An OpenCL back-end for Accelerate

Martin Dybdal – dybber@dybber.dk
University of Copenhagen

6. May 2011

Abstract

Graphics processing units has recently been found useful outside their original domain. The newest graphics hardware provides massive SIMD-parallelism through hundreds of cores. In fields such as bioinformatics, computational finance and physics simulation, researchers are now using GPUs to speed up their computations and the fastest super computers are being built using GPUs instead CPUs.

Accelerate [Lee+09; Cha+11] is a language embedded in Haskell for programming GPUs in a type-safe and purely functional manner using algorithmic skeletons. Accelerate currently only targets NVIDIAs CUDA platform, and are thus only possible to execute on CUDA hardware. In this report I present a prototype implementation of a OpenCL back-end, which makes it possible to execute Accelerate code on all OpenCL enabled devices, such as GPUs, x86/x86_64 CPUs and Cell processors. The prototype OpenCL back-end does not yet perform as good as the CUDA back-end, when executed on similar hardware. I have limited my self from performance tuning and I thus expect that the performance can get much better by selecting better launch parameters for GPU code.

Together with the Accelerate back-end I present a Haskell binding for Accelerate that enables kernel compilation, scheduling and memory management of OpenCL programs.

My contributions also include a set of suggested changes to the Accelerate framework. Most importantly, I suggest that the code-generator for both the CUDA and the OpenCL back-end is modified to use quasi-quotation. This will make the development easier, and makes certain alternative implementation strategies feasible.

Contents

1 Introduction 2

2 GPGPU, CUDA and OpenCL 3

3 Accelerate 7

4 Implementation notes 10

5 Evaluation 13

6 Related Work 15

7 Future Work 16

8 Conclusion 17

9 Bibliography 18

1 Introduction

1.1 Background

Graphical processing units (GPUs) have recently been found useful outside their original field, of computer graphics. In areas such as scientific computing, bioinformatics and computational finance, GPUs have been used to speed up computations. Usage of GPUs for such general purpose problems are commonly done through low-level interfaces, with manual memory management, and a risk of encountering segmentation faults hanging over your head. Not only is programming with such low-level interfaces prone to errors, but you also have to be very careful when issuing jobs to the GPU, as mayor performance penalties may occur. Performance problems can stem from both misaligned memory accesses or too few or too many threads are issued to the GPU processors at once.

Accelerate[1] [Cha+11] is a high-level type safe Haskell framework for executing data-parallel programs on GPUs supporting NVIDIAs CUDA-architure. Using Accelerate, many types of errors are guaranteed not to occur, and the programmer can focus on other problems than aligning memory access properly.

The goal of this project has been to develop a back-end for Accelerate that targets the OpenCL framework. OpenCL is an open standard and not locked to a specific vendor. This means that Accelerate programs will be able to be executed on wider range of hardware devices. Many developers will not have direct access to a machine with NVIDIA GPUs and using OpenCL they will be able to test and execute Accelerate programs locally on their x86/x86_64 CPU. This also implies that a single program can be executed on either or both CPUs and GPUs. Not all problems are solved faster by issuing them to the GPU, by using a heterogeneous interface that supports both CPU and GPU programming, it will be easier to decide and change where the program is to be executed.

This report details my endeavour into the theory of Accelerate programming and I will explain the necessary steps done in the development the prototype OpenCL back-end. I will describe the architecture of modern GPUs and the programming interfaces CUDA and OpenCL in Section 2. In Section 3 I will describe the architecture of Accelerate. Section 4 will lay out the necessary steps for creating the OpenCL back-end. I evaluate on my implementation in Section 5.

1.2 Contributions

With this report I show that it is feasible to implement a OpenCL back-end for Accelerate. My contributions are:

- I have a extended an existing Haskell binding to the OpenCL libraries, such that it is possible for to access the necessary OpenCL functionality from Haskell programs. The existing back-end only made it possible to query certain platform information, my extensions include: Creation, compilation and invocation of GPU programs (kernels), allocation and manipulation of memory objects on the GPU, synchronization primitives and querying additional information.
- I provide a prototype implementation of a OpenCL back-end for Accelerate. This prototype does not cover all of Accelerates collective operations and is not optimized. The collective operations that remains to be implemented does not need any functionality that is not used by the collective operations which work. The prototype implementation thus demonstrates that it is feasible to execute Accelerate programs through OpenCL.
- While developing the prototype implementation of the OpenCL back-end I have identified some limitations of the Accelerate architecture, and provided suggested changes to overcome these limitations.

[1] Accelerate is available at `http://hackage.haskell.org/package/accelerate`

2 GPGPU, CUDA and OpenCL

General purpose computing on graphics processing units (GPGPU) is the technique of using GPUs (graphical processing units) for applications commonly executed on CPUs. GPUs, as their name reveals, where originally designed for computer graphics, but in the beginning of the last decade researchers found that they could speed up their computations in fields as diverse as physics simulation, image processing, computer vision, bioinformatics, computational finance, medical imaging and relational databases, by issuing tasks to the GPU [Har05].

GPUs are cost-effective for these domains, as GPUs show a high degree of data parallelism, which is a common characteristic of problems in these domains. GPUs efficiency stems from their *SIMD parallel*-architecture (single instruction, multiple data) where collections of computational cores execute the same instruction path on each their own data item. For calculations with only limited differences in control flow, this is a more efficient use of the transistors.

Before this new usage of GPUs was uncovered, the hardware-vendors put most of their work into optimizing the speed of the hardware, not precision. Thus, errors in floating point operations was common [HL04] and double precision operations was not supported. After the interest in GPGPU programming was awakened, hardware vendors have improved on these deficiencies, though double precision operations are still vastly costlier performance wise than the same operations on single precision values.

2.1 Programming interfaces

Originally, the only programming interfaces for GPUs were based on concepts such as textures and shader programs. Applications outside the graphics domain had to be encoded to fit the models of graphics programming interfaces such as OpenGL or DirectX and developers needed deep knowledge about the GPU-architecture [NVI09b].

Several frameworks has since been developed for GPGPU programming. NVIDIA developed CUDA for its line of graphics cards and AMD/ATI developed the Stream SDK (initially called "Close to metal") for their cards. To break with the situation where there was no portable interface for GPU-programming, Apple took initiative to the creation of OpenCL. OpenCL is an open standard for programming heterogeneous systems with several kinds of processors, both CPUs and GPUs. There are OpenCL implementations available for, newer NVIDIA and ATI GPUs, the x86 and x86_64 CPU-architectures and the Cell processor.

Although CUDA and OpenCL are more general than the graphics frameworks used previously, they are still low-level interfaces in today's terms. The programmer himself is responsible for administering memory usage himself and manually moving data between main memory and GPU memory. To write even simple programs it is large amounts of surrounding code are necessary. This includes: initialization of OpenCL and connected devices, compilation of GPU specific code, invocation of GPU-functions (kernels) and data-transfers to and from the GPU. To use the GPU effectively, a certain level of knowledge about the architecture of graphics processing units is also necessary. For instance, using the local caches incorrectly or misaligning memory accesses between work items can cause huge performance penalties.

2.2 OpenCL

The terminology of GPU programming is quite different from that connected with programs residing on CPUs. In this section I will describe these differences and detail how a program is executed on a GPU device. The issue of terminology is not helped by the fact that OpenCL and CUDA uses each their own set of terminology. I will not try to cover all these differences here, and will mostly stick to the terminology of OpenCL. Matt Harvey, developer of a CUDA to OpenCL translator called Swan, has summarised some of the terminology differences in a talk named "Experiences porting from CUDA to OpenCL" [Har09].

The execution of a GPU program has to be conducted by an accompanied program executing on the CPU. The traditional CPU is referred to as the

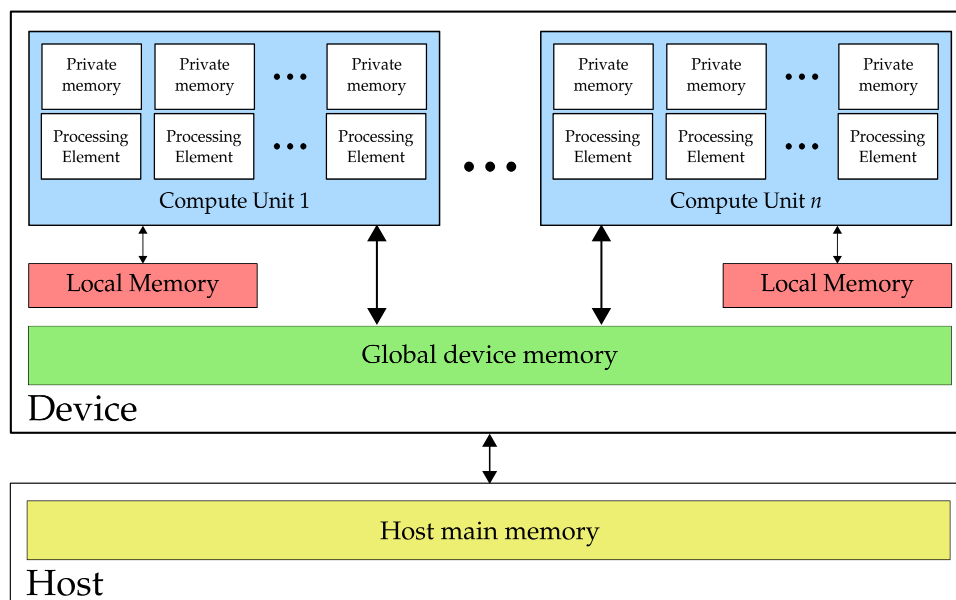

Figure 1: OpenCL GPU terminology: Host, devices, compute units and processing elements. The arrows show the possible path of data movement. By enqueuing read or write operations on the command queue, data can be moved between host memory and device memory. Kernel code is responsible for moving data between global, private and local memory.

host and the GPU is called the *device*. Other concepts such as pointers or arrays are often prefixed with either of these to specify where they reside. A *host pointer* is thus a pointer pointing to *host memory* and a *device-side* array is an array located in *device memory*. Observe, that the OpenCL is not limited to interfacing with GPUs, so the device and the host might be the same hardware unit.

An OpenCL device is divided into compute units, which are in turn divided into individual processing elements. Processing elements performs the actual computation on a GPU. How these concepts are related is shown in Figure 1. Since modern GPUs contains hundreds of processing elements, GPU programs must be written such that their work can be executed independently and in parallel on as many of these processing elements as possible. The smallest unit of work on a GPU is in OpenCL called a *work-item* and represents a single thread of execution on a processing element. These work-items are arranged into equal sized groups called *work-groups* and work-groups are arranged into an n-dimensional grid called an *NDRange*. A *work-group* is executed on a single compute unit, which provides shared memory accessible from all processing elements of that compute unit.

Programming a GPU is done by writing *kernel programs* or simply *kernels*. A kernel specifies the work done by a single work-item of the complete problem. It is the task of kernel program itself, to find the subset of the data that it has to work on. That is, all executions of the kernel receives exactly the same arguments, but the kernel can query where it is located in its work-group and NDRange and use this information to select the appropriate part of the input. When executing tasks on OpenCL devices, it is common to divide the problem into logically separate kernels scheduling them as their dependencies are met.

Job scheduling in OpenCL is done through *command queues* assigned to each device. Kernels, memory accesses and synchronization between work-groups are performed by enqueuing commands to these queues. Synchronization inside the individual work-groups are done through calls to a `barrier`-function inside the kernel. These calls can not occur inside conditionals and shall thus always synchronize all work-items inside the work-group.

OpenCL memory is segmented into *memory objects* (also called *buffers*), that stores linear arrays of bytes. These memory objects are allocated and manipulated from the host by enqueueing certain commands in the command queue of the device. When creating a memory object a reference to a host-representation is returned. To hand these as parameters to the kernels the function *clSetKernelArg* is used. It is only through this function that the host-representation of a memory object can be passed to a kernel executing on the device. The implication of this, is that memory object references must be passed as direct arguments to kernels and can not be passed together with other data inside a struct. This limitation will give us some complications that are not present when using CUDA, and is the cause of the largest difference between how the OpenCL and CUDA back-ends must be implemented, see Section 4.3 which discusses how arrays of tuples are handled.

I have extended an existing Haskell binding for OpenCL, enabling allocation of device memory, compilation and invocation of kernels and moving data between main memory and device memory. This interface will be described in Section 4.1.

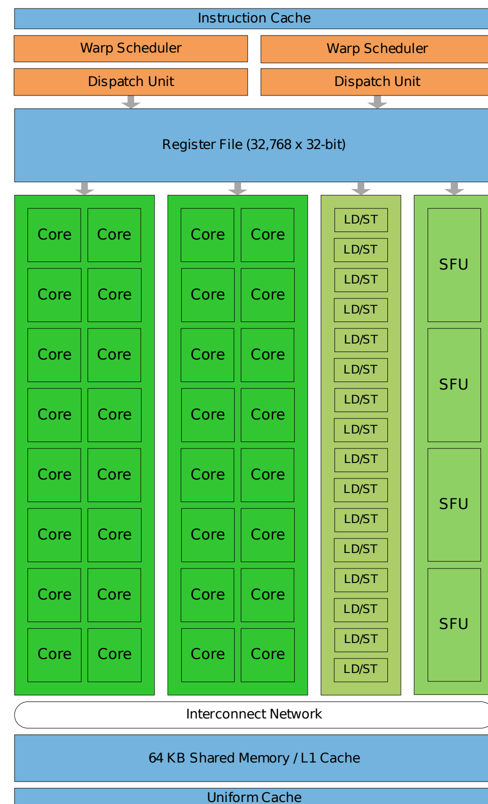

Figure 2: Fermi Streaming Multiprocessor. The illustration is borrowed from NVIDIAs Fermi whitepaper [NVI09b].

2.3 Hardware

As OpenCL is not tied to any particular vendor, there has to be a mapping between the actual hardware platforms and the OpenCL API. It is important to know the hardware particulars to get optimal performance out of the available hardware. I will focus on NVIDIAs GPGPU devices, as it is GPUs from this vendor we have available at the department.

The architecture of NVIDIAs latest line of GPGPU devices is named Fermi. The current Fermi based GPUs consists of up to 512 cores grouped into *streaming multiprocessors* (SM) of 32 cores each. The Tesla NVIDIA C2050, which I have had available for this project are equipped with 448 cores, giving 14 SMs. A streaming multiprocessor, is the OpenCL equivalent of a compute unit. Shared between all SMs is an L2 Cache (768 KB) of global memory [NVI09b]. Figure 2 shows the structure of a streaming multiprocessor in the Fermi architecture. All 32 cores share the same register file (in the top) and 64 KB of local memory (at the bottom) is used both as L1 cache and shared memory between cores. The amount of memory used as cache and as shared memory is configurable. Shared memory and registers are local memory and private memory respectively in OpenCL terminology.

When executing an OpenCL NDRange of work-groups on a Fermi architecture GPU, each work-group is partitioned into *warps* which are groups of 32 work-items. A group of this size matches directly to a single streaming multiprocessor. All work-items in a *warp* always executes exactly the same instruction, this is called *SIMD*, single instruction, multiple data. Moreover, when warps are executed by a SM it can execute two warps from the same work-group simultaneously, as each SM have

access to two schedulers and the register file can be used for both of the two warps independently. This is called *SIMT* (single instruction, multiple thread) as two threads are executed on the same processor simultaneously. Each Streaming multiprocessor can be assigned a certain number of warps (16-48 depending on the device), which it switches between executing. This is useful for hiding latency endured by memory access or data dependencies between instructions. Such dependencies can stall the processor for up to 24 cycles [NVI09a].

Optimization considerations

Determining how the NDRange of work-items are partitioned into work-groups are of importance when optimizing for fast execution. There should at least be as many work-groups as there are streaming multiprocessor, to avoid having a stalled processor. If work-groups have to wait for synchronization with other work-groups it might also be beneficial to have more than one work-group per SM. In a NVIDIA presentation on *OpenCL Optimization* [NVI09a], it is recommended to spawn at least 100 work-groups per streaming multiprocessor (for large problems), if the program should scale well on future devices.

The number of warps should also be considered, as having enough available warps can hide latency from memory transactions. Accessing global memory can stall a streaming multiprocessor for 400-600 cycles, where as local memory access only partakes a couple of cycles [NVI09a]. This is not the largest bottleneck though, as GPUs are presently connected to the host through PCI Express ports which have a throughput limit of 8-16 GB/s (depending on the generation of the device). Accessing global memory on the device can be done at 150 GB/s. The bottleneck of this has been observed by the hardware vendors and AMD has developed a line of products called *AMD Fusion*[2] which combines the CPU and the GPU into the same chip. NVIDIA has also announced plans to develop such *Accelerated Processing Units*, which is the general term for this fusion of processor technologies.

It is not always good for efficiency to have large work-groups with many warps, as each streaming multiprocessor has a limited amount of registers and local memory. All work-items currently executed on a SM shares the same register-file, and work-groups are only assigned to a SM if there are enough available registers for all of its work-items. Thus, to get optimal occupancy, work-groups must be sized such that the size of the register-file is divisible by the total amount of needed registers. A concrete example of this problem is shown in the presentation mentioned above [NVI09a]. An occupancy calculator created by NVIDIA is available as a spreadsheet on the NVIDIA website[3]. The amount of registers and local shared memory needed by a compiled kernel function can be queried through the OpenCL API.

Memory access patterns

Accessing global device memory are always done in segments of 32, 64 or 128 bytes, and these accesses must be aligned such that segments are placed in physical device memory with a segment, starting on addresses that are a multiple of the segment size [NVI10]. When a warp gets executed, the memory transaction of individual work-items are coalesced such that the needed memory can be fetched by a single memory transaction. To minimize the number of memory transactions, it is thus important to write kernels, such that nearby work-items which are scheduled in the same warp, will access memory in the same memory region. Previous architectures had strict rules for how data accesses should be distributed. If simultaneous operations on memory from a warp was out of sequence on such a device, all the operations was performed as separate memory transactions. The same would happen if accesses were not aligned correctly. With the Fermi architecture, accesses are cached such that out of order accesses can be coalesced into single transactions and misalignments can be handled as long as they do not cross 128 byte boundaries. If a segment of 32 bytes are accessed such that this access overlap two physical 32-byte

[2] `http://fusion.amd.com/`

[3] `http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls`

segments, the 64-byte segment containing both of them are loaded from global memory instead.

These considerations are explained in detail in "The CUDA C Programming Guide" by NVIDIA [NVI10] and the guide "AMD Accelerated Parallel Processing OpenCL™" [Adv11] covers the same concepts for GPU devices by AMD.

3 Accelerate

Accelerate [Lee+09; Cha+11] is an array programming language embedded in Haskell, providing a purely functional, type safe interface to GPGPU programming. It is a code-generating domain specific language (DSL), that uses Haskell primitives to structure and specify the programs written in the DSL. For instance, you can use Haskell `let`-bindings and λ-expressions either in the Haskell level or in the level of the embedded language. As an example, the following Accelerate function specifies how to compute a sum of an Accelerate array on the GPU:

```
sum :: (IsNum e, Elt e)
    => Array DIM1 e
    -> Acc (Array DIM0 e)
sum xs = fold (+) 0 (use xs)
```

This has many resemblances to the equivalent program working on Haskells built-in list type:

```
sum :: [e] -> [e] -> e
sum xs = foldl (+) 0 xs
```

I will refer back to this example in the following sections where I explain the architecture of Accelerate. For now, it should be enough to say that members of the type class `Elt` are legal Accelerate array elements, and that the types of Accelerates arrays are shape polymorphic. An array of type `Array DIM1 Float` is a vector of floating point values. The `use` function doesn't represent any computation per se, but indicates that the argument array should be moved from host-memory to GPU-memory.

3.1 The front-end

In Accelerate, as shown in the example above, you write code using Haskell structures such as *let*-bindings and λ-expressions, like `(+)` above, and higher-order functions such as `map`, `fold` and `zipWith`. These programs are not executed directly, but represents term-trees. It is the task of the front-end to build these term trees and prepare them for execution by the back-end.

Term-tree representation

Initially Accelerate constructs a term tree that is represented using *higher-order abstract syntax* (HOAS). With a first order abstract syntax, uses of λ-bound variables would normally be represented by storing an identifier both at binding and use-sites. Accelerates HOAS representation, represents these relationships between λ-binders and variable uses by Haskell λ-expressions. The implication is that we get a nameless structure where we can use the same syntax for lambda-abstractions in both the surrounding Haskell program, and in the embedded DSL.

The initial term tree generated by the `sum` function above, would turn into the following HOAS term tree:

```
sum :: PreAcc Acc (Array DIM0 Float)
sum xs = Fold add (Const 0) (Use xs)
 where
   add = \x y -> PrimAdd float
                  `PrimApp` tup2 (x, y)
   float = FloatingNumType (TypeFloat ...)

tup2 (a, b) = Tuple (NilTup `SnocTup` a
                            `SnocTup` b)
```

The important part regarding the HOAS representation, is that the addition function is represented by a Haskell function where its parameters are used exactly as in Haskell. This makes this representation easy to use for human beings, but when it has to be used in the code-generator, this representation is awkward. To deal with this, the Accelerate front-end converts the HOAS term trees into an equivalent representation, where uses of function-parameters are instead represented with *de Bruijn indices*. In this style, a lambda expression does not mention the variable it binds, instead each use of a variable is represented by a number signifying how many binders are between the actual binding site.

In the following example, the variable x is bound by the outermost binder, and thus the number of binders between the x and its binding-place is one (represented as a Peano number).

```
sum = Fold add (Const 0) (Use xs)
  where
    add = (Lam (Lam
            (Body (PrimAdd float
                       `PrimApp` tup2 (x, y)
                  ))))
    x = Var (SuccIdx ZeroIdx)
    y = Var ZeroIdx
    float = FloatingNumType (TypeFloat ..)
```

Haskell as Meta-language

When constructing the term trees, Haskell conditionals, higher order functions and other programming structures can be used to *generate* Accelerate term trees. Thus, these constructs serves as a meta-language for Accelerate. This can be useful, for instance to unroll loops executed on the device or pre-compute constant terms.

Recovering sharing

Using let-bindings and λ-expressions, programmers can reuse the result of a computation in several places without computing it twice. When a let-binding is used in an Accelerate program though, the result is a term tree where the sharing intended by the let-binding is discarded.

To recover from this problem a technique first mentioned by Andy Gill [Gil09], is used in Accelerate to find sub-expressions of the term tree that originated from the same binding site. This technique uses the GHC-specific module `System.Mem.StableName`, which makes it possible to retrieve a key for each Haskell *thunk* by the function `makeStableName :: a -> IO (StableName a)`. If two such keys are identical, they were generated from the same object, but the opposite is not usually true, because memory objects might be moved around by the garbage collector between two calls to makeStableName. To identify that two sub-expressions originates from the same let-binding, a map is constructed of all sub-expression of the term-tree, indexed by their keys. If two sub-expressions share the same *StableName*, they originated from the same binder.

As the StableName of an object can change, the problem of work duplication is only reduced and thus still occurs. Also, this can only recover values bound by let-bindings and λ-expresssions, but the λ-expressions are not turned into GPU-functions. As explained in [MM10], this is undesirable, as the same λ-expression might be issued plenty of times by a single kernel, thus leading to code explosion. The article on *Nikola* by Geoffrey Mainland also describes how λ-sharing can be recovered.

Array-representation

When introducing the `sum` example above, I briefly mentioned that Accelerates array type was parametrised over its dimensionality. This means that Arrays are shape-polymorphic. This idea is taken from the Repa array framework [Kel+10]. Regular Haskell arrays are also shape-polymorphic, the main difference is that regular Haskell arrays are indexed by tuples, whereas Accelerate uses *snoc*-lists. That is, lists where cons-operations attaches elements to the end instead of the beginning. This is useful, as arrays are represented in row-mayor order and thus increasing the second index of a rank 2 array more frequently than the first index is more cache-friendly. That is, *snoc* presents array indices in the normal order (where rows are first) while giving easy access to most frequently updated index (the column index).

Using a list-representation for indices, operations can also be made shape-polymorphic.

The Accelerate *snoc* lists are defined by:

```
data Z = Z
data tail :. head = tail :. head
```

Where `Z` serves the purpose of *nil* and represents a rank-0 dimensionality (a scalar) and the *snoc*-operator `:.` adds a dimension. With these we can understand the type of the `fold`-operator used in the `sum`-example:

```
fold :: (Shape ix, Elt a)
     => (Exp a -> Exp a -> Exp a)
     -> Exp a
     ->  Acc (Array (ix :. Int) a)
     -> Acc (Array ix a)
```

The input array has a shape that at least are rank-1, which is reduced by one dimension using the operation specified as the first argument. This operation is assumed to be associative, as it would not be possible to parallelise work if the operation was to be applied from left-to-right or right-to-left. The values of type `Exp a` are expressions in the HOAS term tree, with value-type `a`.

This representation also makes it possible to write other interesting array-operations. Accelerate for instance provides a `replicate`-function that allows an array of any dimensionality to be replicated in arbitrary new dimensions.

Host arrays

Arrays on the host are represented as unboxed arrays, where all elements are placed in a continuous memory region. This is a satisfiable way storing arrays over primitive types, as they do not have the storage overhead of boxed arrays and can be mapped directly to C-arrays compatible with the CUDA back-end. The drawback of unboxed arrays is that when evaluating one of them, all elements of the array are forced to be evaluated. As host-arrays are passed directly to a back-end, which often will need to copy the entire array to other devices, this is not a problem, we will have to fully evaluate the elements nonetheless.

Arrays over tuples, could be stored in the same way as for primitive values, but instead they are stored as pairs of arrays. This is due to alignment inefficiencies that can occur using such a representation. This problem, and how it is handled will be explained in detail in Section 4.3.

To make the array representation dynamic in terms of element types, type families are used. An introduction to type families are found in [KPS10], but they are basically a way of providing functions on the type-level and adding associated data type definitions to type class instances. Part of the type class is depicted her:

```
class ArrayElt e where
  type ArrayPtrs e
  indexArr  :: ArrayData e -> Int -> e
  ptrsOfArr :: ArrayData e -> ArrayPtrs e
  (...)
```

Elements e of this type class, are those who can be used as array elements in Accelerate arrays (the `Elt` type class mentioned previously is only sugaring some of the complexity away). The type ArrayPtrs e is then given a concrete type for each element type, making it possible to do this differentiation in representation.

3.2 The CUDA back-end

Unexecuted accelerate programs are represented as values of type `Array a => Acc a`. To obtain the array `a` they are executed by one of several back-ends. The current version of Accelerate provides a CUDA back-end and a reference back-end that executes the accelerate programs directly on the CPU. A back-end for LLVM should also be in the works.

The CUDA back-end is structured in three parts: code-generation, memory management and execution. I will detail each of these in the following sections.

Host–device data transfer

As mentioned when introducing the `sum` example program, the programmer is obligated to insert `use` around arrays that are used in Accelerate expressions. These `use` statements are used to discover which input arrays needs to be transferred to the device, even before code generation. These transfers are done simultaneously with code-generation and kernel compilation, since host–device transfers are limited by the speed of the PCI Express bus (see section 2.3). Thus hiding the latency of memory transfers somewhat.

Code-generation

For each of the collective array operations of Accelerate, a static skeleton of the kernel code is provided. The skeleton is parametrised over the struc-

```
__global__ void map (ArrOut d_out, const ArrIn0 d_in0, const Ix shape) {
    const Ix gridSize = blockDim.x * gridDim.x;
    const Ix globalId = blockDim.x * blockIdx.x + threadIdx.x;
    for (Ix idx = globalId; idx < shape; idx += gridSize) {
        set(d_out, idx, apply(get0(d_in0, idx)));
    }
}
```

Figure 3: CUDA skeleton kernel for the `map` operation.

ture of the input and output arrays. As an example, the CUDA kernel code for the map operation is shown in Figure 3. To instantiate this kernel for a particular input and output type, the types of the input array (`ArrIn0`) and output array (`ArrOut`), as well as type of individual input elements (`TyIn0`) and output elements (`TyOut`) must be defined. In addition three operation working on these types must be defined: `set`, `get0` and `apply`.

The getter and setter methods retrieves values of type `TyIn0` from the input array and writes values of type `TyOut` to the output array, respectively. The `apply` function performs the actual mapping, and thus maps inputs of type `TyIn0` to elements of type `TyOut`. That is, `apply` performs the operation of the λ-abstraction mapped over the array.

In the example, a single work-item applies the function to more than one element in the array, striding over the array until all array items are covered. This is done, as CUDA devices can limit the amount of allowed number of work groups and work items per work group, thus limiting the number of work-items. It should be noted that this limitation is only specified for CUDA, the OpenCL specification [Khr10] places no limitations on the maximum allowed work groups in an NDRange.

3.3 Execution

Execution of kernels are scheduled by traversing the term tree bottom up executing one kernel at a time. Blocking until each is kernel is finished executing. The Fermi architecture supports simultaneous execution of several kernels, and doing so can keep the GPU occupied, hiding latency due to memory transactions (see section 2.3). This could be optimized by constructing a dependency graph over the kernel outputs, such that all kernels without unmet dependencies can be scheduled simultaneously. This would complicate the back-end further, as manual synchronization points would have to be inserted.

4 Implementation notes

The OpenCL back-end I have developed for Accelerate is a modification of the CUDA back-end. Some parts, for instance the code-generator for expressions, mapped seamlessly between the two frameworks, other modifications have been more involved. In this section I will focus on the part of the transition to OpenCL that was required larger modifications of the back-end structure.

4.1 Interfacing with OpenCL

A Haskell interface to OpenCL is a necessary step towards an OpenCL back-end. My hopencl module was initially based on work by Matthew William Cox[4] and certain modules still contain some of his code. I have removed some inefficiencies from the parts of the library he had implemented, such as always requesting all information about devices and platform by many separate API calls, each API call are relatively slow and most of the information are rarely needed. With his the programmer was limited to querying information about the particular OpenCL platform and its devices, so that might have been his goal with the binding.

His version of the module was heavily limited, as it did not support creation, compilation or execution of kernel programs, allocation and manipulation of memory objects, or enqueuing synchronization barriers on the command queues. I have extended on this basic implementation, to support

[4] The version I based my module on is found at http://gitorious.org/hopencl/

these necessary operations of the OpenCL framework. It is still far from a full implementation of the OpenCL standard, though sufficient for the needs of Accelerate.

When programming OpenCL a minimum set of objects are always needed to execute code on the GPU, at least context, platform, device, command queue and kernel objects are necessary. In addition, memory objects are needed to transfer data between the host and the device. Instead of requiring the user to manually deallocate these objects, I have used functionality found in the Haskell foreign function interface, to attach finalisers to these objects, such that they are released as soon as the garbage collector determines they are out of scope. This is done by using `ForeignPtrs` and associated methods, `newForeignPtr` and `withForeignPtr`.

Memory objects are parametrized with a phantom type signifying the element type of arrays. This makes it possible to ensure that the values read or written to such arrays are of the correct types. This would also make it possible to type check kernel arguments in future versions of the library where OpenCL kernels are analyzed by the Haskell interface.

Alternatives

Other Haskell OpenCL interfaces exists. OpenCLRaw[5] is a package which contains bindings for the complete OpenCL specification, version 1.0. The bindings has though not been updated to the current version, version 1.1 of the specification. As the name suggests, OpenCLRaw, is a raw low-level import of the specification. Thus, only providing limited type safety and no automatised memory deallocation.

In January 2011, Benedict R. Gaster from AMD presented a library made internally at AMD to support OpenCL programming within Haskell [Gas11]. Their library is a low-level framework when compared to Nikola, Obsidian and Accelerate, and has most in common with my hopencl library, which it could have replaced. In addition to the functionality I have provided with hopencl, they also provide a some what higher level interface, where they use a reader-monad to avoid passing the OpenCL context, device list, command queues etc. around. Further more, they have created a OpenCL quasiquoter for writing kernels directly within Haskell. This OpenCL interface would be the ideal way to interface with OpenCL from Accelerate, but as it now they are still preparing for releasing it on Hackage, and I therefore had to develop my own library. When it is released, it might be beneficial to move to this OpenCL interface, to reduce the maintenance work required for having several OpenCL modules.

[5]http://hackage.haskell.org/package/OpenCLRaw

4.2 Rewriting CUDA kernels in OpenCL

CUDA C, the kernel language of CUDA, is actually based on C++ and not C. In contrast OpenCL C is based on the C99 standard. This means the transition from the CUDA back-end to OpenCL back-end is not as pleasant as it could be. C++ supports both ad-hoc polymorphism (function overloading) and parametric polymorphism through templates which is not found in C, and both are used in the implementation of the CUDA back-end to write index transformations that operates on indexes of any dimensionality. It is for instance necessary when looping through a 2-dimensional array using a

To tackle the problem of not having parametric polymorphism, I have created concrete instantiations of the functions for each of its possible inputs up to some maximum dimensionality. At the time of code generation, the actual dimensionality of the arrays are known. Therefore we can solve the problem of not having ad-hoc polymorphism by letting the code generator select the correct overloading of the function for us.

Another way of handling this, would be to dynamically generate the exact required function in the code-generator. This would indeed be possible, but the Accelerate code-generator is written by entering abstract syntax trees of generated code directly, which is unpleasant to work with. The amount of code written in this way should be limited as much as possible, as making even small changes later on will require a lot of work.

Using this redirection approach should not incur any performance problems, as these functions are easily inlined by the kernel compiler. An approach

where an extra dimensionality-argument are given to kernels, and the correct implementation is selected at run-time would incur more over head, as conditionals are expensive on SIMD architectures.

4.3 Handling arrays of tuples

The Accelerate implementation for CUDA handles arrays over tuples, by unzipping the arrays and storing one array for each element type of the tuple. For instance, a pair of `floats` is stored as two `float` arrays. This is done to avoid performance degradation due to misaligning the elements in memory, which could happen if they were stored as an array of structs. As explained in section 2.3, even Fermi GPUs are sensitive to such problems, if the structs crosses the wrong boundaries (e.g., a 128 byte boundaries). The structs could of course be padded to avoid alignment errors, but then the amount of unused data transferred to and from the GPU would be drastically increased.

The approach of using a struct with arrays, as the CUDA back-end, does not fit with OpenCL. With CUDA, pointers to device arrays can be given to kernels by enclosing them within a struct. Looking back at the `map` example on Figure 3, if the input to the map operation is a tuple of floats, we define `ArrIn0` as the following struct:

```
typedef struct { float* a1; float* a0; }
```

It is then possible to transfer these arrays two arrays independently, and then hand over this struct, containing the two device pointers, to the kernel. This is not possible with OpenCL, where all memory objects needed in a kernel, must be given as a direct arguments to the kernel through the `clSetKernelArg` function. There are two possible solutions:

The first possible solution is to remove the indirection of the struct, by giving each array of the struct as a separate argument of the kernel. This would indeed be possible, but then the number of kernel arguments will be variable depending on the element types. We would not be able to store the kernels as static algorithmic skeletons, where only a few surrounding definitions are variable. We would again be facing the problem of having to write large amounts of C-code through its abstract syntax, rather than its actual syntax.

The second alternative is to encode tuples of arrays within a single OpenCL array, but in a way that doesn't have performance problems caused by alignment errors. This can be done by placing two arrays after each other in the same OpenCL memory object, making sure that all arrays starts at correctly aligned indexes.

The latter approach is more prone to errors, but is feasible to implement under the circumstances, so it is what I have selected. I have not mad a full implementation of this feature, because of time constraint, but a prototype implementation shows that it is possible.

With this approach an array of 2-tuples is thus represented as an array of the first elements followed by an array of second elements, both placed inside the same OpenCL memory object. The second array must be placed such that eventual alignment restrictions are obeyed. The approach taken by OpenCL is to place all memory objects at a 256 byte boundary [NVI09a], thus by placing the second array at a similar boundary (a multiple of 256 byte distance from the beginning of the memory object), we will avoid inefficiencies due to alignment.

The selected solution has other implications though, to map host arrays to device array references, I use a table indexed by host pointers. As host arrays are represented using Haskell type families (see section 3.1), where only pointers are defined for elements with scalar elements, we are unable to use these pointers to index the table of composite arrays. This was not a problem with the CUDA back-end because each device memory object could only be accessed through the pointer of a single host array.

I therefore suggest that the `ArrayElt` type class (see Section 3.1) is extended, with methods for comparing and acquiring array hashes:

```
eqArrayData   :: ArrayData e
              -> ArrayData e
              -> Bool
```

```
hashArrayData :: ArrayData e -> Int64
```

It will then be possible to store arrays of arbitrary types in a hash table. Implementing these functions are easy for the primitive element types as the equality comparison can map directly to comparing host-pointers and the hash could be obtained by casting the host pointer to an integer. For tuples, these functions can map directly to the values obtained from calling down to the sub-arrays. These changes are made in my prototype implementation.

Currently, the size of arrays are represented several times and often passed along to device allocation and accessor functions. Instead this size could be saved as part of the arrays, I will therefore also suggest that the `ArrayElt` type class is extended with a function for obtaining the size of arrays.

```
sizeArrayData :: ArrayData e -> Int
```

This operation is also necessary for certain memory access functions, for instance when indexing into an array, it now necessary to know the length of the first array in a memory object to determine where the second array begins. Again, this function maps directly to the underlying array representation and is thus not problematic to implement.

4.4 Use of texture buffers

A similar problem to the above occurs in CUDA back-end, when accessing arrays through an indexing operator inside expression code. Such arbitrary look-ups are done by placing the indexed array in a texture buffer and letting the kernel obtain the values from there [Cha+11]. This is possible, as texture buffers does not need to be specified as a kernel argument, and the binding can thus be established in without going through this interface.

This feature of handing over arrays without going through kernel arguments, are not possible in OpenCL. Texture buffers (called *image*-objects in OpenCL) must also be passed through `clSetKernelArg`, as any other kernel argument.

The possible solutions for implementing the indexing operator is the same the above: adding a variable number of arguments to functions or always passing a static argument array which can contain arbitrary arrays that needs to be indexed.

I have not made any attempt at implementing, though it should be possible to adapt the solution from above to also handle this operation.

4.5 Code-generation using quasi-quotation

As noticed through the previous sections, programming the Accelerate code-generator through the abstract syntax heavily limits which solutions are feasible for the encountered problems. The kernel code for the Accelerate `fold`-function would require outrageous amounts of code, if it should be implemented by typing in the abstract syntax tree.

A better alternative to writing the abstract syntax directly, and still having the possibility of dynamically changing the structure of generated code (such as the number of arguments), is to use a quasi-quoter. Geoffrey Mainland has extended GHC with this functionality [Mai07] and developed C and CUDA quasi-quoters which he uses for Nikola [MM10].

As mentioned in Section 4.1, AMD have developed an OpenCL quasi-quoter which could be used. It is though not released yet and I therefore tried to develop my own, based on Mainlands C and CUDA quasi-quoters. This is not a hard task, as the only neccessary changes from C99 to OpenCL C, is the addition of `__local`, `__constant` and `__global` qualifiers. A problem occured though, as the C quasi-quoter and Accelerate use two different C libraries for their AST definitions and are thus conflicting. They use different abstract syntax definitions. It would thus require a complete rewrite, also of the expression code-generator to obtain to implement this.

Based on which of the C libraries that AMDs quasi-quoter have originated from, using it for Accelerate can be either a small or large task.

5 Evaluation

In this section I will discuss the performance on some simple Accelerate programs. As the complete Accelerate language have not been implemented, these programs are not realistic examples of what you might use Accelerate for. In particular, the lack of array of tuples and the indexing-operator, makes

it impossible to run the Black-Scholes option pricing and sparse-matrix vector multiplication examples, which were used to measure the performance of the CUDA back-end in [Cha+11].

I have executed all benchmarks on a computer equipped with a Tesla C2050, containing 448 streaming processors of 1150 Mhz each. This is hosted by two quad-core Intel Xeon® X5550 CPUs X5550 (64-bit, 2.66 GHz) with hyper-threading and 24 GB of memory.

All measurements have been done using the Haskell Criterion package [6] by Bryan O'Sullivan and each measurement is the mean over 100 repeated experiments. Each example program has been tested with input arrays ranging from having 2 million to 20 million single precision floating point values. The graphs on Figure 4 contain error bars on each measurement.

5.1 Benchmarks

I have benchmarked the OpenCL back-end against the CUDA back-end, as the goal of the OpenCL back-end is to get a back-end with equal performance of the CUDA back-end when executed on GPU hardware, but with the ability to move computations freely to other platforms than NVIDIAs CUDA architecture. I have disabled the persistent cache of compiled kernels in the CUDA back-end, to make a fair comparison against the OpenCL back-end where this functionality was not implemented. All benchmarks thus executes all stages of both the Accelerate front-end and back-ends.

I have executed benchmarks on four different programs. Three of them are implemented as single kernels, namely: `map-plus`, which adds a constant expression to all elements of an array, `fold-sum` which calculates the sum over an array, and `saxpy` which uses `zipWith` to calculate the standard function $\alpha x + y$ of the Basic Linear Algebra Subprograms (BLAS) package (the `saxpy` program is shown in Figure 5. The results of executing these three programs are shown on Figure 4(a), Figure 4(b) and Figure 4(c).

The results are not particularly satisfying, as the CUDA versions of the programs performs almost

[6] http://hackage.haskell.org/package/criterion

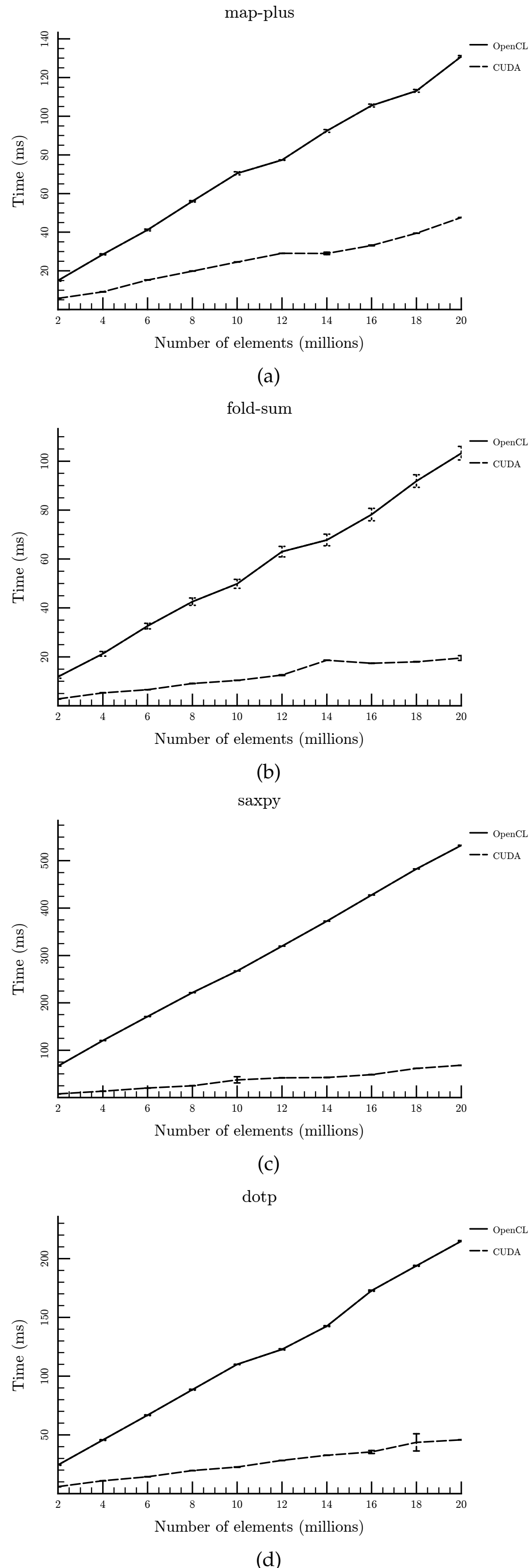

Figure 4

```
saxpy :: Float -> Vector Float
      -> Vector Float -> Acc (Vector Float)
saxpy a xs ys =
  zipWith (\x y -> constant a * x + y)
          (use xs) (use ys)

dotp :: Vector Float -> Vector Float
     -> Acc (Scalar Float)
dotp xs ys =
  let xs' = use xs
      ys' = use ys
  in fold (+) 0 (zipWith (*) xs' ys')
```

Figure 5: Accelerate programs for computing the SAXPY-operation and vector dot product.

an order of magnitude better than the OpenCL version of the same kernels. When problem size increases the difference between the OpenCL and CUDA back-ends gets even more significant.

To test how the back-end behaves when several kernels has to be invoked, I have also benchmarked a program computing the vector dot product of two single-precision floating point vectors. The program is shown in Figure 5 The results of this experiment is shown in Figure 4(d). Again the CUDA version beats the OpenCL back-end by almost an order of magnitude.

5.2 Discussion

The similarity and amount of shared code between the CUDA and OpenCL back-end limits the number things that can have caused the increased problems in running time. The code-generator and front-end of both back-ends are almost identical, these parts should therefore not cause any extra work for OpenCL back-end. The implemented kernel functions for the tested kernels are also close to identical. Especially is it worth to notice that almost no changes were necessary between the CUDA and OpenCL version of the `zipWith` kernel, even though the SAXPY program, which only executes this kernel, is the one with largest performance differences between the CUDA and OpenCL versions. The amount of host-device data-movement is also the same for both instances.

In the OpenCL back-end I have limited myself from optimizing the launch parameters for the kernels, thus I have only selected legal grid and work-group sizes and not used time on optimizing them such that to increase occupancy and decrease the amount of memory transfers between global and local memory. This might be the explanation for the observed performance difficulties.

Alternatively, there might be differences in the implementation of OpenCL and CUDA architecture, but I would not think that such large differences should be observed.

6 Related Work

There has recently been shown quite some interest in Haskell-embedded DSLs for programming GPGPUs. In addition to Accelerate, there has been released two other frameworks for GPGPU programming, namely Obsidian [Sve11] and Nikola [MM10], both targeting the CUDA platform.

6.1 Obsidian

Obsidian [Sve11] is an array manipulation language, for GPGPU programming embedded as a DSL in Haskell. The language provides a set of array operations, which can be composed to specify a CUDA kernel executed on a device. The kernels which one can write this way are limited to 1-dimensional arrays with at most 512 elements. The latter constraint are due to the maximum allowed items in a work-group, because Obsidian are always executed as a single work-group. This is a rather hard limitation, and is probably due to the complexity of code-generation when all composed operations are to be performed the same operation. The use of algorithmic skeletons in Accelerate, has the property that each of them can be written statically and be optimized independently.

To obtain maximum parallelism in Obsidian programs, it is some times necessary to add explicit synchronization steps in between composed operations, such that results are written to the compute units shared memory. This need effects that programmer, as he needs a deep understanding of how each Obsidian construct maps to generated code, as he otherwise might not get the expected speed up due to parallelism.

6.2 Nikola

Nikola [MM10] is like Obsidian an Accelerate a code-generating DSL, for the CUDA platform embedded within Haskell. Nikola programs specifies term trees much in the same style of Accelerate, where sharing introduced with Haskell let-bindings can be recovered in the same way as done in Accelerate (see Section 3.1), in addition λ-abstractions are recovered to avoid code explosion when the same λ-abstraction is issued multiple times. Many of the same constraints as for Accelerate programs, applies to Nikola programs. For instance, closures are disallowed and the only higher-order functions are those built in to the library.

The CUDA back-end of Nikola is however quite different than the one of Accelerate. Nikola programs are like Obsidian limited to a single generated kernel function. This limitation makes some programs in-expressible with Nikola. When kernels are executed on the GPU, the amount of memory needed for output has to be known before hand. Thus, the values of all parameters influencing the size of the output arrays must be known beforehand and collapsing two computations into a single kernel might be troublesome, as the output of the first operation might determine the size of the output of the next operation.

These limitations are somewhat alleviated by the fact that Nikola programs can embed CUDA kernel code directly, thus making it possible to get around the limitations of Nikola. This is made possible through a quasi-quotation library and the use of Template Haskell. Template Haskell also allows CUDA programs to be compiled on either Haskell compile-time or at runtime.

6.3 CUDA x86

A motivation for this project has been the possibility of executing the same Accelerate programs on both x86/x86_64 hardware development machines and on Fermi GPU equipped computers. This will also be possible in the future through the use of the CUDA backend, as *The Portland Group* is developing x86 implementation of CUDA [7], which should be released this year.

[7] `http://www.pgroup.com/resources/cuda-x86.htm`

7 Future Work

7.1 Implementation and optimization

The current version of the OpenCL back-end does not support all accelerate operations. Specifically are segmented fold-operations and all scan operations not handled in my back-end and OpenCL kernel skeletons remains to be implemented.

To simplify the task of developing the back-end, I have also disabled some features of the CUDA back-end before transitioning it to OpenCL. For instance, the CUDA back-end stores compiled kernels in a persistent store on the disk, such that compilation is not necessary when the same program are executed several times.

7.2 Support out of order scheduling

Modern GPGPU devices (such as Fermi based GPUs) supports simultaneous execution of several kernel functions. Currently, both the CUDA and the OpenCL back-ends executes kernels in a blocking manér. Creating a dependency graph between kernel results, would make it possible to schedule several kernels for a single device. This has some complications though, as synchronization now has to be done manually by inserting barriers to the device command queue.

7.3 Scheduling kernels for several devices

Large super computers are being built using GPU hardware, the primary compute power of both the fastest existing super computer and number three on the top500 list of super computers[8], are NVIDIA Fermi GPUs. These super computers are of course clusters of many independent machines, but several GPUs are present in each machine. It would be a worthwhile extension to support execution of Accelerate programs using several GPUs simultaneously.

The Fermi equipped machine that have been available for me is equipped with four NVIDIA Tesla C2050 GPUs, but my implementation of the OpenCL back-end only executed kernels on one of them at a time.

[8] http://www.top500.org/

7.4 Quasi-quotation based code-generation

As explained in Section 4.5, the Accelerate code-generator would be improved by building it with a quasi-quoter instead of writing abstract syntax directly. This would make the development and optimization of the Accelerate framework easier and more efficient. Many of the decisions made for the OpenCL back-end should be reconsidered if such a change was made, as their might be easier or more efficient ways by the use of quasi-quotation.

7.5 Task partitioning and device selection

OpenCL supports programming of heterogenous systems, systems with several kinds of processors, each with their own OpenCL implementation[9]. Selecting the correct device for a given problem has a noticeable impact [GO11]. Currently, the implementation does only make little effort to select the most appropriate effort and does not provide the user of the framework a way for making any cleverer selection. In the paper "A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL" [GO11] an approach for selecting devices for different parts of a computation is presented. Improving the device selection-algorithm of the presented OpenCL-back-end for Accelerate would give a substantial performance improvement. It currently just selects the device with the highest number of FLOPS but as explained in [GO11], this is not all that matters. For example, memory transfer speed is also a concern.

8 Conclusion

I have presented a new prototype back-end for the Accelerate language, that targets the OpenCL platform. This new back-end makes it possible to execute Accelerate code on all OpenCL enabled devices, such as GPUs, x86/x86_64 CPUs and Cell processors. Previously, only NVIDIA CUDA devices were targeted.

When executed on similar hardware, the new prototype OpenCL back-end does not perform as good as the CUDA back-end, when executed on similar hardware. I have limited my self from performance tuning the back-end and I thus expect that performance profiling will show that better kernel launch parameters, can be selected such that the GPU occupancy will increase and/or the amount of memory transactions decreased. Other possible explanations have been ruled out by the fact that the CUDA and OpenCL back-end shares much of their functionality and the generated kernel code are largely similar.

Together with the Accelerate back-end, I have presented a Haskell binding for Accelerate. The binding enables kernel compilation, scheduling and memory management of OpenCL programs.

My contributions also include a set of suggested changes to the Accelerate framework. Most importantly, I suggest that the code-generator for both the CUDA and the OpenCL back-end is modified to use quasi-quotation. This will make the development easier, and makes certain alternative implementation strategies feasible. I also suggest that λ-sharing in the style of Mainland et al. [MM10] is implemented.

[9]For running multiple OpenCL frameworks side-by-side on the same machine, look at the OpenCL extension called `cl_khr_icd`

9 Bibliography

[Adv11] Advanced Micro Devices. *AMD Accelerated Parallel Processing OpenCL™*. 2011.

[Cha+11] M.M.T. Chakravarty et al. "Accelerating Haskell array codes with multicore GPUs". In: *Proceedings of the sixth workshop on Declarative aspects of multicore programming*. ACM. 2011, pp. 3–14.

[Gas11] Benedict R. Gaster. *Making OpenCL™ Simple With Haskell*. 2011. URL: http://developer.amd.com/zones/OpenCLZone/publications/assets/MakingOpenCLSimplewithHaskell.pdf.

[Gil09] A. Gill. "Type-safe observable sharing in Haskell". In: *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*. ACM. 2009, pp. 117–128.

[GO11] Dominik Grewe and Michael O'Boyle. "A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL". In: *Compiler Construction*. Ed. by Jens Knoop. Vol. 6601. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin / Heidelberg, 2011. Chap. 16, pp. 286–305. ISBN: 978-3-642-19860-1.

[Har05] M. Harris. "Mapping computational concepts to GPUs". In: *ACM SIGGRAPH 2005 Courses*. ACM. 2005, 50–es.

[Har09] Matt Harvey. *Experiences porting from CUDA to OpenCL*. 2009. URL: http://www.cse.scitech.ac.uk/disco/mew20/presentations/GPU_MattHarvey.pdf.

[HL04] K. Hillesland and A. Lastra. "GPU floating-point paranoia". In: *Proceedings of GP2* (2004).

[Kel+10] G. Keller et al. "Regular, shape-polymorphic, parallel arrays in Haskell". In: *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*. ACM. 2010, pp. 261–272.

[Khr10] Khronos OpenCL Working Group. *The OpenCL Specification, Version 1.1, Revision 36*. 2010.

[KPS10] Oleg Kiselyov, Simon Peyton, and Jones Chung chieh Shan. *Fun With Type Functions, Version 3*. 2010.

[Lee+09] S. Lee et al. "GPU kernels as data-parallel array computations in Haskell". In: *Workshop on Exploiting Parallelism using GPUs and other Hardware-Assisted Methods (EPAHM 2009)*. Citeseer. 2009.

[Mai07] G. Mainland. "Why it's nice to be quoted: quasiquoting for haskell". In: *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*. ACM. 2007, pp. 73–82.

[MM10] G. Mainland and G. Morrisett. "Nikola: embedding compiled GPU functions in Haskell". In: *Proceedings of the third ACM Haskell symposium on Haskell*. ACM. 2010, pp. 67–78.

[NVI09a] NVIDIA. *OpenCL Optimization Webinar*. 2009. URL: http://developer.download.nvidia.com/CUDA/training/NVIDIA_GPU_Computing_Webinars_Best_Practises_For_OpenCL_Programming.pdf.

[NVI09b] NVIDIA. *Whitepaper: NVIDIA's Next Generation CUDA™ Compute Architecture: Fermi™*. 2009. URL: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[NVI10] NVIDIA. *The CUDA Programming Guide, Version 3.2*. 2010.

[Sve11] J. Svensson. "Obsidian: GPU Kernel Programming in Haskell". In: (2011).