OVERLAPPING DATA TRANSFER AND COMPUTATION ON GPU

Florian Stühlinger, Hannah Thun, Philipp Stark, Raphael Graf

Institute for Atmospheric and Climate Science
ETH Zurich, Switzerland

ABSTRACT

The performance of GPU-accelerated applications, particularly in scientific domains like weather and climate modelling, is often limited by the latency of data transfers between the host (CPU) and the device (GPU). This project of the course *High Performance Computing for Weather and Climate* (HPC4WC) investigates the effectiveness of overlapping data transfer with computation using CUDA streams to mitigate this bottleneck. We apply this technique to two distinct problems: the computationally expensive arccosine function and the 2D stencil computation from the course, representing diffusion solvers in atmospheric models. We implemented these problems in both low-level C++/ CUDA and the high-level domain-specific language GT4Py to analyse performance gains across various problem sizes and computational workloads. Our main result shows that overlapping is highly effective, but its benefit is contingent on a sufficiently high ratio of computation to data transfer. For the stencil computation, we achieve a speedup of up to 1.88x, primarily by improving GPU hardware utilization through concurrent kernel execution, while the arccosine benchmark reaches a speedup of 2.0x with increased computational intensity, effectively hiding memory transfer latency.

1. INTRODUCTION

In recent years, the field of high performance computing has been developing in a new direction. Since single-core performance is starting to meet its physical limits while problem sizes keep increasing, parallel processing, and specialized hardware, such as GPUs, are playing an increasingly vital role. Weather and climate models are no exception to this trend. A crucial measure for increasing forecast accuracy is using finer grids, leading to a growing computational demand. Therefore, efforts are made to incorporate GPUs in these models. While GPUs are able to perform certain operations faster than CPUs, it takes a considerable amount of time to move data from the CPUs memory over to the GPUs memory, which can quickly become the bottleneck of a program. One way to avoid this is trying to hide the data transfer behind the computation by splitting the workload into slices and overlapping the transfer and computation of the different slices. We illustrate this method by applying it to two model problems. The first involves arccosine evaluations and the second a stencil computation like they are found in most numerical weather and climate models.

Our programs are implemented once in C++/ CUDA and once in Python using GT4Py [1]. GT4Py is a domain-specific language for stencil computations. Here, it serves as a reference for the speed of the C++/CUDA implementation. We investigate whether overlapping can increase performance and, if so, how large the workload must be, in comparison to the amount of transferred memory. We also want to see if we can get the CUDA implementations to run faster than the GT4Py versions.

2. BACKGROUND

The main tools in CUDA that enable the goal of overlapping are asynchronous memory copies and streams. Asynchronous memory copies are asynchronous with respect to the host, which means that they can return before the copy is complete. Streams are essentially pipelines for execution on the GPU. Operations such as memory copies or kernels can be assigned to a stream. The operations in a stream are then executed in issue order on the GPU. The data can thus be divided into parts and each part can be processed by a separate stream. Each stream issues a memory copy from the host to the device, a kernel computation, and a memory copy from the device back to the host. While inside of a stream the operations are executed in sequence, operations from different streams can now run concurrently on the GPU.

Arccosine Computation. For the overlapping of memory transfer and computation to be beneficial, the computational workload needs to be sufficiently large. If memory transfer takes much more time than computation on the GPU, the compute engine will wait idle most of the time, and the period in which the compute engine and copy engine are running in parallel is very short. To get a large and adjustable workload, we chose to write a program that performs multiple evaluations of arccosine on a vector of data because the arccosine operation has a relatively high computational intensity.

However, the arccosine function maps values as follows:

$$\arccos : [-1, 1] \to [0, \pi] \tag{1}$$

Since the co-domain is not a subset of the domain, a mapping back to the domain is necessary in order to be able to perform the same operation repeatedly. To this end, we used the following simple map:

$$\phi : [0, \pi] \to [-1, 1], \quad x \mapsto \frac{2x}{\pi} - 1 \tag{2}$$

Leading to a scaled arccos $\psi := \phi \circ \arccos$:

$$\psi : [-1, 1] \to [-1, 1], \quad x \mapsto 2\frac{\arccos(x)}{\pi} - 1 \tag{3}$$

Stencil computation. In weather and climate models, one of the most important type of computations is the stencil computation. A stencil computation is an algorithmic pattern where the value of a point on a grid is updated based on its neighbouring grid points. This is a key operation in these models, since they often rely on finite-difference methods to solve partial differential equations (PDEs) that describe the behaviour of the atmosphere.

For our second benchmark, we implemented a 2D stencil computation that solves a 4th-order diffusion equation. The governing equation for this operation is:

$$\frac{\partial \phi}{\partial t} = -\alpha_4 \Delta_h (\Delta_h \phi) \tag{4}$$

where ϕ describes the diffused quantity, α_4 is the diffusion coefficient and Δ_h is the horizontal Laplacian operator.

This equation is discretized using a finite-difference approach on a 2D grid. The implementation involves applying the Laplacian operator twice, leading to a stencil which uses second order neighbour values. To handle the boundaries of the computational domain, a ”halo” of points is used. These halo points store data from periodic neighbours, which is necessary to correctly compute the stencil at the boundaries.

3. IMPLEMENTATION

For the arccosine benchmark, we implemented our solution in CUDA directly with stream support to focus on overlapping computation and data transfer. In contrast, the stencil benchmark's CUDA implementation followed a two-step process: we first established a correct baseline, which we then extended with streams.

Arccosine CUDA Implementation. The data used for the arccosine computations is created at the beginning of the program by filling an array with samples from a uniform random distribution. This array is split among the given number of CUDA streams. Each stream is issued an asynchronous memory copy of the data to the device, an execution of a set of arccosine computation kernels, and an asynchronous memory copy of the result back to the host. Every execution of the kernel applies a given number of arccosine and intermediate mappings to a single array entry. The number of threads per block used for the kernel execution is set to 256. The number of blocks is adjusted to the size of the array such that there are always enough threads in total to process all entries in the array. Since each thread acts on only one value and none of the values are shared among different threads, we do not use the shared memory of the blocks. Therefore, the number of blocks and threads we are using should not make a significant difference in performance. This was consistent with our observations from experimenting with different numbers of threads per block.

Stencil2d extension for CUDA with streams. As a basis for the second benchmark, the `stencil2d` C++ code from the course was used. It was adapted for CUDA by extending the *Storage3D* class to support GPU data transfer. The computation was ported to the GPU through kernels implementing the *updateHalo* and the *diffusionStep* functions. Streams were introduced by splitting the workload along the z-axis, with each stream responsible for the diffusion step on a distinct slice of the data. Since the horizontal diffusion operator only depends on the z-level being evaluated, the diffusion calculations for different slices can be executed concurrently on the GPU. However, the data transfer takes up only a short time of the whole application which suggests only small improvements by overlapping data transfer and computation. We therefore decided to just distribute the computation over the different streams since this may lead to a different distribution of the workload across the computational units of the GPU.

GT4Py Implementation. We wanted to compare the runtimes of our CUDA codes with GT4Py implementations and see if our CUDA programs can reach the performance of a program in a domain specific language like GT4Py. GT4Py provides the `field_operator` decorator for function definitions. It is meant to mark stencil kernels that GT4Py can run on a n-dimensional field in a more optimal way than stacked for loops in Python. During code generation, GT4Py traverses through the code and tries to apply optimizations but for too many nested arccosine calls Pythons maximum recursion depth limit was reached. Therefore we only used dedicated field_operators up to $2^6 = 64$ arccosine calls and implemented higher call counts by simple for loops. For a GT4Py implementation of the diffusion kernel we took the last version from the solutions in the course repository[2] as a basis and added a timing function to it.

4. EXPERIMENTAL RESULTS

To evaluate the effectiveness of our stream-based implementations, we conducted a series of benchmarks for both the arccosine and the stencil computations. The primary objective was to quantify the speedup achieved by using multiple streams compared to a single-stream baseline. Our methodology consisted of measuring the runtime for each implementation across a range of input data sizes and for a varying number of streams. This allowed us to analyse the performance gains resulting from the overlapping of computation and data transfer.

4.1. Experimental setup

Hardware. All experiments were carried out on the grace hopper nodes of the santis cluster in the ALPS research infrastructure [3][4][5].

Input. For both benchmarks, a similar approach was used. Both benchmarks were tested using a warm-up run and then taking the average time over ten consecutive runs over the same input.

- **Arccos**: The arccosine program was tested for different parameters. All of them varied in size in powers of two, except for the array size, where every second power of two was skipped. The number of arccosine evaluations was varied between 1 and 512, the size of the data array between 8 and 2^{29}, and the number of streams between 1 and 512. The array size was always kept larger than the number of streams, but besides that, all combinations of parameters were part of the measurements.

- **Multiple Arccos Calls**: For benchmarking multiple arccosines calls, the same array size and stream combinations were used, with the number of arccosine calls varied from 1 to 512.

- **Stencil**: The stencil benchmarks were performed with two different approaches. The first one tested the implementation for a constant number of 64 z-levels with varying x- and y-levels, number of time steps and number of streams. The second tested the implementation for a constant number of x- and y-levels and varied the number of time steps, z-levels and number of streams. In both approaches the number of streams were set between 1 and 256 and the number of iterations between 32 and 1024. The number of x- and y-levels in the first approach was between 8 and 4096, leveraging the whole size of the GPU RAM. In the second approach, the z-levels were set between 8 and 65536, again leveraging the whole size of the GPU RAM.

4.2. Results

Arccos. Fig. 1 shows the speedup as a function of the number of streams for array sizes ranging from 2^{19} to 2^{29} elements. In our benchmark run, we also tested smaller array sizes, but only from 2^{19} elements on using streams is beneficial, as the stream overhead is too large compared to the relatively small runtime of a single computation of arcco-

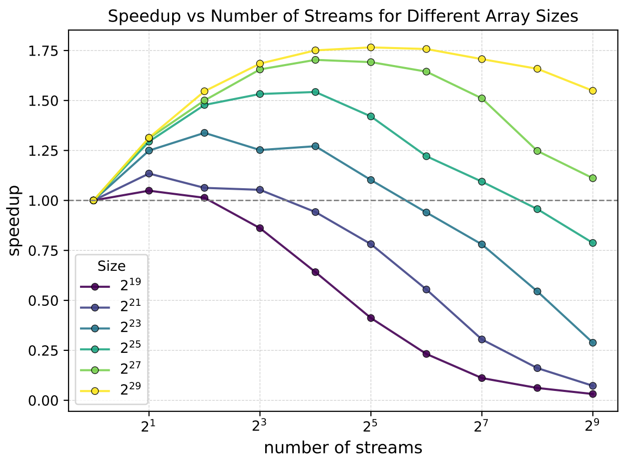

Fig. 1. Speedup vs number of streams plot for different array sizes for a single arccosine call. This plot illustrates that an increase in the number of streams used benefits the speed of the arccosine calculation partly, depending on the system size and the number of streams used. We get a maximal speedup of 1.766 for an array size of 2^{29} and 32 streams.

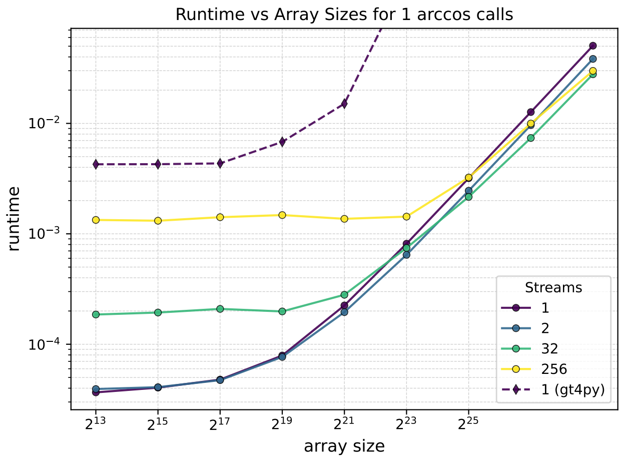

Fig. 2. Runtime vs array size for different numbers of streams for a single arccosine call. For lower array sizes up to 2^{19} the use of multiple streams do not pay off. The GT4Py version is two orders of magnitude slower than the CUDA code.

sine on a GPU. We can also see that the optimal number of streams depends on the array size, and consequently on the chunk size assigned to each stream. Our benchmarking indicates that using many streams is not necessarily beneficial when the chunk size gets too small. Furthermore, we observe that the maximum speedup increases with array size, reaching the highest speedup of 1.766 for the largest tested array size of 2^{29} elements using 32 streams.

In Fig. 2, we compare the actual runtime across different array sizes for four different stream counts: 1, 2, 32 and 256. Our main finding is that using streams creates significant stream management overhead which clearly dominates the overall runtime for most array sizes in the case of a single arccosine call. This overhead increases with the number of streams, making the usage of streams only beneficial for very large array sizes. For example, our benchmarking shows that using 256 streams results in a speedup over a single stream only for extremely large array sizes such as 2^{27} and 2^{29}. As before, we recognize that the optimal number of streams depends on the problem size. We conclude that, due to the relatively low computational cost of a single arccosine evaluation, its computation can only be efficiently overlapped with memory transfer between CPU and GPU for sufficiently large arrays. We also see that GT4Py performs poorly which could have been expected given that GT4Py is meant to perform on stencil problems. To further evaluate the benefits of using streams, we increased the computational workload by testing multiple arccosine calls.

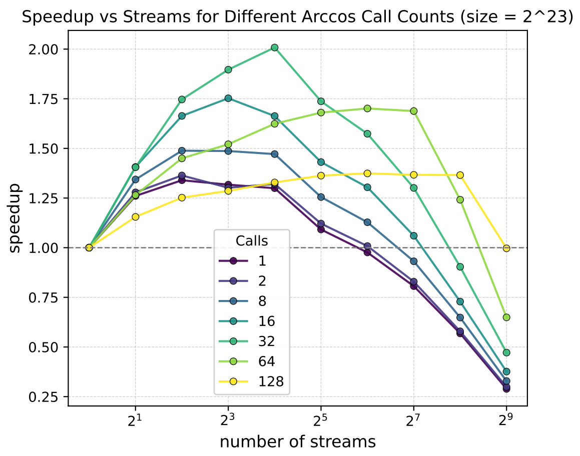

Fig. 3. Speedup vs number of streams plot for different numbers of arccosine calls, array size fixed to 2^{23} = 8,388,608. We see here that the speedup first increases with the number of streams, but if too many streams are used, there is a slowdown. The maximal achieved speedup is 2.0 with 32 function calls and 16 streams.

Multiple Arccos Calls. For the performance analysis, we examine the results for a fixed array size of 2^{23} elements. Fig. 3 displays the speedup as a function of the number of streams for different numbers of arccosine function calls which are illustrated by the coloured graphs. The plot again indicates that speedup strongly depends on the the computational intensity, here influenced by the number of arcossine calls. For 32 function calls, the speedup reaches a maximum value of 2.0, corresponding to the optimal speedup that can be achieved by overlapping computation and data transfer. In contrast, the strongest speed up for a single arcosine evaluation is bounded to 1.34. For more than 32 function calls, the speedup decreases as the GPU's resources are fully used. Since the optimal number of streams also depends on the array size, our finding should be regarded as a case study rather than a general conclusion.

Stencil. Due to only little communication between GPU and CPU and a rather big computational load, we did not expect any speedup by just splitting the stencil computation itself onto different streams. In fig. 4, we see speeds achieving up to 1.88x of the base implementation. This number was seen for nz = 8192 and 512 timesteps.

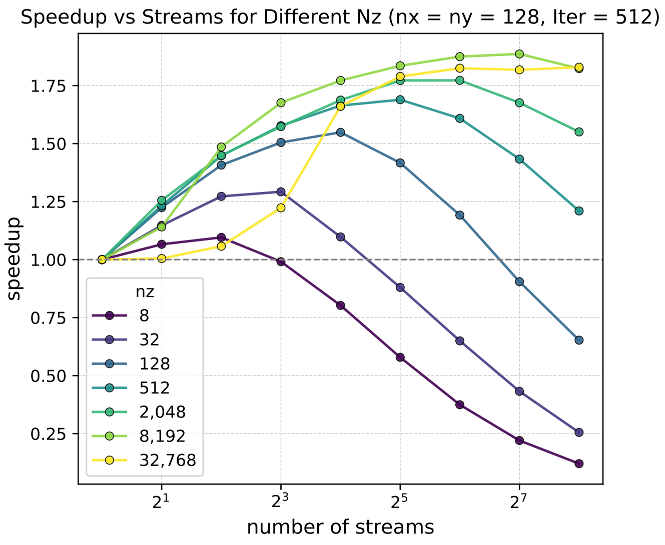

Fig. 4. Speedup vs number of streams plot for different numbers of nz and constant nx, ny and iter for the stencil computation. Two different dependencies are visible, first, that for small systems (i.e. small nz), only low stream numbers even have a gain in runtime, the larger then the system grows, the higher the overall speedup. The second visible dependency is the maximal speedup, that is at a larger number of streams for larger systems.

Our benchmark results indicate that the effectiveness of using multiple streams is highly dependent on the problem size. For smaller systems, the overhead associated with managing multiple streams outweighs the benefits of concurrent execution, leading to only small improvements.

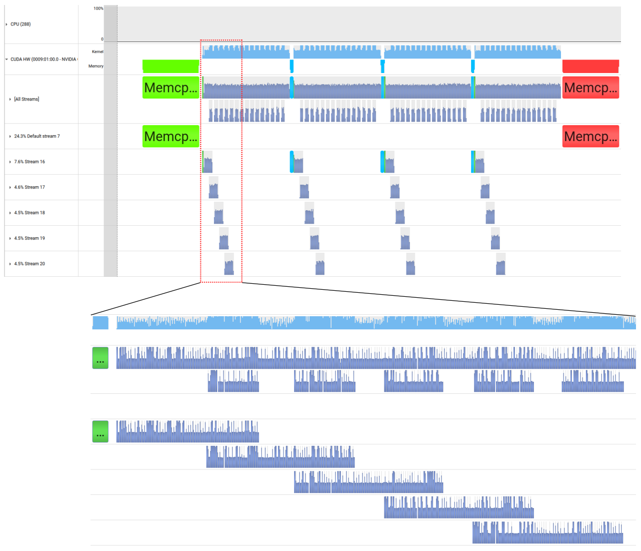

Fig. 5. Nvidia nsight systems analysis for the stencil code over 4 iterations and 16 streams. In the top panel, we see the whole calculation with both memcpy host-device communications at the beginning and at the end of the calculations. In between, we see the different iterations of the algorithm and its distribution over different streams. In the bottom panel, we see a zoomed in view of the distribution over the different streams and with the overlapping calculations, which in total give us the achieved speedup of up to 1.88x.

However, as the system size increases, the computational workload becomes substantial enough to effectively hide the overhead of stream creation. Consequently, for these larger systems, employing a greater number of streams yields significant speedups. Furthermore, we observed a clear trend where the optimal number of streams, which provides the maximum performance gain, shifts to higher values as the overall problem size grows. This performance gain is attributed to a more efficient utilization of the GPU. We split the system in a lot of blocks and threads, enabling the GPU's scheduler to pipeline the execution of different kernels. This strategy allows the computation of one data slice to begin on available hardware resources while another slice's computation is still in progress. This overlap of operations reduces latencies in the computation pipeline, leading to a higher overall hardware occupancy. We can see this in fig. 5, where the streams get executed in order with a slight overlap, leading to our observed speedup.

Comparisons to `Gt4Py`. Fig. 6 shows a runtime comparison of the CUDA and the GT4Py implementations for a fixed problem size, different numbers of iterations and variable numbers of used streams in the CUDA version. It shows that with increasing number of iterations GT4Py gets relatively faster in comparison to the CUDA version

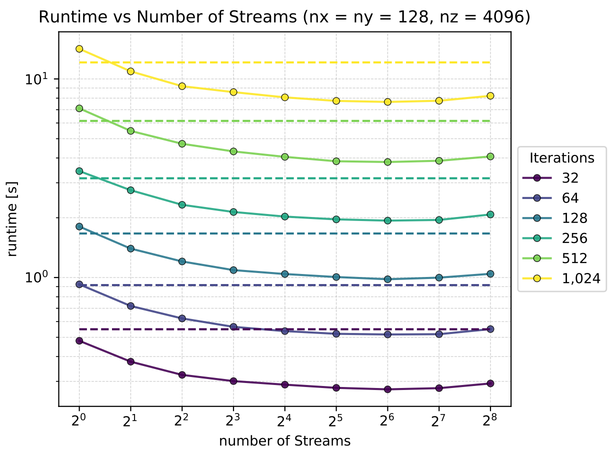

Fig. 6. The runtime for the diffusion stencil written in CUDA for variable number of used streams and iterations in solid lines and written in GT4Py where there is no option to use multiple streams in dashed lines. The used domain size was $nx = ny = 128$ and $nz = 8192$.

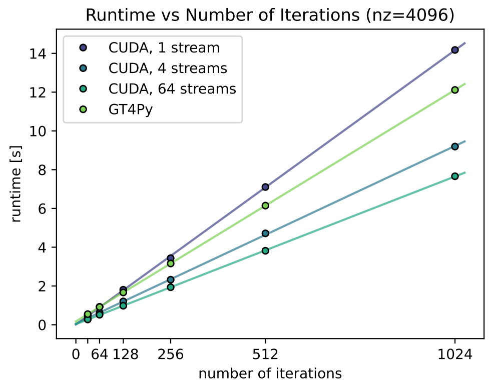

Fig. 7. Runtime for increasing number of diffusion stencil iterations with fixed domain size ($nx = ny = 128$, $nz = 4096$). The lines are linear fits to the data. The CUDA version with 1 stream is slower than the GT4Py version but faster with higher stream counts.

with only 1 stream but is outperformed by CUDA for higher numbers of streams and for this problem size ($nx = ny = 128, nz = 4096$).

Plotting the same data with runtime against number of iterations (see Fig. 7), we observe an almost linear dependence between number of diffusion iterations and runtime as expected. But the same plot for $nz = 8192$ (see Fig. 8) shows a different picture for the GT4Py runtimes. There must be some additional optimizations being applied by GT4Py for $nz = 8192$ in contrast to $nz = 4096$ beause there is a overhead of about four seconds (the y-axis inter-

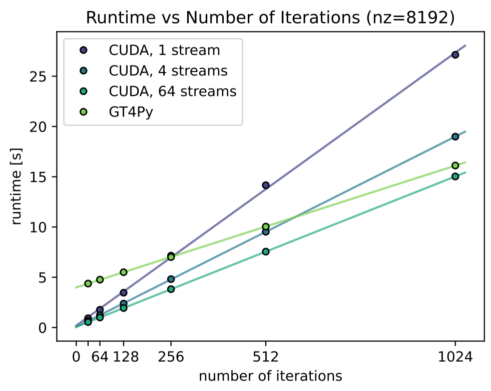

Fig. 8. Runtime for increasing number of diffusion stencil iterations with fixed domain size ($nx = ny = 128$, $nz = 8192$). The lines are linear fits to the data. For this configuration GT4Py has 4s of additional overhead and as a result is only faster than the 1 stream CUDA version for 512 and 1024 iterations but for 1024 iterations it is also faster than the 4 streams version.

cept of the linear fit to the GT4Py data points) but almost the same slope (0.01166 s/iter for $nz = 4096$ vs. 0.01183 s/iter for $nz = 8192$) even though we have double the z-levels and therefore double the compute being done.

5. CONCLUSIONS

In this paper, we investigated the use of CUDA streams to hide data transfer latency and improve performance in GPU-accelerated applications. We successfully implemented and benchmarked this technique for both a computationally intensive arccosine evaluation and the 2D stencil computation from the HPC4WC course.

Our results confirm that overlapping data transfer and computation is a powerful optimization strategy, but its effectiveness is highly dependent on the problem's characteristics. For the arccosine benchmark, a significant speedup was only achieved when the computational workload was sufficiently large to mask the memory transfer time, reaching an optimal speedup of 2.0 with 32 function calls. For the stencil computation, where data transfer is minimal compared to the computation, we still observed a substantial speedup of up to 1.88x. This gain was attributed not to hiding data transfer, but to more efficient hardware utilization, as the GPU scheduler could pipeline the execution of kernels from different streams. Furthermore, our hand-optimized C++/CUDA implementations generally outperformed the high-level GT4Py versions, demonstrating the performance benefits of low-level control.

This work highlights that while CUDA streams can effectively address the data transfer bottleneck, their application requires careful consideration of the compute-to-communication ratio and problem size. The observed performance gains, both from hiding latency and improving hardware occupancy, affirm that stream-based concurrency is an essential technique for developing high-performance scientific code on modern GPU architectures.

6. REFERENCES

[1] Enrique G. Paredes, Linus Groner, Stefano Ubbiali, Hannes Vogt, Alberto Madonna, Kean Mariotti, Felipe Cruz, Lucas Benedicic, Mauro Bianco, Joost VandeVondele, and Thomas C. Schulthess, "Gt4py: High performance stencils for weather and climate applications using python," 2023.

[2] O. Fuhrer, "Hpc4wc: High performance computing for weather and climate," `https://github.com/ofuhrer/HPC4WC/tree/main`.

[3] CSCS, "Alps research infrastructure," `https://www.cscs.ch/computers/alps`.

[4] CSCS, "santis cluster," `https://docs.cscs.ch/clusters/santis/`.

[5] CSCS, "Nvidia gh200 gpu nodes," `https://docs.cscs.ch/alps/hardware/#nvidia-gh200-gpu-nodes`.