UHH Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Bachelorarbeit

Vector Folding for Icosahedral Earth System Models

vorgelegt von

Jonas Tietz

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang:	Informatik
Matrikelnummer:	6702019
Erstgutachter:	Dr. Julian Kunkel
Zweitgutachter:	Nabeeh Jumah
Betreuer:	Nabeeh Jumah, Dr. Julian Kunkel

Hamburg, 2018-03-26

Abstract

The performance of High Performance Computing (HPC) applications become increasingly bound by the access latency of main memory. That is why strategies to minimize accesses to memory and maximize the use of the caches are crucial for any serious HPC application. There is lots of research on the topic of trivial rectangular grids, like using SIMD (single instruction multiple data) instructions, to operate on multiple values at once, or cache blocking techniques, which try to divide the grids into chunks, which fit into the cache. Additionally, there are new interesting techniques for minimizing loads in stencil computations like vector folding.

This thesis will take a look at the theoretical performance benefits, especially vector folding in conjunction with an icosahedral grid, and test them in a simple test case. As a result the performance improved slightly in comparison over traditional vectorization techniques.

Contents

1 Introduction

1.1 Motivation

In physics, engineering, meteorology, among others it is a common problem to have the initial state of a system and try to solve for the state after some amount of time. In the case of meteorology the initial state would be e.g. pressure, wind, temperature, humidity etc at a certain point in time. And the result you want to calculate is the new state (the weather) after some amount of time.

These systems are often described by a set of differential equations. Differential equations describe equations which contain derivatives of themself. Differential equations are divided into two groups. The first are called ordinary differential equations (ODE) and deal with only one independent variable. Yet, a lot of problems that can be solved with differential equations, deal with more than one independent variable. These are solved by using partial differential equations (PDE). Since in meteorology among others, there are usually at least four independent variables, it is important to use PDEs for weather simulations. These variables could for example be three for space, one for each dimesion, and one time variable. A good introduction to differential equations can be found in Bellman et. al. [2].

Unfortunately these equations can not be solved trivially by hand. That is why a numerical method is used. For computer simulations the continuous functions of the input parameters need to be discretised into a grid. The functions can then be evaluated for each grid point, area or volume. Derivatives can be solved by approximating them with methods like the finite difference method.

The finite difference method approximates a given continuous derivative with a difference between two discrete cells divided by their distance. More on that in chapter six by Jacobson et. al. [5]. This can easily be achieved with a so called stencil. A stencil describes the neighboring cells that are needed and the weight associated with them to calculate the new value of a cell. The most basic useful stencil is the 2D five point stencil(see Figure 1.1).

To achieve valid results, the resolution of the grid must be as high as possible and the time steps as small as possible to keep the approximation error small. The problem is, that most stencil code is a 4D problem (three space dimensions and one time) and thus runs in $\mathcal{O}(n^4)$ and therefore will result in greater runtimes and memory footprints.

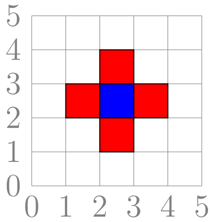


Figure 1.1: 2D five point stencil

1.2 Goal

In this bachelor thesis I will discuss different state of the art techniques to implement icosahedral earth system models. The goal is to implement a new optimization method called vector folding in the context of an icosahedral grid for an earth system test application. Finally the results will be compared with those of C. Yount's paper on vector folding.

1.3 Structure of this thesis

Chapter 2 gives a brief overview over established methods for the kind of computer aided simulations described in Section 1.1. Memory latency problems are discussed as well as cache blocking and basic techniques for data level parallelism.

Chapter 3 explores further techniques, which can be used in icosahedral earth system models. Firstly, I will go into the potential of using dependencies of different stencil to further optimize the performance. In addition to that I explain the difference between basic rectangular grids and icosahedral grids. I will also touch on the topic of the Hilbert space-filling curve to simplify the data layout for those icosahedral grids.

In Chapter 4, I introduce the method of vector folding, which I used in my experiment. I first explain the basic approach to vector folding and then apply it to the icosahedral grid.

Chapter 5 covers the implementation process of the vector folding method for an icosahedral grid. I first lay out my test system, which was used for the measurements. This is followed by each implementation step I took to achieve the end result. Furthermore, I discuss problems and challenges I encountered during the process.

The results are presented in Chapter 6 and Chapter 7 covers the conclusion and provides an overlook over possible future works.

2 Fundamentals

In this chapter, I am going to cover first the basics of numerical simulations and its challenges in computer science. I will then explain two state of the art solutions, which try to reduce these problems.

2.1 Memory Latency Problem

With the increasing speed of the Central Processing Unit (CPU), the problem of memory latency got more and more severe. As a result the access of main memory has become a significant performance bottleneck [7]. That is why CPU manufactures started to include caches.

A cache is a relatively small amount of memory close to the CPU. When loading one value from the main memory, not only that value will be loaded, but a whole cache-line will be stored in the cache for later use. A typical cache line in modern CPUs is 64 bytes. The reason for this is that you will probably need to process data that is close to each other in memory. For example an array stores all its values one after another, and a typical scenario is, that you want to operate on all of them. A typical processor has multiple levels of caches, with different sizes and latencies.

When processing vast amounts of data, it is crucial to process it in such a way that it will be in cache as much as possible. So when you have a simple 2D array you have to process it in memory order. In this example each column is stored consecutively before the beginning of the next row, as can be seen in Figure 2.1.

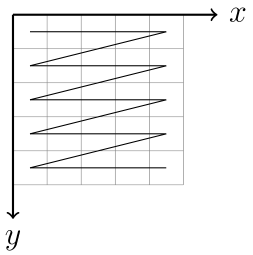


Figure 2.1: Memory layout of a 2D array

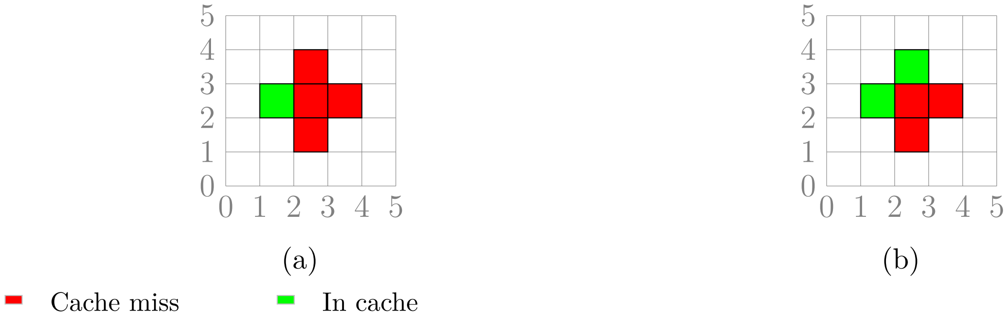
2.2 Cache Blocking

The processing mentioned in Section 2.1 works fine, when you only need the data of that cell or of that row, because it will most likely be in cache. A problem arises when we go back to the 5 point stencil or any other complex enough stencil. This stencil also needs data from the row above and below. This works fine for sufficiently small arrays, because the data of the row above could still be in cache. But when the array gets bigger, the data will be evicted and you will have to load it again, which is suboptimal.

When processing the data as mentioned above with this particular stencil, one datum will definitely be in cache, because it has been processed before and the rest is either a cache miss or in cache depending on if it got loaded in with a cacheline. (see Figure (a) 2.2). This will only get worse, when we move to higher order stencils.

Cache Blocking, as described by Lam et. al. [6], tries to address that problem. It is a technique, where you divide your data into chunks, which fit into the cache (see Figure 2.3). By doing that you can ensure that an already processed row will still be in cache. This means that there are two values definitely in cache, one of which just got processed in the previous step, and the other one got processed in the previous row. The rest behaves like before.(see Figure (b) 2.2).

With higher order and higher dimensional stencil this will lead to increasingly better performance.



(a)

(b)

Figure 2.2: cache behavior of the 5 point stencil

2.3 Vectorization

Another common way to increase the performance of applications is to use single instruction multiple data (SIMD) operations. This is a form of data level parallelism [3]. In a processor, which is able to perform SIMD operations, there are registers, which can hold more than one datum, and instructions which perform on these registers and all of the data at ones. E.g. one instruction could load four integers into a register, add these values to themself and store them back. This would only need three instructions instead of three per integer. This would be twelve in total.

On x64, depending on the processor, you have access to 128 bit wide SIMD-registers up to 512 bit wide SIMD-registers with avx512. Once your data is loaded into such a

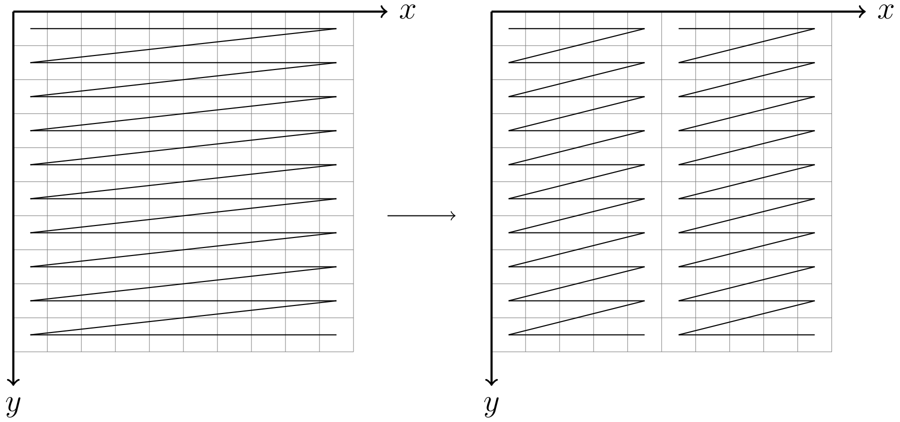


Figure 2.3: Cache Blocking

register you can operate on them using just one instruction and this will therefore lead to more performance. After you are done with all the calculation needed, you have to store them back to memory.

If your data has the right layout, these load and store instructions will only cost you one instruction. But depending on the application, you have to do some work to get the right data in the registers. This overhead will cost you performance.

Another drawback are branches. Branches can not be processed in the usual way. Instead you have to go through all the cases and then use a mask to resolve them later. If you have a loop with early outs, a condition which allows to stop an iteration for a specific datum early, you need to wait until all the values in the register hit the condition for the early out.

Additionally, because of the nature of how to invoke these instructions, you limit your code to a single instruction set architecture or you have to make an implementation for each one. Some of these instructions are also not available on all processors of the same instruction set architecture.

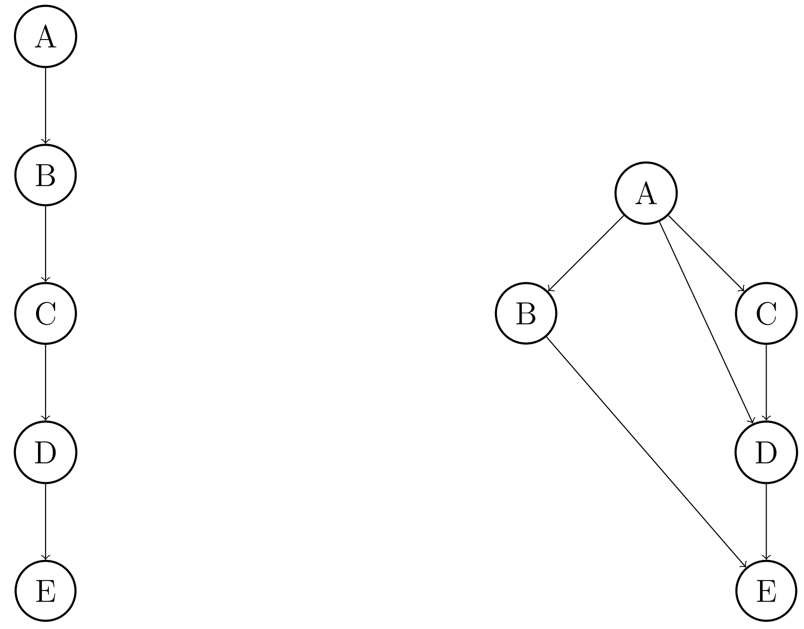
2.4 Chapter Summary

In this and the previous chapter I gave an overview over the basics of numerical analysis and how it is used to solve PDEs, which is needed for earth system model simulations. I then explained why memory latency is a problem and introduced cache blocking as a possible solution to that problem. Finally I talked about SIMD-vectorization, which will become important in the following chapters.

3 Further Techniques

This chapters covers some more advanced techniques for numerical simulations using stencil codes. Especially those used in icosahedral grids. The last section will cover the basics of vector folding which will then be further explored practically in the next chapter.

3.1 Stencil dependencies



(a) Linear dependency

(b) Complex dependencies

Figure 3.1: Dependencies of stencils

Most actual simulations won't have a single stencil to solve a monolithic PDE. It is common practice to split them into smaller PDEs and solve them one after another as mentioned in the section 6.2 in [5] about operator splitting. Some of the stencils that are used to solve those equations are dependent on the previous ones. The normal procedure is to first calculate the first one completely, then the second one etc. Here is some potential for further improvement.

When the previous stencil was just calculated, its data should be in cache. So not using it would waste performance and would thus be inefficient. So instead of processing each stencil at a time, you process all the stencils for each cache block. That way, when the cache blocks are designed the right way, all the data for the stencil should be in cache except the first one, where we still only get the cache behavior of Figure 2.2 (b).

The question is how to design the cache blocks to get the best performance. This question is difficult to answer. The first thing to notice is that the stencils form a dependency graph. If the graph is very linear it might be a good idea to size the cache blocks such that the last one and the current one fit into the cache, because you only ever need the last one. If the graph has a lot of back dependencies to levels higher up in the cache you might want to have more blocks still in cache.

But at some point, when the blocks get too small, the overhead of the cache blocks will get bigger than the performance boost you get from the better cache behavior.

3.2 Icosahedral Grids

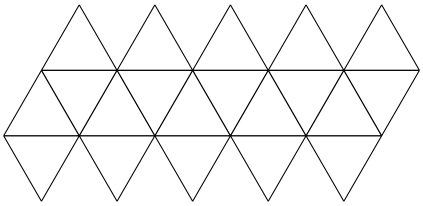


Figure 3.2: Unwrapped icosaheadron

The easiest choice for discretising functions is a simple rectangular grid. They are easy to read and easy to implement. But they carry some disadvantages with them. When designing an earth system model, you need a good projection from your grid onto the sphere of the earth. The problem with rectangular grids is, they over-sample the poles and under-sample the equator.

One solution is to use a grid made out of equilateral triangles to construct an icosahedron. In Figure 3.2 you can see a flattened version of an icosahedron. Because of its sphere-like form it maps way better onto the earth than a cylinder. To increase the resolution, tessellate each triangle into four smaller equilateral triangles(see Figure 3.3). One particular instance of an icosahedric earth system model is described in the ICON paper [9].

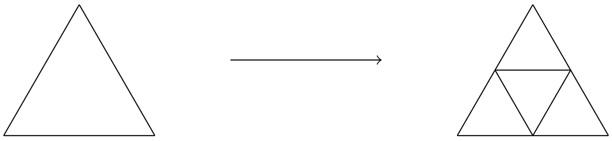
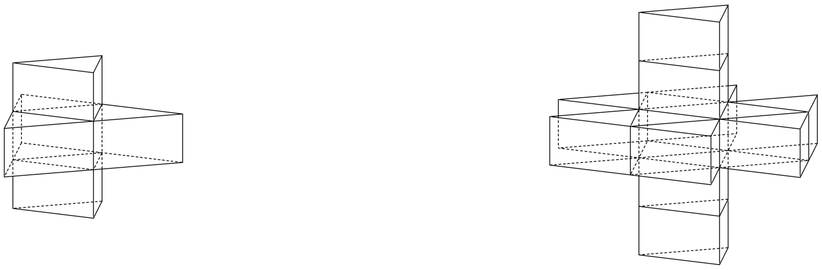


Figure 3.3: Tessellating triangles

Because of the equilateral triangles, the resulting stencils look like in Figure 3.4.



(a) First order Stencil

(b) Second order Stencil

Figure 3.4: Icosahedric Stencil

3.3 Hilbert Curve

All these different ways to layout your data make it complicated to have a modular application. It would be helpful to use an abstraction of the underlying layout. Space filling curves are one way to have for example a 2D layout and transform it to a 1D layout without loosing too much of its spatial properties. The most famous space filling curve is the Hilbert space-filling curve (see Figure 3.5).

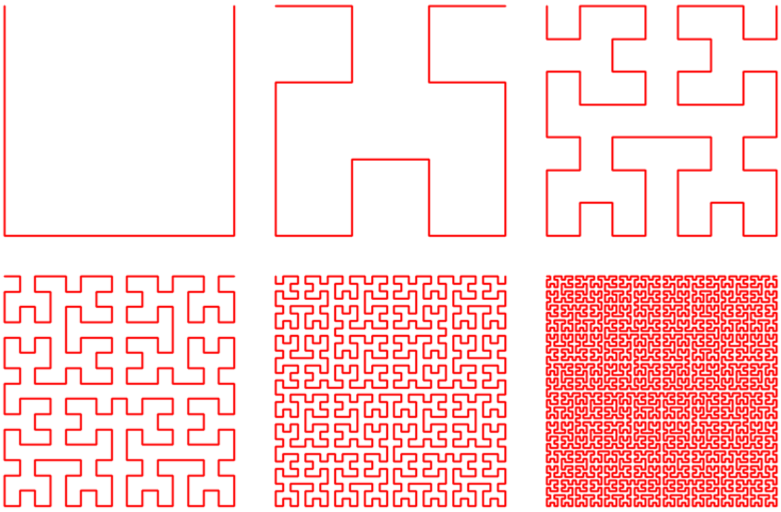


Figure 3.5: Hilbert-Curve on finer and finer grids

Instead of iterating over a complex data-layout, you can simply iterate over a 1D-array. One problem is that you cannot simply get to a neighbour cell just by the index of the current cell. You also have to have a data-structure, which provides you with the neighboring cells of each cell.

3.4 Chapter Summary

In this chapter I explored the possible performance benefits of exploiting the dependencies between different stencils. I then introduce the idea of icosahedral grids as a structure for earth system models. I further explained that the Hilbert space-filling curve is a way of implementing such a structure.

4 Vector folding

In this chapter, I will introduce the method of vector folding. I will first give an brief overview over the fundamentals and then explain how they can be applied to an icosahedral grid. This will later be needed for Chapter 5, because there I will use it in the implementation.

4.1 Standard vector folding

Vector folding is a technique, which builds upon normal SIMD-vectorization. The term was coined by C. Yount in his paper about said technique. I chose use this term, because I also used his method.

The hope is to reduce the footprint of the stencil and overlap load operations by computing cells of multiple dimensions in one vector instead of only one dimension. This was first explored by C. Yount [10]. By altering the memory layout to load for example four values from one row and four from the next row instead of eight from the same row, the overlap of the values needed for the eight stencils is increased.

Figure 4.1 shows the layout of the center points for a given stencil, (a) with normal vectorization and (b) with a 2D-vectorfold. This method assumes that loads are more expensive than other instructions, because more work is needed to fill some SIMD-registers with the right values. The paper by C. Yount [10] called these reconstructions a blend operation. It is also needed to change the memory layout such that one load instruction loads the values from multiple dimensions.

(a) Vectorized

(b) Vector-folded

Figure 4.1: Stencil center points

When comparing the vector folded versions with the vectorized version, the vector folded one will need less aligned load operations and more blend operations.

4.2 Vector folding for icosahedral grids

We can now apply this to an icosahedral grid. In this case we have a second order stencil as shown in Figure 3.4 (b). Using normal SIMD vectorization with AVX2 instructions (8 float wide registers) will result in a stencil footprint like in Figure 4.2. As you can see you will need 13 aligned load operations and 8 blend operations to fill all the necessary registers.

Also some additional overhead is needed for the alternating neighbor.

Using 2D vector folding will result in a footprint which can be seen in Figure 4.3. This will result in 11 aligned loads and 10 blend operations.

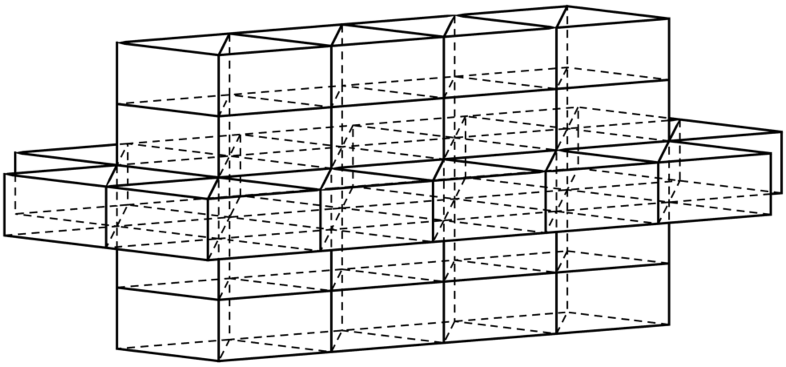


Figure 4.2: Normal vectorization for second order stencil

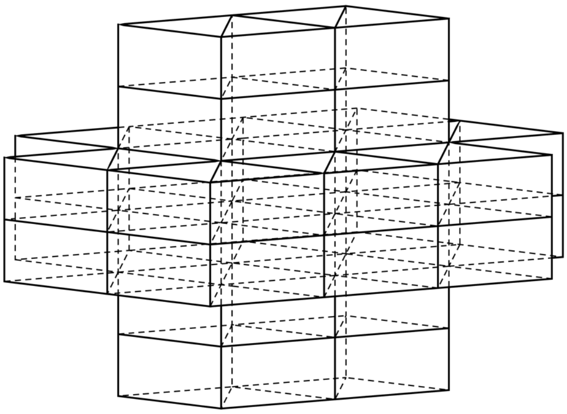


Figure 4.3: 2D vector folded second order stencil

4.3 Chapter Summary

Vector folding is a new and promising method for the optimization of stencil code. That is why I explained the basics of it in this chapter. I then applied vector folding to icosahedral grids. This is important because it is used in the implementation.

5 Implementation

In this chapter I will cover my implementation methodology of vector folding for an icosahedral grid test. Furthermore I will describe the test system I used to measure the performance difference between the vector folding version and the vectorized version.

5.1 Test system

For testing I used an Intel Core i7-7700HQ, which runs at 2.80GHz with 4 cores and 8GB of Ram. The CPU supports SIMD instructions up to AVX2, which gives access to 256 bit SIMD registers (8 single precision or 4 double precision floating point numbers). The GFLOPs per core is according to [1] 3.72 GFLOPs. The compiler I used is GCC version 6.3.0 [8] with the optimization switch -O3

5.2 Testbed

5.2.1 Scalar Version

I implemented the Icosahedric Grid as a two three dimensional array with 512 cells in each dimension. One buffer was used for the data of the timestep, where as the other was used to store the result. Each cell has five neighbors. Left, right, up, down and one alternating front and back. As the stencil I chose a simple second order 14 point stencil like in Figure 3.4 (b), which calculates a blur. All the edges of the grid are wrapped around, such that if you move out of bounds, you are set to the opposite site of that dimension.

I first started out implementing a scalar version with an simple 2D grid and a first order stencil. To enable me to find bugs faster I also implemented a data visualizer, which allowed me to get an fast visual conformation of the whole grid data after each iteration.

I then realized, that for vector folding to work properly, I needed a more complex and higher order stencil. That is why I then moved to a 3d grid with an second order stencil. I used the scalar version as a case to check against to validate my further implementations.

Listing 5.1 contains the main timestep loop, which is mostly shared between all versions discussed in this chapter. I used ftime to time the duration of each time step and stored those timings in an buffer, which will then get written out to csv-file.

In Listing 5.2 you can see the stencil code. WRAP() is a macro, which handles the wrap around if the index leaves the boundary of the array. The blur is implemented by adding up all the cell values of the stencil and dividing them by the number of cells in the stencil.

Between Listing 5.1 and Listing 5.2 the $\mathcal{O}(n^4)$ complexity can easily be seen. Four nested loops. One for the timesteps and one for each dimension.

```
while(IterationCount > 0)
{
  ftime(&Start);
  Timestep(Var,Var2);

  float *** Temp = Var;
  Var = Var2;
  Var2 = Temp;

  ftime(&End);

    // Store the timings until the end of the simulation,
    // where they will be written to disk
  TimingBuffer[ArrayCount(TimingBuffer)-IterationCount] = (long long)(1000*(End.time-Start.time))+(End.millitm -Start.millitm);
  IterationCount--;
}
```

Listing 5.1: Main timestep loop

```
void ScalarTimestep(float ***Var, float ***Var2)
{
  float Weight = (1.0f/14.0f);
  for(int z = 0; z < BUFFER_DIM; z++)
  {
    for(int y = 0; y < BUFFER_DIM; y++)
    {
      for(int x = 0; x < BUFFER_DIM; x++)
      {
        Var2[z][y][x] = Weight*Var[z][y][x];

        // Alternating neighbor
        if(x%2==y%2)
        {
            Var2[z][y][x] += Weight*Var[z][WRAP(y-1)][x];
        }
        else
        {
            Var2[z][y][x] += Weight*Var[z][WRAP(y+1)][x];
        }

        Var2[z][y][x] += Weight*Var[z][y][WRAP(x-1)];
        Var2[z][y][x] += Weight*Var[z][y][WRAP(x+1)];
        Var2[z][y][x] += Weight*Var[WRAP(z-1)][y][x];
        Var2[z][y][x] += Weight*Var[WRAP(z+1)][y][x];

        Var2[z][y][x] += Weight*Var[z][y][WRAP(x-2)];
        Var2[z][y][x] += Weight*Var[z][y][WRAP(x+2)];

        Var2[z][y][x] += Weight*Var[z][WRAP(y-1)][WRAP(x+1)];
        Var2[z][y][x] += Weight*Var[z][WRAP(y-1)][WRAP(x-1)];
        Var2[z][y][x] += Weight*Var[z][WRAP(y+1)][WRAP(x+1)];
        Var2[z][y][x] += Weight*Var[z][WRAP(y+1)][WRAP(x-1)];

        Var2[z][y][x] += Weight*Var[WRAP(z-2)][y][x];
        Var2[z][y][x] += Weight*Var[WRAP(z+2)][y][x];
      }
    }
  }
}
```

Listing 5.2: Scalar stencil code

5.2.2 Vectorized Version

After that I implemented the simple vectorized version. As the instruction set for the vectorization I chose AVX2, so I could do eight single precision floats at once in the inner loop. A description of the available instruction is in the Intel Intrinsic Guide [4]

I loaded all the aligned vectors using the `_mm256_load_ps` instruction. For the blends I either needed one float of the previous/next vector or two and the rest of the other vector. To achieve that I used `_mm256_permutatevar8x32_ps` to put the floats in the right position and `_mm256_blend_ps` to combine them. For example for the x-1 vector I needed to load the x vector and the x-8 vector. I needed to get the highest position float of the x-8 position in the lowest position and all the floats of the x vector one position up. With `_mm256_permutatevar8x32_ps` I implemented a 32 bit word wise rotate. I could then rotate both registers by one float to put all the floats in the right position and combine them with `_mm256_blend_ps` using a mask to decide, which floats to keep from each register.

5.2.3 Vector Folded Version

For the vector folded version I made a 2D fold in the z dimension. therefore the memory layout needed to be changed to (0,0,0), (0,0,1), (0,0,2), (0,0,3), (1,0,0), (1,0,1), (1,0,2), (1,0,3), (0,0,5), (0,0,6), (0,0,7), (0,0,8), (1,0,5), (1,0,6), (1,0,7), (1,0,8) etc. To hold the new data layout the array allocation needed to be altered. The size of the z-dimension is halfed, whereas the x dimension is doubled to fit the extra values.

Because of the new memory layout each aligned vector has now four floats of position z and four of position z+1. To keep the indexing easy I wrote a macro INDEX() as can be seen in Listing 5.3, which is responsible for the transformation of the old indexing to the new indexing. To perform the blend operation in the z dimension, e.g. for the z+1 vector, I needed to get the higher four floats of the lower vector and the lower four floats of the higher vector. This can be achieved by using the instruction `_mm256_permute2f128_ps`. This instruction allows one to make a new vector out of the high and low 128 bits of two vectors.

The instruction takes two vectors and one immediate. The immediate get interpreted by the instruction as two separate values. The first four bits and the following four bits. It uses those to determine which 128 bits to put in the low and which in the high part. A 0 is for the low part of the first argument a, 1 for the high part of the first argument, a 2 is for the low part of the second arguments and a 3 is for the high part of the second argument. If the third bit is set of the 4 bit value the 128 bits are set to zero. I passed 00100001_{bin} as the immediate to get the high 128 bits of the first argument and the low 128 bits of the second.

For the xy-plane I now needed to rotate the floats not over the whole register but in 128 bit units. I could have used `_mm256_permutatevar8x32_ps` again, but by using `_mm256_alignr_epi8` I did not need the `_mm256_blend_ps` instruction. This can be seen in Listing 5.4.

`_mm256_alignr_epi8` takes two vectors and an immediate. It concatenates the low 128 bits of both vectors and the high 128 of both vectors. With the immediate you can then choose which 128 bit part of the two concatenated vectors you want to keep.

As one can see by comparing the Listings 5.3 and 5.4 with the listing 5.2 there is a vast complexity increase and a lot more manual work needs to be done to implement the stencil code. Additionally a major concern is the readability of the code, which is definitely decreased by this kind of assembly-like code.

```
__m256 Left;
__m256 FrontLeft;
__m256 BackLeft;
if(x==0)
{
  Left = _mm256_load_ps(&(INDEX(Var,z,y,BUFFER_DIM-4)));
  FrontLeft = _mm256_load_ps(&(INDEX(Var,z,WRAP(y-1),BUFFER_DIM-4)));
  BackLeft = _mm256_load_ps(&(INDEX(Var,z,WRAP(y+1),BUFFER_DIM-4)));
}
else
{
  Left = _mm256_load_ps(&(INDEX(Var,z,y,x-4)));
  FrontLeft = _mm256_load_ps(&(INDEX(Var,z,WRAP(y-1),x-4)));
  BackLeft = _mm256_load_ps(&(INDEX(Var,z,WRAP(y+1),x-4)));
}

__m256 Right;
__m256 BackRight;
__m256 FrontRight;
if(x==BUFFER_DIM-4)
{
  Right = _mm256_load_ps(&(INDEX(Var,z,y,0)));
  BackRight = _mm256_load_ps(&(INDEX(Var,z,WRAP(y+1),0)));
  FrontRight = _mm256_load_ps(&(INDEX(Var,z,WRAP(y-1),0)));
}
else
{
  Right = _mm256_load_ps(&(INDEX(Var,z,y,x+4)));
  BackRight = _mm256_load_ps(&(INDEX(Var,z,WRAP(y+1),x+4)));
  FrontRight = _mm256_load_ps(&(INDEX(Var,z,WRAP(y-1),x+4)));
}

__m256 Center = _mm256_load_ps(&(INDEX(Var,z,y,x)));
__m256 Front  = _mm256_load_ps(&(INDEX(Var,z,WRAP(y-1),x)));
__m256 Back   = _mm256_load_ps(&(INDEX(Var,z,WRAP(y+1),x)));
__m256 Top2   = _mm256_load_ps(&(INDEX(Var,WRAP(z+2),y,x)));
__m256 Bot2   = _mm256_load_ps(&(INDEX(Var,WRAP(z-2),y,x)));
```

Listing 5.3: 2d vector folding loads

```
__m256 negX1 =
   ↪ (__m256)_mm256_alignr_epi8((__m256i)Center,(__m256i)Left,12);
__m256 negX2 =
   ↪ (__m256)_mm256_alignr_epi8((__m256i)Center,(__m256i)Left,8);
__m256 X1 = (__m256)_mm256_alignr_epi8((__m256i)Right,(__m256i)Center,4);
__m256 X2 = (__m256)_mm256_alignr_epi8((__m256i)Right,(__m256i)Center,8);

__m256 negX1negY1 =
   ↪ (__m256)_mm256_alignr_epi8((__m256i)Front,(__m256i)FrontLeft,12);
__m256 negX1Y1 =
   ↪ (__m256)_mm256_alignr_epi8((__m256i)Back,(__m256i)BackLeft,12);
__m256 X1negY1 =
   ↪ (__m256)_mm256_alignr_epi8((__m256i)FrontRight,(__m256i)Front,4);
__m256 X1Y1 =
   ↪ (__m256)_mm256_alignr_epi8((__m256i)BackRight,(__m256i)Back,4);

__m256 Top1 = _mm256_permute2f128_ps(Center,Top2,0b00100001);
__m256 Bot1 = _mm256_permute2f128_ps(Bot2,Center,0b00100001);

__m256 Mask = _mm256_castsi256_ps(_mm256_set_epi32(0xFFFFFFFF, 0,
   ↪ 0xFFFFFFFF, 0, 0xFFFFFFFF, 0, 0xFFFFFFFF, 0));

// Alternating neighbor
__m256 Neighbor3;
__m256 NeighborBackLeft;
__m256 NeighborBackRight;
__m256 NeighborFrontLeft;
__m256 NeighborFrontRight;
if(x%2!=y%2)
{
  Neighbor3 = _mm256_blendv_ps(Back,Front,Mask);
  NeighborBackLeft = _mm256_blendv_ps(negX1Y1,negX1negY1,Mask);
  NeighborBackRight = _mm256_blendv_ps(X1Y1,X1negY1,Mask);
  NeighborFrontLeft = _mm256_blendv_ps(negX1negY1,negX1Y1,Mask);
  NeighborFrontRight = _mm256_blendv_ps(X1negY1,X1Y1,Mask);
}
else
{
  Neighbor3 = _mm256_blendv_ps(Front,Back,Mask);
  NeighborBackLeft = _mm256_blendv_ps(negX1negY1,negX1Y1,Mask);
  NeighborBackRight = _mm256_blendv_ps(X1negY1,X1Y1,Mask);
  NeighborFrontLeft = _mm256_blendv_ps(negX1Y1,negX1negY1,Mask);
  NeighborFrontRight = _mm256_blendv_ps(X1Y1,X1negY1,Mask);
}
```

Listing 5.4: 2d vector folding blends

5.3 Implementation Complexity

During the implementation process I encountered many problems, although most major problems involved the basic vectorized version and the vector folding version. As I never used SSE or AVX instructions before I had great difficulties finding the right instructions for combining the vectors in the right way. Additionally I found the documentation for some of those instructions hard to follow.

Another difficulty I had was understanding how I needed to combine the loaded vectors to get the right values. The last major problem was getting the changed memory layout to work. I spend lots of time debugging the indexing into the new memory layout before I got it right. Additionally the addresses for the load instructions needed to aligned to 32 byte boundary, which I did not notice at first. This then actually helped me later because it did uncover errors in my indexing, by simply crashing the simulation.

Also due to the implementation details the complexity of the code increased dramatically. Comparing the code in Listing 5.2 with Listings 5.3 and 5.4 you can see a drastic increase in lines of code. Listing 5.2 has 40 lines of code including the whole loop over all three dimension, where as the other listings have 74 lines of code only include the loads and the blend operations. They don't even include the actual stencil code.

Due to the lost time from debugging, I could not explore further techniques, like the exploitation of stencil dependencies.

5.4 Chapter Summary

In this chapter, I talked about the process of implementing vector folding for a simple simulation. I first established a basic scalar version of the stencil code. Afterwards, I changed the code to use SIMD vectorization, which I will then use as a comparison to the vector folding version in the following chapter. I then described how I implemented the vector folding version. Finally I discussed various problems I encountered during the implementation process.

6 Results

For the measurement I let it run for 1000 iterations over the grid on a single core and I measured the time of each of those Iterations. I choose to only measured the iterations instead of the whole program, because I only wanted to know the performance of the loop and not fixed costs like buffer allocation and other set up work. Each buffer of the simulation used roughly 514MB of ram. 512MB for the actual values plus 2MB + 4KB for the pointers in the standard vectorized version and 2MB + 2KB for the vector folding version. Because the simulation used two buffers the total memory footprint should be around 1GB.

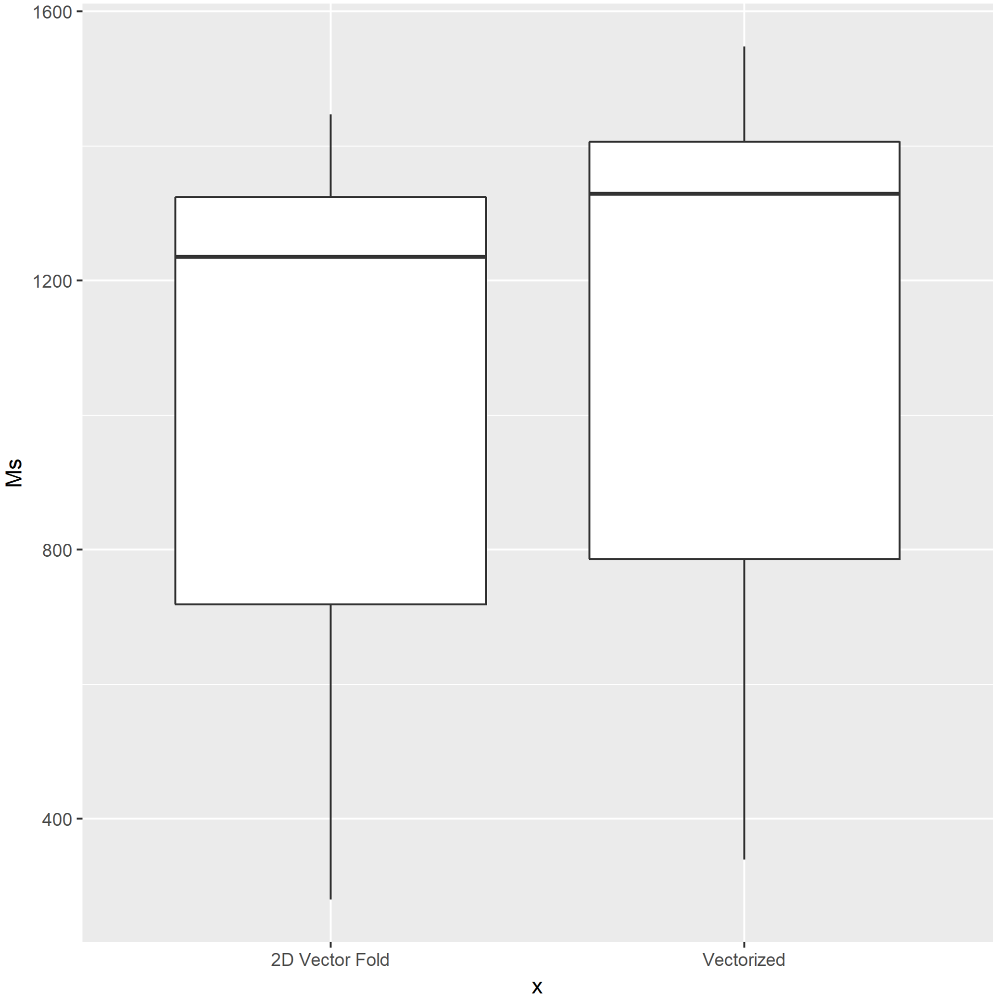


Figure 6.1: Measurement of the Vector Fold and normal Vectorization

The mean of the 2D vector fold version is 1026.58ms and the mean of the normal vectorized version is 1110.46ms. The vector folded version is therefore 8% times faster than the normal version. Unfortunately, there is a wide range of different varying time measures per iteration, which is most likely operating system interference. Because the stencil code uses 14 multiplications and 13 additions per cell and iterates over a total of 512^3 cells, the resulting GFLOPs for the standard vectorization are 3.26 GFLOPs and for the vector folding version 3.53 GFLOPs. When comparing with the maximum GFLOPs per core of the used CPU as mentioned in Section 5.1 that we are already near the maximum amount of GFLOPs of the test system.

This is just a marginal improvement over the simpler vectorized version, but for more iterations and bigger grids the difference will add up. Though when compared to the results in the paper by C. Yount [10], it is less than I expected. The reason for that could be that the stencil is not of a high enough order and therefore does not save enough loads. In C. Yount's paper, the smallest stencil was a 4th order 13 point stencil, while I only used a second order stencil.

7 Summary and future work

The goal of this bachelor thesis was to implement vector folding for an icosahedral grid in order to extend the repertoire of tools and techniques used in earth system models to address the memory latency problem.

It started out with an overview over the fundamental techniques used in earth system models. It first introduced numerical analysis as a basis of these models. It then presented the problem of the increasing disparity between computing speed and memory latency. Cache blocking was then introduced as a standard technique to tackle this problem. It then covered SIMD vectorization as the foundation of vector folding.

The thesis then introduced icosahedral grids and other methods used in simulations of earth system models like the Hilbert space-filling curve. Afterwards, it covered the basics of vector folding and how to apply it to icosahedral grids.

After that a description of the implementation process was given, which gave an overview over the instruction used to achieve the vector folding and the problems which I encountered during the implementation. Finally I presented the performance results of the vector folded version and compared it with the basic vectorized version.

In conclusion, I think this performance optimization is not worth using, if you have to do it by hand, especially for low order stencils, although the effectivness of higher order stencil needs to be tested in the future. It yields not much of a benefit and the implementation is error prone. If a special purpose compiler, like the one used in the YASK paper [11], did the work of the implementation it would definitely be worth using, because the compiler would save the time, which will be lost trying to implement and maintain it.

Further work could be done in testing different orders of stencils and combining them with other standard techniques for earth system model simulations to see how they perform in combination. Also it would be interesting to see the effects of cache prefetching on the overall performance. Additionally, there is the possibility to port the code to the newer AVX512 instructions for a higher amount of data level parallelism and more modern instruction.

Bibliography

[1] Asteroidsathome website. `https://asteroidsathome.net/boinc/cpu_list.php`.

[2] R. Bellman and K.L. Cooke. *Modern Elementary Differential Equations.* (Addison-Wesley series in mathematics.). Dover Publications, 1995.

[3] R. Espasa and M. Valero. Exploiting instruction- and data-level parallelism. *IEEE Micro*, 17(5):20–27, Sep 1997.

[4] Intel. Intel intrinsic guide. `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`.

[5] Mark Z. Jacobson. *Fundamentals of Atmospheric Modeling.* Cambridge University Press, 2 edition, 1999.

[6] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. *SIGPLAN Not.*, 26(4):63–74, April 1991.

[7] Sally A. McKee. Reflections on the memory wall. In Stamatis Vassiliadis, editor, *Proceedings of the 1st conference on Computing frontiers*, page 162, New York, NY, 2004. ACM.

[8] GCC team. Gcc website. `https://gcc.gnu.org/`.

[9] H. Wan, M. A. Giorgetta, G. Zängl, M. Restelli, D. Majewski, L. Bonaventura, K. Fröhlich, D. Reinert, P. Rípodas, L. Kornblueh, and J. Förstner. The icon-1.2 hydrostatic atmospheric dynamical core on triangular grids â€“ part 1: Formulation and performance of the baseline version. *Geoscientific Model Development*, 6(3):735–763, 2013.

[10] C. Yount. Vector folding: Improving stencil performance via multi-dimensional simd-vector representation. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pages 865–870, Aug 2015.

[11] C. Yount, J. Tobin, A. Breuer, and A. Duran. Yask - yet another stencil kernel: A framework for hpc stencil code-generation and tuning. In *2016 Sixth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*, pages 30–39, Nov 2016.

Appendices

List of Figures

List of Listings

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Studiengang XXX selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ort, Datum Unterschrift

Veröffentlichung

Ich bin damit einverstanden, dass meine Arbeit in den Bestand der Bibliothek des Fachbereichs Informatik eingestellt wird.

Ort, Datum Unterschrift