Keldysh Institute • Publication search

Keldysh Institute preprints • Preprint No. 18, 2015

Perepelkina A.Yu., Levchenko V. D.

DiamondTorre Algorithm for High-Performance Wave Modeling

Recommended form of bibliographic references: Perepelkina A.Yu., Levchenko V. D. DiamondTorre Algorithm for High-Performance Wave Modeling // Keldysh Institute Preprints. 2015. No. 18. 20 p. URL: **http://library.keldysh.ru/preprint.asp?id=2015-18&lg=e**

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

A. Perepelkina, V. Levchenko

DiamondTorre Algorithm for High-Performance Wave Modeling

Москва
2015

УДК 519.688

А. Ю. Перепёлкина, В. Д. Левченко

Алгоритм DiamondTorre для высокопроизводительного волнового моделирования

Аннотация. Обсуждаются эффективные алгоритмы решения задач численного моделирования физических сред, темп вычислений в которых для традиционных алгоритмов ограничен пропускной способностью памяти. Рассматривается численное решение волнового уравнения при помощи конечно-разностных схем с явным шаблоном типа «крест» высокого порядка аппроксимации. Построен алгоритм DiamondTorre, учитывающий особенности иерархии памяти и параллельности графических процессоров общего назначения (GPGPU). Преимуществами алгоритма является высокий уровнь локализации данных, а также свойства асинхронности, позволяющие эффективно задействовать все уровни параллелизма GPGPU. Вычислительная интенсивность алгоритма превышает соответствующее значение для лучших алгоритмов с пошаговой синхронизацией, а результате становится возможным преодоление указанного выше ограничения. Алгоритм реализован в рамках модели программирования CUDA. Для схемы второго порядка аппроксимации получена производительность более 50 миллиардов ячеек в секунду на одном устройстве, что в 5 раз превосходит результаты оптимизированного алгоритма с пошаговой синхронизацией.

A. Perepelkina, V. Levchenko

DiamondTorre Algorithm for High-Performance Wave Modeling

Abstract. Effective algorithms of physical media numerical modeling problems solution are discussed. Computation rate of such problems is limited by memory bandwidth if implemented with traditional algorithms. The numerical solution of wave equation is considered. Finite difference scheme with cross stencil and high order of approximation is used. The DiamondTorre algorithm is constructed, with regard for the specifics of GPGPU's (general purpose graphical processing unit) memory hierarchy and parallelism. The advantages of these algorithms are high level of data localization as well as the property of asynchrony, which allows to effectively utilize all levels of GPGPU parallelism. Computational intensity of the algorithm is greater than the one for the best traditional algorithms with stepwise synchronization. As a consequence, it becomes possible to overcome the above-mentioned limitation. The algorithm is implemented with CUDA.For the scheme with second order of approximation the calculation performance of 50 billion cells per second is achieved, which exceeds the result of the best traditional algorithm by a factor of 5.

1 Introduction

On the path to the exascale computations there seems to be a certain stagnation of algorithm ideas. Let us try to analyze the most prominent issues of contemporary computations on the example of wave modeling.

The first issue is the inability to operate on all levels on parallelism with maximum efficiency. It may be solved for some software (testing packages alike LAPACK [1] being the evident example), but remains an open question for the larger part of relevant problems. Also, only one programming instrument is not enough for this. As the supercomputer performance is mostly increased by adding parallelism, the modern Top500 computers [2] are essentially heterogeneous and, as a rule, include GPGPU. The peak performance is achieved by using all levels of parallelism in some ideal way. The sad truth about the expensive supercomputers is that they mostly run the software that does not accomplish this requirement.

To efficiently utilize the supercomputing power the software should be written with account to the model of parallelism of a given system. For physical media simulation, the common methods of utilizing concurrency include domain decomposition [3]. It assumes decomposition into large domains, the data of which fits into the memory attached to a processor (coarse-grained parallelism). The common technology to implement it is MPI [4]. On the other hand, tiling [5] is often used in computer graphics and other applications. Data array is decomposed into small parts, which all lie in the same address space (fine-grained parallelism), and OpenCL [6] and CUDA [7] is a common technology for this. Similar technique is loop nest optimization [8], which arise when initially sequential programs are rewritten to be parallel and a dependency graph [9] has to be analyzed to find optimal algorithm. It is often executed with OpenMP [10] in coarse-grain and loop auto-vectorization [11] in fine-grain. With all parallel technologies, developers have to struggle with the issues of unbalance, large communication latency and low throughput, non-uniform data access [12], necessity of memory coalescing [7].

Second issue in applied computations that deal with processing large amount of data (such as wave modeling) is the deficiency in memory bandwidth. The balance of main memory bandwidth to peak computation performance is below 0.1 byte/flop for the majority of modern computers (see fig. 1), and tends to decrease even further as new computation units appear. To illustrate the scale of this decrease an archaic system NEC-SX is also shown on the graph. Legacy algorithms require the balance to be at near NEC-SX value (or greater) to reach maximal efficiency. In terms of hardware, the access to data takes more time and costs more (in terms of energy consumption) than the computation itself. To cope with this issue most systems have a developed memory subsystem hi-

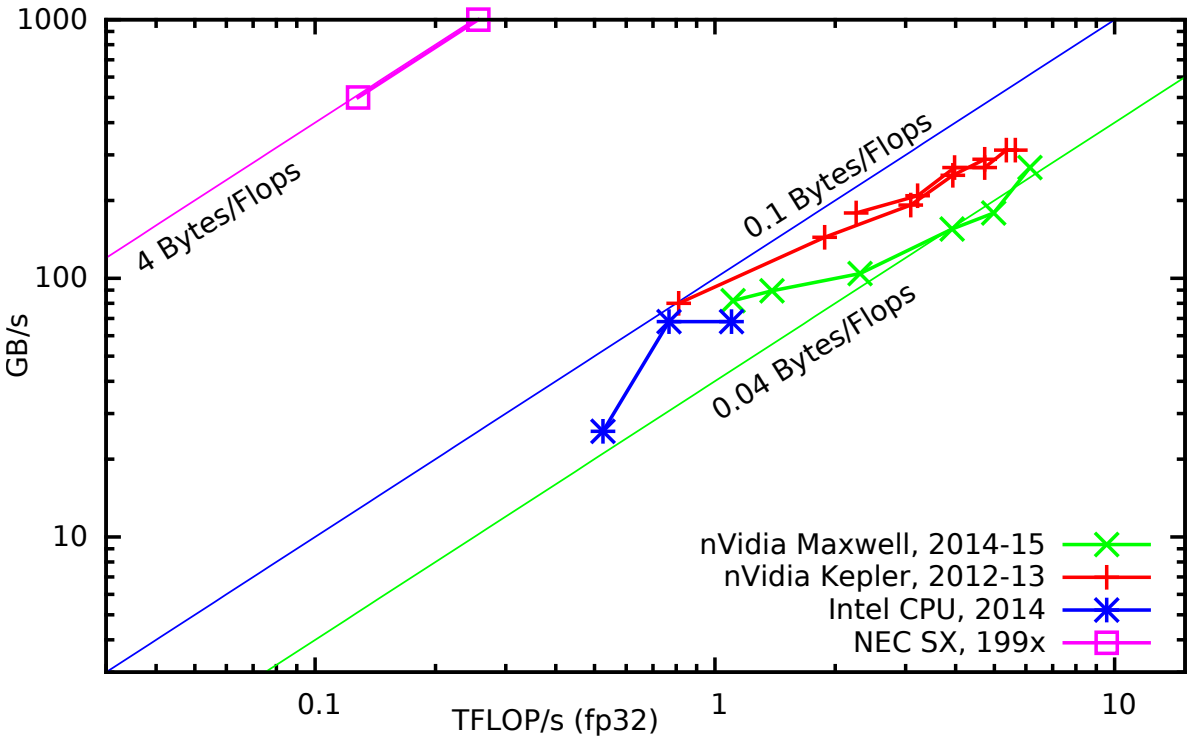

Figure 1: FLOPs and Bandwidth Performance Ratio

erarchy, and the program solutions should use the knowledge of it accordingly. There are several approaches that take this into account on CPU [13, 5]. On GPGPU, CUDA is an instrument that allows working with memory hierarchy levels conveniently [14, 15]

2 Computation models

With the hierarchy of memory subsystems and levels of parallelism, contemporary computers display an extreme complexity.

One helpful tool is a model of pyramidal memory subsystem hierarchy. In fig. 2 in a log-log scale we plot rectangles, which vertical position shows data throughput of the memory level, and the width of rectangle shows data set size. The picture looks like a pyramid for the CPU device. With each level the memory bandwidth is about twice higher and the data size is smaller by a factor of eight.

Locally Recursive non-Locally Asynchronous (LRnLA) algorithms [16, 17] use the divide and conquer approach by decomposing the problem into subproblems recursively in several levels, so that each time the data of subproblems fits into higher level of memory hierarchy. It allows to reach peak performance for the problems, the data of which is as large as the lower level of memory subsystem.

Here we also see that register memory of GPGPU is larger than on CPU, so

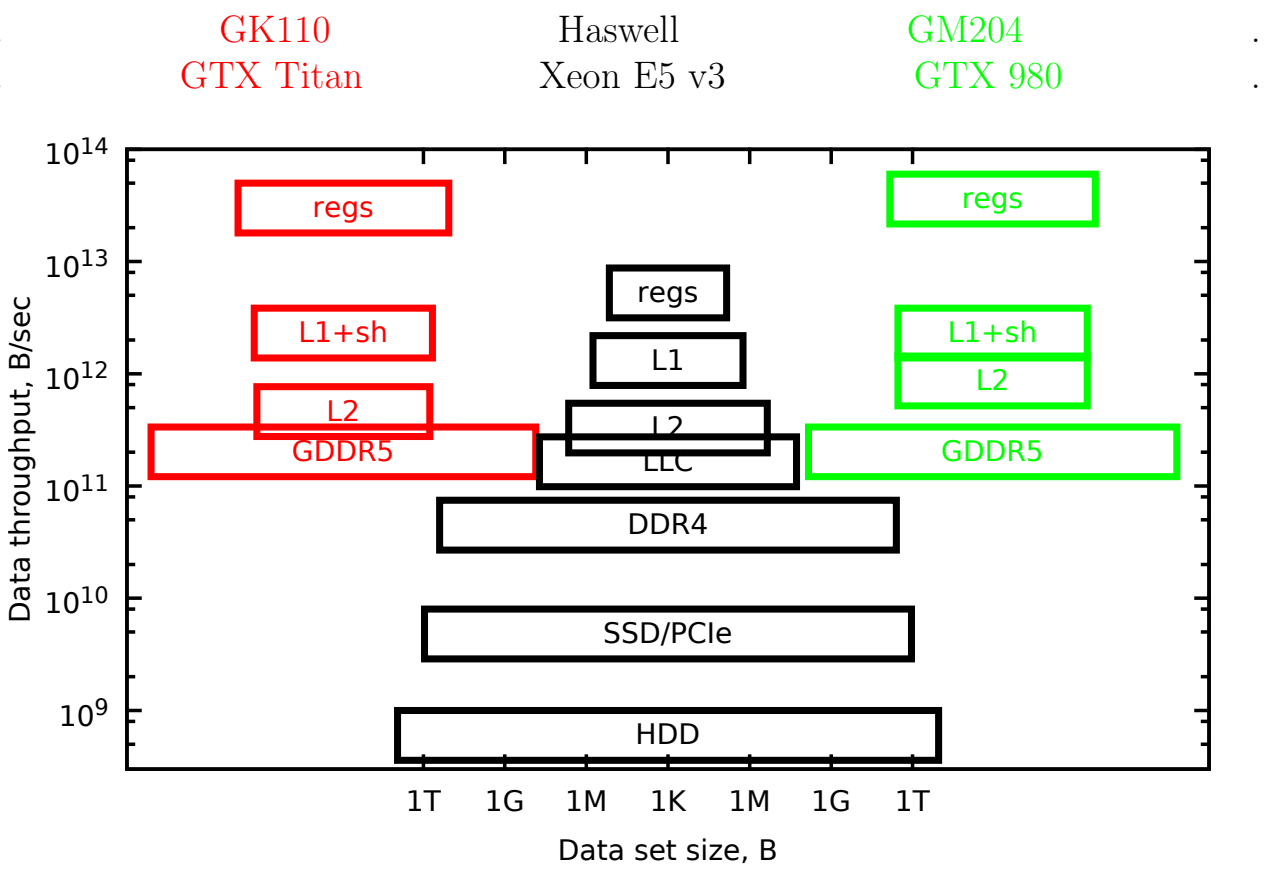

Figure 2: Memory subsystem hierarchy for GPGPU and CPU

it may be used as a main data localization site. This register file is distributed between several multiprocessors. The next significant memory level (GDDR5) has worse memory bandwidth and latency. So instead of recursive decomposition it is important to provide a continuous data flow from device memory to registers and back. In this work we will construct DiamondTorre algorithm which meets this criteria.

The hierarchy is best known to hardware designers, but in the same time it is unacceptable to ignore it in programming. For making the software the complex structure should be simplified as some model. One example of such model is roofline. Introduced in [18] the roofline model is a graph of attainable GFlops per second productivity versus operational intensity. It has two distinct portions, and visually assorts the programs into two categories based on the operational intensity: memory-bound and compute bound. The higher the ceiling rises (this corresponds to increase in peak performance) the more problems fall under the slope and suffer from memory bandwidth limitations.

Table 1: Numerical scheme coefficients

N_O	C_0	C_1	C_2	C_3	C_4
2	-1	1	—	—	—
4	$-5/4$	$4/3$	$-1/12$	—	—
6	$-49/36$	$3/2$	$-3/20$	$1/90$	—
8	$-205/144$	$8/5$	$-1/5$	$8/315$	$-1/560$

3 Problem statement

We deal with problems in numerical simulation of physics. The main scope of the current work is wave modeling, which encompasses a vast range of applications, such as modeling of: elastic media wave propagation, nanophotonics, plasmonics, acoustics. To be specific in the present paper we choose to limit the discussion to the second order acoustic wave equation, but the implications are easily generalized on all numerical methods with local stencil.

$$\frac{\partial^2 F}{\partial t^2} = c^2 \left(\frac{\partial^2 F}{\partial x^2} + \frac{\partial^2 F}{\partial y^2} + \frac{\partial^2 F}{\partial z^2} \right). \tag{1}$$

The problem is to compute the temporal evolution of a field $F = F(x, y, z, t)$ in a finite simulation domain with given initial $\left(F(x, y, z, 0) = F_0(x, y, z)\right)$ and boundary $\left(F|_{boundary} = U(t)\right)$ conditions. The explicit scheme has second order of approximation in time and adjustable order of approximation in space. That is, the mesh is introduced over the domain with Nx, Ny, Nz cells along each corresponding axis and the differentials are approximated by finite sums of the form:

$$\Delta x^2 \left. \frac{\partial^2 F}{\partial x^2} \right|_{x_0,y_0,z_0,t_0} = \sum_{i=0}^{N_O/2} C_i (F|_{x_0+i\Delta x,y_0,z_0,t_0} + F|_{x_0-i\Delta x,y_0,z_0,t_0}), \tag{2}$$

where N_O is the order of approximation (it is even), C_i are numerical constants (sample coefficients are given in table 1), Δ signifies a mesh step along the corresponding axis. Differentials in all four variables are expressed similarly. $N_O = 2$ for time derivative, $N_O = 2, 4, 6..$ and more for space derivatives.

The following computation should be carried out to propagate the field value to a new $(k+1)$th time layer:

$$F|^{k+1} = 2F|^k - F|^{k-1} + c^2 \Delta t^2 \left(\left. \frac{\partial^2 F}{\partial x^2} \right|^k + \left. \frac{\partial^2 F}{\partial y^2} \right|^k + \left. \frac{\partial^2 F}{\partial z^2} \right|^k \right), \tag{3}$$

assuming that all values on the right hand side are known. In the implementation it implies that the field data for two sequential time layers should be stored. Thus

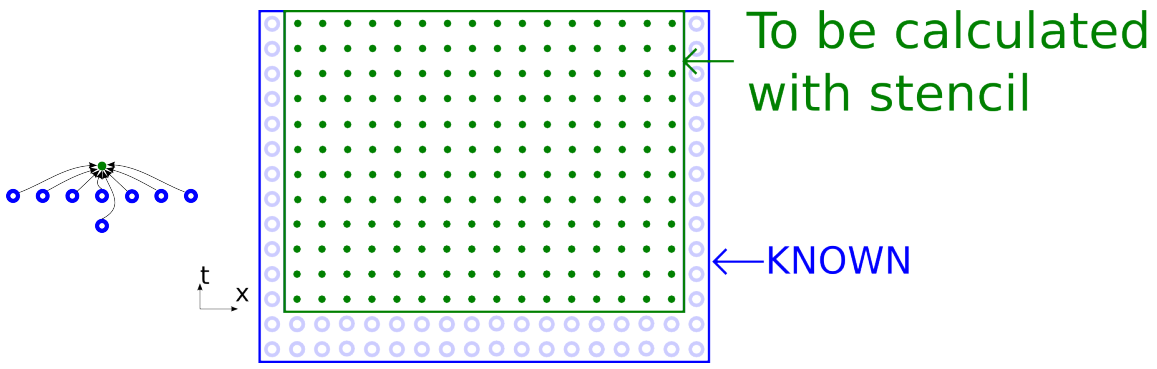

Figure 3: Scheme stencil for 6th order of approximation ($d = 1$) (left). Dependency graph of the problem (arrows are omitted) (right)

it is common to use two program arrays F and G: one for the even time steps, another for the odd ones. To compute, it is necessary for $2 + 3N_O$ values to be loaded from memory, 1 to be saved to memory, and if values like $c^2\Delta t^2/\Delta x^2$ are defined as constant, the amount of FMA (fused multiply-add) operations in the computation is at least $1 + 3N_O$.

By applying the stencil for each point in $(d + 1)$-dimensional mesh (d coordinate axes and and one time iteration axis) we get an entity that we will call a "dependency graph" in this paper. First two layers along time axis is an initial condition, the last layer is desired result of the modeling problem. Between layers the directed edges show data dependencies of the calculations, and calculations are represented by vertices (the top point of the stencil that corresponds to $F|^{k+1}$). First two layers in time and boundary points represent initialization of a value instead of calculation with a stencil. All stencil computations in the graph should be carried out in some particular order.

It is important to trace throughout the study the inherent property of the physical statement of the problem, namely, finite propagation velocity. According to special relativity, there exists a light cone in 4D spacetime, which illustrate the causalities of events (see fig. 4). For a given observer spot, all events that affect it are contained in a past light cone (cone of dependence), and all events that may be affected by the observer are contained in the future light cone (cone of influence). In terms of acoustic wave equation, the slope of the cone is given by the sound speed c.

Numerical approximation of wave equation by explicit scheme with local stencil retains the property, but the cone is transformed according to the stencil shape, and its spread widens. We shall refer to the resulting shapes as "dependency conoid" and "influence conoid" accordingly. The shape of the conoid base for the chosen stencil is a line segment in 1D, rhombus in 2D, octahedron in 3D. We will call this shape "diamond" because of its similarity to the gemstone in

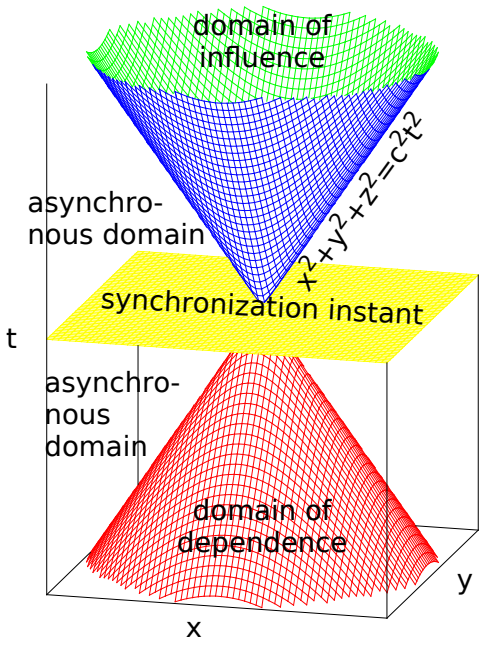

Figure 4: Dependence and influence cones in Minkowsky space

2D and 3D case. The distance between the opposing vertices of d-dimensional orthoplex in the cone base increases by N_O cells along each axis with each time step away from the synchronization instant.

4 Algorithm as a rule of subdividing a dependency graph

In the current approach algorithm is defined as a traversal rule of a dependency graph.

Let us see how an algorithm may be represented by a shape in $(d+1)$-space where the dependency graph is given. If some shape covers some number of graph vertices, it corresponds to an algorithm (or some procedure or function in the implementation) that consists of processing all the calculations in these vertices. This shape may be subdivided into similar shapes, each of which contain smaller number of vertices, in such way, that the data dependencies across each subdivision border are exclusively unilateral. The direction of data dependencies between shapes show the order of evaluation of the shapes. If there is a dependency between two shapes, they must be processed in sequence, if not, they may be processed asynchronously.

After subdivision all the resulting shapes also correspond to some algorithm if given a subdivision rule. By recursively applying this method the smallest shapes contain only one vertex, and correspond to a function performing the calculation. This is a basic idea of LRnLA decomposition.

Let us give an example. The most prevalent way is to process the calculation one time iteration layer after another. The illustration (see fig. 5) by graph subdivision shapes is as follows: a $(d+1)$-dimensional box, which encompasses whole graph of the problem, is subdivided into d-dimensional rectangles, containing all calculations on a certain graph layer. The order of computation in each such layer may be arbitrary: either a loop over all calculations, subdivision into domains for parallel processing, or processing the cells by parallel threads.

Layer-by-layer stepwise calculation is indeed used in almost all physics simulation software, and very few authors have ventured outside the comfort zone. The most notable unfavorable consequences are that during processing each time layer whole data array should be loaded from memory and stored into it, and the parallel processors have to be synchronized. There exist many other dependency graph traversal rules, which require much less synchronization steps and much less memory transfer. More operations may be carried out on the same data. One example is in fig. 5, which arises from tracing dependency/influence conoids.

We shall show now how the optimal algorithm is constructed for a given problem (wave equation with cross-shaped stencil) and for a given implementation environment (GPGPU with CUDA). The illustration is given for two-dimensional problem with $d = 2$ in x—y axes. The dependency graph is plotted in 3D x—y—t space. If we treat each vertex as processing not of one, but of Nz elements, then this illustration is also applicable for 3D simulation problems. Such DiamondTile decomposition is assumed for all further discussion.

- The most compact shape that encompasses the stencil in space coordinates is a diamond. The 2D computational domain is subdivided into diamond shaped tiles. For $N_O = 2$, each tile contains two vertices.
- One diamond tile is chosen on the initial layer. Its influence conoid is plotted. After Nt layers we choose another tile, which lies near the edge of the influence conoid base, on the far side in the positive direction of x-axis. Its dependence conoid is plotted.
- On the intersection of conoids we find a prism (fig. 6).

This prism is a base decomposition shape for DiamondTorre algorithms. Since it is built as an intersection of conoids, the subdivision retains correct data dependencies, and since the shape is a prism, all space may be tiled by this shape (boundary algorithms are the only special case).

Each prism has dependency interconnections only with the prisms, the bases of which are directly adjacent to the base of this prism (see fig. 7). That is, the bases have common edges. The calculations inside the prism depend on calculation result of the two right prisms, and influence the calculations inside the two prisms to the left. The crucial feature is that the calculations inside

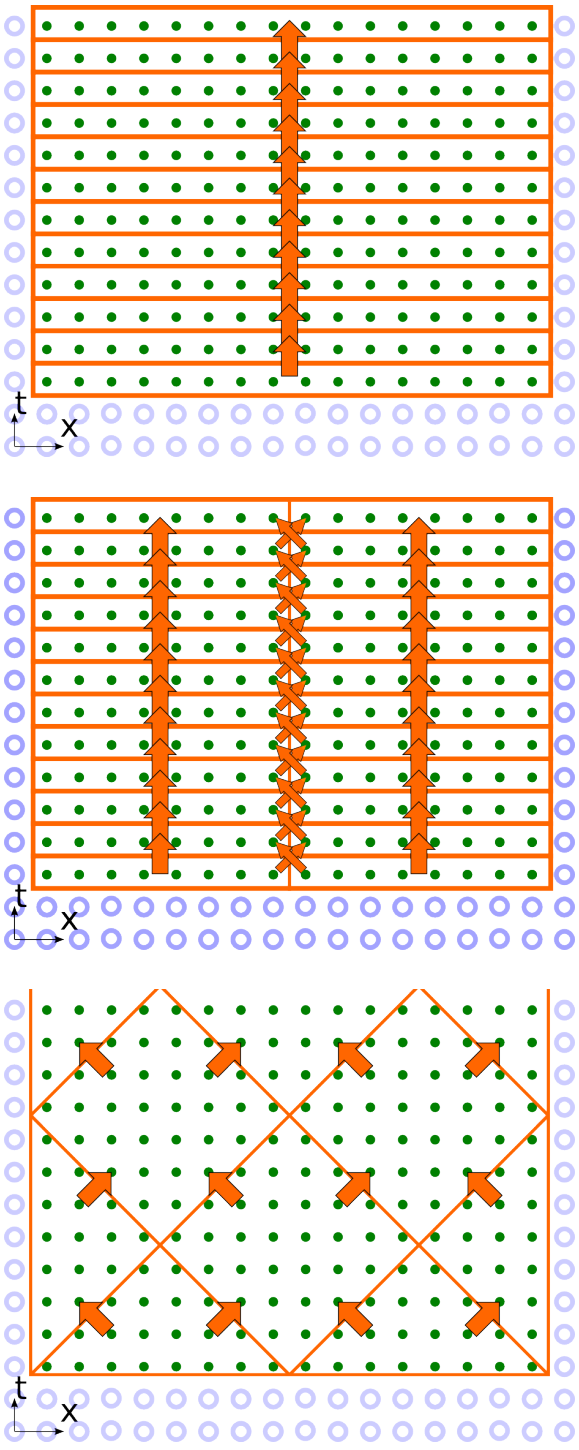

Figure 5: Algorithm as a rule of subdividing a dependency graph: stepwise (top), domain decomposition (center), LRnLA example(bottom). Arrows show data dependencies between subdivision shapes

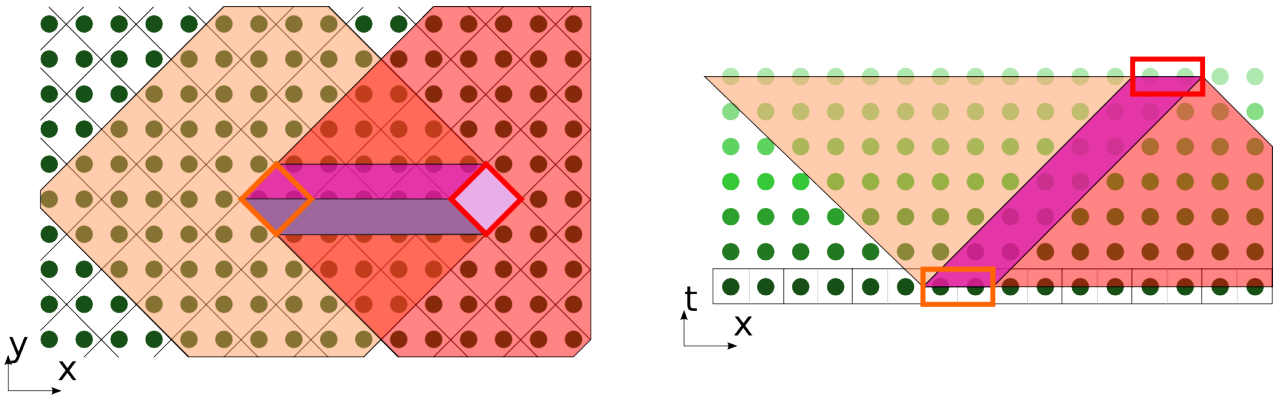

Figure 6: DiamondTorre algorithm construction as an intersection of influence cone (orange) and dependence cone (red) of two tiles. If a point falls onto a shape border, we consider that it is inside the shape if it is on the bottom face; and that it is outside the shape if it falls on the top face

the prisms with common y coordinate, even those the bases of which touch each other, are absolutely independent.

A certain freedom remains with the calibration of prism parameters: height and base size. The distance from the stencil center to its furthest point is defined as stencil half size and equals $ShS \equiv N_O/2$. By default, the base of the prism contains $2 \cdot ShS^2$ vertices. It may be increased in DTS (diamond tile size) along each axis, then the base would contain $2 \cdot ShS^2 DTS^2$ vertices. The height of the prism equals Nt. Nt should be divisible by 2 and DTS.

5 Benefits of LRnLA approach

The goal of the introduction of these algorithms is the solution of the issues which were presented in the introduction, namely, to reduce requirements for memory bandwidth and to increase asynchrony.

To quantitatively compare different algorithms of wave equation modeling in terms of memory bandwidth requirements, we introduce measures of locality and asynchrony of an algorithm. Locality parameter is defined as a ratio of number of dependency graph vertices inside the algorithm shape to the number of graph edges that cross the shape's boundaries. Asynchrony parameter is equal to the number vertices that may be processed concurrently.

The higher are the parameters of locality and asynchrony, the higher performance can be reached for memory-bound problems. While the locality parameter is generally not as high as it could be, the asynchrony parameter is often redundantly large. It is imperative not to increase the asynchrony parameter, but to correctly distribute the concurrent computation on different levels of parallelism.

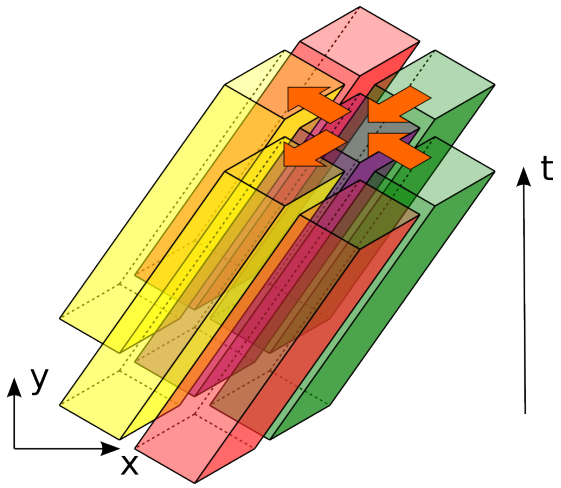

Figure 7: Data dependencies are directed from green prisms into purple one; from purple prism to yellow prisms. Red and purple prisms calculations are completely independent from each other and may be processed asynchronously

Locality parameter has the similar meaning as "operational intensity" measure introduced in the roofline model [18] but differs by a certain factor. The factor is defined from the scheme stencil and is equal to the number of operations per one cell per one time step divided by data size.

Let us calculate the locality parameters for the algorithms, that are introduced above as a subdivision of the dependency graph. For one dependency graph vertex the locality parameter is equal to $1/(3 + 3N_O)$. This subdivision may be illustrated as enclosing each vertex in a box. We will call such algorithm "naive", since it corresponds to direct application of the scheme (3) without account for caching ability specifics on contemporary computers with hierarchical memory subsystem organization.

A row of cell calculations along one axis is asynchronous on one time step. Fine-grain parallelism can be utilized by vectorizing the loop of Nz elements along one (z) axis. The locality parameter increases to $1/(3 + 2N_O)$, the asynchrony parameter is Nz.

More generally, the locality parameter may be increased in two ways. The first method is to use the spatial locality of data of a stepwise algorithm (fig. 5). The quantity of data transfers may be reduced by taking into account the overlapping the scheme stencils. If a scheme stencil stretches in k layers in time ($k = 3$ for the chosen scheme above), it is necessary to load data from $k - 1$ layers, and to save data of 1 time layer. The locality parameter is equal to $1/k$, and this value corresponds to a maximal one for all stepwise algorithms. It is reached only if the algorithm shape covers all vertices of one time layer. In practice this algorithm is impossible since there is not enough space on the upper layers of memory subsystem hierarchy (which means register memory for GPGPU) to

allocate data of all the cells in simulation domain.

Taking the limited size of the upper layer of memory hierarchy into account, we choose tiling algorithm with a diamond shape tile as an optimal. It correspond to DiamondTorre algorithm with $Nt = 1$. The locality parameter in this case is equal to $1/(3+2/DTS+1/DTS^2)$, and it differs from the optimal one for $(1/k = 1/3)$ by a factor of two or less. The asynchrony parameter reaches $Nz \cdot 2DTS^2ShS^2$ since all vertices in a horizontal DiamondTorre slice are asynchronous.

The further increase of locality may be reached through temporal locality, namely, by repeated update of the same cells the data of which is already loaded from memory. DiamondTorre algorithm contains $2ShS^2DTS^2$ calculations on the each of $Nt = 2DTS \cdot n$ (n is some integer number) and requires $2ShS^2(DTS+1)^2 - 2ShS^2DTS^2$ loads and $2ShS^2DTS^2 - 2ShS^2(DTS-1)^2$ saves on each layer, as well as $4ShS^2DTS^2$ initial loads and $4ShS^2(DTS-1)^2$ final saves. The locality parameter for one DiamondTorre amounts to

$$DTS \cdot n/(4n + 2 - 2/DTS + 1/DTS^2), \tag{4}$$

and approaches $DTS/4$ with large Nt.

At this step the transition from fine-grained to coarse-grained parallelism takes place. For a row of asynchronous DiamondTorre (with common y coordinate) asynchrony parameter is increased by a factor of $Ny/(2DTS \cdot ShS)$, which is the amount of DiamondTorres in a row. The locality parameter increases to

$$DTS \cdot n/(n(4 - 1/DTS) + 2 - 2/DTS + 1/DTS^2), \tag{5}$$

and approaches $DTS/(4 - 1/DTS)$ with large Nt.

If the asynchrony of DiamondTorres with different x (and t) positions is involved, the coarser parallel granularity may be utilized. Asynchrony parameter would increase in about $Nx/(2Nt \cdot ShS)$ times.

The roofline model may be plotted with the localization parameter as its horizontal axis. In fig. 10 rooflines for the two target GPUs are plotted in red and green lines. With its use the maximum possible performance for a given algorithm is found as a ceiling point for its localization parameter (black arrows).

6 CUDA implementation

The computing progresses in subsequent stages. A stage consists of processing a row of DiamondTorre algorithm shapes along y axis (fig. 8,9). They may be processed asynchronously, since there are no dependencies between them on any time layers. They are processed by CUDA blocks. Each element of 3D dependency graph that was subdivided into prisms corresponds to processing of Nz elements for 3D problems. Therefore in each DiamondTorre Nz CUDA threads process

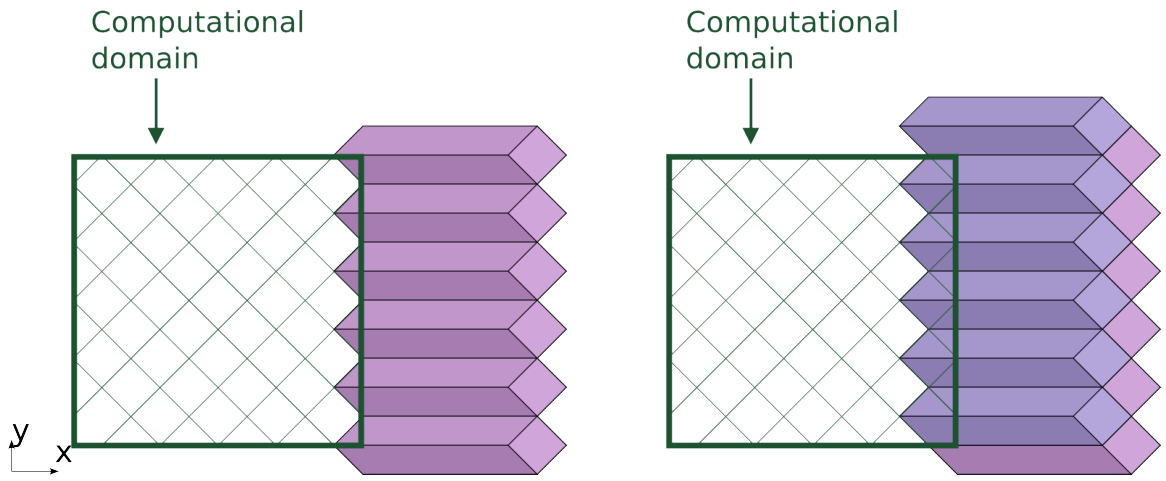

Figure 8: DiamondTorre algorithm implementation with CUDA. First stage (left), second stage (right)

cells along z axis. The DiamondTorre function contains a loop over Nt time layers. Each loop iteration processes cells that fall into the DiamondTile.

It should be noted that as asynchronous CUDA blocks process cells in DiamondTorre's in a row along y axis, the data dependencies are correct even without synchronization between blocks after each time iteration step. The only necessary synchronization is after whole stage is processed. But since there is no conoid decomposition along z axis, CUDA threads within a block should be synchronized. This is important to calculate finite approximations of $\frac{\partial^2 F}{\partial z^2}$ derivative. When CUDA thread processes a cell value, it stores it in shared memory. After synchronization occurs the values are used to compute the finite sum of cell values along z axis. This way it is assured that in one finite sum all values correspond to the same time instant.

Next stage processes a row of DiamondTorre's that is shifted by $ShS \cdot DTS$ in negative x direction, and by same amount in positive y direction. The row is processed like the previous one, and the next one is shifted again in x and y so that by alternating these stages all computation domain is covered.

The first of these rows start near the right boundary of the domain (fig. 8). The upper part of the prisms fall outside the computation domain. These correspond to the boundary functions, in which the loop over the tiles has fewer iterations, and last iterations apply boundary conditions. After boundary prisms are processed (fig. 9), we arrive at the situation when in some cells of the computation domain the acoustic field have the values of the Nt-th iteration step; in some cells the field has its initial values; and other cells have values on time step from 0 to Nt. After all stages are processed, all field values reach Nt-th time step.

All calculations at each time are conducted in a region near the slope on the

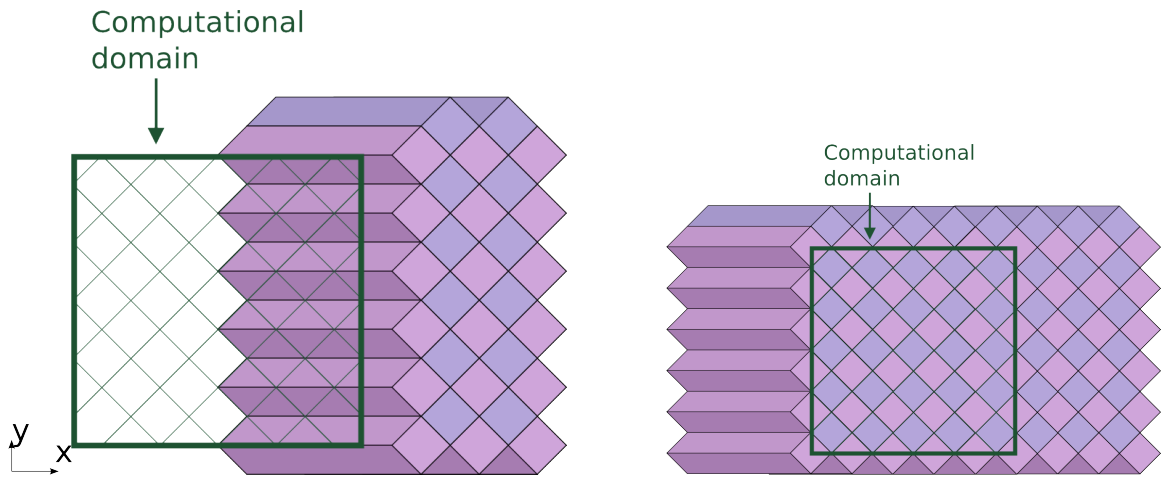

Figure 9: After boundary prisms are processed, in some cells of the computation domain the field have the values of the Nt-th iteration step; in some cells the field has its initial values; and other cells have values on time step from 0 to Nt (left). After all stages are processed, all field values reach Nt-th time step(right)

current DiamondTorre row. This property can be used to implement a so-called "calculation window". Only the data that is covered at a certain stage has to be stored in the device memory, the rest remains in the host memory. This way even big data problems can be processed by one GPGPU device. If the calculation time of one stage equals the time needed to store the processed data and load the data to be processed on the next stage, then the computation reaches maximum efficiency.

To enable multi-GPGPU computation the calculation on each stage may be distributed by subdividing the DiamondTorre row in y axis into several parts, equal to GPGPU number.

7 Results

The algorithm is implemented and tested for various values of Nt, DTS, NO.

In fig. 10 the achieved results for second order of approximation are plotted under the roofline. The lowest point corresponds to FDTD3d result from built-in CUDA examples. It should be noted that the comparison is not exactly fair, since in FDTD3d the scheme is of first order in time, and uses a stencil with one point less ($k = 2$). Other points are from the computation results with DiamondTorre algorithm with increasing DTS parameter, $Nt = 100$.

In fig. 11 the calculation rate is plotted versus parallel levels, measured in warps. It is measured as strong scaling, for a calculation mesh of about $\sim 5 \cdot 10^9$ cells. From 1/32 to 1 on horizontal axis the number of used GPGPU threads rises from 1 to 32. The increase in calculation rate is satisfactorily linear. After 1 the

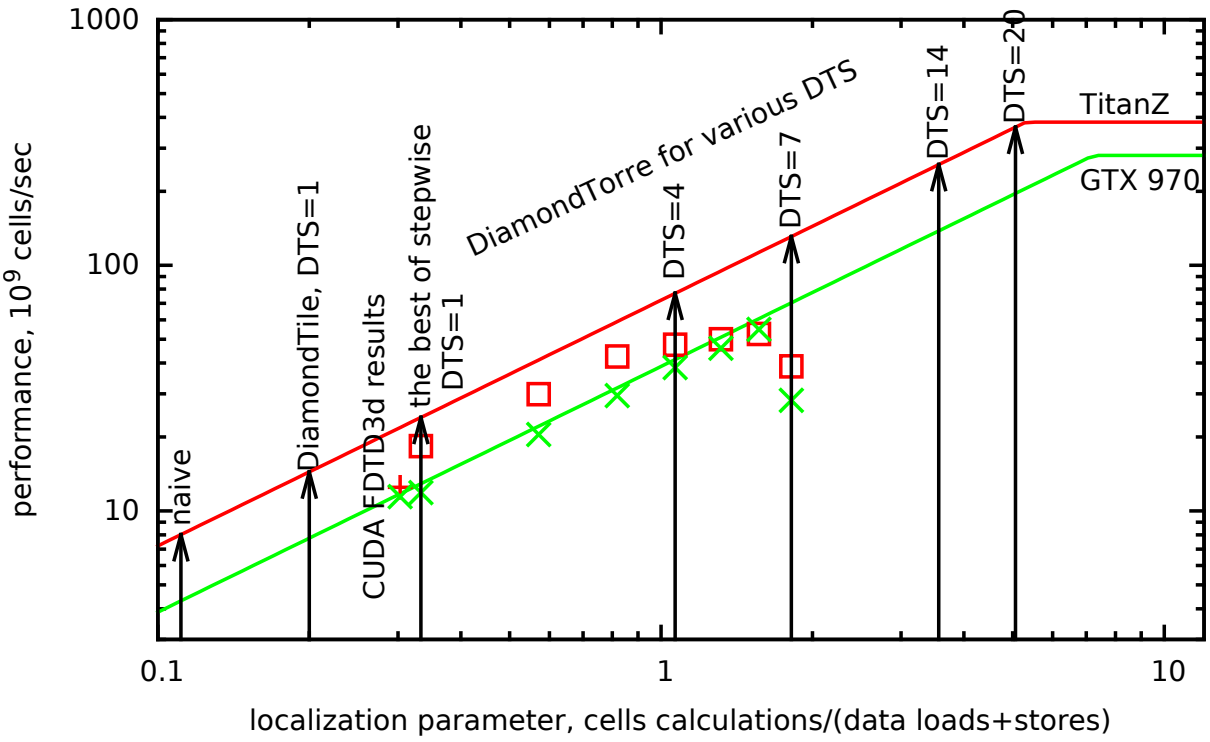

Figure 10: RoofLine Model for Wave Equation on GPGPU

parallel levels increase by adding whole warps up to the maximum number of warps in block (8), with the amount of enabled registers per thread equal to 256. After this the amount of blocks is being increased. The increase of calculation rate remains linear until the number of blocks becomes equal to the number of available multiprocessors. The maximum achieved value is over 50 billions cells per second.

In fig. 12 the achieved calculation rate is plotted with different parameters. The labels on horizontal axis are in the form N_O/DTS. Overall, the results correspond to the analytical predictions (5). With fixed $DTS = 1$ and $N_O = 2, 4, 6, 8$ (first 4 points) the calculation rate is constant (for Maxwell architectures), although the amount of calculation per cell increases. It is explained by the fact that the problem is memory bound. Computation rate increases with DTS for constant N_O since the locality parameter increases. For the rightmost points of the graph, the deviation from the analytical estimate for Kepler architecture is explained by insufficient parallel occupancy.

8 Generalization

The area of DiamondTorre application is not limited to acoustic wave equation. It has also been successfully implemented for finite difference time domain

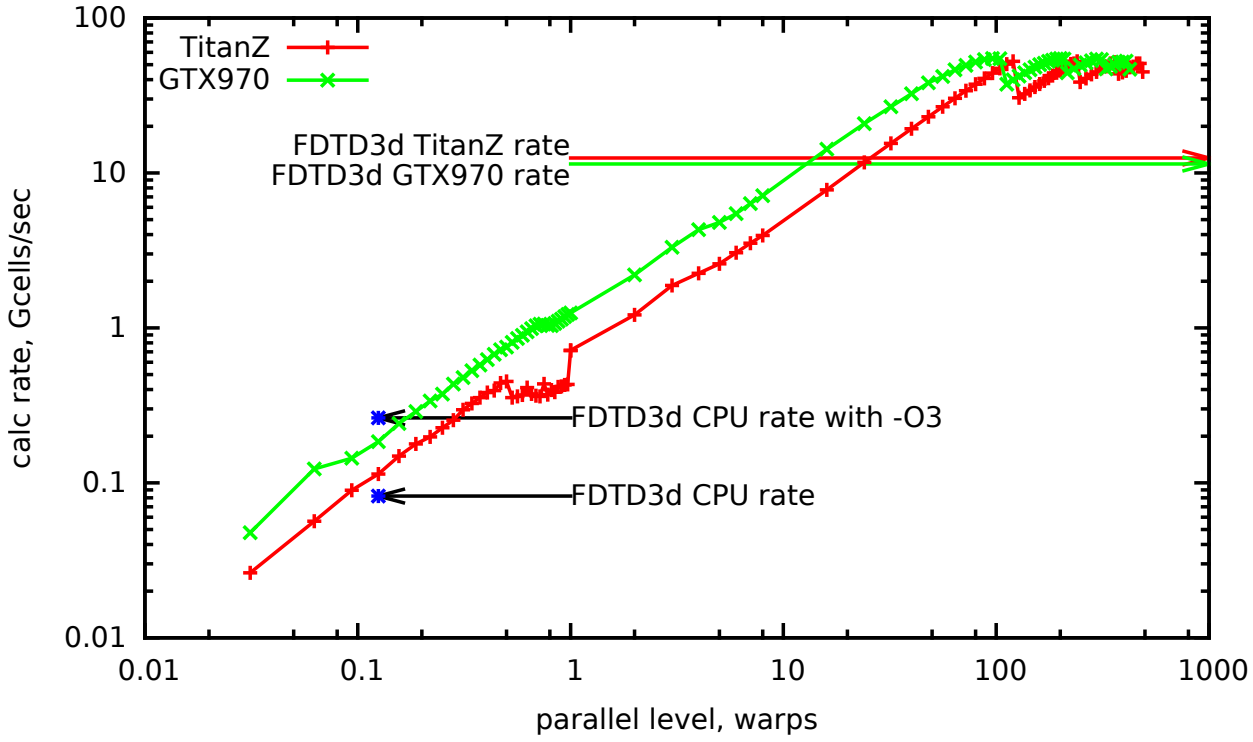

Figure 11: Strong scaling for wave equation. $DTS = 6$, $Nt = 96$, $N_O = 2$

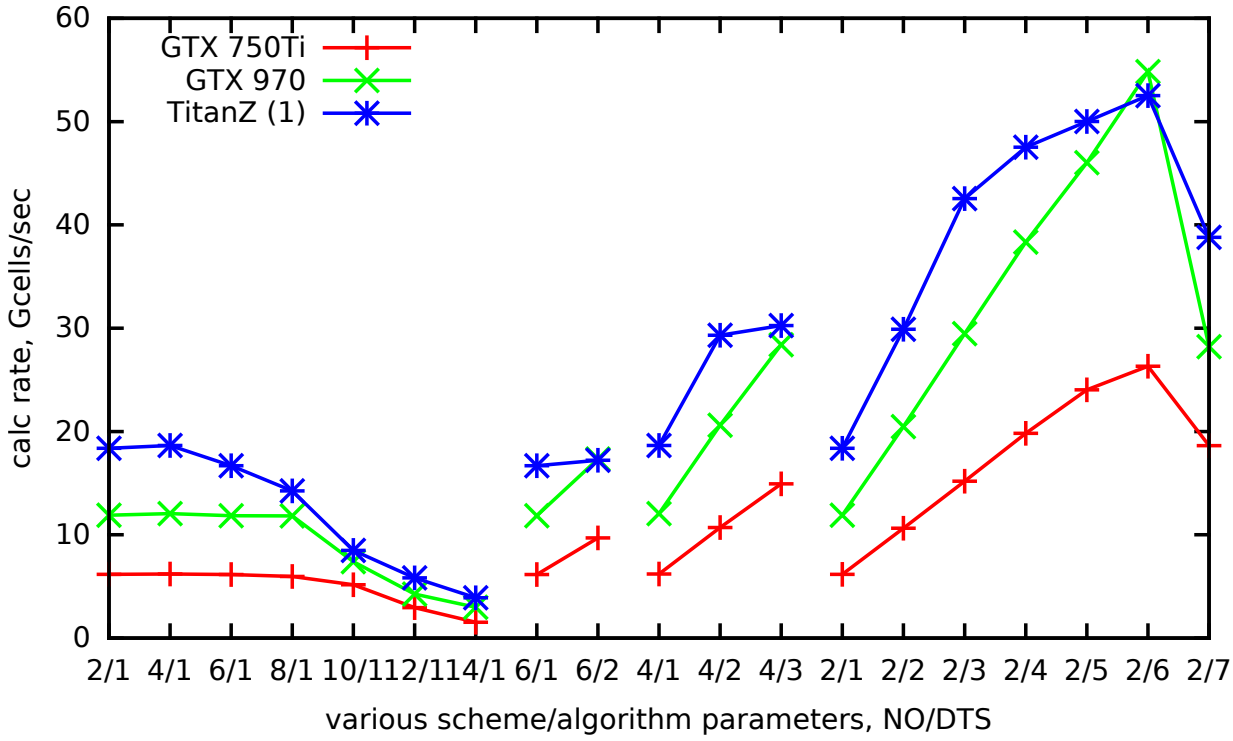

Figure 12: Performance results for different parameters. Horizontal axis labels are in the form N_O/DTS.

methods (FDTD) [19], Runge-Kutta discrete Galerkin method [20], Particle-in-Cell plasma simulation [21]. The LRnLA method of algorithm construction may also be applied for any other numerical methods with local dependencies, and other computer systems and methods of parallelism.

What remains to be discovered is the concept, benefits and area of use of other possible algorithms based on the discussed concept. For example, a fully 3D DiamondTorre dependency graph decomposition.

9 Aknowledgements

The work is supported by RFBR grant 14-01-31483.

References

[1] "LAPACK," retrieved on 2015-03-05. [Online]. Available: http://www.netlib.org/lapack/

[2] "TOP500 list," retrieved on 2015-03-05. [Online]. Available: http://www.top500.org/lists/2014/11/

[3] W. D. Gropp and D. E. Keyes, "Domain decomposition on parallel computers," *IMPACT of Computing in Science and Engineering*, vol. 1, no. 4, pp. 421 – 439, 1989. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0899824889900037

[4] "MPI: A message-passing interface standard version 3.0, message passing interface forum, september 21, 2012," retrieved on 2015-03-05. [Online]. Available: http://www.mpi-forum.org

[5] M. Wolfe, "More iteration space tiling," in *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '89. New York, NY, USA: ACM, 1989, pp. 655–664. [Online]. Available: http://doi.acm.org/10.1145/76263.76337

[6] "OpenCL," retrieved on 2015-03-05. [Online]. Available: https://www.khronos.org/opencl/

[7] "CUDA toolkit 6.5," retrieved on 2015-03-05. [Online]. Available: https://developer.nvidia.com/cuda-downloads

[8] L. Lamport, "The parallel execution of DO loops," *Commun. ACM*, vol. 17, no. 2, pp. 83–93, Feb. 1974. [Online]. Available: http://doi.acm.org/10.1145/360827.360844

[9] D. Bertsekas and J. Tsitsiklis, *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall, Inc., 1989.

[10] "OpenMP application program interface version 4.0 - july 2013," retrieved on 2015-03-05. [Online]. Available: http://openmp.org/wp/openmp-specifications/

[11] "Intel® 64 and IA-32 architectures optimization reference manual," retrieved on 2015-03-05. [Online]. Available: http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html

[12] C. Lameter, "NUMA (non-uniform memory access): An overview," *ACM Queue*, 2013.

[13] H. Prokop, "Cache-oblivious algorithms," Master's thesis, MIT, 1999.

[14] P. Micikevicius, "3D finite difference computation on GPUs using CUDA," in *Proceedings of 2Nd Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-2. New York, NY, USA: ACM, 2009, pp. 79–84. [Online]. Available: http://doi.acm.org/10.1145/1513895.1513905

[15] V. Volkov and J. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, Nov 2008, pp. 1–11.

[16] 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.

[17] A. Perepelkina, I. Goryachev, and V. 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. [Online]. Available: http://iopscience.iop.org/1742-6596/510/1/012042

[18] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009. [Online]. Available: http://doi.acm.org/10.1145/1498765.1498785

[19] V. Levchenko, I. Goryachev, and A. Perepelkina, "Interactive FDTD simulation using LRnLA algorithms," in *Progress In Electromagnetics Research Symposium Abstracts*, Stockholm, Sweden, Aug. 12–15 2013, p. 1002.

[20] L. V. Korneev B.A., "Effective numerical simulation of the gas bubble-shock interaction problem using the RKDG numerical method and the DiamondTorre algorithm of the implementation," *Keldysh Institute Preprints*, no. 97, 2014. [Online]. Available: http://library.keldysh.ru//preprint.asp?lg=e&id=2014-97

[21] A. Y. Perepelkina, V. D. Levchenko, and I. A. Goryachev, "3D3V plasma kinetics code DiamondPIC for modeling of substantially multiscale processes on heterogenous computers," in *41st EPS Conference on Plasma Physics*, ser. Europhysics Conference Abstracts, P. O. Scholten, Ed., no. 38F, EPS. Berlin: European Physical Society, June 2014, p. O2.304.