Keldysh Institute • Publication search

Keldysh Institute preprints • Preprint No. 44, 2016

ISSN 2071-2898 (Print)
ISSN 2071-2901 (Online)

Zakirov A.V., Levchenko V.D., Perepelkina A.Yu., Zempo Yasunari

High performance FDTD code implementation for GPGPU supercomputers

Recommended form of bibliographic references: Zakirov A.V., Levchenko V.D., Perepelkina A.Yu., Zempo Yasunari. High performance FDTD code implementation for GPGPU supercomputers // Keldysh Institute Preprints. 2016. No. 44. 22 p. doi:10.20948/prepr-2016-44-e
URL: http://library.keldysh.ru/preprint.asp?id=2016-44&lg=e

Ордена Ленина
ИНСТИТУТ ПРИКЛАДНОЙ МАТЕМАТИКИ
имени М. В. Келдыша
Российской академии наук

Andrey Zakirov, Vadim Levchenko,
Anastasia Perepelkina, Yasunari Zempo

High performance FDTD code implementation for GPGPU supercomputers

Москва
2016

УДК 519.688

А.В. Закиров, В.Д. Левченко, А.Ю. Перепёлкина, Я. Земпо

Высокопроизводительная реализация конечно-разностного метода FDTD для суперкомпьютеров с графическими процессорами

Аннотация. Описана реализация конечно-разностного метода на сдвинутых сетках (FDTD) для решения задач электродинамики, в том числе нанооптики, требующих больших вычислительных ресурсов. Реализация основана на локально-рекурсивном нелокально-асинхронном (LRnLA) алгоритме DiamondTorre, эффективном при расчетах на графических процессорах общего назначения (GPGPU). Обсуждаются особенности алгоритма DiamondTorre для задач на сдвинутых сетках (на основе ячейки Йи) при реализации на многопроцессорном кластере с гибридной архитектурой. Алгоритмы реализованы с использованием технологий CUDA, OpenMP и MPI в программном комплексе, предназначенном для решения реальных физических задач. Пределы производительности оценены из параметров алгоритма и модели roofline суперкомпьютера TSUBAME 2.5. Полученные оценки сравниваются с реальной производительностью программного комплекса как на одном вычислительном устройстве, так и при параллельном масштабировании в слабой и сильной метриках. При этом достигнута производительность до $0.65 \cdot 10^{12}$ обновлений ячеек в секунду для трёхмерной области с количеством ячеек $0.3 \cdot 10^{12}$.

**Andrey Zakirov, Vadim Levchenko,
Anastasia Perepelkina, Yasunari Zempo**

High performance FDTD code implementation for GPGPU supercomputers

Abstract. An implementation of FDTD (Finite Difference Time Domain) method for solution of optical and other electrodynamic problems of high computational cost is described. The implementation is based on LRnLA (Locally Recursive non-Locally Asynchronous) algorithm DiamondTorre, which is developed specifically for GPGPU (General Purpose Graphical Processing Unit) hardware. The specifics of the DiamondTorre algorithms for staggered grid (Yee cell) and many-GPU devices are shown. The algorithm is implemented in software for real physics calculation with the use of CUDA, OpenMP, MPI technologies. The software performance limits are estimated through algorithms parameters and computer model of TSUBAME2.5. The real performance is tested on one GPU device, as well as on many-GPU cluster with strong and weak scaling tests. The performance of up to $0.65 \cdot 10^{12}$ cell updates per second for 3D domain with $0.3 \cdot 10^{12}$ Yee cells total is achieved.

1 Introduction

Finite Difference Time Domain (FDTD) method [1] is implemented in numerous codes for simulation of electrodynamics. Among other possible applications, it is used for design of optical devices, complex coatings, nanoantennas. The method is easily extended for simulation of other wave processes, such as acoustics and elasticity waves [2].

As it comprises the base of many contemporary calculations, the qualitative acceleration of the code performance would aid many aspects of supercomputer research. This paper deals with one way of rethinking the traditional approach to numerical solution of evolutionary Cauchy problem, the Locally Recursive non-Locally Asynchronous (LRnLA) algorithms [3]. Based on this approach, we have developed software for simulation of real optics on General Purpose Graphical Processing Unit (GPGPU) and measured its efficiency on one device, and on a multi-GPU supercomputer.

1.1 Background

Traditional FDTD implementations are based on algorithms which correspond to von-Neumann model of computations with parallel generalizations by Amdahl (for OpenMP implementation) and Hoare (for MPI implementation). Among the recent parallel models CUDA (OpenCL) should be noted, since these have adequate support of vector level parallelism.

The ideal paradigm of CPU RAM as a main storage of processed data leads to the fact that the majority of numerical scheme implementations use the computational region traversal rules that are layerwise, and linear multidimensional arrays for data structures.

Layerwise synchronization is the most obvious approach: after all field data in the region is updated by one time step, the calculation for next time step begins.

The immediate consequence is a an essential limitation of the maximum locality of processed data. Since the memory subsystem is hierarchical, when the computation region is upscaled, this leads to performance degradation at times when the data size exceeds cache size.

For parallel implementation the layerwise approach requires layerwise synchronization of parallel processors. This leads to the limitation of upscaling set by communication environment latency, and imbalance of computation nodes' loads.

These problems result in extremely low efficiency of applications that use these methods. For example, among FDTD software, Meep [4], Lumerical, have efficiency that is lower than 1%. Other implementations of FDTD on GPGPU is

limited by GPU device memory [5].

1.2 Other space-time approaches

The difference between LRnLA method and the conventional methods is that the optimization of computations deals not only with layerwise computation, but traces data dependencies in 4D time (iteration) and space domain [6–8] (see detailed in later sections).

Although this approach is not wide spread, the basics may be traced in works by other authors. Developed since early 1980s, the so-called loop tiling and loop skewing methods result in similar algorithms for simple domain geometry [9–12]. The research on loop blocking led to creation of cache-aware and cache-oblivious algorithms [13], which were used later for stencil computations of partial derivative equations [14–16]. In 1D simulation these techniques lead to trapezoidal and diamond blocking of space, with generalizations to 2D and 3D.

Among these approaches LRnLA has the following advantages:

- The approach takes account for the complexity of modern computers. The space-time optimization account for all parallel levels, all levels of memory subsystem.
- The theory is built on the model of the computer and allows a priori quantitative estimates of the performance of method implementation.
- The theory applies to any physics simulation with local dependencies, any amount of dimensions.

The main difference lies in the approach of building the algorithms. In LRnLA method, the best algorithm is chosen based on the analysis of both the computer system (by creating a model of memory subsystem and parallel levels) and numerical scheme properties (by constructing a dependency graph and tracing the dependencies). It should be noted that some cache-oblivious algorithms coincide with the algorithms of LRnLA family for lower dimensions.

2 Numerical method

FDTD (Finite Difference Time Domain) [1] numerical method is one of most popular for simulation of wave phenomena, as it is accurate and robust. In the problems of optics and other electrodynamic processes FDTD is used for numerical solution of Maxwell equation in time-space domain:

$$\frac{\partial \vec{D}}{\partial t} = \nabla \times \vec{H}; \qquad \frac{\partial \vec{B}}{\partial t} = -\nabla \times \vec{E}, \tag{1}$$

where $\vec{E}$ and $\vec{D}$ signify electric field, $\vec{H}$ and $\vec{B}$ signify magnetic field. $\vec{E}$ and $\vec{D}$, as well as $\vec{H}$ and $\vec{B}$ are bounded by material equations. In the most simple case, $\vec{D} = \vec{E}, \vec{B} = \vec{H}$. Speed of light is considered equal to 1 in the chosen dimensionless units.

The simulation domain is subdivided with Yee Grid. Electric and magnetic field components are defined in different positions inside the unit cell (fig. 1), and shifted by half-step in time

We use the scheme with 4th order of approximation. Compared to the 2nd order scheme its stencil is wider and for coarser meshes it is substantially more accurate [17]. Scheme stencil is cross-shaped.

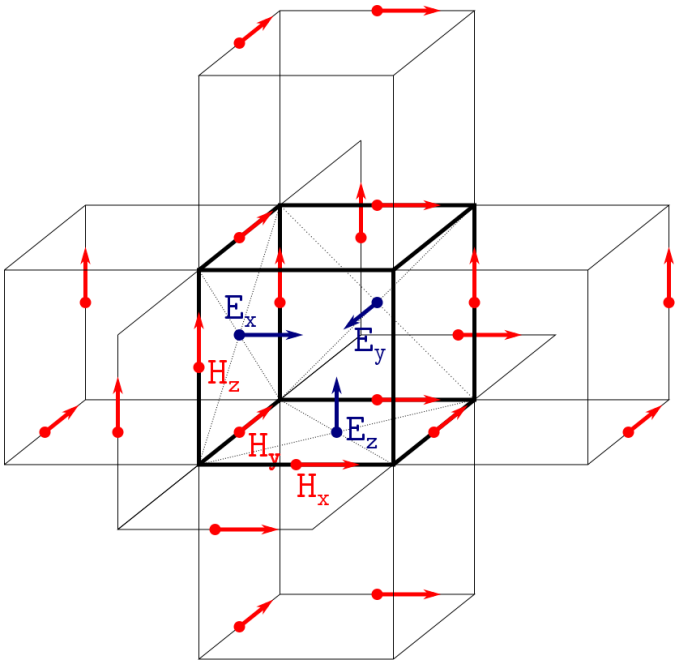

Fig. 1: Yee cell. Apart from one basic cell all fields that influence the electric fields in it are shown (for 4th order scheme).

3 Algorithm description

We use roofline [18] model to estimate the efficiency of the algorithms. Although other models exist [19], roofline model is the most convenient for the current study. It shows only one level of memory subsystem hierarchy. In GPGPU the memory subsystem hierarchy is developed less compared to CPU-type processors, so the roofline model appears to be the most suitable for GPGPU. In it, the peak achievable performance is shown against the operational intensity parameter. Operational intensity is calculated as the ratio of performable operations for the given amount of data to the size of this data. It is an attribute of the algorithm in use. Depending on this parameter, the roofline model subdivides algorithms into two groups: memory bound and compute bound. Naive FDTD method implementation with layerwise synchronization has low operational intensity, and is memory bound.

To increase performance it is necessary to increase operational intensity. It is possible by avoiding stepwise synchronization. This is the main idea of the alternative algorithms for stencil computations (trapezoids, time-slicing and time-skewing [15, 16, 19–23])

LRnLA method suggests tracing the dependencies of the numerical method to find the optimal computation order. The optimization is based on the hardware

model. In this work, the construction of the algorithms and advantages of this method is described on the example of DiamondTorre algorithm.

3.1 DiamondTorre algorithm and its CUDA implementation

We describe an algorithm as computation order that follows from the subdivision of 4D X-Y-Z-t computational region. Each mesh point in this space has dependencies on some amount of points on the previous layer. By subdividing the domain we find that some points are enclosed in a polytopes, the points of which have dependencies on the data from adjacent polytopes. Each polytope corresponds to a procedure of performing computations for all points within it in some arbitrary order. By tracing the dependencies, we find that some shapes have dependencies (direct or indirect), and some are asynchronous. The dependent shapes are to be processed subsequently, asynchronous ones may be processed in parallel. A set of such shapes constitutes the description of the algorithm. By generalization, one shape also comprises an algorithm. Inside some shape in 4D space we can estimate the operational intensity as amount of points in it (proportional to the amount of necessary operations) divided by the amount of separate spatial grid points in it (projection of the shape to 3D spatial grid, proportional to the amount of data necessary for operations).

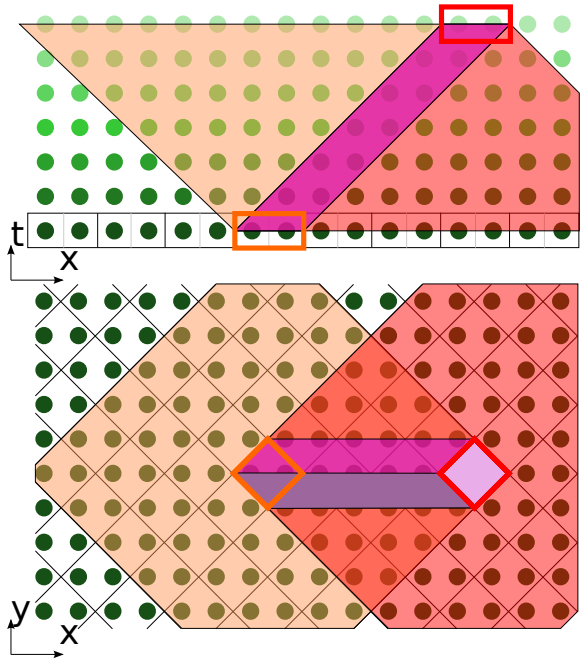

Fig. 2: The intersection of domain of influence and domain of dependency of two 2D diamond-shaped bases in 3D time-space

In 4D time-space the area of influence (and dependence) of some point is presented by a light cone. For FDTD numerical method, in the discrete 4D space of "spatial grid-time iteration" the areas of influence and dependence are presented by 4D pyramids with a 3D diamond shape in their base. The Diamond shape in the base of influence and dependency domains suggests that domain subdivision into the diamond-based shapes (DiamondTile) leads to the highest operational intensity. DiamondTiles are merged into "towers" (DiamondTorre, fig. 2). This decomposition is 3D in X-Y-t domain with 2D diamonds in the base. This way the whole 4D X-Y-Z-t computation-

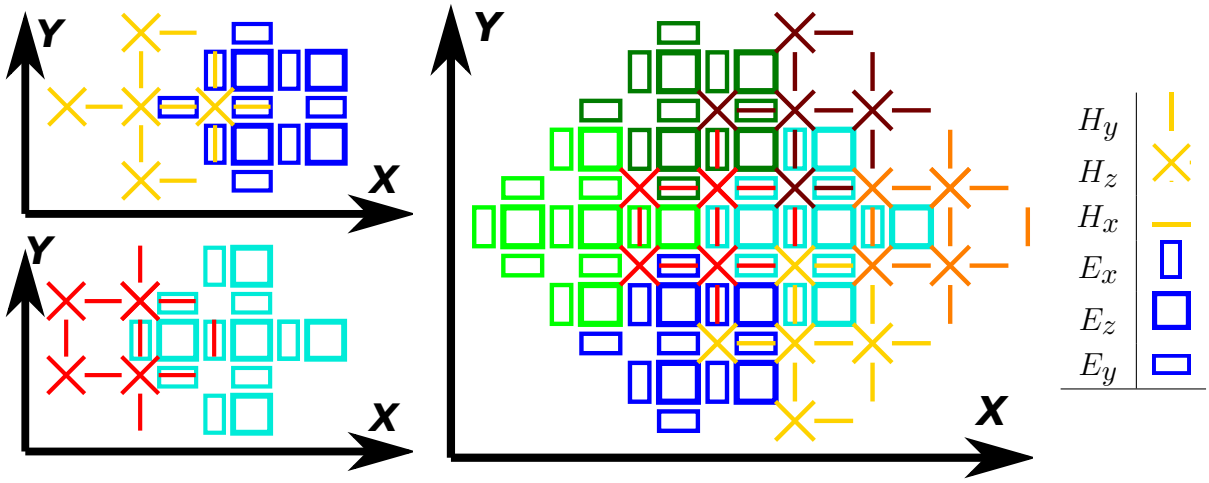

Fig. 3: Field components in one DiamondTile. Two options. (left) Dependencies of one DiamondTile (Type II). Suppose that one type II DiamondTile is loaded into GPU register. To calculate H-field diamond in it three E-field diamonds need to be loaded. To calculate E-field diamond in it three H-field diamonds (to the right, top, and bottom of it) need to be loaded (right).

al region can be tiled with one shape (except for boundaries). The pictured 3D shape spans in two spatial coordinates (X and Y) and one time axis. In CUDA implementation, one DiamondTorre is processed by CUDA-block. Grid points along the remaining coordinate axis (Z) are updated using vector parallelism of CUDA-threads in one block. DiamondTorre size is defined by two parameters: the size of the base `DTS`, and its height `TH`.

More detailed explanation of algorithm, its parameters and implementation for wave equation on 1 GPU can be found in [6]. Here the specifics of its implementation for FDTD method and many-GPU systems are described.

The correspondence of DiamondTile to components of staggered gird is illustrated on fig. 3. This is defined as the basic element for DiamondTile algorithms for FDTD scheme for Maxwell equations with 4th order of approximation and its size is `DTS=1` by definition. It consists of two (E-field and H-field) diamonds that are shifted along time axis (by half time step), as well as along one coordinate axis (by 1.5 spatial step - half width of the chosen scheme stencil). DiamondTorre consists of `TH` DiamondTile pairs, shifted in a similar manner against each other. We refer to the algorithm DiamondTorre as a process of making calculations for all points in the described 4D shape. In GPGPU implementation, DiamondTorre is a CUDA kernel. DiamondTorre's with the same Y-axis position are processed asynchronously by CUDA-blocks.

DiamondTorre base size `DTS`=1 is optimal in this case. Though higher `DTS` leads to higher operational intensity, with `DTS`=2 or more the main limiting factors are the size of register file and instruction cache. Significant performance

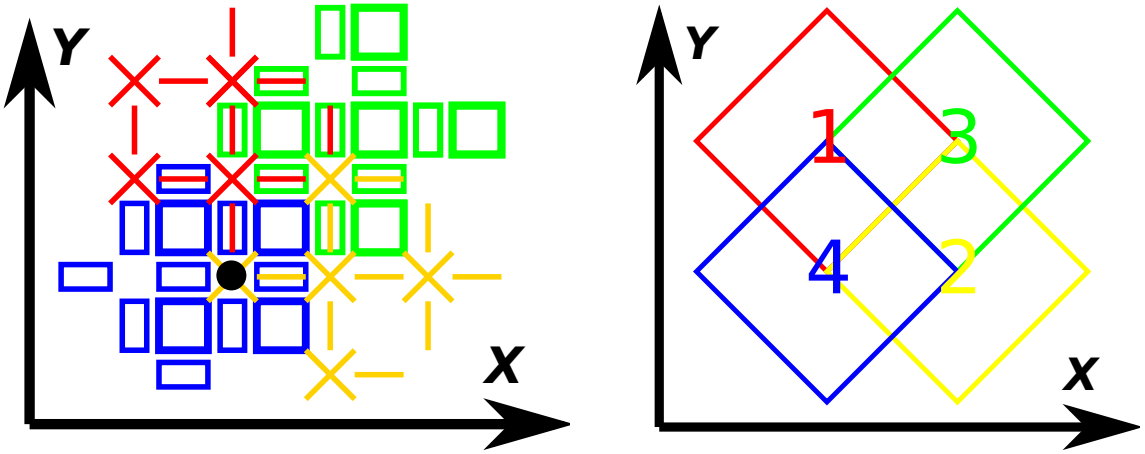

Fig. 4: One element of data structure (left). Each element consists of 4 diamonds (right)

drops are confirmed if the data on one DiamondTile exceeds GPU register file size, or if instruction count in one DiamondTorre kernel exceeds instruction cache size.

The register size is taken as 256 KB. If we choose the number of grid points along Z axis as $Nz = 384$ (double precision) or $Nz = 768$ (single precision), the limit of registers per CUDA-thread is estimated as $256K/(768 \cdot 4) = 85$. This roughly corresponds to the amount of data for `DTS`=1. Also, in this case, one CUDA kernel corresponding to DiamondTorre algorithm contains ~ 1500 instructions. One instruction takes ~ 8 byte, which brings us very close to the instruction cache limit (estimated as 16 Kb).

If we assume that one DiamondTile is loaded into GPU register (fig. 3), to calculate one H-field diamond three E-field diamonds need to be loaded. The resulting H-fields need to be saved. It concludes one half of cell updates in Diamond, for the other half three H-field diamonds are loaded and one H-field diamond is calculated and saved. According to this an estimate of data throughput can be made. With `TH` $\to \infty$, on average, 3 Yee cell data are to be loaded, and 1 Yee cell data should be saved per one Yee cell update. This amounts to 4×6(field in a Yee cell) $\times 8$(byte per field value, double precision) $= 192$ bytes. Operation count for one cell update is about 110 Flop. Operational intensity is obtained as their ratio, 0.57 Flop/Byte.

For any contemporary GPU this problem is memory-bound. The theoretical performance is estimated as $P/192$ Yee cell updates per second, where P is GDDR5 memory bandwidth, 192 is the necessary data throughput, estimated earlier. For example, with NVidia Tesla K20 ($P = 224 \cdot 10^9$ bytes per second) this amounts to $1.167 \cdot 10^9$ Yee cell updates per second (Ys).

Data is stored in linear 2D array, where each element is a set of vectors with length Nz (the amount of points along z-axis). One element is shown on fig. 4.

Such data structure is chosen to provide coalesced data access for the chosen computation algorithms, and to minimize the amount of elements in data load/store operations.

3.2 Calculation window

The important advantage of DiamondTorre algorithm is high performance for large simulation domains, including those, where field data do not fit in device memory. It is easily achieved by updating data in a "calculation window", which moves from right to left (according to the figures above). Data load and save to/from global RAM are performed asynchronously with computations. Only the data for one following group of asynchronous DiamondTorres are loaded. Data of DiamondTorres which are no longer necessary for the current `TH` update are saved and deleted from device memory.

Performance does not decline in case the computation time of DiamondTorre is longer than the time, necessary for memory copy to/from device. Since computation time increases linearly with `TH`, and the copy time is constant, with high enough `TH` host-device transfers are completely concealed. If we do not account for the boundary effects, operational intensity increases linearly with `TH`.

3.3 Small scale performance tests

The described algorithms are implemented in code, which features not only the basic FDTD stencil computations, but also all the required methods for real physics computations.

- FDTD simulation in 3D spatial domain with 4th order accurate scheme in space;
- Perfectly Matched Layer absorbing boundary conditions;
- Total Field/Scattered Field wave source;
- Complex materials according to Drude, Drude-Lorenz model.

Small scale performance tests were conducted to find optimal algorithm parameters. We measure performance in Yee cell updates per second (Ys). This number is usually of 10^9 order, so the main unit is GYs.

Fig. 5 shows the performance results for a problem size $(600 \times (3 \times blocks) \times 384)$, with varied "blocks" parameter. For DiamondTorres lined up along Y axis, one CUDA-block performs computation for one DiamondTorre. The tests were conducted on Tesla K20x. It has 14 streaming multiprocessors (SMs). If the number of involved SMs per device is less than 14, the performance is limited

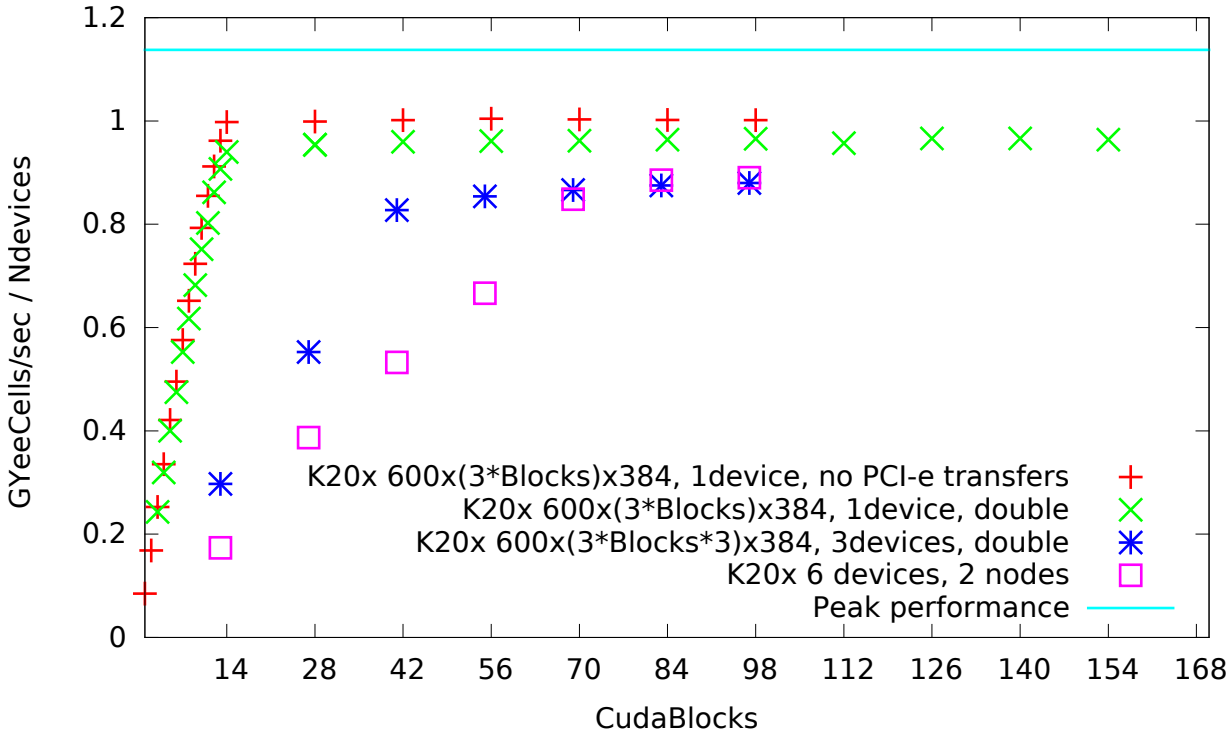

Fig. 5: Performance results for different values of blocks

by GDDR5 access latency. The latency is about 500 clock cycles (about 500 ns). From the Little's law [24], to cover this latency, at least 10^4 transactions are necessary.

Maximum vector size (Nz, also equal to the amount of involved CUDA-threads), for which the register file of Kepler architecture is enough to keep the data of all necessary diamonds, is equal to 384 for double precision. This is why the data throughput is not utilized completely when all 14 SMs are involved ($\sim 14 \cdot 384 \approx 6000$ transactions). It is the main reason why the performance is only 90% from peak one. It becomes more than 10^9 GYs for sufficiently large `TH`.

`TH` is inevitably smaller near region boundaries, so the performance for real problems is slightly lower (few percent).

Performance results for different values of `TH` parameter are shown on fig. 6. Since `TH` parameter is proportional to the algorithm locality, and the vertical axis is performance (normalized by the maximum achieved with the current parameter set), we may plot the roofline model on the same graph to visually comprehend the limitations. For low `TH` performance is limited by PCI-express bandwidth. With the increase of `TH` the shift to limitation of GDDR5 bandwidth is confirmed. Its smoothness is explained by the cut-off of DiamondTorres at the edges of the domain.

The optimal sufficient `TH` is 100, as can be seen on the graph. This value is used in subsequent tests, if not stated otherwise. Similar behaviour of the dependency

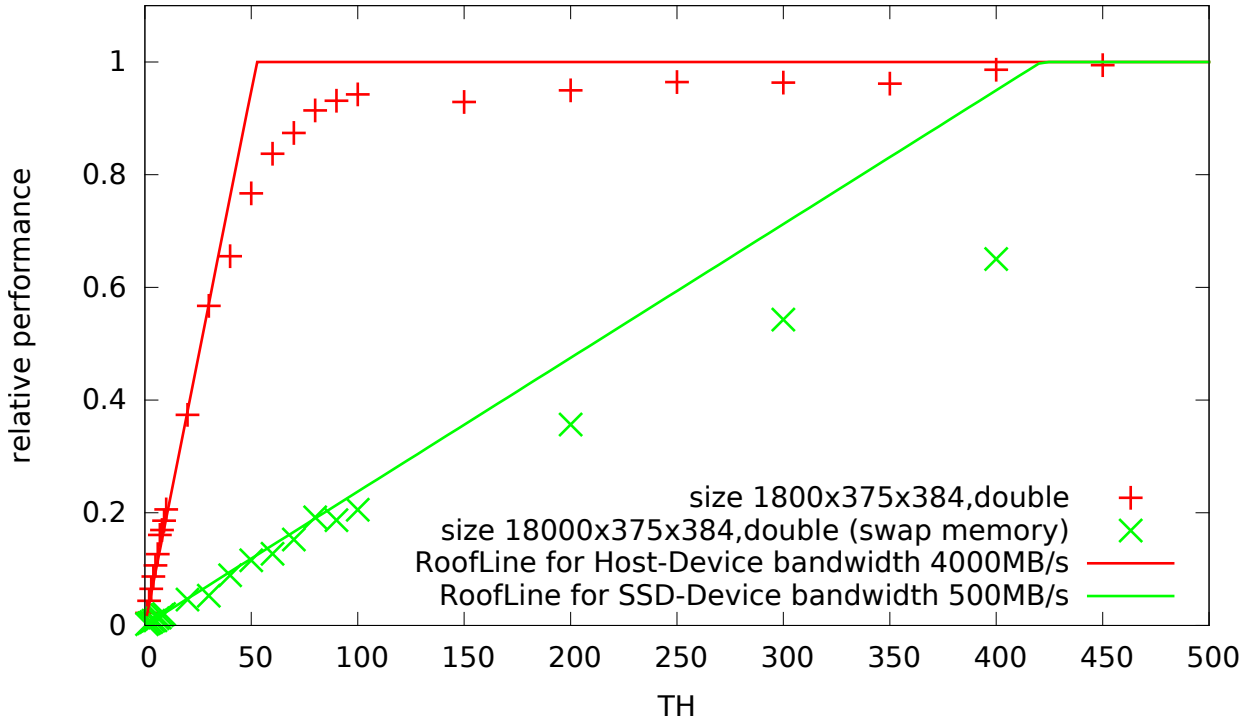

Fig. 6: Performance results for different values of TH

of performance rate on `TH` is observed in case the data is stored on disk (SSD in this case). The optimal `TH` becomes significantly larger (more than 500).

3.4 Concurrency on Y axis

Since all DiamondTorre's standing side-by-side along Y axis are asynchronous, they may be processed by different devices inside one node, as well as by different nodes. (fig. 7) Exactly 2 DiamondTorre's that belong to different devices overlap. The common DiamondTorre is processed by one device in a separate CUDA stream. After this the data are copied to the adjacent device in the same stream. Generally, data are sent through PCI-a and RAM (if Nvidia Peer-to-Peer Memory Access or GPUDirect RDMA technologies are not supported). In this case the data is written to buffer in one continuous block, which will be unpacked on the other device in a similar way. This operation takes just as long as one DiamondTorre processing, plus data transfer time. In this case it is easy to estimate the number of asynchronous DiamondTorre (NA) on one device so that the transfer between devices will be completely covered: $NA > 2 \cdot Nsm + Tsend/Tcalc$. Here Nsm is the amount of Streaming Multiprocessors (SM), Tsend is the time for one DiamondTorre's data transfer between devices, Tcalc is the time of calculating Nsm asynchronous DiamondTorre's.

It may be deduced from previous tests (fig. 5) that data transfers between 3 devices on one node may be concealed completely if the amount of involved CUDA-blocks is more than 42 on each device. For devices installed on different nodes the duration of data transfer `Tsend` is approximately 2 times bigger. In this case data transfers may be concealed completely if the amount of involved CUDA-blocks is more than 70 on each device.

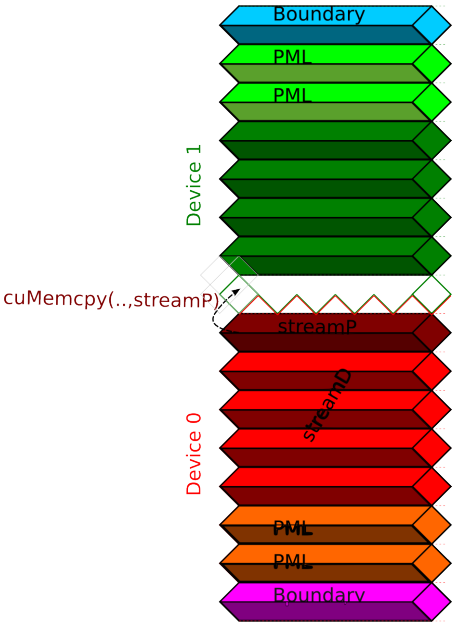

Fig. 7: Asynchronous DiamondTorres. Different colors correspond to different CUDA kernels, which are possibly executed on different devices

3.5 Concurrency on X axis

Additionally, concurrency on X axis is possible. In this case the data are subdivided in blocks in X axis. Data on adjacent nodes overlap by the calculation window size: NW points of X axis (fig. 8). On i-th device data are updated from $(n + i \cdot TH)$-th step to $(n + i \cdot TH + \mathrm{TH})$-th step, where n is an integer number — the time step on which the data of the leftmost node exists on.

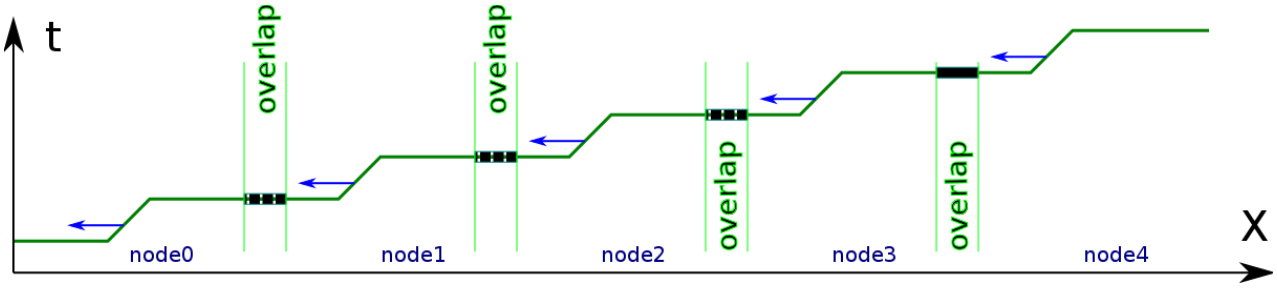

Fig. 8: Concurrency on X axis

Here is a more detailed explanation of the computation and data transfer algorithm. Each node, which has data that does not include the boundary, processes the following operations sequentially (fig. 9, 10):

- Wait to receive overlapping data on the right side from the node on the right (waitR);
- Computing cell updates for the received overlapping data (calcR);
- Non-blocking data transfer to the node on the right (sendR);

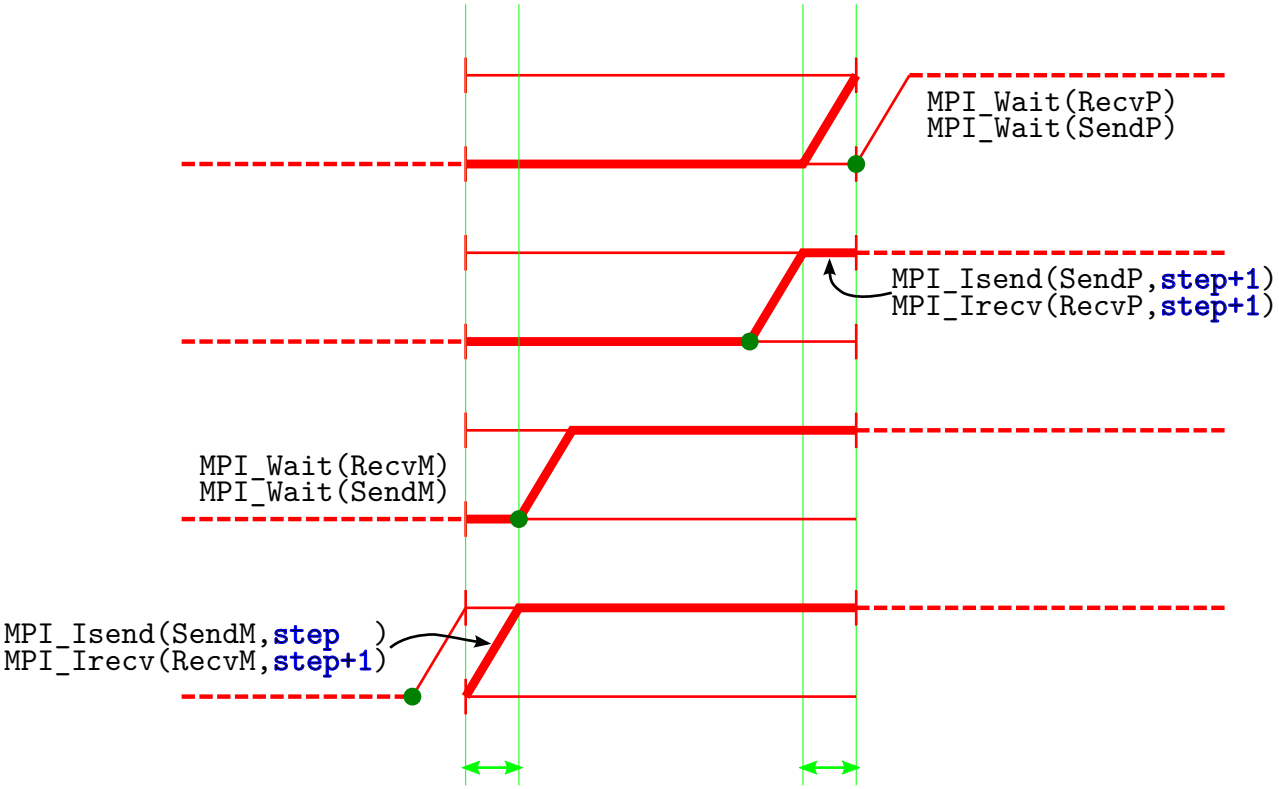

Fig. 9: Key stages of parallel decomposition in x-t domain

- Computing cell updates for data, which does not depend on the overlapping regions (calcM);
- Wait to receive overlapping data on the left side from the node on the left (waitL); these are sent by sendR step of the node to the left;
- Computing cell updates for the overlapping data to the left (calcL);
- Non-blocking data transfer to the node on the left (sendL); these are received during the waitR step of the node to the left.

The dependencies between $(i-1)$-th, i-th and $(i+1)$-th nodes of the described stages are depicted on fig. 10.

In case the execution of calcM takes more time than data transfer time (sendR+sendL), data transfer between nodes will be completely hidden by computations.

4 Parallel scaling results on TSUBAME2.5

The parallel performance of the code has been tested on TSUBAME2.5 supercomputer. Its specifications, that are important for this study, are as follows

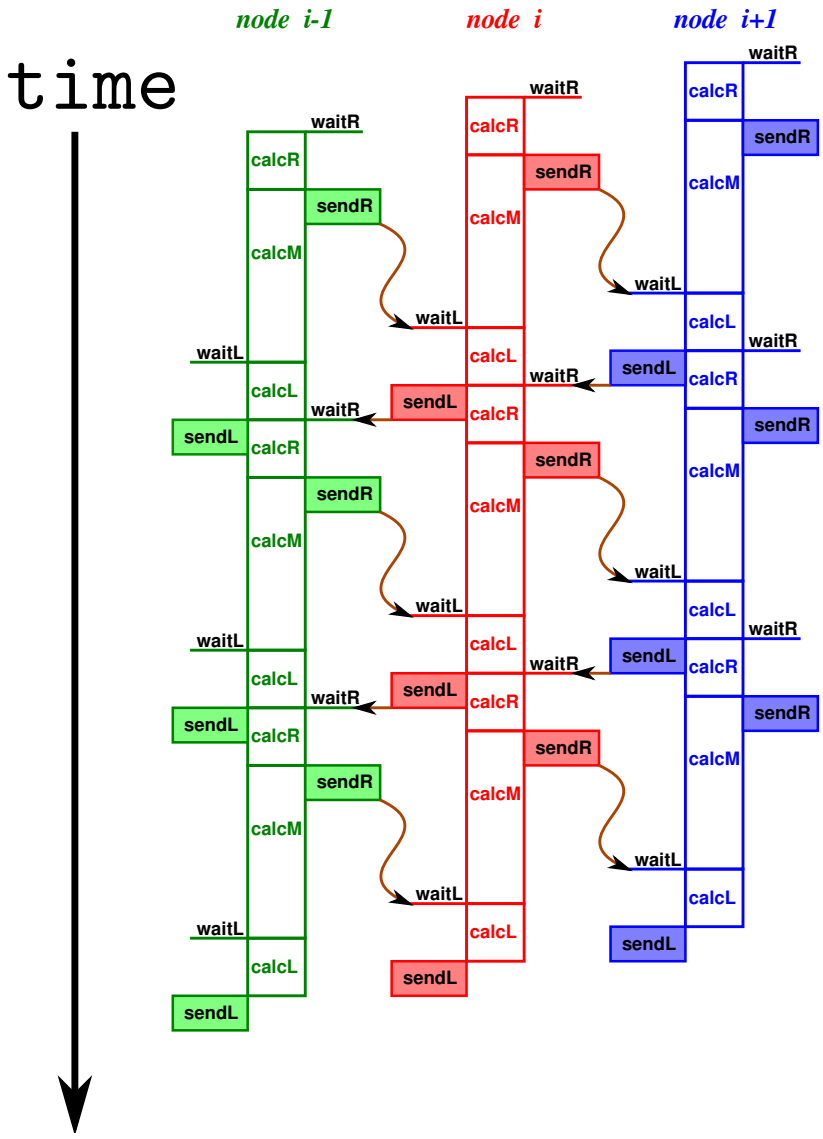

Fig. 10: Time sequence of parallel execution on nodes $i-1$, i, $i+1$

- The amount of available node per one run is up to 300 according to the usage conditions (1403 in total).
- Each node has 3 NVIDIA Tesla K20x GPGPUs installed. Their total GDDR5 memory is $3 \times 5.625 = 16.875$ GB, with 208 GB data throughput each.
- Each node has at least 54 GB, up to 96 GB on several ones. Only a little above 40 GB from it is available for the computation.
- Devices are connected with PCI-e 2.0 with 4 GB/sec throughput in each direction.
- Each node has 120 GB SSD memory.
- Nodes are connected by Infiniband QDR interconnect with up to 4GB/sec throughput.

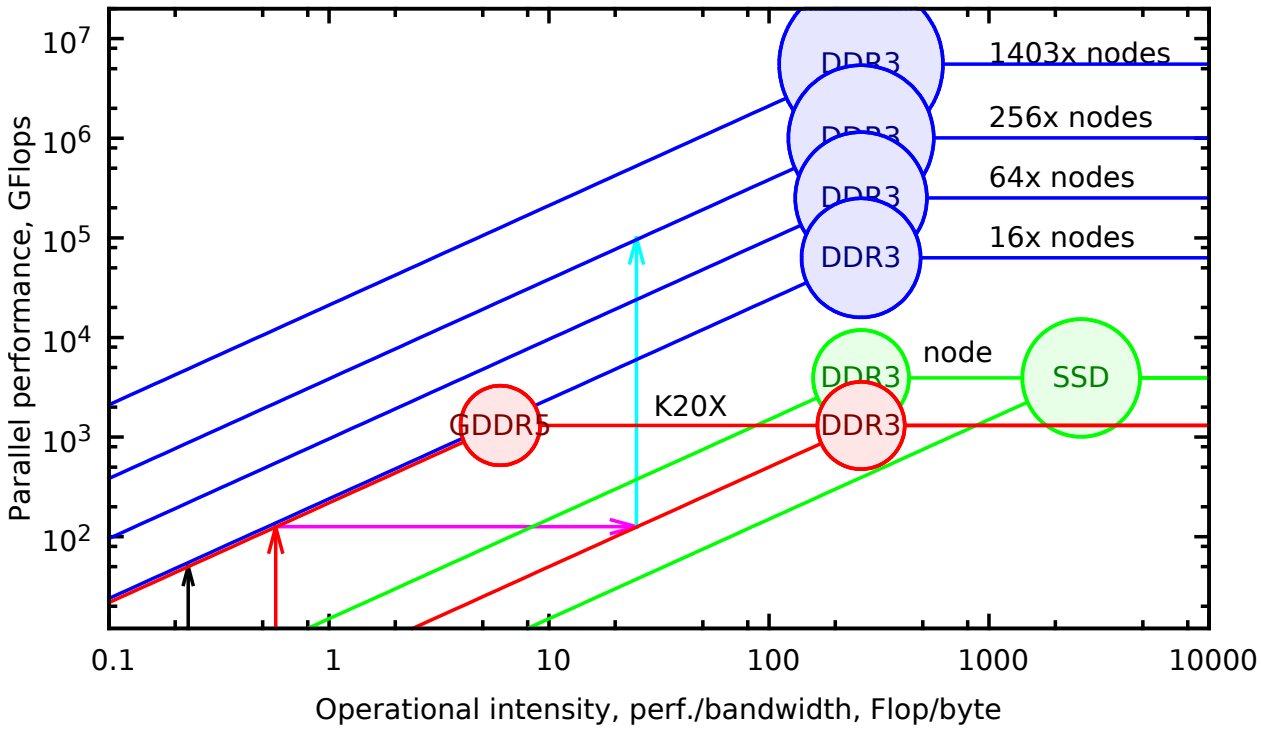

Fig. 11: Roofline graph for TSUBAME2.5

5 Roofline estimation

We plot the roofline graph for the system in use (fig. 11). The memory subsystem is hierarchical, and LRnLA algorithms are constructed with this hierarchy in mind. This is why we plot rooflines for several main memory levels. We also need to account for the size of the level, so the graph is expanded by adding the circles. The size of the circle shows the data size of the memory level. It is positioned on the corner of the corresponding roofline.

Red lines show the roofline for one GPGPU device. Both device memory and node memory may be used, so they both are shown on the roofline. One node (green lines) has 3 devices, so its peak performance is shown to be 3 times higher.

Up to 256 nodes were used in the performance tests, and the whole cluster contains 1403 nodes (blue lines).

The operational intensity of algorithm described in section 3.1 is estimated as 0.57 (red arrow on fig. 11). For this algorithm we take account only of the inner memory of the GPU device. This simple estimate is made by taking $Nt \to \infty$. The difference from the precise number is marginal.

For comparison the operational intensity of the stepwise naive algorithm (no data reuse) is estimated to be 0.23 (black arrow).

When node memory is used the performance is bound by the host-device memory bandwidth (lower red roofline). When the calculation window is used

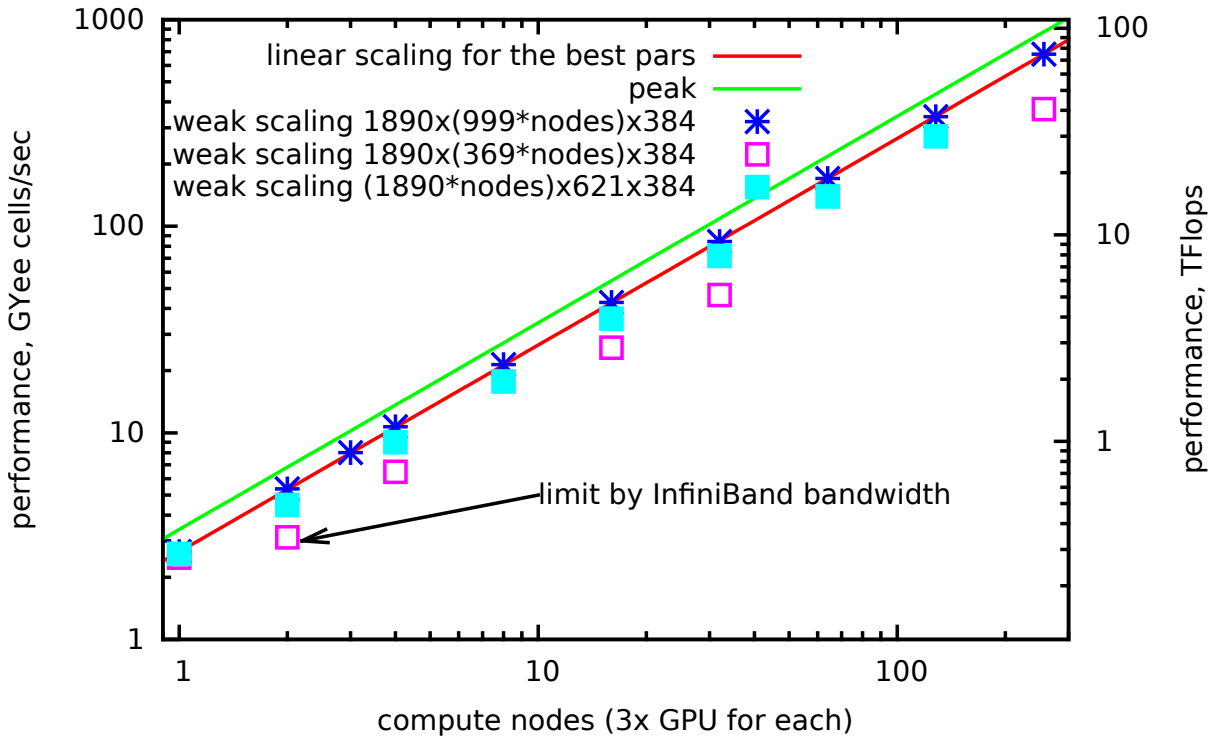

Fig. 12: Weak scaling

the operational intensity rises with `TH` parameter (purple horizontal arrow). The arrow stops at the host-device roofline, since the further shift to the right is not necessary, as the limit of the intra-device algorithm performace can not be overcome. With the use of many nodes, the theoretical performance scales linearly (light blue arrow). The only limiting factor is inter-node communication bandwidth.

5.1 Weak scaling

For weak scaling the domain is scaled proportionally to the amount of nodes used. Both X and Y axis parallelism were tested. Three series were performed:

1. Scaling in Y axis. Data transfers may be concealed completely (112 CUDA-blocks on each device).
2. Scaling in Y axis. Data transfers limit the performance (42 CUDA-blocks on each device, see fig. 5).
3. Scaling in X axis. Each device contains 1890 Yee cells. All 3 devices are involved on each node, performing parallel computation in Y direction.

The results are presented on fig. 12.

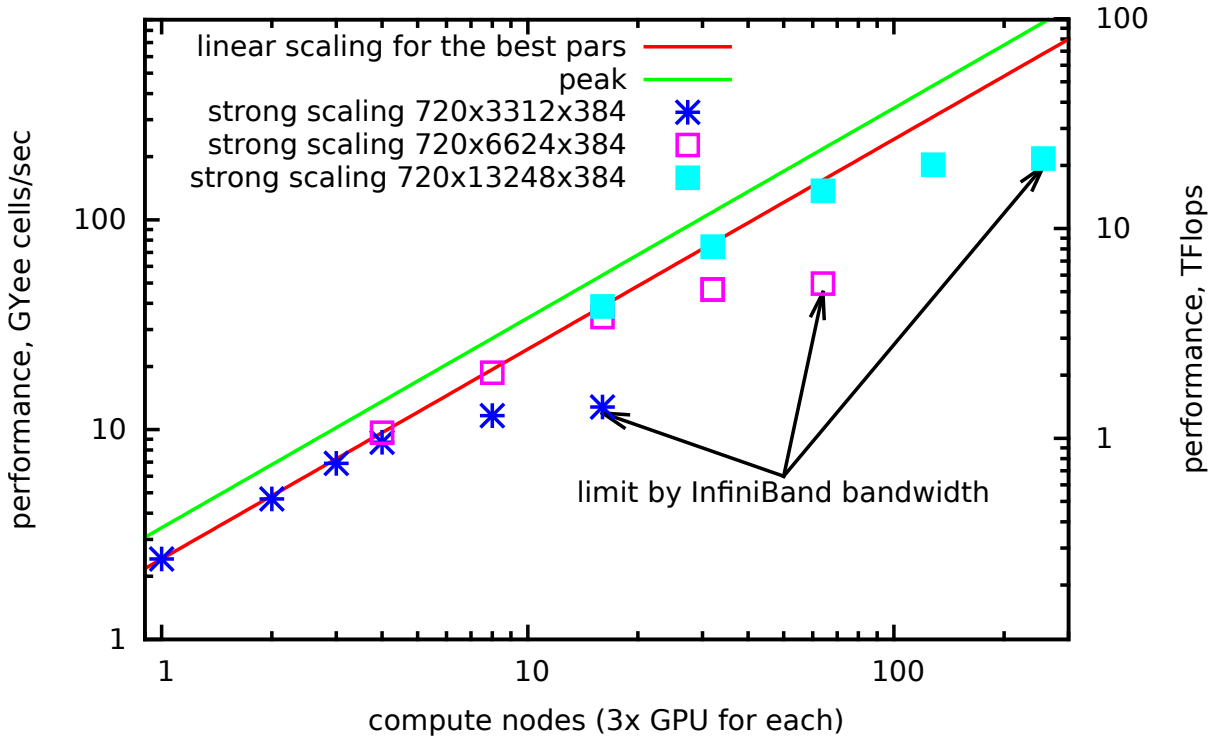

Fig. 13: Strong scaling series

In the first case, parallel efficiency is above 99% as expected. The maximum achieved performance is $0.65 \cdot 10^{12}$ Ys for one computation on domain with $300 \cdot 10^9$ grid points (10 TB data, 256 nodes).

In the second case the efficiency becomes lower when the number of nodes rises from 1 to 2. It is caused by the fact that Infiniband data throughput is lower than data throughput between devices on one node. The following increase in the number of nodes does not lower the efficiency.

In the third case parallel scaling is close to ideal, but is still less than the one for the first series. The main decrease occurs between 1 and 2 nodes. It is caused by a slight imbalance in node utilization.

5.2 Strong scaling

For strong scaling we chose a fixed size domain, and by increasing the amount of nodes, the domain is subdivided to more and more parts, that are to be processed concurrently. Three series were performed (fig. 13):

1. $720 \times 3312 \times 384$ size domain is scaled on 1–32 nodes;
2. 4 times bigger domain ($720 \times 13248 \times 384$) is scaled on 4–64 nodes;
3. 16 times bigger domain ($720 \times 52992 \times 384$) is scaled on 16–256 nodes;

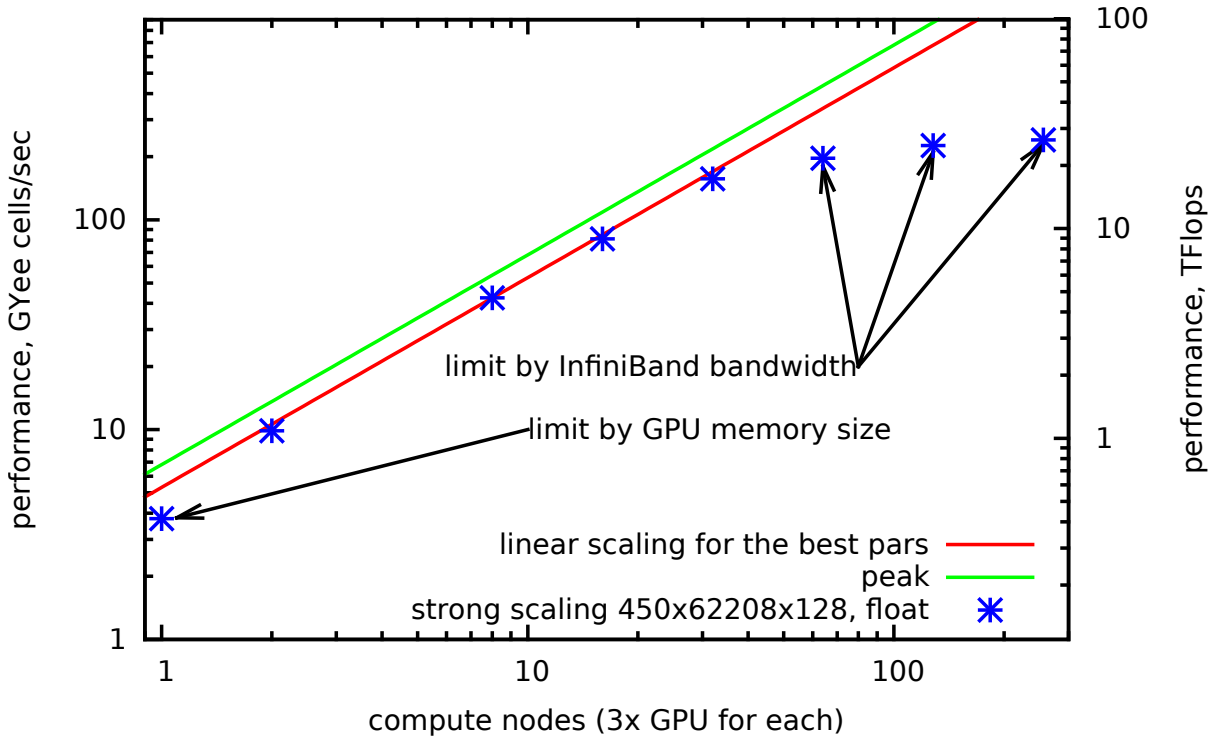

Fig. 14: Strong scaling on Y axis with variable TH

With the increase in amount of parallel nodes the performance decreases since the amount of CUDA-blocks per device becomes lower (see fig 5). At the same time there is a limit on maximum Ny size per device, determined by device memory. It actually may be increased, but the Nx size should be decreased at the same time, as well as DiamondTorre height `TH` with it. This would lower the general performance. By utilizing more nodes to include bigger domain `TH` may be increased again and performance rises.

This dependency on `TH` is shown on fig. 14. Problem size is $450 \times 62208 \times 128$ grid cells. For one node computation `TH` is equal to 15, and increases up to 150 for 8 nodes and higher. For low amount of nodes the speedup is better than linear. It is caused by optimization of `TH` which is only possible when enough data is processed on each node.

The final result is a strong scaling series on a problem with $38400 \times 363 \times 128$ grid points (fig. 15). One node the performance is about 30% from the peak performance, since the size along z axis is not optimal. It is not big enough to conceal GDDR5 access latency, since the amount of simultaneous transactions is too low. But the acceleration is up to 40 times. Only with 128 nodes and above data transfers are taking more time than computation, which leads to decrease in computation rate.

It should be noted that one node has little memory size (only about 3 times more than total device memory), and this becomes the reason for acceleration

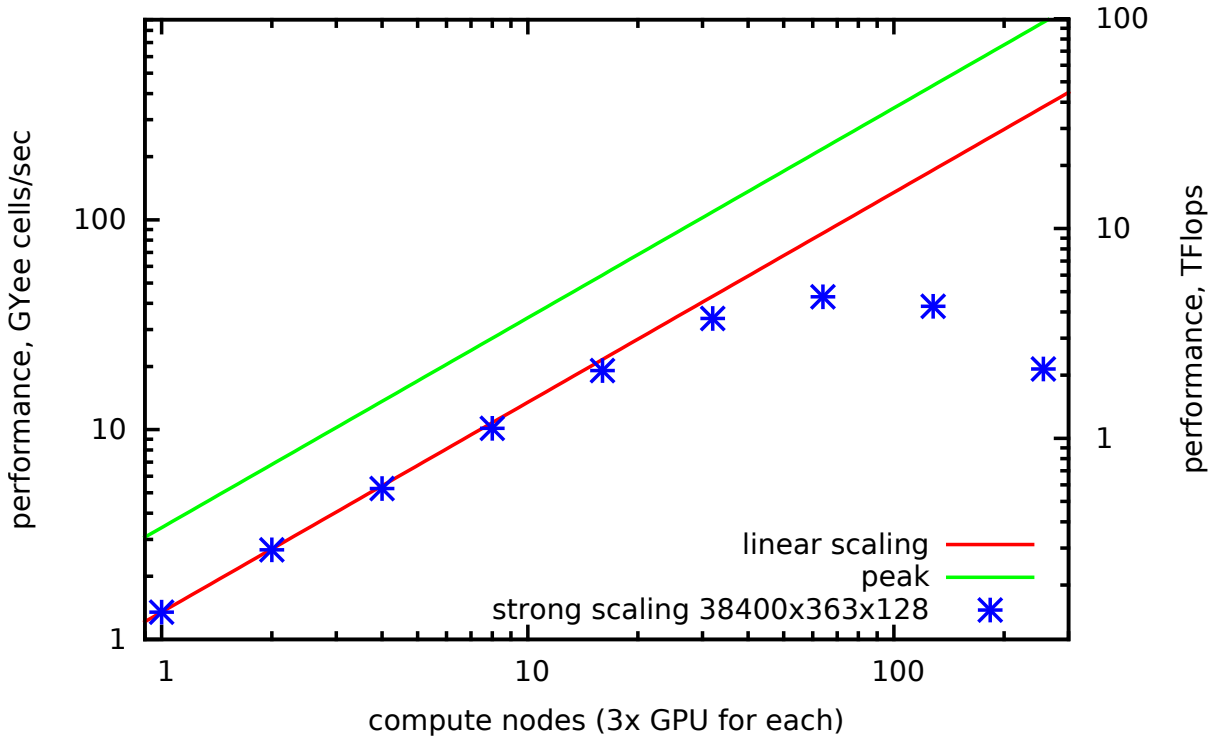

Fig. 15: Strong scaling on X axis

limit. The increase in available memory size should increase acceleration ability for scaling in X axis proportionally.

6 Conclusion

The work can be summarized as follows. The FDTD code has been developed, that allows simulation of real optical phenomena. The distinguishing feature of the code is the use of DiamondTorre LRnLA algorithm, which maximizes the performance on one device, and parallel efficiency for multi-GPU architectures. The software was tested on TSUBAME2.5 supercomputer. High computation rate is achieved (more than 1 billion Yee cell updates per second on one device). The problem size is not limited by device memory. The scaling for ~ 1000 devices is linear for weak scaling, and saturates at ~ 100 for strong scaling.

Algorithm parameters (such as `TH` and problem size) allow not only qualitative, but also quantitative estimates of the performance and parallel scaling. Maximal achieved performance is $0.65{\cdot}10^{12}$ Yee Cells per second for 3D domain with $0.3{\cdot}10^{12}$ Yee cells total. For example, such size for wave optics problems corresponds to 1 cubic millimeter domain. This allows a significant breakthrough in computational nanooptics, by allowing the simulation in domains that were previously too big even for supercomputers. It may be used for simulation of complex optical devices

and substrates.

7 Acknowledgement

The work is supported by Hosei International Fellowship grant, RFBR grant no. 14-01-31483.

References

[1] A. Taflove and S. C. Hagness, *Computational Electrodynamics: the Finite-Difference Time-Domain Method*. Norwood, MA: Artech House, 3rd ed., 2005.

[2] J. Virieux, "P-sv wave propagation in heterogeneous media: Velocity-stress finite-difference method," *Geophysics*, vol. 51, no. 4, pp. 889–901, 1986.

[3] V. Levchenko, "Asynchronous parallel algorithms as a way to archive effectiveness of computations (in russian)," *J. of Inf. Tech. and Comp. Systems*, no. 1, p. 68, 2005.

[4] A. F. Oskooi, D. Roundyb, M. Ibanescua, P. Bermelc, J. D. Joannopoulosa, and S. G. Johnson, "Meep: A flexible free-software package for electromagnetic simulations by the fdtd method," *Computer Physics Communications*, vol. 181, pp. 687–702, 2010.

[5] "Acceleware ltd.." `http://www.acceleware.com/`. Accessed: 2016-04-18.

[6] A. Y. Perepelkina and V. D. Levchenko, "Diamondtorre algorithm for high-performance wave modeling," *Keldysh Institute Preprints*, vol. 18, p. 20, 2015.

[7] A. Y. Perepelkina, I. A. Goryachev, and V. D. Levchenko, "CFHall code validation with 3D3V weibel instability simulation," *Journal of Physics: Conference Series*, vol. 441, no. 1, p. 012014, 2013.

[8] A. Y. Perepelkina, I. A. Goryachev, and V. D. Levchenko, "Implementation of the kinetic plasma code with locally recursive non-locally asynchronous algorithms," *Journal of Physics: Conference Series*, vol. 510, no. 1, p. 012042, 2014.

[9] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," in *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, (New York, NY, USA), pp. 30–44, ACM, 1991.

[10] M. Wolfe, "More iteration space tiling," in *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, Supercomputing '89, (New York, NY, USA), pp. 655–664, ACM, 1989.

[11] M. Wolfe, "Loops skewing: The wavefront method revisited," *International Journal of Parallel Programming*, vol. 15, no. 4, pp. 279–293, 1986.

[12] A. Terrano, "Optimal tilings for iterative pde solvers," in *Frontiers of Massively Parallel Computation, 1988. Proceedings., 2nd Symposium on the Frontiers of*, pp. 227–229, Oct 1988.

[13] H. Prokop, "Cache-oblivious algorithms," 1999.

[14] M. Frigo and V. Strumpen, "Cache oblivious stencil computations," in *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, (New York, NY, USA), pp. 361–366, ACM, 2005.

[15] R. Strzodka, M. Shaheen, D. Pajak, and H.-P. Seidel, "Cache accurate time skewing in iterative stencil computations," in *Proceedings of the International Conference on Parallel Processing (ICPP)*, p. 571–581, IEEE Computer Society, IEEE Computer Society, sep 2011.

[16] R. Strzodka, M. Shaheen, D. Pajak, and H.-P. Seidel, "Cache oblivious parallelograms in iterative stencil computations," in *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, (New York, NY, USA), pp. 49–59, ACM, 2010.

[17] N. V. Kantartzis and T. Tsiboukis, *Higher Order FDTD Schemes for Waveguide and Antenna Structures*. Morgan & Claypool Publishers, 1st ed., 2006.

[18] S. Williams, A. Waterman, and D. A. Patterson, "Roofline: an insightful visual performance model for multicore architectures.," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[19] M. Frigo and V. Strumpen, "The memory behavior of cache oblivious stencil computations," *The Journal of Supercomputing*, vol. 39, no. 2, pp. 93–112, 2007.

[20] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson, "The pochoir stencil compiler," in *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, (New York, NY, USA), pp. 117–128, ACM, 2011.

[21] T. Grosser, A. Cohen, P. H. J. Kelly, J. Ramanujam, P. Sadayappan, and S. Verdoolaege, "Split tiling for gpus: Automatic parallelization using trapezoidal tiles," in *Proceedings of the 6th Workshop on General Purpose Pro-*

cessor Using Graphics Processing Units, GPGPU-6, (New York, NY, USA), pp. 24–31, ACM, 2013.

[22] D. Orozco and G. Gao, "Mapping the fdtd application to many-core chip architectures," in *Parallel Processing, 2009. ICPP '09. International Conference on*, pp. 309–316, Sept 2009.

[23] J. McCalpin and D. Wonnacott, "Time skewing: A value-based approach to optimizing for memory locality," tech. rep., In http://www.haverford.edu/cmsc/davew/cache-opt/cache-opt.html, 1999.

[24] J. D. C. Little, "A proof for the queuing formula: L = lw," *Operations Research*, vol. 9, no. 3, pp. 383–387, 1961.