Performance Analysis and Optimization of the Equelle DSL

Kristian Hasli Johnsen

Thesis submitted for the degree of
Master in Informatics: Technical and Scientific Applications
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Autumn 2018

Performance Analysis and Optimization of the Equelle DSL

Kristian Hasli Johnsen

Performance Analysis and Optimization of the Equelle DSL

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

In recent years there has been an increased interest in creating high-level programming languages for domains that require high performance. The aim is to let the programmer focus on the applications using her domain-specific knowledge while the compiler and backend handles the rest.

In this thesis, we conduct a performance analysis and optimize the CUDA backend of Equelle, a domain-specific language (DSL) for solving partial differential equations (PDE) using the finite volume method (FVM). We also give an introduction to DSLs, a suvey of existing DSLs for high performance computing, as well as an overview of frameworks and tools that have been made to lower the effort of making new DSLs. Concludingly, we present candidates for future work for the CUDA backend.

Contents

List of Figures

List of Listings

List of Tables

Acknowledgements

I would first of all like to thank my supervisors Atgeirr Flø Rasmussen and André R. Brodtkorb for all their guidance and support throughout my thesis work. Working on the Equelle compiler and the CUDA backend has been a perfect fit for my interests.

I would also like to thank my parents Per Reinhart Johnsen and Anne-Mai Hasli for their encouragement and support.

Chapter 1

Introduction

In this thesis we conduct a performance analysis and optimize the CUDA backend of the Equelle DSL. We also give an introduction to domain-specific languages (DSLs), as well as a survey of previous and current developments in the field.

Equelle is a DSL for solving partial differential equations (PDEs), using the finite volume method (FVM). Designing such simulators in a way that is both type safe and efficient is a tough problem when using general-purpose languages, and requires expertise in numerical mathematics as well as an understanding of the underlying hardware and parallel algorithms. Equelle aims to provide a high level syntax that lets the programmer write code that closely resembles the PDEs in their numerical form, and still achieve both safety[1] and good performance. By doing this, the programmer can focus on the mathematics involved, and leave the rest to the compiler and the backend of Equelle.

The down-side to DSLs is that they are very hard to develop. Developing DSLs requires knowledge about the domain it targets, compiler construction as well as the architectures that they are meant to run on. However, in recent years there has been an increased interest in using DSLs. Tools have been developed to make it easier to create new DSLs.

CUDA is Nvidia's proprietary platform for programming GPUs to perform general-purpose computations and is popular in fields that benefit from high parallel performance. CUDA provides its own programming model that lets the programmer have a high degree of control over the GPU's resources, i.e., memory or processor cores. Equelle's CUDA backend was implemented in 2014 [17] so that the system can benefit from the high computational intensity of modern GPUs. It builds on the CPU backend which uses operator overloading extensively to conveniently implement automatic differentiation.

1.1 Motivation

Although the CUDA backend achieves better performance than the CPU backend, it is designed in such a way that the GPU is not fully utilized. In this thesis we analyze the backend by using profilers in order to get an overview of its underlying problems. The results will serve as a basis for our own implementations as well as our suggestions

[1] By safety we mean code that is free of bugs, mathematical errors and logical errors.

for future work. Hopefully, our work can serve as valuable background material for future development on the Equelle DSL. Although we mostly use the implicit heat equation simulation[2] as benchmark case for our analysis, we should still be able to generalize our results to other Equelle programs, since there is a large overlap in the operators they use, and similarities in structure.

1.2 Organization of the Thesis

In the following chapter, we start by giving an introduction to DSLs. We will then go through concepts in compiler construction, and give an account of frameworks and tools that have been made to lower the effort of developing DSLs. We also provide a survey of DSLs that are similar to Equelle, in that they also require high performance. Next, we give a short introduction to CUDA, and to the Equelle language, and finally we go through tools and best practices that are useful when doing CUDA programming. In Chapter 3 we present our performance analysis of the Equelle CUDA backend. In Chapter 4 we go through implementations and the results we have achieved. In Chapter 5 we conclude the thesis and make suggestions for future work.

[2] The complete Equelle code for the implicit heat equation can be found in Appendix A.1.

Chapter 2

Background

2.1 Introduction to Domain-specific Languages

A domain-specific language (DSL) is a language which is designed for a specific purpose. Domain-specific languages have several advantages over the more well known general-purpose languages, such as C++, Java and Lisp. With a syntax that can express the ideas in the domain at hand in a more intuitive way, a DSL lets programmers reason about their programs on a higher level, helping them to be more creative and productive. DSLs also enable the implementation of data structures and algorithms for traversal and computations that is otherwise non-optimal or even infeasible to implement when using a general-purpose language. By applying these ideas to domains that demand high parallel performance, we can write programs that are performance portable[1]. Another benefit that is mentioned in the literature, is reusability. Elements of DSLs such as grammars and domain abstractions can easily be reused, and some DSLs support translation of programs into application libraries for other systems to use as for example plugins. As demonstrated by Bentley [2][2], DSLs can also be used to make new DSLs.

It is not always clear whether a programming language is domain-specific or general-purpose. In fact, several languages that were designed for a specific purpose in the past, are considered general-purpose[3] today. However, one could evaluate this as a continuous spectrum, with pure general-purpose languages like C++ on one side, and a strictly domain-specific one like Diderot on the other. One factor that contributes to defining a language as a DSL is the restrictions put on the programming model. With the proper restrictions, the programming model can be limited to only express concepts in the domain.

Even though the benefits of using a DSL are apparent, there are serious drawbacks to consider. Developing a DSL can be a difficult, tedious and costly process, and a large body of work have been put into finding out how DSLs can be built that are *good enough*, while minimizing the effort needed. What "good enough" entails depends

[1] Performance portability: performance scalability across different hardware architectures with little or virtually no effort.

[2] In his column *Little Languages*, Bentley uses the line drawing language PIC as a back-end for a language that defines chemical structure diagrams. CHEM's output is PIC's input. This approach is called *pipelining*.

[3] Fortran for numerical calculations, COBOL for business applications.

on factors such as performance, size of the user community and the chances of it surviving in the long run.

In 2010, Chafi et al. [7] proposed DSLs as a means to shield the ordinary programmer from the variations and complexities of programming models in modern hardware architectures. Since then, numerous DSLs have been designed to tackle this problem in the domains of image processing, image analysis, scientific visualization, physical simulation and similar areas. Due to the difficulty of developing the DSLs themselves, several frameworks and development tools have been made. In the following sections we present a selection of DSLs, development tools and frameworks, as well as the structure of an optimizing compiler. Please note that this by no means represents the full body of work in this field. The intent is to give an account of current trends.

2.2 Developing Domain-specific Languages

In this section we will give an introduction to the development of DSLs. We start by describing the structure and basic concepts of an optimizing compiler. Afterwards, we give an account of development tools and frameworks that have contributed to lowering the effort of developing DSLs. In the next section we will have a look at DSLs where some of these tools and frameworks have been used.

2.2.1 Structure of a Compiler

A compiler is a computer program that translates one language into another. The input is a program written in the source language, and the output is a corresponding program in the target language. In this text we will focus on a modern perspective where a compiler consists of three major components: a frontend, an optimizer and a backend. Modern compilers and compiler frameworks are often designed to be modular. For instance, an optimizer can have several compatible frontends and backends.

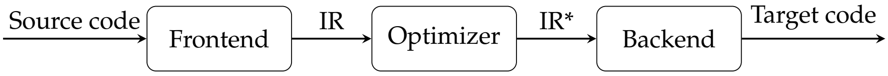


Figure 2.1: The three phases of an optimizing compiler. The compiler frontend takes the source program as input, the compiler then processes the program and outputs the corresponding program in the target language.

Frontend

A compiler frontend contains the following stages:

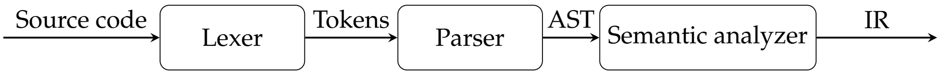


Figure 2.2: The frontend of a compiler.

- Scanning, also called lexing. In this stage, a *Lexer* reads the input (source code) and splits it into segments called *tokens*. The output is a stream of tokens. A lexer program can be implemented by specialized tools such as Flex and JFlex.

- Parsing, also called syntax analysis. The token stream is read by the *Parser* and checked for well-formedness, usually specified as a context-free grammar. The parser produces a tree structure of the program which is simplified to a *abstract syntax tree* (AST). A parser can be implemented using parser generators such as GNU Bison and CUP.

- Semantic analysis. Here the parse tree is checked for semantic correctness, for example by checking types, or checking whether a variable is declared before use. A DSL might for instance check if the dimensions of a vectors and a matrices in multiplications are compatible. In general: "Does this make sense?". The output of this stage is usually an augmented version of the AST, containing relevant information for the next stages. Alternatively, the AST can be converted into an intermediate representation.

An *intermediate representation* (IR for short), is code or a data structure which represents the source program in a format which is meant for optimizations and translation. An IR can be in any form, but there are certain types that are popular, such as various versions of three-address-code[4] (3AC or TAC), or one-address-code (Pascal- or P-code). P-code operates on a stack and is commonly used in bytecode. 3AC is more commonly used in optimizing compilers.

A property commonly inherent in an intermediate representation is static single assignment (SSA), meaning that a variable is read-only once it is assigned a value. This greatly simplifies optimizations.

Optimizer

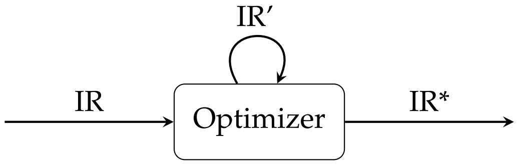


Figure 2.3: The optimizer of a compiler.

The optimizer performs optimizing passes over the intermediate representation. Typical optimizations are:

- Constant folding, which replaces variables by constants in places where it is certain that the value will not change during runtime.

- Dead code elimination, where code which is *unreachable* is removed.

[4] 3AC consists of basic instructions which contain an operation and at most 3 addresses, for instance in memory, or a register identifier. e.g. "t1 = a + b"

- Common subexpression elimination, where identical expressions are identified and possibly replaced by a single variable.

- Domain-specific optimizations such as program rewriting using mathematical identities.

Backend

A backend, or simply a code generator, translates the intermediate representation into code for one or more targets. A *target* can be either another high-level programming language, or a machine specific low-level language.

Typically, when generating code on machine level, the process consists of three main tasks: *instruction selection*, *register allocation and assignment*, and *instruction ordering*. When performing instruction selection, the code generator selects an instruction from the instruction set of the target architecture (or language), which correctly represents the intent of the program. Registers are units of storage located on processing units that are limited in size and few in numbers, but much faster than the main memory. Efficient management of registers is extremely important to obtain efficient code, as continuously fetching data from main memory is very expensive. When generating code for another high-level language (source-to-source), the process consists of mapping concepts in the source language into appropriate concepts in the target language.

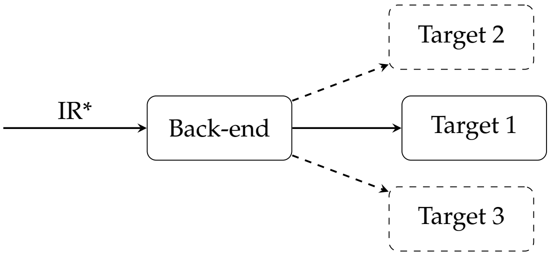


Figure 2.4: The backend of a compiler performs code generation for one or more targets by translating the optimized IR into target-specific code.

2.2.2 Implementing DSLs

There are two main classes of DSLs: those with complete standalone compilers (often called external DSLs), and embedded DSLs (often called internal DSLs). An internal DSL is implemented inside another programming language by defining new language constructs tailored to the domain. Living inside another programming language can lower the development effort of the DSL remarkably, but if the hosting language is not well suited for hosting an embedded language, it could set serious limitations for the implementation. Also, most general-purpose languages do not provide suitable error reporting and debugging for domain-specific implementations.

External DSLs on the other hand, provide their own complete compiler pipeline. This approach has several benefits. Firstly, the syntax of the language can be implemented to more closely resemble the notation of the domain. Secondly,

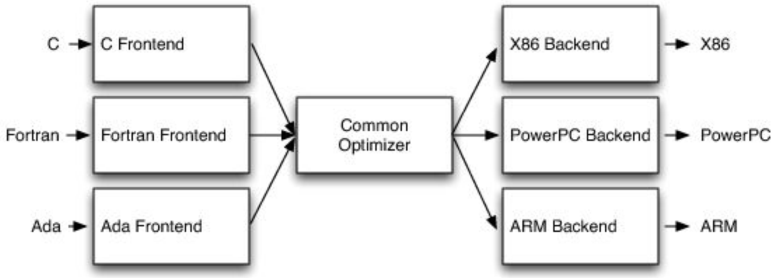


Figure 2.5: LLVM provides an optimizer that can be used by several frontends and backends. Each frontend can be compiled to the same targets. Source: http://www.aosabook.org/en/llvm.html

semantics can be designed to more closely model the constructs in the domain. However, the effort to develop a full compiler can be immense. That is why the *language virtualization* was proposed by Chafi et al. [7]. Virtualization will receive more attention in the sections on Scala.

There are compiler approaches that contain a mix of these two, but we will treat all DSLs that are not pure external ones as internal DSLs.

Development Tools and Frameworks

In this section, we provide a review of some notable tools and frameworks that are either key components in the development of the DSLs included in this survey, or that have otherwise made considerable contributions to the research of developing high performance DSLs with a reasonable amount of effort.

LLVM. The LLVM Project is a collection of modular and reusable compiler and toolchain technologies and has grown to be very popular since its inception in 2003. As we can see in Figure 2.5, LLVM follows the three-part model previously given, where several frontends (programming languages) can use the same optimizer and backend. This is possible because LLVM provides its own standardized intermediate representation (the LLVM IR). One of LLVM's current weaknesses is the limited support for compiling to parallel targets. LLVM is used as a backend to generate efficient code for several of the DSLs in this survey, and is also implemented in the Terra systems programming language discussed further down.

Multi-stage programming, or *staging* for short, is a development technique in which a programmer can define when code will be evaluated (i.e. at which *stage*), as well as employ code generation during runtime. This mechanism has proven to be useful when developing high performance DSLs because it provides the means to add additional layers of abstraction (i.e. domain abstractions) with optimizations. The first stage compiles the the program in a traditional manner, constructing a program representation which is passed on to the future stages. Subsequent stages occur during runtime. Most programming languages do not support staging natively, but some have been extended to support it [36].

Scala is a general-purpose functional programming language which has been extended in order to make it a better host language. There are two inter-related projects: Scala-Virtualized [25] and Lightweight Modular Staging (LMS) [30, 31].

Scala-Virtualized is an experimental branch of the Scala compiler which aims to make embedding DSLs more seamless by providing functionality to define existing Scala constructs in terms of the DSLs. For instance, control structures such as while-loops, if-then-else statements and object constructions can be overloaded or overridden. In addition, new infix-operators can be added to existing types. Scala-Virtualized also uses Scala's implicit parameters which lets DSLs fetch source file information and line numbers, thus improving debugging capabilities.

Lightweight Modular Staging (LMS) is a framework for runtime code generation and embedded compilers. Unlike Scala-Virtualized, LMS does not change the Scala compiler itself, but is implemented as a library which provides functionality that can be used to define IRs, and perform suitable optimizations and translation. LMS uses a type based form of multi-staging, where a staged program is defined purely as a collection of types with dependencies, and the binding times are defined using type annotations[5]. The result is a high level program representation in the form of a graph where elements can move freely ("sea of nodes"). Using such a graph based IR, with a high abstraction level, enables the implementation of relatively straightforward optimizations. For instance, the framework provides global common subexpression elimination out of the box, which ensures no code duplication.

Together, Scala-Virtualized and LMS provide what Scala needs to transform a pure library based embedded language into one that is close to a standalone implementation in terms of *expressiveness*, *performance*, *safety*[6] and with *modest effort*. We say that Scala is *virtualizable* [7]. LMS provides performance because the DSL developer can define domain abstractions that can be optimized from both a generic and domain-specific perspective, and can be translated into low-level optimized code. Scala-Virtualized provides expressiveness with modest effort, as domain abstractions more easily can be mapped to appropriate syntax. Safety is achieved using a technique known as *finally tagless* [6] or *polymorphic embedding* [16]. Lastly, an important part of maintaining modest effort is the ability to reuse ideas and implementations. Delite which is discussed next, is a good example of how Scala can be used for this.

Delite [5, 33] is a compilation and runtime framework for embedded DSLs. It is developed in Scala using Scala-Virtualized and LMS, to make it easier to implement DSLs that are efficient, expressive and parallel performance portable. In the words of Arvind K. Sujeeth et al. [32] "Delite is essentially a Scala library that DSL authors can use to build an intermediate representation (IR), perform optimizations, and generate parallel code for multiple hardware targets."

Traditional compilers often structure their IRs as control flow graphs defining the execution of the code. Delite on the other hand, uses the "sea of nodes" representation from LMS, allowing optimizations to be performed from three perspectives: generic, parallel and domain-specific.

[5] Rep[T], where T is an arbitrary type, is used to define an expression which *represents* the type T in a future stage.

[6] A DSL is considered "safe" if programs are guaranteed to maintain certain properties defined by the DSL. Limited interference from host constructs that are not part of the DSL is important.

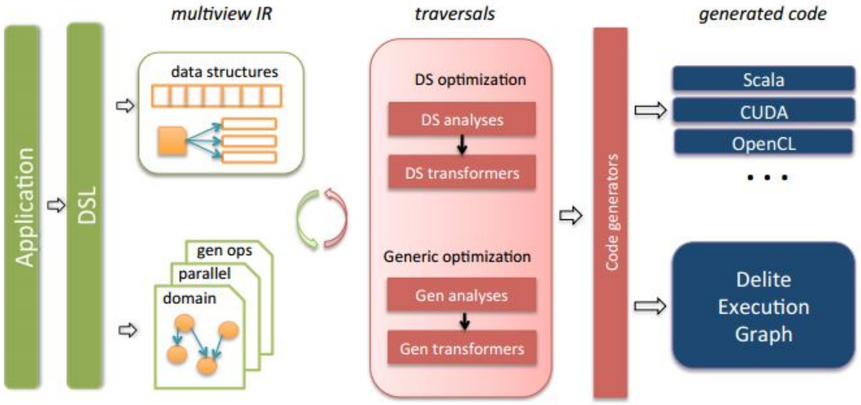


Figure 2.6: The Delite pipeline. The application code is lifted into a DSL representation, which is repeatedly optimized from three perspectives (generic, parallel and domain-specific). After optimizations are complete, code is generated for each target (Scala, CUDA, etc.), as well as an execution graph containing dependencies between computations.

The lowest-level view of the IR is the *generic* one, centered around symbols and definitions. Since the IR nodes are defined as Scala classes, certain optimizations can easily be performed. For instance, during IR construction common subexpression elimination can be performed simply by checking if a node already exists in the graph. Optimizations that can be performed after the IR is constructed include dead code elimination and operator fusions (i.e. loop fusion). The result of these generic optimizations is an optimized program in block structure.

When viewing the IR with regard to *parallelism*, the compiler extends the nodes using predefined *Delite ops* representing parallel patterns (sequential, reduce, map, map-reduce).

In the domain-specific view of an IR, the DSL developer extends the Delite ops with domain-specificity. For instance, using pattern matching to recognize a double matrix transpose and implement the correct transformation.

The main focus of the Delite project is reuse [32]. Delite consists of a large set of components that are both reusable, extendable and overridable. These include code generation implementations, for instance if the DSL developer wishes to create a hand-optimized version for CUDA. Other elements for reuse are the operators (*ops*).

DSLs developed using Delite include OptiML for machine learning, OptiGraph for graph analysis, OptiQL for data querying and OptiMesh for solving PDEs on unstructured meshes[7].

Terra is a low-level programming language which is made with interoperability in mind. It is syntactically embedded in, and metaprogrammed from Lua. Lua[8] is a lightweight, embeddable programming language which supports a diverse selection of programming models: procedural programming, object-oriented programming,

[7] OptiMesh is Liszt implemented in Delite.

[8] https://www.lua.org/about.html

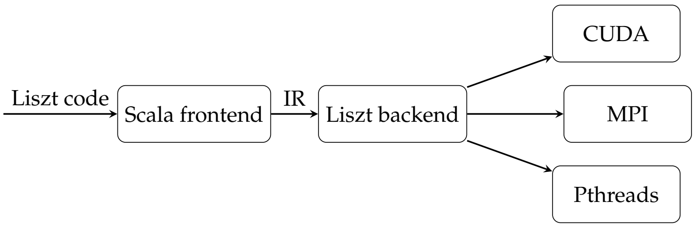


Figure 2.7: Schematic overview of the Liszt compiler.

functional programming, data-driven programming and data description.

A consequence of programming languages not being developed with interoperability in mind, is that *glue code* often is needed to make the parts communicate. For instance, when using C together with Python, glue code to prevent Python objects from being garbage collected is needed. Also, data which is passed between the runtimes might need explicit conversions. Another problem, is that the languages might have an overlap in responsibilities, adding extra layers of complexity.

In **Lua**, Terra entities such as variables, functions and expressions are treated as first-class values (they can be stored in Lua variables, returned from statements etc.), thus eliminating the need for glue code.

The backend of Terra uses LLVM for optimizations and generation of machine code. Terra is modelled to be similar to C in several aspects, with manual memory management using *malloc* and *free*, as well as having similar semantics. A library function (terralib.includec) can be called to parse C files and generate bindings to C code.

2.3 Survey of High-performance DSLs

In this section we provide a survey of a selection of DSLs which demand high parallel performance. They can all be placed in the domains of image processing, image analysis, scientific visualization or physical simulations. An overview of an extended selection can be found in Table 2.1.

Liszt [11] is a mesh based DSL for solving partial differential equations. Its compiler is implemented with a Scala frontend (LMS and Scala-Virtualized) which emits Liszt IRs to the custom Liszt backend using a plugin. The backend then performs transformations and optimizations on the IR, and generates code for CUDA, pthreads[9] and MPI[10]. Liszt's most notable weaknesses are that it is limited to mesh based-simulations, and not well suited for implicit methods.

Ebb [4] is the second generation of Liszt and improves on several of Liszt's weaknesses. A three-layer-model is applied, separating simulation code, definition of data structures for geometric domain and finally runtimes supporting parallel architectures. The model is illustrated in figure 2.8. This model supports utilization of several

[9] Pthreads is a threading implementation based on POSIX, targeting UNIX compatible systems.

[10] MPI, or Message Passing Interface is a standardized and portable API for distributed computing.

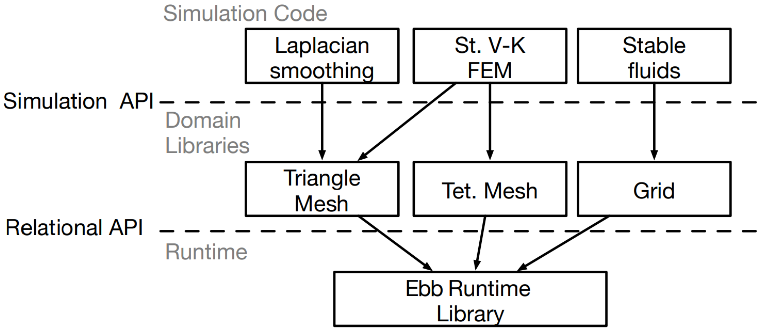


Figure 2.8: The figure shows the three-layer-model of Ebb, where simulation code, domain definitions and runtimes targeting parallelism are separated. Domains can be combined in simulations if wished, as can be seen in the FEM simulation.

geometric domains simultaneously. Ebb is developed using Lua-Terra and thus with interoperability in mind. The authors say that Ebb is "the first step to building an integrated simulation environment".

Simit [23] uses a model which lets the programmer view its domain in two ways: as a graph, which is useful for local computations, and as matrices, which is useful for global linear algebra. Two different code generation platforms are used: LLVM for CPU code, and Nvidia's LLVM based framework NVVM. Simit can be integrated in existing C++ implementations since it is composed of C++ APIs. It is planned that Simit will use the Tensor Algebra Compiler (taco)[11] for its backend in the future.

Diderot [21] is a DSL for portable parallel scientific visualization and image analysis. It was developed due to the lack of mathematical abstraction and difficulty of parallelization in scientific visualization and image analysis. Diderot has the most focus on domain specific notation of the DSLs and uses Unicode encode mathematical notation. $(\nabla, \delta_{ij}, \epsilon_{ijk}, etc)$. Their IR, called EIN lets the compiler perform clever optimizations based on mathematic identities.

Kindlmann et al. chose to develop Diderot as a standalone compiler, because they didn't want to restrict themselves to the limitations of other languages. However, the system can interoperate with other systems by compiling applications to libraries in C, exposing global variables and other parameters. A simple schematic overview of the Diderot compiler can be seen in Figure 2.9.

Halide [24, 27, 28] is a DSL for optimizing parallelism, locality and recomputation in image processing pipelines. The compiler decouples algorithms and schedules, making it possible to achieve high performance without affecting the logic of the implementation. Algorithms in Halide are defined as pipelines of functions. The functions are declarative[12] (meaning no side effects, no explicit loops). The schedules in Halide represent the order in which the functions are executed. Halide uses a

[11] http://simit-lang.org/

[12] Also known as pure functional.

stochastic (genetic) algorithm in order to generate and pick an optimal schedule for a given program. Even though Halide has been proven to be a very efficient DSL, the schedule optimizer has been criticized to not be optimal. Other solutions to the schedule optimization have been proposed more recently [26]. The Halide pipeline can be seen in Figure 2.10. Halide is still in active development at the time of writing.

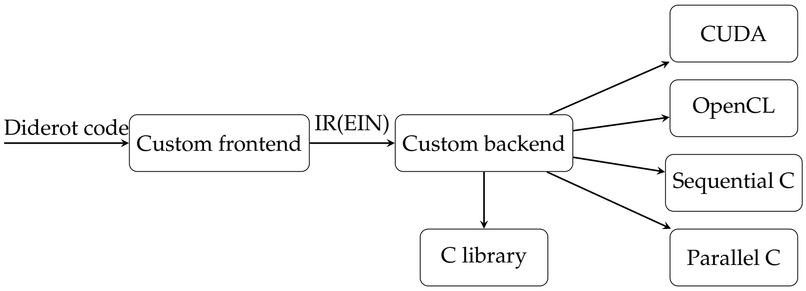


Figure 2.9: Schematic overview of Diderot. EIN is the intermediate representation, which is based on Einstein notation.

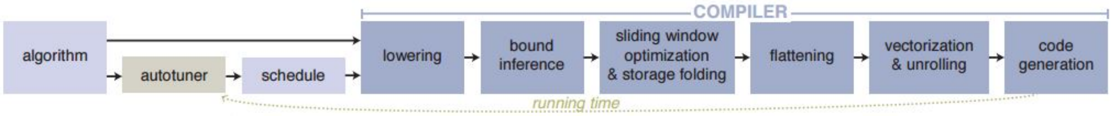


Figure 2.10: The Halide pipeline [28].

Name	Domain	Implementation	Description
Simit [3, 23]	Physical simulation	Implemented as a C++ library, using LLVM and NVVM for code generation.	Simit lets the programmer view domains as both a graph structure which is useful for local computations, and a global matrix/tensor perspective which enables the use of global linear algebra.
Ebb [3, 4]	Physical simulation	Lua-Terra	Ebb is the second generation of Liszt. Employs a 3-layer model which separates simulation code, geometric domains and runtime details.
Diderot [21]	Scientific image analysis and visualization	Standalone	Uses Unicode encoding to implement mathematical symbols in its syntax ($\nabla, \delta_{ij}, \epsilon_{ijk}, etc$).
Halide [28]	High performance image processing	Frontend embedded in C++, backend implemented with LLVM	Separates the algorithm (how) from the schedule (when) of the program. Implements a functional programming model where images are functions $f(x, y)$. Uses a stochastic search algorithm to find optimal execution schedules.
Opt [12]	Non-linear least squares optimization for graphics and imaging	Lua-Terra	A user writes energy functions defined over graphs and images, which are then compiled to highly optimized GPU solvers.
Darkroom [15]	High-level image processing code to hardware pipeline	Lua-Terra	Produces highly optimized pipelines for image processing, competitive with Halide.
OptiML [35]	Machine learning	Delite	Efficient performance portable DSL for quick prototyping of ML algorithms.
Forge [34]	Meta DSL for the Delite framework	Scala/LMS/Scala-Virtualized	Forge is not a high performance DSL, but rather a declaration language for specifying DSLs in the Delite framework.
Liszt [11]	Physical simulation	Frontend in Scala, custom backend	Liszt solves PDEs over unstructured meshes and generated code for MPI, pthreads and CUDA. Its weakness is its limitation to one domain type.
Nebo [13]	Solving PDEs	Embedded in C++	Functional/declarative programming model.
Green-Marl [18]	Graph analysis	Lightweight compiler	DSL for graph analysis.
ImageCL [14]	Portable performance image processing	ROSE compiler framework	It uses a source-to-source compiler which produces candidate implementations in OpenCL, and then picks one of them for a given device, using a machine learning approach.

Table 2.1: Table of related DSLs

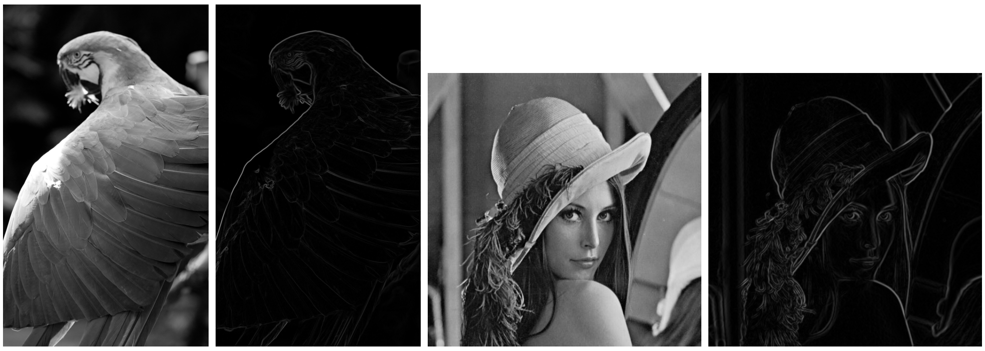


Figure 2.11: Results from the Halide algorithm.

2.4 Halide: Example Program

In this section we will have a closer look at an example program written in Halide, an embedded DSL for digital image processing.

Often when performing image processing tasks, the code will contain an abundance of boilerplate for-loops, which can become hard to read and maintain in larger applications. As we have mentioned, Halide expresses images as functions, which makes it easier to think in terms of input and output, in addition to adhering to the notation found in image processing literature. In this example we will describe a Halide program which performs a simple edge detection, and show the results. We start by describing how we set up Halide, since it turned out to not be straightforward.

2.4.1 Setting up Halide

When setting up Halide, it is important to be aware of which library versions it requires, and to know which compiler versions it supports. For this particular test case, a version of Halide which requires LLVM of version 3.9 or higher, turned out to be incompatible with Clang/LLVM version 5.0.0. The incompatibility was due to a restructuring in the Halide backend. We also experienced the binary release version of Halide support our compiler version. In particular, the binaries compiled for GCC/G++ 5.3 was not compatible GCC/G++ 5.4.

Halide also requires developer versions of the libjpeg and libpng libraries. Anaconda, which is a Python distribution which comes with a package manager might append itself into the `PATH` environment variable. Anaconda only had release versions of the libpng and libjpeg libraries, which caused Halide programs to fail when being run.

The final solution was to use version 3.9 of Clang/LLVM and to download the source code in order to build with GCC 5.4, and remove Anaconda from the PATH variable when running programs.

2.4.2 The program

The algorithm performs edge detection by first reducing noise in the image by applying a blur filter of size 5×5, and then compute the gradient magnitude of the blurred image. Since Halide is an embedded DSL in C++, the code is stored in a .cpp file and we compile it using the G++ compiler. The code can be seen in Listing 1.

Notice the DSL constructs `Func` and `Var`. `Func` is used to define steps in the algorithm pipeline, typically filters and other transformations of the images. `Var` is used to refer to the axes of the images. In the code we first read the image from file using `load_image` (line 7). We then define the pipeline by using `Funcs`: On the lines 10 and 11 we pad the image, by adding zeroes around the edges, before we extend the colour space to avoid overflows later on. In the lines 17 to 30 we define our blur kernel (filter), `k`. We define the blur operation on the lines 34 to 38, and finally we define the gradient magnitude on lines 40 to 43.

We then run the pipeline using the `realize` function (line 46). Finally we execute the algorithm and write the results to the `Buffer` object, before writing to file. We run program on two grayscale images, and the results can be seen in Figure 2.11.

2.5 Introduction to CUDA and Nvidia GPUs

CUDA is Nvidia's proprietary platform for heterogeneous computing. It was released in 2007 and has since been adopted as one of the most commonly used tools in applications that require massive parallel performance, such as scientific computing, machine learning or computational finance. GPU stands for Graphics Processing Unit and is a processor which is designed to perform computations on a *massively parallel* scale. A GPU has considerably more processing units than a traditional CPU. In this section we will go through concepts that are central to CUDA and Nvidia GPUs. For more information about any of these concepts, refer to the CUDA C Programming Guide [9].

2.5.1 Programming Model and Runtime

In the CUDA C programming model, we refer to the CPU as the *host* and the GPU with all its resources is referred to as the *device*. When we use the term *GPU*, we are referring to the computational unit (processor) itself.

The functions/programs that run on the GPU are called *kernels*. CUDA uses a *SIMT* (Single Instruction, Multiple Threads) model, meaning that when an instruction is issued, the operation is performed in parallel across several threads.

The basic execution units are called *threads*, and they all represent one run of a kernel. Threads are organized in larger structures called threadblocks, or just *blocks*, and blocks are organized into *grids*, as can be seen in Figure 2.12. When calling a kernel from the host, the programmer specifies the configuration of threads, blocks and grids. Grids and blocks can be 1, 2 or 3 dimensional. During runtime, blocks get assigned to the GPU's multiprocessors and partitioned into *warps*, which are execution units consisting of 32 threads.

```cpp
#include "Halide.h"
#include "halide_image_io.h"
using namespace Halide::Tools;
using namespace Halide;
int main(int argc, char **argv) {
    // Take a gray 8-bit input
    Buffer<uint8_t> input = load_image("images/gray.png");
    Var x("x"),y("y");
    // Pad the image by clamping
    Func padded("padded");
    padded(x,y) = input(clamp(x, 0, input.width()-2), clamp(y, 0, input.height()-2));
    // Upgrade the image to 16-bit to avoid overflow
    Func input_16("input_16");
    input_16(x, y) = cast<uint16_t>(padded(x, y));

    // Gauss kernel
    Func k("GaussianKernel");
    k(x,y) = 0;
    // First three rows
    k(-2,-2) = 1; k(-2,-1) = 4; k(-2, 0) = 7;
    k(-1,-2) = 4; k(-1,-1) = 16;k(-1, 0) = 26;
    k(0, -2) = 7; k(0, -1) = 26;k(0,  0) = 41;
    k(1, -2) = 4; k(1, -1) = 16;k(1,  0) = 26;
    k(2, -2) = 1; k(2, -1) = 4; k(2,  0) = 7;
    // Fourth and fifth row
    k(-2, 1) = 4; k(-2, 2) = 1;
    k(-1, 1) = 16;k(-1, 2) = 4;
    k(0,  1) = 26;k(0, 2) = 7;
    k(1,  1) = 16;k(1, 2) = 4;
    k(2,  1) = 4; k(2, 2) = 1;

    // Define reduction domain of size 5x5,
    // with origin in the middle
    RDom r(-2,5,-2,5);
    // Perform gaussian blur
    Func blurred("blurred");
    blurred(x,y) = sum(input_16(x+r.x,y+r.y)*k(r.x,r.y));
    blurred(x,y) /= k_divisor;
    // Find gradient magnitude
    Func grad("grad");
    grad(x, y) = cast<uint8_t>(sqrt(pow(blurred(x+1, y) - blurred(x-1, y), 2) +
                 pow(blurred(x, y+1) - blurred(x, y-1), 2)));
    // Allocate buffer for the resulting image
    Buffer<uint8_t> result(input.width(), input.height());
    // Run the algorithm
    grad.realize(result);

    // Store final image in png file
    save_image(result, "ParrotEdgeDetected.png");
    return 0;
}
```

Listing 1: A Halide program which performs edge detection by first blurring the image and then calculating the gradient magnitude of the blurred image.

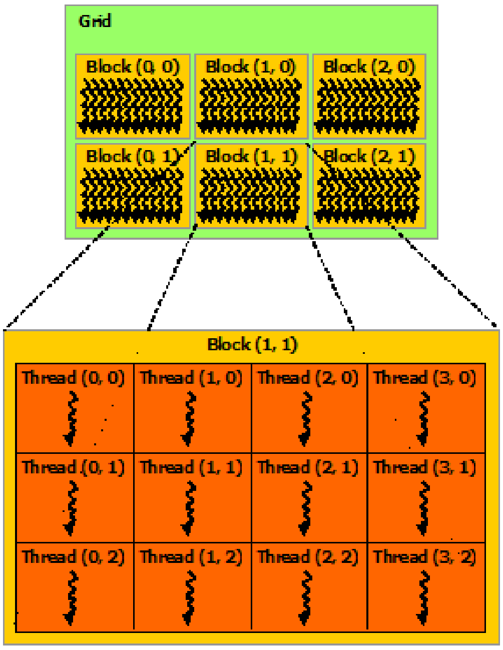


Figure 2.12: Organization of grids, blocks and threads. Source: CUDA C Programming Guide [9]

2.5.2 Synchronous vs Asynchronous Execution

Operations in CUDA can be either *synchronous* or *asynchronous* with regard to the host. A synchronous operation will stall the program until the operation is finished. If an operation is asynchronous, on the other hand, it will run independently of the host, and the program can proceed. Some operations have both synchronous and asynchronous versions, such as memory copies. The synchronous memory copy is called using `cudaMemcpy` while the asynchronous version has the `Async` suffix (`cudaMemcpyAsync`). Multiple asynchronous operations can run concurrently (at the same time), and is implemented using CUDA *streams*.

2.5.3 Hardware Architecture

On Nvidia GPUs, the main execution units are the *Streaming Multiprocessors*, commonly referred to as *SMs*. A GPU has several SMs, each of which have their own set of resources such as CUDA cores, memory (caches, registers, shared memory). SMs contribute to a portable and scalable hardware design, because performance of applications can be improved just by adding more SMs or GPUs to the system, and the behaviour of the application will be the same. Figure 2.13 shows an example of an SM design.

GPUs have a wide variety of applications that demand different types of workloads. Some applications such as scientific computing might demand higher floating point precision than computer games or machine learning. Nvidia categorizes their GPU chips into categories known as *Compute Capabilities (CC)*. Their version numbers consist of a major version and a minor version. For instance, the GTX 1060

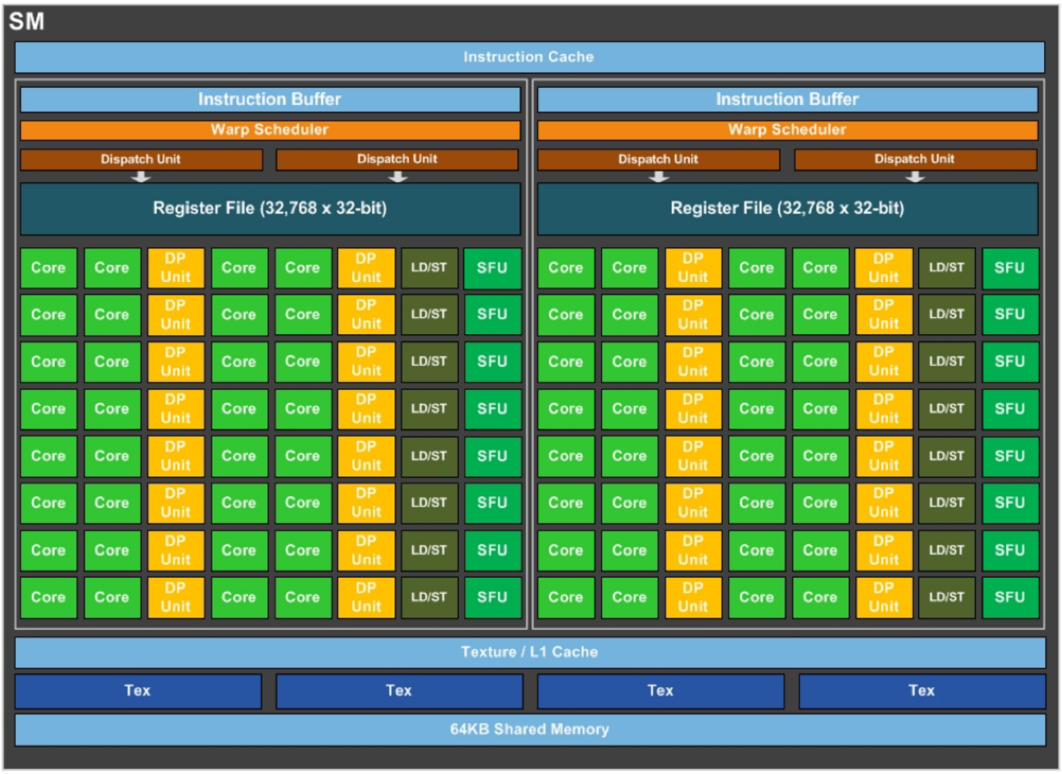


Figure 2.13: An example of a Streaming Multiprocessor design. This in particular is the GP100 SM, which is used in the most high-end GPUs of the Pascal generation, from 2016. This class of SM is especially well suited for double precision workloads, as it has 1 double precision unit for every 2 processing cores.

is of compute capability 6.1, with 6 being the major version and 1 being the minor version. The major version refers to which generation the GPU belongs to and the minor version more specifically states what types of workloads it is designed for. Having this system makes it easier for programmers to get a good insight into the features of the devices they are programming for.

2.5.4 Memory Model

A top priority in CUDA development is to optimize memory usage. The device has several types of memory, all with distinct properties regarding size, caching, read/write, scope and access latency. They are also designed for different access-patterns and using these correctly is absolutely crucial to achieve optimal performance. Figure 2.14 gives an overview of how the spaces are organized on the device. The green area marked with GPU contains the memory types that are located on the GPU chip itself, while the blue DRAM area contains the memory spaces that are located in the off-chip memory. Typically the DRAM memories have a high access latency but also a high memory size. The memory located on the GPU are usually small in size, but quick to access. We will now have a look at each memory type and describe their characteristics.

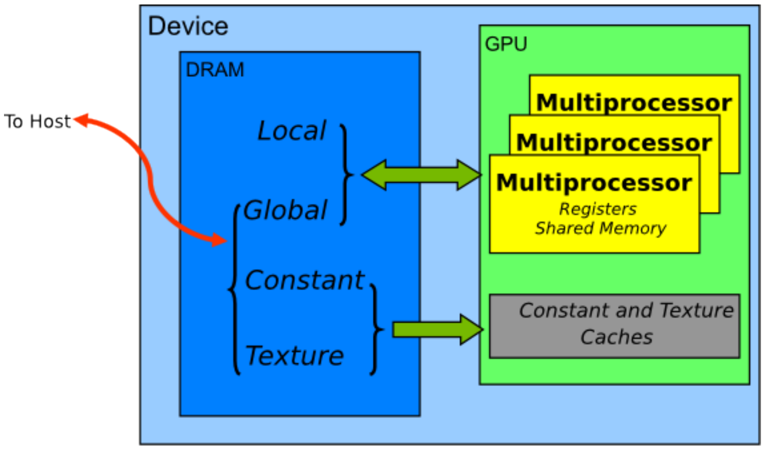


Figure 2.14: CUDA memory spaces and where they are located. Source: CUDA Best Practices Guide [8]

Global memory. Global memory is the largest, also the slowest of the memory types and can potentially consume hundreds of cycles per access. Traditionally global memory has been located in off-chip DRAM[13]. Designing kernels to access global memory according to its intended access pattern is a high priority in CUDA development, as stated by the CUDA Best Practice Guide [8]. Global memory is cached in L2 cache, so repeated access to the same memory segments will improve performance. It is common practice to place values in global memory in the initial steps of development, and then modify the algorithm later on to use other types where it is beneficial.

Local memory. A chunk of local memory is only accessible by a single thread. Local memory is located in DRAM, but also cached (in L2 on newer devices) for faster repeated access. Storage in local memory can be caused by register spilling.

Constant memory. Constant memory is read-only and located off-chip. It is cached in a dedicated constant cache. Constant cache is best utilized when all threads in a warp access the same element, and must be declared at compile-time.

Shared memory. Shared memory is located on the chip, and is shared between threads in a single block. It is very fast if used correctly, but it is not suited for all types of operations as it has strict requirements to access patterns. Shared memory is divided into *memory banks* of a fixed size (4 or 8 bytes depending on generation). If several threads in a warp tries to access the same memory bank using different addresses, a *memory bank conflict* occurs and the accesses are serialized. If all the threads in a warp accesses the same memory bank (same address), its value is broadcasted. If several threads access the same bank, its value is multicasted.

Texture memory. Texture memory is read-only and resides in DRAM, but is cached in a special purpose texture cache (shared by L1 on newer devices) that is optimized for 2D spatial locality. Texture fetches does not cost any global memory reads unless there is a cache miss.

[13] The newest high-end GPUs have replaced it with faster HBM memory (high bandwidth memory) which is on the chip.

Registers. Registers are the fastest memory type, but they are also very limited in both number and size. The scope of a register is limited to a single thread. In traditional serial programming, limiting register usage is a concern when optimizing. In parallel programming however, reusing registers might cause data dependencies and finding a trade-off between register reuse and parallelism is in order.

2.5.5 Latency Hiding and Occupancy

There are many sources of latency on the GPU and a common strategy is to hide this latency by assigning more work to the SMs. The metric we use for this is *occupancy* and is defined as the ratio between the number of warps that are active on the SM and the maximum number of warps it can process.

2.6 Introduction to Equelle

Equelle is a standalone DSL for solving partial differential equations (PDEs) numerically on unstructured grids with an emphasis on the finite volume method (FVM). In this section we will have a look at the most important concepts in Equelle, and give a few examples that demonstrate the use of these concepts. The Equelle Reference Manual [29] can be viewed for additonal information.

2.6.1 Grids in Equelle

The grids in Equelle are structures of interconnected cells, and can be defined in either 1, 2 or 3 dimensions. Cells always have the same dimension as the grid, and pairs of cells are connected by faces. Figure 2.15 shows an example grid in 2 dimensions with cells labeled with numbers and faces labeled with letters.

A *domain* is another important concept in Equelle. A domain is a set of unique grid entities (cells or faces) of the same dimensionality. Equelle provides built-in basic domains for entities that lie on the boundary of the grid, for entities inside the boundary, and for entities of the whole grid. Having access to these domains makes it a lot easier to apply common operations, such as adding boundary conditions and performing computations on the whole grid. `AllCells()`, `BoundaryCells()` and `InteriorCells()` are examples of how these can be accessed. A programmer can also specify custom domains as input to the backend when running a simulator.

2.6.2 Programming Model and Features

Equelle has a variety of features that make it both safe and convenient to use. The code example in Section 2.6.3 demonstrates some of the concepts we mention here.

Vectorization. The syntax of Equelle is highly vectorized. This makes programming in Equelle a lot easier, since the developer does not need to worry about error-prone indexing in loops. Since a for loop in itself is a highly serialized concept, keeping the syntax vectorized also helps to expose parallelism.

Type safety and *dimension consistency.* Safety is a top priority in Equelle and is achieved mainly by implementing strong type checking and consistency of *dimensions*.

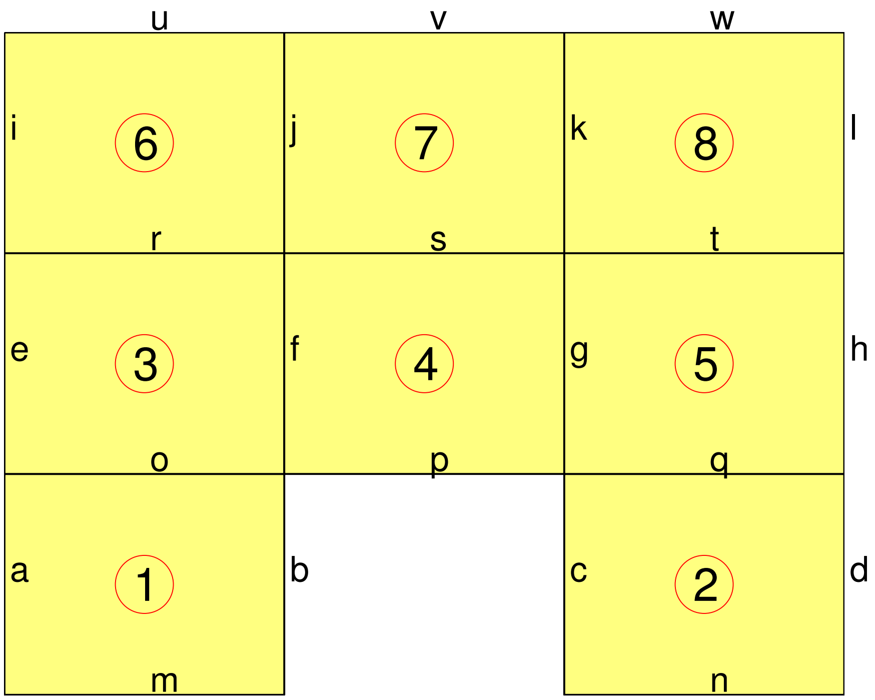


Figure 2.15: An example grid in 2D. The numbers denote the cells and the lowercase letters denote faces. The letters are placed according to where the face normals point. Source: Equelle Reference Manual

Function	**Arguments**	**Returns**
`AllCells`		`Collection Of Cell`
`InteriorCells`		`Collection Of Cell`
`BoundaryCells`		`Collection Of Cell`
`AllFaces`		`Collection Of Face`
`InteriorFaces`		`Collection Of Face`
`BoundaryFaces`		`Collection Of Face`
`FirstCell`	`Collection Of Face`	`Collection Of Cell`
`SecondCell`	`Collection Of Face`	`Collection Of Cell`
`IsEmpty`	`Collection Of Cell`	`Collection Of Bool`
`Centroid`	`Collection Of Cell`	`Collection Of Vector`
`Centroid`	`Collection Of Face`	`Collection Of Vector`
`Normal`	`Collection Of Face`	`Collection Of Vector`

Table 2.2: Built-in grid functions. Source: Equelle Reference Manual [29]

Type	Semantics
`Scalar`	A single floating-point number
`Vector`	A pair or triple of `Scalars` (depending on grid dimension)
`Bool`	A flag that is `True` or `False`
`Cell`	A cell entity of the grid
`Face`	A face entity of the grid

Table 2.3: Basic types in Equelle. Source: Equelle Reference Manual [29]

Dimensions in this context does not refer to spatial dimensions (2D, 3D etc), but rather to units of measurement (metres, feet, degrees Celsius, etc) for quantities (mass, length, temperature, etc).

Variables are by default declared as immutable, meaning that once a variable has been assigned a value, it cannot be reassigned another value. This property is commonly known as static single assignment (SSA) and does not only provide a safety guarantee, but also enables us to implement well-known compiler optimization techniques. However, sometimes one might need to reassign variables, such as in iterative methods. For this reason, Equelle also allows the use of the `Mutable` keyword to make a variable reassignable.

Explicit and implicit methods. Equelle supports both explicit and implicit methods for solving PDEs.

Automatic Differentiation

Equelle uses a technique called automatic differentiation (AD) to compute the derivative (Jacobian) matrices needed for implicit methods. A Jacobian matrix contains all the first order partial derivatives of a given function. If for instance a collection in Equelle represents a temperature field, its Jacobian matrix describes how the temperature of all the entities in the collection change with respect to each other.

$$J_{ij} = \frac{\partial f_i}{\partial x_j} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_j} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_i}{\partial x_1} & \cdots & \frac{\partial f_i}{\partial x_j} \end{bmatrix} \tag{2.1}$$

AD calculates accurate derivatives by using the chain rule on expressions. The derivative matrices can potentially become huge, but since cells in the grid often only interact with cells nearby, the majority of its elements will be zeroes (high degree of sparsity). When a matrix can be considered as *sparse*, using a sparse matrix storage scheme is usually beneficial.

There are several well-known storage schemes, such as coordinate list (COO), compressed sparse row (CSR) and compressed sparse column (CSC). No single scheme is considered to be superior to others, as they are all optimized for matrices of different characteristics. For instance, if a matrix has far more columns than rows, CSR would probably be a better choice than CSC. There are also other factors to consider when choosing a format, for instance memory access patterns. This last factor is especially relevant for GPUs. There has been developed special sparse schemes such specifically to match parallel access patterns.

Types

The basic types of Equelle are `Scalar`, `Vector`, `Bool`, `Cell` and `Face`. These types are often used to associate values with entities in the grid, commonly stored in a `Collection Of` type. In addition, there is the `Sequence Of Scalar` type which is used for timestepping in loops.

Built-in Functions and Operators

Equelle provides several built-in functions. Table 2.2 lists the built-in grid functions, which can be used to query the grid for either entities, or properties of its entities, for instance the centroids of cells or faces, or the face normals.

Two other essential functions in Equelle and for the finite volume method (FVM) are the `Gradient()` function and the `Divergence()` function. `Gradient()` takes a `CollOfScalar` as argument which is defined on `AllCells` and returns a `CollOfScalar` which is defined on `InteriorFaces`. The `Divergence()` function is used to calculate the total flow into each cell. It takes a `CollOfScalar` which is defined on `InteriorFaces`, that represents the *fluxes*. Fluxes represent the actual flow, accounting for physical properties of the grid and the problem being solved.

The `Extend` and `On` operators are used to change the domains that `CollOfScalars` are defined on.

2.6.3 Example Program: Explicit Heat Equation

In this section I will show how to solve the heat equation explicitly in Equelle. The purpose is to give a typical example of what an Equelle program looks like. I will not go into detail on the mathematics or the numerical schemes, but rather present a few fundamental equations and show how they are easily translated into code. Appendix B.1 in the original work on the backend gives a more detailed explanation of the mathematics [17]. The program in its entirety can be found in Listing 2.6.3.

The heat equation is a partial differential equation (PDE) which describes how heat spreads in a medium over time. The equation can be written on the form:

$$\frac{\partial u}{\partial t} - k\nabla^2 u = 0 \tag{2.2}$$

Where k is the conductivity of the material, ∇^2 is the Laplace operator and u is the medium.

The first thing we do in the program, is to fetch necessary values from the parameters. To read the conductivity constant k, we use the following line:

```
k : Scalar = InputScalarWithDefault("k", 0.3)
```

If the parameter file provides a value for k, then the value of the Scalar variable `k` is set to that value. If not, it will default to the value provided in the function call (0.3 in this case). Note that the type definition `: Scalar` is not strictly necessary, but can be useful for readability.

For our loop, we need a list of timestep values. These are provided as a list of values in a separate file linked to by the parameter file. We declare our timestep variable and fill it with values in the following way:

```
timesteps : Sequence Of Scalar
timesteps = InputSequenceOfScalar("timesteps")
```

Next we need the initial temperatures for u. The input to this variable is either a uniform value to be applied to all the cells in the grid, or it can be supplied as a file containing one temperature for each cell. The values are supplied by the parameter `u_initial`.

```
u0 : Collection Of Scalar On AllCells()
u0 = InputCollectionOfScalar("u_initial", AllCells())
```

In the simulation we use a Dirichlet boundary condition to apply an external heat source to u. This is done by setting a subset of the grid's boundary faces' values to appropriate temperatures. In the following code, `dirichlet_boundary` contains the face indices that we apply the values to and `dirichlet_values` contains the values we want to apply to these.

```
dirichlet_boundary : Collection Of Face Subset Of (BoundaryFaces())
dirichlet_boundary = InputDomainSubsetOf("dir_boundary", BoundaryFaces())
dirichlet_values : Collection Of Scalar On dirichlet_boundary
dirichlet_values = InputCollectionOfScalar("dir_values", dirichlet_boundary)
```

In this next snippet we calculate the transmissibility of the interior faces. The transmissibility is a measure of how much heat the face is able to transfer. It is calculated by multiplying k with the area of the face, and dividing by the distance between its first and second cell. Each face is connected to two cells, each of which can be accessed using `FirstCell(face)` and `SecondCell(face)`. We can see below that the areas of the faces are fetched by using pipes ('`|`'). The distance between the cells are found by using their centroids, which are vectors that point to their midpoints. By subtracting one from the other and taking the norm of the two vectors, we get the distance between the cells.

```
ifaces = InteriorFaces()
first = FirstCell(ifaces)
second = SecondCell(ifaces)
itrans : Collection Of Scalar On ifaces
itrans = k * |ifaces| / |Centroid(first) - Centroid(second)|
```

Next, we find the transmissibility of the boundary faces. This is mostly the same, except that the boundary faces only have a single cell that can either be the first, or the second cell. We use the ternary if to check if the first cell is empty. If it is, we pick the second cell, if it is not, we pick the first cell. Instead of using the distance between cells, we find the distance between the face and its cell.

```
bf = BoundaryFaces()
bf_cells = IsEmpty(FirstCell(bf)) ? SecondCell(bf) : FirstCell(bf)
btrans = k * |bf| / |Centroid(bf) - Centroid(bf_cells)|
```

We now define a function to calculate the flux of the interior faces. For this we use the transmissibility we calculated earlier and the gradient of u. It uses the standard form of the law of heat conduction, $\vec{q} = -k\nabla u$. The flux is the amount of heat that passes through a given face, also giving the direction of the flow through its sign.

```
computeInteriorFlux : Function(u : Collection Of Scalar On AllCells()) ...
                       -> Collection Of Scalar On InteriorFaces()
computeInteriorFlux(u) = {
    -> -itrans * Gradient(u)
}
```

Before we can compute the flux of the boundary faces, we must find the correct signs. We find this using the ternary if, in the same way as choosing the correct cells for the faces:

```
bf_sign = IsEmpty(FirstCell(bf)) ? (-1 Extend bf) : (1 Extend bf)
dir_sign = bf_sign On dirichlet_boundary
```

Now we define a function for computing the flux of the boundary faces. The computation is done by multiplying the transmissibility by the signs we just found, and then multiplying by the difference between the boundary cell values and the heat source values. This is the same operation as the one we performed on the interior cells. The difference in the values of the boundary cells and the heat sources, is the same as the gradient along the boundary.

```
computeBoundaryFlux : Function(u : Collection Of Scalar On AllCells()) ...
                       -> Collection Of Scalar On BoundaryFaces()
computeBoundaryFlux(u) = {
    # Compute flux at Dirichlet boundaries.
    u_dirbdycells = u On (bf_cells On dirichlet_boundary)
    dir_fluxes = (btrans On dirichlet_boundary) * dir_sign ...
                 * (u_dirbdycells - dirichlet_values)
    -> dir_fluxes Extend BoundaryFaces()
}
```

The last part of the program is the loop that uses everything we have already defined to solve the heat equation. The loop iterates over the timesteps and solves once per entry. The code starts by defining a `Mutable Collection Of Scalar`. We need it to be mutable because it will be reassigned in every iteration. Next we use the functions we defined to compute the fluxes. By extending them both to `AllFaces()` and adding them together, we place all the fluxes in the same domain. In the solution step, we divide the timestep `dt` by the volumes of all cells in the grid. This ensures that the heat spreads at the correct rate no matter what the cell size is. We then take the divergence of the fluxes, effectively performing $\nabla \cdot (k\nabla u)$.

```
u : Mutable Collection Of Scalar On AllCells()
u = u0
For dt In timesteps
{
    # Compute fluxes.
    ifluxes = computeInteriorFlux(u)
    bfluxes = computeBoundaryFlux(u)
    fluxes = (ifluxes Extend AllFaces()) + (bfluxes Extend AllFaces())

    # Solve for current timestep.
    u = u - (dt / |AllCells()|) * Divergence(fluxes)
}
```

This concludes our example program, and we will now proceed to the next section, where we describe the CUDA backend.

```
# Heat diffusion constant.
k : Scalar = InputScalarWithDefault("k", 0.3)
# Input timesteps.
timesteps : Sequence Of Scalar
timesteps = InputSequenceOfScalar("timesteps")
# Input initial temperatures.
u0 : Collection Of Scalar On AllCells()
u0 = InputCollectionOfScalar("u_initial", AllCells())
# Input boundary condition domain, a subset of BoundaryFaces, and boundary heat values.
dirichlet_boundary : Collection Of Face Subset Of (BoundaryFaces())
dirichlet_boundary = InputDomainSubsetOf("dir_boundary", BoundaryFaces())
dirichlet_values : Collection Of Scalar On dirichlet_boundary
dirichlet_values = InputCollectionOfScalar("dir_values", dirichlet_boundary)
# Compute interior transmissibilities.
ifaces = InteriorFaces()
first = FirstCell(ifaces)
second = SecondCell(ifaces)
itrans : Collection Of Scalar On ifaces
itrans = k * |ifaces| / |Centroid(first) - Centroid(second)|
# Compute boundary transmissibilities.
bf = BoundaryFaces()
bf_cells = IsEmpty(FirstCell(bf)) ? SecondCell(bf) : FirstCell(bf)
btrans = k * |bf| / |Centroid(bf) - Centroid(bf_cells)|

# Function for computing flux for interior faces.
computeInteriorFlux : Function(u : Collection Of Scalar On AllCells()) ...
                      -> Collection Of Scalar On InteriorFaces()
computeInteriorFlux(u) = {
    -> -itrans * Gradient(u)
}

# Compute sign to get correct gradient.
bf_sign = IsEmpty(FirstCell(bf)) ? (-1 Extend bf) : (1 Extend bf)
dir_sign = bf_sign On dirichlet_boundary

# Function for computing flux for boundary faces.
computeBoundaryFlux : Function(u : Collection Of Scalar On AllCells()) ...
                      -> Collection Of Scalar On BoundaryFaces()
computeBoundaryFlux(u) = {
    # Compute flux at Dirichlet boundaries.
    u_dirbdycells = u On (bf_cells On dirichlet_boundary)
    dir_fluxes = (btrans On dirichlet_boundary) * dir_sign * ...
                 (u_dirbdycells - dirichlet_values)
    -> dir_fluxes Extend BoundaryFaces()
}

# Need mutable variable for the loop.
u : Mutable Collection Of Scalar On AllCells()
u = u0
For dt In timesteps
{
    # Compute fluxes.
    ifluxes = computeInteriorFlux(u)
    bfluxes = computeBoundaryFlux(u)
    fluxes = (ifluxes Extend AllFaces()) + (bfluxes Extend AllFaces())

    # Solve for current timestep.
    u = u - (dt / |AllCells()|) * Divergence(fluxes)
}
```

Listing 2: An Equelle program which solves the heat equation explicitly.

2.6.4 CUDA Backend Design

The CUDA backend is based on the CPU backend, which is implemented in C++ and uses operator overloading extensively. By using operator overloading, it gets simpler to implement mathematical expressions since they can be written in a form that is closer to its mathematical notation. We will see examples of this in the sections below. Table 2.5 shows a complete list of all the implemented arithmetic operators. For a more complete explanation of the backend, please refer to the previous work [17].

CollOfScalar

The `CollOfScalar` class represents a `Collection Of Scalar` in the Equelle language. Its values are stored in a `CudaArray` object. The collection's derivatives are stored in a `CudaMatrix` object. The matrix is only used if the Equelle program uses the implicit method. The arithmetic operators of `CollOfScalar` uses automatic differentiation when calculating its derivatives. Let's say we want to calculate $u * v + w$ where u, v and w are collections. By applying the chain rule we get $uv' + u'v + w'$ for the derivative, and finding the new collection's value happens in a elementwise manner. By using the operator overloading approach, this operation is straight-forward to implement. The code in Listing 3 demonstrates how operator overloading simplifies implementations.

```cpp
CollOfScalar u;
CollOfScalar v;
CollOfScalar w;

/* Set the values of u, v and w and compute their derivatives. */

CollOfScalar result = u*v+w;

// This is what happens implicitly
CudaArray new_values = u.val_*v.val_ + w.val_; // u * v + w
CudaMatrix new_der = u.val_*v.der_ + u.der_*v.val_ + w.der_; // uv' + u'v + w'
CollOfScalar result(new_values, new_der);
```

Listing 3: C++ code which demonstrates how to use `CollOfScalar` to implement mathematical expressions.

CudaArray

`CudaArray` is essentially a vector, with operator overloading for element-wise arithmetic and comparison. When such an operator is performed, it performs the operations using CUDA kernels. The class has a pointer to its values which are located on the device. Listing 4 shows how the class can be used in code. When an object is initialized, it allocates memory and when the object's destructor, it frees the memory.

```
// Create two CudaArrays containing 1000 elements.
int N = 1000;
CudaArray array1(N);
CudaArray array2(N);

/* Set the values of the arrays to arbitrary numbers */

// Element-wise addition
CudaArray array_arith_result = array1 + array2;

// Element-wise check for equality
CollOfBool array_comp_result = array1 == array2;

// Element-wise check on whether array1's elements are greater than array2's
CollOfBool array_comp_result = array1 > array2;
```

Listing 4: C++ code showing how to use `CudaArray`. Initialization of the `CudaArray` values is omitted.

CudaMatrix

`CudaMatrix` represents a matrix in sparse storage format. It uses the *compressed sparse row* (CSR) storage format. CSR is a widely used format and is supported by both CUSP and cuSPARSE. A matrix in CSR format consists of 3 data arrays: A row pointer, column indices and the nonzero matrix values. The row pointer array contains the cummulative number of nonzero values for each row, and always has a leading zero element. The column index array stores the indices of the columns that the nonzero values are located in, in row-major order. The nonzero values are stored in the same order. Figure 2.16 shows an example of a dense matrix and its corresponding CSR representation.

`CudaMatrix` uses cuSPARSE for most of its arithmetic operators, such as multiplication, addition and subtraction. For sparse matrix multiplication, it uses the `csrgemm` procedure. For matrix-vector multiplication, it uses the `csrmv` procedure. Finally, for sparse matrix addition and subtraction, it uses `csrgeam`.

CollOfVector

`CollOfVector` is another data class which represents a collection of vectors, for instance centroids or normals. Like the other data classes, it overloads basic arithmetic operators in addition to the indexing operator ('[]'). Its data is stored using a `CudaArray` object, which is useful since many of the operations for `CollOfVector` are performed element-wise. `vec_collection[1]` returns the second component of all the vectors in `vec_collection` as a `CollOfScalar` rather then the second vector in the collection. It also has functions for dot products and norms.

DeviceGrid

The `DeviceGrid` class is responsible for constructing and managing the Equelle grid, which is stored on the device. Grids are constructed on the CPU using the OPM

$$\begin{pmatrix}1 & 0 & 3 & 0\\ 0 & 0 & 4 & 5\\ 2 & 0 & 0 & 0\\ 0 & 6 & 0 & 7\end{pmatrix}$$

(a)

rowPtr	0	2	3	5	7		
colInd	0	2	2	3	0	1	3
values	1	3	4	5	2	6	7

(b)

Figure 2.16: A dense matrix (a) and its corresponding sparse representation in compressed sparse row (CSR) format. The length of a row pointer array in a CSR structure is always one more than the number of rows in the matrix it represents, and always starts with a 0. The column index array and the value array each contain a number of elements equal to the number of nonzero values in the matrix.

library [19], and then transferred to the device. The grid remains on the device until the Equelle program is finished. Table 2.4 lists the data arrays of the grid. `DeviceGrid` also implements functions for creating and accessing the built-in domains.

LinearSolver

The `LinearSolver` class implements iterative solvers and preconditioners using the CUSP library. It currently supports the following solvers: CG, BiCGStab and GMRes. CG and BiCGStab can both be used with a diagonal (or Jacobi) preconditioner and GMRes does not support any preconditioners.

EquelleRuntimeCUDA

`EquelleRuntimeCUDA` is the main class in an Equelle CUDA program. It works as a wrapper for the other classes and keeps track of the runtime state, contains the grid, handles file input (e.g. parameters) and output and printing to terminal.

2.7 Tools and Best Practices

Nvidia provides a set of tools for measuring performace and debugging CUDA programs. They also provide a best practices guide that makes recommendations for how to perform optimization work and for what to focus on when programming for their GPUs. In this section, we will have a look at some of these best practices as well as the tools that have been used in this thesis.

2.7.1 Best Practices

Many fields have certain established guidelines that are widely accepted as effective. We will now go through some of the best practices that are important to both CUDA development and performance evaluation.

Variable	Size (Elements)	Data Type (Bytes)	Description
`cell_centroids_`	One vector per cell.	`double` (8)	Vectors pointing to the centers of the cells. Each vector contains 1 double for each cell.
`face_centroids_`	One vector per face.	`double` (8)	Vectors pointing to the centers of the faces. Each vector contains 1 double for each face.
`face_normals_`	One vector per face.	`double` (8)	Face normals describing their direction.
`cell_volumes_`	One per cell.	`double` (8)	Volumes of cells.
`face_areas_`	One per face.	`double` (8)	Areas of faces.
`cell_facepos_`	One per cell + 1	`int` (4)	Starting indices for each cell's faces in `cell_faces_`.
`cell_faces_`	`size_cell_faces`	`int` (4)	Stores the relations between cells and their faces.
`face_cells_`	2 per cell.	`int` (4)	Stores 2 cell indices per face, pointing to the faces' *first* and *second* cells.

Table 2.4: The table contains information about the data arrays that make up the grid. They are constructed on the CPU using the OPM library and then copied as-is to the device memory. Note that vectors (centroids and normals) have one element for each dimension of the grid.

CUDA Development

During development we follow the *CUDA Best Practices Guide* [8] for guidelines on how to develop and optimize CUDA programs.

Figure 2.17 illustrates the APOD process which is described in the guide. APOD stands for Assess, Parallelize, Optimize and Deploy, and is an iterative process where we first assess our program to get an overview of its bottlenecks and find possible solutions to them. If the bottlenecks contain any serial code, we parallelize is, then we optimize the code, and finally we deploy our solution to reap the benefits as early as possible. The steps are then repeated.

We can use several approaches when assessing our application, but at the center in CUDA development is the use of *profilers* to measure the actual performance of programs. Profilers are specialized programs made specifically for gathering and analyzing runtime data.

Returns	**Lhs**	**Operator**	**Rhs**
`CudaArray`	`CudaArray`	`+ - * /`	`CudaArray`
`CudaArray`	`Scalar`	`* /`	`CudaArray`
`CudaArray`	`CudaArray`	`* /`	`Scalar`
`CudaMatrix`	`CudaMatrix`	`+ - *`	`CudaMatrix`
`CudaMatrix`	`CudaMatrix`	`*`	`CudaArray`
`CudaMatrix`	`Scalar`	`*`	`CudaMatrix`
`CudaMatrix`	`CudaMatrix`	`*`	`Scalar`
`CudaMatrix`	`N/A`	`-`	`CudaMatrix`
`CollOfScalar`	`CollOfScalar`	`+ - * /`	`CollOfScalar`
`CollOfScalar`	`CollOfScalar`	`* /`	`Scalar`
`CollOfScalar`	`Scalar`	`* /`	`CollOfScalar`
`CollOfScalar`	`N/A`	`-`	`CollOfScalar`
`CollOfVector`	`N/A`	`-`	`CollOfVector`
`CollOfVector`	`CollOfVector`	`+ -`	`CollOfVector`
`CollOfVector`	`Scalar`	`*`	`CollOfVector`
`CollOfVector`	`CollOfVector`	`* /`	`Scalar`
`CollOfVector`	`CollOfVector`	`* /`	`CollOfScalar`
`CollOfVector`	`CollOfScalar`	`*`	`CollOfVector`

Table 2.5: Overloaded arithmetic operators in the CUDA backend.

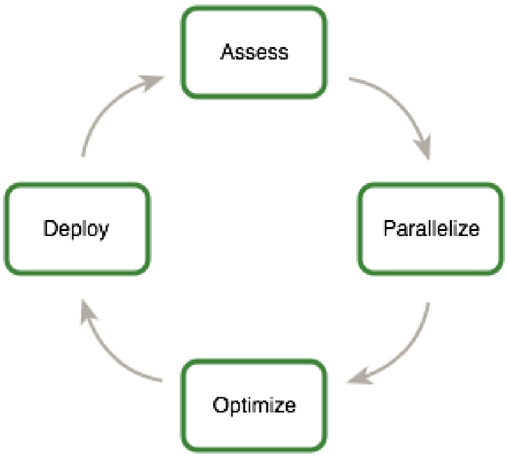


Figure 2.17: A diagram depicting the APOD development process described in the *CUDA Best Practice Guide* [8]. APOD is an iterative process where we first assess our program to get an overview of its hotspots and find possible solutions to them. We parallelize serial code where possible, we then optimize and lastly we deploy our changes. We then repeat the steps.

Performance Evaluation

Performance evaluation of computer systems is a difficult task and has been a well-established research field for decades.

One might say that "the only requirement for validation is that the results should not be counterintuitive." [20]. This means that when we measure the performance of a system, with the correct assumptions, we should to a certain degree be able to estimate the impact when changing the factors of the system. In our performance analyses in this thesis, we will follow this principle for validation. If we apply this correctly, it will allow us to generalize our results and extract more information from our measurements. Validating results could include the use of analytical modelling to estimate performance, and then measure the system to see if the assumptions are correct. If the measurements do not correlate with the modelling, then the results should be investigated in more detail to find the cause of the unintuitive results.

2.7.2 Tools

In this section we will go through the tools that have been used for profiling and debugging in this thesis. These are the profilers provided by Nvidia as well as their debugger, and the GPU monitoring application nvidia-smi.

Profilers and Debuggers

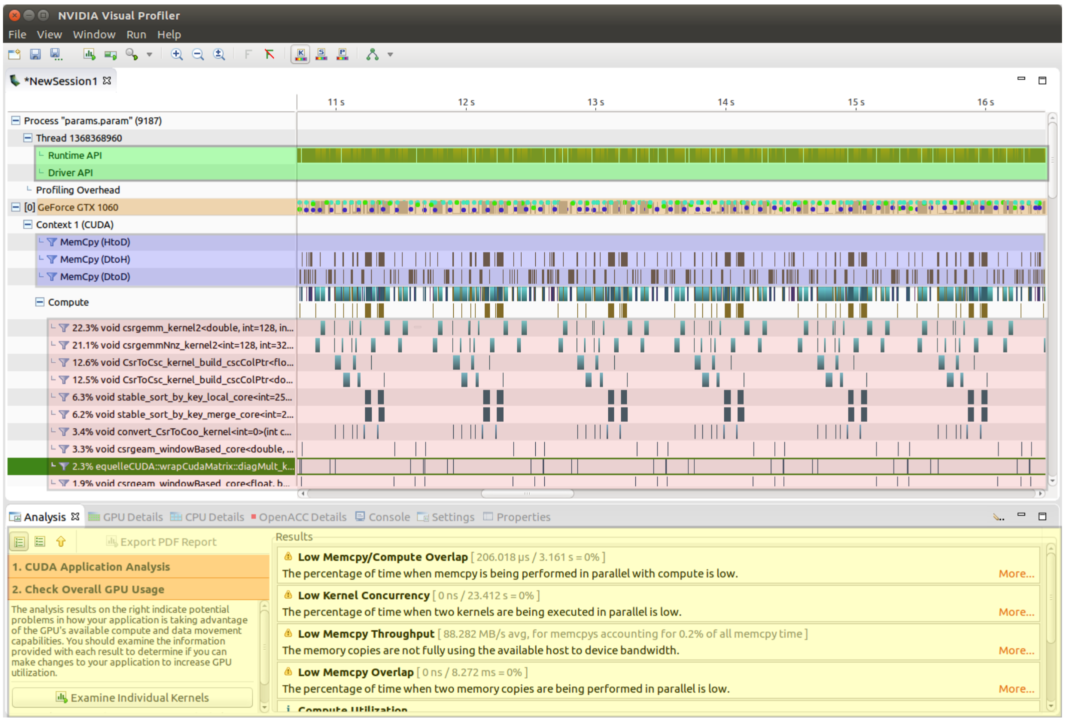
For profiling and debugging we are using Nvidia's supplied toolkit of debugger and profilers. For debugging we are using CUDA-GDB, which is based on the GNU Project Debugger, gdb[14]. and provides the same user interface as well as additional functionality for CUDA debugging, such as kernel debugging. We are using two different but interconnected profilers: the command-line profiler *nvprof* and the *Nvidia Visual Profiler*. nvprof lets us collect and view profiling data from the command-line. By giving appropriate command-line options we can customize the data from nvprof to include dependency analyses, CPU-traces, metrics such as floating-point operations per second and several others. The output can be specified as well, for instance redirecting it to a file that can be read by the Visual Profiler.

When CPU profiling is enabled, nvprof will sample the CPU's program counter (PC) at a given frequency (100hz as default), ultimately generating a trace which shows where in the code the CPU spends most of its time. The trace can be viewed from 3 different perspectives: *top-down*, *bottom-up* and *flat*. When displayed top-down, it will show the call hierarchy starting at the first call (usually the main function) and continues deeper down, showing the time portion of each sub-call.

The Nvidia Visual Profiler (nvvp) is an advanced profiler which visualizes CUDA programs as timelines and provides an easy-to-use GUI. The timeline view is very useful for inspecting programs in detail, with the possibility to do in-depth analysis of both single kernels and whole-program analyses. It also has a guided analysis mode that guides the developer to possible bottlenecks, giving advice on where to optimize.

In addition to their own profilers, Nvidia provides APIs that both lets us alter the output of the profilers, as well as making our own. By using the Nvidia Tools

[14] https://www.gnu.org/software/gdb/


NVIDIA Visual Profiler
File View Window Run Help
*NewSession1
11 s
12 s
13 s
14 s
15 s
16 s
Process "params.param" (9187)
Thread 1368368960
Runtime API
Driver API
Profiling Overhead
[0] GeForce GTX 1060
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
MemCpy (DtoD)
Compute
22.3% void csrgemm_kernel2<double, int=128, in...
21.1% void csrgemmNnz_kernel2<int=128, int=32...
12.6% void CsrToCsc_kernel_build_cscColPtr<flo...
12.5% void CsrToCsc_kernel_build_cscColPtr<do...
6.3% void stable_sort_by_key_local_core<int=25...
6.2% void stable_sort_by_key_merge_core<int=2...
3.4% void convert_CsrToCoo_kernel<int=0>(int c...
3.3% void csrgeam_windowBased_core<double, ...
2.3% equelleCUDA::wrapCudaMatrix::diagMult_k...
1.9% void csrgeam_windowBased_core<float, b...
Analysis
GPU Details
CPU Details
OpenACC Details
Console
Settings
Properties
Export PDF Report
1. CUDA Application Analysis
2. Check Overall GPU Usage
Examine Individual Kernels
Results
More...
Compute Utilization

The analysis results on the right indicate potential problems in how your application is taking advantage of the GPU's available compute and data movement capabilities. You should examine the information provided with each result to determine if you can make changes to your application to increase GPU utilization.

Low Memcpy/Compute Overlap [206.018 µs / 3.161 s = 0%]
The percentage of time when memcpy is being performed in parallel with compute is low.

Low Kernel Concurrency [0 ns / 23.412 s = 0%]
The percentage of time when two kernels are being executed in parallel is low.

Low Memcpy Throughput [88.282 MB/s avg, for memcpys accounting for 0.2% of all memcpy time]
The memory copies are not fully using the available host to device bandwidth.

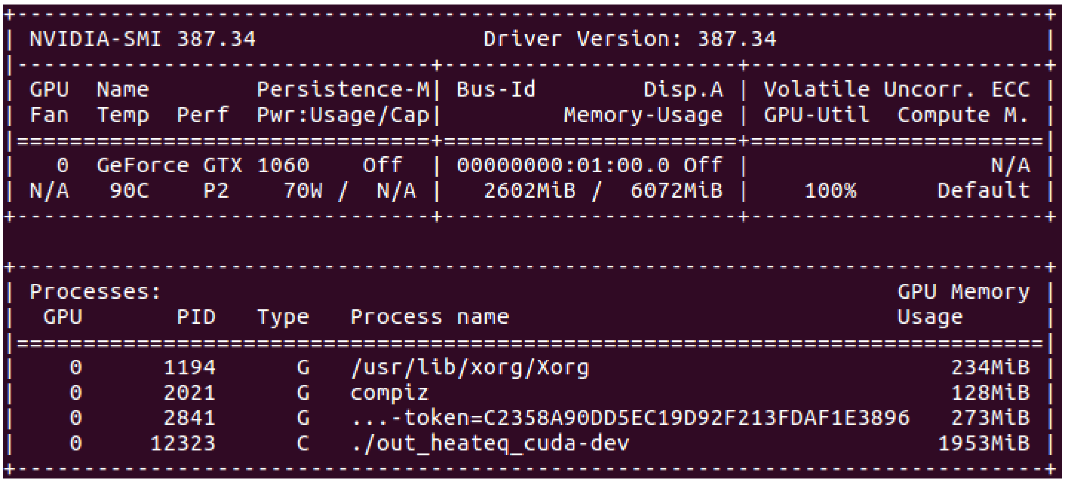
Low Memcpy Overlap [0 ns / 8.272 ms = 0%]
The percentage of time when two memory copies are being performed in parallel is low.

Figure 2.18: This is the default view of the Nvidia Visual Profiler. It is in guided mode and the GPU analysis has been run. The green area is the timeline for CUDA and driver API activity, the orange area shows GPU activity, the blue area shows the memory transfers with one line for each type (host-to-device, device-to-host and device-to-device), the red area shows the timeline for the kernel calls with one line for each kernel. Finally, the yellow currently shows the Analysis view with information from the GPU analysis. It can also change to views for Console, GPU details, and CPU details.

Extension (NVTX) API, we can annotate the Nvidia Visual Profiler's timeline by tagging ranges of the program. This makes interpretation of the timelines much easier. If we run the annotated code with nvprof, we also get summaries for each range.

Nvidia-smi

Nvidia System Management Interface (nvidia-smi) is a command-line utility for monitoring Nvidia GPUs. It can be used to gather metrics such as memory usage, compute utilization, temperature, clock speeds of both memory and multiprocessors and power consumption. We can also use the tool to change the behaviour of our device by changing variables. Figure 2.19 shows the output after running nvidia-smi in its default mode. At the top we see the driver version (387.34) and in the table below we can read information such as the GPU model (GTX 1060), current temperature (90C), power consumption (70W) and total memory usage in mebibytes (MiB). The table at the bottom shows all the processes that are running on the GPU, including both graphics and compute applications. Graphics processes are labelled G and compute processes are labelled C. In this example, we are running a version of



```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 387.34                 Driver Version: 387.34                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce GTX 1060    Off  | 00000000:01:00.0 Off |                  N/A |
| N/A   90C    P2    70W /  N/A |   2602MiB /  6072MiB |    100%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    0      1194      G   /usr/lib/xorg/Xorg                           234MiB |
|    0      2021      G   compiz                                       128MiB |
|    0      2841      G   ...-token=C2358A90DD5EC19D92F213FDAF1E3896   273MiB |
|    0     12323      C   ./out_heateq_cuda-dev                       1953MiB |
+-----------------------------------------------------------------------------+
```

Figure 2.19: This is the console output of nvidia-smi while running the implicit heat equation simulation. From this screen we can see that the process takes up 1952 MiB of memory. We also see that the temperature of the GPU is 90 degrees Celsius.

the implicit heat equation, which uses 1953 MiB of memory.

With nvidia-smi we can generate timeseries of metrics for compute applications by using the `query-compute-apps` command line option in a loop. The loop interval is specified using the `lms` command line option, and specified in milliseconds and the output is in CSV format. This functionality is very useful for benchmarking memory usage, as the profilers do not have this kind of functionality at the time of writing. In this thesis, we are using the command line in Listing 5 to gather timeseries of memory usage. Since we are running on Ubuntu, we can use the '>' symbol to redirect the output to the file `memory_usage.csv`.

```
nvidia-smi --query-compute-apps=used_memory --format=csv,noheader,nounits
           -lms 1 > memory_usage.csv
```

Listing 5: Command line for writing GPU memory usage to a csv file.

The output is a stream of integers separated by line feeds, which can easily be copy-pasted into spreadsheets or read by programs, and be analyzed. The numbers are in mebibytes (MiB), which is 1024^2 bytes, as opposed to a megabytes (MB) which are 1000^2 bytes.

2.7.3 CUDA Libraries

This section contains brief descriptions of libraries that are used in the CUDA backend. Using these libraries lets us parallelize code with a much lower effort than if we developed every algorithm from scratch. The libraries listed that come with the CUDA Toolkit are actively developed and maintained.

Thrust is a useful library for parallel algorithms which is included in the CUDA toolkit. It is based on the C++ Standard Template Library (STL), and includes similar functionality such as efficient implementations of common algorithms (sorting,

GPU	FP64 Throughput (GigaFLOP/s)	Memory Bandwidth (GB/s)	Memory Amount	CUDA Cores Total (FP64)	Core Clock
GTX 1060	133	192	6 GB	1280 (40)	1670Mhz
NVS 5200M	10	14.4	1 GB	96 (8)	625Mhz
Tesla K40	1430	288	12 GB	2880 (960)	706Mhz

Table 2.6: List of GPUs used with Equelle. In the CUDA cores column, the numbers in parentheses denote the number of cores that are capable of performing double precision operations. We are using the GTX 1060 which is a GPU designed for single precision.

reductions, prefix sums), general and special purpose iterators, data structures such as vectors and maps that can reside on both the host and the device, and more. Thrust also supports several backends for its algorithms, such as OpenMP, device (GPU) and host (CPU).

cuSPARSE is a CUDA library that comes with the CUDA toolkit, which provides a set of subroutines for sparse matrices. Examples of functionality it provides are: basic components for building solvers and preconditioners, subroutines for transpositions and other modifications sparse structures, as as well known arithmetic operators such as generalized multiplication (`gemm`). cuSPARSE supports a variety of storage schemes for sparse matrices such as compressed sparse row (CSR), compressed sparse column (CSC) and a hybrid format (HYB), as well as subroutines for conversion between these and editing of them.

CUSP "is an open source C++ library of generic parallel algorithms for sparse linear algebra and graph computations on CUDA architecture GPUs" [10]. It is not included in the CUDA Toolkit, but is available on GitHub as an open source project. CUSP uses common sparse storage formats such as CSR and is therefore compatible with cuSPARSE. The last major version of CUSP was released in 2015. Most of the recent updates to CUSP are bugfixes and adding compatibility for newer GPUs and CUDA versions, as can be seen in the CUSP commit history on GitHub [1].

2.7.4 Hardware Setup

The GPU we are using is a Geforce GTX 1060 for laptops, with 6 GBs of on-board DRAM memory, with a bandwidth of 192 GB/s. The GPU has 10 Streaming Multiprocessors (SMs) with 128 CUDA cores each. It is a GPU in the upper middle-range meant for computer games and is therefore designed for single precision computations. The peak performance is listed in the Nvidia Visual Profiler as 4.276 teraFLOP/s for single precision, and 133 gigaFLOP/s for double precision. The reason for this discrepancy is that only 4 of the 128 cores on every SM is capable of performing double precision computations, which means that we only have 40 cores for these computations. Since Equelle mainly works with double precision floating point numbers, this greatly decreases the potential for performance gains compared to other GPUs tailored for double precision.

In the initial work on the CUDA backend, two GPUs were used for testing: The NVS 5200M laptop GPU and the Tesla K40 compute GPU. Like the GTX 1060, the NVS is designed with single precision in mind, and only 8 out of its total 96 cores are double precision-capable, and has a throughput of 10 gigaFLOP/s. Tesla K40 on the other hand is a pure compute GPU with 2880 cores where 1/3 of them are double precision capable, with a core frequency of 706 Mhz. The peak double precision performance is 1.43 teraFLOP/s, roughly 11 times more than what the GTX 1060 is capable of.

Chapter 3

Performance Analysis

It is important to know what the bottlenecks of a program are when deciding on optimizations. In this chapter we perform an analysis of the Equelle CUDA backend, mainly by using the implicit heat equation simulator. We will first give an explanation of the program so that we can better understand the computations that are involved. The knowledge we get from this study will help us to understand our profiling results and to perform further analysis. After we have studied the program, we will use profilers to measure the performance of the backend and to find out what its underlying problems are. We will then use our results to decide on optimizations in the next chapter.

3.1 Program Overview

In this section we will have a look at the overall structure of the implicit heat equation program with a focus on the main computation phase. The complete Equelle code can be found in Section A.1.

Listing 6 contains pseudocode that demonstrates a typical case of solving an equation implicitly, using the Newton-Raphson method. For each timestep, a solution to u is found by iteratively minimizing the *residual* of u. The residual and its associated Jacobian is calculated using a *residual function* and it is considered to be *small enough* when its norm is below a chosen tolerance value.

The residual computation for the heat equation is the following:

```
residual = u - u0 + (dt / (cv * vol)) * Divergence(fluxes)
```

where u is the collection of values that we update after each iteration. At the beginning of each iteration, u has an identity matrix as its derivative. u0 is the starting point of the current timestep and it has no derivative. It will get updated in every timestep, but it stays constant across Newton iterations. dt is the timestep length, and its value is specified as a parameter to the program. The step length *might* vary for each timestep, but it is usually uniform. cv is a scalar value which represents the specific heat capacity of the cells. vol is the volumes of the grid cells, which has been precomputed in the setup phase. Divergence(fluxes) is the divergence of the heat flow on every cell on the grid. fluxes is the heat flow of the grid and is defined

```
for $n$ in $timesteps$ do
    $residual, jacobian \leftarrow computeResidual(u)$;
    $i \leftarrow 0$;
    while $twoNorm(residual) < tolerance$ and $i < maxiterations$ do
        $du \leftarrow linearSolve(residual, jacobian)$;
        $u \leftarrow u - du$;
        $residual, jacobian \leftarrow computeResidual(u)$;
        $++i$;
    end while
    $u_n \leftarrow u$
end for
```

Listing 6: Pseudocode for solving an equation implicitly using Newton's method. `u` contains the solution vector and is iteratively updated using Newton iterations.

on `AllFaces`. The final result of the computation which is stored in `residual`, is a `Collection Of Scalar` which is defined on `AllCells`, with a derivative matrix of size `AllCells`×`AllCells`.

Listing 7 shows the timestepping in Equelle. The built-in function `NewtonSolve` is called with the residual function and the initial guess `u_guess` as arguments. The Equelle backend then computes the residual and updates the solution, as described in Listing 6.

Listing 8 defines the residual function in Equelle. The fluxes are calculated on lines 2 to 4. `computeInteriorFlux` and `computeBoundaryFlux` are called to compute the interior and boundary fluxes before adding them together (line 4). Notice that since `ifluxes` is defined on `InteriorFaces` and `bfluxes` is defined on `BoundaryFaces`, they both need to be extended to `AllFaces` before being combined.

Listing 9 defines `computeInteriorFlux`, which computes the interior flux by multiplying the negative interior transmissibilities (`-itrans`) by the gradient of `u`. When the gradient of `u` is taken, its derivative is computed by multiplying a fixed sparse matrix depending on the grid. The transmissibilities also stay the same since they are computed in the setup phase. Thus, `computeInteriorFlux` will yield the same derivative every time for a given grid. The values of the resulting collection is defined on `InteriorFaces` and contains the heat that passes through each of the faces.

Similarly, `computeBoundaryFlux` (Listing 10) computes the fluxes along the Dirichlet boundary. The values of `u` in the Dirichlet boundary cells in `u` are first stored in `u_dirbdycells` by using the `On` operator. The flux of the Dirichlet boundary is then found by multiplying the transmissibilities, face direction (sign) and the change in temperature. The derivative matrix of the boundary flux will remain the same throughout the simulation as well. The fluxes are extended to the `BoundaryFaces` domain before being returned.

```
For dt In timesteps {
    computeResidualLocal(u) = {
        -> computeResidual(u, u0, dt)
    }
    u_guess = u0
    u = NewtonSolve(computeResidualLocal, u_guess)
    u0 = u
}
```

Listing 7: Equelle code which defines the timestepping loop for the implicit heat equation. `NewtonSolve` is called with the residual function and an initial guess as its arguments.

```
1  computeResidual(u, u0, dt) = {
2      ifluxes = computeInteriorFlux(u)
3      bfluxes = computeBoundaryFlux(u)
4      fluxes = (ifluxes Extend AllFaces()) + (bfluxes Extend AllFaces())
5      residual = u - u0 + (dt / (cv * vol)) * Divergence(fluxes)
6      -> residual
7  }
```

Listing 8: Equelle code which defines the `computeResidual` function. The interior fluxes (`ifluxes`) and boundary fluxes (`bfluxes`) are first computed, and then added together (line 4). The residual function is then evaluated.

```
computeInteriorFlux(u) = {
    -> -itrans * Gradient(u)
}
```

Listing 9: Equelle code which defines the `computeInteriorFlux` function. The flux of the interior faces are calculated by multiplying the negative transmissibilities by the gradient of u. The result is defined on `InteriorFaces`.

```
computeBoundaryFlux(u) = {
    u_dirbdycells = u On dir_cells
    dir_fluxes = dir_trans * dir_sign * (u_dirbdycells - dir_val)
    -> dir_fluxes Extend BoundaryFaces()
}
```

Listing 10: Equelle code which defines the `computeBoundaryFlux` function. First we define a new variable containing the values of the cells along the Dirichlet boundary. As we can see from the extend operator in the return statement, the result is defined on `BoundaryFaces`.

3.2 Program Analysis

In this section we perform a manual analysis of the program by using simple established program analysis concepts that are implemented in many modern compilers. They can be extended to perform many optimizations such as data flow analysis for identifying "dead" variables[1] and dependency analysis, for instance for eliminating common sub-expressions.

We start by writing the residual computation in *Three Address Code (TAC or 3AC)*. TAC is an extensively used intermediate representation (IR) which simplifies program analysis and optimizations. The format is defined as follows: each statement contains at most 2 source addresses (right-hand-side), and one destination address (left-hand-side). Each statement represents one instruction/operation. In order to rewrite the code we insert temporary variables, using names such as `t1`, `t2`, etc. The result can be seen in Listing 11. The format is essentially a linearization of the AST of a program and simplifies operations such as instruction reordering.

Figure 3.1 contains a directed acyclic graph (DAG) for the residual computation. The operations it represents are evaluated from the bottom, and upwards, so the last operation to be performed is addition. The blue nodes represent constants (zero derivative). White nodes represent functions or variables representing domains, and the grey nodes represent operations and variables that will have different values/results across iterations. Another property of the grey nodes is that they involve calculations on the derivatives. It should be noted that the graph does not represent the correct order of operations when it comes to associativity, as some of the nodes are reordered in order to make the graph more readable. Table 3.1 shows an overview of the operators that are used in the residual computation, which can be useful for analyzing the impact that an optimization might have.

Since the graph is a DAG, we can exploit several of its properties. For instance, if an operation node has two or more parents, we have common sub-expressions. As we can see in the graph, common subexpressions is not a problem. Since we know that the blue nodes will not change across iterations[2], we can also tell that any blue node that has an operator as a parent can be precomputed, reducing the number of calculations in each iteration. In this case, the following expressions can be precomputed: (1) `-trans`, (2) `dir_trans * dir_sign` and (3) `(dt / (cv * vol))`. This is valuable information, since the compiler can detect such expressions and perform optimizations after analyses on the AST.

[1] The analysis is called live variable analysis in the compiler literature.

[2] `u0` will get updated in every timesteps, but is constant across Newton iterations in a single timestep.

```
1  t1 = -itrans
2  t2 = Gradient(u)
3  ifluxes = t1 * t2
4  u_dirbdycells = u On dir_cells
5  t3 = u_dirbdycells - dir_val
6  t4 = dir_trans * dir_sign
7  dir_fluxes = t4 * t3
8  bfluxes = dir_fluxes Extend BoundaryFaces()
9  t5 = ifluxes Extend AllFaces()
10 t6 = bfluxes Extend AllFaces()
11 fluxes = t5 + t6
12 t7 = cv * vol # Scalar * CollOfScalar
13 t8 = dt / t7
14 t9 = u - u0
15 t10 = t9 + t8
16 t11 = Divergence(fluxes)
17 t12 = t10 * t11
```

Listing 11: The residual computation written in Three Address Code (TAC) format. Since the sequence contains no branches, it forms what is known as a *basic block*. The statements on the lines 1 to 11 is the flux computation and 12 to 17 is the final residual computation.

Operator	computeBoundaryFlux	computeInteriorFlux	computeResidual	**Total**	**Comment**
*	2	1	2	5	Lines 3, 6, 7, 12, 17
Extend	1	0	2	3	Lines 8 to 10
+	0	0	2	2	Lines 11 and 15
Binary -	1	0	1	2	Lines 5 and 14
On	1	0	0	1	Line 4
/	0	0	1	1	1 Scalar / CollOfScalar, Line 13
Unary -	0	1	0	1	Line 1
Gradient	0	1	0	1	Line 2
Divergence	0	0	1	1	Line 16
Total	5	3	9	17	

Table 3.1: The table contains the number of calls for each operation in the residual computation. The columns contain numbers for each function which is called, as well as the total. The line numbers listed refers to the location of the operations in the TAC code (Listing 11).

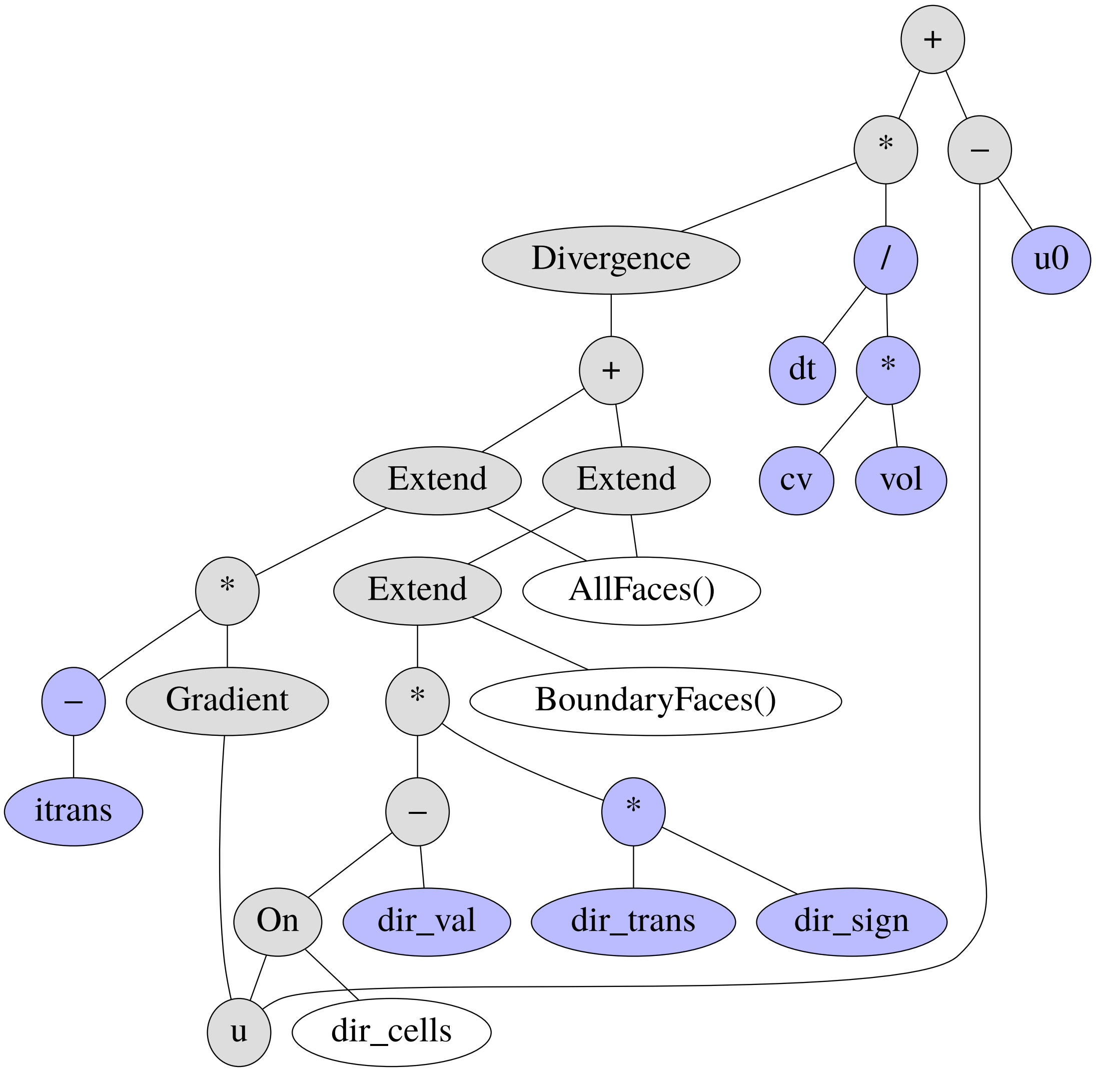


Figure 3.1: A directed acyclic graph (DAG) describing the residual computation. The blue nodes represent constants (no derivative), grey nodes represent collections with derivatives and white nodes are domains used with the On and Extend operators. Also note that `dt` and `cv` are scalars. Lastly, note that the graph depicts the relation between the operators, but does not retain the correct associativity with respect to the code.

3.3 Memory Usage

This section describes the memory usage for the implicit heat equation program. We assume a grid of size 250 × 100 × 100 (Table 3.2). We focus on the grid itself as well as the variables that are defined in the Equelle program, excluding the temporaries that are created in the backend. Overhead caused by temporary objects will be discussed in the next section, where we measure performance using the profilers and nvidia-smi.

The intent of this analysis is to estimate the memory impact of these variables. By using very simple reasoning about our program, we can also look for optimization possibilities. When we measure performance by profiling, we can also use the results from this section when inspecting the memory usage in the nvidia-smi output.

First we calculate the total footprint of the grid itself (excluding domains). The Equelle grid contains 8 data arrays that describe its geometry and the relations between its cells and faces. Table 3.3 lists the arrays, as well as their memory sizes. The total memory usage of the grid is 633.84 MBs. Table 3.4 lists the collections that are stored during the residual computation. They are all destroyed after each residual computation. Their total size is 777.6 MBs. If we include the `u` parameter with an identity matrix as its derivative, the total is 837.6. The size of the gradient matrix is 208.32 MB. The size of the divergence matrix is 190 MBs. Variables in the setup phase account for 222.56 MBs. In the next section, when we profile the application, we will use these numbers to validate our assumptions.

x	y	z	`AllCells`	`AllFaces`	`InteriorFaces`	`BoundaryFaces`
250	100	100	2,500,000	7,560,000	7,440,000	120,000

Table 3.2: The table shows the number of entities in each domain for a given grid (x, y, z).

Variable	**Elements ∗ Data Type**	**Size (MB)**
`cell_centroids_`	(3 ∗ 2,500,000) ∗ 8	60
`face_centroids_`	(3 ∗ 7,560,000) ∗ 8	181.44
`cell_facepos_`	(2,500,000 + 1) ∗ 4	10
`cell_faces_`	(15,000,000) ∗ 4	60
`cell_volumes_`	(2,500,000) ∗ 8	20
`face_areas_`	(7,560,000) ∗ 8	60.48
`face_cells_`	(2 ∗ 7,560,000) ∗ 4	60.48
`face_normals_`	(3 ∗ 7,560,000) ∗ 8	181.44
Total	-	633.84

Table 3.3: The table shows the size of each individual array in the grid as well as the total. The centroids and normals are vector types and are therefore multiplied by the number of dimensions in the grid.

	Values		**Derivative**			**Total**
Variable	*Elements*	*Size (MB)*	*Rows*	*NNZ*	*Size (MB)*	*Size (MB)*
`ifluxes`	7,440,000	59.52	7,440,000	14,880,000	208.32	267.84
`bfluxes`	120,000	0.96	120,000	20,000	0.72	1.68
`fluxes`	7,560,000	60.48	7,560,000	14,900,000	209.04	269.52
`residual`	2,500,000	20	2,500,000	17,380,000	218.56	238.56
Total	-	140.96	-	-	636.64	777.6

Table 3.4: Size information for all collections that are stored on the device in `computeResidual`. All of these are destroyed and reallocated for each timestep.

Matrix	**Rows**	**NNZ**	**Size (MB)**
`Gradient`	7,440,000	14,880,000	208.32
`Divergence`	2,500,000	15,000,000	190
Total	-	-	398.32

Table 3.5: The table contains size information for the gradient and divergence matrices, which are constructed once and stored on the device throughout the simulation.

3.4 Profiling

In this section we use profilers and nvidia-smi to measure the performance of the CUDA backend, which will help us to locate bottlenecks. We will get insights into the cost of the operations involved, both at a high and a low level of the code. For our profiling, we are using the system which was described in Section 2.7[3], running CUDA 8.0, and GPU driver version 396.54.

First we use the Nvidia Visual Profiler to generate timelines. This will let us get an overview of the program execution. Figure 3.2 shows a timeline with NVTX markers that we have added for the first two timesteps, which perform two iterations each. Above the markers is the CUDA API activity, and underneath it shows GPU activity. In the first iteration, computing the interior flux and the residual takes more time than in subsequent iterations. The reason is that the gradient and the divergence matrices are constructed on the CPU and copied to the device for reuse. We can clearly see that it is done on the CPU as there is no GPU activity and no CUDA API activity in those periods. Now we move on to study one single iteration, so that we can get more information about each computation step.

Figure 3.3 shows the steps in a single residual computation. The steps are: (1) computing the interior flux, (2) computing the boundary flux, (3) adding the interior and boundary fluxes together and finally (4) computing the residual. Adding the fluxes together is the most expensive step, and we can se a relatively high amount of GPU activity in that region.

At the end of step 1, 3 and 4, there are periods where there is CUDA API activity, but no GPU activity. It is mostly `cudaFree` that is being called, so we can assume that the gaps are due to cleanup steps. Since `cudaFree` is a blocking call, no new kernels can be launched until it is finished. In Figure 3.4 we see the timeline we get after

[3] Laptop with i7-7700HQ, GTX 1060 with 6 GBs of memory, 16 GB RAM, running on Ubuntu 16.04.

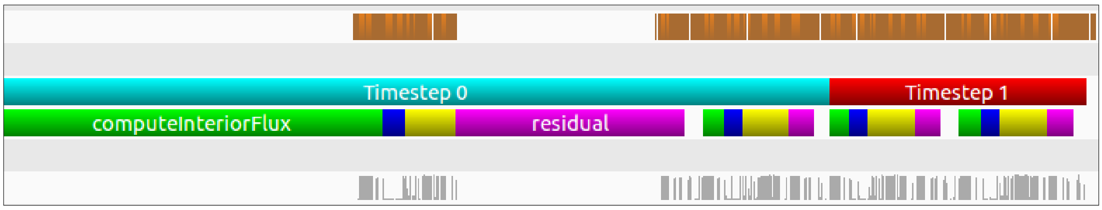


Figure 3.2: Timeline for the first two timesteps with markers. The top shows CUDA API calls and the bottom shows GPU activity. The large gaps in CUDA and GPU activity are due to the construction of the gradient and the divergence matrices on the CPU.

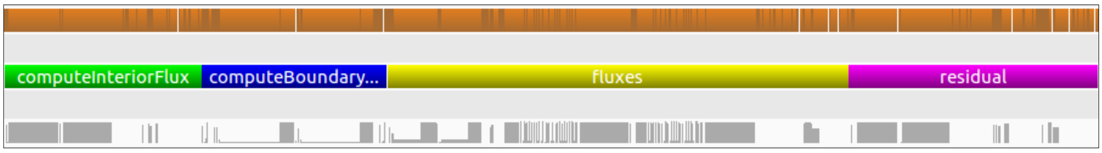


Figure 3.3: Timeline for a single residual computation with NVTX markers for each step. The top shows CUDA API calls and the bottom shows GPU activity.

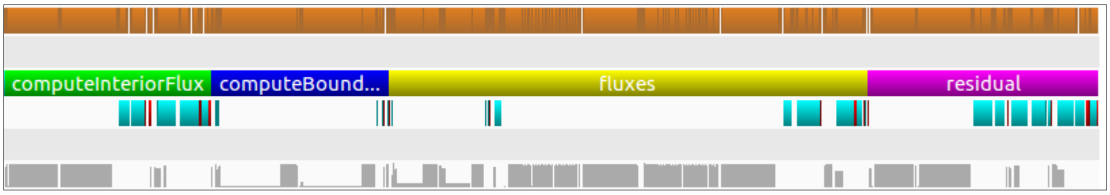
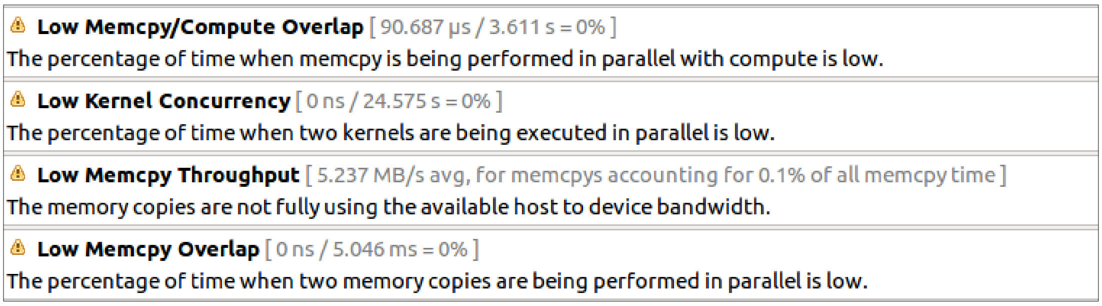


Figure 3.4: Timeline for one residual computation, with markers for each step as well as for the destructors of `CudaMatrix` and `CudaArray`. The markers for the destructors are placed below the ones for the computation steps. Notice that there is no GPU activity where the destructors (`cudaFree`) are called.



⚠ **Low Memcpy/Compute Overlap** [90.687 μs / 3.611 s = 0%]
The percentage of time when memcpy is being performed in parallel with compute is low.

⚠ **Low Kernel Concurrency** [0 ns / 24.575 s = 0%]
The percentage of time when two kernels are being executed in parallel is low.

⚠ **Low Memcpy Throughput** [5.237 MB/s avg, for memcpys accounting for 0.1% of all memcpy time]
The memory copies are not fully using the available host to device bandwidth.

⚠ **Low Memcpy Overlap** [0 ns / 5.046 ms = 0%]
The percentage of time when two memory copies are being performed in parallel is low.

Figure 3.5: The figure shows the output from the Nvidia Visual Profiler after running its GPU analysis in *Guided Mode*.

```
sudo nvprof --cpu-profiling on --cpu-profiling-frequency 500Hz
     --cpu-profiling-mode top-down --cpu-profiling-thread-mode separated
     ./out_heateq_cuda ../heateq/params.param
```

Listing 12: nvprof command line for profiling the CPU. We specify that the CPU's program counter will be sampled 500 times every second. Our application launches four threads, but only one of them does anything CUDA related. We therefore set the thread mode to *separated*.

adding markers for the destructors of `CudaMatrix` and `CudaArray`. Our assumption is confirmed, as the timeline clearly shows that the destructors are dominating those regions.

Using the Nvidia Visual Profiler's guided mode, we run a GPU analysis which gives us the output in Figure 3.5. The output shows that the memory throughput is low and that there is very little overlap between CUDA operations such as memory copies and kernel calls. It comes as no surprise that there is little overlap, as the use of several CUDA streams have not been implemented in Equelle. As for the memory bandwidth issues, we need to investigate a bit more.

We can inspect the individual memory copies by using the *GPU Details* tab. Each entry shows the type of transfer (HtoD/DtoH/DtoD, asynchronous/synchronous), the transfer size as well as the achieved throughput of the transfers. We can also click on the entries in order to have them highlighted in the timeline. When we inspect the host-to-device (HtoD) transfers individually, we immediately recognize the transfer sizes from our previous analysis (Section 3.3), most of them being for the gradient and divergence matrices, and the grid. 15 out of 17 transfers are done synchronously. The achieved throughputs are mostly between 5.5 and 11 GB/s, so nowhere near the 5.236 MB/s average that the profiler lists in Figure 3.5. Moving on to the device-to-host transfers (DtoH), we start to see where the low bandwidth issue comes from. There are thousands of transfers of 4 and 8 bytes with an achieved throughput of between 0.28 MB/s and 12 MB/s. The timeline shows that these small transfers of 4 bytes are done in relation to kernels for sparse matrix multiplication. Most notably the transfers are made interleaved with the `stable_sort_by_key` kernels, which are called before the kernel that finds the new sparsity pattern (`csrgemmNnz`) of the result matrix, and the multiplication kernel (`csrgemm`). They are called when cuSPARSE performs multiplications which involve transposed matrices. The `Extend` operator is the only part of Equelle which uses this `csrgemm` version.

In Figure 3.6, we see the timeline for the same iteration after adding markers for `gemmNnz` and `gemm`. We can observe that the GPU utilization varies a great deal across each call. For the multiplication in `computeInteriorFlux` the utilization is high, but in `computeBoundaryFlux` the GPU utilization is low for the majority of the time for both `gemmNnz` and `gemm`. For our simulation we know that there are very few entities on the boundary compared to the interior part of the grid, so the SMs of the GPU aren't fully utilized.

Now we move on to generating a CPU trace, in order to understand where the program actually spends most of its time. We use the command line in Listing 12 to generate the trace. The simulation takes 43.19 seconds.

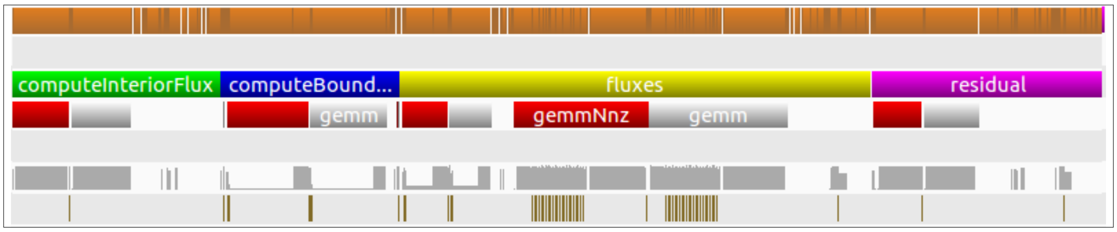


Figure 3.6: Timeline for a single iteration with markers for each step and for matrix multiplication. `gemmNnz` is the procedure which finds the new sparsity pattern of the matrix and `gemm` is the multiplication procedure. Below the markers we see the GPU activity. For the multiplication kernels, the GPU utilization varies a great deal. The bottom of the timeline shows device-to-host transfers. Notice the sequences of small transfers during multiplication.

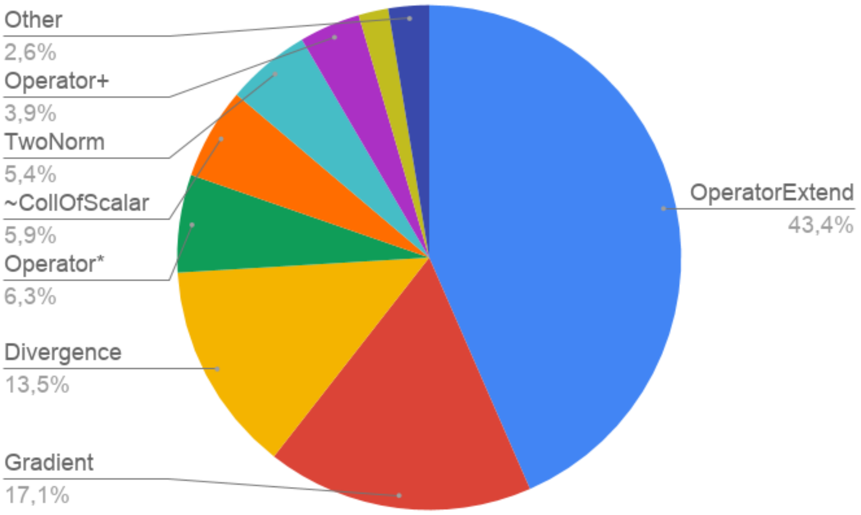


Figure 3.7: Time distribution for the implicit heat equation reported by the CPU trace. We can clearly see that the extend operator is a bottleneck, taking 43.4 percent of the total execution time. The secondary bottlenecks are the gradient and divergence operators.

Figure 3.7 shows the time distribution of the most dominating parts of the code. The results clearly show that the `Extend` operator is a bottleneck, taking over 40 percent of the total simulation time. Figure 3.8 shows the call graph we get from the trace, for the `Extend` operator. `operatorExtend` calls both `extendToFull` and `extendToSubset`. `extendToFull` is a function which extends a domain to the whole grid. `extendToSubset` calls `extendToFull` first, and then restricts it to the subdomain.

When we look at the gradient's trace, we see that sparse matrix multiplication is a bottleneck there as well. For the divergence operator, we see exactly the same as for the gradient. Since the time to construct the matrices would not increase as we run the simulation for more timesteps, we do not need to consider this as a bottleneck.

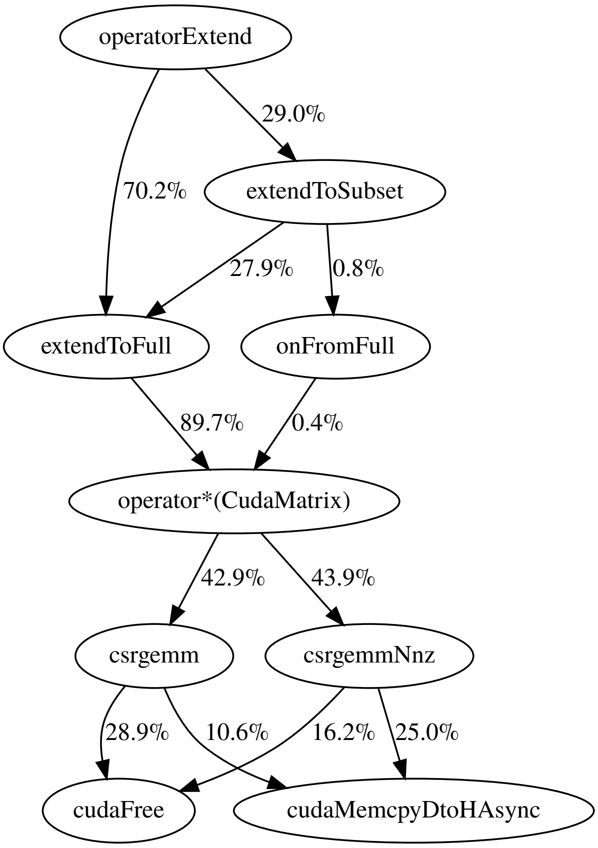


Figure 3.8: The figure shows a call graph for the `Extend` operator with percentages of its total runtime. `cudaFree` and asynchronous device-to-host copying are responsible for 80.7%. Most of the remaining time is spent on memory management of `CudaArray` and `CudaMatrix`. In `extendToFull` alone we lose about 9% to memory management.

The trace also shows that for `operator*`, which is the `CollOfScalar` multiplication, 80% of the time is spent in the `CudaMatrix` destructor. For `twoNorm`, 87.7% of the time is spent performing matrix multiplication.

We have generated traces for 7 different grid sizes using 100 timesteps to see if there is a shift in bottlenecks when we scale the grid. The results can be seen in Figures 3.9 and 3.10. They show that the operators keep their relative places for all the grid sizes.

Now we run nvprof in its summary mode to get statistics on GPU activity and CUDA API calls (Listing 13). The top two kernels are related to the sparse matrix multiplication (`csrgemm`), where the multiplication kernel itself and the kernel for finding the non-zero pattern of the resulting matrix are equally dominating at around 41% in total, with 280 calls being made to each of them. This equates to 14 calls per timestep. Next on the list is device-to-device memory transfers, with 12.34% of the time and 7,929 calls being made. Thirdly we see that there are calls to kernels for converting CSR to CSC format, which is used to transpose matrices in cuSPARSE. They are called 120 times each, or 6 times per timestep. The `sort_by_key` kernels account for more than 10% and are called 1,840 times each. We also see that sparse

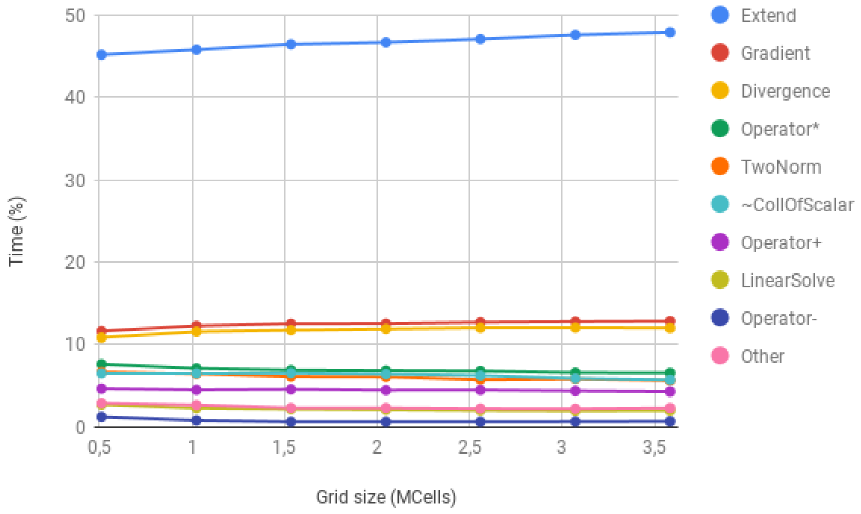


Figure 3.9: The chart shows the percentage of the total time for the most significant parts from the CPU trace, and how it changes as we increase the grid size. What we see is that the extend operator takes a larger percentage as the grid size increases. We also see that there is no shift in bottlenecks. Figure 3.10 supplements this chart and shows how the percentages of each operator changes between each grid sample. The data was collected using 100 timesteps.

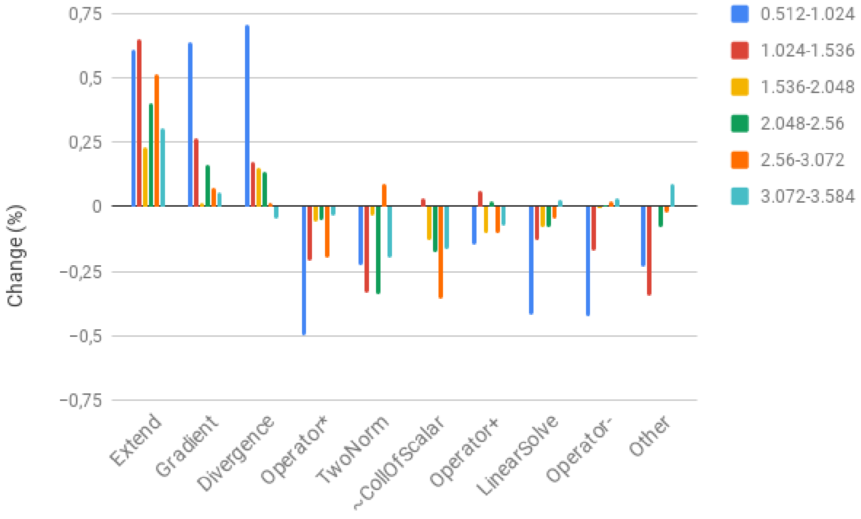


Figure 3.10: The chart shows that the extend, gradient and divergence operators take increasingly a larger portion of the total time as the grid size is scaled up. The columns denote scaling between different grid sizes. 0.512-1.024 means that the grid was scaled from 0.512 to 1.024 megacells. It is worth noting that for the minus operator, the percentages decreases for the first half and then starts to increase. Even though this might indicate a shift, our sample size is too small for us to make any conclusions. The data was collected using 100 timesteps.

```
Time(%)      Time     Calls     Name
 20.61%  5.72949s       280     void csrgemm_kernel2
 20.34%  5.65499s       280     void csrgemmNnz_kernel2
 12.34%  3.43084s      7929     [CUDA memcpy DtoD]
 10.08%  2.80228s       120     void CsrToCsc_kernel_build_cscColPtr<float, ...>
 10.05%  2.79374s       120     void CsrToCsc_kernel_build_cscColPtr<double, ...>
  5.27%  1.46576s      1840     void stable_sort_by_key_local_core
  5.26%  1.46206s      1840     void stable_sort_by_key_merge_core
  2.79%  776.41ms       240     void convert_CsrToCoo_kernel
  2.68%  745.86ms       120     void csrgeam_windowBased_core<double, ...>
  1.90%  529.46ms       200     equelleCUDA::wrapCudaMatrix::diagMult_kernel(double*, ...)
  1.51%  418.74ms       120     void csrgeam_windowBased_core<float, ...>
  1.34%  372.09ms      2000     void stable_sort_by_key_stop_core<...>

Time(%)      Time     Calls       Avg       Min       Max  Name
 60.85%  24.3709s     12781  1.9068ms  4.7090us  306.25ms  cudaFree
 28.62%  11.4624s      7340  1.5616ms  5.4750us  51.143ms  cudaMemcpyAsync
  8.32%  3.33155s     12780  260.68us  3.7580us  121.55ms  cudaMalloc
  0.86%  343.24ms       240  1.4301ms  136.79us  5.5551ms  cudaEventSynchronize
  0.69%  274.47ms      7009  39.159us  5.9120us  27.979ms  cudaMemcpy
  0.42%  169.58ms     20548  8.2530us  3.4120us  2.9027ms  cudaLaunch
```

Listing 13: Output from running the implicit heat equation simulation with nvprof's summary mode. The top table shows the most time consuming operations on the device with percentages of the total GPU time. The bottom table shows the most time-consuming CUDA API and driver calls, with cudaFree taking 60.85% of the time. Less relevant entries are omitted. Also notice that cudaFree is called one time more than cudaMalloc. The reason is that cusparseCreate(), which initializes cuSPARSE, calls cudaFree implicitly.

matrix addition (csrgeam) is called 120 times, or 6 times per timestep.

The bottom table gives an overview of the calls being made to the driver API or to the runtime API. cudaFree is called 12,781 times and accounts for over 60% of the total runtime. Performing that many cudaFree calls is bad for two reasons: (1) freeing memory is in itself an expensive operation, (2) cudaFree is a blocking/synchronizing call, which means that when it is called, all the processes running on the device must finish before memory can be freed and the program can proceed. This can potentially be very bad for performance as it prohibits asynchronous activities such as kernels and asynchronous memory copies from running independent of the CPU. We should avoid synchronization whenever possible. Blocking operations include cudaFree, cudaMalloc and cudaMemcpy. By using output to console, we find that 3,090 of the cudaFrees are from the CudaMatrix destructor, and 1,785 is from the CudaArray destructor. Further we see that cudaMemcpyAsync is performed 7,340 times (367 per timestep) and takes over 28% of the time. cudaMalloc is called one time less than cudaFree, at 12,780. The table also shows that 20,548 kernels are being launched.

In our next benchmarks we measure the overall performance of Equelle for the implicit heat equation. A common metric for measuring performance when dealing with grids of cells, is the number of cells that the system can process per second.

We are using different grid sizes to determine at point the CUDA backend gets fully saturated. We run the simulation for 20 timesteps. When the GPU is saturated, we should see that the simulation time scales linearly with the amount of work we add.

Figure 3.11 shows the performance we achieve with 28 different grid sizes, measured in megacells per second. The throughput is 1.18 megacells per second for a grid which is 3.5 megacells large. The performance increases sharply when increasing the grid on the lower end, but flattens out as the grids get larger. The tendency is expected, as it is typical for how performance scales on parallel processors as the workload is scaled up.

Figure 3.12 shows the total amount of time it takes to complete 0 to 5 timesteps using a grid size of 2.5 megacells. When we run the simulation for 0 timesteps, it only sets up the grid and the initial variables and then does the cleanup phase, which only takes approximately 1 second. When we increase the timesteps to 1, we see a large leap (6 seconds) in the simulation time. A large part of this increase comes from the construction of the gradient and the divergence matrices of the grid (on the CPU). As we would expect, the time scales linearly from 1 timestep and upwards. Each timestep above 1 adds approximately 1.9 seconds.

In our next benchmarks, we will have a look at the actual memory usage. We collect the data using nvidia-smi. Note that the data in the charts are in mebibytes (MiB) and not MB[4]. The charts have been annotated manually by the use of the timelines generated by the Nvidia Visual Profiler as well as matching the memory sizes we estimated in Section 3.3, to make clear what causes the memory activity. The annotations also indicate what data we have stored on the device throughout the simulation.

In Figure 3.13 we see the memory activity for the first two timesteps. The very first activity we see is the allocation of the grid followed by the construction of the interior faces (marked `ifaces`). The grid takes up just over 600 MiB, which is what we estimated in Section 3.3. As we have already observed in the annotated timelines, every timestep performs two residual computations/iterations. The very first computation takes a lot longer than the subsequent ones because the gradient and the divergence matrices are being constructed on the CPU. We have also observed that there is no CUDA activity in these two periods, which we can clearly see in the chart as there is no memory being allocated nor freed.

In Figure 3.14 we can see the `itrans` section, with the rest of the setup phase. The chart is divided into sections that first set the variables containing the `FirstCell` and the `SecondCell` of the interior faces. In the `itrans` section, there is a big overhead to the computation, which loads 776 MiB into memory (at its peak) to evaluate the whole expression. The expression is:

```
itrans = k * |ifaces| / |Centroid(first) - Centroid(second)|
```

The timeline shows that the calculation is performed in the following order: (1) Find centroids of `first`, then find the centroids of `second`. (2) Perform the subtraction of the two centroid collections. (3) Find the norm of the centroids. (4) Find the norm of `ifaces`. (5) Perform the division between the norms. (6) Multiply the `Scalar k` by the previous result. Two collections of centroids will alone account for 170.3 MiB. At the

[4] One MiB is 1024^2 bytes, while a MB is 1000^2 bytes.

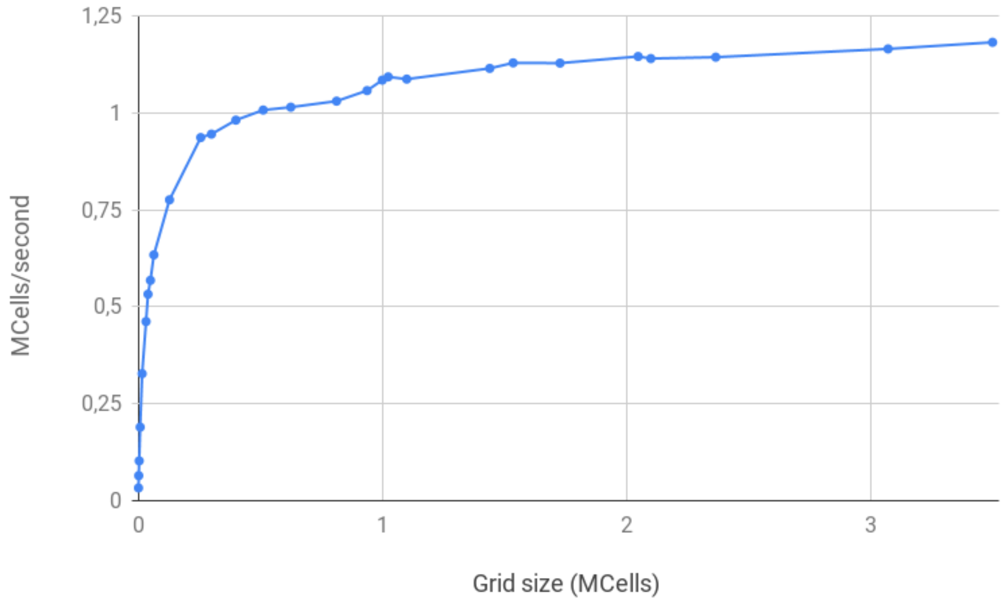


Figure 3.11: The chart shows that the throughput sharply increases and then flattens out as we scale up the grid. The data is collected using 28 distinct grid sizes, running for 20 timesteps.

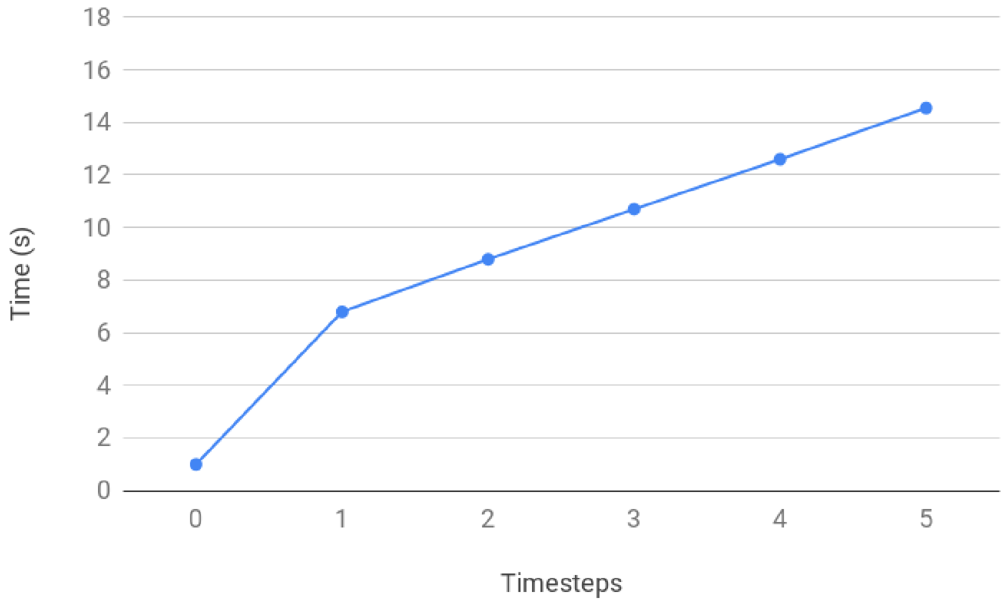


Figure 3.12: The chart shows how much time it takes to complete from 0 to 5 timesteps using a grid of 2.5 megacells. As expected, the time scales linearly with respect to the number of timesteps. Each timestep above 1 adds approximately 1.9 seconds.

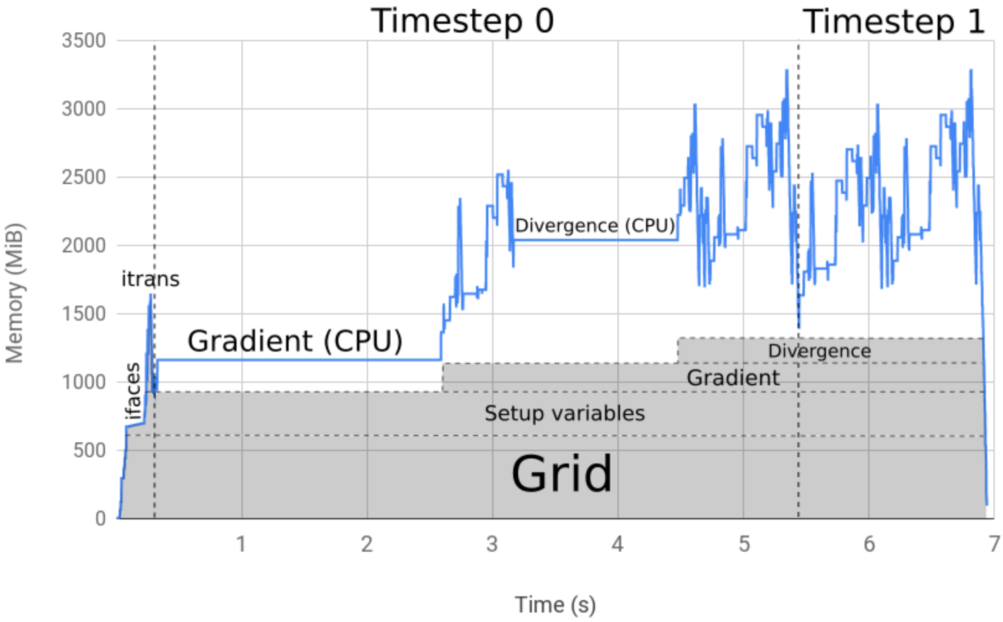


Figure 3.13: The chart shows the memory usage in mebibytes (MiB) for the first two timesteps. The annotated areas show the amount of memory that remains on the device until the end of the simulation. The residual is computed 4 times, and as we have seen earlier, the long inactive periods are caused by the construction of the gradient and divergence matrices on the CPU. After the divergence is calculated, the total of memory that remains on the device is more than 1,350 MiB (more than 1,400 MB).

beginning of the calculation, we can see that each of the centroid operations account for approximately double of the estimated size, suggesting that there is an overhead equal to one temporary `CollOfVector` in each of them. When reading the Dirichlet boundary data, there is not much variation in the memory usage. The data arrays are quite small (only 20,000 elements each). The flat parts are where the CPU is reading the data. In the next section of the chart where the Dirichlet transmissibilities are being set, the `On` operator is performed on three collections to yield `dir_cells`, `dir_sign` and `dir_trans`.

Figure 3.15 shows memory usage for one iteration, annotated to show each computation phase and significant operations. We will use the chart as a reference in our optimization chapter.

The last thing we do is to determine the largest grid we can run on the GTX 1060. A grid of size 350 × 100 × 100 (3,500,000 megacells) seems to work fine, but scaling it up to 365 × 100 × 100 causes a memory overflow. The grid contains 3,650,000 cells and 11,033,000 faces.

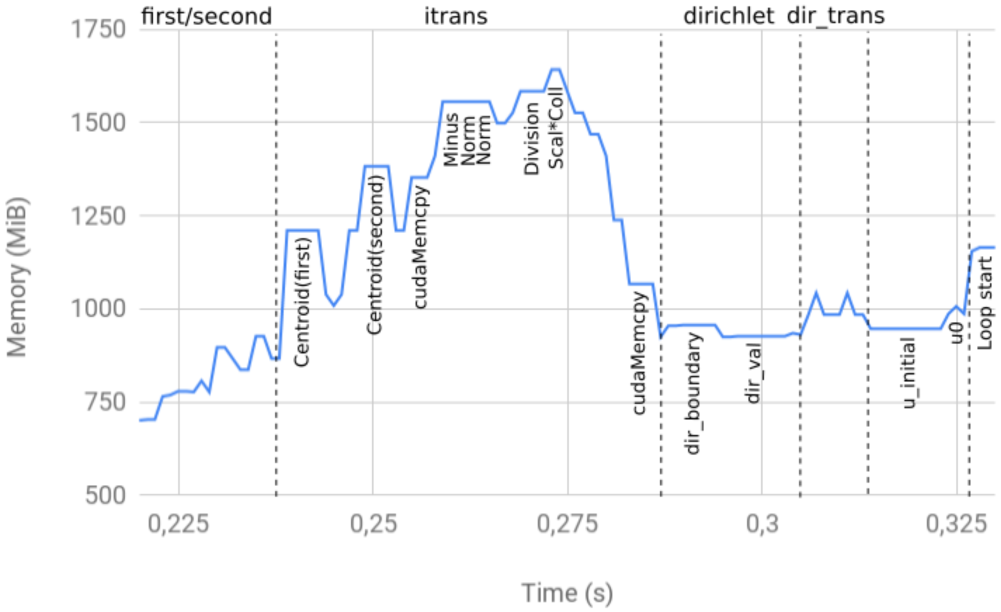


Figure 3.14: The chart shows the setup phase in the implicit heat equation program. `itrans` is corresponds to the spike we see in 3.13, and is the calculation of the interior transmissibilities. It shows the whole setup step in the implicit heat equation, which includes (1) Setting the collections containing the first and second cell of the interior faces. (2) Computing the transmissibility of the interior faces. (3) Reading the Dirichlet boundary values from file. (4) Setting the Dirichlet transmissibility. (5) Read the initial temperatures into `u_initial` and set `u0`. (6) Finally we see the start of the timestepping. Evaluating the `itrans` expression demands 776 MiB of memory.

3.5 Additional Observations

In this section we perform measurements and analyses that are not strictly related to any of our simulations. Since the simulators written in Equelle will most likely only use a subset of the language's available functionality, we should conduct some experiments that highlight other parts of Equelle.

3.5.1 Destructor Calls Per Operator

We already know that the CUDA backend performs a large number of kernel launches and calls to expensive operations such as `cudaFree` and `cudaMalloc`. Table 3.6 gives an overview of the calls being made to the destructors of `CudaArray` and `CudaMatrix` in addition to the total amount of `cudaFree` they account for. The data was collected using the `autoDiff` test in the CUDA backend test suite. Calls to `cudaFree` which is being done outside of the destructors are not included, as the table is meant to give an indication of the amount of temporary objects in Equelle. Since the number of temporary objects can depend on the compiler being used and the optimization level, we choose to use actual measurements instead of making estimates based on reading

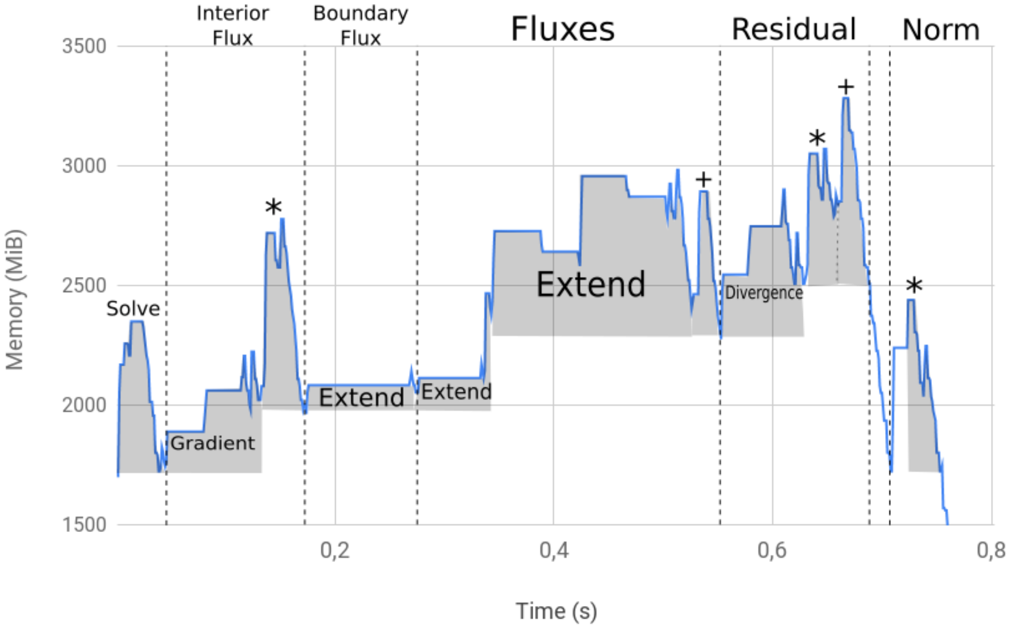


Figure 3.15: The chart shows the memory usage of a single iteration, measured in mebibytes (MiB). The most dominant portions of the program are highlighted. Asterisk (∗) denotes `CollOfScalar` multiplication and plus (+) denotes `CollOfScalar` addition. The memory usage reaches its peak of 3,285 MiB in the residual section when performing an addition. Notice how much of the time is spent performing the `Extend` operator.

the code. We have measured using G++ version 5.5 with the `-O3` optimization option. There are two versions of the divergence operator, one which calls `cudaFree` 30 times and one which calls `cudaFree` 10 times. The difference is due to the first one being called on a collection that needs to be extended to `AllFaces` first. The latter is already defined on `AllFaces`, so no extension is needed.

3.5.2 Incorrect Results from GPU Solvers

We have experienced issues with the implicit two-phase flow simulation. When we run it with any of the GPU solvers and preconditioners, the results are not comparable to those of the CPU (which we assume to be correct). After running for a while, all the pressure values in the output becomes `nan`. We do not know what the reason is. We get comparable results to the CPU if we use the CPU solver with the GPU backend.

3.5.3 Race Conditions in csrgemm

In the implicit two-phase flow simulation, the results vary from run to run. After debugging we determined that the varying results are due to race conditions in the `gemm` procedure when we use a small grid. The race conditions were detected using `cuda-memcheck` with its racecheck tool when compiling for CUDA 9.

3.5.4 Explicit Zeroes in CudaMatrix

When we run the `autoDiff` test from the CUDA backend test suite, it fails because a matrix has the wrong `nnz` value. We get the following error message: `"Wrong number of nnz: 41072 should be 11392"`, so the number of nonzeroes is approximately 3.6 times too high. This is caused by a call to the ternary if operator in a previous step[5]. In Section 4.12 we will have a closer look at the ternary if operator and how these explicit zeroes end up in a `CudaMatrix`'s values.

3.5.5 Slow csrgeam in CUDA 9

The sparse matrix addition provided by cuSPARSE is 8 times slower in CUDA 9 than in CUDA 8. The kernels get launched with CUDA grids that are 8 times larger. We made a forum post in the Nvidia Developer Forums where Nvidia confirms that it is most likely a bug [6]. This is one of the reasons for not using CUDA 9 in our work on Equelle. We have submitted a bug ticket.

3.6 Summary

We have now analyzed the Equelle program which solves the heat equation implicitly with a focus on the timestepping and residual computation part of the program. We have used simple reasoning about the program as measuring its performance with profilers and `nvidia-smi` to look for ways of improving the Equelle compiler and CUDA backend. Our analyses have been done using a variation of grids, but we mainly used a grid of 2,500,000 cells and 7,560,000 faces for the benchmarks.

In our analysis we have established where the program spends most of its time. The Equelle `Extend` operator is a major bottleneck, and it takes 43.4 percent of the total time when we use a grid of 2.5 megacells. It is the extension of the matrices that takes 90 percent of the time, and is performed by doing a matrix multiplication with a transposed matrix, using cuSPARSE `gemm`. This multiplication procedure calls `cudaFree` implicitly, which makes the program stall until it is finished, which prohibits asynchronous execution. The transposition process itself, which is done as an initial step for both the procedure that finds the sparsity pattern of the matrix, as well as for the multiplication kernel, makes several thousand calls to `cudaMemcpyAsync` and kernels used for sorting indices. Using matrix multiplication is not necessary to perform the extension, as all that needs to be done to extend the matrix, is to change its row pointer array. Its `nnz` variable as well as the column index array and the value array will stay the same. In Chapter 4 we will optimize the operator and observe the results.

We have also seen that `cudaFree` stalls the program for approximately 60 percent of the time and is called 12,781 times when the simulation runs for 20 timesteps. Half of the stall time is related to cuSPARSE routines, and the rest comes from temporary data objects. As we have mentioned, not only is `cudaFree` an expensive call, but it is also a blocking call which stalls our program. This clearly makes it a performance

[5] The test suite uses regression testing, so each test is dependent on the results of preceding tests.

[6] https://devtalk.nvidia.com/default/topic/1038554/gpu-accelerated-libraries/slow-cusparsedcsrgeam-in-cuda-9-2/

issue, and we should look into ways of reducing the number of calls. Both by reducing the use of temporary objects in the backend as well as procedures that use it implicitly. In Chapter 4 we will attempt to do this by using *move semantics*, which is a concept in modern C++ designed specifically to reduce the cost of temporary objects.

Next in the list of observations, is the large fluctuation in memory usage. We have seen in our charts that certain operations cause spikes in the memory utilization on the device. For instance, for the `CollOfScalar` multiplication operator we could observe that 80 percent of the time were spent in the `CudaMatrix` destructor. This should be looked into, as solving it will allow us to run larger simulations, since we will be able to use larger grids without causing the memory to overflow. At this point the largest grid we can run for the implicit heat equation on the GTX 1060[7], is of size $365 \times 100 \times 100$, containing 3,650,000 cells and 11,033,000 faces.

Also related to memory usage, are data on the device that gets allocated and used once. We can optimize our program by freeing this memory once it has been used. In the implicit heat equation program, examples of two such variables are `first` and `second`, which contain the first cell and second cell of the interior faces. Performing the optimization is possibly something that needs to be done at compile-time.

The `twoNorm` operator computes the sum of squares of the values in a `CollOfScalar`. We have seen that the operator spends close to 90 percent of its time performing matrix multiplication, which is not necessary, since the matrix values are not needed for this calculation. We will optimize the `twoNorm` opera

In addition to finding bottlenecks, we have established that the simulation scales as expected when we increase either the grid size or the timesteps. When we increase the number of timesteps, each one adds the same amount of time. We also saw that when we add more work (larger grid) the typical case for parallel computers occurs: If the workload is too small, then the efficiency is low (not enough work for the GPU). If we add additional work, we see that the performance increases sharply in the beginning, and then we receive less of an increase. This observation can be seen in Figure 3.11. Equelle's peak performance for the implicit heat equation is approximately 1.18 megacells per second.

We have also highlighted other issues by running the two-phase flow simulation, as well as the test suite of the CUDA backend. First we found that the GPU solvers and preconditioners do not give results that are comparable to those of the CPU backend. We have also found that the `gemm` routine from the cuSPARSE library gives different results from run to run when used on the two-phase flow simulation. Running cuda-memcheck's racecheck tool showed that race conditions occur when Equelle grids are small. In addition we found that the `csrgeam` routine is 8 times slower in CUDA 9 than it is in CUDA 8. When we ran the `autoDiff` test suite, we also found that the ternary if operator causes explicit zeroes to get stored in the values of `CudaMatrix` objects.

[7] Note that up to 1 GB of memory is taken by graphical programs, making the effective memory size 5 GB.

Operator	~CudaArray	~CudaMatrix	cudaFree
Subset On Subset	3	10	33
Divergence(CollOfScalar$_{AD,InteriorFaces}$)	3	9	30
Ternary if	2	9	29
CollOfScalar / CollOfScalar$_{AD}$	4	8	28
CollOfScalar$_{AD}$ / CollOfScalar$_{AD}$	3	7	24
CollOfScalar$_{AD}$ / CollOfScalar	3	7	24
CollOfScalar$_{AD}$ * CollOfScalar	2	6	20
CollOfScalar$_{AD}$ * CollOfScalar$_{AD}$	1	5	16
Extend(InteriorCells, AllCells)	1	5	16
CollOfScalar * CollOfScalar$_{AD}$	1	5	16
Gradient(CollOfScalar$_{AD}$)	1	4	13
Sqrt(CollOfScalar$_{AD}$)	3	3	12
Scalar / CollOfScalar$_{AD}$	3	3	12
Divergence(CollOfScalar$_{AD,AllFaces}$)	1	3	10
CudaMatrix * CudaMatrix	0	3	9
On(CollOfScalar$_{AD}$,AllCells, InteriorCells)	2	2	8
CollOfScalar$_{AD}$ + CollOfScalar$_{AD}$	2	2	8
CollOfScalar$_{AD}$ * Scalar	2	2	8
CollOfScalar$_{AD1}$ - CollOfScalar$_{AD1}$	2	2	8
CollOfScalar$_{AD}$ - CollOfScalar	2	2	8
CollOfScalar - CollOfScalar$_{AD}$	2	2	8
-CollOfScalar$_{AD}$	2	2	8
CollOfScalar$_{AD}$ + CollOfScalar	1	1	4
Scalar * CollOfScalar$_{AD}$	1	1	4
CollOfScalar$_{AD1}$ - CollOfScalar$_{AD2}$	1	1	4
CollOfScalar$_{AD}$ / Scalar	1	1	4
Diagonal CudaMatrix * CudaMatrix	0	1	3

Table 3.6: The table gives an overview of the number of times the destructors of CudaArray and CudaMatrix are called for a selection of Equelle operators. This is before we have implemented any optimizations. CollOfScalar$_{AD}$ denotes a CollOfScalar with a derivative. The CudaFree values only account for the calls being made from the destructors. ~CudaMatrix calls cudaFree 3 times and ~CudaArray calls it 1 time. The measurements have been done using output to console.

Chapter 4

Implementation and Results

In this chapter, we describe implemented changes that address some of the issues we found in Chapter 3. We will follow the iterative development approach recommended by Nvidia (Section 2.7.1). For each implementation we perform an analysis, implement the changes we see fit and then go back to assessing our application to find another bottleneck to optimize.

We will perform six implementations. We first optimize the `Extend` operator which we have found to be the bottleneck in the implicit heat equation program. We then move on to implementing move semantics, which will reduce the cost of temporary objects. Thirdly we optimize the `CollOfScalar` multiplication operator, which is not memory efficient. In our fourth optimization, we eliminate matrix multiplication from the two-norm computation, since it is not needed. Our fifth implementation is an AST rewriter and is the first step towards making Equelle into an optimizing compiler. In our sixth and final implementation, we remove the explicit zeroes in `CudaMatrix` objects that are caused by the ternary if operator. Finally we summarize our results and perform a conclusive measurement.

4.1 Optimizing the Extend Operator

In our performance analysis we found that the `Extend` operator is the dominating bottleneck when solving the heat equation implicitly, taking more than 40 percent of the total simulation time. Specifically, the long execution time is caused by the use of transposed matrices in sparse matrix multiplication (`csrgemm`) when extending the derivative.

4.1.1 Design and Implementation

Performing the extension with a multiplication is completely unnecessary, as the only change in the new matrix is the number of rows. This means that the only operation we need to do is to allocate a new row pointer array, place the row pointer values from the original row pointer into the newly allocated row pointer and finally fill in the gaps.

The implementation consists of three steps and can easily be implemented using the Thrust library and is as follows:

1. Fill the new row pointer array with zeroes.
2. Insert the row pointer values from the original domain into their corresponding places in the extended domain. This is implemented with the *scatter* algorithm.
3. Use the *inclusive scan* algorithm to change the remaining zeroes to the correct non-zero values.

The code in Listing 14 implements the steps above.

A scan algorithm takes a binary operator which it applied to the input array from left to right. With the inclusive variant, earlier results are carried over and influences the results in subsequent evaluations. When using the maximum operator, the array values are set to the highest value that has previously been encountered. Figure 4.1b shows the result after evaluating the values {0 1 0 0 2 0 3 4 0}.

```cpp
// 1.1) Set up the output matrix. Also allocates row pointer array.
CudaMatrix der(full_size, in_data.der_.cols(), in_data.der_.nnz());

// Copy csrColInd, csrVal and fill csrRowPtr with zeroes
thrust::copy(thrust::device, in_data.der_.csrColInd(),
             in_data.der_.csrColInd()+in_data.der_.nnz(), der.csrColInd());
thrust::copy(thrust::device, in_data.der_.csrVal(),
             in_data.der_.csrVal()+in_data.der_.nnz(), der.csrVal());
// 1.2) Fill row pointer array with zeroes.
thrust::fill(thrust::device,der.csrRowPtr(),der.csrRowPtr()+der.rows()+1, 0.0);

// 2) Map values in set being extended to the new domain.
thrust::scatter(thrust::device, in_data.der_.csrRowPtr()+1,
                in_data.der_.csrRowPtr()+in_data.der_.rows()+1,
                from_set.begin(), der.csrRowPtr()+1);

// 3) Fill in the gaps of the rowPtr.
// {0, 0, 2, 0, 0, 4, 0, 5} becomes
// {0, 0, 2, 2, 2, 4, 4, 5}
thrust::maximum<int> binary_op;
thrust::inclusive_scan(thrust::device, der.csrRowPtr(),
                       der.csrRowPtr()+der.rows()+1, der.csrRowPtr(), binary_op);
```

Listing 14: C++ code implementing the optimized extend operator for sparse matrices. The extension is performed in three steps: (1) allocate new row pointer array of zeroes, (2) insert old row pointer into the new row pointer, (3) fill the remaining gaps.

4.1.2 Performance Results

Figure 4.4 illustrates the improvement. The optimized version of the extend operator shows a dramatic improvement over the old version. When running the heat equation with the same grid as before, `Extend` takes between 1.4 and 1.5 seconds, rather than

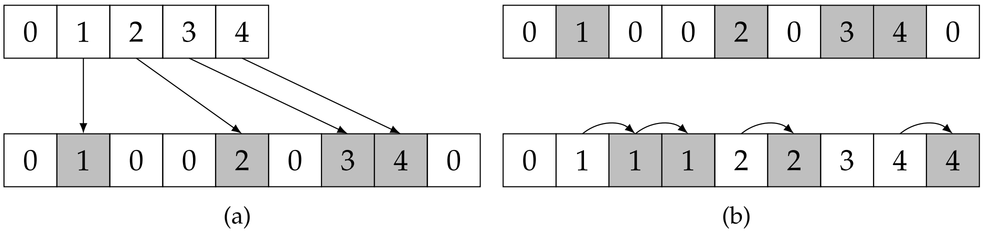


Figure 4.1: Extension of the row pointer in two steps. Figure (a) illustrates the mapping of the subset into the extended domain. Figure (b) illustrates the inclusive scan algorithm that fills in the missing values.

18 seconds, speeding it up by a factor of between 12x and 12.85x! The total runtime of the simulation is reduced from 43 seconds to 27 seconds. Previously we found that the `Extend` operator called the `CudaArray` destructor once and the `CudaMatrix` destructor 5 times. After our optimization, we see that the calls to the `CudaMatrix` destructor is reduced from 5 to 1.

Listing 15 shows the output from nvprof's summary mode after we have performed the optimization. The number of calls to `cudaFree` has been reduced by 2,800, from 12,781 to 9,981 and we see that the time has gone from 24.37 seconds to 14.55 seconds. The number of calls to `cudaMemcpyAsync` has gone from 7,340 to 1,240, and its execution time from 11.45 seconds to 4.258 seconds. Notice also that the number of calls to `cudaLaunch` have been reduced from 20,548 to 4,448.

In our initial profiling the transposition kernels (`CsrToCsc` etc.) were also listed, taking over 20% of the execution on the GPU. As we can see, they are no longer present. We also notice that the number of device-to-device (DtoD) copies have been reduced from 7,929 to 6,629. Its total time, however, is not that different has not changed much. It has gone from 3.43 to 3.12 seconds.

In Figure 4.2 we see the new time distribution in percent. The `Extend` operator now takes 5.3% of the total execution time, down from between opproximately 40%. Our new bottleneck is the gradient operator (28%), followed by the divergence (22.8%), `CollOfScalar` multiplication (10.7%), the `CollOfScalar` destructor (9.8%) and the two-norm (9.6%, used in the Newton loop). Figure 4.4 shows the before and after comparison for the program as a whole and for the extend operator isolated.

Figure 4.3 shows an updated memory activity chart. The `Extend` operator has been reduced to short spikes in increased memory usage instead of being large flat structures, like the gradient and the divergence. These spikes represent the temporary objects that are returned at the end of the function. In the next section we will use move semantics from modern C++ to reduce the overhead of such objects.

Figure 4.5 shows a memory chart for the extension of the interior fluxes and the boundary fluxes, before and after the optimization. The red line is for the optimized version, and the blue line is for the unoptimized version. Notice that the `Extend` operator now takes much less time to execute, but it still causes a large increase in memory usage. In the next section we will implement move semantics, which reduces the cost of temporary objects by reusing their data members.

```
Time(%)       Time     Calls     Name
 26.28%  3.26302s       160     void csrgemm_kernel2<double, ...>
 25.11%  3.11821s      6629     [CUDA memcpy DtoD]
 23.90%  2.96730s       160     void csrgemmNnz_kernel2<...>
  6.03%  748.17ms       120     void csrgeam_windowBased_core<double, ...>
  4.27%  530.19ms       200     wrapCudaMatrix::diagMult_kernel(double*, ...)
  3.32%  412.14ms       120     void csrgeam_windowBased_core<float, ...>
  1.29%  160.25ms       320     wrapCudaMatrix::initDiagonalMatrix(double*, ...)
  1.23%  152.94ms        17     [CUDA memcpy HtoD]

Time(%)       Time     Calls        Avg        Min        Max  Name
 64.64%  14.5528s      9981  1.4551ms  1.0540us  304.30ms  cudaFree
 18.91%  4.25821s      1240  3.4340ms  5.6590us  52.981ms  cudaMemcpyAsync
 12.49%  2.81227s      9980  281.79us  2.3240us  148.54ms  cudaMalloc
  1.48%  332.85ms       220  1.5130ms  127.99us  5.6105ms  cudaEventSynchronize
  1.21%  271.53ms      5909  45.951us  6.3960us  35.511ms  cudaMemcpy
  0.84%  189.24ms       480  394.26us  3.1990us  3.9312ms  cudaDeviceSynchronize
  0.25%  56.603ms      4448  12.725us  4.0910us  2.6597ms  cudaLaunch
```

Listing 15: Output from nvprof after we have optimized the extend operator. We see that matrix multiplication is called 120 times less, which brings the total number of calls to `csrgemm` and all its associated procedures down to 160. Notice also the substantial difference in calls to `cudaLaunch`, which is reduced from 20,548 to 4,448. That's a difference of 16,100, or a reduction of almost 80%!

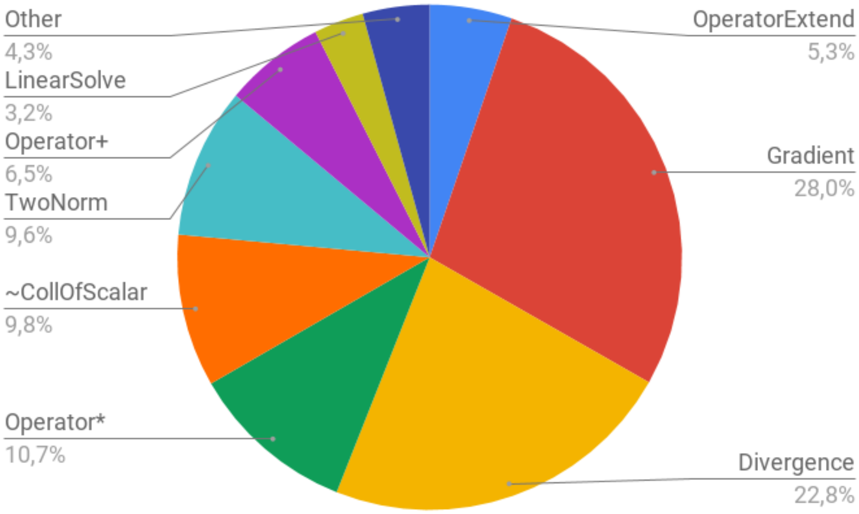


Figure 4.2: Time distribution for the implicit heat equation after optimizing the extend operator. In the initial profiling the extend operator took over 43 percent of the time, but now it takes 5.3 percent. It is sped up by a factor of 12x to 12.85x. We see now that the gradient and divergence operators are the main bottlenecks.

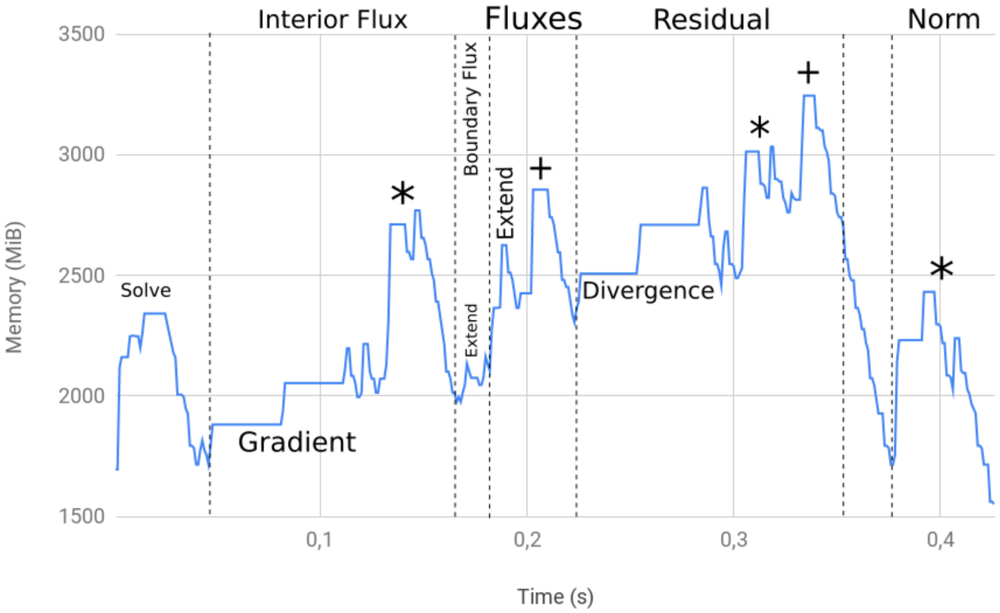


Figure 4.3: Memory usage for one iteration, after we have optimized the extend operator. The chart shows the same iteration as Figure 3.15. Notice that the extend operators have been reduced to spikes in memory usage over a short timespan, rather than large flat structures like the gradient and divergence. Also notice that the total time is down to approximately 0.425 seconds, from 0.76 seconds.

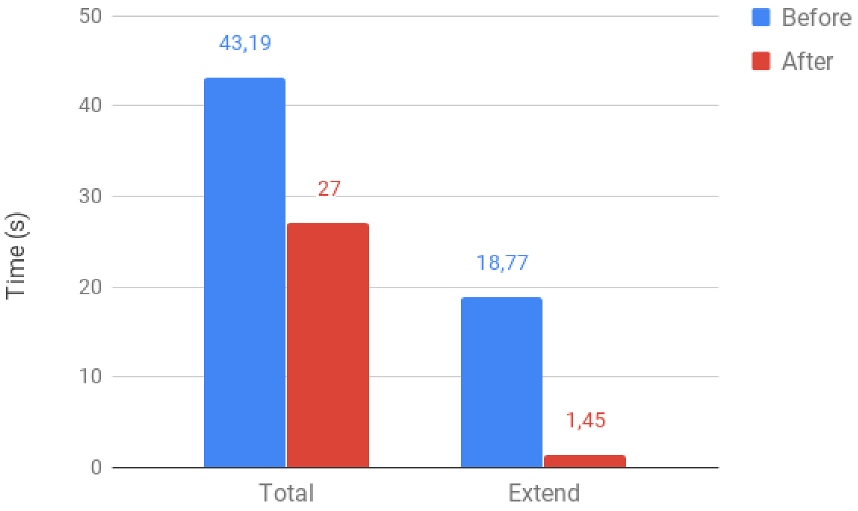


Figure 4.4: The chart shows the execution time in seconds before and after we optimized the extend operator. The left bars show before and after for the total time and the right bars show before and after for the extend operator itself.

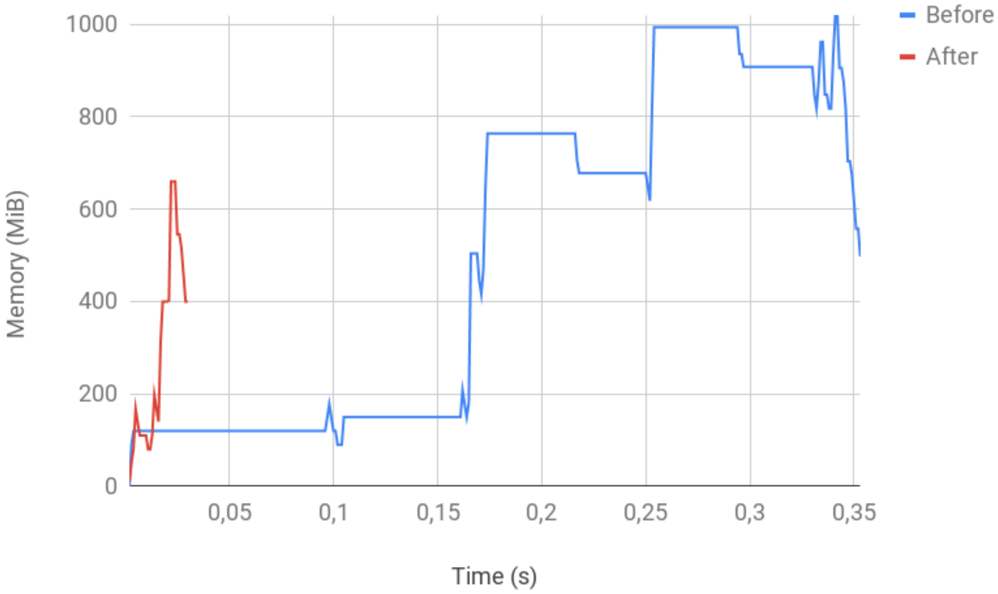


Figure 4.5: The chart shows the memory activity and run time of the two extend operator calls when adding the fluxes together, for both before (blue) and after (red) the optimization of the extend operator. The extend operator takes considerably less time, but we see that it still uses a lot of memory.

4.2 Move Semantics

As we now know, there are a large number of temporary objects that affect the performance of Equelle. They cause runtime overhead by calling `cudaFree`, `cudaMalloc`, and performing memory transfers on the device. In addition to being expensive calls, `cudaFree` and `cudaMalloc` are blocking calls that inhibit concurrency. We have also seen in previous profiling that we get spikes in memory usage which can cause the memory to flow over, preventing us from scaling up our simulations. Figure 4.6 shows a memory activity chart for a single iteration, highlighting some of the spikes created by temporary objects. In this section we implement *move semantics* in the CUDA backend to reduce the impact of temporary objects.

4.2.1 Design and Implementation

Move semantics is a concept in modern C++ that came with the C++11 standard. It reduces the overhead of temporary objects by *moving* rather than copying their values into other objects. We can identify these temporary objects by using *rvalue references* in our parameters in C++, which are denoted using double ampersands (`&&`), instead of the single ampersands used in traditional by-reference parameters.

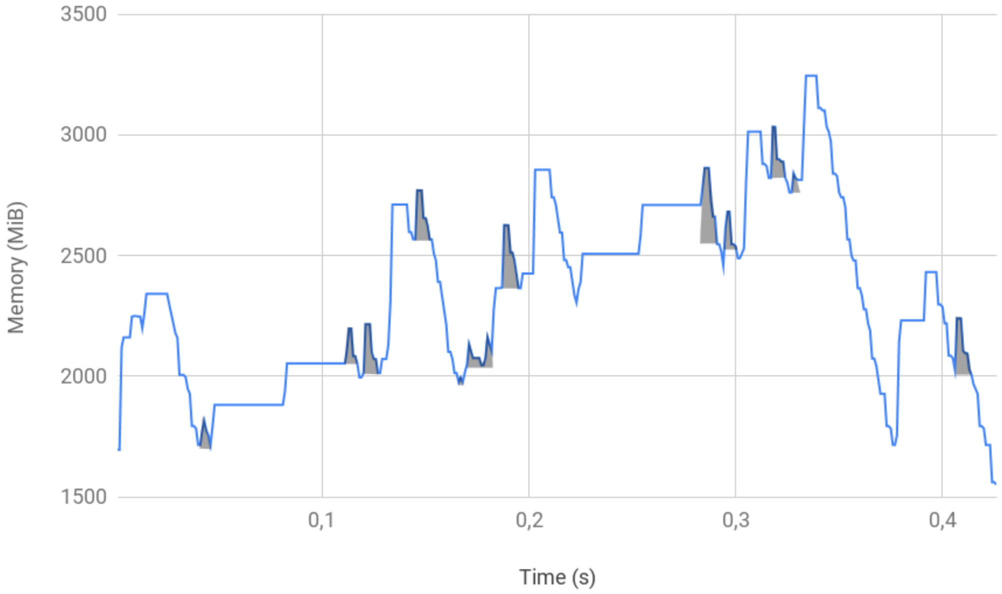


Figure 4.6: The chart highlights some of the spikes in memory usage created by the temporary objects returned by functions/operators. They can be addressed using move semantics in C++. The chart shows the same iteration as in previous charts.

Constructors and Assignment Operators

It is common practice to implement both a *move constructor* and a *move assignment operator* for a class with data members which could be reused at some point. A move constructor takes an rvalue reference to an object as its argument. It sets the member variables of the object it constructs to be empty, and then swaps them with those of the input object, effectively leaving the temporary object with no data (nothing to clean up!). The move assignment operator swaps the content of the temporary object with the content of the assignee. Since the assignee might already have data in it, the result will often be that the temporary object receives data that will get destroyed when its destructor is called.

Consider the case of `a = (b + c)` with the assumption that we have already implemented move assignment. Since the expression `(b + c)` produces a temporary object, the move assignment operator will be called, swapping the contents of the temporary with the contents of `a`. If `a` is empty, then there will be no cleanup since the object will become empty. If `a` contains any data, however, the temporary object from `(b + c)` will receive the contents of `a`, which will be destroyed when it leaves scope. This is still better than if we did not have the move assignment implemented. If the copy assignment was called instead, then the contents of `a` must first be destroyed, then the data of the temporary object is copied into `a`, and finally the temporary is destroyed. To summarize:

```
// Move constructor
CudaArray::CudaArray(CudaArray&& coll) noexcept
    : size_(coll.size_),
      dev_values_(0)
{
    std::swap(dev_values_, coll.dev_values_);
}

// Move assignment operator
CudaArray& CudaArray::operator=(CudaArray&& other) noexcept
{
    size_ = other.size_;
    std::swap(dev_values_, other.dev_values_);
    return *this;
}
```

Listing 16: Move semantics for `CudaArray`. They both set the size variable of the new `CudaArray` and swaps its data pointer with the input object's. In the move constructor, the data pointer of the input object is always set to 0, leaving it empty. In the move assignment operator the left-hand-side object might have data, so the input object `other` is potentially left with data that will be cleaned up.

- Move assignment cost:
  - Empty assignee: 0 copies, 0 cleanup
  - Non-empty assignee: 0 copies, 1 cleanup
- Copy assignment cost:
  - Empty assignee: 1 copy, 1 cleanup
  - Non-empty assignee: 1 copies, 2 cleanup

In the CUDA backend for Equelle we want to implement move semantics for our main data classes: `CollOfScalar`, `CudaMatrix`, `CudaArray` and `CollOfVector`. We start with `CudaArray` since it is the simplest of the three, and its move members are needed to implement move semantics for both `CollOfScalar` and `CollOfVector`.

Listing 16 shows how the move constructor and move assignment operator for `CudaArray` is implemented. Notice that we use the double ampersand notation (`&&`) in the parameters. In the initializer list we first copy the size variable and set the data variable `dev_values_` to `0`, and then call the `std::swap()` function to swap the pointers of the input object and the constructed object. As we have described, this will leave `coll` with a null pointer and nothing to clean up.

The move assignment operator also starts out by setting the size variable, and then it switches data pointers, leaving the temporary with the previous contents of the assignee. This will cost a maximum of 1 `cudaFree`, but 0 if the temporary is left empty.

In the implementation for `CudaMatrix`, we implement our own swap method (Listing 17). It will perform the move operation by swapping the three data pointers `csrVal_`, `csrRowPtr_`, `csrColInd_` and all the other variables. We use the swap method to implement the standard move constructor and move assignment operator for CudaMatrix in the same manner as we did for CudaArray, as can be seen in Listing 18. The move constructor will not cause any memory management. The move assignment operator will cost 3 cudaFrees if the temporary object is assigned any data but 0 if it is left empty.

```
// Swaps the variables of the caller and "other"
void CudaMatrix::swap(CudaMatrix& other) noexcept
{
    std::swap(nnz_, other.nnz_);
    std::swap(csrVal_, other.csrVal_);
    std::swap(csrColInd_, other.csrColInd_);
    std::swap(rows_, other.rows_);
    std::swap(csrRowPtr_, other.csrRowPtr_);
    std::swap(cols_, other.cols_);
    operation_ = other.operation_;
    diagonal_ = other.diagonal_;
}
```

Listing 17: Swap method for CudaMatrix, used for moving.

```
// Move constructor
CudaMatrix::CudaMatrix(CudaMatrix&& mat) noexcept
    : rows_(0),
      cols_(0),
      nnz_(0),
      csrVal_(0),
      csrRowPtr_(0),
      csrColInd_(0),
      sparseStatus_(CUSPARSE_STATUS_SUCCESS),
      cudaStatus_(cudaSuccess),
      description_(0),
      operation_(mat.operation_),
      diagonal_(mat.diagonal_)
{
    swap(mat);
    createGeneralDescription_("CudaMatrix move constructor");
}

// Move assignment operator:
CudaMatrix& CudaMatrix::operator=(CudaMatrix&& other) noexcept
{
    swap(other);
    return *this;
}
```

Listing 18: Move semantics for CudaMatrix.

In Listing 19 we have the implementations for three move constructors and the move assignment operator for CollOfScalar. Since the data members of CollOfScalar are not pointers, but rather CudaArray and CudaMatrix, we take a different approach to implementing move semantics. We must force the use of their move constructors, rather than using the swap functions. This is done by using the C++ Standard Library function std::move(), which tells the compiler to treated an object as an *rvalue* (basically a temporary object). In the code, we use the function in the initializer list of the constructors to force the move assignment operators of the member in the move assignment operator.

As before, for the two move constructors that only have rvalue references there is no memory management occuring. For the constructors where either of the arguments are lvalue references, it is more expensive:

- CollOfScalar(const CudaArray& val, CudaMatrix&& der) copies a CudaArray, causing one cudaMalloc and one cudaMemcpy.

- CollOfScalar(CudaArray&& val, const CudaMatrix& der) copies a CudaMatrix, causing three cudaMallocs and three cudaMemcpy.

The best case for operator=(CollOfScalar&& other) is when the assignee is an empty CollOfScalar and all that needs to be done is a move operation. In the worst case, however, the temporary gets assigned data to both its variables and the cost is 4 cudaFrees.

Again, to compare to the copy alternatives: The constructors that copy both members need to allocate memory for each of the objects and copy them. This amounts to 4 cudaMalloc and 4 cudaMemcpy. The copy assignment operator will need to destroy the contents of the assignee, then allocate them again, then copy the variable of the temporary and finally destroy the temporary, ending up at a cost of 8 cudaFree, 4 cudaMalloc and 4 cudaMemcpy.

In Listing 20 we see the implementations of move semantics for CollOfVector. Only a regular move constructor is implemented, in a standard way: std::move is used to force the move semantics of its data member which is a CudaArray.

We also add std::move() to certain parts of the backend code in order for it to use the newly implemented move semantics. Listing 21 shows the change we must do in the addition operator for CollOfScalar. We change CollOfScalar(val, der) to CollOfScalar(std::move(val), std::move(der)) and CollOfScalar(val) to CollOfScalar(std::move(val)).

Now that we have implemented move semantics for the constructors and assignment operators of Equelle's main data classes, we can move on to implementing it for other parts of the code, such as arithmetic operators.

```
// Move constructor
CollOfScalar::CollOfScalar(CollOfScalar&& coll) noexcept
    : val_(std::move(coll.val_)),
      der_(std::move(coll.der_)),
      autodiff_(coll.autodiff_)
{
}

// Move constructor from CudaArray and CudaMatrix.
// Both val and der are moved.
CollOfScalar::CollOfScalar(CudaArray&& val, CudaMatrix&& der) noexcept
    : val_(std::move(val)),
      der_(std::move(der)),
      autodiff_(true)
{
}

// Move constructor from CudaArray and CudaMatrix
// Only der is moved. val is copied.
CollOfScalar::CollOfScalar(const CudaArray& val, CudaMatrix&& der) noexcept
    : val_(val),
      der_(std::move(der)),
      autodiff_(true)
{
}

// Move constructor from CudaArray and CudaMatrix
// Only der is moved. val is copied.
CollOfScalar::CollOfScalar(CudaArray&& val, const CudaMatrix& der) noexcept
    : val_(std::move(val)),
      der_(der),
      autodiff_(true)
{
}

// Assignment move operator
CollOfScalar& CollOfScalar::operator=(CollOfScalar&& other) noexcept
{
    val_ = std::move(other.val_);
    autodiff_ = other.autodiff_;
    if ( autodiff_ || other.autodiff_) {
        der_ = std::move(other.der_);
    }
    return *this;
}
```

Listing 19: Move semantics for CollOfScalar.

Operator Overloading

In order to implement move semantics for a function or an overloaded operator, at least on of the inputs must be of the same type as the output. Using this knowledge should let us easily identify targets for move semantics.

```
// Move constructor
CollOfVector::CollOfVector(CollOfVector&& coll)
    : elements_(std::move(coll.elements_)),
      dim_(coll.dim_),
      vector_setup_(coll.numVectors())
{
}
```

Listing 20: Move semantics for `CollOfVector`.

```
CollOfScalar equelleCUDA::operator+ (const CollOfScalar& lhs,
                                     const CollOfScalar& rhs)
{
    //CudaArray val = lhs.val_ + rhs.val_;
    CudaArray val = lhs.val_ + rhs.val_;
    if (lhs.autodiff_ || rhs.autodiff_) {
        CudaMatrix der = lhs.der_ + rhs.der_;
        //return CollOfScalar(val, der);
        return CollOfScalar(std::move(val), std::move(der));
    }
    //return CollOfScalar(val);
    return CollOfScalar(std::move(val));
}
```

Listing 21: `std::move` insertion in `operator+` return values.

Take for instance the `itrans` expression:

```
itrans = k * |ifaces| / |Centroid(first) - Centroid(second)|
```

Three of the operations in the expression have inputs and outputs of the same type. For the `itrans` computation, this applies to:

- `Scalar * CollOfScalar`
- `CollOfVector - CollOfVector`
- `CollOfScalar / CollOfScalar`

The first is `k * |ifaces|`, which multiplies a `Scalar` by a `CollOfScalar` and returns a `CollOfScalar`. The temporary object from `|ifaces|` can be reused. The second operation is the division operator, which takes two `CollOfScalars` as its arguments. Since both sides in this case are temporaries, we can reuse either of them. Thirdly, the minus operator for `CollOfVector` has the same type in its input and output.

When implementing move arithmetic operators, the typical case is to perform the operation in-place on the temporary we want to reuse, and return it using the move constructor (e.g. `return CudaArray(std:move());`). Move division is implemented for `CudaArray` in Listing 22.

```
CudaArray equelleCUDA::operator/(CudaArray&& lhs, CudaArray&& rhs) {
    kernelSetup s = lhs.setup();
    division_kernel<<<s.grid, s.block>>>(lhs.data(), rhs.data(), lhs.size());
    return CudaArray(std::move(lhs));
}
```

Listing 22: Move division operator for `CudaArray`. It performs division in-place on `lhs` before its data is moved into the returned object.

For the residual computation, we have 3 operators that are compatible with move semantics:

- `CollOfScalar * CollOfScalar`
- `Scalar * CollOfScalar`
- `Scalar / CollOfScalar`

We implement these operators in the backend and will see the results in the next section.

4.2.2 Performance Results

We now measure the performance improvement after implementing move semantics, starting with only the move constructors and move assignment operators implemented.

In Figure 4.7 we see the memory usage for one iteration, after move semantics for the constructors and the assignment operators are implemented. Notice that all the spikes marked in Figure 4.6 are now gone. We also see that the new peak in memory usage is lower, now at 3,013 MiB, down from 3,285 before we implemented move semantics. Calls to `cudaFree` have been reduced by 30 percent (2,990 less), down to 6,991 which now accounts for 10.5 seconds. This is a reduction of 4 seconds, or 27 percent. We can also observe that the total execution time of the simulation has gone down from 27 seconds to 21.4, which is an improvement of approximately 20 percent.

Figure 4.8 shows the memory usage after we have added move semantics for the arithmetic operators. Inspection of the data shows that the memory usage in general is 20 MiB lower, which is the size of a `CollOfScalar` on the `AllCells` domain. The first multiplication operator's memory usage has gone down by 280 MiB and both the second multiplication's and the addition's memory usage has gone down by 262 MiB. `cudaFree` is now called 6,423 times, which accounts for 9.84 seconds. The current execution time is 20.7 seconds. Our implementation has improved both the time and the memory usage of the backend. In the next section we will look further into the multiplication operator to make one more improvement.

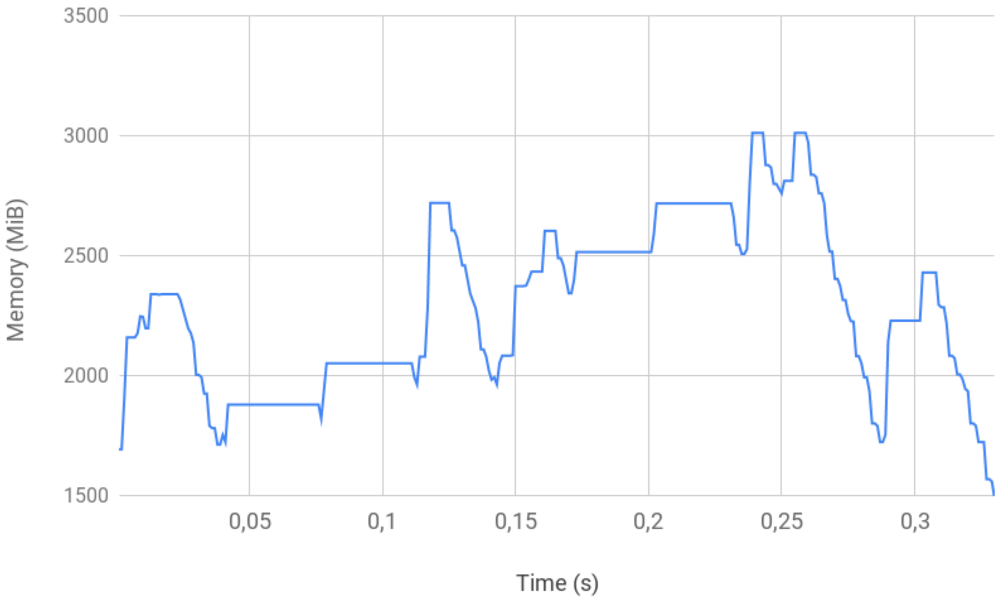


Figure 4.7: Memory usage after implementing move semantics for constructors and assignment operators, for one iteration. Notice that all the small spikes highlighted in Figure 4.6 are gone. The new peak memory usage is 3,013 MiB, down from 3,285.

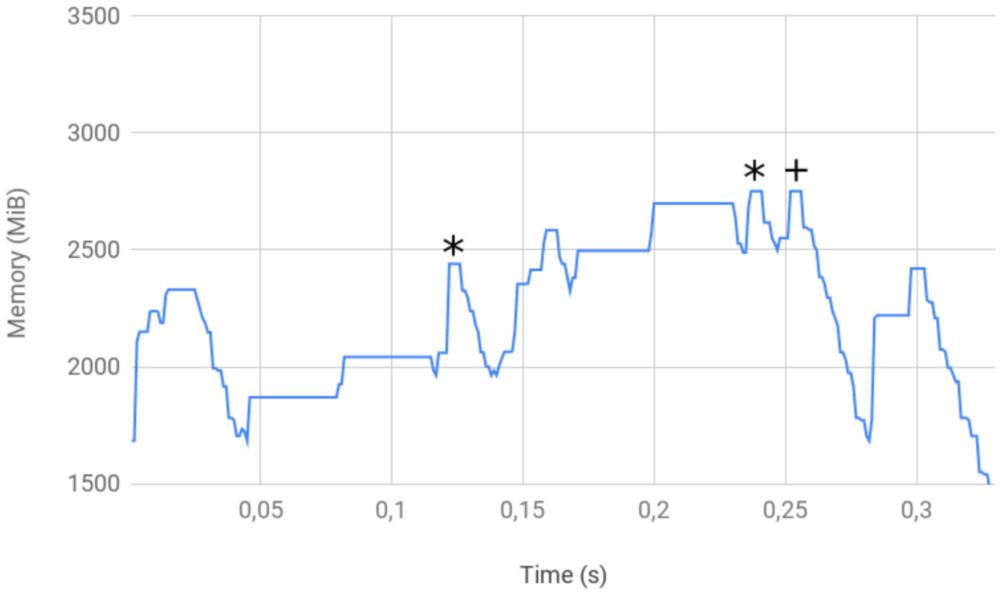


Figure 4.8: Memory usage after implementing move semantics for arithmetic operators. We see that there are significant improvements for the multiplication and addition operators, which are annotated.

4.3 Optimizing CollOfScalar Multiplication

In this section we will optimize the `CollOfScalar` multiplication by reducing its memory usage footprint. The first multiplication creates a spike of 380 MiB and the second a spike of 262 MiB.

The first of the two occurs when computing the interior flux, when the negative transmissibility is multiplied by the gradient of `u`:

```
-itrans * Gradient(u)
```

And the second one occurs in the residual computation in the following operation:

```
(dt / (cv * vol)) * Divergence(fluxes)
```

The backend code can be seen in Listing 23. On the lines 4 to 6 we see where the memory usage goes up. We see that the values of the two `CollOfScalar` are copied into two `CudaMatrix` objects before being used in diagonal multiplications with the derivatives of the collections, and adding the results together. Each of the diagonal matrices will have a size which is equal to the number of elements it receives from the value array, in addition to having the row pointers and the column index arrays. Since they are diagonal matrices, they do not need to store the row pointer and column indices, since they will both contain sequential numbers up to its size.

```
1  CollOfScalar equelleCUDA::operator*(const CollOfScalar& lhs, const CollOfScalar& rhs)
2  {
3      CudaArray val = lhs.val_ * rhs.val_;
4      if ( lhs.autodiff_ || rhs.autodiff_ ) {
5          CudaMatrix diag_u(lhs.val_);
6          CudaMatrix diag_v(rhs.val_);
7          CudaMatrix der = diag_v*lhs.der_ + diag_u*rhs.der_;
8          return CollOfScalar(std::move(val), std::move(der));
9      }
10     return CollOfScalar(std::move(val));
11 }
```

Listing 23: Unoptimized `CollOfScalar` multiplication code from the backend.

4.3.1 Design and Implementation

Diagonal multiplication in the CUDA backend is implemented using a member function of `CudaMatrix` which takes a `CudaMatrix` as argument. When using it, the caller needs to be a diagonal matrix. We choose a simple implementation where we add a modified version, where the argument is a `CudaArray` representing a diagonal matrix. By using the `CudaArray` directly instead of constructing a wrapper `CudaMatrix`, we avoid generating and storing column indices and row pointers. For collections that are on the `InteriorFaces` domain when we use our grid which contains 7,440,000 faces, this equates to saving 56.76 MiB per matrix. This means that we should see an improvement equal to double this number in addition to temporary objects involved in the calculation.

```
CudaMatrix CudaMatrix::diagonalMultiply(const CudaArray& lhs_diag_mat) const
{
    // Check if this is empty
    // Check if this is diagonal
    CudaMatrix out = *this;
    kernelSetup s(this->rows_);
    wrapCudaMatrix::diagMult_kernel<<<s.grid, s.block>>>(out.csrVal_,
                                                         out.csrRowPtr_,
                                                         lhs_diag_mat.data(),
                                                         this->rows_);
    return CudaMatrix(std::move(out));
}
```

Listing 24: Diagonal matrix multiply function without CudaMatrix wrapper. This is used in the improved CollOfScalar multiplication in Section 4.3. The parameter CudaArray represents a diagonal matrix, which is multiplied from the left.

The implementation can be seen in Listing 24. As we can see, the caller matrix is copied, and then diagMult_kernel which performs the diagonal multiplication is called. In Listing 25 we see the CollOfScalar multiplication operator which uses the new function.

```
CollOfScalar equelleCUDA::operator*(const CollOfScalar& lhs, const CollOfScalar& rhs)
{
    CudaArray val = lhs.val_ * rhs.val_;
    if ( lhs.autodiff_ || rhs.autodiff_ ) {
        CudaMatrix der = lhs.der_.diagonalMultiply(rhs.val_)
                         + rhs.der_.diagonalMultiply(lhs.val_);
        return CollOfScalar(std::move(val), std::move(der));
    }
    return CollOfScalar(std::move(val));
}
```

Listing 25: Improved CollOfScalar multiplication code from the backend. The diagonalMultiply() function treats a CudaArray as a diagonal matrix.

4.3.2 Performance Results

The results in memory usage after we have improved the CollOfScalar multiplication can be seen in Figure 4.9. The multiplications that are annotated have both improved, with the first one now using 236 MiB less memory and the second using 60 MiB less memory. We also see that the two-norm operator now has an extra spike in memory usage, which is most likely a temporary object. The execution time is now 21.514 seconds and the number of calls to cudaFree is 5583 (57.24%, 10.1 seconds). The number of cudaMemcpy calls is 2311. cudaLaunch is called 4128 times. In the next section we will improve the two-norm operator, which is bottlenecked by matrix multiplication, which it does not need to perform.

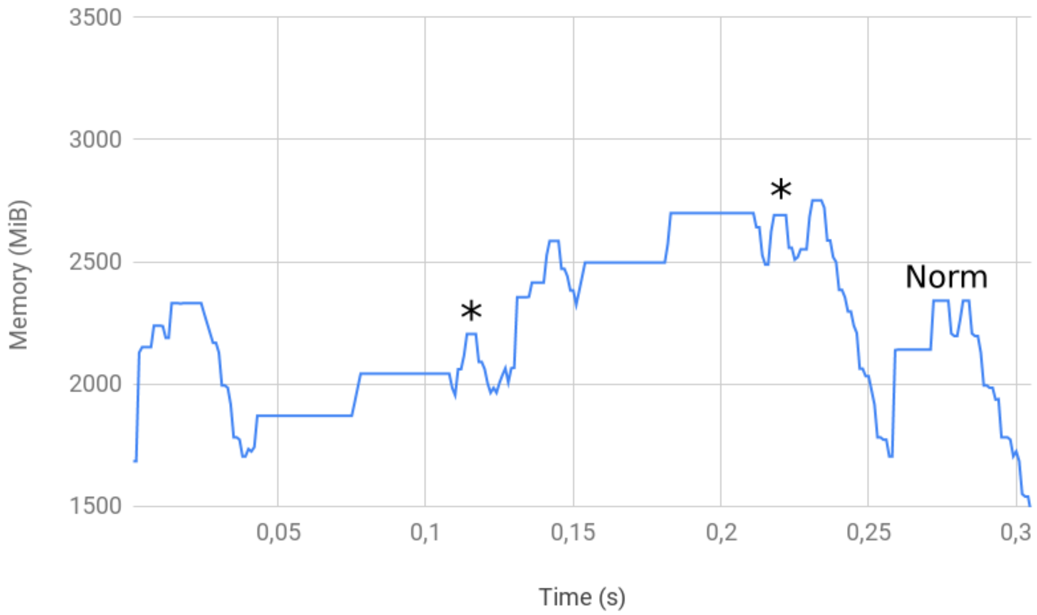


Figure 4.9: Memory usage after optimizing the CollOfScalar multiplication. The first multiplication has been significantly improved, using 236 MiB less. The second multiplication uses 60 MiB less memory. Thirdly we see that the norm operator now produces a temporary object, indicated by a spike like the ones we have seen in previous charts.

4.4 Optimizing TwoNorm

The two-norm operator is a scalar value which is calculated as the sum of all the values squared. In iterative methods such as the Newton's method which Equelle uses, it is used to determine whether a residual of the current solution is small enough. In this section we will have a look at how to optimize it, as it is far more expensive than it needs to be. When calculating the norm of a `CollOfScalar`, it is multiplied by itself, both its derivative and its values. Before the optimization, the execution time for the `twoNorm` function is 2.35 seconds, accounting for 10.26% of the total execution time.

4.4.1 Design and Implementation

We use a straight-forward approach where we implement a member function which returns a `CollOfScalar`, containing the squares of the caller's values. The implementation of the square function can be seen in Listing 26, along with its use in the `twoNorm` function.

4.4.2 Performance Results

The results we get after improving the `twoNorm` calculation can be seen in Figure 4.10. The execution time of the program has been reduced from 21.51 to 19.47 seconds and twoNorm has been reduced from 2.35 seconds to approximately 0.2 seconds.

```
CollOfScalar CollOfScalar::squareVals() const
{
    return CollOfScalar(std::move(val_*val_));
}

Scalar EquelleRuntimeCUDA::twoNorm(const CollOfScalar& vals) const
{
    return std::sqrt( sumReduce(vals.squareVals()) );
}
```

Listing 26: C++ code which implements the square function we have added and the twoNorm function where it is used.

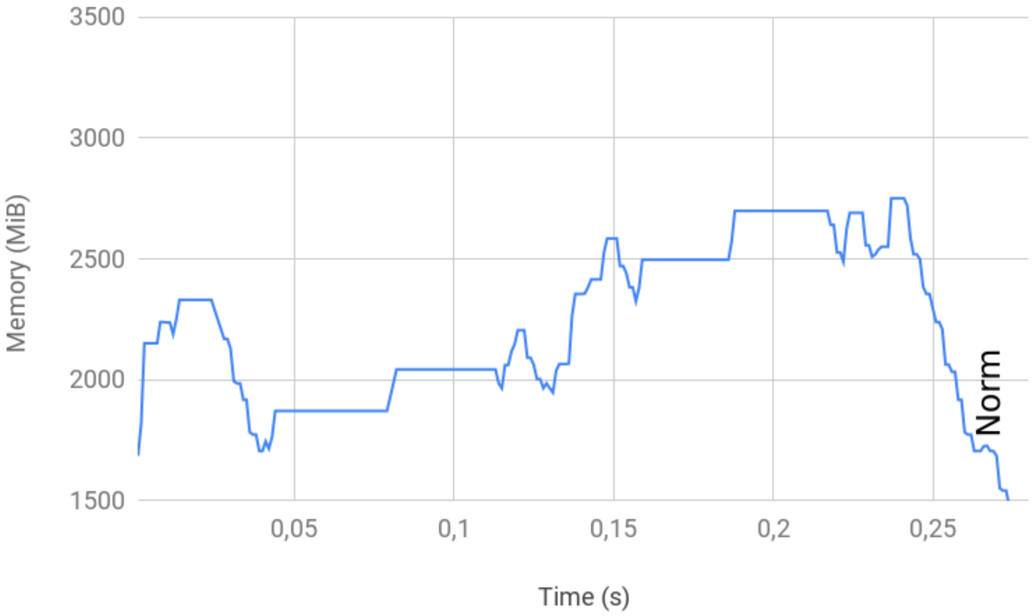


Figure 4.10: Memory usage after improving the two-norm calculation.

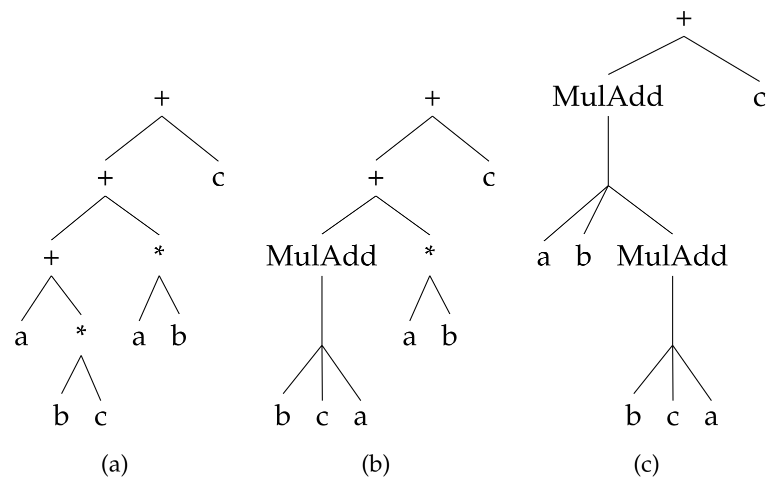


Figure 4.11: Substitution of $a + b * c$ and $a * b + c$ in an AST sub-tree representing $((a + b * c) + a * b) + c$. The algorithm runs recursively through the AST in a pre-order fashion and finds two instances of $a + b * c$. The result is $MulAdd(a, b, MulAdd(b, c, a)) + c$.

4.5 Abstract Syntax Tree Rewriting

A common compiler optimization technique is to rewrite the intermediate representation (IR) of a program. In this section we discuss an AST rewriter which we have implemented in the Equelle compiler. The rewriter performs pattern matching on sub-trees which are modified or replaced based on rules that are specified in its implementation. In our example pattern match scheme, we fuse the multiplication and addition operators to make a multiply-add operation. We expect no big performance improvement from this implementation at this point, as we have established that memory operations limit the performance too much. The implementation will, however, be applicable in the future, and across backends, as it is applied before code generation. We might say that it is a first step towards making the Equelle compiler into an *optimizing compiler*.

4.5.1 Design and Implementation

The rewriting step happens after the semantic analysis[1] has been performed, and before the code generation. Figure 4.11 gives an example of what a rewriting of a multiplication and an addition looks like.

4.5.2 Results

As previously mentioned, there will be no discussion about the performance results as this section only demonstrates the AST rewriting itself. In the implicit heat equation simulation, the rewriter finds one multiply-add in the residual computation, which generates the following C++ code:

[1] Rewriting occurs after the program has been proven to be correct.

```
// Pattern matching for multiply-add
if (current->op() == Add) {
    // Dynamic cast to BinaryOpNode* works as a type check.
    // If it isn't a BinaryOpNode* then the result is nullptr
    auto* lhs = dynamic_cast<BinaryOpNode*>(current->getChild(0));
    auto* rhs = dynamic_cast<BinaryOpNode*>(current->getChild(1));
    // Accounts for a*b+c and a+b*c
    BinaryOpNode* mulOpNode;
    int addOpNodeIndex = 0;
    if (lhs != nullptr && lhs->op() == Multiply) {
        mulOpNode = lhs;
        // child index of rhs
        addOpNodeIndex = 1;
    } else
    if (rhs != nullptr && rhs->op() == Multiply) {
        mulOpNode = rhs;
        //child index of lhs
        addOpNodeIndex = 0;
    } else {
        return;
    }

    auto replacementNode =
        new MultiplyAddNode(dynamic_cast<ExpressionNode*>(mulOpNode->getChild(0)),
                    dynamic_cast<ExpressionNode*>(mulOpNode->getChild(1)),
                    dynamic_cast<ExpressionNode*>(current->getChild(addOpNodeIndex)));
    replaceNode(childIndex, current, replacementNode);
    deleteNode(mulOpNode);
}
```

Listing 27: Code from the AST rewriter that implements fusion of multiplication and addition.

```
const CollOfScalar residual =
        er.multiplyAdd((dt / (cv * vol)), er.divergence(fluxes), (u - u0));
```

Listing 28: Generated C++ code after operator fusion using the AST rewriter.

4.6 Explicit Zero Compression

The ternary if operator is a common feature in many modern programming languages and is often called an inline if-statement. It takes three arguments: one condition and two expressions. The two expression parameters are used to construct an output value of the same type, and therefore needs to be compatible. In the case where the expressions are two collections, they must contain the same basic value type, be of the same dimensions and be on the same domain (if any). The syntax is as follows[2]:

`(value > 0) ? iftrue : iffalse`

In the example, `value > 0` is the condition that generates the predicate. The result

[2] The parentheses can be omitted, but are added for readability.

$$\begin{pmatrix} 0 & \times & \times \\ \times & 1 & \times \\ \times & \times & 0 \end{pmatrix} \times \begin{pmatrix} 1 & \times & \times \\ 3 & \times & 5 \\ \times & 7 & 9 \end{pmatrix} = \begin{pmatrix} 0 & \times & \times \\ 3 & \times & 5 \\ \times & 0 & 0 \end{pmatrix} \tag{4.1}$$

$$\begin{pmatrix} 1 & \times & \times \\ \times & 0 & \times \\ \times & \times & 1 \end{pmatrix} \times \begin{pmatrix} \times & 2 & \times \\ 4 & 6 & \times \\ 8 & \times & \times \end{pmatrix} = \begin{pmatrix} \times & 2 & \times \\ 0 & 0 & \times \\ 8 & \times & \times \end{pmatrix} \tag{4.2}$$

$$\begin{pmatrix} 0 & \times & \times \\ 3 & \times & 5 \\ \times & 0 & 0 \end{pmatrix} + \begin{pmatrix} \times & 2 & \times \\ 0 & 0 & \times \\ 8 & \times & \times \end{pmatrix} = \begin{pmatrix} 0 & 2 & \times \\ 3 & 0 & 5 \\ 8 & 0 & 0 \end{pmatrix} \tag{4.3}$$

Figure 4.12: An example of the three steps involved in the ternary if operator, demonstrating how zeroes get stored explicitly. The first two are multiplications by a predicate stored in diagonal matrices. The last step adds the two results together to yield the final derivative matrix.

is a `Collection Of Bool` of the same size as the `value` collection, with 1's in the positions where the condition is true, and 0's where it is false. For insance, if `value` contains `{0,1,4,3,-3}`, the predicate would be `{0,1,1,1,0}`. `(value > 0) ? 2 : 3` is also valid and returns `{3,2,2,2,3}`.

When performing the ternary if on the values of collections, it simply evaluates the predicate in an element-wise manner and picks values from the `iftrue` collection for predicate values that are true (or a 1), and picks values from the `iffalse` collection where the predicate is false (or a 0). Evaluating the predicate `{0,1,0,1,0}` with `{2,2,2,2,2}` as `iftrue` and `{3,3,3,3,3}` as `iffalse` would yield `{3,2,3,2,3}`.

In the case of the derivatives, the operation is performed using three steps. In the first step the predicate array is turned into a diagonal matrix and multiplied with the `iftrue` collection's derivatives.

In step 2, the values of the predicate are flipped so that 0's become 1's and 1's become 0's. Using the flipped predicate, we perform the same operation as in the previous step on the `iffalse` collection. The final step is to add the two matrices together.

A problem with this approach is that zeroes are stored in the non-zero structure of the sparse representation, resulting in excessive storage and computational overhead. Consider the following example:

Let us say we have two collections containing 3 scalars each, and we have generated the predicate `{0,1,0}`. The `iftrue` derivative matrix contains the CSR values `{1,3,5,7,9}` and `iffalse`'s matrix contains `{2,4,6,8}`. The equations 4.1, 4.2, and 4.3 show the three steps, demonstrating how the zeroes get stored.

4.6.1 Design and Implementation

In order to avoid the storage of zeroes in our sparse matrices, we choose to remove them in a consecutive step. The cuSPARSE library has a compression procedure which is designed for this purpose. The CSR-to-CSR compression takes two steps. First `cusparseDnnz_compress` is called to find the sparsity pattern of the resulting matrix, including an array containing the number of nnz per row. The information is used to allocate memory for the new matrix, and then `cusparseDcsr2csr_compress` is called to perform the compression. The implementation is added as a function in the `CudaMatrix` class and returns a compressed copy of the matrix when called. The code is located in Listing 29. The line `der = der.removeZeroes();` is added in the `wrapEquelleRuntimeCUDA::trinaryIfWrapper` function.

4.6.2 Results

After implementing the zero compression function, we run the `autoDiff` test again. The result is that the test program fails on an earlier test than before, saying "`Wrong number of nnz: 49267 should be 60632`". Notice that before the implementation, the `CudaMatrix` contained too many non-zeroes, but now it contains fewer than the CPU matrix which it is compared to. Investigating further with console output shows that the `CudaMatrix` now has the correct amount of non-zeroes, but the matrix which is from the CPU backend contains explicit zeroes. This means that both the CPU backend and the CUDA backend stores explicit zeroes for different operations.

4.7 Summary of Results

In this chapter we have made improvements to the Equelle compiler and CUDA backend, with a focus on the implicit heat equation simulation. We used a grid of 2.5 megacells and ran it for 20 timesteps.

First we optimized the `Extend` operator, which we had previously found to be the bottleneck. The execution time of the operator was reduced from 18.77 seconds to 1.45 seconds. The total execution time of the simulation was reduced from 43.19 to 27 seconds.

Furthermore, we have implemented move semantics to address the cost of temporary objects. After implementing move semantics for constructors, assignment operators and arithmetic operators, we saw that the execution time was reduced from 27 seconds to less than 21 seconds.

Next, we optimized `CollOfScalar` multiplication, since its memory usage was not very efficient. The reason for the large increase in memory usage when evaluating these multiplications, was that `CudaArrays` were being wrapped into `CudaMatrix` objects representing diagonal matrices, generating both row pointers and column indices. After our optimization, we did not see a significant improvement in execution time. However, the memory usage for the two multiplications went down by 236 MiB and 60 MiB.

Our fourth optimization was an improvement of the `twoNorm` operator, which is performed once for each Newton iteration. The two-norm performed an unnecessary

```cpp
CudaMatrix CudaMatrix::removeZeroes() const
{
  if (isEmpty()){
    return CudaMatrix();
  }
  CudaMatrix out;
  out.rows_ = rows_;
  out.cols_ = cols_;

  double tol = std::numeric_limits<double>::epsilon();

  int *nnzPerRow;
  int *newNNZ;
  cudaStatus_ = cudaMallocManaged( &nnzPerRow, sizeof(int) * rows_ );
  checkError_("Failed to allocate memory for nnzPerRow in CudaMatrix::removeZeroes()");
  cudaStatus_ = cudaMallocManaged( &newNNZ, sizeof(int));
  checkError_("Failed to allocate memory for newNNZ in CudaMatrix::removeZeroes()");
  memset( nnzPerRow, 0, sizeof(int) * rows_ );

  // Get nnz information about the the compressed matrix
  cusparseDnnz_compress(CUSPARSE, rows_, description_, csrVal_,
                        csrRowPtr_, nnzPerRow,
                        newNNZ, tol);
  out.nnz_ = *newNNZ;

  // Allocate memory for the compressed matrix
  if (out.nnz_ > 0) {
      cudaMalloc((void**)&out.csrRowPtr_, sizeof(int)*(rows_+1));
      cudaMalloc((void**)&out.csrColInd_, sizeof(int)*out.nnz_);
      cudaMalloc((void**)&out.csrVal_, sizeof(double)*out.nnz_);
  } else {
      cudaFree(nnzPerRow);
      cudaFree(newNNZ);
      return CudaMatrix();
  }

  // Compress the matrix
  cusparseDcsr2csr_compress( CUSPARSE, rows_, cols_, description_, csrVal_,
                             csrColInd_, csrRowPtr_,
                             nnz_,  nnzPerRow,
                             out.csrVal_, out.csrColInd_,
                             out.csrRowPtr_, tol);
  cudaFree(nnzPerRow);
  cudaFree(newNNZ);
  return out;
}
```

Listing 29: Function for compressing a sparse matrix. It uses the cuSPARSE procedure `cusparseDcsr2csr_compress` to remove zeroes in a `CudaMatrix`.

matrix multiplication. After our improvement we saw a decrease in both execution time and memory usage. The total time was 19.47 seconds. The memory improvement (before and after) can be seen in Figure 4.9 and Figure 4.10.

The fifth implementation was an abstract syntax tree rewriter, meant for improving or modifying the structure of Equelle programs. We demonstrated how to implement operator fusion of multiplication and addition. We did not implement the fused operator in the backend, as the intent was to give an example of how to implement a rewriting scheme.

The last implementation was a function for removing explicit zeroes in sparse matrices, which we had observed in our initial analysis. The implementation was tested on the `autoDiff` test in the Equelle test suite, and showed that the compression was successful. However, the test suite failed at another test, because the CPU matrix contained explicit zeroes.

4.7.1 Final Profiling

As a conclusion to our analysis and optimization work, we now profile the implicit heat equation program once more to measure the total impact of our implementations. We compare our results to those of Chapter 3. A summary of the metrics can be seen in Tables 4.1 and 4.2.

First we measure the number of cells that the backend can process. Before our optimizations, we found that the backend processed 1.18 megacells per second for a grid of 3.5 megacells, running for 20 timesteps. When we now run the same benchmark it shows that the throughput is 3.77 megacells per second, which is a 3.19x speedup.

Figure 4.13 shows the new distribution reported by the CPU trace. What we see is that the gradient and divergence operators now are the main bottlenecks, and that the extend operator which used to be the bottleneck only takes 3.4 percent of the total execution time.

In order to push the system, we increase the grid to the largest it can handle. Before our optimizations, it was 3.65 megacells. The new maximum is 4.8 megacells. The performance when running for 100 timesteps is 3.7 megacells per second. Since this is approximately the same performance as for the former grid, it suggests that it is the peak performance.

The chart in Figure 4.14 shows the memory usage for one Newton iteration, before and after our optimizations. The red line is after our optimizations and the blue line is before. Notice that the iteration takes more than 7.5 milliseconds to complete before the optimizations and it only takes slightly more than 2.5 milliseconds after. Also notice the improvement in memory usage, where the peak has gone down by 534 MiB.

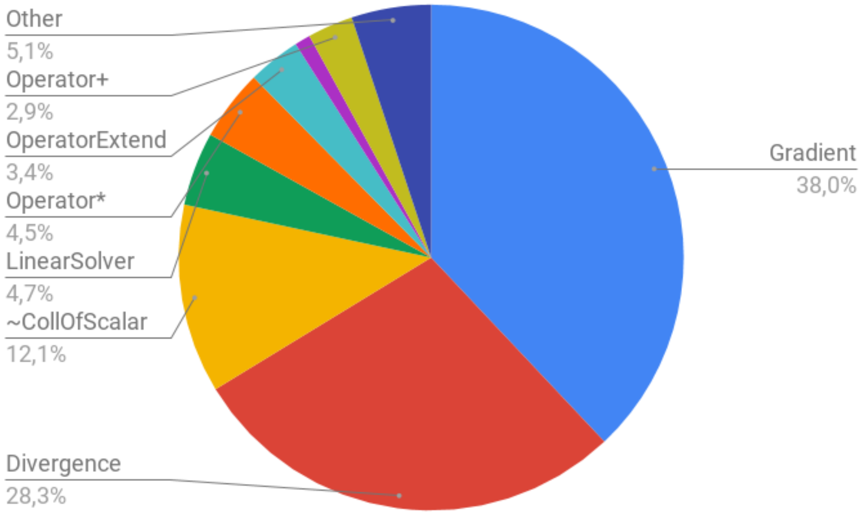


Figure 4.13: Time percentages reported by the CPU trace after optimizations. The extend operator which used to be the bottleneck now only takes 3.4 percent of the execution time. The gradient and divergence operators are clear bottlenecks.

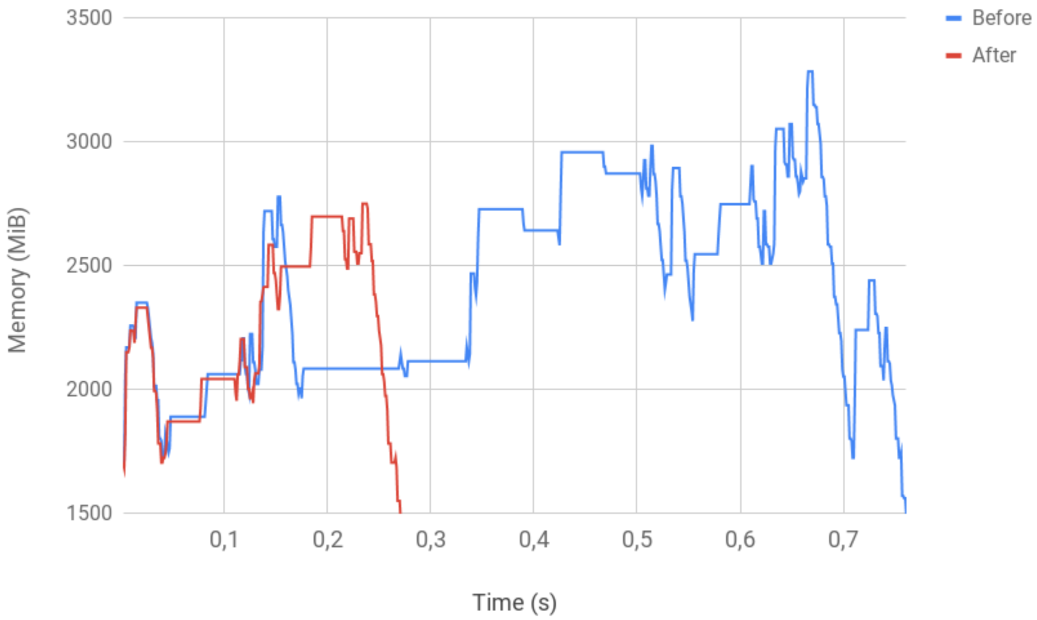


Figure 4.14: Memory usage for one Newton iteration, before and after our optimizations.

Metric	**Before**	**After**	**Improvement**
Cell throughput (megacells)	1.18	3.77	3.19x
Newton iteration time	7.5ms	2.5ms	3x
Memory usage	3,285 MiB	2,751 MiB	534 MiB
Max grid size (megacells)	3.65	4.8	1.315x
Timestep time	1.9s	0.65s	2.9x

Table 4.1: The table shows an overview of various improvements from before and after our optimizations. Cell throughput is measured using a grid of 3.5 megacells for 20 timesteps. The other entries are measured with a grid of 2.5 megacells running for 20 timesteps.

Operation	**Before (s)**	**After (s)**	**Reduction (%)**
`cudaFree`	23.87	8.24	34.5
`memcpyAsync`	11.46	3.76	32.81
`cudaMalloc`	3.33	1.55	53.4
`DtoD` copy	3.43	0.98	71.4
Total	41.46	14.52	-

Table 4.2: The table shows the reduction in time spent performing memory operations. The numbers are measured using a grid of 2.5 megacells, running for 20 timesteps.

Chapter 5

Conclusion

In this thesis, we have analyzed and improved the compiler and CUDA backend of Equelle, a DSL for solving partial differential equations (PDEs) using the finite volume method (FVM). We have also given an introduction to DSLs, including tools and frameworks for developing them, as well as a survey of DSLs for high performance computing.

In our performance analysis in Chapter 3, we found that the implicit heat equation simulation was bottlenecked by the `Extend` operator, which took over 40 percent of the execution time. We also found that the performance of the backend is limited by operations related to memory management on the device, mostly because of the frequent allocation and deallocation of memory in temporary objects. Furthermore, operators such as the `CollOfScalar` multiplication and addition caused spikes in memory usage. We also found that when using CUDA 9, the cuSPARSE matrix addition (`csrgeam`) is 8 times slower than in CUDA 8. Running the `autoDiff` test from the test suite showed that the ternary if operator causes zeroes to get stored in `CudaMatrix` values. Lastly, we performed additional analysis which showed that the GPU solvers do not provide sufficient solutions to the two-phase flow simulation, and that there are race conditions in the `csrgemm` that cause the results to vary across runs.

In Chapter 4 we implemented optimizations that improved the performance of the CUDA backend, addressing some of the previous findings. First, we optimized the `Extend` operator, which was sped up by a factor of 12 to 12.85x. Next we implemented move semantics to reduce the memory management cost of temporary objects, which yielded a speedup of approximately 1.25x, as well as improving the overall memory usage. In our third optimization we improved the memory usage of the `CollOfScalar` multiplication, which lowered the peak memory usage. Next, we optimized the `twoNorm` operator, which improved both its memory usage and execution time. Its execution time was reduced from 2.35 seconds to 0.2 seconds, which is a speedup of 11.75x. Our fifth implementation is an AST rewriter, which allows us to modify subtrees of the AST. The rewriter will let us experiment with transformations that can optimize our program, for instance by changing operators or removing unneeded computations. In our last implementation we added functionality for removing explicit zeroes from the value array of a `CudaMatrix`. The final profiling showed that the cell throughput for the implicit heat equation, which is a good metric for the overall performance, has increased by a factor of 3.19x[1]. Furthermore, the largest grid we can run before the memory flows over is over 30 percent larger.

5.1 Future Work

In this section we make suggestions for future development of the Equelle CUDA backend, based on our findings in Chapters 3 and 4. The list is in prioritized order, keeping the work-to-benefit ratio in mind.

After our optimizations, the gradient and the divergence operators are the most time consuming parts of the code. Together they account for over 60 percent of the application execution time, most of which are spent performing matrix multiplication using cuSPARSE `csrgemmNnz and csrgemm`. Also, memory management overhead is still performance limiting. The frequent allocation, deallocation and copying of memory on the device are time consuming, and operations such as `cudaFree`, `cudaMalloc` and `cudaMemcpy` block asynchronous execution, and should be a consideration when looking into future development.

- **Implement additional parallel linear solvers and preconditioners.** This should arguably be the first priority in further development. Whether the issues when running the two-phase flow simulation[2] were due to poor conditioning of the linear system or poor performance from the solvers, or a mix of the two we do not know. This will need additional investigation. The first step should be to identify an implementation of a preconditioner/solver combination that have been proven to work well for the equations in question. The implicit two-phase simulation is a good candidate for testing, as that is the one we have been struggling with, and it is much more complex than the implicit heat equation simulation, and so it is a more interesting program to optimize. The CUDA toolkit supplies a code example that implements the BiCGStab solver with Incomplete Lower-Upper (ILU for short) factorization for preconditioning. Adapting the code to fit into the Equelle backend would make a good starting point to see if the results when solving the two-phase flow example are comparable to those of the CPU.

- **Memory management in the CUDA backend.** As we have observed through our profiling in our initial assessment (Chapter 3) and after our optimizations/experiments (Chapter 4), the memory management in Equelle is by far the main performance issue. Calls to `cudaMalloc` and `cudaFree` are not only expensive calls in themselves, but are also blocking, inhibiting asynchronous execution and efficient concurrency. Solving this might need a redesign of the backend.

- **Alternatives to cuSPARSE.** cuSPARSE has proven to be both unreliable and costly when used in the Equelle compiler. In addition to this, Nvidia seems to be unresponsive even after bugs are filed regarding the issues we have experienced,

[1] Measured using a grid of 3.5 megacells running for 20 timesteps.

[2] Two-phase issues are described in Section 3.5.2.

including race conditions[3] and slow routines[4]. The bug report that we submitted for the slow `csrgeam` routine is not publicly available, but we have made a post in the Nvidia developer forum where Nvidia confirms that it is most likely a bug. Looking for and comparing alternatives that are more efficient and that follow a more transparent release cycle should be a priority and can be a worthwhile investment for further development. Looking for open source alternatives that let us debug in more detail, should bugs appear, should be a priority. It is planned that *taco* (Tensor Algebra Compiler), will receive GPU support [22]. This should be a very interesting alternative. Concludingly, it should be noted that CUDA 10 has since been released, and we have not verified that the issues we encountered remain.

- **Explore more IR/AST transformations.** Implementing more compiler optimizations before the code generation step, for instance by extending the AST rewriter (Section 4.5) is interesting for further development. It is necessary to transform programs in order to implement both global and local optimizations. One such optimization is the reordering of calculations. In many cases the order in which the operations are performed, does not affect the result, but it can have a significant impact on the performance. For instance, consider the expression `u * u0 * u1`, where `u` has a derivative and `u0` and `u1` does not. With this ordering, the calculation will involve 2 matrix multiplications, since the derivative of `u` first is multiplied by the values in `u0`, where the result has a derivative, which is then multiplied by `u1`, causing another matrix multiplication. If we evaluate `u0 * u1` first, however, only one matrix multiplication will be performed since the first calculation does not involve any derivatives.

- **Implement NewtonSolveSystem in the CUDA backend.** `NewtonSolveSystem` is an extension of `NewtonSolve` and is currently not supported in the CUDA backend. Doing this has not been a priority in this thesis. Doing so would make a great addition to the language, and might expose more interesting themes to look into. Implementing solvers and preconditioners should, however, be a priority before this.

- **Implement code generation of CUDA kernels for arbitrary expressions.** The backend still calls a large number of small kernels, which is sub-optimal for several reasons. By moving larger portions of the program into large kernels, the backend will benefit from the aggressive optimizations that the CUDA compiler (nvcc) performs. One thing to keep in mind is that sparse matrix operations in most cases[5] can not be included in the generated kernels, as cuSPARSE can not be called from within a kernel.

- **Exploit knowledge about the sparsity patterns of matrices.** We know that the sparsity patterns of the derivatives will not change between Newton iterations. For this reason, we do not need to recalculate the sparsity pattern for

[3] Race conditions in `csrgemm` when we use small Equelle grids.

[4] `csrgeam` is 8 times slower in CUDA 9 due to what seems to be a bug (confirmed by Nvidia in our forum post).

[5] Sparse matrix multiplication can be included when the left-hand-side matrix is diagonal.

every step, using cuSPARSE subroutines such as `csrgeamNnz` and `csrgemmNnz`. Since `csrgemmNnz` is about as expensive as `csrgemm`, an implemention of this optimization might halve the cost of matrix multiplication when applicable. For instance, this is applicable for the gradient and divergence calculations, which are bottlenecks in the implicit heat equation.

Appendix A

Complete Code Samples

A.1 Implicit Heat Equation

```
# Heat conduction with Diriclet boundary conditions.

# This example is intended to show how a relatively simple
# model can be implemented in Equelle. It shows how to use
# units properly, how to write functions, and how to solve
# implicit problems. It also shows how to implement general
# Dirichlet type boundary conditions.

# Heat diffusion constant.
# Default value within range given for granite:
#   http://en.wikipedia.org/wiki/List_of_thermal_conductivities
k = InputScalarWithDefault("k", 2.85) * 1 [Watt / (Meter*Kelvin)]

# Volumetric heat capacity.
# Default value corresponds to granite:
#   http://en.wikipedia.org/wiki/Volumetric_heat_capacity
cv = InputScalarWithDefault("cv", 2.17e6) * 1 [Joule / (Kelvin * Meter^3)]

# Compute interior transmissibilities.
ifaces = InteriorFaces()
first = FirstCell(ifaces)
second = SecondCell(ifaces)
itrans = k * |ifaces| / |Centroid(first) - Centroid(second)|

# Compute flux for interior faces.
computeInteriorFlux(u) = {
    -> -itrans * Gradient(u)
}

# Support for Dirichlet boundaries
dir_boundary = InputDomainSubsetOf("dir_boundary", BoundaryFaces())
dir_val = InputCollectionOfScalar("dir_val", dir_boundary) * 1 [Kelvin]

# Compute boundary transmissibilities and orientations.
bf = BoundaryFaces()
bf_cells = IsEmpty(FirstCell(bf)) ? SecondCell(bf) : FirstCell(bf)
bf_sign = IsEmpty(FirstCell(bf)) ? (-1 Extend bf) : (1 Extend bf)
btrans = k * |bf| / |Centroid(bf) - Centroid(bf_cells)|
dir_cells = bf_cells On dir_boundary
dir_sign = bf_sign On dir_boundary
dir_trans = btrans On dir_boundary

# Compute flux for boundary faces.
computeBoundaryFlux(u) = {
    # Compute flux at Dirichlet boundaries.
    u_dirbdycells = u On dir_cells
    dir_fluxes = dir_trans * dir_sign * (u_dirbdycells - dir_val)
    # Extending with zero away from Dirichlet boundaries,
    # which means assuming no-flow elsewhere.
    -> dir_fluxes Extend BoundaryFaces()
}

# Compute the residual for the heat equation.
vol = |AllCells()|
computeResidual(u, u0, dt) = {
    ifluxes = computeInteriorFlux(u)
    bfluxes = computeBoundaryFlux(u)
    # Extend both ifluxes and bfluxes to AllFaces() and add to get all fluxes.
    fluxes = (ifluxes Extend AllFaces()) + (bfluxes Extend AllFaces())
    residual = u - u0 + (dt / (cv * vol)) * Divergence(fluxes)
    -> residual
}

# u_initial is user input (u is the unknown, temperature here)
u_initial = InputCollectionOfScalar("u_initial", AllCells()) * 1 [Kelvin]

# Sequences are ordered, and not associated with the grid
# as collections are.
timesteps = InputSequenceOfScalar("timesteps") * 1 [Second]

# u0 must be declared Mutable, because we will change it
# in the For loop further down.
u0 : Mutable Collection Of Scalar On AllCells()
u0 = u_initial

# Output initial conditions
Output("u", u0)
Output("maximum of u", MaxReduce(u0))

For dt In timesteps {
    computeResidualLocal(u) = {
        -> computeResidual(u, u0, dt)
    }
    u_guess = u0
    u = NewtonSolve(computeResidualLocal, u_guess)
    Output("maximum of u", MaxReduce(u))
    u0 = u
}
```

Bibliography

[1] Nathan Bell, Steve Dalton and Nvidia Research. *cusplibrary Repository*. URL: https://github.com/cusplibrary/cusplibrary/commits (visited on 17/10/2018).

[2] Jon Bentley. 'Programming Pearls: Little Languages'. In: *Commun. ACM* 29.8 (Aug. 1986), pp. 711–721. ISSN: 0001-0782. DOI: 10.1145/6424.315691.

[3] Gilbert Louis Bernstein and Fredrik Kjolstad. 'Perspectives: Why New Programming Languages for Simulation?' In: *ACM Trans. Graph.* 35.2 (May 2016), 20e:1–20e:3. ISSN: 0730-0301. DOI: 10.1145/2930661.

[4] Gilbert Louis Bernstein et al. 'Ebb: A DSL for Physical Simulation on CPUs and GPUs'. In: *ACM Trans. Graph.* 35.2 (May 2016), 21:1–21:12. ISSN: 0730-0301. DOI: 10.1145/2892632.

[5] Kevin J. Brown et al. 'A Heterogeneous Parallel Framework for Domain-Specific Languages'. In: *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*. PACT '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 89–100. ISBN: 978-0-7695-4566-0. DOI: 10.1109/PACT.2011.15.

[6] Jacques Carette, Oleg Kiselyov and Chung-chieh Shan. 'Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages'. In: *J. Funct. Program.* 19.5 (Sept. 2009), pp. 509–543. ISSN: 0956-7968. DOI: 10.1017/S0956796809007205.

[7] Hassan Chafi et al. 'Language Virtualization for Heterogeneous Parallel Computing'. In: *SIGPLAN Not.* 45.10 (Oct. 2010), pp. 835–847. ISSN: 0362-1340. DOI: 10.1145/1932682.1869527.

[8] NVIDIA Corporation. *CUDA C Best Practices Guide*. URL: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html (visited on 23/06/2018).

[9] NVIDIA Corporation. *CUDA C Programming Guide*. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (visited on 28/10/2018).

[10] NVIDIA Corporation. *CUSP*. URL: https://developer.nvidia.com/cusp (visited on 17/10/2018).

[11] Zachary DeVito et al. 'Liszt: A Domain Specific Language for Building Portable Mesh-based PDE Solvers'. In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '11. Seattle, Washington: ACM, 2011, 9:1–9:12. ISBN: 978-1-4503-0771-0. DOI: 10.1145/2063384.2063396.

[12] Zachary DeVito et al. 'Opt: A Domain Specific Language for Non-linear Least Squares Optimization in Graphics and Imaging'. In: *arXiv preprint arXiv:1604.06525* (2016).

[13] Christopher Earl et al. 'Nebo: An efficient, parallel, and portable domain-specific language for numerically solving partial differential equations'. In: (Jan. 2016).

[14] Thomas L. Falch and Anne C. Elster. 'ImageCL: An Image Processing Language for Performance Portability on Heterogeneous Systems'. In: *CoRR* abs/1605.06399 (2016). arXiv: 1605.06399. URL: http://arxiv.org/abs/1605.06399.

[15] James Hegarty et al. 'Darkroom: Compiling High-level Image Processing Code into Hardware Pipelines'. In: *ACM Trans. Graph.* 33.4 (July 2014), 144:1–144:11. ISSN: 0730-0301. DOI: 10.1145/2601097.2601174.

[16] Christian Hofer et al. 'Polymorphic Embedding of DSLs'. In: *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*. GPCE '08. Nashville, TN, USA: ACM, 2008, pp. 137–148. ISBN: 978-1-60558-267-2. DOI: 10.1145/1449913.1449935.

[17] Håvard Heitlo Holm. 'A CUDA Back-End for the Equelle Compiler'. MA thesis. Norges Teknisk-Naturvitenskapelige Universitet, Fakultet For Naturvitenskap Og Teknologi, 2014.

[18] Sungpack Hong et al. 'Green-Marl: A DSL for Easy and Efficient Graph Analysis'. In: *SIGARCH Comput. Archit. News* 40.1 (Mar. 2012), pp. 349–362. ISSN: 0163-5964. DOI: 10.1145/2189750.2151013.

[19] Open Porous Media Initiative. *Open Porous Media Initiative*. URL: https://opm-project.org/ (visited on 01/11/2018).

[20] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques For Experimental Design, Measurement, Simulation, and Modeling, NY: Wiley*. Apr. 1991. ISBN: 0471503361.

[21] Gordon L. Kindlmann et al. 'Diderot: a Domain-Specific Language for Portable Parallel Scientific Visualization and Image Analysis'. In: *IEEE Trans. Vis. Comput. Graph.* 22.1 (2016), pp. 867–876. DOI: 10.1109/TVCG.2015.2467449.

[22] Fredrik Kjolstad. *GPU Code Generation*. URL: https://github.com/tensor-compiler/taco/issues/129 (visited on 14/10/2018).

[23] Fredrik Kjolstad et al. 'Simit: A Language for Physical Simulation'. In: *ACM Trans. Graph.* 35.2 (May 2016), 20:1–20:21. ISSN: 0730-0301. DOI: 10.1145/2866569.

[24] Tzu-Mao Li et al. 'Differentiable programming for image processing and deep learning in Halide'. In: *ACM Trans. Graph. (Proc. SIGGRAPH)* 37.4 (2018), 139:1–139:13.

[25] Adriaan Moors et al. 'Scala-virtualized'. In: *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation*. PEPM '12. Philadelphia, Pennsylvania, USA: ACM, 2012, pp. 117–120. ISBN: 978-1-4503-1118-2. DOI: 10.1145/2103746.2103769.

[26] Ravi Teja Mullapudi et al. 'Automatically Scheduling Halide Image Processing Pipelines'. In: *ACM Trans. Graph.* 35.4 (July 2016), 83:1–83:11. ISSN: 0730-0301. DOI: 10.1145/2897824.2925952.

[27] Jonathan Ragan-Kelley et al. 'Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines'. In: *ACM Trans. Graph.* 31.4 (July 2012), 32:1–32:12. ISSN: 0730-0301. DOI: 10.1145/2185520.2185528. URL: http://doi.acm.org/10.1145/2185520.2185528.

[28] Jonathan Ragan-Kelley et al. 'Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines'. In: *SIGPLAN Not.* 48.6 (June 2013), pp. 519–530. ISSN: 0362-1340. DOI: 10.1145/2499370.2462176.

[29] Atgeirr Flø Rasmussen. *Equelle Reference Manual*. URL: https://github.com/sintefmath/equelle/blob/master/doc/EquelleReference.pdf (visited on 17/10/2018).

[30] Tiark Rompf. 'Reflections on LMS: Exploring Front-end Alternatives'. In: *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala*. SCALA 2016. Amsterdam, Netherlands: ACM, 2016, pp. 41–50. ISBN: 978-1-4503-4648-1. DOI: 10.1145/2998392.2998399.

[31] Tiark Rompf and Martin Odersky. 'Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs'. In: *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*. GPCE '10. Eindhoven, The Netherlands: ACM, 2010, pp. 127–136. ISBN: 978-1-4503-0154-1. DOI: 10.1145/1868294.1868314.

[32] Arvind K. Sujeeth et al. 'Composition and Reuse with Compiled Domain-specific Languages'. In: *Proceedings of the 27th European Conference on Object-Oriented Programming*. ECOOP'13. Montpellier, France: Springer-Verlag, 2013, pp. 52–78. ISBN: 978-3-642-39037-1. DOI: 10.1007/978-3-642-39038-8_3.

[33] Arvind K. Sujeeth et al. 'Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages'. In: *ACM Trans. Embed. Comput. Syst.* 13.4s (Apr. 2014), 134:1–134:25. ISSN: 1539-9087. DOI: 10.1145/2584665.

[34] Arvind K. Sujeeth et al. 'Forge: Generating a High Performance DSL Implementation from a Declarative Specification'. In: *SIGPLAN Not.* 49.3 (Oct. 2013), pp. 145–154. ISSN: 0362-1340. DOI: 10.1145/2637365.2517220.

[35] Arvind K. Sujeeth et al. 'OptiML: An Implicitly Parallel Domain-specific Language for Machine Learning'. In: *Proceedings of the 28th International Conference on International Conference on Machine Learning*. ICML'11. Bellevue, Washington, USA: Omnipress, 2011, pp. 609–616. ISBN: 978-1-4503-0619-5. URL: http://dl.acm.org/citation.cfm?id=3104482.3104559.

[36] Walid Taha and Tim Sheard. 'Multi-stage Programming with Explicit Annotations'. In: *SIGPLAN Not.* 32.12 (Dec. 1997), pp. 203–217. ISSN: 0362-1340. DOI: 10.1145/258994.259019. URL: http://doi.acm.org/10.1145/258994.259019.