VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

PERFORMANCE ENGINEERING OF STENCILS OPTIMIZATION IN GEOMETRIC MULTIGRID

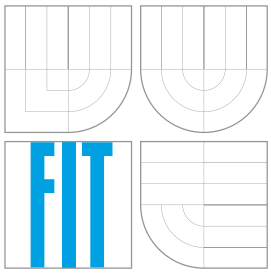
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. RADIM JANALÍK

BRNO 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

OPTIMALIZACE VÝPOČTU V MULTIGRIDU

PERFORMANCE ENGINEERING OF STENCILS OPTIMIZATION IN GEOMETRIC MULTIGRID

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. RADIM JANALÍK

VEDOUCÍ PRÁCE
SUPERVISOR

Doc. Ing. JIŘÍ KUNOVSKÝ, CSc.
Prof. Dr. OLAF SCHENK

BRNO 2015

Abstrakt

V této práci představujeme blokovou metodu pro zlepšení lokality v cache paměti u výpočtů typu stencil a dva nástroje, Pluto a PATUS, které tuto metodu používají ke generování optimalizovaného kódu. Provádíme různá měření a zkoumáme zrychlení výpočtu při použití různých optimalizací. Nakonec implementujeme vyhlazovací krok v multigridu s různými optimalizacemi a zkoumáme jak se tyto optimalizace projeví na výkonu multigridu.

Abstract

In this work we present spatial and temporal blocking methods to exploit cache locality in stencil computations and two state of the art optimizers, Pluto and PATUS, that use these methods to generate optimized code. We perform various measurement to investigate the speedup using different optimizations. At the end we implement smoothing step in multigrid with different optimizations and measure impact of these optimizations on the performance of multigrid.

Klíčová slova

Stencil, optimalizace, Pluto, PATUS, tiling, spatial blocking, temporal blocking, aritmetická intenzita, roofline model, multigrid.

Keywords

Stencil, optimization, Pluto, PATUS, tiling, spatial blocking, temporal blocking, arithmetic intensity, roofline model, multigrid.

Citace

Radim Janalík: Performance Engineering of Stencils Optimization in Geometric Multigrid, diplomová práce, Brno, FIT VUT v Brně, 2015

Performance Engineering of Stencils Optimization in Geometric Multigrid

Declaration

Hereby I declare that I was working on this master thesis independently under the guidance of Prof. Dr. Olaf Schenk and Doc. Ing. Jiří Kunovský, CSc.
I referenced all resources I used in this work.

.....
Radim Janalík
May 27, 2015

Acknowledgment

I would like to thank Prof. Dr. Olaf Schenk for his great guidance and many valuable advices. I would also like to thank Doc. Ing. Jiří Kunovský, CSc. for his support and allowing the cooperation with Prof. Schenk.

© Radim Janalík, 2015.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Contents

Chapter 1

Introduction

Many scientific applications depend on solving boundary value systems of partial differential equations. Large scale systems are solved on supercomputers that provide enought computational power and memory. Finite difference method, which is based on stencil computation, is often used to solve such a systems. But finite difference method in its basic form is far from optimal. To prevent waste of resources, it is necessary to optimize the computations as much as possible.

Geometric Multigrid is one of the methods how to optimize solving of partial differential equations. It is a mathematical approach that allows us to speed up convergence of finite difference method. It uses many grids of different sizes and transformations between them to make the solution converge quickly.

But for solving the system, Geometric Multigrid uses the same stencil code as the basic finite difference method. For stencil codes is typical very low arithmetic intensity. Stencil computations are limited by memory bandwidth and cannot fully utilize computational power of the computer.

In this work we focus on spatial blocking and temporal blocking optimizations, also known as tiling methods. These optimizations increase cache locality. Elements loaded from main memory to cache are kept in cache as long as possible. This results in reduction of traffic between main memory and cache and allows using more of the computational power.

In chapter 2 we provide introduction to Geometric Multigrid. We describe the basic idea of Multigrid that quickly eleminate the error and therefore leads to speeding us the convergence. Then we describe all of the components of multigrid and show how they work together.

In chapter 3 we describe different stencil kernel operations that we later use in our experiments. On these kernels we show their limitation by memory throughtput so they cannot utilize all of the computational power. At the end of the chapter we explain spatial blocking and temporal blocking techniques to increase cache locality.

In chapter 4 we show a basic implementation of stencil code and two state of the art optimizers, Pluto and PATUS, that use spatial and temporal blocking techniques. In this chapter we also describe how we combined these optimizers to use advantages of both in the same time.

Finally chapter 5 describes experiments we performed. In the first experiment an effect of vectorization that is done by compiler is shown. Next two experiments show the effect of spatial and temporal blocking on the performance. In the last experiment we implement smoothing step in multigrid with different optimizations and show how different optimizations affect the solution.

Chapter 2

Geometric Multigrid

Finite difference method is well known method for solving boundary value systems of differential equations. The value at the boundary is known and the goal is to compute the distribution of value in space. To do this, we discretize the continuous space and create uniformly distributed grid of points. Crucial question is how many points should the grid contain. More points means better approximation of analytical solution, but the solution can take a lot of time. On the contrary using grid with less points the solution will converge quickly, but with lower precision.

Geometric multigrid is widely used method to speed up convergence of numerical methods. Multigrid uses several grids of different sizes and transformations between them [1]. On a coarse grid it gets rough estimate of the solution and later the solution is improved on a fine grid. Solution on a grid is done using stencil code, the same way as it is done in finite difference method.

2.1 Residual equation

To explain the theory we use the same notation as is used in [1].

$$Au = f \tag{2.1}$$

represents a system of linear equations. Vector u denotes exact solution and v denotes it's approximation. Error e can by expressed as

$$e = u - v \tag{2.2}$$

However we don't know the exact solution u, so we cannot compute the error. But we can compute residual

$$r = f - Av \tag{2.3}$$

For exact solution u, residual equals zero. From equations 2.1, 2.2 and 2.3 we can derive residual equation 2.4

$$Ae = r \tag{2.4}$$

We can see that equations 2.1 and 2.4 are very similar, so the error approximation can be computed the same way as approximation of exact solution. We only need to replace f

by the residual r. If we have some approximation v, we can get residual from equation 2.3. Then we can compute approximation of error using equation 2.4 and finally use equation 2.2 to get new approximation of exact solution.

$$u = v + e \tag{2.5}$$

2.2 Restriction and Interpolation

We cannot compute error estimation on the same grid as the solution v. If we do so, it will have the same effect as performing more iterations on equation 2.1 and it will not lead to desired speedup. To get the speedup we need to compute error estimation on coarse grid.

We need to define two operators to move between different grids. The operator for moving from fine grid to coarse grid is called *restriction operator* and the one for moving from coarse grid to fine grid is called *interpolation operator*. It is possible to move between any two grid sizes, but for practical use we can assume that the number of points in every dimension changes by factor of two ($2n + 1 \to n + 1$ or $n + 1 \to 2n + 1$).

Restriction operator $I_h^{2h} v^h = v^{2h}$ transforms fine grid v^h to coarse grid v^{2h}. The easiest, but not very good, way how to do the transformation is just to take appropriate points from the fine grid.

$$v_{i,j}^{2h} = v_{2i,2j}^{h}, \quad 1 \leq i, j \leq \frac{n}{2} - 1 \tag{2.6}$$

Disadvantage of this approach is, that the information carried by thrown-away points is lost. Better solution is to take weighted average of neighboring points. Figure 2.1 shows which points on 2D grid are averaged. Equation 2.7 describes precise formula of the restriction.

$$\begin{aligned} v_{i,j}^{2h} &= \frac{1}{16}\left(v_{2i-1,2j-1}^{h} + v_{2i-1,2j+1}^{h} + v_{2i+1,2j-1}^{h} + v_{2i+1,2j+1}^{h}\right) \\ &+ \frac{1}{8}\left(v_{2i,2j-1}^{h} + v_{2i,2j+1}^{h} + v_{2i-1,2j}^{h} + v_{2i+1,2j}^{h}\right) \\ &+ \frac{1}{4} v_{2i,2j}^{h}, \qquad 1 \leq i, j \leq \frac{n}{2} - 1 \end{aligned} \tag{2.7}$$

Interpolation operator $I_{2h}^{h} v^{2h} = v^h$ transforms coarse grid v^{2h} to fine grid v^h. Very easy and sufficient is linear interpolation. Value of point that has its equivalent on coarse grid is taken without any change. The value of the other points can be calculated as average of the closest points on coarse grid. Figure 2.2 shows which points on 2D grid are averaged. Equation 2.8 describes precise formula of the interpolation.

$$\begin{aligned} v_{2i,2j}^{h} &= v_{i,j}^{2h}, \\ v_{2i+1,2j}^{h} &= \frac{1}{2}\left(v_{i,j}^{2h} + v_{i+1,j}^{2h}\right), \\ v_{2i,2j+1}^{h} &= \frac{1}{2}\left(v_{i,j}^{2h} + v_{i,j+1}^{2h}\right), \\ v_{2i+1,2j+1}^{h} &= \frac{1}{4}\left(v_{i,j}^{2h} + v_{i+1,j}^{2h} + v_{i,j+1}^{2h} + v_{i+1,j+1}^{2h}\right), \quad 1 \leq i, j \leq \frac{n}{2} - 1 \end{aligned} \tag{2.8}$$

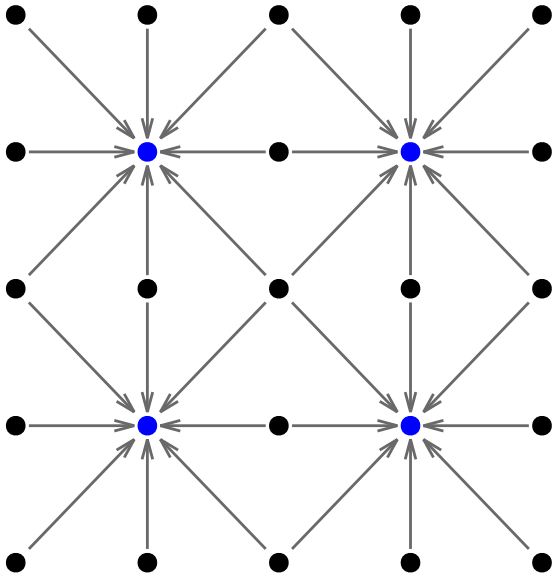


Figure 2.1: Restriction operation. Arrows show which points from fine grid (black and blue) are averaged to get point on coarse grid (blue).

2.3 V-cycle

We can look at the error as any other function. And as any function can be decomposed to sum of sine functions with different frequencies, we can decompose error as well. Doing that, we can investigate how amplitude of every single sine function changes over time.

We find out that all the amplitudes decrease, as the whole error do. But while the high frequency errors wanish out within a few iterations, the low frequency errors remain for hundrets of iterations. This is because when the wave length is short, then value of every point is quickly spreaded within its half-wave, but when the wave length is long, then it takes many iterations.

If we resample the function to coarse grid, the frequency remains the same, but the wave length in matter of grid points gets shorter. We can see transformation of three different functions from grid with 13 points to grid with 7 points in the figures 2.3 and 2.4.

As the wave length gets shorter when we move to coarse grid, the error decreases faster, as if the frequency would be higher. But there is a limit for that. The limit is Nyquist-Shannon sampling theorem, which says that the sampling frequency must be at least double of the signal frequency. If this condition is not met, then the frequency appears to be lower than it actually is. We can see this at the third sine function in the figures 2.3 and 2.4. At the grid with 13 points everything is fine, but after transformation to grid with 7 points the sampling theorem dosn't hold anymore. The points now represent sine function with lower frequency (the blue one). It means that this error will decrease slower on the coarse grid.

In the context of solving equation 2.1 if we move to coarse grid, the low frequency errors become high frequency errors and vice versa. To make convergence faster, we can first perform a few iterations of numerical method to get rid of high frequency errors. Then we can move to coarse grid and perform a few iterations to get rid of low frequency errors (which are high frequency errors on coarse grid) and go back to fine grid. However the

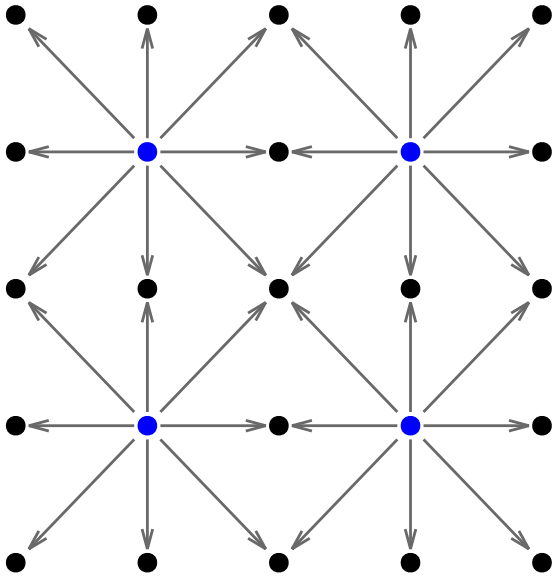


Figure 2.2: Interpolation operation. Arrows show which points from coarse grid (blue) are averaged to get point on fine grid (black).

transformation introduces some error and also solution on coarse grid is not as precise as on fine grid. So after returning to fine grid we need to perform several additional iterations.

We don't need to move to coarse grid only once. While solving on coarse grid, we can apply the same idea and move to even coarse grid. This gets us to the deffinition of V-cycle. First we perform ν_1 iterations on equation 2.1 and compute residual by equaion 2.3. Then we restrict residual r to coarse grid and set u on coarse grid to zeros. Then we call recursively V-cycle for coarse grid, but we swap r and u for the call. After the computation on coarse grid is finished we interpolate the residual r from coarse to fine grid and correct the solution according the equation 2.5. Finally, after the correction we perform ν_2 additional iterations on equation 2.1. We can see pseudocode of V-cycle in the algorithm 2.1.

Function $V^h(v^h, f^h)$:
    If Ω^h = coarsest grid:
        Perform ν_c iterations on $A^h u^h = f^h$ with initial guess v^h
    Else:
        Perform ν_1 iterations on $A^h u^h = f^h$ with initial guess v^h
        $f^{2h} \leftarrow I_h^{2h}(f^h - A^h v^h)$
        $v^{2h} \leftarrow 0$
        $v^{2h} \leftarrow V^{2h}(v^{2h}, f^{2h})$
        $v^h \leftarrow v^h + I_{2h}^h v^{2h}$
        Perform ν_2 iterations on $A^h u^h = f^h$ with initial guess v^h

Algorithm 2.1: Recursive definition of V-Cycle. Taken from [1].

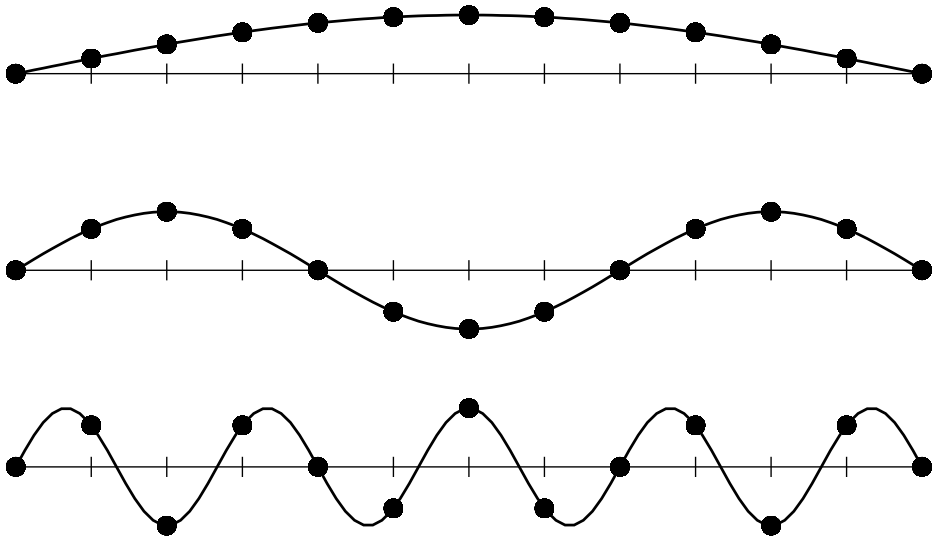


Figure 2.3: Sine functions with 1, 3 and 9 half-waves discretized at grid with 13 points.

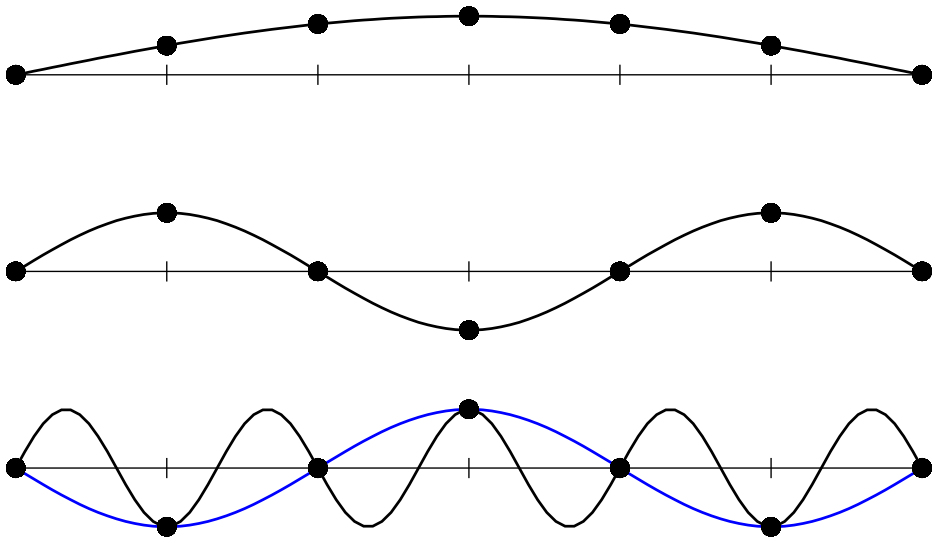


Figure 2.4: Sine functions with 1, 3 and 9 half-waves discretized at grid with 7 point. For the third function sampling theorem does not hold, so it appears as if the function would have lower frequency.

Chapter 3

Stencil code performance engineering

Stencil code is frequently used class of computations in many different areas of scientific computing, for example partial differential equation solvers, celular automata or in image processing. Stencil code is an iterative computation on a structured grid. Every point is updated based on values of its neighboring points.

Very important thing is how individual elements are accessed within one iteration. If there are distinct sets of input and output points, typically one input and one output grid, then it is called *Jacobi iteration*. As there is no dependency between two points, the points can be visited in any order. This is very convenient for optimizations and for paralelization.

If some points are used as both input and output in one iteration, then the order in which the nodes are visited is important and cannot be changed. This case is called *Gauss-Seidel iteration* [2]. This limitation makes optimizations and paralelization difficult. In this work we will focus only on more suitable case for optimizations, the Jacobi iteration.

In the figure 3.1 we can see four examples of stencil kernels. Three of them we use for experiments that are discussed in chapter 5. In the figures 3.1a and 3.1b there are 2D and 3D Laplacian kernels. To update value of an element, only values of its neighbors are needed. These kernels are also used at smoothing step in Geometric Multigrid. In the figures 3.1c and 3.1d there are 2D and 3D Longrange kernels. These kernels need values from radius of 4 from the updated element. Increasing cache locality for these kernels is more complicated because of limited cache size.

3.1 Arithmetic intensity

The arithmetic intensity of an algorithm is the average number of floating point operations per one byte of DRAM memory traffic [4].

$$I = \frac{\#\text{ flops}}{\#\text{ loads and stores}} \tag{3.1}$$

For stencil computations is typical that memory access pattern is regular. Update of every grid point requires the same amount of floating point operations, reads and writes the same amount of data. Strictly speaking, the number of read items is not constant because data are loaded from DRAM to cache by entire cache line, not one by one element. However the arithmetic intensity is defined as an average, so to determine arithmetic intensity we

can assume loading one by one element.

$$\begin{aligned} A[x,y,z] &= 0.125 * (B[x+1,y,z] + B[x-1,y,z] \\ &\quad + B[x,y+1,z] + B[x,y-1,z] \\ &\quad + B[x,y,z-1] + B[x,y,z+1]) \\ &+ 0.25 * B[x,y,z] \end{aligned} \tag{3.2}$$

Figure 3.1 shows memory access patterns of four stencil kernels. Let's show how to determine the arithmetic intensity of 3D Laplacian kernel (figure 3.1b). The kernel operation is also shown in the equation 3.2. We can see that there are 6 additions and 2 multiplications, it is in total 8 floating point operations. Values of 7 elements are needed, 2 are already loaded in cache after update of previous point, 5 need to be load from memory and one element will be stored, which requires 2 memory operations in case of Write Allocate[1]. This is 7 transfers between memory and cache. We perform calculations in double precision, every element has 8B. This gives us 56B of memory transfer and the arithmetic intensity is

$$I = \frac{8}{56} = 0.14 \tag{3.3}$$

We can see arithmetic intensity of the other kernels in the table 3.1.

Stencil kernel	**Arithmetic intensity [flops/byte]**
2D Laplacian	0.15
3D Laplacian	0.14
2D Longrange with variable coeficients	0.16
3D Longrange with variable coeficients	0.14

Table 3.1: Arithmetic intensity of different stencil kernel operations.

3.2 Roofline model

In [4] there is introduced a performance model that can be used in performance engineering. Because of its shape it is called roofline model. In the figure 3.2 we can see the roofline model of Intel Xeon E5-2660 v2 Ivy Bridge Processor.

The roofline model is a graph that shows dependance of floating point performance (flops/s) on arithmetic intensity (flops/byte) of the algorithm. The graph consits of two lines, the horizontal line shows the peak performance and the other one shows performance for arithmetic intensity where peak performance cannot be achieved because of a memory bandwidth limitation. This two lines intersect in a point where the procesor achieves maximum performance and maximum memory throughput in the same time. This ballanced state is achieved for arithmetic intensity

$$I = \frac{\text{Peak performance}}{\text{Memory bandwidth}} \tag{3.4}$$

We can measure Peak performance and Memory bandwidth by microbenchmark or find these values in the CPU specification.

[1] Write Allocate requires a variable to be loaded to cache before updated value is stored. If the variable is not in cache yet, then write requires to perform both load and store instructions, which takes 2 cycles.

If we increase arithmetic intensity from this point, the performance will stay on the maximum possible value and memory throughput goes down, the line on the right is horizontal. On the other side if we decrease arithmetic intensity, the memory throughput stays on the maximum while performance decreases. The left line goes from zero (for arithmetic intensity equals to zero) to the balanced point.

If we know the roofline model of given architecture and the arithmetic intensity of some algorithm, we can tell whether the algorithm is CPU of memory bounded. If the algorithm lays in the left part of the graph, it is memory bounded. We are not able to fully utilize the CPU because the memory bandwidth is the bottleneck. Our focus should be to improve the memory efficiency of the algorithm. If the algorithm lays in the right part of the graph, it is CPU bounded. We should look for algorithm that doesn't need so many arithmetic operations, e.g. by avoiding some useless computations.

3.3 Supercomputer architectures

According to the list of 500 most powerful supercomputers [5] up to 2008 in all of the supercomputers were using only classical CPUs. Since most of the supercomputers were equiped by Intel Xeon processors. In the figure 3.2 we saw roofline model of Intel Xeon E5-2660v2 Ivy Bridge processor. In the graph we can see that the balanced point is at arithmetic intensity about 2.9 flops/byte.

In 2008 the most powerful computer and the first computer with peak performance over 1 petaflop/s, Roadrunner, was equiped by hybrid processors IBM PowerXCell 8i. This is 9 cores processor with one PPE[2] cores and eight SPE[3] cores. Peak performance of this processor is 102.4 Gflop/s in double precision and memory bandwidth 25.6 GB/s, which means that the peak performance can be acheved with arithmetic intensity 4 flops/byte or higher [6].

From 2010 Nvidia Tesla accelerator apears. It is GPU device with many cores and very low energy consumption. The 2010 model Tesla C2050 has 448 cores, performance 515 Gflop/s in double precision and memory bandwidth 144 GB/s [7]. To utilize the performance arithmetic intensity 3.6 flops/byte is needed. The newest model of CUDA accelerator of HPC, Tesla K80 released in 2014, wits 4992 cores, double precision peak performance 2.91 Tflop/s and 480 GB/s memory bandwidth needs arithmetic intensity at least 6 flops/byte to use all its computational power [8].

In 2012 first supercomputers with Intel Xeon Phi coprocessors appear. Xeon Phi is a coprocessor unit with 61 cores, theoretical peak performance in double precision 1208.29 Gflop/s, memory bandwidth 352 GB/s in the latest model 7120A [9]. Required arithmetic intensity is at least 3.4 flops/byte.

We can see that the trend in supercomputer design is using energy efficient accelerators that require high arithmetic intensity to use all the computational power. For algorithms, it is inportant to follow this trend, otherwise it is not possible to use all the advantages that supercomuters offer.

[2] Power Processor Element
[3] Synergistic Processing Elements

3.4 Tiling

Tiling is set of techniques for increasing arithmetic intensity of algorithm by reusing elements that are loaded to cache. We can see the principle in the figure 3.3. In normal order when the whole row is updated before proceeding to the next row, the element is loaded to cache, the update is done and after a while the element is released from cache to free space for another element. When the computation continues to the next row, the element must be loaded again when updating an element nearby.

If we split the grid to blocks with width of the block so small that several lines of the block fit in the cache, then the element is loaded, all the updates that require this element are done and after that the element is released. This saves many load operations and therefore increases the arithmetic intensity. This optimization is called *spatial blocking*.

Tiling can also be done in time domain. If we set tile size small enough that the whole tile fits in the cache, then after updating the whole tile, all the elements remain in the cache. We can immediately start next iteration in this tile and achieve additional increase of arithmetic intensity. Scheduling of work is more complicated than at spatial blocking because it is necessary to ensure that for each update only values from previous iteration are used. This technique is called *temporal blocking*.

In the section 3.1 we showed how to compute arithmetic intensity of the basic algorithm without any optimization. Now we show on the same example how arithmetic intensity increases using spatial blocking. We found that arithmetic intensity of 3D Laplacian kernel (equation 3.2) is 0.14. Using spatial blocking the number of floating point operations remains the same, but but number of elements loaded from main memory changes. Number of elements to read goes down from 5 to 1 and 1 element to write remains the same. It is 8 floating point operations and 3 memory operations (one for load and two for write allocate). This gives us arithmetic intensity

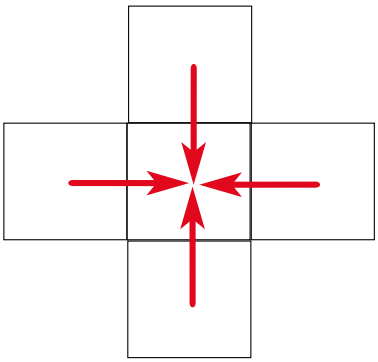
$$I_{spatial} = \frac{8}{3 * 8} = 0.33 \tag{3.5}$$

which is about twice better than without spatial blocking, so we can expect twice better performance because the algorithm is memory bounded.

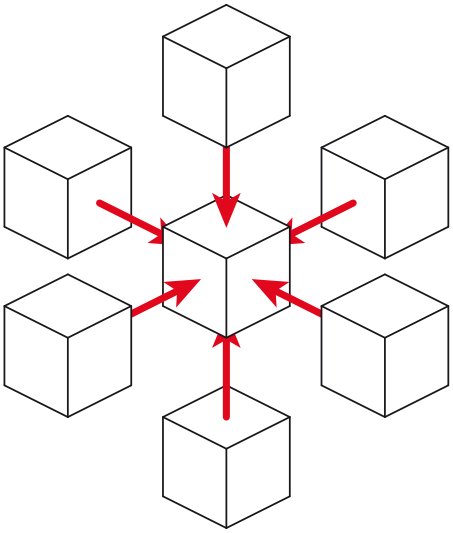
If we add temporal blocking optimization, then the number of loads and stores remains the same as at the spatial blocking, but number of floating point operations increases as more iterations are performed. The arithmetic intensity for k iterations is then

$$I_{temporal} = k * I_{spatial} \tag{3.6}$$

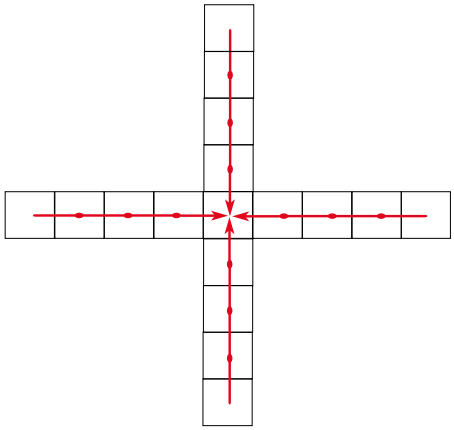
In chapter 4 we describe two optimizers and code generators, Pluto and PATUS, that implement the spatial blocking and temporal blocking strategies to increase arithmetic intensity.



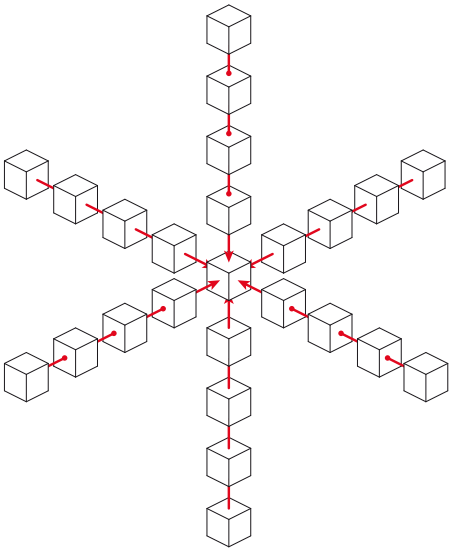
(a) 2D 5point Laplacian stencil.



(b) 3D 7point Laplacian stencil.



(c) 2D 17point Longrange stencil.



(d) 3D 25point Longrange stencil.

Figure 3.1: Examples of stencil kernels. Figures 3.1b and 3.1d taken from [3].

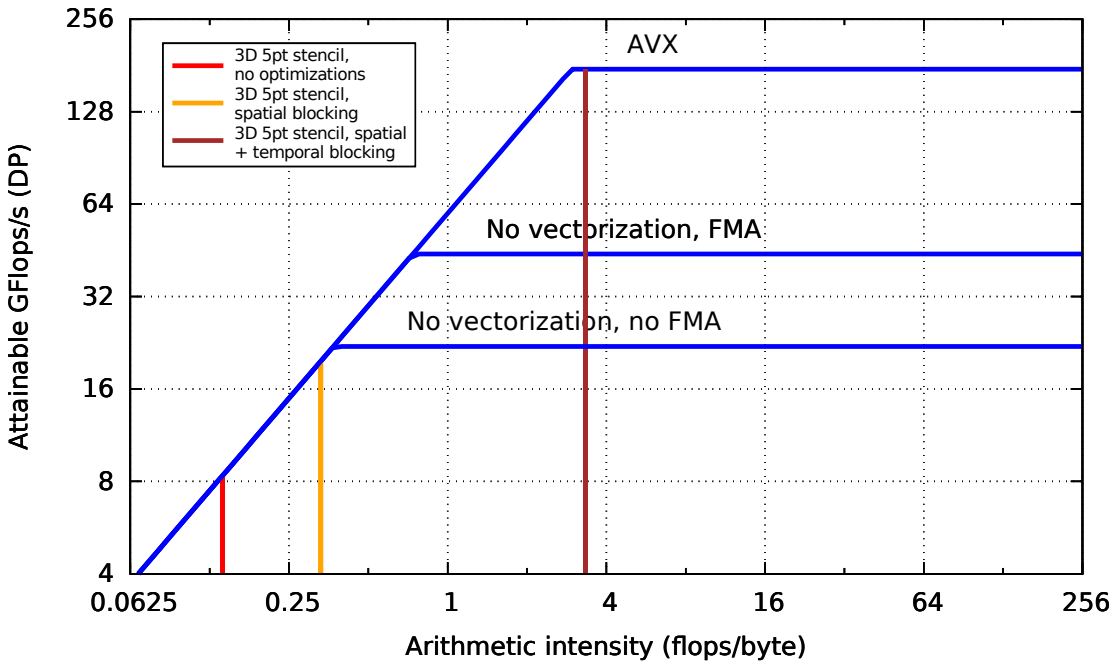
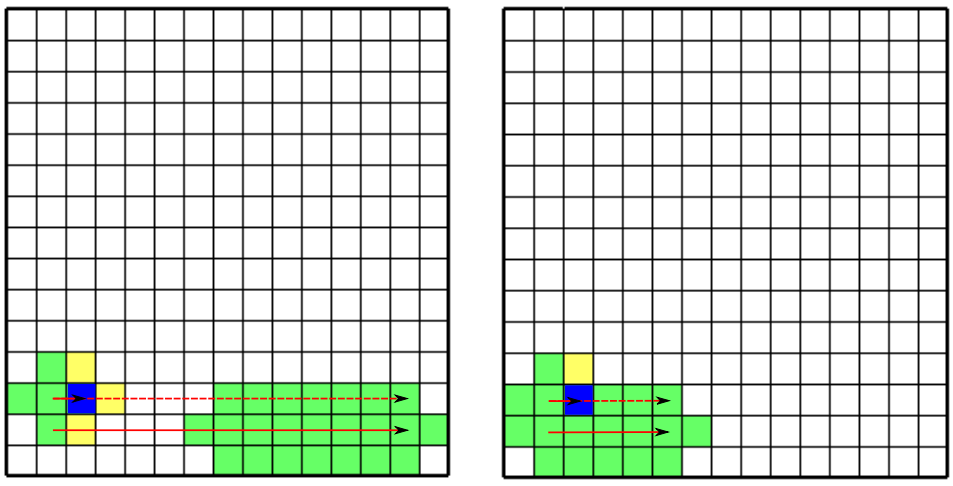


Figure 3.2: Roofline model of Intel Xeon E5-2660 v2 Processor.



(a) Without spatial blocking.

(b) Spatial blocking.

Figure 3.3: Example of spatial blocking. Blue - item to update, green - item already in cache, yellow - item needs to be loaded to cache.

Chapter 4

Stencil code generators

There are several ways how to generate a stencil code. The easiest is basic implementation that is very simple but its performance is far from optimal as we saw in section 3.1 and 3.4. In the listing 4.1 we can see the basic implementation of 3D Laplacian stencil.

In this chapter we introduce two state of the art optimizers and code generators, Pluto and PATUS, that can be used to generate better stencil code. They use spatial blocking and temporal blocking strategies (section 3.4) to generate optimized code. On the example of 3D Laplacian stencil code we show how to use these tools. Basic implementation of this stencil is in the listing 4.1.

```
for (t = 0; t < T; t++) {
    for (i = 1; i < N+1; i++) {
        for (j = 1; j < N+1; j++) {
            for (k = 1; k < N+1; k++) {
                A[(t+1)%2][i][j][k] =
                    0.125 * (A[t%2][i+1][j][k] + A[t%2][i-1][j][k]
                            + A[t%2][i][j+1][k] + A[t%2][i][j-1][k]
                            + A[t%2][i][j][k-1] + A[t%2][i][j][k+1])
                    + 0.25 * A[t%2][i][j][k];
            }
        }
    }
}
```

Listing 4.1: Basic implementation of 3D Laplacian stencil code.

4.1 Pluto

Pluto is a general purpose optimization and paralelization tool. It takes C source code as an input and produces optimized code. It is able to apply spatial and temporal blocking strategies as well as vectorization and paralelization using OpenMP [10].

In the source code we define regions that we want optimize using `#pragma scop` and `#pragma endscop`. Pluto takes these regions and replaces them with the optimized and paralelized code.

To generate optimized stencil code we can take the basic implementation (listing 4.1), enclose the code by `#pragma scop` and `#pragma endscop` and let Pluto generate the code.

However Pluto doesn't support all of the C constructs and has some constraints on using control variables in loops. Because of that it can be quite tricky to use Pluto compiler.

In our case there is problem with modulo operator in the index of array to select which array is used as an input and which one as an output. To avoid using modulo in index of array, we used if statement instead. Then the code can be optimized by Pluto without any problems. The final code ready to optimize by Pluto is in the listing 4.2.

```
#pragma scop
for (t = 0; t < T; t++) {
    for (i = 1; i < N+1; i++) {
        for (j = 1; j < N+1; j++) {
            for (k = 1; k < N+1; k++) {
                if ((t+1)%2) {
                    A[1][i][j][k] =
                        0.125 * (A[0][i+1][j][k] + A[0][i-1][j][k]
                                + A[0][i][j+1][k] + A[0][i][j-1][k]
                                + A[0][i][j][k-1] + A[0][i][j][k+1])
                         + 0.25 * A[0][i][j][k];
                } else {
                    A[0][i][j][k] =
                        0.125 * (A[1][i+1][j][k] + A[1][i-1][j][k]
                                + A[1][i][j+1][k] + A[1][i][j-1][k]
                                + A[1][i][j][k-1] + A[1][i][j][k+1])
                         + 0.25 * A[1][i][j][k];
                }
            }
        }
    }
}
#pragma endscop
```

Listing 4.2: Stencil code implementation ready to optimize by Pluto compiler.

4.2 PATUS

PATUS (Parallel Auto-Tuned Stencils [2]) is a software framework which generates optimized stencil codes for different hardware architectures. Stencil is specified by domain specific language and on the basis of this specification a stencil code for target architecture is generated. The advantage of using the domain specific language is that code for different target architectures can be generated based on the same stencil specification.

As a specialized tool for stencil computations, PATUS generates very good code. It implements spatial blocking optimization and uses auto-tuner to get tile size that maximizes the performance on the target computer. However PATUS doesn't support temporal blocking optimization, so as we can see in the section 5.3, for more iterations Pluto achieves better performance than PATUS.

The generated code together with the auto-tuned parameters can be used in any application. We can see example of PATUS stencil specification in the listing 4.3.

```
stencil kernel (
      double grid A(0 .. N+1, 0 .. N+1, 0 .. N+1),
      double grid B(0 .. N+1, 0 .. N+1, 0 .. N+1)
      )
{
   iterate while t < 1;
   domainsize = (1 .. N, 1 .. N, 1 .. N);

   operation
   {
      A[x, y, z] = 0.125 * (B[x+1, y, z] + B[x-1, y, z]
                          + B[x, y+1, z] + B[x, y-1, z]
                          + B[x, y, z-1] + B[x, y, z+1])
                 + 0.25 * B[x, y, z];
   }
}
```

Listing 4.3: Stencil code specification for PATUS code generator.

4.3 Combination of Pluto and PATUS

As we can see in the section 5.2, PATUS achieves very good performance on one iteration. Because PATUS supports only spatial blocking and not temporal blocking, Pluto overperforms PATUS on more than one iteration. We would like to improve performance on more than one iteration by extending PATUS generated code of temporal blocking approach. To do it we decided to generate by Pluto and PATUS code for the same stencil operation and then manually combine them.

PATUS generates a function that is called from OpenMP parallel region. The function detects ID of the thread and number of threads and performs one iteration on the selected area of the grid. Programmer cannot change this behavior and it cannot be combined with the paralelization done by Pluto. So we let Pluto to do only tiling and not the parallelization.

First we generated code by PATUS and let auto-tuner to find the best tiling configuration. Then we let Pluto to do tiling with the tile size found by auto-tuner.

The output of Pluto contains many nested for-loops with boundary conditions not readable for human. But after some experiments like printing the boundary conditions of the loops and watching the traversal on the grid in debugger, the main idea is clear. The outer loops take care about selecting the tile position on the grid and temporal blocking and the two (2D) of three (3D) innermost loops take care about iterating within one tile.

Finally, we replaced the innermost loops by the function generated by PATUS. From the range on which every loop iterates we determined the position of the tile and the tile size and passed these information to the function. In this way the Pluto code takes care about spatial and temporal blocking and PATUS takes care about the computation within the tile.

Performance achived by the combination of these two optimizers we can see in the section 5.3.

Chapter 5

Experiments

We make all our experiments on Emmy cluster at Regionalen Rechenzentrums Erlangen. This cluster is equipped by 10 cores Intel Xeon E5-2670v2 Ivy Bridge processors [11]. All the benchmark applications we use are compiled by the newest Intel compiler, version 15.0.2.

In our first experiment we show the effect of vectorization on the performance. In the next experiment we measure performance and memory bandwidth when using different number of cores. In the third experiment we show the importance of spatial blocking in stencil computations. We perform these experiments with these four stencil codes:

- 2D Laplacian on grid size 4002^2
- 3D Laplacian on grid size 150^3
- 3D Laplacian on grid size 502^3
- 3D Long-Range on grid size 150^3

In the last experiment we measure the effect of different optimizations of stencil code on the time of solution in Geometric Multigrid.

For the first three experiments we used *heat* example that is included to Pluto compiler [10]. Based on this example various stencil codes with different optimizations were implemented. The multigrid implementation for fourth experiment we got from [12] and implemented different smoothing step optimizations.

To perform the measurements as precise as possible we use likwid-pin tool that allows us to pin threads to physical cores [13]. The pinning ensures that during the experiment all the threads remain on the selected physical cores. This makes the measurements more precise and it also ensures the same performance when we rerun the measurements. We pin all threads to cores on one socket, so the application cannot take an advantage of running on two sockets, as it would do otherwise.

To measure memory bandwidth we used likwid-perfctr tool [13]. This tool measures statistics of memory usage within defined regions in the source code.

5.1 Impact of vectorization

In the first experiment we measur the the effect of vectorization on the floating point performance of different stencil codes. We used the basic implementation and the code optimized by Pluto, both compiled with and without vectorization. We measure the performance on

1 and 10 cores of 10-cores Intel Ivy Bridge processor. In this experiment only one iteration was performed, so the temporal blocking in Pluto doesn't take an effect.

In the figure 5.1 we present the results on 1 core and in the figure 5.2 on 10 cores. Tables A.1 and A.2 (in the Appendix A) contain the measured values. On 1 core we can see that the performance without vectorization is about the same for all Laplacian stencil codes. With vectorization the performance is about 1.5-2 times better. But on 10 cores there is only minimal improvement with vectorization, in the case of icc version no improvement at all. We can also see that the performance is only 5 times better compared to the measurement on 1 core. Using 10 cores, the memory bandwidth is saturated, so the vectorization or adding more cores cannot improve the performance.

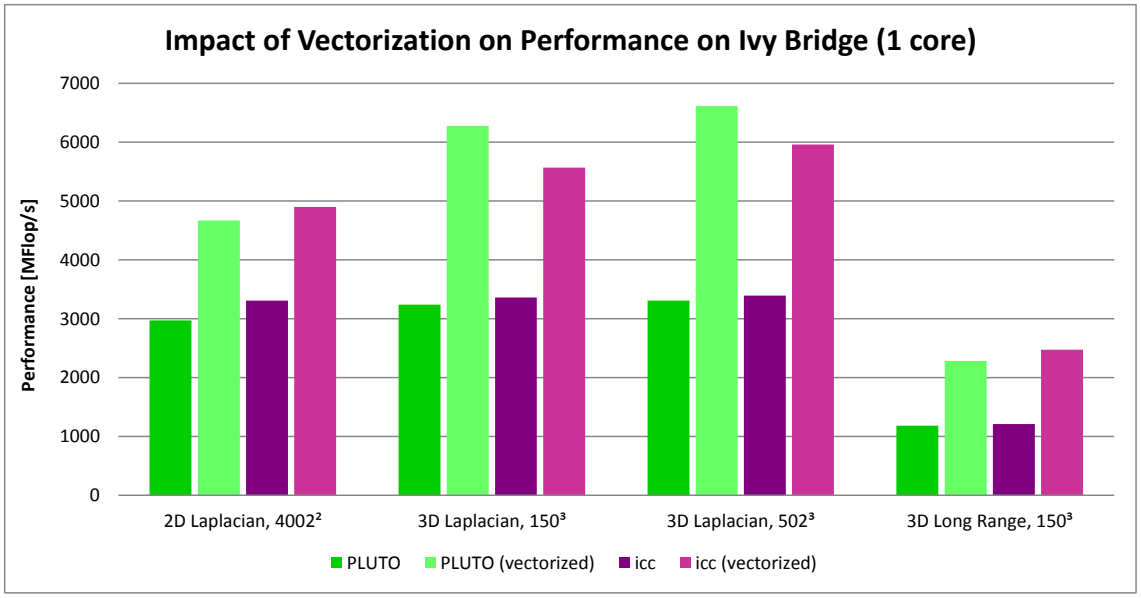


Figure 5.1: Impact of Vectorization on Performance on 1 core of Ivy Bridge processor.

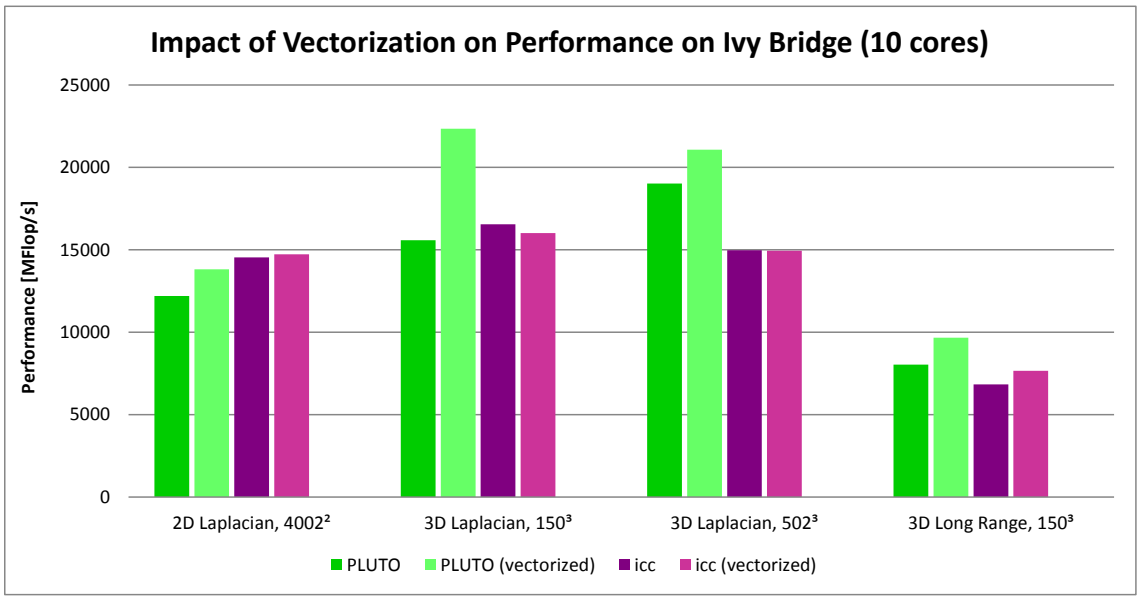


Figure 5.2: Impact of Vectorization on Performance on 10 cores of Ivy Bridge processor.

5.2 Spatial blocking performance

This experiment is performed by running 1 to 10 threads at an Intel Ivy Bridge 10-cores socket, with different kernel operations and different grid sizes. Auto-tuning in PATUS is performed at the 10-threads experiments only. For each of the thread scaling results we report the performance and the memory bandwidth usage of the corresponding threading experiment.

Figures 5.3 to 5.5 show performance and memory bandwidth for different kernels. The measured values we can find in the table B.1 (Appendix B). In both 2D and 3D we can see tha PATUS performs very well up to 5 threads. Then the memory bandwidth saturates and the performance does not increase anymore.

Performance and memory bandwidth of PLUTO with vectorization is in 3D more or less the same as in case of PATUS. In 2D PATUS overperforms Pluto for lower numbers of threads, but for 7 and threads and more Pluto achieves the same performance. Non-vectorized version of Pluto has obviously worse performance, but eventually it gets to the same performance as the other versions because of the memory bottleneck.

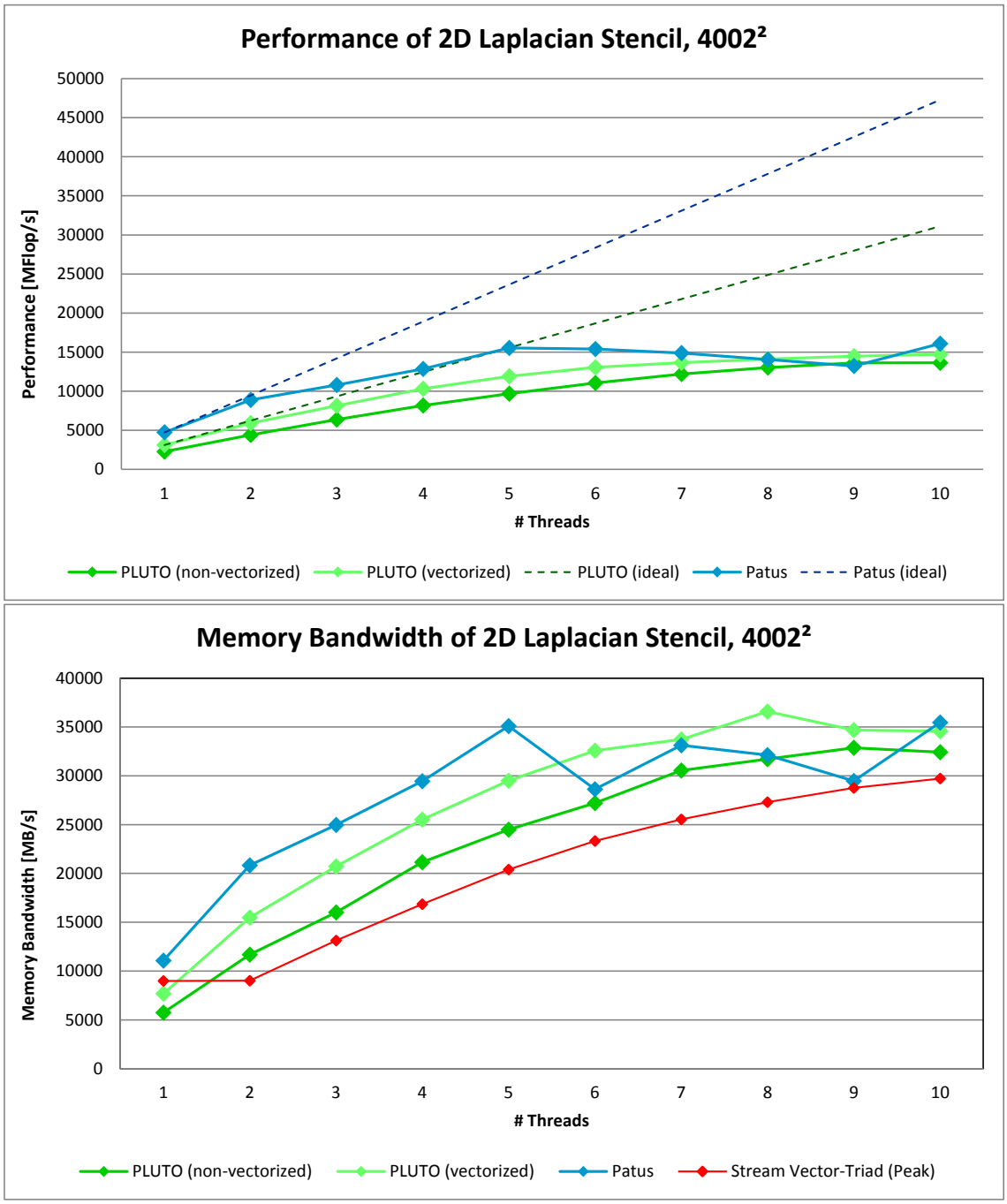


Figure 5.3: Performance and Memory bandwidth of 2D Laplacian stencil (grid size=4002^2) at 1 iteation on Intel Ivy Bridge processor.

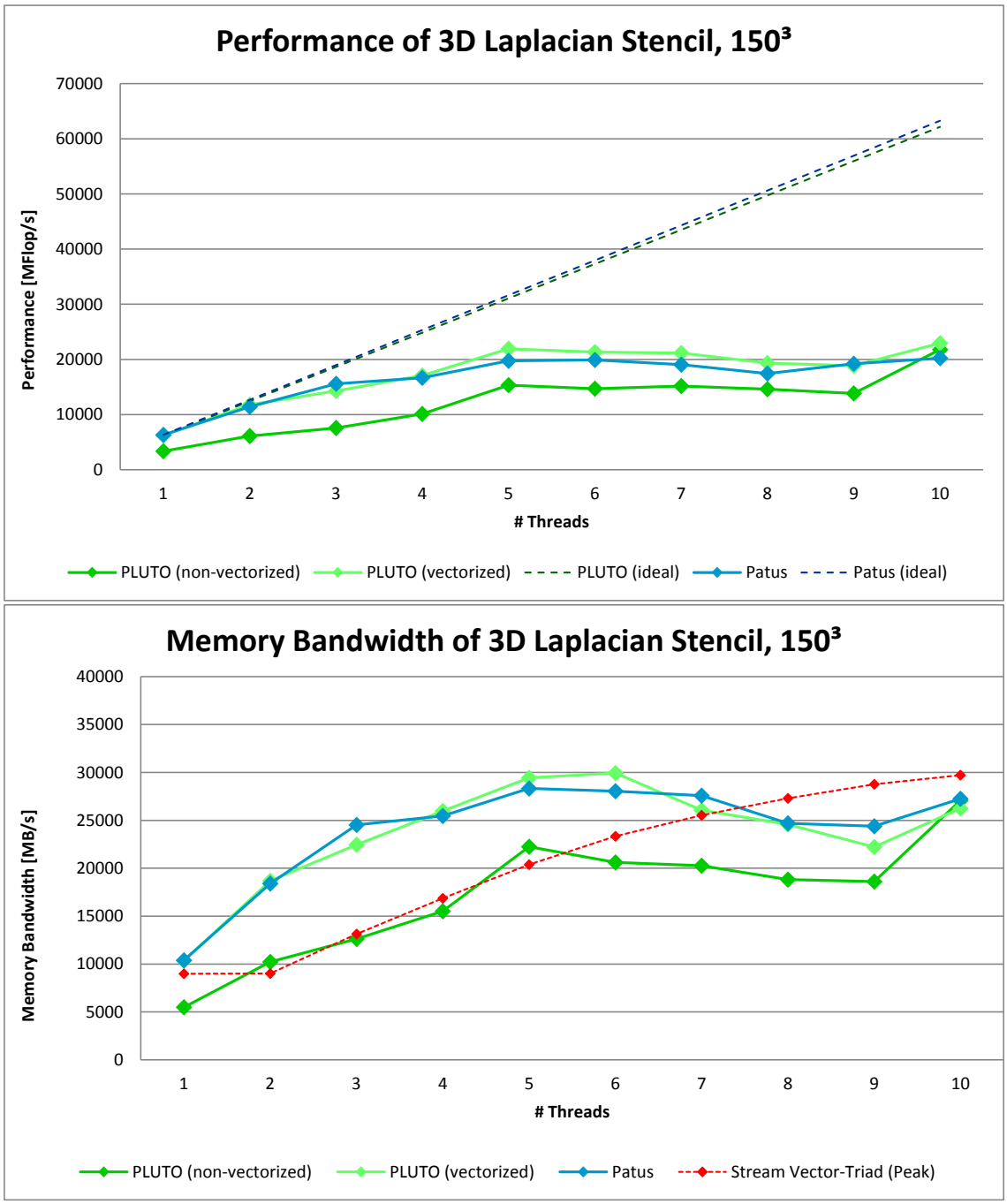


Figure 5.4: Performance and Memory bandwidth of 3D Laplacian stencil (grid size=150^3) at 1 iteation on Intel Ivy Bridge processor.

Figure 5.5: Performance and Memory bandwidth of 3D Laplacian stencil (grid size=502^3) at 1 iteation on Intel Ivy Bridge processor.

5.3 The importance of temporal blocking

As we saw in the roofline model (figure 3.2), to achieve maximum performance, it is necessary for the algorithm to have arithmetic intensity at least 3 flops/byte. Stencil computations have much lower arithmetic intensity, as we showed in the section 3.1. In section 3.4 we showed that using spatial blocking we can increase the arithmetic intensity approximately twice. However this is not enough to get the full performance, we need to add another optimization – temporal blocking.

In this experiment we performed 1 to 20 iterations on 1 and 10 cores of Intel Ivy Bridge socket, for the same stencil codes as at previous experiments. We measured the performance and memory bandwidth of the basic implementation without optimizations, optimized version by Pluto with temporal blocking and combination of Pluto and PATUS, where the temporal blocking is done by Pluto and iteration within one tile is done by PATUS. We can see results in the figures 5.6 to 5.13 or in Appendix C and D.

We can see that performance and bandwidth of the icc version (the basic implementation) with increasing number of iterations stays constant. Pluto uses temporal blocking optimization, so with the increasing number of iterations, the arithmetic intensity decreases. Because of that the memory bandwidth decreases and performance increases, as we can see in the results.

The combination of Pluto and PATUS uses the same optimizations as stand-alone Pluto, so we would expect similar behavior. On one core the behavior of Laplacian stencils is similar to Pluto, but the behavior of Longrange stencil is different. For 1 and 2 iterations it starts similar as Pluto, but then the memory bandwidth starts increasing. It seems that the tile size found by auto-tuner is too large for temporal blocking. For 3 and more iterations all of the elements don't fit in the cache, which causes more traffic between memory and cache.

Very strange behavior we can see at the combination of Pluto and PATUS on 10 cores. Even though 10 cores were used, the performance and bandwidth is the same as on 1 core. It seems that only one thread is doing all the work and the other threads are not doing anything. This is probably caused by the way how we combined Pluto and PATUS. PATUS code is called to compute every tile one by one, but in the results we can see that this is not optimal. It is necessary to find a way how to call PATUS code for several tiles in parallel. This will be matter of our future work.

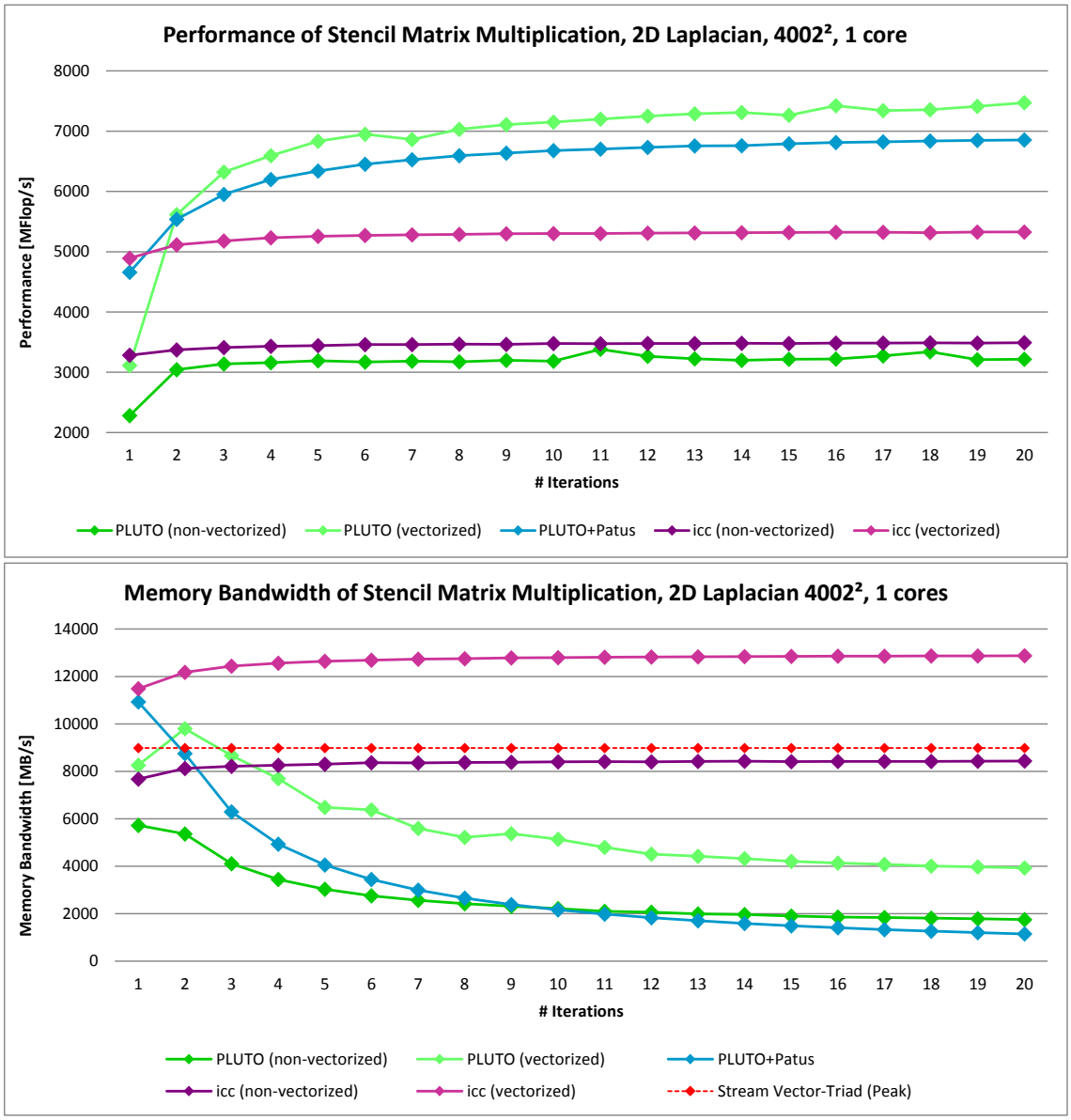


Figure 5.6: Performance and Memory bandwidth of 2D Laplacian stencil (grid size=4002^2) on 1 core of Intel Ivy Bridge processor.

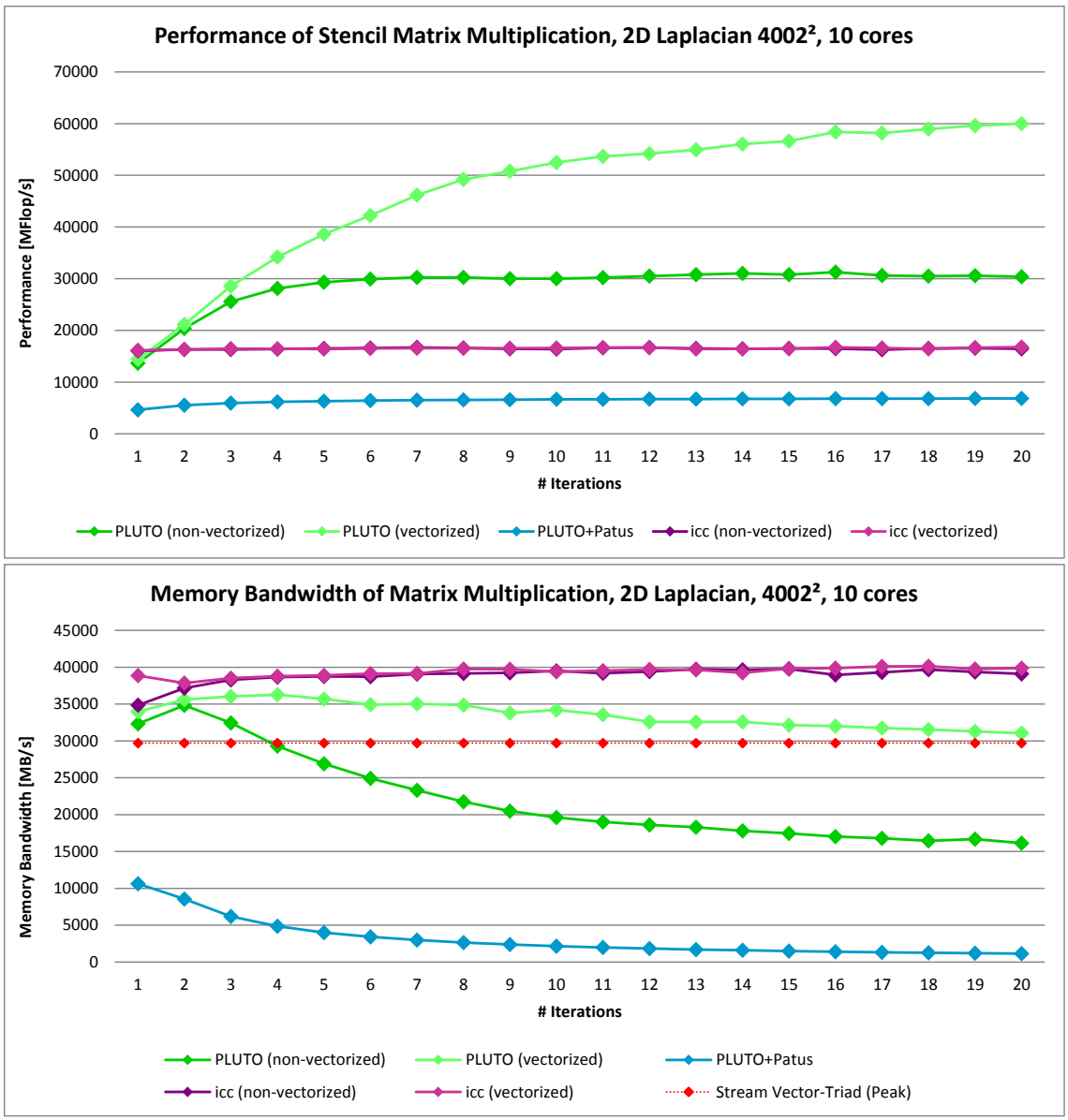


Figure 5.7: Performance and Memory bandwidth of 2D Laplacian stencil (grid size=4002^2) on 10 cores of Intel Ivy Bridge processor.

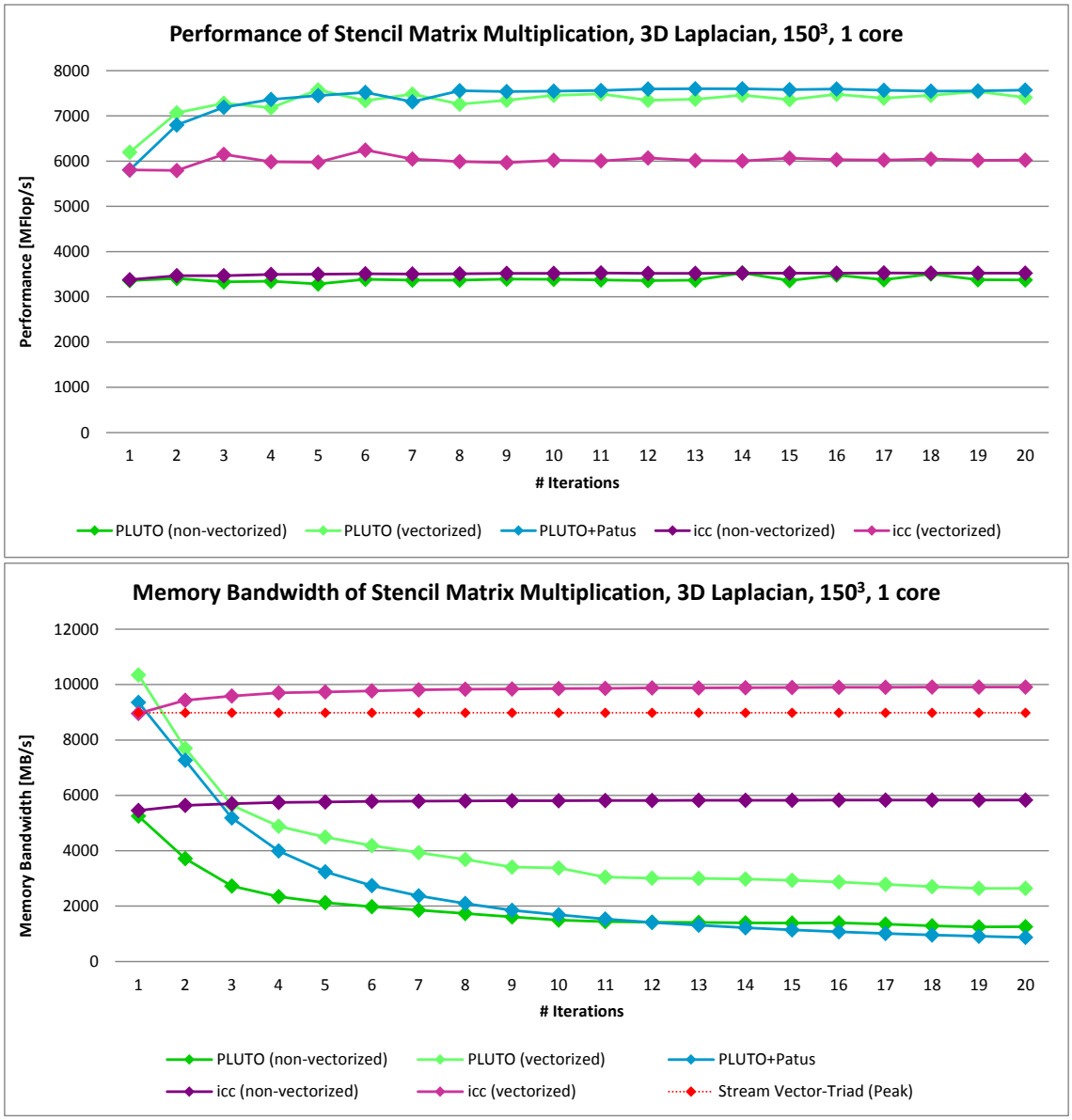


Figure 5.8: Performance and Memory bandwidth of 3D Laplacian stencil (grid size=150^2) on 1 core of Intel Ivy Bridge processor.

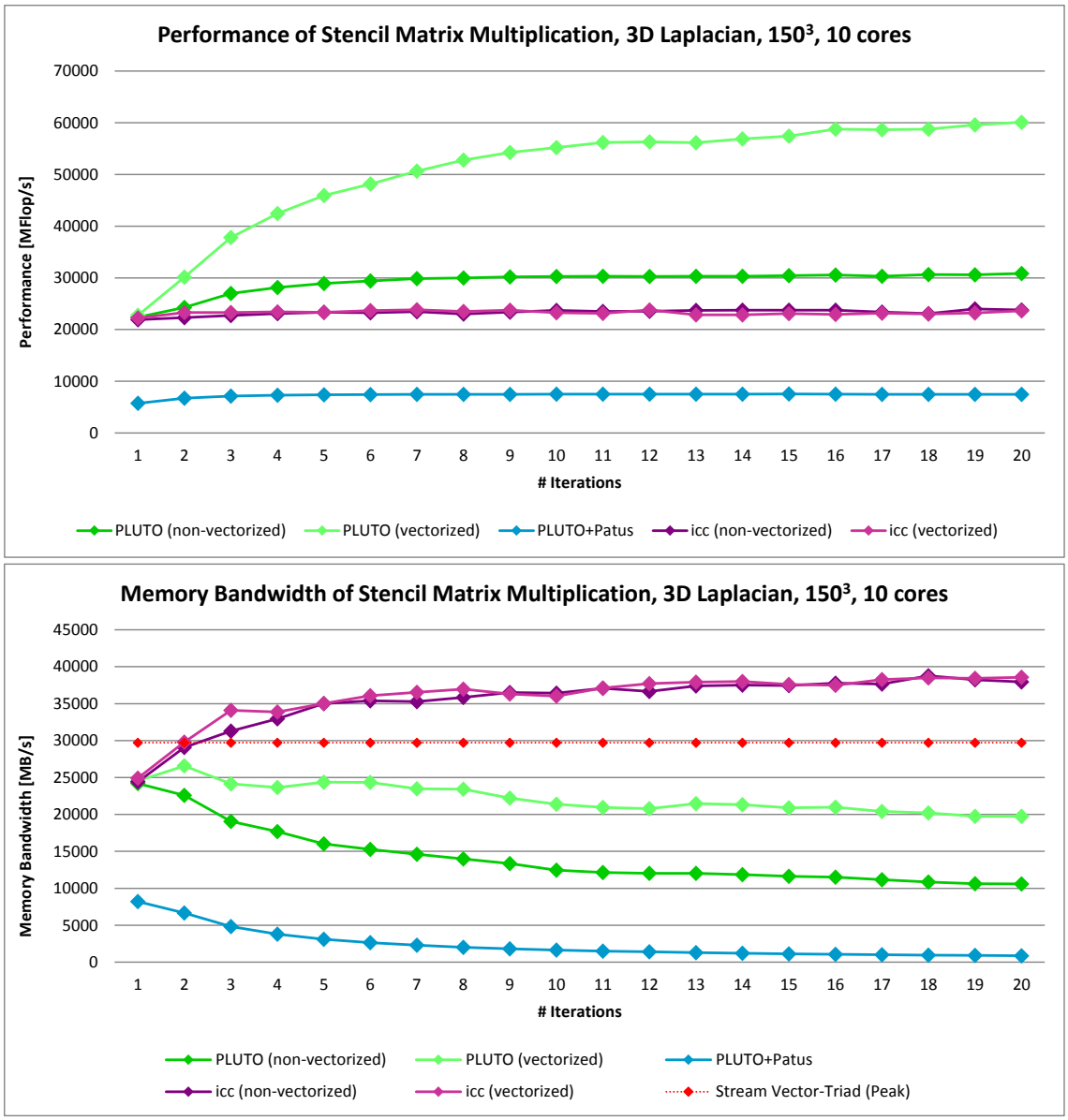


Figure 5.9: Performance and Memory bandwidth of 3D Laplacian stencil (grid size=150^2) on 10 cores of Intel Ivy Bridge processor.

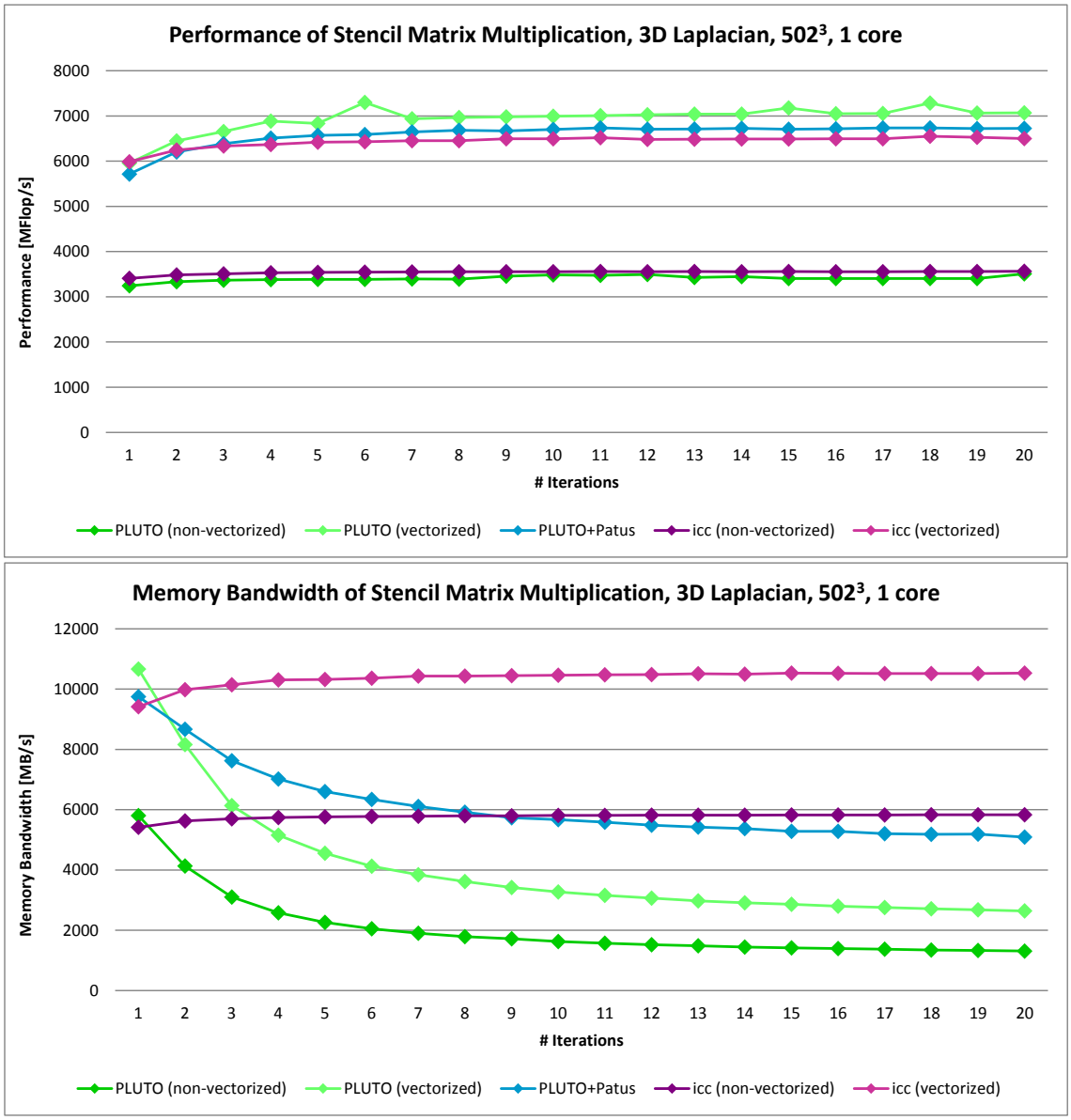


Figure 5.10: Performance and Memory bandwidth of 3D Laplacian stencil (grid size=502^2) on 1 core of Intel Ivy Bridge processor.

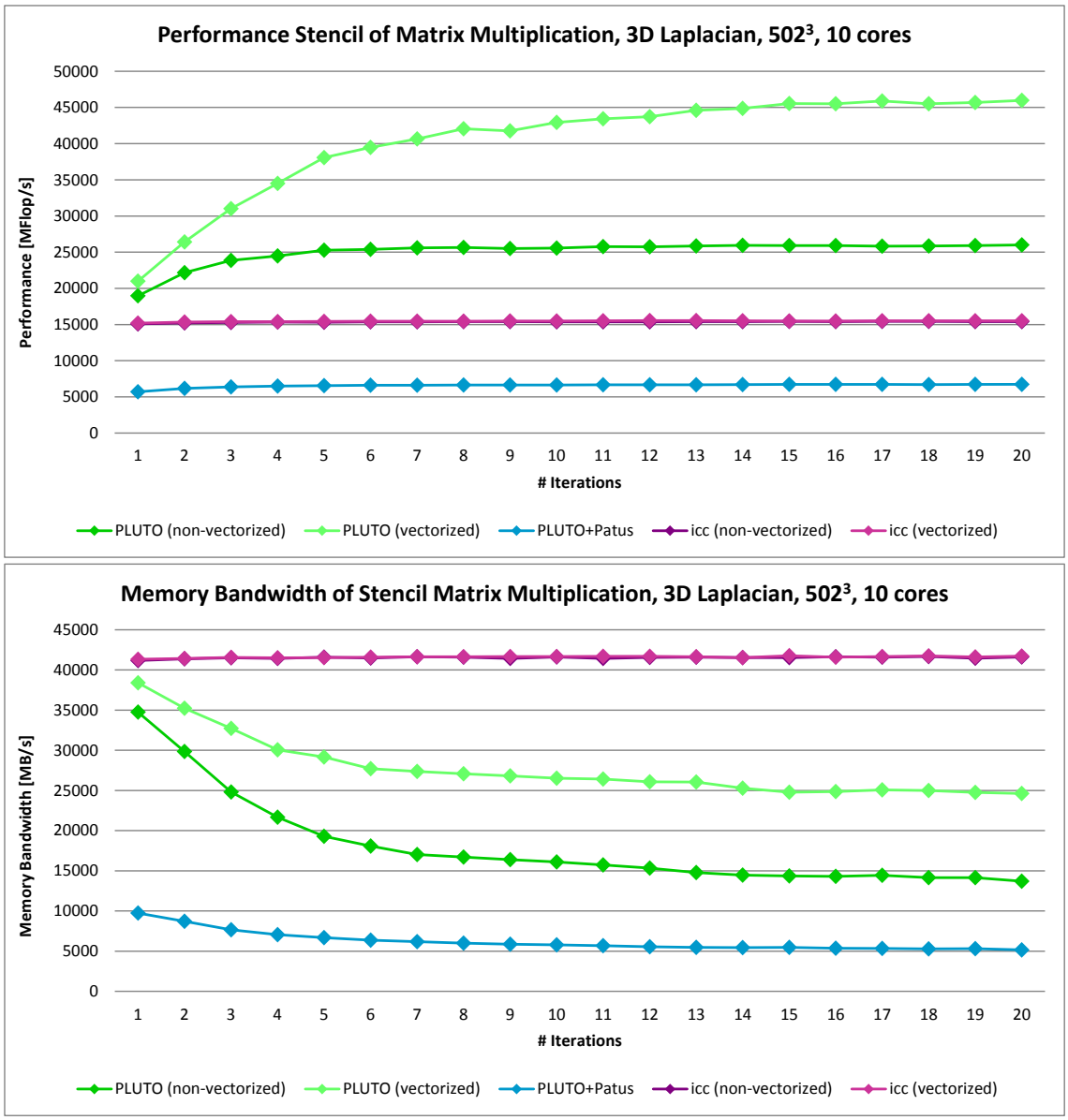
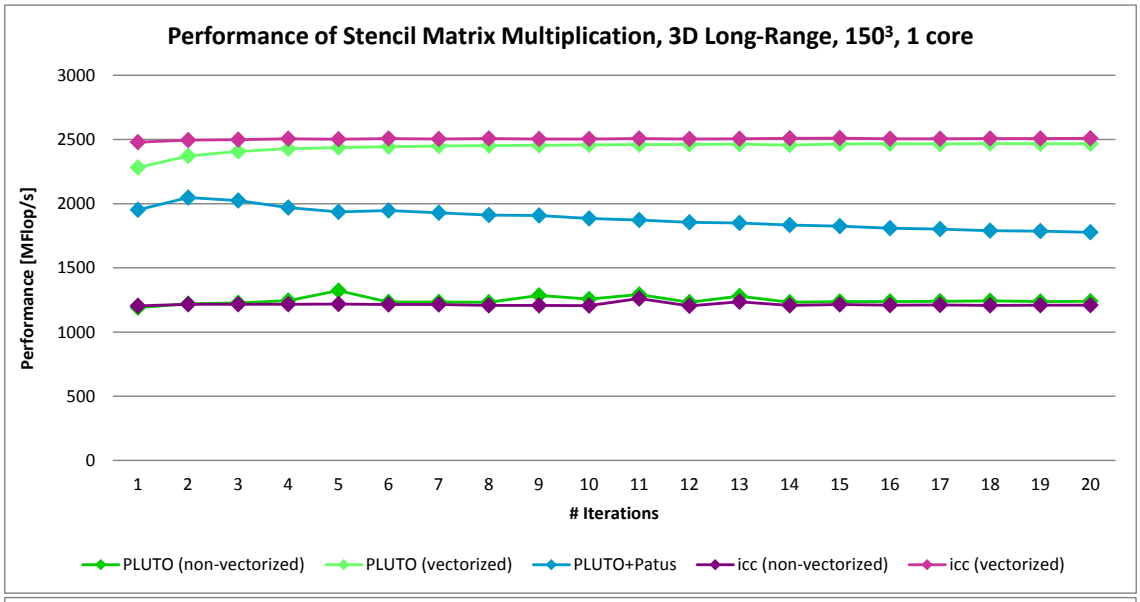


Figure 5.11: Performance and Memory bandwidth of 3D Laplacian stencil (grid size=502^2) on 10 cores of Intel Ivy Bridge processor.



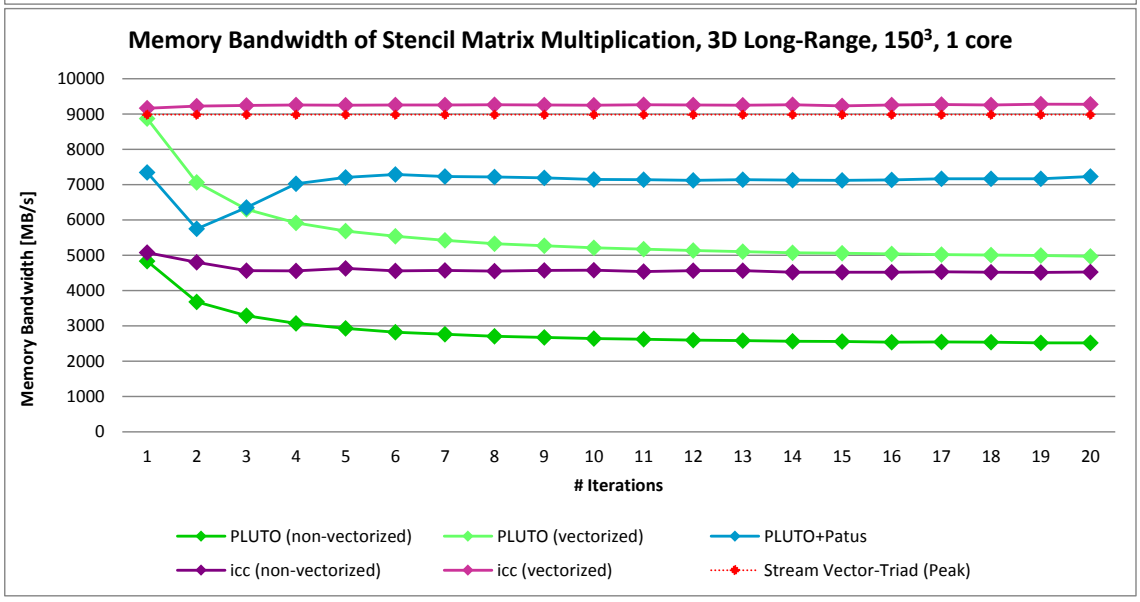


Figure 5.12: Performance and Memory bandwidth of 3D Longrange stencil (grid size=150^2) on 1 core of Intel Ivy Bridge processor.

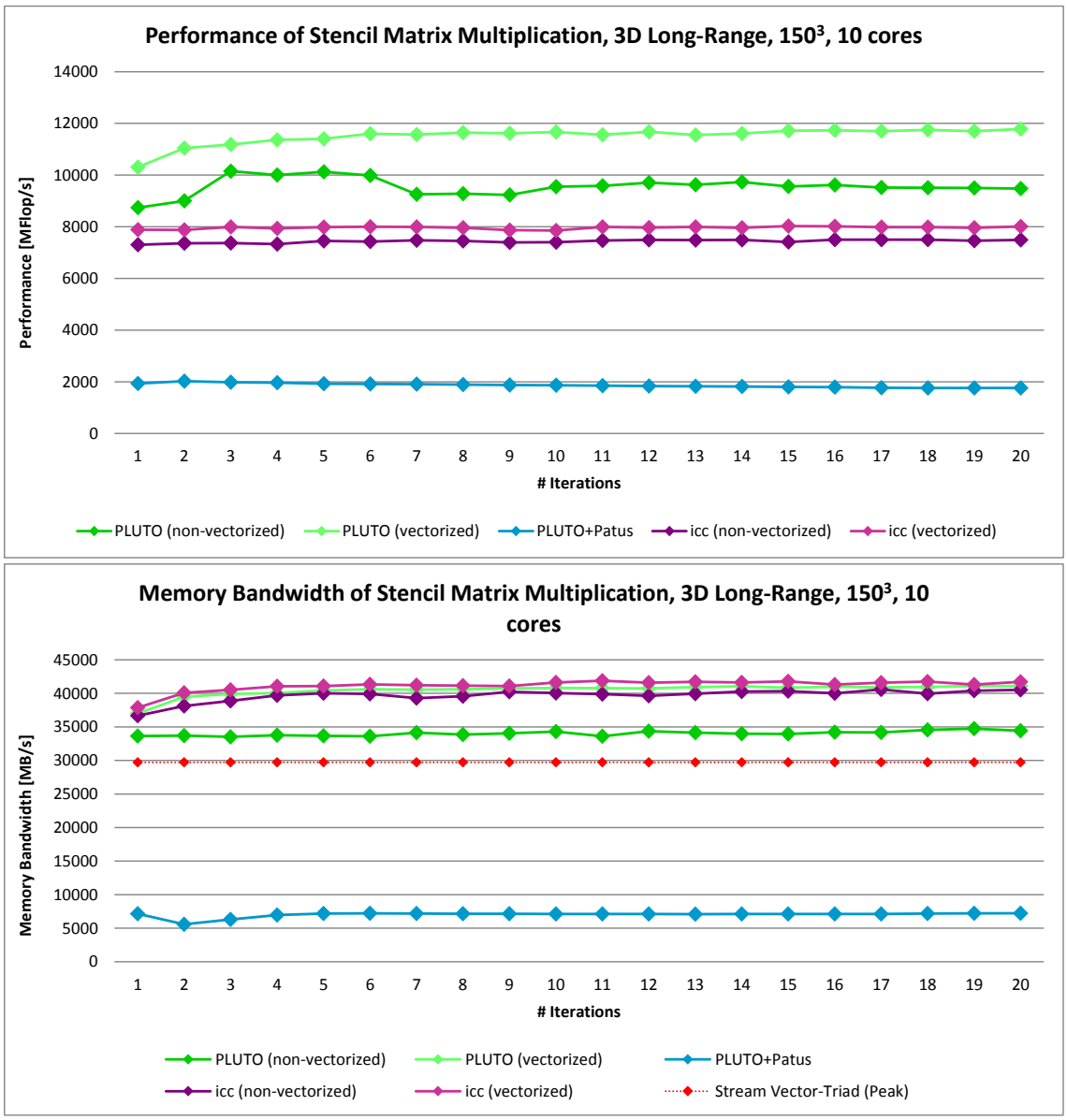


Figure 5.13: Performance and Memory bandwidth of 3D Longrange stencil (grid size=150^2) on 10 cores of Intel Ivy Bridge processor.

5.4 Geometric Multigrid

In the last experiment we investigate how different stencil optimizations influence the time of solution in Geometric Multigrid. We have the same multigrid code with different implementations of the smoothing step – basic implementation, implementation optimized by Pluto with both spatial and temporal blocking and code generated by PATUS. The autotuning in PATUS is performed for three different grid sizes (31^2, 1023^2 and 4095^2) on 10 cores.

In the figures 5.14 and 5.15 or in the tables 5.1 and 5.2 we can see the times of solution on the grid size = 16383^2 on 1 and 10 cores of Intel Ivy Bridge processor. Second column of the tables show how many V-cycles are needed to achieve required precision 10^{-12}. As we can expect, for more smoothing steps, less V-cycles are needed. We can also see that best performing PATUS tuning is for grid size=1023^2.

Next thing we can see is that the best number of smoothing steps for basic implementation (mgm) and PATUS is 2 on both one and ten cores, but for Pluto it is 4 on one core and 6 or 8 on ten cores. As we saw in the previous experiment (section 5.3), thanks to temporal blocking Pluto achieves much better performance when more iterations are performed. So it is preferable for Pluto to perform less V-cycles with more smoothing steps compared to PATUS.

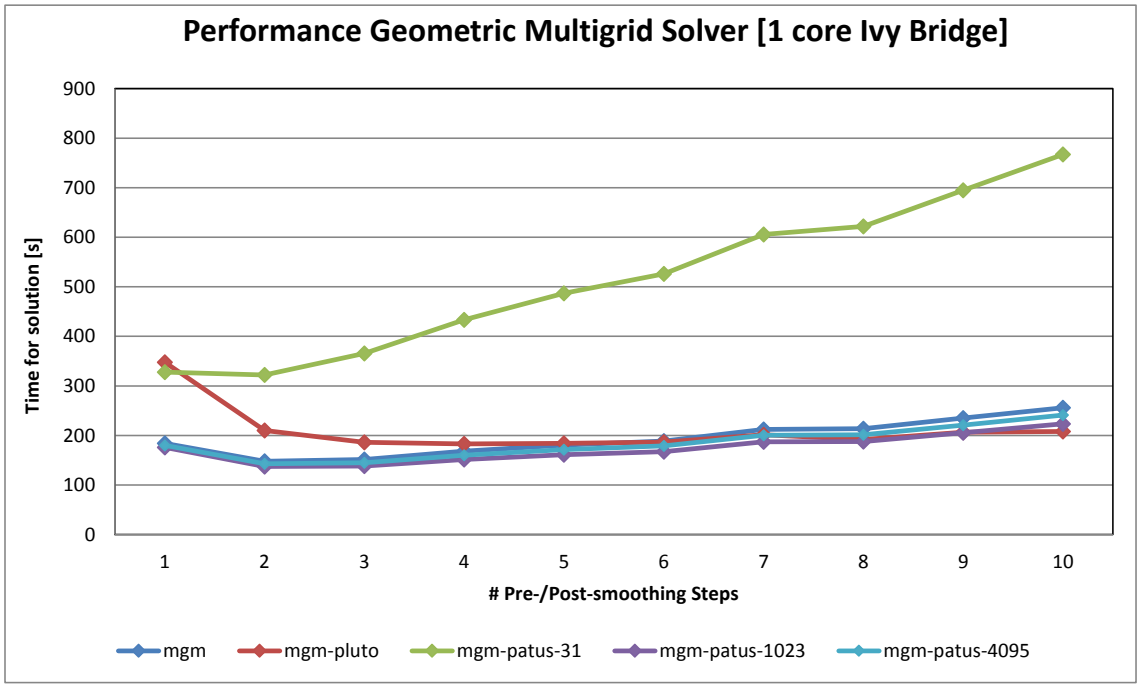


Figure 5.14: Times of solution of Geometric Multigrid for different numbers of Pre- and Post-smoothing steps and different optimizations of the smoothing step stencil code. Measured on 1 core of Intel Ivy Bridge processor on grid size=16383^2.

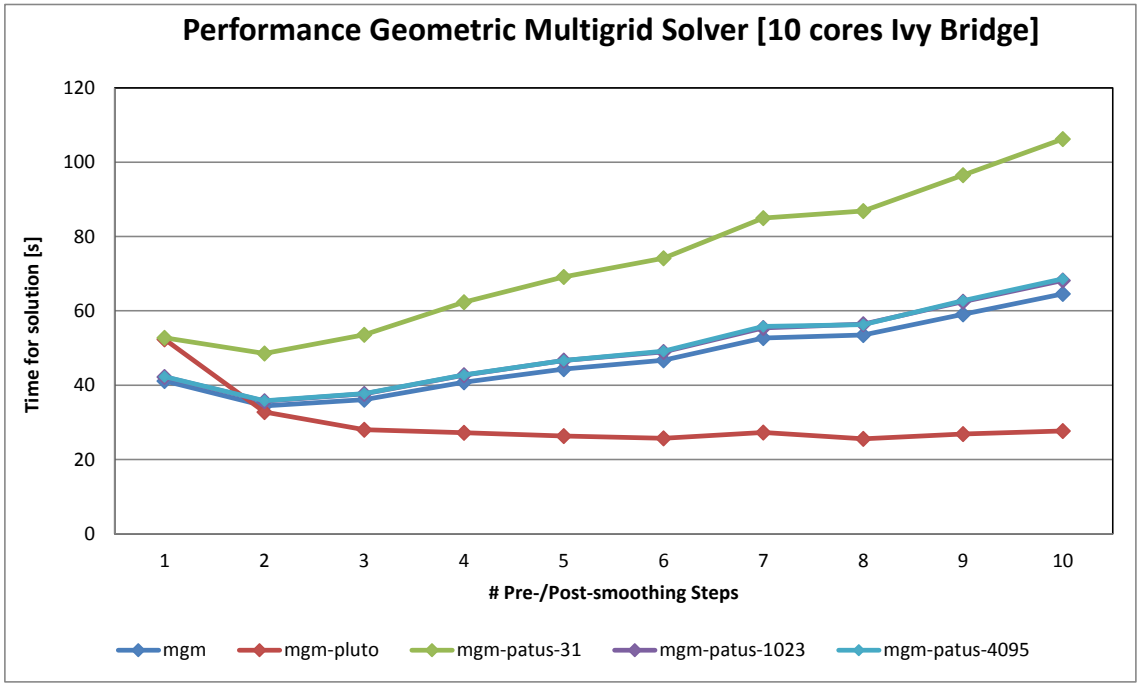


Figure 5.15: Times of solution of Geometric Multigrid for different numbers of Pre- and Post-smoothing steps and different optimizations of the smoothing step stencil code. Measured on 10 cores of Intel Ivy Bridge processor on grid size=16383^2.

Pre- / Post-smoothing steps	V-cycles	icc-15.02	Pluto	PATUS (31)	PATUS (1023)	PATUS (4095)
1	28	184.0	347.8	328.0	176.1	179.8
2	17	147.9	210.1	322.0	137.3	142.9
3	14	151.5	185.9	365.7	138.0	145.0
4	13	168.0	182.9	433.4	151.6	160.3
5	12	180.3	184.0	486.8	161.2	171.3
6	11	188.8	186.7	526.0	167.2	178.9
7	11	212.1	201.3	605.5	187.0	200.2
8	10	213.7	191.5	621.8	187.8	201.7
9	10	234.9	206.5	694.9	205.7	220.8
10	10	255.8	208.1	767.0	223.5	241.1

Table 5.1: The time of solution (in s) in Geometric Multigrid for different optimizations of smoothing step and different numbers of Pre-/Post-smoothing steps on one core of an Intel Ivy Bridge processor.

Pre- / Post-smoothing steps	V-cycles	icc-15.02	Pluto	PATUS (31)	PATUS (1023)	PATUS (4095)
1	28	41.1	52.4	52.8	42.3	42.3
2	17	34.4	32.8	48.5	35.7	35.8
3	14	36.1	28.0	53.6	37.7	37.7
4	13	40.8	27.2	62.4	42.7	42.8
5	12	44.3	26.3	69.1	46.6	46.6
6	11	46.7	25.7	74.2	48.9	49.1
7	11	52.7	27.3	85.0	55.4	55.8
8	10	53.5	25.6	86.9	56.5	56.3
9	10	59.1	26.9	96.5	62.5	62.7
10	10	64.6	27.7	106.2	68.2	68.6

Table 5.2: The time of solution (in s) in Geometric Multigrid for different optimizations of smoothing step and different numbers of Pre-/Post-smoothing steps on ten cores of an Intel Ivy Bridge processor.

Chapter 6

Conclusion

In this work we studied the principles of performance engineering of stencil codes. We studied spatial blocking and temporal blocking strategies to exploit cache locality. We got familiar with two state of the art stencil compilers, Pluto and PATUS, which implement these methods and allow generating optimized stencil code.

Then we implemented various stencil codes with different optimizations. We measured floating point performance and memory bandwidth to see the influence of different optimizations. We performed all the measurements on RRZE Emmy Cluster, equiped by Intel Ivy Bridge processors.

We also tried to combine both optimizers while generating a stencil code to use advantages of both. PATUS as a specialized stencil generator tool produces very good code, but it doesn't support temporal blocking, so it cannot benefit from performing several iterations. Pluto is a general purpose optimizer. It takes advantage over PATUS when more iterations are performed because it implements temporal blocking.

We managed to combine these two tools and created working code. This code however performs well only on one core. On more cores it doesn't split the work between threads well. Solving these problems is a matter of future work.

Finally, we studied Geometric Multigrid, a widely used method to solve partial differential equations. We implemented smoothing step of multigrid with different optimizations and measured a time of solution with these optimizations.

In chapter 2 of this work we introduce Geometric Multigrid. There is described how multigrid works, how using several grids help make the solution converge quickly and there is also description of all parts of multigrid.

The next chapter focuses on performance engineering of stencil code. There is explained how to analyze memory usage of an algorithm and why performance of stencil code is limited by memory throughput. There are also discussed different types of architectures used in recent years to show the trend of using various accelerators, which require higher arithmetic intensity than classical CPUs to utilize the performance.

Chapter 4 describes the optimizers Pluto and PATUS and the way how we combined them. And in the last chapter there are described the the measurements we performed and the results are discussed.

As is shown in the section 3.3, in the last years a lot of new supercomputers are equiped by accelerators. Usually are used Nvidia CUDA devices and Intel Xeon Phi. We can expect that these devices will be used also in the future, so it is good to be prepared for that. The work started in this thesis could continue by investigating stencil code optimizations for these accelerators. All of the devices are suitable for tasks that can be computed in

parallel on high number of cores. In the stencil codes, all of the points can be updated independently, so it is easy to paralelize. But the accelerators require higher arithmetic intensity than regular processors to work effectively. To increase arithmetic intensity, the same approach as is described in this work can be applied.

Modification of program to run on Xeon Phi is not difficult, so there shouldn't be any problem to try the same experiments as are described in this work. Modifications to run program on CUDA are more difficult, but there shouldn't be any serious problem as well. PATUS is able to generate code for CUDA architecture, but with Pluto some problems may appear. Parallelization on CUDA is done different way form CPU, but tiling done by Pluto should work on CUDA as it works on CPU.

Bibliography

[1] William L. Briggs, Van Emden Henson, and Steve F. McCormick. *A Multigrid Tutorial (2Nd Ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[2] Matthias Christen. *Generating and Auto-Tuning Parallel Stencil Codes*. PhD thesis, University of Basel, Switzerland, 2011.

[3] Stencil code. `http://en.wikipedia.org/wiki/Stencil_code`, 2014-07-25 [cit. 2014-01-14].

[4] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.

[5] Top 500 [online]. `http://www.top500.org`, 2015-05-25 [cit. 2015-05-25].

[6] Richard Walsh, C. Earl Joseph, Steve Conway, and Jie Wu. White paper with its new powerxcell 8i product line, ibm intends to take accelerated processing into the hpc. August 2008.

[7] Tesla c2050 board specification. `http://www.nvidia.com/docs/IO/43395/Tesla_C2050_Board_Specification.pdf`, 2010-05-07 [cit. 2015-05-25].

[8] Tesla k80 gpu accelerator board specification. `http://images.nvidia.com/content/pdf/kepler/Tesla-K80-BoardSpec-07317-001-v05.pdf`, 2015-01-30 [cit. 2015-05-25].

[9] Intel xeon phi coprocessor peak theoretical maximums [online]. `http://www.intel.com/content/www/us/en/benchmarks/server/xeon-phi/xeon-phi-theoretical-maximums.html`, [cit. 2015-05-25].

[10] Pluto - an automatic parallelizer and locality optimizer for multicores [online]. `http://pluto-compiler.sourceforge.net/`, 2014-07-27 [cit. 2014-01-12].

[11] Emmy cluster [online]. `http://www.rrze.fau.de/dienste/arbeiten-rechnen/hpc/systeme/emmy-cluster.shtml`, 2014-12-10 [cit. 2015-05-21].

[12] P. Ghysels and W. Vanroose. Modeling the performance of geometric multigrid on many-core computer architectures. *SIAM Journal on Scientific Computing*, 37(2):194–216, 2015.

[13] J. Treibig, G. Hager, and G. Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*, San Diego CA, 2010.

Appendix A

Performance with and without vectorization on one and ten cores

Stencil	icc-15.02		PLUTO	
	no-vector	vector	no-vector, no temp.	vector, no temp.
2D Laplacian (4002^2)	3.3	4.9	3.0	4.7
3D Laplacian (150^3)	3.4	5.5	3.2	6.3
3D Laplacian (502^3)	3.4	6.0	3.3	6.6
3D Long-Range (150^3)	1.2	2.5	1.2	2.3

Table A.1: The impact of vectorization on the performance (in Gflops/s) for four different stencil operators on one core of Intel Ivy Bridge processor.

Stencil	icc-15.02		PLUTO	
	no-vector	vector	no-vector, no temp.	vector, no temp.
2D Laplacian (4002^2)	14.5	14.7	12.2	13.8
3D Laplacian (150^3)	16.6	16.0	15.6	22.3
3D Laplacian (502^3)	15.0	15.0	19.0	21.0
3D Long-Range (150^3)	6.8	7.7	8.0	9.7

Table A.2: The impact of vectorization on the performance (in Gflops/s) for four different stencil operators on ten cores of Intel Ivy Bridge processor.

Appendix B

Spatial blocking results on one core

Thread scaling of 5-point 2D stencil (grid size=4002^2) with constant coefficient on a 10-core Intel Ivy Bridge socket.

	PATUS (vector.)		PLUTO (non vector.)		PLUTO (vector.)	
	Perf.	mem.bandwidth	Perf.	mem.bandwidth	Perf.	mem.bandwidth
1	4.73	11.07	2.27	5.74	3.11	7.68
2	8.87	20.83	4.39	11.69	5.91	15.49
3	10.80	24.97	6.37	16.00	8.15	20.71
4	12.86	29.45	8.16	21.15	10.32	25.52
5	15.54	35.10	9.68	24.49	11.92	29.50
6	15.39	28.63	11.06	27.21	13.05	32.58
7	14.88	33.13	12.21	30.55	13.65	33.72
8	14.04	32.13	13.02	31.71	14.10	36.59
9	13.22	29.47	13.61	32.87	14.47	34.68
10	16.08	35.47	13.64	32.43	14.71	34.58

Thread scaling of 7-point 3D stencil (grid size=150^3) with constant coefficient on a 10-cores Intel Ivy Bridge socket.

	PATUS (vector.)		PLUTO (non vector.)		PLUTO (vector.)	
	Perf.	mem.bandwidth	Perf.	mem.bandwidth	Perf.	mem.bandwidth
1	6.33	10.39	3.36	5.47	6.21	10.35
2	11.42	18.41	6.12	10.21	11.76	18.67
3	15.58	24.54	7.58	12.62	14.29	22.45
4	16.66	25.47	10.13	15.52	17.07	25.97
5	19.73	28.33	15.34	22.24	21.93	29.44
6	19.91	28.04	14.67	20.61	21.30	29.94
7	19.04	27.58	15.16	20.26	21.15	26.07
8	17.44	24.70	14.59	18.81	19.32	24.58
9	19.22	24.39	13.82	18.62	18.86	22.21
10	20.22	27.24	21.78	27.07	22.94	26.24

Thread scaling of 7-point 3D stencil (grid size=502^3) with constant coefficient on a 10-cores Intel Ivy Bridge socket.

	PATUS (vector.)		PLUTO (non vector.)		PLUTO (vector.)	
	Perf.	mem.bandwidth	Perf.	mem.bandwidth	Perf.	mem.bandwidth
1	5.94	10.16	3.25	5.85	5.97	10.67
2	11.07	18.82	6.35	11.41	11.70	20.56
3	16.07	27.48	9.18	16.50	15.90	29.17
4	18.82	32.43	12.55	23.45	19.48	35.19
5	21.68	37.33	14.23	25.68	20.13	36.33
6	21.50	37.25	16.22	29.68	20.68	37.48
7	21.90	38.27	18.75	33.49	21.79	39.70
8	21.87	38.51	20.91	38.57	21.73	39.55
9	22.60	40.17	20.35	37.43	21.81	39.63
10	22.15	40.16	19.02	34.90	21.00	38.16

Thread scaling of 25-point long-range 3D stencil (grid size=150^3) with constant coefficient on a 10-cores Intel Ivy Bridge socket.

	PATUS (vector.)		PLUTO (non vector.)		PLUTO (vector.)	
	Perf.	mem.bandwidth	Perf.	mem.bandwidth	Perf.	mem.bandwidth
1	2.05	7.98	1.20	5.08	2.28	8.89
2	3.67	14.77	2.22	9.28	4.12	16.26
3	5.70	22.08	2.84	13.31	5.17	20.37
4	6.78	27.22	3.75	14.30	6.27	24.54
5	8.15	31.42	5.34	20.95	8.39	32.90
6	8.78	34.10	5.54	20.64	8.36	32.24
7	9.58	35.65	5.65	20.49	8.19	31.24
8	9.38	36.20	5.92	20.10	7.77	29.79
9	9.47	36.48	5.61	25.18	8.01	29.80
10	10.03	36.91	8.80	33.27	10.37	37.01

Table B.1: Spatial-tiling performance (in Gflops/s) and memory bandwidth usage (in Gbytes/s) for two state-of-the-art stencil compilers for various stencils on one node of Intel Ivy Bridge processor.

Appendix C

Temporal blocking results on one core

Spatial blocking Perf. (in Gflops/s) and mem.bandwidth usage (in Gbytes/s) for two state-of-the-art stencil compilers for various stencils on one core of Intel Ivy Bridge processor.

	Arithmetic Intensity	PLUTO (non-vec) Perf.	PLUTO (vec) Perf.	PLUTO-PATUS Perf.	icc-15.02, non-vec Perf.	icc-15.02, vec Perf.
1	0.25	2.28	3.11	4.66	3.28	4.89
2	0.50	3.04	5.62	5.54	3.37	5.11
3	0.75	3.14	6.32	5.95	3.41	5.18
4	1.00	3.16	6.59	6.20	3.43	5.23
5	1.25	3.19	6.83	6.34	3.44	5.26
6	1.50	3.17	6.95	6.45	3.46	5.27
7	1.75	3.18	6.86	6.53	3.46	5.28
8	2.00	3.17	7.03	6.59	3.47	5.29
9	2.25	3.20	7.11	6.64	3.46	5.30
10	2.50	3.19	7.15	6.68	3.48	5.30
11	2.75	3.38	7.20	6.70	3.47	5.30
12	3.00	3.27	7.25	6.73	3.48	5.31
13	3.25	3.22	7.29	6.76	3.48	5.31
14	3.50	3.20	7.31	6.76	3.48	5.31
15	3.75	3.22	7.26	6.79	3.48	5.32
16	4.00	3.22	7.42	6.81	3.49	5.32
17	4.25	3.27	7.34	6.82	3.48	5.32
18	4.50	3.34	7.36	6.84	3.49	5.32
19	4.75	3.21	7.41	6.85	3.48	5.33
20	5.00	3.22	7.47	6.85	3.49	5.33
	Arithmetic Intensity	PLUTO (non-vec) mem.bandw.	PLUTO (vec) mem.bandw.	PLUTO-PATUS mem.bandw.	icc-15.02, non-vec mem.bandw.	icc-15.02, vec mem.bandw.
1	0.25	5.73	8.26	10.93	7.67	11.48
2	0.50	5.36	9.80	8.75	8.13	12.17
3	0.75	4.10	8.67	6.29	8.21	12.43
4	1.00	3.44	7.70	4.93	8.26	12.56
5	1.25	3.03	6.48	4.05	8.30	12.64
6	1.50	2.75	6.37	3.44	8.36	12.68
7	1.75	2.57	5.59	3.00	8.36	12.73
8	2.00	2.42	5.22	2.66	8.38	12.75
9	2.25	2.32	5.37	2.38	8.39	12.78
10	2.50	2.22	5.13	2.17	8.40	12.79
11	2.75	2.10	4.80	1.99	8.41	12.81
12	3.00	2.06	4.51	1.83	8.40	12.82
13	3.25	2.00	4.42	1.70	8.42	12.83
14	3.50	1.96	4.33	1.59	8.43	12.84
15	3.75	1.90	4.21	1.49	8.41	12.85
16	4.00	1.86	4.13	1.41	8.41	12.86
17	4.25	1.84	4.08	1.33	8.42	12.85
18	4.50	1.81	4.01	1.26	8.42	12.87
19	4.75	1.78	3.97	1.20	8.43	12.87
20	5.00	1.75	3.93	1.15	8.43	12.87

Table C.1: Temporal-spatial tiling scaling of 5-point stencil (grid size=4002^2) with constant coefficients on one core of an Intel Ivy Bridge socket.

	Arithmetic Intensity	PLUTO (non-vec) Perf.	PLUTO (vec) Perf.	PLUTO-PATUS Perf.	icc-15.02, non-vec Perf.	icc-15.02, vec Perf.
1	0.33	3.36	6.20	5.80	3.38	5.81
2	0.67	3.41	7.07	6.80	3.47	5.79
3	1.00	3.33	7.28	7.19	3.47	6.15
4	1.33	3.34	7.18	7.37	3.50	5.99
5	1.67	3.29	7.58	7.45	3.50	5.97
6	2.00	3.39	7.34	7.52	3.51	6.25
7	2.33	3.37	7.48	7.31	3.51	6.05
8	2.67	3.37	7.26	7.56	3.51	5.99
9	3.00	3.39	7.34	7.54	3.52	5.97
10	3.33	3.39	7.46	7.55	3.52	6.02
11	3.67	3.38	7.49	7.56	3.53	6.00
12	4.00	3.36	7.35	7.59	3.52	6.07
13	4.33	3.37	7.37	7.60	3.52	6.02
14	4.67	3.53	7.46	7.60	3.52	6.00
15	5.00	3.36	7.36	7.58	3.52	6.07
16	5.33	3.48	7.48	7.59	3.53	6.03
17	5.67	3.38	7.39	7.57	3.53	6.02
18	6.00	3.51	7.45	7.55	3.53	6.05
19	6.33	3.38	7.54	7.55	3.52	6.02
20	6.67	3.37	7.41	7.57	3.53	6.03
	Arithmetic Intensity	PLUTO (non-vec) mem.bandw.	PLUTO (vec) mem.bandw.	PLUTO-PATUS mem.bandw.	icc-15.02, non-vec mem.bandw.	icc-15.02, vec mem.bandw.
1	0.33	5.25	10.35	9.36	5.45	8.96
2	0.67	3.72	7.70	7.26	5.64	9.44
3	1.00	2.72	5.64	5.18	5.70	9.58
4	1.33	2.34	4.89	3.99	5.75	9.70
5	1.67	2.13	4.49	3.24	5.76	9.73
6	2.00	1.98	4.18	2.74	5.78	9.77
7	2.33	1.85	3.93	2.37	5.79	9.81
8	2.67	1.73	3.68	2.09	5.80	9.84
9	3.00	1.61	3.41	1.85	5.80	9.84
10	3.33	1.50	3.38	1.69	5.81	9.86
11	3.67	1.44	3.05	1.53	5.81	9.86
12	4.00	1.42	3.01	1.41	5.82	9.88
13	4.33	1.41	3.01	1.31	5.82	9.88
14	4.67	1.40	2.98	1.22	5.82	9.89
15	5.00	1.39	2.94	1.15	5.82	9.89
16	5.33	1.40	2.88	1.08	5.83	9.90
17	5.67	1.35	2.79	1.01	5.83	9.90
18	6.00	1.29	2.70	0.96	5.83	9.91
19	6.33	1.25	2.65	0.91	5.83	9.91
20	6.67	1.26	2.64	0.87	5.83	9.91

Table C.2: Temporal-spatial scaling of 7-point stencil (grid size=150^3) with constant coefficients on one core of an Intel Ivy Bridge socket.

	Arithmetic Intensity	PLUTO (non-vec) Perf.	PLUTO (vec) Perf.	PLUTO-PATUS Perf.	icc-15.02, non-vec Perf.	icc-15.02, vec Perf.
1	0.33	3.24	5.96	5.72	3.41	5.99
2	0.67	3.33	6.45	6.20	3.48	6.25
3	1.00	3.37	6.65	6.39	3.51	6.33
4	1.33	3.38	6.88	6.51	3.53	6.37
5	1.67	3.39	6.83	6.57	3.54	6.42
6	2.00	3.38	7.30	6.59	3.54	6.43
7	2.33	3.39	6.94	6.64	3.55	6.45
8	2.67	3.39	6.96	6.68	3.55	6.45
9	3.00	3.46	6.98	6.67	3.55	6.50
10	3.33	3.48	6.99	6.70	3.56	6.50
11	3.67	3.47	7.01	6.74	3.56	6.52
12	4.00	3.49	7.02	6.71	3.55	6.48
13	4.33	3.43	7.04	6.71	3.56	6.49
14	4.67	3.45	7.04	6.73	3.55	6.49
15	5.00	3.40	7.18	6.70	3.56	6.49
16	5.33	3.40	7.05	6.71	3.56	6.49
17	5.67	3.41	7.05	6.73	3.56	6.50
18	6.00	3.41	7.28	6.73	3.56	6.55
19	6.33	3.40	7.06	6.72	3.56	6.53
20	6.67	3.51	7.07	6.72	3.56	6.50
	Arithmetic Intensity	PLUTO (non-vec) mem.bandw.	PLUTO (vec) mem.bandw.	PLUTO-PATUS mem.bandw.	icc-15.02, non-vec mem.bandw.	icc-15.02, vec mem.bandw.
1	0.33	5.81	10.67	9.75	5.41	9.41
2	0.67	4.13	8.16	8.67	5.63	9.98
3	1.00	3.10	6.14	7.63	5.70	10.14
4	1.33	2.58	5.15	7.02	5.74	10.31
5	1.67	2.26	4.55	6.60	5.76	10.32
6	2.00	2.05	4.13	6.34	5.77	10.37
7	2.33	1.91	3.84	6.10	5.79	10.44
8	2.67	1.79	3.62	5.92	5.80	10.43
9	3.00	1.72	3.42	5.73	5.80	10.45
10	3.33	1.63	3.27	5.67	5.81	10.46
11	3.67	1.57	3.16	5.58	5.81	10.48
12	4.00	1.52	3.07	5.48	5.82	10.49
13	4.33	1.48	2.98	5.42	5.82	10.51
14	4.67	1.45	2.91	5.37	5.82	10.50
15	5.00	1.42	2.86	5.28	5.82	10.53
16	5.33	1.39	2.80	5.28	5.83	10.53
17	5.67	1.37	2.76	5.21	5.83	10.52
18	6.00	1.35	2.71	5.18	5.83	10.52
19	6.33	1.33	2.68	5.19	5.83	10.52
20	6.67	1.31	2.64	5.09	5.83	10.53

Table C.3: Spatial-temporal scaling of 7-point stencil (grid size=502^3) with constant coefficients on one core of an Intel Ivy Bridge socket.

	Arithmetic Intensity	PLUTO (non-vec) Perf.	PLUTO (vec) Perf.	PLUTO-PATUS Perf.	icc-15.02, non-vec Perf.	icc-15.02, vec Perf.
1	0.29	1.19	2.28	1.95	1.20	2.48
2	0.58	1.22	2.37	2.05	1.22	2.50
3	0.87	1.23	2.41	2.02	1.22	2.50
4	1.16	1.24	2.43	1.97	1.22	2.51
5	1.45	1.32	2.44	1.94	1.22	2.50
6	1.74	1.23	2.44	1.95	1.21	2.51
7	2.03	1.23	2.45	1.93	1.21	2.50
8	2.32	1.23	2.45	1.91	1.21	2.51
9	2.61	1.28	2.45	1.91	1.21	2.50
10	2.90	1.26	2.46	1.88	1.21	2.50
11	3.19	1.29	2.46	1.87	1.26	2.51
12	3.48	1.23	2.46	1.86	1.20	2.50
13	3.77	1.28	2.46	1.85	1.24	2.51
14	4.06	1.23	2.46	1.83	1.21	2.51
15	4.35	1.24	2.46	1.83	1.21	2.51
16	4.64	1.24	2.47	1.81	1.21	2.51
17	4.93	1.24	2.47	1.80	1.21	2.51
18	5.22	1.24	2.47	1.79	1.21	2.51
19	5.51	1.24	2.47	1.79	1.21	2.51
20	5.80	1.24	2.47	1.78	1.21	2.51
	Arithmetic Intensity	PLUTO (non-vec) mem.bandw.	PLUTO (vec) mem.bandw.	PLUTO-PATUS mem.bandw.	icc-15.02, non-vec mem.bandw.	icc-15.02, vec mem.bandw.
1	0.29	4.84	8.87	7.35	5.08	9.16
2	0.58	3.68	7.07	5.75	4.80	9.22
3	0.87	3.29	6.30	6.35	4.57	9.25
4	1.16	3.07	5.92	7.02	4.56	9.26
5	1.45	2.93	5.69	7.20	4.63	9.25
6	1.74	2.82	5.54	7.29	4.56	9.26
7	2.03	2.76	5.42	7.23	4.57	9.26
8	2.32	2.71	5.33	7.22	4.55	9.26
9	2.61	2.67	5.27	7.19	4.57	9.26
10	2.90	2.64	5.21	7.15	4.57	9.25
11	3.19	2.62	5.17	7.14	4.54	9.27
12	3.48	2.60	5.13	7.12	4.56	9.26
13	3.77	2.58	5.11	7.14	4.57	9.25
14	4.06	2.57	5.07	7.13	4.52	9.26
15	4.35	2.56	5.06	7.12	4.52	9.23
16	4.64	2.54	5.04	7.13	4.52	9.26
17	4.93	2.55	5.02	7.17	4.53	9.27
18	5.22	2.54	5.01	7.17	4.52	9.26
19	5.51	2.52	5.00	7.17	4.52	9.28
20	5.80	2.52	4.97	7.23	4.53	9.27

Table C.4: Spatial-temporal scaling of long-range stencil (grid size=150^3) with variable coefficients on one core of an Intel Ivy Bridge socket.

Appendix D

Temporal blocking results on ten cores

Spatial blocking Perf. (in Gflops/s) and mem.bandwidth usage (in Gbytes/s) for two state-of-the-art stencil compilers for various stencils on 10 cores of Intel Ivy Bridge processor.

	Arithmetic Intensity	PLUTO (non-vec) Perf.	PLUTO (vec) Perf.	PLUTO-PATUS Perf.	icc-15.02, non-vec Perf.	icc-15.02, vec Perf.
1	0.25	13.66	14.43	4.64	16.03	16.15
2	0.50	20.37	21.13	5.52	16.30	16.29
3	0.75	25.58	28.55	5.93	16.30	16.48
4	1.00	28.13	34.19	6.17	16.38	16.45
5	1.25	29.30	38.62	6.31	16.51	16.40
6	1.50	29.91	42.23	6.43	16.59	16.49
7	1.75	30.26	46.16	6.50	16.71	16.55
8	2.00	30.24	49.22	6.57	16.60	16.56
9	2.25	30.00	50.80	6.61	16.42	16.56
10	2.50	30.02	52.48	6.66	16.37	16.58
11	2.75	30.20	53.68	6.67	16.64	16.68
12	3.00	30.50	54.20	6.71	16.68	16.69
13	3.25	30.78	54.93	6.73	16.53	16.37
14	3.50	31.01	56.05	6.76	16.43	16.42
15	3.75	30.77	56.61	6.77	16.50	16.47
16	4.00	31.27	58.38	6.78	16.49	16.72
17	4.25	30.64	58.16	6.80	16.28	16.61
18	4.50	30.52	58.95	6.81	16.51	16.43
19	4.75	30.56	59.62	6.82	16.54	16.66
20	5.00	30.36	59.98	6.83	16.42	16.76
	Arithmetic Intensity	PLUTO (non-vec) mem.bandw.	PLUTO (vec) mem.bandw.	PLUTO-PATUS mem.bandw.	icc-15.02, non-vec mem.bandw.	icc-15.02, vec mem.bandw.
1	0.25	32.32	33.96	10.63	34.85	38.86
2	0.50	34.83	35.60	8.56	37.16	37.84
3	0.75	32.43	36.05	6.20	38.29	38.51
4	1.00	29.31	36.27	4.87	38.66	38.78
5	1.25	26.90	35.70	4.01	38.75	38.90
6	1.50	24.91	34.93	3.42	38.73	39.14
7	1.75	23.32	35.02	2.98	39.07	39.14
8	2.00	21.75	34.85	2.64	39.17	39.76
9	2.25	20.48	33.80	2.37	39.23	39.72
10	2.50	19.62	34.20	2.15	39.49	39.39
11	2.75	19.02	33.55	1.97	39.18	39.52
12	3.00	18.61	32.60	1.83	39.40	39.69
13	3.25	18.30	32.55	1.70	39.74	39.63
14	3.50	17.81	32.57	1.59	39.63	39.23
15	3.75	17.47	32.14	1.49	39.76	39.83
16	4.00	17.02	32.00	1.41	38.96	39.87
17	4.25	16.79	31.75	1.33	39.30	40.09
18	4.50	16.45	31.56	1.26	39.67	40.14
19	4.75	16.67	31.29	1.20	39.35	39.77
20	5.00	16.13	31.05	1.15	39.11	39.89

Table D.1: Temporal-spatial tiling scaling of 5-point stencil (grid size=4002^2) with constant coefficients on 10 cores of an Intel Ivy Bridge socket.

	Arithmetic Intensity	PLUTO (non-vec) Perf.	PLUTO (vec) Perf.	PLUTO-PATUS Perf.	icc-15.02, non-vec Perf.	icc-15.02, vec Perf.
1	0.33	22.39	22.75	5.73	21.91	22.24
2	0.67	24.29	30.15	6.74	22.30	23.28
3	1.00	26.98	37.83	7.16	22.73	23.32
4	1.33	28.14	42.47	7.29	23.08	23.41
5	1.67	28.91	45.95	7.38	23.33	23.34
6	2.00	29.40	48.13	7.43	23.24	23.65
7	2.33	29.88	50.66	7.47	23.48	23.83
8	2.67	30.00	52.79	7.48	23.03	23.50
9	3.00	30.19	54.27	7.46	23.37	23.76
10	3.33	30.27	55.18	7.52	23.71	23.25
11	3.67	30.31	56.18	7.51	23.50	23.14
12	4.00	30.25	56.30	7.53	23.54	23.77
13	4.33	30.30	56.17	7.52	23.69	22.86
14	4.67	30.32	56.87	7.52	23.75	22.86
15	5.00	30.45	57.41	7.55	23.76	23.10
16	5.33	30.56	58.78	7.50	23.75	22.91
17	5.67	30.32	58.66	7.47	23.40	23.18
18	6.00	30.65	58.78	7.48	23.08	23.01
19	6.33	30.60	59.59	7.48	23.99	23.20
20	6.67	30.86	60.10	7.47	23.78	23.64
	Arithmetic Intensity	PLUTO (non-vec) mem.bandw.	PLUTO (vec) mem.bandw.	PLUTO-PATUS mem.bandw.	icc-15.02, non-vec mem.bandw.	icc-15.02, vec mem.bandw.
1	0.33	24.19	24.55	8.20	24.41	24.91
2	0.67	22.59	26.57	6.65	29.08	29.78
3	1.00	19.05	24.14	4.82	31.28	34.09
4	1.33	17.67	23.64	3.77	32.95	33.86
5	1.67	16.00	24.37	3.11	35.03	35.00
6	2.00	15.28	24.34	2.64	35.40	36.07
7	2.33	14.60	23.48	2.29	35.27	36.54
8	2.67	13.97	23.42	2.01	35.84	36.98
9	3.00	13.35	22.21	1.80	36.50	36.31
10	3.33	12.46	21.39	1.63	36.42	36.02
11	3.67	12.14	20.96	1.50	37.08	37.12
12	4.00	12.02	20.78	1.40	36.65	37.71
13	4.33	12.02	21.46	1.29	37.39	37.92
14	4.67	11.85	21.34	1.20	37.53	38.01
15	5.00	11.63	20.89	1.13	37.45	37.61
16	5.33	11.52	20.98	1.06	37.76	37.47
17	5.67	11.18	20.41	1.00	37.65	38.27
18	6.00	10.86	20.22	0.95	38.77	38.49
19	6.33	10.63	19.75	0.90	38.24	38.43
20	6.67	10.59	19.76	0.87	37.94	38.56

Table D.2: Temporal-spatial scaling of 7-point stencil (grid size=150^3) with constant coefficients on 10 cores of an Intel Ivy Bridge socket.

	Arithmetic Intensity	PLUTO (non-vec) Perf.	PLUTO (vec) Perf.	PLUTO-PATUS Perf.	icc-15.02, non-vec Perf.	icc-15.02, vec Perf.
1	0.33	18.96	20.98	5.70	15.09	15.19
2	0.67	22.17	26.42	6.17	15.21	15.33
3	1.00	23.88	31.02	6.39	15.29	15.39
4	1.33	24.46	34.50	6.50	15.34	15.40
5	1.67	25.26	38.08	6.56	15.32	15.44
6	2.00	25.38	39.49	6.61	15.37	15.45
7	2.33	25.60	40.64	6.61	15.37	15.46
8	2.67	25.64	42.04	6.63	15.39	15.47
9	3.00	25.50	41.76	6.64	15.40	15.50
10	3.33	25.56	42.93	6.65	15.38	15.48
11	3.67	25.76	43.42	6.67	15.37	15.52
12	4.00	25.74	43.73	6.68	15.38	15.53
13	4.33	25.86	44.61	6.68	15.40	15.54
14	4.67	25.95	44.88	6.71	15.40	15.51
15	5.00	25.91	45.55	6.72	15.42	15.49
16	5.33	25.92	45.50	6.72	15.40	15.48
17	5.67	25.84	45.89	6.73	15.44	15.53
18	6.00	25.85	45.50	6.71	15.43	15.52
19	6.33	25.93	45.69	6.73	15.39	15.51
20	6.67	26.00	45.97	6.74	15.40	15.51
	Arithmetic Intensity	PLUTO (non-vec) mem.bandw.	PLUTO (vec) mem.bandw.	PLUTO-PATUS mem.bandw.	icc-15.02, non-vec mem.bandw.	icc-15.02, vec mem.bandw.
1	0.33	34.74	38.40	9.74	41.17	41.33
2	0.67	29.84	35.24	8.72	41.37	41.43
3	1.00	24.81	32.71	7.65	41.50	41.55
4	1.33	21.67	30.06	7.06	41.42	41.51
5	1.67	19.29	29.16	6.68	41.58	41.54
6	2.00	18.07	27.71	6.36	41.49	41.59
7	2.33	17.03	27.36	6.17	41.64	41.65
8	2.67	16.70	27.07	6.00	41.59	41.65
9	3.00	16.40	26.81	5.86	41.43	41.67
10	3.33	16.11	26.53	5.77	41.61	41.67
11	3.67	15.74	26.41	5.68	41.42	41.69
12	4.00	15.34	26.07	5.54	41.57	41.70
13	4.33	14.79	26.06	5.47	41.59	41.65
14	4.67	14.48	25.27	5.45	41.53	41.54
15	5.00	14.36	24.77	5.47	41.53	41.77
16	5.33	14.30	24.86	5.35	41.64	41.59
17	5.67	14.44	25.07	5.34	41.58	41.68
18	6.00	14.16	24.99	5.28	41.68	41.74
19	6.33	14.13	24.76	5.32	41.45	41.62
20	6.67	13.70	24.62	5.16	41.61	41.73

Table D.3: Spatial-temporal scaling of 7-point stencil (grid size=502^3) with constant coefficients on 10 cores of an Intel Ivy Bridge socket.

	Arithmetic Intensity	PLUTO (non-vec) Perf.	PLUTO (vec) Perf.	PLUTO-PATUS Perf.	icc-15.02, non-vec Perf.	icc-15.02, vec Perf.
1	0.29	8.74	10.31	1.93	7.30	7.88
2	0.58	9.00	11.05	2.02	7.36	7.88
3	0.87	10.15	11.18	1.98	7.37	7.99
4	1.16	10.01	11.37	1.96	7.33	7.94
5	1.45	10.13	11.41	1.93	7.45	7.98
6	1.74	9.99	11.60	1.92	7.43	8.00
7	2.03	9.26	11.57	1.91	7.48	8.00
8	2.32	9.28	11.64	1.89	7.45	7.96
9	2.61	9.23	11.62	1.88	7.40	7.87
10	2.90	9.55	11.67	1.87	7.40	7.86
11	3.19	9.58	11.57	1.85	7.47	7.99
12	3.48	9.71	11.67	1.84	7.49	7.97
13	3.77	9.62	11.55	1.82	7.49	7.99
14	4.06	9.73	11.61	1.82	7.49	7.96
15	4.35	9.56	11.72	1.80	7.42	8.03
16	4.64	9.62	11.74	1.79	7.50	8.02
17	4.93	9.52	11.70	1.77	7.50	7.98
18	5.22	9.51	11.75	1.76	7.50	7.98
19	5.51	9.50	11.70	1.76	7.46	7.96
20	5.80	9.48	11.78	1.76	7.50	8.01
	Arithmetic Intensity	PLUTO (non-vec) mem.bandw.	PLUTO (vec) mem.bandw.	PLUTO-PATUS mem.bandw.	icc-15.02, non-vec mem.bandw.	icc-15.02, vec mem.bandw.
1	0.29	33.65	37.00	7.16	36.66	37.86
2	0.58	33.69	39.47	5.58	38.12	40.08
3	0.87	33.52	39.92	6.28	38.89	40.51
4	1.16	33.77	40.05	6.97	39.71	41.06
5	1.45	33.68	40.43	7.17	40.02	41.09
6	1.74	33.61	40.62	7.20	39.93	41.35
7	2.03	34.14	40.56	7.19	39.30	41.22
8	2.32	33.85	40.61	7.14	39.58	41.16
9	2.61	34.06	40.73	7.14	40.24	41.10
10	2.90	34.32	40.81	7.11	40.04	41.64
11	3.19	33.61	40.78	7.12	39.90	41.88
12	3.48	34.37	40.75	7.12	39.64	41.60
13	3.77	34.13	40.94	7.10	39.95	41.72
14	4.06	33.98	41.06	7.11	40.28	41.64
15	4.35	33.96	40.83	7.13	40.34	41.78
16	4.64	34.21	40.98	7.12	40.03	41.31
17	4.93	34.18	40.91	7.12	40.58	41.58
18	5.22	34.56	40.97	7.19	39.94	41.74
19	5.51	34.74	41.09	7.20	40.39	41.31
20	5.80	34.44	41.08	7.23	40.52	41.72

Table D.4: Spatial-temporal scaling of long-range stencil (grid size=150^3) with variable coefficients on 10 cores of an Intel Ivy Bridge socket.

Appendix E

Content of attached CD

- multigrid - multigrid source codes + measurements
  - geometric_multigrid_tiled_avx_icc_31 - tuned for tile size 31
  - geometric_multigrid_tiled_avx_icc_1023 - tuned for tile size 1023
  - geometric_multigrid_tiled_avx_icc_4095 - tuned for tile size 4095
- stencils - stencil source codes + measurements
  - exa2ct-radim-olaf-may15 - source codes
  - exa2ct-radim-olaf-may15_run - source codes + binaries + results
  - report-results - results used in this report
- tex - LaTeX source codes
- DP_Radim_Janalik.pdf - technical report