Parallelizing the Edge application for GPU-based systems using the SkePU skeleton programming library

by

Oskar Sjöström

LIU-IDA/LITH-EX-A--15/001--SE

2015-02-12
Master Thesis

Parallelizing the Edge application for GPU-based systems using the SkePU skeleton programming library

by

Oskar Sjöström

LIU-IDA/LITH-EX-A--15/001--SE

2015-02-12

Supervisor: Usman Dastgeer and Lu Li
Examiner: Christoph Kessler
Abstract

SkePU is an auto-tunable multi-backend skeleton programming library for multi-GPU systems. SkePU is implemented as a C++ template library and has been developed at Linköping University. In this thesis the CFD flow solver Edge has been ported to SkePU. This combines the paradigm of skeleton programming with the utilization of the unstructured grid structure used by Edge. In order to do this certain extensions have been made to the SkePU library. The performance of the ported implementation has been evaluated to identify if a performance gain can be achieved by parallelizing this type of application with the help of SkePU. A moderate speedup of the application has been achieved given the size of the ported section of the Edge application. Another important outcome of the project is the provided feedback for further development of the SkePU framework.
Acknowledgments

I would like to thank my examiner Christoph Kessler and my supervisors Usman Dastgeer and Lu Li for all their time and efforts. I would also like to thank Soon-Heum Ko and Torben Rasmussen from the National Supercomputing Centre for all the help they provided during the project. I also acknowledge the Swedish National Infrastructure for Computing (SNIC) for providing resources at LUNARC.
Chapter 1

Introduction

This thesis project is a collaboration between the Department of Computer and Information science at Linköping University and the Swedish National Supercomputer Center (NSC). As part of the continued development of the SkePU programming framework several projects have been carried through to evaluate how SkePU can be used to improve the portability and performance of various applications. In this project the CFD flow solver Edge is parallelized with the aid of SkePU. Edge was selected after evaluating several candidates of applications that could be of interest to port into SkePU. The first step of the project was for some initial analysis of the Edge code to be made. Following this the functions with the largest impact on the performance of the application would be ported into C++. After porting these functions to C++ they would be parallelized with the aid of the SkePU library. It was expected at the start of the project that modifications to SkePU might be necessary in order to be able to support the types of computations present in Edge. Once an implementation had been produced its performance and portability should be evaluated.

1.1 Motivation

The main motivation behind the project is to show the viability of SkePU. If widely used applications can be modified with reasonable effort to use SkePU and possibly achieve a performance increase in doing so it would be of great value. Thesis projects related to SkePU also serve as a valuable source of feedback for future development of the SkePU library. Additionally any improvement of the Edge software will be beneficial to the developers of Edge at the Swedish Defence Research Agency (FOI).

1.2 Project goals

The following is a list of the goals that the project hoped to achieve.

- Porting the most computationally intensive subroutines of Edge to C++.
- Rewriting the code by equivalent combinations of SkePU skeletons.
- Evaluating the result and optimizing the implementation.
- Investigate performance portability for different GPU configurations.
- Providing feedback and suggestions for the further design of SkePU.

1.3 Project requirements

These are the project requirements agreed to at the start of the project.

Priority 1 requirements shall be achieved, priority 2 requirements are hopefully achieved.

- The resulting program shall produce the same output as the original Edge code. (1)
- The program shall be able to run on an NVIDIA Fermi GPU (1)
- The new implementation shall be written in C++ using SkePU (1)
• The new implementation shall give improved performance over the old Edge code (2)
• The program can be easily modified to run on alternative hardware platforms (2)

1.4 Thesis outline

The structure of this report is as follows:
• Background on multicore programming, skeleton programming, the SkePU framework and the Edge application is given in Chapter 2.
• Chapter 3 contains a description of the work that has been done in the project.
• Detailed results of the performance of the application and the various experiments that have been conducted are shown in Chapter 4.
• In Chapter 5 there is a discussion of projects and work closely related to this thesis.
• Feedback for further development of the SkePU framework is presented in Chapter 6.
• Chapter 7 is dedicated to conclusions drawn from the project.
• Appendix A contains the source code for the modifications done to SkePU.
• In Appendix B, the raw data that the results chapter is based on is presented.
Chapter 2

Background

This chapter provides background on the SkePU framework, the Edge application as well as general theory of multicore programming.

2.1 Multicore and GPU programming

For a long time the performance of computer architectures based on a serial processing mode has been able to increase according to Moore's law. Performance has increased by increasing the clock frequency of the processor and by packing a larger number of transistors per chip. More recently computer architectures have moved towards architectures based on multiple processor cores. The move away from single processor architectures are mainly due to no longer being able to increase performance by increasing the operating frequency of the processor. The reasons for this were formulated by David Patterson as the three walls [1].

- The memory wall
  
The speed of memory accesses has not increased by the same rate as the operating frequency of the processor. In the past operations such as multiplications would require more time than a memory access. This is no longer true. Increasing operating frequency further would increase the gap between the speed of processors and memory. This leads to many applications being bottlenecked by memory operations.

- The ILP wall
  
  It becomes harder and harder to find ways to utilize additional instruction level parallelism. When a sufficient level of instruction level parallelism cannot be obtained all parts of the processor are not being efficiently made use of.

- The power wall
  
  The dynamic power consumption of a processor increases cubically as the operating frequency is increased. This has the effect that the increased computing power becomes more and more expensive. These diminishing returns make it so that there comes a point at which increasing the operating frequency is no longer worth it. There is also a threshold for how high of an operating frequency can be used while relying on a regular computer fan for cooling. Once this threshold has been passed the cooling required is substantially more expensive.

In parallel with the move towards multicore architectures there has also been an increase in use of GPU-accelerated computing. GPU acceleration works by allowing the computationally intensive parts of the code to be executed on a GPU which contains a larger number of cores than a CPU. GPUs were originally designed for calculating individual pixel values of images which in itself is a massively parallel problem. As the evolution of multi-core architectures progressed more and more people saw the potential benefits of using GPUs for general purpose computations. The main drawback was initially the difficulty in creating programs that were able to utilize the GPU. With the development of new programming interfaces such as CUDA and OpenCL these difficulties have been somewhat
mitigated. Despite these advances GPU programming offers a low level of abstraction compared to the languages used in modern sequential programming.

2.2 Skeleton programming

One drawback of multicore solutions is the lack of a unified programming model for these new architectures. Creating sequential programs for single processor architectures for the most part require little knowledge of how the hardware functions. The unified programming model for sequential programming also gives a lot of portability across various platforms. To make use of the full capacity of multicore systems the programmer needs to learn how to handle communication and synchronization between processors. As there is a great variety in the way multicore architectures function programs typically need to be custom made for a specific architecture in order to make use of its full potential. One possible solution to these problems is the concept of skeleton programming.

The main goal of skeleton programming is to hide these platform specific issues of programming from the programmer. By providing a set of generic components whose underlying implementations take care of these issues the programmer is given a simple interface to work with. These generic components are what we refer to as skeletons. The interface provided by these skeletons allows for the programmer to create a program that seemingly works in a sequential manner. This has several benefits. The code becomes more portable as it does not need to be customized for a specific architecture. It also allows more developers to work with multicore programming without requiring every programmer to be an expert on multicore architectures.

Complications arise when there is a problem which does not nicely fit into the pre-defined skeletons. In such cases custom solutions may still be required. As the frameworks of skeleton programming are expanded and more research is done in the area the types of algorithms that fit into the skeleton programming paradigm will also be increased. This will hopefully allow for a unified programming model to emerge, similar to the one available for programming single processor systems. As a stepping stone towards this several different libraries for skeleton programming have been developed. One such library is the SkePU library that is being used in this project.

In general there are two types of skeletons related to multicore programming. There are task parallel skeletons and data parallel skeletons. Task parallel skeletons are used to execute multiple tasks concurrently. Data parallel skeletons which are more relevant to this project, are used to perform the same task on multiple elements of data in parallel. Data parallel skeletons are best suited to be utilized by GPU systems that are designed to efficiently execute the same code on multiple processors.

2.3 SkePU

This section will give an overview of the SkePU library with a focus on the parts that are the most relevant to this project. SkePU was originally created as a master thesis project by Enmyren [2] at Linköping University. Further development of SkePU has been made in majority by Kessler and Dastgeer. The SkePU project has since then resulted in a number of publications [3]. SkePU was first introduced and published in the proceedings of the 2010 HLPP workshop in Baltimore [4]. The work by Dastgeer, Li and Kessler [5] on the implementation of automatic selection for the multiple backends was published in the proceedings of the APPT conference. The addition of smart containers by Dastgeer and Kessler [6] was presented at the HLPP symposium. The SkePU smart containers are also thoroughly described in Dastgeer's doctoral thesis [7].

SkePU is an open source C++ template library for skeleton programming. It is used by including the SkePU header files of each skeleton and container type used in the application. This means that there is no installation process required and programs using SkePU are compiled using regular C++ /CUDA compilers.
Templates in C++ is a feature that allows the programmer to define classes and functions with the help of generic types. With the help of this an implementation can be made that is capable of handling inputs of different types rather than defining a custom implementation for each one. There are similar features available in many other languages such as Generics in Java and Ada.

SkePU makes use of templates in several ways. Each skeleton is represented by a class object. These class objects are instantiated with template parameters. The different backends of each skeleton are implemented as member functions which are also called using template parameters. The benefits of this is that each skeleton can be called using different types of user functions and the user functions themselves can be customized to use the most suitable parameter types.

2.3.1 Supported skeletons and custom data types

There are numerous possible algorithmic skeletons, the following is a brief overview of the skeletons that have been implemented in the SkePU library.

SkePU provides six different data-parallel skeletons

Map
The map skeleton produces an output vector or matrix where every element is a function of corresponding elements of the input vectors or matrices respectively. The map skeleton can take up to three inputs in the current implementation.

Maparray
Maparray works similar to map with a couple of exceptions. Maparray always takes two inputs and each element of the result vector or matrix is calculated as a function of the corresponding element from one of the inputs and any number of elements from the other one.

Mapoverlap
Mapoverlap is another variant on the map skeleton. In mapoverlap each element in the output vector or matrix is calculated as a function of the corresponding and multiple adjacent elements of the input.

Reduce
A reduction computes a scalar value by applying a commutative associative binary operator across all elements of its input.

Mapreduce
Mapreduce functions as a combination of the map and the reduce skeleton. It produces the same result as performing a reduction on the results of a mapping.

Scan
Scan computes a series of reductions such that each element at index $n$ in the output corresponds to a reduce operation of elements 1..$n$ from the input.

SkePU also contains a single task-parallel skeleton

Farm
By integration with the StarPU runtime system the farm skeleton allows for independent tasks of possibly different types to be run in parallel.

Data types

The library also contains implementations of two custom datatypes to be used along with the skeletons. These datatypes are referred to as Vector and Matrix. The Vector is based on the STL vector and represents a one dimensional array. The Matrix represents a two dimensional array and works similarly to the Vector with some added functionality to adapt to the MapOverlap skeleton. These two datatypes are used for the majority of inputs and outputs of skeleton calls.
2.3.2 Features

A very important feature of SkePU is the ability to run the same code with different backend implementations. The task-parallel skeletons in SkePU have support for sequential CPU, OpenMP, Cuda and OpenCL backends. This makes SkePU applications extremely portable as well as allowing the most efficient backend to be chosen for each situation. The backend that is active can either be chosen by the programmer directly in the user code or the SkePU library can choose which backend to use. The library chooses which backend to use by automatically generating execution plans containing configuration and backend selection. It is also possible to manually generate new execution plans. As an example this could be used by the programmer to select backend based on problem size.

SkePU contains certain autotuning features. These are based mainly on off-line calculations. To be able to adapt during runtime SkePU has been integrated with the StarPU runtime system. This allows for online tuning of the execution plans.

The Vector and Matrix containers used for passing operand data to skeletons can be used to wrap existing data structures. The containers are then be used to automatically keep track of valid copies of the data and where it is stored.

2.3.3 User function Implementation

The user functions in SkePU are the parts of the program that will be parallelized. With the help of a macro language users can specify functions that will expand into a struct. This struct is then used by the SkePU library when executing a skeleton. The macro language defines different types of functions that can be used. Each skeleton can make use of a subset of these types of functions. For instance the maparray skeleton requires user functions to be of the type ARRAY_FUNCTION. As the macro definitions can only expand statically a different macro definition is required for any change in the user function such as number of inputs. Figure 2-1 shows an example of a macro expansion taken from Usman Dastgeers licentiate thesis[8].

```cpp
BINARY_FUNC(plus_f, double, a, b, struct plus_f
  return a+b;
)

// EXPANDS TO: =====>
struct plus_f
  { skue::FuncType funcType;
    std::string func_CL;
    std::string datatype_CL;
    plus_f()
    {
      funcType = skue::BINARY;
      funcName_CL.append("plus_f");
      datatype_CL.append("double");
      func_CL.append("double plus_f(double a,double b)\n" "+\n" "return a+b;\n"\n"");
    }

double CPU(double a, double b)
{
  return a+b;
}

device__ double CU(double a, double b)
{
  return a+b;
}

Figure 2-1: A SkePU macro expansion
2.3.4 Applications previously ported to SkePU

A number of common applications have been ported into SkePU previously. These programs show that SkePU is capable of parallelizing many common algorithms. Test applications for the following program types are included in the SkePU distribution [3].

- a Runge-Kutta ODE solver
- separable and non-separable 2D image convolution filters
- Successive Over-Relaxation (SOR)
- Coulombic potential grid application
- N-body simulation
- LU Decomposition
- Mandelbrot fractals
- Smooth Particle Hydrodynamics (SPH, fluid dynamics shocktube simulation)
- Pearson Product-Moment Correlation Coefficient (PPMCC)
- Mean Squared Error (MSE)

A simple example of a SkePU program, taken from Usman Dastgeer’s licentiate thesis [8] is shown in Figure 2-2.

```cpp
1 # include <iostream>
2 # include "skepu/vector.h"
3 # include "skepu/mapreduce.h"
4
5 BINARY_FUNC ( plus_f , double , a, b, 
6 return a+b; 
7 )
8
9 BINARY_FUNC ( mult_f , double , a, b, 
10 return a*b; 
11 )
12
13 int main ()
14 {
15 skepu :: MapReduce < mult_f , plus_f > dotProduct ( new mult_f ,
16 new plus_f );
17
18 skepu :: Vector < double > v1 (500 ,4) ;
19 skepu :: Vector < double > v2 (500 ,2) ;
20
21 double r = dotProduct (v1 ,v2);
22
23 std :: cout <<" Result : " <<r <<"\n";
24
25 return 0;
26 }
27
28 // Output
29 // Result: 3000
30
```

Figure 2-2: Dot product computation implemented with SkePU

This shows how to compute the dot product of two vectors using the mapreduce skeleton.

2.4 Edge

Edge is a computational fluid dynamics (CFD) flow solver developed by the Swedish Defence Research Agency (FOI). Several of FOI’s partners have contributed to the development, notably SAAB Aerosystems and The Royal Institute of Technology (KTH) [9]. Edge is used mainly for calculations within the field of aerodynamics but it has many potential uses. The users of Edge
include among others the Swedish National Supercomputer Center (NSC), Volvo Aero and Saab Aerosystems [10]. Examples of other flow solvers similar to Edge are SU2 developed at Stanford University and FLUENT by ANSYS. Edge is written in the Fortran programming language in compliance with the Fortran 95 ISO/ANSI standard [11]. The Edge software consists of five larger parts [12]:

• Preprocessor
Before running the flow solver the input mesh files require preprocessing to be adapted to the input format of the main program.
• Flow solver
The flow solver performs the computations required to complete the experiments provided by the input mesh files.
• Helper programs
The helper programs provide additional ways of adjusting the input and output data of Edge.
• Application programs
Additional programs that let the Edge user manipulate the mesh files.
• Xedge
Xedge is the graphical user interface for Edge. Xedge is made in Java and is an optional part of the software.

In this project we work with the flow solver portion of the program. The other parts were only investigated to the extent that was needed to facilitate proper configuration and testing.

A distinguishing feature of Edge is that it works on unstructured grids with arbitrary elements. An unstructured grid does not have a direct mapping between how data is stored in memory and the physical connection between nodes. That Edge works with arbitrary elements refers to the type of geometrical elements the grid consists of. Arbitrary elements means that Edge accepts grids composed of any type of polygons.

Edge does have support for parallelization through Message Passing Interface (MPI) which can be used when running Edge on a CPU cluster. This parallelization is implemented through domain partitioning. What this means is that the data that is being operated on is split into pieces and these are treated by the flow solver individually. The message passing is then used to communicate the results from the flow solver to update boundary values in between iterations.

2.4.1 Computational fluid dynamics
Computational fluid dynamics is a field of science which solves problems involving fluid flows with the help of numerical methods. Fluid flows are mathematically modelled with the help of systems of partial differential equations. These equations can then be solved with the help of computer software. An example of a field where CFD is used is meteorology. Atmospheric phenomena such as rain and wind can be modelled as fluid flows and CFD solvers are used to create weather forecasts. In the case of Edge the software is mostly used within aerodynamics where it can be used to design the shapes of airplanes.

2.4.2 The multigrid method
Multigrid is the name of a group of methods used when solving differential equations numerically. These methods have been commonly applied in the field of computational fluid dynamics. A review of some of the important aspects of using multigrid for CFD can be found in the Journal of Computational Mathematics [13].

Multigrid can be implemented in numerous ways, what the different solutions have in common is that they solve problems by working on multiple levels of coarseness. The input grid or mesh is altered to create a set of inputs of varying coarseness. The number of grids used for a specific problem is referred to as the level of multigrid being used. The benefit of using multigrid is that it increases the
rate of convergence. Calculations are made on the coarser grids and the results are propagated to the finer grid levels. The iterations made on the coarser grids will take less time. This is both due to the fact that there are fewer nodes to make calculations for and because Edge makes use of simplifying assumptions when calculating the coarser grid levels [12].
Chapter 3

Design

This chapter will detail the different steps of the solution, from the initial analysis of the code to the final implementation. The structure of this chapter roughly mimics the chronological progression of the project. Necessary limitations and design choices will be brought up in the section most closely relating to them.

3.1 Profiling and analysis

To determine which parts of the Edge code were most suitable for porting an initial profiling run was done for a large scale test case (run on a cluster with MPI) by Soon-Heum Ko of NSC. The parameters of this profiling are shown in table 3-1. This profiling was meant to find the most performance critical functions of the Edge code. This profiling was done using the Scalasca software [14].

Table 3-1. Profiling parameters

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Linux x86, 64bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler</td>
<td>IFORT, icc</td>
</tr>
<tr>
<td>Edge version</td>
<td>5.2.1</td>
</tr>
<tr>
<td>CPU-Cores</td>
<td>64</td>
</tr>
<tr>
<td>Testcase</td>
<td>Wing-body with iterations set to 1000</td>
</tr>
</tbody>
</table>
Figure 3-1. Results from the initial profiling of the Edge code

In Figure 3-1 the results of this profiling are shown. The function that the application spends the largest amount of time executing is the Artvis function which is part of the flow solver. Roughly 29% of the total execution time of Edge is spent in the Artvis function. Upon inspecting the source code of the flow solver we can see that the Artvis function consists of two nested loops performing calculations on a large data set. This should provide a large amount of data parallelism to exploit with the help of parallelization. Based on these results the first function to be ported into C++ and later parallelized with SkePU was the Artvis function.

3.2 Porting

This section will describe the different steps during the part of the project where parts of the Edge code was ported from Fortran into C++. The porting to C++ was necessary to be able to make use of the SkePU framework.

The parts of the Edge code that are ported to C++ still need to run together with the rest of the application. To do this the C++ functions will be called from within the Fortran code. To avoid the name mangling that is used by the C++ compiler each function is given a C interface which is adapted to suit the way Fortran function calls operate. The ported functions are compiled separately and linked together with the Fortran code during compilation of Edge. Once compiled Edge functions in the same manner as before the C++ code was introduced.
3.2.1 Verification

The C++ and later SkePU implementations need to be verified to ensure that the functionality of the Edge code remains intact. For this verification the test case rae_case10 was used. This test case does not contain a typical input for the edge software. The test case is designed so that changes in the calculations have a larger effect on the output. This makes it ideal for verifying the functionality of our code as any incorrect behavior is more likely to show itself in the result.

Figure 3-2. Sample output of Edge

Figure 3-2 shows an output from a run of the Edge software. The important part of this output when looking to verify the functionality of the program is the RMS_RES value which contains the L2 norm of the physical property. The Artvis routine updates a number of arrays containing the gradient of physical properties. If this output remains unchanged then we assume that the same result has been obtained in the new implementation as in the original.

This output is given at a fairly low resolution which means that there could possibly be errors in the application that do not manifest themselves for this particular test case. To get more detailed information about the behavior of the ported function the sum of the values calculated inside it were printed at a high resolution. These values were then compared between the C++ and Fortran versions of the code.

Table 3-2. Comparison of outputs between Fortran and C++ version of Artvis

<table>
<thead>
<tr>
<th>Variable</th>
<th>Fortran output</th>
<th>C++ output</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sum of FL1</td>
<td>-1.09000346244339</td>
<td>-1.09000346244347</td>
<td>8.10^{-14}</td>
</tr>
<tr>
<td>Sum of FL2</td>
<td>74.521905171943</td>
<td>74.521905171943</td>
<td>3.4.10^{-11}</td>
</tr>
<tr>
<td>Sum of FL3</td>
<td>71.1961065609901</td>
<td>71.1961065609854</td>
<td>4.7.10^{-12}</td>
</tr>
<tr>
<td>Sum of FL4</td>
<td>0.0000000000000000E+000</td>
<td>0.0000000000000000E+000</td>
<td>0</td>
</tr>
<tr>
<td>Sum of FL5</td>
<td>-361579.634253773</td>
<td>-361579.634253736</td>
<td>-3.7.10^{-8}</td>
</tr>
</tbody>
</table>

Table 3-2 shows a comparison of these outputs. We can see that there is a small difference in the outputs. There could be various factors that contribute, the most likely being a difference in the way
math functions are implemented in the different languages. The difference in output was deemed negligible after observing that a similar change in output occurred when compiling the same code on a different hardware platform.

### 3.2.2 Performance

Once the functionality of the C++ version had been verified the performance was compared to the original Fortran version. Despite the functionality being equivalent it could be that calling the C++ functions in this manner causes some overhead affecting the performance. Another possibility is that the different implementations of underlying math functions within the two languages could cause a change in performance. Due to the difference in the way multidimensional arrays are implemented in C++ and Fortran it is also important to maintain the order in which data is accessed. If the order of data accesses were reversed it would cause a significant downgrade in performance.

![Figure 3-3. Execution times of the Artvis function using Fortran and C++ implementations for various testcases](image-url)
To compare the performance the execution times of the Artvis function was measured using both the original Fortran code and the new C++ version. The parameters of this profiling are shown in Table 3-3 and the results are shown in Figure 3-3. This profiling shows that the C++ port gives a performance degradation of roughly 30% for the Artvis function. Later on when porting additional functions this degradation did not show, meaning that it is specific to the Artvis function. Based on this it was concluded that the degradation most likely stems from the implementation of the power function. In Fortran this is an intrinsic function while in C++ it is implemented as part of a math library.

Table 3-3. Profiling parameters for the C++/Fortran comparison

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating System</td>
<td>Linux x86, 64bit</td>
</tr>
<tr>
<td>Compiler</td>
<td>GFortran, GCC</td>
</tr>
<tr>
<td>Edge version</td>
<td>5.3.0</td>
</tr>
<tr>
<td>CPU-Cores</td>
<td>1</td>
</tr>
</tbody>
</table>

3.3 SkePU extensions

The Artvis function is a part of the flow solver whose role is to calculate and update certain flux values. When considering how to implement a parallel version of this function we first need to look at the structure of the code. Figure 3-4 shows a simplified look at the structure of the Artvis function.

```c
for IC = 1 .. NCOL
    for IED = ICOL(IC,1) .. ICOL(IC,2)
        IP1 = IEDN(IED,1)
        IP2 = IEDN(IED,2)
        // computations
        DV1 = RO[IP1] - RO[IP2];
        DV2 = RO[IP1]*UU[IP1] - RO[IP2]*UU[IP2];
        DV3 = RO[IP1]*VV[IP1] - RO[IP2]*VV[IP2];
        // more computations
        ..
        // resulting flux values
        FL1 = ..
        FL2 = ..
        // update data structures with computed fluxes
        array1[IP1] += FL1
        array1[IP2] -= FL1
        array2[IP1] += FL2
        array2[IP2] -= FL2
```

Figure 3-4: Artvis function code structure

As it turns out the part of the function that uses the calculated fluxes to update an existing data structure is not possible to parallelize. The indices used (IP1, IP2) will not be unique to each thread. This means that there may be collisions when trying to update these arrays. Unfortunately this cannot be solved with a simple critical section or usage of locks. This is because locking the update of an array would effectively serialize the code when ideally only one element of the array would be locked. To instead detect or avoid these collisions would require substantial communication between threads which is especially undesirable when working with GPUs. Due to this a decision was made to leave this part of the function sequential.

Ideally in skeleton programming the sequential code should be broken down into various skeletons. Considering the skeletons available in SkePU and the structure of the Artvis code in the Edge flow solver the skeleton that at first seems the most suited for the problem is the mapoverlap skeleton. Mapoverlap is designed to fit algorithms where neighboring elements are used to calculate the next step of an iteration. The calculations in Artvis are based on neighboring elements but
neighboring elements in the mesh are not necessarily neighbors in terms of how the data is stored in memory. The Reduce and mapreduce skeletons are used when dealing with calculations that use every element of a vector as its input. The most common usage for the two skeletons are different kinds of summations. No such calculations are relevant to the code in question. Scan performs similar types of computation as the reduce skeletons and is also not relevant. Farm is a task-parallel skeleton while in Edge we work with a large amount of data-parallelism. This leaves us with the map and maparray skeletons. The problem with using the map skeleton is that it expects a one to one mapping of indices and outputs which is not necessarily present when working with indirect addressing and an unstructured grid.

### 3.3.1 Using existing SkePU skeletons

An attempt was made to implement parallelization without making modifications to the SkePU library. To make use of the maparray skeleton SkePU has a special type of user function called ARRAY_FUNC. The macro definition of ARRAY_FUNC is shown in Figure 3-5.

```c
#define ARRAY_FUNC(name, type1, param1, param2, func)\
struct name\
  {\
   typedef type1 TYPE;\
   typedef type1 CONST_TYPE;\
   skepu::FuncType funcType;\
   type1 dummy;\
   type1 getConstant() {return dummy;};\
   name()\
   {\
    funcType = skepu::ARRAY;\
   }\
   type1 CPU(type1 * param1, type1 param2)\
   {\
    func\
   }\
   __device__ type1 CU(type1 * param1, type1 param2)\
   {\
    func\
   }\
  };
```

**Figure 3-5: Macro definition of ARRAY_FUNC**

As we can see from this definition the user function needs to be implemented such that

- The input and output are of the same type
- Each call to maparray has only a single input where the entire array can be accessed.

A typical line of code inside the section of Edge that we are looking to parallelize looks similar to this:

```c
DV2 = RO[IP1]*UU[IP1] - RO[IP2]*UU[IP2];
```

Where RO and UU are arrays of double precision floating point numbers. Here there are two input arrays that need to be accessed at different indices with no way of separating them from each other. To use maparray at least one of the two inputs would need to be reordered such that it can be accessed in a regular fashion. There is also not a direct relation between the values of the two indices IP1 and IP2 which means that even for a single array input the skeleton would require additional inputs to provide information about which parts of the array are going to be accessed. An alternative to using maparray is to make use of the map skeleton. To use the map skeleton each input needs to be reordered such that each element appears in the order it will be accessed. As it is possible for the same index to be used at multiple points this also leads to some duplication of data. To implement this the above example calculation can be broken down into several smaller calculations:

- reorder RO to match the values of IP1
• reorder RO to match the values of IP2
• reorder UU to match the values of IP1
• reorder UU to match the values of IP2
• calculate \( RO[IP1]*UU[IP1] \)
• calculate \( RO[IP2]*UU[IP2] \)
• calculate DV2 as the difference of the two previous calculations

As many of the intermediate results can be reused each such calculation would on average require three different calls to the map skeleton. As can be expected this implementation had very poor performance for a number of reasons: The large number of skeleton calls cause a lot of overhead; Instantiation of additional data structures as well as duplication and reordering of data leads to an increase in the total calculations required. After this evaluation it became clear that a modification of SkePU was required to be able to find a sufficient implementation.

3.3.2 Modifying SkePU

The goal in modifying SkePU was to create a variant of the maparray skeleton that allows for an arbitrary number of read-only inputs. This would allow the parts of the Edge flow solver that we are looking to parallelize to be implemented with a single skeleton call causing minimal overhead.

Before a solution was chosen some different ways of solving the issues were considered. The most elegant solution would be to have a parameter with a dynamic size. This could be implemented as list of lists or with variadic templates. The main issue with these types of constructs is that they are not compatible with CUDA. It is possible to pass a class object such as an STL vector to a device function but it is not possible to use its member functions, unless those member functions are also declared as device functions which would break the portability of the code. C++11 features such as variadic templates that could elegantly solve the problem are also not currently supported by CUDA.

As one of the main features of SkePU is to have code that is highly portable and works across different platforms it was desirable to have a solution that maintains this.

Another way to approach the problem is to attempt to pack the input data into a single vector. This vector could then be passed into the existing maparray implementation. Issues with this is the high overhead it causes. If the entire application was made from scratch the data structures could be adapted to suit SkePU. However since we are only modifying a small portion of a large existing application it is not feasible to alter the data structures used by the rest of the program. This would mean that the data needs to be packed each time before a SkePU skeleton is used and decomposed back after the skeleton call, creating overhead that grows in proportion to the problem size.

It would also not be entirely compatible with the existing maparray skeleton due to limitations in the way it has been implemented. For this approach to work there needs to be knowledge of the offset to each input. This data also needs to be available on the GPU which requires the maparray skeleton to accept additional inputs. Maparray is also limited in that it expects the output to be of the same type as its input. The ported functions of the Edge flow solver create multiple outputs which do not map to multiple inputs. Splitting the function into multiple parts so that each skeleton call only produces a single output is also undesirable. This is due to a large amount of redundant calculations having to be made as several intermediate results are used to create multiple outputs.

Ultimately the decision was made between two different types of implementations. One option was to set a cap to the amount of inputs that the skeleton would be able to handle. An implementation with a set number of inputs would be made where each individual input would be handled the same way as a single input was in the previous implementations. The user would then fill any extraneous inputs with dummy data which would be ignored by SkePU. There are two main drawbacks to this approach.

• A maximum number of accepted inputs would have to be set rather than allowing an arbitrary number of inputs.
• It would potentially create very messy user code.
The second option was to use a struct encapsulating arrays of an arbitrary size. The individual elements of this struct can then be accessed inside a device function. The main drawbacks of this approach is that the user has to access data directly rather than via an accessor method and the implementation is not as easily integrated into the existing SkePU framework. The latter approach was chosen in order to have an implementation supporting any number of inputs. The details of this implementation are described in the following section.

### 3.3.3 New maparray implementation

The new implementation of maparray was designed with the goal to allow an arbitrary number of read-only inputs. To make this possible a new data structure needed to be added. The purpose of this data structure (henceforth referred to as 'multi-vector') is to encapsulate all the input vectors that are required by the user function. For compatibility with Cuda the multi-vector was implemented as a basic struct. This allows it to be passed as an input when invoking a Cuda device function and the struct members can be accessed from within such a device function.

The multi-vector consists of two structs. One struct to hold arrays of input data and a second one to hold pointers to such arrays. Both the structs consists of three members: two array pointers to store the location of the input data on the host and on the device and one scalar value containing the size of the arrays. The interface to the structure consists of a number of member functions that can be used on the host side when creating the multi-vector structure. Inside the user function an interface like this is not possible while keeping compatibility with Cuda. Instead the user needs to access each data element directly inside the struct. A nice feature of this kind of implementation is that it can encapsulate data of any type without restrictions. Below shows an example of what the user code would look like when using the multi-vector.

In main code:
```cpp
int *input1 = {2,3,8,4};
double *input2 = {1.3,5.2};
skepu::Vector<double> result(numthreads);
MultiVector C;
C.allocData(2);
C.addData(0,4*sizeof(int),input1);
C.addData(1,2*sizeof(double),input2);
userfunc(threadIDs,C,result); // threadIDs unused in example below
```

Inside user function:
```cpp
int *input1 = (int *) C.device_arr[0].device_el; // {2,3,8,4}
double *input2 = (double *) C.device_arr[1].device_el; // {1.3,5.2}
return input2[0] + input1[2]; // 1.3 + 4
```

When calling the CUDA backend of this skeleton a deep copy of the multi-vector structure is performed to transfer all the input parameters to the GPU. The GPU addresses are stored in the device portion of the multi-vector which is then used inside the user function. This is followed by a call to the maparray kernel where the multi-vector is passed as an input to each invocation of the user function. When calling a backend that does not make use of the GPU the pointer to the device portion of the multi-vector is set to the value of the host pointer. This allows the code within the user function to run on any backend without modification. The templates parameters used by the maparray skeleton have also been modified to allow for the input and output parameters to be of different types.

This implementation does have certain limitations that are important to acknowledge. The possible memory management allowed by the multi-vector structure is limited relative to the smart containers available for the SkePU Vector and Matrix data types. The smart containers allow the framework to keep track of data transfers of individual elements between devices. In the case of the multi-vector the input arrays are moved in their entirety regardless of the state of the data available on the device. The multi-vector implementation also does not provide a type-safe interface which puts additional demands on the programmer to ensure the correctness of the code.
The full source code of this maparray implementation can be viewed in Fel: Det gick inte att hitta referenskällan.

### 3.3.4 Porting additional functions

Based on the previous profiling results the code of the flow solver was inspected to find additional functions with a similar code structure to the Artvis function. The function that takes up the second largest amount of execution time is the Convflux function which consists of the same type of loop structure as Artvis. It also works on a similar data set. The size of the computations inside the loop is however considerably smaller. To get a better view of the performance of this type of parallelization this function was also ported to C++ and SkePU.
Chapter 4

Results and discussion

This section details the experiments that have been conducted to measure the performance of the new Edge implementation. The main focus in these tests was the execution time and speedup achieved after porting the two most computationally expensive functions (Artrvis and Convflux). To better evaluate the benefits and drawbacks of GPU assisted execution measurements of data transfer times between GPU and CPU were also done. Information regarding the hardware used during these tests is shown in table 4-2.

It should be noted that it is not possible to set an arbitrary problem size. Running the Edge software requires an input mesh file. The input files describe various physical problems and their compositions include certain variations outside of the number of nodes they contain. For these experiments five different test cases were used, these are detailed in table 4-1.

There is also a hard cap on the problem size that is possible to run on a given GPU. Because the entire input data needs to be available on the GPU it has to fit on the GPU memory. This means that the hardware available during these measurements is capable of executing test cases with at most ten million nodes.

Table 4-1. Listing of test cases used

<table>
<thead>
<tr>
<th>Name</th>
<th>Size (# of nodes)</th>
<th>Multigrid level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static Mixer</td>
<td>2786</td>
<td>4</td>
</tr>
<tr>
<td>Rae_case10</td>
<td>22088</td>
<td>4</td>
</tr>
<tr>
<td>Heating Coil</td>
<td>29826</td>
<td>4</td>
</tr>
<tr>
<td>yf17</td>
<td>97104</td>
<td>4</td>
</tr>
<tr>
<td>Wing body</td>
<td>4074967</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 4-2. Test parameters

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Linux x86, 64bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>NVIDIA Fermi</td>
</tr>
<tr>
<td>Compiler</td>
<td>GFortran, NVCC</td>
</tr>
<tr>
<td>Edge version</td>
<td>5.3.0</td>
</tr>
<tr>
<td>CPU-Cores</td>
<td>16</td>
</tr>
</tbody>
</table>
4.1 Data transfer overhead

When working with general purpose computing on GPUs it is important to consider the overhead of transferring data from the CPU to the GPU. In order to measure this overhead the Nvidia visual profiler was used along with nvprof. When running a CUDA application together with nvprof information is collected during runtime concerning activity on the GPU including data transfers to and from the CPU. Table 4-3 shows how large a portion of the execution time was spent on data transfers as well as time spent in the ported regions of the application.

We can see that the overhead from data transfers makes up a significant portion of the execution time. It is also worth noting that the time spent moving data onto the GPU is similar in size to the time it takes to transfer the results back onto the CPU. One common technique for circumventing this kind of overhead in Cuda is to make use of streams [15] to be able to overlap data transfers with computations. This however relies on knowing which data will be accessed at what time. With the way the Edge code functions this was not a feasible way to handle the problem. The best way to mitigate the overhead from data transfers with the help of SkePU would be to integrate the multi-vector structure introduced in this report together with the existing smart containers. This would allow for less data transfers to be made by avoiding replacing data that has not been changed during previous iterations. Such an implementation is however outside of the scope of this thesis.

<table>
<thead>
<tr>
<th>Test case</th>
<th>Multigrid</th>
<th>Host to Device (s)</th>
<th>Device to Host (s)</th>
<th>Data transfer</th>
<th>Ported section</th>
</tr>
</thead>
<tbody>
<tr>
<td>rae_case10</td>
<td>ON</td>
<td>1.908</td>
<td>1.004</td>
<td>3.55%</td>
<td>45.12%</td>
</tr>
<tr>
<td>rae_case10</td>
<td>OFF</td>
<td>0.229</td>
<td>0.124</td>
<td>7.06%</td>
<td>53.98%</td>
</tr>
<tr>
<td>wing-body ON</td>
<td>18.882</td>
<td>17.824</td>
<td>11.06%</td>
<td>47.29%</td>
<td></td>
</tr>
<tr>
<td>wing-body OFF</td>
<td>9.13</td>
<td>8.588</td>
<td>11.66%</td>
<td>55.81%</td>
<td></td>
</tr>
<tr>
<td>Static mixer ON</td>
<td>0.311</td>
<td>0.229</td>
<td>2.35%</td>
<td>49.90%</td>
<td></td>
</tr>
<tr>
<td>Static mixer OFF</td>
<td>0.064</td>
<td>0.048</td>
<td>2.80%</td>
<td>85.85%</td>
<td></td>
</tr>
<tr>
<td>Heating coil ON</td>
<td>3.014</td>
<td>2.53</td>
<td>5.23%</td>
<td>43.35%</td>
<td></td>
</tr>
<tr>
<td>Heating coil OFF</td>
<td>0.698</td>
<td>0.57</td>
<td>7.93%</td>
<td>65.11%</td>
<td></td>
</tr>
<tr>
<td>yf17 ON</td>
<td>13.764</td>
<td>12.481</td>
<td>8.36%</td>
<td>40.15%</td>
<td></td>
</tr>
<tr>
<td>yf17 OFF</td>
<td>0.23</td>
<td>0.124</td>
<td>0.77%</td>
<td>61.53%</td>
<td></td>
</tr>
</tbody>
</table>
4.2 Performance compared to original implementation

The most important experiment to make is one that compares the new SkePU based implementation with the original Fortran code. As was noted in section 3.2.2 the porting to C++ gave some degradation on the performance of the Edge code. In this chapter comparisons will be made with the sequential C++ version rather than the original Fortran version. This allows us to get a better understanding of how the parallelization impacts the performance.

Figure 4-1 shows partial results of a profiling run done with the intent of better understanding the application. In Figure 4-1 the numbers below the function names represent, in order: portion of execution time used by this branch, portion of execution time used by this node and number of times the function was called. This profiling data shows that while both the Artvis and the Convflux function occupy a substantial amount of the execution time and perform a similar type of algorithm there are important differences in their behavior. The Convflux function is called substantially more often than the Artvis function but each function call takes less time to finish. Because of this the overhead caused by both SkePU and data transfers between the CPU and GPU have a larger impact on the performance of the Convflux function.

![Figure 4-1. Partial results from profiling the sequential C++ version of Edge](image)
Figure 4-2 shows a comparison of execution times for the different implementations. We can see that the Sequential code does perform better than the SkePU version when running only a single thread. When running SkePU with either the OpenMP or the CUDA backend enabled there is however significant improvement. From comparing the two sequential versions we can note that SkePU does create a significant amount of overhead. This overhead is larger than what has been observed for previous applications ported to SkePU. When investigating this overhead it is clear that it is not from a static source such as creating the multi-vector structure. Instead it is each skeleton call showing a lesser performance for the sequential computations compared to the C++ version. This is most likely due to the restructuring of the code which had to be done when modifying the code to use SkePU. In the SkePU version of the code the sequential part is split from the parallel computations, this leads to a decrease in the temporal locality of the code which negatively effects the performance. Despite this the parallel versions do show an improvement that overcomes this.

When comparing the speedup of the two ported functions there is a distinct difference in the results. The Artvis functions with its heavier computations give significantly better results when running on the GPU than the Convflux function does. This shows that to overcome the overhead from transferring data to the GPU requires a fairly large amount of computations. Comparisons of performance for the Artvis function can be seen in Figure 4-3 while Figure 4-4 shows comparisons for the Convflux function.
Figure 4-3. Performance comparison of different backends for the Artvis function

Figure 4-4. Performance comparison of different backends for the Convflux function
4.3 Parallelization scaling with problem size

To investigate how well the implementation scales with the problem size the application was tested with five different test cases. For each test case the application was executed using sequential, OpenMP and CUDA SkePU backends. When performing a test like this there are several things that can affect the results. Some of the factors that can affect the execution time of a program are:

- Applications run by other users concurrently with the experiment
- Cache issues
- Operating system context switches

To minimize the impact of these variations and avoid outliers in terms of execution time each test was conducted three times. During development a noticeable variation in execution times had been noted. This development typically took place during office hours where there was significant activity on the server. The testing that the results in this section are based on took place at nighttime. The results of these tests showed only very small variations in execution times. Due to this it was deemed satisfactory to use the test producing the median value of the total execution time.
Figure 4-5 shows how the speedup of the parallel backends relative to the sequential backend changes based on the problem size. We can see that for the total execution time of the application the OpenMP backend performs better than the CUDA backend. It is to be expected that the CUDA backend in general performs poorly for small problem sizes. Figure 4-6 shows that the OpenMP backend also outperforms the CUDA backend in regards to the Convflux function. The additional overhead that is necessary when performing calculations on the GPU requires there to be a certain amount of parallelized computations to make up for it. When there are not enough calculations to
make up for the overhead the performance suffers. Figure 4-7 shows a speedup comparison for the Artvis function. Here we can see that the GPU assisted implementation does perform the best once there is a sufficiently large amount of computations.

Looking at these various comparisons we can see that all of the graphs flatten out towards the end. The speedup does not continue to increase indefinitely as the problem size grows. One reason for this is that a significant part of the functions was not possible to parallelize. This is a case where Amdahl's law can be applied, Amdahl's law states that the overall speedup of an application with multiple work rates is limited by the performance of the slower part. When applied to parallel programming the multiple work rates consist of the sequential and parallel part of a program. Amdahl's law can also be expressed as a formula for calculating speedup of an application with multiple work rates. In this formula P is the portion of the workload that is executed in parallel and S is the amount of speedup in the parallel region.

\[
\text{Overall speedup} = \frac{1}{(1-P)+\frac{P}{S}}
\]

This formula can also be used to calculate the maximum possible speedup when parallelizing only part of an application.

\[
\lim_{s \to \infty} \frac{1}{(1-P)+\frac{P}{S}} = \frac{1}{(1-P)}
\]

The question then becomes, how much of the workload in Edge has been parallelized? We can estimate this by comparing the measured speedup with the maximum possible speedup with a given number of processors.

\[
P \approx \frac{1}{S} - 1
\]

Using the numbers for the highest speedups that have been achieved in these tests P was estimated for the different parts of the application. Based on this estimate we can also calculate the maximum speedup that can be achieved when parallelizing this portion of the program by plugging the estimated value of P into the formula for maximum possible speedup.

<table>
<thead>
<tr>
<th>Program</th>
<th>Parallelized portion</th>
<th>Possible speedup with infinite processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edge</td>
<td>0.618</td>
<td>2.61</td>
</tr>
<tr>
<td>Artvis</td>
<td>0.828</td>
<td>5.80</td>
</tr>
<tr>
<td>Convflux</td>
<td>0.658</td>
<td>2.93</td>
</tr>
</tbody>
</table>

This shows two things. For one any improvement to the implementation of the parallel portion would not give a significant improvement in the overall performance of the application. It also explains why the GPU based implementation does not perform significantly better than the OpenMP backend. When the OpenMP backend already performs at close to maximum possible speedup increasing the number of threads that run in parallel only has a marginal effect on the total performance. To show the plausibility of these estimates we can also compare them to the results of the original profiling of
the Edge application. In this profiling the Artvis function accounted for 29% of the execution time and the Convflux function accounted for 19%. Achieving a two times speedup of the overall application when parallelizing these two functions is in line with what could be expected.

**Figure 4-7. Speedup comparison of the Artvis function for OpenMP and Cuda backends**

### 4.4 Multigrid experiment

When using Edge there is an option to enable or disable multigrid. Multigrid is often used as an alternative way to speed up the application without using parallelization. When measuring the speedup of an implementation multigrid is typically turned off as it can potentially lower the relative performance increase. When running Edge with multigrid enabled the application is effectively using multiple problem sizes during the same run. Figure 4-8 shows a comparison of the speedup achieved with the CUDA backend with and without multigrid. We can note that running Edge without multigrid yields a higher speedup. This difference in speedup does however decrease as the problem size becomes larger. This is due to the coarser grids becoming larger while the disparity in speedups between problem sizes becomes smaller. The values acquired with multigrid disabled are more accurate in terms of how determining how the performance of the implementation changes with the problem size. Due to this the speedup comparisons in the previous section have been made with multigrid disabled.
Figure 4-8. Speedup comparison with and without multigrid
Chapter 5

Related work

Recently there have been several thesis projects conducted at Linköping University related to SkePU. The most recent of these is the master thesis by Cuello [17]. The thesis has worked on incorporating support for the Movidius Myriad1 platform into SkePU. A new type of backend has been added to the SkePU library to allow software built for the Myriad1 to make use of SkePU. A functioning backend was developed but future work will be required before it is feasible to use in practice.

Another thesis more similar to this one was carried out by Sundin [18]. In her thesis on the adaption of algorithms used in sonar data processing a number of algorithms are ported to make use of GPUs. A comparison is made between the performance of CUDA, OpenCL and SkePU implementations. Sundin brings up the problems that arise due to SkePU only allowing a limited number of inputs and outputs, similarly to what was encountered during this thesis. In that case the problem could be circumvented for one of the algorithms by the addition of a new macro definition for the user function. The other algorithms were at the time not implemented in SkePU. The additions made in this thesis could potentially allow for the porting of these types of algorithms into SkePU as well.

Another extension of SkePU was the topic of the master thesis by Mangaraj Fel: Det gick inte att hitta referenskällan. In the thesis the matrix container is extended to allow data to be distributed across a cluster of CPUs with the help of MPI. The implementation is tested with the help of an application for solving partial differential equations. These tests show promising results for the viability of using SkePU also on this type of cluster architecture.

Work in similar areas is also taking place at other universities. An example of work done in a similar area at Universidad de La Laguna is DPSKEL [20]. DPSKEL is a programming skeleton for dynamic programming. It was implemented in CUDA and enables the user to solve certain sets of dynamic programming problems by providing a sequential description of the problem in C++. The performance of this skeleton was extensively tested on various platforms [21]. This skeleton is a continuation of work previously done to create a tool for parallel dynamic programming [22]. This work was based on automata theory which allowed the implementation to be as generic as possible. This was done by viewing dynamic programming as multistage finite automata.

A library more closely related to SkePU is SkelCL [23], a skeleton library implemented in OpenCL. Unlike SkePU each skeleton has only one single implementation. This means that the library is more tightly coupled with the way OpenCL programs interact with the GPU. SkelCL has some overlap with SkePU in the skeletons currently implemented but there are some differences in their functionality. For instance, in SkePU the number of input arguments of the user functions are limited whereas SkelCL allow an arbitrary number of arguments. In general SkelCL is more limited in that it offers fewer skeletons, data containers and implementation types. In return SkelCL is more flexible in terms of memory management as well as the types of inputs each skeleton can handle. This is mainly a byproduct of the fact that SkelCL is targeted exclusively at OpenCL while SkePU needs to take into account the different types of implementations of each skeleton.

Another important part of the project is how to handle the use of an unstructured mesh and indirect addressing. Charles Ferenbaugh addresses parts of the problem in his work [24]. The article brings up
a GPU version of an unstructured mesh kernel implemented with CUDA. To minimize the cost of data movement between the CPU and GPU the data movement is overlapped with computations. A chunk of data is moved onto the GPU, computations are made and the corresponding output is then sent back to the CPU. To overcome data locality issues three different strategies are investigated.

- Reordering of data on the CPU and moving it onto the GPU
- Moving data onto the GPU and then reordering it
- Moving data onto the GPU and relying on hardware caches to enhance performance

A different approach to the subject is brought up in [25]. An unstructured mesh compressible flow solver is deployed on a GPU. To improve its performance the calculations are refactored to minimize the number of memory accesses and increase the number of floating point operations.

Another interesting project in the field of parallelization is the Galois system [26]. The Galois system is similar to skeleton programming in that it attempts to provide the programmer with a sequential interface. While the goal of not exposing the programmer to problems such as communication and synchronization is the same as in skeleton programming the underlying techniques used are very different. The Galois system works by utilizing speculative execution to enable parallelization of code that can not ordinarily be parallelized. This works in a similar fashion to how advanced single core architectures will often use speculative execution to utilize further instruction level parallelism.
Chapter 6

SkePU feedback

The main drawback of skeleton programming is that many applications and algorithms do not fit into the available skeletons. In SkePU there is also a problem with the flexibility of many of the skeleton implementations, partially this is due to the portability requirements. Another reason is that the underlying architecture of the SkePU user functions makes the skeletons lack in flexibility. To be able to map as many programs as possible to the SkePU skeletons they need to allow for more variations of the function parameters. A move away from the static design of the C precompiler macros could be very beneficial in achieving this.

During the course of the project both of these issues became apparent as several attempted solutions proved to not be feasible. Algorithmic Skeletons rely on the regularity and structure that most programs are based upon. When working on an unstructured grid as the case is with Edge it becomes harder to find ways to map computations in a regular way. Investigating flow solvers that work on structured grids may prove to be more rewarding. Applications with more regular access patterns would allow the execution of the skeletons to become more optimized. Optimizing data accesses is especially important when working with GPUs due to the underlying architecture.

The solution for Edge presented in this thesis does not match the specifics of the computations in the Edge code. Rather this is a very general solution that can encompass a lot of different kinds of programs. The downside of this is that is not the most efficient solution. In maparray every element of the input vectors are made available inside the user function, each call to the Artvis and Convflux user functions will however only use a small number of elements from each input vector. Ideally a solution would be designed such that only the elements that are actually used are passed as inputs to the user function. This could be done by designing a more flexible variant of the MapOverlap skeleton. The interface of such an implementation could look like

\[ \text{MapOverlap}(\text{ArrayList, OffsetList, ID}) \]

Where the ArrayList parameter contains all the input arrays used by the function, OffsetList contains the offset to the neighbours of the node that the calculations revolve around and the ID parameter would be the index of this node. To actually implement such an implementation within the SkePU framework would however pose many challenges. One problem would be how to handle the parameter passing while still conforming to the multiple backends that SkePU aims to support.

One addition that could improve the current solution is integration with the SkePU smart containers. In the case of Edge this would be problematic to implement due the interaction between the C++ and Fortran code. In the case of a pure C++ program integration with smart containers would be more viable. One way to do this would be to make the multi-vector aware of a potential overlap between the data inside the multi-vector and data that resides within a smart container. Transfers of this data could then be handled outside of the call to maparray by the programmer which would allow for further optimization.
Another issue with SkePU in its current form is the design of the header files. SkePU works by including the header files of the data structures and skeletons being used in the program. When working with a larger project consisting of multiple files this poses a problem because the header files contain not only the declarations of the data structures and skeletons but also their definitions. This means that including the same header file in multiple places causes compilation errors. The programs currently implemented with SkePU are fairly small which is likely why this problem has not been encountered previously. If SkePU aims to be more widely used this needs to be redesigned to make it possible to use SkePU also in larger projects without requiring tedious workarounds.
Chapter 7

Conclusions

When evaluating the project it is important to look back at the goals that were set at the start of the project. Have we achieved what we set out to?

Project goals
- Porting the most computationally intensive subroutines of Edge to C++.
- Rewriting the code by equivalent combinations of SkePU skeletons.
- Evaluating the result and optimizing the implementation.
- Investigate performance portability for different GPU configurations.
- Providing feedback and suggestions for the further design of SkePU.

The two subroutines that use the largest portion of the applications total execution time have been ported from Fortran to C++. These functions were then parallelized with the help of SkePU. To make this possible there was a modification of the maparray skeleton which allows a skeleton call to be made with an arbitrary number of read only inputs. This accounts for the first two goals of the project. Further the performance of the new Edge implementation has been evaluated. When considering the amount of overhead generated by SkePU during sequential execution and the necessary overhead from data transfers when using a GPU the best solution for Edge is likely to use a pure OpenMP implementation for parallelization. Despite these issues there is a substantial gain in execution time achieved by the SkePU implementations and as the library is further developed these types of applications can be a viable target for parallelization with SkePU. Throughout the project an effort has been made to optimize the code in order to get the best possible performance. In the best case scenario the speedup compared to the original Fortran code is roughly 15%.

One goal that turned out to be more difficult to accomplish than initially expected was investigating how the applications performs for different hardware configurations. Attempts were made to evaluate the performance on other platforms but this has not been possible. Due to the nature of the licensing agreement for the Edge code certain demands need to be met regarding the testing environment. This limits the possibilities of testing the application. An attempt was made at using the Erik GPU cluster at LUNARC for testing but due to problems with their file system which are yet to be resolved no such tests were possible.

In this project some of the potential weaknesses of the SkePU framework have been encountered. One of these is the lacking flexibility of the macros used to define the user functions of SkePU. In terms of feedback other than what has been brought up in this thesis the meetings that have taken place during the course of the project have allowed for reporting encountered bugs and other issues to be brought up.
Appendix A

Source code for the new maparray implementation

This appendix contains the code for the maparray implementation that has been adjusted to be able to take an arbitrary number of read only inputs.

A.1 Multi-vector Structure

```c
struct Arr
{
    void *host_el;
    size_t size;
    void *device_el;
};

struct MultiVector
{
    Arr *host_arr;
    size_t size;
    Arr *device_arr;

    Arr * allocData(int numinputs)
    {
        Arr *data = (Arr *) malloc(numinputs*sizeof(Arr));
        host_arr = data;
        size = numinputs;
        return data;
    }

    void addData(int index, size_t size, void * data)
    {
        host_arr[index].host_el = data;
        host_arr[index].size = size;
    }

    void freeData()
    {
        free(host_arr);
    }
};
```

A.2 Operator Macro

```c
#define VAR_FUNC(name, inputtype, outputtype, containertype, param1, param2, func)\
\struct name\
{
  \skepu::FuncType funcType;
  bool isConst;\
  name()\
  {
    funcType = skepu::UNARY;\
    isConst = false;\
  }\
  outputtype CPU(inputtype param1, containertype param2)\
```
A.3 CPU implementation

```cpp
{\func\n  \_device__ outputtype CU(inputtype param1, containertype param2)\n  {\func\n  }\n};
```

```cpp
A.3 CPU implementation

template <typename MapArrayFunc>
template <typename in, typename out>
void MapArray<MapArrayFunc>::CPU(Vector<in>& input, MultiVector P, Vector<out>& output)
{
    input.updateHost();
    P.device_arr = P.host_arr;
    for (int i=0;i<P.size;i++)
    {
        P.device_arr[i].device_el = P.device_arr[i].host_el;
    }

    output.invalidateDeviceData();
    int size = input.size();
    for(int i=0;i<size;i++)
    {
        output[i] = m_mapArrayFunc->CPU(input[i], P);
    }
}
```

A.4 OpenMP implementation

```cpp
A.4 OpenMP implementation

template <typename MapArrayFunc>
template <typename InputIterator, typename OutputIterator>
void MapArray<MapArrayFunc>::OMP(InputIterator inputBegin, InputIterator inputEnd,
MultiVector P, OutputIterator outputBegin)
{
    size_t n = inputEnd - inputBegin;
    omp_set_num_threads(m_execPlan->numOmpThreads(n));

    //Make sure we are properly synched with device data
    outputBegin.getParent().invalidateDeviceData();
    inputBegin.getParent().updateHost();
    inputBegin.getParent().updateHost();

    P.device_arr = P.host_arr;
    for (int i=0;i<P.size;i++)
    {
        P.device_arr[i].device_el = P.device_arr[i].host_el;
    }

    #pragma omp parallel for
    for(size_t i = 0; i < n; ++i)
    {
        outputBegin(i) = m_mapArrayFunc->CPU(inputBegin(i), P);
    }
}
```
A.5 Cuda implementation

```c
void cudacheck(cudaError_t error, char* message)
{
    if(error!=cudaSuccess) { fprintf(stderr,"ERROR: %s:
%s\n",message,cudaGetErrorString(error)); exit(-1); }
}

namespace skepu
{
    template <typename MapArrayFunc>
    template <typename InputIterator, typename OutputIterator>
    void MapArray<MapArrayFunc>::CU(InputIterator inputBegin, InputIterator inputEnd,
    MultiVector P, OutputIterator outputBegin)
    {
        int deviceID = m_environment->bestCUDADevID;
        CHECK_CUDA_ERROR(cudaSetDevice(deviceID));
        clock_t t;
        t = clock();
        // move MultiVector structure to gpu
        for (int i=0;i<P.size;i++)
        {
            cudacheck(cudaMalloc((void**)&P.host_arr[i].device_el,P.host_arr[i].size),"malloc host_arr");
            cudacheck(cudaMemcpy(P.host_arr[i].device_el,P.host_arr[i].host_el,P.host_arr[i].size,cudaMemcpyHostToDevice),"memcpy host_arr");
        }
        t = clock() - t;
        t = clock();
        cudacheck(cudaMalloc((void**)&P.device_arr,P.size*sizeof(Arr),"malloc MultiVector");
        cudacheck(cudaMemcpy(P.device_arr,P.host_arr,P.size*sizeof(Arr),cudaMemcpyHostToDevice),"memcpy MultiVector");
        t = clock() - t;
        // Setup parameters
        size_t n = inputEnd-inputBegin;
        BackEndParams bp=m_execPlan->find_(n);
        size_t maxBlocks = bp.maxBlocks;
        size_t maxThreads = bp.maxThreads;
        size_t numBlocks;
        size_t numThreads;
        numThreads = std::min(maxThreads, n);
        numBlocks = std::min((size_t)1, std::min( (n/numThreads + (n%numThreads == 0 ? 0:1)), maxBlocks));
        // Copies the elements to the device
        typename InputIterator::device_pointer_type_cu in_mem_p = inputBegin.getParent().updateDevice_CU( inputBegin.getAddress(), n, deviceID, true, false);
        MapArrayKernel_VAR<<<numBlocks,numThreads>>>("m_mapArrayFunc, in_mem_p->getDeviceDataPointer(),P, out_mem_p->getDeviceDataPointer(), n);

        out_mem_p->changeDeviceData();
        t = clock();
        out_mem_p->copyDeviceToHost();
        t = clock() - t;
        // free gpu memory
        for (int i=0;i<P.size;i++)
```
cudacheck(cudaFree(P.host_arr[i].device_el),"free data");
cudacheck(cudaFree(P.device_arr),"free MultiVector");
}

template <typename in, typename out, typename ArrayFunc>
__global__ void MapArrayKernel_VAR(ArrayFunc mapArrayFunc, in* input, MultiVector P, out* output, size_t n)
{
    size_t i = blockIdx.x * blockDim.x + threadIdx.x;
    size_t gridSize = blockDim.x*gridDim.x;

    while(i < n)
    {
        output[i] = mapArrayFunc.CU(input[i], P);
        i += gridSize;
    }
}
**Appendix B**

**Complete results table**

This appendix contains the complete results table for the experiment that was conducted to measure the performance of the application.

**B.1 Execution times**

<table>
<thead>
<tr>
<th>Testcase</th>
<th>Multigrid</th>
<th>Implementation</th>
<th>Function</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>rae_case10</td>
<td>ON</td>
<td>C++</td>
<td>main</td>
<td>63</td>
</tr>
<tr>
<td>rae_case10</td>
<td>ON</td>
<td>C++</td>
<td>artvis</td>
<td>10.451</td>
</tr>
<tr>
<td>rae_case10</td>
<td>ON</td>
<td>C++</td>
<td>convflux</td>
<td>9.479</td>
</tr>
<tr>
<td>rae_case10</td>
<td>ON</td>
<td>sequential SkePU</td>
<td>main</td>
<td>87</td>
</tr>
<tr>
<td>rae_case10</td>
<td>ON</td>
<td>sequential SkePU</td>
<td>artvis</td>
<td>13.919</td>
</tr>
<tr>
<td>rae_case10</td>
<td>ON</td>
<td>sequential SkePU</td>
<td>convflux</td>
<td>26.804</td>
</tr>
<tr>
<td>rae_case10</td>
<td>ON</td>
<td>cuda</td>
<td>main</td>
<td>95</td>
</tr>
<tr>
<td>rae_case10</td>
<td>ON</td>
<td>cuda</td>
<td>artvis</td>
<td>5.325</td>
</tr>
<tr>
<td>rae_case10</td>
<td>ON</td>
<td>cuda</td>
<td>convflux</td>
<td>38.463</td>
</tr>
<tr>
<td>rae_case10</td>
<td>ON</td>
<td>openmp</td>
<td>main</td>
<td>78</td>
</tr>
<tr>
<td>rae_case10</td>
<td>ON</td>
<td>openmp</td>
<td>artvis</td>
<td>3.908</td>
</tr>
<tr>
<td>rae_case10</td>
<td>ON</td>
<td>openmp</td>
<td>convflux</td>
<td>19.811</td>
</tr>
<tr>
<td>rae_case10</td>
<td>OFF</td>
<td>C++</td>
<td>main</td>
<td>5</td>
</tr>
<tr>
<td>rae_case10</td>
<td>OFF</td>
<td>C++</td>
<td>artvis</td>
<td>1.808</td>
</tr>
<tr>
<td>rae_case10</td>
<td>OFF</td>
<td>C++</td>
<td>convflux</td>
<td>1.968</td>
</tr>
<tr>
<td>rae_case10</td>
<td>OFF</td>
<td>sequential SkePU</td>
<td>main</td>
<td>10</td>
</tr>
<tr>
<td>rae_case10</td>
<td>OFF</td>
<td>sequential SkePU</td>
<td>artvis</td>
<td>2.512</td>
</tr>
<tr>
<td>rae_case10</td>
<td>OFF</td>
<td>sequential SkePU</td>
<td>convflux</td>
<td>4.702</td>
</tr>
<tr>
<td>rae_case10</td>
<td>OFF</td>
<td>cuda</td>
<td>main</td>
<td>7</td>
</tr>
<tr>
<td>rae_case10</td>
<td>OFF</td>
<td>cuda</td>
<td>artvis</td>
<td>0.751</td>
</tr>
<tr>
<td>rae_case10</td>
<td>OFF</td>
<td>cuda</td>
<td>convflux</td>
<td>4.088</td>
</tr>
<tr>
<td>rae_case10</td>
<td>OFF</td>
<td>openmp</td>
<td>main</td>
<td>6</td>
</tr>
<tr>
<td>rae_case10</td>
<td>OFF</td>
<td>openmp</td>
<td>artvis</td>
<td>0.666</td>
</tr>
<tr>
<td>rae_case10</td>
<td>OFF</td>
<td>openmp</td>
<td>convflux</td>
<td>3.502</td>
</tr>
<tr>
<td>Static Mixer</td>
<td>ON</td>
<td>C++</td>
<td>main</td>
<td>15</td>
</tr>
<tr>
<td>Module</td>
<td>Status</td>
<td>Backend</td>
<td>Mode</td>
<td>Module</td>
</tr>
<tr>
<td>---------------</td>
<td>--------</td>
<td>---------</td>
<td>-------</td>
<td>--------</td>
</tr>
<tr>
<td>Static Mixer</td>
<td>ON</td>
<td>C++</td>
<td>artvis</td>
<td>1.503</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>C++</td>
<td>convflux</td>
<td>2.447</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>sequential SkePU</td>
<td>main</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>sequential SkePU</td>
<td>artvis</td>
<td>1.994</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>sequential SkePU</td>
<td>convflux</td>
<td>8.012</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>cuda</td>
<td>main</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>cuda</td>
<td>artvis</td>
<td>0.743</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>cuda</td>
<td>convflux</td>
<td>11.096</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>openmp</td>
<td>main</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>openmp</td>
<td>artvis</td>
<td>0.528</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>openmp</td>
<td>convflux</td>
<td>6.101</td>
</tr>
<tr>
<td></td>
<td>OFF</td>
<td>C++</td>
<td>main</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>OFF</td>
<td>C++</td>
<td>artvis</td>
<td>0.746</td>
</tr>
<tr>
<td></td>
<td>OFF</td>
<td>C++</td>
<td>convflux</td>
<td>0.462</td>
</tr>
<tr>
<td></td>
<td>OFF</td>
<td>sequential SkePU</td>
<td>main</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>OFF</td>
<td>sequential SkePU</td>
<td>artvis</td>
<td>0.995</td>
</tr>
<tr>
<td></td>
<td>OFF</td>
<td>sequential SkePU</td>
<td>convflux</td>
<td>3.226</td>
</tr>
<tr>
<td></td>
<td>OFF</td>
<td>cuda</td>
<td>main</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>OFF</td>
<td>cuda</td>
<td>artvis</td>
<td>0.372</td>
</tr>
<tr>
<td></td>
<td>OFF</td>
<td>cuda</td>
<td>convflux</td>
<td>3.1</td>
</tr>
<tr>
<td></td>
<td>OFF</td>
<td>openmp</td>
<td>main</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>OFF</td>
<td>openmp</td>
<td>artvis</td>
<td>0.265</td>
</tr>
<tr>
<td></td>
<td>OFF</td>
<td>openmp</td>
<td>convflux</td>
<td>2.75</td>
</tr>
<tr>
<td>Wing-body</td>
<td>ON</td>
<td>C++</td>
<td>main</td>
<td>523</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>C++</td>
<td>artvis</td>
<td>183.727</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>C++</td>
<td>convflux</td>
<td>160.092</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>sequential SkePU</td>
<td>main</td>
<td>752</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>sequential SkePU</td>
<td>artvis</td>
<td>249.021</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>sequential SkePU</td>
<td>convflux</td>
<td>315.834</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>cuda</td>
<td>main</td>
<td>384</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>cuda</td>
<td>artvis</td>
<td>45.907</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>cuda</td>
<td>convflux</td>
<td>134.383</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>openmp</td>
<td>main</td>
<td>368</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>openmp</td>
<td>artvis</td>
<td>56.371</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>openmp</td>
<td>convflux</td>
<td>121.623</td>
</tr>
<tr>
<td></td>
<td>OFF</td>
<td>C++</td>
<td>main</td>
<td>241</td>
</tr>
<tr>
<td>Scenario</td>
<td>Condition</td>
<td>Language</td>
<td>Module</td>
<td>Value</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------</td>
<td>----------</td>
<td>----------</td>
<td>---------</td>
</tr>
<tr>
<td>Wing-body</td>
<td>OFF</td>
<td>C++</td>
<td>artvis</td>
<td>91.797</td>
</tr>
<tr>
<td>Wing-body</td>
<td>OFF</td>
<td>C++</td>
<td>convflux</td>
<td>62.39</td>
</tr>
<tr>
<td>Wing-body</td>
<td>OFF</td>
<td>sequential SkePU</td>
<td>main</td>
<td>350</td>
</tr>
<tr>
<td>Wing-body</td>
<td>OFF</td>
<td>sequential SkePU</td>
<td>artvis</td>
<td>124.238</td>
</tr>
<tr>
<td>Wing-body</td>
<td>OFF</td>
<td>sequential SkePU</td>
<td>convflux</td>
<td>161.579</td>
</tr>
<tr>
<td>Wing-body</td>
<td>OFF</td>
<td>cuda</td>
<td>main</td>
<td>163</td>
</tr>
<tr>
<td>Wing-body</td>
<td>OFF</td>
<td>cuda</td>
<td>artvis</td>
<td>24.063</td>
</tr>
<tr>
<td>Wing-body</td>
<td>OFF</td>
<td>cuda</td>
<td>convflux</td>
<td>69.661</td>
</tr>
<tr>
<td>Wing-body</td>
<td>OFF</td>
<td>openmp</td>
<td>main</td>
<td>160</td>
</tr>
<tr>
<td>Wing-body</td>
<td>OFF</td>
<td>openmp</td>
<td>artvis</td>
<td>28.658</td>
</tr>
<tr>
<td>Wing-body</td>
<td>OFF</td>
<td>openmp</td>
<td>convflux</td>
<td>61.868</td>
</tr>
<tr>
<td>Heating Coil</td>
<td>ON</td>
<td>C++</td>
<td>main</td>
<td>99</td>
</tr>
<tr>
<td>Heating Coil</td>
<td>ON</td>
<td>C++</td>
<td>artvis</td>
<td>14.877</td>
</tr>
<tr>
<td>Heating Coil</td>
<td>ON</td>
<td>C++</td>
<td>convflux</td>
<td>25.915</td>
</tr>
<tr>
<td>Heating Coil</td>
<td>ON</td>
<td>sequential SkePU</td>
<td>main</td>
<td>146</td>
</tr>
<tr>
<td>Heating Coil</td>
<td>ON</td>
<td>sequential SkePU</td>
<td>artvis</td>
<td>21.174</td>
</tr>
<tr>
<td>Heating Coil</td>
<td>ON</td>
<td>sequential SkePU</td>
<td>convflux</td>
<td>63.046</td>
</tr>
<tr>
<td>Heating Coil</td>
<td>ON</td>
<td>cuda</td>
<td>main</td>
<td>111</td>
</tr>
<tr>
<td>Heating Coil</td>
<td>ON</td>
<td>cuda</td>
<td>artvis</td>
<td>4.691</td>
</tr>
<tr>
<td>Heating Coil</td>
<td>ON</td>
<td>cuda</td>
<td>convflux</td>
<td>43.173</td>
</tr>
<tr>
<td>Heating Coil</td>
<td>ON</td>
<td>openmp</td>
<td>main</td>
<td>102</td>
</tr>
<tr>
<td>Heating Coil</td>
<td>ON</td>
<td>openmp</td>
<td>artvis</td>
<td>5.087</td>
</tr>
<tr>
<td>Heating Coil</td>
<td>ON</td>
<td>openmp</td>
<td>convflux</td>
<td>33.278</td>
</tr>
<tr>
<td>Heating Coil</td>
<td>OFF</td>
<td>C++</td>
<td>main</td>
<td>19</td>
</tr>
<tr>
<td>Heating Coil</td>
<td>OFF</td>
<td>C++</td>
<td>artvis</td>
<td>7.588</td>
</tr>
<tr>
<td>Heating Coil</td>
<td>OFF</td>
<td>C++</td>
<td>convflux</td>
<td>5.023</td>
</tr>
<tr>
<td>Heating Coil</td>
<td>OFF</td>
<td>sequential SkePU</td>
<td>main</td>
<td>30</td>
</tr>
<tr>
<td>Heating Coil</td>
<td>OFF</td>
<td>sequential SkePU</td>
<td>artvis</td>
<td>10.397</td>
</tr>
<tr>
<td>Heating Coil</td>
<td>OFF</td>
<td>sequential SkePU</td>
<td>convflux</td>
<td>13.891</td>
</tr>
<tr>
<td>Heating Coil</td>
<td>OFF</td>
<td>cuda</td>
<td>main</td>
<td>17</td>
</tr>
<tr>
<td>Heating Coil</td>
<td>OFF</td>
<td>cuda</td>
<td>artvis</td>
<td>2.389</td>
</tr>
<tr>
<td>Heating Coil</td>
<td>OFF</td>
<td>cuda</td>
<td>convflux</td>
<td>8.653</td>
</tr>
<tr>
<td>Heating Coil</td>
<td>OFF</td>
<td>openmp</td>
<td>main</td>
<td>16</td>
</tr>
<tr>
<td>Heating Coil</td>
<td>OFF</td>
<td>openmp</td>
<td>artvis</td>
<td>2.574</td>
</tr>
<tr>
<td>Heating Coil</td>
<td>OFF</td>
<td>openmp</td>
<td>convflux</td>
<td>7.403</td>
</tr>
<tr>
<td>yf17</td>
<td>OFF</td>
<td>C++</td>
<td>main</td>
<td>351</td>
</tr>
<tr>
<td></td>
<td>Mode</td>
<td>Language</td>
<td>Module</td>
<td>Time</td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
<td>----------</td>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>yf17</td>
<td>OFF</td>
<td>C++</td>
<td>artvis</td>
<td>56.501</td>
</tr>
<tr>
<td>yf17</td>
<td>OFF</td>
<td>C++</td>
<td>convflux</td>
<td>107.309</td>
</tr>
<tr>
<td>yf17</td>
<td>ON</td>
<td>sequential SkePU</td>
<td>main</td>
<td>548</td>
</tr>
<tr>
<td>yf17</td>
<td>ON</td>
<td>sequential SkePU</td>
<td>artvis</td>
<td>83.951</td>
</tr>
<tr>
<td>yf17</td>
<td>ON</td>
<td>sequential SkePU</td>
<td>convflux</td>
<td>265.959</td>
</tr>
<tr>
<td>yf17</td>
<td>ON</td>
<td>cuda</td>
<td>main</td>
<td>340</td>
</tr>
<tr>
<td>yf17</td>
<td>ON</td>
<td>cuda</td>
<td>artvis</td>
<td>15.025</td>
</tr>
<tr>
<td>yf17</td>
<td>ON</td>
<td>cuda</td>
<td>convflux</td>
<td>119.879</td>
</tr>
<tr>
<td>yf17</td>
<td>ON</td>
<td>openmp</td>
<td>main</td>
<td>319</td>
</tr>
<tr>
<td>yf17</td>
<td>ON</td>
<td>openmp</td>
<td>artvis</td>
<td>17.023</td>
</tr>
<tr>
<td>yf17</td>
<td>ON</td>
<td>openmp</td>
<td>convflux</td>
<td>102.726</td>
</tr>
<tr>
<td>yf17</td>
<td>OFF</td>
<td>C++</td>
<td>main</td>
<td>66</td>
</tr>
<tr>
<td>yf17</td>
<td>OFF</td>
<td>C++</td>
<td>artvis</td>
<td>28.419</td>
</tr>
<tr>
<td>yf17</td>
<td>OFF</td>
<td>C++</td>
<td>convflux</td>
<td>20.101</td>
</tr>
<tr>
<td>yf17</td>
<td>OFF</td>
<td>sequential SkePU</td>
<td>main</td>
<td>114</td>
</tr>
<tr>
<td>yf17</td>
<td>OFF</td>
<td>sequential SkePU</td>
<td>artvis</td>
<td>41.314</td>
</tr>
<tr>
<td>yf17</td>
<td>OFF</td>
<td>sequential SkePU</td>
<td>convflux</td>
<td>52.515</td>
</tr>
<tr>
<td>yf17</td>
<td>OFF</td>
<td>cuda</td>
<td>main</td>
<td>51</td>
</tr>
<tr>
<td>yf17</td>
<td>OFF</td>
<td>cuda</td>
<td>artvis</td>
<td>7.52</td>
</tr>
<tr>
<td>yf17</td>
<td>OFF</td>
<td>cuda</td>
<td>convflux</td>
<td>23.062</td>
</tr>
<tr>
<td>yf17</td>
<td>OFF</td>
<td>openmp</td>
<td>main</td>
<td>48</td>
</tr>
<tr>
<td>yf17</td>
<td>OFF</td>
<td>openmp</td>
<td>artvis</td>
<td>8.527</td>
</tr>
<tr>
<td>yf17</td>
<td>OFF</td>
<td>openmp</td>
<td>convflux</td>
<td>20.54</td>
</tr>
</tbody>
</table>
References


