The Eleventh International Conference on
Engineering Computational Technology
23–25 August, 2022 | Montpellier, France

Proceedings of the Eleventh International Conference on
Engineering Computational Technology
Edited by B.H.V. Topping and P. Iványi
Civil-Comp Conferences, Volume 2, Paper 8.5
Civil-Comp Press, Edinburgh, United Kingdom, 2022, doi: 10.4203/ccc.2.8.5
©Civil-Comp Ltd, Edinburgh, UK, 2022

MRT Lattice Boltzmann Method on multiple Graphics Processing Units with halo sharing over PCI-e for non-contiguous memory

T.O.M. Forslund
Division of Fluid and Experimental Mechanics
Luleå University of Technology, Sweden

Abstract

The Lattice Boltzmann Method (LBM) has been shown to be well suited for implementation on Graphics Processing Units (GPUs). The benefit of GPU implementations compared to CPU is in the reduction of computational time, by as much as 2 orders of magnitude. This staggering difference is due to how computations for LBM are both explicit and local, meaning that it can make full use of the GPUs capabilities, like most other cellular automata methods. Although GPUs have a significantly larger performance in terms of floating-point operations per second (FLOPS) compared to a CPU it has two significant drawbacks; First, the complexity of the calculations is limited due to the relative simplicity of the GPU core design compared to a CPU, secondly, the memory of a GPU is usually limited in comparison, ranging from a few GB up to ≈ 100 GB for high-end enterprise cards. Because the LBM method is suitable for execution on GPUs the first point is not necessary to consider. But the second point becomes a limitation as larger, or more highly resolved computational domains are of interest. This can be remedied by distributing the computations across several GPUs executing in parallel. The GPUs share values in overlapping regions called halo-values that need to be transferred each time step. If the memory is contiguous then each transfer can be executed as a single efficient memory transfer call that utilizes the PCI-e lanes efficiently. If this is not the case then support exists for copying of so-called strided memory which has a constant offset between values for either single strided (2D) or double strided (3D). These functions practically result in bad PCI-e lane utilization and to remedy this a method is proposed, the halo-values are calculated and packed into a contiguous memory buffer that is then communicated between the GPUs via the PCI-e lanes. It is shown that the method introduces some additional overhead compared to single GPU execution but maintains a reasonable 70% performance compared to the single GPU case.

Keywords: lattice Boltzmann method, GPU, multi-GPU programming

1 Introduction

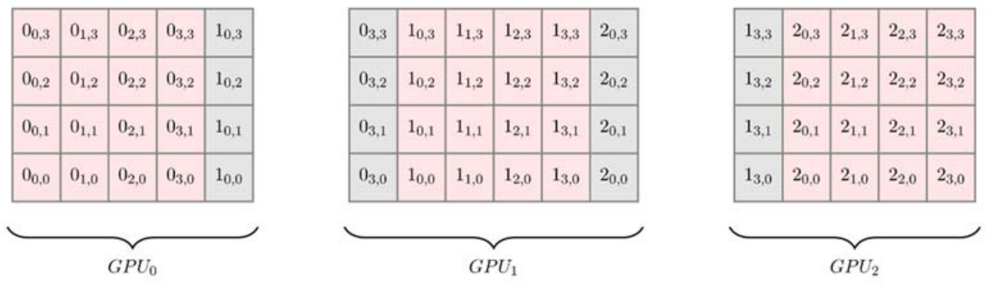

Figure 1: The halo exchange regions between three GPUs labeled 0, 1 and 2. Grey values are halos and red are inner nodes

The Lattice Boltzmann Method (LBM) has been shown to be well suited for implementation on Graphics Processing Units (GPUs) [1, 2, 3]. Benefits of GPU implementations compared to CPU is in the reduction of computational time, by as much as 2 orders of magnitude [2]. This staggering difference is due to how computations for LBM are both explicit and local, meaning that it can make full use of GPU capabilities, like most other cellular automata methods. Although GPUs have a significantly larger performance in terms of floating-point operations per second (FLOPS) compared to a CPU it has two significant drawbacks; First, the complexity of the calculations is limited due to the relative simplicity of the GPU core design compared to a CPU, secondly, the memory of a GPU is usually limited in comparison, ranging from a few GB up to ≈ 100 GB for high-end enterprise cards. Because the LBM method is suitable for execution on GPUs the first point is not necessary to consider. But the second point becomes a limitation as larger, or more highly resolved computational domains are of interest. Therefore the computational domain can be divided into several separated computational blocks distributed across the GPUs as graphically illustrated in Figure 1. The values called halos are those that overlap between the GPUs, these must be transferred at each time step. If the memory is contiguous, as it can be arranged for this simple linear case presented in Figure 1 then each transfer can be executed as a single efficient memory transfer call that utilizes the PCI-e lanes efficiently. If this is not the case then support exists for copying of so-called strided memory which has a constant offset between values [4] for either single strided (2D) or double strided (3D). These functions practically result in bad PCI-e lane utilization and to remedy this a method is proposed, the halo-values are calculated and packed into a contiguous memory buffer that is then communicated between the GPUs across the PCI-e lanes. The program structure can be summarized as.

1. Halo part
    a. Calculate the halos
    b. Pack the halos into container memory
    c. Send container memory
    d. Unpack container
2. Inner blocks
    a. Calculate the inner nodes for each GPU

2 Methods

The algorithm is implemented in the Compute Unified Device Architecture (CUDA). The layout of the program is presented as pseudo-code snippets with comments that clarify the code operation. The type of LBM model used in this work is a 3-Dimensional 27 distribution model (D3Q27), for additional details regarding the model see [5]. The calculation of the halos and main body call the same function queueCollideAndStream(args) which is the function that combines the streaming and collision steps as defined in as defined in [1, 5]. The for loop goes across all GPU devices and queues the halo and transfer calculation for each halo-stream.

```
for(int i = 0; i < nGPUs; i++)
{
        cudaSetDevice(i);
        queueBC<T>(&blockBC[i], &gridBC[i], varBC[i], haloStream[i]);
        queueCollideAndStream<T>(&blockHalo[i], &gridHalo[i], varHalo[i], haloStream[i]);
        pack(&blockTrans, &gridTrans, varPack[i], haloStream[i]);

}

for(int i = 0; i < nConnections; i++)
{

        cudaMemcpyPeerAsync(f_in[i], gpu_in[i], f_out[i], gpu_out[i], size, haloStream_conn[i]);
}

for(int i = 0; i < nGPUs; i++)
{
        unPack(&blockTrans, &gridTrans, varHalo[i], haloStream[i]);
}
```

Where the boundary condition function queueBC(args) is executed first and as part of the halo-stream. The varHalo[i] object contains all the variables relevant for the function and is not a single argument in the actual implementation. The values are then packed using the packing function and copied between the GPUs using the PCI-e lanes by using the cudaMemcpyPeerAsync(args) function. After this the values are unpacked and the next computation can begin. Running in parallel with this calculation and transfers are the inner calculations, those that are not boundaries or halos. The calculation procedure here is simply a call to the queueCollideAndStream(args) function.

```
for(int i = 0; i < nGPUs; i++)
{
        cudaSetDevice(i);
        queueCollideAndStream<T>(&block[i], &grid[i], var[i], innerStream[i]);
}
```

After this call the function starts over at the beginning of the main loop again.

3 Results

The algorithm is executed on a channel cylinder test case using a workstation with three RTX8000 GPUs, the geometry and boundary conditions can be seen in Figure 2. The case consists of a constant velocity inlet (green), a zero-gradient outlet (red) and walls (light blue). The domain is subdivided into three equally large computational domains for GPU 1 (green), 2 (blue) and 3 (red). A render of the iso-surfaces of vorticity is presented in Figure 3 which shows that the overlap region between the computational blocks is well behaved.

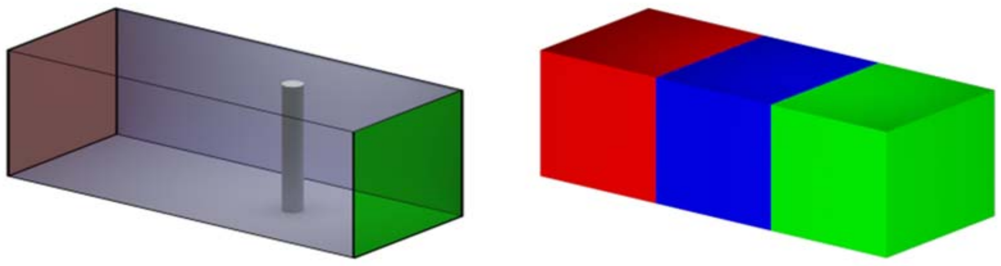

Figure 2: The boundary condition and geometry (left), and the division of the computational blocks (right).

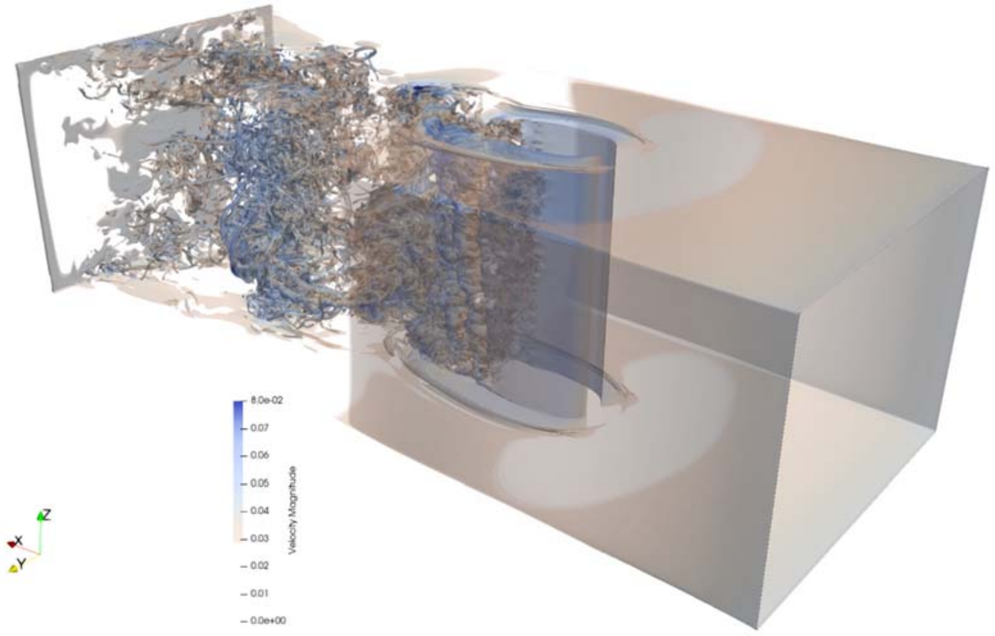

Figure 3: Iso-surface of the magnitude of the vorticity coloured by the velocity magnitude for the block size of 144^3.

The calculation speed of the algorithm is measured in Lattice Updates Per Second (LUPS). A single RTX8000 GPU performs ≈ 2200 MLUPS with the D3Q27 model, therefore in the best-case scenario of full parallel execution without bottlenecks a performance of ≈ 6600 MLUPS is expected. The measured performance from the simulations can be seen in Table 1. As we can see the performance is ≈ 70% of the expected value for the largest simulation domain.

n^3	n_{el}	$MLUPS$
24^3	1658880	1760
48^3	13271040	2890
96^3	106168320	4042
144^3	358318080	4719

Table 1: the performance in MLUPS for different base lattice sizes n^3 and total amount of elements n_{el}.

4 Conclusions and Contributions

In conclusion the PCI-e lane is sufficient for transferring the halo values between the GPUs without adding any significant bottlenecks to the calculation given that the ratio of halo-nodes to inner nodes are high. Even if the memory is not contiguous the packing algorithm adds the possibility of transferring the halos fast between the GPUs by adding a small memory allocation overhead. As the amount of GPUs utilized increase the amount of transferred memory compared to the inner memory per GPU is expected to increase, i.e. the amount of halo nodes in total compared to inner nodes per GPU. A quick estimation of whether a given application will bottleneck due to this limitation can be carried out by keeping the expected execution time of the PCI-e transfers below the execution time for each inner block, i.e. we have the inequality in Equation (1)

$$\frac{2\,n_{halo}}{S_{PCI}} = \frac{n_{inner}}{S_{GPU}} \tag{1}$$

Where n_{halo} is the amount halo elements (in total), S_{PCI} is the memory transfer rate of the PCI-e lane, n_{inner} is the amount of inner elements per GPU and S_{GPU} is the memory transfer rate of the GPU. Rearranging we have Equation (2)

$$\frac{S_{GPU}}{S_{PCI}} = \frac{n_{inner}}{2\,n_{halo}} \tag{2}$$

Assuming cubic elements that are connected across all edges with halos the ratio n_{inner}/n_{halo} will scale by the relation in Equation (3)

$$\frac{n_{inner}}{n_{halo}} = \left(\frac{n_{inner}}{6\,n_{inner}{}^{2/3} n_{GPU}}\right) \tag{3}$$

Where n_{GPU} is the amount of GPUs. Using the 3rd gen PCI-e RTX8000 as an example the value $S_{PCI}/S_{GPU} \approx 128/672$, the value for S_{PCI} is based on the number of PCI-e lanes which varies depending on the hardware. This gives a required amount of inner nodes per GPU to avoid bottle-necking in Equation (4)

$$n_{inner} = (\frac{8064\, n_{GPU}}{128})^3 \quad (4)$$

The maximum amount of inner nodes for an RTX8000 card running a D3Q27 model is ≈ 200M , therefore we see that approximately 9 cards can operate on the same host before the PCI-e lanes become a limitation. Changing the connectivity of the GPUs to a 2D or 1D grid or increasing the amount of PCI-e lanes does not impact the n_{GPU}^3 term, this means that the method is not viable beyond more cards than ≈ 10.

Acknowledgements

This work was made possible by funding from the Swedish Research Council, grant 2017-04390.

References

[1] N. Delbosc, J. L. Summers, A. I. Khan, N. Kapur, and C. J. Noakes, "Optimized implementation of the Lattice Boltzmann Method on a graphics processing unit towards real-time fluid simulation," Computers and Mathematics with Applications, vol. 67, no. 2, pp. 462–475, 2014.

[2] N. Delbosc, "Real-Time Simulation of Indoor Air Flow Using the Lattice Boltzmann Method on Graphics Processing Unit," no. September, 2015.

[3] F. Kuznik, C. Obrecht, G. Rusaouen, and J. J. Roux, "LBM based flow simulation using GPU computing processor," Computers and Mathematics with Applications, vol. 59, no. 7, pp. 2380–2392, 2010.

[4] NVIDIA, "Cuda C Programming Guide," Programming Guides, no. September, pp. 1–261, 2015.

[5] K. Suga, Y. Kuwata, K. Takashima, and R. Chikasue, "A D3Q27 multiple-relaxation-time lattice Boltzmann method for turbulent flows," Computers and Mathematics with Applications, vol. 69, no. 6, pp. 518–529, 2015.