Procedia Computer Science
Volume 51, 2015, Pages 1494–1503
ICCS 2015 International Conference On Computational Science

Multi-GPU Implementations of Parallel 3D Sweeping Algorithms with Application to Geological Folding

Ezhilmathi Krishnasamy[1,2], Mohammed Sourouri[2,3], and Xing Cai[2,3]

[1] Department of Engineering and Management, Linköping University, SE-58183, Linköping, Sweden
ezhkr601@student.liu.se
[2] Simula Research Laboratory, P.O. Box 134, 1325 Lysaker, Norway
mohamso@simula.no, xingca@simula.no
[3] Department of Informatics, University of Oslo, P.O. Box 1080 Blindern, 0316 Oslo, Norway

Abstract

This paper studies the CUDA programming challenges with using multiple GPUs inside a single machine to carry out plane-by-plane updates in parallel 3D sweeping algorithms. In particular, care must be taken to mask the overhead of various data movements between the GPUs. Multiple OpenMP threads on the CPU side should be combined multiple CUDA streams per GPU to hide the data transfer cost related to the halo computation on each 2D plane. Moreover, the technique of peer-to-peer data motion can be used to reduce the impact of 3D volumetric data shuffles that have to be done between mandatory changes of the grid partitioning. We have investigated the performance improvement of 2- and 4-GPU implementations that are applicable to 3D anisotropic front propagation computations related to geological folding. In comparison with a straightforward multi-GPU implementation, the overall performance improvement due to masking of data movements on four GPUs of the Fermi architecture was 23%. The corresponding improvement obtained on four Kepler GPUs was 47%.

Keywords: NVIDIA GPU, CUDA programming, OpenMP, 3D sweeping, anisotropic front propagation

1 Introduction

Motivated by higher energy efficiency, a new trend with high-end computing platforms is the adoption of multiple hardware accelerators such as general-purpose GPUs and many-integrated-core coprocessors per compute node. The most prominent example is Tianhe-2, which is the current No. 1 system on the TOP500 list [1]. Three Xeon Phi coprocessors can be found in each of Tianhe-2's 16,000 compute nodes. With respect to clusters that have multi-GPU nodes, the TSUBAME 2.5 system is known for having three NVIDIA Tesla K20x GPUs per compute node.

Together with this new hardware configuration trend, there comes the challenge of programming. In addition to properly offloading portions of a computation to the individual

Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2015

doi:10.1016/j.procs.2015.05.339

accelerators to achieve higher computing speed, a new issue arises regarding accelerator-to-accelerator data transfers. Although MPI communication should be used between accelerators residing on different compute nodes, intra-node data transfers between cohabitant accelerators have the possibility of using low-level APIs that incur less overhead than the MPI counterpart.

The hardware context of this paper is using multiple NVIDIA GPUs within the same compute node. As the computational domain, we choose to study parallel 3D sweeping algorithms, which have a causality constraint in that 2D planes have to updated one by one in sequence, where parallelism exists between mesh points lying on the same 2D plane. Following our previous work in the simpler scenario of 3D stencil computations [2], we want to investigate how to apply multiple CUDA streams, multiple OpenMP threads and NVIDIA GPUDirect [3] to 3D sweeping algorithms for masking the overhead of GPU-to-GPU data transfers. Another objective of this paper is to quantify the actual performance gain of using multiple GPUs for simulating 3D anisotropic front propagation, compared with using a single GPU [4]. To our knowledge, previous work on using GPUs to simulate anisotropic front propagation all targeted single-GPU platforms, such as [5, 6, 7]. Therefore, this paper is also novel with respect to using multiple GPUs for this particular computational problem.

2 Parallel 3D sweeping algorithms

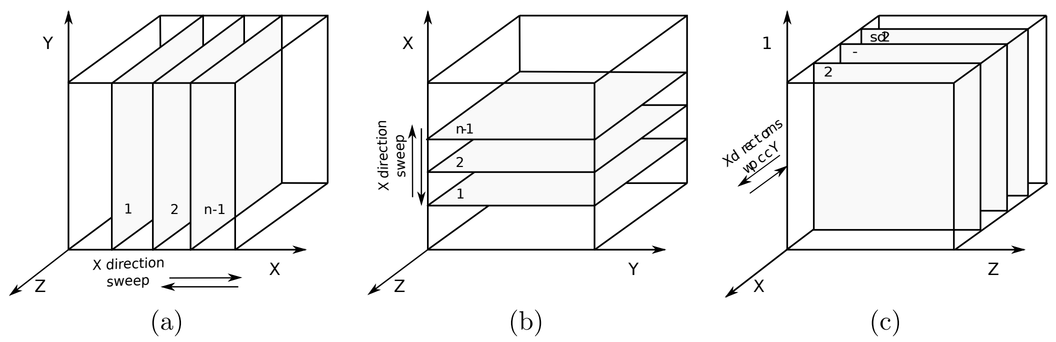


Figure 1: The 2D planes involved in the six sub-sweeps that constitute a parallel 3D sweeping algorithm.

In this paper, we consider parallel 3D sweeping algorithms that use a Cartesian grid, which is of dimension $(n_x + 2) \times (n_y + 2) \times (n_z + 2)$. The mesh point values, denoted by $T_{i,j,k}$, are iteratively updated by sweeps each being made up of six sub-sweeps that alternate between the positive and negative x, y and z-directions. Each sub-sweep consists of plane-by-plane updates that move consecutively through the 3D mesh. Computations of the mesh points that lie on a 2D update plane are independent of each other, thus parallelizable. The following pseudocode shows one sub-sweep along the positive x-direction and another sub-sweep in the opposite direction (see Fig. 1(a)):

```
for $i = 2 \to n_x$ do
    for all $j \in [1, n_y]$ and $k \in [1, n_z]$ do
        Update $T_{i,j,k}$ using values $T_{i-1,j\pm a,k\pm b}$, $a \in \{0,1\}, b \in \{0,1\}$
    end for
end for
for $i = n_x - 1 \to 1$ do
```

```
    for all $j \in [1, n_y]$ and $k \in [1, n_z]$ do
        Update $T_{i,j,k}$ using values $T_{i+1,j\pm a,k\pm b}$, $a \in \{0,1\}, b \in \{0,1\}$
    end for
end for
```

The four other sub-sweeps (two in the y-direction and two in the z-directions) are similar. We also remark that the sub-sweeps update only the interior mesh points, i.e., $1 \leq i \leq n_x$, $1 \leq j \leq n_y$, $1 \leq k \leq n_z$. The boundary points assume known solution values. In the above pseudocode, updating a mesh point relies on 9 mesh points that lie on the preceding plane, see Fig. 2 for an example. A concrete application of such a sweeping algorithm can be to simulate anisotropic front propagation that is described by the following static Hamilton-Jacobi equation:

$$F\|\nabla T(\mathbf{x})\| + \psi\left(\mathbf{a} \cdot \nabla T(\mathbf{x})\right) = 1, \qquad (1)$$
$$\text{given } \; T = t_0 \; \text{ on } \Gamma_0,$$

where $T(\mathbf{x})$ can model the first-arrival time of a propagating front that originates from the initial surface Γ_0. When the viscosity solution of (1) is used to model geological folding, vector $\mathbf{a}$ marks the axial direction of the fold, with F and ψ being nonzero constants. For details about the mathematical model and the derivation of sweeping-based numerical schemes, we refer the reader to [7] and the references therein.

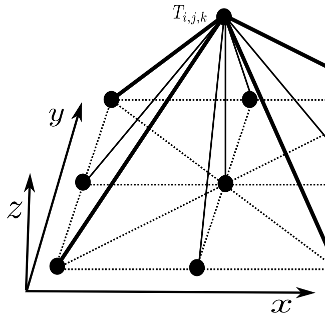


Figure 2: An example of data dependency associated with sub-sweeps along the z-direction.

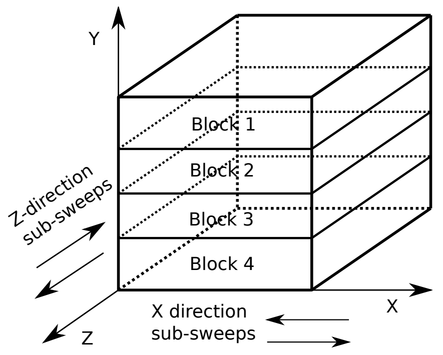


Figure 3: A partitioning of the 3D Cartesian grid that suits parallelization of sub-sweeps in both x and z-directions.

3 Parallelization

3.1 Mesh partitioning

Parallelism within parallel 3D sweeping algorithms exists among mesh points that lie on the same 2D update plane, but not across the planes. That is, parallelization can be realized by dividing each 2D plane among multiple computing hardware units, such as GPUs. Due to the rectangular shape of a 2D plane, it is natural to assign each GPU with a rectangular subdomain. Moreover, since each sub-sweep moves along a given spatial direction, from the top (or bottom) 2D plane to the bottom (or top) plane, all the 2D planes associated with one sub-sweep can use the same partitioning. This means that, for sub-sweeps along a specific spatial direction, the 3D Cartesian grid should be partitioned by cutting surfaces parallel with the sub-sweeping direction. However, we recall that the sub-sweeps alternate between the three spatial directions, thus there does not exist a universal partitioning that works for all x, y and z-directions. The best partitioning strategy is to let sub-sweeps of two spatial directions share one partitioning, switching to another partitioning for the third spatial direction. Fig. 3 shows a partitioning of the 3D Cartesian grid that can be used to parallelize sub-sweeps in both x and z-directions.

3.2 Data transfers

There are two types of data transfers that must be carried out between the GPUs. The first type of data movement happens on each 2D (yz or xz or xy) plane, for the purpose of communicating results of the halo computation (i.e., on mesh points that lie immediately beside a neighbor) from one subdomain to another. Consider for example the partitioning shown in Fig. 3, which can be shared by sub-sweeps in the x and z-directions. Then parallel computing on each yz-plane (inside a x-directional sub-sweep) will require one subdomain to send (and receive) n_z point values to (and from) each of its two neighbors. When updating each xy-plane (inside a z-directional sub-sweep), n_x point values are exchanged between a pair of neighboring subdomains. It should be noted that this halo-induced type of communication happens once per 2D plane.

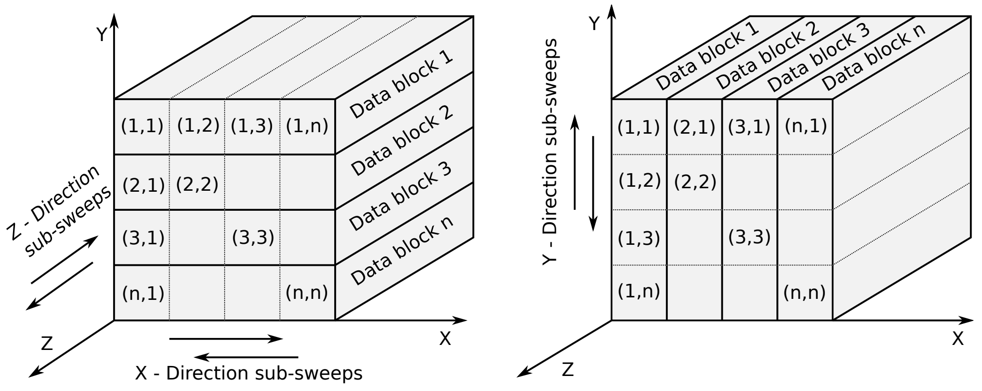


Figure 4: An example of volumetric data shuffle in connection with changing the grid partitioning.

The second type of data transfer is due to the need of switching the grid partitioning between some of the sub-sweeps. For example, the partitioning shown in Fig. 3 cannot be used for sub-sweeps along the y-direction. In connection with switching the grid partitioning, a 3D volumetric data shuffle among all GPUs is necessary. A concrete example of data shuffle involving four GPUs is shown in Fig. 4. There, each GPU can keep one fourth of its data, but has to exchange one fourth of its data with each of the other three GPUs. This volumetric data shuffle happens only twice per sweep (every six sub-sweeps). In other words, the second type of communication is considerably less frequent than the first type of halo-induced communication.

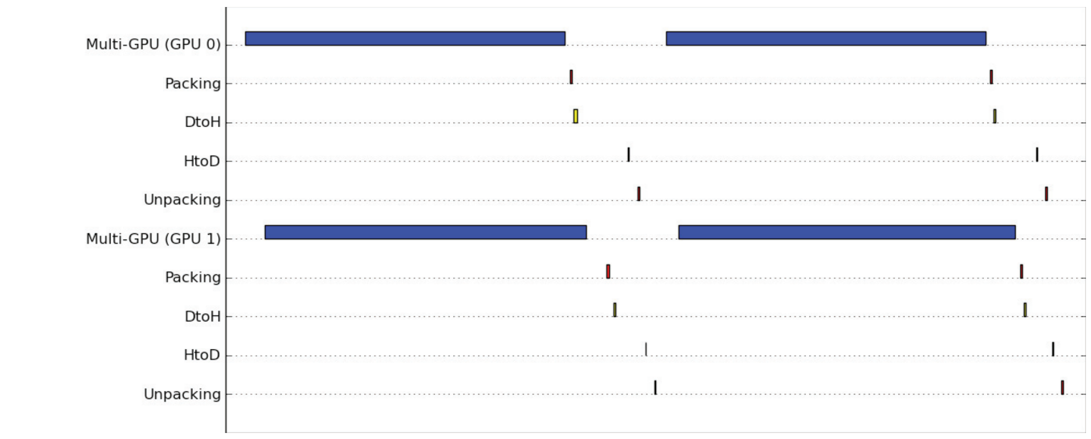
4 CUDA implementations

4.1 Plain multi-GPU implementation

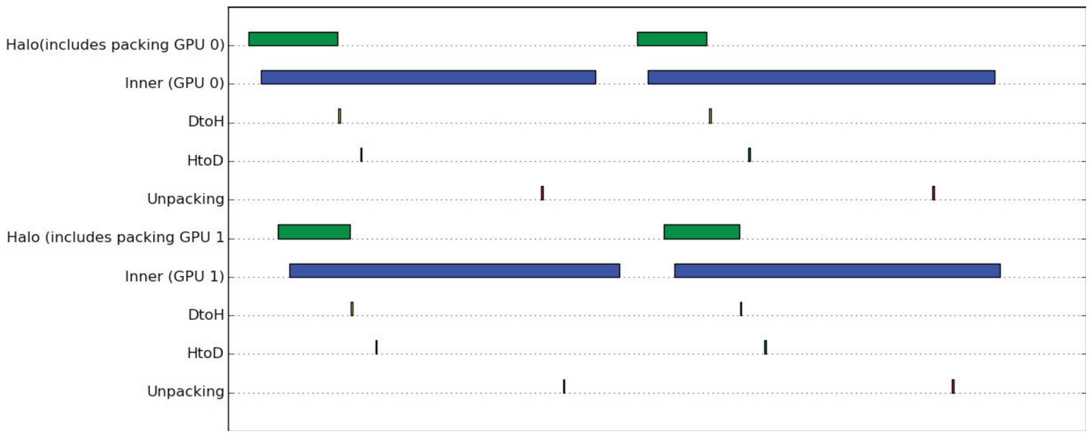
If there already exists a single-GPU CUDA implementation of a 3D sweep algorithm, it is a relatively simple programming task to write a plain implementation that uses multiple GPUs residing on the same compute node. Of the existing single-GPU code, its six kernels associated with the six different sub-sweeps can be reused by each GPU to work within its assigned subdomain of a 2D plane. Additional kernels have to be implemented to support the two types of data transfers: halo-induced communication and 3D volumetric data shuffle. The following pseudocode segment shows one sub-sweep in the positive x-direction for the case of using two GPUs:

```
for (i=2; i<=nx; i++) {
   cudaSetDevice(0);
   compute-kernel: update all points on GPU0's part of a yz-plane
   pack-kernel: fill a buffer containing halo values needed by GPU1

   cudaSetDevice(1);
   compute-kernel: update all points on GPU1's part of a yz-plane
   pack-kernel: fill a buffer containing halo values needed by GPU0

   cudaMemcpy (GPU0 buffer to host buffer H0)
   cudaMemcpy (GPU1 buffer to host buffer H1)
   cudaMemcpy (host buffer H0 to GPU1 buffer)
   cudaMemcpy (host buffer H1 to GPU0 buffer)

   cudaSetDevice(0);
   unpack-kernel: handle the incoming H1 data from host

   cudaSetDevice(1);
   unpack-kernel: handle the incoming H0 data from host
}
```

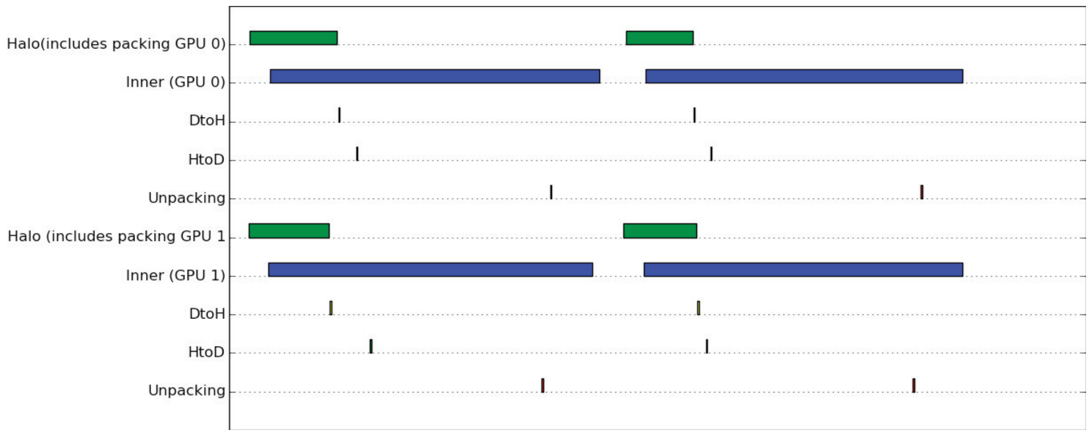
There are two problems with the plain implementation above. First, due to the default synchronous CUDA stream on each GPU, the halo-induced communication will not start until all the mesh points on the subdomain 2D plane are updated. There is thus no possibility of hiding the overhead of this communication, as shown in Fig. 5(a). The second problem is the use of synchronous data copies (`cudaMemcpy` for both device-host and host-device transfers), meaning that only one data transfer can happen at a time. The same problem also applies to the second type of communication: 3D volumetric data shuffles (not shown in the above code segment).



(a) Plain 2-GPU implementation: the default synchronous CUDA stream per GPU



(b) Improved 2-GPU implementation version 1: two CUDA streams per GPU



(c) Improved 2-GPU implementation version 2: two CUDA streams and one OpenMP thread per GPU

Figure 5: The timeline view (provided by `nvprof`) of two subsequent sub-sweeps in three 2-GPU implementations.

4.2 Improvement 1

The key to hiding the overhead of halo-induced communications is to overlap this type of data transfers with computations of the interior points (i.e., non-halo points) on each GPU. For this purpose, every data-packing kernel from the plain implementation is extended (as a halo-kernel) to compute its line of "halo points" as well as packing. Moreover, each GPU adopts at least two CUDA streams, one being responsible for updating its interior mesh points, the other ("halo stream") for independently executing the data-packing kernels. Consequently, asynchronous device-host and host-device data transfers (by calling `cudaMemcpyAsync`) can be enabled by using the halo streams. The following pseudocode implements this improvement, and the effect can be seen in Fig. 5(b) for the case of two GPUs.

```
for (i=2; i<=nx; i++) {
   cudaSetDevice(0);
   halo-kernel using halo_stream(0)
   compute-kernel over interior points using compute_stream(0)
   cudaMemcpyAsync (GPU0->H0) using halo_stream(0)

   cudaSetDevice(1);
   halo-kernel using halo_stream(1)
   compute-kernel over interior points using compute_stream(1)
   cudaMemcpyAsync (GPU1->H1) using halo_stream(1)

   cudaStreamSynchronize halo_stream(0)
   cudaStreamSynchronize halo_stream(1)

   cudaSetDevice(0);
   cudaMemcpyAsync (H0->GPU1) using halo_stream(1)
   unpack-kernel using halo_stream(1)

   cudaSetDevice(1);
   cudaMemcpyAsync (H1->GPU0) using halo_stream(0)
   unpack-kernel using halo_stream(0)

   cudaSetDevice(0);
   cudaStreamSynchronize compute_stream(0);
   cudaSetDevice(1);
   cudaStreamSynchronize compute_stream(1);
}
```

4.3 Improvement 2

The situation can be improved further. We note from Fig. 5(b) that the start of the kernels on GPU1 has a delay with respect to those on GPU0. This is because both GPUs are controlled by the same host CPU thread, which first initiates the kernels on GPU0 and then those on GPU1. This delay will become more severe when more GPUs are involved. To solve the above problems, we adopt the strategy of using multiple OpenMP threads on the CPU side, as proposed in [2]. That is, one controlling OpenMP thread is now in charge of each GPU. The entire code will thus be wrapped into an OpenMP parallel region, and the thread ID will dictate the responsibility of an OpenMP thread. The effect of this improvement is clearly visible in Fig. 5(c). The pseudocode is not shown due to page limit.

4.4 Improvement 3

As can be seen in Fig. 5(c), the combination of multiple CUDA streams and one controlling OpenMP thread per GPU can result in halo-induced data exchanges being carried out while computations on the interior mesh points proceed. The overhead of this type of communication can therefore be effectively hidden, even though the communication is relayed through the CPU host. For the second type of communication, i.e., 3D volumetric data shuffles between switches of the grid partitioning, relaying data via CPU is unnecessarily costly if there is hardware support for direct peer-to-peer (P2P) communication [3] between the GPUs. Specifically, to draw benefit from P2P and enable bi-directional data transfers, the `cudaMemcpyPeerAsync` function should be simultaneously called by the controlling OpenMP threads. Afterwards, the CUDA streams that execute the asynchronous P2P communication must be properly synchronized, via `cudaStreamSynchronize` called by the multiple controlling CPU threads, to make sure that the shuffled data has arrived.

Finally, we remark that the above three improvements were first discussed in [8]. However, the multi-GPU implementations used in this paper have made substantial adjustments (and corrections) of those in [8].

5 Experiments and measurements

5.1 Hardware platforms

We tested our multi-GPU implementations on two GPU clusters, *Erik* [9] and *Zorn* [10], for running 3D simulations of anisotropic front propagation. In particular, one 4-GPU node on the Erik cluster was used, where each GPU is of type NVIDIA Tesla K20m. The CPU host has dual-socket 8-core Intel Xeon E5-2650 2.0 GHz processors. It is important to notice that the four GPUs are organized as two PCIe "islands", meaning that the GPU0-GPU1 and GPU2-GPU3 pairs have a direct PCIe connection in between, whereas across-pair traffic is subject to a slower speed. On the Zorn cluster, we used one of its 4-GPU nodes where each GPU is of type Tesla C2050 and the CPU host consists of dual-socket 4-core Intel Xeon E5620 2.4 GHz processors. The four GPUs are also organized as two pairs with intra-pair PCIe connection. CUDA v5.5 was used on both platforms.

5.2 An example of geological folding

To compare with the single-GPU implementation from [4], we have chosen to simulate the same example of geological folding. More specifically, $F = 1.1$, $\psi = 1.0$ and $\mathbf{a} = (-0.34, 0.4, 0.7)$ were used in (1). The 3D spatial domain has length 10 in all three directions. The initial surface Γ_0 is shown in the left plot of Fig. 6, whereas the simulation result is depicted in the right plot. (The numerical results from the multi-GPU implementations were all verified by those produced by the original single-GPU implementation.)

5.3 Time measurements

Table 1 summarizes the time measurements that were obtained on the Erik and Zorn systems. Two problem sizes were tested: $n_x = n_y = n_z = 512$ and $n_x = n_y = n_z = 640$. Each simulation ran 8 sweeps, i.e., 48 sub-sweeps in total. All computations used double precision. Recall from Section 4 that the first improvement to the plain multi-GPU implementation is to let each GPU use multiple CUDA streams. That is, the number of CUDA streams per GPU equals the

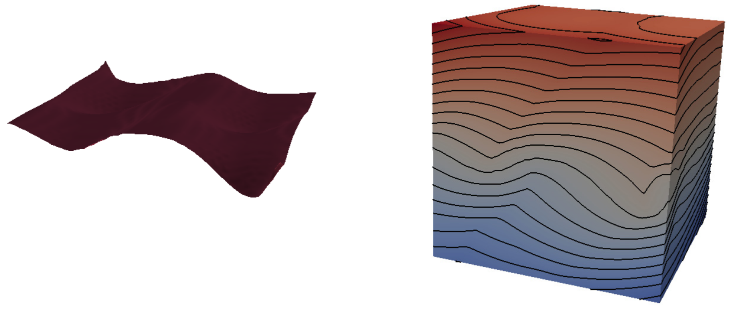


Figure 6: The initial surface Γ_0 (left plot) and the simulation result of (1) after running 8 sweeps.

number of neighbors plus one. Halo computations are thus carried out as early as possible, and the data transfers (type 1 communication) between neighboring GPUs can be carried out while computations on the interior mesh points proceed. As can be seen in Table 1, the benefit of the first improvement is more obvious for the 4-GPU cases. Likely, the second improvement (assigning one controlling OpenMP thread per GPU) also has a clearer advantage for the 4-GPU cases. For the third improvement, using P2P data communication for the 3D volumetric data shuffles (type 2 communication) instead of relaying data via the CPU, the benefit is more profound for the 2-GPU cases. This is due to the fact that GPU0 and GPU1 have a direct PCIe connection that provides hardware support for P2P. When four GPUs are used, although GPU2 and GPU3 also form a pair with direct PCIe connection, data transfers across the two pairs (e.g. GPU0-GPU2) still have to relay through the CPU, thus not fully enjoying the performance benefit of P2P data communication.

Grid size	$n_x = n_y = n_z = 512$		$n_x = n_y = n_z = 640$	
	Time on Erik	Time on Zorn	Time on Erik	Time on Zorn
Single-GPU	31.71	64.95	59.54	123.34
2-GPU plain impl.	24.75	44.50	45.05	82.14
2-GPU improved v1	24.35	41.40	44.73	81.36
2-GPU improved v2	22.39	40.38	39.58	75.42
2-GPU improved v3	19.37	37.40	35.34	69.64
4-GPU plain impl.	23.28	30.77	41.03	54.61
4-GPU improved v1	20.68	28.54	38.65	51.56
4-GPU improved v2	14.31	25.54	25.18	46.83
4-GPU improved v3	12.48	23.57	21.91	43.02

Table 1: Time measurements (in seconds) of running eight sweeps of various multi-GPU implementations.

6 Concluding remarks

In comparison with single-GPU implementations of parallel 3D sweeping algorithms, the use of multiple GPUs introduces the complicating issue of having to switch between two grid partitionings and the resulting 3D volumetric data shuffles among all the GPUs. These come on top of the conventional halo-induced data exchanges between neighboring GPUs. In other words, parallelizing a 3D sweeping algorithm is more difficult than parallelizing a regular 3D finite difference method. The achievable parallel efficiency will consequently be lower due to the costly 3D volumetric data shuffles. Nevertheless, our time measurements have shown that with a proper use of multiple multiple CUDA streams per GPU, in combination with adopting one controlling OpenMP thread per GPU and P2P data communication offered by NVIDIA GPUDirect, we can secure satisfactory parallel efficiency. This means that using multiple GPUs for performing parallel 3D sweeps is a viable technical solution, especially considering the benefit of aggregating the device memory of the GPUs to solve larger problems that exceed the memory limit of a single GPU.

As future work, the current work can be extended to the scenario of multiple compute nodes, each with one or several GPUs. Asynchronous CUDA memory copies have to be replaced with suitable non-blocking MPI calls. Then, really huge-scale simulations can be made possible.

Acknowledgments

The work of the first author was supported by an Erasmus grant. The second author was funded by FRINATEK grant No. 214113/F20 from the Research Council of Norway. The simulations were performed on resources provided by the Swedish National Infrastructure for Computing (SNIC) at LUNARC and PDC. The Abel system within Norway's NOTUR facilities was used for testing purposes.

References

[1] TOP500 Supercomputing Sites.
URL `http://top500.org/lists/2014/11/`

[2] M. Sourouri, T. Gillberg, S. B. Baden, X. Cai, Effective multi-GPU communication using multiple CUDA streams and threads, in: Proceedings of 20th International Conference on Parallel and Distributed Systems (ICPADS 2014), IEEE, 2014.

[3] NVIDIA GPUDirect.
URL `https://developer.nvidia.com/gpudirect`

[4] T. Gillberg, M. Sourouri, X. Cai, A new parallel 3D front propagation algorithm for fast simulation of geological folds, Procedia Computer Science 9 (2012) 947–955.

[5] W.-K. Jeong, R. T. Whitaker, A fast iterative method for eikonal equations, SIAM Journal on Scientific Computing 30 (5) (2008) 2512–2534.

[6] O. Weber, Y. S. Devir, A. M. Bronstein, M. M. Bronstein, R. Kimmel, Parallel algorithms for approximation of distance maps on parametric surfaces, ACM Transactions on Graphics 27 (4) (2008) 1–16.

[7] T. Gillberg, Fast and accurate front propagation for simulation of geological folds, Ph.D. thesis, University of Oslo, Norway (2013).

[8] E. Krishnasamy, Hybrid CPU-GPU parallel simulations of 3D front propagation, Master's thesis, Linköping University, Sweden (2014).

[9] The Erik GPU cluster at LUNARC.
URL `http://www.lunarc.lu.se/Systems/ErikDetails`

[10] An overview of Zorn, PDC's GPU cluster.
URL `https://www.pdc.kth.se/resources/computers/zorn`