Software Development Case Study: The Acceleration of a Distributed Application using GPUs

Martin Kuhnel
martinkuhnel@chevron.com
Chevron
Houston, Texas, USA

Alex Loddoch
loddoch@chevron.com
Chevron
Houston, Texas, USA

Tao Sun
tao.sun@chevron.com
Chevron
Houston, Texas, USA

ABSTRACT

We present a practical approach for the acceleration of an industrial and scientific application using graphics processing units (GPUs). Our original application is a computational stratigraphy codebase that couples fluid flow and sediment deposition submodels. The application uses domain decomposition and a halo exchange to split the workload among multiple workers in a distributed system. Our methodology abstracts and conserves the host data structures while re-writing computational elements in the GPU programming language CUDA. Utilizing high performance GPU machines in the Azure cloud, we show a minimum 90x speedup compared to a high-end CPU based cluster. In this poster, we give a brief description of the original algorithm, followed by a discussion of required software changes and additions. Although this case study focuses on a specific example, we hope this approach inspires similar efforts in other applications.

CCS CONCEPTS

• **Software and its engineering → Parallel programming languages**; **Software development methods**; • **General and reference → Performance**; • **Hardware → Hardware accelerators**; • **Computing methodologies → Parallel algorithms**; **Massively parallel and high-performance simulations**.

KEYWORDS

Software development, simulation, performance, acceleration, domain decomposition, GPU, MPI, CUDA, NVLink

1 POSTER SUMMARY

1.1 Starting Point

Our goal is to accelerate a proprietary geology modelling application termed CompStrat [1][4]. The application models the flow of sediment-carrying water through a basin over a large time frame. The flow of this water erodes and deposits different types of sediments over the course of the simulation, creating stratigraphic layers throughout the basin. The current algorithmic approach uses a finite volume spatial discretization scheme, usually in the form of a Cartesian grid, computing the flow and sediment fluxes while updating a column of deposited sediments for each cell. This process is repeated for upwards of ten million iterations with a variable time step.

Since each cells' computation is independent of the others, the flux calculations can be done in parallel as long as information about neighbors is kept up to date among workers. The original parallelization scheme uses MPI to perform domain decomposition, splitting the grid among multiple ranks, and to perform the halo exchange and synchronization of the workers. The application in this state is compute-bound, as the flux calculations can be computationally difficult and must be repeated for each cell, edge, and sediment type. The goal of our work is to run this workload on GPUs which offer more parallelism and can accelerate the large computational bottleneck.

This document is intended only for use by Chevron for presentation at the SC23 conference. No portion of this document may be copied, displayed, distributed, reproduced, published, sold, licensed, downloaded, or used to create a derivative work, unless the use has been specifically authorized by Chevron in writing.

1.2 Past Software Development Hurdles

Running this complex workload, mostly consisting of C++ code, on NVIDIA GPUs requires porting the computation to CUDA kernels and managing GPU memory [3]. This effort requires a good understanding of the existing data structures, computation, and message passing systems. Two past approaches provide insights into some of the hurdles this development encounters.

One approach is to rewrite the application to be entirely GPU native. This approach can potentially generate the most performant code, but is not viable from a time-to-production standpoint. Not only does it require more code to be written, it is also difficult to test against the original code and requires more work to be compatible with original input and output standards. Additionally, any new developments made in the original code will have to be added to the new replacement.

The second approach is to kernel swap the computationally intense flux calculations. While this approach conserves a large portion of the original application and is easy to unit test, it suffers in performance due to the excessive device copies. Each iteration requires moving data to the GPU for compute and back to the host for bookkeeping. This approach benefits from being "in-place" and preserving the original code structure, but needs to be expanded to be more GPU native.

1.3 GPU Acceleration Approach

Our final approach is a compromise between these two strategies. In order to preserve the reliability and compatibility benefits of the in-place approach, we make sure to keep all setup, input and output, and CPU data structures constant during development. The GPU intakes all necessary data into its own data structures which are abstracted from the existing host structures. The GPU will be able to run the entire compute heavy simulation with minimal communication to the host and we can take full advantage of its parallelism. This also allows for faster and easier unit tests: as kernels are developed and integrated, the code can be directly compared to the original algorithm.

1.4 Domain Mapping

One crucial development to maintain compatibility with the original data structures is abstracting the domain and its decomposition. Our goal is to allow for radical domain changes to be integrated using C++ in the original codebase without the need to change the new GPU code. Before moving data to the GPU at the start of the simulation, these index maps are generated by looping through the host domain. Information for each cell such as the location of its neighbors is parsed from the CPU structures and appended to arrays that will be copied to the GPU. Therefore, the GPU no longer concerns itself with the shape or decomposition of the domain. Instead the GPU works in directly indexed arrays that are mapped to one another. A GPU kernel can simply act in parallel on all of the active cells and retrieve the state, location, and neighbors of those cells from larger arrays that contain the data of all cells.

1.5 Memory Buffer

While the simulation remains GPU native for the entirety of the computational work, it must deal with the limitation posed by GPU memory. GPU cards have far less memory available than the original CPU nodes and therefore cannot contain the entire sediment grid, which grows as material is deposited. To solve this issue, the GPU only works on a buffer of a fixed size containing anywhere from ten to ten thousand of the top layers in our sediment columns. When the buffer either fills up or bottoms out, the buffer is copied back to the host and reset to a moving midpoint which minimizes the amount of synchronizations that need to occur. The fixed-size nature of the buffer ensures that it will fit within the GPU memory throughout the entire simulation and allows the size to be changed depending on which GPU generation is being used.

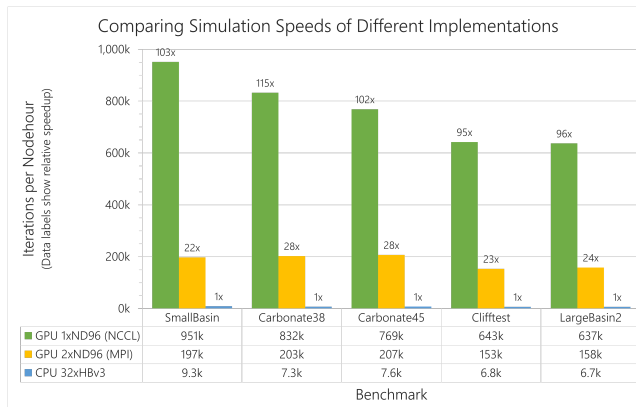
1.6 Halo Exchange

Similar to the original MPI code, it is beneficial to split the work among multiple GPUs. With this in-place approach, we assign one MPI rank to one available GPU, using the PCI bus topology to make the optimal selections. This ensures that each GPU has its own domain section to work on and conserves the underlying decomposition. We can also use the MPI structure of the original CPU code by copying the data in the halo back to the host where it is packed and transferred using MPI. However, this introduces a large bottleneck with many data copies.

Depending on the hardware available, the halo exchange can be accelerated with the use of NVLink, which we use via the NCCL library to perform GPU-to-GPU transfers [2]. To fit within the abstract GPU domain, a new index map is introduced to map overlapping halos on the GPU to its neighboring devices. The NCCL library then allows us to piggy-back off of the existing MPI communicator to create a NCCL communicator that is aware of the rank to GPU mapping. The new NCCL sends and receives can use the NVLink fabric present on our 8-way machines, greatly reducing the communication time. These two approaches give us the flexibility to make optimal use of various types of hardware while keeping the original MPI system untouched.

1.7 Results

Throughout this development process, we performed benchmarks of various sizes to understand our computational speedup. The results for five internal production-scale benchmarks are shown in Figure 1. The new GPU code with NCCL runs 95 to 115 times faster than the CPU code running on a distributed cluster. Switching the halo exchange to MPI allows us to run benchmarks on two GPU machines, but the communication bottleneck creates a substantial slowdown.



	SmallBasin	Carbonate38	Carbonate45	Clifftest	LargeBasin2
GPU 1xND96 (NCCL)	951k	832k	769k	643k	637k
GPU 2xND96 (MPI)	197k	203k	207k	153k	158k
CPU 32xHBv3	9.3k	7.3k	7.6k	6.8k	6.7k

Figure 1: The speed of the simulation given in iterations per hour of computation on a given SKU. These results are from five production-scale benchmarks with five million iterations and meshes on the order of one million cells. The CPU code ran on 32 Azure HBv3 nodes with 120 AMD cores and the GPU codes ran on the Azure ND96 SKUs with 8 A100 GPUs.

This speedup outweighs the extra cost of the more expensive GPU nodes in Azure, allowing us to capture substantial compute cost savings. Additionally, the speedup allows for much faster workflows, less wait time, and higher resolution outputs. Longer simulations originally taking multiple weeks on upwards of 40 CPU nodes can now comfortably be run over the weekend on a single GPU node.

1.8 Future Work

One issue that has surfaced in our testing is a lack of memory on the GPU nodes. The original algorithm is comfortably split among many CPU nodes, distributing the memory load. As shown by our two node benchmarks, the GPU implementation is best run on a single node, which now has to store the entire sediment grid in memory. Therefore, it is necessary for us to find and implement a new memory management system such as offloading memory to disk storage.

Other future work will include implementation and benchmarking on the new H100 GPUs and continued optimization of the GPU workflow. This approach can also be applied to other similar applications, further improving the efficiency of our compute environment.

ACKNOWLEDGMENTS

To RJ Souza, for the work on the MPI halo overlap code and to Kevin Baker, for the help with understanding and managing the original code.

REFERENCES

[1] Lisa Goggin Maisha Amaru, Tao Sun and Ashley Harris. 2017. Integration of computational stratigraphy models and seismic data for subsurface characterization. *The Leading Edge* 36, 11 (November 2017). https://doi.org/10.1190/tle36110947a1.1

[2] NVIDIA [n. d.]. *NVIDIA Collective Communication Library (NCCL) Documentation*. NVIDIA. https://docs.nvidia.com/deeplearning/nccl/user-guide/docs.

[3] NVIDIA 2023. *CUDA Toolkit Documentation 12.2 Update 1*. NVIDIA. https://docs.nvidia.com/cuda/.

[4] Tao Sun. [n. d.]. Integrated Reservoir Characterization and Modeling with Computational Stratigraphy. In *AAPG Distinguished Lecture Series, 2022-23 Season*. https://www.aapg.org/career/training/in-person/distinguished-lecture/presentation/articleid/64181?utm_medium=website&utm_source=Right4Pane