GPU acceleration of DALES with OpenACC
Research proposal

Caspar Jungbacker

June 7, 2023

1 Background

The atmospheric boundary layer (ABL) is a highly turbulent flow consisting of a wide spectrum of turbulent eddies (whirling motions). Explicitly solving the Navier-Stokes equations for all scales of these eddies is called direct numerical simulation (DNS). For DNS, the computational mesh has to be fine enough to represent even the smallest eddies, making it very computationally demanding (Moeng & Sullivan, 2015). Instead of resolving all scales, one could choose to only resolve the largest, most energetic scales explicitly and model the smaller scales. This approach is known as large-eddy simulation (LES). The Dutch Atmospheric Large-Eddy Simulation (DALES) model, developed by Heus et al. (2010), is an LES model that is used for fundamental research of atmospheric processes.

Although LES is computationally less expensive than DNS for a given domain size, it remains resource-intensive. The majority of the computing time arises from loops that perform some calculation on all grid points, such as calculating derivatives. Typically, these calculations are independent of one another, meaning that they can be carried out in parallel. Graphics Processing Units (GPUs) are particularly well suited for parallel computing; unlike Central Processing Units (CPUs), which consist of up to tens of processing cores, a modern GPU can have thousands of processing cores performing calculations in parallel (Elster & Haugdahl, 2022), offering massive potential speed up for applications.

Moving calculations from CPUs to GPUs is a nontrivial task. Programmers need to adapt their software to effectively utilize a GPU. GPUs can be programmed using standard programming languages like C/C++, Fortran, Python, and others. To this end, specialized programming models are available that allow the programmer to manage a GPU. Examples of such programming models are: Compute Unified Device Architecture (CUDA) from NVIDIA, Open Computing Language (OpenCL), and Heterogeneous-Compute Interface for Portability from AMD. These models require rewriting the original CPU code into code that can run on GPUs, so-called *kernels* (Owens et al., 2008). Alternatively, one can choose to offload calculations using a programming model based on *compiler directives*. Compiler directives are instructions to the compiler to perform some action on a piece of code. These directives look like comments. Instead of rewriting code, the programmer places compiler directives over parallelizable loops to offload them to a GPU. The compiler will then generate the kernels by itself. An example of a directive-based programming model is Open Accelerators (OpenACC). Working with OpenACC offers multiple benefits over traditional, lower-level programming models like CUDA (Herdman et al., 2012):

- Productivity: OpenACC requires minimal addition of lines of code to an existing codebase. This makes it possible for a programmer to accelerate large sections of a program in a relatively short amount of time.
- Single source code: since OpenACC directives are essentially comments within the source code, only one version of the source code needs to be maintained. This approach is less error-prone compared to maintaining separate CPU and GPU versions, as required by CUDA and OpenCL. When a CPU version of the code is desired, the code can be recompiled for CPUs, and the OpenACC directives will be disregarded.
- Performance: CUDA and OpenCL applications require more manual optimizations of algorithms than OpenACC. For this reason, it is not uncommon that an OpenACC implementation can outperform a CUDA or OpenCL application, especially when little time has been spent on optimizing the application. However, since CUDA and OpenCL allow more fine-grained control over the GPU hardware, algorithms can be optimized much further than possible with OpenACC.

2 Related work

Niemeyer and Sung (2014) have examined the status of GPU computing in the field of computational fluid dynamics (CFD). To demonstrate the potential benefits of using GPUs for CFD, two case studies were performed: a 2D Laplace equation, resembling the Poisson equation that is often found in CFD codes, and a lid-driven cavity flow. Four implementations were tested: single-core CPU, multi-core CPU with OpenMP, GPU with CUDA, and GPU with OpenACC. For mesh sizes up to 512^2, the wall-clock time for the GPU implementations exceeded that of the multi-core CPU implementation. While the authors do not explicitly explore the possible causes of this behavior, it can be argued that the increase in wall-clock time is due to data transfer between the CPU and GPU. For larger mesh sizes, the GPU implementations outperformed the CPU implementations. Specifically, for the Laplace equation, the GPU solver showed a speedup of about 4.6, while for the lid-driven cavity flow, the speedup was about 2.8. Remarkably, Niemeyer and Sung (2014) showed that as the mesh size increases, the wall-clock time of the OpenACC implementation converged to that of the CUDA implementation.

DALES itself has been ported to GPUs before by Schalkwijk, Griffith, Post, and Jonker (2012). To this end, the original Fortran code of DALES was translated to C++, and calculations were moved to the GPU using CUDA, resulting in the GPU-resident Atmospheric Large-Eddy Simulation (GALES) model. Schalkwijk et al. (2012) found that GALES was able to reduce the wall-clock time per time step by a factor of 2 compared to DALES, although it should be noted that GALES uses single precision floating point numbers in most parts of the simulation, while DALES uses double precision. This is an important distinction to make, as GPUs are particularly well-optimized for single-precision floating point arithmetics. Since then, the company Whiffle has adopted GALES and further developed it into the GPU-Resident Atmospheric Simulation Platform (GRASP). GRASP is often used for very accurate simulations of windfarms (Verzijlbergh, 2021).

Costa (2018) has developed a tool for DNS of turbulent flows, called CaNS (Canonical Navier-Stokes solver). The dynamical core of CaNS is very similar to that of DALES: both use finite-difference discretization on a structured, staggered grid, an FFT-based solver for the pressure, and third-order Runge Kutta time integration. Unlike DALES, CaNS does not include a subgrid-scale model. Parallelization of CaNS is achieved by domain decomposition into pencils along two directions, with further shared-memory parallelization via OpenMP. CaNS was later adapted for GPUs using CUDA Fortran (Costa, Phillips, Brandt, & Fatica, 2021). CUDA Fortran provides a Fortran interface to the CUDA programming model and includes CUF Kernels. CUF Kernels are compiler directives that can be placed above loops to tell the compiler that the loop can be executed on the GPU. NVIDIA's cuFFT library was used to perform the FFT calculations on GPUs. Performance analysis was done on two systems: an NVIDIA DGX Station, a system comparable in size to a modern desktop PC and containing 4 Tesla V100 32 Gb GPUs, and an NVIDIA DGX-2, a standard 19-inch server chassis containing 16 Tesla V100 32 Gb GPUs. Costa et al. (2021) found that for the same problem size, one would need about 6100 to 11200 CPU cores to match the wall-clock time per time step of the 16 Tesla V100s in the NVIDIA DGX-2. This is still a conservative estimate, as linear scaling was assumed for the CPU code, whereas in reality, performance often scales sublinearly for a given problem size due to overhead introduced by communication between processes. For reference, the first phase of the new TU Delft DelftBlue supercomputer has around 11,000 CPU cores, indicating that GPUs can offer a significant reduction in energy consumption.

3 Objectives

The reviewed literature shows that GPUs can speed up CFD simulations considerably. This allows researchers to save time, perform more experiments and use finer or larger domains. The main objective of this thesis is to accelerate DALES by leveraging GPUs through a directive-based programming model so that DALES users experience these benefits as well. To successfully reach this goal, the following questions have to be answered:

1. **What are the main computational bottlenecks in DALES?**

   Identifying computational bottlenecks will reveal which parts of DALES would benefit the most from GPU acceleration.

2. **How can the components of DALES that rely on external modules be offloaded to GPUs?**

   For example, the subroutine that solves the Poisson equation relies on the FFTW module. In order to get maximum performance, these modules should be able to run on GPUs as well. If this is not possible, replacement modules should be sought.

3. **How can multiple GPUs be used?**

   When more than one GPU is used, the need for communication between GPUs arises. The current communication back end of DALES needs to be tested with GPUs to find bottlenecks. Other back ends can be explored to improve performance.

4. **How does the performance of the GPU implementation of DALES compare to the CPU implementation?**

   Performance should be tested for a variety of cases, grid configurations, and the number of GPUs used.

4 Methodology

4.1 Profiling

Before attempting to accelerate a model like DALES, the most time-consuming parts of the code should be identified. This process is called profiling, for which various tools exist. NVIDIA NSight Systems is such a tool, and is particularly well-suited for GPU applications (NVIDIA, n.d.). NSys automatically tracks CPU-GPU interactions, including data transfer and GPU utilization, but does not track CPU utilization. To track CPU activity, a custom timer module based on the NVIDIA Tools Extension (NVTX) library is used. With NVTX, CPU code can be tracked by NSys as well. The necessary code for NVTX has been added to DALES already. NSys comes with a GUI to visually inspect the profile (Figure 1), which will be used to track the performance of DALES as its components are offloaded to the GPU.

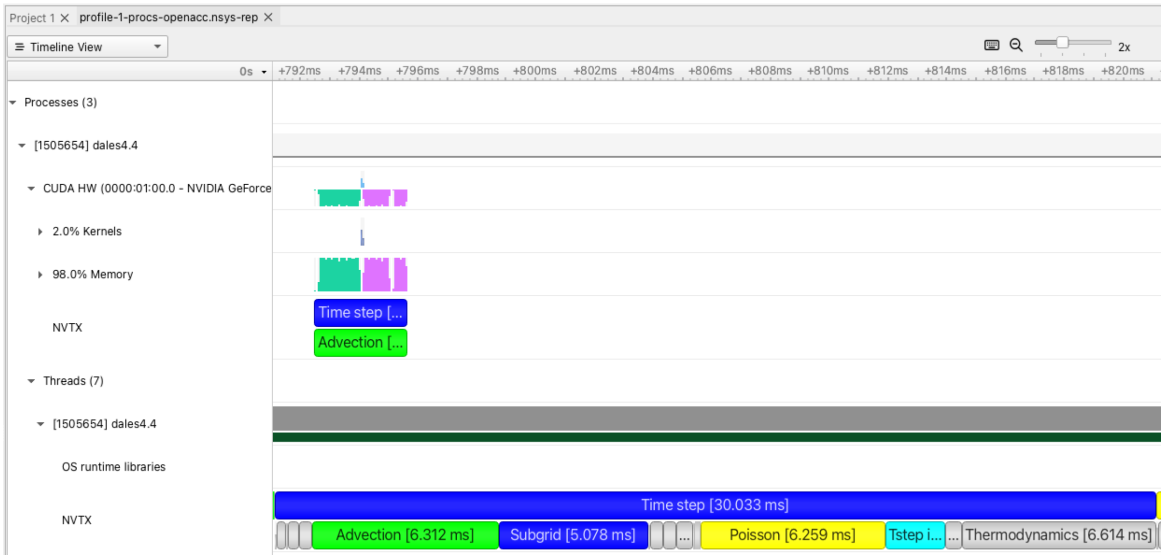

Figure 1: A screenshot of the NSys GUI. The main subroutine calls in DALES' time steps have been annotated with NVTX markers and show up at the bottom of the image, with their execution time between brackets. To demonstrate the GPU tracking capabilities of NSys, one advection kernel has been offloaded to the GPU (`advecu_2nd`), which shows up under "CUDA HW". It can be seen that NSys automatically separates kernel execution time from time spent on memory operations, allowing for easy identification of performance bottlenecks.

4.2 OpenACC

According to the literature, OpenACC is a straightforward programming model to implement into an existing code base, while still offering good performance, especially when data transfers are properly optimized. This makes it a good choice for accelerating DALES. In Figure 1 a profile of DALES can be found. It can be seen that during a single time step, most time is spent on advection, sub-grid, Poisson, and thermodynamics modules. These modules involve parallelizable loops, making them well-suited for OpenACC.

The first step in getting DALES running on GPUs is to place OpenACC directives over all parallelizable loops. An example of a loop with an OpenACC directive can be found in Listing 1. Once a loop or a group of loops has been offloaded, the data transfers can be optimized. As mentioned previously, memory transfers between CPU and GPU can contribute significantly to the wall-clock time of an application. Hence, it is important to minimize the amount of data transfers and optimize data locality (i.e., the location of data). With OpenACC, the programmer does not have to be explicit about data transfers, as the compiler can determine when to move data by itself. However, to ensure optimal data locality, it is good practice to explicitly copy data to and from the GPU. In OpenACC, this is also done via compiler directives. This approach will also be applied to DALES. Ideally, all fields will be copied to the GPU before the time loop begins, and data will only be copied back to the CPU for writing output files, with no additional data transfer in between. This may not be possible to realize for DALES, as some algorithms may not be parallelizable, meaning that these are better run on the CPU. In this case, intermediate data transfers are required.

```fortran
!$acc parallel loop collapse(3) default(present)
do k = 1, kmax
    do j = 1, jmax
        do i = 1, imax
            c(i,j,k) = a(i,j,k) + b(i,j,k)
        end do
    end do
end do
```

Listing 1: Example of a Fortran loop decorated with an OpenACC directive. The directive `!$acc parallel loop` tells the compiler that the following loop can be executed in parallel. The `collapse(3)` clause collapses the three nested loops into one big loop, exposing more parallelism, and the `default(present)` clause tells the compiler that the arrays `a`, `b` and `c` already exist on the GPU and no further data transfer is needed.

4.3 Solving the Poisson equation

In DALES, a solution for the pressure π is obtained by solving the following Poisson equation:

$$\frac{\partial^2 \pi}{\partial x_i^2} = \frac{\partial}{\partial x_i}\left(-\frac{\partial \overline{u}_i \overline{u}_j}{\partial x_j} + \frac{g}{\theta_0}\overline{\theta}_v \delta_{i3} + \mathcal{F}_i - \frac{\partial \tau_{ij}}{\partial x_j}\right) \tag{1}$$

DALES provides two methods to solve this equation: using Fast Fourier Transforms (FFTs), or using iterative solvers. Among these, the FFT-based solver is the fastest. DALES has the ability to use the highly optimized Fastest Fourier Transform in the West (FFTW) library (Frigo & Johnson, 1997). FFTW is not capable of running on GPUs, however. NVIDIA provides an FFT library in their HPC SDK called cuFFT, which does run on GPUs and can be used instead of FFTW. cuFFT has a similar interface as FFTW, making the conversion straightforward.

4.4 Multiple GPUs

Currently, DALES is parallelized with MPI by decomposing the domain into slabs or pencils. OpenACC introduces further parallelization by distributing loop iterations among GPU threads. This means that MPI and OpenACC can be combined to run DALES on more than one GPU, for example by decomposing the domain into a number of slabs or pencils that equals the number of available GPUs, with each MPI process bound to its own GPU. An important thing to note is that the communication overhead increases with this approach, as halo transfers now also include copying data from GPU to CPU in addition to communication between CPUs. To mitigate this performance penalty to some degree, an MPI implementation that is aware of GPU memory can be used, which combines and optimizes the data transfer and MPI call.

The FFT-based Poisson equation solver relies heavily on communication between processes (and therefore, GPUs). This is because it requires transposing the data in between FFT calculations, which in turn requires a redistribution of the data. Currently, the transpose operations are implemented manually in DALES, using MPI `Alltoall` operations. When FFTW is replaced by cuFFT, these transpose operations must also be adopted for the GPU. Alternatively, the multi-process version of cuFFT, cuFFTMp, can be used. This library enables multi-process, multi-GPU FFTs, without the need to transpose the data and perform `MPI_Alltoall` manually. In fact, inter-GPU communication is optimized using another communication library called NVSHMEM (NVIDIA Developer, n.d.), which offers superior performance compared to regular MPI communications. cuFFTMp can bind to an existing MPI communicator, and handle data transposition internally. Yet another option is to use the cuDecomp library as described by Romero, Costa, and Fatica (2022). cuDecomp does not offer any FFT functionality but functions as an abstraction layer for the domain decomposition and communications. Given a certain number of MPI processes, cuDecomp automatically determines the optimal domain decomposition. For example, given 32 processes, a 3D domain can be decomposed into 1×32, 2×16, or 4×8 chunks, each potentially yielding different performance. Additionally, cuDecomp supports multiple communication backends (MPI, NVSHMEM, NCCL), from which it selects the best performing during run time. This can improve performance significantly, especially when GPUs are distributed over multiple nodes. Because of its promising performance benefits on current-generation supercomputers, implementing cuDecomp is an objective of this project.

4.5 Verification and performance testing

Modifying source code carries the risk of introducing bugs. While OpenACC requires minimal rewriting of source code, some modifications may be necessary for optimization purposes. Therefore, the updated source code needs to be validated against the original unmodified code to ensure that the program logic stays intact. Because of the chaotic nature of turbulence, one cannot simply perform an element-wise comparison of two arrays after a number of time steps. Instead, for some simulation case, multiple runs will be done, each with identical settings but different random seeding. From these runs, ensemble statistics can be calculated, like the mean and standard deviation of a quantity. The correctness of the modified code can be verified by comparing the value of some quantity against its mean from the ensemble; if the value is located within two standard deviations (95% confidence interval) from the ensemble mean, one can say that the modified code remains correct with a reasonable level of certainty. Multiple cases will be validated, as different cases can stress the components of DALES differently.

There are multiple metrics available to define the performance of a GPU program. One obvious metric is the speedup in the wall-clock time of using one CPU versus using one GPU for multiple grid sizes. When using more than one GPU, the scaling of the wall-clock time with the number of GPUs used is a good indication of the parallel efficiency. The scaling is measured by calculating the speedup that is gained from using more GPUs for a fixed problem size. As for the verification, this will also be done for multiple cases.

5 Planning

My planning can be found in Figure 2. The indicated time needed for the programming tasks is indicative since this will be largely determined by how many issues will be encountered along the way. Testing and validation of the code will be done iteratively, which is why it is scheduled for almost the whole duration of the thesis. Near the end of the project, when as much code is offloaded as possible, more comprehensive performance assessments will be done (also scheduled under "Testing and validation"). I will be writing the report parallel to the programming and research, I plan to devote about a third of my time to writing.

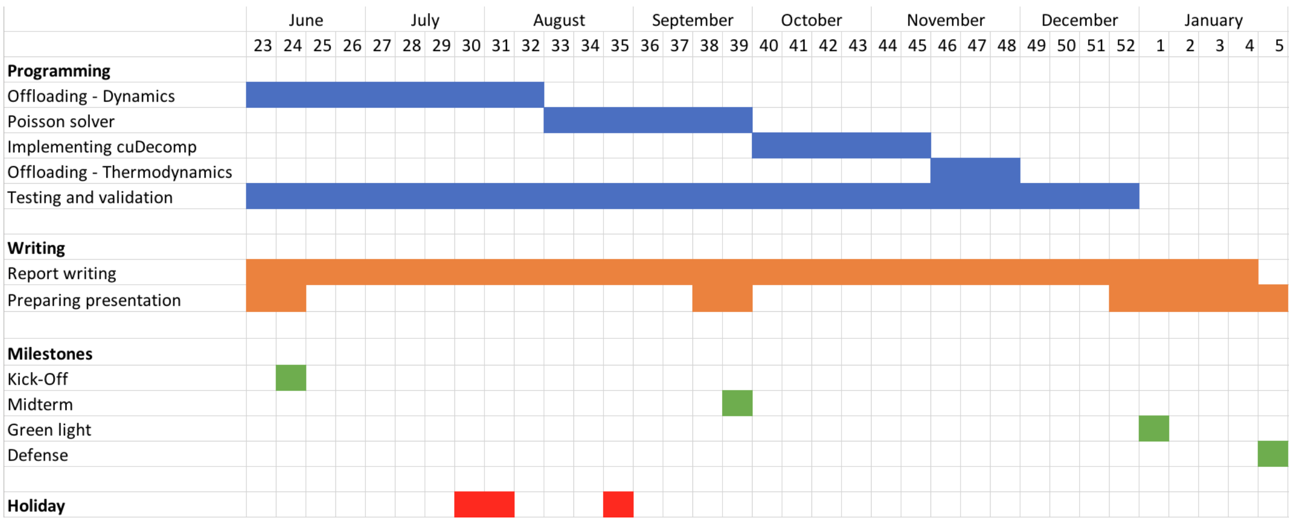

Figure 2: Weekly planning

References

Costa, P. (2018, October). A FFT-based finite-difference solver for massively-parallel direct numerical simulations of turbulent flows. *Computers & Mathematics with Applications*, *76*(8), 1853–1862. Retrieved 2023-05-25, from https://www.sciencedirect.com/science/article/pii/S089812211830405X doi: 10.1016/j.camwa.2018.07.034

Costa, P., Phillips, E., Brandt, L., & Fatica, M. (2021, January). GPU acceleration of CaNS for massively-parallel direct numerical simulations of canonical fluid flows. *Computers & Mathematics with Applications*, *81*, 502–511. Retrieved 2023-05-26, from https://linkinghub.elsevier.com/retrieve/pii/S0898122120300092 doi: 10.1016/j.camwa.2020.01.002

Elster, A. C., & Haugdahl, T. A. (2022, March). Nvidia Hopper GPU and Grace CPU Highlights. *Computing in Science & Engineering*, *24*(2), 95–100. doi: 10.1109/MCSE.2022.3163817

Frigo, M., & Johnson, S. G. (1997, September). *The fastest Fourier transform in the west* (Tech. Rep. No. MIT-LCS-TR-728). Massachusetts Institute of Technology.

Herdman, J. A., Gaudin, W. P., McIntosh-Smith, S., Boulton, M., Beckingsale, D. A., Mallinson, A. C., & Jarvis, S. A. (2012). Accelerating hydrocodes with OpenACC, OpenCL and CUDA. *Proceedings - 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, SCC 2012*, 465–471. doi: 10.1109/SC.COMPANION.2012.66

Heus, T., Van Heerwaarden, C. C., Jonker, H. J. J., Pier Siebesma, A., Axelsen, S., Van Den Dries, K., ... Vilà-Guerau De Arellano, J. (2010, September). Formulation of the Dutch Atmospheric Large-Eddy Simulation (DALES) and overview of its applications. *Geoscientific Model Development*, *3*(2), 415–444. Retrieved 2023-05-23, from https://gmd.copernicus.org/articles/3/415/2010/ doi: 10.5194/gmd-3-415-2010

Moeng, C.-H., & Sullivan, P. (2015). NUMERICAL MODELS | Large-Eddy Simulation. In *Encyclopedia of Atmospheric Sciences* (pp. 232–240). Elsevier. Retrieved 2023-05-26, from https://linkinghub.elsevier.com/retrieve/pii/B9780123822253002012 doi: 10.1016/B978-0-12-382225-3.00201-2

Niemeyer, K. E., & Sung, C.-J. (2014, February). Recent progress and challenges in exploiting graphics processors in computational fluid dynamics. *The Journal of Supercomputing*, *67*(2), 528–564. Retrieved 2023-05-29, from https://doi.org/10.1007/s11227-013-1015-7 doi: 10.1007/s11227-013-1015-7

NVIDIA. (n.d.). *NVIDIA Nsight Systems.* Retrieved 2023-05-25, from https://developer.nvidia.com/nsight-systems

NVIDIA Developer. (n.d.). *NVSHMEM.* Retrieved from https://docs.nvidia.com/nvshmem/api/index.html

Owens, J., Houston, M., Luebke, D., Green, S., Stone, J., & Phillips, J. (2008, May). GPU Computing. *Proceedings of the IEEE*, *96*(5), 879–899. Retrieved 2023-06-04, from http://ieeexplore.ieee.org/document/4490127/ doi: 10.1109/JPROC.2008.917757

Romero, J., Costa, P., & Fatica, M. (2022, July). Distributed-memory simulations of turbulent flows on modern GPU systems using an adaptive pencil decomposition library. In *Proceedings of the Platform for Advanced Scientific Computing Conference* (pp. 1–11). New York, NY, USA: Association for Computing Machinery. Retrieved 2023-06-02, from https://dl.acm.org/doi/10.1145/3539781.3539797 doi: 10.1145/3539781.3539797

Schalkwijk, J., Griffith, E. J., Post, F. H., & Jonker, H. J. J. (2012, March). High-Performance Simulations of Turbulent Clouds on a Desktop PC: Exploiting the GPU. *Bulletin of the American Meteorological Society*, *93*(3), 307–314. Retrieved 2023-05-22, from https://journals.ametsoc.org/doi/10.1175/BAMS-D-11-00059.1 doi: 10.1175/BAMS-D-11-00059.1

Verzijlbergh, R. (2021). Atmospheric flows in large wind farms. *Europhysics News*, *52*(5), 20–23. Retrieved 2023-05-31, from https://www.europhysicsnews.org/10.1051/epn/2021502 doi: 10.1051/epn/2021502