The development of the LRnLA algorithm for cross stencil numerical schemes for the CUDAGPU model of parallelism

Anastasia Perepelkina and Vadim Levchenko
Keldysh Institute of Applied Mathematics RAS, Miusskaya sq. 4, Moscow, Russia
E-mail: mogmi@narod.ru

Abstract. This document describes the recent progress in the constructing of a 3D and time (3D1T) spatial and temporal blocking algorithm for cross stencil numerical scheme implementation. This method serves the purpose of raising the operational intensity of the memory-bound problems and increasing the performance limit at least several times compared to the algorithm without temporal blocking. The method of construction of LRnLA algorithms involves choosing the prism shape that subdivides the dependency graph of the problem into a series of tasks with prescribed dependencies between each other. The algorithm is adjusted to the GPU architecture, taking into account all available levels of parallelism. The proposed algorithm had been implemented and called DiamondCandy. The obtained efficiency in our current implementation is 59% of the limit estimated by theoretical analysis.

1. Introduction
The rapid advancement of the hierarchy of memory levels and parallelism options of the modern hybrid computers cause the demand for the development of new algorithms for high performance codes. This is extremely important in the codes for numerical simulation of physical phenomena. Typical evolutionary problems of electrodynamics, fluid dynamics, plasma physics and such store some big data array as an initial condition, and all values in it are updated several million times. This puts such kind of problems to the memory bound domain [1].

This work deals with one way of space-time blocking approach, the Locally Recursive non-Locally Asynchronous (LRnLA) algorithms [2, 3, 4]. Space and, more importantly, time blocking algorithms [5, 6] raise operational intensity, allow for more data reuse, and may increase the performance by several orders of magnitude. The search for optimal blocking shapes in 3D1T (3D with time) is one of the issues [7]. In simulation of hydrodynamics with the RKDG method [8], and in other important applications, many modern and robust schemes are based on the cross stencil. The method of algorithm construction should correspond to the scheme stencil [2], so for the cross stencil in 3D, the subdivision should be based on an octahedron. Thus the task of search for optimal 3D1T blocking shapes can not be split into 1D1T problems, leaving one parallelization method for each direction. The 3D1T blocking shape for cross stencils, called DiamondCandy, has been developed [4], however, its GPU implementation is not easy due to the overhead of a fully 3D1T blocking [9]. We plan to overcome this problem. In this paper we show the process of adaptation of the LRnLA algorithm DiamondCandy to GPU computing.
Figure 1. (a) Torre algorithm in 1D1T dependency graph. The prism base in 1D is a horizontal line segment. Torres are evaluated in the order from right to left. Inside the Torre the cells are updated stepwise. (b) DiamondCandy shape in 3D is constructed as a combination of a regular octahedron with two tetrahedra. It is a hexagon, so it may be subdivided into similar shapes. It is a Torre base in 3D1T.

2. **CUDA GPU specifics**

The NVidia GPU model may be simplified as follows. The computation is performed by the device. In it, the main storage of the simulation data is the global memory. The calculation is distributed between CUDA blocks. The block is assigned to an SM and is provided with an amount of shared memory. This amount is relatively small, but the access to shared memory is faster by an order of magnitude.

Inside one CUDA block, the calculation is distributed between several warps, which perform vector operations. However, the vector operations are usually not written in code. The CUDA programming model requires for the operations to be specified for one CUDA thread, one of the 32 elements of a warp. Each one is provided by an amount of memory in the fastest memory level, the register file.

This way, inside the GPU, the operations are parallelized between CUDA-blocks. Inside CUDA-blocks, they are parallelized between warps. In a warp, they are processed by several CUDA-threads. The data is localized in the device memory. In one CUDA-block, it may be exchanged between warps with the use of the shared memory. In a warp, the data in the registers of a CUDA thread may be exchanged with other CUDA threads with vector shuffle operations [10].

3. **Algorithm adaptation to a CUDA GPU**

In the traditional approach, for parallel processing, the domain may be decomposed spatially, and the decomposition approach may vary. But, in any case, an outer loop in time exists. Only after one time step is processed in all cells, the next update is started. This loop imposes a restriction: all data values have to be read and written at least once per one update in time of the whole region. We call this approach 'stepwise'.

Among the variety of LRnLA algorithm construction patterns [2], in the current work, we have chosen the slanted prisms (Torre, Figure 1(a)) [3] with a specific hexahedral shape in its base (DiamondCandy, Figure 1(b)) [4] for the dependency graph subdivision.

We demonstrate the algorithm construction with a second order finite difference scheme for the numerical solution of the 3D wave equation. The scheme uses 3 layers in time. To implement it, two copies of the data array are used. We call them \(f\) and \(g\) layers. For the \(f\) update the neighboring data from array \(g\) is used and vice versa.

The numerical scheme determines the amount of data per cell, that is, two floating point values, for \(f\) and for \(g\). For the algorithm construction, the concrete decomposition rules and the sizes of the prism base are chosen so that it fits the hardware specifics. Taking account of the basic CUDA GPU structure (Sec. 2), we have constructed the following algorithm.

To describe it, we start with a minimal element of construction. The smallest DiamondCandy, which can uniformly tile the whole domain, contains two mesh nodes. We choose a pair of cells adjacent to each other in \(z\)-axis direction, and call it an element (Figure 2(a)).
One Torre is assigned to a CUDA block. The base of the prism is a DiamondCandy with $N \times N \times N$ elements. Due to the hexahedral shape, DiamondCandies may be tiled or subdivided similar to a cube. The axes of the tiling are $\vec{e}_1 = (1, 0, 1)$, $\vec{e}_2 = (0, 1, 1)$, $\vec{e}_3 = (-1, -1, 0)$. Choosing $N = 16$ is convenient because $16^3 = 4096$ cells require 64 KB of memory, which fits the register file, and 16 is a power of 2.

One CUDA block updates $f$ inside one DiamondCandy of $16^3$ elements. All CUDA threads in this CUDA block are synchronized, and then the base shifts by 1 cell in $z$-axis direction and CUDA block updates the $g$ cells there. After the next shift by 1 cell, $f$ is updated and so on.

Among these elements, the computations are distributed as follows. One warp updates a layer of $16 \times 2 \times 16$ elements. One CUDA thread updates $2 \times 2 \times 4$ elements.

When $2 \times 2 \times 4$ pairs of values of $g$ are updated, $3 \times 3 \times 5$ pairs of $f$ values are required as the stencil dependencies. In the registers of one thread, we keep $2 \times 2 \times 4$ values of $f$ and $2 \times 2 \times 4$ values of $g$. $g$ values are shifted by 1 cell in $z$ direction from the $f$ values. The existing $f$ values are used in the stencil computation. Then we perform a series of data exchanges to get the remaining values. In the course of exchange, the existing $2 \times 2 \times 4$ $f$ values are replaced by $2 \times 2 \times 4$ $f$ values shifted by 2 cells in the $z$-axis direction. This corresponds to the starting position, with $f$ and $g$ exchanged. So, the computation continues in a similar way for the $f$ update. As a whole, the DiamondCandy base moves in the $z$ direction.

The shift by 2 cells in the $z$-axis direction is performed as a sum of shifts in the $\vec{e}_3$, $\vec{e}_1$, and $\vec{e}_2$ directions in that order.

Data exchange in a warp in the $\vec{e}_3$, and then in the $\vec{e}_1$ directions, is performed by vector shuffle operations. This way, the shared memory is not used for this exchange. $2 \times 2 \times 1$ elements are sent to the adjacent thread in the $-\vec{e}_3$ direction. $1 \times 2 \times 4$ elements are sent to the adjacent thread in the $-\vec{e}_1$ direction.

Data exchange between warps is performed through the shared memory. $16 \times 1 \times 16$ elements are saved at the beginning of the exchanges and are read by adjacent warps when necessary. Thus, the exchange in the $\vec{e}_2$ direction is performed.

Data exchange with global memory is performed once every 4 time steps: 2 updates of $f$ values and 2 updates of $g$ values in the Torre base. The data are loaded into the shared memory arrays, which are called ‘hoods’, since they closely encompass the Torre base at the top (Figure 2(c)). It is used for the elements on the boundary of the Torre base during the 3 exchanges in the $\vec{e}_3$, $\vec{e}_1$, and $\vec{e}_2$ directions. When
the data are used, the values which were updated in the current Torre, and are not to be updated in the current Torre anymore, are written in its place. At the end of the four time steps, these data are written into the global memory, and the data for the next stages are written into these arrays.

Here, the four time step stride is chosen for better data locality and coalescing. The L1 cache line size is 128 B, and this is the size of 8 elements. Thus, it is preferable to read the data in portions of 8 elements. For this reason, the special data structure has been proposed to contain the cell data. One tile is defined as $2 \times 2 \times 2$ elements of $g$ and $f$. These tiles are packed in a 3D array. Spatially, they are tiled in the $\vec{e}_1$, $\vec{e}_{II}$, and $\vec{e}_3$ directions (Figure 2(b)). The $\vec{e}_{II}$ direction is chosen so that the computable domain would be elongated along the Torre shift direction, which is parallel to the $z$-axis in this case.

Several Torres should be run to correctly cover the dependency graph between two synchronization instants. They are run by parallel CUDA-blocks (80 in NVidia Volta V100) with an additional synchronization to ensure the correct dependencies. The first Torre is run from the top boundary of the domain. After four time steps are processed in one Torre, the same four updates in time may be processed in its three neighbors. These neighbors are Torres, the bases of which are adjacent to the base of the current one in $-\vec{e}_1$, $-\vec{e}_2$, and $-\vec{e}_3$ directions. In the current implementation, external synchronization in the $\vec{e}_2$ direction is made by means of a sequential loop over the $\vec{e}_{II}$ axis from top to bottom. Thus, the Torres, the bases of which are tiled in the $\vec{e}_1$ and $\vec{e}_3$ directions, are started at once. The dependencies between them are ensured by a system of semaphores that are implemented manually, with an array of integers corresponding to each Torre base. When the four field updates are processed in a Torre, its semaphore value is incremented. The next four field updates are started only when the corresponding values of its neighbors in $-\vec{e}_3$ and $-\vec{e}_1$ directions are not less than its own value.

4. Performance Benchmark

The performance of the code implementation of the DiamondCandy algorithm for the wave equation has been tested on NVidia Volta V100. The performance estimation of it with the use of the Roofline [1] model (Figure 3) is performed with a procedure described in [2].

The operational intensities for the task of updating the whole domain by 1000 time steps (‘Domain’ in the graph), for the sub-task of running 80 Torres, which are synchronized each 4 time steps (DCs), for the sub-task of one CUDA-block (DC), for the sub-task of one warp, and for the subtask of one CUDA thread is estimated, and all corresponding limitations are found.

The whole data structure is taken as 26GB and fits the device memory. The layer of 80 Torres (DCs

![Figure 3. The Roofline model for the NVidia Volta V100. The obtained performance result is marked with a cross.](image-url)
in the graph) is synchronized during the run, and the amount of data that are exchanged between them may be localized in the LLC cache. That is why one Torre (DC) exchange with the global memory is through the LLC cache and its efficiency is limited by LLC bandwidth. Its base fits the shared memory, so the exchange between warps is limited by the shared memory access. In the result, the minimal limit is 309 billion cell updates per second (GCps).

In comparison, the limit of the theoretical ideal stepwise implementation is estimated as two reads ($f$ and $g$) and one write ($f$ or $g$) from the global memory of each value per each update in all cells of the domain. It is 82.8 GCps.

The performance obtained in our tests reaches up to 183 GCps. This is 18% of the peak computing performance of the device. Since it is still only $\sim 60\%$ of the estimated Roofline limit, the implementation can be improved. It may be possible with better load balancing, less overhead for offset computation, less shared memory use.

5. Conclusion
The DiamondCandy LRnLA algorithm for CUDA GPU has been constructed, implemented and tested. The obtained performance is at least twice higher than the performance of the theoretical ideal traditional implementation.

However, the current work is just a step in the process of finding a better solution of DiamondCandy CUDA GPU implementation. In the current work, the new data structure is introduced. With the shift along the $\mathbf{\vec{e}}_I$, the space is filled with the DiamondCandy tiles without introducing even/odd rows as it was done in [4]. In [4], access to the even and odd rows complicated the offset computation. Compared to previous attempts [3], the size of the base of the Torre is increased to $16^3$ elements, which lowers its surface to interior ratio, and, consequently, the raises the operational intensity. This was achieved by considering the warp parallelism and CUDA thread parallelism as two separate levels, so that the space, required for the exchange of data through the shared memory, was decreased.

The current result may be already applied for the implementation of other cross-stencil schemes, such as RKDG simulation of hydrodynamics [8] and other CFD schemes that use flux formulation. Thus, we plan to test out other variations of the implementation of the proposed algorithm.

The work is supported by the Russian Science Foundation (project #18-71-10004).

References
[1] Williams S, Waterman A and Patterson D 2009 Communications of the ACM 52 65–76
[2] Levchenko V D and Perepelkina A Y 2018 Lobachevskii Journal of Mathematics 39 552–561
[3] Levchenko V, Perepelkina A and Zakirov A 2016 Computation 4 29
[4] Perepelkina A, Levchenko V and Khilkov S 2018 The Journal of Supercomputing ISSN 1573-0484 URL https://doi.org/10.1007/s11227-018-2461-z
[5] Habich J, Zeiser T, Hager G and Wellein G 2009 Enabling temporal blocking for a lattice Boltzmann flow solver through multicore-aware wavefront parallelization 21st International Conference on Parallel Computational Fluid Dynamics pp 178–182
[6] Endo T, Midorikawa H and Sato Y 2019 Software Technology That Deals with Deeper Memory Hierarchy in Post-petascale Era (Singapore: Springer Singapore) pp 227–248 ISBN 978-981-13-1924-2
[7] Wonnacott D G and Strout M M 2013 IMPACT 2013 3
[8] Korneev B and Levchenko V 2016 Detailed numerical simulation of shock-body interaction in 3D multicomponent flow using the RKDG numerical method and "DiamondTorre” GPU algorithm of implementation Journal of Physics: Conference Series vol 681 (IOP Publishing) p 012046
[9] Maruyama N and Aoki T 2014 Optimizing stencil computations for NVIDIA Kepler GPUs Proceedings of the 1st International Workshop on High-Performance Stencil Computations, Vienna pp 89–95
[10] NVIDIA Corporation 2019 CUDA C Programming Guide pg-02829-001_v10.1 ed URL https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf