FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

BSc Thesis in Computer Science

Jakob Stokholm Bertelsen `<rjl577>`

Implementing a CUDA Backend for Futhark

Supervisor: Troels Henriksen

January 2019

Abstract

Futhark is a data-parallel functional programming language whose compiler is presently capable of translating to GPGPU code through the OpenCL framework. This project details the implementation of an additional backend for the Futhark compiler targeting the CUDA framework. The backend is empirically evaluated through testing with the Futhark test suite, and by a performance comparison with the existing OpenCL backend. The results show that the CUDA backend passes all tests, and that it, for the majority of benchmark programs, performs similarly to the OpenCL backend. There is, however, a number of benchmark programs for which the CUDA backend is either significantly faster or slower than the OpenCL backend, and an exact reason for this difference has not been found.

Contents

1. Introduction and Motivation

Futhark is a data parallel, purely functional programming language that comes with an optimizing compiler for generating GPGPU code through the OpenCL framework [2, 1]. Its supported output languages are C through the standard OpenCL C API, Python through the PyOpenCL library, and C# through the Cloo library. For each of these languages, the compiler supports the generation of standalone executables as well as libraries that can be linked against by larger applications.

This project, as its primary goal, covers the implementation of an additional backend to the Futhark compiler for generating GPGPU code through the CUDA framework developed by NVIDIA. There are two main reaons behind choosing to add a CUDA backend to the compiler:

1. CUDA is more widespread than OpenCL, meaning that there are more CUDA programs with which to compare the performance of Futhark programs. Comparing the performance of a Futhark program to the performance of a CUDA program gives a more accurate result if the Futhark program also uses the CUDA framework, since this ensures that any performance differences between OpenCL and CUDA do not factor in.

2. Although most CUDA devices support OpenCL, there are some who do not. The addition of a CUDA backend would thus expand the range of devices that Futhark programs can run on.

The project focuses only on outputting C code with CUDA.

An introduction to the CUDA programming model can be found in chapter 2, while chapter 3 documents the implementation of the backend. In chapter 4, the backend is tested using the Futhark test suite, and a performance comparison with the OpenCL backend is performed using the Futhark benchmark suite. Lastly, chapter 5 summarizes the project and briefly looks at possible future work.

2. The CUDA Programming Model

This chapter gives a brief introduction to programming with CUDA, and touches on parts of the framework that are relevant to understanding the most important choices made in the implementation of the CUDA backend. Fully in-depth information on programming with CUDA can be found in the official documentation [5].

CUDA follows a heterogenous programming model in which a host CPU orchestrates the execution of parallel code on one or more CUDA-enabled devices. CUDA programs are written in C/C++, and various language extensions are used to, among other things, specify if functions should be located on the host or the device, and to call device functions from the host. Since the host and the device each have their own memory, an important part of writing CUDA programs is the management of device memory (allocation/deallocation) and the copying of data between the device and host.

Device functions that are callable from the host are called *kernels*. When a kernel is called, it is executed in parallel by a number of threads, as specified by the host when the call is made. These threads are, conceptually, arranged into 3-dimensional *blocks*, which are again arranged into a 3-dimensional *grid*. Within each block, threads can share data with each other through a fast type of memory called shared memory, and accesses to memory can by made safe through the use of various synchronization functions.

Representing threads in a multidimensional manner the way CUDA does is often helpful when writing parallel code, since parallel code is often in problem domains with multidimensional aspects. Examples of such problem domains include matrix operations and physics simulations.

2.1 An Example Program

Listing 2.1 shows a simple example of a kernel, `add_kernel`, for adding together two matrices, and a corresponding wrapper function, `add`, located on the host.

```
__global__ void add_kernel(float *a, float *b, float *c, int width
    , int height)
```

```
{
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;
  if (x >= width || y >= height) { return; }
  int idx = y * width + x; // Row-major index
  c[idx] = a[idx] + b[idx];
}

#define CEIL_DIV(a,b) (((a) + (b) - 1) / (b))

__host__ float *add(float *a, float *b, int width, int height)
{
  size_t memsize = width * height * sizeof(float);
  float *d_a = NULL, *d_b = NULL, *d_c = NULL;
  float *c = (float *)malloc(memsize);

  // Allocate device memory
  cudaMalloc(&d_a, memsize);
  cudaMalloc(&d_b, memsize);
  cudaMalloc(&d_c, memsize);

  // Copy input matrices from host to device memory
  cudaMemcpy(d_a, a, memsize, cudaMemcpyHostToDevice);
  cudaMemcpy(d_b, b, memsize, cudaMemcpyHostToDevice);

  // Invoke kernel with 2-dimensional block and grid
  dim3 block(32, 32, 1);
  dim3 grid(CEIL_DIV(width, block.x), CEIL_DIV(height, block.y),
    1);
  add_kernel<<<grid, block>>>(d_a, d_b, d_c, width, height);

  // Copy output matrix from device to host memory
  cudaMemcpy(c, d_c, memsize, cudaMemcpyDeviceToHost);

  cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
  return c;
}
```

Listing 2.1: A simple matrix addition kernel with a host wrapper (error handling omitted for brevity).

The host wrapper performs the following actions:

1. `cudaMalloc` is used to allocate memory on the device.
2. `cudaMemcpy` is used to copy the input to the allocated buffers on the device.
3. The block and grid dimensions with which to launch the kernel are calculated. Specifically, we choose a fixed 2-dimensional block size of 32-by-32, and then

set the grid dimensions accordingly, in order to have enough threads for each thread to calculate a single element of the output matrix. Note that if the height or width of the matrices are not multiples of 32, more threads will be created than are needed. Choosing a fixed block dimension and setting the grid dimension accordingly in this way is a commonly used approach when writing CUDA programs.

4. The `add_kernel` kernel is launched with the calculated block and grid dimensions.

5. The result matrix is copied back to host memory with the `cudaMemcpy` function and all allocated device buffers are freed with `cudaFree`.

It can be seen that the host wrapper makes use of several non-standard C features, which are CUDA extensions:

- An execution space specifier, `___host___`, specifies that the function is to be located on the host. This is also the default when no specifier is given.
- A `dim3` vector type is used for specifying grid and block dimensions.
- A special triple-chevron syntax (`<<<...>>>`) is used to launch a kernel with the specified block and grid dimensions.

Now, looking at `add_kernel`, we can tell that this is indeed a kernel from its `___global___` execution space specifier. Had this specifier instead been `___device___`, then the function would be located on the device, but it would not be a kernel – i.e., it would not be callable from the host. Looking at the body of the kernel, it can be seen that a number of builtin variables are used. In general in CUDA device code, the following builtin variables make up important language extensions:

`threadIdx`	The index of the current thread within its block.
`blockIdx`	The index of the current block within the grid.
`blockDim`	The block dimensions.
`gridDim`	The grid dimensions.

All of these variables are of the vector type `dim3`, and their elements can be accessed using the `x`, `y`, and `z` member variables – e.g., `threadIdx.x`.

With this in mind, we can see that the kernel does the following:

1. The block and thread indices are used to to calculate the x- and y-coordinates of the element that a thread will be calculating in the output matrix. Due to the nature of matrix addition, these coordinates also specify the element that will be read from each of the two input matrices.

2. Next, a guard is in place that ensures a thread will terminate if its calculated coordinates are outside the bounds of the matrices. This handles the issue of more threads than necessary being spawned by the host when the matrix dimensions are not a multiple of 32.

3. Then, each thread calculates its index into the matrices from the x- and y-coordinates – assuming row-major order.

4. Lastly, each thread reads an element from each of the two input matrices, calculates the sum, and then stores the result in the output matrix.

The CUDA toolkit includes the tool `nvcc`, which can be used for various CUDA-related translations, including the translation of `.cu`-files to executable files [3]. If the code in listing 2.1 is stored in a file `add.cu`, and an entry point is added to this file, then this file can be translated to an executable using

```
$ nvcc -o add add.cu
```

2.2 Compilation of CUDA Programs

In the translation of a `.cu`-file, the first step taken by `nvcc` is to invoke a preprocessor that separates the device code and the host code, since these are handled differently.

The device code is translated into a fat binary that can, depending on options given to `nvcc`, contain several different images that are appropriate for different devices. For the host code, the first step taken is to expand all language extensions into valid C/C++ code. For kernel launches, for example, this means replacing the triple-chevron syntax with calls to the appropriate functions from the CUDA runtime API (see section 2.2.2). Then, the fat binary generated from the device code is embedded into the C/C++ code, and finally, `nvcc` calls a general-purpose C/C++ compiler, such as `g++`, to compile the source code into an executable file.

2.2.1 Device Code Translation in CUDA

Translating device code to something that can run on a CUDA device is a two-step process. First, the device code is translated into an intermediate assembly language, PTX [6], before it is translated into a cubin object – a type of binary object with machine code that is directly executable by a compatible CUDA device. When translating CUDA programs, it is often useful to include several different versions of the device code as both PTX code and cubin objects in the embedded fat binary. This is because of different compatibility properties of PTX code and cubin objects.

In order to identify specifications and features of different devices, NVIDIA associates a *compute capability* with every device. The compute capability of a device is written $A.B$, where A is its major version number, and B is its minor version number. NVIDIA uses this to define the compatibility of PTX and cubin objects: Cubin objects compiled for a target compute capability $A.B$ are only compatible with devices of compute capability $A.C$ where $C \geq B$. PTX, however, is fully backwards compatible, meaning that PTX compiled for a compute capability $A.B$ can be compiled to a cubin object of any compute capability equal to or greater than $A.B$.

When PTX code is included in a fat binary, it is JIT-translated into a cubin object by the CUDA driver when it is loaded at runtime. This JIT-translation adds to the startup time of the program, and is the main disadvantage of using PTX. The advantage of PTX is, of course, its backward compatibility, which guarantees that it can always be JIT-translated to a cubin object with any compute capability equal to or greater than that of the PTX code. It is worth noting, however, that when PTX is compiled to a cubin object of a higher compute capability, the binary code may not make use of all features available to that compute capability. This means that the performance may not be as good as it would have been if the cubin was compiled from PTX code of a higher compute capability. Including cubin objects has the advantage of a lower startup time due to not having to JIT-translate, but the disadvantage of having a narrower compatibility range.

As an example, in the compilation of a CUDA program stored in a file `add.cu`, the following `nvcc` command includes cubin objects for compute capabilities 3.0, 5.0, and 6.0, while also including PTX for compute capability 6.0:

```
$ nvcc -o add add.cu \
  -gencode arch=compute_30,code=sm_30 \
  -gencode arch=compute_50,code=sm_50 \
  -gencode arch=compute_60,code=[compute_60,sm_60]
```

Runtime compilation with NVRTC

Through the NVIDIA Runtime Compilation (NVRTC) library, CUDA also supports runtime compilation of device code to PTX for any compute capability [7]. This is useful if an application wants to dynamically control options for PTX code generation, if the application wishes to modify device code before compiling it, or if the device code is given to the program from an external source at runtime.

2.2.2 The Driver API and the Runtime API

In the code in listing 2.1, the functions `cudaMalloc` and `cudaMemcpy` are functions from the CUDA *runtime* API. Additionally, when `nvcc` processes host code, the triple-chevron syntax for kernel invocation is expanded to a call to the `cudaLaunchKernel` function, which is also exposed by the runtime API. Using the runtime API is a convenient way to create handwritten CUDA programs, since a lot of complexity is hidden, and since a lot of (often boilerplate-like) setup work is done implicitly.

The driver API is an alternative to the runtime API which has the same functionality, but provides a higher degree of control of how PTX code is loaded and provides functions for the management of CUDA contexts [4]. A CUDA context is a representation of a state that must be maintained between calls to different API functions, and in the runtime API, the creation and management of this context is handled implicitly. When performing runtime compilation with NVRTC, it is necessary to use functions from the driver API to load the generated PTX code.

We will now see an example of a simple CUDA program that uses the driver API.

2.3 Another Example Program

We consider another simple CUDA program. The kernel in this program takes a list of integers and turns each element into the sum of its original value and its two neighbors. The device code can be seen in listing 2.2, and the host code can be seen in listing 2.3. **Note** that this is a very contrived example, since the kernel only supports being launched with a single block (i.e., with grid dimensions $(1, 1, 1)$), which means only relatively small arrays can be handled.

This example serves to show how shared memory and synchronization can be used, and to give an impression of how the driver API is used.

```
extern "C" __global__ void neighbor_sum_kernel(int *arr)
{
  extern __shared__ int shared_mem[];
  int idx = threadIdx.x;

  // Read into shared memory
  shared_mem[idx] = arr[idx];

  // Wait for all threads to have written to shared memory
  __syncthreads();

  // Read an element and its neighbors
  int left = idx > 0 ? shared_mem[idx - 1] : 0;
  int here = shared_mem[idx];
  int right = idx < blockDim.x - 1 ? shared_mem[idx + 1] : 0;

  // Calculate sum and write back to global memory
  arr[idx] = left + here + right;
}
```

Listing 2.2: `sum.cu`, device code for a neighborhood sum kernel.

In the device code, a pointer is declared to a shared memory array of integers. This type of shared memory is called *dynamic* shared memory in CUDA, since it is the responsibility of the host to tell CUDA how much shared memory should be allocated. This allows for the shared memory buffer to change in size between kernel launches, which can be useful, since allocating too much shared memory can negatively affect performance (see section 2.3.1). It is only possible for a kernel to have one dynamic shared memory buffer. CUDA also supports static shared memory, which can be used when a kernel uses a constant amount of shared memory.

In the kernel code, each thread copies an element of the array into the shared memory buffer, and then uses the synchronization function `__syncthreads` to wait for all other threads to have written their element to shared memory. Then, each thread accesses its own element and its neighbors from shared memory, computes the sum, and stores it in the appropriate element of the global memory array [1].

Note that the kernel is annotated with `extern "C"` to prevent the name of the kernel from being mangled when it is translated to PTX.

```
#include <stdlib.h>
#include <stdio.h>
#include <cuda.h>
#include <nvrtc.h>

CUfunction neighbor_sum_kernel;
void do_stuff();

void neighbor_sum(int *buf, int len)
{
  CUdeviceptr d_buf;
  cuMemAlloc(&d_buf, len * sizeof(int));
  cuMemcpyHtoD(d_buf, buf, len * sizeof(int));

  void *args[] = { &d_buf };
  cuLaunchKernel(neighbor_sum_kernel,
      1, 1, 1,             // Grid dimensions (1, 1, 1)
      len, 1, 1,           // Block dimensions (len, 1, 1)
      len * sizeof(int),   // Size of dynamic shared memory
      NULL,                // Stream identifier
      args,                // Kernel arguments
      NULL);               // Extra options (not used)
```

[1] As a terminology side note, memory that is allocated on the device by the host with `cudaMalloc` (runtime API) or `cuMalloc` (driver API) is said to be in global memory. There are many more details to the memory model of CUDA, but they are not explored in this report.

```

  cuMemcpyDtoH(buf, d_buf, len * sizeof(int));
  cuMemFree(d_buf);
}

int main()
{
  CUcontext ctx; CUdevice dev; CUmodule module;

  cuInit(0);                          // Initialize driver API
  cuDeviceGet(&dev, 0);               // Get handle for 0th device
  cuCtxCreate(&ctx, 0, dev);          // Create a CUDA context
  cuModuleLoad(&module, "sum.ptx"); // Load PTX
  cuModuleGetFunction(                // Get function handle
      &neighbor_sum_kernel, module, "neighbor_sum_kernel");
  do_stuff();                         // Run program
  cuModuleUnload(module);             // Clean up module
  cuCtxDestroy(ctx);                  // Clean up context
  return 0;
}
```

Listing 2.3: `main.c`, host code for calling the neighborhood sum kernel (error handling omitted for brevity).

The host code in this example is a standard C file that can be compiled with any general-purpose C compiler. It contains a wrapper function for invoking the `neighbor_sum_kernel` kernel, `neighbor_sum`, and an entry point that shows (in a simplified manner) the setup work necessary for being able to invoke kernels when using the driver API. Looking at the entry point, the following things take place:

1. The driver API is initialized using `cuInit`. The parameter given to this function is unused.

2. A device handle for the 0th device is retrieved with `cuDeviceGet`. This assumes that there is at least one device available. In a normal application, a programmer would want to enumerate the available devices to select the best possible fit according to some criteria. These criteria could be compute capability, memory size, number of cores, and/or many other things.

3. A CUDA context is created using `cuCtxCreate`. Note that this context does not explicitly need to be passed to all following driver API functions, since CUDA keeps track of which context is *current* using an internal state. When creating a new context in this manner, it is automatically set as the current context internally for the calling thread. [2]

[2] Context management is a complicated feature, and many details are left out here for brevity.

4. PTX code is loaded from the `sum.ptx` file using `cuModuleLoad`. This function is also capable of loading cubin objects or fat binaries. Had this example used NVRTC instead of simply loading pre-existing PTX code, this step would be replaced with calls to the appropriate NVRTC compilation functions, which would return the PTX code as a `char` array. This array would then be passed to `cuModuleLoadData` instead.

5. A handle to the `neighbor_sum_kernel` kernel is retrieved using `cuModuleGetFunction`.

6. Some program logic is called in the `do_stuff` function (the implementation of this is not shown in listing 2.3), which is where one or more calls to the `neighbor_sum` host wrapper would occur.

7. Lastly, the module initialized from PTX is unloaded with `cuModuleUnload`, and the current CUDA context is destroyed with `cuCtxDestroy`.

In the `neighbor_sum` host wrapper that invokes the kernel, `cuMemAlloc` and `cuMemFree` are used for memory allocation and freeing, while `cuMemcpyHtoD` and `cuMemcpyDtoH` are used for copying between the host and device. The call to `cuLaunchKernel` is the driver API equivalent to the triple-chevron syntax (which is expanded to a call to the `cudaLaunchKernel` function by `nvcc`). The arguments to `cuLaunchKernel` are not surprising: grid dimensions, block dimensions, shared memory size, and kernel arguments. The stream identifier argument, which is `NULL` in this example, can be used to execute several different kernels concurrently on devices that support it. Setting this identifier to `NULL` means that a so-called legacy default stream is used. Details on streams will not be explained further here, but can be found in the official documentation.

To run the example program, the device code in listing 2.2 needs to be translated to PTX – which `nvcc` can do – and the host code in listing 2.3 needs to be compiled using a general-purpose C compiler while linking against the driver API, `libcuda`:

```
$ nvcc -o sum.ptx -ptx sum.cu
$ gcc -o sum -lcuda main.c
```

Note that the host code in listing 2.3 does not define the `do_stuff` function. A version of the host code that does define this function can be found in appendix A.

2.3.1 Brief Notes on the CUDA Hardware Model

CUDA devices can be said to be made up of a number of streaming multiprocessors (SMs) and a global memory space. Each SM also has its own memory spaces used for shared memory and registers.

When a kernel is launched, its blocks are scheduled across the SMs of the device. Within each block, threads execute instructions in lockstep in small groups called *warps*. An SM is capable of executing several blocks concurrently, but the number of blocks that can execute concurrently depends on the amount of resources used by the corresponding kernel, the amount of resources available on the SM, and on the maximum number of warps that can execute concurrently on an SM. Resources in this context refer to shared memory usage and the number of registers used. Since kernels are scheduled on SMs block-wise, and since a greater block size means a greater number of warps is used, the size of a block also influences the number of blocks that can execute concurrently on an SM.

The term *occupancy* refers to the degree to which a kernel is letting SMs launch as many warps as the hardware allows. If the resource usage of a kernel is too high, then the number of warps (and thus also the number of blocks) that can be launched is limited, resulting in bad occupancy. For example, if the block size of a kernel would result in exactly half of the available warps per SM being launched, but the block takes up just 51% of the resources of the SM, then only a single block from this kernel can be scheduled on an SM at a time. This will negatively affect performance, since the hardware is not being utilized as effectively as it could be.

To improve the occupancy of a kernel, CUDA provides a number of helper functions that can help determine an optimal block size based on the shared memory usage and register usage of a kernel. It is also possible to set limits on the maximum amount of registers a kernel can use when compiling from PTX to binary code.

2.4 CUDA in relation to OpenCL

To better understand the choices made in the implementation of the CUDA backend, we now take a brief look at some of the most important similarities between CUDA and OpenCL.

OpenCL is an open standard that specifies an API for computing on heterogenous platforms, and it has many similarities with CUDA. The programming model followed is, much like in CUDA, one in which a single host processor orchestrates the execution of parallel code on one or more so-called compute devices. An important difference is that while CUDA is only supported by NVIDIA GPUs, compute devices in OpenCL include not only other types of GPUs, but also many other types of devices used for accelerating computations. The memory model in OpenCL, also as in CUDA, is one in which the host and its compute devices have disjoint memory spaces, and the host is responsible for managing memory on the compute devices (including allocation, deallocation, and transfer).

Device code is another area in which there is a great amount of similarity between the two frameworks. OpenCL device code is written in an extended version of C called OpenCL C, which has many similarities with CUDA device code:

- Functions that are callable from the host are also known as kernels, and marking a function as a kernel is done with the `__kernel` qualifier, corresponding to the `___global___` qualifier in CUDA.

- Threads (called work items) are conceptually arranged into 3-dimensional work groups, which correspond directly to blocks in CUDA. Work groups are again arranged 3-dimensionally in the same way that blocks are arranged into a grid in CUDA. The dimensions of the work group and the number of work groups is specified by the host when a kernel is launched.

- *Local* memory, corresponding directly to shared memory in CUDA, is a part of memory that is shared between the work items of a work group, and is faster than global memory. As with dynamic shared memory in CUDA, the amount of local memory to be allocated for a kernel is specified by the host when the kernel is launched. Note that from this point on, OpenCL local memory will be referred to as simply shared memory.

  There is, however, a minor difference in the way this type of memory is passed to a kernel: in CUDA, dynamic shared memory is accessed through a single pointer declared as `extern` in the device code, while in OpenCL, shared memory is passed to a kernel as pointer arguments.

- In the same way that CUDA device code has builtin variables that allow threads to access information such as thread and block indices, OpenCL C has builtin functions that allow work items to access similar information. In fact, each of these builtin functions in OpenCL can be expressed in terms of the CUDA builtin variables, as seen in table 2.1. Note that in OpenCL, the numbers 0, 1, and 2 are used to denote the x, y, and z dimensions, respectively. For example, `get_local_id(2)` in OpenCL corresponds to `threadIdx.z` in CUDA.

OpenCL	CUDA
`get_local_id(d)`	`threadIdx.[x|y|z]`
`get_group_id(d)`	`blockIdx.[x|y|z]`
`get_local_size(d)`	`blockDim.[x|y|z]`
`get_num_groups(d)`	`gridDim.[x|y|z]`
`get_global_size(d)`	`gridDim.[x|y|z] * blockDim.[x|y|z]`
`get_global_id(d)`	`blockIdx.[x|y|z] * blockDim.[x|y|z] + threadIdx.[x|y|z]`

Table 2.1: OpenCL work item functions expressed using CUDA builtin variables

One important difference between OpenCL C and CUDA device code is that OpenCL requires that all pointers are given an address space specifier, denoting which memory space it points into. For global memory, this specifier is `__global` and for shared memory, this specifier is `__local`. In CUDA, there is no such requirement for pointers.

As for the host code, the OpenCL interface is conceptually very similar to the driver API in CUDA. Similarities include concepts like contexts, streams (command queues in OpenCL), and the dynamic loading of compiled device code from other sources. In OpenCL, the most common approach is to perform compilation of device code at runtime, which is an approach that can also be used in CUDA with the NVRTC library.

2.4.1 Choices for the CUDA Backend in Futhark

The similarity with OpenCL is one of the reasons why, in the Futhark CUDA backend, the choice was made to use the driver API with NVRTC. Increased similarity with the existing OpenCL backend eases the implementation, and possibly makes maintenance of the backends easier in the future.

The additional features provided by the driver API (compared to the runtime API) may also become of use in the future. For example, the ability to tell a Futhark-generated library to use an already existing context may be of interest to larger applications that make use of other CUDA-accelerated libraries. As for NVRTC, it is also possible that Futhark will, in the future, make use of device information obtained at runtime to tune the generation of PTX code, which could improve performance. Usage of the NVRTC library also necessitates the use of the driver API, since the runtime API cannot be used to load PTX dynamically.

3. Implementing a CUDA Backend for Futhark

This chapter documents the implementation of the CUDA backend. Since it is not relevant to the implementation of a new backend, we will not concern ourselves with the compilation steps leading up to the backend. Broadly speaking, these steps can be summed up as:

1. Parsing the source code,
2. type checking the parsed source code,
3. converting the type checked program to the `Prog` type, which is used as the internal representation of a Futhark program up until the backend, and then
4. running the `Prog` through a GPU pipeline of transformations that decide which parts of the program should be put into kernels and perform a wide variety of optimizations.

After these steps, the program being processed reaches the backend.

3.1 Backend Design of the Futhark Compiler

We first take a look at the existing backend design of the compiler, focusing especially on the OpenCL backend targeting the C language. The job of this backend is to translate the `Prog` representation to a number of strings with C code, which will, depending on command line arguments given to the compiler, either be translated to a single executable or be output as raw source files for use as a library.

The first step in this translation process is to translate the `Prog` to an imperative intermediate language called ImpCode. The most important type used for describing ImpCode, `Code`, can be seen in listing 3.1.

```
-- In module Futhark.CodeGen.ImpCode
data Code a = Skip
```

```
             | Code a :>>: Code a
             | For VName IntType Exp (Code a)
             | While Exp (Code a)
             | DeclareMem VName Space
             | DeclareScalar VName PrimType
             | DeclareArray VName Space PrimType [PrimValue]
             | Allocate VName (Count Bytes) Space
             | Free VName Space
             | Copy VName (Count Bytes) Space VName
                  (Count Bytes) Space (Count Bytes)
             | Write VName (Count Bytes) PrimType Space
                   Volatility Exp
             | SetScalar VName Exp
             | SetMem VName VName Space
             | Call [VName] Name [Arg]
             | If Exp (Code a) (Code a)
             | Op a -- Custom operation
             -- Some data constructors not shown
```

Listing 3.1: ImpCode representation

As it can be seen, ImpCode supports common imperative constructs such as for-loops, while-loops, function calls, and if-statements. It follows static single assignment form, and several memory operations are supported, including allocation, deallocation, and various copy operations. With each memory operation, memory space annotations are associated, indicating which memory space to allocate or deallocate in, or which memory spaces to copy a certain piece of memory between. The memory spaces Futhark distinguishes between are host memory, global memory, and shared memory.

It is useful for Futhark to have an imperative IL, since all of its supported output languages (C, Python, C#) are imperative. This means that a significant part of the work of translating from the functional `Prog` representation to an imperative language can be handled in a single place for all output languages.

An important feature of the imperative language is the support for a custom operation. In the translation from the `Prog` representation to ImpCode, the host-side logic and device-side logic are turned into ImpCode with different custom operations. For the host-side ImpCode, the custom operation most importantly allows for the invocation of kernels, while for the device-side ImpCode, the custom operation allows code to access, among other things, thread and group indices. The full custom operation type used with device-side ImpCode is described by the `KernelOp` type, and can be seen in listing 3.2.

```
data KernelOp = GetGroupId VName Int
              | GetLocalId VName Int
              | GetLocalSize VName Int
              | GetGlobalSize VName Int
```

```
                  | GetGlobalId VName Int
                  | GetLockstepWidth VName
                  | Atomic AtomicOp
                  | Barrier
                  | MemFence
```

Listing 3.2: Device-side custom operation type.

After translating to ImpCode, the next step is to translate the device-side ImpCode to OpenCL C code. Futhark has a generic ImpCode-to-C translation module that performs the majority of this translation, while allowing its caller to implement the translation of the custom operation. In the case of translating the device-side ImpCode with the `KernelOp` operation, this translation involves replacing, for example, the `GetGroupId` and `GetLocalId` constructors with calls to their OpenCL C equivalents, i.e., `get_group_id` and `get_local_id`. The resulting OpenCL C code is stored as a string and combined with a prelude that defines a number of mathematical operations.

After this, the final step of the translation process is to translate the host-side ImpCode to C code with OpenCL API calls. For this, the generic ImpCode-to-C translation module is used again. The module is given instructions on how to translate the custom host-side operation, which as its primary goal allows for the launching of kernels. Translation of the host-side operation involves, for kernel launches, inserting the appropriate calls to the OpenCL API. Additionally, the generic translation module is given instructions on how to translate memory operations such as allocation/deallocation on the device, and memory transfers between the host and device. This also involves inserting appropriate OpenCL API function calls.

After the host-side ImpCode has been translated, the resulting C code is combined with:

- The OpenCL C device code as a string (with the prelude containing mathematical functions prepended to it).

- A hardcoded runtime system providing a number utilities, including OpenCL context initialization, functions for performing OpenCL runtime compilation, and functions for managing memory.

- Boilerplate-like code for initialization of kernels, gathering of useful debug info during execution, and for configuration of the Futhark program prior to execution. This configuration code allows users of the Futhark program – through command line arguments if an executable is produced, and through exported functions if a library is produced – to configure, for example, the device to execute device code on, and the default dimensions to use for kernel work groups.

After the combination of these parts, the job of the backend is complete, and the combined C code is either compiled with `gcc` or output as raw files for use as a library.

3.2 Adding the CUDA Backend

The CUDA backend implementation is made with the goal of reusing as much of the existing code as possible. This reduces code duplication and preserves maintainability.

As discussed in section 2.4, there is a fundamental similarity between programming with CUDA and programming with OpenCL. This similarity means that there is no need to modify the ImpCode generation, since the custom operations on both the host and device sides are sufficient for generating CUDA code.

3.2.1 Device Code Generation

As for device code generation, the similarity means that a simple prelude can perform most of the work necessary to turn OpenCL C into valid CUDA device code, and this is the approach taken in the CUDA backend. The prelude includes definitions of functions like `get_local_id` and `get_group_id` that evaluate to their corresponding CUDA expressions, as outlined in table 2.1. For `get_local_id`, for example, the definition can be seen in listing 3.3. Since the function is only ever invoked with a numerical constant, the switch statement is evaluated and reduced to a single statement at (device code) compile-time, resulting in no additional overhead in the generated PTX code.

```
static inline int get_local_id(int d)
{
  switch (d) {
    case 0: return threadIdx.x;
    case 1: return threadIdx.y;
    case 2: return threadIdx.z;
    default: return 0;
  }
}
```

Listing 3.3: `get_local_id` for CUDA.

In addition to defining these commonly used functions, the CUDA prelude also defines:

- Fixed-size integer types such as `int8_t`. This is necessary since including `stdint.h`, the C header that would normally define these types, causes issues with NVRTC compilation. In OpenCL C, these definitions are built in.

- Wrapper functions for translating OpenCL C atomic operations to CUDA atomic operations.
- Empty macro definitions of the address space specifiers that OpenCL C requires all pointers to be annotated with. These specifiers have no direct equivalent in CUDA.
- The same mathematical operations also included in the OpenCL device code prelude.

One difference between OpenCL and CUDA device code that cannot be solved in this prelude is the difference in how shared memory is passed to kernels. As mentioned in section 2.4, in OpenCL, shared memory buffers are accessed through pointer parameters given to the kernel, while in CUDA, (dynamic) shared memory is accessed through a single `extern` pointer. This pointer can be declared globally, which is done in the aforementioned prelude, as follows:

```
extern volatile __shared__ char shared_mem[];
```

Since dynamic shared memory is accessed as a single buffer in CUDA, special care needs to be taken when a kernel in a Futhark program requires several shared memory buffers. To handle this, when generating CUDA kernels, the pointers that would normally be passed to OpenCL kernels are replaced with integers specifying offsets into the shared memory buffer. These offsets are then used with the `shared_mem` pointer to know where each buffer starts.

To sum up, CUDA device code generation is implemented as a thin layer on top of the existing OpenCL C generator, with the only difference being the inclusion of the special CUDA prelude and the change in passing of shared memory to kernels.

3.2.2 Host Code Generation

For host code generation, there is no reuse of code from the existing OpenCL backend. As in the OpenCL backend, however, the generic ImpCode-to-C translation module is used. The module is given instructions on how to translate the custom operation, which in the host-side ImpCode most importantly covers the invocation of kernels. In the CUDA backend, such an invocation is translated to a call to the `cuLaunchKernel` function from the driver API. Additionally, the module is given instructions on how to translate various memory operations. For example, memory copying between host and device, or between memory buffers on the device, are translated to the appropriate calls to `cuMemcpyDtoH`, `cuMemcpyHtoD`, and `cuMemcpy`. As in the OpenCL backend, the C code resulting from this translation is combined with

- The CUDA device code as a string, i.e., the generated kernels with the CUDA prelude prepended.
- A hardcoded runtime system providing a number of CUDA-specific utilities, including context initialization, device code compilation through NVRTC, device selection, and functions for managing memory.
- Boilerplate-like code for various initialization and configuration functionality. This code plays the same role as the boilerplate-like code generated in the OpenCL backend.

Lastly, as in the OpenCL backend, the resulting C code is either compiled with `gcc` or output as raw files for use as a library.

4. Empirical Evaluation of the CUDA Backend

Futhark has an extensive suite of test programs which can be used for testing the CUDA backend. Running the tests is done with the `futhark-test` utility as follows:

```
$ futhark-test --compiled --compiler=futhark-cuda --exclude=
   no_opencl tests/*
```

At the time of writing, there are valid Futhark programs for which the compiler cannot generate GPGPU code due to limitations. Since these compiler limitations occur prior to and during ImpCode generation, they are also present in the CUDA backend. Passing `-exclude=no_opencl` prevents running test programs that trigger these limitations.

Running the above command reports that all 1972 test cases run passed. The same command can be executed on the entire benchmark suite, which reports that all 400 test cases run passed. Thus, in terms of correctness, this indicates that the CUDA backend is not (much) worse than the OpenCL backend. We now take a look at the performance of the CUDA backend.

4.1 Performance Comparison with the OpenCL Backend

Using the `futhark-benchmark` tool with the Futhark benchmark suite, the performance of the two backends is measured. After this, the results of the benchmarks are compared using a benchmark comparison tool provided by Futhark:

```
$ futhark-bench --compiler=futhark-opencl -r 50 \
    --exclude=no_opencl --ignore-files=/lib/ \
    --json opencl.json futhark-benchmarks/*
$ futhark-bench --compiler=futhark-cuda -r 50 \
    --exclude=no_opencl --ignore-files=/lib/ \
    --json cuda.json futhark-benchmarks/*
$ cmp-bench-json.py opencl.json cuda.json
```

The output from the comparison above is a list of benchmarks with their relative speedups achieved by using the CUDA backend. The full output from this command can be seen in appendix B. Benchmarks are performed on an NVIDIA 970 GPU, and all benchmark runtimes are an average of 50 runs. The benchmark results show that about 60% of runtimes are within roughly a 10% range of their OpenCL equivalents. Of the other runtimes, slightly more than half are faster in OpenCL than in CUDA.

A useful detail about the OpenCL implementation on NVIDIA GPUs is that the OpenCL device code is in fact translated to PTX. This means that performance differences between CUDA and OpenCL device code can be understood by comparing the two PTX translations. Additionally, in OpenCL-compiled Futhark programs, if no kernels in the OpenCL device code use shared memory, then the generated PTX can be extracted and used directly with the same Futhark program when compiled with the CUDA backend.

Now, looking at one of the benchmarks that showed a significantly slower runtime in CUDA than in OpenCL, `accelerate/ray/trace.fut` with a speedup of `0.47x`, this program uses no shared memory. As mentioned, this means that the OpenCL-generated PTX could be used with the same program when compiled using the CUDA backend, and when this was attempted, the runtime was roughly the same as when running the OpenCL-compiled program. This shows that the cause of this problem – at least for the `trace.fut` program – is not in the generated host code, but somewhere in the device code. Specifically, the issue is either with the definitions in the CUDA prelude, or in the translation of the device code to PTX itself. Exactly what is to blame has not been determined, but futher analysis of the two PTX translations would likely do so.

5. Conclusion and Future Work

In terms of correctness, the implementation of the CUDA backend can certainly be considered a success because of its ability to pass all tests in the comprehensive Futhark test suite. There is, however, still plenty of possible future work to do on the backend:

- As discussed in section 2.3.1, the occupancy related CUDA functions could be used to dynamically determine the ideal block size to launch a kernel with. This approach would take into account both the resources required by the kernel, and the specifications of the device it is to be launched on, which could possibly result in improved performance.

- Adding better configuration options for CUDA contexts. Larger applications linking to a Futhark library may want to be able to tell the library to use an already existing context, particularly if the application uses other libraries that are CUDA-accelerated.

- Looking into the causes behind the difference in performance between OpenCL and CUDA in some Futhark programs. This will likely require a close look at the PTX code generated by CUDA and OpenCL.

- Adding support for other languages. Like the OpenCL backend does, it may be of interest to have the CUDA backend support more than C/C++ output. For Python, for example, the PyCUDA library could be used.

Bibliography

[1] Troels Henriksen. “Design and Implementation of the Futhark Programming Language”. PhD thesis. Universitetsparken 5, 2100 København: University of Copenhagen, Nov. 2017.

[2] Troels Henriksen et al. “Futhark: Purely Functional GPU-programming with Nested Parallelism and In-place Array Updates”. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: ACM, 2017, pp. 556–571. ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062354. URL: http://doi.acm.org/10.1145/3062341.3062354.

[3] NVIDIA. *CUDA Compiler Driver NVCC.* https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html. Accessed: 2019-01-06.

[4] NVIDIA. *CUDA Driver API.* https://docs.nvidia.com/cuda/cuda-driver-api/index.html. Accessed: 2019-01-06.

[5] NVIDIA. *CUDA Programming Guide.* https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Accessed: 2019-01-06.

[6] NVIDIA. *CUDA PTX ISA.* https://docs.nvidia.com/cuda/parallel-thread-execution/index.html. Accessed: 2019-01-06.

[7] NVIDIA. *Runtime Compilation.* https://docs.nvidia.com/cuda/nvrtc/index.html. Accessed: 2019-01-06.

A. Driver API Example Program

```c
#include <stdlib.h>
#include <stdio.h>
#include <cuda.h>
#include <nvrtc.h>

CUfunction neighbor_sum_kernel;
void do_stuff();

void neighbor_sum(int *buf, int len)
{
  CUdeviceptr d_buf;
  cuMemAlloc(&d_buf, len * sizeof(int));
  cuMemcpyHtoD(d_buf, buf, len * sizeof(int));

  void *args[] = { &d_buf };
  cuLaunchKernel(neighbor_sum_kernel,
      1, 1, 1,            // Grid dimensions (1, 1, 1)
      len, 1, 1,          // Block dimensions (len, 1, 1)
      len * sizeof(int),  // Size of dynamic shared memory
      NULL,               // Stream identifier
      args,               // Kernel arguments
      NULL);              // Extra options (not used)

  cuMemcpyDtoH(buf, d_buf, len * sizeof(int));
  cuMemFree(d_buf);
}

int main()
{
  CUcontext ctx; CUdevice dev; CUmodule module;

  cuInit(0);                           // Initialize driver API
  cuDeviceGet(&dev, 0);                // Get handle for 0th device
  cuCtxCreate(&ctx, 0, dev);           // Create a CUDA context
  cuModuleLoad(&module, "sum.ptx");    // Load PTX
  cuModuleGetFunction(                 // Get function handle
      &neighbor_sum_kernel, module, "neighbor_sum_kernel");
  do_stuff();                          // Run program
  cuModuleUnload(module);              // Clean up module
  cuCtxDestroy(ctx);                   // Clean up context
  return 0;
}

void do_stuff()
{
  int arr[] = { 100, 200, 50, 0, 300, 1 };
  int n = sizeof(arr) / sizeof(arr[0]);

  for (int i = 0; i < n; i++) { printf("%3d ", arr[i]); }
  printf("\n");

  neighbor_sum(arr, n);

  for (int i = 0; i < n; i++) { printf("%3d ", arr[i]); }
  printf("\n");
}
```

Listing A.1: `main.c`, full host code for calling the neighborhood sum kernel (error handling omitted).

B. Benchmark results: CUDA vs. OpenCL

```
Sune-ImageProc/interp.fut
  data/fake.in:                                               1.00x

Sune-ImageProc/interp_cos_plays.fut
  data/fake.in:                                               0.59x

accelerate/canny/canny.fut
  data/lena256.in:                                            1.03x
  data/lena512.in:                                            1.01x

accelerate/crystal/crystal.fut
  #0 ("200i32 30.0f32 5i32 1i32 1.0f32"):                     1.00x
  #4 ("2000i32 30.0f32 50i32 1i32 1.0f32"):                   0.98x
  #5 ("4000i32 30.0f32 50i32 1i32 1.0f32"):                   0.98x

accelerate/fft/fft.fut
  data/256x256.in:                                            0.93x
  data/128x512.in:                                            0.92x
  data/64x256.in:                                             1.09x
  data/512x512.in:                                            0.99x
  data/1024x1024.in:                                          1.00x
  data/128x128.in:                                            1.07x

accelerate/fluid/fluid.fut
  benchmarking/medium.in:                                     0.91x

accelerate/hashcat/hashcat.fut
  rockyou.dataset:                                            0.72x

accelerate/kmeans/kmeans.fut
  data/trivial.in:                                            1.22x
  data/k5_n50000.in:                                          1.09x
  data/k5_n200000.in:                                         1.03x

accelerate/mandelbrot/mandelbrot.fut
  #0 ("800i32 600i32 -0.7f32 0.0f32 3.067f32..."):            0.72x
  #1 ("1000i32 1000i32 -0.7f32 0.0f32 3.067f32...."):         0.72x
  #2 ("2000i32 2000i32 -0.7f32 0.0f32 3.067f32...."):      0.71x
  #3 ("4000i32 4000i32 -0.7f32 0.0f32 3.067f32...."):      0.71x
  #4 ("8000i32 8000i32 -0.7f32 0.0f32 3.067f32...."):      0.71x

accelerate/nbody/nbody.fut
  data/1000-bodies.in:                                     0.38x
  data/10000-bodies.in:                                    0.43x
  data/100000-bodies.in:                                   0.43x

accelerate/pagerank/pagerank.fut
  data/small.in:                                           1.24x
  data/random_medium.in:                                   1.00x

accelerate/ray/trace.fut
  #0 ("800i32 600i32 100i32 50.0f32 -100.0f32.0f32 1..."): 0.47x

accelerate/tunnel/tunnel.fut
  #0 ("10.0f32 800i32 600i32"):                            0.74x
  #1 ("10.0f32 1000i32 1000i32"):                          0.74x
  #2 ("10.0f32 2000i32 2000i32"):                          0.74x
  #3 ("10.0f32 4000i32 4000i32"):                          0.74x
  #4 ("10.0f32 8000i32 8000i32"):                          0.74x

finpar/LocVolCalib.fut
  LocVolCalib-data/small.in:                               0.76x
  LocVolCalib-data/medium.in:                              0.95x
  LocVolCalib-data/large.in:                               1.03x

finpar/OptionPricing.fut
  OptionPricing-data/small.in:                             1.32x
  OptionPricing-data/medium.in:                            0.83x
  OptionPricing-data/large.in:                             1.06x

jgf/crypt/crypt.fut
  crypt-data/medium.in:                                    1.01x

jgf/crypt/keys.fut
  crypt-data/userkey0.txt:                                 1.34x

jgf/series/series.fut
  data/10000.in:                                           1.38x
  data/100000.in:                                          1.38x
  data/1000000.in:                                         1.38x

misc/bfast/bfast.fut
  data/sahara.in:                                          1.01x

misc/heston/heston32.fut
  data/1062_quotes.in:                                     1.10x
  data/10000_quotes.in:                                     0.95x
  data/100000_quotes.in:                                    0.89x

misc/heston/heston64.fut
  data/1062_quotes.in:                                      1.02x
  data/10000_quotes.in:                                     1.16x
  data/100000_quotes.in:                                    1.24x

misc/radix_sort/radix_sort_blelloch_benchmark.fut
  data/radix_sort_10K.in:                                   1.13x
  data/radix_sort_100K.in:                                  1.04x
  data/radix_sort_1M.in:                                    1.01x

misc/radix_sort/radix_sort_large.fut
  data/radix_sort_10K.in:                                   1.47x
  data/radix_sort_100K.in:                                  1.08x
  data/radix_sort_1M.in:                                    1.01x

parboil/mri-q/mri-q.fut
  data/small.in:                                            1.07x
  data/large.in:                                            1.03x

parboil/sgemm/sgemm.fut
  data/tiny.in:                                             1.32x
  data/small.in:                                            1.00x
  data/medium.in:                                           0.84x

parboil/stencil/stencil.fut
  data/small.in:                                            1.01x
  data/default.in:                                          1.02x

parboil/tpacf/tpacf.fut
  data/small.in:                                            1.00x
  data/medium.in:                                           1.00x
  data/large.in:                                            1.00x

rodinia/backprop/backprop.fut
  data/small.in:                                            1.07x
  data/medium.in:                                           1.01x

rodinia/bfs/bfs_asympt_ok_but_slow.fut
  data/4096nodes.in:                                        1.20x
  data/512nodes_high_edge_variance.in:                      1.09x
  data/graph1MW_6.in:                                       1.03x
  data/64kn_32e-var-1-256-skew.in:                          1.05x

rodinia/bfs/bfs_filt_padded_fused.fut
  data/4096nodes.in:                                        1.10x
  data/512nodes_high_edge_variance.in:                      1.05x
  data/graph1MW_6.in:                                              1.01x
  data/64kn_32e-var-1-256-skew.in:                                 1.01x

rodinia/bfs/bfs_heuristic.fut
  data/4096nodes.in:                                               1.07x
  data/512nodes_high_edge_variance.in:                             1.11x
  data/graph1MW_6.in:                                              1.01x
  data/64kn_32e-var-1-256-skew.in:                                 1.02x

rodinia/bfs/bfs_iter_work_ok.fut
  data/4096nodes.in:                                               1.03x
  data/512nodes_high_edge_variance.in:                             0.98x
  data/graph1MW_6.in:                                              1.01x
  data/64kn_32e-var-1-256-skew.in:                                 1.02x

rodinia/cfd/cfd.fut
  data/fvcorr.domn.097K.toa:                                       0.95x
  data/fvcorr.domn.193K.toa:                                       0.95x

rodinia/hotspot/hotspot.fut
  data/64.in:                                                      0.86x
  data/512.in:                                                     0.77x
  data/1024.in:                                                    0.75x

rodinia/kmeans/kmeans.fut
  data/100.in:                                                     0.93x
  data/204800.in:                                                  0.91x
  data/kdd_cup.in:                                                 0.96x

rodinia/lavaMD/lavaMD.fut
  data/3_boxes.in:                                                 0.78x
  data/10_boxes.in:                                                0.92x

rodinia/lud/lud-clean.fut
  data/16by16.in:                                                  0.96x
  data/64.in:                                                      0.99x
  data/256.in:                                                     0.97x
  data/512.in:                                                     0.95x
  data/2048.in:                                                    0.91x

rodinia/lud/lud.fut
  data/16by16.in:                                                  1.01x
  data/64.in:                                                      0.99x
  data/256.in:                                                     0.99x
  data/512.in:                                                     0.98x
  data/2048.in:                                                    0.95x

rodinia/myocyte/myocyte.fut
  data/small.in:                                                   0.87x
  data/medium.in:                                                   1.11x

rodinia/nn/nn.fut
  data/medium.in:                                                   1.18x

rodinia/nw/nw.fut
  data/large.in:                                                    0.99x

rodinia/particlefilter/particlefilter.fut
  data/128_128_10_image_10000_particles.in:                         0.97x
  data/128_128_10_image_400000_particles.in:                        0.94x

rodinia/pathfinder/pathfinder.fut
  data/medium.in:                                                   1.00x

rodinia/srad/srad.fut
  data/image.in:                                                    1.02x
```

Listing B.1: Output from the `cmp-bench-json.py` tool when comparing OpenCL benchmark to CUDA benchmark