Overlapping stencil computations on the CPU and the GPU

Nikolaos Tselepidis
ntselepidis@student.ethz.ch

Michal Sudwoj
msudwoj@student.ethz.ch

Abhimanyu Bhadauria
bhadaura@student.ethz.ch

August 2020

Abstract

In this project, we experimentally study various different approaches for implementing a fast 3D stencil code, often used in weather and climate simulations, that overlaps computations on the CPU and the GPU, in an attempt to efficiently utilize modern hardware and achieve maximum performance. We start by developing optimized code able to run on the CPU and the GPU, individually, and then we fuse them with an efficient strategy, that allows for an additional increase in performance. In our work, we compare three alternative parallel programming models; OpenMP, OpenACC, and CUDA. We present extensive numerical results for all three approaches along with discussions.

1 Introduction

Weather and climate modeling is an extremely compute-intensive task that always relied on the state-of-the-art in high-performance computing, in order to enable large-scale simulations and accurate predictions in reasonable amount of time. For much of it's history, this meant running climate models on multi-core clusters, using distributed and shared memory parallelism. However, with the advent GPGPU computing and the wide availability of modern hardware accelerators, which not only allow much higher peak performance, but are also much more energy efficient, providing a significant FLOP/Watt advantage, climate codes are gradually adapting to GPUs.

CPUs are general purpose processing units with a small number of powerful cores that are latency-optimized allowing for complex instruction level parallelism. They usually achieve this by means of a larger cache, larger instruction set, and a larger control unit. Most modern CPUs have less than a 100 cores.

GPUs on the other hand, consist of a large number of simple cores that are throughput-optimised for fast "component-wise" computations on arrays. Therefore, they excel at applying the same instruction concurrently on a very large number of inputs. This aspect of the GPU architecture lends itself well to graphical rendering applications. However, this same aspect has been widely utilized in scientific computing problems where a bulk of the computational effort involves applying the same instruction to a large array of data. Modern GPUs can have 1000s of cores, revealing their potential for massive parallelism.

This is where its rising prevalence in weather and climate codes comes in. Simulation of weather and climate models typically involve solving a system of conservation laws, i.e mass, momentum, and energy, known together as the Navier-Stokes-Fourier equations, as well as some additional equations for air and water balance. These equations are given below:

$$\frac{d}{dt}\mathbf{v} = -2\boldsymbol{\Omega} \times \mathbf{v} - \frac{1}{\rho}\nabla_3 p + \mathbf{g} + \mathbf{F} \tag{1a}$$

$$C_v \frac{d}{dt}(\rho q) + p\frac{d}{dt}\left(\frac{1}{\rho}\right) = J \tag{1b}$$

$$\frac{\partial}{\partial t}(\rho) = -\nabla_3 \cdot (\rho \mathbf{v}) \tag{1c}$$

$$\frac{\partial}{\partial t} = -\nabla_3 \cdot (\rho \mathbf{v} q) + \rho(E - C) \tag{1d}$$

$$p = \rho R T \tag{1e}$$

Numerical solution of PDEs such as eq. (1) involves moving from a continuous domain to a discrete equivalent, and solving the PDEs approximately on a finite number of grid points. A wide class of methods for the solution of the discrete problem can be casted

as stencil computations, where the solution at every grid point is updated based on a fixed pattern, that involves local computations in the neighborhood of every grid point. This fixed pattern all over the computational domain can be exploited for extracting parallelism on GPUs.

In this project, we experimentally study various approaches based on the programming models OpenMP, OpenACC, and CUDA, for implementing an efficient 3D stencil code. We examine the performance one can achieve using CPU or GPU, individually, as well as CPU and GPU cooperatively, [1] to perform stencil computations. As a benchmark, we utilize a simplified set of equations, by reducing our focus to the 4th-order non-monotonic diffusion equation [2], i.e.:

$$\begin{aligned} \frac{\partial \phi}{\partial t} &= -\alpha \boldsymbol{\nabla^4 \phi} \\ &= -\alpha \boldsymbol{\Delta} \left(\boldsymbol{\Delta \phi} \right) \end{aligned} \tag{2}$$

Such numerical diffusion can be used in a weather or climate model as a means of controlling small-scale noise. Although much simpler than the full system of equations (1), the underlying mechanism of the stencil computations remains the same, and hence, can be utilized as a model problem for evaluating the efficiency of weather and climate codes.

In section 2, we present the different programming models we used for optimizing our stencil code implementations for CPUs and GPUs, along with some implementation details. In section 3, we showcase our results and analyze the viability of domain decomposition across the CPU and the GPU, in an attempt to cooperatively utilize the available hardware of modern compute nodes so as to maximize performance. We compare the performance results across the different programming paradigms we implemented and argue on them.

2 Stencil Computations

Given the continuous 4th order PDE, eq. (2), we discretize the two derivatives in space and time. We choose 2nd order central-differences for the space derivatives along with an explicit Euler time-stepping scheme. Hence, we obtain the following the stencils:

$$\begin{aligned} &\boldsymbol{\Delta} \phi^n_{i,j} = \\ &\left(-4\phi^n_{i,j} + \phi^n_{i-1,j} + \phi^n_{i+1,j} + \phi^n_{i,j-1} + \phi^n_{i,j+1} \right) / \Delta x \Delta y \end{aligned} \tag{3a}$$

$$\partial_t \phi^n_{i,j} = \left(\phi^{n+1}_{i,j} - \phi^n_{i,j} \right) / \Delta t \tag{3b}$$

The truncation of the original continuous domain implies that suitable boundary conditions need to be imposed. We assume periodic boundary conditions.

The main skeleton of the stencil code we implemented is given below.

Algorithm 1 Stencil code structure

```
1: for all iterations do
2:     UpdateHalo(in)                    ▷ Periodic BCs
3:     ApplyStencil(in, out)     ▷ 4th-order diffusion
4:     if not in last iteration then
5:         Swap(in, out)
6:     end if
7: end for
```

It should be noted, that in the UpdateHalo() step we impose the boundary conditions. Moreover, we need to mention that since we have to deal with a 4th-order diffusion term i.e. eq. (2), the stencil eq. (3) is applied twice inside the ApplyStencil() routine.

2.1 Stencils on Multicore CPUs

OpenMP. In our CPU-only stencil code implementation we utilize OpenMP to achieve multithreading. OpenMP is a directive based programming paradigm that follows the fork-join model, where the master thread forks a specified number of slave threads and the system divides a task among them. The threads then run concurrently. The parallelization of a program can be achieved using simple `#pragma omp` directives, and no substantial modification of the serial code are needed. In our CPU-only stencil code, we parallelized the halo update using `#pragma omp for collapse(2) nowait` and the two applications of the stencil eq. (3) inside every z-slice using `#pragma omp parallel for` on the 2D-loop.

2.2 Stencils on GPUs

OpenMP. OpenMP also provides support for offloading computation to different "accelerators", currently GPUs. This is done in a similar manner

to CPU OpenMP code, by annotating code with `pragmas` (C++) or directives (Fortran). The directive `target data` can be used to manually manage data movement to and from the GPU, while the `target` directive runs the surrounded code on the accelerator. Finer tuning of parallelization can be achieved using `teams`, `distribute` and `parallel do`. Due to the hierarchical nature of these directives, loop order is just as important for achieving good performance as it is in sequential code. However, a big advantage of the OpenMP programming model, is the fact that the actual C++ or Fortran code does not require modification aside from adding directives.

OpenACC. OpenACC is another directive based programming paradigm for offloading work to accelerators, which is hardware agnostic, hence it can be used across different accelerators. The primary advantage of OpenACC, as with other directive based programming models, is the ease of deployment which does not require making major changes to code - such as would be needed if using a more lower level model such as CUDA. We use the `pragma acc parallel` directive to designate a scoped region where the code to be accelerated lies.

However, to gain the most out of OpenACC, it is always good to write code in such a way that optimisations are unlocked. In our case this means using the correct loop order (having the index with the smallest stride on the inner most loop), eliminating dependencies and possible aliasing, and collapsing loops. The most important directive, from where the bulk of the speedup comes is `pragma acc loop`, which offloads all the computation of the loop to the GPU.

For hybrid CPU-GPU problems, data movement is a significant bottleneck, we make use of unstructured data regions using the `pragma acc enter data` and `pragma acc exit data` to store data on the accelerator until it is manually exited.

CUDA. CUDA (Compute Unified Device Architecture) is a parallel computing platform and application programming interface (API) model created by NVIDIA. It is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels. Writing CUDA kernels requires a lot more effort than parallelizing the code using compiler directives like in OpenMP and OpenACC. However, it gives to the programmer more control over the application. Additionally, since every kernel launch is asynchronous, i.e. control returns to the CPU immediately after the kernel call, it allows writing applications that utilize CPU and GPU concurrently.

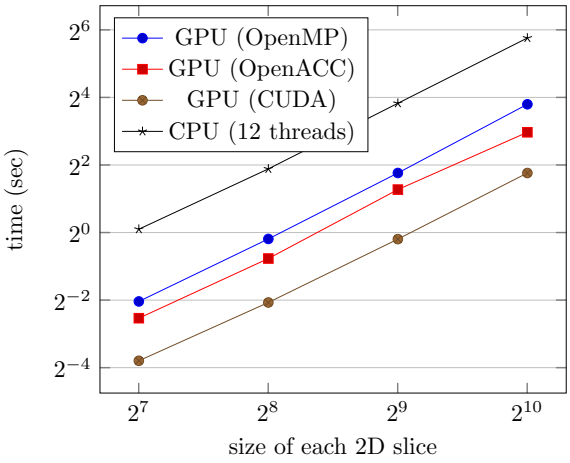

Figure 1: Performance of different parallelization approaches on the CPU and the GPU individually, for increasing problem size and using `doubles`. We define the size as $S \times S \times 64$ for 1024 iterations, and vary S from 128 to 1024.

For our CUDA implementation of the stencil code, we perform a 3D decomposition of the data and execute a CUDA kernel on square 8×8 blocks of every 2D slice. In every timestep, we load the required submatrices from the device memory into the shared memory, also considering the halo update, and then we apply the two Laplacians, eq. (3), locally. After the shared memory buffers are loaded, as well as after the first application of the Laplacian, local synchronization of the threads in a block is required. This is performed by calling `__synctreads()`. In the end of every time-loop, before proceeding to the next iteration, we add an extra barrier, i.e. `cudaDeviceSynchronize()`. After the time-loop is completed, we load the data from the GPU memory back to RAM.

2.3 Overlapping Computations

Due to the nature of the physics of the atmosphere, the stencil computations at each z-level are independent of other levels, and this lends itself well to a simple decomposition of z-levels between the CPU and GPU. Hence, during the computation of the stencil, a portion of the z-levels are offloaded to the GPU, while the rest remain on the CPU. It is important to note that while data transfer speeds

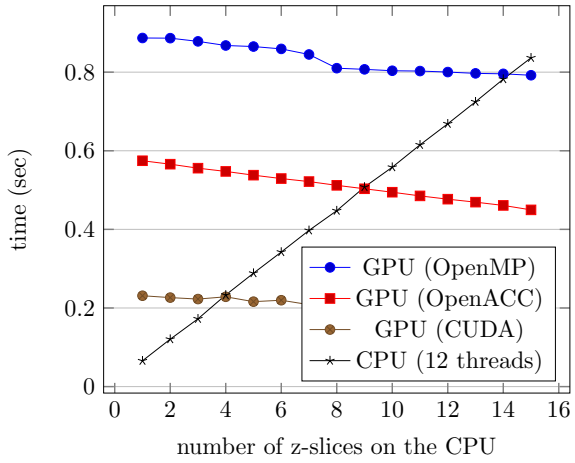

Figure 2: Performance of the CPU-only and GPU-only code for increasing number of z-slices on the CPU and decreasing number of z-slices on the GPU, for fixed problem size $S = 256$, and for total number of z-slices equal to 64.

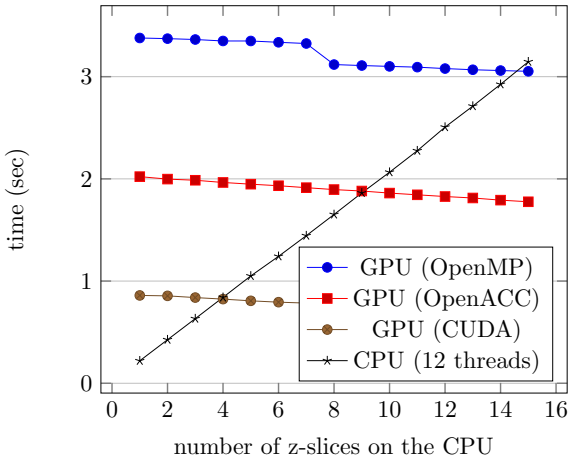

Figure 3: Performance of the CPU-only and GPU-only code for increasing number of z-slices on the CPU and decreasing number of z-slices on the GPU, for fixed problem size $S = 512$, and for total number of z-slices equal to 64.

on the GPU are extremely fast, moving data to and from the CPU to the GPU is extremely expensive, and as such must be minimized. Therefore, once the necessary data is transferred to the GPU, it is only copied back to the CPU *after* at the end of the iterations. Since data at independent z-levels do not need information from other z-levels, even across timesteps, this leads to no dependency issues. It should be mentioned, that at the end of every iteration we synchronize GPU and CPU. In a distributed-memory implementation, inter-node communication would take place there.

3 Results

In this section, we examine the performance of our CPU-only, GPU-only and hybrid CPU-GPU stencil code implementations based on different programming models, i.e. OpenACC, OpenMP, and CUDA. All the numerical experiments were performed on Piz Daint, which has the following specifications; each hybrid computational node consists of 64GB RAM, an Intel® Xeon® E5-2690 v3 @ 2.60GHz (12 cores), as well as an NVIDIA® Tesla® P100 with 16GB. Our CPU implementation utilizes all 12 cores of the node using OpenMP. Our GPU implementations are based on either directive-based programming models, i.e. OpenACC and OpenMP, or CUDA. In all of our experiments we use double-precision arithmetic, and we execute the stencil application for 1024 iterations/timesteps.

Below we give the compilers along with the respective compiler flags we used for each one of our implementations:

- CPU (OpenMP): GCC 8.3.0 using `-O3 -ffast-math -funroll-loops -fopenmp`
- GPU (OpenMP): Cray Fortran using `-O3 -hfp2 -ra`
- GPU (OpenACC): PGI C++ using `-O3 -fast -ta=tesla,cc60`
- GPU (CUDA): nvcc 7.0.1 using `-arch=sm_60`

In fig. 1, we present the performance of our stencil code implementations on the CPU and GPU, individually, for fixed number of slices in the z-direction, i.e. $nz = 64$, and increasing sizes of the 2D slices.

It can be observed, that the CUDA GPU-only implementation of the stencil code achieves the highest speed-up ($\sim$ 16x) versus the CPU-only multi-threaded implementation. Furthermore, it can be seen that a significant speedup ($\sim$4x) can be obtained by using either of the two directive-based GPU acceleration paradigms, which involve much less effort to deploy.

In figs. 2 to 4, we show the performance of our CPU-only and GPU-only implementations for

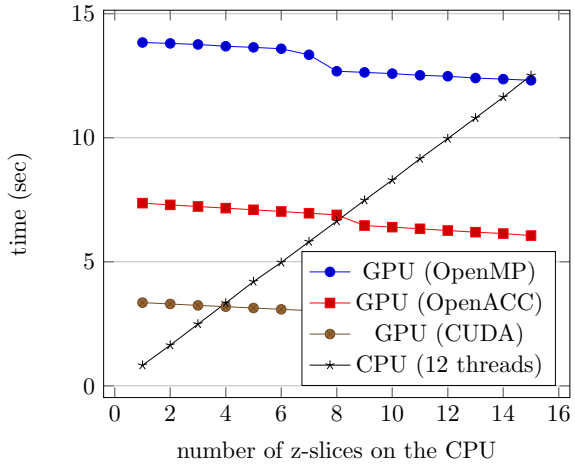

Figure 4: Performance of the CPU-only and GPU-only code for increasing number of z-slices on the CPU and decreasing number of z-slices on the GPU, for fixed problem size $S = 1024$, and for total number of z-slices equal to 64.

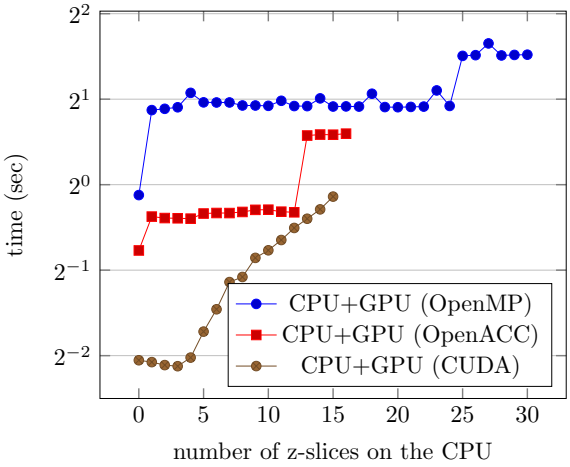

Figure 5: Performance of the hybrid CPU/GPU code for fixed problem size $S = 256$ and increasing offloaded work on the CPU.

increasing number of z-slices on the CPU (starting from 1) and decreasing number of z-slices on the GPU (starting from $64 - 1 = 63$), while keeping fixed the sizes of the 2D slices. Specifically, we start with 1 slice on the CPU and 63 on the GPU, then we proceed with 2 slices on the CPU and 62 slices on the GPU, and so on. The point where the performance lines of the CPU-only and GPU-only codes intersect, gives an estimate of the maximum number of z-slices that can be offloaded on the CPU in order to achieve maximum performance of the hybrid CPU-GPU implementations.

The figs. 2 to 4, indicate that the CUDA implementation is already very fast, and only some minor speedup can be achieved by offloading up to 3 slices to the CPU. Offloading more than 3 slices would lead to a performance drop since the CPU would become the bottleneck for the hybrid CPU/GPU implementation. However, for the OpenACC and OpenMP implementations, it seems that there is more to gain by offloading part of the computation to the CPU. The OpenACC and OpenMP implementations should be able to achieve a speed-up by offloading $\sim$ 8 and $\sim$ 15 slices, respectively. This can simply be attributed to the fact that the codes accelerated using directives are not as optimized as manually optimised CUDA code.

In figs. 5 to 7, we present the performance of our hybrid CPU-GPU implementations for increasing number of z-slices on the CPU, and for fixed sizes of the 2D slices.

We can see that only in the case of the CUDA implementation do we actually get a speed-up by offloading some slices to the GPU. This speed-up also agrees well with the results from figs. 5 to 7, with optimal performance being reached for the case with 3 slices offloaded to the CPU.

On the other hand, for both the OpenMP and the OpenACC implementations, no performance is gained by having the CPU do part of the computation. In both, we observe an initial jump in runtime after offloading a single slice to the CPU. The reason for this behaviour is not immediately obvious, however, since the runtime remains approximately constant thereafter for a while, it is possible that this is the result of overheads from spawning new threads inside the OpenMP `parallel` region. In both of these versions another jump in runtime is also observed approximately the 10th and 25th offloaded z-slice respectively. It is not completely clear what causes this jump. It is possible that at this level the size of the offloaded arrays fully populate the cache of the CPU leading to a drop in performance, however, these phenomena needs to be investigated further.

4 Conclusion

In our project, we investigated how stencil code implementations based on different GPU programming paradigms, i.e. OpenMP, OpenACC,

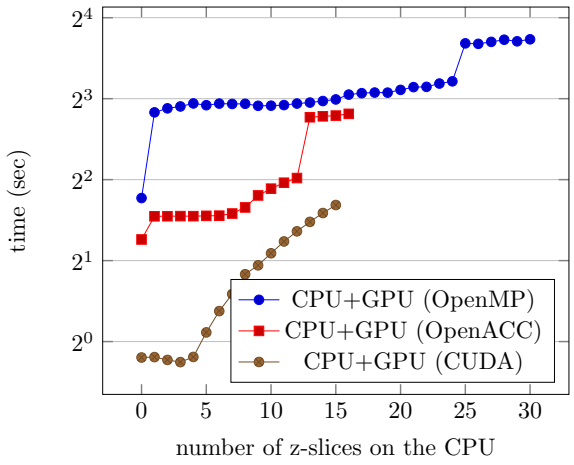

Figure 6: Performance of the hybrid CPU/GPU code for fixed problem size $S = 512$ and increasing offloaded work on the CPU.

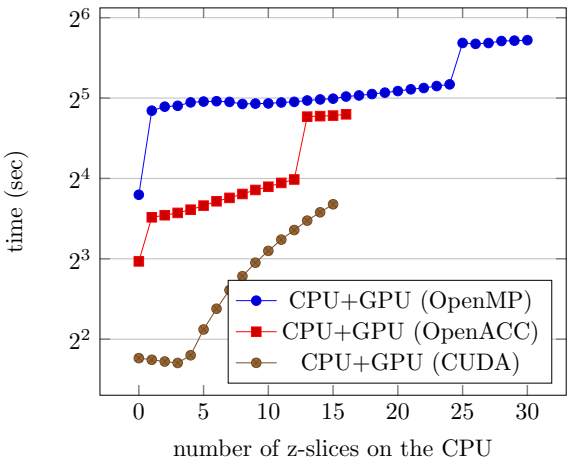

Figure 7: Performance of the hybrid CPU/GPU code for fixed problem size $S = 1024$ and increasing offloaded work on the CPU.

and CUDA compare to each other in terms of performance. Moreover, we compared these implementations against a parallel stencil code implementation targeted for multicore CPUs, and examined if running computations on the CPU and GPU concurrently can boost performance. We found that due to the difference in performance between the OpenMP, OpenACC and CUDA GPU implementations, the amount of computations that has to be offloaded to the CPU, in order to improve performance, varies greatly. In fact, while implementing the algorithm in CUDA is not straightforward, the hand-optimized version exhibits excellent performance, to the point where very little slices can be kept on the CPU without causing slowdown. Meanwhile, the OpenMP and OpenACC version reach this point much later. While in theory, and also according to (figs. 2 to 4), there is scope for speed-up on all three versions, and even more so on the the OpenMP and OpenACC versions, due to their GPU implementations not being as optimised as the CUDA version, we see no performance gain in these cases.

References

[1] M. Papadrakakis, G. Stavroulakis, and A. Karatarakis. A new era in scientific computing: Domain decomposition methods in hybrid CPU–GPU architectures. *Computer Methods in Applied Mechanics and Engineering*, 200(13):1490 – 1508, 2011.

[2] M. Xue. High-Order Monotonic Numerical Diffusion and Smoothing. *Monthly Weather Review*, 128(8):2853–2864, 08 2000.