Bachelor thesis

An Intel® Xeon Phi™ Backend for the ExaStencils Code Generator

Thomas Lang

Supervisor: Prof. Christian Lengauer, Ph.D.
Tutor: Dr. Armin Größlinger

27th April 2016

Abstract

Stencil computations are an essential part of scientific calculations especially for numerical solving of partial differential equations and are a field of ongoing research. The ExaStencils project provides a high-level way to describe such calculations by defining a domain specific langugage. This abstract way includes a code generator that is able to transform an input written in the domain specific language into a general-purpose language like C++ which can be executed on a target hardware. However, the performance of the generated code varies for different platforms.

In this thesis, we focus on a specific target hardware, the Intel® Xeon Phi™ coprocessor, a many-core coprocessor card for executing highly parallel code. Firstly, we give an overview about this specific hardware. Next, we describe high-level optimizations for enhancing performance, like vectorization, that were added to the code generator and the corresponding implementation details. Empirical experiments measured on this coprocessor show the significant impact on performance of our optimizations compared to the original generated code.

Acknowledgements
This thesis has only been made possibly through the support of many people.

First of all, I want to thank Prof. Christian Lengauer, Ph.D., who gave me the opportunity of working on a part of the ExaStencils project. So I could gain some insights in real-life high performance computing and in the state-of-the-art of these fields.

Next, I want to thank my tutor Dr. Armin Größlinger, who supported me in many points like gaining me access to the cluster network used by the chair of programming, setting up the working environment and answering numerous questions about the project.

Furthermore, I am grateful for support from Stefan Kronawitter MSc., who introduced me into the current code generator and answered several questions about existing features or where potential for optimizations lies.

Lastly, I appreciate the remaining team of the chair of programming for giving me various hints on code optimization.

Contents

List of Figures

Listings

Chapter 1

Introduction

The ExaStencils project is a new approach in reaching exascale performance for multigrid stencil computations. This goal is reached by focussing on a specific domain and optimize applications using domain specific expert knowledge.

The chosen domain for this project are stencil computations, which are very important for a wide range of scientific computations, especially for solving partial differential equations (PDEs).

The main challenge here lies in generating efficient code that can solve such PDEs. To make the program usable for non-experts in programming and still ensure accuracy and efficiency, it was decided to create a *domain specific langugage* (DSL) named *ExaSlang* to gain the needed level of abstraction. To get executable code that can really solve the given problem, a code generator was written in `Scala`. This program generates a `C++` output that can be compiled and run. During the individual compilation phases, it performs many optimizations either given as input from the user or implied from previous steps to gain a maximum of performance [LGK+14].

In this thesis, we provide a measurement of the runtime of the currently generated code and a way and additions to the code generator to gain a better performance when using the Intel® Xeon Phi™ coprocessor.

The next chapter gives a general overview over the architecture and workflow of both the ExaStencils project and the current code generator.

In chapter 3, we describe performance enhancing methods and implementation details for that.

This is succeeded by an analysis of the runtime of the generated code before and after adding the implementation discussed in this thesis.

Finally, it is concluded how performance can be gained through several adaptions and what further features not added yet may be useful to improve the runtime on the coprocessor.

Chapter 2

Background

2.1 ExaStencils

The ExaStencils project aims for exascale performance when solving partial differential equations numerically. In the following sections, we take a look at the workflow of the project and the code generator itself.

2.1.1 Project Overview

An essential part of scientific calculations is the solving of partial differential equations, like a heat equation which describes the changement of temperatures in a given region over time. In a general coordinate system such an equation has a form of:

$$\frac{\partial U}{\partial t} = \kappa \nabla^2 U$$

where κ is the thermal diffusivity and U is the temperature [Wei16]. A numerical approach to solve this is to represent the region in a multi-dimensional discrete grid and to use stencil computations to compute the solution point-per-point.

A stencil is a point in a multi-dimensional grid whose value can be computed by combining arithmetically itself, its neighboured values and weights. This is schematically shown in Figure 2.1. A stencil computation is the act of applying this calculation to every point in the entire grid [Fre14].

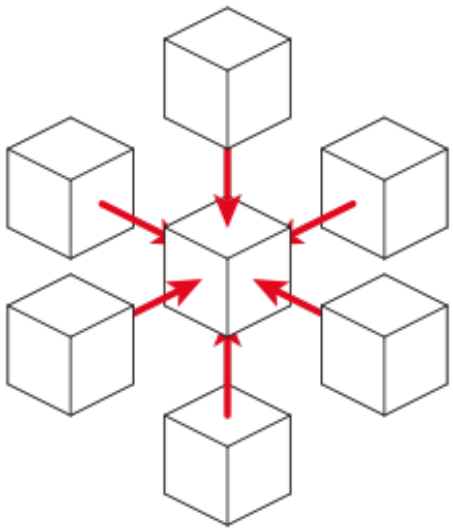

Figure 2.1: A three-dimensional six-point stencil (taken from [Gen13])

Depending on the equation, the grid used for iterative solving can have more than one dimension, e.g. a three-dimensional stencil computation as shown in Figure 2.1 can be realized in a single grid with each point having three-dimensional coordinates. In order to enhance performance, this computations can be done on multiple grids of different cell sizes to reduce memory consumption. This is done by executing *VCycles* which consist of *smoothing*, *coarsening* and *prolongation*. Smoothing is the approximation of the solution of the system of linear equations which reduces the difference between the exact and the calculated solution. To achieve this, different numerical algorithms can be used for smoothing. After smoothing, the remaining system can be represented on a grid of same size but with fewer points what is called coarsening. This process is used recursively until the coarsest level is reached. On this level the equation system can be solved directly. After this, the solution is prolongated to finer levels using interpolation, with smoothing taking place on each level until the finest level is reached [RBK15].

The purpose of this project is the efficient and highly parallel execution of stencil computations.

To achieve that, the project runs through several steps with each of them uses its own optimizations on their specific domain using expert knowledge.
In detail, these steps are [exa16]:

1. Adaption of the mathematical problem
2. Formulation in a DSL
3. Domain-specific optimizations customized for stencil codes
4. Loop optimization in the polyhedral model
5. Platform-specific adaptions

In the very first step the mathematical problem is transformed into a numerically stable, scalable and efficient way. In the second step, this very abstract form is translated into a domain specific language. The resulting program then walks through an entire four layer software stack and is further optimized using expert knowledge on the computability of stencil codes and parallelization. Finally, the generated code is tuned towards the target hardware [exa16].
This workflow can be visualized as in Figure 2.2.

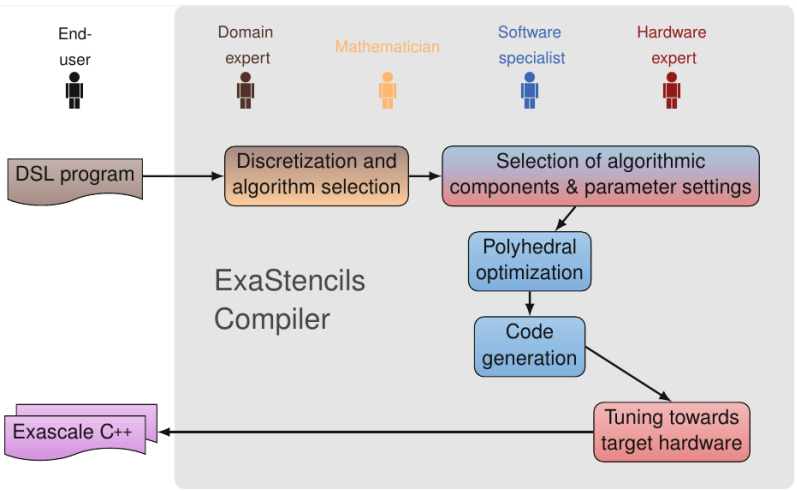

Figure 2.2: Workflow in the project (taken from [LGK+14])

Today, these computations are implemented in any general-purpose language like `C/C++` or `Fortran` and derivates that exploit parallelism, mainly by using `OpenMP` or `MPI`.
However, this implementation can be complicated and different for different domains and platforms and it needs experts in programming to do that. To avoid that, ExaStencils provides the above already mentioned domain specific approach to reduce this complexity [LGK+14].

2.1.2 The ExaSlang DSL

This abstraction ultimately resulted in the creation of the ExaSlang (**ExaS**tencils **lang**uage) domain specific language, used for solving partial differential equations using stencil computations. All together, a four layer language stack is planned, as shown in Figure 2.3.

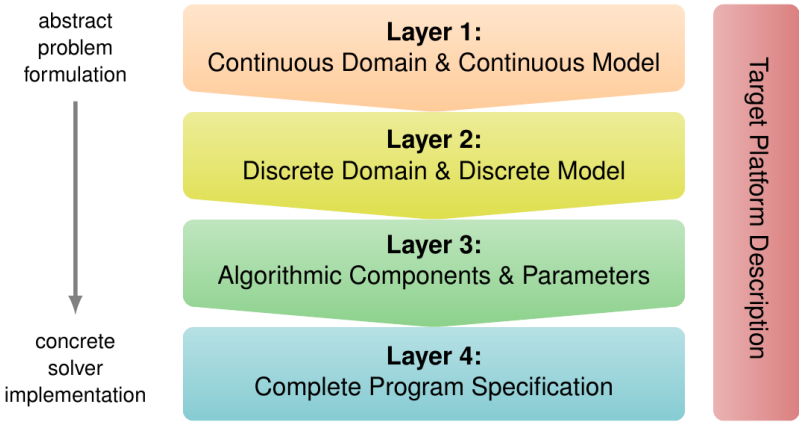

Figure 2.3: ExaSlang software stack (taken from [LGK+14])

This DSL is used for formulating a problem on a very abstract and mathematical oriented level. This is the *Layer 1* DSL. All models and domains in this layer are assumed continuous just like in the mathematical definition.
Next comes the *Layer 2* language that discretizes the domain and the problem to make it computable programmatically. This includes the choice of data types and the first problem transformation.
This is followed by *Layer 3*. This layer specifies mathematical operators, several mathematical techniques for solving (like smoothers) and also the conversion from fields to arrays.
Finally, *Layer 4* describes the complete program specification in a `Scala`-like syntax and the last level in this DSL stack. An example Layer 4 file is shown in Appendix D. This further specifies the operations depending on the several levels of the multigrid computation and the communication between that levels. Also, third-party libraries can be included here and other optimizations can be made.
All these layers are sharing a target hardware description that enables the generator to do different optimizations for several target machines on each layer [KSK+15].
It also should be noted that only the *Layer 4* level exists currently. But for our optimizations for the Intel®Xeon Phi™this is sufficient because this optimizations tune the code towards this specific hardware and do not rely on the above DSL layers.

Because we manipulate the *Layer 4* level only, we take a closer look at this DSL. A simple and very important code example in this DSL is a simple smoother like shown in Listing 2.1.

```
Function SmootherT(): Unit {
    loop over fragments {
        repeat 2 times with contraction [1,1,1] {
            loop over SolutionT@finest {
                SolutionT[nextSlot]@finest = SolutionT[active]@finest
                  + (0.8 / diag(Laplace@finest)
                    * (RHST@finest - Laplace@finest * SolutionT[active]@finest))
            }
            advance SolutionT@finest
        }
    }
}
```

Listing 2.1: Single Jacobi smoother in the Layer 4 DSL

In this example, we see a single Jacobi smoother in the Scala-like syntax of the *Layer 4* DSL. It uses nested loops and specifications on which grids which code must be executed, e.g. `SolutionT[active]@finest` means that the current field of `SolutionT` on the finest grid is accessed. The remaining parts of the code are communication and computational details we will not explain here.

The ExaStencils code generator can process this program and converts it into C++ code. Listing 2.2 shows a simplified excerpt of the generated code.

```
void SmootherT() {
    for (int fragmentIdx = 0; fragmentIdx < 1; ++fragmentIdx) {
        for (int z = 1; z < 577; ++z) {
            for (int y = 1; y < 577; ++y) {
                double* const p1 = &fieldData_RHST[1][346912*z+592*y];
                double* const p2 =
                  &slottedFieldData_SolutionT[currentSlot_SolutionT[1]%2][1][348096*z+592*y];
                double* const p3 =
                  &slottedFieldData_SolutionT[(currentSlot_SolutionT[1]+1)%2][1][348096*z+592*y];

                for (int x = 1; x < 577; ++x) {
                    p3[x+1743448] =
                      (0.19999999999999996*p2[x+1743448])
                        +(0.16666666666666669*p1[x+1390024])
                          +(0.13333333333333336*(p2[x+1395352]
                            +p2[x+1742856]+p2[x+1743447]
                              +p2[x+1743449]+p2[x+1744040]+p2[x+2091544]));
                }
            }
        }
        ...
    }
}
```

Listing 2.2: Single Jacobi smoother transformed into C++ code

As we can see that the constants and field accesses as well as the different dimensions of the generated code are created entirely by the code generator which gives even more abstraction on the *Layer 4* level. We also can observe several optimizations already made by the code generator like the address precalculation.

2.1.3 Code Generator

To manage the four layers of the ExaSlang DSL, a code generator was implemented in `Scala`. This generator reads the input in the domain specific language (currently *Layer 4*). The input is then lexed and parsed into an tree-like intermediate representation. On this intermediate representation, several optimizations are made, e.g. function inlining or dead code elimination. After all these steps, the generated code will finally be transformed into `C++` code[1] and pretty-printed into corresponding files.

There are two possible usages of this generator planned.
On the one hand, one can generate code in individual levels and then manually optimize that code for the specific domain. On the other hand, the code generator does all optimizations. For that, it uses expert knowledge from several domains like software specialists in loop optimization or hardware specialists in tuning towards a specific hardware.

All optimizations done by the code generator are implemented using *strategies*. Each such strategy iterates over the whole program in a *Visitor*-like pattern and affects the desired parts of it. These are replaced with a new state resulted by applying one or more transformations on the old state. This new state replaces the entire sub-tree from the manipulated node, including itself.
We want to discuss a few optimizations relevant for this thesis in more detail: A high-level optimization is the loop optimization using the polyhedral model. As said, this strategy computes the loop and access dependencies, eliminates dead code, searches an optimal schedule, tiles the dimensions and recreates the abstract syntax tree. This aims for loops with a very high potential for parallelism.
Furthermore, the low-level approach of address precalculation can be used. This computes base pointers of array accesses used in the loop and with this, it prevents expensive calculations in the innermost loops that could be done before [KSK+15]. In advance, this leads to simplified inner loops which then have a higher potential for Vectorization.

Later on, we will implement such a strategy for pushing further vectorization specifically for the Intel® Xeon Phi™.

[1] The currently supported `C++` standard is `C++11`.

2.2 Intel®Xeon Phi™

In 2013 [Cor13a], Intel® announced a new manycore system called Intel®Xeon Phi™. As stated before, the generated code does not perform as well on this machine as it would be possible. In this section, we will look at the structure of such a machine and its unique advantages over other existing systems.

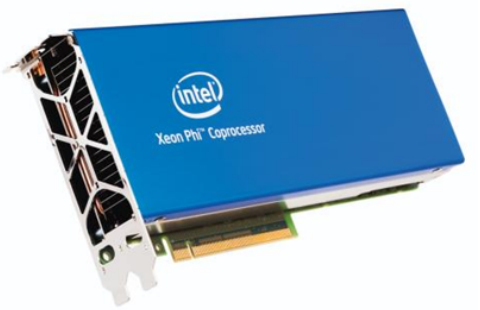

Figure 2.4: An Intel®Xeon Phi™coprocessor

2.2.1 Architecture

The Intel®Xeon Phi™(aka *Many Integrated Core* or MIC card) is a coprocessor, which means it is a card that can be physically installed on an existing Intel® computer. It differs from other systems in that it was not designed to execute serial code, but highly parallel code instead.

In this thesis we use an Intel®Xeon Phi™3120P coprocessor card that has 57 CPU cores with each supporting four threads simultaneously, which makes a maximum of 228 threads at a time. The cores are connected in a bi-directional 512 bit wide ring BUS that interconnects the coherent 512kB L2 caches of the cores. These hardware features are shown schematically in Figure 2.5.

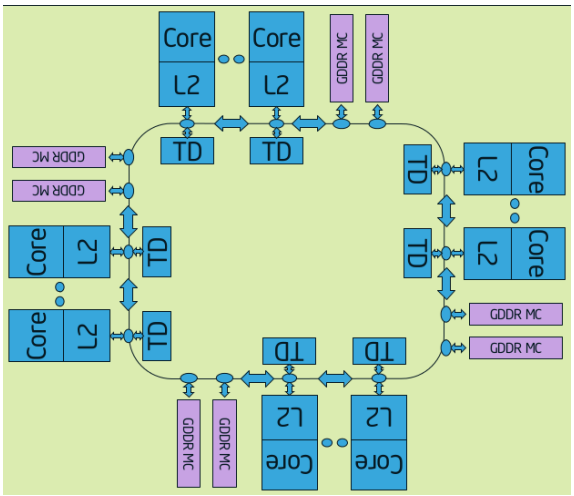

Figure 2.5: Intel®Xeon Phi™core architecture(taken from [Cor12c])

In contrast to other Intel®products, this hardware does not support *out-of-order execution*. This principle enables processors to execute statements in another order than specified by the program. As the Intel®Xeon Phi™does not support that, every instruction is executed like in the program. This may lead to a waste of cycles caused by CPU stalls and should be compensated by using techniques improving instruction throughput like hyperthreading [CD13].

Additionally, there are two pipelines: One for executing scalar and one for executing vectorized code. The vector processing unit supports *Single Instruction Multiple Data* operations. With that more than one data item at a time can be processed in a single instruction [Cor13d].

With knowledge gained from above, we expect a huge improvement in the theoretical peak performance. The coprocessor used for this thesis can process 8 double precision data items on 2 sockets, totalling 16 floating point operations per clock cycle. In total, the theoretical peak performance can be computed as [Cor13c]:

$$\text{PEAK} = 16\frac{\text{FLOPs}}{\text{clock}} \times 57\text{cores} \times 1.1\text{GHz} = 1003.2\frac{\text{GFLOPs}}{\text{sec}}$$

So at maximum about one TFLOP per second is possible when using double precision. In case of using single precision, twice the amount of data items can be processed, so then about two TFLOPS are possible.

2.2.2 Execution Models

As stated earlier, the Intel® Xeon Phi™ is just a card that is physically installed on a machine in addition to the main CPU. In total, there are three different execution models usable:

1. Native execution
2. Offload execution
3. Symmetric execution

In the following sections, we take a closer look at these three models.

Native execution

The coprocessor is able to run its own operating system. This enables users to connect to the coprocessor directly using `ssh` and execute applications. It must be noted, that the Intel® Xeon Phi™ is *not* binary compatible with other Intel® processors, so cross-compilation is needed for using this model.
This model provides the easiest way for porting applications to this hardware. This is as easy by simply adding the switch `-mmic` to the compile and link commands [Rah13].

With this technique, we can easily execute our program on the coprocessor without changing any code. However, this does not mean that the code gets faster by itself. Empirical tests show that the code executed can even get slower! In order to compensate this and improve performance, we can apply several techniques like vectorization or parallelization.
These topics will be discussed in Chapter 3.

Offload Execution

We know that the coprocessor is designed for executing highly parallel and vectorized codes. But in reality, every application code has sequential parts like environmental setup or memory management, that are not very performant on this card. This problem can be solved by using the offloading execution model.

This model uses both the host and the coprocessor. The main workflow takes place on the host while suitable calculations are executed on the MIC card.
This is basically the act of copying data from the master node (the host) to the coprocessor, executing a massively parallel algorithm and re-copying the result back to the host [Rah13].

We can realize that by adding offloading pragmas. Consider the following example of a matrix multiplication that uses offloading:

```
void doMult(unsigned int size, double* restrict A, double* restrict B, double* restrict C)
{
    #pragma offload target(mic:0) in(A:length(size*size)) \
                    in(B:length(size*size)) out(C:length(size*size))
    {
        #pragma omp parallel for shared(A, B, C, size)
        for (int i = 0; i < size; ++i)
            for (int j = 0; j < size; ++j)
                for (int k = 0; k < size; ++k)
                    C[i + j*size] += A[i + k*size] * B[k + j*size];
    }
}
```

Listing 2.3: Offloading using pragmas

As we can see, the only difference to a non-offloaded variant is the additional pragma. This pragma tells the compiler the offloading target (a MIC device) and what and how many data items to copy to the device (`A` and `B`) and what to copy back from it (`C`). In a working example, the memory for the three pointers is allocated and free'd on the host processor.

While this would be the most useful model for the ExaStencils project, this is *not* usable at all. This is caused by the fact that this project uses multi-dimensional arrays to represent the multi-dimensional grid. These arrays can be expressed in C/C++ as multiple pointers (like `double ***p;`) or multi-dimensional arrays of pointers (like `double* p[2][10];`). And the current Intel® C compiler does neither support the first nor the second type [Cor13e].

Symmetric Execution

As its name suggests, this execution model makes the application run both on the host and the coprocessor. These two programs are communicating with each other using message passing, e.g. with MPI. So there is no explicit master node like when using offloading, but each hardware is treated as a single node in a cluster environment [Rah13].

Intel® Cilk™ Plus Offloading

Besides the above mentioned models, the Intel® Cilk™ Plus language extension also can make use of offloading.
This can happen on global variables only that have to be allocated and free'd as shared memory.
We will explain this process in detail on the following simple example.

```
double** _Cilk_shared dp;

_Cilk_shared void init() {
    dp = (double**) _Offload_shared_malloc(5 * sizeof(double*));
    for (int i = 0; i < 5; ++i) dp[i] = _Offload_shared_malloc(4 * sizeof(double));
}

_Cilk_shared void modifyShared()
{
    for (int i = 0; i < 5; ++i)
        for (int j = 0; j < 4; ++j)
            dp[i][j] = (double)42 + i + j;
}
```

```
15 _Cilk_shared void destroy()
   {
       for (int i = 0; i < SIZE1; ++i) _Offload_shared_free(dp[i]);
       _Offload_shared_free(dp);
   }
20
   void modifyLocal()
   {
       dp[2][3] = -7;
   }
25
   int main(int argc, char* argv[])
   {
       _Cilk_offload init();
       _Cilk_offload modifyShared();
30     modifyLocal();
       _Cilk_offload destroy();
       return 0;
   }
```

Firstly, we notice that the global variable that should be used for offloaded computations has the additional attribute `_Cilk_shared` which marks it as a shared memory variable. As such, we have to allocate it properly using the allocation intrinsic `_Offload_shared_malloc`. Of course, we have to free it with its intrinsic counterpart `_Offload_shared_free`, as any mix of this intrinsics and the standard memory allocation will result in undefined behaviour.
Next, we will take a look where we actually use offloading. To do so, called functions have to have the additional attribute `_Cilk_shared`. Any computation in these functions then can take place on the coprocessor. Now, when we call these functions using `_Cilk_offload`, the function will be executed on the coprocessor. But, if we call it just as usual, the function is executed on the host [Cor12b].

Currently, the usage of this approach on the ExaStencils project runs out of memory, so it is not implemented in the code generator.

Chapter 3

An Intel®Xeon Phi™Backend

3.1 Vectorization

To exploit the full potential of the Intel®Xeon Phi™, highly parallel and vectorized code must be used. The term vectorization means the conversion of scalar into vectorized code, which can be done either by the programmer or by a compiler.

3.1.1 Introduction

The code normally produced by a programmer or generator is *scalar*, which means that every value used is a single value. As a consequence, a single instruction can compute at most one data item.
In contrast to that, on the Intel®Xeon Phi™another technology is used: SIMD.
SIMD (*Single Instruction Multiple Data*) has the special ability to compute more than one data item in the same instruction, the operations will be applied element by element [Cor13b].
These two concepts are illustrated in Figure 3.1.
A concrete example on the Intel®Xeon Phi™is shown in Figure 3.2.

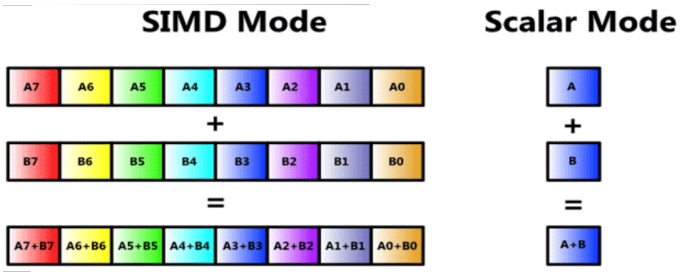

Figure 3.1: Illustration of a SIMD instruction and a scalar instruction(taken from [Cor11])

```
zmm0 = { 0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0 }
zmm1 = { 8.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0 }

zmm2 = { 9.0, 9.0, 9.0, 9.0, 9.0, 9.0, 9.0, 9.0 }
k3   = {  1 ,  1 ,  0 ,  1 ,  0 ,  1 ,  1 ,  1  }

vaddpd zmm2 {k3}, zmm0, zmm1

zmm2 : { 8.0, 8.0, 9.0, 8.0, 9.0, 8.0, 8.0, 8.0 }
```

float64 vectors

mask register

Figure 3.2: IMCI instruction (taken from [Cor13d])

As we see, two vectors of eight values can be added *in one instruction* and saved into a target vector. Furthermore, it is possible to mask these operations, so in the example two vector entries are not computed. All together, this computation can take place in one instruction, which takes around four clock cycles [Cor13d].

As mentioned above, the coprocessor is capable of using 512 bits for each vector (eight double precision or 16 single precision data items) at a time. This is twice as much as when using *Intel® Advanced Vector Extensions (AVX)* on other architectures, which suggests that highly parallel code should be executed on this hardware. Later, we will verify this empirically.

3.1.2 Manual Vectorization

As stated, a programmer can actively write vectorized code or change existing code so it can be vectorized.

The manual approach can be achieved through the usage of *intrinsic* functions. These are C-style functions hand-optimized for the Intel® Xeon Phi™ that provide easy access to Intel® vector instructions without a need to write assembly code. A full list of all available intrinsic functions can be viewed under [Cor16b]. The only supported vector instruction set on the used MIC card is Intel® *Initial Many Core Instructions* (IMCI).

As a typical example, we optimize the matrix multiplication for 8×8 matrices using IMCI intrinsics:

```
1  void doMult(double* restrict A, double* restrict B, double* restrict C)
   {
       __mm512d a_line, b_line, c_line;
4      for (int i = 0; i < 64; i += 8) {
           a_line = _mm512_load_pd(A);
           b_line = _mm512_set1to8_pd(B[i]);
           c_line = _mm512_mul_pd(a_line, b_line);
8
           for (int j = 1; j < 8; ++j) {
               a_line = _mm512_load_pd(&A[j*8]);
               b_line = _mm512_set1to8_pd(B[i+j]);
12             c_line = _mm512_fmadd_pd(a_line, b_line, c_line);
           }

           _mm512_store_pd(&C[i], c_line);
16     }
   }
```

Listing 3.1: 8×8 matrix multiplication using intrinsics

Here, we use the Intel® Xeon Phi™ vector intrinsics to process 8 double values at once. Note that the first three statements in the outer loop are the unrolled first step of the inner loop to avoid an additional initialization of the accumulator `c_line` with zero. These instructions then load the needed data. The real computation then is done in line 12, which just computes

```
c_line += a_line * b_line;
```

This computation can be realized using a *fused multiply add*, which can perform one multiplication and one addition in a single instruction. Finally, the result is stored back into matrix `C`.

The current code generator also supports generating code with vector intrinsics even for the Intel® Xeon Phi™. As we will see in Section 4.2, this already leads to a good performance, but we can push it further with other techniques we will also see shortly.

Another approach is the usage of the Intel® Cilk™ Plus Array Notation as seen in Section 3.4.

3.1.3 Auto-vectorization

As we can see in the previous section, it is an extreme effort to write all vectorized code without support from tools that can do this automatically. Because of this, the Intel® C Compiler (ICC) supports auto-vectorization. This is an attempt to automatically vectorize existing code without interference of any programmer.
Furthermore, this compiler can report what sections have been vectorized and which have not and can give information about estimated potential speedups [Cor12a].
In order to support auto-vectorization, a programmer can make use of several constructs:

1. Data alignment
2. Pragmas
3. Intel® Cilk™ Plus Array Notation
4. Elemental functions

We will describe these points in detail later.

3.1.4 Loop Vectorization

As the most expensive parts of the execution of the genereated ExaStencils code are the loops running the smoothing algorithms, loop vectorization is the main focus of this thesis.
The basic principles of vectorization also apply here. But for improving performance, the compiler checks for the following points [Cor13b]:

1. Is the loop run condition invariant of the body?
2. Are any base pointers used in the loops invariant?
3. Is there any aliasing between used pointers?
4. Are the operations used in the loop associative?
5. Will a vectorized version be faster than the scalar version?

Based on that, the compiler make its decision whether to and how to vectorize the loop, or not to do so. If any of these conditions is not fulfilled, the loop will not be auto-vectorized. But if there is some more information about the loop e.g. that there is no aliasing, and the compiler does not recognize it, the programmer can give the compiler a hint what to do. This will be described in detail later in the section on pragmas.

3.2 Data Alignment

For efficient vectorization, it is very important how data is aligned. It is, because if the data is aligned correctly, a compiler can maximize the amount of vectorized code.

3.2.1 Data Alignment on the Intel® Xeon Phi™

As mentioned, data alignment is important for assisting in generation of vectorized code. So, the first step is to align the data correctly in the first place.

The optimal data alignment differs from hardware to hardware, for the Intel® Xeon Phi™ the optimal value is 64 bytes [Cor15]. So, we have to align the data on a 64 byte boundary for getting the best performance.

In order to do so, we have to do two different things:

1. Align the base pointer of the array.
2. Align the dynamically allocated memory.

In detail we demonstrate this on the following very short example[1]:

```
1  double* A[10] __attribute__((aligned(64)));

   void func()
   {
5      for (int i = 0; i < 10; ++i)
           A[i] = (double*) _mm_malloc( 2 * (i + 1) * sizeof(double), 64);

       for (int i = 0; i < 10; ++i)
           if (A[i]) {
10             _mm_free(A[i]);
               A[i] = 0;
           }
   }
```

Listing 3.2: Data alignment on Intel® MIC

In Listing 3.2 we demonstrate how to align data on a 64 byte boundary for Intel® MIC. Firstly, the base pointer of the array of pointers is aligned. Next, the dynamically allocated memory is also aligned using the allocation intrinsic `_mm_malloc`. Of course, we have to free the dynamically allocated memory with an appropriate method. A mix of these allocation intrinsics and regular `C/C++` allocation will result in undefined behaviour [Cor15].

3.2.2 Data Alignment on loops

When vectorizing loops, it is quite difficult (or nearly impossible) to restructure it to vectorize the whole loop. In order to vectorize most of the code, the Intel® C compiler will split up a loop in three parts if necessary: a peel, a main and a remainder loop. This is shown in Figure 3.3.

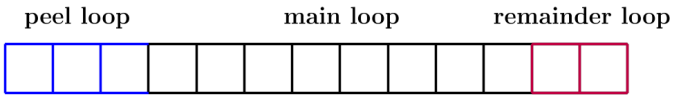

Figure 3.3: Schematic split up loop

As the names suggest, the main loop is that part of the original loop that fits perfectly into the used data alignment. The peel and remainder loops are left-over iterations that can be executed (and even vectorized) separately [Cor16f].

[1] Note that this alignment attribute syntax works on Linux and Mac only, on Windows use the older `__declspec(align(64))`.

The following simple example of a simple `for` loop will demonstrate this:

```
#define SIZE 500

double A[SIZE][SIZE];
double B[SIZE][SIZE];
double C[SIZE][SIZE];

void func()
{
    for (int i = 0; i < SIZE; ++i)
        for (int j = 0; j < SIZE; ++j) {
            A[i][j] = i + j;
            B[i][j] = i - j;
            C[i][j] = 0.0;
        }
}
```

Listing 3.3: Loop that will be split up

After compiling with `-vec-report5`, the Intel® C compiler prints a file containing at least the following vectorization informations:

```
LOOP BEGIN at file1.cpp(33,9)
<Peeled loop for vectorization>
remark #15301: PEEL LOOP WAS VECTORIZED
LOOP END

LOOP BEGIN at file1.cpp(33,9)
remark #15399: vectorization support: unroll factor set to 2
remark #15300: LOOP WAS VECTORIZED
remark #15451: unmasked unaligned unit stride stores: 1
remark #15475: --- begin vector loop cost summary ---
remark #15476: scalar loop cost: 13
remark #15477: vector loop cost: 0.810
remark #15478: estimated potential speedup: 12.500
remark #15487: type converts: 2
remark #15488: --- end vector loop cost summary ---
LOOP END

LOOP BEGIN at file1.cpp(33,9)
<Remainder loop for vectorization>
remark #15301: REMAINDER LOOP WAS VECTORIZED
LOOP END
```

As one can see, three loops were generated and vectorized separately. Futhermore, this output contains additional information about the estimated speedup because of vector arithmetic, or the unroll factor introduced by the compiler.

3.2.3 Loop Unrolling

In the optimization output above, we see that the unroll factor was set to 2.
This means that every statement in the loop body was duplicated and the access index in the second statement is incremented, while the loop iterator is incremented by two instead of one.
This technique may serve very well to restructure a loop in order to vectorize it. All parts of the loop that could not be restructured like that will be executed in a separate loop as mentioned above.

This is already implemented in the ExaStencils code generator. In that strategy, there can be distinguished between two variants of loop unrolling: *interleaving* and *non-interleaving* unrolling [KSK+15].

Lets imagine the following simple loop:

```
for (int i = 0; i < 8; i += 1) {
    a[i] = i - 4;
    b[i] = i + 2;
}
```

Now we will show up the difference of both unrolling variants:

```
        for (int i = 0; i < 8; i += 2) {
            a[i]   = i - 4;
            b[i]   = i - 2;
            a[i+1] = (i+1) - 4;
            b[i+1] = (i+1) + 2;
        }
```

Listing 3.4: Non-interleaving loop unrolling

```
        for (int i = 0; i < 8; i += 2) {
            a[i]   = i - 4;
            a[i+1] = (i+1) - 4;
            b[i]   = i + 2;
            b[i+1] = (i+1) + 2;
        }
```

Listing 3.5: Interleaving loop unrolling

As we can see, in the non-interleaving variant, the whole loop body was duplicated and appended. In contrast to that, using the other variant each individual statement was duplicated and appended before it was proceeded with the next statement, provided that no data dependency is violated [LK15]. Especially the latter variant could bring benefits when used on in-order architectures like the Intel® Xeon Phi™ because the base pointer for each array has to be loaded once only. In the other case of *non-interleaving unrolling*, for each statement an individual load instruction will be generated instead.

3.3 Pragmas

As we noticed before, some loops will not be vectorized although it could be useful. This may result from efficiency heuristics done by the target compiler, so, this is a serious limitation of it. But we can give some hints to the compiler so it should try to vectorize even these loops. These hints come in form of pragmas.
The main pragmas usable for vectorization are `ivdep`, `vector` and `simd`.

3.3.1 Vector Dependencies

The main obstacle when vectorizing code are the dependencies in loops, which means that the state of one loop iteration depends on another one. In general, there are three different types of dependencies: *flow, anti* and *output* dependence. These three describe the possible conflicts that can happen in code.

First, we know that two read-only operations on data do not influence each other, so two read operations are independent on the Intel® Xeon Phi™.
The first true conflict that can happen is a *flow* dependence. This describes that data is read after it was written in a previous statement.

Another type of dependence is the *anti* dependence what is the exact opposite of the *flow* dependence. This means, that here memory is written after it was read.
Lastly, there exists the so called *output* dependence. This one arises if some memory is overwritten after it was already written by another statement before [Golo8].

Now we will show up these dependencies on a small example code. Here we use the scalar case, but the same applies for vectorized code (element per element).

```
a[0] = 0;    // S1
b[0] = a[0]; // S2
a[0] = 42;   // S3
b[0] = -1;   // S4
```

The above example shows all three dependence types at once:

- There exists a *flow* dependence between statements S1 and S2 because in the latter statement the value of `a[0]` is read after it was written by statement S1.
- Also we see an *anti* dependence between statements S2 and S3. Here, the value of `a[0]` is written after it was read in S2.
- Lastly, there is an *output* dependence between the second and the fourth statement because in both statements the same data item is written.

As stated, data dependencies represent conflicts that must be handled.
The Intel®C compiler does an analysis of the input code and tries to detect these conflicts. It further distinguishes between assumed and proven dependencies. In the latter case, the compiler cannot solve this problem and does not vectorized the code, while in the first case it does not have enough information to do so, and it still aborts vectorization.
So, if there is information about dependencies between data items, the programmer can tell these to the compiler to enable vectorization where it would be denied otherwise. For the specific case of this thesis, this happens by using pragmas.

3.3.2 #pragma ivdep

IVDEP stands for *Ignore Vector DEPendencies*, so this pragma gives the compiler a hint that there are no vector dependencies in the following code. Normally, any assumed dependence will be treated as proven dependence to prevent vectorization. This pragma will override that heuristic [Cor16c] listed as point three in Section 3.1.4.
However, if there are proven data dependencies in a loop, it will not be vectorized.

Lets suppose the following example[2]:

```
void func(int *a, int k, int c, int m)
{
    #pragma ivdep
    for (int i = 0; i < m; ++i)
        a[i] = a[i + k] * c;
}
```

Listing 3.6: Usage of `#pragma ivdep`

As one can see, the value of `k` is not known at compile time. This is an assumed dependence that prevents vectorization. But if we annotate the loop inside the function with this pragma, we give a hint that this should be ignored, and so it will be vectorized. Of course, this is only useful, if the array limits of `a` are not violated.

[2] Code taken from https://software.intel.com/en-us/node/514541

3.3.3 #pragma vector

Similar to the above pragma, this gives the compiler a hint not to use its default optimization heuristics. There are multiple strategies usable with this pragma. The desired strategy must be passed as an argument, like in `#pragma vector always`, where `always` is the argument. As for our purposes, the only useful attributes are `always` and `nontemporal`. While the first argument tells the compiler to ignore *any* of its efficiency heuristics like listed in Section 3.1.4, the second one instructs the compiler to generate streaming stores, a common technique for improving performance [Cor16e]. Streaming stores allow the program to overwrite a full cache line without reading it before, making room for other, probably more frequently used data [Hag08].

3.3.4 #pragma simd

In contrast to just giving hints to the compiler as the above two pragmas, this one enforces the vectorization of loops whenever possible, regardless of any compiler heuristics what means it disables all checks listed in Section 3.1.4. As a consequence, loops that will be vectorized anyway by the compiler are not affected by this.
But still, if there is absolutely no chance for vectorization like when a function call is made within it, a warning will be produced and the loop will stay scalar [Cor16d].

Just suppose the following example[3]:

```
void add(double* a, double* b, double* c, double *d, double* e, int n)
{
    #pragma simd
    for (int i = 0; i < n; ++i)
        a[i] = a[i] + b[i] + c[i] + d[i] + e[i];
}
```

Listing 3.7: Usage of `#pragma simd`

In this example, the compiler will not auto-vectorize because there are too many unknown pointers. But when the `simd` pragma is applied, this is ignored and the loop will be vectorized.

However, if this approach is included in the ExaStencils code generator, some dependencies are destroyed what lead to incorrect behaviour of the generated code. Because of this, this pragma is not used, but a workaround using a combination of other pragmas was created instead. This workaround will be described in Section 3.6.

3.3.5 Aliasing and restrict

At this time, we should mention aliasing and the `restrict` keyword.

Aliasing is a common problem when writing code. It means that the exact same data item can be accessed through two or more different ways.
A simple example would be the assignment `int i = 0, *j = &i;`
Here, the saved data can be accessed both by the variables `i` and `j`. This is especially a problem when memory is accessed through pointers.

But if we have the information that two pointers have distinct memory address spaces, we can apply the `restrict` keyword to them. This marks them with the stated behaviour for the compiler and enables it to do more sophisticated optimizations [Cor12a] and the vectorizing unit can assert the third point listed in Section 3.1.4. The following short example demonstrates how this is done:

[3] Code taken from https://software.intel.com/en-us/node/514582

```
1  void func1(double* a, double* b, double* c, int n)
   {
       for (int i = 0; i < n; ++i)
           c[i] = a[i] + b[i];
5  }

   void func2(double* restrict a, double* restrict b, double* restrict c, int n)
   {
       for (int i = 0; i < n; ++i)
10         c[i] = a[i] + b[i];
   }

   int main(int argc, char** argv)
   {
15     int size = 5;
       double* restrict a = (double*) malloc(size * sizeof(double));
       double* restrict b = (double*) malloc(size * sizeof(double));
       double* restrict c = (double*) malloc(size * sizeof(double));

20     func1(a, b, c, size);
       func2(a, b, c, size);

       free(a); free(b); free(c);
       return 0;
25 }
```

Listing 3.8: Using `restrict` for anti-aliasing

When `func1` is called, runtime checks have to be done to properly schedule load and store statements, but this is not the case when we call `func2` instead. Because we applied the `restrict` keyword to the parameters, the compiler knows that the memory address spaces of the pointers are distinct.

It should be noted here that the Intel® C compiler *does not* know the `restrict` keyword by default. This has to be enabled by using the compile flag `-restrict`. However, this may affect portability. An alternative to this is to tell the compiler that there is no aliasing with the switch `-fno-alias` [Cor12a].

Furthermore, it is *not* necessary to use the `restrict` keyword or the `-fno-alias` switch when the #pragma simd is used, this will mark the annotated loop with hints for passing all checks listed in Section 3.1.4.

By examining the code generated by the ExaStencils code generator, we see that we can safely assume that there is no aliasing. Furthermore, we notice that there are quite a few loops that will not be vectorized by the Intel® C compiler because of assumed dependencies.

3.4 Intel® Cilk™ Plus Array Notation

The Intel® Cilk™ Plus Array Notation is an extension to the `C/C++` language and syntactic sugar for loops, that should be vectorized and parallelized.
The syntax fully replaces a loop and is inspired by mathematical domain specific languages like GNU Octave or Matlab. Just like in these languages, we can access the full array at once or

get slices from it. If the dimension of the array is not known at compile time, the first and last index has to be specified in slice notation. The chosen operation will be applied element-per-element [Cor13b].

As a simple example, we replace an unit-stride access loop over three arrays of the same size with this notation:

```
for (int i = 0; i < 8; ++i)
    A[i] = B[i] + C[i];
```

Listing 3.9: Original loop

```
A[:] = B[:] + C[:];
```

Listing 3.10: Altered using Intel® Cilk™ Plus

In Listing 3.10 we can see the simplified syntax, where all array elements are accessed. It should be noted, that all accesses use the same index here, but we cannot control if e.g. the first element is accessed before the last one.
Such code then can be vectorized and parallelized with more flexibility by the target compiler, as it can determine its optimal schedule for array accesses on itself.

This language extension allows some more operations to be done on arrays. A few examples are shown in Listing 3.11.

```
/* Generating an array slice.
 *
 * Syntax:  array[start:length:stride]
 *
 * B[0] = A[0];
 * B[1] = A[2];
 * B[2] = A[4];
 */
int B[] = A[0:3:2];

/* Using standard operators. */
A[:] = B[:] + C[:];

/* Call a function on every element. */
A[:] = add_2(B[:]);

/* Sum up all array entries. */
int sum = __sec_reduce_add(A[:]);

/* Mask operations. */
if (A[:] < 0) {
    A[:] = 0; /* For every negative value, set it to zero. */
}
```

Listing 3.11: Operations on arrays using Intel® Cilk™ Plus Array Notation

As we cannot specify any particular order between the individual array elements, this approach is not usable at all for our project as we need to specify indices of neighboured stencils cells.

3.5 Elemental Functions

Another technique for vectorizing code is the use of *elemental functions*.
Such a function is simply a scalar function annotated with `__attribute__((vector))`. The target compiler then generates if possible both a scalar and a vectorized version. Depending on the calling mechanism (like Intel® Cilk™ Plus Array Notation) a version is chosen [Cor13b].

One common case is to implement a scalar function and then use the Intel® Cilk™ Plus Array Notation to use the vectorized version and execute it in parallel.

This case is illustrated in the next example[4].

```
1  __attribute__((vector))
   __attribute__((vector(uniform(b, c))))
   double func(double a, double b, double c)
   {
5      return a + 7 * b - 42 * c;
   }

   int main(int argc, char* argv[])
   {
10     double a[] = {1, 2, 3, 4}, b[] = {0, 8, 3, 7};
       double c[] = {-1, 9, 4, 2};
       double d[];

       d = func(a[:], b[:], c[:]);
15     return 0;
   }
```

Listing 3.12: Elemental function in combination with Intel® Cilk™ Plus Array Notation

Here, we wrote the function `func`, that executes one scalar statement, and annotated it with the target compiler attribute `__attribute__((vector))`. This instructs it to generate the scalar version as shown and a vectorized one. When we call that function on our arrays using the Intel® Cilk™ Plus Array Notation, this function will be executed for each pairs of values from the three source arrays in a vectorized version. If there are hardware ressources existent for parallel programming, this can be executed even in parallel.

As we can also see, we can generate multiple vectorized and optimized versions by simply adding another attribute. With this, there will be two vectorized versions generated, one particular for uniform values of `b` and `c`, and a second one for the other case. The second attribute can also be omitted safely, but this may become a loss of performance, while on the other hand, if the first one is removed and if it is called with non-uniform code, the scalar (!) version will be executed [Cor13b].

As of the current project state, there are only a few actual functions which make massively use of loops and work on multiple data, so this technique is also not useful on the ExaStencils project.

[4] Also here applies the rule that on Windows `__declspec(vector)` must be used instead of the attribute syntax.

3.6 Implementation Details

After we discussed the theoretical background, it is time to show what parts of the above mentioned things were added to the compiler in order to enhance the performance of the generated code.
This optimizations only are made for Linux and Windows target installations, as we use the native execution model, which is not supported on OSX systems.

3.6.1 Data Alignment

We already know that correct data alignment is essential for vectorization and thus performance when using the Intel®Xeon Phi™. We also know that the optimal data alignment for our coprocessor is 64 bytes, as it can handle 64 bytes in one vector instruction. Because of that, the target compiler attribute `__attribute__((aligned(64)))` for Linux target systems or `__declspec(align(64))` for Windows target installations, repectively, was added for all global pointer-like data.

As mentioned in a former section, also the dynamically allocated memory should be aligned on this 64 byte boundary. So the default memory allocation was changed from `new double[size]` to `(double*) _mm_malloc(size * sizeof(double) + 64, 64)` for any allocation parameter `size`. The latter statement utilizes the IMCI memory allocation intrinsic and ensures that it is aligned on our desired 64 byte boundary. Also the free'ing of memory was changed from `delete[] pointer` to `_mm_free(pointer)` to ensure predictable behaviour.

Additionally, 64 bytes in size were added to ensure safe padding which in advance helps the target compiler vectorizing and thus improves performance [Cor14].

3.6.2 Loop Vectorization through Pragmas

The next step in enhancing performance is to push vectorization. To do so, the best way is to add the pragmas explained in Section 3.3. Empirical tests showed that the `simd` pragma destroyed some dependencies which resulted in incorrect program behaviour. This problem was solved by using a combination of the `ivdep` and the `vector` pragmas. Summing up, a transformation like shown in Listing 3.13 was implemented that transforms the innermost loops to new ones with pragmas added.

```
case class VectorIvdepPragma(loop: Option[ForLoopStatement],
                             omp: Option[OMP_ParallelFor]) extends Statement {
    override def prettyprint(out: PpStream): Unit =
        out << "#pragma vector always\n#pragma vector nontemporal\n#pragma ivdep\n"
            << loop.getOrElse(omp.get)
}

object XeonPhiVectorization extends DefaultStrategy("Vectorizing for the Intel(R) Xeon Phi(TM)") {
    override def apply(applyAtNode: Option[Node]) = {
        this.transaction()

        this.execute(new Transformation("Adding vectorization pragmas", {
            case o: OMP_ParallelFor
              if (isInnermost(o.body.body) && hasNoAllocationOrFree(o.body.body)) =>
                new VectorIvdepPragma(None, Some(
                  new OMP_ParallelFor(o.body, o.additionalOMPClauses, o.collapse)))
            case l: ForLoopStatement
              if (isInnermost(l.body) && hasNoAllocationOrFree(l.body)) =>
                new VectorIvdepPragma(Some(
                  new ForLoopStatement(l.begin, l.end, l.inc, l.body, l.reduction)), None)
        }, false))

        this.commit()
    }

    def isInnermost(loopBody: ListBuffer[Statement]): Boolean =
        (true /: loopBody.map {
            s =>
                s match {
                    case _: ForLoopStatement => false
                    case _: Scope            => false
                    case _: OMP_ParallelFor  => false
                    case _                   => true
                }
        })(_ && _)

    def hasNoAllocationOrFree(loopBody: ListBuffer[Statement]): Boolean =
        (true /: loopBody.map {
            s =>
                s match {
                    case AssignmentStatement(_, Allocation(_, _), _)               => false
                    case ConditionStatement(_, ListBuffer(FreeStatement(_), _*), _) => false
                    case _                                                         => true
                }
        })(_ && _)
}
```

Listing 3.13: Excerpt of the implemented strategy

Firstly, we created a `case class` to represent our pragmas. In the strategy we will search for loops or parallelized loops that do not have any other loops or OpenMP statements in their bodies by using the *Visitor*-like behaviour of transformations. As an optimiziation, we also ignore such loops where memory is allocated or free'd. These loops can be found by using the above implemented helper functions which are simply left folds which evaluate to `true` if they are suitable for being annotated with our pragmas. For all of the valid loops, we do so by simply wrapping them in our newly created `case class` which overrides the pretty-printing behaviour.

The following example shows how this transformation changed the loop shown in Figure 3.14 into the one shown in Figure 3.15.

```
    for (int y = 0; y <= 3; ++y) {
        for (int x = 1; x <= 511; ++x) {



            for (int i = 1; i <= 511; ++i) {
                /* calculations */
            }
        }
    }
```

Listing 3.14: Before adding the pragmas

```
    for (int y = 0; y <= 3; ++y) {
        for (int x = 1; x <= 511; ++x) {
            #pragma vector always
            #pragma vector nontemporal
            #pragma ivdep
            for (int i = 1; i <= 511; ++i) {
                /* calculations */
            }
        }
    }
```

Listing 3.15: After adding the pragmas

As explained in the previous sections, these pragmas give hints to the target compiler to vectorize these loops even if it would not be done according to its heuristics and to generate streaming stores.

3.6.3 Loop Iteration Counting

It was already stated that the Intel® C compiler does code analysis and tries to predict the loop trip count in order to optimize the loops maybe by applying unrolling or other strategies. However, the compiler is not able to do so on some loops. But, we can tell it the number of iterations by using the `loop count` pragma.
So, another transformation was implemented that pre-calculates the number of iterations of the innermost loops and applies the pragma onto it.

Applied to the example from the previous section, that loop will be transformed into:

```
for (int y = 0; y <= 3; ++y) {
    for (int x = 1; x <= 511; ++x) {
        #pragma loop count 511
        #pragma vector always
        #pragma vector nontemporal
        #pragma ivdep
        for (int i = 1; i <= 511; ++i) {
            /* calculations */
        }
    }
}
```

Listing 3.16: Previous example with `loop count` pragmas added

3.6.4 Target Compiler Switches

In addition to code manipulation strategies, we added numerous compile flags.
In total, the added switches are listed in the below table, a detailed description for each switch can be showed up in the Intel® C++ compiler reference that is online available under [Cor16a].

Switch	Description
`-mmic`	Cross-compile for native execution on Intel® Xeon Phi™.
`-fno-alias`	Assume no alias in whole program.
`-opt-assume-safe-padding`	Assume safe padding for vector instructions.
`-opt-prefetch=4`	Use the maximum of compiler generated prefetch instructions.
`-opt-calloc`	Substitute calls to `calloc` with calls to `_intel_fast_calloc`.
`-opt-malloc-options=2`	Add additional configurations for memory allocations.
`-fno-exceptions`	Disable generation of exception tables.
`-restrict`	Enable usage of `restrict` keyword.
`-qopt-threads-per-core`	Number of threads per core will be used by the application.
`-qopt-streaming-cache-evict=0`	Set cache eviction level to 0.
`-opt-gather-scatter-unroll=2`	Alternate unroll sequence for gather/scatter instructions with an unroll factor of 2.
`-ftls-model=global-dynamic`	Set thread local storage (TLS) model.

Because of the fact that we use native execution, we also have to apply the `-mmic` switch to the linker.

3.6.5 Other Modifications

Besides the above mentioned adaptions, we also implemented other modifications.
Firstly, we enhanced parallelism by lowering the threshold for parallelizing with OpenMP. This lower limit was set the the configurable number of OpenMP threads to compensate the delay for thread creation.
In this context, we also manipulated the generation of these pragmas. In detail, the chunk size of workload distributed among the threads was explicitly set, so we changed `schedule(static)` into `schedule(static,1)`. In other words, the scheduling algorithm remains the same, but the chunk size was set to one to increase the cache reuse.

Finally, we also manipulated the thread distribution of the threads onto the several cores of the Intel® Xeon Phi™, as this has quite a significant impact on the performance.
Basically, there are three types [McC13]:

`compact` In this distribution the threads are distributed as close as possible on the first cores, so the first four threads will be mapped to the first core, the next four to the second and so on. Now, when not all possible 228 threads are in use, some cores will not actually do some work.

`scatter` This is a more advanced thread mapping where the threads are evenly assigned on all cores[5]. A closer look on the mapping shows that the first thread is mapped to the first core, the second to the second core, and so on. The 58th thread then is mapped to the first thread again.
A disadvantage when using this model is the fact that we have to copy memory from one

[5] Of course, this does only work if there are more than 57 threads in use.

core to another if the latter wants to access memory that physically lies on the first. This then leads to a performance loss, although minor only, as we got a high bandwidth on the bi-directional bus.

`balanced` Based on the `scatter` mapping, this slightly different mapping exists for the Intel® Xeon Phi™ only. It divides the threads just like the scattered case to the cores, but now adjacent threads lie on adjacent cores. Now imagine a total of threads of 171 (three threads per core). Then, the first three are mapped to the first core, the next three to the second core and so on.
We expect this type to deliver the best performance for our project because in stencil computations we use adjacent memory cells very often. This is also validated by our empirical measurements.

Figure 3.4 shows these three types schematically with four cores and eight threads.

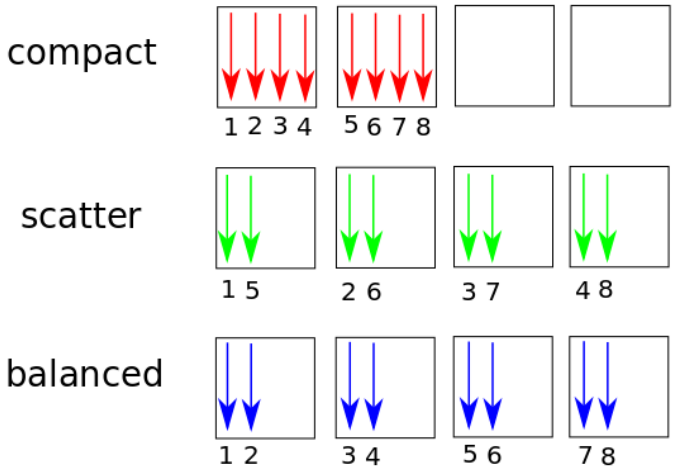

Figure 3.4: The three types of thread affinity

Chapter 4

Runtime Analysis

In the previous chapter we showed techniques for improving the performance of the generated code. Now it is time to measure, how much performance we gained by using these methods.

4.1 Setup

For our experimental tests, we use an Intel® Xeon Phi™ just like stated in Section 2.2.1. So it is a coprocessor with 57 cores, where each is clocked at 1.1 GHz and supports up to 4 hyper-threads. As calculated further above, the maximum aggregated peak performance lies at 1.003 TFLOPs. As for our tests, the software revision is the *XeonPhiBackend* branch at the commit status `c9eb5c7ebcbc12c7af320debd6f6e0a3e590dc5a`. The used target compiler is the Intel® C compiler, version 16.0.1.150.

In total, we test a variety of configurations:

Mnemonic	**Description**
ICC, novec	No vectorization.
ICC, autovec	Auto-vectorization only.
ICC, pragmas	Additional pragmas.
ICC, pragmas, flags	Additional pragmas and compile flags.
ICC, intrinsics	Usage of intrinsic functions.
ICC, intrinsics, pragmas	Usage of intrinsic functions, additional pragmas and compile flags.

The first configuration is our performance baseline. It describes the generated multi-grid code without any vectorization, only with the basic optimizations done already by the compiler.
The next test feature is the same as above but with the basic auto-vectorization done by the Intel® C compiler.
Then we add our optimization strategies for adding the pragmas stated before.
This is followed by the addition of optimizing compile flags mentioned in Section 3.6.4.
While these four configurations benchmark the auto-vectorization variants, the last two rows in the above table test the manual vectorization using intrinsic functions. There we distinguish between the pure intrinsics and these functions with the additional pragmas and compile flags. For each of these two variants, the configuration property `opt_vectorize=true` was added to the knowledge file shown in Appendix A.

For every of these alternatives, we test it with $1, 2, 4, 8, 16, 32, 57, 114, 171$ and 228 threads and no loop unrolling as this results in the best vectorization and thus performance. Additionally, we test the same configurations for the single-grid cycle using the appended Layer 4 file. Finally, we calculate the mean values over 20 runs, totalling 2400 runs.

4.2 Experimental Results

In our experiments, for each of the 120 configurations defined above four metrics are measured:

1. Average runtime per VCycle
2. Absolute runtime
3. Speedup
4. LUPS/FLOPS

The first metric describes the average time spent in a VCycle which includes the full cycle of coarsening, solving and smoothing. This is probably the most sufficient metric because the several VCycles are the most time consuming parts of the program.
Advantageously, the timers to measure this are generated automatically by the ExaStencils code generator.

A plot for this metric is shown in Figure 4.1, a detail plot showing the measurements of the optimized code only is shown in Figure 4.2.

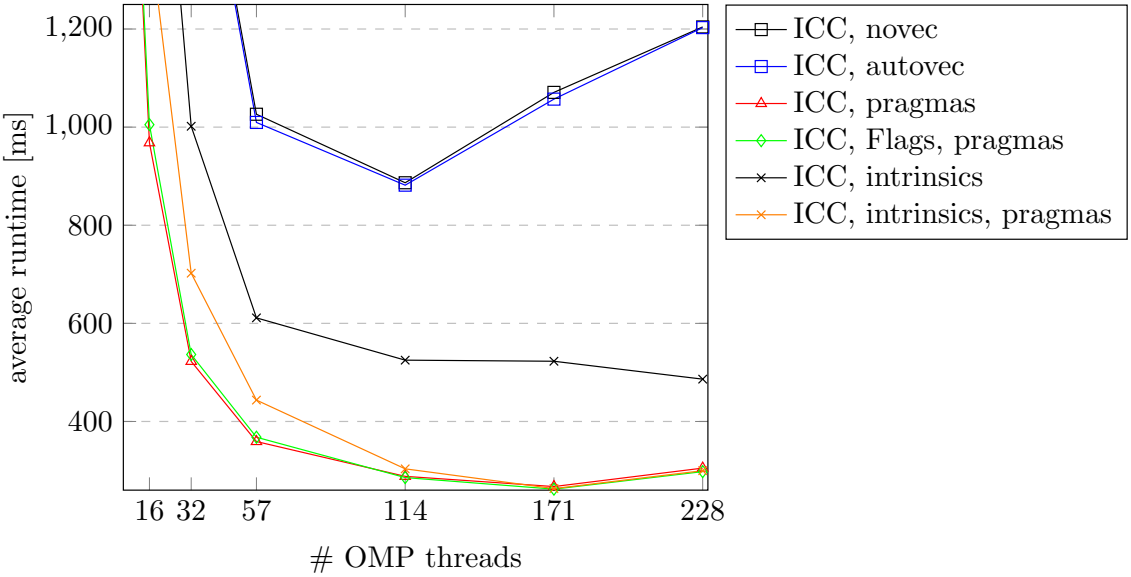

Figure 4.1: Average runtime per VCycle

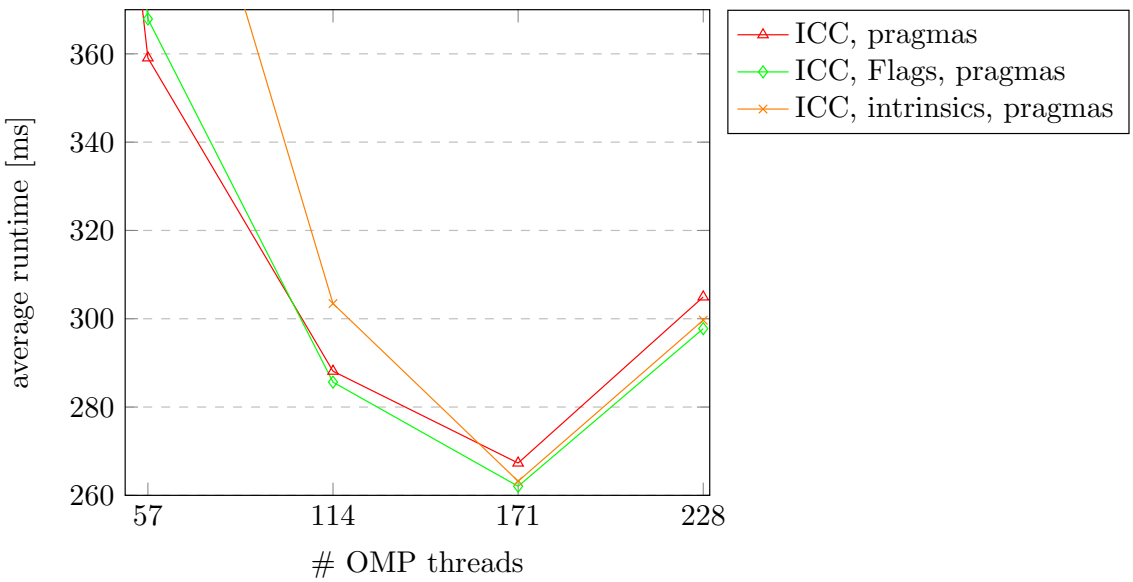

Figure 4.2: Detail plot from Figure 4.1

In these plots we see that our optimizations using pragmas bring some serious speedup. Especially interesting is the fact, the our vectorizing pragmas even improve the code using intrinsic functions. This may be due to the fact that there are still loops remaining in the generated code that do not use intrinsics, but are auto-vectorized because of our pragmas.
Summing up, our added optimizations enhance the performance of the generated code, even beyond the performance of low-level intrinsic code.

A similar behaviour can be observed when measuring the absolute time for solving the system. This is quite logical, as the lower bound for the whole runtime are the VCycles we saw already. Similarly to the first metric, the timers necessary for this measurements are also automatically generated.
A plot of these measured absolute runtimes can be viewed in Figure 4.3.

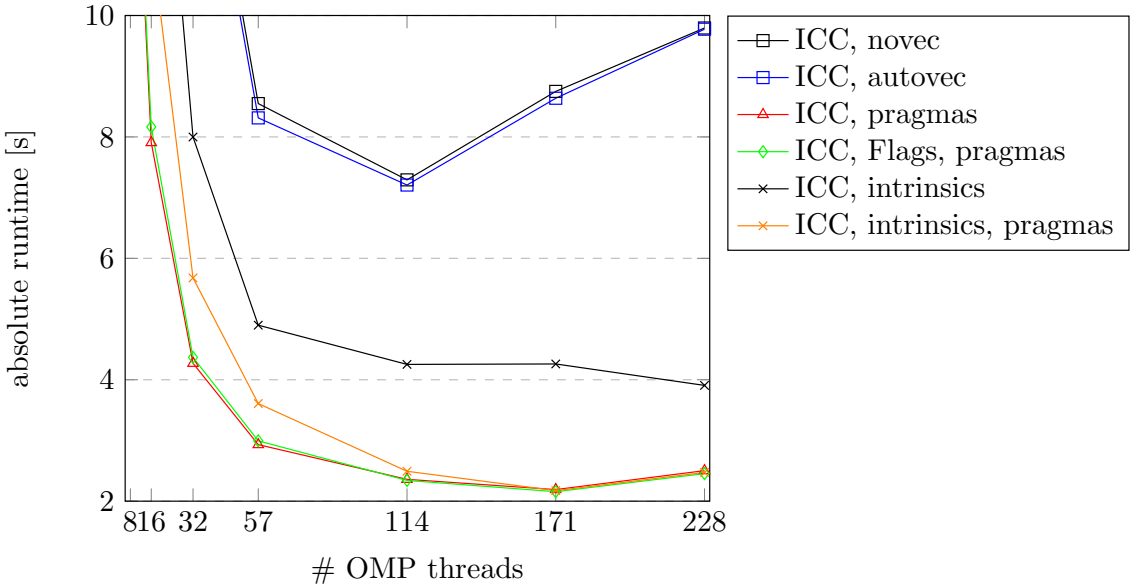

Figure 4.3: Absolute solving time

Based on the above measurement of the absolute solving time, we calculated the actual speedup achieved. The baseline for this speedup was the generated code without any vectorization (with mnemonic novec).
The calculated values are visualized in Figure 4.4.

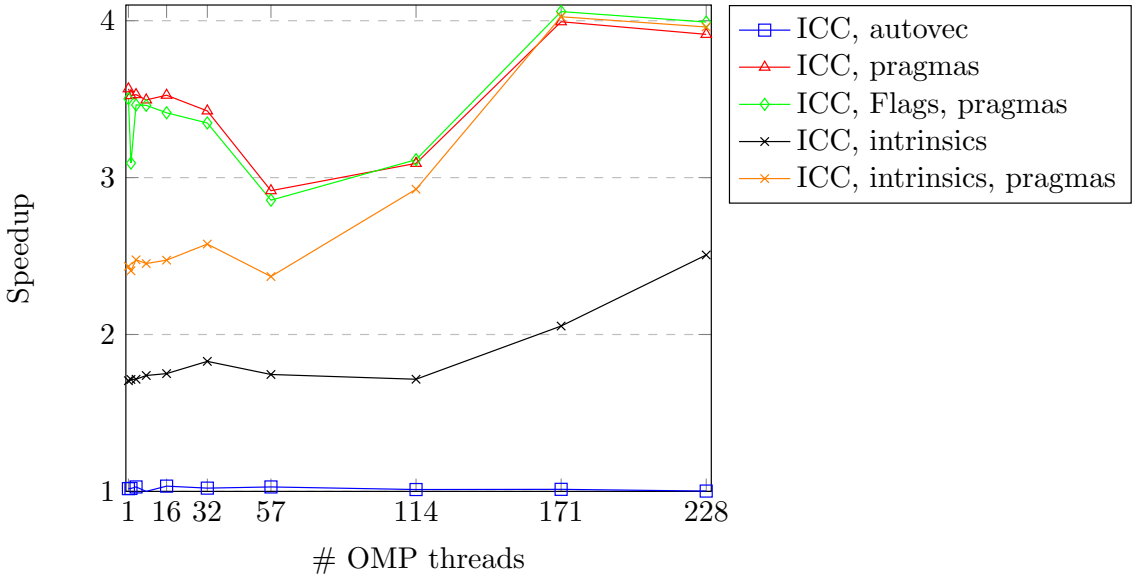

Figure 4.4: Speedup in runtime

The results shown in this plot are not surprising as we know how the program behaves already from the first two metrics. Interestingly, when 57 or 114 threads are used, the non-vectorized and auto-vectorized variants have the lowest runtime, and so does the speedup, while the optimized variants get two times faster with a doubled number of OpenMP threads.
In the best cases, we reach a speedup of over four times, while the *worst* optimized runs still provide a speedup of at least 2.5.

Finally, we take a look at the throughput achieved by the program. The desired metric here are *Lattice Updates per second* (LUPS). This is measured by using the adapted knowledge file shown in Appendix B in combination with a custom Layer 4 file shown in Appendix D. Using this single-grid cycle, this metric is identical to the number of loop iterations divided by the runtime needed. The measured results are shown in Figure 4.5.

As one can easily see, our optimizing strategy provides up to four times the amount of lattice updates per second as reached without the added pragmas. And just like expected from the previous measurements, these pragmas even enhance the performance of the generated code using intrinsics.

Another common performance metric are *floating point operations per second* (FLOPS). This metric can be easily calculated out of the measured LUPS by multiplying them with the amount of floating point operations per loop iteration. In our specific single-grid cycle, a three-dimensional (or six-point) stencil is calculated, which needs the values of its six neighbours and its own value and three weights, totalling ten floating point operations per iteration.
By applying this calculation on our measurements, we see that we get a practical peak performance of 27.3 GFLOPS.

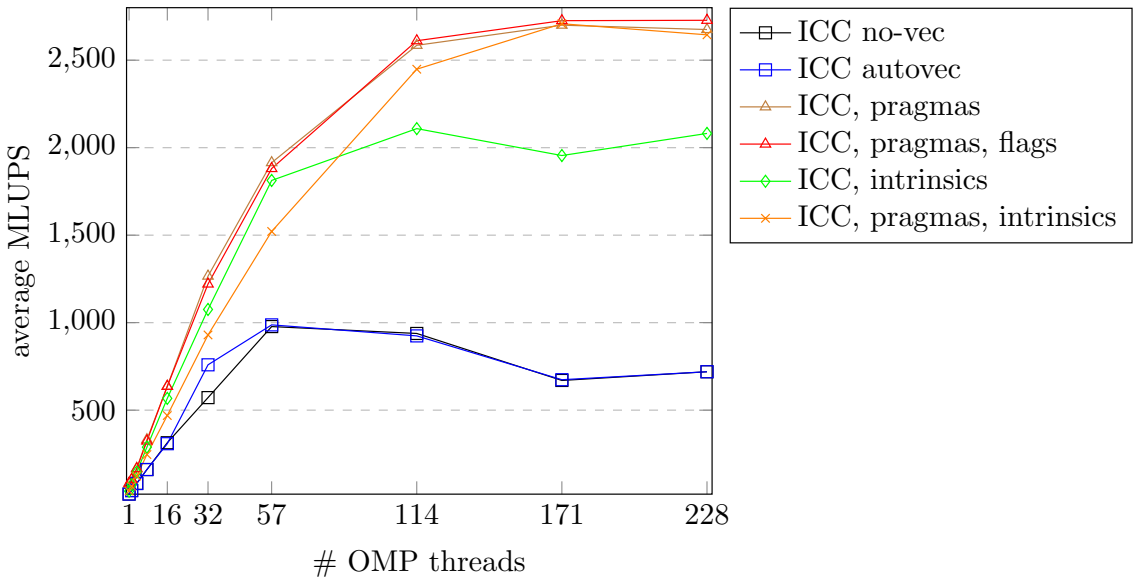

Figure 4.5: Average MLUPS

4.3 Analysis with perf

In addition to our extensive measurements on the runtime, we also analyzed the code using the Linux specific tool *perf*. This tool is a lightweight profiler included in the Linux kernel capable of reading hardware performance counters. The most useful information we get with this method is that about 57% of the whole execution time is spent in the OpenMP library. This indicates that the synchronization of the OpenMP threads takes much time, which is plausible as we use a massive amount of `parallel for` pragmas with a high amount of threads which all have to synchronize after the loop. However, we cannot reduce the amount of synchronization to ensure the correctness of the generated code.

4.4 Summary

As we saw in the previous section, our high-level optimizations using vectorization pragmas and compile flags result in quite a speedup, up to over four times compared to the non-vectorized variant.

However, considering a calculated theoretical peak performance of 1003.2 GFLOPS, we barely reach three percent of this peak. One possible explanation for this is the fact, that in our native execution model every single part of the code is executed on the coprocessor only. This includes memory management and a quite high amount of serial code which performs less well on the Intel® Xeon Phi™ as on other processors. This may be caused by the fact that the clock speed of each Intel® Xeon Phi™ core is 1.1 GHz what is only about a third of a modern Xeon processor.

Chapter 5

Conclusion and Further Work

In this thesis, we gave an overview over Intel®'s many integrated core system named Intel® Xeon Phi™. This is succeeded by a proposal of performance enhancing methods specifically for this coprocessor. Most notably, we explained how vectorization works. In detail, we saw that we can use both manual vectorization or auto-vectorization to process multiple data in a single instruction. Manual vectorization can be used by either using a high-level approach called Intel® Cilk™ Plus Array Notation, where the special notation represents a data vector, or the low-level approach using intrinsic functions which represent assembly instructions on a more typesafe level. In contrast to that, we also use auto-vectorization using pragmas. The target compiler then can automatically vectorize this annotated code if possible. Furthermore, we also discussed the impact of correct data alignment on the performance of the generated code.

We implemented this optimizations as an individual strategy in the ExaStencils code generator. This strategy consists of code transformations which affect the desired parts of the code on the project's intermediate representation. Additionally, we also manipulated already present strategies e.g. the generation of target compiler switches.

Finally, we measured the impact of our optimizations on the target code using four metrics. All measurements are consistent and show a significant performance increase compared to the original generated code. By applying our custom optimizations, a peak speedup of over 4× can be observed, while the *worst* optimized variant still provides a speedup of at least 2.5×.

Summing up, we showed techniques for enhancing performance specifically when using the Intel® Xeon Phi™. However, we also explained that there are technical restrictions which limit the usability of techniques that can increase the performance even more. Further work is necessary to bypass these limitations. Especially, more research should be done on using the Intel® Cilk™ Plus shared memory programming model, so serial code like environmental setup is executed on a host processor while heavy parallel and vectorized parts are executed on the coprocessor. Another starting point for further optimizations is the reduction of the overhead done by OpenMP barriers, to improve the alignment of array accesses or to optimize cache reusage.

Appendices

A Knowledge file for multi-grid cycles

```
simd_avoidUnaligned = true
dimensionality = 3
minLevel = 0
maxLevel = 9
omp_enabled = true

omp_numThreads = 228

omp_useCollapse = true
omp_parallelizeLoopOverFragments = false
omp_parallelizeLoopOverDimensions = true
experimental_useStefanOffsets = true

mpi_enabled = false
mpi_numThreads = 1

l3tmp_generateL4 = true
l3tmp_smoother = "Jac"
l3tmp_numPre = 2
l3tmp_numPost = 2

poly_optLevel_fine = 3
poly_tileSize_x = 0
poly_tileSize_y = 128
poly_tileSize_z = 0

opt_useAddressPrecalc = true
data_alignFieldPointers = true
data_alignTmpBufferPointers = true

opt_unroll = 1
l3tmp_genTemporalBlocking = false
```

B Knowledge file for single-grid cycles

```
simd_avoidUnaligned = true
dimensionality = 3
minLevel = 8
maxLevel = 9
omp_enabled = true

omp_numThreads = 1

omp_useCollapse = true
omp_parallelizeLoopOverFragments = false
omp_parallelizeLoopOverDimensions = true
experimental_useStefanOffsets = true

mpi_enabled = false
mpi_numThreads = 1

l3tmp_generateL4 = false
l3tmp_smoother = "Jac"

poly_optLevel_fine=0
poly_tileSize_x = 0
poly_tileSize_y = 128
poly_tileSize_z = 0

opt_useAddressPrecalc = true
data_alignFieldPointers = true
data_alignTmpBufferPointers = true

opt_unroll = 1
l3tmp_genTemporalBlocking = true
```

C Platform file for Intel®Xeon Phi™

```

targetOS = "Linux"
targetCompiler = "ICPC"
targetCompilerVersion = 16
targetCompilerVersionMinor = 0
targetHardware = "CPU"
simd_instructionSet = "IMCI"
hw_cpu_numCoresPerCPU = 57

hw_cpu_name = "Intel(R) Xeon Phi(TM)"
```

D Layer 4 file for measuring LUPs

```
Domain global< [ 0, 0, 0 ] to [ 1, 1, 1 ] >

Layout FullTempBlockable< Real, Node >@all {
    innerPoints = [ 576, 576, 576 ]
    ghostLayers = [ 5, 5, 5 ]
    duplicateLayers = [ 1, 1, 1 ]
}

Layout PartTempBlockable< Real, Node >@all {
    innerPoints = [ 576, 576, 576 ]
    ghostLayers = [ 4, 4, 4 ]
    duplicateLayers = [ 1, 1, 1 ]
}

Field SolutionT< global, FullTempBlockable, 0.0 >[2]@finest
Field RHST< global, PartTempBlockable, None >@finest

Stencil Laplace@finest {
  [ 0,  0,  0] =>  4.8
  [ 1,  0,  0] => -0.8
  [-1,  0,  0] => -0.8
  [ 0,  1,  0] => -0.8
  [ 0, -1,  0] => -0.8
  [ 0,  0,  1] => -0.8
  [ 0,  0, -1] => -0.8
}

Globals {
}

Function LUPs() : Real {
  Variable dimSize : Integer = 576
  return(dimSize * dimSize * dimSize)
}

Function SmootherT() : Unit {
  loop over fragments {
    repeat 2 times with contraction [1,1,1] {
      loop over SolutionT@finest {
        SolutionT[nextSlot]@finest =
            SolutionT[active]@finest + (0.8 / diag(Laplace@finest)
                * (RHST@finest - Laplace@finest * SolutionT[active]@finest))
      }
      advance SolutionT@finest
    }
  }
}

Function InitFields ( ) : Unit {
  loop over SolutionT@finest sequentially {
    SolutionT[active]@finest = native('((double)std::rand()/RAND_MAX)')
  }
  loop over RHST@finest sequentially {
    RHST@finest = 0
  }
}
```

```
Function BenchmarkT() : Unit {
  print('Cache warmup')
  repeat 1 times {
    SmootherT()
  }
  print('Starting benchmark (temporal blocking)')
  startTimer(benchTTimer)
  repeat 5 times {
    SmootherT()
  }
  stopTimer(benchTTimer)
  Variable time : Real = getTotalFromTimer(benchTTimer)
  print('Runtime: ', time)
  print('MLUPs: ', (LUPs() * 10) / time / 1e3)
}

Function Application() : Unit {
  startTimer(setupWatch)
  initGlobals()
  initDomain()
  InitFields()
  stopTimer(setupWatch)
  print('Total time to setup: ', getTotalFromTimer(setupWatch))
  BenchmarkT()
  destroyGlobals()
}
```

E Script for automatic tests

```
#!/bin/bash
#########################################################
## Simple bash script for computing average runtimes. ##
##                                                     ##
## Author:  Thomas Lang                                ##
## Version: 1.4, 2016-03-24                            ##
#########################################################

## Settings
TOTAL_LINES_PER_RUN_EXA=11           # Number of lines written by the multigird in each run.
TOTAL_LINES_PER_RUN_LUPS=5           # Number of lines written by the single grid in each run.
NUMBER_OF_RUNS=20                    # Total number of runs per configuration.
OUTPUT_FILE_EXA='count_exa.txt'      # File where the output of the multigrid was written into.
OUTPUT_FILE_LUP='count_lup.txt'      # File where the output of the single grid was written into.
MEASURES_EXA='measurements_exa.txt'  # File where multigrid results come.
MEASURES_LUP='measurements_lup.txt'  # File where LUPs results come.
WHERE='EXES'                         # Directory where executables lie.
LUPS="$WHERE/lups"                   # Prefix for lups applications.

## helper variables
offset_exa=1
offset_lup=1

## Doing some file checks ...
[ -f "$OUTPUT_FILE_EXA" ] && >$OUTPUT_FILE_EXA || touch $OUTPUT_FILE_EXA
[ -f "$OUTPUT_FILE_LUP" ] && >$OUTPUT_FILE_LUP || touch $OUTPUT_FILE_LUP
[ -f "$MEASURES_EXA" ] && >$MEASURES_EXA || touch $MEASURES_EXA
[ -f "$MEASURES_LUP" ] && >$MEASURES_LUP || touch $MEASURES_LUP

echo "Starting measurements ..." >> $MEASURES_EXA
echo "Starting measurements ..." >> $MEASURES_LUP

for ex in $(find $WHERE -type f)
do
    absolute_runtime=0
    average_runtime=0
    num_lups=0

    for (( i=1; i<=$NUMBER_OF_RUNS; i++ ));
    do
    if [[ $ex = $LUPS* ]]; then
        ./$ex >> $OUTPUT_FILE_LUP
    else
        ./$ex >> $OUTPUT_FILE_EXA
    fi
    done

    ## Magical counting starts here ...
    for (( i=1; i<=$NUMBER_OF_RUNS; i++ ))
    do
    if [[ $ex = $LUPS* ]];
    then
        # get as string
        head_lines=$(expr $offset_lup \* $TOTAL_LINES_PER_RUN_LUPS)
        str=$(head -n $head_lines $OUTPUT_FILE_LUP | tail -n 1)

        # parse and add
```

```
        arr=($str)
        num_lups=$(echo "scale=4; ${arr[1]} + $num_lups" | bc)
        offset_lup=$(expr $offset_lup + 1)
    else
        # get as string
        head_lines=$(expr $offset_exa \* $TOTAL_LINES_PER_RUN_EXA)
        str=$(head -n $head_lines $OUTPUT_FILE_EXA | tail -n 2)

        # parse and add
        arr=($str)
        absolute_runtime=$(echo "scale=4; ${arr[8]} + $absolute_runtime" | bc)
        average_runtime=$(echo  "scale=4; ${arr[13]} + $average_runtime" | bc)
        offset_exa=$(expr $offset_exa + 1)
    fi
    done

    if [[ $ex = $LUPS* ]];
    then
        lll=$(echo "scale=4; ( $num_lups / $NUMBER_OF_RUNS )" | bc)
        echo "-----------------------------------------------------------------" >> $MEASURES_LUP
        echo "Configuration: "$ex >> $MEASURES_LUP
        echo "Average number of LUPs over "$NUMBER_OF_RUNS" runs: "$lll"  [MLUPs]" >> $MEASURES_LUP
        echo "-----------------------------------------------------------------" >> $MEASURES_LUP
    else
        abs=$(echo "scale=4; ( $absolute_runtime / $NUMBER_OF_RUNS )" | bc)
        avg=$(echo "scale=4; ( $average_runtime / $NUMBER_OF_RUNS )"  | bc)
        echo "-----------------------------------------------------------------" >> $MEASURES_EXA
        echo "Configuration: "$ex >> $MEASURES_EXA
        echo "Absolute runtime over "$NUMBER_OF_RUNS" runs:           "$abs"  [ms]" >> $MEASURES_EXA
        echo "Average runtime per VCycle over "$NUMBER_OF_RUNS" runs: "$avg"  [ms]" >> $MEASURES_EXA
        echo "-----------------------------------------------------------------" >> $MEASURES_EXA
    fi

    >$OUTPUT_FILE_EXA
    >$OUTPUT_FILE_LUP
    offset_exa=1
    offset_lup=1
done

echo "Finished measurements!" >> $MEASURES_EXA
echo "Finished measurements!" >> $MEASURES_LUP
```

Bibliography

[CD13] C-DAC: *hyPACK 2013 Intel® Xeon Phi™: Tuning and Performance*. Website, 2013. – Available online under http://cdac.in/index.aspx?id=pdf_xeon-phi-tips-tun-perf-hypack; looked up on 22nd March 2016

[Cor11] Corporation, Intel®: *Introduction to Intel® Advanced Vector Extensions*. Website, 2011. – Available online under https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions; looked up on 5th April 2016

[Cor12a] Corporation, Intel®: *A Guide to Vectorization with Intel® C++ Compilers*. Website, 2012. – Available online under https://software.intel.com/sites/default/files/8c/a9/CompilerAutovectorizationGuide.pdf; looked up on 18th February 2016

[Cor12b] Corporation, Intel®: *Intel® Xeon Phi™: Offload Compilation*. Website, 2012. – Available online under https://software.intel.com/sites/default/files/Beginning%20Intel%20Xeon%20Phi%20Coprocessor%20Workshop%20Offload%20Compiling%20Part%202.pdf; looked up on 5th April 2016

[Cor12c] Corporation, Intel®: *Intel® Xeon Phi™ X100 Family Coprocessor - the Architecture*. Website, 2012. – Available online under https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner; looked up on 5th April 2016

[Cor13a] Corporation, Intel®: *Intel® Xeon Phi™ Coprocessor 3100 series*. Website, 2013. – Available online under http://ark.intel.com/de/products/series/75808?_ga=1.10100469.1719564821.1453408114; looked up on 26th April 2016

[Cor13b] Corporation, Intel®: *Intel® Xeon Phi™ Coprocessor Advanced Topics in Vectorization*. Website, 2013. – Available online under https://software.intel.com/sites/default/files/Intel%C2%AE_Xeon_Phi%E2%84%A2_Coprocessor_Advanced_Topics_in_Vectorization.pdf; looked up on 1st March 2016

[Cor13c] Corporation, Intel®: *Intel® Xeon Phi™ Product Family - Peak Theoretical Performance*. Website, 2013. – Available online under http://www.intel.com/content/www/us/en/benchmarks/server/xeon-phi/xeon-phi-theoretical-maximums.html; looked up on 21st March 2016

[Cor13d] Corporation, Intel®: *Intel® Xeon Phi™ Coprocessor Architecture Overview*. Website, 2013. – Available online under http://www.training.prace-ri.eu/uploads/tx_pracetmo/MIC_Intro_Architecture.pdf; looked up on 17th February 2016

[Cor13e] Corporation, Intel®: *Restrictions on offloaded code using a pragma*. Website, 2013. – Available online under https://software.intel.com/en-us/node/522493; looked up on 21st March 2016

[Cor14] Corporation, Intel®: *Utilizing Full Vectors and Use of Option -qopt-assume-safe-padding*. Website, 2014. – Available online under https://software.intel.com/en-us/articles/utilizing-full-vectors; looked up on 26th March 2016

[Cor15] Corporation, Intel®: *Data Alignment to Assist Vectorization*. Website, 2015. – Available online under https://software.intel.com/en-us/articles/data-alignment-to-assist-vectorization; looked up on 29th February 2016

[Cor16a] Corporation, Intel®: *Compiler Reference*. Website, 2016. – Available online under https://software.intel.com/en-us/node/512846; looked up on 4th April 2016

[Cor16b] Corporation, Intel®: *Intel® Intrinsics Guide*. Website, 2016. – Available online under https://software.intel.com/sites/landingpage/IntrinsicsGuide/#techs=KNC; looked up on 22nd March 2016

[Cor16c] Corporation, Intel®: *ivdep*. Website, 2016. – Available online under https://software.intel.com/en-us/node/514541; looked up on 28th February 2016

[Cor16d] Corporation, Intel®: *simd*. Website, 2016. – Available online under https://software.intel.com/en-us/node/514582; looked up on 28th February 2016

[Cor16e] Corporation, Intel®: *vector*. Website, 2016. – Available online under https://software.intel.com/en-us/node/514586; looked up on 28th February 2016

[Cor16f] Corporation, Intel®: *What are PEEL and REMAINDER loops? (Fortran and C vectorization support)*. Website, 2016. – Available online under https://software.intel.com/en-us/articles/what-are-peel-and-remainder-loops-fortran-vectorization-support; looked up on 29th February 2016

[exa16] *Advanced Stencil-Code Egineering (ExaStencils)*. Website, 2016. – Available online under http://www.exastencils.org/; looked up on 23rd March 2016

[Fre14] Freitag, Michael: *Analysis and Extension of Existing Tiling Algorithms for Stencil computations*. Online, 2014. – Bachelor thesis, available online under http://www.infosun.fim.uni-passau.de/cl/arbeiten/freitag-b.pdf; looked up on 4th April 2016

[Gen13] Gentryx, User:: *A three-dimensional six-point stencil*. Website, 2013. – Available online under https://en.wikipedia.org/wiki/File:3D_von_Neumann_Stencil_Model.svg; looked up on 26th April 2016

[Gol08] Goldstein, Seth: *Data Dependence in Loops*. Website, 2008. – Available online under http://www.cs.cmu.edu/afs/cs/academic/class/15745-s09/www/lectures/lect6-deps.pdf; looked up on 29th February 2016

[Hag08] Hager, Georg: *A case for the non-temporal store*. Website, 2008. – Available online under https://blogs.fau.de/hager/archives/2103; looked up on 4th April 2016

[KSK+15] Kronawitter, Stefan ; Schmitt, Christian ; Kuckuk, Sebastian ; Hannig, Frank ; Teich, Jürgen ; Lengauer, Christian ; Köstler, Harald ; Rüde, Ulrich: *ExaSlang and the ExaStencils code generator*. Online, 2015. – Presentation at PASC'15 in Zurich, Switzerland. Available online under https://www10.cs.fau.de/publications/talks/2015/Kuckuk_Zuerich_PASC15_2015-06-02.pdf; looked up on 25th February 2016

[LGK+14] Lengauer, Christian ; Grösslinger, Armin ; Kronawitter, Stefan ; Grebhan, Alexander ; Apel, Sven ; Bolten, Matthias ; Hannig, Frank ; Köstler, Harald ; Rüde, Ulrich ; Teich, Jürgen ; Kuckuk, Sebastian ; Rittich, Hannah ; Schmitt, Christian: ExaStencils: Advanced Stencil-Code Engineering. In: *Euro-Par 2014 Workshops, Part II* (2014), S. 553 – 564

[LK15] Lengauer, Christian ; Kronawitter, Stefan: *Optimizations Applied by the ExaStencils Code Generator*. Website, 2015. – Available online under http://www.infosun.fim.uni-passau.de/publications/docs/KroLe2015tr.pdf; looked up on 4th April 2016

[McC13] McCalpin, John D.: *Native Computing and Optimization on the Intel® Xeon Phi™ Coprocessor*. Online, 2013. – Available online under https://portal.tacc.utexas.edu/documents/13601/933270/MIC_Native_2013-11-16.pdf/56b4a5c9-be24-4c41-8625-eee21879ca8b; looked up on 29th March 2016

[Rah13] Rahman, Rezaur: *Intel® Xeon Phi™ Coprocessor Architecture and Tools - The Guide for Application Developers*. Apress Open, 2013. – ISBN 978–1–4302–5926–8

[RBK15] Rittich, Hannah ; Bolten, Matthias ; Kahl, Karsten: *The Mathematics of ExaStencils*. Website, 2015. – Available online under http://materials.dagstuhl.de/files/15/15161/15161.HannahRittich.Slides.pdf; looked up on 16th February 2016

[Wei16] Weisstein, Eric W.: *Heat Conduction Equation*. Website, 2016. – Available online under http://mathworld.wolfram.com/HeatConductionEquation.html; looked up on 26th April 2016

Statement of Authorship

I, Thomas Lang, hereby certify that this bachelor thesis has been composed by myself and describes my own work unless otherwise stated. All references and verbatim extracts have been quoted and all sources of information have been specifically acknowledged. In addition, this thesis has not been accepted in any previous application for a degree.

Passau, 27th April 2016

(Thomas Lang)