Keldysh Institute • Publication search

Keldysh Institute preprints • Preprint No. 225, 2018

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

Perepelkina A.Y., Levchenko V.D.

The DiamondCandy Algorithm for Maximum Performance Vectorized Cross-Stencil Computation

Recommended form of bibliographic references: Perepelkina A.Y., Levchenko V.D. The DiamondCandy Algorithm for Maximum Performance Vectorized Cross-Stencil Computation // Keldysh Institute Preprints. 2018. No. 225. 23 p. doi:10.20948/prepr-2018-225-e
URL: http://library.keldysh.ru/preprint.asp?id=2018-225&lg=e

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

Anastasia Perepelkina, **Vadim Levchenko**

The DiamondCandy Algorithm for Maximum Performance Vectorized Cross-Stencil Computation

Москва
2018

УДК 519.688

Перепёлкина А.Ю., Левченко В.Д.

Алгоритм DiamondCandyV для повышения производительности конечноразностных вычислений с применением векторизации

На основе поиска оптимальной пространственно-временной декомпозиции 3D1T пространства операций построен новый алгоритм DiamondCandyV для эффективной реализации конечно-разностных вычислений с использованием аппаратной векторизации. Базовый элемент разбиения пространства операций DiamondCandy получен пересечением областей зависимостей и влияния (конусоидов) для схем с шаблоном типа крест. Благодаря этому алгоритм характеризуется высокой вычислительной интенсивностью и локализует обрабатываемые данные на верхних уровнях иерархии памяти современных компьютеров. Ключевой особенностью алгоритма является легко реализуемая в программном коде поддержка двух основных средств повышения производительности современных процессоров, а именно аппаратной векторизации (с использованием SIMD расширения AVX) и вычислительных потоков с общей памятью (many-core CPU). Обсуждаются детали программной реализации поддержки параллельности различных уровней на примере численного решения волнового уравнения. Результаты тестирования реализации алгоритма показывают повышение производительности на порядок по сравнению с традиционными алгоритмами с пошаговой синхронизацией. Также, в отличие от традиционного подхода, с увеличением размера обрабатываемых данных производительность не деградирует.

Keywords: конечно-разностные вычисления, LRnLA, волновое уравнение, time skewing, многоядерные процессоры, векторизация

Anastasia Perepelkina, Vadim Levchenko

The DiamondCandy Algorithm for Maximum Performance Vectorized Cross-Stencil Computation

An advance in the search for the 4D time-space decomposition that leads to an efficient vectorized cross-stencil implementation is presented here. The new algorithm is called DiamondCandy. It is built from the dependency and influence conoids of the scheme stencil. It has high locality in terms of the operational intensity, SIMD parallelism support, and is easy to implement. The implementation details are shown to illustrate how both instruction and data levels of parallelism are used for many-core CPU. The test run results show that it performs an order of magnitude better than the traditional approach, and that the performance does not decline with the increase of the data size.

Keywords: Stencil, LRnLA, Wave Equation, time skewing, many-core

1 Introduction

We focus on the finite difference numerical solution of the Cauchy problem for the 3D wave equation:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \triangle u. \tag{1}$$

The goal is to find the most efficient implementation of the numerical scheme on a many-core processor with vector processing units. For example, on a Xeon Phi KNL. The results may be naturally generalized on all similar explicit stencil based numerical schemes.

The most intuitive way to implement the scheme in code is to write 4 nested loops:

```
for it in 0..Nt
  for ix in 0..Nx
    for iy in 0..Ny
      for iz in 0..Nz
        scheme update...
      end
    end
  end
end
```

This way, firstly, all field data is updated from the initial values to the values on time step `it=1`. Then all data is updated from `it=1` to `it=2`. Actually, the majority of codes are written like this. The algorithms of this kind are called *"stepwise"* in this paper since all field values are updated in time step by step.

The iteration on each time step is over a rectangular region. It is the easiest way to write programs. "For" loops are a natural instrument to iterate over the rectangular domains that is present in any relevant programming language. Moreover, human brain intuitively perceives the spacetime in Euclidean metrics, not as a Minkowski space.

The stepwise order of calculation is not the only correct one. The computation may be executed in any order which does not conflict with the data dependencies. And this order is one of the less efficient in terms of taking advantage of the acceleration techniques that are present in the modern computer hardware.

With the stepwise approach, field values all over the domain need to be loaded, updated and stored to progress each one time step. The data is large for most relevant applications. Thus, the problem with the stepwise algorithm that solves has a low arithmetic intensity and it is in the memory-bound category [1].

In terms of parallelism, the domain decomposition method is the most usual approach here. With the stepwise approach, the parallel processors need to be synchronized by exchanging data to continue to each next time step. This becomes the bottleneck of the parallel efficiency.

2 Time-space decomposition

The idea to neglect the straightforward loops and, instead, to find optimal schedules from the analysis of data dependencies has appeared long ago [2]. Now, when the high-performance systems are essentially hybrid with many levels of parallelism and many levels of cache, it receives even more attention.

Locally-Recursive non-Locally Asynchronous (LRnLA) algorithms [3] present one of the methods to optimize the performance with a time-space decomposition of computations.

Both the real world physics and the explicit numerical schemes have a finite speed of information propagation. Thus, with respect to any point, the time-space domain can be split into the cone of dependence, the cone of influence, and the absolutely asynchronous domain. In case of a numerical scheme, the cone shape is defined by the scheme stencil, so the more general term 'conoid' is used. For the cross stencil of finite difference schemes, the conoid base is a diamond.

The general way to construct an LRnLA algorithm is to take two time instants and a base shape on both of them. Then, plot the area of influence of the bottom one and the area of dependence of the top one, and find their intersection. The shape is called the conoid of dependence-influence. The time-space simulation domain may be subdivided into conoids. The fact that each conoid can be subdivided into similar conoids to optimize cache use is the "locally recursive" property of the algorithms. The fact that distant conoids may be computed without synchronization and data exchanges is the "non-locally asynchronous" property of the algorithms.

The shape in the time-space can be thought of as a code block for computation of the points inside it. Then an LRnLA algorithm is defined as a shape with its subdivision rule [4].

To be specific about time-space description we use nD1T (e.g., 2D1T, 3D1T) for the field of n spatial dimensions and time, and nD (2D, 3D) for the n-dimensional slice of it at one time instant. The simulation is nD if it solves partial differential equations in n coordinate axes and time.

Let us take a cubic region in 3D and a point in its center as the conoid bases. The resulting conoids are pyramids in 3D1T, which correspond to the Minkowski cone in the real world. The first LRnLA algorithms [5, 3] were

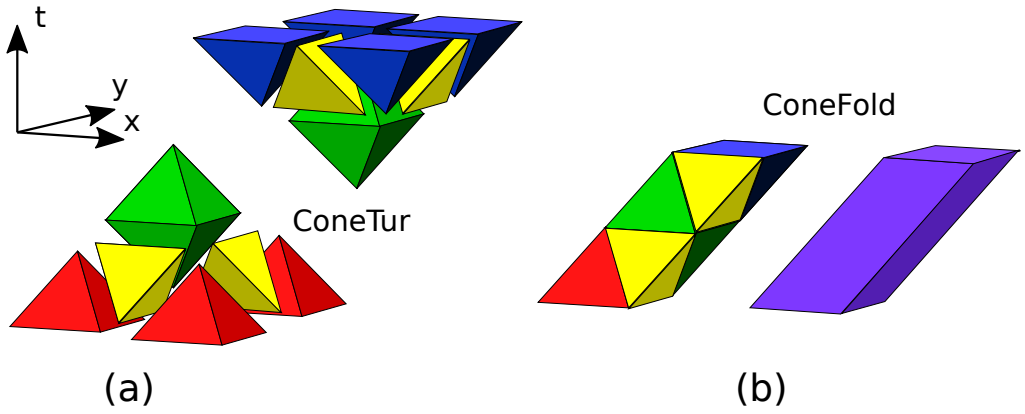

Figure 1: Some LRnLA algorithm shapes projected to the 2D1T time-space

built like that. The name of the first algorithm was ConeTur (Fig. 1a).

The idea was used in the implementation of the Euler scheme of Vlasov equation [6], finite difference time domain code [7]. Even the more complex schemes, like the particle-in-cell simulation, were found to have finite propagation speed. Indeed, the particle-in-cell scheme includes particles, which move freely inside the cells mesh. However the speed of the particles is naturally limited, so the construction of dependency cones works as well. The 2D and 3D particle-in-cell codes were implemented with LRnLA algorithms and used for computational heavy research problems [8].

Some recent advances in time-space decomposition by other authors present the similar decomposition [9, 10].

The ConeTur algorithm had some key aspects. It is easily generalized to higher dimensions by a superposition of 1D1T shapes. With the recursive subdivision for 1D simulation, the resulting shapes in 1D1T time-space were triangles (upright and upside-down) and diamonds, and there were special cases at the boundaries. In 2D1T the pyramids are tiled along with octahedrons and tetrahedrons, and the number of special shapes for the boundaries increases even more. For 3D1T the programming effort becomes unreasonable, so code generators were used. Even then, the code even for the simple schemes is too complex. The performance efficiency is limited since the large code does not fit the instruction cache. Moreover, the compilation time becomes too large.

If the conoids of different shapes are combined, it is possible to find a uniform tiling of the time-space. The algorithm that is constructed as a combination of ConeTur shapes is called ConeFold [11, 12] (Fig. 1b).

With the use of ConeFold the codes for simulation of optics with the FDTD scheme, seismic waves with the Levander scheme [12], plasma with the particle-in-cell scheme [11] were implemented and used for real physics problems. The

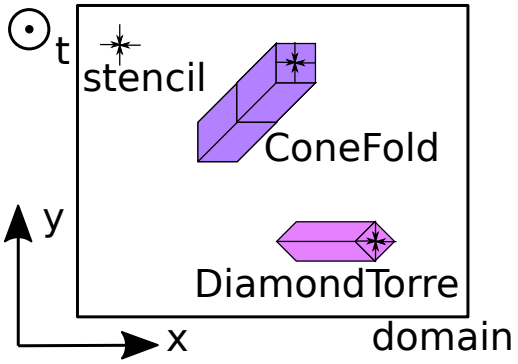

Figure 2: The difference between DiamondTorre and ConeFold conoids

conoids with the shape similar to ConeFold, but different decomposition rules, are used to construct algorithms for various levels of parallelism.

Other ways to construct a uniform time-space tiling are presented by other authors. For example, the wavefront [13] or time-skewing method [14]. Among these, the optimum locality is reached when the time-space decomposition provides blocks, localized in 3D1T [15].

The shapes that arise in this methods present rectangular shapes in each 3D space slice. However, it is not optimal for the computation asynchrony and for the data locality of cross stencil applications.

The need for more asynchrony which arose with the introduction of GPU to scientific computing led to the construction of the DiamondTorre LRnLA algorithm [4].

DiamondTorre is constructed in 2D1T. The conoid bases are 2D diamonds, shifted by several grid spaces from each other. This way, the conoids are prisms in the 2D1T space-time. Indeed, they are similar to the 2D1T projection of ConeFold, but rotated by 45 degrees, so that rectangles become diamonds (Fig. 2). The DiamondTorre prisms in a row along the y axis are asynchronous, *with no dependencies between each other*. Thus, the computation in these may be distributed between streaming multiprocessors.

In addition, it is better localized in memory. The data necessary for the execution of the operations inside one shape may be estimated as its projection to spatial coordinates, 2D in case of DiamondTorre. The projection of DiamondTorre is smaller than the projection of ConeFold-based algorithms (Fig. 2). Thus, the data for the update of the points inside one shape may fit into higher levels of cache, while the same number of operations are carried out.

The algorithm was implemented with the use of GPU. It was used in optics [16], seismic wave propagation [17], plasma physics [18], gas dynamics [19].

The drawback of the DiamondTorre is that it is based on a 2D decomposition

of the spatial domain. While a 2D plane may be subdivided into diamonds, the 3D space subdivision requires an octahedron and two types of tetrahedrons. Thus, the complexity of the algorithm description rises. One previous attempt is reported in [20], however, the overhead for vectorization outweighed the locality benefits.

The current paper is about our recent find in this research. The new algorithm is based on a conoid that is homogeneous, may be recursively subdivided into similar shapes, and is based on a diamond (octahedron) shape that provides high locality for the cross-stencil.

The proposed polyhedral subdivision of the iteration space resembles polyhedral compilation techniques [21, 22] in a way. In these, authors aspire to a more general goal to optimize the loop traversal rule for any computation, by analyzing the code without the knowledge of the numerical schemes in its base. These provide the useful terminology and mathematical tools to describe the complex algorithmic structures, but suffer from the complexity of the optimization problems for real computation applications.

In LRnLA method the polyhedra are devised from the shape of the dependency conoid. This ensures better locality and correct flow of data dependencies. Another distinction of LRnLA algorithms among other time-space approaches is the recursivity of subdivision. Each nD1T shape that appears after the domain subdivision is subdivided again into smaller similar shapes. The decomposition rules vary on some levels of decomposition to adapt to the properties of the cache memory and the method of parallelism.

One more important point to think about is SIMD vectorization. On CPU, the SIMD parallelism may accelerate the computation up to 16 times. The automatic vectorization in modern compilers is optimized for simple loops, but not for the complex time-space decomposition algorithms. Vectorization of localized blocks of data proves to be efficient for stencil computing [23]. In the construction of the algorithm, the attention to the vectorization options remains important.

3 The DiamondCandy algorithm

The construction here is carried out for the specific example of the finite difference scheme for the wave equation with the cross stencil. The time deriva-

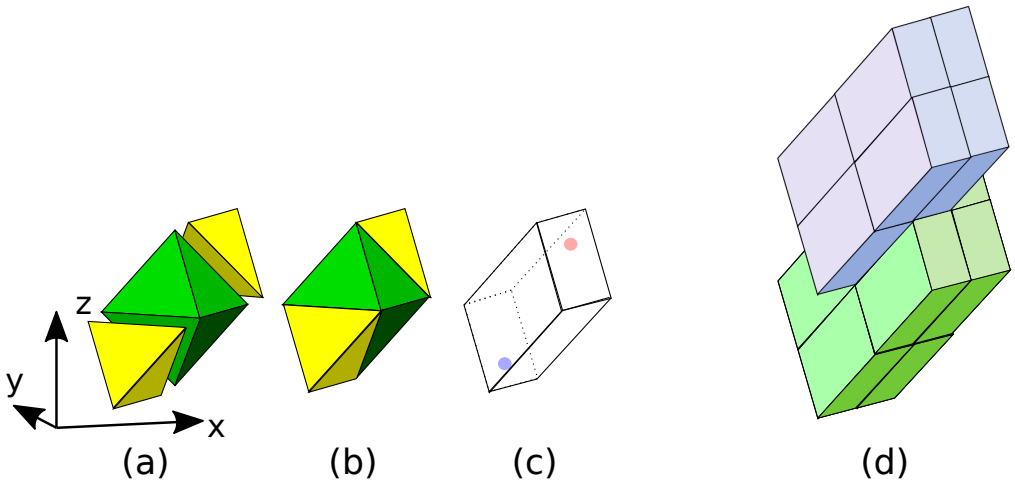

Figure 3: (a),(b): the DiamondCandy base shape (candy) construction. (c): its correspondence with the grid points. (d): f (green) and g (blue) values combined into one structure

tive uses 3 points, space derivatives use 5 points.

$$\begin{aligned}\frac{\partial^2 u}{\partial t^2} &= \frac{u(t+\Delta t) - 2u(t) + u(t-\Delta t)}{\Delta t^2};\\ \frac{\partial^2 u}{\partial x^2} &= \frac{1}{\Delta x^2}\left(-\frac{1}{12}(u(x+2\Delta x) + u(x-2\Delta x)) + \right.\\ &\left. +\frac{4}{3}(u(x+\Delta x) + u(x-\Delta x)) - \frac{5}{2}u(x)\right).\end{aligned} \tag{2}$$

Since 3 points in time are used, two data arrays are necessary. These are called f and g here, so that $f = u(t + k\Delta t)$ for odd k, and $g = u(t + k\Delta t)$ for even k.

The 3D domain may be decomposed into the octahedron and tetrahedron honeycomb. The shape of the conoid base is chosen as a combination of the octahedron and the two adjacent tetrahedrons (Fig. 3a,b). The bases on the two time layers are shifted along the z axis from each other, and the prism is constructed in 3D1T.

The resulting shape is advantageous in three aspects. Firstly, it closely encompasses the octahedron that is the base of the natural conoids for the cross stencil. Thus, it provides high locality and tiles the whole space at the same time. Secondly, it is isomorphic to a cube, so just as a cube may be subdivided into 8 smaller cubes, the chosen shape may be recursively decomposed into 8 smaller similar shapes. Thirdly, it is local with respect to all 3 spatial axes.

This shape is called a trigonal trapezohedron in geometry. In the search for a better word we decided to call the shape 'candy'.

The construction is essentially a generalization of the DiamondTorre algorithm to the 3D1T decomposition. Thus, the reader may refer to [4] for a

better understanding of the illustrations.

When placed on a computational grid, the smallest candy encompasses 2 grid points (Fig. 3c). Their relative positions are $(0,0,0)$ and $(1,1,1)$. This is called 'the base pair' hereafter.

The terminology provided for the DiamondTorre algorithm [4] applies here as well.

The basic element of space-time decomposition is defined by the stencil spatial size. Stencil half size (`ShS`) is the parameter of the algorithm and it defines the candy size. For the chosen scheme it equals 2. Thus, the minimal element of the data structure is $2 \times 2 \times 2 = 8$ base pairs combined into a bigger candy. The candy for the g values is defined shifted up from the candy for f values for `ShS` grid steps in the z direction (Fig. 3d). The example of the code for the structure is

```
struct cell {
  ftype gx1y1z2, gx0y0z2, gx2y1z3, gx1y0z3,
        gx1y2z3, gx0y1z3, gx2y2z4, gx1y1z4,
        gx2y2z3, gx1y1z3, gx3y2z4, gx2y1z4,
        gx2y3z4, gx1y2z4, gx3y3z5, gx2y2z5;
  ftype fx1y1z0, fx0y0z0, fx2y1z1, fx1y0z1,
        fx1y2z1, fx0y1z1, fx2y2z2, fx1y1z2,
        fx2y2z1, fx1y1z1, fx3y2z2, fx2y1z2,
        fx2y3z2, fx1y2z2, fx3y3z3, fx2y2z3;};
```

The values in the variable names correspond to the relative position on the x–y–z grid.

The definition of this element is convenient, since it results in a more homogeneous code. In addition, this data structure ensures local and coalesced access for the DiamondCandy algorithm. The computation of all f values in the candy with this size depends on the data from the g values of the 8 candies, and vice versa.

To use the AVX vectorization the data from 32 points are combined into 4 vectors of 8 elements so that in each pair the points belong to two different vectors.

```
struct cellV { ftypeV g[2], f[2]; };
```

The algorithm building block consists of the computation of one f candy followed by the computation of the g candy ShS mesh steps above it in z direction. In 3D1T, it is a prism shape slanted in z–t direction, just as DiamondTorre is slanted in x–t. The bottom base of the prism is the f candy; the upper base of the prism is the g candy. Several more alternating f and g candy computations may be added to increase the prism height. The number of f

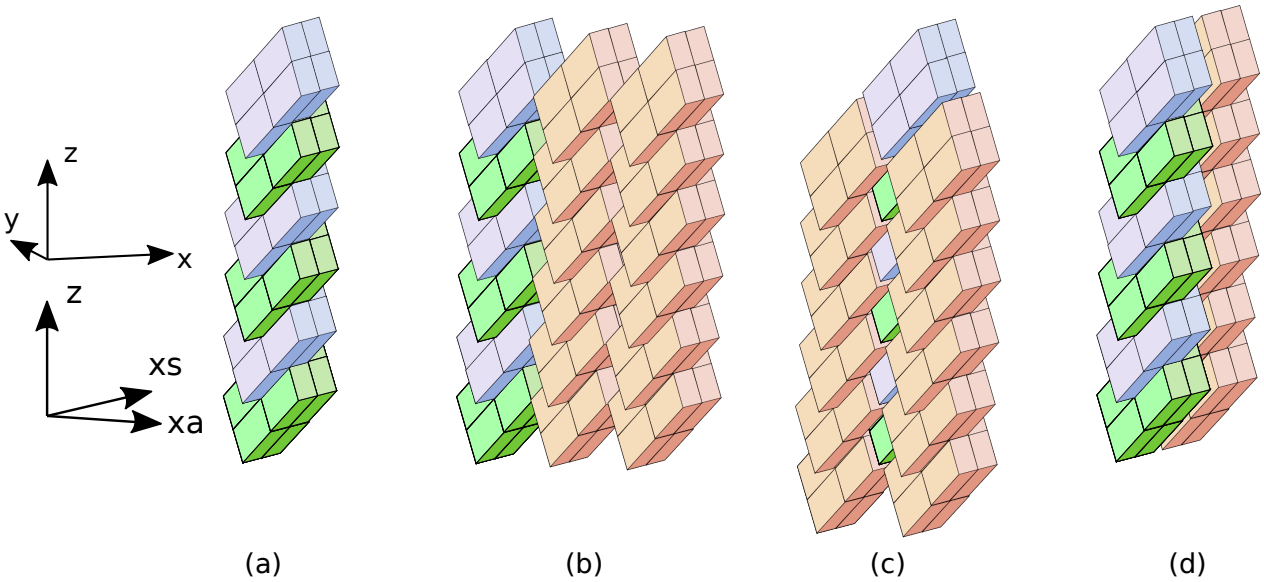

Figure 4: (a): A DiamondCandy projection onto the 3D space. (b): an example of asynchronous DiamondCadies. (c) and (d): the three DiamondCandies that depend on the first one.

and g updates in the prism is the algorithm parameter, NT. So the number of time steps in it is $2NT$. This algorithm, and the 3D1T shape corresponding to it, are called DiamondCandy.

The projections of the DiamondCandy to the 3D space are presented on Fig. 4.

To perform the computation of the whole 3D1T domain, the 3D space is tiled with candies. Then the DiamondCandy algorithm should be started from each of the bases. The appropriate order is defined by the dependencies between the DiamondCandies. By looking at the data dependencies we find asynchronous and dependent DiamondCandies.

One example of an asynchronous set of DiamondCandy algorithms is a row, where their bases are shifted by $ShS \cdot (1, -1, 0)$ (Fig. 4b). For the convenience, the axis x_a (asynchronous axis) is introduced in the direction $(1, -1, 0)$. The axis that is orthogonal to x_a and z is called x_s (synchronous axis). Its direction is $(1, 1, 0)$.

We illustrate the algorithm with the actual code listings here. We used this specific code for the performance runs. The performance results are studied in section 5. The code was verified to produce valid simulation results.

The cell data is organized into a 3D array in x_s, x_a, z axes. In this case, odd and even rows in z-direction need to be distinguished since the access to neighboring candies follows a different pattern. In the x–y point of view, odd an even rows are shifted by (0,ShS,0) with respect to each other.

The smallest element of computation is

```
inline void calc_even(int icxs, int icxa, int icz){
  stencil(get_cellV(icxs+0,icxa+0,icz+0).f,
       get_cellV(icxs+0,icxa+0,icz-2).g,
       get_cellV(icxs-1,icxa+0,icz-2).g,
       get_cellV(icxs+0,icxa-1,icz-1).g,
       get_cellV(icxs-1,icxa-1,icz-1).g,
       get_cellV(icxs+0,icxa+0,icz-1).g,
       get_cellV(icxs-1,icxa+0,icz-1).g,
       get_cellV(icxs+1,icxa+0,icz+0).g,
       get_cellV(icxs+0,icxa+0,icz+0).g
       );
  stencil(get_cellV(icxs+0,icxa+0,icz+0).g,
       get_cellV(icxs+0,icxa+0,icz+0).f,
       get_cellV(icxs-1,icxa+0,icz+0).f,
       get_cellV(icxs+0,icxa-1,icz+1).f,
       get_cellV(icxs-1,icxa-1,icz+1).f,
       get_cellV(icxs+0,icxa+0,icz+1).f,
       get_cellV(icxs-1,icxa+0,icz+1).f,
       get_cellV(icxs+1,icxa+0,icz+2).f,
       get_cellV(icxs+0,icxa+0,icz+2).f
       );}
```

The `stencil(ftypeV* f, ftypeV gB,... )` function contains intrinsic vector operations for the computation of the cross stencil for the vector `f`. A similar function is needed when the computation starts from the bases on the odd rows.

This element is iterated several times for the prism of height NT. The row in the x_a axis is computed asynchronously.

```
#pragma omp parallel for
for (int icxa=0; icxa<Ncxa; icxa++) {
  for (int it=0; it<NT; it++){
    calc_even(icxs,icxa,icz+2*it);
  }
}
```

Actually, more DiamondCandies than just one row of them are asynchronous. The iteration between them would require a bit more complex code, and is not in the scope of the current work.

The data dependencies from one DiamondCandy are directed to the next DiamondCandy in the x_s axis, and to the two DiamondCandies below it (Fig.

4c,d).

Thus, to get the correct simulation results, first of all, the rows of DiamondCandies along the top boundary of the domain are computed. Among them, the row with the smallest x_s is started first. Then the simulation progresses to the lower layer. The next layer is odd in the illustrated case. Omitting the special cases on the boundary, the example code is:

```
for (int icz = Ncz-2; icz>=0; icz-=2){
  for (int icxs=0; icxs<Ncxs; icxs++)
#pragma omp parallel for
    for (int icxa=0; icxa<Ncxa; icxa++) {
      for (int it=0; it<NT; it++){
        calc_even(icxs, icxa, icz+2*it);
      }
    }
  for (int icxs=0; icxs<Ncxs; icxs++)
#pragma omp parallel for
    for (int icxa=0; icxa<Ncxa; icxa++) {
      for (int it=0; it<NT; it++){
        calc_odd(icxs, icxa, icz+2*it);
      }
    }
}
```

The boundary rows that are not shown here contain different calculations since the boundary condition applies. As is, the algorithm forbids periodic boundary condition in z and x_s directions.

For more data access locality, the DiamondCandies may be combined into bigger similar shapes. The parameter of the size of the conoid is called DTS (Diamond Tile Size). By definition, the candies that contain ShS^3 basic pairs, have $DTS = 1$. Bigger DiamondCandy has DTS^3 candies in its base, combined into a candy shape themselves. In the code above, it would result in several calls of `calc_odd()` and `calc_even()` instead of one, at the appropriate coordinates to constitute a bigger candy shape. For example, for even DTS,

```
for (int icz = Ncz-2; icz>0; icz-=2*DTS) {
  for (int icxs=0; icxs<Ncxs; icxs+=DTS) {
#pragma omp parallel for
    for (int icxa=0; icxa<Ncxa; icxa+=DTS) {
      for (int it=0; it<NT; it+=DTS){
        EVENDTS(icxs,icxa,icz+2*it);
      }
```

```
    }
#pragma omp parallel for
    for (int icxa=0; icxa<Ncxa; icxa+=DTS) {
      for (int it=0; it<NT; it+=DTS){
        EVENDTS(icxs-DTS/2,icxa-DTS/2,icz-DTS+2*it);
      }
    }
  }
}
void par::EVENDTS(int icxs, int icxa,int icz){
  for (int it=0; it<DTS; it++){
    for (int ie1=DTS-1; ie1>=0; ie1--){
      for (int ie2=DTS-1; ie2>=0; ie2--){
        for (int ie3=DTS-1; ie3>=0; ie3--){
          int ixs=((ie1+ie2)>>1)-ie3;
          int ixa=((ie1-ie2)>>1);
          int iz = (ie1+ie2);
          if (iz%2==0)
            calc_even(icxs+ixs,icxa+ixa,icz+iz+2*it);
          else
            calc_odd(icxs+ixs,icxa+ixa,icz+iz-1+2*it);
        }
      }
    }
  }
}
```

Upper limit of the DTS is determined by the fact that all data needed for one candy and the halo determined by the stencil dependencies should fit the L2 (LLC) cache size.

4 Locality

The locality parameter is proportional to the ratio of arithmetic operations to the data load/store operations. We derive a general formula here. We consider the DiamondTorre with algorithm decomposition dimension $d = 2$ and the DiamondCandy with $d = 3$ at the same time. Let o be the operation count per tile update (candy or diamond). $o = 2 \cdot 8 \cdot 32$ FLOP for the DiamondCandy, $o = 2 \cdot 4 \cdot 32 \cdot Nz$ for the DiamondTorre. Let s be the data for one candy of ShS^3 base pairs (or diamond of ShS^2 in the case of DiamondTorre). The data structure contains a candy for f and for g, so its size is $2s$. In our case, in

DiamondCandy, $s = 64$ byte for single precision and $s = 128$ bytes for double precision. In DiamondTorre, $s = 32 \cdot Nz$ single precision, $s = 64 \cdot Nz$ double precision. For one dD1T shape $NT \cdot DTS^d \cdot 2o$ operations are needed. The loaded values are the following.

- The f and g candy that will be updated on the first step $(DTS^d \cdot 2s)$;
- On each time step the data required for computation, but not loaded yet $(NT((DTS+1)^d - DTS^d) \cdot 2s)$.

Only the data that will no longer be updated in one DiamondCandy needs to be saved to the lower levels of memory. So we estimate the number of saved values by considering:

- the f and g candy that were updated on the final step $(DTS^d \cdot 2s)$
- on each step the values that will not be updated anymore $((NT-1)(DTS^d - (DTS-1)^d) \cdot 2s)$

The operations to data ratio is

$$\frac{DTS}{4 + (2DTS - 2 + 1/DTS)/NT} \frac{o}{s} \tag{3}$$

for the DiamondTorre with $d = 2$ and

$$\frac{DTS}{6 + 2/DTS^2 + (2DTS - 3 + 3/DTS - 1/DTS^2)/NT} \frac{o}{s} \tag{4}$$

for the DiamondCandy with $d = 3$.

Since a row of asynchronous shapes is computed at the same time, the data traffic is reduced on each time step by one diamond in the DiamondTorre and by (DTS+1) candies in the DiamondCandy. If we consider a row of NA asynchronous shapes, the number of operations and the amount of data is multiplied by NA, then the data for DiamondTorre is reduced by $NT \cdot NA$, the data for DiamondCandy is reduced by $NT \cdot NA \cdot (DTS + 1)$. The final formula is

$$\frac{DTS}{4 - 1/DTS + (2DTS - 2 + 1/DTS)/NT} \frac{o}{s} \tag{5}$$

for the DiamondTorre and

$$\frac{DTS}{6 - 1/DTS + 1/DTS^2 + (2DTS - 3 + 3/DTS - 1/DTS^2)/NT} \frac{o}{s} \tag{6}$$

for the DiamondCandy with $d = 3$.

5 Results

The described algorithm was implemented and the performance was tested on two computers: a desktop computer with Intel Core i5-6400 processor (SKL-S), and a high-performance node with Intel Xeon Phi 7250 (KNL). To use the AVX2 instructions on the first one and AVX512 instructions on the second one, vector length is set equal to 8 in both implementations, but the data type is changed from float to double.

Three algorithms are compared.

- A stepwise algorithm with the support of SIMD and OpenMP parallelization
- DiamondTorre algorithm [4] implemented for CPU with the use of SIMD (instead of CUDA threads) and OpenMP (instead of CUDA blocks).
- DiamondCandy algorithm described in section 3.

For the simulation, a cubic domain with $64 \times 64 \times 64$ mesh points is taken. Then it is upscaled up to the RAM limit.

The performance is measured in billions of cell updates per second. As for the algorithm parameters, NT was taken large enough so that the performance does not change much with the increase in NT. DTS was varied both for DiamondTorre and DiamondCandy. The best result from the multiple runs is plotted.

For Xeon Phi processor, the simulation was performed in the flat mode if the data is less than 16GB, and in the cache mode if it exceeds 16GB.

We see that in general LRnLA algorithms perform better than the stepwise algorithm for the large data sizes.

5.1 Core i5-6400

On SKL-S the smallest simulation data size fits the LLC cache, so we see an abrupt drop after the first point for the stepwise and DiamondTorre algorithms.

The DiamondTorre performs better than the DiamondCandy in general since the implementation uses less SIMD vector rearrange operations. One axis in the DiamondTorre is not included in the space-time decomposition. There is a one-dimensional loop in the z axis, and it is easily vectorized with the built-in compiler tools. A bit more complex intrinsic vector operations take place in the stencil computation for the DiamondCandy.

For the large data size, the DiamondTorre exhibits a drop in performance, while the DiamondCandy performance remains stable. This is what is achieved

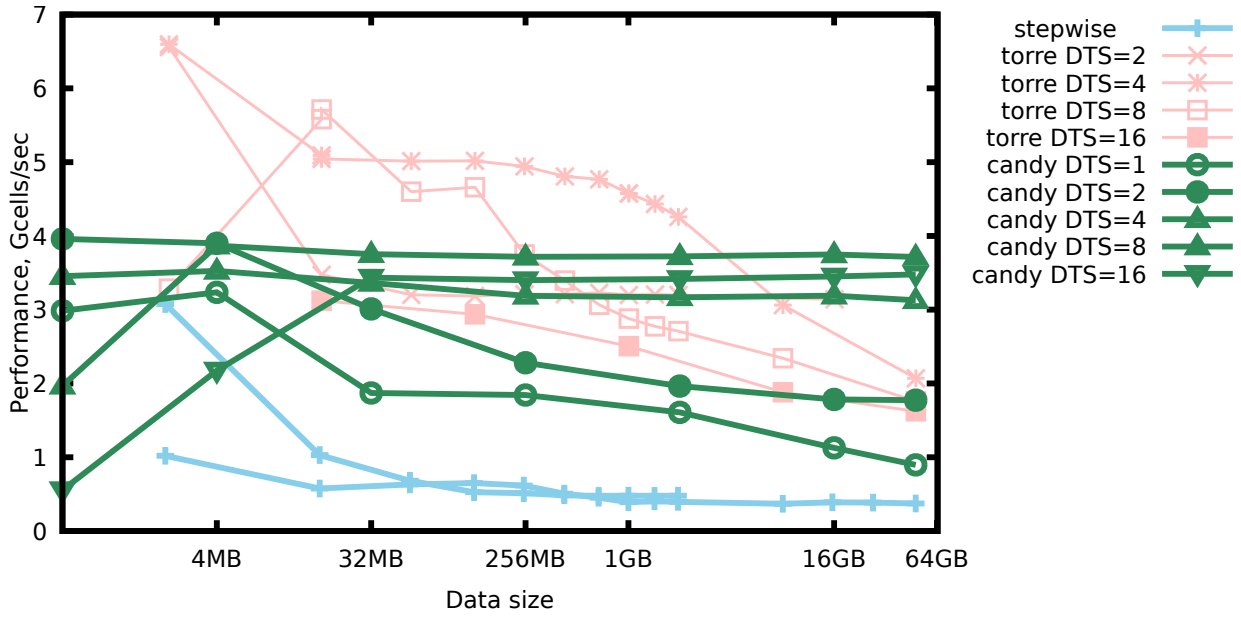

Figure 5: Performance results on Intel Core i5-6400. The lower of the blue lines shows the performance of the stepwise algorithm without OpenMP parallelisation

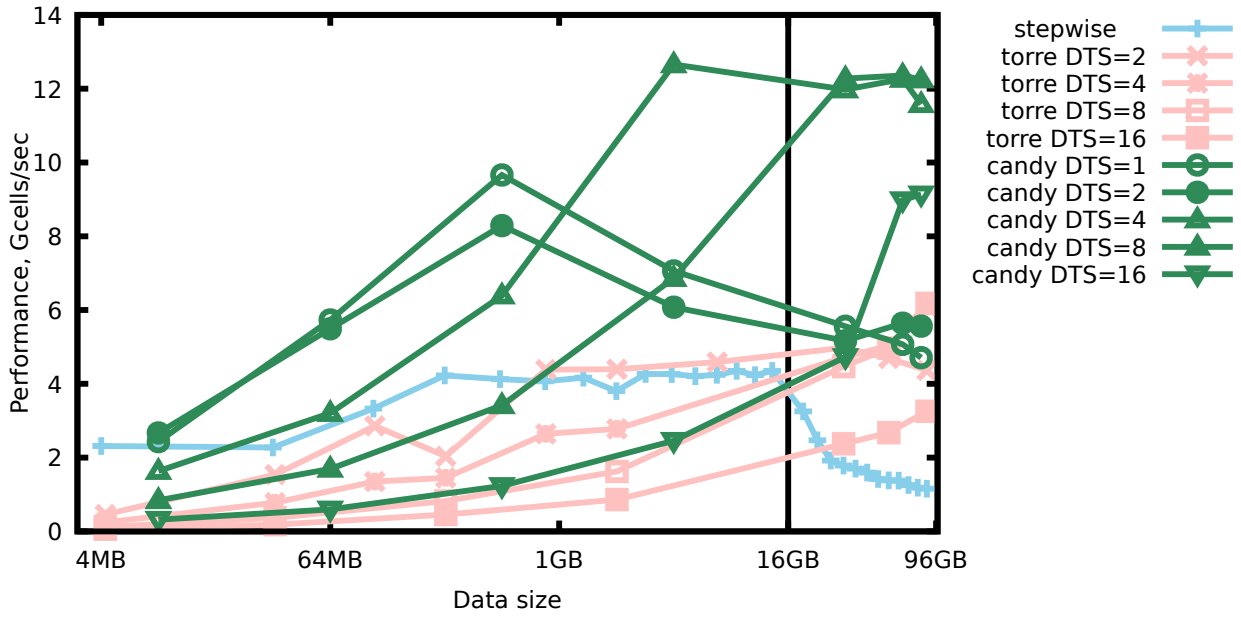

Figure 6: Performance results on KNL

by the use of the full 3D1T decomposition. DiamondTorre performs well until the data of its base fits the higher levels of memory hierarchy. Since there is no decomposition on the third axis, each DiamondTorre deals with the data that is scaled by the number of cells in z direction. So DiamondTorre performs well for the simulations where the number of cells along one axis remains small. This actually takes place in some physics problems, where the simulation domain is a flat rectangle, so in other simulation setups DiamondTorre could remain more efficient than DiamondCandy.

5.2 Xeon Phi

On Xeon Phi the stepwise algorithm performs as expected for a typical memory-bound problem, showing a rapid performance decrease when switching to the cache mode.

DiamondTorre performance does not decrease in the cache mode but, generally, does not show better results than the stepwise algorithm. For small sizes, the decomposition is not enough to use all available threads. For larger sizes, the number of cells in the third axis is too large for the data of one DiamondTorre to fit higher cache levels. DiamondCandy overcomes the limitation of both stepwise and DiamondTorre algorithms.

Weak scaling test was performed on the KNL (Fig. 7). The simulation domain was scaled with the number of threads used in the test. The size is set to $16 \times (DTS \cdot threads) \times 1024$. The scaling is close to the optimal up to 64 threads, that is, one thread per core.

6 Efficiency

6.1 Parallel efficiency

To see how efficient the DiamondCandy traversal algorithm is, a special benchmark is created. With a minimal amount of data, the `stencil()` function for one candy is run multiple times. No data locality problems arise, so it is expected to be the maximum performance of the current implementation of the vectorized computation. When several threads are used, they compute separate problems without synchronization.

For 1 thread run the performance is $\sim 30\%$ of the peak value on KNL and $\sim 50\%$ on SKL-S. For maximum number of threads, the performance is 50% of the peak value both on KNL and SKL-S. Since we are dealing with memory bound problems, for actual simulation no more than 1 thread per core is set for KNL runs. Thus, the performance results are reduced to 30% efficiency

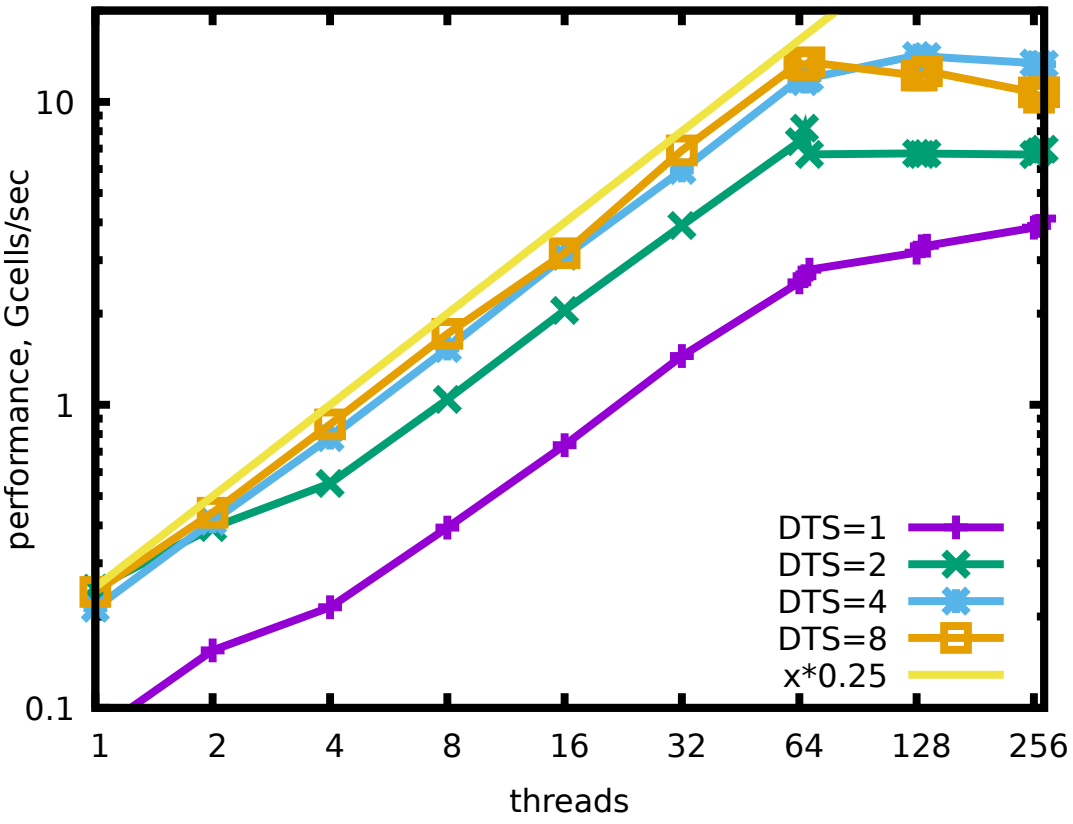

Figure 7: Performance results on KNL

regardless of the traversal algorithm.

6.2 Memory efficiency

The roofline model [1] is a helpful tool to see the limitations of the algorithm. It shows the limitations of memory bandwidth and computational performance on one graph.

We develop the estimations of the algorithm performance efficiency based on this model and compare with the results we got in the performance tests in section 5 (Fig. 8,9).

The vertical axis is performance, the horizontal axis is the operational intensity. Operational intensity is the number of FLOP per byte of memory traffic. It is proportional to the locality parameter discussed in section 4.

The basic roofline is composed of two line segments. To the right, the horizontal line shows the peak compute performance limit of the computer. To the left, the inclined line shows the memory bandwidth level. We plot it for several memory levels.

We add the estimations of the operational intensity for the studied algorithms to the graph.

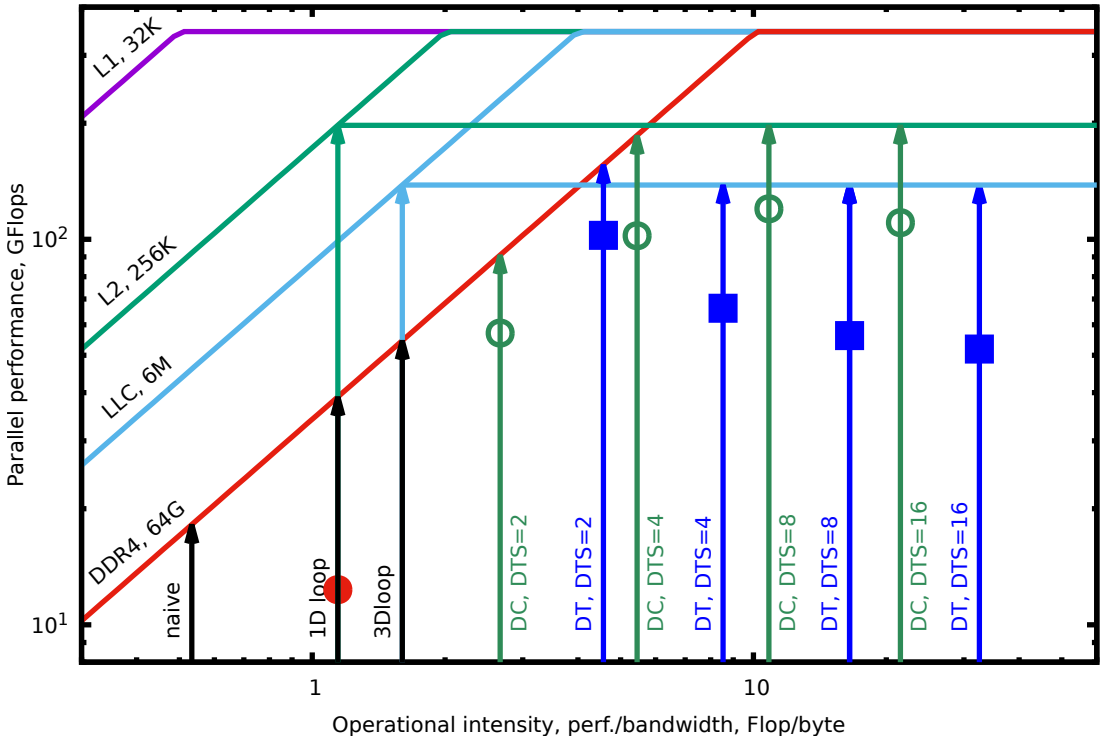

Figure 8: Roofline model for SKL-S with algorithm estimations and performance results. DC — DimondCandy, DT — DiamondTorre.

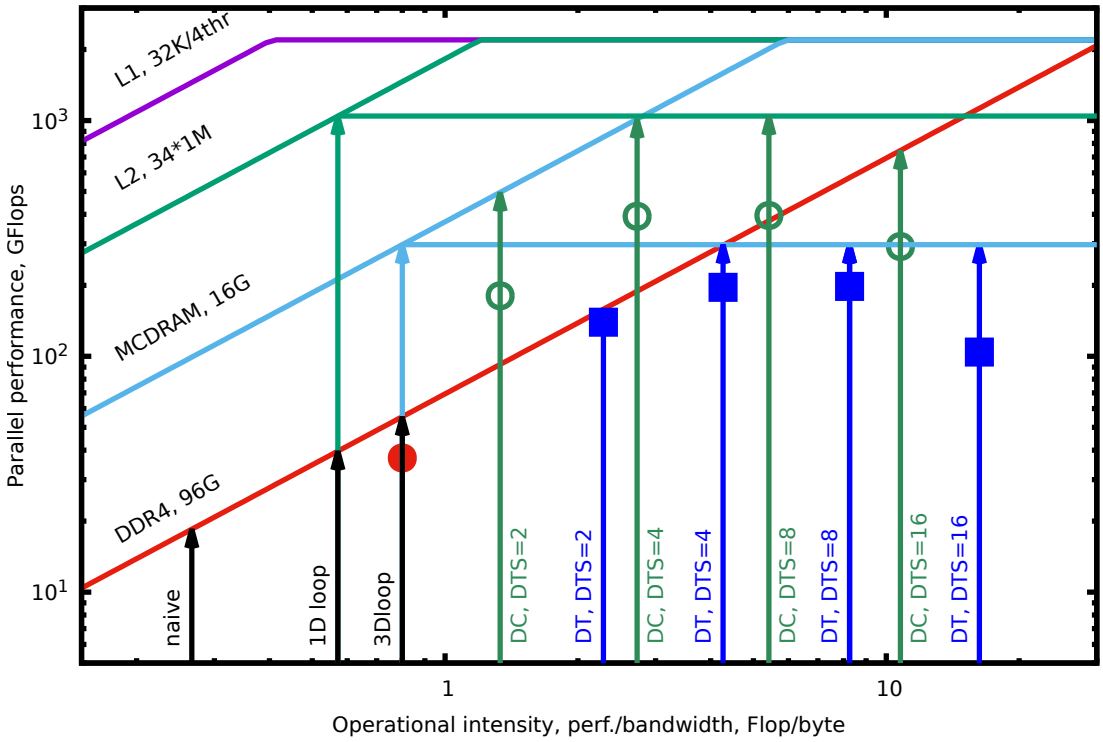

Figure 9: Roofline model for KNL with algorithm estimations and performance results. DC — DimondCandy, DT — DiamondTorre.

We assume that there are 32 FLOP per cell update in the chosen scheme, $s = 8$ byte per value on KNL, $s = 4$ byte per value on SKL-S.

The black arrows show the estimate of the stepwise algorithm with different ideal implementations.

- *Naive* is the stepwise algorithm without data reuse. The stencil takes 15 values, so the intensity is estimated as $32/(15s)$.
- In *1D loop* algorithm it is assumed that data is cached when looping over the z-axis. 4 of the stencil points are already in cache, and the estimate is $32/(11s)$.
- In the *3D loop* it is assumed that all previously loaded data is stored in the cache, so only 5 new points need to be loaded per cell update.

The high point of the black arrows is at the DDR4 level because the relevant simulation domain size does not fit the higher levels of memory.

The operational intensity of the LRnLA algorithms is estimated for $NT \gg 1$ as

$$\frac{DTS}{4 - 1/DTS}\frac{1}{s} \quad \text{[FLOP/byte]} \tag{7}$$

for DiamondTorre [4] and

$$\frac{DTS}{6 - 1/DTS + 1/DTS^2}\frac{1}{s} \quad \text{[FLOP/byte]} \tag{8}$$

for DiamondCandy.

Another limitation for DiamondCandy is the traversal rule inside it. For higher DTS it is also a 3D loop on each spatial 3D slice of the candy. Thus, the stepwise limitation applies. However, since one DiamondCandy uses less data, the data is localized in higher memory levels. The data required in one spatial slice is estimated as $2 \cdot 2 \cdot s(ShS(DTS + 1))^3$ for DiamondCandy and $2 \cdot 2 \cdot s(ShS(DTS + 1))^3$ for the DiamondTorre. The performance limit is decided by the 3D loop (z-loop for DiamondTorre) limit for the memory level into which the required data fits. To show this limitation we plot the horizontal lines that go to the right from the arrow points for stepwise algorithms.

The markers over the arrows show the results from section 5 for large data size.

7 Discussion

We have constructed a new LRnLA algorithm DiamondCandy for cross-stencil scheme implementation.

Its novelty among the temporal blocking schemes is that its base follows the diamond (octahedron) shape, which is the base of the dependency conoid of the cross-stencil. At the same time, it homogenously tiles the 3D1T domain, can be recursively subdivided into similar shapes. It results in a better locality, parallel ability, ease of implementation. It makes use of the AVX2/AVX512 vectorization.

The test performance results showed its advantage over the stepwise algorithm and our previous invention, the DiamondTorre LRnLA algorithm. The fact that the performance does not decrease for the large data size is the most important achievement since real physics problems require large computation domains.

The implementation still has some potential. Namely, the vectorization method is inferior to that of the DiamondTorre and stepwise algorithms. The reason is that the compilers and the hardware are better at vectorizing simple loops, than complex structures. Moreover, DiamondCandy has more ability to parallelize than was currently used. When one asynchronous row is computed, three rows may be started after it at the same time. The bases of the two of them are at the same time step (below in z axis, and 'behind' in x_s axis), and one is located NT time steps after at the same place where the first one ended. This may be achieved with mutex-based synchronization in the implementation. To get even more data access locality, the data storage may be organized in larger structures, or a Morton curve may be used.

The algorithm is illustrated here for the wave equation, so it may be used, for example, for problems in acoustic simulation. But the main significance is that it may be used in any memory-bound cross stencil computation. It also may be used for other stencils, if they are enclosed by a diamond (octahedron) shape.

The MPI level of parallelism was not shown in this work. It is expected to be implemented in a similar way it was done for large-scale DiamondTorre computations [16]. With DiamondTorre, data transfer was concealed completely by computation with the right parameter choice. The same parameters are applied in the DiamondCandy description, so good scaling is expected.

We plan to use DiamondCandy in our future wave modeling applications, such as optic and elastic wave propagation. We expect that the advance in time-space approach to the numerical solution of the systems of partial differential equations will help to raise the efficiency of the supercomputer use, and conquer new frontiers in technological and scientific progress.

The authors that Colfax Intl. for providing access to the computers with Xeon Phi processors, education on its use, helpful tips on code implementation and execution.

References

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

[2] R. M. Karp, R. E. Miller, and S. Winograd, "The organization of computations for uniform recurrence equations," *Journal of the ACM (JACM)*, vol. 14, no. 3, pp. 563–590, 1967.

[3] V. D. 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] V. Levchenko, A. Perepelkina, and A. Zakirov, "DiamondTorre algorithm for high-performance wave modeling," *Computation*, vol. 4, no. 3, p. 29, 2016.

[5] V. D. Levchenko, "Simulation of super-large-size plasma physics problems on cheap computational sytems using.local schemes taking into account the dependence region," in *16th International Conference on the Numerical Simulation of Plasmas*, pp. 147–150, 1998.

[6] L. In'kov and V. D. Levchenko, "Optimization of PIC method in the SUR code via object-oriented plasma model," *Keldysh Institute Preprints*, no. 133, 1995.

[7] A. V. Zakirov and V. D. Levchenko, "The effective algorithm for 3D modeling of electromagnetic waves' propagation through photonic crystals," *Keldysh Institute Preprints*, p. 21, 2008.

[8] N. Elkina and V. D. Levchenko, "SUR/MP: Parallel PIC 3D code. I. Maxwell equations," *Keldysh Institute Preprints*, pp. 48–1, 2000.

[9] T. Malas, G. Hager, H. Ltaief, H. Stengel, G. Wellein, and D. Keyes, "Multicore-optimized wavefront diamond blocking for optimizing stencil updates," *SIAM Journal on Scientific Computing*, vol. 37, pp. C439–C464, jan 2015.

[10] U. Bondhugula, V. Bandishti, and I. Pananilath, "Diamond tiling: Tiling techniques to maximize parallelism for stencil computations," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 5, pp. 1285–1298, 2017.

[11] A. Y. Perepelkina, V. D. Levchenko, and I. A. Goryachev, "Implementation of the kinetic plasma code with Locally Recursive non-Locally Asynchronous Algorithms," in *Journal of Physics: Conference Series*, vol. 510, p. 012042, IOP Publishing, 2014.

[12] A. Zakirov, *Application of the Locally Recursive non-Locally Asynchronous algorithms in the full wave modeling (in Russian).* PhD thesis, MIPT, Moscow, 2012.

[13] L. Lamport, "The parallel execution of DO loops," *Communications of the ACM*, vol. 17, no. 2, pp. 83–93, 1974.

[14] D. Wonnacott, "Achieving scalable locality with time skewing," *International Journal of Parallel Programming*, vol. 30, no. 3, pp. 181–221, 2002.

[15] T. Muranushi and J. Makino, "Optimal temporal blocking for stencil computation," *Procedia Computer Science*, vol. 51, pp. 1303–1312, 2015.

[16] A. Zakirov, V. Levchenko, A. Perepelkina, and Y. Zempo, "High performance FDTD algorithm for GPGPU supercomputers," in *Journal of Physics: Conference Series*, vol. 759, p. 012100, IOP Publishing, 2016.

[17] A. Zakirov, V. Levchenko, A. Ivanov, A. Perepelkina, T. Levchenko, and V. Rok, "High-performance 3D modeling of a full-wave seismic field for seismic survey tasks," *Geoinformatika*, no. 3, pp. 34–45, 2017.

[18] 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, PO Scholten, Ed*, no. 38F, p. O2.304, 2014.

[19] B. Korneev and V. Levchenko, "Detailed numerical simulation of shock-body interaction in 3D multicomponent flow using the RKDG numerical method and "DiamondTorre" GPU algorithm of implementation," in *Journal of Physics: Conference Series*, vol. 681, p. 012046, IOP Publishing, 2016.

[20] V. Levchenko and A. Perepelkina, "The DiamondTetris algorithm for maximum performance vectorized stencil computation," in *International Conference on Parallel Computing Technologies*, pp. 124–135, Springer, Cham, 2017.

[21] P. Feautrier, "Some efficient solutions to the affine scheduling problem: Part I. one-dimensional time," *International Journal of Parallel Programming*, vol. 21, pp. 313–348, October 1992.

[22] F. Quilleré, S. Rajopadhye, and D. Wilde, "Generation of efficient nested loops from polyhedra," *International journal of parallel programming*, vol. 28, no. 5, pp. 469–498, 2000.

[23] C. Yount and A. Duran, "Effective use of large high-bandwidth memory caches in HPC stencil computation via temporal wave-front tiling," in *Proceedings of the 7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems*, PMBS '16, (Piscataway, NJ, USA), pp. 65–75, IEEE Press, 2016.