FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

DEPARTMENT OF COMPUTER SCIENCE

Project outside course scope

Accelerating Ocean Modelling

Adressing performance bottlenecks of the ocean modelling framework Veros

Till Grenzdörffer `vmt184@alumni.ku.dk`

Supervisor
Cosmin Eugen Oancea `cosmin.oancea@di.ku.dk`

1 Introduction

Currently many scientists are using purely sequential software for ocean modelling, leading to long simulation times and inefficient use of modern hardware. The aim of this project is to tackle this problem by introducing and optimizing highly parallel code that uses the potential of modern GPUs to accelerate the modelling process.

The ocean modelling framework this project is based on is Veros [HJE+18]. It is written in python and uses Jax [BFH+18] to parallelize large parts of the computations. However, there is reason to believe that the parallel code generated by Jax (specifically on GPUs) is not optimal and warrants the implementation of specific bottlenecks in a language like CUDA, possibly generated by Futhark.

Another part of this project is the integration of CUDA code into Jax through XLA calls. This allows an easy use of the parallel code through a python interface.

2 Tridiagonal Solver

In this project I first investigated the implementation of a tridiagonal solver that is used throughout Veros. The requirement for this solver is that it solves many (ca. 50000) tridiagonal systems with comparably small diagnonals (ca. 100) as fast as possible.

I considered two algorithms for solving tridiagonal systems for this task and experimented with different performance optimizations for both.

Thomas Algorithm

One simple algorithm for solving a tridiagonal system is the Thomas Algorithm. Pythonlike pseudocode for it is shown in fig. 0.1. We can see that in the both for-loops there are true RAW-dependencies across iterations, which means that the algorithm itself is not trivially parallelizable. However, in our case we want to solve many small tridiagonal systems, so we can parallelize over the different systems and leave the Thomas Algorithm in its sequential form. As long as we have enough systems to fully utilize the GPU it is running on, this implementation seems like a good approach.

Figure 0.1: Thomas algorithm in pythonlike pseudocode

```
for i in range(1, n):
    w = a[i] / b[i - 1]
    b[i] += -w * c[i - 1]
    d[i] += -w * d[i - 1]

out[-1] = d[-1] / b[-1]

for i in range(n - 2, -1, -1):
    out[i] = (d[i] - c[i] * out[i + 1]) / b[i]
```

Thomas Algorithm - Coalesced

In the trivial algorithm, successive iterations of the recurrent loop access neighboring data in memory. Since we assign one tridiagonal system to each thread, neighboring threads access data with the stride of the size of the tridiagonal system. This means that the accesses are uncoalesced and we can improve the performance significantly by changing the underlying data layout. In this case the change is trivial - we just need to transpose the input matrix, which can be done efficiently using shared-memory tiling and tranpose the result back afterwards.

The benchmarks show clearly that the benefit of the transposition on the main kernel outweighs the cost of the transposition and thus increases the performance by a factor of XXXX.

Thomas Algorithm - Loop unrolling

Since in our use case the size of the tridiagonal systems is relatively small, it might make sense to unroll the recurrent loops. However, this is only possible if we know the size of the systems at compile time, or compile the CUDA kernel at runtime (for example using CuPy [OUN+17]). However, I expect the benefit of this to be too small to warrant the significant overhead of JIT compilation.

Flat version

The flat version of the tridiagonal solver is based on the observation, that all recurrences in the Thomas algorithm can be replaced with a scan with an appropriate operator. For example, after inlining the computation of w and replacing the +=, we get the following line:

```
b[i] = b[i] - a[i] * c[i - 1] / b[i - 1]
```

As explained in [ABB+16] in more detail, this statement matches the pattern $b_i = a_i + c_i / b_{i-1}$, which can be replaced with a scan that uses 2x2 matrix multiplication as its operator. Similarly, the other recurrences can be replaced by a scan with linear function composition.

I implemented this flat algorithm in CUDA using the thrust library, which provides a scan implementation with custom operators on the GPU. For comparison, I also included the Futhark implementation of the authors of [ABB+16] from the futhark benchmarks repository.

Compared to the Thomas algorithm, this algorithm is more computationally expensive, however if the size of the tridiagonal systems is large enough or we have sufficiently parallel hardware, the benefits of parallelization should outweigh the additional computational load.

Flat version - shared memory

Since my initial implementation of the flat algorithm used thrust scans, the computation involved the launch of 6 kernels in total. This means that there are a lot of redundant accesses to global memory.

In order to evaluate the effect of this, I also implemented a version of the flat algorithm that is computed completely within shared memory.

Here a block is assigned to each tridiagonal system (given that the system is not too large to fit shared memory), and the flat algorithm is computed using intra block scans. For this I used parts of the code handouts from the PMPH course.

Precision

In order to be useful, all of the implementations of course need to produce results that match a reference implementation. However, due to the nature of floating point computation, this will almost never be exact.

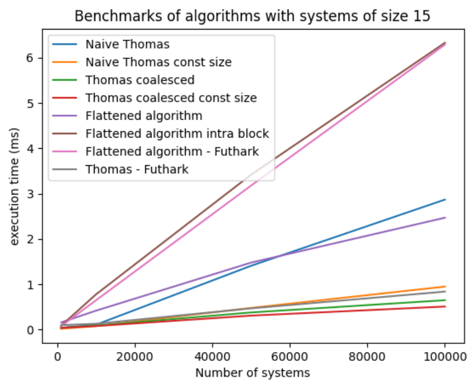
Interestingly, the error margin varied a lot between the algorithms. The flat implementations suffered far greater imprecision, probably at least partly due to the increased parallelism and thereby more indeterminstic order of execution, but also just due to the higher number of floating point calculations.

However, in Veros the tridiagonal systems this solver is supposed to work on are guaranteed to be diagonally dominant. Testing the algorithms on systems with this property showed that all algorithms produce meaningful results.

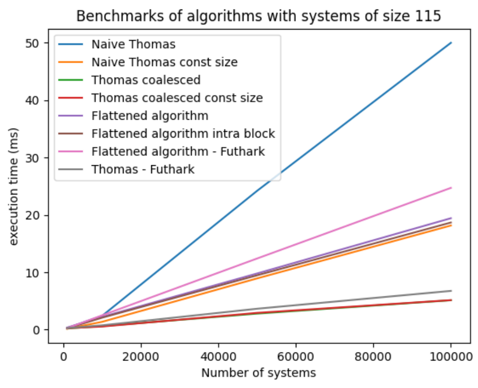
Futhark

In addition to the CUDA implementations, I also included a sequential and a flat implementation (from futhark benchmarks) in the benchmarks in order to see how close the high level description of the algorithms compiled by the futhark compiler will come to CUDA implementations that were optimized by hand.

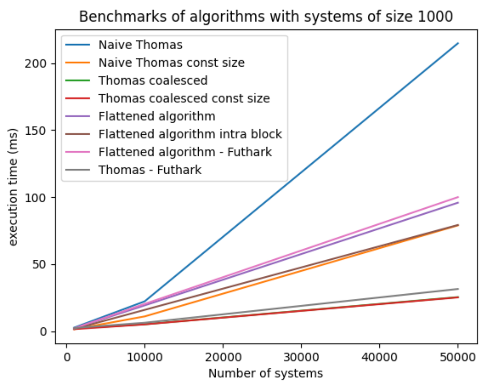
Benchmarks



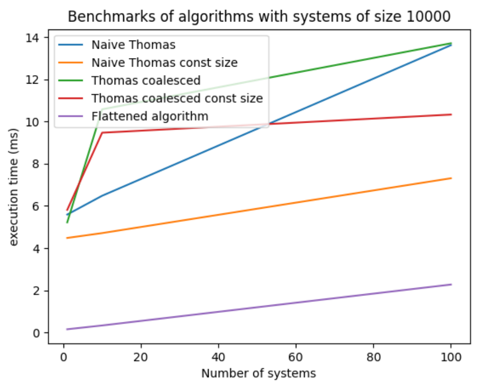
(a) Tridiagonal systems of size 15



(b) Tridiagonal systems of size 115



(c) Tridiagonal systems of size 1000



(d) Tridiagonal systems of size 10000

The performance of the different algorithms are shown in figs. 0.2a-0.2d. All benchmarks were executed on a RTX 2070 Super GPU. The algorithms were tested on a different number of systems, and on systems of sizes 15, 115, 1000 and 10000. In our target application the size 115 is the most relevant and the algorithms should perform well on ca. 50000 systems.

In fig. 0.2a we can see that for small systems the flat algorithms are not performing very well. Because of the small size of the systems, the overhead for parallelizing the recurrences outweighs the benefits of the parallelization.

The sequential algorithms perform similarly well, though it is noteworthy that the performance of the uncoalesced Thomas algorithm with compile-time known size of the systems performs almost on par with the coalesced versions. The small stride of 15 still leads to decent locality of reference and the transpositions needed for the coalesced versions could make up the rest of the difference.

For our target system size of 115 we can see that the naive Thomas implementation performs by far the worst. The flattened versions now achieve a similar performance as the trivial Thomas algorithm with compile-time know size. The best performance is shown by the coalesced versions, independant of whether the size of the systems is known at compile time. However, the futhark implementation follows right behind. It is important to mention here that the futhark implementation does not contain any special precautions that would result in coalesced access. The futhark compiler deduced the need of transpositions itself and came up with a solution that almost matches hand-optimized CUDA code.

With systems of size 1000 we see the same pattern as for size 115. Apparanetly the systems are still not large enough to reap the benefits of the increased parallelism (at least on the machine used). However, we can see a noticable improvement in performance by the intra block flat version as compared to the implementation that uses thrust. The use of shared memory to avoid global memory accesses starts to pay off.

Lastly, I tested the algorithms on smaller batches of large systems of size 10000. Here the best implementation is indeed the flat version. The sequential version are all similarly slow. The sharp bend in the performance of the coalesced versions is due to the transposition being a no-op for a single system. The futhark and intra block implementations are not included here, since the systems were too large to fit into shared memory.

3 Turbulent kinetic energy

After integrating the tridiagonal solver into Jax (see section 4), I decided to look at one example context in which the solver would be used.

The turbulent kinetic energy benchmark is a larger routine that uses different components like the tridiagonal solver. It acts on a three dimensional grid and contains a lot of stencil computations.

Components

The turbulent kinetic energy routine mainly consists of the following parts:

1. Initializing a lot of tridiagonal systems
2. Solving these systems
3. Multiple 2D stencil operations
4. The Superbee scheme

After integrating my tridiagonal solver and profiling the benchmark using tensorboard, which is well supported by Jax, I found that the Superbee scheme was the largest bottleneck.

Superbee scheme

The superbee scheme is a stencil calculation that also works on a 3D grid. It computes three flux grids, one for each dimension and each using a stencil along the respective dimension. The indexing pattern of the stencil is shown in fig. 0.3 and a 2D projection of it is visualized in fig. 0.4.

A very straightforward observation about this pattern is, that there is an overlap of the three computations. This means that when implementing this scheme in CUDA it makes sense to fuse the computations together, such that the value $grid[x, y, z]$ only has to be read once from global memory.

Figure 0.3: Superbee stencil access pattern

```
for x in range(width):
    for y in range(height):
        for z in range(depth):
            flux_east[x,y,z] =  f(grid[x-2,y,z],
                                  grid[x-1,y,z],
                                  grid[x,  y,z],
                                  grid[x+1,y,z])

            flux_north[x,y,z] = f(grid[x,y-2,z],
                                  grid[x,y-1,z],
                                  grid[x,y,  z],
                                  grid[x,y+1,z])

            flux_top[x,y,z] =   f(grid[x,y,z-2],
                                  grid[x,y,z-1],
                                  grid[x,y,z  ],
                                  grid[x,y,z+1])
```

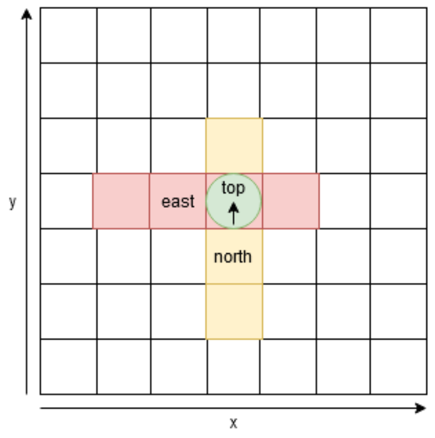


Figure 0.4: 2D projection of the stencil shape. The green dot symbolizes the flux in "top" direction, going along the z axis.

Tiled stencil computation

Benchmarks

4 Interfacing with Jax

Uses XLA Integrating CUDA code with Cython Needs modified compiler... blah

Bibliography

[ABB$^+$16] Andreetta, Christian ; Bégot, Vivien ; Berthold, Jost ; Elsman, Martin ; Henglein, Fritz ; Henriksen, Troels ; Nordfang, Maj-Britt ; Oancea, Cosmin E.: Finpar: A parallel financial benchmark. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 13 (2016), Nr. 2, S. 1–27

[BFH$^+$18] Bradbury, James ; Frostig, Roy ; Hawkins, Peter ; Johnson, Matthew J. ; Leary, Chris ; Maclaurin, Dougal ; Necula, George ; Paszke, Adam ; VanderPlas, Jake ; Wanderman-Milne, Skye ; Zhang, Qiao: *JAX: composable transformations of Python+NumPy programs*. http://github.com/google/jax. Version: 2018

[HJE$^+$18] Häfner, D. ; Jacobsen, R. L. ; Eden, C. ; Kristensen, M. R. B. ; Jochum, M. ; Nuterman, R. ; Vinter, B.: Veros v0.1 – a fast and versatile ocean simulator in pure Python. In: *Geoscientific Model Development* 11 (2018), Nr. 8, 3299–3312. http://dx.doi.org/10.5194/gmd-11-3299-2018. – DOI 10.5194/gmd–11–3299–2018

[OUN$^+$17] Okuta, Ryosuke ; Unno, Yuya ; Nishino, Daisuke ; Hido, Shohei ; Loomis, Crissman: CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations. In: *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*, 2017