x) = func(*(in + id.x));
 },{{(distance + 127) / 128}, {128}});

kernel(in.begin(), out.begin(), size);
```

Listing 8: Implementation of `gpu::transform`.

Listing 8 shows the implementation of `gpu::transform` for an Nvidia GPU. For each element of the input range, a GPU thread is launched which: a) loads one element form the global memory, b) applies the given function (`func`) to the element, and c) stores the result back to the global memory of the GPU. Our implementation configures the underlying kernel to use 128 threads per block. This is a platform-specific choice, and auto-tuning [4] or similar techniques can be used to pick appropriate values.

Reduce To efficiently implement the `gpu::reduce` algorithm, we make use of the *multi-stage programming* support in PACXX [8] for embedding values known at the runtime of the host program into the GPU code. This enables additional optimization opportunities like aggressive loop unrolling. PACXX provides a dedicated function (**`stage`**) which evaluates expressions prior to the GPU kernel execution and the computed values are then automatically embedded into the GPU program.

```
 1  template <class InRng, class T, class Fun>
 2  auto reduce(InRng&& in, T init, Fun&& fun) {
 3    // 1. preparation of kernel call
 4    ...
 5    // 2. create GPU kernel
 6    auto kernel = pacxx::kernel(
 7     [fun](auto&& in, auto&& out,
 8           int size, auto init) {
 9      // 2a. stage elements per thread
10      int ept = stage(size / glbSize);
11      // 2b. start reduction computation
12      auto sum = init;
13      for (int x = 0; x < ept; ++x) {
14        sum = fun(sum, *(in + gid));
15        gid += glbSize; }
16      // 2c. perform reduction in shared memory
17      ...
18      // 2d. write result back
19      if (lid = 0) *(out + bid) = shared[0];
20     }, glbSize, lclSize);
21    // 3. execute kernel
22    kernel(in, out, distance(in), init);
23    // 4. finish reduction on the CPU
24    return std::accumulate(out, init, fun); }
```

Listing 9: Implementation sketch of the `gpu::reduce` algorithm making use of multi-staging.

Listing 9 shows a sketch of the implementation of the `gpu::reduce` algorithm in PACXX. For brevity and clarity we concentrate on the parts relevant for the multi-staging optimization. After some preparations, a GPU kernel is created in line 6 using the PACXX-provided `kernel` function. The following lambda expression contains the code executed on the GPU which calls the reduction operation `fun` in line 14. In line 10 we make use of the **`stage`** function: it indicates that the expression passed as its argument is evaluated on the CPU *prior* to the kernel call and that the evaluated value is embedded into the GPU code as a compile-time constant. Here this value is the number of elements processed per thread which is used as the upper bound in the following loop in line 13. As a consequence of the staging, this loop can be completely unrolled, since the upper bound is now known at the GPU compilation time, which results in a significant performance gain as we will see in the evaluation section. The kernel continues with a reduction performed in shared memory, synchronized across all threads in the same block (indicated in line 17); finally, the first thread of each block writes the computed result back to global memory (line 19). This kernel is launched in line 22, and in the last line the results of all blocks are reduced to the final result on the CPU. Note that the user-provided function `fun` is used seamlessly on both GPU and CPU. This is possible due to the two-pass compilation approach performed by PACXX when generating code for CPU and GPU (Section 5.1).

5.3 Implementation of Multi-Staging in PACXX

This section describes how the **`stage`** function is implemented in PACXX. In the kernel compilation pass, the staged code, e.g., `size / glbSize` in Listing 9, is separated from the kernel code and used to generate a corresponding function which is embedded in the binary and will be evaluated at runtime on the host prior to the execution of the kernel. The **`stage`** function call is removed from the kernel and replaced by a dummy call instruction which acts as a placeholder and is replaced at runtime with the value obtained from evaluating the staged function on the host.

When a kernel is executed at runtime, three steps are performed:

1) the kernel's parameters and launch configuration are set;
2) the staged functions are just-in-time compiled, evaluated, and the kernel's IR is modified;
3) the kernel is just-in-time compiled and launched.

To evaluate the staged function in step 2), its IR is loaded from the executable and just-in-time compiled for the host architecture. The function is then evaluated and in the kernel's IR the placeholder call instructions are replaced with the evaluated constant value. After the staged values have been embedded into the kernel program, PACXX performs additional optimizations on the code such as aggressive unrolling to take advantage of the new knowledge introduced by the staged values. Finally, the kernel program is lowered to either PTX [15] or SPIR [7] code linked with the corresponding CUDA or OpenCL runtime and executed.

The compiled kernel code is cached by PACXX to minimize the just-in-time compilation overhead. All staged functions are evaluated again if the kernel is launched multiple times: PACXX checks if the staged values have changed and only performs a kernel recompilation if necessary.

6. Experimental Evaluation

In this section we evaluate our composable GPU programming approach. We are interested in the performance of the composable C++ code compared to low-level monolithic CUDA code and high-level Thrust code, as well as the performance impact of multi-staging and the overhead caused by our JIT compiler at runtime.

6.1 Experimental Setup

We evaluate our approach on a system equipped with one Nvidia K20c GPU and an Intel Xeon E5-1620v2 CPU. The operating system is Ubuntu 16.04.01 LTS, and CUDA version 7.5 is used with the corresponding driver (version 361).

All benchmarks are compiled by Nvidia's `nvcc` compiler with `-O3` and `-arch=sm_35`. In advance, a warm-up kernel is executed to bring the GPU into the P0 state (highest performance). The kernel execution timings are recorded using the Nvidia command line profiler (`nvprof`). We perform 1000 iterations of each algorithm and report the average runtimes. For benchmarks with multiple kernels (e.g., Thrust benchmarks where `reduce` is used), we sum up the runtimes of the individual kernels. Throughout the plots we show the speedup $S = \frac{T_{Thrust,CUDA}}{T_{PACXX}}$ over Thrust and CUDA.

For benchmarks which study the advantage of multi-staging in PACXX, we also compare to CUDA code which is JIT compiled and has been optimized by applying the same multi-stage optimization manually. We use the Nvidia Runtime Compilation library (`nvrtc`) for the JIT compilation which was introduced in CUDA 7.5. We also evaluate against the most recent CUDA beta version 8.0.27.1.

6.2 Performance of Composable GPU Programming

In this subsection we evaluate the performance achieved for code written with our composable API compared to code written in Thrust and CUDA. We use four small benchmarks: vector addition (*vadd*), *saxpy*, *dot product*, and a vector *sum*. We evaluate a larger application in Section 6.5.

Implementation of Benchmarks We showed the implementations of the *dot product* in Listing 6 and *saxpy* in Listing 7. Both benchmarks are written as a composition of views. In case of the *dot product* the views are fused with a final `gpu::reduce`. The *saxpy* code uses only views and our `gpu::async` function. The *vadd* benchmark uses `view::zip` together with `gpu::transform`. The *sum* benchmarks makes no use of views and only uses `gpu::reduce`.

The CUDA implementations of these benchmarks are low-level codes operating on raw pointers and exploiting the parallelism explicitly. The simple CUDA code of the *dot product* shown in Listing 1 highlights additional problems, such as thread synchronization. Our pattern-based API clearly avoids these pitfalls and raises the level of abstraction significantly. We used an optimized CUDA version of Listing 1 for the performance comparison.

The Thrust implementations of our benchmark applications also avoid the pitfalls of CUDA, but do not allow for a composable programming style. The *dot product* is implemented using the specialized `inner_product` function, as shown in Listing 2. The *sum* benchmark used the `thrust::reduce` function and the *vadd* and *saxpy* benchmarks are written using `thrust::transform`. Different to our API, a monolithic style is used in the Thrust implementations.

Performance Results We evaluate each benchmark with 11 input sizes of single precision floating point values. Figures 2 and 3 show the performance results compared to the optimized CUDA code. The results for the vector addition and saxpy benchmark (Figure 2) show that our API implemented using PACXX achieves equal performance to the low-level CUDA kernels. The `transform` implementation in Thrust uses 1024 threads per block and launches a fixed number of blocks. Using this strategy, each thread computes multiple elements and Thrust achieves significantly better performance for the 2^{17} input size of the saxpy benchmark.

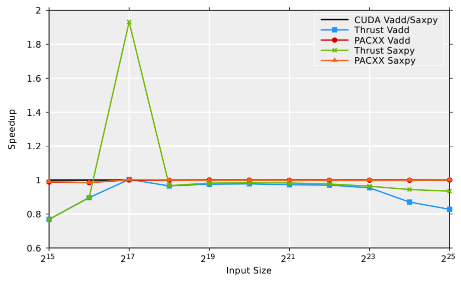


Figure 2: Speedup of our approach and Thrust compared to CUDA for the *vector addition* and *saxpy* benchmark.

Figure 3 shows the performance of our approach for *dot product* and *sum* in comparison to Thrust and CUDA. All three approaches follow different implementation strategies for these benchmarks. The CUDA version from Nvidia's SDK [14] uses a tree-based reduction in local memory and avoids synchronization inside a warp. The Thrust reduction algorithm computes the result in two GPU kernels while the PACXX and the CUDA versions execute only one GPU kernel and finish the computation on the CPU. Using two kernels with Thrust yields a significant overhead for smaller input sizes, which is clearly visible in the graph. The multi-staging optimized `gpu::reduce` implementation in PACXX clearly outperforms the Thrust reduction and even beats the low-level CUDA code for larger input sizes.

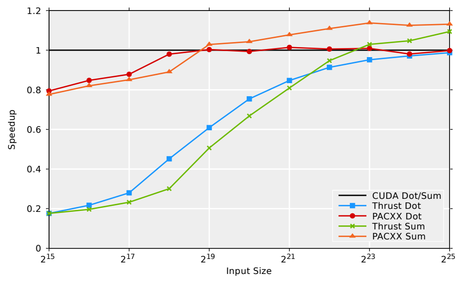


Figure 3: Speedup of our approach and Thrust compared to CUDA for the *dot product* and *sum* benchmarks.

Summary This section shows that our composable programming style using views does not introduce a performance overhead. In contrary, our optimized GPU-enabled algorithms outperform the corresponding low-level CUDA and Thrust code. Even the specialized `inner_product` implementation of Thrust used in the dot product benchmark is outperformed by our composable implementation where views are fused with our generic reduction algorithm.

6.3 Performance Impact of Multi-Staging

We continue our evaluation by studying the multi-stage optimization applied in PACXX. As described in Section 5, multi-staging is used in our `gpu::reduce` algorithm to specialize the generated GPU code at runtime for the particular input size, which enables aggressive loop-unrolling.

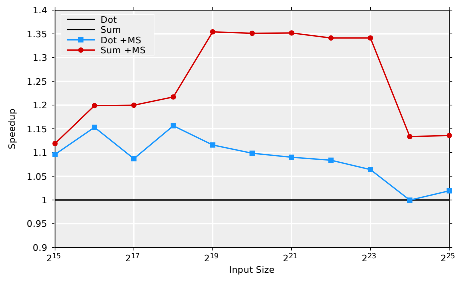


Figure 4: Performance improvement via multi-stage (MS) optimization for the *dot product* and *sum* benchmarks.

Figure 4 shows the relative performance improvement when applying multi-staging in the `gpu::reduce` algorithm, for the *dot product* benchmark shown in Listing 6 and for the *sum* benchmark that adds up all elements of an array. We observe that multi-staging improves performance by up to 1.35× depending on the input size. For input sizes larger than 2^{24}, the performance advantage declines, as the number of threads in a block is increased and, therefore, the number of loop iterations executed by each individual thread decreases: the effect of unrolling the loop becomes less visible. The improvements for the dot product are lower, because the kernel performs more loads from the global memory which dominate the kernel runtime, making the loop unrolling a less important performance factor.

6.4 Just-in-time Compilation Overhead

PACXX performs an offline and an online compilation step. In this section, we study the overhead introduced at runtime by the PACXX JIT compiler by comparing the compilation times of PACXX with the JIT compiler library `nvtrc` recently introduced for CUDA.

Figure 5 shows the compilation time for the sum and the dot product benchmarks. We compare PACXX with `nvrtc` from CUDA 7.5 and 8.0 RC. The compilation of PACXX is 15 times (for dot product) and almost 20 times (for sum) faster as compared to CUDA 7.5, and 9-12 times faster compared to CUDA 8.0 RC. This is because the PACXX offline compiler prepares the GPU code generation and, therefore, the PACXX JIT compiler operates directly on the LLVM IR.

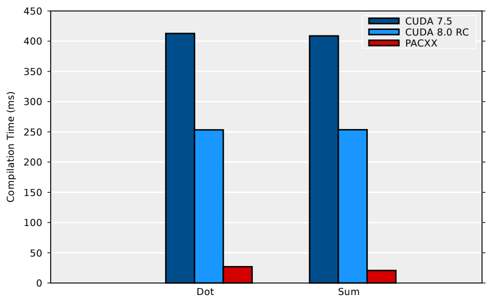


Figure 5: The just-in-time compilation overhead for the *dot product* and *sum* benchmarks.

The costly front-end operations — parsing, semantic checks and building of the abstract syntax tree — are not performed in PACXX at runtime as done in `nvrtc`.

6.5 Application Study: FlexBox

FlexBox [5] is a flexible MATLAB image processing toolbox for finite-dimensional convex variational problems which often consist of non-differentiable parts with linear operators. FlexBox uses GPUs to accelerate the solution of such image processing tasks. It offers a high-level notation familiar to applied mathematicians. This notation is mapped in FlexBox's implementation to computational patterns which are executed on the GPU.

FlexBox Background FlexBox takes advantage from the fact that many variational problems in image processing can be written in the form

$$\arg\min_x G(x) + F(Ax), \tag{1}$$

where A denotes a linear operator and both G and F are proper, convex and lower-semicontinuous functions. Problem (1) refers to the so-called primal formulation of the minimization problem and x is known as the primal variable. It can be shown (see [17]) that minimizing (1) is equivalent to solving the primal-dual formulation

$$\arg\min_x \arg\max_y G(x) + < y, Ax > -F^*(y). \tag{2}$$

FlexBox uses a primal-dual scheme [3] to avoid (computationally) inefficient operator inversion and to get reliable error estimates. The primal-dual scheme can be sketched up as follows: For $\tau, \sigma > 0$ and a pair $(\hat{x}^0, y^0) \in X \times Y$ we iteratively (k denotes the iteration) solve:

$$y^{k+1} = prox_{\sigma F^*}(y^k + \sigma A \hat{x}^k) \tag{3}$$
$$x^{k+1} = prox_{\tau G}(x^k - \tau A^T y^{k+1}) \tag{4}$$
$$\hat{x}^{k+1} = 2x^{k+1} - x^k \tag{5}$$

Here, $prox_{\tau G}$ (resp. σF^*) denotes the proximal operator:

$$prox_{\tau G}(y) := \arg\min_v \frac{||v - y||_2^2}{2} + \tau G(v) \tag{6}$$

which can be interpreted as a compromise between minimizing G and being close to the input argument y. The efficiency of primal-dual algorithms relies on the fact that the prox-problems are computationally efficient to solve.

In the primal-dual algorithm (5), the application of the linear operator A can be decoupled as follows:

$$\tilde{y} := y^k + \sigma A \tilde{x}^k, \text{ and } \tilde{x} := x^k - \tau A^T y^{k+1}. \tag{7}$$

As stopping criterion, FlexBox uses the primal-dual residual proposed in [6].

Image Denoising with FlexBox As an example we evaluate the Rudin-Osher-Fatemi (ROF) model [18] which has very popular applications in image denoising. An example image showing the effect of denoising is shown in Figure 6a. The primal formulation reads

$$\arg\min_u \frac{1}{2}||u - f||_2^2 + \alpha||\Delta u||_{1,2}, \tag{8}$$

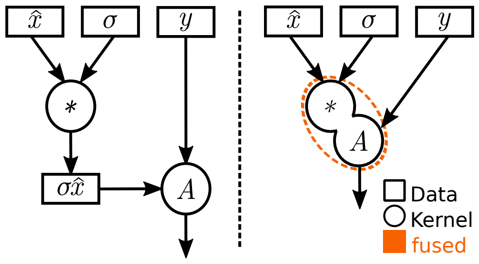
where the first part fits the unknown u to the input image f and the second part refers to the isotropic total variation, which penalizes the total sum of jumps in the solution.

While originally implemented in MATLAB, FlexBox offers an additional C++ backend to increase execution efficiency. FlexBox was recently extended with GPU capabilities based on Thrust. FlexBox is written in an object-oriented and modular way. Throughout the code, algorithms such as `transform` and `reduce` are used to implement the mathematical transformations which can be applied to an image in any order (required by equations like (8)). Adding GPU execution to FlexBox using Thrust is fairly easy: STL algorithms are replaced by the Thrust equivalents and Thrust's `thrust::device_vector` is used instead of `std::vector`.

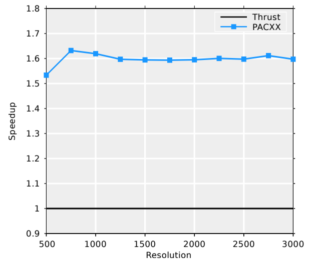
Profiling the Thrust version shows that 11 GPU kernels are executed in each iteration of our denoising application. A part of the data flow graph showing computations operating on data is presented in Figure 6b (left). Following the modular implementation of FlexBox, the two computations shown in the graph result in two kernel executions in Thrust with a temporary vector required that stores the result of multiplying each element of $\tilde{x}$ with σ.

We re-implemented FlexBox using our composable programming approach. Only slight modifications to the source code were required. Functions with Thrust algorithms now use corresponding views from our API which are automatically fused. Figure 6b (right) shows the data-flow graph with our API. The two computations are fused into a single efficient kernel, despite the fact that these two computations are described in separate header files. We were able to reduce the number of kernels executed in the denoising benchmark from 11 to 9 kernels. Furthermore, no more temporary vectors are needed which reduces the memory footprint on the GPU.

(a) Image used in the FlexBox evaluation: the source image (left) and the denoised image (right).



(b) Data-flow graph of implementations in Thrust (left) and our API (right) where two operations are fused into a single kernel.



(c) Performance improvements of PACXX: 1.6× compared to Thrust.

Figure 6

Comparing the performance of the image denoising application implemented with FlexBox, we see that the implementation using views in our API outperforms Thrust by 1.6×. This demonstrates the advantages of our approach which improves composability *and* improves performance due to the user-controlled fusion of views into efficient GPU kernels.

7. Conclusion

This paper presented a *composable* GPU programming approach using C++ and the STL. Generic, reusable patterns are composed to develop real-world GPU programs. This raises the abstraction level and avoids many issues of low-level coding in CUDA and OpenCL. In contrast to comparable approaches, such as Thrust, our implementation combines *eager actions and algorithms* with *lazy views* which are guaranteed to be fused into efficient GPU kernels. Our LLVM-based compiler uses multi-stage programming techniques to aggressively optimize GPU code at runtime with negligible runtime overhead. Our approach shows comparable performance to low-level CUDA code and significant performance improvements of up to 1.6× compared to Thrust code, as demonstrated by our real-world image denoising application.

Acknowledgments

We thank Nvidia Corp. for generously donating the hardware used in our experiments.

References

[1] AMD. Bolt C++ template library. 2014. Version 1.2.

[2] N. Bell and J. Hoberock. Thrust: A productivity-oriented library for CUDA. *GPU Computing Gems Jade Edition*, 2011.

[3] A. Chambolle and T. Pock. A first-order primal-dual algorithm for convex problems with applications to imaging. *J. of Mathematical Imaging and Vision*, 40(1):120–145, 2011.

[4] C. Cummins, P. Petoumenos, M. Steuwer, and H. Leather. Autotuning OpenCL workgroup size for stencil patterns. In *ADAPT@HiPEAC 2016*, 2016.

[5] H. Dirks. A flexible primal-dual toolbox. *arXiv preprint arXiv:1603.05835*, 2016.

[6] T. Goldstein, M. Li, X. Yuan, E. Esser, and R. Baraniuk. Adaptive primal-dual hybrid gradient methods for saddle-point problems. *arXiv preprint arXiv:1305.0546*, 2013.

[7] K. O. W. Group. The SPIR specification. 2014.

[8] M. Haidl, M. Steuwer, T. Humernbrum, and S. Gorlatch. Multi-stage programming for GPUs in C++ using PACXX. In *GPGPU@PPoPP*. ACM, 2016.

[9] M. Harris. Optimizing parallel reduction in CUDA. 2007.

[10] isocpp. *Technical Specification for C++ Extensions for Parallelism [N4578]*, 2015.

[11] T. L. McDonell, M. M. T. Chakravarty, G. Keller, and B. Lippmeier. Optimising purely functional GPU programs. In *ICFP*. ACM, 2013.

[12] Microsoft. C++ AMP: Language and programming model. 2012. Version 1.0.

[13] E. Niebler and C. Carter. N4569: C++ extensions for ranges. 2016. `https://github.com/ericniebler/range-v3`.

[14] Nvidia. The CUDA software developer kit. . Version 7.5.

[15] Nvidia. PTX: Parallel thread execution ISA. . Version 4.2.

[16] R. Reyes and V. Lomüller. SYCL: Single-source C++ accelerator programming. In *ParCo*, volume 27 of *Advances in Parallel Computing*. IOS Press, 2015.

[17] R. T. Rockafellar. *Convex analysis*. Princeton University Press, 2015.

[18] L. I. Rudin, S. Osher, and E. Fatemi. Nonlinear total variation based noise removal algorithms. *Physica D: Nonlinear Phenomena*, 60(1):259–268, 1992.

[19] M. Steuwer, P. Kegel, and S. Gorlatch. SkelCL — A portable skeleton library for high-level GPU programming. In *IPDPS Workshop Proceedings*, pages 1176–1182. IEEE, 2011.

[20] M. Steuwer, C. Fensch, S. Lindley, and C. Dubach. Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance OpenCL code. In *ICFP 2015*, pages 205–217. ACM, 2015.