YAGAL
Yet Another GPGPU Library

DPW106F18

Jonathan Hastrup

Morten Mandrup Hansen

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Selma Lagerlöfs Vej 300
9220 Aalborg Ø

Title:
YAGAL: Yet Another GPGPU Abstraction Library

Theme:
Programming Technology

Project period:
01/02/2018 -
08/06/2018

Project group:
dpw106f18

Members:
Jonathan Hastrup
Morten Mandrup Hansen

Supervisor:
Lone Leth Thomsen

No. of Pages: 93
No. of Appendix Pages: 4
Total no. of pages: 97
Completed: 08/06/2018

General purpose GPU (GPGPU) programming requires a developer to learn how to program in a new programming model to be effective. There are already related works addressing GPGPU, but they either utilize a custom compiler, and thereby enforce the compiler choice of the developer, or they utilize *OpenCL* as a target language.
This thesis documents the development of *YAGAL*, a GPGPU abstraction framework, that utilizes the *CUDA Driver API* for device management, and *LLVM* for *PTX* code generation at run-time, to be compiler independent without relying on *OpenCL*.
With the library, we explore the option of building kernels through an action abstraction that allows chaining of function invocations on a vector object to generate and execute a kernel on the GPU.
We compare the framework to the state of the art, in terms of both static measurements, and usability by using Cognitive Dimensions of Notations, to understand how *YAGAL* performs.
We reflect upon our thesis work in terms of technology choice, selected related works, framework implementation, and comparison.
We conclude that the development of a GPGPU framework is indeed possible with the configuration chosen, but there appear to exist other options that are equally performant and expressive with simpler development.

Preface

Reading Instructions

This thesis is meant to reflect and document the work of group dpw106f18 at their 10th semester at Aalborg University. This report have been written during spring 2018, and the material presented is based upon what was available at this time.

The reference material is available in the bibliography at the end of the report, and includes links to official documentation, websites, articles, scientific papers, and books.

The intended audience for this thesis is readers with some previous experience or understanding of both GPGPU development and *C++*.

A huge thanks to our supervisor, Lone Leth Thomsen from Aalborg University's department of computer science. Even tough we must have been a difficult group to work with, we appreciate all the help and support you provided throughout the thesis. Thank you!

This thesis is intended to be read in sequential order, as the chapters correspond to the tasks we performed in the order we worked on them. These chapters are;

Introduction
: This chapter contains the motivation and problem statement of the thesis, and in addition covers our development process and prerequisites for the thesis.

Related Works
: This chapter contains our investigation of related works that are similar to ours.

Design Principles
: This chapter contains the design guidelines that we followed to during development.

Framework Design
: This chapter contains the design of our framework, including our approach, the design of the API, and the design of the underlying architecture.

Framework Implementation
: This chapter contains documentation of our implementation and decisions taken during the implementation.

Problems
: This chapter contains the biggest issues that was encountered during development, and our proposals for how these issues can be solved.

Framework Demo and Comparison
: This chapter contains a presentation of our comparison methodology, how

we used it, and an evaluation of our work and of the related works.

Reflection
This chapter contains our reflections upon the thesis as a whole, which includes our choice of related works, our design, our choice of technologies, our implementation, and our comparison.

Conclusion
This chapter contains the conclusion of the thesis.

Future work
This chapter contains a description of how this thesis could move forward, if development were to continue.

Definitions

This is a list of terms heavily used in this report, with their definitions.

GPU
Graphics Processing Unit.

GPGPU development
General Purpose GPU development. To develop general purpose applications targetting GPU hardware.

STL
The C++ Standard Template Library.

Host
A machine executing a program, might delegate some tasks to a device. Covering both processing unit and memory.

Device
A device that can be given a computational task from a host. Usually a GPU, including processing units and memory.

BLAS
Stands for *Basic Linear Algebra Subprograms* and refers to functions that perform common linear algebra operations.

Developer
"A developer", "the developer", or "developers" in this report refers to one or multiple person that use *YAGAL* within their project.

Action
An abstraction of an intended action that should be performed on a collection, such as adding a value to all elements of a vector.

Contents

1. Introduction

In this chapter we present the motivation for the thesis, the problem statement that is derived from the motivation, the tasks and priorities we define based on our problem statement, the development process, and initial design decisions.

1.1 Motivation

It have been shown, that some heavy computational problems can be solved drastically faster using GPUs as accelerators, compared to traditional CPU execution. This have led to the development of multiple frameworks and tools to enable development on this platform.

In a previous project[1], we analyzed existing languages enabling GPGPU development. The goal of that project was to find representatives of different language groups, and compare the development experiences of using them. We concluded that GPGPU development generally is very difficult, as it requires the adaption of a new programming model, which gives a very steep learning curve for a developer. This leads to a lot of time spent learning the programming model, rather than solving problems. In some cases GPU acceleration is not the solution to performance issues, which means that a developer potentially wastes her time learning this programming model.

We found that frameworks for GPGPU development in compiled languages generally require a very specific compilation process, often by forcing the developer to use a custom compiler developed as a piece of the framework. This means that developers who already depend on some specific compiler for some feature might be unable to also incorporate the GPGPU framework.

In this project we want to create a framework, prioritizing high abstraction over absolute performance. This framework should allow developers to build accelerated applications without limiting their compiler choice. During the development of the framework we want to investigate the implications of not implementing the framework in the form of a compiler.

1.2 Problem Statement

Based on our motivation in Section 1.1, we describe our goal as a problem statement. Then we specify which tasks we need to perform to achieve it, and delimit our goal.

Can a GPGPU framework to abstract the underlying programming model be created, as a library that does not limit the developers choice of compiler, and how does it compare to other frameworks that do?

1.2.1 Tasks

We have defined the following tasks to reach the goal described in our problem statement:

Create overview of related works
: As others have made attempts at making GPGPU development simpler, we can learn from their design choices and results.

Research framework design principles
: Identify good practices for developing frameworks.

Design framework
: Design the architecture and API of the framework.

Implement framework
: Implement the design and discuss the challenges, and possible solutions, that arise from limitations.

Develop demo applications
: Showcase features what is possible in the framework, and to compare with implementations in related works.

Evaluation of our framework
: Review how our framework compares to the related works.

1.2.2 Delimitation

The problem can be approached in various ways, but the following are not the priority of the project:

Outperforming CUDA/OpenCL
: While we want the framework to perform well, we do not attempt to reach better performance than CUDA and OpenCL, as this would be unfeasible due to cost of abstractions.

Being "Complete"
: While we want to provide the functionality for a developer to develop GPGPU applications, we prioritize experimentation over being complete, as we want to experiment with which abstractions can work.

1.3 Development Process

In section 1.2.1, we define the tasks that need to be performed to reach the goals in our problem statement. These tasks have a linear dependency which motivates us to follow a waterfall inspired development process. This process is separated into two major parts; Design and implementation of our framework, and comparison between our framework and other frameworks. Before we start the comparison part, we will have completed the design and implementation part.

The implementation was done in incremental steps. Each step corresponds to a implementation of a feature that is a foundation for upcoming steps. The implementation is explained throughout chapter 5.

1.4 Thesis Prerequisites

In our previous thesis[1], we compared multiple languages with frameworks supporting development targeting the GPU. We observed how *CUDA* and *OpenCL* are the frameworks that offer the most explicit device control, and can be the choice for developers with knowledge of how to fine tune a GPGPU application. Based on this experience we make some decisions, that set the direction of the thesis.

1.4.1 Language Selection

Our goal of this thesis is the construction of a framework that can assist in providing applications with an initial benefit of GPGPU acceleration, where our framework is replaceable by another framework with a more low-level programming model when the developer is prepared for the steep learning curve of traditional GPGPU development. It is therefore convenient to keep our framework in the same language these low-level approaches. As *CUDA* and *OpenCL* are used from *C++* and *C* these are relevant. The abstractions allowed in *C++*, in the form of templates and lambdas, makes it more attractive for us, as we want to provide high-level abstractions. As such *C++* is our language of choice, and the framework we create in this thesis is based on *C++*.

1.4.2 Platform

Generating code to execute on a GPU can be done in various ways, either targeting a high-level languages like *CUDA*, or assembly like languages such as *PTX*. Targeting a high-level language gives the benefit of the compilation tool chain being able to perform optimizations, while targeting an assembly like language gives more explicit control.

LLVM[2] is a collection of compiler tools and technologies, which can be used to construct compilers and other tool chains for specific needs. One of the technologies of *LLVM* is the *LLVM Intermediate Representation* language, which can be constructed using *LLVM* tools. *LLVM Intermediate Representation* is a a platform agnostic language that can be compiled to platform specific code for any supported platform. A platform that is supported is *NVIDIA GPU*s, through the *NVPTX Back-End*, which allows us to compile *LLVM Intermediate Representation* to *PTX* code. Targeting *LLVM Intermediate Representation* allows us to generate code on a higher abstraction level than *PTX*, with optimizations provided by *LLVM*. Compared to a high-level language such as CUDA, we get more explicit control over the execution. We choose to target *LLVM Intermediate Representation*, as it seem to be of a convenient abstraction level between *CUDA* and *PTX*, and as it allows for expanding to other platforms than *NVIDIA GPU*s based on the available *LLVM Back-Ends*.

Using *LLVM Intermediate Representation* as the target platform, we can generate *PTX* code. To launch kernels written in *PTX* we need to use the *CUDA Driver API*. Memory management on the device also need to be done through this API, as it is done outside kernels. This decision requires YAGAL to be used on a system with a *NVIDIA GPU*, but can allow for easier transition for the developer into *CUDA* as *YAGAL* constructs use *CUDA* constructs.

2. Related Works

Before we design our framework, we consider other frameworks that attempt to reach similar goals. This allow us to make more informed decisions when we design our framework, as we then know how others have approached similar problems, and what their result was. We also learn how others have provided abstractions for the underlying architecture, giving us inspiration for our API design.

2.1 Selection of Related Works

To direct the content of this chapter we go through the frameworks we want to cover, and why we chose them. We then describe our examination of each framework.

The frameworks we examine in this chapter are:

Thrust
: Thrust is promoted by *NVIDIA*, who are the developers of *CUDA*, as a high level interface to GPU Programming[3].

C++ AMP
: C++ AMP is promoted by *Microsoft*, who is the developers of *DirectX*, as a *C++* language extension to enable data-parallel acceleration[4].

Bolt
: Bolt is developed by the *HSA Foundation* to provide a high level library to provide abstractions on top of low level programming models.[5].

SkelCL
: SkelCL is a research project that attempts to make GPU development easier with a concept called algorithmic skeletons.[6].

PACXX
: PACXX is a research project that attempts to make GPU development easier by combining host and device code in standard *C++*.[7].

We consider the frameworks in regard to:

Goals
: Understanding the goals of a framework help us identify the motivation behind its design choices.

Programming Model
: Understanding the programming model of a framework helps us form an overview of which approaches have been tried, and what is currently possible.

To give a demonstration of the programming model, we implement the Single-Precision A * X plus Y, *SAXPY*, computation. It is a calculation where the iterations of the main loop can be executed in any order. *SAXPY* is seen in listing 2.1.

```c
for (int i = 0; i < N; i++){
    result[i] = a * x[i] + y[i];
}
```

Listing 2.1: The *SAXPY* computation in C.

Implementation
Investigating the implementation helps us to understand how the programming model have been facilitated, and can give us insight of different approaches. While not all related works are open source, we are still able to get an architectural overview of their process.

Key Points
We put an emphasis on the most critical points of the frameworks, from our perspective, giving us some points to keep in mind when we design our own framework.

2.2 Thrust

Thrust is a *C++* library that allow developers to implement high performance applications with minimal programming effort. This section is based on *Thrust*'s overview document[8] and *Github* page[9].

2.2.1 Goals

The aim of *Thrust* is to make high performance application development as easy as possible. It is designed to be similar to *STL*, with the intention of being concise, readable, and efficient. It is intended to supply developers with containers and fundamental algorithms, with user defined behavior, rather than specific numeric algorithms as provided by *BLAS*. It is also intended to be interoperable with *CUDA*.

2.2.2 Programming Model

Thrust is modeled on *STL*, and follows the model of calling functions with iterators as arguments to instruct where input, and output is located.

Listing 7.5 shows how the *SAXPY* computation can be implemented in *Thrust*, and the usage of iterators to manage data access is shown.

The execution of *SAXPY* is done in line 17, and shows how iterators are used to define input and output locations.

```
 1  size_t N = 1 << 29;
 2  float a = 11;
 3
 4  //initialize host vectors
 5  thrust::host_vector<float> h_x(N);
 6  thrust::host_vector<float> h_y(N);
 7
 8  //fill with random data
 9  std::generate(h_x.begin(), h_x.end(), rand);
10  std::generate(h_y.begin(), h_y.end(), rand);
11
12  //copy to device
13  thrust::device_vector<float> d_x = h_x;
14  thrust::device_vector<float> d_y = h_y;
15
16  //perform saxpy
17  thrust::transform(d_x.begin(), d_x.end(), d_y.begin(), d_x.
        ↪ begin(), [=]__device__(float x, float y){return a * x + y
        ↪ ;});
18
19  //copy results back to host vector
20  h_x = d_x;
```

Listing 2.2: *Thrust SAXPY* example.

2.2.3 Implementation

Thrust is a designed to be compiled by the *NVIDIA* compiler, *nvcc*, as seen on figure 2.1.

The memory management is performed through types defined in the library, abstracting direct allocation management, and functions can be defined as lambdas with the annotation *__device__* which is the *CUDA Runtime* annotation for kernels. With *nvcc* as the compiler, *Thrust* can achieve the high-level design shown in listing 7.5, where the *Thrust* data types interface with *STL* algorithms and use an extended *C++11* syntax for lambdas.

The compilation process is static, and results in a single executable with both host and *CUDA Runtime* code included.

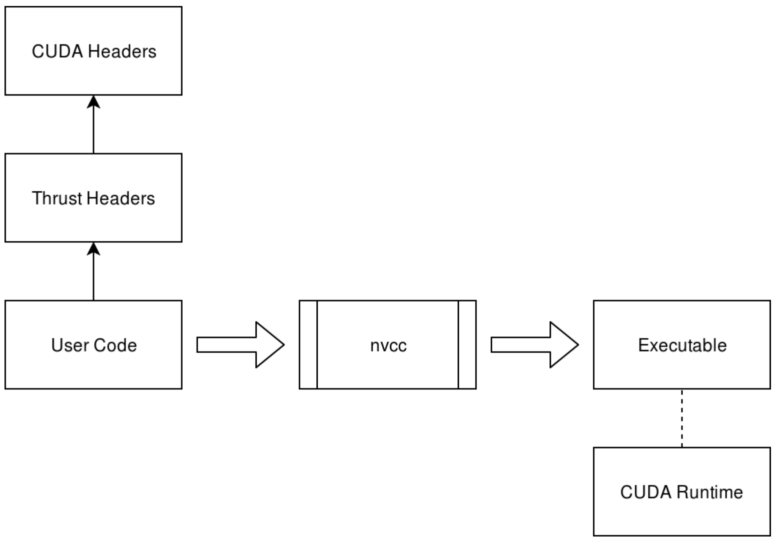

Figure 2.1: Thrust Compilation Process.

2.2.4 Key Points

The API of *Thrust* is structured to imitate that of *STL*, and a *C++* developer is therefore familiar with it. It can be very verbose with multiple operations on the same container since the usage of the containers is based on iterators.

Thrust is a header only implementation, but is dependent upon the *CUDA Runtime*, and requires the *NVIDIA* compiler *nvcc* to be used.

2.3 C++ AMP

C++ AMP stands for Accelerated Massive Parallelism, and is a framework that allows a developer to write code to be executed on data-parallel hardware, and is built upon *Direct3D*. *C++ AMP* is developed by *Microsoft* as a framework and as an open standard for implementing parallelism in *C++*. Their choice of *Direct3D* is interesting since *OpenCl* and *CUDA* existed at the time. The information discussed in this section was gained from *Microsoft*'s *C++ AMP* page [4].

2.3.1 Goals

The goal of the *C++ AMP* specification is to provide a way of writing code for data parallel hardware directly within the *C++* language. *Microsoft* implemented the specification based upon *Direct3D*, and the *HSA Foundation* later did it for

OpenCL.

2.3.2 Programming model

A feature of *C++ AMP* is that kernel functions is here expressed in *C++* as restricted lambdas, meaning that a subset of *C++* is available.

Construction of matrices is done by first creating an array, and then wrap it with the `array_view` that is provided by *C++ AMP*. To show an example, an array is constructed below:

```
float matrix[] = {1, 2, 3, 4};
```

To construct a matrix with two dimensions, the `matrix` array is wrapped with `array_view`:

```
array_view<float, 2> mat(2, 2, matrix);
```

Here the `<float, 2>` specifies that the `mat` matrix consist of floating point values and two dimensions. `(2, 2, matrix)` indicates that the `mat` matrix will have two rows and two columns, and will be populated with the data from the `matrix` array;

Listing 2.3 shows *SAXPY* implemented in *C++ AMP*. The `array_views` are constructed at line 11 to 13. It is still needed to specify the views, even though this example only utilize one dimension. The `z_v` `array_view` is at line 14 marked with the `discard_data()` function. This is done to indicate that `z_v` is used purely as an output container, and to avoid wasting resources transferring it to device since the contents will be overwritten. At line 16 the function `parallel_for_each()` method is called and given two arguments. `z_v.extend` indicates the compute domain. The lambda at line 19 are marked with `restrict(amp)` which states that the lambda should be executed on device and that only a subset *C++* functionality is available for execution.

```
1  const size_t N = 1024;
2  float a = 10;
3
4  std::array<float, N> x;
5  std::array<float, N> y;
6  std::array<float, N> z;
7
8  std::generate(x.begin(), x.end(), rand);
9  std::generate(y.begin(), y.end(), rand);
10
11 array_view<const float, 1> x_v(size, x);
12 array_view<const float, 1> y_v(size, y);
13 array_view<float, 1> z_v(size, z);
14 z_v.discard_data();
15
```

```
parallel_for_each(
    z_v.extent,

    [=](index<1> idx) restrict(amp){
        z_v[idx] = a * x_v[idx] + y_v[idx];
    }
)
```

Listing 2.3: *SAXPY* computation in *C++ AMP*.

2.3.3 Implementation

C++ AMP is a framework that enables simple manipulation of large dimensional arrays by introducing a new language feature called `restricted` for *C++*.

Data is managed within regular `std::arrays`. `array_views` can be constructed to represent these arrays as matrices on the GPU and is used to manipulate them.

Lambdas are used to construct the logic of a kernel for execution on the GPU. The *C++ AMP* keyword `restricted` is required on the lambda to indicate that the lambda is intended for execution on the GPU, and has limited functionality.

To execute a kernel, the kernel defining lambda is given to the `parallel_for_-each` function which also manages copies to and from the device of the data described by the `array_views`.

The compilation chain of *C++* AMP can be seen in Figure 2.2. Here the *C++ AMP* headers are included within the user code, and is compiled by the *Microsoft Visual C++* compiler (*MSVC*). The result is an executable file that utilizes *Direct3D* to execute on the GPU. In addition to providing a library that must be included in the code, *C++ AMP* adds an additional keyword to the language and therefore requires a compiler that implements the specification, such as the *MSVC* compiler for *windows*. The *C++ AMP* specification is however open such that other compiler vendors could support it in the future.

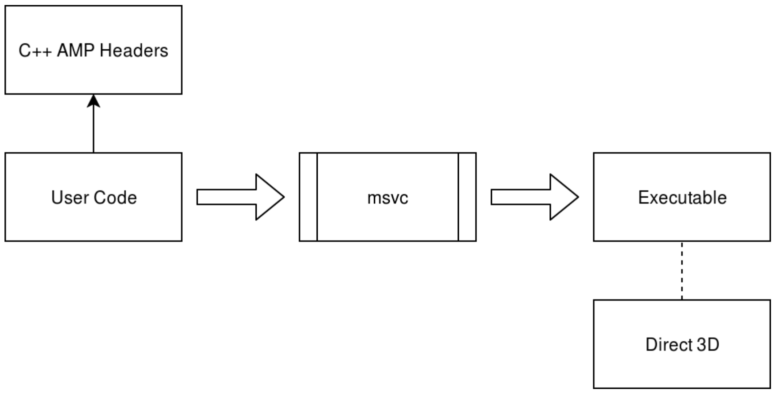

Figure 2.2: C++ AMP Compilation Process.

2.3.4 Key Points

A unique feature of *C++ AMP* is that it outputs to *Direct3D*. We assume that this decision might have been made due to *Direct3D* being developed and maintained by *Microsoft* as well.

C++ AMP is meant to extend *C++* with parallelism, but using arrays on the GPU requires that they are wrapped within an *array_view*, and lambdas are restricted.

C++ AMP is dependent upon the *MSVC* compiler.

2.4 Bolt

Bolt is a library providing abstractions for heterogeneous computing. This section is based on *Bolt*'s documentation[5] and *Github* page[10].

2.4.1 Goals

The goal of *Bolt* is to provide high performance library, that contain implementations of common algorithms, following the structure of *STL*. It is intended to make heterogeneous development easier, and is designed to provide an application that can execute on either a CPU or any OpenCL capable unit.

2.4.2 Programming Model

Bolt is modeled on *STL*, and follows the model of calling functions with iterators as arguments to instruct where input and output is located.

Bolt provide functions for modifying *STL* containers, and the library determines whether the computation should happen on host or device.

The example shown in Listing 2.4, shows how the library interfaces with an *STL* vector.

From line 11 until the next comment it is shown how the function is defined and implemented with the *C++ AMP* back-end. It is done with a *C++11* lambda and *C++ AMP*'s restrict classifier.

From line 17 it is shown how, instead of a lambda, a functor is needed when using the *OpenCL* back-end. The functor is then defined inside a BOLT_FUNCTOR macro to statically generate relevant *OpenCL* code.

```
 1  const size_t N = 1024;
 2  float a = 10;
 3
 4  std::vector<float> x(N);
 5  std::vector<float> y(N);
 6  std::vector<float> z(N);
 7
 8  std::generate(x.begin(), x.end(), rand);
 9  std::generate(y.begin(), y.end(), rand);
10
11  //bolt with c++ amp back-end
12  auto saxpyLambda = [=] (float xx, float yy) restrict(cpu,amp) {
13    return a * xx + yy;
14  };
15  bolt::transform(x.begin(), x.end(), y.begin(), z.begin,
        ↪ saxpyLambda);
16
17  //bolt with opencl back-end
18  BOLT_FUNCTOR(SaxpyFunctor,
19    struct SaxpyFunctor{
20      float _a;
21      SaxpyFunctor(float a): _a(a) {};
22      float operator() (const float& xx, const float& yy){
23        return _a * xx + yy;
24      };
25    };
26  );
27  boltcl::transform(x.begin(), x.end(), y.begin(), z.begin,
        ↪ SaxpyFunctor(a));
```

Listing 2.4: *SAXPY* computation in *Bolt*.

2.4.3 Implementation

Bolt is an abstraction library on top of either *C++ AMP* or *OpenCL*.

The developers of *Bolt* has made an effort in making it compatible with *STL* types and algorithms. To this end, data can be implicitly managed with *STL* vectors, where data needed for a kernel will be copied to the device before execution, and back to the host after. *Bolt* also provides device specific types for explicit memory management such as `device_vector` which allocates data directly in device memory.

Lambda expressions are used to express kernel functions when *C++ AMP* is the targeted back-end. The procedure is similar to how it is done in *C++ AMP* which is described in Section 2.3, but *Bolt* will handle the interaction with data and the `array_views` of *C++ AMP* is therefore not needed. The `restricted` keyword is still used to tell the compiler that only a subset of *C++* functionality is available within the lambda scope.

Functors are used to express kernel functions when using *OpenCl* as the targeted back-end. A functor allows the construction of a struct that can be called as a regular function, by overloading the parenthesis operator.

One of the two supported back-ends is *C++ AMP*, and the compilation chain can be seen on Figure 2.4. Here the user code has included the *Bolt* headers which in turn make use of the *C++ AMP* headers. *MVSC* is used for compilation since it is a compiler that supports *C++ AMP*. This results in an executable file that utilizes *Direct3D* for execution on the GPU.

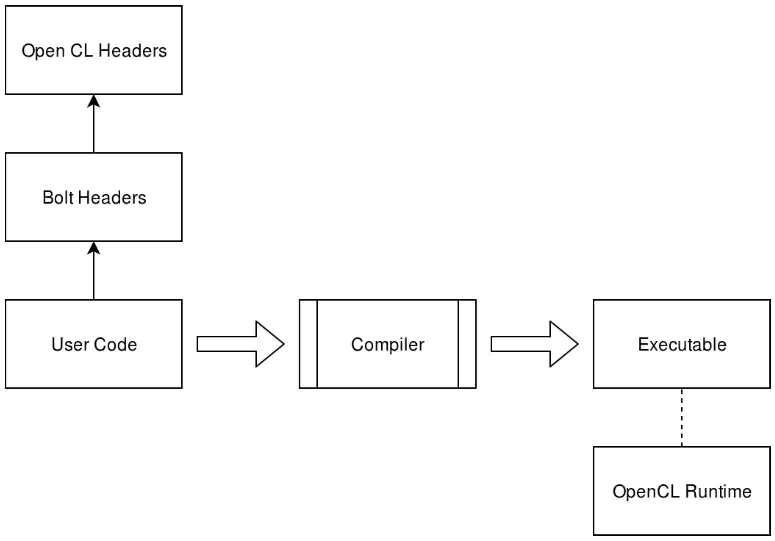

Figure 2.3: *Bolt*, targeting *OpenCL*, compilation process.

The other of the two supported back-ends is *OpenCL*, and the compilation chain can be seen on Figure 2.3. It is the same procedure as with *C++ AMP*, except that *Bolt* makes use of *OpenCL* headers, it is not dependent upon the *MVSC* compiler, and it utilizes the *OpenCL* run-time.

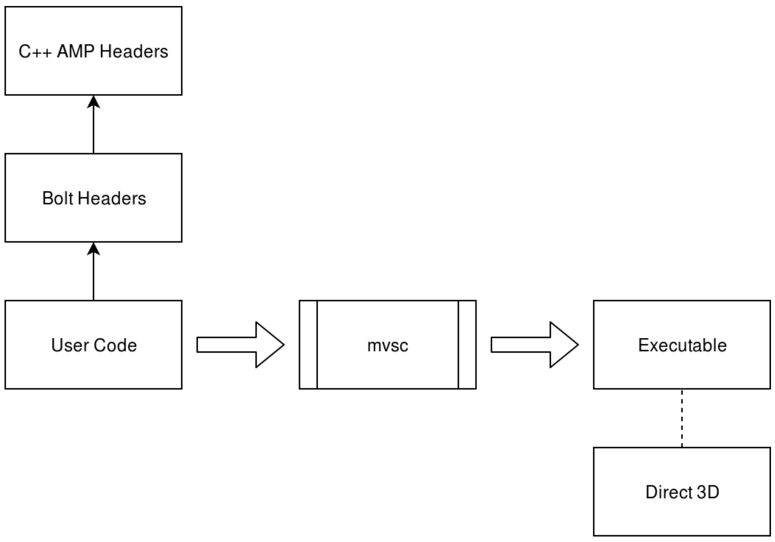

Figure 2.4: *Bolt*, targeting *C++ AMP*, compilation process.

2.4.4 Key Points

STL containers are more seamlessly integrated than they are in *C++ AMP* since no `array_views` are needed. *Bolt* manages allocations on a device, and provides `device_array` for cases where data should be manually managed.

As a library on top of other frameworks it shows some code artifacts of the underlying framework. With *C++ AMP* as target, the use of the restrict classifier on lambdas is necessary. `BOLT_FUNCTOR` is a macro is used to overcome the language gap between *C++* and *OpenCL C*, when *OpenCL* is set as targeted back-end. Both examples show that workarounds to support the target back-end sometimes will show up in the API.

2.5 SkelCL

SkelCL (Skeleton Computing Language) is a library aiming to provide abstractions for parallel programming on multi GPU systems. It is developed as a research project by Michel Steuwer et.al at University of Münster, Germany. This section is written based upon the information available on their website [11] and in their paper [6].

2.5.1 Goals

The developers of *SkelCL* state that programming for GPUs result in complex, lengthy and error prone programs. This is due to the process of writing GPU code typically being reliant on low-level programming approaches as seen with *OpenCL* and *CUDA*.

To avoid the pitfalls of the traditional low-level approaches, the library *SkelCL* provides abstractions in the form of algorithmic patterns, parallel container data types, and management of transfers between host and device.

SkelCl can be used on single GPU systems, but is developed for systems with multiple GPUs, and introduces the feature called *data (re)distributions* which manages data among the available GPUs.

2.5.2 Programming Model

The programming model is centered around *parallel skeletons*, which is pre-implemented high-level patterns that can be customized for a given problem. The available skeletons are *map*, *zip*, *reduce*, *scan*, *mapOverlap*, end *allpairs*.

An implementation of the *SAXPY* computation in *SkelCL* is shown in listing 2.5. After *SkelCL* is initialized, which happens at line 4, skeletons can be constructed. The `Zip` skeleton is specified with the parameters `<float(float,float)>`, indicating that the resulting `Zip` function expects two floats, and that a single float will be returned. The given string specifies the function of the skeleton. At line 13 the calculation is performed based on the constructed skeletons.

```
 1  size_t N = 1024;
 2  float a = 10;
 3
 4  skelcl::init();
 5
 6  Zip<float(float,float)> saxpy("float func(float x, float y,
      ↪ float a){return a * x + y;}");
 7
 8  skelcl::Vector<float> X(N);
 9  skelcl::Vector<float> Y(N);
10  skelcl::init(X.begin(), X.end());
11  skelcl::init(Y.begin(), Y.end());
12
13  saxpy(out(Y), X, Y, a);
```

Listing 2.5: The *SAXPY* computation in *SkelCL*.

2.5.3 Implementation

SkelCL is a library built upon *OpenCL*. This allows host and kernel code to be contained within one source file, as opposed to the traditional *OpenCL* approach. The implementation is done entirely in a library, which allows the developer to use it without enforcing a compiler choice.

Memory management is handled through either the `Vector` or the `Matrix` template classes, where the allocations are managed on the device.

Kernel functionality is provided through *C++* constructs that generate *OpenCL* code, which is then given to the *OpenCL* run-time. The generated code is constructed as a *OpenCL* kernel string, by providing different hard coded content based on the used skeleton, concatenated with the user provided string, such as the one in listing 2.5 line 6. It provides the features of *OpenCL* with minimum analysis of user code.

The compilation chain is shown in figure 2.5, where the highlight is the library layers with *SkelCL* in front of *OpenCL*, and the use of a non-specified compiler, as the implementation is a library.

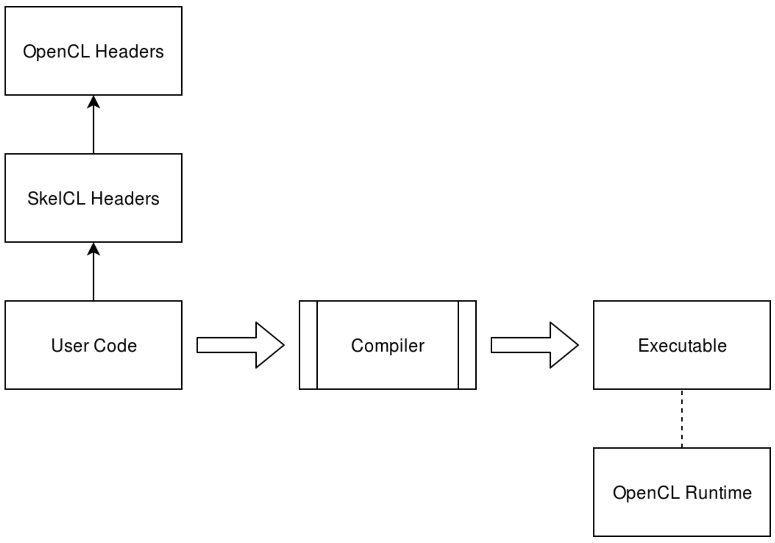

Figure 2.5: SkelCL compilation process.

2.5.4 Key points

A key point of *SkelCL* is the data containers it provides, namely vectors and matrices. They are available on both host and device. When one of these data containers is allocated or deallocated on the host, it is automatically also allocated or deallocated on the device(s). Furthermore, memory transfers between host and device are managed implicitly.

Another key point of *SkelCL* is how it is designed to function on systems with multiple GPUs. The *data (re)distribution mechanism* describes how a container is distributed among the available GPUs. This feature abstracts the need to manage which parts of the container gets assigned to which GPU. The data containers can be considered as self contained entities. A developer must specify a model for how the data should be distributed, with the available options being *single*, *copy*, *block*, and *overlap*.

A last key point is that *SkelCL* generate *OpenCL* code based upon skeletons which reduces the amount of needed analysis of user code. This results in an *OpenCL* string that can be delivered to the *OpenCL* runtime for execution.

2.6 PACXX

PACXX is a unified programming model that uses a custom compiler based on *Clang* and *LLVM*. It is a research project created by Michael Haidl and Sergei Gorlatch, both from University of Münster, Germany. The information in this section is based upon their *PACXX* paper released in 2014[7]. *PACXX* is not officially released yet and there are no given release date, but the compiler can be found on *Github*[12].

2.6.1 Goals

The *PACXX* paper states that *OpenCL* and *CUDA* are error-prone since with these approaches, host code is written in *C*/*C++* with a restricted, *C*-like API to handle memory management, and device specific code is written with a parallel programming model. The aim of *PACXX* is to avoid the traditional pitfalls of GPU programming by unifying host and device code and thereby allowing a developer to utilize *C++14* and *STL* features.

2.6.2 Programming Model

As the goal of *PACXX* suggests, the programming model is similar to a regular *C++* approach, such that the developer will not have to change mindset when programming. There are some exceptions; The developer still needs to evaluate the threads and blocks she want to use. The developer must use the `kernel` class that *PACXX* provides to construct a kernel function. Lastly, *PACXX* generates and compiles device code at run-time, and there are no restrictions as to what a kernel function can call, but all code used in combination with a kernel must be known at run-time. Another restriction is that functions from pre-compiled libraries cannot be used by a kernel function.

Listing 2.6 shows a *SAXPY* implementation using *PACXX*. A lambda function called `saxpy` is created on line 6, which describes *SAXPY*. The thread id will be fetched, as seen on line 7, and then the elements corresponding to that thread of each vector will be used for the *SAXPY* computation. The amount of threads and blocks are determined at line 12 and 13. Then, at line 15, the kernel function is constructed using the *PACXX* provided `kernel` class. The *SAXPY* computation is executed at line 16.

```
 1  main() {
 2    size_t = 1 << 24;
 3    float a = 2;
 4    std::vector<float> x(n), y(n), z(n);
 5
 6    auto saxpy = [](const float& a, const vector<float>& x, const
        ↪ vecotr<int>& y, vector<float>& z) {
 7      auto i = Thread::get().global.x;
 8      if (i >= x.size()) return;
 9      z[i] = x[i] * a + y[i];
10    };
11
12    size_t threads = 128;
13    size_t blocks = (n + (threads * 2 - 1)) / (threads * 2);
14
15    auto saxpy_gpu = kernel(saxpy, {{blocks}, {threads}});
16    saxpy_gpu(a, x, y, z);
17  }
```

Listing 2.6: *SAXPY* computation in *PACXX*.

2.6.3 Implementation

PACXX is a framework centered around a compiler created with *LLVM*, named *pacxx++*. Using a special purpose compiler allows for quality of life features, such as defining kernel functionality in *C++11* lambdas, at the cost of forcing that compiler to be used by the developer. The compilation is multi-staged, meaning that the main program is statically compiled, and the kernels are compiled before use, allowing kernels to vary depending on run-time state.

An important feature of *PACXX* is that the code written by the developer is standard C++, and can be compiled and executed on many platforms. The *pacxx++* compiler is able to identify lambdas that can be compiled for a GPU, and create a unified programming experience.

On figure 2.6 it is shown how a program compiled with *pacxx++* includes the *PACXX Run-time*. This run-time is able to generate *LLVM Intermediate Repre-*

sentation code and translate that to either *PTX* code for the *CUDA Run-time* or *SPIR* code for the *OpenCL Run-time*, using the *LLVM* support for those targets.

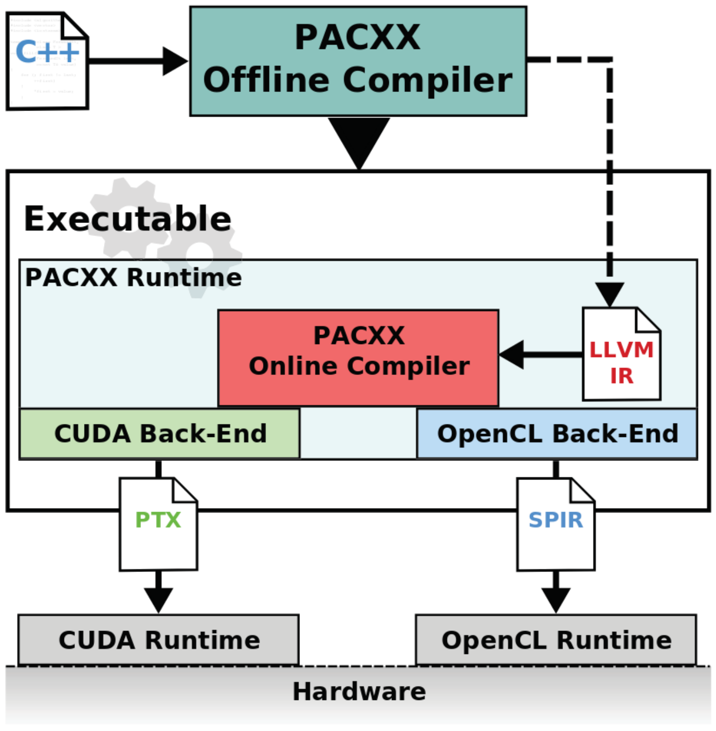

Figure 2.6: PACXX Compilation Process[13].

2.6.4 Key Points

PACXX allows a developer to write device specific code with *C++14* lambdas with few restrictions, and must still take threads and blocks into account.

PACXX uses *LLVM* to generate *PTX* and *SPIR* code at run-time. This is interesting as it provides more opportunities and freedom for abstractions than a static header library would. *PACXX* is also multi-staged such that run-time compiled kernels can vary depending on state.

2.7 Summary

Through considering the frameworks in this chapter, we gained some insights which allow us to make better decisions in the design of *YAGAL*.

2.7.1 Compilation

The compilation processes are diverse. The following variants have been shown in this chapter:

- *Thrust* is a library extension over *CUDA*, and inherits the compilation process of *CUDA*.
- *C++ AMP* is a standard that requires the compiler to implement it.
- *Bolt* is a library extension over either *OpenCL* or *C++ AMP*, and the compilation process is either using any compiler and linking to the *OpenCL* library, or using a compiler that implements the *C++ AMP* standard.
- *SkelCL* is a library that wraps the construction of *OpenCL kernel string*s within objects, and the compilation can be done using any compiler and by linking to the *OpenCL* library.
- *PACXX* is a framework and a compiler where developers can define kernel functions with limited lambdas and regular *C++*. The kernels are then generated during run-time.

Only *SkelCL*, and *Bolt* when targeting *OpenCL*, allow any compiler to be used. The common part is that both generate *OpenCL* kernels as strings, ready for the run-time.

PACXX is interesting as it implements a compiler that takes normal *C++* code and compiles some of it to GPU kernels using *LLVM* and its intermediate representation. The use of *LLVM* is enabling support for both *NVIDIA*'s *PTX* and *Khronos Group*'s *SPIR* platform. It has the drawback of requiring a specific compiler, which might be troublesome for some developers to integrate.

2.7.2 Data Storage

The general approach observed for managing allocations is by providing container classes such as `thrust::host_vector` to wrap that functionality. This seems like a reasonable choice, as it helps the developer perform allocations, free memory, and transfer data between host and device, through constructors and destructors.

Accessing data from these containers is generally performed through iterators or direct access with syntax as an array access. This seems like a convenient choice, but it might invite the developer to perform logic on the CPU by copying data back and forth, which seem counter intuitive as many small copies are slower than a few large copies, due to the memory transfer overhead[14].

2.7.3 Lambdas

The general approach observed for modifying data is to provide some high-level functions such as `boltcl::transform` that take some representation of a function to perform on each element. This function is represented in various ways, either through functors, strings, or *C++ 11* lambdas with additional syntax. Lambda handling is interesting since the related works each have a unique way of handling them. A distinction can be made between those that require specific compilers, and those that does not. Those that require specific compilers are able to freely add syntax to define new kinds of lambdas. Those that do not require a specific compiler require the developers to provide the logic in such a way that the library can extract the intent.

In general, supporting lambdas seems to be a problematic but useful feature, that appear most elegant when done by a framework that provides a compiler with support for syntax expansion.

3. Design Principles

As we are inexperienced in writing and documenting *C++* libraries and APIs, we need to identify some guidelines to consult during our development process. The guidelines are presented in this section along with the knowledge we gain as a result. This knowledge is utilized for designing the framework in chapter 4.

We consider *The Little Manual of API Design*[15], as it is written by *Jasmin Blancehette*, who have experience from *The Qt Company* in creating *C++* APIs. We also consider the *Standard Library Guidelines*[16] from *isocpp.org*, as guidelines for the standard library of *C++* can be relevant to us as well.

3.1 The Little Manual of API Design

The Little Manual of API Design is written by *Jasmin Blancehette*, and contains key insights into API design that were discovered during development of the *Qt application development framework* by *Trolltech*, which later became *The Qt Company*. It provides some core API design principles which can assist us in our framework design phase. The principles cover good characteristics of APIs, the design process, to guidelines. We focus mostly on the characteristics since the design process is focused on collecting information and extending existing APIs.

The Little Manual of API Design describes good characteristics of an API as follows:

Easy to learn and memorize
The API should use meaningful naming conventions that are consistent throughout the entirety of the library. The API should be minimal such that it is easy to memorize, and consistent, such that the developer can reapply the knowledge gained in one area of the API in another.

Leads to readable code
The API should lead the developer towards readable code. The API should not force a developer to write excess of boilerplate code nor specify irrelevant information.

Hard to misuse
The API should be designed in a way that it minimizes the risk of using it wrong. This includes not forcing the developer to call methods in a strict order and avoid implicit side effects.

Easy to extend
Frameworks and libraries can be extended over time, and this should be kept in mind during the design of the API.

Complete
This is an ideal to pursue. Since it might be impossible to create a *com-*

plete API, it should allow developers to extend it or customize it to fit their needs.

3.2 Standard Library Guidelines

The *Standard Library Guidelines* are a refined version of a set of ideas from the ISO standards committee members from 2012. The guidelines have been assembled in order to assist the *C++* community in getting libraries accepted and included in the standard. The guidelines cover writing documentation and examples, designing library components and public interfaces, and coding conventions. The guidelines are very general, and gives an idea about what a *C++* developer would expect of a library.

3.3 Strategy for use of guidelines

We consider the points of *The Little Manual of API Design* to be principles worth considering when making general decisions regarding the API design. The points of the *Standard Library Guidelines* is much more specific to some implementation details, and as such we use those to look up specific things, such as naming conventions.

4. Framework Design

The design of the framework, including API and architecture, is described in this chapter. The thought process behind decisions are included here to provide reasoning for the choices when multiple options are present.

4.1 Design Approach

We consider designing *YAGAL* a task that can be separated in two sub tasks; designing the API, and designing the architecture.

The API is the front-end of the framework, which is what a developer is going to see when using the framework. Designing the API is the task of defining the programming model of the framework, including what functions and types are made available to developers. The API design dictates the usage and learning experience for developers, and som principles to guide the development of this part is covered in chapter 3.

The architecture is the structure of the framework, its run-time environment, its compilation process, and platform support. Architectural design decisions define which use cases are possible to facilitate, and have an impact on how a developer will incorporate the framework into her work flow.

4.1.1 Design Order

We expect the order of design to be able to change the final outcome, as both API and architecture design can set requirements and limitations for the other. The order of which these get designed is important, and different approaches are discussed here.

API First

When the API is designed before the architecture, we expect the following pros and cons, in no specific order:

Pros:

- The API design can be done with ease of use as the primary goal.
- The API design can be done without having to let architectural details propagate into it.
- The architecture design can be developed towards specific tasks.

Cons:

- The architecture can be limited due to some API design decisions being unfeasible on some platforms, such as requiring features of a specific vendor.
- The API usage might be too different from the underlying implementation, resulting in some computations being more expensive than needed, such as abstracting memory location can result in multiple redundant data transfers.
- The API design might impose requirements on the architecture that are difficult to implement or self conflicting, such as two actions in the API requiring two different architectures to be meaningful.
- The API design might require features that make the compilation process convoluted for the developer, such as requiring custom pre-processing before compilation.

In general, this approach is more likely to make *YAGAL* ergonomic to use, but it might be at at the cost of architectural simplicity.

Architecture First

When the architecture is designed before the API, we expect the following pros and cons:

Pros:

- The framework structure will be simpler to explain.
- Performance optimizations will be more convenient to implement.
- Allow a convenient tool chain, with a well defined compilation process.

Cons:

- The API design might have implementation details revealed.
- The API design might be constrained, due to limitations in the architecture.

In general, this approach is more likely to make *YAGAL* consistent, but it might come at a cost in the form of limitations in the API.

Decision

As the goal of this thesis is to make a framework that makes it fast and simple for a developer to try GPGPU accelerated code, we choose to design the API first, as it will put the ergonomics of the framework first. As making the underlying architecture work is an implementation problem for us, it is not a concern for a developer.

This decision is one we reflect on in chapter 8, as it is done on our assumption at the time.

4.2 API Design

In this section we discuss the design of the API, and the reasoning behind the design decisions.

4.2.1 Goals

As it is stated in the motivation in section 1.1, we prioritize high abstraction over absolute performance. We avoid exposing kernel logic and put a layer of abstraction upon the general GPU model by managing kernel function setup and setup of blocks and threads for the developer.

In the following sections we discuss some of the options and decisions taken while designing *YAGAL*.

4.2.2 Data Types

We want to provide a developer with types that makes it convenient to work with the GPU. It is difficult to define what makes types accessible and easy to work with since there are multiple factors in play, such as the context of the programming language, the context of application, and what an individual developer sees as convenient. In this section and subsections, we explore some of these factors in the form of memory model, the actual data types, and how types are accessed, in order to determine how the types of *YAGAL* should be designed.

Memory Model

We consider two general methods for managing the underlying memory of *YAGALs* data types. One is that the data types represent a unified memory layout, where the actual location of the data is handled by *YAGAL*, involving any data transfers required to perform computations. The other is to provide methods for a developer to control the location of the data, giving her more work, but more explicit control.

In the case of *YAGAL* being able to manage memory transfers between host and device implicitly, data should be available and behave as regular *STL* data types when used on host, and be available for use on device. *Bolt* and *PACXX* manage memory this way, as seen in related works in chapter 2. An advantage of this approach is that a developer does not need to be concerned about memory when working with *YAGAL*. A downside is that the developer loses control of when transfers are happening, which can come at a high performance cost. It may also create difficulties for a developer when she want to replace *YAGAL* with *CUDA*.

Memory in *C++* is managed explicitly, and an experienced *C++* developer is used to know and be in control of how and where data is stored. As *YAGAL* is a *C++* library, it makes sense to let developers manage memory themselves. Based on this we decided that *YAGAL* will facilitate ways to manually manage where data is allocated and be able to control transfers between host and device. In related works in chapter 2 we saw that *Thrust*, *C++ AMP*, and *SkelCL* manages memory this way.

We decided that *YAGAL* will provide developers with the means to manage allocation and transfers of data between host and device themselves, instead of handling it implicitly.

Supported Types

There are many types that *YAGAL* can potentially provide. To stay within the time frame of this thesis, we decide to keep the current amount of provided types to a minimum. The types we have considered are `arrays`, `vectors`, and `maps`.

`vectors` are contiguous memory that can be used as arrays with the possibility of being dynamically resized, in contrast to `arrays` which are contiguous memory with a static size. The frameworks in chapter 2 all provide `vectors` with the exception of *C++ AMP* that uses `arrays` and `array_views`. Based on this, we want *YAGAL* to support `vectors`, but it is unclear to what degree of complications the implementation of `vectors` would introduce, due to their dynamic functionality. `arrays` on the other hand, appear to be more simple to implement due to them being statically sized. We will therefore focus on implementing `vectors` and have `arrays` as fallback solution, as the possibility of being able to dynamically resize a data collection is a quality that makes it easier for developers to use.

In regard to multidimensional support for `vectors` and/or `arrays`, it is convenient to have as it makes some tasks easier to implement, but it is of low priority as it is a more specialized feature, that is not needed by all developers, in contrast to single dimension arrays or vectors. It would be useful for *YAGAL* to support it, but it will not be prioritized.

The `map` data container is an interesting option, since the GPU could be utilized to perform lookups. This however does not contribute to the overall goal of *YAGAL*, as we have not observed this feature in any of the tools covered in chapter 2 and the introduction of this feature does not contribute to *YAGAL* being replaceable. `map` will therefore not be supported.

Accessing Data

When a data container is in use, there are multiple possible methods of accessing the data that can be expected.

When a single element is required, either for read or write, we consider two options:

- Using `container.get(index)` and `container.set(index, value)` to read and modify.
- Using `container[index]` to read and modify.

We chose to implement get and set functions, as the square bracket accessing can be developed on top of these at a later point if needed.

It is more problematic when more than a single element is needed. There are multiple options we consider:

- Providing iterators to provide iteration over data, as is tradition in *C++*.
- Providing a pointer to the first element and a number of elements to let a developer control access, as is tradition in *C*.
- Providing the number of elements to let the developer use single element accessing methods.
- Providing casting rules to let a developer cast a collection to another type that provides the needed accessing methods at the cost of a data transfer to host.

We chose to provide a developer with the device pointer to the data, and the number of elements contained, giving the developer information enough to use the single element accessors. This also allow direct access to the memory for other frameworks if multiple frameworks are used together.

Providing iterators is problematic as it motivates the developer to make many small copies back and forth between device and host. Instead we chose to provide a method to create a *STL* vector containing a copy of the device data, which can provide iterators for the device data copy. The device data copy can then be copied back to the device. This results in only two transfers per collection being required, which is to prefer compared to two transfers per element in a collection.

4.2.3 Functions

We chose to experiment with a different function structure compared to the related works in chapter 2. Where the related works generally define some kind of kernel function, and then apply it to a collection, we want to build the kernel on the collection to make the kernel definition appear as methods on the collection.

Calling Execute on a Collection

We build a kernel on a collection lazily, when a developer use an execute method. She can build up kernel functionality by appending method calls on the collection. The methods represent actions that the developer want to perform on the collection. The actual kernel function will be generated and executed based on the stored actions, when she calls the execute method on the collection that have actions queued.

An advantage of this approach is that it could prevent unnecessary data transfers between host and device since it might be possible to bundle necessary logic within a single kernel function.

A disadvantage of this approach is that it could be a cause for confusion for the developer. When the developer applies `add(5)` to a collection, a developer might expect the addition to have already taken place.

Function Chaining

We want to make a single kernel able to execute multiple actions, such as adding and multiplying a value in sequence. To enable this each function call must return a reference to the collection object so that multiple actions can be queued before the final execute call.

This will allow a program, that adds 5 to all elements of a collection, before multiplying them with 2, to be written as `collection.add(5).multiply(2).exec()`, which we find to be intuitive to read in comparison to the definition and usage separation of kernels from the related works.

Allowing chaining of functions also allow the framework to determine how, and if, kernels should be split, such that the example above would result in executing a single kernel doing both actions, rather than executing an add kernel and a multiply kernel in sequence.

Primitive Functions

Functions should be provided to give developers a baseline of functionality. These functions represent simple action to append to collections. Such a function could be `collection.add(X)` which will add `X` to each member of the collection.

Here are some primitive functions representing actions that each take either a single value or a collection:

`collection.add(X)`
: Add a value `X` to each element of the collection or add entries of a given collection `X` to the corresponding entries of the collection.

`collection.sub(X)`
: Subtract a value `X` from each element of the collection or subtract entries of a given collection `X` from the corresponding entries of the collection.

`collection.div(X)`
: Divide each element in the collection by value `X` or divide each entry by each corresponding entry of a given collection `X`.

`collection.mult(X)`
: Multiply each value of the collection by the value `X` or multiply each entry of the collection by each corresponding entry of the collection `X`.

These functions also serve as a way for us to get started experimenting with code generation since the operations involved are relatively simple. We implement these primitive functions to provide base functionality.

Higher Order Functions

We implement higher order functions in order to allow us to experiment with different methods of passing one function to another. Higher order functions provide an additional layer of abstractions for the developer.

Here is a list of higher order functions that have been considered for implementation:

Transform
: Takes a function that describes how to transform each element in the collection.

Sort
: Sorting a collection requires another collection traversal strategy compared to a transform.

Contains
: Checking whether a collection contains an element that satisfies some predicate require the ability to return a value.

Filter
: Creating a new collection with some elements from the original collection

requires the ability to dynamically share where potential elements should be inserted between threads.

The list above is not an exhaustive list as there are other higher order functions that can be relevant. We prioritize working on transform, as it is a function that allow us to focus on function passing, while reusing the collection traversal strategy of the primitive functions. If successfully implemented we can revisit other higher-order functions.

4.3 Architecture Design

In this section we discuss the design decisions made regarding the architecture of *YAGAL*, including use of external technologies, and compatibility.

4.3.1 Using LLVM and CUDA Driver API

In related works in chapter 2 we see that there are frameworks which use *LLVM* to build a compiler that output either *PTX* or *SPIR* code. We also see that there are frameworks that, without a compiler, generate *OpenCL* code. We have not found any frameworks that, without a compiler, translates to *CUDA* or *PTX*. This is interesting, and we want to attempt this, to figure out why that can be.

A part of *LLVM* is the intermediate language, known as *LLVM Intermediate Representation*. It is designed to be higher level than most assembly languages, and is used by *LLVM* as a source of compilation when compiling to a final target, such as *PTX*. *LLVM Intermediate Representation* is a higher level language than *PTX*, and is thereby easier to generate code for. The idiomatic use of *LLVM* is to build a compiler with the passes provided by the *LLVM*, and additional passes defined by the compiler developer, but as we want to avoid building a compiler that a developer is forced to use to be able to utilize our framework, we can not do this. Instead we want to use the *LLVM* framework by including it in our library in such a way, that the user of *YAGAL* compiles the necessary components of *LLVM* into her executable. This allows us to generate *LLVM Intermediate Representation* and use one of *LLVM*s back-ends to target *PTX* inside the *YAGAL* library.

When we can generate *PTX* code we can focus on executing it, which is done using the *CUDA Driver API*. The *CUDA Driver API* differs from the *CUDA Runtime API* by being slightly more verbose, but allowing *PTX* to be invoked, and not requiring the *NVIDIA* compiler. Using the *LLVM* library in conjunction with the *CUDA Driver API* allows us to handle code generation and execution without requiring a specific compiler.

Using the *LLVM* library and the *CUDA Driver API* does impose dependencies that the developer is required to have available. While the compiler choice is still up to the developer, the need to install these two libraries is still present as a prerequisite to using *YAGAL*.

Figure 4.1 illustrates an overview of *YAGAL*. We show that the source is clean *C++* with nothing but *YAGAL* library inclusion. This source, when compiled, results in an executable that contains the logic of the program, together with the *YAGAL* logic to both generate *LLVM Intermediate Representation* and manage memory, together with the *LLVM Translator* which translates the *LLVM Intermediate Representation* to *PTX*. The run-time requirement of a *CUDA Driver* is present as it is the part that facilitates memory management and code execution on the GPU.

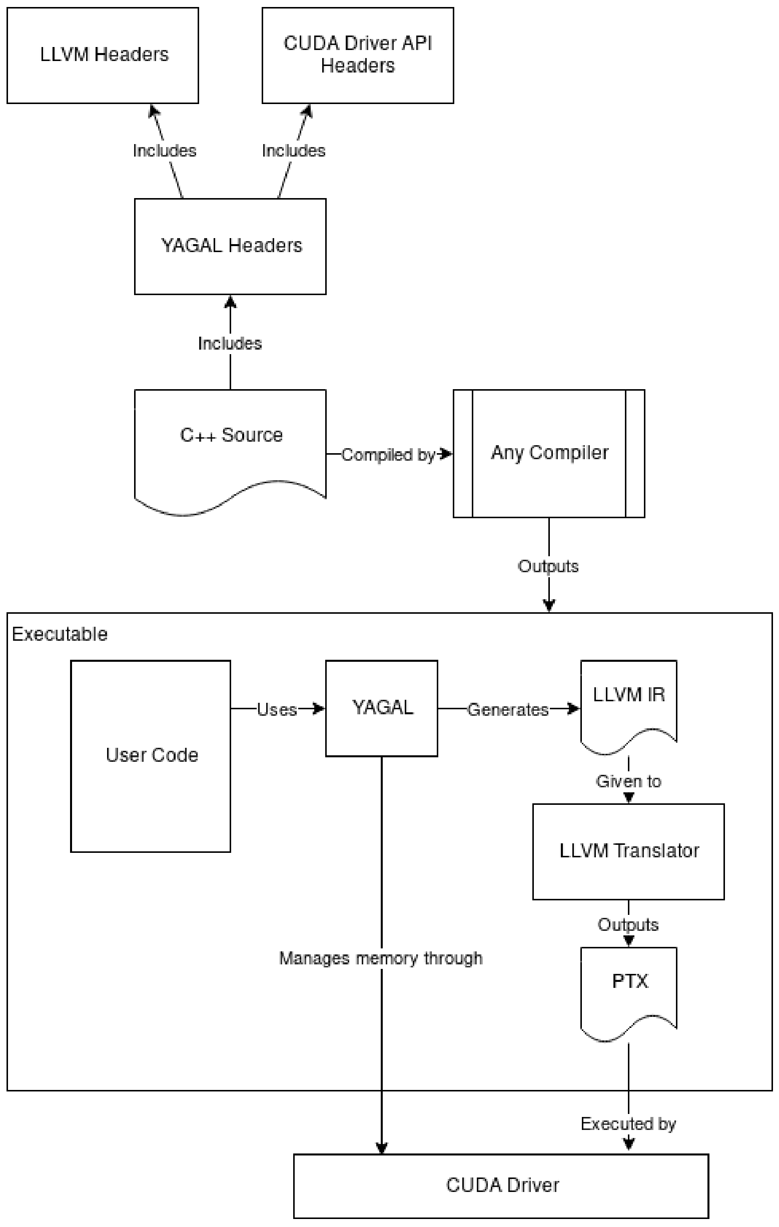

Figure 4.1: Architectural overview of YAGAL.

4.3.2 CUDA Compatibility

In cases where programs are unable to be expressed in *YAGAL*, we want to provide a fall back solution. We will do this through letting the developer execute arbitrary *PTX* code, through the *CUDA Driver API*. To allow this we want our abstractions to provide access to the relevant device pointers to the GPU.

This solution still require the developer to have *PTX* code that solves the problem. Normally this would mean writing *CUDA* code, and compile that to *PTX*, which is a long workaround. As such it is not ideal, but we decide that it is better to provide a problematic workaround, rather than none.

5. Framework Implementation

The framework implementation is performed in incremental steps, as described in the development process section 1.3. This chapter contain implementation details of the steps in order of completion.

5.1 Management of Device Memory

We have created the template class `yagal::Vector<T>` to represent a vector of elements of type T, where the data is allocated on the device. This class is constructed to follow the idea of section 4.2.2, where we state that we want the developer to have explicit control of where memory is located. A code snippet with a stripped down version of the class, only showing the relevant content, is shown in Listing 5.1, and line number references in this section refer to that listing.

The `yagal::Vector` class contains a device pointer that is used by the *CUDA* driver API, and an integer that indicates the number of elements stored beginning at that address. The memory layout is shown on table 5.1. The lifetime of the object is directly related to the lifetime of the data. When the object is created the necessary allocation is made on the device through the *CUDA* driver API, and when the object is destroyed, the allocated memory is freed.

Main memory	Device memory
Data pointer Data size	Data

Table 5.1: Information locations of a `yagal::Vector`.

`yagal::Vector` uses an `std::vector<T>` to provide a constructor and casting options for data transfers. The constructor copies data from an `std::vector` to our `yagal::Vector` with *CUDA*s `memcpy` function as a part of the object initialization, as shown at line 9. To transfer data back, we overload the casting operator to an `std::vector`, copying the data from the device to the result `std::vector` as seen at line 23.

Accessing single data elements is done through `get` and `set` functions, as seen at line 30. These functions perform the needed memory transfer for the developer. They are intended only for tasks regarding a single value, as a transfer is needed for every use, and manipulations of the entire vector should be done either in main memory by the CPU, or in device memory by the GPU.

We chose not to implement an iterator for the `yagal::Vector` in order to avoid motivating CPU computations on device memory. This makes it different from an `std::vector`, but iterating over elements can still be done, by copying the data from the device to an `std::vector`.

There is no support for resizing of a `yagal::Vector`. Implementing this is troublesome as there is no such thing as `realloc` in the *CUDA Driver API*. A possible workaround is to initially allocate all available memory and manage that in a layer below the `yagal::Vector`, however then we also make it impossible for a *YAGAL* program to use the *CUDA Driver API* for any other tasks, which we feel is a unnecessary restriction. This does show up in the API of *YAGAL*, as there is no functions such as `push_back()` or `resize()` on the `yagal::Vector`, and an initialization with the size of zero will cause a compile time error, as there is no possible meaning of it.

```
namespace yagal{
    template<typename T>
    class Vector{
    private:
        CUdeviceptr _devicePtr;
        size_t _count;

    public
        // Constructors
        Vector(int elementCount)
            : _count(elementCount)
        {
            _devicePtr = yagal::cuda::malloc(_count * sizeof(T)
                ↪ );
        }

        Vector(const std::vector<T>& source)
            : _count(source.size())
        {
            _devicePtr = yagal::cuda::malloc(_count * sizeof(T)
                ↪ );
            yagal::cuda::copyToDevice(_devicePtr, source.data()
                ↪ , _count * sizeof(T));
        }

        // Cast (Copy out)
        operator std::vector<T>(){
            std::vector<T> result(_count);
            yagal::cuda::copyToHost(result.data(), _devicePtr,
                ↪ _count * sizeof(T));
            return result;
        }

        // Accessors
        T getElement(int index){
            T result;
            yagal::cuda::copyToHost(&result, _devicePtr + (
                ↪ index * sizeof(T)), sizeof(T));
            return result;
        }

        void setElement(int index, T value){
            yagal::cuda::copyToDevice(_devicePtr + (index * sizeof(T)), &value, sizeof(T));
        }

        // Destructors
        ~Vector(){
            yagal::cuda::free(_devicePtr);
        }

    }
}
```
Listing 5.1: Vector class, showing only code relevant to memory management.

5.2 Execute PTX

Executing *PTX* code on the GPU is a central part of *YAGAL*'s functionality, and the library achieves this by interacting with the *CUDA* driver API. All *CUDA* related code in *YAGAL* are contained within the `yagal::cuda` namespace.

To execute the *PTX* code, it is passed to the function `executePtxWithParams` as a string, along with a vector containing the input parameters as seen on Listing 5.2. The last two parameters corresponds to block and grid dimensions. The optimal values for these vary depending on algorithm and data sizes. We chose to default to `128, 1, 1` for both, as it results in $128 * 128 = 16384$ instances of the kernel, which is sufficient to make use of all threads of any current GPU. When tested on sample calculations we found this to be a well performing configuration compared to other configurations. Results from these tests can be found in appendix A.

To ensure that the GPU is ready, the `initIfNeeded()` function at line 11 is called which creates a *CUDA* context if one does not already exists.

At line 15 a *CUDA* module is created by calling `cuModuleLoadDataEx()`, which is part of the *CUDA C* API. It takes *PTX* code as a `C-string`, and outputs to the provided module container.

A function handle, which is represented by a string containing the name of the function, is needed in order to determine which function within the module to execute. The handle is identified by calling `cuModuleGetFunction()` at line 19. Here a function called `kernel` is requested from within the module. `kernel` is the hardcoded name of a function that is created by *YAGAL*.

The module is executed on the GPU at line 23 by a call to the function `cuLaunchKernel` and the module is then unloaded at line 35 by `cuModuleUnload()`. The module have now been executed and the functionality within it is not needed anymore, therefore the module is unloaded.

```
int executePtxWithParams(
        const std::string& ptx,
        const std::vector<CUdeviceptr*>& kernelParams,
        std::tuple<int,int,int> blockDimensions = {128, 1, 1},
        std::tuple<int,int,int> gridDimensions = {128, 1, 1}){
    CUmodule    cudaModule;
    CUfunction  function;
    CUlinkState linker;
    int         devCount;

    initIfNeeded();

    // Create module for object
    checkCudaErrors(
        cuModuleLoadDataEx(&cudaModule, ptx.c_str(), 0, 0, 0));

    // Get kernel function
    checkCudaErrors(cuModuleGetFunction(
        &function, cudaModule, "kernel"));

    // Kernel launch
    _p.info() << "cuda kernel launching" << std::endl;
    checkCudaErrors(cuLaunchKernel(function,
        std::get<0>(gridDimensions),
        std::get<1>(gridDimensions),
        std::get<2>(gridDimensions),
        std::get<0>(blockDimensions),
        std::get<1>(blockDimensions),
        std::get<2>(blockDimensions),
        0, NULL,
        (void**)kernelParams.data(),
        NULL));

    // Cleanup
    checkCudaErrors(cuModuleUnload(cudaModule));
    _p.info() << "cuda kernel executed successfully"
        << std::endl;

    return 0;
}
```

Listing 5.2: int executePtxWithParams().

5.3 Queuing Actions

To modify a `yagal::Vector`, we create a concept of actions. These actions contain the necessary information for generating a kernel, that executes the corresponding logic.

An action is an intended task to be performed on the `yagal::Vector`, and is constructed as a result of a developer calling a function on the `yagal::Vector`. For instance, calling `myVector.add(5)` will result in an `AddAction` being created with a copy of the parameters, 5 in this example, needed to perform the calculation. The developer must then call `myVector.exec()` to perform all applied actions. We use the actions as the source of our code generation, as we generate code to perform exactly the content of the action.

The template class `Vector<T>` is expanded, as seen in Listing 5.3. A vector of `Action<T>` elements are added to the fields at line 7, it contains the actions needing to be performed on the `Vector<T>`, in the order they were added. Functions have been added to generate actions that are placed in this vector. An example of this is `Vector<T>& add(T value)` at line 12, which creates an action that represents adding a value to all elements in the `Vector<T>`, and places it at the back of the `_actions` vector. The other added function is `Vector<T>& exec()`, which is the function that consumes the action vector, generates *LLVM IR* based on the consumed actions, translates the *LLVM IR* to *PTX*, loads the *PTX* on the device, and executes it. The content of the `Vector<T>& exec()` function is not the focus of this section, and is explained in section 5.4.6.

```
 1  namespace yagal{
 2      template <typename T>
 3      class Vector{
 4      private:
 5          /* omitted fields */
 6
 7          std::vector<std::shared_ptr<internal::Action>> _actions
                ↪ ;
 8
 9          /* omitted functions */
10
11      public:
12          Vector<T>& add(T value) {
13              _actions.emplace_back(new internal::AddAction<T>(
                    ↪ value));
14              return *this;
15          }
16
17          /* omitted functions */
18
19          Vector<T>& exec(){
20              /* omitted logic */
21          }
22      }
23  }
```

Listing 5.3: Vector<T> action additions.

Figure 5.1 contain a class diagram of the inheritance for actions. The template class `Action<T>` is the single top level class in the hierarchy and is shown at the top of the class diagram. The every bottom level class is a concrete action that can be performed on a `yagal::Vector<T>` and are shown at the bottom of the class diagram as `AddAction` and `AddVectorAction`. These actions are grouped by mid level classes, that are shown in the class diagram as `SimpleAction` and `ParameterAction`, which define the input parameters. An example is the `AddAction<T>` being a `SimpleAction<T>`, as it takes a single input value to perform the action on an element of the `yagal::Vector<T>`, and the `SimpleAction<T>` being an `Action<T>` to allow us to contain it with other actions in the vector on line 7.

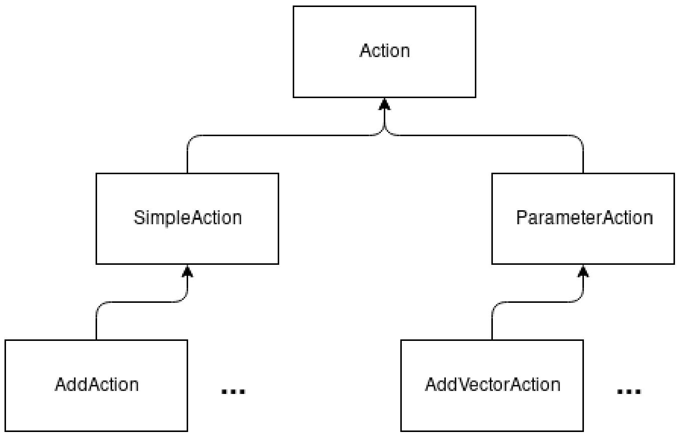

Figure 5.1: Class diagram for *Action*.

5.4 Generation of LLVM IR

We generate *LLVM Intermediate Representation*, which we now refer to as *LLVM IR*, which can later be transformed to *PTX* code, and be executed. We use the tools provided by the *LLVM* library to generate this *LLVM IR*, as they manage details such as variable names and references.

The *LLVM IR* is based on a nested structure as seen on Figure 5.2, where `LLVM Module` is the program, `LLVM Function` is a function that is a member of the module, `LLVM Basic Block` is a code block, and `LLVM Instruction` is an instruction.

When a `yagal::Vector`'s `exec` function is called, the queued actions are used as source for the *LLVM IR* generation. The procedure for *LLVM IR* generation is as follows:

1. Generate *LLVM Module* with platform information.
2. Generate *LLVM Function* with parameter information extracted from the

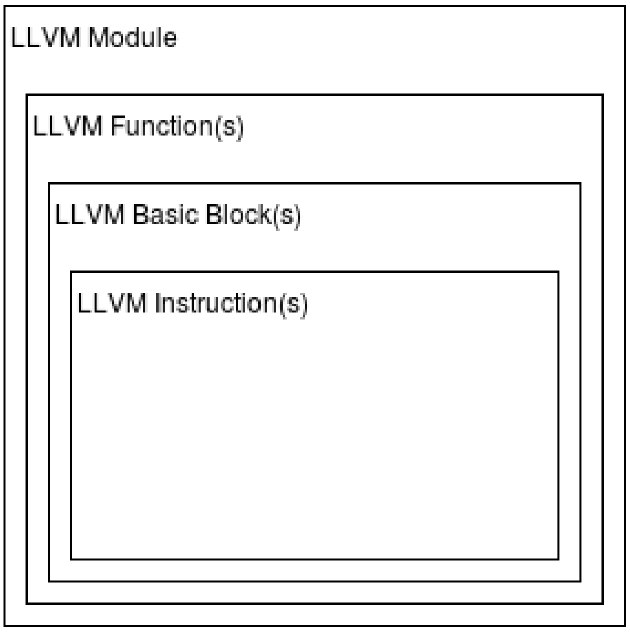

Figure 5.2: The LLVM IR hierarchy.

set of actions on the `Vector`.

3. Generate *LLVM Basic Block* to perform logic of a single action.
4. Repeat step 3 until all actions have a corresponding *LLVM Basic Block*.
5. Connect *LLVM Basic Block*s by branching instructions to create program flow.
6. Generate *LLVM Metadata* for functions.

The benefit of this procedure is that it allows multiple actions to be compiled to a single kernel function, and then be executed on the GPU in a single step.

The implementation of the procedure is presented in the following subsections.

5.4.1 Generating an LLVM Module

The *LLVM Module* is the outer most abstraction of *LLVM IR*. In our implementation we encapsulate it in the class `IRModule`. This class contains functions needed to generate *LLVM IR*, and the *LLVM Module* as a member variable that are accessed through these functions. A listing showing an extract of the class with the relevant content can be seen in listing 5.4.

Before we can create any *LLVM* objects, an `LLVMContext` is needed. A `LLVMContext` is a container of the state that *LLVM* is in, and is declared at line 8.

With the context we create the module in the constructor chain at line 23, along with the initialization of the intrinsics which will be used by the kernel to fetch information at run-time. An architecture must then be defined for the module. We provide the string "nvptx64-nvidia-cuda", as it is a key in the *LLVM* library, that is used to look up target information during code generation.

```
 1 namespace yagal::generator{
 2     //Representation of a module in llvm ir.
 3     //Contains the logic to configure a llvm module and provide
         ↪  functions for the rest of the library to add
         ↪ functionality to a module
 4     class IRModule{
 5     public:
 6         /* Some fields omitted */
 7
 8         llvm::LLVMContext context;
 9         llvm::Module module;
10
11         uint64_t elementsToHandle;
12         std::vector<llvm::Function*> kernels;
13         std::vector<llvm::BasicBlock*> userBlocks;
14
15         //Core function variables
16         llvm::Function* getThreadIdxIntrinsic;
17         llvm::Function* getBlockIdxIntrinsic;
18         llvm::Function* getBlockDimxIntrinsic;
19         llvm::Function* getGridDimxIntrinsic;
20         llvm::Value* currentIndexValue;
21
22         IRModule(uint64_t numberOfElements):
23             module("yagalModule", context),
24             getThreadIdxIntrinsic(llvm::Intrinsic::
                 ↪ getDeclaration(&module, llvm::Intrinsic::
                 ↪ nvvm_read_ptx_sreg_tid_x)),
25             getBlockIdxIntrinsic(llvm::Intrinsic::
                 ↪ getDeclaration(&module, llvm::Intrinsic::
                 ↪ nvvm_read_ptx_sreg_ctaid_x)),
26             getBlockDimxIntrinsic(llvm::Intrinsic::
                 ↪ getDeclaration(&module, llvm::Intrinsic::
                 ↪ nvvm_read_ptx_sreg_ntid_x)),
27             getGridDimxIntrinsic(llvm::Intrinsic::
                 ↪ getDeclaration(&module, llvm::Intrinsic::
                 ↪ nvvm_read_ptx_sreg_nctaid_x)),
28             elementsToHandle(numberOfElements)
29         {
30             //Set platform specific variables for the module.
31             module.setTargetTriple("nvptx64-nvidia-cuda");
32
33             _p.debug() << "ir module constructed" << std::endl;
34         }
35
36         //Create a function ready for insertion with a
             ↪ IRBuilder.
```

```
        llvm::Function* createKernel(int numberOfParameters){
            /* omitted */
        }

        //Creates the return point of a kernel, and links blocks together, to effectively make them labels.
        void finalizeKernel(llvm::Function* kernel){
            /* omitted */
        }

        //Update metadata of module to correctly tag the kernel functions.
        void updateMetadata(){
            /* omitted */
        }

        /* other functions omitted */
    };
}
```

Listing 5.4: The IRModule class.

With the module created, we can build *LLVM Function*s to create our kernel.

5.4.2 Generating an LLVM Function

An *LLVM Function* can be mapped to a GPU kernel, as a GPU kernel is a function callable on the GPU. The code we use to generate an *LLVM IR* function is shown in listing 5.5.

Before we can declare the function, we need to prepare argument types, as seen from line 2 to 5 by providing type information and address space of parameters. The second parameter to `getFloatPtrTy`, `1` indicates which address space the parameter is in, with 1 being the global memory of the device.

We construct the function as a member of the module on line 7, by providing type information, linking method for external functions, name, and containing module. The type information consists of the return type `void`, as we expect no direct return value from a kernel, the previously constructed parameter types, and whether the number of parameters can vary. The linkage refers to the visibility of the function, with external meaning that it can be accessed from other modules.

To make sure the function follows the calling convention and parameter parsing of the platform, we use the `setCallingConvention` function, as seen on line 14.

```
1  llvm::Function* createKernel(int numberOfParameters){
2      std::vector<llvm::Type *> kernel_arg_types;
3      for(int i = 0; i < numberOfParameters; i++){
        kernel_arg_types.push_back(llvm::Type::getFloatPtrTy(context, 1));
    }

    auto kernel = llvm::Function::Create(
        llvm::FunctionType::get(llvm::Type::getVoidTy(context), kernel_arg_types, false),
        llvm::Function::ExternalLinkage,
        llvm::Twine("kernel"),
        &module
    );

    kernel->setCallingConv(llvm::CallingConv::PTX_Kernel);

    return kernel;
}
```

Listing 5.5: The createKernel function.

With the function created we can build *LLVM Basic Block*s within them, to provide functionality to the kernel.

5.4.3 Generating an LLVM Basic Block Based On an Action

The functionality needed of a kernel is dictated by the actions described in section 5.3. All actions have a function that generates a corresponding *LLVM Basic Block*. This function can be seen on line 6 in listing 5.6.

First we create an *LLVM Basic Block* at line 10, by providing the `LLVM Context` of the *LLVM IR* container, a name, and the function that should contain it. We then create an `LLVM IRBuilder`, which is an object that provides functions for generating *LLVM IR* from *LLVM* objects, and adds the block to the vector of blocks, that should be managed by the *LLVM IR* container.

After constructing the *LLVM Basic Block*, we get a pointer to the first kernel parameter, which is the vector we are manipulating. We can now build the *LLVM IR*.

We use the `IRBuilder` to create the following instructions as seen in listing 5.6:

- Line 20 creates the `load` instruction, that loads the index of the value that should be modified.
- Line 21 creates the `getElementPtr` instruction, that gets the pointer to the correct element of the vector based on index.
- Line 22 creates the `load` instruction, that loads the value of the vector before modification.

- Line 23 creates a constant value based on the value the AddAction was constructed with.
- Line 24 creates the add instruction for floating point values, that adds the constant to the loaded value, and assigns a temporary value for the result.
- Line 25 creates the `store` instruction, that stores the temporary value at the address of the element pointer.

```
1  template <typename T>
2  class AddAction : public SimpleAction<T>{
3  public:
4      AddAction(T v): SimpleAction<T>(v) {}
5
6      void generateIR(yagal::generator::IRModule& ir, llvm::Function* kernel, int& inputVectorCounter){
7          _p.debug() << "generateIR for add action called." << std::endl;
8
9          //Prepare the block to fill in
10         auto actionBlock = llvm::BasicBlock::Create(ir.context, llvm::Twine(ir.getNextBasicBlockName()),kernel);
11         llvm::IRBuilder<> builder(actionBlock);
12         ir.userBlocks.push_back(actionBlock);
13
14         //Prepare argument
15         auto vecVal = kernel->arg_begin();
16         vecVal->setName("vec");
17
18         //Build llvm ir
19         int alignment = 4;
20         auto indexVar = builder.CreateAlignedLoad(ir.currentIndexValue, alignment, "i");
21         auto ptrVal = builder.CreateGEP(vecVal, indexVar, "ptr");
22         auto tmpVal = builder.CreateAlignedLoad(ptrVal, alignment, "tmp");
23         auto inputConst = llvm::ConstantFP::get(llvm::Type::getFloatTy(ir.context), (float)this->value);
24         auto retVal = builder.CreateFAdd(tmpVal, inputConst, "ret");
25         builder.CreateAlignedStore(retVal, ptrVal, alignment);
26     }
27 };
```

Listing 5.6: The AddAction class.

Now that we can build a *LLVM Basic Block* we are almost ready to execute the code. There still is no return statement in the function, and the code is still invalid as not all possible execution paths return.

5.4.4 Connecting LLVM Basic Blocks

Now that we can generate *LLVM Basic Block*s in a function, we need to go through all basic blocks of a function and ensure that they are connected in a way that leads to the function executing all basic blocks and returning when done.

To control the flow, two additional basic blocks are created, one representing the entry, and one representing the exit. The entry block contains an unconditional branch to the first basic block, and the exit block contains the return statement. We then iterate all basic blocks in the function, and add a branch instruction at the end, leading to the next basic block. When we reach the last basic block, we make it branch to the exit block. Figure 5.4.4 shows the resulting flow, and with that we are done modifying the content of the function.

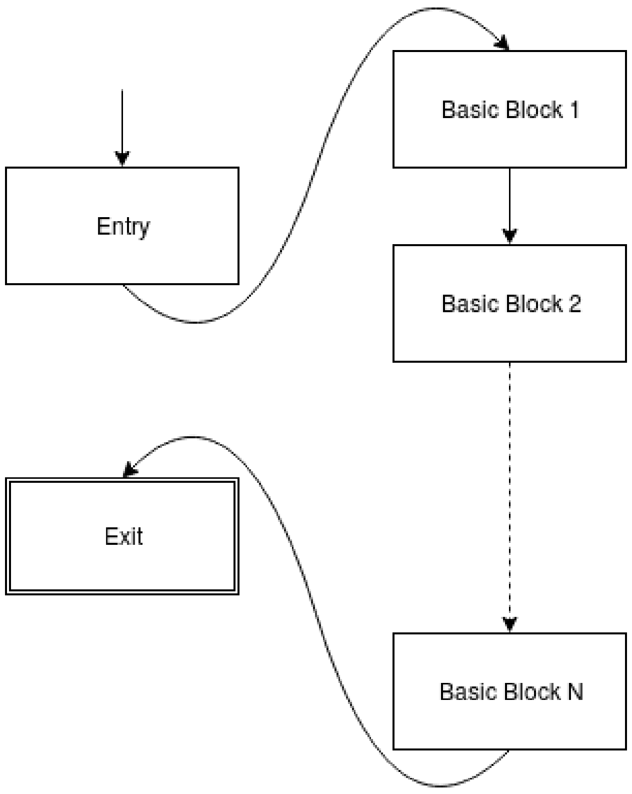

Figure 5.3: The connections of Basic Blocks.

5.4.5 Generating LLVM Metadata

The *LLVM Function* is ready to be executed on the GPU, but is not marked as a GPU kernel yet. The *NVPTX* back end requires us to provide the function we want marked as a kernel, and to do this we use *LLVM Metadata.* On listing 5.7 we show how we use the *LLVM API* to create a metadata node tagging our function, named "kernel", as a kernel function.

```
void updateMetadata(){
    auto metadata = module.getOrInsertNamedMetadata("nvvm.annotations");
    auto oneconstant = llvm::ConstantInt::get(llvm::Type::getInt32Ty(context), 1);
    std::vector<llvm::Metadata *> ops{
        llvm::ValueAsMetadata::getConstant(module.getNamedValue("kernel")),
        llvm::MDString::get(context, "kernel"),
        llvm::ValueAsMetadata::getConstant(oneconstant)
    };
    auto metadata_node = llvm::MDTuple::get(context, ops);
    metadata->addOperand(metadata_node);
}
```

Listing 5.7: The updateMetadata function.

5.4.6 Putting The Steps Together

Now that we have all necessary components for building *LLVM IR*, we can put them together. We do this with the `exec` function on the `yagal::Vector<T>`.

At line 2 in Listing 5.8 we construct the `IRModule`, which will construct an *LLVM Module* ready for *LLVM Function* insertion, as described in section 5.4.1.

At line 5 to 12 we prepare the parameters to kernel function. We do this by collecting the `CUdeviceptrs` of all vectors required by actions queued on the vector, so they are ready to be provided to the kernel.

At line 15 to 19 we generate an *LLVM Basic Block* as described in section 5.4.2, and create a basic block for each action in this function as described in section 5.4.3.

At line 22 we link the basic blocks of the function together, to complete the function, as described in section 5.4.4.

At line 23 we create the metadata needed to let the compiler identify the function as a kernel, as described in section 5.4.5

At line 27 we can translate the constructed *LLVM IR* to *PTX*, and be ready to execute it. How the translation is performed is covered in section 5.5.

At line 32 we call the function described in section 5.2, to execute the kernel with our parameters.

At line 35 we empty the action vector to be ready for a new action queue on the same vector.

```cpp
Vector<T>& exec(){
    yagal::generator::IRModule ir(_count);

    //Count number of cuda parameters needed, starting at 1 to
        ↪ include the vector itself.
    std::vector<CUdeviceptr*> devicePointers({&_devicePtr});
    for (auto& a : _actions){
        if(a->requiresCudaParameter()){
            auto pa = static_cast<internal::ParameterAction<T
                ↪ >*>(a.get());
            auto ptr = pa->getDevicePtrPtr();
            devicePointers.push_back(ptr);
        }
    }

    //Generate llvm ir blocks.
    int inputVectorCounter = 0;
    auto kernel = ir.createKernel(devicePointers.size());
    for (const auto& a : _actions){
        a->generateIR(ir, kernel, inputVectorCounter);
    }

    //Link blocks and update metadata.
    ir.finalizeKernel(kernel);
    ir.updateMetadata();

    //Generate code
    _p.debug() << ir.toString() << std::endl;
    yagal::generator::PTXModule ptx(ir);
    auto ptxSource = ptx.toString();
    _p.debug() << ptx.toString() << std::endl;

    //Execute kernel
    yagal::cuda::executePtxWithParams(ptxSource, devicePointers
        ↪ );

    //Cleanup
    _actions.clear();

    return *this;
}
```

Listing 5.8: The exec function.

5.5 Replacing LLVM LLC

LLVM LLC is a command line tool that takes *LLVM IR* as input and converts it to assembly language for a specified architecture such as *PTX*. Since *LLVM LLC* is a command line tool, it becomes difficult to include this in *YAGAL*, since this

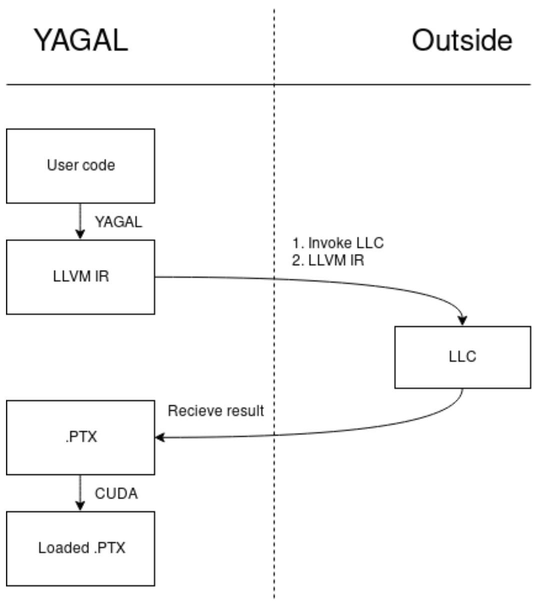

Figure 5.4: Yagal using LLC.

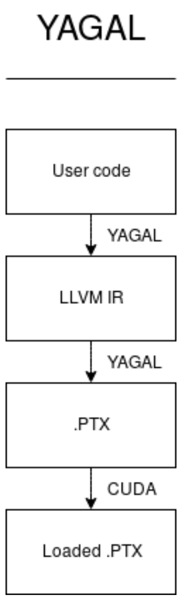

Figure 5.5: YAGAL with LLC replacement.

would require the library to start, and be dependent upon, an external process. This can be seen on Figure 5.4 where *YAGAL* starts the *LLVM LLC* process and provides it with the *LLVM IR*, and must await the result.

The source code for *LLVM LLC* is open source and available on *GitHub*[17], and we have implemented a subset of it to fit our use case by using the original implementation as an example.

The subset of *LLVM LLC* we implemented involves *PTX* generation as single purpose. It have been implemented directly into *YAGAL* in order to avoid being dependent upon an external processes. This has resulted in the flow which is shown by Figure 5.5 where the *LLVM LLC* tool is no longer involved in the process.

The *PTX* generation is contained within the `PTXModule` class. This class functions as a container to generate and hold *PTX* code based on a given `IRModule`. The constructor of the `PTXModule` class can be seen in Listing 5.9, and line references in this section is referring to this Listing.

The `initializeLlvmTargetIfNeeded` function invokes *LLVM C* macros if they haven not already been invoked. This initializes the *LLVM* backend for *NPTX* which is needed for converting *LLVM IR* to *PTX*. The `initializeLlvmPassRegistryIfNeeded` function specifies which passes the *NPTX* backend should perform upon the *LLVM IR*.

At lines 5 through 15 we set the variables needed for the target machine. These variables represent options that would have been provided to *LLVM LLC* as command line arguments.

At line 27 we initialize the `PassManager`. A `PassManager` is an *LLVM* structure that manage the compilation passes. The compilation passes we need are specified in the `initializeLlvmPassRegistryIfNeeded` function.

A `LLVMTargetMachine` is instantiated at line 34 along with a `MachineModuleInfo` pointer. These contain meta information regarding the target machine which are necessities for LLVM.

The data layout for the `PtxModule` is set at line 39. The data layout describes how data is to be laid out in memory. The data layout for the GPU in our test machine is `e-i64:64-i128:128-v16:16-v32:32-n16:32:64`.

The given passes are registered and initialized by call to the `addPassesToEmitFile` function at line 41. The output buffer is also registered here, which is where we can read the resulting *PTX* code fom.

Finally, the passes are executed at line 45. The result is then witted to a the `_string` variable, which is a member of the `PtxModule` class, at line 47. The `_string` variable is later in the process passed to CUDA through `yagal::cuda::cudaHandler` for execution.

```
 1  PTXModule(IRModule& ir){
 2      initializeLlvmTargetIfNeeded();
 3      initializeLlvmPassRegistryIfNeeded();
 4  
 5      std::string arch("nvptx64");
 6      llvm::Triple triple(llvm::Twine("nvptx64-nvidia-cuda"));
 7      std::string error;
 8      const llvm::Target *target(
 9          llvm::TargetRegistry::lookupTarget(
10              arch, triple, error));
11      std::string cpuStr("sm_20");
12      std::string featureStr("");
13      llvm::CodeGenOpt::Level optLevel(
14          llvm::CodeGenOpt::Aggressive);
15      llvm::TargetOptions options;
16  
17      std::unique_ptr<llvm::TargetMachine>
18          targetMachine(target->createTargetMachine(
19          triple.getTriple(),
20          cpuStr,
21          featureStr,
22          options,
23          llvm::None,
24          llvm::CodeModel::Small,
25          optLevel));
26  
27      llvm::legacy::PassManager passManager;
```

```

    llvm::SmallVector<char, 512> buffer;
    auto bufferStream =
        std::make_unique<llvm::raw_svector_ostream>(buffer);
    auto outputStream = bufferStream.get();

    llvm::LLVMTargetMachine &llvmtm =
        static_cast<llvm::LLVMTargetMachine&>(*targetMachine);
    llvm::MachineModuleInfo *mmi =
        new llvm::MachineModuleInfo(&llvmtm);

    ir.module.setDataLayout(
        targetMachine->createDataLayout());
    targetMachine->addPassesToEmitFile(
        passManager, *outputStream,
        llvm::TargetMachine::CGFT_AssemblyFile, false, mmi);

    passManager.run(ir.module);

    _string = std::string(buffer.begin(), buffer.end());

    _p.debug() << "ptx module constructed" << std::endl;
}
```

Listing 5.9: `PTXModule` constructor based on a given `IRModule`.

5.6 Matching Number of Threads with Vector Elements

As described in section 5.4, we can generate *PTX* code from actions to modify vectors. There is however a problem which is not handled at this point: The element modified by a kernel is chosen, by using the kernels index as index in the vector. This mean that in cases of a vector having more elements than threads, the execution would not modify all values in the vector, and in the case of more threads than elements, the execution would go out of bounds, possibly corrupting other data.

The solution to this problem is to implement logic for the kernels that prevents execution of logic on elements that are out of bounds, and ensure that all elements in a vector will be modified exactly once.

With threads being organized in blocks, and blocks being organized in a grid, we use a method called striding, where each thread of each block handle a specific index in the vector before jumping to the next dedicated index. This logic is implemented to follow the process:

1. Set current index value to $threadIdX + blockDimX * blockIdX$.
2. If the index value is valid in the vector, handle logic, otherwise exit.

3. Increment the current index value by $blockDimX * gridDimX$.
4. repeat from step 2.

We only consider the x dimension, although we can use up to three, as a `yagal::Vector` only have a size in a single dimension.

We implement this feature, by extending any function we generate with two *LLVM Basic Block*s and extending the entry block. The entry block will have the logic for the initialization of the index value, and branch to the first new block, rather than the first user block. The first new block is for checking whether the condition of the index being valid is met, and branch based on that to either the user block chain or the exit block. The second new block is for incrementing the index value by the number of threads, and branch to the condition block. The last user block will be changed to branch to the increment block, rather than the exit block. This implementation is illustrated on figure 5.6, where it is shown how the different blocks are connected.

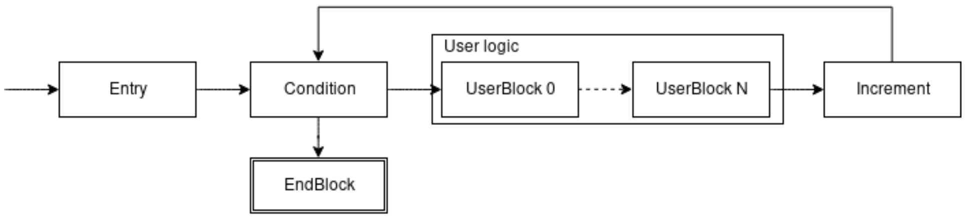

Figure 5.6: How blocks are connected within our generated *LLVM IR*.

5.7 LLVM Optimizations

The use of *LLVM* have provided advantages for *YAGAL*, such as *LLVM IR* and optimizations of generated code. This section contains some examples of where *YAGAL* have made use of these advantages.

The *LLVM* code generation tools have multiple optimization levels for the generated code. We have made use of it for the conversion of *LLVM IR* to *PTX*. This is done by setting the code generation optimization flag with `CodeGenOpt::Level optLevel(CodeGenOpt::Aggressive)`. To show an example we have the code in Listing 5.10 where a `yagal::vector` is made and two `add` functions chained on it. When equivalent *PTX* code is generated without optimizations, it will result in the code shown on listing 5.11. With the aggressive flag set, the resulting *PTX* code results in what can be see on listing 5.12. What can be seen here is that the non-optimized version consists of more labels and have multiple unnecessary instructions. The optimized version have rearranged the code by having the add instructions happen under the same label at lines 29 and 30, and having only a single store to global instruction at line 31. These kind of optimizations would have been difficult to make if we were generating *PTX* directly and not utilizing *LLVM IR*.

Another major advantage of utilizing *LLVM IR* is that it have a higher level of abstraction that *PTX*. *LLVM IR* allows function definitions with return types and provide unlimited amount of registers. This means that we do not need to take platform and hardware specifics into account since we let *LLVM* handle it during code generation.

```
yagal::Vector<float> v({1.0, 2.0, 3.0});

v.add(5).add(5).exec();
```

Listing 5.10: Chained `yagal::Vector::add`.

5.8 More Accessible PTX

In the design at chapter 4 we mention that we want to allow execution of external *PTX* code. To do this, we implement functionality to execute custom *PTX* and export of generated *PTX*.

We provide *PTX* management through two functions on the `yagal::Vector`. The first function is an overload of the `exec` function that, instead of generating *PTX* and executing it, executes a given string of *PTX*. The second function generates the *PTX* based on the actions on a vector, but instead of executing it, returns it as a string.

5.8.1 Overload of exec

The `exec` overload is shown in listing 5.13. The parameters to the function are in order:

ptxSource
: The *PTX* to execute as a *STL* string.

otherVectors
: A vector of the other relevant vectors `CUdeviceptrs`.

blockDimensions & gridDimensions
: The parameters used to start the kernel, these are optional and defaults to the values mentioned in section 5.2.

The function collects all `CUdeviceptrs`, including the one on the vector which

```
.version 3.2
.target sm_20
.address_size 64

  // .globl kernel

.visible .entry kernel(
  .param .u64 kernel_param_0
)
{
  .local .align 4 .b8   __local_depot0[4];
  .reg .b64   %SP;
  .reg .b64   %SPL;
  .reg .pred  %p<2>;
  .reg .f32   %f<5>;
  .reg .b32   %r<12>;
  .reg .b64   %rd<8>;

  mov.u64   %SPL, __local_depot0;
  cvta.local.u64  %SP, %SPL;
  ld.param.u64  %rd1, [kernel_param_0];
  mov.u32   %r1, %ntid.x;
  mov.u32   %r2, %tid.x;
  mov.u32   %r3, %ctaid.x;
  mul.lo.s32  %r4, %r3, %r1;
  add.s32   %r5, %r2, %r4;
  st.u32  [%SP+0], %r5;
  bra.uni   LBB0_1;
LBB0_1:
  ld.u32  %r6, [%SP+0];
  setp.lt.u32   %p1, %r6, 100;
  @%p1 bra  LBB0_4;
  bra.uni   LBB0_3;
LBB0_2:
  mov.u32   %r7, %ntid.x;
  mov.u32   %r8, %nctaid.x;
  mul.lo.s32  %r9, %r7, %r8;
  ld.u32  %r10, [%SP+0];
  add.s32   %r11, %r10, %r9;
  st.u32  [%SP+0], %r11;
  bra.uni   LBB0_1;
LBB0_3:
  ret;
LBB0_4:
  ld.s32  %rd2, [%SP+0];
  shl.b64   %rd3, %rd2, 2;
  add.s64   %rd4, %rd1, %rd3;
  ld.global.f32   %f1, [%rd4];
  add.rn.f32  %f2, %f1, 0f40A00000;
  st.global.f32   [%rd4], %f2;
  bra.uni   LBB0_5;
LBB0_5:
  ld.s32  %rd5, [%SP+0];
  shl.b64   %rd6, %rd5, 2;
  add.s64   %rd7, %rd1, %rd6;
  ld.global.f32   %f3, [%rd7];
  add.rn.f32  %f4, %f3, 0f40A00000;
  st.global.f32   [%rd7], %f4;
  bra.uni   LBB0_2;
}
```

Listing 5.11: Non-optimized generated PTX.

```
.version 3.2
.target sm_20
.address_size 64

  // .globl kernel

.visible .entry kernel(
  .param .u64 kernel_param_0
)
{
  .reg .pred  %p<2>;
  .reg .f32   %f<4>;
  .reg .b32   %r<9>;
  .reg .b64   %rd<4>;

  ld.param.u64  %rd1, [kernel_param_0];
  mov.u32   %r1, %ntid.x;
  mov.u32   %r5, %tid.x;
  mov.u32   %r6, %ctaid.x;
  mad.lo.s32  %r8, %r6, %r1, %r5;
  mov.u32   %r7, %nctaid.x;
  setp.lt.u32   %p1, %r8, 100;
  @%p1 bra  LBB0_3;
  bra.uni   LBB0_2;
LBB0_3:
  mul.wide.s32  %rd2, %r8, 4;
  add.s64   %rd3, %rd1, %rd2;
  ld.global.f32   %f1, [%rd3];
  add.rn.f32  %f2, %f1, 0f40A00000;
  add.rn.f32  %f3, %f2, 0f40A00000;
  st.global.f32   [%rd3], %f3;
  mad.lo.s32  %r8, %r1, %r7, %r8;
  setp.lt.u32   %p1, %r8, 100;
  @%p1 bra  LBB0_3;
LBB0_2:
  ret;
}
```

Listing 5.12: Optimized generated PTX.

the function is called on at line 2 to 5, and forwards the arguments to the `executePtxWithParams` function shown in section 5.2.

```
1  Vector<T>& exec(const std::string& ptxSource, const std::vector
       ↪ <CUdeviceptr*>& otherVectors, std::tuple<int, int, int>
       ↪ blockDimensions = {128, 1, 1}, std::tuple<int, int, int>
       ↪ gridDimensions = {128, 1, 1}){
2      std::vector<CUdeviceptr*> devicePointers({&_devicePtr});
3      for(const auto& e: otherVectors){
4          devicePointers.push_back(e);
5      }
6  
7      //Execute kernel
8      yagal::cuda::executePtxWithParams(ptxSource, devicePointers
           ↪ , blockDimensions, gridDimensions);
9  }
```

Listing 5.13: exec overload to just execute *PTX*.

5.8.2 Introduction of exportPtx

The function that creates *PTX* code based on actions is named `exportPtx` and is shown in listing 5.14.

The single parameter it takes determines if the queued actions should be deleted after *PTX* generation at line 5.14. This is implemented to allow a refactoring of the initial exec function to use this function for generating *PTX* and still be able to access the vectors provided to the actions. When at the default value, true, it clean the vector, allowing a new queue of actions to be filled.

The function performs the logic that was previously found in exec, as it presented in section 5.4.6, except the preparation of devicePointers being replaced by a count of them at at line 5 to 10, and no execution of *PTX* is happening.

```
1  std::string exportPtx(bool clearActions = true){
2      yagal::generator::IRModule ir(_count); lst:exportPtx:irmodule
3  
4      //Count number of cuda parameters needed, starting at 1 to
           ↪ include the vector itself.
5      int devicePointerCount = 1;
6      for (auto& a : _actions){
7          if(a->requiresCudaParameter()){
8              devicePointerCount++;
9          }
10     }
11 
12     //Generate llvm ir blocks.
13     int inputVectorCounter = 0;
14     auto kernel = ir.createKernel(devicePointerCount);
15     for (const auto& a : _actions){
16         a->generateIR(ir, kernel, inputVectorCounter);
```

```
    }

    //Link blocks and update metadata.
    ir.finalizeKernel(kernel);
    ir.updateMetadata();

    //Link blocks and update metadata.
    ir.finalizeKernel(kernel);
    ir.updateMetadata();

    //Generate code
    _p.debug() << ir.toString() << std::endl;
    yagal::generator::PTXModule ptx(ir);
    auto ptxSource = ptx.toString();
    _p.debug() << ptx.toString() << std::endl;

    //Cleanup
    if(clearActions){
        _actions.clear();
    }

    return ptxSource;
}
```

Listing 5.14: `exportPtx` to build *PTX* code.

5.8.3 PTX Posibilities

Using raw *PTX* it is possible to work around the build time from *YAGAL*. Allowing the developer to control when to generate the *PTX* and when to execute it separately can be beneficial in some cases.

We have made some time measurements on different methods of the different ways a kernel can be created and executed with *YAGAL*. The code we measured is shown in listing 5.15. The purpose of the logic is to take a vector, and increment all values by one. Any measurements noted in this section are done on a test machine described in section 7.1. They are only intended comparison relative to each other and should not be compared to other results on other setups.

On line 13 we use the simplest form; we queue the add action with the value 1, and immediately generate and execute the kernel. This was measured to take 22.20 milliseconds.

On line 18 we only construct the *PTX*; we queue the add action with the value 1, as previously, and output the string to a variable. This was measured to take 21.92 milliseconds.

On line 23 we execute the previously generated *PTX*. This was measured to take 0.16 milliseconds.

For this example generating the *PTX* was about 137 times more time consuming than executing it, which mean that if the computation were to happen multiple times, it could be beneficial for the developer to only generate the kernel once and apply it multiple times later.

```cpp
typedef std::chrono::high_resolution_clock Clock;
void timeBuild(){
    //Initialize a considerably big vector
    std::vector<float> src(1<<16);
    std::srand(0);
    std::generate(src.begin(), src.end(), std::rand);
    yagal::Vector<float> v(src);


    auto t0 = Clock::now();

    //Use the default method of building and executing a kernel
    v.add(1).exec();

    auto t1 = Clock::now();

    //Use exportPtx to generate ptx and execute it later
    auto ptx = v.add(1).exportPtx();

    auto t2 = Clock::now();

    //Execute previously generated ptx
    v.exec(ptx,{});

    auto t3 = Clock::now();

    /* omitted delta time calculations and output statements */
}
```

Listing 5.15: Building and executing *PTX* in action

6. Challenges and Possible Solutions

While designing and implementing *YAGAL* we encountered unexpected challenges. In this chapter we will present the major ones, and our proposals for how they can be worked around.

6.1 Anonymous Functions

We consider anonymous functions as an important part of a framework that allows construction of other functions, which in the context of this thesis refers to the construction of GPU kernels. Implementing support for anonymous functions in *YAGAL* to enable developers to extend functionality is not a straightforward task, and different approaches are discussed in this section. The code examples shown in this section are not supported by *YAGAL*, but show how anonymous functions could look as a result of each of our proposed approaches. The examples present a predicate, checking whether a squared value of a vector element is above 1000, which should be given to other non-existant *YAGAL* functions such as `filter` to provide their logic.

6.1.1 Known Approaches

An initial thought is that *C++11* provides expressive lambdas, which seem ideal for a framework such as *YAGAL*. The problem with expanding upon this construct is that it requires a compiler to be built in order to determine how to handle the lambda body, as the body is not available for a library to inspect[18], and an important part of the lambda is the body, since that is the part needed for code generation. In the related works, in chapter 2, we see that *C++ AMP* and *Thrust* both take the approach of using extended *C++* lambdas. In both cases they have extended the language with keywords in order to annotate the lambdas, and thereby tell their compiler know how the lambda should be treated differently. Even though *PACXX* lambdas can be defined with clean *C++* syntax, it still requires a specific compiler in order to generate kernels that are based upon them. In *YAGAL* we chose not to enforce the compiler choice on the developer, and thereby rendering this approach as not possible.

Another method, which is used by *SkelCL* as described in section 2.5, and *Bolt* on *OpenCL* as described in section 2.4, is to generate a kernel as a *OpenCL* string by concatenation of strings. This approach involves user defined strings that describe the anonymous function. A problem with the approach is that it requires the framework to be based upon *OpenCL*, whereas *YAGAL* is based on *CUDA*. With *OpenCL* being syntactically very similar to *C++*, it is a usable solution for allowing a developer to specify lambdas. In contrast, providing similar functionality, where the user provides the lambda as a *PTX* string, is far less intuitive due to the difference between *C++* and *PTX*.

As *YAGAL* is not based on *OpenCL*, and we do not provide a special purpose compiler, none of the related works provide a model we can re-use in *YAGAL*.

6.1.2 Alternative Approaches

To work around this problem we have tried to rethink the way kernel logic can be constructed. We came up with two ideas for constructing it. One approach is to extend the library with meta types that represent logic that would otherwise have been provided through lambdas. Another approach is to create a *CUDA C* string, and use external tools to compile it to *PTX* at run-time. These approaches are discussed in the following subsections.

Creating Meta Types

A function body could be represented by objects representing the expected syntax. An example of what this could look like is shown in listing 6.1 as being represented with meta types in listing 6.2. The different components of the kernel can be constructed like the current action implementation described in chapter 5, allowing *LLVM Intermediate Representation* to be generated.

```
bool pred(float element){
    return element * element > 1000f;
}
```

Listing 6.1: Pseudo *C* code showing a device function.

```
auto builder = KernelBuilder.New<bool, float>();
builder.addParameter(Parameter<float>("element"));
builder.addStatement(ReturnStatement<bool>(
    GreaterThanComparison<float>(
        builder.multiply<float>(
            builder.getValue<float>("element"),
            builder.getValue<float>("element")
        ),
        builder.createConstant<float>(1000)
    )
));

auto pred = builder.getResult();
```

Listing 6.2: Code showing how construction of device function in a meta type solution could be done.

This solution is extremely verbose, and difficult to read, compared to the logic that is being constructed. A developer would have an easier time by simply using another framework. Implementation wise it is a big task to implement all the different kinds of operations that would be possible. It would be ideal for this approach, to put the verbose constructions behind a more approachable system, so that the developer can define the logic with a more readable method.

Wrapping by String Interpretation

The meta types presented in the previous section in listing 6.2 could be the result of a parser. The parser could parse a well formatted *C++* string, and create the objects based upon the content of the string. With this approach it would be simpler to make use of existing compiler techniques, such as utilizing a syntax tree, compared to using a kernel builder as shown on 6.2. This tree could contain all information needed for performing code generation with *LLVM*, but would practically mean that *YAGAL*, on top of integrating a considerable part of *LLVM*, needs to include other components of a compiler, that being a lexer, a parser, and syntactical objects for building syntax trees.

A function that performs the same calculation as the above meta type example, could with such a solution look like listing 6.3. It is similar to the example shown in 6.1, which is what makes it attractive.

```
auto pred = yagal::predicateFromString<bool, float>(
    "bool pred(float element){
        return element * element > 1000f;
    }"
);
```

Listing 6.3: Code showing possible construction of device function with string interpretation.

Even though the solution in listing 6.3 looks more write- and readable than the example shown in listing 6.2 it is not a solution without problems. Requiring compilation of the string to be done at run-time will make it more difficult to get errors or warnings presented to the developer, when compared to static compilation. There is also the minor annoyance of having strings in user code that provide functionality, as it generally makes it difficult for tools, such as an IDE, to assist the developer in writing and checking code, e.g. by providing syntax highlighting or error detection.

Wrapping by C-Style Macros

Bolt, targeting *OpenCL*, uses *C* macros to wrap function objects. This allows it to have the function, after the macros interaction, syntactically checked by the compiler, as the result would still be code. It also allows it to generate an *OpenCL* code string based upon this function. As *C++* and *OpenCL* are syntactically similar in how a function is constructed, the major part of code generation is done by copying the content of the macro to a string constant. If such a solution should work for *YAGAL*, it could be done by targeting one of the other approaches, such as the previously mentioned string interpreter, by constructing a string for that.

```
auto pred = YAGALIFY(
    bool pred(float element){
        return element * element > 1000f;
    }
);
```

Listing 6.4: Code showing possible construction of device function with macro, named YAGALIFY, and *C++* lambda.

This solution can be considered cleaner than using strings only, but it does add a layer of complexity to the already high stack.

Generating CUDA C

A different approach, compared to the previously mentioned approaches, is to let a developer define a function in a string, and wrap that function in the needed components to make it a valid *CUDA C* function. That function can then be sent to the external tool *nvrtc*[19], *NVIDIA Run-time Compiler*, which generates the *PTX* code needed for that function. Even though this approach requires an additional external dependency, it does have the benefit solving the task without inventing a new language and the means of compiling it, as the previous examples did.

Example usage can is shown in listing 6.5. Notable is the lack of *CUDA* keywords, such as `__DEVICE__` to declare that it is a device function. This can be inserted by the library before parsing it to *nvrtc*, along with replacing other keywords we could introduce with intrinsics. This would relieve the user of having to learn some of the *CUDA* specifics involved when writing kernels.

```
auto myKernel = yagal::ptxFromCudaString(
    "bool pred(float element){
        return element * element > 1000f;
    }"
);
```

Listing 6.5: Code showing a possible construction of a device function based on a string. The string is being sent to the library, where it get extended to correct *CUDA C*, before being sent to *nvrtc*.

6.1.3 What Is Implemented in YAGAL

In *YAGAL* we have no implementation of anonymous functions, or functions using them. We are convinced there is a need for them, but the approaches are too involved for the time constraints of this project. The meta type solution is more straightforward to generate *LLVM Intermediate Representation* for, but would require too much development time, and it would be inconvenient for a developer to express functions with. The ideas of parsing strings, or using macros, to make the expressions less verbose could alleviate the problems of the first method, but will still require too much development time. The final solution, using *nvrtc* might be the one that is the most fitting for *YAGAL*, as we already rely on executing *PTX*. It does however come with the cost of a tighter coupling with *CUDA*.

When a method of implementing anonymous functions is done, implementing functions to use them would be possible.

6.2 Compilation Time

We implemented a subset of *LLVM LLC*, as described in section 5.5, in order to translate *LLVM IR* to *PTX* code. The consequence of this decision came in the form of additional compilation time. Even though we have stripped most of the functionality of *LLVM LLC* to only contain the *PTX* conversion, the inclusion of *LLVM* headers still have a significant impact on compilation time.

We have two proposals to how to overcome the compilation time impact for compilation of programs using *YAGAL*. We could further strip the included *LLVM* headers, or we could bundle pre-compiled binaries of our reimplementation of LLVM LLC with *YAGAL*, requiring the developer to link to these at compile time.

6.2.1 Stripped Headers

The reimplementation of *LLVM LLC* was stripped of anything unrelated to the conversion of *LLVM IR* to *PTX* code, but there are still significant overhead. This is due to the *LLVM LLC*'s dependency to the rest of *LLVM*. To further reduce the impact compilation time, a step deeper could be taken in order to investigate the headers used by *LLVM LLC* and strip those of any unneeded functionality. This would be a lot of work, and would be very difficult to maintain, as updates get released for *LLVM*. If other approaches are available they would be preferable compared to this.

6.2.2 Pre-compiled Binaries

Our reimplementation of *LLVM LLC* could be pre-compiled and bundled with *YAGAL*. This would severely cut down the compilation time, since there is no longer a need for a developer to compile it every time she compiles a *YAGAL* program. This would however require a developer to link to the library binaries when compiling a *YAGAL* program.

6.2.3 What Is Implemented in YAGAL

In *YAGAL* we have not implemented pre-compiled headers, even though it is our preferred approach. We chose to prioritize implementing features higher, and as such it was not done. Before this would be ready for a developer to use, it is definitely a point that should be addressed.

7. Framework Demo and Comparison

This chapter contains the comparison between *YAGAL* and the frameworks from the related works chapter 2. This includes our comparison methodology, demo applications, and the comparison results.

7.1 Comparison Process

We want to investigate how *YAGAL* performs against the related works and how usable it is. To do this we have split the comparison in two parts; the first part covers the superficial qualities, being those that are easy to measure and compare, and the second part cover the use of *Cognitive Dimensions of Notations* to construct a vocabulary for evaluation of the usability.

The evaluation and comparison will be done based on a demo application that is implemented in each of the frameworks. First the superficial qualities are recorded and compared to an equivalent implementations in *CUDA* for the GPU and in plain *C++* for the CPU. Then the usability is evaluated. This is done for each of the implementations. The findings of the evaluations are then compared to each other in the final section of this chapter.

YAGAL does not support anonymous functions, as explained in section 6.1. As a consequence; *YAGAL* is limited in terms of what can be achieved with it compared to the related works. Because of this, we have chosen to implement a demo application, that can be implemented in *YAGAL*.

The demo application is SAXPY, as it was presented for each of the related works in chapter 2. The application calculated SAXPY with vectors of size 536870912, which is the largest possible allocation size to have twice on our testing device.

Our testing device is the following machine: Hardware:

CPU: Intel Core i7-920

GPU: Zotac Geforce GTX 1070 Mini

Software:

OS Ubuntu Server 17.10

LLVM LLVM version 7.0.0svn

C++ Compiler g++ from gcc 7.2.0

CUDA Driver nvidia-384

CUDA Compiler nvcc 8.0.61

The comparison is limited to only one of the related works, which is *Thrust*, due to our choice of platform. *C++ AMP* is dependent upon the *msvc*, which is only supported on *Windows* platforms. *SkelCL* is supported on *Unix* systems, but requires an *AMD* GPU due to *Nvidia* drivers not supporting the *C++* extension for *OpenCL*. We had issues compiling *PACXX*, as the build script did not generate one of the necessary files for linking the compiler.

7.2 General Comparison

The General part of the comparison serves as a static metric for comparison of the frameworks. It shows how each of the frameworks perform, and is used to show how *YAGAL* measures against these. This section contain the implementation of the *SAXPY* computation for the CPU, *CUDA*, *YAGAL*, and *Thrust* with their corresponding measurements. The section ends with a comparison between *YAGAL* and *Thrust* with the CPU and *CUDA* measurements as a reference point. All time measurements are done around the execution of the kernels, excluding any data transfer.

The measurements for each implementation consist of:

Performance
The execution time of the demo application.

Lines of code
A count how many lines of code necessary for the implementation of the demo application.

Size of executable
The size of the compiled executable.

7.2.1 CPU

Listing 7.1 contain the implementation of *SAXPY* executing on the CPU. The CPU implementation serve as a reference point for the overall comparison. The size of the vectors are set at line 7 by bit-shifting 1 by 29, and the constant a is set in the following line. The vectors are initialized beginning at line 10 and are filled with random values beginning at line 13. The computation is performed at line 16.

```
 1 #include <vector>
 2 #include <iostream>
 3 #include <algorithm>
 4 
 5 int main(void)
 6 {
 7     size_t N = 1 << 29;
 8     float a = 11;
 9 
10     std::vector<float> x(N);
```

```
    std::vector<float> y(N);

    std::generate(x.begin(), x.end(), rand);
    std::generate(y.begin(), y.end(), rand);

    std::transform(x.begin(), x.end(), y.begin(), x.begin(),
        ↪ [=](float x, float y)->float{return a * x + y;});

    return 0;
}
```

Listing 7.1: CPU *SAXPY* implementation.

The measurements for the CPU *SAXPY* computation in listing 7.1:

Performance: 12931,49 milliseconds

Lines of code: 7

Size of executable: 24 kilobytes

7.2.2 CUDA

Listing 7.2 contain the *CUDA* implementation of the *SAXPY*. The *CUDA* implementation serve as a reference point for the overall comparison. The kernel function for *SAXPY* is defined beginning at line 3. The size of the vectors are set at line 10 by bit-shifting 1 by 29, and the constant a is set in the following line. The vectors are initialized beginning at line 13 and filled with random data beginning at line 17. Memory for the vectors are allocated on device beginning at line 22 and the data are copied to device beginning at line 25. The *SAXPY* computation kernel is then performed at line 28 and copied back to host beginning at line 30.

```
 1 #include <iostream>
 2
 3 __global__ void kernel(int n, float a, float* x, float* y){
 4     for( int i = blockIdx.x * blockDim.x + threadIdx.x; i < n;
           ↪ i += blockDim.x * gridDim.x){
 5         x[i] = a * x[i] + y[i];
 6     }
 7 }
 8
 9 int main(void){
10     int N = 1 << 29;
11     float a = 11.0;
12
13     float *h_x, *h_y, *d_x, *d_y;
14     h_x = (float*)malloc(N*sizeof(float));
15     h_y = (float*)malloc(N*sizeof(float));
16
17     for(int i = 0; i < N; i++){
        h_x[i] = rand();
        h_y[i] = rand();
    }

    cudaMalloc(&d_x, N*sizeof(float));
    cudaMalloc(&d_y, N*sizeof(float));

    cudaMemcpy(d_x, h_x, N*sizeof(float),
        ↪ cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, h_y, N*sizeof(float),
        ↪ cudaMemcpyHostToDevice);

    kernel<<<128, 128>>>(N, a, d_x, d_y);

    cudaMemcpy(h_x, d_x, N*sizeof(float),
        ↪ cudaMemcpyDeviceToHost);
    cudaMemcpy(h_y, d_y, N*sizeof(float),
        ↪ cudaMemcpyDeviceToHost);
}
```

Listing 7.2: *CUDA SAXPY* Implementation.

The measurements for the *CUDA SAXPY* computation in listing 7.2:

Performance: 0,044 milliseconds

Lines of code: 18

Size of executable: 572 kilobytes

7.2.3 YAGAL

Listing 7.3 shows *SAXPY* implemented in *YAGAL*. The size of the `yagal::Vectors` are set at line 6 by bit-shifting 1 by 29, and the `yagal::Vectors` are filled with random data at line 12 and 13 by utilizing `std::generate`. The `yagal::Vectors` are then instantiated at line 15 and 16 based on the generated `std::vectors`.

The *SAXPY* computation is set up at line 18 by invoking a chain of functions on the `yagal::Vector` called `d_x`. The chained functions are `multiply()` with argument `a`, then an `add()` with the argument `d_y`. Finally the kernel is constructed and executed by the final function in the chain; *exec()*.

```
1  #include "yagal/vector.hpp"
2  #include <vector>
3  #include <iostream>
4
5  int main(){
6      size_t N = 1 << 29;
7      float a = 11;
8
9      std::vector<float> h_x(N);
```

```
    std::vector<float> h_y(N);

    std::generate(h_x.begin(), h_x.end(), rand);
    std::generate(h_y.begin(), h_y.end(), rand);

    yagal::Vector<float> d_x(h_x);
    yagal::Vector<float> d_y(h_y);

    d_x.multiply(a).add(d_y).exec();
}
```

Listing 7.3: *YAGAL SAXPY.*

Listing 7.4 shows how *YAGAL* can be used to generate and store *PTX* code and utilize the `exec()` function to execute it upon the given collection.

```
//.. Omitted ..//

int main(){
    //.. Omitted ..//

    auto ptx = d_x.multiply(a).add(d_y).exportPtx();

    d_x.exec(ptx, {d_y.getDevicePtrPtr()});
}
```

Listing 7.4: *YAGAL SAXPY* utilizing *PTX* generation.

The measurements for the *YAGAL SAXPY* computation in listing 7.1:

Performance with PTX generation: 57,933 ms

Performance without PTX generation: 35,011 ms

Lines of code with PTX generation: 9

Lines of code with PTX generation: 10

Size of executable: 48 megabyte

7.2.4 Thrust

Listing 7.5 shows *SAXPY* implemented in *Thrust*. The vectors used for the computation are initialized as `host_vectors` starting line 5 and they are filled with random data starting at line 9. The `host_vectors` are then copied to device starting at line 13 by initializing `device_vectors`. The *SAXPY* computation are then executed at line 17 by using *Thrust*'s `transform`, which takes iterators of the `device_vectors` and an anonymous device function. Finally the result are copied back to host at line 20.

```
1 size_t N = 1 << 29;
```

```
float a = 11;

//initialize host vectors
thrust::host_vector<float> h_x(N);
thrust::host_vector<float> h_y(N);

//fill with random data
std::generate(h_x.begin(), h_x.end(), rand);
std::generate(h_y.begin(), h_y.end(), rand);

//copy to device
thrust::device_vector<float> d_x = h_x;
thrust::device_vector<float> d_y = h_y;

//perform saxpy
thrust::transform(d_x.begin(), d_x.end(), d_y.begin(), d_x.
    ↪ begin(), [=]__device__(float x, float y){return a * x + y
    ↪ ;});

//copy results back to host vector
h_x = d_x;
```

Listing 7.5: *Thrust SAXPY* example.

The measurements for the *Thrust SAXPY* computation in listing 7.1:

Performance: 0,103 ms

Lines of code: 10

Size of executable: 860 kilobyte

7.2.5 Comparison

The results of the general part of the evaluation are compared in this section.

Performance

The Highest execution time is the CPU implementation with 12931.49 milliseconds, and the lowest execution time is *CUDA* with 0,044 milliseconds. *Thrust*, with an execution time of 0.103 milliseconds, is 2.34 times slower than *CUDA*.

The two execution times of *YAGAL*, which are 57.933 and 35,011 with and without *PTX* generation respectively, are both multiple times slower than *Thrust*, which is interesting, as it shows that there is some major difference between the *PTX* we generate, and the code that get executed through *Thrust*.

Lines of code

There are no major differences in lines of code between the implementations. The CPU implementations is 7 lines and *YAGAL* and *Thrust* is 10 lines each. The only outlier are *CUDA*, taking 18 lines of code.

CUDA is, based on the measurement, more verbose the rest. This is due to that a kernel function must be explicitly defined. Both *YAGAL* and *Thrust* handles kernel construction implicitly.

Size of executable

The smallest of the executeables are the CPU implementation at 24 kilobytes. The *CUDA* and *Thrust* executables are more even in size with the sizes of 572 kilobytes and 860 kilobyte respectively. The outlier in these measurements are *YAGAL* with a size of 48 megabytes.

The large size of *YAGAL*'s executable is a result of including *LLVM* headers, which resulted in a ten times larger executable, when it was not stripped for debug symbols. We do not consider the size of the executable a big problem, as a machine with a GPU, doing computation of a caliber where GPGPU can be useful, probably have the memory available.

7.3 Usability Evaluation

To perform usability evaluation we use *Cognitive Dimensions of Notations* as a set of points we can evaluate for the compared frameworks.

Cognitive Dimensions of Notations are a collection of usability principles that is intended to provide a vocabulary for discussion and evaluation of a given system. We base our understanding of the notations on the paper *Notational Systems – the Cognitive Dimensions of Notations framework*[20]. In this project we use it to reflect upon the API of *YAGAL* and use it to compare it to the related works.

Not all of the dimensions are equally interesting when used to evaluate an API, and we have chosen to ignore those that do not make sense when evaluating code related solutions, such as those related to icons and other visual or auditive features. Below is a list of the dimensions we use, and our interpretation of them:

Viscosity
: Describes a systems resistance to change. How easy can code be changed and if there are barriers that prevent changes.

Visibility
Describes the ability to view components easily. Whether the components are intuitive and if information is made available for the developer. Abstractions that hide information can reduce visibility.

Premature commitment
Describes constraints for the order in which a developer must complete tasks. This includes whether the developer is forced to perform tasks in a certain order, make premature decisions before information is made available, and whether the decision of a developer can be reversed or changed at a later stage of the process.

Hidden dependencies
Describes the dependencies between entities in the system where there is no transparent link for the developer. This includes whether changes in one part of the system lead to consequences in another, and whether dependencies between components are made clear to the developer.

Role-expressiveness and Consistency
Describes to what degree a developer can infer the purpose of an entity in the system and if they are consistent with the rest of the system. This includes how intuitive the API is, whether the API can be used in multiple ways, and whether a developer, that has learned one part of the API, can reliably figure other parts of the API.

Error-proneness
Describes whether the system could invite a developer to make mistakes, and if some of the constructs of the system could lead to misuse and result in errors. It also describes how the system handles and prevents developer errors.

Abstraction
Describes how the system handles and provide abstractions. It also describes the availability and support for developer defined abstractions. In some systems this could involve determining the minimum and maximum levels of abstractions achievable. A systems with too many abstractions could potentially make the system difficult to learn.

Closeness of mapping
Describes the closeness of the system representation and the domain it models. For this comparison, it describes how the system relates and maps to working directly with the programming model of GPUs.

Diffuseness
Describes the degree of verbosity of the system. This includes the display space needed to express functionality. We interpret this as the amount of code.

Hard mental operations

Describes whether the system demands cognitive resources of the developer. This includes the complexity of the notation, and whether a developer can work out the notation within their own mental processing.

7.4 YAGAL

The usability evaluation of *YAGAL* is contained in this section.

7.4.1 Usability Evaluation

In this section we consider the cognitive dimensions presented in section 7.3. For *YAGAL* we have implementation specific information that we do not have about the other frameworks, and these influence our evaluation.

Viscosity

YAGAL has low viscosity. A developer can freely chain functions together making changes in logic welcome. The computations can be performed on multiple data sets regardless of size. The user code can be used on different *CUDA* compatible devices, compiled by various compilers without changes.

Visibility

If the developer know the underlying kernel model, *YAGAL* has low visibility, but if the expectation is to modify collections with methods, it is more intuitive. There is a limited amount of constructs to learn, so it is easy to keep an overview.

Premature Commitment

YAGAL has a strict order of how tasks should be performed; a developer must have a collection in order to build kernels. A developer must have appended actions on a collection to use `exec()` or a `exportPtx()` to build kernels.

A developer can split chaining upon collections, as long as she ends the chain with an `exec()` or a `exportPtx()`.

Hidden dependencies

During allocation of the first `yagal::Vector` *YAGAL* will create a `context` in the *CUDA driver API* which will be reused in later usage of the GPU.

Role-expressiveness and Consistency

Usage of method overloading, in the cases of `exec()` and *YAGAL*s action functions such as `add()`, can lead to confusion for a developer.

`exec()` either consumes the stored actions on a collection to build and execute a kernel, or it takes a *PTX* code string and other device pointers to execute that kernel.

`add()` either creates an *add action* with another `yagal::Vector` as argument, or a single value.

Error-proneness

The use of `exec()` can cause a developer confusion, as it can be misleading as to when a collection is modified. To show an example, `vec.add(5);` has no `exec()` in the chain, and `vec` have therefore not been modified yet.

Abstraction

YAGAL has no support for anonymous functions and no support for expressing kernels that have more advanced functionality than traversing a collection. This severely limits which abstractions a developer can build with *YAGAL*.

Closeness of mapping

The execution model of *YAGAL* and low level APIs, such as *CUDA*, are very different. In *YAGAL* the developer does not build kernels directly, as they enqueue action upon collections. When managing memory, the abstractions of *YAGAL* are a thin layer above the memory allocations, allowing direct access to addresses if needed.

Diffuseness

The actions of *YAGAL* make the way a developer expresses and execute kernels compact, compared to writing a kernels, and strategies for applying them.

Hard mental operations

The API of *YAGAL* is based on actions, that can be perceived as building blocks, which gives a developer a mental model of the kernel she is executing. Due to the separation of logic into small actions, the mental operations required to express the logic have also been divided into more digestible pieces.

7.4.2 Summary

Based on the general comparison, *YAGAL* completes the *SAXPY* computation about 350 times faster than the corresponding single core CPU implementation. The *SAXPY* definition and execution can be expressed with a single expression.

YAGAL have, in terms of usability, negative and positive areas. The negatives include restricted abstractions, the compilation feedback loop, and role expressiveness.

The positive areas include the use of actions as abstractions which provide a simple mental model of the process being executed. The actions of *YAGAL* allow a developer to express and execute logic in a compact statements.

7.5 Thrust

The usability evaluation of *Thrust* is contained in this section.

7.5.1 Usability Evaluation

In this section we consider the cognitive dimensions of notations presented in section 7.3. The evaluation is based on our exposure to and experiences with *Thrust*.

Viscosity

Thrust have low viscosity, as there are no barriers to change. A developer do not have to adjust the code in multiple places to adapt to changes.

Visibility

There are generally a high degree of visibility in *Thrust*. Data management is intuitive since a developer is given explicit control in the form of `host_vectors` and `device_vectors`, and are made responsible for copying the data between host and device.

There are some constructions where specific internals are hard to infer such as the transform function, that applies a transformation on all elements of a vector. It is here difficult to determine how many threads are executed, and how different memory layers used.

Premature Commitment

Thrust enforces no order in which tasks must be completed. Anonymous functions for instance can be declared when it is needed or it can be prepared in advance. The only enforced task is that *device_vector*s must be prepared before use.

Hidden Dependencies

The first call to the `Thrust` API involves that activity on device, also starts an instance of the *CUDA Run-time API* causing a time overhead. Even though the overhead is negligible, it is important to keep in mind when timing tasks.

Role-expressiveness and Consistency

Thrust imitates *STL* constructs, and a developer can therefore have expectations as to how things should be done. This impacts role-expressiveness and consistency both positively and negatively. Positive wise a developer will feel familiar with the constructs of *Thrust*. Negative wise, it is not in every case that the constructs of `Thrust` behaves the same as in *STL*, due to them being parallelized expectations of order of operations can fail.

Error-proneness

Due to the similarity to to *STL*, a developer might have expectation as to how the constructs function, which impacts error-proneness negatively.

Due to the introduction of an extra set of vectors for each computation, the number of iterators required to perform tasks are also doubled and iterators are already a source for hard to debug problems.

Abstraction

Thrust provides various higher-order functions for general purpose algorithms. If a specific function is needed that can not be built using their abstractions, then it required by a developer to write kernel functions in *CUDA C*.

Closeness of mapping

As *Thrust* provide a higher abstraction level than *CUDA*, it hides the underlying details, such as the execution model, and memory layers. This is significant step away from the underlying model.

Diffuseness

Anonymous function can be verbose in Thrust, as they are defined in functors rather than *C++* lambdas, unless the compiler have been passed the `-expt-extended-lambda` flag. This flag will allow it to pass *C++* lambdas as shown in the *Thrust* implementation in listing 7.5.

Hard Mental Operations

Since *Thrust* manages the amount of threads and blocks, as well as memory allocations, there are fewer mental operations required when compared to working directly in *CUDA*.

7.5.2 Summary

Based on the general comparison, thrust completes the *SAXPY* compilation about 290 thousand times faster than the equivalent CPU implementation. This number seem extreme, but there are a few factors to remedy them; the CPU implementation was done at a default level of optimization, without considering how much CPU time the operating system provided it. The *SAXPY* computation can be expressed with a single line of code.

Thrust have, in terms of usability, mainly positive areas. The few negatives we have noted include *Thrust*s similarity to *STL*, which can raise false expectations for a developer, and that the expressions of *Thrust* are not as expressive as kernels written directly in *CUDA*.

7.6 Usability Comparison

This section contains the comparison between the usability of *YAGAL* and *Thrust*.

In general, *YAGAL* and *Thrust* have multiple design decisions in common, such as the use of a vector class to model memory allocations and functions that work on those. We emphasize the differences of the two frameworks below, and discuss the consequences for *YAGAL*.

The viscosity of *YAGAL* and *Thrust* are similar, with an exception in regards to changes to the compiler. If the developer of *Thrust* need to change compiler, she would have to compile the thrust component isolated from the rest of the project, as it require a specific compiler, namely *nvcc*, where with *YAGAL* it would be possible to simply compile it with the new compiler.

The visibility of *YAGAL* and *Thrust* are different in the level of abstraction, and the use of actions make it more difficult to infer the logic of the underlying system, compared to the code written in *Thrust*. Access to threads and blocks model is not possible in *Thrust*, as it is in *YAGAL*, which is interesting, as that is a lower level model that could be expected available in *Thrust*.

In regards to premature commitment, *YAGAL* require more than *Thrust*, as the strict order the code most follow to use *YAGAL*, require the developer to perform a commitment to how she solves a problem earlier than *Thrust*.

Thrust and *YAGAL* are similar in regards to hidden dependencies, as both introduce an overhead at the first activity on the GPU.

Role expressiveness is a case of the two frameworks being different, but achieving similar results. We consider both to be clear about what the different constructs do, but where *YAGAL* have possibly unexpected method overloading, *Thrust* have a *STL* inspired API, that sometimes might have surprising deviations from *STL*.

In regards to error-proneness, both *Thrust* and *YAGAL* have issues. *Thrust* uses iterators, which is a source of errors. *YAGAL* do not use iterators in order to avoid this issuei Instead *YAGAL* introduces the exec function, which must be used to do any transformation, which can be easy to forget. An argument can be made for iterators having less impact, as developers of *C++* already are used to those.

The biggest difference of the frameworks are in regards to abstraction. Where *Thrust* is limited to use a set of general purpose higher order functions, *YAGAL* does not implement such a feature. This strictly limits the abstractions that can be build around *YAGAL*.

The closeness of mapping is different between the frameworks. In *YAGAL* we introduce the action abstraction, which make it further from the underlying logic, compared to *Thrust* where the content of the lambdas are written in *CUDA C*.

The diffuseness of the both frameworks is low, as it is possible to compactly define and use kernels. Without enabling lambda support in *Thrust*, it does get more verbose and less compact.

In regards to hard mental operations, both frameworks provide abstraction over thread and block definitions, and allow the developer to focus on the logic. Whether the developer is comfortable with the *YAGAL* actions, is what decides whether it is more or less mentally straining.

The usability of the frameworks are comparable, with the major differences being in regard to the action abstraction *YAGAL* provides. It is not fair to conclude that *Thrust* have lower usability due to its strictly better support for abstraction by supporting anonymous functions. We do, however, consider actions as a good usability abstraction as it gives a simple mental model for issuing changes on an object.

8. Reflection

This chapter contains the reflections on the topics covered in this thesis. Each paragraph is initiated with a title, that describes the covered topic.

Related works

Predicting the Thesis Result

During the investigation of related works, it became clear that there was no single configuration that the frameworks adhered to. This was true in regards to requirements for run-time and compilers, but also for their way of handling anonymous functions. The most challenging technical problem of this project is to support anonymous functions, and as every related work handles this topic differently, it could be considered a warning; that this would become a big challenge. Instead of investigating further how to handle anonymous functions in general, we decided to focus the development effort on exploring the combination of run-time and compiler-dependent/compiler-independent choice combination not seen in the related works. This mean that the effort went to making *YAGAL* a compiler-independent framework running on top of the *CUDA Driver API*, which also supports a new abstraction regarding how to build and execute kernels, while experimenting with integrating *LLVM* in such a framework. This decision led to us trying to solve too many challenges simultaneously, where only a subset of the challenges was accomplished. A delimitation of the problem could have been beneficial for the project, to steer the focus.

Design

API First or Architecture First

Designing the API first, and then an architecture to fit it, worked out as expected. It may have been at the cost of how the underlying architecture is presented to the developer, but it was convenient to have a set of features needed by the API to design the architecture towards.

The Fit of the Action and Execution Abstractions

The abstraction we created to cover kernels was the concept of actions, where a kernel can be constructed by putting together a chain of commands, was a different approach to the usual kernel definition model. We were inspired to take a different approach to this, as we saw both *SkelCL* with its algorithm skeletons, and *PACXX* presenting lazy evaluation in a paper[21]. We found the action model to be simpler to comprehend, compared to the usually more explicit definition of kernels. It is not without issues, and it can be very verbose to express simple kernels when nesting calculations. The abstraction is interesting, and there might be problems where this approach can be a better fit than GPU kernel development.

The Fit of the Vector Abstraction

The abstraction we created to cover memory allocation was the *YAGAL* vector. It was designed to be deliberately different from *STL*s vector, while being compatible with it through copy constructor and casting. This resulted in a construction that controlled allocation and copies between data and device, through a single interface. This abstraction was convenient, and could be used as a base for a matrix construction in the future.

Choice of Framework Technologies

Not Developing YAGAL as a Compiler

The decision to not implement *YAGAL* as a compiler, but rather as a library, was interesting. The only related works that did this was based on *OpenCL*, which we chose to differ from, and *Thrust* which required *NVIDIA*s compiler. The problems we encountered might have been easier to solve if we could build a compiler to read the lambdas, and possibly even introduce new keywords to the language that could identify device functions, like *Thrust* does it with `__device__`. While it could solve some problems, it would also be less interesting, as it would not have served the purpose of exploring the field of compiler-independent GPGPU frameworks.

Choosing CUDA over OpenCL

The decision to base the solution on the *CUDA Driver API* was another interesting one. The related works, that are compiler-independent, are building the kernels as *OpenCL* code strings, so with *YAGAL* we attempted a new approach for a compiler-independent solution. Targeting *PTX* as the code language was done as it is executable by the *CUDA Driver API*, and *LLVM* supports it as a backend for its code generators. If *YAGAL* was targetting *OpenCL*, we would probably have progressed further in development, but it would have impacted the novelty of the project, as there already exist other compiler-independent frameworks that target *OpenCL*, those being *Bolt* and *SkelCL*. As the goals we attempt to accomplish simultaneously with *YAGAL* have been plenty, this could still be argued a better choice.

Using LLVM

The decision to use *LLVM* was inspired by *PACXX*, as they use the *PTX* back-end of *LLVM* to generate *PTX* code. As *YAGAL* is implemented as a library, we saw an opportunity in utilizing *LLVM* for code generation at run-time. This was done by including the necessary components of *LLVM* in *YAGAL*, as code that would get compiled into the final executable keeping *YAGAL* compiler independent. *LLVM* is meant for compiler construction, and the experience of using it outside of that domain was interesting, as documentation was directed at compiler construction. With *LLVM* we got the *LLVM Intermediate Representation*. It provided us with a higher level of expressiveness compared to *PTX*. This expressiveness combined with the many features *LLVM* brings in regards to code generation and optimization was a great benefit. Generally, the use of *LLVM* has been a learning experience, where we took a well established framework for one domain, namely compiler creation, and adopted it to another.

Implementation

Replacing LLC

We chose to make a re-implementation of the *LLVM* tool *LLC* as a part of *YAGAL*. *LLC* is the tool used to translate *LLVM Intermediate Representation* to the various targets *LLVM* supports, including *PTX*. We did this to make the executable, that is built with *YAGAL*, avoid interaction with other processes as much as possible due to performance and to be self contained. We did not perform performance measurements to ensure the necessity of this, which could have been interesting to see. We are, however, confident that performing the translations in memory is faster than outputting intermediate code and running an external process, that may or may not be present on the executing system, to get the same effect.

The Compilation Time

One of the problems of *YAGAL* is the long compilation time; for the *SAXPY* example it took 43 seconds, which is a long time for such a simple program. The compilation time is mainly due to the inclusion of large parts of the *LLVM* library. This could possibly have been worked around with a smarter compilation chain compared to linking everything in a single translation unit, as we did. We chose not to focus much on this problem during development, as we valued other features higher during our development time. This might be worth investigating at a later point, but there are still more pressing issues.

Anonymous Functions

Another more functionality impairing problem is the lack of anonymous functions. We consider the difficulty of providing this feature a product of the early design decisions, as we did not expect the choice of being compiler-independent without relying on *OpenCL* to hinder the implementation. This could have been avoided if we had taken these decisions after getting a better understanding of how they would impact the implementation of anonymous functions.

The exportPtx Conflict

We decided to have support for loading external *PTX* for execution, and as a natural continuity on this we also wanted to allow the user to export the *PTX* that the developer had constructed using *YAGAL*s actions. This gave some benefits, such as being able to generate kernels at one point and executing them later, and giving the developer the option to not repeatedly generate the same kernel for multiple executions of the same logic. But the benefits came at the price of divulging the inner components of the framework, and hurting the consistency of our abstractions.

Evaluation

Involving More People in Evaluation

The evaluation of *YAGAL* was done by us. This is not ideal, as we have understanding of the frameworks inner mechanism, where another person would have a less biased view on it. As we treat *YAGAL* to be an experiment, we do consider the current evaluation to be sufficient as a method of showing the current state of the system. However, for getting a fair evaluation that avoid bias, more people should be involved.

SAXPY as Example

Another critical point of the evaluation is the implemented algorithm. We chose to implement *SAXPY*, as it is within what is currently possible with *YAGAL*. Comparing *YAGAL* based on performance or any other metric against others is not fair, as the other frameworks can implement much more advanced algorithms than *YAGAL*. Other algorithms of increasing complexity should be used to get a more fair impression.

9. Conclusion

GPGPU development is not an easy task, and presents a steep learning curve for a developer getting into it. There are multiple frameworks that attempt to solve this, and they do this by providing various higher level constructs. These frameworks have different dependencies and requirements that the developer must fulfill, including compiler and run-time requirements.

In this project we perform an experimental design and implementation of *YAGAL*, with the aim of being compiler-independent with a comprehensible programming model with design decisions not covered by related work.

The project started based on the problem statement:

Can a GPGPU framework to abstract the underlying programming model be created, as a library that does not limit the developers choice of compiler, and how does it compare to other frameworks that do?

Throughout the project we have worked with the following tasks, which have been derived from the problem statement:

Create an overview of related works
: We have identified a set of frameworks that provide abstractions for GPGPU development, and analyzed their qualities.

Research framework design principles
: We have identified a set of design principles that we have followed during development.

Design the framework
: We have designed *YAGAL* with a novelty not found in the related works. *YAGAL* is the result of our choice of compilation method and used technologies, prioritizing experimentation over features.

Implement the framework
: We have implemented a major subset of the design, and provided a discussion about methods of proceeding regarding the discovered problems.

Implement demo application
: We have implemented an algorithm to show the features of *YAGAL*, and to provide material for evaluating and comparing *YAGAL* to the related works.

Evaluate the design and implementation of the framework
: We have evaluated *YAGAL* in regards to measured performance and usability, using the cognitive dimensions of notations, and compared it to the related works.

We have designed *YAGAL* to be an compiler-independent framework, with a run-time based on the *CUDA Driver API*. None of the related works that utilize *CUDA* are compiler independent, which is what inspired this combination.

We have provided abstractions for both the memory management in the form of a vector construction, and the construction of kernels in the form of actions. A developer does not have to perform manual allocation and copies on the device, our vector construction handles those tasks for her. A developer also does not have to learn the kernel abstraction of a lower level framework such as *CUDA* or *OpenCL*, as she can apply transformations on *YAGAL* vectors using the actions as building blocks to express the kernel logic.

We have utilized the *LLVM* library to provide an intermediate representation, for our generated kernels. For our run-time *PTX* code generation, we have made a single purpose re-implementation of *LLVM LLC*, which gets packaged in the final executable.

We have discovered that targeting a language that is not *OpenCL* without introducing a compiler results in anonymous functions being problematic to implement. We have proposed a set of possible solution strategies for this problem, including the expected difficulty and implications.

Finally we conclude that it is indeed possible to create a compiler-independent framework that utilizes the *CUDA Driver API*, but when high level abstractions are needed, those require significantly more work to implement compared to other designs.

10. Future Work

This chapter contains the topics that need to be addressed in the future, if the development on *YAGAL* were to continue.

The project was defined under the assumption, that somewhere there is a need for an abstraction library for GPGPU development, that does not enforce the developers choice of compiler. Before any more time should be dedicated on further developing *YAGAL*, it should be made clear whether there is a purpose of doing so.

We proposed some approaches to the anonymous function challenge presented in section 6.1. If development were to continue, it would make sense to further investigate those, how they impact the action design, and implement one of them. The most approachable proposed solution would probably be to rely on *nvrtc* to translate *CUDA C* to *PTX*, and get inspiration from the related works that target *OpenCL* on how to construct the *CUDA C* code from user constructs.

The current performance evaluation is grounded in a *SAXPY* implementation. If more constructs get implemented to enable more advanced algorithms in *YAGAL* is it important to see how the performance compares to the related works. Even as an abstraction library, the domain of GPGPU development is performance oriented.

Bibliography

[1] Andreas Steen Andersen & Christian Lundtofte Sørensen & Henrik Djernes Thomsen & Jonathan Hastrup & Morten Mandrup Hansen & Thomas Heldbjerg Bork. “A Comparative Study of Programming languages for the GPU”. Aalborg University, 2017.

[2] llvm-admin team. *The LLVM Compiler Infrastructure*. Seen 09/03/2018. URL: `http://www.llvm.org/`.

[3] NVIDIA. *Thrust's NVIDIA Page*. Seen 05/03/2018. URL: `https://developer.nvidia.com/thrust`.

[4] Microsoft. *Microsoft's C++ AMP page*. Seen 19/02/2018. URL: `https://msdn.microsoft.com/en-us/library/hh265137.aspx`.

[5] Inc. Advanced Micro Devices. *Bolt's Documentation*. Seen 19/02/2018. URL: `https://hsa-libraries.github.io/Bolt/html/index.html`.

[6] Michel Steuwer and Sergei Gorlatch. “SkelCL: Enhancing OpenCL for High-Level Programming of Multi-GPU Systems”. In: *Parallel Computing Technologies* 339.2 (2013), pp. 258 –272. ISSN: 0302-9743. DOI: `10.1007/978-3-642-39958-9`.

[7] Michael Haidl and Sergei Gorlatch. “PACXX: Towards a Unified Programming Model for Programming Accelerators Using C++14”. In: LLVM-HPC ’14 (2014), pp. 1–11. DOI: `10.1109/LLVM-HPC.2014.9`. URL: `http://dx.doi.org/10.1109/LLVM-HPC.2014.9`.

[8] Jared Hoberock and Nathan Bell. *Thrust's Overview*. Seen 20/02/2018. URL: `https://github.s3.amazonaws.com/downloads/thrust/thrust/Thrust%3A%20A%20Productivity-Oriented%20Library%20for%20CUDA.pdf?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=AKIAISTNZFOVBIJMK3TQ%2F20180220%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-Date=20180220T090430Z&X-Amz-Expires=300&X-Amz-SignedHeaders=host&X-Amz-Signature=493b9df1b251f35fd967d9ceb7c6b8df3e9d8da85`

[9] Jared Hoberock and Nathan Bell. *Thrust's Github Page*. Seen 20/02/2018. URL: `https://thrust.github.io/`.

[10] Inc. Advanced Micro Devices. *Bolt's Github Page*. Seen 19/02/2018. URL: `https://github.com/HSA-Libraries/Bolt`.

[11] Michel Steuwer. *SkelCL Website*. Seen 19/02/2018. URL: `https://skelcl.github.io/`.

[12] Khronos Group. *Github profile for PACXX*. Seen 05/03/2018. URL: `https://github.com/pacxx`.

[13] Michael Haidl et al. “Multi-stage Programming for GPUs in C++ Using PACXX”. In: *Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit*. GPGPU ’16. Barcelona, Spain: ACM, 2016, pp. 32–41. ISBN: 978-1-4503-4195-0. DOI: `10.1145/2884045.2884049`. URL: `http://doi.acm.org/10.1145/2884045.2884049`.

[14] Michael Boyer. *Memory Transfer Overhead*. seen 24/05/2018. URL: `https://www.cs.virginia.edu/~mwb7w/cuda_support/memory_transfer_overhead.html`.

[15] Jasmin Blanchette. “The Little Manual of API Design”. Trolltech, a Nokia company, June 19, 2008.

[16] ISO C++ Library Working Group. *Standard Library Guidelines.* Seen 16/03/2018. URL: `https://isocpp.org/std/library-design-guidelines`.

[17] llvm-admin team. *Github mirror of the LLVM project.* Seen 30/4/2018. URL: `https://github.com/llvm-mirror/llvm`.

[18] cppreference.com maintainers. *C++ Lambda Reference.* seen 28/5/2018. URL: `http://en.cppreference.com/w/cpp/language/lambda`.

[19] NVIDIA. *NVRTC Documentation.* Seen 31/05/2018. URL: `https://docs.nvidia.com/cuda/nvrtc/index.html`.

[20] Alan F. Blackwell and Thomas R.G. Green. *Notational Systems – the Cognitive Dimensions of Notations framework.* 31-05-2018. URL: `http://www.cl.cam.ac.uk/~afb21/publications/BlackwellGreen-CDsChapter.pdf`.

[21] Michael Haidl et al. “Towards Composable GPU Programming: Programming GPUs with Eager Actions and Lazy Views”. In: *Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores.* PMAM’17. Austin, TX, USA: ACM, 2017, pp. 58–67. ISBN: 978-1-4503-4883-6. DOI: `10.1145/3026937.3026942`. URL: `http://doi.acm.org/10.1145/3026937.3026942`.

Part I

Appendix

A. Kernel Launch Parameter Test

Code used to measure:

```
void cpuTest(){
    std::vector<float> src(1 << 29);
    std::srand(0);
    std::generate(src.begin(), src.end(), std::rand);

    auto t0 = Clock::now();
    std::transform(src.begin(), src.end(), src.begin(), [](
        ↪ float x){return x+1;});
    auto t1 = Clock::now();

    std::cout
        << std::chrono::duration_cast<std::chrono::milliseconds
            ↪ >(t1 - t0).count() << " ms"
        << " on cpu:"
        << std::endl;
}

void defaultParamterTest(){
    std::vector<float> src(1 << 29);
    std::srand(0);
    std::generate(src.begin(), src.end(), std::rand);
    yagal::Vector<float> v(src);

    auto ptx = v.add(1).exportPtx();

    for(int x = 1; x <= 1024; x *= 2){
        for(int y = 1; y <= 1024; y *= 2){
            auto t0 = Clock::now();
            v.exec(ptx, {}, {x,1,1}, {y,1,1});
            auto t1 = Clock::now();

            std::cout
              << std::chrono::duration_cast<std::chrono::
                  ↪ milliseconds>(t1 - t0).count() << " ms"
              << " with "
              << "blockDim: {" << x <<",1,1}, gridDim: {" << y
                  ↪ << ",1,1}: "
              << std::endl;
        }
    }
}
```

Output of both functions, after fixing indentation and sorting by ascending times:

```
21    ms with blockDim: {1024,1,1}, gridDim: {1024,1,1}
21    ms with blockDim: {1024,1,1}, gridDim: {512,1,1}
21    ms with blockDim: {512,1,1},  gridDim: {1024,1,1}
22    ms with blockDim: {1024,1,1}, gridDim: {256,1,1}
22    ms with blockDim: {128,1,1},  gridDim: {1024,1,1}
22    ms with blockDim: {256,1,1},  gridDim: {1024,1,1}
22    ms with blockDim: {256,1,1},  gridDim: {512,1,1}
22    ms with blockDim: {512,1,1},  gridDim: {512,1,1}
23    ms with blockDim: {1024,1,1}, gridDim: {128,1,1}
```

```
23     ms  with  blockDim:  {128,1,1},   gridDim:  {128,1,1}
23     ms  with  blockDim:  {256,1,1},   gridDim:  {64,1,1}
23     ms  with  blockDim:  {512,1,1},   gridDim:  {256,1,1}
24     ms  with  blockDim:  {1024,1,1},  gridDim:  {16,1,1}
24     ms  with  blockDim:  {1024,1,1},  gridDim:  {64,1,1}
24     ms  with  blockDim:  {128,1,1},   gridDim:  {512,1,1}
24     ms  with  blockDim:  {256,1,1},   gridDim:  {256,1,1}
24     ms  with  blockDim:  {512,1,1},   gridDim:  {128,1,1}
24     ms  with  blockDim:  {512,1,1},   gridDim:  {32,1,1}
24     ms  with  blockDim:  {64,1,1},    gridDim:  {1024,1,1}
24     ms  with  blockDim:  {64,1,1},    gridDim:  {256,1,1}
27     ms  with  blockDim:  {1024,1,1},  gridDim:  {32,1,1}
27     ms  with  blockDim:  {128,1,1},   gridDim:  {256,1,1}
27     ms  with  blockDim:  {256,1,1},   gridDim:  {128,1,1}
27     ms  with  blockDim:  {32,1,1},    gridDim:  {1024,1,1}
27     ms  with  blockDim:  {512,1,1},   gridDim:  {64,1,1}
27     ms  with  blockDim:  {64,1,1},    gridDim:  {512,1,1}
29     ms  with  blockDim:  {128,1,1},   gridDim:  {64,1,1}
29     ms  with  blockDim:  {256,1,1},   gridDim:  {32,1,1}
29     ms  with  blockDim:  {32,1,1},    gridDim:  {256,1,1}
29     ms  with  blockDim:  {512,1,1},   gridDim:  {16,1,1}
29     ms  with  blockDim:  {64,1,1},    gridDim:  {128,1,1}
30     ms  with  blockDim:  {1024,1,1},  gridDim:  {8,1,1}
31     ms  with  blockDim:  {32,1,1},    gridDim:  {512,1,1}
46     ms  with  blockDim:  {128,1,1},   gridDim:  {32,1,1}
46     ms  with  blockDim:  {32,1,1},    gridDim:  {128,1,1}
46     ms  with  blockDim:  {64,1,1},    gridDim:  {64,1,1}
47     ms  with  blockDim:  {16,1,1},    gridDim:  {1024,1,1}
47     ms  with  blockDim:  {256,1,1},   gridDim:  {16,1,1}
47     ms  with  blockDim:  {512,1,1},   gridDim:  {8,1,1}
49     ms  with  blockDim:  {1024,1,1},  gridDim:  {4,1,1}
55     ms  with  blockDim:  {16,1,1},    gridDim:  {256,1,1}
57     ms  with  blockDim:  {16,1,1},    gridDim:  {512,1,1}
77     ms  with  blockDim:  {32,1,1},    gridDim:  {64,1,1}
77     ms  with  blockDim:  {64,1,1},    gridDim:  {32,1,1}
78     ms  with  blockDim:  {128,1,1},   gridDim:  {16,1,1}
78     ms  with  blockDim:  {256,1,1},   gridDim:  {8,1,1}
80     ms  with  blockDim:  {512,1,1},   gridDim:  {4,1,1}
83     ms  with  blockDim:  {1024,1,1},  gridDim:  {2,1,1}
87     ms  with  blockDim:  {16,1,1},    gridDim:  {128,1,1}
91     ms  with  blockDim:  {8,1,1},     gridDim:  {1024,1,1}
104    ms  with  blockDim:  {8,1,1},     gridDim:  {512,1,1}
109    ms  with  blockDim:  {8,1,1},     gridDim:  {256,1,1}
132    ms  with  blockDim:  {4,1,1},     gridDim:  {1024,1,1}
141    ms  with  blockDim:  {32,1,1},    gridDim:  {32,1,1}
142    ms  with  blockDim:  {128,1,1},   gridDim:  {8,1,1}
142    ms  with  blockDim:  {64,1,1},    gridDim:  {16,1,1}
143    ms  with  blockDim:  {256,1,1},   gridDim:  {4,1,1}
144    ms  with  blockDim:  {512,1,1},   gridDim:  {2,1,1}
147    ms  with  blockDim:  {16,1,1},    gridDim:  {64,1,1}
148    ms  with  blockDim:  {1024,1,1},  gridDim:  {1,1,1}
170    ms  with  blockDim:  {4,1,1},     gridDim:  {512,1,1}
171    ms  with  blockDim:  {4,1,1},     gridDim:  {256,1,1}
171    ms  with  blockDim:  {8,1,1},     gridDim:  {128,1,1}
```

```
226   ms with blockDim: {2,1,1},    gridDim: {1024,1,1}
282   ms with blockDim: {32,1,1},   gridDim: {16,1,1}
283   ms with blockDim: {128,1,1},  gridDim: {4,1,1}
283   ms with blockDim: {16,1,1},   gridDim: {32,1,1}
283   ms with blockDim: {256,1,1},  gridDim: {2,1,1}
283   ms with blockDim: {512,1,1},  gridDim: {1,1,1}
283   ms with blockDim: {64,1,1},   gridDim: {8,1,1}
291   ms with blockDim: {2,1,1},    gridDim: {256,1,1}
291   ms with blockDim: {4,1,1},    gridDim: {128,1,1}
291   ms with blockDim: {8,1,1},    gridDim: {64,1,1}
299   ms with blockDim: {2,1,1},    gridDim: {512,1,1}
416   ms with blockDim: {1,1,1},    gridDim: {1024,1,1}
551   ms with blockDim: {1,1,1},    gridDim: {512,1,1}
553   ms with blockDim: {1,1,1},    gridDim: {256,1,1}
557   ms with blockDim: {8,1,1},    gridDim: {32,1,1}
558   ms with blockDim: {2,1,1},    gridDim: {128,1,1}
559   ms with blockDim: {16,1,1},   gridDim: {16,1,1}
559   ms with blockDim: {32,1,1},   gridDim: {8,1,1}
559   ms with blockDim: {4,1,1},    gridDim: {64,1,1}
560   ms with blockDim: {64,1,1},   gridDim: {4,1,1}
562   ms with blockDim: {128,1,1},  gridDim: {2,1,1}
565   ms with blockDim: {256,1,1},  gridDim: {1,1,1}
1095  ms with blockDim: {1,1,1},    gridDim: {128,1,1}
1098  ms with blockDim: {4,1,1},    gridDim: {32,1,1}
1099  ms with blockDim: {2,1,1},    gridDim: {64,1,1}
1100  ms with blockDim: {16,1,1},   gridDim: {8,1,1}
1100  ms with blockDim: {8,1,1},    gridDim: {16,1,1}
1105  ms with blockDim: {32,1,1},   gridDim: {4,1,1}
1111  ms with blockDim: {64,1,1},   gridDim: {2,1,1}
1112  ms with blockDim: {128,1,1},  gridDim: {1,1,1}
2151  ms with blockDim: {2,1,1},    gridDim: {32,1,1}
2167  ms with blockDim: {1,1,1},    gridDim: {64,1,1}
2167  ms with blockDim: {4,1,1},    gridDim: {16,1,1}
2179  ms with blockDim: {8,1,1},    gridDim: {8,1,1}
2193  ms with blockDim: {32,1,1},   gridDim: {2,1,1}
2200  ms with blockDim: {64,1,1},   gridDim: {1,1,1}
2214  ms with blockDim: {16,1,1},   gridDim: {4,1,1}
4258  ms with blockDim: {1,1,1},    gridDim: {32,1,1}
4294  ms with blockDim: {4,1,1},    gridDim: {8,1,1}
4324  ms with blockDim: {2,1,1},    gridDim: {16,1,1}
4366  ms with blockDim: {8,1,1},    gridDim: {4,1,1}
4436  ms with blockDim: {16,1,1},   gridDim: {2,1,1}
4436  ms with blockDim: {32,1,1},   gridDim: {1,1,1}
8514  ms with blockDim: {2,1,1},    gridDim: {8,1,1}
8685  ms with blockDim: {8,1,1},    gridDim: {2,1,1}
8698  ms with blockDim: {4,1,1},    gridDim: {4,1,1}
8847  ms with blockDim: {16,1,1},   gridDim: {1,1,1}
8901  ms with blockDim: {1,1,1},    gridDim: {16,1,1}
10134 ms on cpu
16857 ms with blockDim: {1,1,1},    gridDim: {8,1,1}
17076 ms with blockDim: {2,1,1},    gridDim: {4,1,1}
17216 ms with blockDim: {8,1,1},    gridDim: {1,1,1}
17247 ms with blockDim: {4,1,1},    gridDim: {2,1,1}
26513 ms with blockDim: {1,1,1},    gridDim: {4,1,1}
```

```
26644 ms with blockDim: {2,1,1},    gridDim: {2,1,1}
26793 ms with blockDim: {4,1,1},    gridDim: {1,1,1}
45750 ms with blockDim: {1,1,1},    gridDim: {2,1,1}
46143 ms with blockDim: {2,1,1},    gridDim: {1,1,1}
84420 ms with blockDim: {1,1,1},    gridDim: {1,1,1}
```