Extending the Problem Data Size for GPU Simulation Beyond the GPU Memory Storage with LRnLA Algorithms

Anastasia Perepelkina\textsuperscript{1}, Vadim Levchenko\textsuperscript{1}, Andrey Zakirov\textsuperscript{2}

\textsuperscript{1}Keldysh Institute of Applied Mathematics Miusskaya sq., 4 Moscow, 125047, Russia
\textsuperscript{2}Kintech Lab Ltds 12, 3rd Khoroshevskaya str., Moscow, 123298, Russia
E-mail: mogmi@narod.ru

Abstract. To use the CPU RAM to store the data of a GPU simulation, the data exchange between CPU and GPU is required. With LRnLA algorithms, the computation without data exchange is made longer, and the data exchange may be concealed. In this work, the concept is demonstrated with the Lattice Boltzmann method simulation with ConeTorre LRnLA algorithm on GPU. A new method of data storage is proposed. The data on the synchronization planes are stored in tiles, and on the wavefront slopes a new data structure is used, which is optimized for cell data exchange between LRnLA sub-tasks. The performance tests show less than 5% overhead for CPU-GPU communication, and the GPU performance persists for simulations where the main storage site is the CPU RAM.

1. Introduction
In stencil schemes for computational physics, the performance limits are often imposed by the memory bandwidth of the system. Generally, stepwise algorithms are used, and all data of the problem has to be loaded at least once per time iteration. In parallel execution, data exchange between processes is required at least once per time iteration.

With Locally Recursive non-Locally Asynchronous (LRnLA) algorithms [1], sub-tasks are found in the dependency graph, which require fewer synchronizations, and more data may be reused. This eases the bottlenecks imposed by insufficient memory bandwidth. LRnLA algorithms may be applied to different types of data storage, and to different kinds of parallelism.

In this work, LRnLA algorithm theory is applied specifically to the problem of the data exchange between GPU device memory and CPU RAM. Generally, GPU devices have ten times higher memory bandwidth, so an expected acceleration with GPU is an order of magnitude. On the other hand, the GPU device memory is limited, while the CPU RAM may be extended to be several orders of magnitude larger. Thus, the data size of the problems that fit GPU memory is restricted by relatively small sizes, of the order of $\sim 10$ GB.

If the simulation data is stored in the CPU RAM, the data exchange between CPU and GPU is required. However, the PCI-e bandwidth is low. When the data exchange operations are included, the overall time of the code execution may be comparable to the CPU version. This way, the use of GPU becomes meaningless. To increase the size of the GPU accelerated tasks multi-GPU setups are used. It is convenient but expensive due to the GPU device costs.
Another way is to use hybrid simulations, where the task is partially accelerated on GPU [2]. However, since the CPU performance is an order of magnitude lower than the GPU performance, it makes sense to use GPU for computation, and CPU for storage. Some efforts to overcome the low bandwidth problem has been made with a lossy data compression [3]. Temporal blocking approach had been used to mitigate the performance loss when the data exchange between CPU and GPU takes place [4]. With such methods, the main storage site can be ambiguously large, as even SSD space may be used with little performance loss [5].

With LRnLA algorithms the problem of the limitation of the problem data size by the low speed of access to the larger storage have been solved both for GPU [6] and Xeon Phi [7] co-processors. In [6], the implementation of the FDTD scheme [8] with the DiamondTorre LRnLA algorithm allowed concealing the data copy and avoiding the performance loss. The algorithm itself was better suited for cross stencils, and the LRnLA decomposition was 2D.

Here we propose the theory as well as the implementation of the new LRnLA algorithm FArShWin. It allows solving 3D stencil problems, where the size is comparable to the CPU storage, and, at the same time the performance corresponds to the GPU accelerated codes.

2. The Algorithm Description

2.1. Lattice Boltzmann Method

A standard LBM [9] may be described as a two-stage numerical scheme in terms of the discrete PDF (particle distribution function) variables $f_q(^{\vec{x}}_i)$ at the mesh nodes $^{\vec{x}}_i$. One of these steps is a local collision operation $\Omega$ at node $^{\vec{x}}_i$. The other is a streaming step, which consists of $Q$ separate transfers of $f_q$ values in the $^{\vec{c}}_q$ directions across all $x_i$ nodes. In the d-dimensional space the LBM scheme with $Q$ discrete velocities is defined by the following expressions. For each mesh node $^{\vec{x}}_i$, for each time instant $^{t_k}$,

$$f_q(^{\vec{x}}_i,^{t_k}) = \Omega(^{f_q}_{^*}(^{\vec{x}}_i,^{t_k}),...^{f_q}_{Q}(^{\vec{x}}_i,^{t_k})),$$

$$^{f_q}_{q}(^{\vec{x}}_i + ^{\vec{c}}_q,^{t_k} + 1) = ^{f_q}_{q}(^{\vec{x}}_i,^{t_k}), \quad q = 1..Q.$$

The number of values that are needed per lattice site is $Q$ floating point values ($s$ bytes, $s = 76$ for $Q = 19$), and the number of the floating point operations is determined by the choice of the collision operator. Here, we estimate the number of operations per lattice site update as $o = 360$, which corresponds to the BGK collision with the equilibrium expression as a second order polynomial. It is convenient for performance comparisons, and since this collision is one of the less computationally heavy, makes the problem more memory-bound.

2.2. ConeTorre

In this work, the ConeTorre LRnLA algorithm is used on GPU as it is described in [10, 11, 12]. The streaming scheme is compact [13]. In it, only one data array copy is needed, since the ‘in-place modify’ property takes place, and per each $^{f_i}$ update one load and store is required everywhere. The update is performed in cell groups of $2^d$ cells, where the data of the group is loaded, updated, and stored, and $2^d$ full cell updates are performed without the access to the neighbor groups. The groups are shifted on the even and odd time steps. One CUDA-block processes a ConeTorre with a base of $n^3$ cells, one cell per CUDA thread. These cells are loaded into the register. A CUDA thread updates a cell, and received the cell data from the CUDA thread which has a cell shifted by $(1, 1, 1)$ in $(x, y, z)$. The threads on the right borders of the cube receive the data from the global memory, and the threads on the left borders of the cube save the updated cell data to the global memory. It repeats $N_T$ times, and then the whole upper base of the ConeTorre is stored into the global memory. ConeTorre comprise a TorreFold: several CUDA-blocks are run simultaneously, and the execution of the ConeTorre levels inside them is controlled by the dependencies and implemented with mutexes [10].
In this work, for the first time, a special data structure is used for the LRnLA algorithm data exchange. During the ConeTorre execution, the data exists either on the synchronization plane, or in a partially computed state on the ConeTorre slope. Here we define two data structures. The first one is tile data structure for storage of information corresponding to the synchronization instants. It contains the array of data of cell groups in a Z-curve, or in tiles with any d-dimensional indexing. The cell group data contains \( f_i \) values in the SoA fashion for coalesced read/write, and the groups are stored in the Z-curve array for local addressing. The second data structure (Functionally Arranged Shadow - FArSh) contains data, arranged in a way in which it is received by a ConeTorre from its neighboring ConeTorre.

The ConeTorre base is a cube with linear size \( n \), and, initially, one diagonal row is assigned to each cell of the cube on the right border (that is, to each cell in the one-cell wide cube gnomon). The cells in one row correspond to \( N_T \) cell states, each cell in a row is shifted by \((1, 1, 1, 1)\) in \((x, y, z, t)\) from the previous one. After the execution of the ConeTorre with this cube as its base, FArSh is shifted to the positions on the left borders of the cube.

The use of FArSh as a special structure for data exchange is helpful on many levels, and here is trivialized the data exchange with CPU RAM. Here we store the simulation data in the initial position of the BaseTile structure in CPU RAM (Fig. 1). It is a cube of linear size \( N > M \), and we send its sub-tiles of size \( M \) one by one to the GPU. FArSh is also stored in GPU.

Threads inside one warp exchange data by warp-shuffles. For exchanges between warps, shared memory is used. FArSh is used as the data structure in the shared memory as well.

The asynchronous exchange is implemented with CUDA streams. Each tile has to be sent to GPU, computed, and sent back to CPU. If we start to load the second tile as soon as the first one starts to be updated in a ConeTorre in GPU, it should be completely loaded by the time the first tile is processed, so GPU continues calculation uninterrupted. The calculation of one ConeTorre should be long enough to conceal the tile copy operation. With LRnLA, the ConeTorre height \( N_T \) is adjusted for this.

### 2.3. Performance Estimation

Let us consider a TorreFold on a cubic domain in \( d \) dimensions with linear size of \( M \) and \( M^d \) cells total, and assume the data fits the GPU memory with bandwidth equal to \( T_{GPU} \):

\[
 s M^d \leq M_{GPU}.
\]

Let us estimate the performance \( P_{CT} \) of the ConeTorre algorithms with a linear size of \( n \) which constitute this TorreFold. Here we assume that the data of \( n^d \) cells, \( s \) byte per cell, is localized in the SM register file, and the data exchange is through shared memory or L2 cache.
To process a ConeTorre, $o^d N_T$ operations are required and data throughput is $2sdn^{d-1} \gamma N_T$ bytes. Here $dn^{d-1}$ is the most significant term of the expression of the cube gnomon $((n+1)^d - n^d)$, and $\gamma = ((n+1)^d - n^d)/dn^{d-1} = 1 + O(1/n)$ is the correction term. Its value may differ in some cases, depending on the presence of diagonal dependencies in the stencil.

According to the Roofline model [14, 1],

$$P_{CT} \leq \min \left( P_{GPU}, \frac{T_{GPU}}{1 + d \gamma N_T/n} \right),$$  \hspace{1cm} (4)

If memory-boundness is assumed, and $N_T$ is large enough, the performance is bound by

$$P_{CT} < \frac{n^d T_{GPU}}{2s},$$ \hspace{1cm} with $d N_T \gg n \gg 1$.

Next, let us consider the processing of a task with the base size of $N$ cells in $d$ dimensions, and with the use of FArSh data structure. The size of the BaseTile is $sN^d$ bytes. The data on the ConeTorre slopes is stored in FArSh, and its size is $sdN^{d-1} \gamma N_T$. The total data size is $sN^d(1 + d \gamma N_T/N)$.

In case the local boundary conditions are in use, the problem data is the cube with $N^d$ cells total, and the FArSh data overlaps it in the course of TorreFold execution, creating a copy of the cells that are on the current ConeTorre slope position. Before and after the TorreFold is called, FArSh covers an area outside the domain, and thus, contains no relevant data. On the other hand, periodic boundary of the Bloch type [8] may be used. Here we assume that all data both in the tile and in the FArSh is relevant at the start of the TorreFold. During the execution the data is not doubled. Thus, the data size is exactly $sN^d(1 + dN_T/N)$ (in fact, here, $\gamma \equiv 1$).

Since the FArSh data is in GPU memory, $sdN^{d-1} N_T \leq M_{GPU}$. The BaseTile is localized in the larger storage, $sN^d \leq M_{CPU}$. These conditions replace the similar condition (3). When the

$$dN_T/N = M_{GPU}/M_{CPU},$$ \hspace{1cm} (5)

condition is satisfied the simulation area size can reach the maximal size, and the data size of the whole problem may be up to $M_{GPU} + M_{CPU}$. Thus, the available storage of different memory levels are added up.

Let us consider one ConeTorre with base size of $n^d$ cells. The FArSh data are in GPU, but the base data of the TorreFold has to be sent from CPU to GPU, and after the TorreFold is executed, it has to be sent back to the BaseTile in CPU. If the data exchange is asynchronous with computation and its rate is $T_{CPU \leftrightarrow GPU}$, we require:

$$t_{exch} = 2sn^d/T_{CPU \leftrightarrow GPU} \leq t_{calc}.$$

Combining it with the performance estimate (4) we get

$$P_{FArSh/Win} \leq \min \left( P_{GPU}, \frac{T_{GPU}}{1 + d \gamma N_T/n} \frac{N_T}{2s}, T_{CPU \leftrightarrow GPU} N_T \frac{\gamma}{2s} \right).$$ \hspace{1cm} (7)

If $t_{calc}$ in (6) is estimated under the assumption an 'ideal' (with maximum possible data reuse) stepwise algorithm is used for $N_T$ time steps, and the data is localized in the GPU memory,

$$t_{exch} \leq t_{ideal SW} = 2sn^d N_T/T_{GPU},$$

and the condition for $N_T$ is $N_T \geq T_{GPU}/T_{CPU \leftrightarrow GPU}$. With (5), the condition for hardware is

$$N_T = \frac{N M_{GPU}}{d M_{CPU}} \geq \frac{T_{GPU}}{T_{CPU \leftrightarrow GPU}}.$$

If the limitation on $N$ is considered, the final condition for the possibility of simulating the tasks localized in the CPU RAM with the GPU speed is

$$d \cdot \frac{M_{CPU}}{M_{GPU}} \cdot \frac{T_{GPU}}{T_{CPU \leftrightarrow GPU}} \leq N \leq \sqrt[4]{\frac{M_{CPU}}{s}}.$$
3. Performance Results

The implementation was tested on a desktop workstation with NVidia RTX2070 GPU and Intel Core i7-7800X CPU. Its parameters are $M_{\text{GPU}} = 8\, \text{GB}$, $M_{\text{CPU}} = 128\, \text{GB}$, $T_{\text{CPU\leftrightarrow GPU}} = 2 \times 15.8\, \text{GB/s}$, $T_{\text{GPU}} = 448\, \text{GB/s}$. According to the estimates in Section 2.3, limitations on $N_T$ and $N$ are $14 \leq N_T \leq 38$ and $672 \leq N \leq 1200$. Since the ranges are wide enough, this system actually allows problems with size of the CPU RAM to be processed with the GPU memory throughput as the performance limit. To prove the estimates we vary the parameters in the following ranges $1 \leq N_T \leq 128$, $64 \leq N \leq 1024$. With $N = 1024$, the BaseTile size equals to $1024^3$ cells total (76 GB data).

The ConeTorre and TorreFold parameters $n$ and $M$ are chosen to be 8 and 64 correspondingly. With $n = 8$, 512 CUDA-threads are used for a ConeTorre execution. Thus, occupancy is sufficient even when 1 CUDA-block per SM is run. The GPU in use has 36 SMs, which leads to the choice of the TorreFold parameter $M = 64$. This way, $(M/n)^3 = 512 \gg 36$, and this is enough for balanced parallel load.

The performance of D3Q19 LBM is measured in billion lattice node updates per second, GLups. In Fig. 2, 3, the ‘calc only’ line shows performance in case no data exchange is performed between CPU and GPU, and the other curves are plotted with taking the exchange times into account.

In Fig. 2a, the dependency of performance on the ConeTorre height $N_T$ is shown. As expected, with higher $N_T$ the performance is higher due to data reuse in the GPU register. In the ‘calc only’ mode, the saturation is achieved with $N_T \geq 16$. When the data exchange is implemented synchronously (with one stream) the overhead for data copy operations decreases the performance significantly. However, with the use of asynchronous exchange by several streams the performance is restored when $N_T \geq 32$. In Fig.2b, the dependency of the performance on the problem size is shown for $N_T = 32$. The right portion of the plot confirms no loss in performance when the data size exceeds GPU storage capacity.

Strong scaling with $N = 1024$ (Fig. 3a) shows the data copy is the limiting factor in the synchronous mode (1 stream) when the number of CUDA blocks is higher than 8, while below the computation itself is too slow that the data movement is concealed. Just two asynchronous streams are enough to reach full parallel capacity. The overhead produced by asynchronous data copy is below 5% (Fig. 3b).

4. Conclusion

We have developed the new FArShWin algorithm for faster solution of 3D stencil problems, where the size is comparable to the CPU storage and the performance is limited by the GPU.

Figure 2. (a) Performance vs ConeTorre height $N_T$ with $N = 512$ and (b) vs data size with $N_T = 32$.
memory throughput. The parameter estimates presented in this paper prove the relevance of the algorithm for a wide range of GPU-based computer systems, and for a wide range of numerical schemes. The performance benchmarks of the code, described in the paper demonstrate no loss in performance when the data size exceeds GPU storage capacity. The overhead is less than 5% when the asynchronous CUDA streams are used. We have obtained the performance of up to 2.4 GLUPs for the D3Q19 LBM code, which is at least 3.5 times higher than the CPU LRnLA implementation for the problem of same size on the same computer [13].

The work is supported by the Russian Science Foundation, grant #18-71-10004.

References
[1] Levchenko V and Perepelkina A 2018 Lobachevskii Journal of Mathematics 39 552–561
[2] Riesinger C, Bakhtiari A, Schreiber M, Neumann P and Bungartz H J 2017 Computation 5 48
[3] Sun G, Kang S and Jun S W 2020 BurstZ: a bandwidth-efficient scientific computing accelerator platform for large-scale data Proceedings of the 34th ACM International Conference on Supercomputing pp 1–12
[4] Shimokawabe T, Endo T, Onodera N and Aoki T 2017 A stencil framework to realize large-scale computations beyond device memory capacity on GPU supercomputers Cluster Computing (CLUSTER) pp 525–529
[5] Midorikawa H and Tan H 2015 Locality-aware stencil computations using flash SSDs as main memory extension 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (IEEE) pp 1163–1168
[6] Zakirov A, Levchenko V, Perepelkina A and Zempo Y 2016 High performance FDTD algorithm for GPGPU supercomputers Journal of Physics: Conference Series vol 759 (IOP Publishing) p 012100
[7] Perepelkina A, Levchenko V and Khilkov S 2019 The Journal of Supercomputing 75 7778–7789
[8] Tallove A and Hagness S C 2005 Computational electrodynamics: the finite-difference time-domain method (Artech house)
[9] Succi S 2001 The lattice Boltzmann equation: for fluid dynamics and beyond (Oxford university press)
[10] Levchenko V, Zakirov A and Perepelkina A 2019 GPU implementation of ConeTorre algorithm for fluid dynamics simulation Parallel Computing Technologies, PaCT 2019, Lecture Notes in Computer Science vol 11657 ed Malyshekin V (Cham: Springer) pp 199–213
[11] Levchenko V, Zakirov A and Perepelkina A 2019 LRnLA lattice Boltzmann method: A performance comparison of implementations on GPU and CPU International Conference on Parallel Computational Technologies (Springer) pp 139–151
[12] Zakirov A, Perepelkina A, Levchenko V and Khilkov S 2020 Streaming techniques: Revealing the natural concurrency of the lattice Boltzmann method Journal of Supercomputing, to be published
[13] Perepelkina A, Levchenko V and Zakirov A 2020 New compact streaming in LBM with ConeFold LRnLA algorithms Communications in Computer and Information Science, to be published
[14] Williams S, Waterman A and Patterson D 2009 Communications of the ACM 52 65–76