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 [1]. It is written in python and uses Jax [2] 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 [3].

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. A tridiagonal system of size n is shown in fig. 1, where $\boldsymbol{a}$, $\boldsymbol{b}$ and $\boldsymbol{c}$ are vectors of size n and $a_0 = c_n = 0$

The diagonals $\boldsymbol{a}$, $\boldsymbol{b}$ and $\boldsymbol{c}$, as well as $\boldsymbol{d}$ are given and the task of the solver is to find $\boldsymbol{x}$.

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 & 0 & 0 & 0 & 0 \\ a_2 & b_2 & c_2 & 0 & 0 & 0 & 0 & 0 \\ 0 & a_3 & b_3 & c_3 & 0 & 0 & 0 & 0 \\ 0 & 0 & a_4 & b_4 & c_4 & 0 & 0 & 0 \\ 0 & 0 & 0 & a_5 & b_3 & \dots & 0 & 0 \\ 0 & 0 & 0 & 0 & \dots & \dots & \dots & 0 \\ 0 & 0 & 0 & 0 & 0 & \dots & \dots & c_{n-1} \\ 0 & 0 & 0 & 0 & 0 & 0 & a_n & b_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \dots \\ \dots \\ \dots \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \\ \dots \\ \dots \\ \dots \\ d_n \end{bmatrix} \tag{1}$$

The requirement for the solver in Veros is that it solves many (ca. 50000) tridiagonal systems of comparably small size (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 eq. 1, where a, b, c, d and x refer to the respective vectors shown in eq.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 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]

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

for i in range(n - 2, -1, -1):
    x[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.

Of course this is only beneficial if the time saved in the main kernel is larger due to coalesced access than the time spent tranposing the inputs and results.

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 [4]). 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 [5] 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 [5] 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.

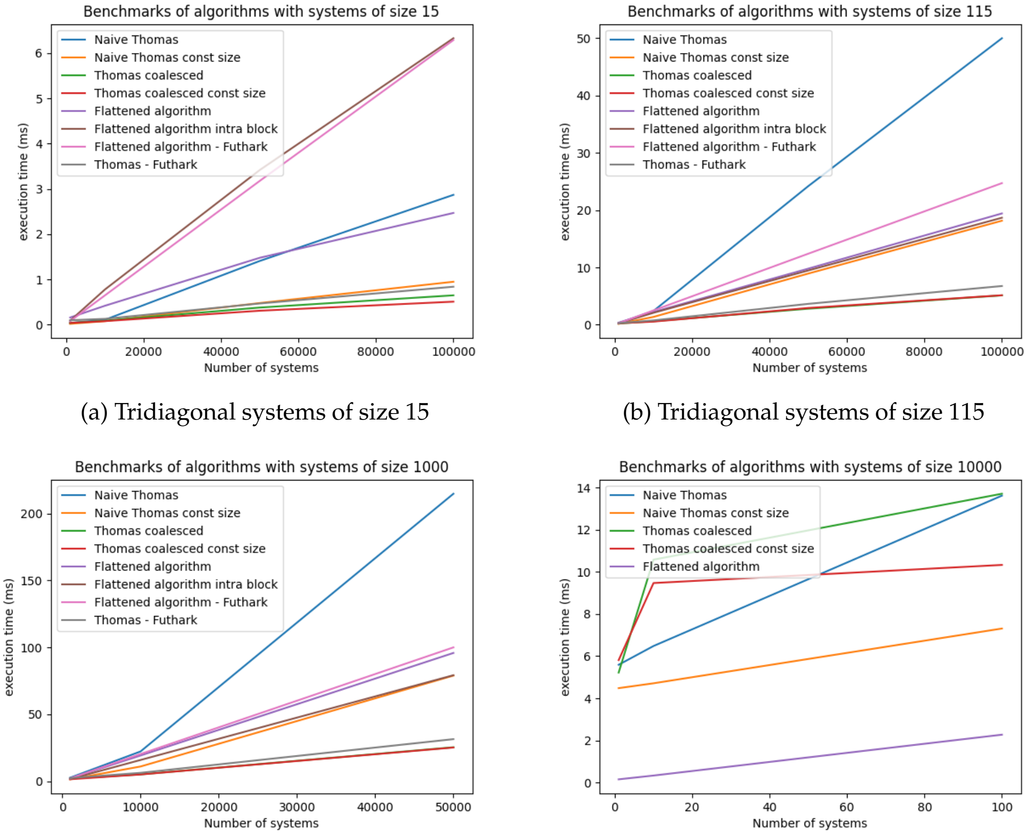
Interestingly, the error margin varied a lot between the algorithms. The flat implementations suffered far greater imprecision, probably at least partly due to the increased parallelism and thereby more indeterminstic order of execution, but also just due to the higher number of floating point calculations.

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. 2a-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. 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 instructions 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.

Conclusion

For this particular application (and the hardware I tested the algorithms on) we can see that due to the small size of the tridiagonal systems it makes most sense to have an algorithm that works sequentially on each of the systems and optimizes the memory layout such that this algorithm can be executed on many systems in parallel.

The flattened algorithm make sense in other cases though. If there are few or even only a single large tridiagonal system to solve, they outperform the sequential approach. If the system is small enough to fit into shared memory we can compute the solution within a single kernel call.

We can see that there are lots of parameters that influence which solution is the most optimal. Thus we can use apply incremental flattening [6] here to compile different versions of the algorithm and choose dynamically, which version is going to be executed, depending on system capabilities and the size of the problem.

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.

This routine is responsible for qunatifying the effect of small-scale water turbulence on the overall state of the simulation. Since we work on a discrete grid with a fixed resolution, effects that occur within one grid cell will not be taken into account. Thus it is important to use the quantifcation of this effect, offered by the turbulent kinetic energy routine, such that these turbulence do not need to be calculated in detail.

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 part of a stencil calculation that also works on a 3D grid. Its purpose is to calculate the influence of advection of tracers within the water and will result in three values for each of the grid cell. One value corresponding to flux along the first dimension of the grid (named flux_east), one along the second dimension (named flux_north) and one along the third dimension (named flux_top).

Since I do not have sufficient knowledge of the physics and mathematics behind this operation, I looked at this problem purely from a coding perspective. More specificly I tried to understand the pattern of memory accesses and tried to map the computation efficiently to CUDA code.

The indexing pattern of the stencil is shown in fig. 3 and a 2D projection of it is visualized in fig. 4.

Figure 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])
```

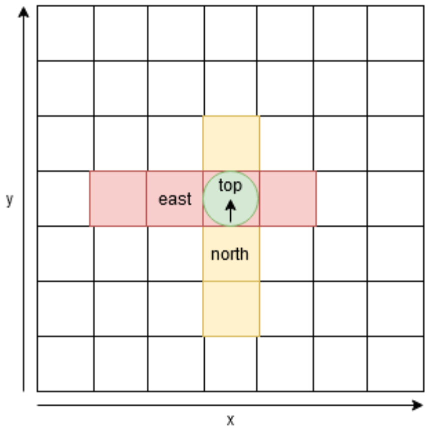


Figure 4: 2D projection of the stencil shape. The green dot symbolizes the flux in "top" direction, going along the z axis.

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 and can be stored in registers.

Tiled stencil computation

In order to save more accesses to global memory, we can observe that successive iterations of this stencil computation will also read almost the same data. For example, in the iteration $x = 0; y = 0; z = 2$ we read $grid[0, 0, 0]$, $grid[0, 0, 1]$,$grid[0, 0, 2]$ and $grid[0, 0, 3]$ and in iteration $x = 0; y = 0; z = 3$ we read $grid[0, 0, 1]$, $grid[0, 0, 2]$,$grid[0, 0, 3]$ and $grid[0, 0, 4]$. We can see that 75% of the memory locations are shared between the iterations.

In order to use this to our advantage, we need to make use of shared memory. By tiling the loops and loading 3D blocks of the data into shared memory, we can avoid a lot of global memory accesses. This *overlapped tiling* approach is described in [7]. Pseudocode for the tiled approach is shown in fig. 5.

We can see that there are still some redundant loads on the border of the tiles, however if the tile is large enough these are negligible.

Loading the grid data to shared memory in coalesced fashion is not exactly trivial, since there is a mismatch between the number of threads that are loading the data ($TILE\_SIZE^3$) and the number of entries to be loaded from global memory: $(TILE\_SIZE + 3)^3$.

For this I used the lmod construct...

Turbulent Kinetic Energy in Futhark

After further profiling of the turbulent kinetic energy Jax implementation with custom operations for the tridiagonal solver and the superbee scheme, it became apparent that most of time is not spent on computation. A lot of gather

Figure 5: Tiled Superbee stencil

```
for xx in range(0, width, TILE_SIZE):
    for yy in range(0, height, TILE_SIZE):
        for zz in range(0, depth, TILE_SIZE):
            # load block grid[xx-2:xx+TILE_SIZE+1,
            #                 yy-2:yy+TILE_SIZE+1,
            #                 zz-2:zz+TILE_SIZE+1]
            # to shared memory in coalesced fashion
            shared_memory[...] = grid[...]
            for x in range(TILE_SIZE):
                for y in range(TILE_SIZE):
                    for z in range(TILE_SIZE):
    # wrong indentation so code fits the page
    # all below this is inside all loops
                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])
```

operations were performed, each in a seperate kernel, which led to the assumption that a lot of arrays are created for intermediate results. Most of these operations seemed to correspond to array slicing which was used extensively in the Jax code.

To check this hypothesis I implemented the full routine in Futhark. Interestingly, the futhark code (compiled to opencl) ran slightly slower than the Jax code with GPU backend. However, after using Futhark's autotune feature the program sped up by a factor of 4, outperforming Jax by a huge margin. This again shows the importance of the use of incremental flattening to match the problem proportions and the available hardware resources.

Benchmarks

4 Interfacing with Jax

In order to use the optimized CUDA code within the Veros framework, I decided to use the wrappers around Tensorflow's [8] XLA Custom Calls that are provided by the Jax library. The custom calls are used to encapsulate native code and directly pass pointers to device memory, which can then be read and written by the CUDA kernels. The whole pipeline is shown in fig. 6.

An interesting feature of the XLA custom call API is the option to define a memory layout for the data passed to the native code. We can use this for the implementation of the tridiagonal solver. In the python routines that use the tridiagonal solver the systems are stored in such a way that consecutive elements of one diagonal are stored consecutively in memory and all n systems are stored one after the other. However with the custom calls we can decide to store consecutive elements with a stride of n, effectively transposing the input and allowing for coalesced access with the implementation of the Thomas algorithm shown in section 2.

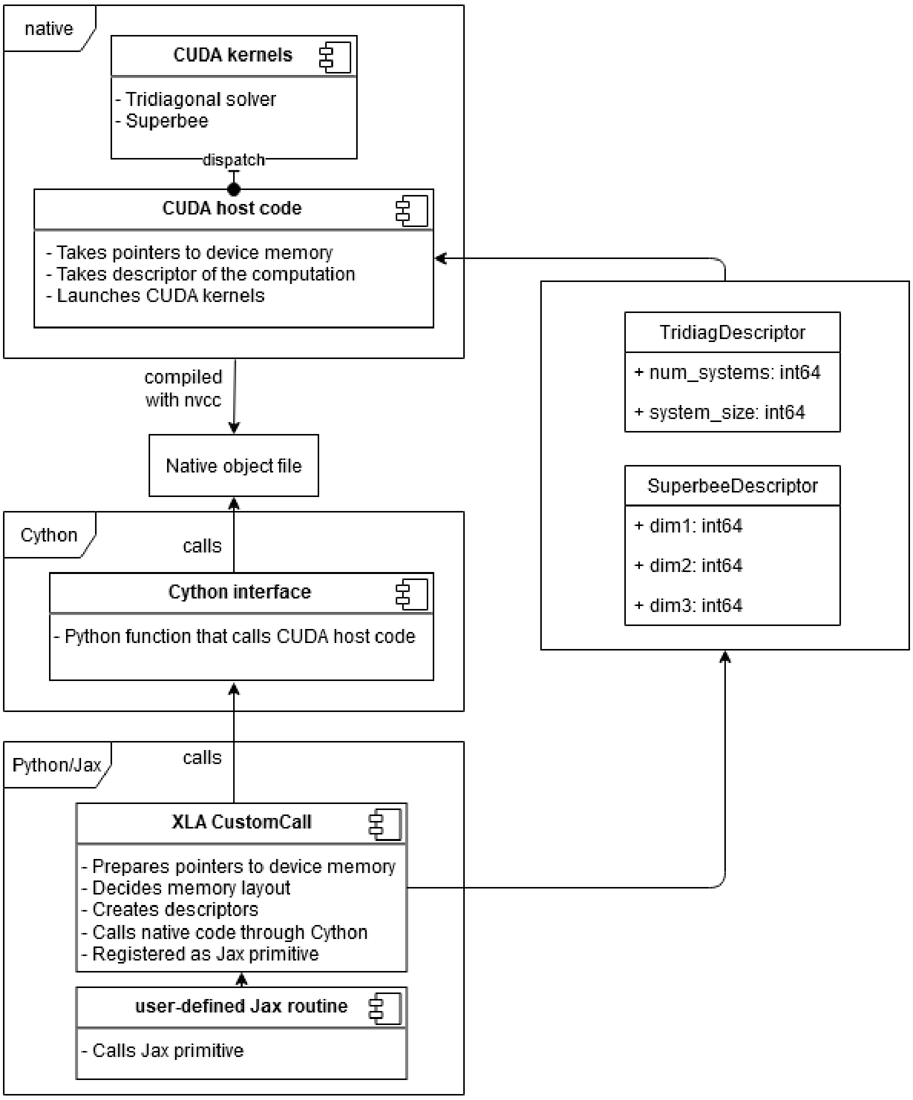


Figure 6: Interaction between Jax, Cython and CUDA.

Bibliography

[1] D. Häfner, R. L. Jacobsen, C. Eden, M. R. B. Kristensen, M. Jochum, R. Nuterman, and B. Vinter, "Veros v0.1 – a fast and versatile ocean simulator in pure python," *Geoscientific Model Development*, vol. 11, no. 8, pp. 3299–3312, 2018.

[2] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, "JAX: composable transformations of Python+NumPy programs," 2018.

[3] T. Henriksen, N. G. W. Serup, M. Elsman, F. Henglein, and C. E. Oancea, "Futhark: Purely functional gpu-programming with nested parallelism and in-place array updates," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, (New York, NY, USA), pp. 556–571, ACM, 2017.

[4] R. Okuta, Y. Unno, D. Nishino, S. Hido, and C. Loomis, "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.

[5] C. Andreetta, V. Bégot, J. Berthold, M. Elsman, F. Henglein, T. Henriksen, M.-B. Nordfang, and C. E. Oancea, "Finpar: A parallel financial benchmark," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 2, pp. 1–27, 2016.

[6] T. Henriksen, F. Thorøe, M. Elsman, and C. Oancea, "Incremental flattening for nested data parallelism," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, (New York, NY, USA), pp. 53–67, ACM, 2019.

[7] B. Hagedorn, L. Stoltzfus, M. Steuwer, S. Gorlatch, and C. Dubach, "High performance stencil code generation with lift," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pp. 100–112, 2018.

[8] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pp. 265–283, 2016.