Simulating the Euler equations on a cluster of GPUs using Python

André R. Brodtkorb*† and Martin L. Sætra†,‡

†Oslo Metropolitan University, Department of Computer Science, Oslo, Norway
‡Norwegian Meteorological Institute, Oslo, Norway

July 3rd, 2022

Abstract

GPUs have become a household name in High Performance Computing (HPC) systems over the last 15 years. However, programming GPUs is still largely a manual and arduous task, which requires expert knowledge of the physics, mathematics, and computer science involved. Even though there have been large advances in automatic parallelization and GPU execution of serial code, it is still difficult to fully utilize the GPU hardware with such approaches. Many core numeric GPU codes are therefore still mostly written using low level C/C++ or Fortran for the host code.

Several studies have shown that using higher level languages, such as Python, can make software development faster and with fewer bugs. We have developed a simulator based on PyCUDA and mpi4py in Python for solving the Euler equations on Cartesian grids. Our framework utilizes the GPU, and can automatically run on clusters using MPI as well as on shared-memory systems.

Our framework allows the programmer to implement low-level details in CUDA C/C++, which is important to achieve peak performance, whilst still benefiting from the productivity of Python. We show that our framework achieves good weak and strong scaling on both shared-memory and distributed-memory GPU systems.

Keywords: GPU computing, CFD, conservation laws, finite-volume methods, Python, CUDA, MPI

1 Introduction

GPU computing started with proof-of-concept codes in the 1990s [1], continued with dedicated programming languages and platforms in the 2000s [2, 3, 4, 5], and is now becoming a necessity within High Performance Computing (HPC). Most of the top level HPC systems in the world [1] are currently equipped with a large GPU partition that applications need to utilize efficiently in order to get resource allocations on the systems.

*Personal preprint

[1]See `https://eurohpc-ju.europa.eu/about/our-supercomputers`, `https://www.olcf.ornl.gov/frontier/`, and `https://www.top500.org/`.

The most common way of efficiently utilizing GPU devices has been to write specialized kernels in Nvidia CUDA C++ for the "inner loops", and calling these kernels from a C/C++ or Fortran host application through an API.

Although most existing HPC codes are written in C/C++ and Fortran, a growing portion of scientific software is developed in Python, which is currently one of the most popular programming languages for scientific computing applications. [6] Within STEM education and training, Python is also a popular choice, leading to a new generation of scientific programmers and researchers which is more proficient in Python than the traditional compiled languages. For experimentation and prototyping, Python is of particular interest due to its low verbose code and high productivity[7, 8], how it can be combined with C++ and Fortran code, and the wide range of available tools and third-party libraries for numerical and scientific applications [2].

Holm, Brodtkorb and Sætra [9] investigated GPU computing with Python for single-GPU systems, and found that the overhead of using Python for stencil-based simulator codes, compared to low-level languages, is negligible with regards to performance. The energy efficiency does vary some between different numerical schemes, optimization levels and GPUs, but the run time is unsurprisingly the most important factor. They note that the productivity of working with Python is significantly higher than using C++. Herein, we take the first step to extend these findings to a multi-GPU setting. We show that Python code with GPU acceleration can be moved from the prototype stage into HPC production code, obtaining good results in terms of parallel efficiency and scaling on two HPC systems with very different characteristics, also without architecture-specific optimization. Our code can be run through scripts for batch runs, but is also easily used interactively (in a read-eval-print manner) from Jupyter notebooks, also for multi-GPU simulation with MPI. We are simulating the Euler equations using finite-volume methods with a stencil-based high-resolution time-stepping scheme for our experiments. The Euler equations describe adiabatic and inviscid flow, and have a wide range of applications, from numerical weather prediction to simulation of air flow around an aircraft wing. The methods we use may also be applied to other hyperbolic conservation and balance laws, and have been demonstrated to achieve high performance both on single GPUs [10, 11] and multiple GPUs [12] for the shallow-water equations and stencil-based solvers in general.

There exist other similar Python-based frameworks that utilize GPUs for computation fluid dynamics. Witherden, Farrington and Vincent [13] describes PyFR, a framework for solving the Euler and Navier-Stokes equations on unstructured grids using both CPUs and GPUs. The framework relies on a domain specific language to specify point-wise kernels that are interpreted at runtime and used to generate CUDA kernels that run on the GPUs. Walker and Niemeyer [14] apply the swept rule for solving the two-dimensional heat equation and Euler equations on heterogeneous architectures. They note that great care must be taken when designing a solver for such architectures in order to achieve decent speed-up. Oden [15] investigates the differences between native CUDA C++ code and CUDA code written in Python using Numba, using both microbenchmarks and real applications. She shows that the Numba versions only reach between 50% and 85% of the performance of the native CUDA C++ for compute-intensive benchmarks. This suggests that it is still necessary to "hand code" the inner loops in order to get maximum performance on the GPU.

The use of Python for scientific computing and machine learning has exploded over the last decade. [16, 17] For HPC applications, one major challenge is to balance productivity with efficiency. In the work presented here, the highest efficiency is always within reach since the numerical schemes are written in CUDA C++,

[2] See `https://numfocus.org/sponsored-projects/affiliated-projects` and `https://numfocus.org/sponsored-projects`.

allowing for fine-tuning the code to a particular GPU. At the same time, high productivity is facilitated by keeping all other parts of the code in Python. Moreover, by using only standard Python and well-known and mature third-party libraries, debugging and profiling of the code is kept manageable.

2 Materials and Methods

This section details the mathematical discretization, its implementation on GPU and extension to multiple GPUs. We also outline how the profiling of the MPI+GPU application was performed.

2.1 Mathematical formulation

The two-dimensional Euler Equations are a simplification of the more complex Navier-Stokes equations, and can be written

$$\begin{bmatrix} \rho \\ \rho u \\ \rho v \\ E \end{bmatrix}_t + \begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho v u \\ u(E+p) \end{bmatrix}_x + \begin{bmatrix} \rho v \\ \rho u v \\ \rho v^2 + p \\ v(E+p) \end{bmatrix}_y = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \tag{1}$$

in which ρ is the fluid density, the vector $[\rho u, \rho v]$ represents the momentum, E is the total energy, and p is the pressure. The total energy can be written as

$$E = \frac{1}{2}\rho(u^2 + v^2) + p/(\gamma - 1), \tag{2}$$

in which γ is the adiabatic exponent. We can also write the equations on vector form,

$$\frac{\partial Q}{\partial t} + \frac{\partial F(Q)}{\partial x} + \frac{\partial G(Q)}{\partial y} = 0,$$

in which Q is our vector of conserved variables, and F and G are the source terms that govern the fluid dynamics. We can then discretize our spatial derivatives, and end up with the following ordinary differential equation that we have to solve in time:

$$\frac{\partial Q}{\partial t} = -\frac{\partial F(Q)}{\partial x} - \frac{\partial G(Q)}{\partial y}. \tag{3}$$

There are multiple ways of solving these equations. We can for example use the classical Lax-Friedrichs numerical scheme, which gives us the following discretization in two dimensions:

$$\begin{aligned} Q_{i,j}^{n+1} = \frac{1}{4}\left[Q_{i+1,j}^n + Q_{i-1,j}^n + Q_{i,j+1}^n + Q_{i,j-1}^n\right] \\ - \frac{\Delta t}{2\Delta x}\left[F(Q_{i+1,j}^n) - F(Q_{i-1,j}^n)\right] \\ - \frac{\Delta t}{2\Delta y}\left[G(Q_{i,j+1}^n) - G(Q_{i,j-1}^n)\right], \end{aligned} \tag{4}$$

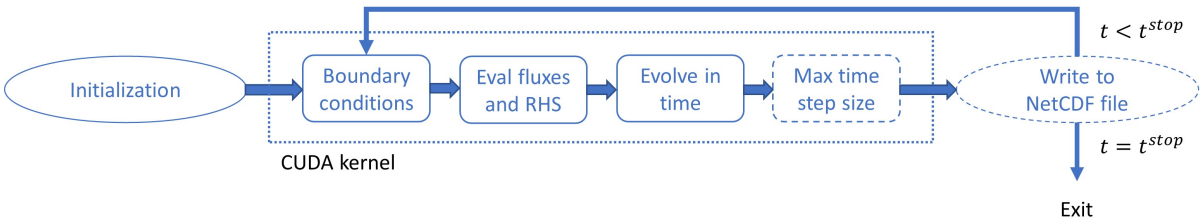

Figure 1: Program flow on a single GPU. All the boxes within the dotted square represent stages within the CUDA kernel. After completing all the CUDA kernel stages, the simulation is advanced one step in time. The dashed boxes are optional, and if the max time-step size if not computed, a fixed time-step size must be given.

with a time step restricted by a CFL condition. This scheme is a two-dimensional scheme, in which we compute the new solution based on a two-dimensional stencil.

It is well known that this classical first order scheme is numerically dissipative, leading to overly smeared solutions as time progresses. We can improve on this, by using a higher-order scheme. In this work, we have used a second-order accurate MUSCL-Hancock scheme, which in one dimension can be written

$$\bar{Q}_i{}^L = Q_i^L + \frac{1}{2}\frac{\Delta t}{\Delta x}\left[F(Q_i^L) - F(Q_i^R)\right], \tag{5}$$

$$\bar{Q}_i{}^R = Q_i^R + \frac{1}{2}\frac{\Delta t}{\Delta x}\left[F(Q_i^L) - F(Q_i^R)\right]. \tag{6}$$

Here, $\bar{Q}_i{}^L$ denotes the reconstructed value at the left hand side of cell i, and equivalently $\bar{Q}_i{}^R$ denotes the reconstructed value at the right hand side. This means that we have the two states, $\bar{Q}_i{}^R$ and $\bar{Q}_{i+1}{}^L$, at the interface between cells i and $i+1$, and we use the Harten-Lax-van Leer (HLL) approximate Riemann solver with these two states to compute the HLL flux, $\overline{F}_{i+1/2}$, across the interface. The scheme is second-order accurate in one dimension [18]. To extend it to two-dimensional problems, we use a dimensional splitting scheme that is second-order accurate every other time step [19, 18],

$$Q^* = Y^{\Delta t} X^{\Delta t} Q^n, \tag{7}$$

$$Q^{n+2} = X^{\Delta t} Y^{\Delta t} Q^*. \tag{8}$$

Here, $X^{\Delta t}$ and $Y^{\Delta t}$ are the one-dimensional operators that advance the solution in time along the x and y-axis, respectively:

$$X^{\Delta t} = Q^n - \Delta t\left[\overline{F}_{i+1/2,j} - \overline{F}_{i-1/2,j}\right], \tag{9}$$

$$Y^{\Delta t} = Q^n - \Delta t\left[\overline{G}_{i,j+1/2} - \overline{G}_{i,j-1/2}\right]. \tag{10}$$

2.2 Program structure

The simulator is written using Python (3.7.12), NumPy (1.21.6), netcdf4 (1.5.8), PyCUDA [20] (2021.1), mpi4py [21, 22, 17] (3.1.3), OpenMPI (4.1.0), and CUDA (11.4.1) to combine the power of multiple GPUs in a distributed-memory or shared-memory architecture. The traditional programming language for HPC has been Fortran – which is evident from the large array of Fortran programs running on supercomputers today. The common explanation for this is that nothing can surpass Fortran in terms of performance. Whilst it may be true that Fortran typically runs faster than Python, the approach of using Python has some merits. [23] We use Python here to combine traditional MPI-based supercomputing with GPU computing in a hybrid approach. The libraries we use have a base/core in C/C++, which gives us the flexibility of Python with close to the speed of compiled languages.

The program structure for a single-GPU simulation is shown in Figure 1. Simulator initialization constructs initial conditions, a CUDA context and a simulator ready for time-stepping. Initial conditions and all simulation parameters needed to re-run the simulation is saved to a NetCDF file for easy reproducible results. Each time step consists of enforcing the boundary conditions upon reading in data from the previous time step, evaluation of new fluxes and any right-hand side source terms, forward time integration, and an optional computation of the maximum time-step size (for the next time step) based on the CFL condition. If the maximum time-step size is not computed, a fixed time-step size must be provided. Intermediate results can be written to the NetCDF file at prescribed simulation times.

2.3 CUDA implementation

The numerical scheme outlined in the Section 2.1 is implemented as a single-GPU kernel that computes either Equation 7 or Equation 8 depending on a flag sent as a variable. This means that after two kernel invocations, both equations are computed once and the solution is evolved two time steps (recall that the numerical scheme is only second-order accurate every second time step). The GPU is completely managed through PyCUDA: The device is controlled through the device API (`pycuda.driver`), the CUDA C++-kernels are compiled using just-in-time compilation (`pycuda.compiler`), and data is stored using GPU Arrays (`pycuda.gpuarray.GPUArray`). CUDA kernels are run using prepared function calls.

If we examine the data dependencies of the numerical scheme (i.e., the numerical stencil), we observe that the kernel requires a two-row halo in all directions, as shown in Figure 2. The kernel is then implemented in a standard way, using shared memory on the GPU to store the physical and reconstructed variables. The principles from our earlier implementations of numerical schemes for shallow-water simulations [10, 11] are applied. By computing both in the x-dimension and the y-dimension for each kernel execution we avoid having to read the physical values twice, at the expense of an extra row and column of halo cells. In this work, we have used a fixed CUDA block size of 16 by 8 cells, but the code does contain an autotuner which can be used to optimize the block size for performance on a particular GPU (see Section 4.2 in [9]).

In addition to the numerical scheme presented for the Euler equations, the simulator also contains various numerical schemes for solving the two-dimensional shallow-water equations.

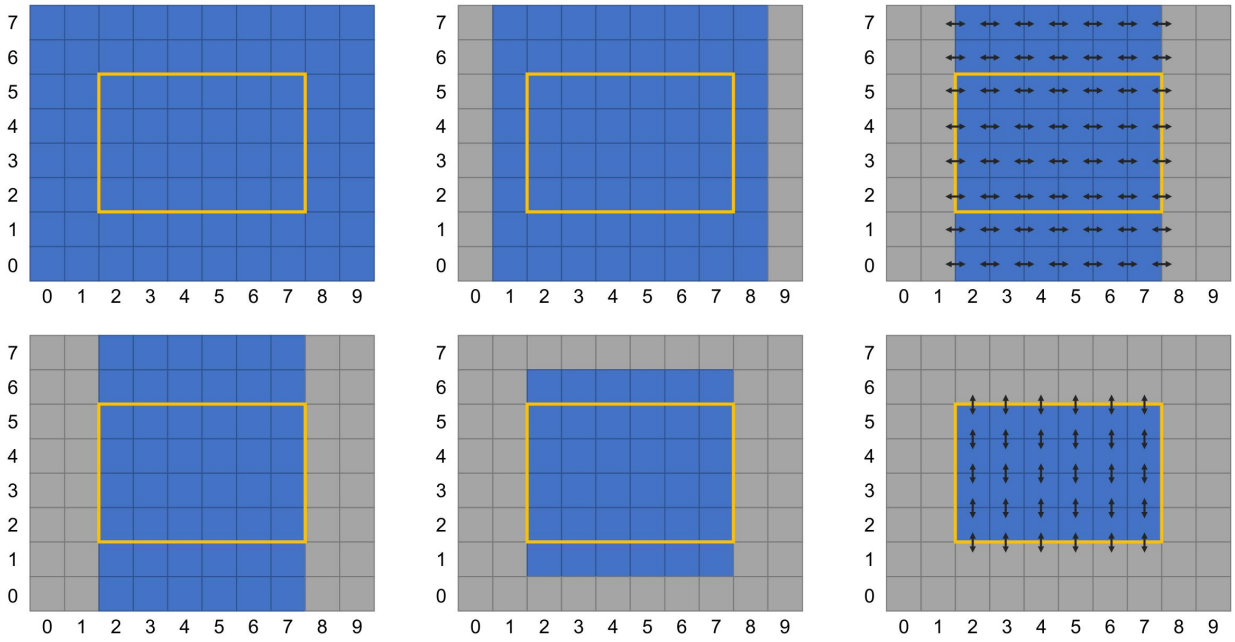

Figure 2: Halo region for the CUDA kernel when computing Equation 8. The top row shows how the input values (left) are used to compute the slopes (center) along the x-axis. The slopes are consequently used to compute the face fluxes (right). Equivalently for the y-axis, the bottom row shows how the input values (left) are used to compute the slopes (center), and finally the fluxes along the y-axis. Thus, an input grid of 8×10 cells is used to compute the fluxes for the inner 4×6 cells.

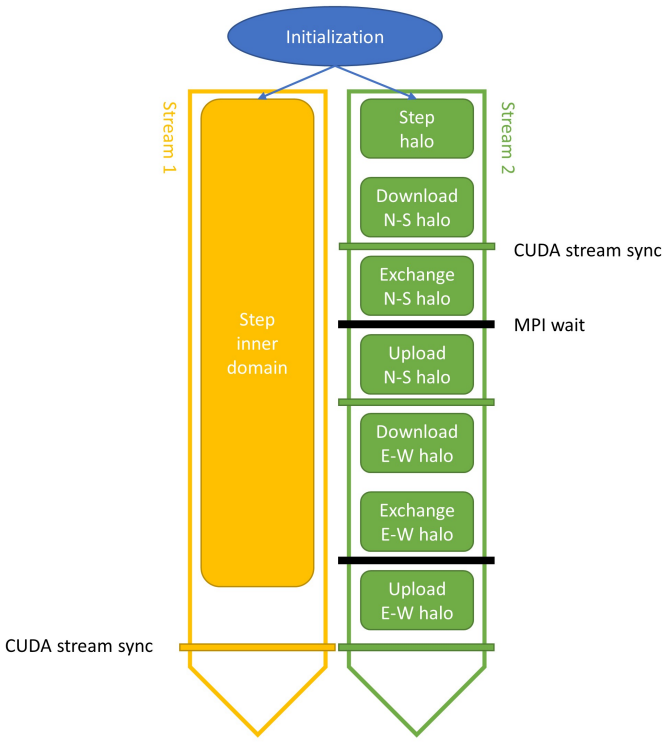

Figure 3: By using two CUDA streams we can perform memory operations and computations simultaneously. The inner domain is advanced one time step in Stream 1 and the halo in Stream 2. Since the inner domain usually is much bigger than the halo, the compute time spent on the inner domain in Stream 1 will mask the time used for exchanging halo regions with neighboring subdomains in Stream 2, including downloading (GPU to CPU), send/receive with MPI, and uploading (CPU to GPU). The step stage executes the same kernel for both streams, shown in Figure 1, but for different areas of the domain.

2.4 Extension to multiple GPUs

For multi-GPU simulations we introduce Python classes for an MPI grid and an MPI simulator. Each MPI process has its own MPI simulator and CUDA context. The MPI grid handles domain decomposition and the necessary bookkeeping, keeping track of which subdomains that need to communicate and where the global boundaries are found. Both one- and two-dimensional domain decomposition are supported. In both cases the global Cartesian grid is decomposed uniformly between all MPI ranks, such that each subdomain will have at most two neighbors in 1D and at most four neighbors in 2D. The MPI simulator extends the base simulator class and adds the necessary functionality for exchanging halo cells between neighbouring subdomains and computing global Δt for simulations with a variable time-step size. All MPI calls are done through mpi4py.

In Figure 3, all the stages of a multi-GPU simulation are shown (from the perspective of one subdomain/MPI rank), divided between two CUDA streams. To minimize the communication overhead, we use asynchronous memory transfers that overlap with computations. This enables us to hide most of the cost associated with downloading, exchanging and uploading halo cells each time step. There are some prerequisites for this: Host memory needs to be pagelocked, the memory operations must be asynchronous and they must

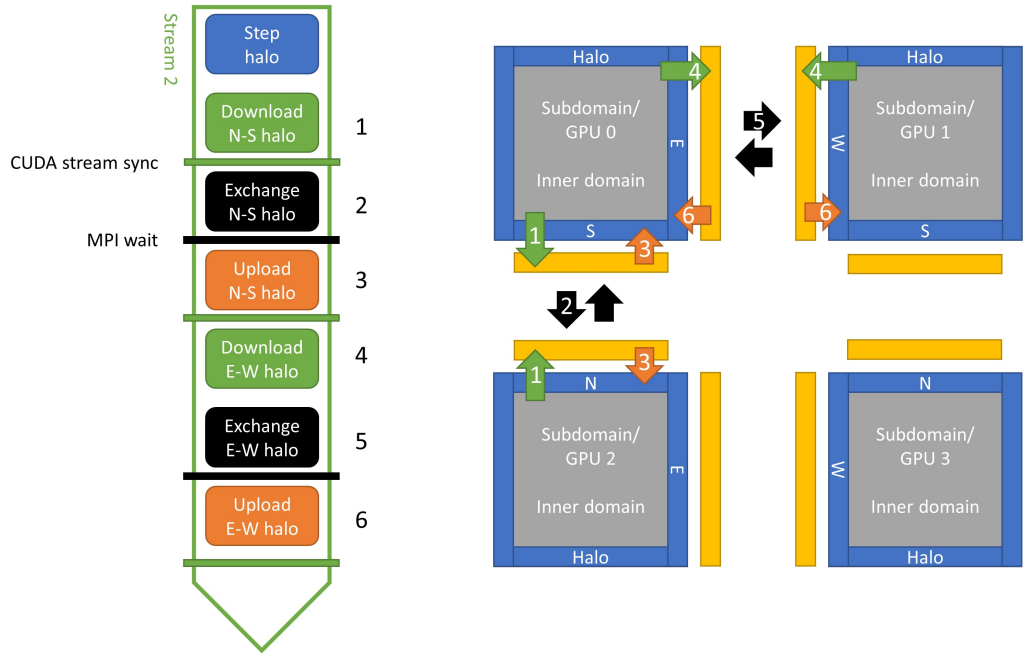

Figure 4: This schematic outline shows the different stages executed in Stream 2 and how they relate to the computational grid. Note that the halo computed in the first stage includes both the regions that will be copied to neighboring subdomains (inne halo) and the areas where values from neighboring subdomains will be copied into (outer halo).

be issued on a different stream than the computational kernel(s) that is intended to run simultaneously. The issue order of computational kernels and memory operations on the two streams may impact the degree of overlap. Furthermore, the size of the computational domains and different hardware specifications (GPU generation and class) may also lead to a varying degree of overlapping execution between the two streams.

Figure 4 shows a schematic outline of the complete halo cell exchange performed in Stream 2, related to the computational grid. The halo cell buffers for all four boundaries of each subdomain is allocated through PyCUDA's device interface. They are allocated once as pagelocked memory at simulation initialization, and then used throughout the rest of the simulation. These buffers are also used for the global boundaries for certain boundary conditions, e.g., periodical boundaries. Memory transfers are performed asynchronously by calling PyCUDA Memcpy2D objects, specifying the stream we want the memory operation issued to. The sizes of the halo regions are easy to adjust to permit the use of other numerical schemes with different stencil sizes.

The exchange of halo regions with neighboring subdomains is done with matching non-blocking MPI Isend/Irecv pairs with MPI wait at the global synchronization points. The corner cells are transferred first in the north-south direction, then in the east-west direction. This minimizes the size of the transfer buffers and avoids diagonal transfers between subdomains, at the expense of an extra MPI synchronization to ensure that all north-south exchanges are done before any east-west exchanges start. As the internal domain is being

computed simultaneous to the full halo exchange process, including all but the final synchronization point, this effectively hides most of the communication overhead.

2.5 Profiling multi-GPU applications

For a complete documentation of the tools used herein, we refer to the user guides. The walk-through given here describes our use of the tools and may also serve as a quickstart guide for others.

The Nvidia Visual Profiler and the `nvprof` command line interface (CLI) tool have long been the go-to profiling and analysis tools for GPU applications, the latter for remote profiling on servers. In 2018, Nvidia introduced a new suite of Nsight Tools; Systems [3], Compute, and Graphics, which is now the new standard profiling tools. Nsight Systems (`nsys`) for complete system profiling and analysis, including OpenGL, OpenMP, MPI, and CPU sampling; Nsight Compute (`ncu`) for GPU kernel profiling and analysis; and Nsight Graphics (`ngfx`) for profiling graphics applications.

This is how we profiled our code with Nsight Systems on the remote systems (using CLI):

1. Install Nsight Systems (x86_64, IBM Power, and ARM SDSA target versions are available in the CUDA Toolkit)
2. Run `nsys profile <application executable>`
3. To trace MPI, add the -t option. Here we are tracing CUDA, NVTX, OS runtime and MPI: `-t cuda,nvtx,osrt,mpi`
4. Copy or move `report1.qdrep` (standard name) to local machine
5. Open `report1.qdrep` with `nsys-ui` to view profile

It is also possible to do live profiling of a remote host by connecting your locally running Nsight Systems UI to a daemon running on the remote system, but this is generally not recommended due to security issues with unencrypted connection and plain-text password storage.

3 Results

The simulator has been profiled using Nsight Systems for multi-GPU and MPI performance. Furthermore, the simulator has been benchmarked on two systems, by instrumenting the Python code to measure run times, demonstrating good weak and strong scaling. We describe the experiment setup in terms of simulation case and hardware, and present the results.

3.1 Experiment description

We have simulated the Kelvin-Helmholtz instability, which models the dynamics occurring at the interface between two fluids with different speeds. An example of such a simulation is shown in Figure 5. This phenomenon occurs naturally, e.g., in a stratified atmosphere when there is a layer of fast moving air over a slower moving layer of air.

[3] `https://docs.nvidia.com/nsight-systems/UserGuide/index.html`

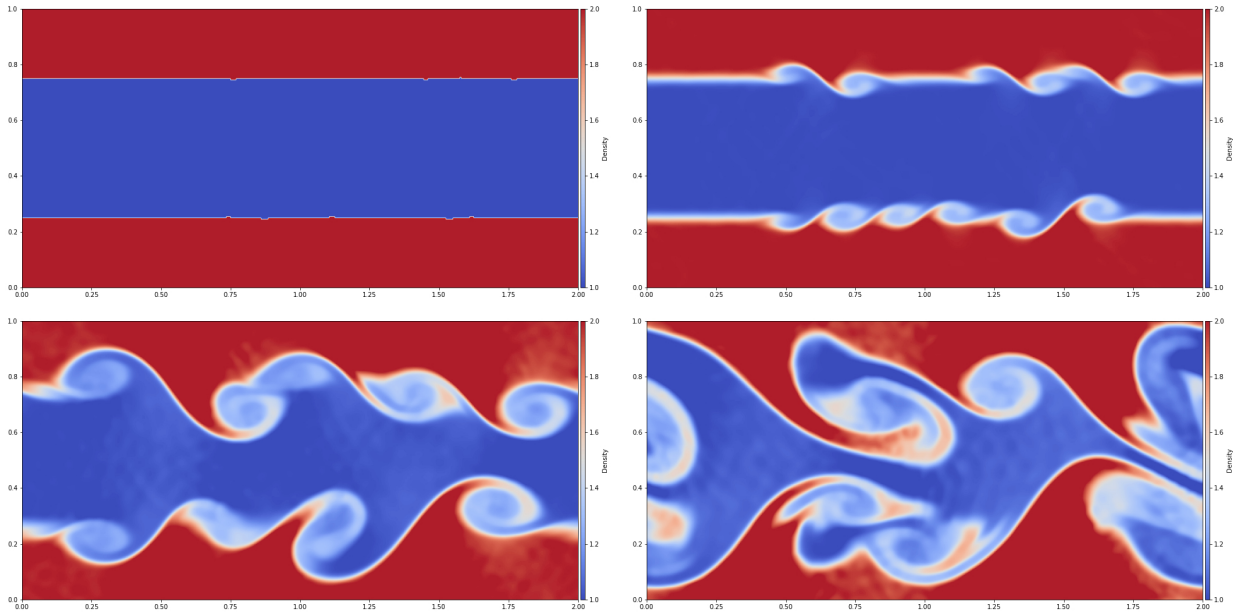

Figure 5: A simulation of the Kelvin-Helmholtz instability. The four panels show density in the initial conditions and after 1, 2 and 3 seconds of simulation time. See animation at `https://www.youtube.com/watch?v=ejCzlwIT--w`

We generate a periodic test case in which both the upper and lower quarter of the fluid move towards the left, whilst the center moves towards the right. We define the two regions with the following properties:

$$\rho_1 = 1.0, \qquad u_1 = 0.5 \qquad v_1 = 0 \tag{11}$$

and

$$\rho_2 = 2.0, \qquad u_2 = -0.5 \qquad v_2 = 0. \tag{12}$$

The total energy is computed using Equation 2, in which we set the adiabatic exponent $\gamma = 1.4$. The interface between the two regions is slightly perturbed to trigger the instability.

3.2 Benchmark systems

The scaling benchmark has been run on two different systems with very different characteristics, but on both systems the Slurm Workload Manager was used to launch all jobs.

The first system, the DGX-2 [4], is a shared-memory architecture with 16 Nvidia Tesla V100 GPUs. All the GPUs have 32 GB of local memory each (total of 512 GB) and share 1.5 TB main memory. The peak performance in double precision is 125 teraflops. We had exclusive access to this system while running the experiments.

The second system, Saga [5], is a distributed-memory system and a part of the Norwegian research infrastructure. The system has eight nodes with four Nvidia Tesla P100 GPUs installed in each node. The

[4] `https://www.ex3.simula.no/resources`
[5] `https://documentation.sigma2.no/hpc_machines/saga.html`

Figure 6: Screendump of timeline in the Nsight Systems UI zoomed in on one time step on one GPU (in a two-GPU simulation), showing overlap of memory operations (in Stream 14) and MPI communication (designated MPI), and the computational kernel for the inner domain (in Stream 15). The CUDA API calls are shown in the bottom row. Note that the timeline is split over two rows. The color codes for the bars are as follows: Blue is compute kernel execution or API call, purple is download (GPU to CPU), green is upload (CPU to GPU), gray is MPI Isend/Irecv or wait, red is memory API calls, and yellow is CUDA thread synchronization.

GPUs have 16 GB of local memory each. Each node has two CPUs with 24 cores and 384 GiB memory each. The nodes are connected by InfiniBand HDR. This system was shared with other users while running the experiments.

3.3 Multi-GPU profiling

From Figure 6 we see that the memory operations (downloading from, and uploading to, the GPU) and MPI communication and synchronization with other subdomains are performed simultaneously with the execution of the computational kernel for the inner domain. As the domain size is increased, so is the ratio of the inner domain area to the halo area. Thus, at some point, the compute time for the inner domain will hide all the cost of halo cell exchange process except the API calls and other fixed overheads. Since there are four conserved variables (see Equation 1) and up to four neighboring subdomains (see Figure 4), this results in a total of 16 sets of downloading, uploading and exchanging halo cell buffers for the subdomains which are not on the global domain boundary.

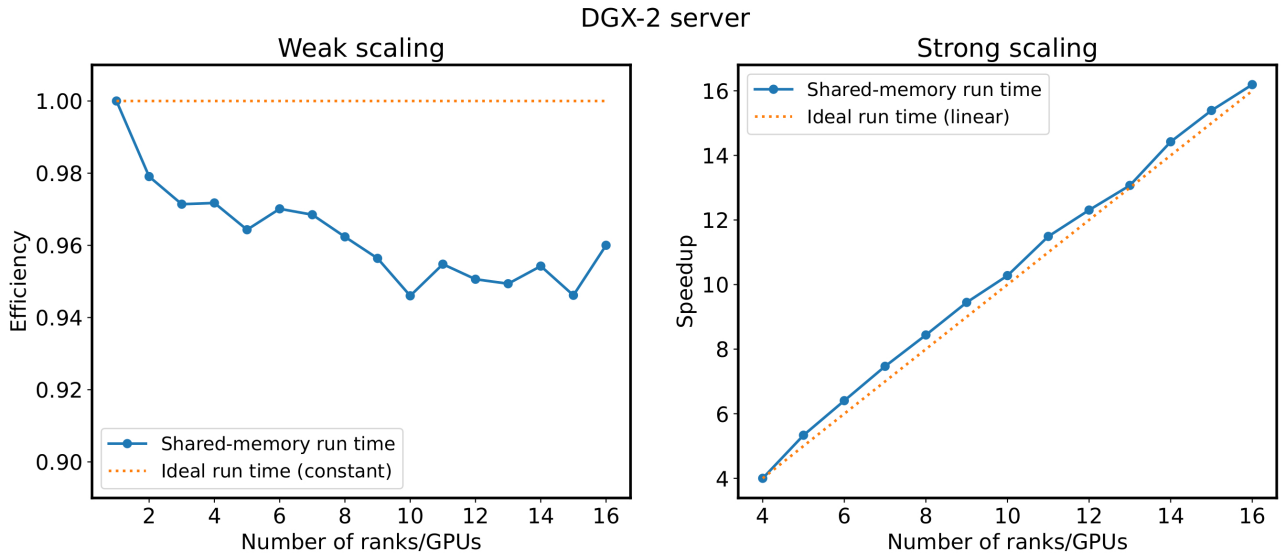

Figure 7: Results from scaling experiments on DGX-2 shared-memory system. Weak scaling (1–16 GPUs) is shown in the left panel and strong scaling (4–16 GPUs) is shown in the right panel. Run times are normalized with respect to the single rank/GPU experiment for weak scaling and the four ranks/GPUs for strong scaling.

3.4 Multi-GPU scaling

For all scaling experiments, the domain size is set such that close to all available memory is used on each GPU, and we run the simulation for 200 time steps with a fixed time-step size. For weak scaling the subdomain size per GPU is kept fixed, and we run experiments using from one GPU up to the available number GPUs in the system. For strong scaling the global domain size is fixed, and we run experiments using from four GPUs (the domain size is scaled to fit on four GPUs) up to the available number of GPUs in the system. All initialization, writing of results to NetCDF files and cleanup after the last time step are not included in the time measurements, meaning that we measure only the stages of Stream 1 and Stream 2 shown in Figure 3.

Results from scaling experiments on the DGX-2 system is shown in Figure 7. For weak scaling, shown in the panel to the left, we have a minimum efficiency of 94% compared to the single-GPU run. There are some variations in the efficiency in the interval between 94% and 96% efficiency from 9–16 GPUs. This suggests that there are still some overheads connected to the halo cell exchange that are not completely hidden. For strong scaling, shown in the panel to the right, we observe linear scaling from 4–16 GPUs, meaning that no significant extra cost in terms of run time is incurred when going from 4 to 16 GPUs.

Results from scaling experiments on the Saga cluster is shown in Figure 8. There are two graphs for weak scaling, shown in the panel to the left, one for shared memory on a single node and one for distributed memory on 1–4 nodes with one GPU on each node. In both graphs, measurements are normalized with respect to the single-node single-GPU run time. The single-node weak scaling have a minimum efficiency of 98% and the multi-node weak scaling have a minimum efficiency of 90%. Again, this shows that there is still some overheads connected to the halo cell exchange that is not completely hidden. We attribute the lower efficiency in Saga (for the distributed-memory run times), compared to the shared-memory run times, to the

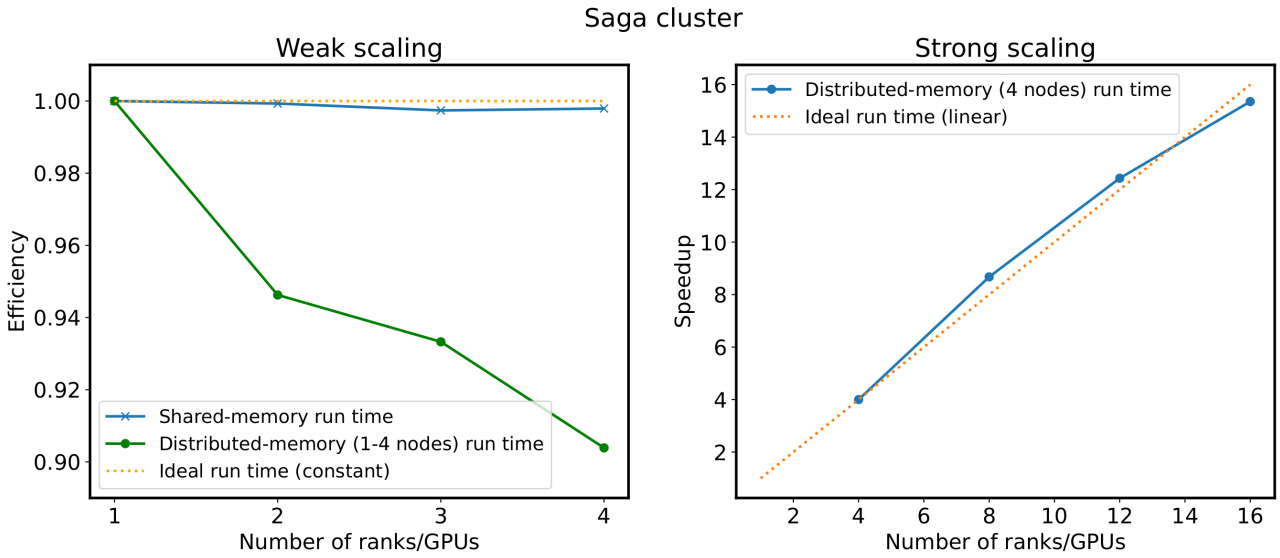

Figure 8: Results from scaling experiments on the Saga cluster. Weak scaling (1–4 GPUs on a single node and 1–4 nodes with one GPU on each node) is shown in the left panel and strong scaling (4–16 GPUs divided equally between four nodes) is shown in the right panel. Run times are normalized with respect to the single rank/GPU experiment for weak scaling and the four ranks/GPUs for strong scaling.

interconnect and the fact that this is a shared system with many active users. We would have liked to use up to 16 GPUs for the weak scaling experiments, but limitations due to congestion and Slurm setup made this difficult. Strong scaling, shown in the panel to the right, scales linearly from 4 to 16 GPUs.

4 Discussion

We have described a Python-CUDA C++-based prototype framework for multi-GPU simulation of the Euler equations using finite-volume methods. By implementing everything but the numerical schemes in Python, the code is kept short and manageable, which facilitates productivity. No domain specific language has been used and only a limited number of well-known and mature third-party libraries. This enhances the readability of exception stack traces and reduces the vulnerability of discontinued third-party libraries for GPU acceleration, inter-process communication, domain decomposition, etc. The numerical scheme is implemented in CUDA C++, which gives a high level of flexibility and the possibility to optimize for a particular GPU. Autotuning of CUDA block size is implemented and can give increased performance portability between different generations and classes of GPUs. The code can be adapted to simulate other conservation laws and to use different numerical schemes. The simulator has been shown to be efficient through profiling and scalability experiments on both shared-memory and distributed-memory systems. As the amount of scientific Python code bases grows together with the number of GPU-accelerated HPC-systems, these are promising results.

The results show that there are still improvements to be made, particularly for distributed-memory systems without a fast interconnect. One possible option is to use ghost cell expansion [24, 12], which would increase the size of the halo cell regions, allowing for running multiple time steps for each halo cell exchange.

Hybrid MPI+OpenMP (threads) would decrease the number of total MPI messages, but since inter-process communication costs already are efficiently hidden by overlapping with computation on the shared-memory system, this seems like a high investment in code complexity for a low return in terms of reduced run time. So-called MPI+MPI (MPI-integrated shared memory) and experimentation with processor affinity are also something to be explored.

Future work includes adding more applications and numerical schemes, with extension to three dimensions. A more robust implementation of asynchronous communication that take GPU (PCIe, NVLink and NVSwitch) and system interconnect into account could further increase the scaling performance by leveraging CUDA-aware MPI, UCX, and RDMA. [25, 26] Domain decomposition for load balancing in heterogeneous systems (CPUs and GPUs, or different generations and classes of GPUs) would allow for better performance portability for simulations on heterogeneous systems and multi-GPU simulations. The scaling benchmark should also be run on larger systems to investigate the limits of scalability and how the workload manager's resource allocation and management may affect the performance, depending on which flags that are used. Comparison studies on alternatives to CUDA C++ would give more insight into efficiency-productivity considerations, including HIP (by using HIPIFY), SYCL and even pragma/directive-based approaches. This would also break the current dependency on Nvidia GPUs.

The code is available at `https://github.com/babrodtk/ShallowWaterGPU`

Author contribution (CRediT statement)

André R. Brodtkorb – Conceptualization; Methodology; Software; Writing - Original Draft; Writing - Review & Editing; Visualization; Funding acquisition.

Martin L. Sætra – Conceptualization; Software; Validation; Investigation; Writing - Original Draft; Writing - Review & Editing; Visualization; Funding acquisition.

Acknowledgments

The research contribution of Sætra is funded by The Norwegian Research Council under the project "HAV-VARSEL – Personalized ocean forecasts in a two-way data flow system" (310515).

The shared-memory simulations were performed on a system provided by Simula Research Laboratory under the eX3 project [6].

The Saga cluster simulations were performed on resources provided by Sigma2 – the National Infrastructure for High Performance Computing and Data Storage in Norway under project number nn9882k.

References

[1] E. Larsen and D. McAllister. Fast matrix multiplies using graphics hardware. In *ACM/IEEE Conference on Supercomputing*, pages 55–55, New York, NY, USA, 2001. ACM.

[6] `https://www.ex3.simula.no/`

[2] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Tim Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, March 2007.

[3] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[4] Andre R. Brodtkorb, Christopher Dyken, Trond R. Hagen, Jon M. Hjelmervik, and Olaf O. Storaasli. State-of-the-art in Heterogeneous Computing. *Scientific Programming*, 18(1):1–33, 2010.

[5] André R. Brodtkorb, Trond R. Hagen, and Martin L. Sætra. Graphics processing unit (GPU) programming strategies and trends in GPU computing. *Journal of Parallel and Distributed Computing*, 73(1):4–13, January 2013.

[6] Lorena A. Barba. The Python/Jupyter Ecosystem: Today's Problem-Solving Environment for Computational Science. *Computing in Science & Engineering*, 23(3):5–9, May 2021.

[7] S. Nanz and C. A. Furia. A comparative study of programming languages in rosetta code. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 778–788, May 2015.

[8] Lutz Prechelt. An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl for a search/string-processing program. Technical report, Karlsruhe Institute of Technology, 2000.

[9] Håvard H. Holm, André R. Brodtkorb, and Martin L. Sætra. GPU Computing with Python: Performance, Energy Efficiency and Usability. *Computation*, 8(1):4, March 2020.

[10] André R. Brodtkorb, Martin L. Sætra, and Mustafa Altinakar. Efficient shallow water simulations on GPUs: Implementation, visualization, verification, and validation. *Computers & Fluids*, 55(0):1–12, 2012.

[11] André R. Brodtkorb and Martin L. Sætra. Explicit shallow water simulations on GPUs: Guidelines and best practices. In *XIX International Conference on Water Resources, CMWR 2012, June 17–22, 2012*. University of Illinois at Urbana-Champaign, 2012.

[12] Martin Sætra and André Brodtkorb. Shallow water simulations on multiple GPUs. In Kristján Jónasson, editor, *Applied Parallel and Scientific Computing*, volume 7134 of *Lecture Notes in Computer Science*, pages 56–66. Springer Berlin / Heidelberg, 2012.

[13] F. D. Witherden, A. M. Farrington, and P. E. Vincent. PyFR: An open source framework for solving advection–diffusion type problems on streaming architectures using the flux reconstruction approach. *Computer Physics Communications*, 185(11):3028–3040, November 2014.

[14] Anthony S. Walker and Kyle E. Niemeyer. Applying the Swept Rule for Solving Two-Dimensional Partial Differential Equations on Heterogeneous Architectures. *Mathematical and Computational Applications*, 26(3):52, September 2021.

[15] Lena Oden. Lessons learned from comparing C-CUDA and Python-Numba for GPU-Computing. In *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 216–223, March 2020.

[16] Lorena A. Barba, Andreas Klöckner, Prabhu Ramachandran, and Rollin Thomas. Scientific Computing With Python on High-Performance Heterogeneous Systems. *Computing in Science & Engineering*, 23(4):5–7, July 2021.

[17] Zane Fink, Simeng Liu, Jaemin Choi, Matthias Diener, and Laxmikant V. Kale. Performance Evaluation of Python Parallel Programming Models: Charm4Py and mpi4py. In *2021 IEEE/ACM 6th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*, pages 38–44, November 2021.

[18] E. Toro. *Shock-Capturing Methods for Free-Surface Shallow Flows*. John Wiley & Sons, Ltd., 2001.

[19] R. F. Warming and Richard M. Beam. Upwind Second-Order Difference Schemes and Applications in Aerodynamic Flows. *AIAA Journal*, 14(9):1241–1249, September 1976.

[20] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, B. Catanzaro, Paul Ivanov, and Ahmed Fasih. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. *Parallel Computing*, 38(3):157–174, 2012.

[21] Lisandro Dalcín, Rodrigo Paz, and Mario Storti. MPI for Python. *Journal of Parallel and Distributed Computing*, 65(9):1108–1115, September 2005.

[22] Lisandro Dalcin and Yao-Lung L. Fang. Mpi4py: Status Update After 12 Years of Development. *Computing in Science & Engineering*, 23(4):47–54, July 2021.

[23] Greg Wilson, D. A. Aruliah, C. Titus Brown, Neil P. Chue Hong, Matt Davis, Richard T. Guy, Steven H. D. Haddock, Kathryn D. Huff, Ian M. Mitchell, Mark D. Plumbley, Ben Waugh, Ethan P. White, and Paul Wilson. Best Practices for Scientific Computing. *PLOS Biology*, 12(1):e1001745, January 2014.

[24] C. Ding and Y. He. A ghost cell expansion method for reducing communications in solving PDE problems. ACM/IEEE Conference on Supercomputing, pages 55–55, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[25] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R. Tallent, and Kevin J. Barker. Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):94–110, January 2020.

[26] Jaemin Choi, Zane Fink, Sam White, Nitin Bhat, David F. Richards, and Laxmikant V. Kale. GPU-aware Communication with UCX in Parallel Programming Models: Charm++, MPI, and Python. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 479–488, June 2021.