DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Evaluation and Prediction of Execution Times for OpenCL-based Computations on GPGPU Systems

Alexander Pöppl

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Evaluation and Prediction of Execution Times for OpenCL-based Computations on GPGPU Systems

Evaluierung und Vorhersage von Ausführungszeiten für OpenCL-basierte Berechnungen auf GPGPU-Systemen

Author:	Alexander Pöppl
Supervisor:	Prof. Dr. Helmut Seidl
Advisor:	Dipl. Phys. Alexander Herz
Date:	August 15th, 2014

Ich versichere, dass ich diese Master's Thesis selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

I confirm that this master's thesis is my own work and I have documented all sources and material used.

München, den 12. August 2014 Alexander Pöppl

Acknowledgments

I'd like to thank everyone for the support given to me whilst writing this thesis. Thank you Neil for correcting my English. Thank you to all the people I talked to about my ideas. Thank you everyone who helped me in little ways.

Abstract

This thesis presents a run time prediction model for domain iterations on GPUs. The model includes predictions about transferring data to and from the compute device, the influence of the size of the work-group and kernel base costs. On top of that, a cost model is established for basic operations and memory accesses, taking into account effects of having multiple basic operations or memory accesses in one kernel, and the properties of different kinds of memory access classes. The thesis also describes the benchmark suite developed to take the measurements the model is based on. It also explains the additions made to the funkyIMP compiler in order to facilitate the execution of code on the GPU. The thesis concludes by giving an evaluation of the predictions made with the help of the model.

In dieser Arbeit wird eine Modell zur Vorhersage von Ausführungszeiten für Domain Iterations auf GPUs präsentiert. Dieses Modell beinhaltet Vorhersagen über den Transfer von Daten zu und von der Grafikkarte, den Einfluss der Größe der Work-Group, und den Grundkosten bei der Ausführung von Kernels auf der GPU. Dazu wird ein Kostenmodell für Rechenoperationen und Speicherzugriffe etabliert, das auch die Effekte von mehreren Rechenoperationen oder Speicherzugriffen innerhalb eines Kernels berücksichtigt. Außerdem werden Speicherzugriffe in verschiedene Klassen eingeteilt und separat bewertet. In der Arbeit wird auch die Benchmark Suite beschrieben, die im Laufe der Masterarbeit entstanden ist, um die Messungen die dem Modell zu Grunde liegen vorzunehmen. Weiterhin werden die Erweiterungen beschrieben, die am funkyIMP Compiler vorgenommen wurden, um die Ausführung von Code auf der GPU zu ermöglichen. Abschließend wird eine Bewertung der Vorhersagen, die mit dem Modell erstellt wurden durchgeführt.

Contents

Outline of the Thesis

Part I: Introduction and Theory

CHAPTER 1: INTRODUCTION

This chapter presents a general introduction and motivation to the topic of parallel and general purpose GPU computing.

CHAPTER 2: THEORY

This chapter introduces the relevant technologies that are used in the implementation, namely OpenCL and FunkyIMP.

Part II: Methodology

CHAPTER 3: EXTENSIONS TO THE FUNKYIMP COMPILER

This chapter presents the extensions that were made to the funkyIMP compiler in order to accommodate the automated compilation of code on the GPU.

CHAPTER 4: THE BENCHMARK SUITE

This chapter presents the benchmark suite that was written to create a model of the runtime of the individual operations in OpenCL.

Part III: Runtime Model and Results

CHAPTER 5: A RUNTIME PREDICTION MODEL

This chapter introduces presents the actual runtime model for OpenCL operations.

CHAPTER 6: RESULTS AND EVALUATION

In this chapter, the results achieved in the course of this thesis are shown. The quality of the predictions made utilizing the runtime prediction model is evaluated on a quantitative scale.

Part IV: Further Work and Conclusion

CHAPTER 7: FUTURE WORK

There are topics that did not fit in the scope of this thesis. This chapter discusses several of these.

CHAPTER 8: CONCLUSION

In this chapter the results of this thesis are evaluated on a qualitative scale.

Part I.

Introduction and Theory

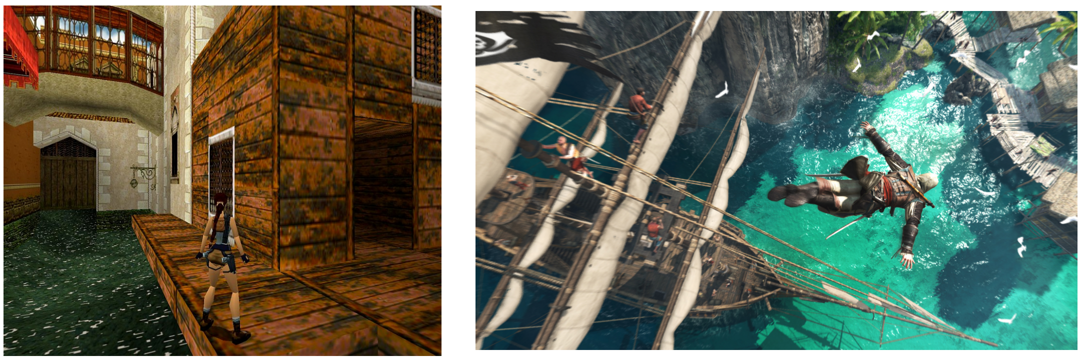
1. Introduction

After a period of incredible advances in processing power in the Eighties, Nineties and early 2000s, progress in the field of classical single core CPUs has slowed considerably in the last 10 years. Given the ever increasing complexity of modern computer programs, different solutions had to be found. Once single core performance could not be increased at a sufficient speed anymore, the tendency to execute code in parallel arose instead. For example, modern CPUs usually have at least two cores, a lot of them even four or more. This trend is no longer specific to traditional computers such as PCs and servers, but can also be found on mobile devices such as smart phones and tablets.

There are two different ways to parallelize applications, i.e. task and data parallelism. Task parallelism is an instance of coarse-grained parallelism. Multiple components of the program are run concurrently, each executing its own instructions and following its own control flow. Multiple CPU cores are the prime example for the facilities that task parallel programs are executed on. Data parallelism usually denotes a more fine-grained approach to parallelism where the same instructions are executed on multiple data elements. These kind of instructions are called **S**ingle **I**nstruction **M**ultiple **D**ata (SIMD) instructions. For example, with the Intel AVX vector extensions, it is possible to multiply a vector of eight 32 bit integer values component-wise with another, in one instruction. In applications that need to have the same operation applied on large amounts of data, this can save a considerable amount of time.

One of the most remarkable developments in the last two decades is the Graphics Processing Unit (GPU). Starting off as a simple I/O device to display two dimensional content on the computer screen, it evolved with the ever increasing demand of graphical applications. Graphical applications typically require the same operation (like for example a shader) to be executed on a large chunk of data, such as a texture file. To fulfill these requirements and satisfy the ever increasing performance demands, GPUs evolved into powerful, massively parallel processors. Due to the nature of the applications, the parallelism of GPUs is data parallel. [4, 13]

Video games are good examples of these increasingly complex graphical applications. Figure 1.1 shows two computer games, *Tomb Raider II* from 1997 and *Assassin's Creed IV* from 2013. They are an example for the performance gains that have been achieved in the last two decades. Today shaders are used to create realistic lighting and water effects, in the early 2000s, programming GPUs was difficult, and developers had to use tricks such as executing multiple passes to get the required results. With the introduction of the Pixel and Vertex shader, people started writing general purpose code for the GPU. Writing general purpose code for these shaders was possible, but difficult. One had to write the shader as if it were displayed, then extract the result from the device's memory. [13, 4]



(a) Tomb Raider II (1997) (b) Assassin's Creed (2014)

Figure 1.1.: Comparison of complexity of video game graphics. Modern GPUs can display graphics that are more complex, by several orders of magnitude.

Some years later, Pixel and Vertex shader units were unified into more general purpose unified Shader units, and around the same time the term of General Purpose GPU (GPGPU) Computing came into use. Microsoft introduced the DirectCompute API, NVidia the CUDA API and the Khronos Group the OpenCL API to enable developers to move parts of the execution of their programs to the GPU. All of these APIs enabled the programmer to execute general purpose code on the GPU without having to use facilities and APIs designed to display graphics. All of them use a C like language that is compiled for the individual GPUs at run time. [7, 8, 17]

1.1. Motivation

Code executed in such a matter can be significantly faster than normal sequential code, especially if it is heavy on computations. Unfortunately, there is a rather high overhead, as resources need to be transferred to the device[1]. Figure 1.2 compares the execution time of applying an operation on every element of a two-dimensional array, with different sizes, using conventional methods and with the execution using a GPU and OpenCL. The first thing one notes is that the overhead alone makes it not viable to perform kernels on the GPU for small arrays. For these arrays, in this example those with less than about $5 * 10^4$ elements, the time spent on copying alone is larger than just executing the whole computation conventionally using the CPU. For larger arrays, there is a significant benefit to performing the computation on the GPU. This is especially true when comparing just the execution of the operation. If there is more then one operation that is performed on the GPU, and the data is already on the GPU, then the difference in execution time is about a single order of magnitude.

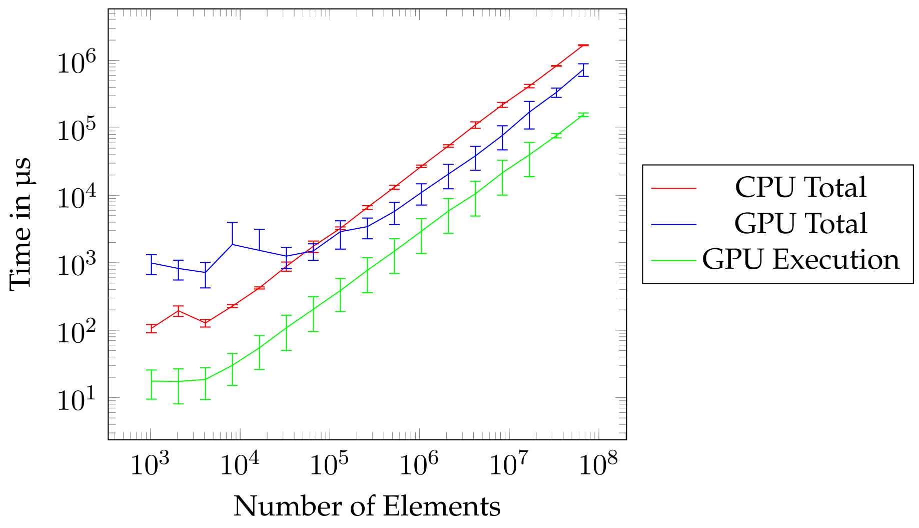


Figure 1.2.: Execution time for the execution of a kernel with different number of elements. The red graph shows the time spent on executing the kernel on the CPU, using parallel for loops and vectorization. The blue graph shows the total execution time for the GPU, including the transfer of the memory to and from the device. The green graph shows the time spent executing the kernel on the GPU.

There are, however, some downsides to executing code on the GPU. There is still a great amount of boilerplate code involved when writing OpenCL code. A lot of repetitive calls to C APIs need to be made, memory needs to be managed manually, work-group sizes determined and kernels enqueued for execution. Additionally, it may not always be a good idea to outsource the computation. OpenCL has certain restrictions, and is not suited for every kind of computation. Additionally, as seen above, there might not even always be

[1]The computing device used, for example the GPU of the PC.

a performance benefit. This thesis is part of an effort to address these concerns by implementing a compiler that automatically generates code both for the CPU and the GPU[2], and selects the right kind of execution based upon the available resources and the program that is executed. Deciding whether to execute a computation on the GPU requires knowledge about the costs that come with the execution. Chapter 5 presents such a model.

[2]See chapter 3

2. Theory

In this chapter the technologies, programming languages and algorithms that form the foundation for this thesis will be explained in detail. The underlying technology used to perform GPGPU computing is the OpenCL[1] computing environment. As the intent of the thesis is to provide runtime predictions for a scheduler in a compiler, the other important part is that compiler and the programming language it belongs to, the FunkyIMP[2] programming language.

2.1. OpenCL

OpenCL was originally released by Apple in 2008 in conjunction with the 10.6 release of OS X. Subsequently, stewardship for the project was transferred to the Khronos Group, which is also responsible for OpenGL, and then released as an open standard for general purpose GPU computing. It is now widely used, with implementations for the major operating systems, as well as a wide range of devices. There are implementations by Apple for OS X, and by NVidia, AMD and Intel for their GPU and CPU products, respectively. [7, 11]

Thus, in contrast to other GPGPU computing frameworks such as the CUDA framework maintained by NVidia, OpenCL is not constricted to a single vendor or type of device. A valid OpenCL kernel may be executed on any device that supports OpenCL, without the need for modifications. This makes the technology viable to be used in an environment with a multitude of heterogeneous devices, enabling an on-demand distribution of data parallel tasks onto available devices. [21]

The following sections will give an overview of the OpenCL architecture, its execution model, memory model, and the C dialect used to write OpenCL kernels.

2.1.1. The OpenCL platform architecture

The OpenCL platform consists of a host and multiple devices. Figure 2.1 illustrates the basic architecture. The Platform, also referred to as Host, is the environment the main program is executed on. It is not an OpenCL device itself, and the code does not have to be written in OpenCL. Any Programming language can be used, although the APIs for interfacing with OpenCL are written in C. There are wrappers for other Programming languages, such as C++. The platform uses these APIs to control the devices. [8]

[1]OpenCL homepage: https://www.khronos.org/opencl/
[2]FunkyIMP homepage: http://www2.in.tum.de/funky

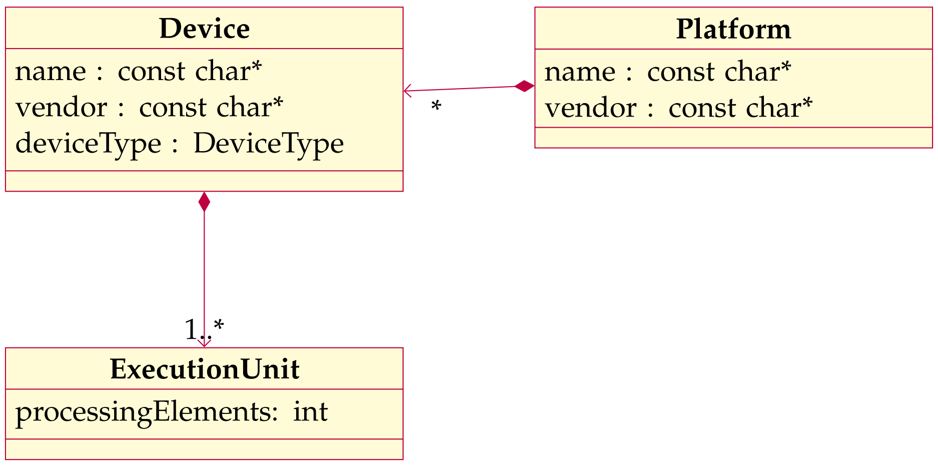


Figure 2.1.: UML class diagram of the OpenCL platform

The Devices execute the actual data parallel programs, also known as kernels. Devices are hardware entities such as a GPU, a CPU or accelerators (in embedded environments) that can perform computations on their own. Each device consists of one or more execution units. For example, an execution unit may map to a core of a CPU. These execution units in turn have a number of ProcessingElements, each of which models the most fine-grained element of parallelism, e.g. a Shader Core. ProcessingElements may be called in a SIMD (**S**ingle **I**nstruction **M**ultiple **D**ata) fashion resulting in a completely parallel execution or in a SPMD (**S**ingle **P**rogram **M**ultiple **D**ata) fashion, wherein each ProcessingElement retains its own Program Counter. [8]

For an example configuration of the platform architecture, see figure A.1 in the Appendix.

2.1.2. The OpenCL execution model

In order to execute code in OpenCL, the following resources need to be managed:

- The **device** that is to be used for the computations.
- The **kernel**, a function that is to be executed on the OpenCL device.
- The **program**, the actual source of the program that is to be executed.
- The **memory** objects the kernel is executed on.

This functionality is bundled in the so-called **context**. Interactions with the device are done using **command queues**. These are used to execute operations, manage the device's memory, or to determine the order the commands are executed in. [8]

The OpenCL programs are written in OpenCL C, which will be discussed in detail in section 2.1.4. They are then executed in parallel over a defined computation domain with

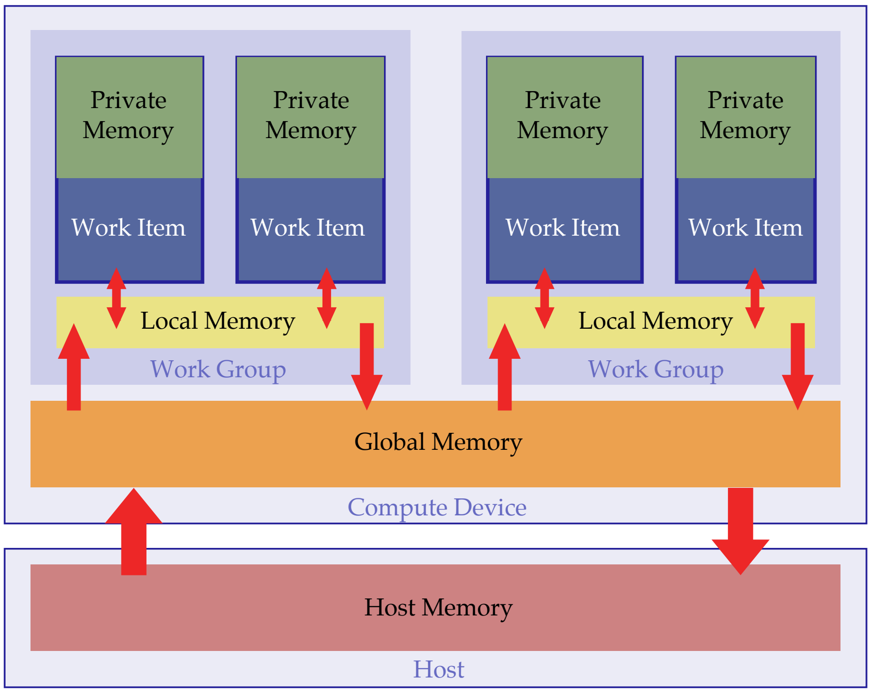


Figure 2.2.: The OpenCL memory hierarchy.

each thread, also known as **work item**, handling one element of that domain. This element might be a single computation on one value in an array, or more involved operations on a column in a matrix. The work items are again grouped into **work-groups**[3], which enable synchronization within a work-group. There is also a level in the memory hierarchy that is shared within a work-group. The different work-groups on the other hand are completely independent from each other. As there are many threads executed at the same time, they usually share one program counter, executing the same instruction on multiple threads at once. This has some interesting effects. If there are control structures or branches in the code, every branch will still have to be executed, and results of the computations will have to be canceled out if the computation is not to be done on that particular work item. This also translates to performance penalties for very control-flow-heavy programs. [20]

2.1.3. The OpenCL memory model

An important design aspect for OpenCL is the closeness to the hardware. This implies the need for a memory architecture that maps well to real graphics hardware. To achieve this, a multi-layer memory architecture has been chosen. Consider figure 2.2 for an overview. The fastest type of memory is the **private memory**. Every work item has its own, and it is typically the part of memory where all local variables and parameters used in a kernel are stored. For a NVidia graphics card, the access time would typically be comparable to a register access. However, this is not guaranteed. [20, 17]

[3]Sometimes also known as wave fronts.

Next in the memory hierarchy is the local memory. This part of memory is shared amongst the work items of a single work-group. On NVidia graphics cards, local memory is located in the off-chip DRAM, resulting in longer latencies for memory accesses, typically around 200-300 cycles. [17, 10]

There also is a special area of memory reserved for constant data. Data in Constant memory is immutable for the duration of the kernel execution, and can thus be cached for quicker access. This usually results in shorter latencies, for NVidia graphics cards it is usually about as fast as a register access. [10]

Lastly, there is the Global Memory. It has the greatest amount of available memory, and it is shared across the whole device. On NVidia graphics cards it is usually implemented as a dedicated off-chip DRAM. This implies a high access latency, and as with the local memory, it is around 200-300 cycles. [10, 20]

It is also notable that there is no concept of memory protection on the GPU. Segmentation faults are not reported, that means if a work item modifies a memory segment outside of its own memory segment, anything can happen, up to and including gibberish on the screen or a full crash of the operating system. When implementing a new OpenCL kernel, it is usually a good idea to test it on a CPU device first, as those usually implement memory protection, and a segmentation fault will be reported instead of causing random undefined behavior on the system. [20]

2.1.4. The OpenCL C dialect

OpenCL kernels are written in a slightly modified version of C99, a language called OpenCL C. It features some additional data types for handling vector types, as well as well as implementations for basic operations on them. Vectors can have a length of 2, 3, 4, 8 or 16. The following data types are supported: `char`, `short`, `int`, `long` and their unsigned counterparts, as well as `float` for floating point values. The operations are the usual operators, and they are applied component-wise (see equation 2.1). [8]

$$\begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} = \begin{pmatrix} a_1 + b_1 \\ a_2 + b_2 \\ a_3 + b_3 \\ a_4 + b_4 \end{pmatrix} \tag{2.1}$$

There are several limitations as well, the most important one shall be explained here. For the full list, see [8]. In OpenCL, function calls are implemented via inlining, that means that functions that are called from within a kernel simply have their body copied into the original kernel, replacing the parameters by the function call arguments. This implies two limitations. The inlining has to be a finite process, that means that *recursion is not possible*, specifically, there may not be any cycles in the function call graph. It also means that all the called code has to be known at (kernel) compile time. The second implication is therefore the impossibility of function pointers, as they would enable the programmer to call any code at runtime, thus rendering the inlining impossible. There are some more limitations for pointers. They may only be declared in conjunction with either the `global`, the `local` or the `private` address space qualifier. Pointers of one address space specifier may not be assigned to a pointer with a different address space specifier. [8]

Figure 2.3 shows a very simple OpenCL kernel. It illustrates several elements that are common to all kernels. In the function signature, kernels are marked as such using the `kernel` keyword. The return type of a kernel is always `void`. Parameters are used for the communication between the kernel and the host. There are parameters that point to the data transferred to the GPU by the host, and a pointer to the memory segment that is to hold the result. By default, the parameters reside in the `private` address space. The `constant` and `global` address space specifiers denote the type of memory the pointers are pointing to. Local variables, such as `work_item_id` are also in the `private` address space by default. The function `get_global_id(0)` is called to determine the id of the current work item. [8]

```
kernel void vector_add(constant float *a, constant float *b,
                        global float *c)
{
        int work_item_id = get_global_id(0);
        c[work_item_id] = a[work_item_id] + b[work_item_id];
}
```

Figure 2.3.: This is an example of an OpenCL-Kernel. It adds the content of two vectors and writes the result into a third one. `get_global_id(0)` returns the current work item number, in this case it corresponds to the position in the arrays.

2.2. The FunkyIMP programming language

The FunkyIMP programming language is developed by Alexander Herz as a race and deadlock-free, implicitly parallel, functional programming language. It employs task as well as data parallelism in order to maximize the number of computations that are executed in parallel. This is done without losing the ability to handle I/O and events. Thus funkyIMP can be considered a general purpose programming language. For this thesis, the type system, with its polyedric array domains, is of most interest. It is explained below. [9]

Most functional languages use single linked lists as their main way of storing data in bulk. For these lists, a wide range of operations, such as `map`, `filter` or `foldl`[4] are available. These operations enable the programmer to write very concise, declarative code, focusing on what to achieve instead of how to do it. A simple example of this is the reversing of lists. Using functional code with lists, this can be expressed as below. The example is written in Standard ML.

```
val aList = [1,2,3,4]
fun rev l = foldl nil (fn (a,b) => a::b) l
val reversed = rev aList (*Evaluates to [4,3,2,1]*)
```

The code is short and very declarative, and the meaning should be clear at first glance. However, the implementation of the same functionality is more difficult with arrays. In Standard ML, arrays are declared in the structure `Array`. To reverse an array, one has to write code such as the following.

```
val arr = Array.fromList [1,2,3,4] (*{|1,2,3,4|}*)

fun reverse_array arr =
  let
    val arr_len = Array.length arr
    val arr_work = Array.array (arr_len, Array.sub (arr, 0)) (*1*)
    fun appi_op (p,v) =
      let
        val cur_pos = arr_len - 1 - p (*2*)
      in
        Array.sub (arr, cur_pos)
      end
    val x = Array.modifyi appi_op arr_work
  in
    arr_work
  end

val arr_rev = reverse_array arr (*Evaluates to {|4,3,2,1|}*)
```

This code is much more complicated. It needs to use array indices and several temporary variables. Additionally, `Array`s are mutable in SML, and the implementation has to

[4]For descriptions of those functions, please see B.2

make use of this. To sum it up, the declarative nature of the code is largely absent, instead it is solved with an iterative approach. With that added complexity comes several possible problems. First, consider `(*1*)`. This line will fail to execute if the array has zero elements, a case that should be handled using a conditional statement as well, otherwise the execution will fail with an exception at run time. The other index operations also need to perform bounds checks at runtime, and would raise exceptions if the accesses were outside the array bounds. Programming errors like that can easily happen, for example one might forget to subtract the 1 at line marked with `(*2*)`.

How does this complexity arise? In most programming languages, arrays are treated as one-dimensional pieces of memory, just as in the example above. In a functional environment, and with immutable values, the actual structure with which the values are saved in memory might not matter too much. It may be more important to have a type system that enables reasoning about array size and structure at compile time, enabling the programmer to create subarrays that fulfill certain criteria, and to perform operations on them. In other cases, such as the reverse example, it might be enough to provide a different way to address the original array.

FunkyImp allows the programmer to do that. Given an one-dimensional array defined as

$$\text{domain } arr_{1D}\{x\}() = \{(i) : i < x\}$$

one can define the reversed array as follows:

$$\text{domain } arr_{rev}\{x\}() : arr_{1D}(a) = \{arr_{1D}\{x\}(i) : a = x - i - 1\}$$

Now all the programmer needs to do is create a new variable and cast the old value into the new one, e.g.:

```
int[arr_1D{10}] x = new int[arr_1D{10}];
int[arr_rev{10} y = (int[arr_rev{10}]) x;
```

2.2.1. Multidimensional Arrays

Multidimensional arrays are another problem that are efficiently solved by polyedric domains. This is shown with the implementation of a matrix as an example. There are several different approaches to solve this problem. These will be illustrated with the following example:

$$M \in \mathbb{R}^{3\times 4}$$
$$M = \begin{pmatrix} 1 & 2 & 26 & 4 \\ 5 & 6 & 14 & 8 \\ 42 & 7 & 3 & 89 \end{pmatrix}$$
$$v = M_{i,j}$$

The least involved approach is to simply implement the matrix as a one-dimensional array. In an imperative language such as C, it might be implemented as follows:

```
void function(int i, int j) {
  float *M = (float *) malloc(3*4*sizeof(float));
  //... Init values ...
  float v = M[i*3+j];
  free(M);
}
```

This approach is the most simple one to implement in the compiler, but leaves most of the computational burden for the programmer. For one, the structure of the matrix is not expressed in the type of `M`. If the programmer wants to use the matrix in another function, they will have to manually transfer all the information about the structure of the matrix to that other function. If that is not done, nothing prevents the programmer from interpreting the matrix as, say, for example, a $\mathbb{R}^{6\times 2}$ matrix. Additionally, array access is not straightforward. Again, the programmer has to maintain the semantic meaning of the array. If they want to access an element of the matrix, they will have to compute the right index for the array index themselves. To sum it up, this is the approach that is closest to the actual memory structure of the computer, but it does assume careful attention by the programmer.

Another approach is to implement the matrix as an array of arrays. This might be implemented as follows, for example in Standard ML:

```
val i = (*...*)
val j = (*...*)

open Array
val m = array (3, array (j, 4))
(*... Init values ... *)
val v = sub (sub (m, i), j)
```

This approach is still quite simple to implement from a compiler point of view. It has several advantages from a type safety point of view. It is now no longer possible to reinterpret the matrix as something other than a two-dimensional array with the given dimensions. This type safety, however, comes at a cost. It is now no longer guaranteed that the matrix is stored in one continual segment of memory. That might have negative side effects on the performance. Additionally, from a semantic standpoint, an array of arrays is still a slightly different concept than a two-dimensional array.

The FunkyIMP approach combines type safety while still retaining the continuous memory segment by taking the address computations out of the programmer's hands and generating the necessary code during complilation. It might be implemented as follows:

```
domain one_d{x} = {(a) | a < x}
domain two_d{x, y} : one_d{x*y} (o) = {(a,b) | a<x & b<y}

float M[two_d{3,4}] = new float[two_d{3,4}];
//... Init values ...
//init code
```

```
float v = M[i,j];
```

The first line defines a simple one-dimensional array. Usually, that will be predefined. The second line describes a two-dimensional array with a rows and b columns. It inherits the properties of a one-dimensional array with the length of $a * b$. Then, constraints for the indexing parameters are specified. Here, the parameters may only be within the limits set by a and b. Constraints specifying that $x, y \geq 0$ are implicitly added. In line 3, the newly created domain is used to build a matrix with a size of 4×3. After the values are initiated, the programmer can simply retrieve them using an indexed access. This solution captures the semantics of a matrix, while still retaining the runtime benefits of the C implementation, as it basically performs the same index computations. In contrast to the C example, here they are generated by the compiler.

2.2.2. Map and Reduce

Recall the listing from the beginning of section 2.2. It describes several operations typically implemented on lists in functional programming languages, namely `map`, `filter` and `foldl`. They provide a simple and easily understandable way to iterate on lists. These operations (except for `filter`[5]) on arrays are implemented as language features in funkyIMP.

To illustrate their functionality, the following operations will be performed, the first using map and the second one using reduce.

$$M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}^T$$

$$v = \sum_{m_{ij}: i,j \in \mathbb{N}_3} m_{ij}$$

The code looks as follows:

```
int M[two_d{3,3}] = new int[two_d{3,3}].\(i,j){ i * 3 + j };
M't = M.\(i,j) { M[j,i] };
int v = M.\reduce {0} (i,j, accum) { M[i,j] + accum };
```

The `.\` performs the map operation. It returns a new Array with the same type as the old one. The second operation, the `.\reduce` describes how to collect the contents of the array into a single value.

For this thesis, the `map`-Operation[6] is considered the most interesting. The original matrix is immutable and the new value only depends on the old matrix (and other, also immutable, variables). Hence the operation may be parallelized readily. Chapter 3 will highlight the steps to be taken in order to incorporate the generation of code for the GPU-accelerated `map`-operation into the funkyIMP compiler.

[5]It does not make too much sense to remove elements from an array that cannot be resized.
[6]Also referred to as a domain iteration in later chapters.

Part II.

Methodology

3. Extensions to the funkyIMP Compiler

The funkyIMP compiler is a heavily modified version of the Java compiler. Its front end has been customized to parse code that adheres to the funkyIMP language specification. More pertinent for this thesis is the compiler back end. It has been replaced completely to emit C++11-compatible code instead of Java byte code. This generated code is then compiled to native machine code using a GNU g++ compiler. In order to support the execution of code on the GPU, several modifications had to be made to the existing code base. This chapter will discuss them in detail, starting with the overview of the process at a high level, and then explaining each component individually.

3.1. System Overview

Offloading a computation to the graphics card takes several steps. In the beginning, one must gather all the necessary data and code that will be used in the computation. To do this, all symbols used in the computation need to be collected, including those found in functions called within the original kernel. Functions need to be translated into OpenCL-C, local variables are passed as parameter and memory segments need to be processed and copied to the GPU. As OpenCL does not have a sophisticated array type system, some additional information such as the size of the array in bytes need to be computed. Afterwards, code to allocate space on the GPU to copy the content of the array on host side to the GPU is generated. Further steps to generate the code on the host side include the allocation of space for the computation results on the GPU, the actual invocation of the computation, and the copying of the result of the computation back to the host.

In addition to the generation of the boilerplate code to invoke the OpenCL kernel, code for the actual kernel itself needs to be generated as well. This is done, in accordance with the rest of the compiler architecture, by utilizing the Visitor Pattern[1]. It iterates over the Syntax tree, generating code recursively.

There also is a runtime component for the funkyIMP language. There is a library[2] that encapsulates the OpenCL API calls with a simple object-oriented C++ API. The library has been slightly changed to incorporate some additional features, such as querying frequently needed device information.

The following sections will give a detailed overview of the additions to the funkyIMP compiler developed in the course of this thesis. They will follow the steps the compiler performs to output the necessary C++ code.

[1]As defined in [6]
[2]See C++ Wrapper [15]

3.2. Code generation for the Host

This section details the generation of code to handle the host side of the kernel execution, the compilation of the kernel, its execution, and the management of the necessary variables.

3.2.1. Collection of Identifiers

The first step to be performed is to collect all the identifiers needed for the execution of the kernel. There are several types of identifiers to be considered. The first type is the **local variable**. Local variables are self-contained values that can be set simply as a parameter for the kernel. The second type of identifier is the **array**. Arrays are also set as a parameter. In addition, the memory segments they point to need to be copied to the GPU. The third type of identifier holds **functions**. In the function body, other functions, arrays or local variables may be accessed too. This necessitates an analysis of the body of the function for identifiers. Functions also need to be compiled. The last type of identifiers to be considered are **Iteration Arguments**. These need not be copied, as they can be computed from the work item id of the kernel. They will be computed on the GPU at runtime.

To gather the necessary information about the kernel, the following analysis is performed. Let $\mathcal{V}$ be the set of all the possible syntactical elements of the language (e.g. Identifiers, keywords, Literals, ...) and let $\mathcal{T}$ be the set of all syntax trees defined as in equation 3.1. Let $\mathcal{L} \subseteq \mathcal{V}, \mathcal{A} \subseteq \mathcal{V}, \mathcal{F} \subseteq \mathcal{V}$ be the set of all local variables, the set of all arrays and the set off all functions, respectively. Let $\mathcal{I}$ denote the set of iteration parameters of the domain. An abstraction of the syntax tree looks as follows:

$$\begin{aligned} &v \in \mathcal{V} \\ &t = \begin{cases} \text{Node}(v, t_1, \ldots, t_n) \\ \text{Leaf}(v) \end{cases} \end{aligned} \tag{3.1}$$

With those definitions the set of all identifiers needed in a kernel, denoted as $\mathcal{E} \subseteq \mathcal{L} \cup \mathcal{A} \cup \mathcal{F}$, can be computed as can be seen in equation 3.2. Intuitively, the syntax tree is iterated over, and in the case where there are any identifiers that denote functions, their bodies will also need to be checked for identifiers as. For an example, see C.1 in the appendix.

$$\begin{aligned} [\![\text{Node}(T_1, T_2, \ldots T_n)]\!]^\sharp &= [\![T_1]\!]^\sharp \cup [\![T_2]\!]^\sharp \cup \cdots \cup [\![T_n]\!]^\sharp \\ [\![\text{Leaf}(i)]\!]^\sharp &= \{i\} && (i \in \mathcal{L} \cup \mathcal{A}) \\ [\![\text{Leaf}(f)]\!]^\sharp &= \{f\} \cup [\![\text{resolve}(f)]\!]^\sharp && (f \in \mathcal{F}) \end{aligned} \tag{3.2}$$

$$\begin{aligned} resolve &: \mathcal{F} \to \mathcal{T} \\ resolve(f) &= t_f && \text{where } t_f \text{ is the body of } f. \end{aligned}$$

Figure 3.1.: Analysis to extract the set of used identifiers from an expression.

Some of the identifiers captured in $\mathcal{E}$ may not be necessary. The iteration arguments are generated on the GPU at runtime, as they depend on the number of the work item they are executed upon. Hence, they need to be taken from the set. Additionally, any variables that are locally defined in called functions may be taken from the set as well. This yields the final set of needed identifiers $\mathcal{E}^* = \mathcal{E} \setminus (\mathcal{I} \cup \mathcal{L}_{local}$

3.2.2. Code for needed variables

All array variables that need to be copied to the GPU have to be processed. They have to be mapped to the more simple, C-style memory segment while keeping the information about the array's structure intact. Specifically, the lengths of the array's dimensions need to be obtained. Additionally, code for the allocation of the memory on the GPU has to be generated. For the purpose of code generation, with $\mathbb{T}_p$ denoting the set of primitive types (e.g `float`, `int`, `double`, `char`,...), an array may be described as follows.

$$A = \langle m, d, T \rangle$$
$$T \in \mathbb{T}_p, d \in \mathbb{N}^n, m \in T^s \ where \ s = \prod_{k=1}^{n} d_k$$

With this definition the code generation function $code_{Host}$ for array variables can be defined as given in equation 3.3. The first step is to extract a simple pointer to the array's memory segment using the FunkyIMP runtime method `toNative()`. The next step is to allocate enough space on the GPU to fit the memory segment just extracted. The total size of that segment is calculated from the size of the primitive data type of the array and the product of the sizes of each dimension. Additionally, for each dimension, a local variable, holding its size, is allocated. Finally, the memory segment is copied over to the GPU.

Code generation for local variables with a primitive type is less complex than conde generation for arrays. They can be defined as $L = \langle T, v \rangle$ with $T \in \mathbb{T}_p$ and $v \in T$. Their code generation function is given in equation 3.4.

$$\begin{aligned} code_{Host}\ A \ = & \ T \ * m = A \rightarrow \text{toNative}(); \\ & \text{ocl\_mem}\ m_{GPU} \ = \\ & \quad \text{device.malloc}\left(\text{sizeof}\left(T * \prod_{k=1}^{n} d_k\right), \text{CL\_MEM\_READ\_ONLY}\right); \\ & \text{int}\ m_{dim_0} = d_1; \\ & \quad \vdots \\ & \text{int}\ m_{dim_n} = d_n; \\ & m_{GPU}\ .\text{copyFrom}(m); \end{aligned} \tag{3.3}$$

$$code_{Host}\ L \ = T\ a = v; \tag{3.4}$$

3.2.3. The return value

The return value of a kernel has to be treated slightly differently. There is no direct counterpart on the host, as the object is created by the domain iteration. Hence, memory for the return value needs to be allocated on the GPU as well as on the host. In contrast to the arrays in the section above, the memory is not allocated in read only mode but in write only mode. The return value cannot be referenced from funkyIMP code, so it is not necessary to make it readable. In addition, the memory is not initialized to anything and contains undefined values.

$$\begin{aligned} code_{Host}^{begin}\ R\ &= T\ *ret = \text{new}\ T\left[\prod_{k=1}^{n} d_k\right]; \\ &\text{ocl\_mem}\ ret_{GPU}\ = \\ &\quad \text{device.malloc}\left(\text{sizeof}\left(T * \prod_{k=1}^{n} d_k\right), \text{CL\_MEM\_WRITE\_ONLY}\right); \end{aligned} \tag{3.5}$$

The return value needs to be copied back to the host after the computation, and then also translated back into a format compatible with the funkyIMP runtime. The code generation function is shown in equation 3.6. First a new `LinearArray` object is created. Then a new `Version` object ($ret_{funky'}$) is built. It is a version of the array enriched with information about the structure of the array, and the value that is returned as the result of the computation.

$$\begin{aligned} code_{Host}^{end}\ R\ &= ret_{GPU}\ .\text{copyTo}(ret); \\ &\text{funky} :: \text{LinearArray} < T > *ret_{funky} = \\ &\quad \text{new funky} :: \text{LinearArray} < T > \left(\prod_{k=1}^{n} d_k, ret\right); \\ &\text{funky} :: \text{LinearArray} < T >:: \text{Version}\ *ret_{funky'} = \\ &\quad \text{new funky} :: \text{LinearArray} < T >:: \text{Version}(ret_{funky}, n, d_1, \ldots, d_n); \end{aligned} \tag{3.6}$$

3.2.4. Code generation for the Domain iteration

There are more tasks that need to be performed apart from the allocation of variables. The actual code for the GPU needs to be compiled and the parameters set, and then the code needs to be executed. Equation 3.7 shows the code generation function $code_{Host}$, a formal definition of the whole process. The kernel is first compiled for execution on the target device. Then code for all variables gathered with the analysis presented in section 3.2.1 will be generated. The process is similar for the allocation of the return value. Afterwards, the kernel arguments need to be set. For primitive variables it is enough to pass the address they are stored at, while for an array the actual OpenCL memory object and the variables holding its dimensions need to be given. Subsequently the kernel is executed with $|R|$ work items and a work group size that is the greatest even divider of the number of work

$$
\begin{aligned}
code_{Host}\ P \ = \ & \text{ocl\_kernel}\ \ k(\&device, \text{“tmp/k.cl”}); \\
& code_{Host}\ V_1 \\
& \quad \vdots \\
& code_{Host}\ V_n \\
& code^{begin}_{Host}\ R \\
& k\,.\,\text{setArgs}(args(V_1), \ldots, args(V_n)) \\
& k\,.\,\text{run}(workgroup(|R|), |R|) \\
& \text{device}\,.\,\text{finish}(); \\
& code^{end}_{Host}\ R \\
& \text{return}\ ret_{funky'}
\end{aligned}
\tag{3.7}
$$

$$\{V_1, \ldots, V_n\} = (\llbracket P \rrbracket^{\sharp}_{ident} \cup R) \setminus (\mathcal{I} \cup \mathcal{F}) \tag{3.8}$$

$$
args\ V = \begin{cases}
\&V & V \in \mathcal{L} \\
\begin{array}{l} V.\,\text{mem}() \\ \quad \&v_{dim_0} \\ \quad\quad \vdots \\ \quad \&v_{dim_n} \end{array} & V \in \mathcal{A}
\end{cases}
\tag{3.9}
$$

$$|\langle \_, d, \_ \rangle| = \prod_{k=1}^{n} d_k \tag{3.10}$$

$$workgroup\ n \ = \ \max\{w |\ w > 0 \land w < n \land w < W_{max} \land w \mod n = 0\} \tag{3.11}$$

Figure 3.2.: Code generation function for the whole domain iteration.

items that is less than or equal to the maximum work group size of the GPU W_{max}. Section C.2 in the appendix shows the actual output of the code generation process for a domain iteration. The identifier names in the output generated by the compiler differ slightly from the ones in the hypothetical output of the the code generation function $code_{Host}$ specified in this section. This deviating naming schema is utilized in the implementation to avoid duplicate identifier names.

3.3. Code Generation on the GPU

In this section the part of the code generation that produces code for the GPU is explained. There are two steps to the process: The first step is to gather all the functions that are called from the kernel, and to build a dependency graph. The second step is to generate code for them and the kernel. The kernel is a function as well, but there are several special properties that need to be considered.

3.3.1. Dependent Function Analysis

One of the most important ways to structure code is the use of functions and procedures. Unfortunately, support for function calling in OpenCL C is somewhat limited. Function calls will be inlined, and hence it is impossible to generate code for programs with recursive function calls. [8]

This necessitates an analysis of the called functions. The call graph of a function can be obtained from the funkyIMP compiler. The call graph in general has the properties of a directed graph, each function being represented by a vertex, and each function call being an edge. All functions that are reachable from the kernel vertex[3] of the call graph need to be included in the translation. Recursion is not allowed, hence a way to detect it is needed. In terms of graph theory, a function f is recursive iff $\mathrm{Path}(f, f)$. It can be seen that this is the case if the graph has a cycle. In case the graph is not recursive, the graph needs to be topologically sorted so that the code for functions can be generated in a way that makes forward declarations unnecessary.

$$\begin{aligned}
G = \langle V, E \rangle\,,\; E \subseteq V \times V & \qquad \text{(A directed graph)} \\
\mathrm{Path}(v_1, v_2) \Leftrightarrow (v_1, v_2) \in E \lor \exists v' : (v_1, v') \in E \land \mathrm{Path}(v', v_2) & \qquad \text{(Path between } v_1\text{, } v_2\text{)} \\
\mathrm{DAG}(G) \Leftrightarrow \neg\exists (v_1, v_2) \in E : \mathrm{Path}(v_1, v_2) \land \mathrm{Path}(v_1, v_2) & \qquad \text{(Directed Acyclic Graph)} \\
\mathrm{Sorted}([v_1, \ldots, v_n]) \Leftrightarrow \neg\exists\, 1 \le k \le n : \exists\, 1 \le l \le n : \mathrm{Path}(v_l, v_k) & \qquad \text{(Topologically sorted List)}
\end{aligned}$$

Section C.4 in the appendix illustrates these principles with an example. The code generation for functions is discussed in the next section.

3.3.2. Code generation for Functions

For the compilation of functions there are two tasks that need to be performed. The first is to generate the function signature, the second to generate the function body itself. The first step will be described here, while the second step is covered below, in section 3.3.3, where code generation for GPU kernels is described.

Function Headers in C need to have a unique name, whilst funkyIMP allows overloaded functions. For this reason, the compiler needs to generate a unique name from the function specified in funkyIMP. Additionally, if there are any array types, they need to be translated to the appropriate pointer types. The code generation function is given in equation 3.12.

[3] i.e $\mathrm{Path}(kernel, f)$

$$\begin{aligned}
\mathit{code}_{GPU}\ T_{ret}\ f(\text{params})\{\text{body}\} &= \mathit{code}_{GPU}\ T_{ret} \\
&\quad \mathit{name}\ (T_{ret}\ f(\text{params})) \\
&\quad (\mathit{code}_{GPU}\ \text{params}) \\
&\quad \mathit{code}_{GPU}\ \text{body} \\
\mathit{code}_{GPU}\ T &= T && (T \in \mathcal{T}_p) \\
\mathit{code}_{GPU}\ T &= T* && (\text{for others}) \\
\mathit{code}_{GPU}\ \text{params} &= \mathit{code}_{GPU}\ T_1\ t_1, \\
&\quad \vdots \\
&\quad \mathit{code}_{GPU}\ T_n\ t_n \\
\mathit{code}_{GPU}\ \text{body} &= \{\mathit{code}_{GPU}(S_1; \ldots S_m)\}
\end{aligned} \tag{3.12}$$

$$\mathit{name}(T_{ret}\ f(\mathrm{T_1 t_1}, \ldots \mathrm{T_n t_n})) = T_{ret}\_f\_T_1\_\ldots\_T_n$$

3.3.3. Code Generation for GPU Kernels

The kernel itself is also an OpenCL function, but it has some special properties. Some of these stem from OpenCL and some are limitations imposed by the manner in which funkyIMP handles the domain iterations. OpenCL states that kernels are to be marked with a special `kernel` keyword. It is also required that the return type of all kernels is void. The result is stored in a section of the global memory, and a pointer to that section is passed as a parameter. There are also some special considerations for the parameters. Arrays need the `global` address space specifier, whilst primitive values need no address space qualifier which makes them implicitly `private`. This leads to the slightly modified code generation function specified in equation 3.13.

$$\begin{aligned}
\mathit{code}_{GPU}\ T_{ret}\ f(\text{params})\{\text{body}\} &= \text{\_\_kernel void}\ k \\
&\quad (\mathit{code}_{GPU}\ \text{params}, \mathit{code}_{GPU} V_R) \\
&\quad \mathit{code}_{GPU}^{kernel}\ \text{body}\ V_R \\
\mathit{code}_{GPU}\ V &= T_V\ n_V && (T_V \in \mathcal{T}_p) \\
\mathit{code}_{GPU}\ V &= \text{global}\ T_V * n_v, && (T_V \notin \mathcal{T}_P) \\
&\quad \text{int}\ V_{dim_0}, \\
&\quad \vdots \\
&\quad \text{int}\ V_{dim_n} \\
\mathit{code}_{GPU}\ \mathrm{V_1, \ldots, V_n} &= \mathit{code}_{GPU}\ V_1, \\
&\quad \vdots \\
&\quad \mathit{code}_{GPU}\ V_n
\end{aligned} \tag{3.13}$$

The parameters V_1 to V_n are equal to the ones specified in equation 3.8. Their order is also identical. The name of the kernel (k) also matches the name of the kernel on the host side.

The code generation function given above does not define the code generation for the body of the kernel. In section 3.3.2, when translating helper functions, it was enough to simply compile the statements of the function. The domain iteration however is only an expression, not a series of statements, hence its result needs to be assigned to something so that it does not get lost. In this case it is the element of the return value that is currently processed. That element is determined by querying for the work item ID of the currently executed work item and reconstructing the original position in the array from that. The process can be formalized with the code generation function given in equation 3.14.

$$\begin{aligned}
code_{GPU}\ \{e\}\ \langle m, d, T\rangle = \{ & \\
& \text{int}\, id_{workItem} = \text{get\_global\_id}(0); \\
& \text{int}\, pos_{dim_0} = id_{workItem}/(stride\ d\ 0)\ \%\ d_0; \\
& \quad \vdots \\
& \text{int}\, pos_{dim_n} = id_{workItem}/(stride\ d\ n)\ \%\ d_n; \\
& m[id_{workItem}] = code_{GPU}\ e; \\
\} & \\
stride\ d\ k = \prod_{i=k+1}^{n} d_i &
\end{aligned} \tag{3.14}$$

The reconstruction of the position in the array is taking advantage of a simplification of the problem domain. It is assumed that the array is an n-dimensional (hyper)-rectangular object. With that simplification one can calculate the position in the current dimension by first dividing the absolute position by the stride. The stride is the amount of memory that has to be skipped in order to get to the element that has the next position in the current dimension. The position in the current dimension can be obtained by taking the result of the division modulo of the size of the current dimension.

3.3.4. Code Generation for funkyIMP Constructs

There are other funkyIMP constructs that need to be translated to OpenCL C. They are described in this section.

The Block Statement

In C, as well as in funkyIMP, the block statement is used to open a scope for variables and group several statements into a unit, for example to execute them multiple times in a loop. Code generation is fairly straightforward, as there are no significant differences between funkyIMP and OpenCL.

$$\begin{aligned} \mathit{code}_{GPU}\ \{S_1; \dots S_n\} = \{ & \\ & \mathit{code}_{GPU}\ S_1 \\ & \quad\vdots \\ & \mathit{code}_{GPU}\ S_n \\ \} & \end{aligned} \tag{3.15}$$

The `if` Statement

The if statement, like the block, is very similar in funkyIMP and C. The `if` statement is used to branch the execution and enable conditional execution of statements. It is important to note that due to the fact that code is executed in a SIMT fashion, if one work item in a work-group executes a branch, all the other work items in the work-group also have to execute this branch, even if they did not take the branch[4]. For code that is heavy with branches, this may lead to a significant performance overhead. Whenever possible, branches should be avoided, or replaced by conditional assignments. The code generation function for the `if` statement looks as follows.

$$\begin{aligned} \mathit{code}_{GPU}\ \text{if } (\mathit{condition}) S_1 \text{ else } S_2 = & \text{ if}(\mathit{code}_{GPU} \mathit{condition}) \\ & \quad \mathit{code}_{GPU}\ S_1 \\ & \text{ else} \\ & \quad \mathit{code}_{GPU}\ S_2 \end{aligned} \tag{3.16}$$

Assignments

Assignments in funkyIMP are slightly more involved. The language has a linear type system that only allows single assignment of identifiers. To reduce unnecessary type annotations, one may assign a value of the same type as the old identifier by suffixing the original name. The example below illustrates this.

```
int a = 10;
a'x = 20;
a'y = a'x * 20;
```

To translate this construct into OpenCL C, the the original type of the suffixed variable needs to be looked up. Additionally, C does not allow “'” in identifier names, so they need to be replaced as well.

$$\begin{aligned} \mathit{code}_{GPU}\ (T\ n = e) &= \mathit{code}_{GPU}\ T \quad n = \mathit{code}_{GPU}\ e; \\ \mathit{code}_{GPU}\ (n's = e) &= \mathit{code}_{GPU}\ T_n \quad n\_s = \mathit{code}_{GPU}\ e; \end{aligned} \tag{3.17}$$

[4]In this case, they ignore the results made in the branch.

Arithmetical Expressions

Arithmetical expressions in funkyIMP are either expressions with an unary or a binary operator application. As the compiler holds expressions in a tree structure and a one-dimensional string is to be generated, parentheses need to be inserted at appropriate places in order to ensure the semantic equality of the output. Let $op <_P e$ denote that the operator precedence of op is less then any part of e, likewise for the other comparison operators.

$$\begin{aligned}
code_{GPU}\ \mathrm{op}_\mathrm{U}\ e &= \mathrm{op}\, code_{GPU}\ e && (op_U \leq_P e) \\
code_{GPU}\ \mathrm{op}_\mathrm{U}\ e &= \mathrm{op}\ (code_{GPU}\ e) && (op_U >_P e) \\
code_{GPU}\ e_1\, \mathrm{op}_\mathrm{B}\ e_2 &= code_{GPU}\ e_1\, \mathrm{op}_\mathrm{B}\ code_{GPU}\ e_2 && (op_B <_P e_1, op_B \leq_P e_2) \\
code_{GPU}\ e_1\, \mathrm{op}_\mathrm{B}\ e_2 &= (code_{GPU}\ e_1)\, \mathrm{op}_\mathrm{B}\ code_{GPU}\ e_2 && (op_B \geq_P e_1, op_B \leq_P e_2) \\
code_{GPU}\ e_1\, \mathrm{op}_\mathrm{B}\ e_2 &= code_{GPU}\ e_1\, \mathrm{op}_\mathrm{B}\ (code_{GPU}\ e_2) && (op_B <_P e_1, op_B >_P e_2) \\
code_{GPU}\ e_1\, \mathrm{op}_\mathrm{B}\ e_2 &= (code_{GPU}\ e_1)\, \mathrm{op}_\mathrm{B}\ (code_{GPU}\ e_2) && (op_B \geq_P e_1, op_B >_P e_2)
\end{aligned} \tag{3.18}$$

$$\begin{aligned}
\mathrm{op}_\mathrm{U} &\in \{*, !, ++, --, \tilde{}, \&\} \\
\mathrm{op}_\mathrm{B} &\in \{+, -, *, /, \%, <<, >>, \&, |, <, <=, >, >=, ==\}
\end{aligned}$$

Array Accesses

Array accesses in funkyIMP have the form `arr[a,b,d]`. Each of the values in the index expression indicates the position of the element in the corresponding dimension. This essentially hides the complexity of the address computations from the developer. These computations need to be generated for the OpenCL C code. Due to the simplifications noted in section 3.3.3, the actual offset within the array can be computed by simply multiplying the strides of the dimensions with the indices, and adding up the results. The code generation function looks as follows.

$$\begin{aligned}
code_{GPU}\ e_0[e_1, \ldots, e_n] &= code_{GPU}\ e_0 \\
&\quad [code_{GPU}\ e_1 * stride\ d\ 1 \\
&\quad + \quad \vdots \\
&\quad + code_{GPU}\ e_n * stride\ d\ n] \\
stride\ d\ k &= \prod_{i=k+1}^{n} d_i \\
\langle m, d, T \rangle &= resolve\ e_0
\end{aligned} \tag{3.19}$$

Type Annotations

Type annotations are also fairly simple to translate, as arrays are just pointers on the OpenCL C side, and the primitive values simply stay as they are. That leads to the code generation function below.

$$\begin{aligned}
code_{GPU}\ T &= T && (T \in \mathcal{T}_P) \\
code_{GPU}\ T &= T* && (T \notin \mathcal{T}_P)
\end{aligned} \tag{3.20}$$

Identifiers

For identifiers, the code generation itself is trivial. However, the management of the different variables is more complicated, as there are different variables that need to be correctly mapped. There are the parameters and local variables, which may simply be called, and there are the iteration variables of the domain iteration. For these variables, the appropriate local variable generated at the beginning of the kernel code generation needs to be chosen by using the *resolve* function.

$$\begin{aligned} code_{GPU}\ i &= resolve\ i && (i \in \mathcal{I}) \\ code_{GPU}\ i &= i && (i \notin \mathcal{I}) \end{aligned} \tag{3.21}$$

Literals

Code generation for literal values is straightforward. To avoid unnecessary casts it is important to add a suffix for the literals as necessary. The code generation function shown below reflects this.

$$\begin{aligned} code_{GPU}\ v &= v\,\mathrm{L} && (T_i = \text{long}) \\ code_{GPU}\ v &= v\,\mathrm{f} && (T_i = \text{float}) \\ code_{GPU}\ v &= v && (T_i \notin \{\text{long}, \text{float}\}) \end{aligned} \tag{3.22}$$

4. The Benchmark Suite

In order to make reliable predictions about the suitability of running a computation on the GPU, a model of the costs of the operations that can be performed is needed. There are several kinds of costs to consider, for example time spent on transferring the memory to and from the GPU, and the execution of the kernel itself, which should be decomposed into smaller units. In order to obtain these measurements, a benchmark suite was implemented. The design of that suite is explained in this chapter.

4.1. Configuration

The Benchmark suite comes with a CMake Configuration file `CMakeLists.txt`. CMake is a configuration tool that can generate project files or Makefiles for every major platform. The Benchmark suite has been tested with Mac OS X 10.9, GNU/Linux (Ubuntu 13.04), and Windows 8.1. The prerequisites to running the suite are as follows:

- An OpenCL compatible computing device. While CPUs are supported as well, a dedicated GPU is recommended.
- A runtime environment for OpenCL. For OpenCL capable GPUs, it is usually bundled with the display driver. On OS X, the runtime environment is provided by Apple.
- The runtime library for C++. On OS X and Linux, this will not be a problem, as they usually ship with `libstdc++.dylib` or `libc++.so`. On Windows the libraries `MSVCP120.dll` (Visual C++ 2012 Runtime library) and `MSVCR120.dll` (C Runtime Library) need to be distributed with the code.

The build process is backed by the CMake[1] configuration tool. To generate the platform specific build files, run `cmake <path to CMakeLists.txt>` from any folder. Usually this is done from an empty folder (`build`) inside the source directory. On Unix systems a makefile will be generated, and on Windows a Visual Studio project[2]. These can then be built in the usual way. There are some other requirements for building, however. The first is a compiler that is C++11-compliant. On Windows, the C++ compiler included in Visual Studio 2013 is required. On Linux and OS X, any gcc version above or equal to 4.7 or any clang version above or equal to 3.1 suffices. Additionally, Header files for OpenCL need to be installed. OS X provides them by default, on GNU/Linux there are usually packages available that contain them. On Windows, one installs the SDK that is compatible with the GPU with which the computer has been shipped.

[1] `http://www.cmake.org` CMake is a configuration tool that manages the build process in a platform independent manner.

[2] `http://www.visualstudio.com/` Visual Studio is an IDE for Windows that was developed by Microsoft

4.2. System Design

The Benchmark Suite is an object oriented C++ application. There are two components. One is responsible for driving the benchmark, while the other provides facilities for managing the data that is being collected by the benchmark driver component. Consider figures A.2 and A.3 for UML 2.0 class diagrams of the Benchmark Suite. The first diagram shows the general structure of the application, while the second one shows the different `KernelProvider` subclasses.

4.2.1. Driving the Benchmark

The benchmark is started from the `main` function of the application. From there, the user may specify which GPU is to be tested. Based upon the input of the user, a computing device is selected and the benchmarks are run. There are benchmarks to test the run time of empty kernels, time for memory accesses, basic operations and the influences arising from changing the work-group size. There are also benchmarks to measure the time spent on transferring memory to and from the device. The benchmarks themselves will be discussed in detail in chapter 5, where the runtime model is discussed.

Each benchmark has three components. The first one is an instance of a subclass of the `KernelProvider` class, which may provide one or more kernels to be executed, for example different basic operations, or different kinds of memory accesses. One may obtain the kernel using the `getKernelString()` method. The class also manages the execution of a single sample in the benchmark, essentially encapsulating all the functionality that is unique to a type of benchmark. For a single run that usually implies allocating the memory segment that is to be executed, creating and compiling the kernel and transferring memory to the GPU. Afterwards, a timed run is performed, the resulting memory segment is copied back to the device and the time spent on the execution is returned.

The second component is responsible for the orchestration of a single benchmark. Consider the control flow graph given in figure A.4 in the appendix. It illustrates the typical run of a single benchmark. On the top level, a value for the metric that is to be used is generated, and if the value is smaller then a certain threshold, a given number of samples will be executed using the `KernelGenerator`. After the samples are taken, the standard error percentage of the samples is computed. If the percentage is below a given threshold, the results will be stored and the next value for the given metric will be benchmarked. Otherwise, the process is repeated with a new batch of samples until the standard error percentage of all samples is below a given value. The current implementation uses the target standard error percentage of 2%. Once all values are benchmarked, the results are stored using the facilities explained in section 4.2.2.

The third component is the generator function, a function that takes as input the number of the current run and returns a value that is used as a metric for subsequent runs. Different benchmarks might differ in their requirements for the size of the metric, e.g. some might require a exponential or a linear progression of these, and the increase of the metric might differ as well.

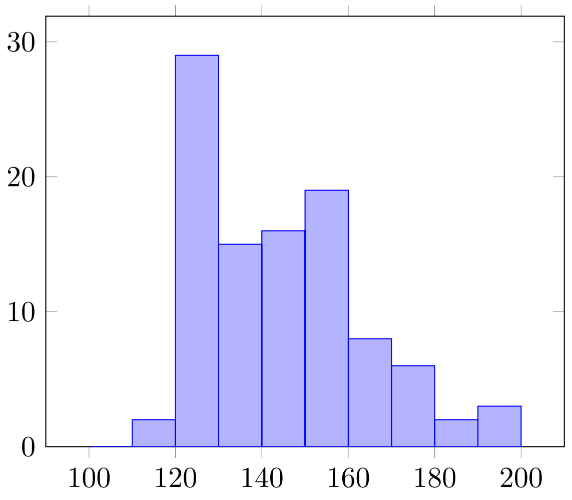


Figure 4.1.: Sample value distribution of a benchmark for a single size. It can be seen that the run time is not always the same, but spread out over an interval.

4.2.2. Collecting Measurements

A single benchmark unit is usually run multiple times. This is done since the execution time for a single sample is not always the same, but rather is spread over an interval. Figure 4.1 shows a sample distribution of run times. To store these values, the class `ResultAnalyzer` is used. It stores the values collected by the benchmark in a `std::vector` and calculates statistics. The `ResultAnalyzer` is able to calculate the arithmetic mean of the values, the standard deviation, and the Standard error percentage. The definitions for the arithmetic mean, the standard deviation and the standard error are given in the equations below. [14].

$$\mathrm{avg}(V) = \frac{\sum_{v \in V} v}{|V|} \tag{4.1}$$

$$\mathrm{stdDev}(V) = \sqrt{\frac{\sum_{v \in V}(v - \mathrm{avg}(V))^2}{|V|}} \tag{4.2}$$

$$\mathrm{stdErr}(V) = \frac{\mathrm{stdDev}(V)}{\sqrt{|V|}} \tag{4.3}$$

The *standard deviation* is a metric used to describe how the values are spread out over an interval. It is used to visualize this spread of execution times. It may also be interesting to see the approximate error of the average of the distribution. This metric is used to determine the point at which the mean value gathered by the suite is sufficiently precise, so that the execution of samples may be stopped. [14]

It is also possible to merge two `ResultAnalyzers` using the overloaded + operation. The resulting object has the values of both pre-existing analyzers. After a merge, the three

Metric	Runtime	StdDev	StdErr
v_1	r_{v_1}	s_{v_1}	e_{v_1}
v_2	r_{v_2}	s_{v_2}	e_{v_2}
$\vdots$	$\vdots$	$\vdots$	$\vdots$
v_n	r_{v_n}	s_{v_n}	e_{v_n}

Table 4.1.: Sample table generated by the class `TableGenerator`.

metrics will then have to be recomputed.

Usually, a benchmark will be executed with a variation in at least one metric, for example the size of the memory segment upon which the kernel is executed. For each value in the metric a table is generated. It will contain the size of the metric, the average runtime, the standard deviation and the standard error. Such a table is depicted in table 4.1. The functionality for this is located in the class `TableGenerator`.

In some cases another dimension is needed, in order to group a set of related benchmarks together. This is be the case, e.g. when looking at the basic operation types `+`,`-`,`*` and `/`. The class `AccumulatedReportGenerator` provides this functionality. It acts as a container for `TableGenerators`, and is able to produce tables of tabulator-separated values. These tables can be simply imported into data visualization software, e.g. SciDAVis[3].

[3]`http://scidavis.sourceforge.net` SciDAVis is a free data analysis and plotting tool.

Part III.

Runtime Model and Results

5. A Runtime Prediction Model

In this chapter, a runtime model for the execution of domain iterations on the GPU will be established. There are two components that need to be modeled for the runtime model. The first is the overhead for the transfer of memory to and from the GPU. For many computations, this will be a bottleneck, and the part of process the majority of the run time is spent on. It is discussed in detail in section 5.1. The other part of the model deals with the execution of the kernel of the kernel on the GPU. Here, a number of parameters influence the execution time. These are discussed in section 5.2.

5.1. Memory Transfer Costs

Data for GPGPU computations need to be copied from the host to the device and back again. This process causes a significant runtime overhead, as the CPU and the GPU typically do not share the same memory resources. Hence, memory will need to be copied back and forth. In modern PCs, GPUs are typically connected to the CPU via the *PCI Express 2.0* BUS[1]. This bus offers a peak transfer performance of $1\frac{GBit}{s}$ per lane. Typically, GPUs have 16 lanes available to them, in case of Multi-GPU systems or notebooks, only 8 lanes might be available for each GPU. For the latter[2] case, this amounts to a theoretical performance of $1\frac{GB}{s}$. The RAM of the PC is also directly connected to the CPU. In the case of DDR3-SDRAM, the current state of memory technology, the theoretical maximum bandwidth would be between $6.4 - 17.1\frac{GB}{s}$. This effectively makes the PCI Express bus a bottleneck when only considering the bandwidth. [12, 19]

Another point to consider is latency. The signals on the BUS systems travel with a high but finite velocity. For printed circuit boards, this velocity is around half the speed of light, or about $150\frac{mm}{ns}$. With modern CPU and GPU architectures having clock rates of several GHz, several cycles may be needed for signals holding the request for data to reach the memory, and several more for the data to reach the GPU. Added to that is the is the latency of the memory module. For smaller memory segments, that combined latency is the defining factor in determining the time spent on copying memory. Measurements supporting this assertion can be found in [5] and in the measurements below in figure 5.1. [2, 5]

$$t_{trans}(x) = b^{-1} * x + l_{prop} \tag{5.1}$$

[1] For example in the Chipset Architecture of the X79 Intel chipset [12]

[2] The author of the thesis performed most measurements using a Notebook GPU, hence it will be used for the examples given here.

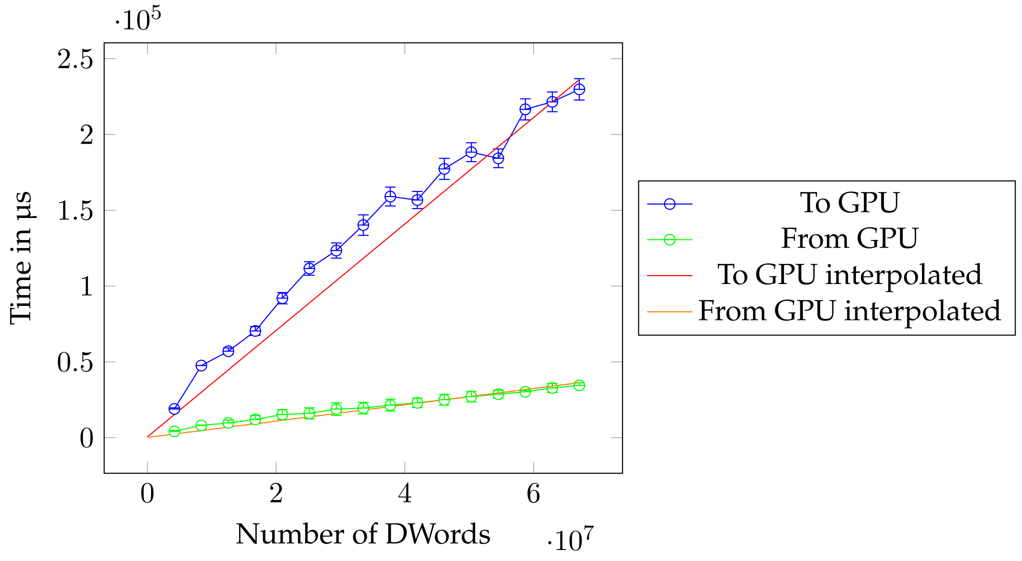


Figure 5.1.: Transfer times for memory. Time spent copying to the GPU is shown in blue, and time spent copying from the GPU is shown in green.

The structure above makes it a reasonable assumption that the variable is the amount of memory copied to and from the device. The propagation latency l_{prop} is only observed once per transfer, which makes it a constant offset. The reciprocal of the bandwidth b determines how many seconds it takes to copy one byte. It needs to be multiplied by the number of bytes copied. The result of this is described in equation 5.1. To test this claim, a benchmark that copies data to and from the GPU was performed, and the time spent on both transfers was recorded. The results of these measurements are shown in figure 5.1. It may be seen that there is significantly more time spent copying data to the GPU than on copying the results back again, although the size of the data remains the same. This effect is also referred to in [5]. The details of the GPU architectures and the runtimes are not documented, however, and hence no explanation for the cause of the effect can be given.

From the values of figure 5.1, formulae for the transfer times of host-to-device (equation 5.2) and device-to-host (equation 5.3) can be derived. For the test device, a linear interpolation yields the following data:

$$t_{\rightarrow GPU_{rMBP}} = 0.00357997\mu s * x + 516.413\mu s \tag{5.2}$$

$$t_{\leftarrow GPU_{rMBP}} = 0.00053337\mu s * x + 173,626\mu s \tag{5.3}$$

Equations 5.2 and 5.3 are also shown in figure 5.1. One might observe that the linear structure of the model described above fits well with the data gathered experimentally.

5.2. Base Cost for Running Kernels

As previously noted, computations on the GPU are performed in a massively parallel fashion, with each work item being computed in its own thread. These threads need to be started, categorized and put into thread blocks, which might cause a noticeable amount of overhead. To measure the costs, an empty kernel that only queries its own id is executed, with a different number of work items being tested. The kernel can be seen in B.1.1. It is basically an empty kernel, except for the function call to inquire the id of the work item the kernel is executed on. This operation has to be performed whenever a kernel is executed, as it has to be used to determine which element in the memory should be accessed.

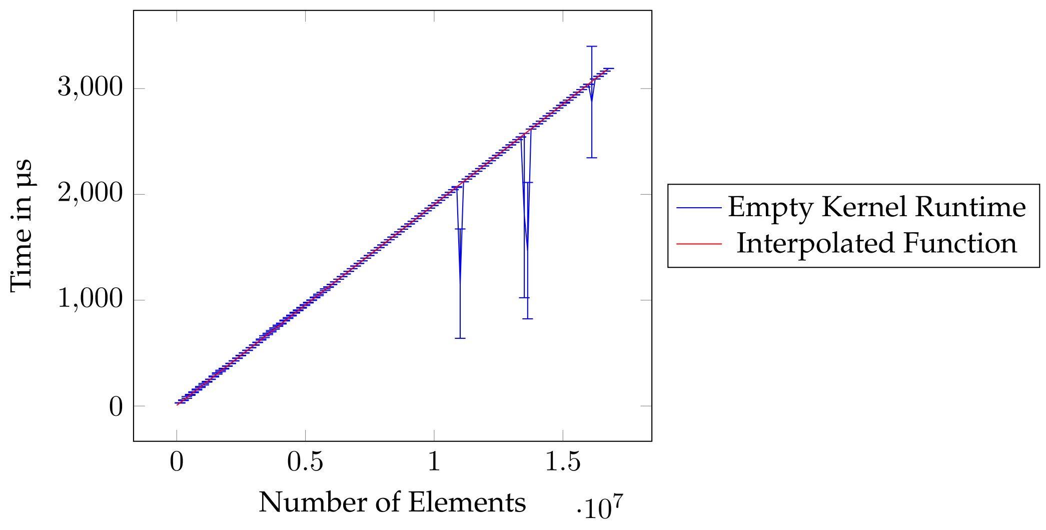


Figure 5.2.: Execution time for empty Kernels. The experimentally gathered data points are shown in blue, and the Interpolated Function is shown in red.

As it can be seen here, the runtime scales linearly against the number of work items that are executed on the GPU. With that in mind the function T_{Base} (see equation 5.4) for the runtime of empty kernels may be assigned for the run time of empty kernels. Equation 5.5 shows a sample assignment of the values based upon the results of the benchmarks executed on the GT-650M.

$$T_{Base}(x) = t_{Element} * x + c \tag{5.4}$$
$$T_{Base_{rMBP}} = 0.18988566ns * x + 5.72652\mu s \tag{5.5}$$

5.3. Influence of the Work-Group Size

In OpenCL, work items are grouped into so-called *work-groups*. Each work-group has the same number of work items. The total number of work items has to be divisible by the work-group size. Work-groups are executed in sequence. Once all work items within a work-group have finished executing, the execution of work items belonging to the next work-group will be started. From this follows that the work-group needs to be sufficiently large in order to utilize all the available processors on the GPU. Additionally, if the number of work items within a work-group is sufficiently large, the scheduler will employ techniques to hide the latency of loading data from the off-chip global memory. [8, 17]

To evaluate this behavior, another benchmark has been implemented. In this case, the kernel in question (shown in section B.1.2 in the appendix) is executed with a varying work-group size on an array with roughly 2^{20} elements. The imprecision is due to the fact that the work-groups need to be an even divider of the total number of work items. At that number of elements the amount of memory will vary by only 0.1%, which makes the impact of the memory size negligible.

The data obtained by executing the test, shown in figure 5.3 validates the claims made above. For the GPU the benchmark was executed on, there is a severe speed penalty when working with a work-group size below sixteen. In the worst case with a work-group only containing a single work item, the execution time is higher by a factor of fifty compared to using the maximum number of work-items in a work-group. With sixteen to one hundred work items in a work-group, there is still a significant decrease in execution time for bigger work-groups. Above this, no more significant increases in execution speed were observed.

From the data obtained, the formula given in equation 5.6 can be deduced. It is a sum of two inverse exponential functions. The subexpression denoted as **A** marks the influence resource utilization has on the program. The subexpression denoted as **B** shows a decline that is less steep than the first. It models the influence the scheduler has on the execution time. Equation 5.7 shows a concrete assignment of the variables using the data obtained from the benchmark.

$$M_{WG}(x) = \underbrace{B_1 * e^{\frac{-x}{t_1}}}_{\text{A}} + \underbrace{B_2 * e^{\frac{-x}{t_2}}}_{\text{B}} + y_0 \tag{5.6}$$

$$M_{WG_{rMBP}}(x) = 13.04 * e^{\frac{-x}{12.21124}} + 26.10 * e^{\frac{-x}{2.28329}} \tag{5.7}$$

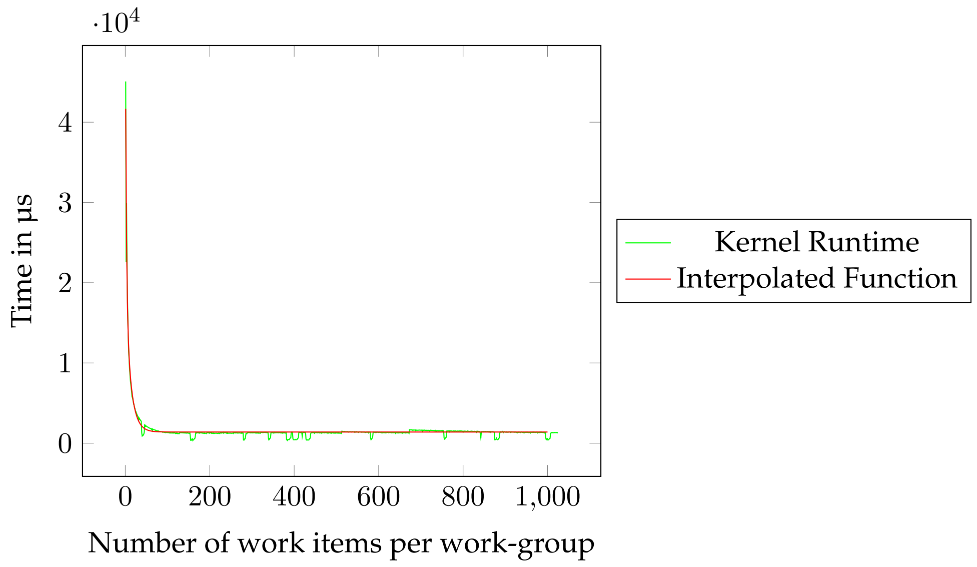


Figure 5.3.: Execution time for different work-group sizes. The experimentally-gathered data is shown in green, the Interpolated Function in red.

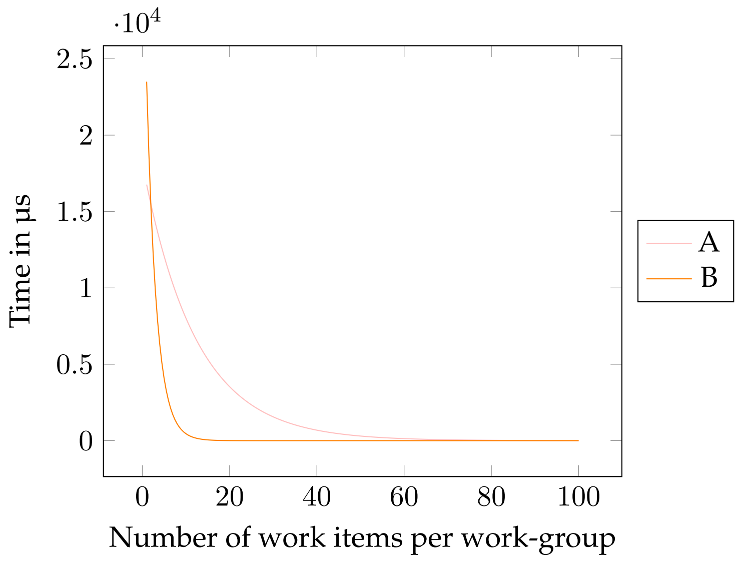


Figure 5.4.: The two different components of the function 5.6 derived from the data given in figure 5.3.

5.4. Basic Operations

The four basic operations, $+$, $-$, $*$ and $/$, are discussed here as applied to various types of data, such as floating point values and integers. The documentation of the NVidia OpenCL platform gives some estimates about the projected performance, specifying how many operations can be completed per clock cycle. This metric is established by looking at a single SIMT-Element (warp) with 32 elements, which synchronously executes one instruction comprising of 32 operations. One may examine the number of cycles a warp takes to be executed, and hence deduce the operations throughput from this. [17]

According to the documentation, for Additions, Subtractions and Multiplications of floating point numbers, throughput is identical at **8** operations per clock cycle. Division should be slower, at **0.88** operations per cycle. For integer arithmetic, Addition and Subtraction are at **8** operations per cycle. Multiplication of integers should be slower, at **2** operations per cycle, and Division and Modulo are slower again. In case of literals, the compiler will try to replace the Division and Modulo by Bit-Shifts or Bitwise AND-Operations. A single Bit-Shift or AND-Operation should again be at **8** operations per cycle. The actual performance benefit depends upon the literal.[17]

To evaluate these claims, another benchmark was implemented. The variable, once again, is the size of the memory segment that is iterated upon. Tests are conducted for the four basic operation types, both for floating point and integer arithmetic. This alone would not suffice to deduce the time spent on the operations, however, as it is not possible to write a kernel that only performs a single basic operation. For example, there always is the base cost discussed earlier, and a memory accesses to save the result is needed so the compiler does not eliminate the operation. To mitigate this, first a benchmark that only performs a read and a write was executed, and the runtime difference between that one and the benchmark with the basic operations is deducted to be the time spent on the operation. The kernels used for the benchmark are shown in section B.1.3 in the appendix.

The data from the benchmark (figure 5.5) shows that for floating point arithmetic, only use of the division causes a significant computational overhead. The other operations seem to be behaving identically, causing next to no overhead at all. This is more or less in line with the NVidia OpenCL programming guide[3]. However, the evaluation of the model revealed the costs spent on the division operation to be smaller in most cases. This effect is discussed in section 7.1.

For integer operations (figure 5.6), the division is also the slowest operation. The other operations are running at a similar speed. A speed difference between Additions and Subtractions on the one side, and Multiplication on the other could not be reproduced with this benchmark. It was observed that the floating point division is much slower then its integer division counterpart, presenting a stark contrast to the information given in the NVidia OpenCL programming guide.

[3]see [17]

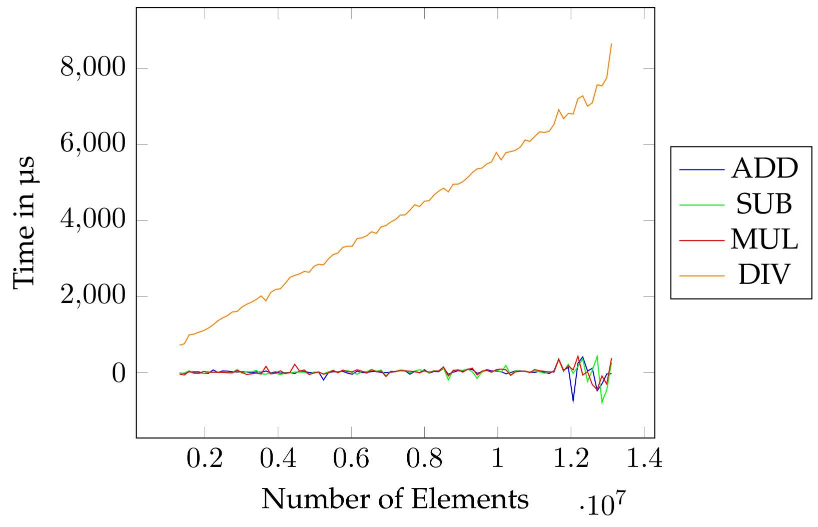


Figure 5.5.: Execution time for basic operations on floating point values. The Standard Deviation is omitted from this diagram for legibility reasons. For diagrams containing the Standard Deviation, see figure A.5 (Addition), figure A.6 (Subtraction), figure A.7 (Multiplication) and figure A.8 (Division) in the appendix.

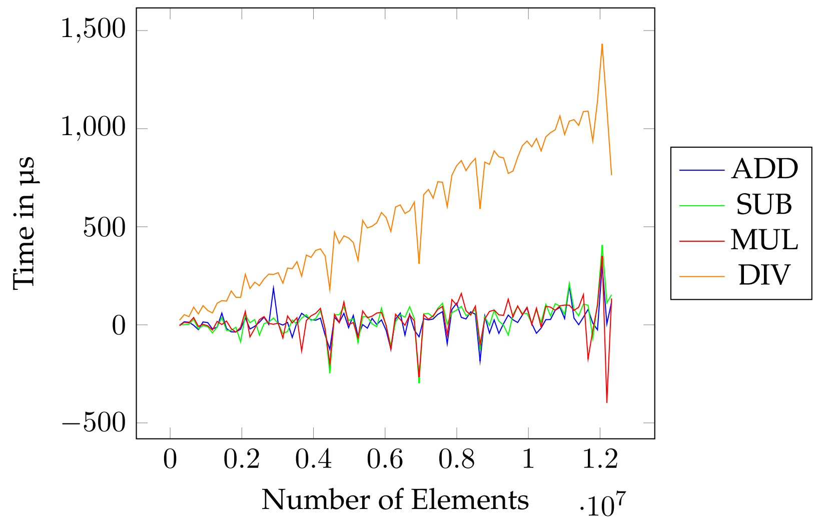


Figure 5.6.: Execution time for basic operations on integer values. The Standard Deviation is omitted from this diagram for legibility reasons. For diagrams containing the Standard Deviation, see figure A.9 (Addition), figure A.10 (Subtraction), figure A.11 (Multiplication) and figure A.12 (Division) in the appendix.

Nevertheless, the run times for each operation are observed mostly to scale linearly with the number of elements that have to be computed. Hence, the equations that model the runtime were extracted from the data of the benchmarks, yielding the equations 5.8 as a result.

$$\begin{aligned}
T^{Op}_{type}(x) &= t_{Element} * x + c \qquad (5.8)\\
T^{+}_{float}(x) &= 3.2679908306701ps * x\\
T^{-}_{float}(x) &= 4.42040597789466ps * x\\
T^{*}_{float}(x) &= 4.44336371720927ps * x\\
T^{/}_{float}(x) &= 0.000586512010350116\mu s * x + 2.28467688281392\mu s\\
T^{+}_{int}(x) &= 3.28593145810764ps * x\\
T^{-}_{int}(x) &= 6.1655964896141ps * x\\
T^{*}_{int}(x) &= 4.83050202348605ps * x\\
T^{/}_{int}(x) &= 8.97819795983561 * 10^{-2}ns * x + 3.02532315741367\mu s
\end{aligned}$$

5.4.1. Multiple Basic Operations Per Kernel

Tests conducted while just using the results from the previous sections and scaling them linearly did not produce accurate predictions. This became especially apparent on tests performed with kernels that contain multiple floating point divisions, as they have the most significant computational overhead. To investigate this effect, another benchmark was devised. This time, the size of the memory segment is held the same, at 256kB, and the number of operations that are to be executed on the data is varied. The kernel is shown in section B.1.3 in the appendix.

For integer arithmetics, the benchmark (see figure 5.7) reveals that for the basic operations, no noticeable difference was observed in how many operations of a particular type are performed in a kernel. A notable exception is the division operation. The compiler seems to be able to infer that if there are a certain number of divisions by a literal performed in sequence on the same number, the result will always be zero. In this case it can be eliminated[4].

Floating point arithmetic behave slightly differently. For Addition, Subtraction and Multiplication no increase in run time may be observed for the first few operations, and after that a slight increase with each added operation can be observed. That delay before the increase may be due to the program being memory-bound. With few operations, these can be executed whilst other threads are waiting for the memory access to be completed.

Divisions behave more erratically. Figure 5.9 shows that the first two operations seem to adhere to the cost model deduced and shown in section 5.4, but afterwards they start

[4]In the benchmark, divisions by seven are used, and $c \in \text{int} \implies c/7^{12} = 0$. Also, 7^{12} does not overflow, as the operations are applied sequentially.

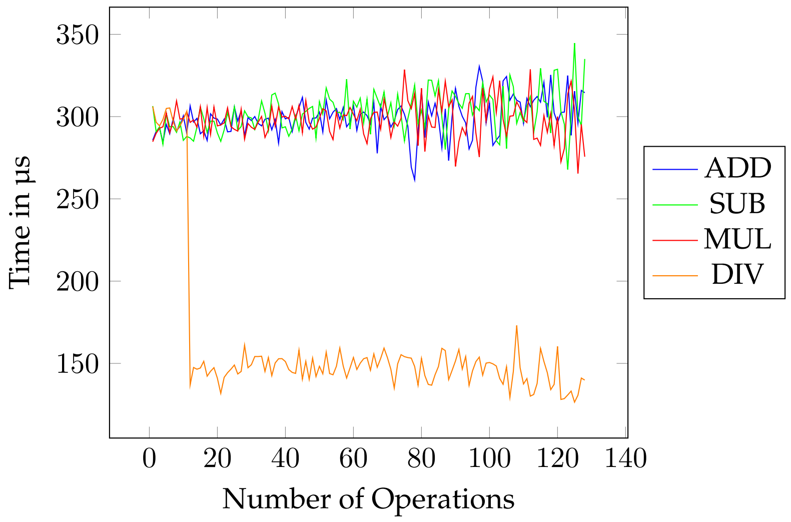


Figure 5.7.: Total execution time for multiple basic operations on integer values. The x axis denotes the number of operations of a single type that are executed within a single kernel. The Standard Deviation is omitted for legibility reasons. In this case, it is fairly large, about $\pm 50\mu s$ for the memory size of 256kB.

taking longer to complete, and then, subsequent to 64 operations, they become even more expensive than before. The first two operations can be explained with the memory read shadowing the cost of the operation. The real cost of the operation becomes apparent afterwards. The second inflexion is not clear, however.

For the Addition, Subtraction and Multiplication, the formula 5.9 can be derived. It takes the number of operations as input, and returns a multiplier to be applied to the linear runtime in order to get the final runtime. For x values smaller than a threshold, the formula will return a multiplier that returns the runtime of a single operation. If x is above the threshold, the formula will return the multiplier that takes the shadowed operations into account. This multiplier will gradually get closer to one as the number of operations that are shadowed, x_{sat}, gets smaller compared to the total number of operations performed in the kernel.

$$M(x) = \begin{cases} \frac{1}{x} & (x \leq x_{sat}) \\ \frac{x_{sat}-x}{x} & (x < x_{sat}) \end{cases} \tag{5.9}$$

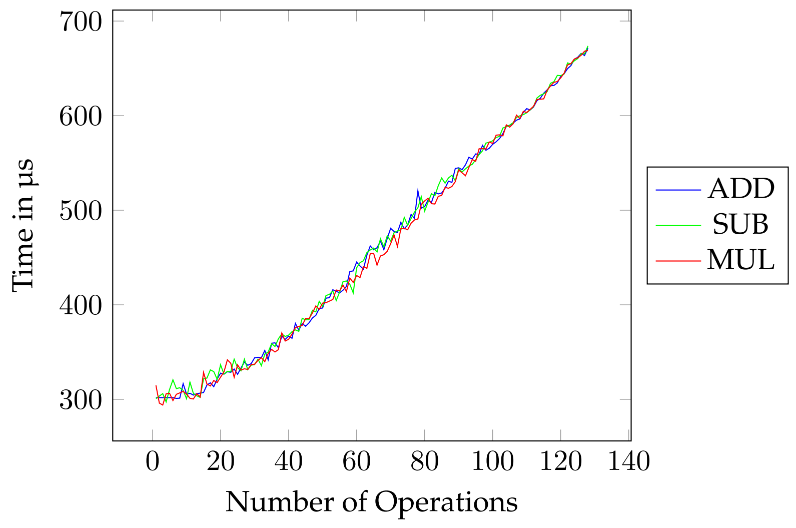


Figure 5.8.: Total execution time for multiple basic operations (ADD, SUB, MUL) on floating point values. The x axis denotes the number of operations of a single type that are executed within a single kernel. The Standard Deviation is omitted for legibility reasons. In this case, it is fairly small, about $\pm 10\mu s$ for the memory size of 256kB.

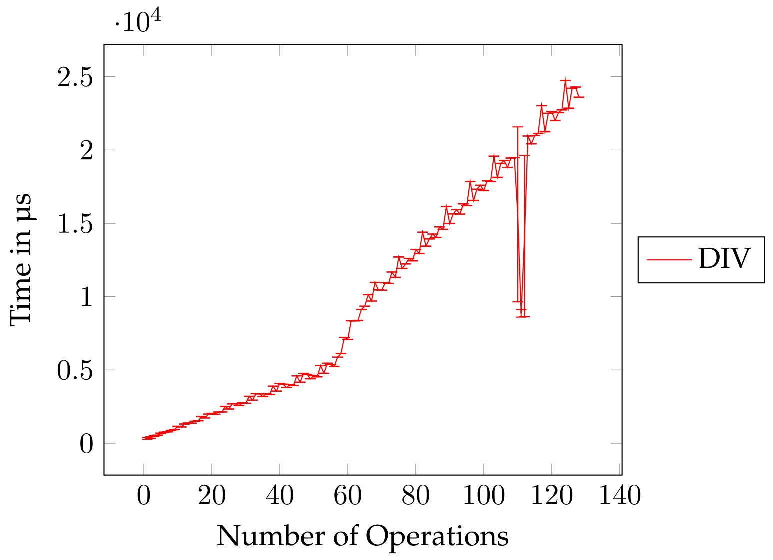


Figure 5.9.: Total execution time for multiple basic operations on floating point values. The x axis denotes the number of divisions that are executed within a single work item.

5.5. Memory Accesses

As already discussed in section 2.1.3, there is a memory hierarchy in OpenCL. Recall that there are four different kinds of memory, *private*, *local*, *constant*, and *global*. Global memory is most expensive, with access incurring a cost of about 300 cycles. It is shared amongst all work items. Local memory is shared amongst the work items belonging to the same work-group. It typically offers a faster access speed compared to global memory. Private memory is used for local variables, and is private to one work item. It is typically very fast, most likely implemented as a register. The initial Benchmark uses a benchmark without a memory access as the base line, and compares the times spent on the private access, the local access and the global access. Again, the variable is the size of the array the kernel is executed upon. [18]

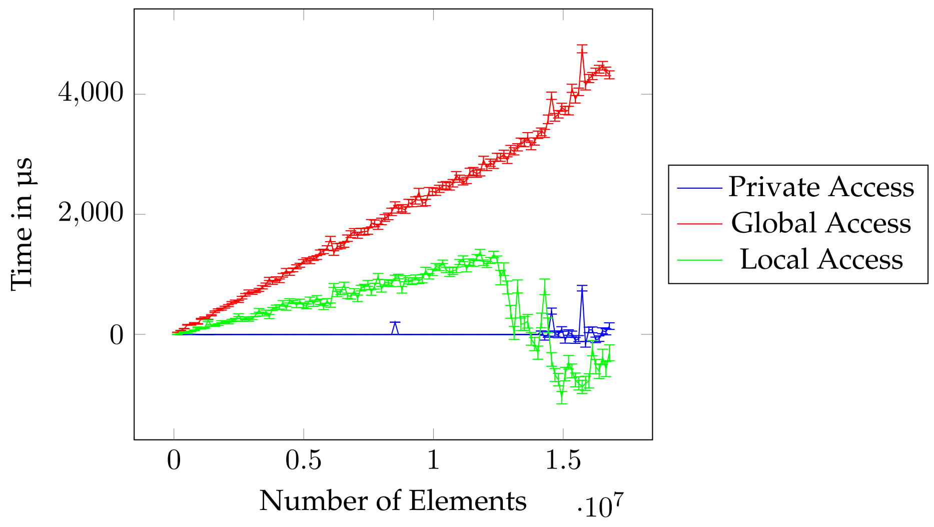


Figure 5.10.: Execution times for memory access operations. This is the first benchmark, comparing private, local and global accesses.

The first conclusion drawn from the benchmark (see figure 5.10) is that a private memory access is de-facto free, yielding a result shown in equation 5.10. Global Accesses are scaling linearly according to input size. The costs of a global memory access can be calculated as described in equation 5.11. Local accesses also scale linearly according to input size. However, their access speeds are significantly faster, with them requiring only about half of the time global accesses need to complete. Equation 5.12 describes how to describe the time spent on a local access operation for a kernel with x work items.

$$T_{private}(x) = 0s \tag{5.10}$$
$$T_{global}(x) = 0.247444074967131ns * x \tag{5.11}$$
$$T_{local}(x) = 0.1038783725ns * x \tag{5.12}$$

Unfortunately, this first approach did not yield accurate results with real world data. Thus, several modifications to the model had to be made. The first one was the classification of global memory accesses into several categories, measuring their performance individually (see section 5.5.1) and the second one was to look at the behavior when there are multiple memory accesses in one kernel (see 5.5.3).

5.5.1. Memory Access Classes

The device the benchmarks are conducted on was a NVidia GPU with a Compute Capability Level of 3.0, which in itself has certain implications. There are two cache levels, L1 being exclusive to each processor, and L2 being shared amongst all of them. There is also a read-only cache which is used to speed up read-only accesses. There are also problematic kinds of accesses, e.g. if an address of a memory access falls into more then one bank, then these requests have to be fulfilled sequentially. [17]

To test the behavior, a series of benchmarks were performed. All of them were performed with a varying memory size. The kernel that was used in the benchmarks can be found in section B.1.4. The following memory accesses were performed in the kernel:

- **Constant Address Read:** In this test, a global read is performed, with each work item accessing the same address in memory. (`output[x] = input[0x3ff];`)
- **Interval Address Read:** In this test, a global read within a small segment of the global memory is performed. This is done by zeroing away the higher bits of the work item id with an arithmetic and operation. (`output[x] = input[x & 0x3ff];`)
- **Continuous Address Read:** In this test, the global read is performed on the whole memory segment, with each segment accessing the element that corresponds to its work item. (`output[x] = input[x];`)
- **Constant and Interval Address Read:** This test combines the first and the second test. There is a constant address memory access and one that is within a fixed interval. (`output[x] = input[x & 0x3ff] + input[x & 0x3ff];`)
- **Constant and Continuous Address Read:** In this test, two reads are performed, the first with a constant address shared by all the work items, and the second read with a continuous access. (`output[x] = input[0x3ff] + input[x & 0x3ff];`)
- **Two Equal Continuous Address Reads:** This test performs two reads as well. In this case both accesses have the same address. They are also both continuous accesses. (`output[x] = input[x] + input[x & 0x3ff];`)
- **Two Different Continuous Address Reads:** The last test also performs two continuous reads. However, this time the addresses are not the same, with the second access being bit-shifted once to the right. (`output[x] = input[x] + input[x>>1]`)

The result of the benchmark (see figure 5.11) offers further insights. The first of which is that accesses to constant addresses are significantly faster then continuous accesses, which indicates that the data can and will be held in a cache. Caches offer access speeds that are an order of magnitude slower than registers, but, depending on the cache level, several orders of magnitude faster then global memory. Accesses will also be cached if they are contained in an interval that fits the size of the cache. Continuous accesses are the test case described as global access in section 5.5. They are considerably slower. However, when there are more then one accesses to the same address performed within a single kernel, the cache also is utilized, making the second access to the same address essentially without cost. The other test with multiple continuous accesses indicates that a second memory access is cheaper than the first one. This is discussed in section 5.5.1.

In some automated tests[5], however, the runtime deviated significantly[6] from the predictions made using the model, discussed above. This behavior seems to correlate with the presence of memory accesses with a particularly complex access pattern.

To evaluate this, a number of kernels that contained memory accesses with randomly generated index expressions were executed. The result can be seen in figure A.13 in the appendix. For the test the kernels were sorted by the number of nodes in the index expression tree. They ranged from one to twenty nodes, with twenty kernels for each. It may be observed that there were a number of kernels that take less than two hundred microseconds to complete, with most of those being kernels with a low number of nodes. Other kernels took significantly longer, with execution times of several milliseconds. These memory accesses may not have been able to utilize the cache, just as the scheduler may not have been able to hide the latency of loading values from memory when the accesses occur in a nonlinear fashion. This kind of memory access will henceforth be referred to as a **complex memory access**.

5.5.2. Classifying Memory Accesses

The memory accesses tested in section 5.5.1 represent different classes of memory accesses. For predictions to be reliable, there also needs to be a way to classify memory accesses into these categories as they are encountered. Practically, this is implemented in the FunkyIMP compiler using the funkyIMP array type system with an instance of the Visitor pattern iterating over the kernel syntax tree and collecting the data from it.

The data consists of the number of nodes in the syntax tree, and indicators for circumstances that make a memory access complex. It also detects cases where the classification cannot be performed statically, for example when the address being accessed is based on the result of a function call or a memory access. Additionally, the Visitor collects the identifiers that are used in the addressing. In section 2.2 it was noted that FunkyIMP supports multi-dimensional array types and iterations over them. In the body of the domain iteration each dimension has its identifier, and together they may be used to access the current

[5] See chapter 6

[6] The actual runtime was more than twice as high as anticipated.

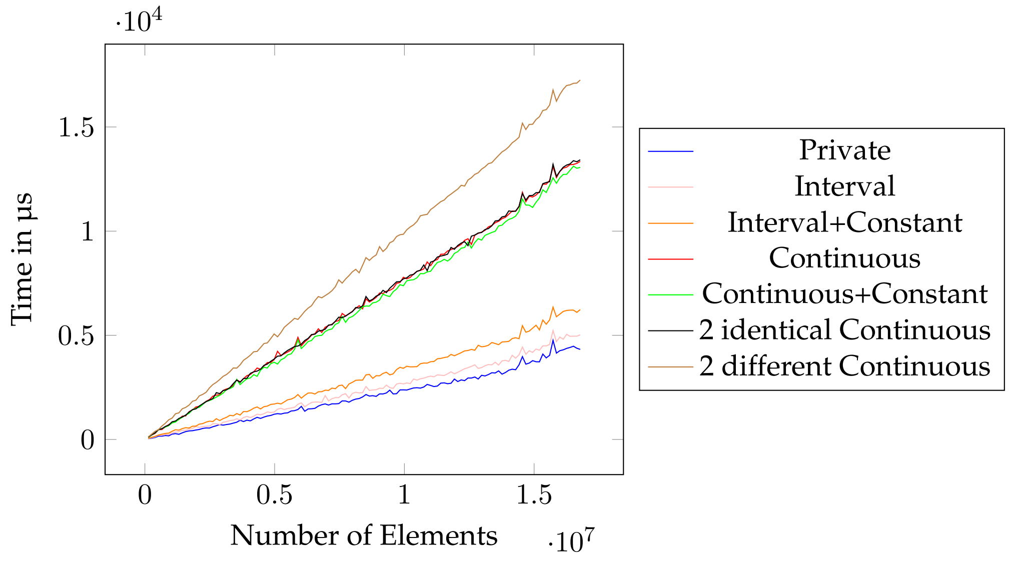


Figure 5.11.: Execution times for memory access operations. This is the second benchmark, comparing different kinds of and global accesses. The Standard deviation (Comparatively small with $\sim 2\%$) is omitted for legibility reasons. All operations contain a write to global memory, the first operation is given as a reference.

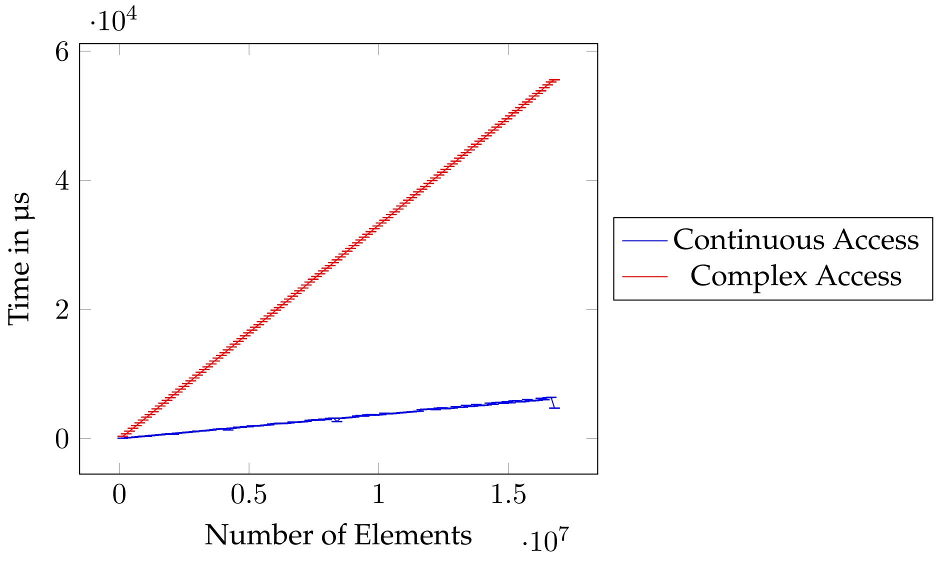


Figure 5.12.: Execution times for memory access operations. This benchmark compares a normal continuous access with a complex one. The kernel with the complex access can be seen in section B.1.4 in the appendix.

$$\begin{aligned}T_{const}(x) &= 45.9504ps * x\\ T_{ivl}(x) &= 45.9504ps * x\\ T_{cont}(x) &= 0.521932ns * x + 3\mu s\\ T_{complex}(x) &= 3.5202282ns * x + 3\mu s\end{aligned}$$

Figure 5.13.: Functions specifying the Runtime for memory access operations. The first one is for constant address accesses, the second one for accesses that are limited to small intervals, the third for continuous accesses. The last one is for accesses that are designed to be as random as possible, to avoid optimizations, and represent a worst-case scenario.

element.

These identifiers are the only values that differ from iteration to iteration. The classification of the memory access therefore depends solely upon their position in the index expression. The simplest case is in the total absence of any identifier from the iteration whatsoever. In such a case, all work items would access the same element in memory, making this a *constant access*. Otherwise, the sizes of the dimensions that are dynamically accessed are multiplied, yielding the number of elements that are accessed in the course of the iteration. If that value is below a certain threshold (corresponding to the GPU cache size), the memory access is then classified as a *cached access*.

For memory accesses that do not fit the categories described above, the classes *normal access* and *complex access* have been introduced. Complex accesses occur if the addresses accessed vary greatly, and the accesses are not conducted in a linear fashion. This is the case for index expressions that are very complex[7] or expressions that use an identifier for a dimension that is more fine-grained than the current one. Otherwise, the read is classified as *normal read*.

5.5.3. Multiple Memory Accesses in one Kernel

For basic operations (see section 5.4), experiments showed that they do not necessarily behave linearly as regards to the number of times they appear in a work item. It is not unreasonable to assume similar behavior with memory accesses. Hence, another benchmark was implemented. This benchmark evaluates the behavior when there is more then one memory access within a single work item. It is executed with 2^{20} elements. The kernel used in the benchmark is displayed in section B.1.4 in the appendix. The kernel calculates the sum of multiple memory accesses, as shown below.

$$\text{memory}[work\_item] \leftarrow 42 + \sum_{k=0}^{n} \text{memory}[work\_item + k]$$

[7]Which means a complicated syntax tree that cannot be simplified by the compiler.

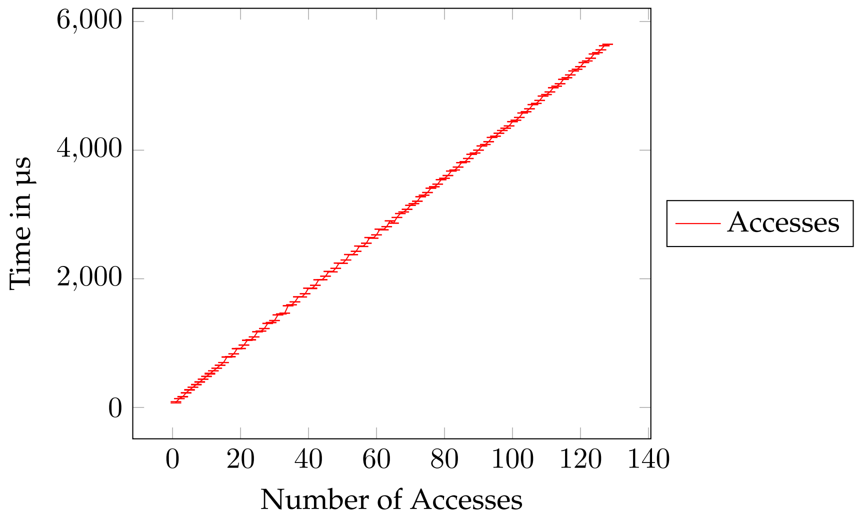


Figure 5.14.: Total execution time for multiple memory accesses from global memory. The x axis denotes the number of accesses that are executed within a single kernel. The accesses are of the scheme `memory[work_item + k ]`.

The benchmark revealed that the runtime grows linearly with the number of memory accesses, however not with the rate predicted in section 5.5.1. Instead there is a fixed cost for the first access, with a lower cost occurring every time the memory is accessed. This can be described by the following formula:

$$M(x) = \frac{1}{x} + 0.17 \tag{5.13}$$

6. Results and Evaluation

In this chapter, the results of this thesis are presented and evaluated. the chapter is divided into three sections. Section 6.1 serves as an introduction and discusses the tools and configurations that the other two sections are based upon. In section 6.2 the results of the thesis are described, namely the analysis built using the runtime model described in chapter 5 and the predictions made using that analysis. In section 6.3 the quality of the predictions made by the analysis is evaluated and discussed on a quantitative scale.

6.1. Preamble

6.1.1. Generation of Random Domain Iterations

For the tests conducted in 6.3, randomly generated domain iterations were used. A Java utility[1] was built to generate these. Domain iterations are generated randomly, and embedded into the template shown in section B.3.2 in the appendix. The resulting class acts as the test driver for the individual test cases, executing a single domain iteration with different memory sizes ranging from 2^{10} elements to $2^{26} (= 67108864)$ elements.

The utility generates the expression tree that forms the body of the domain iterations recursively. The tree consists of several different subtypes of expressions, as listed below.

- **Literal Expressions** hold floating point literals, e.g. `5.3f`.
- **Identifier Expressions** are expressions consisting of an identifier that contains a private variable, e.g. `x`.
- **Simple Array Access Expressions** hold a simple array access to the memory address that matches the current work-item, e.g. `matrix[x,y]`.
- **Random Array Access Expressions** hold an arbitrary array access. This is a recursive type, with expression trees generated recursively for each dimension. These may consist of identifiers, literals, or binary expressions. To avoid negative values and divisions by zero, there can be no subtractions and no divisions. Additionally, to avoid out of bounds array accesses, the accesses must be taken modulo the array bounds, e.g. `matrix[342%HEIGHT, (x*2)%WIDTH]`.
- **Local Array Access Expressions** hold an arbitrary access to an array that is located in local memory. The index expression is generated recursively. The same restrictions as for the index expressions for random array accesses apply. To avoid out of bounds accesses, only the seven least significant bits will be considered. (e.g.

[1]Included in the funkyIMP repository. Requires Java 8

`xxx[(x*23)&0x7F]`) The indexed array has to have the same name that is passed to the funkyimp compiler using the `LOCAL_HACK` switch. When the argument is set, the compiler will convert accesses to the identifier specified in the argument. (e.g. `-LOCAL\_HACK "xxx"`)

- **Binary Expressions** each hold a basic operator ($+$, $-$, $*$, $/$) and two subexpressions, e.g. `3.434 - (x/y)`

Whenever a new tree is generated, it is created with a random type. Each type of tree is assigned a probability. Binary expressions have a probability of 0.5, all other expression types a probability of 0.1. To avoid the expression trees from growing too large, and hence causing the stack to overflow, the number of nodes in a tree is capped at three hundred.

The code generator may be configured by entering values in a standard Java properties file, and passing it as a command line parameter for the utility. One may choose whether to allow divisions and complex memory accesses, and to limit the number of nodes in the generated domain iterations with a lower and an upper bound. One may also specify the number of examples that are generated, and the directory they are written to. For a description of the keys used in the configuration file, see section D.1 in the appendix.

6.1.2. Setup for Automated Testing

Test classes are created automatically with the help of a shell script that uses the generator and template files to create test cases. It may be called by executing the command `./generate.sh` in the `AutomatedTest` directory in the funkyIMP repository. The script runs the Code generator with a specified number of examples, and moves each test case to its own folder, adding in a copy of a Makefile and a template for the result table.

The test cases are executed by calling `./execute.sh` in the automated test directory. The script recurses into each of the generated directories, invoking `make clean` to remove any previously generated files. Afterwards it invokes `make` to compile the funkyIMP source. The `Makefile` is configured to compile the code in a way that causes the program to be executed five times, with run times being recorded for each run. In the course of the compilation the runtime for the kernel and the time spent on memory transfer will be predicted. The script captures and saves these times into a file. Next the newly generated executable is started. It prints the average runtime and the standard deviation between the different runs before terminating. This information is collected, and, finally, everything is merged into a single CSV file.

6.1.3. The Platform

The model has been developed and tested on a MacBook Pro with an Intel Core i7-3740QM processor, 16 GB of RAM and on OS X 10.9. This notebook has a dedicated, *Kepler Architecture* based, NVidia GT-650M GPU featuring 1024 MB of exclusive video memory, and two compute cores with a combined pipeline width of 384. It is clocked at 900 MHz.

6.2. Results

6.2.1. A Runtime Prediction Analysis

In chapter 5, a model of the execution time for kernels executed on the GPU was presented. The model was used to create an analysis that is able to predict execution times. It is presented here in this section. Let $\mathcal{C}$ denote the set of all cost types that are considered in the analysis. The update operator $\oplus$ is used to increment the number of occurrences of a particular type in a set of $\mathcal{C} \times \mathbb{N}$ tuples. The set update operator $\uplus$ denotes a shorthand for applying the update operator on all elements of the left-hand side to the right-hand side. The definitions are shown below.

$$\begin{aligned}\mathcal{C} = \{&+_{int}, -_{int}, *_{int}, /_{int}, +_{float}, -_{float}, *_{float}, /_{float}, BASE\} \\ &\cup \left\{A_{Private}, A_{Local}, W_{Global}, R^{Constant}_{Global}, R^{Cached}_{Global}, R^{Complex}_{Global}, R^{Continuous}_{Global}\right\}\end{aligned} \tag{6.1}$$

$$\begin{aligned}\oplus :&\quad \mathcal{C} \times \mathbb{N} \to \mathcal{P}(\mathcal{C} \times \mathbb{N}) \to \mathcal{P}(\mathcal{C} \times \mathbb{N}) \\ (c,n) \oplus M =& \begin{cases}\{(c,n)\} \cup M & ((c,\_) \notin M) \\ \{(a,b)|(a,b) \in M \wedge a \neq c\} \cup \{(c, n+n')|(c,n') \in M\} & ((c,\_) \in M)\end{cases}\end{aligned} \tag{6.2}$$

$$\begin{aligned}\uplus :&\quad \mathcal{P}(\mathcal{C} \times \mathbb{N}) \to \mathcal{P}(\mathcal{C} \times \mathbb{N}) \to \mathcal{P}(\mathcal{C} \times \mathbb{N}) \\ \emptyset \uplus M &= M \\ \{(a_1,n_1),(a_2,n_2),...(a_k,n_k)\} \uplus M &= (a_1,n_1) \oplus (\{(a_2,n_2,...,a_k,n_k)\} \uplus M)\end{aligned} \tag{6.3}$$

The abstract semantics function collects the number of times each language construct is used. It is defined below, in equation 6.4. The function $accessType$, utilized in the analysis, returns the cost type of the index expression given as a parameter, according to the rules discussed in section 5.5.2.

$$\begin{aligned}[\![]\!]^\sharp :&\ \mathcal{T} \to \mathcal{P}(\mathcal{C} \times \mathbb{N}) \\ [\![x]\!]^\sharp &= \{(\mathrm{A}_{\mathrm{Private}}, 1)\} && (storage\ x = Private) \\ [\![x]\!]^\sharp &= \{(\mathrm{A}_{\mathrm{Local}}, 1)\} && (storage\ x = Local) \\ [\![e_1 op_{float}\ e_2]\!]^\sharp &= (op_{float}, 1) \oplus ([\![e_1]\!]^\sharp \uplus [\![e_2]\!]^\sharp) \\ [\![e_1 op_{int}\ e_2]\!]^\sharp &= (op_{int}, 1) \oplus ([\![e_1]\!]^\sharp \uplus [\![e_2]\!]^\sharp) \\ [\![e_0[e_1,...e_k]]\!]^\sharp &= (accessType\ (e_0[e_1,...,e_k]), 1) \oplus \bigoplus_{e_l=e_0}^{e_k} [\![e_l]\!]^\sharp \\ [\![program]\!]^\sharp &= (BASE, 1) \oplus [\![e_1;...e_k]\!]^\sharp\end{aligned} \tag{6.4}$$

The total execution time of a program is computed from the sum of the run times of the cost types, based on the number of work items that are to be executed and the size of the work-group. The formula is shown in equation 6.6. The run times of the cost types may be computed using the function $assignCosts$ defined in equation 6.5. This utilizes the multiplicity and runtime functions deduced in chapter 5.

$$\text{assignCosts} : \mathcal{C} \to \mathbb{N} \to \mathbb{N} \to \mathbb{R} \tag{6.5}$$

$$\text{assignCosts}\, c\, n_{Ops}\, n_{Input} = n_{Ops} * T_c(n_{Input}) * M_c(n_{Ops})$$

$$T_{program}(n_{Input}, n_{work-group}) = M_{WG}(n_{work-group}) * \sum_{(c,n)\in[\![program]\!]^\sharp} \text{assignCosts}\ c\, n_{Ops}\, n_{Input} \tag{6.6}$$

6.2.2. Runtime Predictions

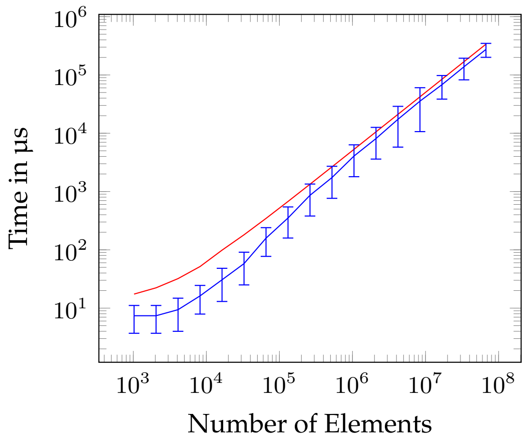
The analysis presented in the previous section was implemented in Java and utilizes the classes and data structures of the funkyIMP compiler to analyze domain iterations. The result of this analysis is presented in tabular form. It displays every cost type, the number of its occurrences, and the combined time estimated to be spent on it. An example output for a prediction is depicted in figure 6.1.

Cost Type	# in Kernel	Time
FLOAT_ADD	1	54.82778805217169
FLOAT_SUB	0	0.0
FLOAT_MUL	0	0.0
FLOAT_DIV	0	0.0
INT_ADD	2	55.128781833866825
INT_SUB	0	0.0
INT_MUL	3	81.04237583646253
INT_DIV	4	1509.3169877866271
LOCAL_ACCESS	0	0.0
PRIVATE_ACCESS	1	0.0
GLOBAL_WRITE	0	0.0
CONSTANT_GLOBAL_READ	0	0.0
CACHED_GLOBAL_READ	0	0.0
GLOBAL_READ	1	4981.5752014161835
COMPLEX_GLOBAL_READ	0	0.0
BASE_COST	1	3191.479259200592
TOTAL_COST	**X**	**9873.370394125905**

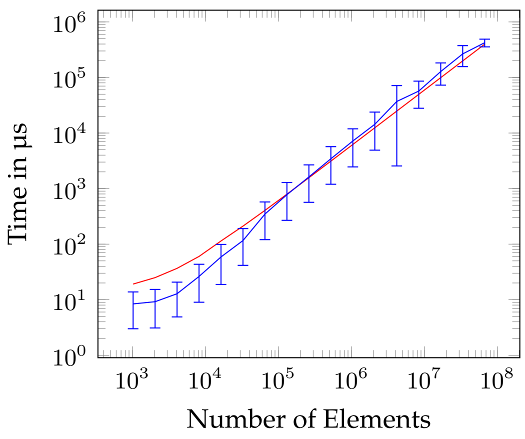
Figure 6.1.: The table shows the predictions for the domain iteration `x + matrix[x,y]` on a matrix with 4096×4096 elements.

In figure 6.2, the execution times of four different domain iterations are compared to predictions that were made during their compilation. The funkyIMP compiler is able to generate profiling code that executes the program multiple times, and to return the arithmetic mean and standard deviation of the runtime for the whole program and for parts marked for profiling. This was used to compare the predictions to real-world data. Each domain iteration was executed on a range of matrices with the number of elements increasing from 2^{10} to 2^{26}.

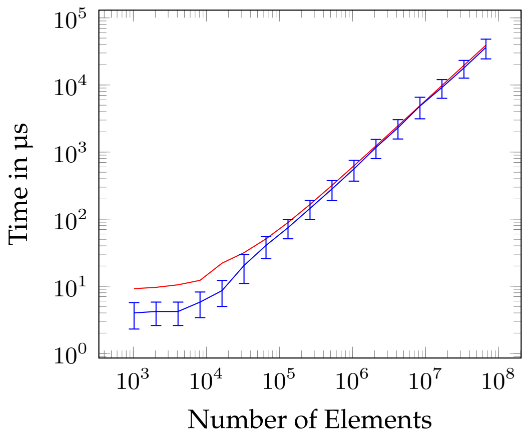
For data sets with less than fifty thousand elements the prediction generated estimations that were significantly too large. This inaccuracy may be due to incorrectly deduced fixed costs for the individual cost functions. Consider, for example, the cost function for a *continuous read* from global memory, $T_{cont}(x) = 0.521932ns * x + 3\mu s$. It has a constant offset of $3\mu s$. For small array sizes this has the greatest share of the total estimated time for the operation. The same holds for other cost functions with constant offsets. The cost functions are fitted to the data obtained by running the benchmark suite. Most have a constant offset, due to imprecisions in the data, hence causing mis-predictions for small array sizes. A quantitative analysis of the runtime predictions is given in section 6.3.



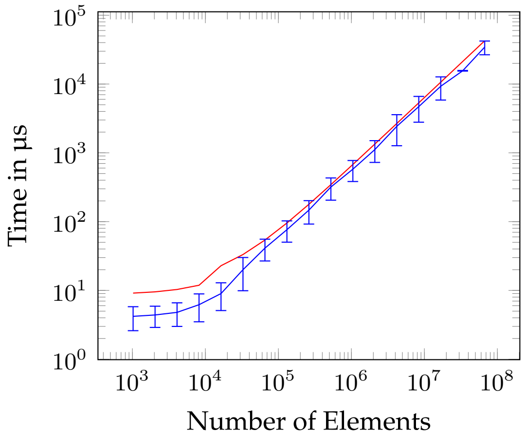
(a) `428.3741f + ((matrix[1 % HEIGHT, 1 % WIDTH] + matrix[x,y]) + matrix[y % HEIGHT, x % WIDTH])`



(b) `(((matrix[x,y] * matrix[1 % HEIGHT, 572 % WIDTH]) - matrix[y % HEIGHT, y % WIDTH]) + matrix[y % HEIGHT, 1 % WIDTH]) + matrix[x,y]`



(c) `matrix[x,y] * (matrix[x,y] * matrix[x,y])`



(d) `matrix[x,y] + matrix[1 % HEIGHT, 1 % WIDTH]`

Figure 6.2.: Comparison of predictions against actual execution time. To increase readability both axes use logarithmic scale. The prediction is depicted in red, the observed execution time in blue, with error bars marking the standard deviation.

6.3. Evaluation

The quality of the predictions obtained using the runtime prediction analysis was evaluated with two tests. The first one tested the quality of the predictions made for a largely unrestricted set of randomly generated sample domain iterations, the second one for a more restricted set, to evaluate the predictions for domain iterations that are likely to occur in productive environments.

6.3.1. Unrestricted Sample Set

The first test was executed with one thousand randomly generated domain iterations using the techniques and the platform described in section 6.1. The number of nodes in the expression tree was limited to the interval of $[2; 50]$. Other than that, no restrictions were applied.

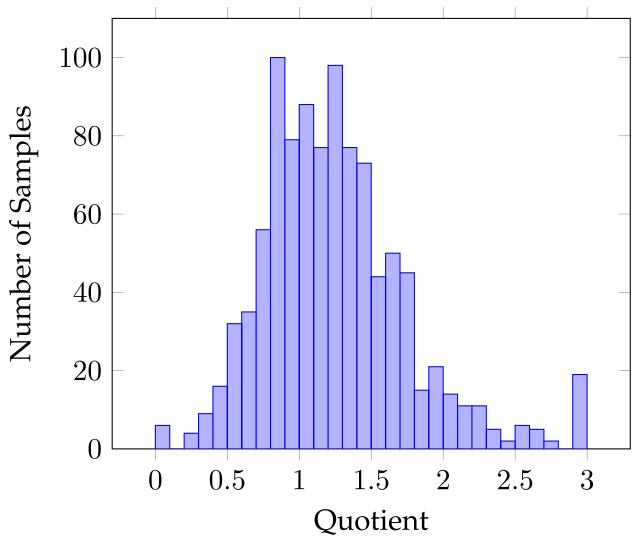


Figure 6.3.: Distribution of benchmark results for the test with an unrestricted set of domain iterations. The X axis denotes the quotient $\frac{t_{prediction}}{t_{result}}$. The closer this quotient is to 1, the better the prediction. A variation of this diagram, showing the distance between predictions and results, is shown in figure A.21 in the appendix.

Figure 6.3 displays the distribution of the quotient $\frac{prediction}{result}$ amongst the test cases in the first test. Each data point holds the result of one test case. The perfect result for a test case would be 1.0, and this holds for all of them. For the first test, the arithmetic mean of the quotient was **1.262** (rounded to four significant figures), indicating at a slight overestimation of the execution times. The standard deviation is at **0.5644**, with a standard error of **0.01785**. From this distribution, some conclusions about the nature of predictions may be drawn.

The first one is that predictions tend to be too high, with the mean of the ratios being twenty-five percent above the expected value. Hardware specific optimizations may be the cause of this. An execution unit may be able to perform several operations at once, or start the execution of an operation while waiting for a memory access to complete. This effect was already discussed in section 5.4.1, in the context of multiple basic operations in the same kernel. It is also likely to be present for other types of operations.

Imprecision whilst gathering the values may be a reason for the observed spread of these values. There may be a performance degradation when working with different screen resolutions, or with several screens, as more GPU time is required for refreshing the content that is being displayed. Added to this are applications being executed in the background. These may also require the GPU, thus possibly causing the execution of OpenCL kernels to be interrupted.

A third reason may be inaccurate predictions in regards to division and memory accesses. Normal memory accesses that are incorrectly classified as complex, as well as complex ones classified as normal may have an adverse effect on the quality of the runtime prediction. Division operations also have been observed to behave unpredictably regarding their runtime behavior. Time spent on divisions may depend on the values used on both sides of the operator, which makes a prediction difficult, as these are rarely known at compile time. Another explanation may be the optimizations discussed above. There may be several divisions executed at once, in a pipeline or while waiting for a memory access to complete.

6.3.2. Restricted Sample Set

The second test was executed with a thousand randomly generated samples. Here, however, several restrictions were put in place in order to obtain domain iterations that may conceivably be used in real world applications. The number of nodes in the expression tree was limited to the interval $[2; 6]$, and memory accesses were limited to at most two nodes per index expression. Division was not allowed[2].

Figure 6.3 displays the distribution of the quotient $\frac{prediction}{result}$ amongst the test cases in the second test. Again, each data point holds the result of one test case, with the perfect result being 1.0. For this test, the arithmetic mean of the quotient was **1.281** (rounded to four significant figures), again indicating an overestimation of the execution times. The standard deviation is at **0.3813**, with a standard error of **0.01206**.

Whilst the Standard Deviation of the second test's distribution is lower than the first, the arithmetic mean of the values was similar. The smaller spread of the values is caused by the absence of the complex memory accesses, as these tend to be difficult to predict. As with the first test, hardware specific optimizations may be a cause of the unexpected

[2]Runtime prediction for divisions has proven problematic on the test machine, an effect particular to that GPU. See section 7.1

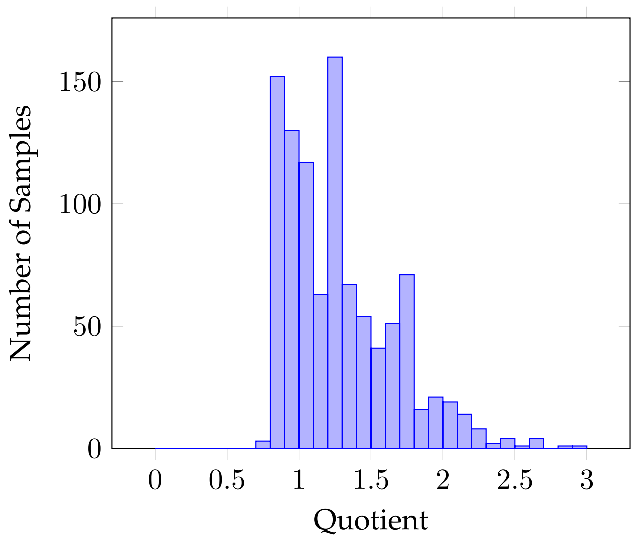


Figure 6.4.: Distribution of benchmark results for the test using the restricted example set. The X axis denotes the quotient $\frac{t_{prediction}}{t_{result}}$. The closer the quotient is to 1, the better the prediction. A variation of this diagram, showing the distance between predictions and results, figure A.22, can be found in the appendix.

performance gains.

Additionally, predictions for kernels with local memory accesses appear to be less accurate than those without. It is conceivable that there are cache effects influencing the runtime, similar to the those observed with global memory. Values that have been read once may remain available in cache. This makes further accesses to the same values as being essentially without cost. At the time of writing, these effects are not incorporated into the runtime model.

Part IV.

Further Work and Conclusion

7. Further Work

There are some issues and topics that did not fit within the scope of this thesis. For one, there is further work on the funkyIMP compiler. There are language features that have not been implemented in the OpenCL backend yet. Implementing these would make the compiler more useful in real world applications. These language features are discussed in section 7.3. Then there is the model itself. While it predicts the run time sufficiently precisely in the majority of cases, there are some cases where it fails to predict the runtime accurately. Details of these issues can be found in section 7.1. Additionally, in order to be viable for real world use, the model needs to be tested and adapted for other GPU architectures. This is discussed in section 7.2.

7.1. Refining the model

While the model gives good, or at least acceptable[1], predictions for a large majority of the domain iterations, there are some cases where the model fails to predict the runtime with an acceptable precision. Consider, for example, the kernel given in figure 7.1. The results, given in figure 7.2, reveal a significant discrepancy between the prediction and the run time of the kernel. The predictions made utilizing the model give an estimation that is too high, on average, by a factor of about three. It is possible to output a detailed report in the form of a table, with predictions for the time spent on each operation. For the this kernel, the table shown in figure 7.3 has been generated.

Two problems may be inferred from the data given in the table. The first problem is an incorrectly classified memory access. The analyzer is unable to recognize that the memory accesses `matrix[x,y]` and `matrix[x%HEIGHT, y%WIDTH]` always access the same elements. If they were recognized as identical, then the second one would be classified as cached, and therefore be assigned no costs[2]. This is not the case here, as evidenced by the two `GLOBAL READ`s in figure 7.3. Memory accesses that are recognized as identical only show up as one access in the detailed report. The incorrectly assigned memory access is therefore treated as independent, and assigned the full run time cost. The second problem is hinted at by the costs assigned to the floating point division. The predicted time that would be spent on that operation alone exceeds the observed run time for the kernel. This effect could be explained by shadowing, i.e. it is possible that the GPU performs the division while waiting for a memory access to complete, thus effectively shadowing the costs of the division.

[1]The prediction does not deviate by more than 50%
[2]see section 5.5.1

```
cancel new float[two_d{HEIGHT, WIDTH}].\[x,y]{
  matrix[x,y] * (y * (((0.585229f
        + ((x - matrix[x % HEIGHT, y % WIDTH])
    - matrix[492 % HEIGHT, 958 % WIDTH]))
    * (matrix[x,y] / 767.6984f))
    * 661.48236f))
};
```

Figure 7.1.: This is a domain iteration for which the prediction is remarkably inaccurate. The identifiers `HEIGHT` and `WIDTH` are replaced by the actual sizes of the benchmark.

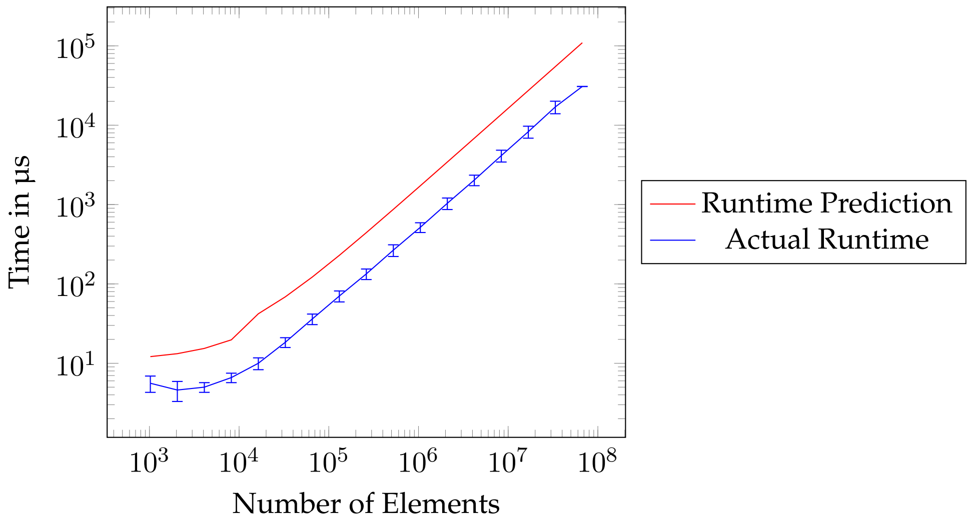


Figure 7.2.: This graph shows the runtime of the domain iteration shown in figure 7.1. It can be seen that there is a significant difference between the predicted and the actual runtime. To increase readability, both axes are logarithmic scale.

Another problem is compiler optimizations. The OpenCL compiler runs optimizations over the generated kernel. There are several optimizations, such as available expressions or constant propagation, that simplify the expression without changing the result. Thus, the code that is generated by the compiler may be different from the code given to the module analyzing the domain iteration. These optimizations are specific to the compiler used on the particular OpenCL platform. Factoring in these optimizations would require examining the used compilers on a case by case basis, as the optimizations might vary according to the different platform vendors.

Cost Type	# in Kernel	Time
FLOAT ADD	1	219.311
FLOAT SUB	2	0.0
FLOAT MUL	4	0.0
FLOAT DIV	1	39362.439
INT ADD	8	220.515
INT SUB	0	0.0
INT MUL	9	324.170
INT DIV	4	6028.192
LOCAL ACCESS	0	0.0
PRIVATE ACCESS	2	0.0
GLOBAL WRITE	0	0.0
CONSTANT GLOBAL READ	1	3083.679
CACHED GLOBAL READ	0	0.0
GLOBAL READ	2	47479.643
COMPLEX GLOBAL READ	0	0.0
BASE COST	1	12748.737

Figure 7.3.: This is the result of the analysis for the kernel shown in figure 7.1. and a matrix size of 8192×8192. Although the access `matrix[x,y]` occurs twice, it is only registered once, as the second occurrence will be cached.

7.2. Other GPU architectures

The model presented in chapter 5 was developed on a NVidia GT-650M, a notebook GPU with the Kepler microarchitecture. In order to use the model for other GPUs, the benchmark suite had to be executed on these GPUs as well. Up to now, the other GPUs that the benchmark suite has been executed on are the NVidia Quadro K4000, a professional workstation GPU, the AMD Radeon 5770 desktop GPU and the Intel HD 4000 integrated GPU.

However, the evaluation has only been carried out on the GT-650M. This is due to the fact that the performance prediction is currently integrated into the funkyIMP compiler. Unfortunately, installing that compiler is a non-trivial process, and an installation on Microsoft Windows is not possible at all. The best course of action would be to factor the performance evaluation into an external tool that analyzes the generated OpenCL code directly. This would also increase the applicability of the analyzer, as it would no longer depend on the funkyIMP language constructs, and could instead support all OpenCL constructs.

The Quadro GPU measurements look similar to the measurements taken from the GT-650M. This should not be not too surprising, as both GPUs share a common architecture, i.e. the Kepler microarchitecture. That being said, the Quadro K4000 is significantly faster than the GT-650M. The only notable behavioral difference is on division, which is as fast as the other basic operations. A diagram illustrating this can be found as figure A.14 in the appendix.

7.2.1. Intel HD 4000 (Integrated in the Ivy Bridge CPU)

The Intel HD 4000 GPU is an GPU that is packaged on the Ivy Bridge Architecture Intel CPUs. Integrated graphics solutions are built differently than dedicated GPUs. For one, they typically do not have any dedicated video memory. Instead, they share them main memory of the system with the CPU. Some integrated solutions have access to caches, but they typically share some of these with the CPU. The HD 4000 is no exception in this respect. Nevertheless, the performance of the HD 4000 is comparable to, or even exceeds, the performance of entry-level dedicated GPUs available on the market at the time the Ivy Bridge CPUs were released. [22]

The Intel HD 4000 has a total number of 16 execution units, each clocked at 1.2GHz. It may use up to 1GB of the main memory of the system, and the processor's L3 cache. Additionally, it has its own set of faster caches. The maximum work-group size is 512. [16]

The execution of the benchmark revealed a number of deviations from the model described in chapter 5. The first becomes apparent when looking at figure A.17, depicting the execution time for an OpenCL computation with a varying number of work items in the work-group. In contrast to the result of the GT-650M (see figure 5.3), the HD 4000 does not exhibit the inverse exponential behavior discussed in section 5.3. Instead, whenever work-group size $w \bmod 16 = 1$, there was a spike in the run time, with the first spike having the greatest effect on this and later ones becoming gradually smaller. This provides a clue as to the degree of parallelism. In OpenCL each work-group holds a fixed number of work items that is specified prior to the execution of the kernel. For the execution of work items from a new work-group to be initiated, all the work items in the prior work-group have to finish their execution. The Intel HD4000 has 16 execution units. Hence, with a work-group size of 17 the utilization will be $\frac{17}{32}$, with a work-group size of 33 the degree of utilization will be $\frac{33}{48}$, and so on. This can be generalized with the formula shown in equation 7.1, leading to the multiplicity function seen in equation 7.2.

$$U(w, n_{XU}) = \frac{w \bmod n_{XU}}{n_{XU}} + \frac{\lfloor \frac{w}{n_{XU}} \rfloor}{\lceil \frac{w}{n_{XU}} \rceil} \tag{7.1}$$

$$M(t_{max}, U) = \frac{t_{U_{max}}}{U} \tag{7.2}$$

In terms of the kernel itself, basic operations appear to bring no costs at all with them. Measurements (see figure A.15) reveal the costs of basic operations to be negligible, and the Standard Deviation is judged to be too great to make reliable predictions. This effect could be researched more thoroughly, in order to determine whether the basic operations are shadowed by other operations in some manner.

Memory accesses made within the kernel also show a different behavior compared to the one discussed in section 5.5. Consider measurements taken from the various kinds of memory accesses[3] shown in figure A.16 in the appendix. Here, a number of discrepancies that could be profitably addressed in future revisions of the runtime prediction model, are discussed. Firstly, there is a performance anomaly occurring when handling more than 10 million elements. After that point the results stopped increasing linearly, as expected. Instead, the kernels were observed to be executing *faster*, even though the kernel was executed with a greater number of elements. This effect could be researched further.

The fact that there seem to be very little costs attached to constant and interval accesses also hints at caches with fast access speeds. Also, a single continuous access also does not have a major impact on the run time. Significant increases in the run time behavior may only be observed when there is a second continuous access performed within a kernel. This effect occurs both for memory accesses that reference different addresses, as well as for accesses referencing the same addresses. This could also be researched further and incorporated into the model.

7.2.2. AMD Radeon HD 5750

The AMD Radeon HD 5750 is mid-range desktop GPU released in 2009. It was among the first AMD GPUs to support OpenCL. It is clocked at 700MHz, with nine execution units, and 1GB GDDR5 video memory. The maximum work-group size is 256.
[1, 3]
The execution of the benchmark suite reveals a behavior similar to the one observed with the NVidia GPUs. Basic operations behave the same way, except that the time they take to complete is not observable for single operations. For multiple operations, the observed behavior is the same as with the GT-650M. Again, the first few operations do not lead to a significant increase in execution time, afterwards, there is a constant amount of time added for each operation. The measurements are depicted in figure A.20.

The runtime behavior in respect to the size of the work-group resembles the one observed in the Intel HD4000. Consider figure A.18 for the data gathered experimentally. Again, the runtime decreases as the size of the work-group increases, up to a work-group size of 64. Afterwards, the execution time doubles, and starts decreasing again.

When taking measurements on the HD 5750, performance anomalies similar to the one occurring on the HD 4000 were observed. While the measurements that were taken indicate a suitability of the runtime prediction model for AMD GPUs, further measurements could be taken in order to gather valid data for all cost types and therefore verify that claim.

[3]The same kinds of accesses as described in section 5.5.1.

7.3. Work on the compiler

The funkyIMP OpenCL backend supports a subset of all the available operations in the funkyIMP programming language, as described in chapter 3. The functionality is enough to perform basic computations, however there are more language features that have to be implemented in order to use it in a productive manner.

Full Type System Support has not been implemented as yet. Currently the compiler backend only supports iterations over multidimensional arrays. This is only a subset of the abilities of the type system. For example, it is possible to describe subsets of such an array. The following example describes a partition of a matrix into three parts, i.e. the trace, the upper and the lower triangle.

$$\text{domain trace}\{x,y\} : \mathit{one\_d}\{x\}(j) = \{\mathit{two\_d}\{x,y\}(a,b) \mid j = a \wedge a = b \wedge x = y\} \tag{7.3}$$
$$\text{domain utriag}\{x,y\} = \{\mathit{two\_d}\{x,y\}(a,b) \mid b \geq a+1 \wedge x = y\} \tag{7.4}$$
$$\text{domain ltriag}\{x,y\} = \{\mathit{two\_d}\{x,y\}(a,b) \mid b < a \wedge x = y\} \tag{7.5}$$

Domain iterations are also supported on these projections. The OpenCL compiler backend, however, does not support them. To implement support for these, the code generation has to be modified slightly. First, the host code needs to copy the whole memory segment that houses the domain. Implementing this feature is an opportunity to optimize the copying of memory between device and host. For example, if an iteration over the upper triangle is followed by an iteration over the lower triangle, the memory segment that contains both of them should only be copied once, as it cannot be changed owing to the linear type constraint.

The kernel will be executed with as many work items as there are elements in the domain. In the kernel there needs to be a mapping from the current work item id to the coordinates of the element that is being worked on presently, to access the correct element in the memory segment. Projections may also be stacked, so there might be multiple translations until the actual type is reached.

Nested Domain Iterations are another concept that needs to be supported. Currently, there is no possibility to execute a piece of code more then once, as OpenCL does not allow recursion, and the funkyIMP language only allows loops in the form of domain iterations. The current OpenCL backend does not support this, however. For the code executed on the GPU, the barvinok[4] library can translate the domain iterations to `for`-loops. These loops could then be used in the OpenCL kernel, after making sure the identifiers are unique. For loops that have a defined size, loop unrolling may be applied in order to reduce the overhead caused by the control flow operations.

[4]library for counting the number of points in polytopes. (http://freecode.com/projects/barvinok)

Loops may also present a challenge for the Runtime model, as the run time may not scale linearly with the number of iterations in the loop. There might be effects due to the caches used that might change the run time, so one cannot simply multiply the run time for a single run with the number of times the loop body is executed. Furthermore, control flow instructions are expensive, and need to be additionally factored into the run time prediction.

Dynamic Arrays should be straightforward to implement. These cannot change the size of their dimensions within the domain iteration, so they behave just as their static counterparts from the OpenCL point of view. For the host side, the static information from the domain type that is used to determine the number of work items, and the size of the memory allocated on the GPU, needs to be replaced by function calls to the funkyIMP runtime.

Object-Oriented concepts might prove difficult to implement. The FunkyIMP language supports classes, inheritance and polymorphy. It is possible to map object-oriented structures with virtual function calls to C, for example by adding pointers to the addresses of virtual functions as fields of the structure modeling the translated class. However, there is the restriction in OpenCL C that pointers to functions are specifically not allowed. Another approach might be to generate OpenCL C code for all possible matches of the object, and then choosing the appropriate implementations, and optionally the superclass implementation, dynamically at runtime using a field indicating the runtime type of the object. Unfortunately, this approach does not work for code that is not known at compilation time, which makes its use in libraries difficult.

8. Conclusion

The purpose of the work described in this thesis was to build a model for the prediction of the time spent on the execution of OpenCL kernels. In order to create this model, an OpenCL backend to the funkyImp compiler developed by Alexander Herz (see [9]) has been implemented. The results presented in this thesis show that predictions about OpenCL programs without loops with a potentially unlimited number of iterations are possible, with the quality of the predictions being acceptable for typical, compute focused OpenCL kernels. The model is based on the run time behavior of Kepler microarchitecture GPUs and should be flexible enough to be adopted to other types of NVidia GPUs. Some work would need to be done to adopt it to integrated GPUs and dedicated GPUs manufactured by AMD.

The predictions are not ends in themselves. A scheduler that manages the resources of a compute node can use them as a metric, and use them to approximate the run time of a more complex computation. This is done by creating a task schedule that uses the metrics generated with the help of the model to find the schedule that utilizes the available hardware as optimally as possible, thus also approximating the run time for the whole program. In some cases it may be useful to perform a computation on the GPU even if it is slower than the CPU, because it frees the CPU to do other tasks while waiting for the result of the GPU.

This makes the predictions an important part of taking the reasoning about hardware details out of the programmer's hands. The creation of schedules means letting the compiler decide which computing resource is used for a computation. Instead of worrying about boilerplate code and problems of parallel software, the programmer can instead focus on the algorithm, while still producing a program that optimally uses the available resources. The compiler and the runtime environment will then handle all the details of the resource allocation.

Appendix

A. Diagrams

A.1. OpenCL

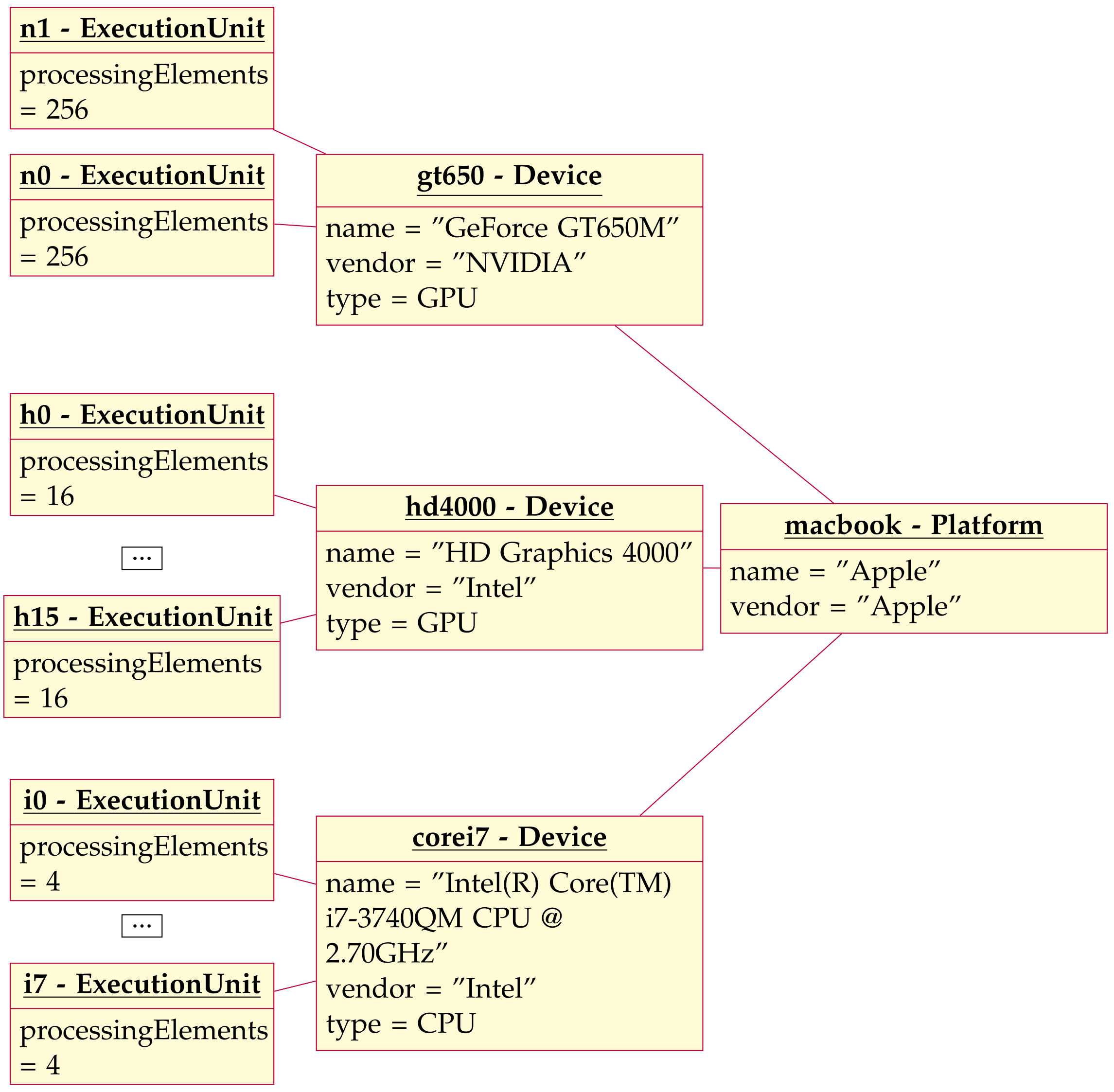


Figure A.1.: UML object diagram of a possible configuration of an OpenCL computing environment. The platform in this case would be OS X, with Apple as the platform vendor. The configuration has three devices. One of them is a dedicated Graphics Card, the GT650M, the other the HD4000 graphics that is integrated in the CPU. Lastly there is the CPU itself.

A.2. Benchmark Suite System Design

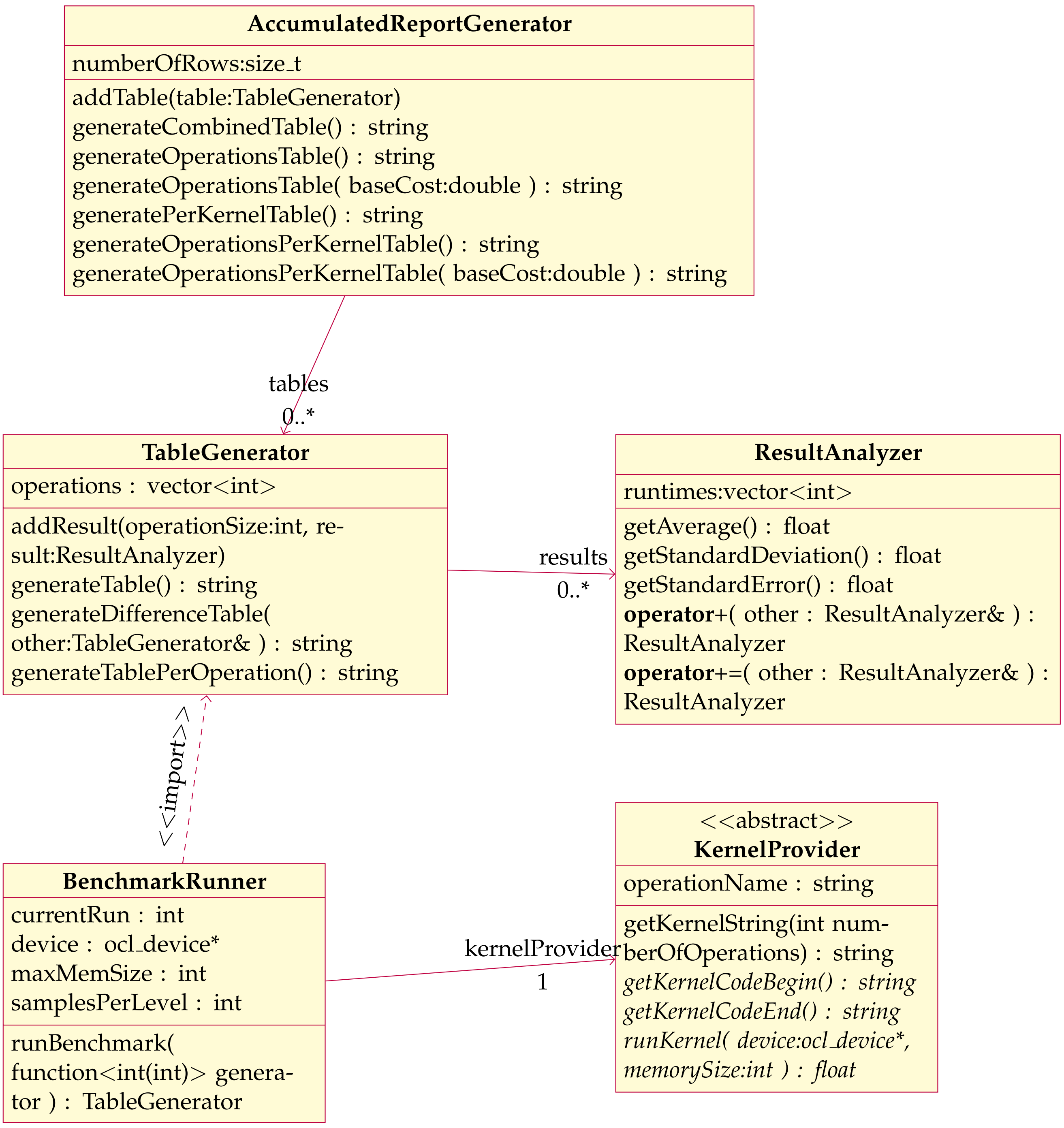

AccumulatedReportGenerator
numberOfRows:size_t
addTable(table:TableGenerator)
generateCombinedTable() : string
generateOperationsTable() : string
generateOperationsTable(baseCost:double) : string
generatePerKernelTable() : string
generateOperationsPerKernelTable() : string
generateOperationsPerKernelTable(baseCost:double) : string
tables
0..*
TableGenerator
operations : vector<int>
addResult(operationSize:int, result:ResultAnalyzer)
generateTable() : string
generateDifferenceTable(other:TableGenerator&) : string
generateTablePerOperation() : string
results
0..*
ResultAnalyzer
runtimes:vector<int>
getAverage() : float
getStandardDeviation() : float
getStandardError() : float
operator+(other : ResultAnalyzer&) : ResultAnalyzer
operator+=(other : ResultAnalyzer&) : ResultAnalyzer
<<import>>
BenchmarkRunner
currentRun : int
device : ocl_device*
maxMemSize : int
samplesPerLevel : int
runBenchmark(function<int(int)> generator) : TableGenerator
kernelProvider
1
<<abstract>>
KernelProvider
operationName : string
getKernelString(int numberOfOperations) : string
getKernelCodeBegin() : string
getKernelCodeEnd() : string
runKernel(device:ocl_device*, memorySize:int) : float

Figure A.2.: UML class diagram that shows the System Design of the Benchmark Suite. The subclasses of `KernelProvider` are omitted for space reasons. They can be found in figure A.3.

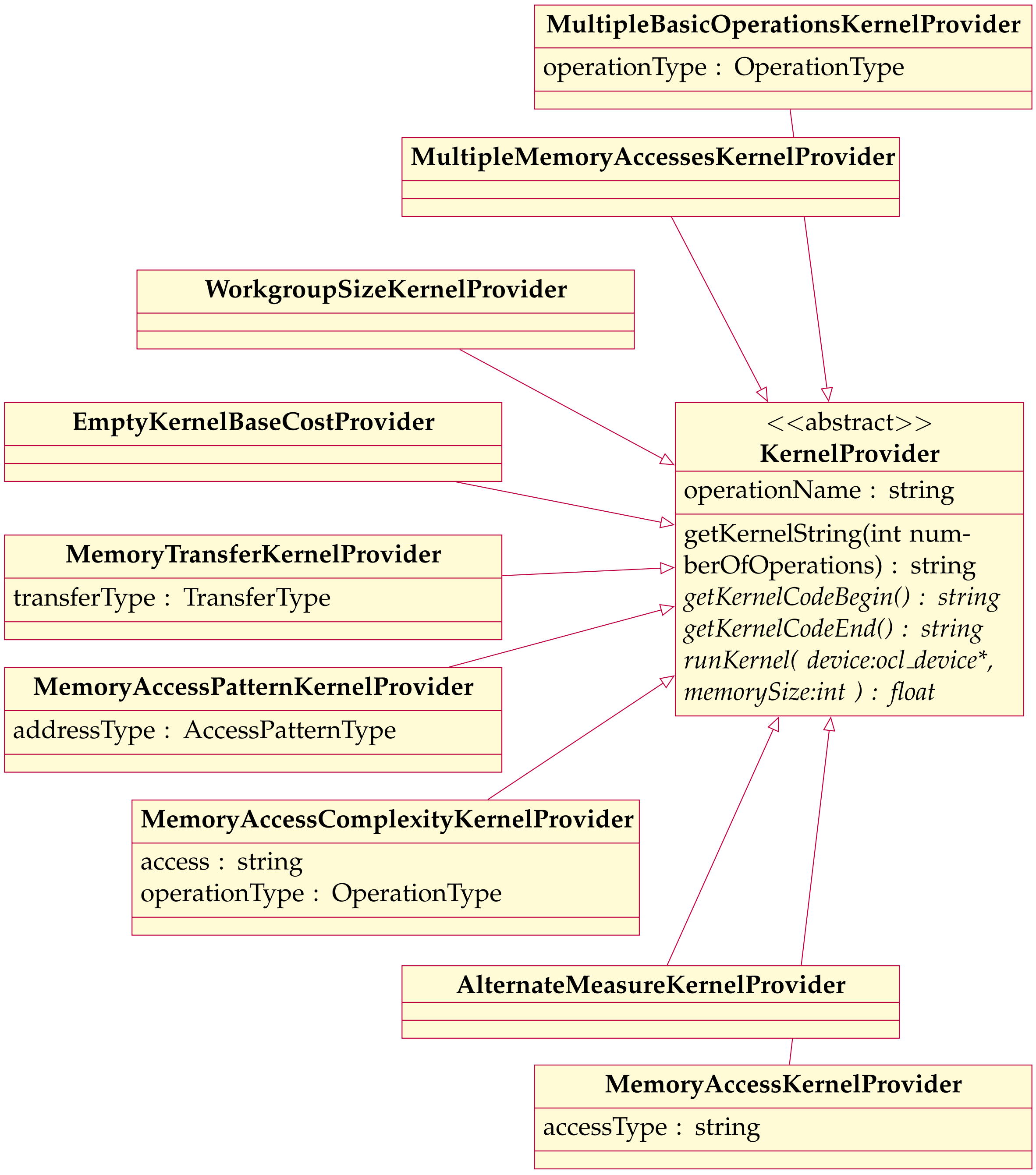

MultipleBasicOperationsKernelProvider
operationType : OperationType
MultipleMemoryAccessesKernelProvider
WorkgroupSizeKernelProvider
EmptyKernelBaseCostProvider
MemoryTransferKernelProvider
transferType : TransferType
MemoryAccessPatternKernelProvider
addressType : AccessPatternType
MemoryAccessComplexityKernelProvider
access : string
operationType : OperationType
AlternateMeasureKernelProvider
MemoryAccessKernelProvider
accessType : string
<<abstract>>
KernelProvider
operationName : string
getKernelString(int numberOfOperations) : string
getKernelCodeBegin() : string
getKernelCodeEnd() : string
runKernel(device:ocl_device*, memorySize:int) : float

Figure A.3.: The `KernelProvider` subclasses. Each subclass implements one actual benchmark run.

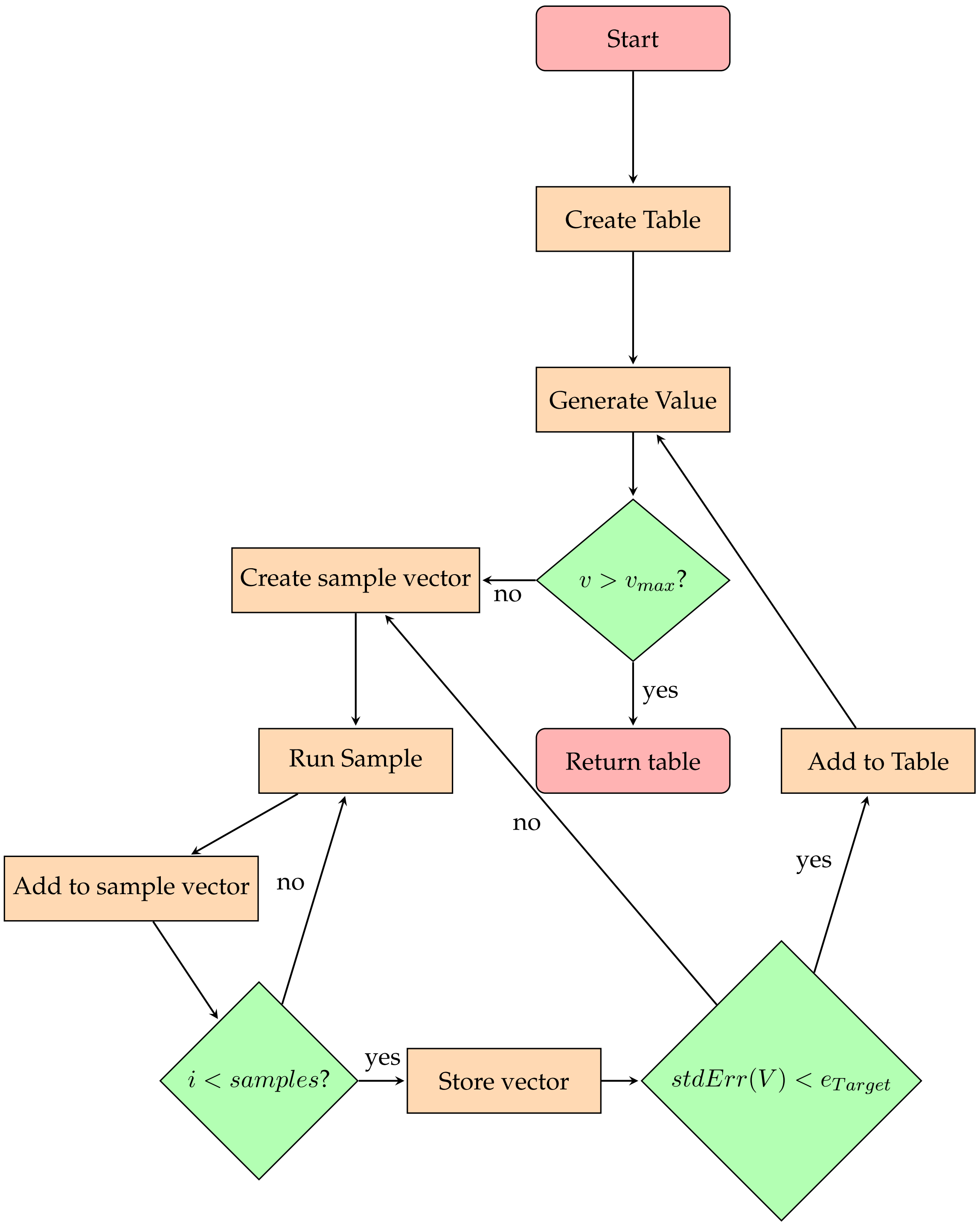


Figure A.4.: Control flow graph illustrating the iterative Process the Benchmark Suite uses.

A.3. Runtime Model

This section contains additional diagrams illustrating the performance of various operations on the GPU. If it is not specified otherwise, the measurements were taken on a NVidia GT-650M.

A.3.1. Floating Point Arithmetics

The Execution time for floating point arithmetics is depicted in figures A.5 to A.8.

A.3.2. Integer Arithmetics

The Execution time for integer arithmetics is depicted in figures A.9 to A.12.

A.3.3. Experiments into Memory access complexity

To research different classes of memory complexity, a test with a kernel with a memory access with an increasing number of nodes in its index expression tree is performed. The results are shown in figure A.13.

A.3.4. Measurements from the NVidia Quadro K4000

There have been measurements taken with the NVidia Quadro K4000 workstation GPU. Measurements revealing a behavior diverging from the one observed in the NVidia GT-650M are depicted here in figure A.14.

A.3.5. Measurements from the Intel HD4000

There have been measurements taken with the Intel HD4000 integrated GPU. Measurements revealing a behavior diverging from the one observed in the NVidia GT-650M are depicted here in figures A.15, A.16 and A.17.

A.3.6. Measurements from the AMD Radeon HD 5750

Measurements taken on the AMD Radeon HD 5750 are depicted here, in figures A.18, A.19 and A.20

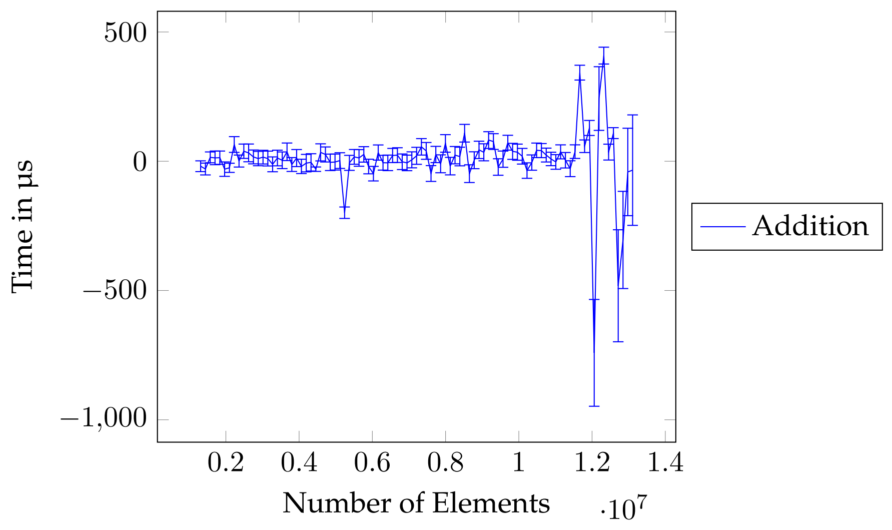


Figure A.5.: Execution time for the basic add operation on floating point values.

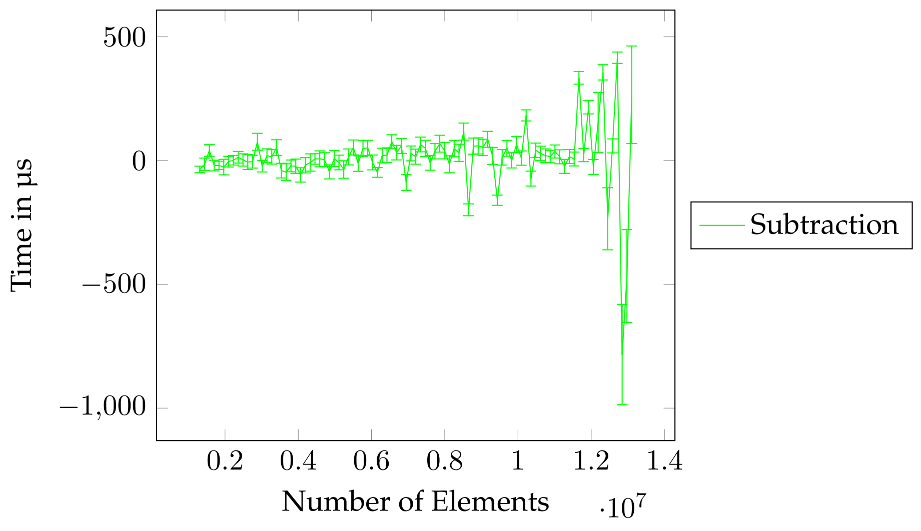


Figure A.6.: Execution time for the basic subtract operation on floating point values.

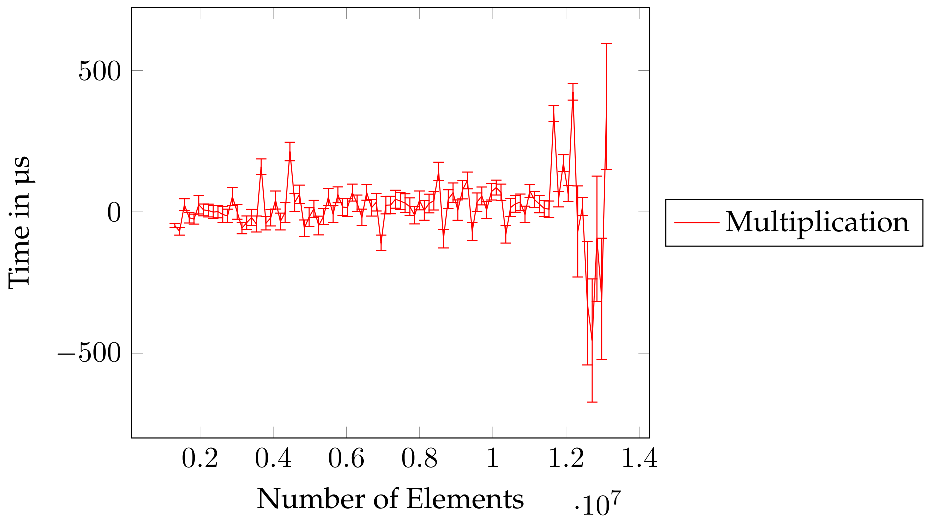


Figure A.7.: Execution time for the basic multiply operation on floating point values.

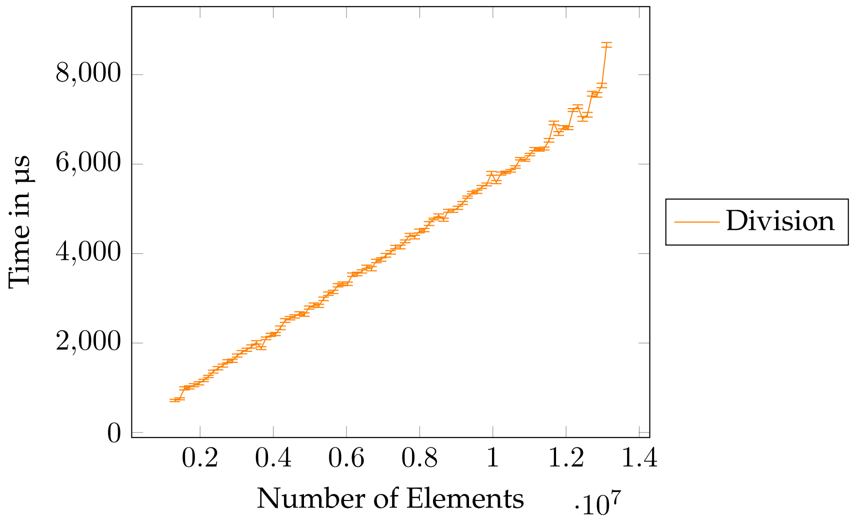


Figure A.8.: Execution time for the basic division operation on floating point values.

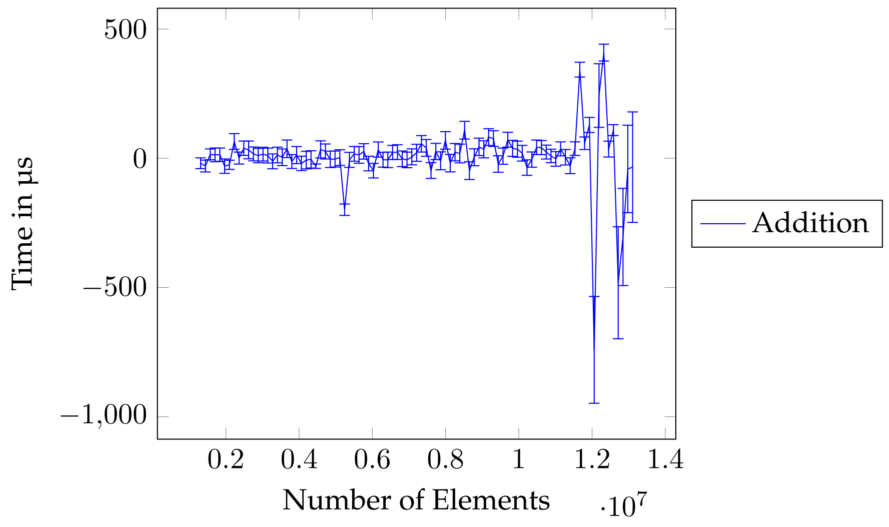


Figure A.9.: Execution time for the basic add operation on integer values.

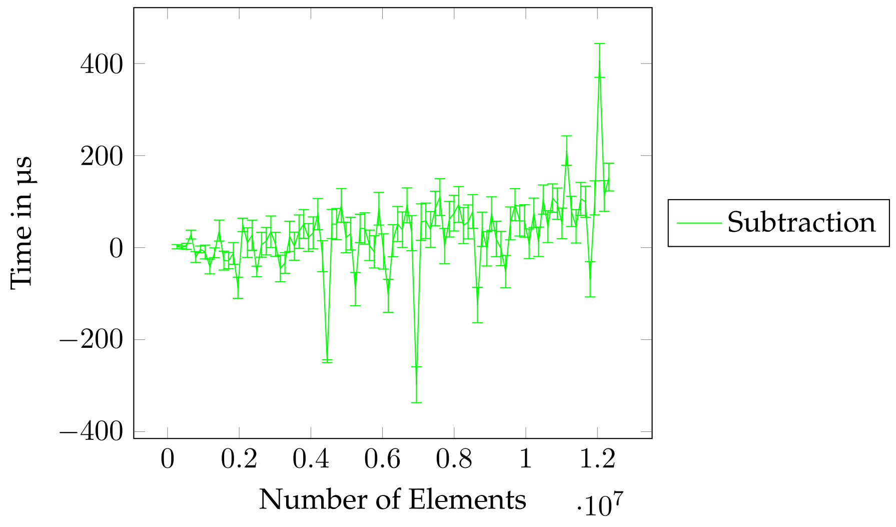


Figure A.10.: Execution time for the basic subtract operation on integer values.

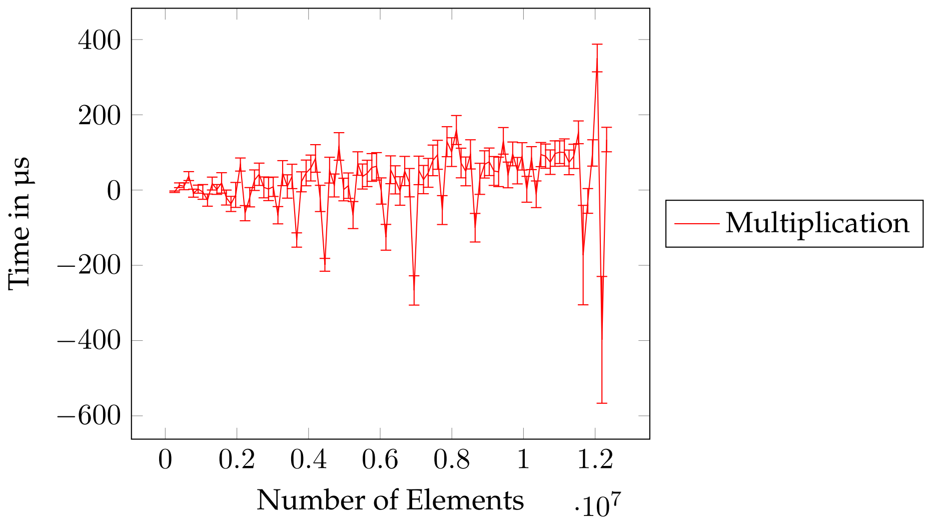


Figure A.11.: Execution time for the basic multiply operation on integer values.

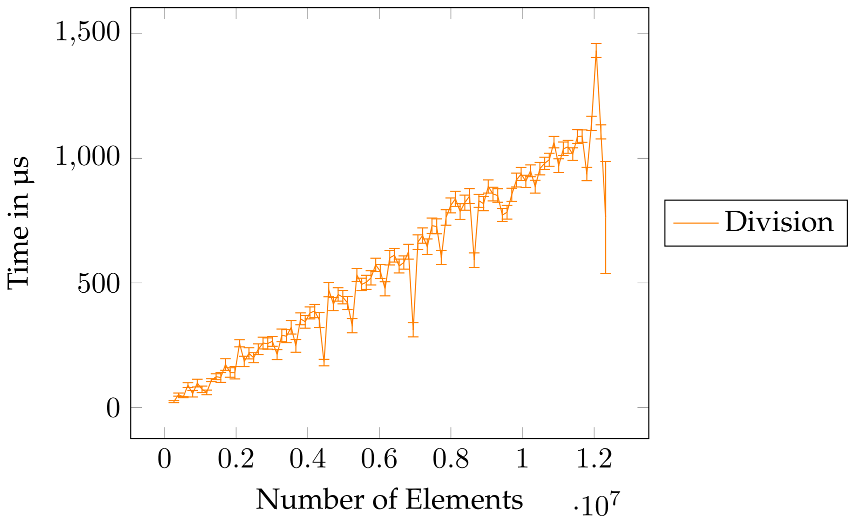


Figure A.12.: Execution time for the basic division operation on integer values.

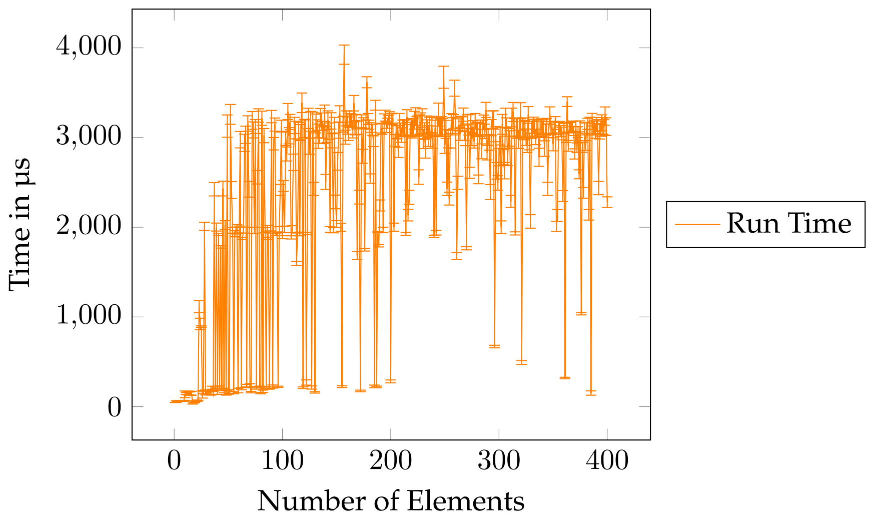


Figure A.13.: Execution time for different randomly generated memory accesses. The number of nodes in the index expression tree are increased every twenty samples.

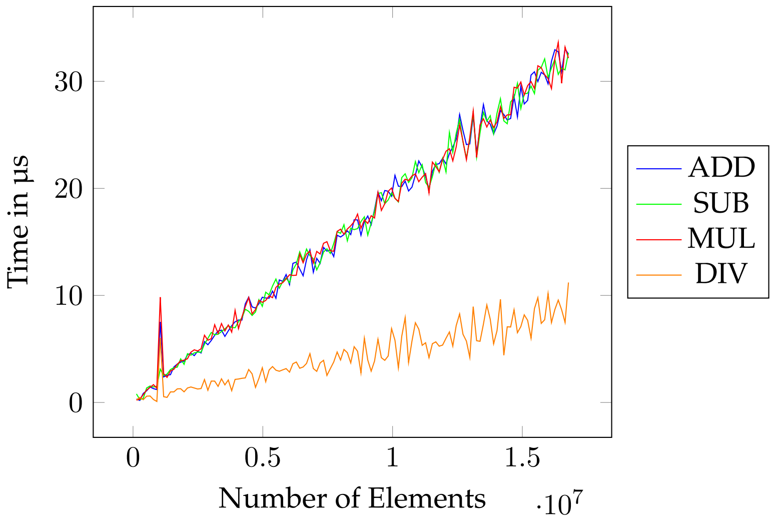


Figure A.14.: Execution time for basic operations on floating point values on the NVidia Quadro K4000 Workstation GPU. The Standard Deviation is omitted from this diagram for legibility reasons.

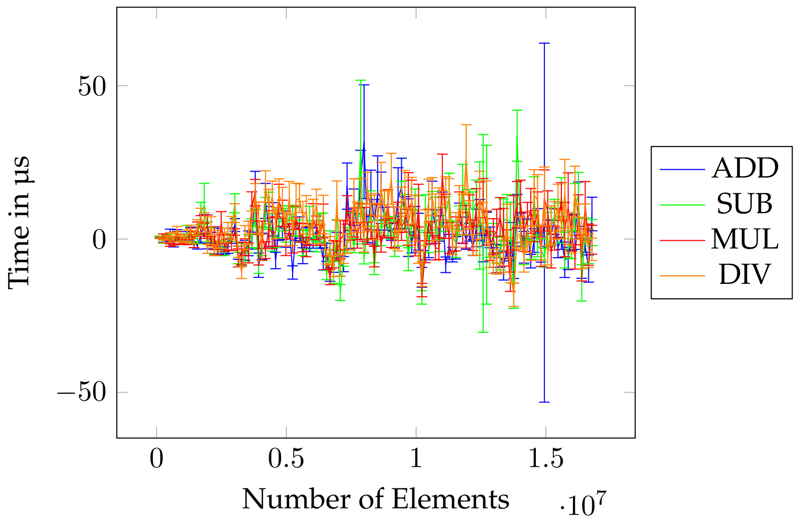


Figure A.15.: Execution time for basic operations on floating point values on the Intel HD4000 integrated GPU. Unfortunately there is not too much to be gained from this diagram, other than the fact that the time spent on basic operations cannot be measured the way it was done on the GT-650M.

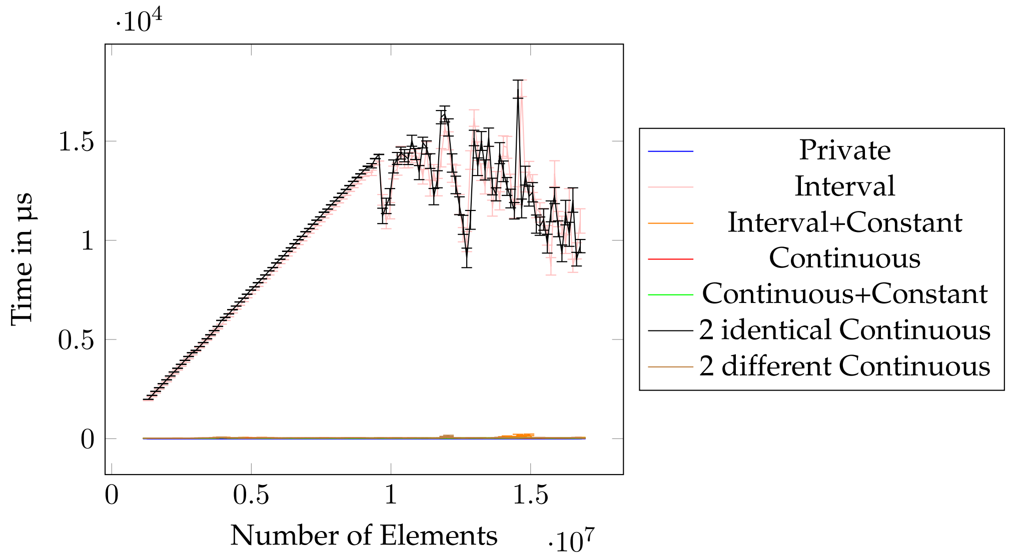


Figure A.16.: Execution times for memory access operations on the Intel HD4000 integrated GPU. This benchmark compares different kinds of global accesses. The Standard deviation (Comparatively small with $\sim 2\%$) is omitted for legibility reasons. All operations contain a write to global memory, the first operation is given as a reference.

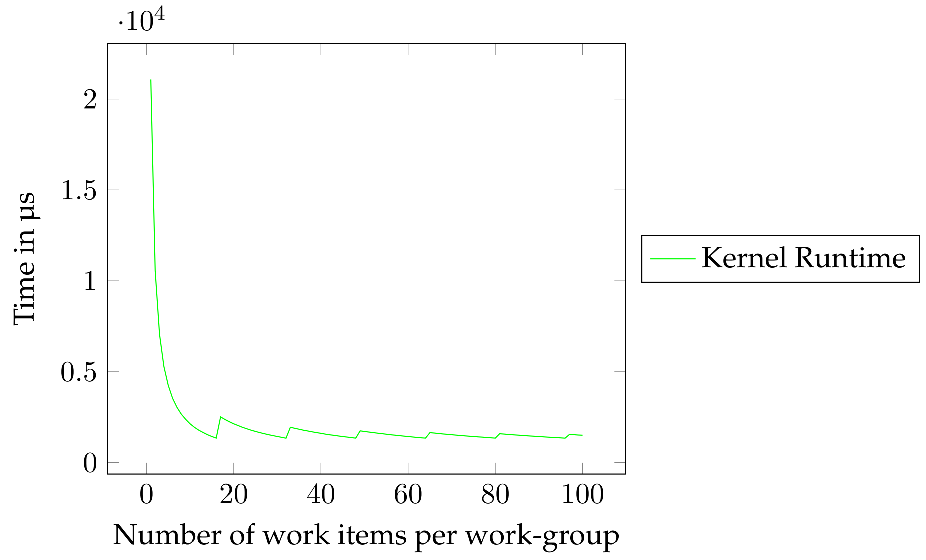


Figure A.17.: Execution time for different work-group sizes. Although the greatest work-group size of the HD4000 is 512, the graph is limited to work-group sizes up to 100 in order to show the difference from the graph depicting the GT-650M's performance.

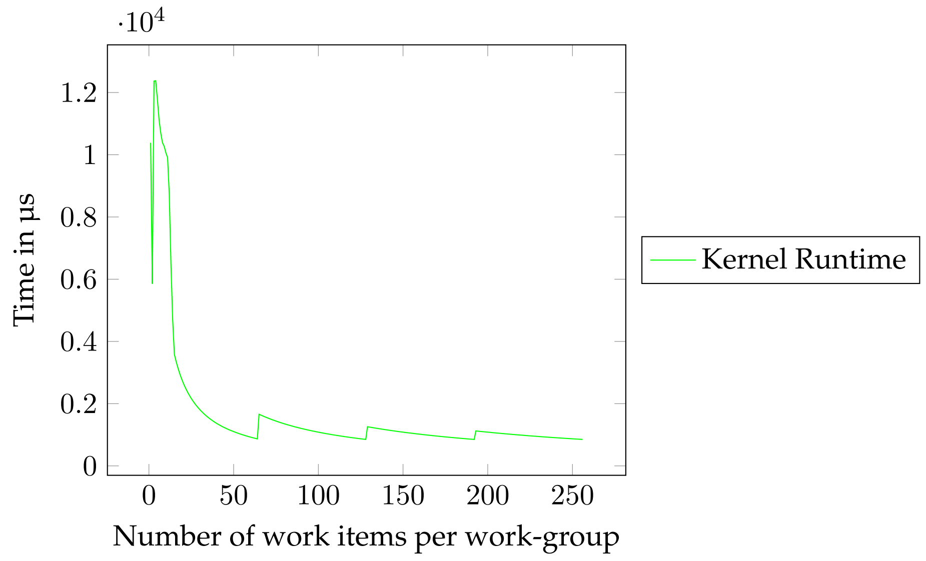


Figure A.18.: Execution time for different work-group sizes. The data reveals the a periodic runtime behavior similar to the one observed in with the Intel HD 4000.

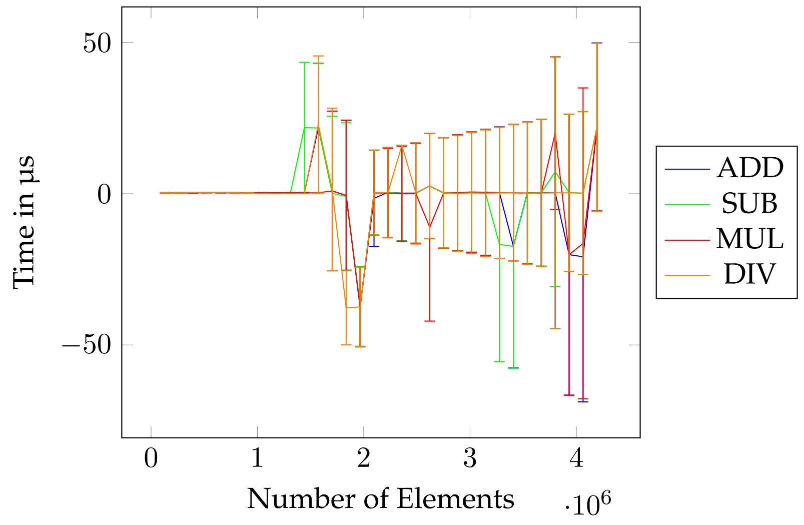


Figure A.19.: Execution time for basic operations on floating point values on the AMD Radeon HD 5750 GPU.

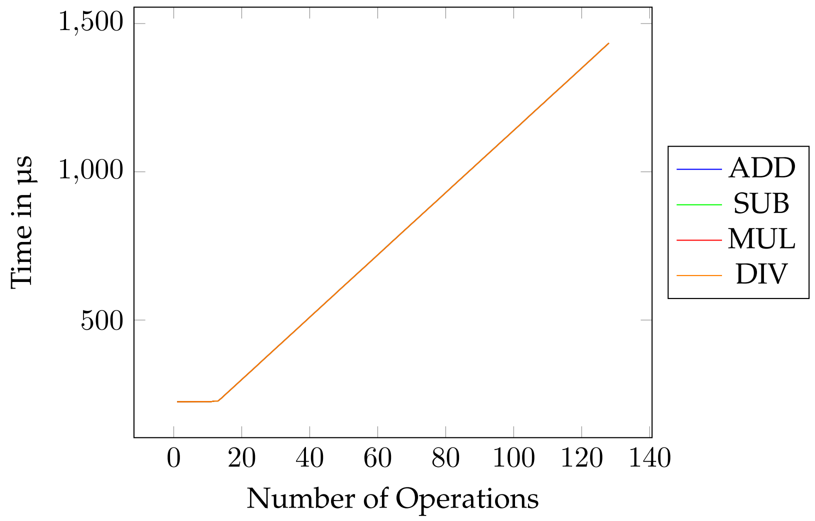


Figure A.20.: Total execution time for multiple basic operations (ADD, SUB, MUL and DIV) on floating point values. The x axis denotes the number of operations of a single type that are executed within a single kernel. The Standard Deviation is omitted for legibility reasons. In this case, it is fairly small, about $\pm 10\mu s$ for the memory size of 256kB.

A.4. Results

A.4.1. Distance between Predictions and Results in Automated Tests

There is an alternative way to illustrate the quality of a prediction. This is done by depicting the distance between the prediction and the result as follows:

$$d(p,a) = \frac{\max\{p,a\}}{\min\{p,a\}} \tag{A.1}$$

This computes the quotient of the larger to the smaller value, resulting in values that are always greater or equal than one. While information about the kind of the prediction error is lost, instead one may gain information about how far the numbers are off, e.g. less than 20%. The figures below show the distance for the results of the tests described in chapter 6.

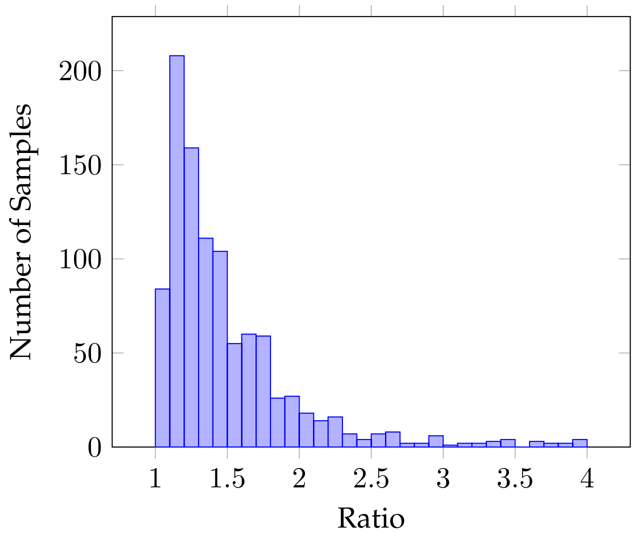


Figure A.21.: Distribution of benchmark result distances for the test with the unrestricted example set. The closer the distance is to 1, the better the prediction.

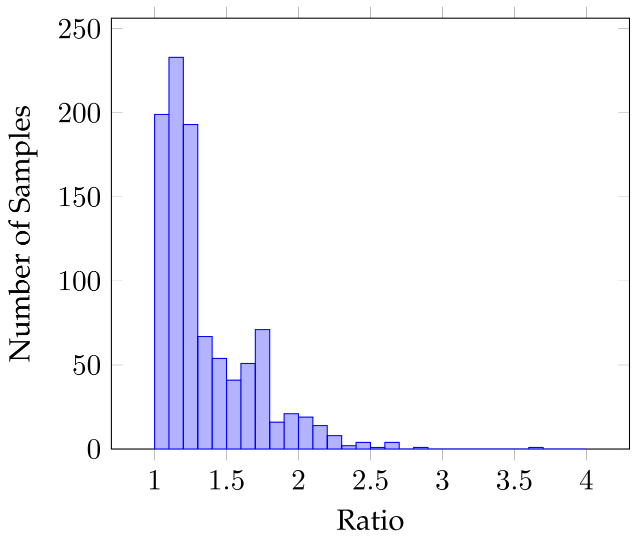


Figure A.22.: Distribution of benchmark result distances for the test with the restricted example set. The closer the distance is to 1, the better the prediction.

B. Code Samples

B.1. OpenCL Code

B.1.1. An Empty Kernel

This is basically an empty OpenCL C kernel. The only operation that is in its body determines the global id of the thread it is executed in. In the context of the Benchmark Suite, it is used to determine the basic cost that occurs whenever a kernel is run.

```
__kernel void empty(__global float *memory)
{
    get_global_id(0);
}
```

B.1.2. Kernel used for the Work-Group Size Benchmark

This kernel is used for determining the impact of the work-group size. The kernel only reads a segment of the global memory, and stores it again. It serves to show the benefits of using work-groups that are as large as possible.

```
__kernel void wg_kernel(global float *memory) {
    int work_item = get_global_id(0);
    memory[work_item] = memory[work_item] / 42.f;
}
```

B.1.3. Kernels used to measure Basic Operations

Kernel used to measure basic Performance

This Kernel is used to determine the cost of executing basic operations. The **defines** at the top of the code are inserted in the code below, and then executed.

```
#define CODE_BASE int x = get_global_id(0); \
                  output[x] = input[x];
#define CODE_ADD  int x = get_global_id(0); \
                  output[x] = input[x]+f;
#define CODE_SUB  int x = get_global_id(0); \
                  output[x] = input[x]-f;
```

```
#define CODE_MUL  int x = get_global_id(0); \
                  output[x] = input[x]*f;
#define CODE_DIV  int x = get_global_id(0); \
                  output[x] = input[x]/f;

__kernel void basic_op(global float *input, float f,
                       global float *output)
{
        CODE_<BENCH>;
}
```

Kernel used to measure behavior with multiple operations

This kernel is used to analyze the runtime behavior in the presence of multiple operations. The number of operations is varied, and the operator and type is changeable as well. The operation is applied on the variable a certain number of times, and then written back to memory.

```
#define TYPE float
#define OP +
#define OPERATION i = i OP 7

__kernel void multiple_ops(global TYPE *memory) {
    int work_item = get_global_id(0);
    TYPE i = 2147483648;
    OPERATION;
    ...
    OPERATION;
    memory[work_item] = i;
}
```

B.1.4. Kernels used to measue Memory Accesses

Kernel used to measure Basic Memory Access Performance

This Kernel is used to determine the cost of executing memory accesses. The defines at the top of the code are inserted in the code below, and then executed.

```

#define BENCH_CST      output[x] = input[0x3ff];
#define BENCH_CNT      output[x] = input[x];
#define BENCH_IVL      output[x] = input[x & 0x3ff];
#define BENCH_IVLCST   output[x] = input[0x7ff] + input[x&0x3ff];
#define BENCH_CNTCST   output[x] = input[x] + input[0x3ff];
```

```
#define BENCH_CNTCNT  output[x] = input[x] + input[x];
#define BENCH_CNTCNT2 output[x] = input[x] + input[x>>1];

__kernel void memory_access(global float *input, float f,
                            global float *output)
{
        int x = get_global_id(0);
        CODE_<BENCH>;
}
```

Kernel used to measure Complex Memory Access Performance

This Kernel is used to determine the cost of executing complex memory accesses. The calculations in the beginning are used to make the access as random as possible.

```
__kernel void memory_access(global float *input,
            float f, global float *output)
{
    int work_item = get_global_id(0);
        unsigned int x = (work_item>>11) & 2047;
        unsigned int y = work_item & 2047;
    output[work_item] = input[((x + (415397 + x))
         * (785 * ((x + 705)
         * (y * ((x + 95) * x)))))]
}
```

Kernel used to measure Multiple Memory Access Performance

This kernel is used to determine the cost scaling of using multiple memory accesses in one kernel. The benchmark is executed multiple times, each time with another number of memory accesses. The accesses are inserted at the `/* +...  */` comment.

```

__kernel void multiple_access(global float *memory) {
    int work_item = get_global_id(0);
    memory[work_item] = 42.f + memory[work_item + 0]
                         /* + memory[work_item + 1] + ... */;
}
```

B.2. Functional Code

B.2.1. The `map` function

The function `map` applies a function `f` to each element of a list, and returns a list with all the results, while preserving the order.

```
fun map f nil      = nil
  | map f (x::xs) = x :: map f xs
```

B.2.2. The `filter` function

The function `filter` applies a predicate `p`, that is a function that returns a boolean value to each value of the list, and returns a list with all elements for which the predicate returns true, while preserving the original order.

```
fun filter p nil      = nil
  | filter p (x::xs) = if p x then (x::filter p xs)
                              else filter p xs
```

B.2.3. The `foldl` function

The function `foldl` applies a function `f` to each element of a list and the result of the previous application of `f` or a parameter given at the start. It returns the result of the last application of `f` (or the start value if the list was empty).

```
fun foldl s f nil      = s
  | foldl s f (x::xs) = foldl (f(s,x)) f xs
```

B.3. FunkyIMP Code

B.3.1. Full Example class

The following code is an example of what valid funkyIMP code looks like. First, a domain for three-dimensional arrays is created, and then a class with three static methods is defined. The `main` method functions similarly to the main method in C++, with the first parameter being the number of command line arguments, and the second being the actual arguments. In `main` a new array (`test`) is created and initialized. Then `test'a` is assigned the result of `f`, which performs a domain iteration on `test`.

```
domain three_d{x,y,z}: one_d{x*y*z}(o) =
{
  (j,k,l) | j<x & k < y & l < z
}

public class cur
{
  static int g(int x)
  {
    cancel 2*x;
  }

  static int[three_d{256,256,256}] f(int ma[three_d{256,256,256}])
  {
    cancel ma.\[a,b,c] {ma[c,a,b]+g(a)};
  }

  static int main(int argc, inout unique String[one_d{-1}] argv)
  {
    int test[three_d{256,256,256}]
      = new int[three_d{256,256,256}].\[x,y,z]{x+y+z};
    test'a = f(test);
    finally 0;
  }
}
```

B.3.2. Automated Test Template class

The following code is used as a template for the automated Test suite. It uses macros to execute a test over different array sizes. The code that is to be executed in the domain iteration is inserted at the `/*!!*/` comment.

```
import ffi.stdio;
import stdlib.vector;
import stdlib.gVector;
```

```
#define TYPE float

#define FUNCNAME(HEIGHT, WIDTH) transpose_ ## HEIGHT ## _ ## WIDTH

#define BENCHMARK(HEIGHT, WIDTH) static \
                        TYPE[two_d{HEIGHT, WIDTH}] \
        FUNCNAME(HEIGHT,WIDTH) (TYPE matrix[two_d{HEIGHT, WIDTH}])\
    {\
        cancel new TYPE[two_d{HEIGHT, WIDTH}].\[x,y]{/*!!*/};\
    }

#define MATRIX_NAME(HEIGHT, WIDTH) matrix_ ## HEIGHT ## _ ## WIDTH
#define BENCHMARK_CALL(HEIGHT, WIDTH) \
    TYPE MATRIX_NAME(HEIGHT, WIDTH)[two_d{HEIGHT, WIDTH}] \
        = new TYPE[two_d{HEIGHT, WIDTH}].\[x,y]{(x*10)+(9-y)};\
    FUNCNAME(HEIGHT, WIDTH)(MATRIX_NAME(HEIGHT, WIDTH))

domain two_d{x,y}:one_d{x*y}(o) = { (j,k) | j<x & k<y }

public class cur
{

    BENCHMARK(32,32)
    BENCHMARK(32,64)
    BENCHMARK(64,64)
    BENCHMARK(64,128)
    BENCHMARK(128,128)
    BENCHMARK(256,128)
    BENCHMARK(256,256)
    BENCHMARK(256,512)
    BENCHMARK(512,512)
    BENCHMARK(512,1024)
    BENCHMARK(1024,1024)
    BENCHMARK(2048,1024)
    BENCHMARK(2048,2048)
    BENCHMARK(4096,2048)
    BENCHMARK(4096,4096)
    BENCHMARK(4096,8192)
    BENCHMARK(8192,8192)

    static int main(int argc, inout unique String[one_d{-1}] argv)
    {
        BENCHMARK_CALL(32,32);
        BENCHMARK_CALL(32,64);
        BENCHMARK_CALL(64,64);
        BENCHMARK_CALL(64 ,128);
        BENCHMARK_CALL(128 ,128);
        BENCHMARK_CALL(256 ,128);
        BENCHMARK_CALL(256 ,256);
        BENCHMARK_CALL(256 ,512);
        BENCHMARK_CALL(512 ,512);
        BENCHMARK_CALL(512 ,1024);
        BENCHMARK_CALL(1024 ,1024);
        BENCHMARK_CALL(2048 ,1024);
        BENCHMARK_CALL(2048 ,2048);
        BENCHMARK_CALL(4096 ,2048);
        BENCHMARK_CALL(4096 ,4096);
        BENCHMARK_CALL(4096 ,8192);
        BENCHMARK_CALL(8192, 8192);
        finally 0;
    }
}
```

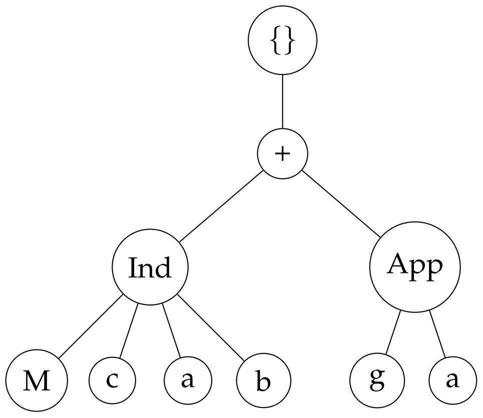
C. Examples

C.1. Collection of identifiers

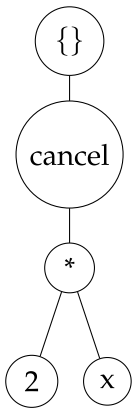
In this example, the sample code given in B.3.1 will be used to illustrate the analysis given in section 3.2.1. The identifier collection analysis described in section 3.2.1 will be performed upon the domain iteration C.1. First, the syntax tree is constructed, see figure C.2a. As in the iteration the function `g` is called and its syntax tree (figure C.2b) is also needed. Now the analysis, in form of equation 3.2 may be applied to the tree, yielding $\mathcal{E} = \{M, c, a, b, g, x\}$ as a result. Taking out unneeded identifiers yields the final set of interesting variables, $\mathcal{E}^* = \{M, g\}$

```
cancel ma.\[a,b,c] {ma[c,a,b]+g(a)};
```

Figure C.1.: Sample code that will be translated



(a) The syntax tree for the domain iteration C.1



(b) The syntax tree for the function `g`.

Figure C.2.: Syntax trees for the code specified in C.1

$$
\begin{aligned}
\mathcal{E} &= [\![\{M[c,a,b] + g(a)\}]\!]^\sharp \\
&= [\![M[c,a,b] + g(a)]\!]^\sharp \\
&= [\![M[c,a,b]]\!]^\sharp \cup [\![g(a)]\!]^\sharp \\
&= [\![M]\!]^\sharp \cup [\![c]\!]^\sharp \cup [\![a]\!]^\sharp \cup [\![b]\!]^\sharp \cup [\![g]\!]^\sharp \cup [\![a]\!]^\sharp \\
&= \{M\} \cup \{c\} \cup \{a\} \cup \{b\} \cup \{g\} \cup [\![\{\text{cancel}\, 2 * x\}]\!]^\sharp \cup \{a\} \\
&= \{M, c, a, b, g\} \cup [\![\text{cancel}\, 2 * x]\!]^\sharp \\
&= \{M, c, a, b, g\} \cup [\![2 * x]\!]^\sharp \\
&= \{M, c, a, b, g\} \cup [\![2]\!]^\sharp \cup [\![x]\!]^\sharp \\
&= \{M, c, a, b, g\} \cup \emptyset \cup \{x\} \\
&= \{M, c, a, b, g, x\}
\end{aligned}
$$

Figure C.3.: Computation of the set $\mathcal{E}$ of all needed identifiers

C.2. Generation of Host Code

This section illustrates the process of code generation for the host side, as described in section 3.2 with the domain iteration C.1 with the following listing. The listing shows the code the compiler generates from the function `f` specified in section B.3.1. The code is slightly edited to make it more readable.

```
funky::LinearArray< int >::Version*
  cur::f_int_three_d_256_256_256___int_three_d_256_256_256__
  (funky::LinearArray< int >::Version* ma)
{
  return [&]() ->funky::LinearArray< int >::Version* {
    if (1) {
      return
        ([&]()->funky::LinearArray< int >::Version*
        {
          funky::LinearArray< int >::Version*__VAL_TMP0=ma;
          funky::LinearArray< int >::Version*__EXP_TMP0
            = __VAL_TMP0->getNewVersion();

          // Compile the kernel
          ocl_kernel f_0(&device,"tmp/f_0.cl");

          // Translate ma into a format
          // that is understandable by OpenCL
          int *__ma_0=ma->toNative();
          ocl_mem __ma_GPU_0
```

```
            = device.malloc(sizeof(int)*16777216,
                              CL_MEM_READ_ONLY);
          int __ma0_dim_0=256;
          int __ma0_dim_1=256;
          int __ma0_dim_2=256;

          // Create the return value
          int *__return_val_0=new int[16777216];
          ocl_mem __return_val_GPU_0
            = device.malloc(sizeof(int)*16777216,
                              CL_MEM_WRITE_ONLY);
          // Copy ma to the GPU
          __ma_GPU_0.copyFrom(__ma_0);

          // Set the Kernel Arguments
          f_0.setArgs(__ma_GPU_0.mem(),&__ma0_dim_0,
                      &__ma0_dim_1,&__ma0_dim_2,
                      __return_val_GPU_0.mem());

          //Run the kernel and wait for it to be done.
          int id = f_0.timedRun(1024, 16777216);
          device.finish();

          // Translate the return value back to the
          // funkyIMP format.
          __return_val_GPU_0.copyTo(__return_val_0);
          funky::LinearArray<int> *__return_LINARR0
            = new funky::LinearArray<int>(16777216,
                                            __return_val_0);
          funky::LinearArray< int >::Version* __return_0
            = new funky::LinearArray<int>
                ::Version(__return_LINARR0,3,256,256,256);
              return __return_0;
            }) ();
        } else {
            // Generated CPU iteration
        }
    }();
}
```

C.3. Generation of GPU Code

This section illustrates the process of code generation for the GPU side, as described in section 3.3, using the domain iteration C.1 with the following listing. The listing shows code the OpenCL kernel the compiler generates from the domain iteration expression body in function `f` specified in section B.3.1, with a slightly edited layout to make it fit the page.

```
//FUNCTION HEADER DECLS
int  __s_int_g_int(int  __x);

//FUNCTION BODY DECLS
int  __s_int_g_int(int  __x)
{
    return 2*(__x);
}


//KERNEL CODE
__kernel void f_0(__global int *__ma_0, int __ma0_dim_0,
        int __ma0_dim_1, int __ma0_dim_2,
        __global int *__return_0)
{
    size_t __CUR_POS_0=get_global_id(0);
    int __a_0=(__CUR_POS_0/65536)%256;
    int __b_0=(__CUR_POS_0/256)%256;
    int __c_0=(__CUR_POS_0/1)%256;
    __return_0[__CUR_POS_0]=((__ma_0)[(__b_0) + (__a_0) * 256L
        + (__c_0) * 65536L])+(__s_int_g_int(__a_0));
}
```

C.4. Function dependencies

In this section the dependencies between functions will be illustrated. Consider the following example:

```
static int[two_d{5,5}] kernel(int ma[two_d{5,5}]) {
    cancel ma.\[r,c]{ma[c,r] * a(c)};
}

static int a(int i) {
    cancel b((int)f(i)*3) + f(2);
}

static int b(int i) {
    int x = 5 * (7 + i);
    x'doubl = 2*x;
    cancel d(x'doubl);
}

static int d(int i) {
    if (!(i < 0)) {
        cancel f(i*2) - f(5);
    } else if (i == 0) {
        cancel 9;
    } else {
        cancel g(i*3)*f(3);
    }
}

static int f(int i) {
    cancel g(i*3);
}

static int g(int i) {
    cancel h;
}

static int h(int x) {
    cancel 42;
}
```

This listing can be translated into the call graph given in figure C.4. The graph does not contain any cycles, that means that a topological order exists. It looks as follows: $L = [h, g, f, d, b, a]$.

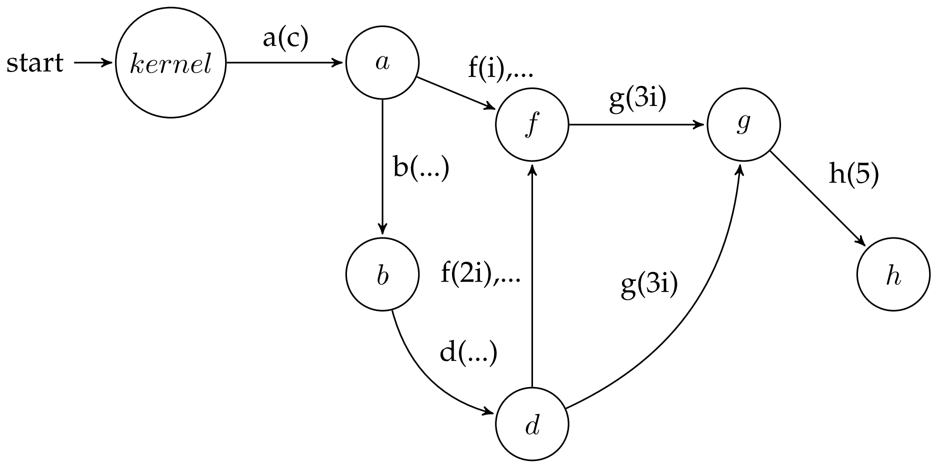


Figure C.4.: Callgraph of the code listing given in section C.4

D. Configurations

D.1. Random Example Code Generator Configuration

The tool that generates example domain iterations can be configured using a standard Java Properties file. These files consist of key-value pairs and comments[1]. The syntax for the key-value pairs is as follows:

```
com.example.property=exampleValue
com.example.num=233
```

There are six proterties in the configuration file of the Code Generator. They are depicted in the table below:

Key	Description
edu.tum.funky.codegen.allow-complex	Boolean value that indicates whether to allow complex memory accesses with more than one node.
edu.tum.funky.codegen.allow-float-div	Boolean value that indicates whether to allow floating-point divisions.
edu.tum.funky.codegen.min-nodes	Integer value that indicates the minimum number of nodes in the example domain. It must be positive.
edu.tum.funky.codegen.max-nodes	Integer value that indicates the maximum number of nodes in the example domain. It must be positive.
edu.tum.funky.codegen.generated-examples	Integer value that indicates the number of examples that will be generated. It must be positive.
edu.tum.funky.codegen.output-path	String that marks the location the Code generator will output its example classes to.

[1]Comments are marked with #

Bibliography

[1] Guru 3D. Radeon hd 5750 review (crossfire), October 2009. http://www.guru3d.com/articles_pages/radeon_hd_5750_review_(crossfire),3.html (Retrieved on August 10th, 2014).

[2] Alexander Pakosta Alexander Weiler. High-speed layout guidelines. Technical report, Texas Instruments, November 2006.

[3] AMD. Radeon hd 5750. http://www.amd.com/en-us/products/graphics/desktop/5000/5750 (Retrieved on August 10th, 2014).

[4] David Blythe. Rise of the graphics processor. *Proceedings of the IEEE*, 96(5):761–778, 2008.

[5] Yusuke Fujii, Takuya Azumi, Nobuhiko Nishio, Shinpei Kato, and Masato Edahiro. Data transfer matters for gpu computing. 2013.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[7] The Khronos Group. The khronos group releases opencl 1.0 specification. Press Release, December 2008. https://www.khronos.org/news/press/the_khronos_group_releases_opencl_1.0_specification (Retrieved on March 11th, 2014).

[8] The Khronos Group. The opencl specification. Technical Specification, November 2012. Version 1.2, https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf (Retrieved on March 11th, 2014).

[9] Alexander Herz. funkyimp. Wiki, January 2013. http://www2.in.tum.de/funky (Retrieved on March 19th, 2014).

[10] Wen-Mei Hwu, Christopher Rodrigues, Shane Ryoo, and John Stratton. Compute unified device architecture application suitability. *Computing in Science & Engineering*, 11(3):16–26, 2009.

[11] Apple Inc. Opencl programming guide for mac. Programming Guide, August 2012. https://developer.apple.com/library/mac/documentation/Performance/Conceptual/OpenCL_MacProgGuide/OpenCL_MacProgGuide.pdf (Retrieved on March 12th, 2014).

[12] Intel. X79 express chipset block diagram. Diagram on a Website, December 2011. http://www.intel.de/content/www/de/de/chipsets/performance-chipsets/x79-express-chipset-diagram.html (Retrieved on June 3rd, 2014).

[13] Vahid Jalili-Marandi and Venkata Dinavahi. Simd-based large-scale transient stability simulation on the graphics processing unit. *Power Systems, IEEE Transactions on*, 25(3):1589–1599, 2010.

[14] U. Kuckartz, S. Rädiker, T. Ebert, and J. Schehl. *Statistik: Eine Verständliche Einführung*. VS Verlag fur Sozialwissenschaften GmbH, 2010.

[15] David Medina. Opencl c++ wrapper. GitHub, February 2013. https://github.com/dmed256/OpenCL-Wrapper (Retrieved on April 8th, 2014).

[16] notebookcheck.com. Intel hd graphics 4000. Product Review. http://www.notebookcheck.com/Intel-HD-Graphics-4000.69166.0.html (Retrieved on August 5th, 2014).

[17] Nvidia. Opencl programming guide for the cuda architecture. Programming Guide, September 2012. http://hpc.oit.uci.edu/nvidia-doc/sdk-cuda-doc/OpenCL/doc/OpenCL_Programming_Guide.pdf (Version 4.2, Retrieved on June 18th, 2014).

[18] Nvidia. Opencl programming guide for the cuda architecture. Programming Guide, September 2012. http://hpc.oit.uci.edu/nvidia-doc/sdk-cuda-doc/OpenCL/doc/OpenCL_Programming_Guide.pdf (Version 4.2, Retrieved on June 18th, 2014).

[19] Elektronik-Kompendium.de (Patrick Schnabel). Ddr-sdram (ddr1 / ddr2 / ddr3 / ddr4). Website, January (?) 2014. http://www.elektronik-kompendium.de/sites/com/1312291.htm (Retrieved on June 4th, 2014).

[20] Jonathan Tompson and Kristofer Schlachter. An introduction to the opencl programming model. *Person Education*, 2012.

[21] Rafael Alejandro Vejarano, Phuong Thi Yen, and Jeong-Gun Lee. Parallel acceleration on manycore systems and its performance analysis: Opencl case study. *International Journal of Software Engineering & Its Applications*, 7(3), 2013.

[22] xbitlabs.com. Intel hd graphics 4000 and intel hd graphics 2500 review. Product Review, June 2012. http://www.xbitlabs.com/articles/graphics/display/intel-hd-graphics-4000-2500.html (Retrieved on July 8th, 2014).