A Stencil computation algorithm optimization

Ivan Buccella
Department of Computer Science
University of Salerno
Italy

***Abstract*—This article tries to apply several types of optimizations to a Stencil computation algorithm, in order to minimize the computation execution time. In this document are treated the concepts of OpenMP multithreading code optimizations, loop interchange optimization, parallel loops optimizations, and loop tiling optimizations. From the experiment results, the best optimization to apply to the chosen Stencil computation algorithm is loop tiling. It also tries to increase the algorithm dimension on the optimized code in order to compare the performances increase/decrease.**

I. Introduction

Stencil computation has been a research topic for decades in a variety of domains including computational electromagnetics [1], solution of PDEs using finite difference or finite volume discretizations [2], and image processing for CT and MRI imaging [3][4]. In the finite-difference computation process, stencil computation is often iteratively used to solve the differential operator. In these scientific and engineering applications, Stencil is often the most vital and time-consuming computing kernel [5].

The general computation pattern of stencil computation is that the central point accumulates the contributions of neighbor points in every axis of the Cartesian coordinate system [5]. The number of neighbor points in each axis or grid step of the stencil computation corresponds to the accuracy of the stencil. The more neighbor elements involved in the computation, the higher accuracy the computation obtains. The computation is then repeated for every point of the grid domain as an iterative operator of the PDEs. Due to the structure of the stencil computation, two inherent problems exist:

- First, it is the non-continuous memory access pattern. There exist distances among elements needed for the computation except those in the innermost or unit-stride dimension. Many more cycles in latencies are required to access these points.
- Second, it is the low arithmetic intensity and poor data reuse. Just one point is updated with all the elements loaded. The data reuse between two updates is also limited within the unit-stride dimension, while the other dimensions' elements that are expensive to access have no data reuse at all.

Motivation. On the stencil computation various optimization approaches have been discussed. The main contributions for stencil optimization can be divided into two aspects: blocking and parallelism optimizations [5].

Blocking optimizations aim at improving the data locality of both space and time dimensions. They are highly related to the tiling strategies widely employed in the modern multi-level cache hierarchy architectures.

Parallelism optimizations refer to the techniques that explore parallelism at diverse levels, including data-level parallelisms, such as SIMD, thread-level parallelism, such as block decomposition, and instruction-level parallelism. They tend to make full use of the potential advantages of modern processors' many- or multi-core architectures.

However, these optimizations can be categorized according to their complexity of implementation (programmer efforts), benefit improvement (performance) and implementation tightness regarding hardware (dependence) [6].

Related work. In the article '*Acceleration of real-life stencil codes on GPUs*' [7] there is a detailed description of the loop tiling transformation, the CUDA programming model, and GPU architecture consists. Then, it explores the possible tiling approaches that can be applied to such stencil patterns; it describes 3 interesting approaches: 2D tiles, time-skewing+time-tiling transformation, and performing redundant computations inside tiles.

The article '*High-Performance Code Generation for Stencil Computations on GPU Architectures*' [8] introduces an automatic code generation scheme for stencil computations on GPU architectures. This scheme uses overlapped tiling to provide efficient time tiling on GPU architectures, which are massively threaded but are susceptible to performance degradation due to branch divergence and a lack of memory coalescing.

However, this article tries to apply multi-core architecture optimizations using OpenMP instead of using CUDA programming models and GPU architecture like in the related works.

II. Background

In order to understand the described work, it is important to know something about multi-core, and code optimizations using OpenMP.

The multi-core architecture. Multicore refers to an architecture in which a single physical processor incorporates the core logic of more than one processor. A single integrated circuit is used to package or hold these processors. These single integrated circuits are known as a die. Multicore architecture places multiple processor cores and bundles them as a single physical processor. The objective is to create a system that can

complete more tasks at the same time, thereby gaining better overall system performance [9].

OpenMP. The OpenMP API supports multi-platform shared-memory parallel programming in C/C++ and Fortran. The OpenMP API defines a portable, scalable model with a simple and flexible interface for developing parallel applications on platforms from the desktop to the supercomputer [10].

Loop interchange. In compiler theory, loop interchange is the process of exchanging the order of two iteration variables used by a nested loop. The variable used in the inner loop switches to the outer loop, and vice versa. It is often done to ensure that the elements of a multi-dimensional array are accessed in the order in which they are present in memory, improving the locality of reference [11].

Optimization flags. Without any optimization option, the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. Statements are independent: if you stop the program with a breakpoint between statements, you can then assign a new value to any variable or change the program counter to any other statement in the function and get exactly the results you expect from the source code. Turning on optimization flags makes the compiler attempt to improve the performance and/or code size at the expense of compilation time and possibly the ability to debug the program. The compiler performs optimization based on the knowledge it has of the program. Depending on the target and how the compiler was configured, a slightly different set of optimizations may be enabled at each -O level (*-O0*,*-O1*,*-O2*,*-O3*) [12].

Parallel loops. A parallel for loop is a for loop in which the statements in the loop can be run in parallel: on separate cores, processors, or threads. In OpenMP the *parallel loop* construct is a shortcut for specifying a parallel construct containing a loop construct with one or more associated loops and no other statements [15].

Loop tiling. Loop tiling transformation (or simply tiling) consists of partitioning an iteration domain into many regular and uniform tiles [7]. A tile is thus a subset of iteration points. Executing a tile corresponds to the execution of all the iteration points it contains. A tiles execution is atomic, which means if a tile execution starts, it cannot be interrupted until it reaches its last iteration point. This condition implies that no data can be exchanged between two concurrent tiles. This transformation allows data to be accessed in blocks (tiles), with the block size defined as a parameter of this transformation. Each loop is transformed into two loops: one iterating inside each block (intratile) and the other one iterating over the blocks (intertile) [13].

III. The applied method

The main goal of this experiment is to minimize as soon as possible the computation time of the chosen algorithm for Stencil Computation. This operation is done by using the above explained techniques: *loop interchange*, *optimization flags*, *parallel loops* and *loop tiling*.

Starting Alogrithm. First of all, it's important to show what is the used algorithm that the experiment tries to improve.

```
for (x = 1; x < N − 1; x++)
{
    for (y = 1; y < N − 1; y++)
    {
        B[x][y] = a ∗ A[x][y] + b ∗ (A[x − 1][y] + A[x
            + 1][y] + A[x][y − 1] + A[x][y + 1]);
    }
}
```

Loop interchange. The first step is editing the starting algorithm by exchanging the order of the iteration variables **x** and **y** used by nested loops. The result is two different algorithms:

Algorithm 1: The starting algorithm

```
for (x = 1; x < N − 1; x++)
{
    for (y = 1; y < N − 1; y++)
    {
        B[x][y] = a ∗ A[x][y] + b ∗ (A[x − 1][y] + A[x
            + 1][y] + A[x][y − 1] + A[x][y + 1]);
    }
}
```

Algorithm 2: A new algorithm with x and y swapped

```
for (y = 1; y < N − 1; y++)
{
    for (x = 1; x < N − 1; x++)
    {
        B[x][y] = a ∗ A[x][y] + b ∗ (A[x − 1][y] + A[x
            + 1][y] + A[x][y − 1] + A[x][y + 1]);
    }
}
```

Optimization flags. The next step is editing the optimization flag used for compiling the source code; the clang compiler offers four different flag specifications: O0, O1, O2, and O3. The result is four different commands:

Command 1: Flag -O0

```
clang code.c −o code.out −fopenmp −O0
```

Command 2: Flag -O1

```
clang code.c −o code.out −fopenmp −O1
```

Command 3: Flag -O2

```
clang code.c -o code.out -fopenmp -O2
```

Command 4: Flag -O3

```
clang code.c -o code.out -fopenmp -O3
```

Parallel Loops. The next step is editing the code by applying the parallel loops optimization. In order to distribute the execution of a loop into multiple threads, it has been used the "pragma omp parallel for" [15][16] directive; this directive opens a parallel region and schedules the loop into multiple threads. The directive could be applied to each loop, also on both. For applying the directive on both loops, it has been used the "pragma omp parallel for collapse(2)" [17] directive which is able to distribute two nested loops on multiple threads. The result of this step is three different algorithms:

Algorithm 3: Parallel loop on X

```
#pragma omp parallel for
for (x = 1; x < N - 1; x++)
{
    for (y = 1; y < N - 1; y++)
    {
        B[x][y] = a * A[x][y] + b * (A[x - 1][y] + A[x
            + 1][y] + A[x][y - 1] + A[x][y + 1]);
    }
}
```

Algorithm 4: Parallel loop on Y

```
for (x = 1; x < N - 1; x++)
{
    #pragma omp parallel for
    for (y = 1; y < N - 1; y++)
    {
        B[x][y] = a * A[x][y] + b * (A[x - 1][y] + A[x
            + 1][y] + A[x][y - 1] + A[x][y + 1]);
    }
}
```

Algorithm 5: Parallel loop on X and Y

```
#pragma omp parallel for collapse (2)
for (x = 1; x < N - 1; x++)
{
    for (y = 1; y < N - 1; y++)
    {
        B[x][y] = a * A[x][y] + b * (A[x - 1][y] + A[x
            + 1][y] + A[x][y - 1] + A[x][y + 1]);
    }
}
```

Loop tiling. The next step is editing the code by applying tiling optimization. In order to apply this type of optimization it has been necessary to cut the **A** matrices into multiple smaller *tiles*; due to it has been introduced into the code a new variable "tile_size" which expresses the tile size, and two nested loops that have the goal of executing the code on the different tiles. On the tiling loops (with *xx* and *yy* variables) it has been applied the "pragma omp parallel for collapse(2)" directive in order to distribute the matrices tiles on multiple threads. Different tile sizes have been chosen depending on several points of view like L2 Cache size, etc. The result of this step is four different algorithms, identical but with different tile sizes:

Algorithm 6: Loop tiling with size of 16384

```
const int tile_size = 16384;
#pragma omp parallel for collapse (2)
for (int xx = 0; xx < N; xx += tile_size)
{
    for (int yy = 0; yy < N; yy += tile_size)
    {
        for (x = xx + 1; x < MIN(xx + tile_size, N -
            1); x++)
        {
            for (y = yy + 1; y < MIN(yy + tile_size, N
                - 1); y++)
            {
                B[x][y] = a * A[x][y] + b * (A[x -
                    1][y] + A[x + 1][y] + A[x][y - 1]
                    + A[x][y + 1]);
            }
        }
    }
}
```

Algorithm 7: Loop tiling with size of 10922

```
const int tile_size = 10922;
... same as before
```

Algorithm 8: Loop tiling with size of 8192

```
const int tile_size = 8192;
... same as before
```

Algorithm 9: Loop tiling with size of 5461

```
const int tile_size = 5461;
... same as before
```

Dimension augmentation. The next step is editing the *tiled* code by applying two more dimensions to the original stencil code. In order to apply this step, it has been necessary to edit the internal loops (with *x* and *y* variables) condition from "MIN(xx + tile_size, N - 1)" to "MIN(xx + tile_size, N - 3)". Because of the tiling optimization, the code has been executed on different tile sizes depending on several points of view like L2 Cache size, etc. The result of this step is four different algorithms, identical but with different tile sizes:

Algorithm 10: Dimension augmentation + loop tiling with size of 16384

```
const int tile_size = 16384;
#pragma omp parallel for collapse(2)
for (int xx = 0; xx < N; xx += tile_size)
{
    for (int yy = 0; yy < N; yy += tile_size)
    {
        for (x = xx + 3; x < MIN(xx + tile_size, N -
            3); x = x + 3)
        {
            for (y = yy + 3; y < MIN(yy + tile_size, N
                - 3); y = y + 3)
            {
                B[x][y] = a * A[x][y] + b * (A[x -
                    3][y] + A[x - 2][y] + A[x - 1][y]
                    + A[x + 1][y] + A[x + 2][y] + A[x
                    + 3][y] + A[x][y - 3] + A[x][y -
                    2] + A[x][y - 1] + A[x][y + 1] +
                    A[x][y + 2] + A[x][y + 3]);
            }
        }
    }
}
```

Algorithm 11: Dimension augmentation + loop tiling with size of 10922

```
const int tile_size = 10922;
... same as before
```

Algorithm 12: Dimension augmentation + loop tiling with size of 8192

```
const int tile_size = 8192;
... same as before
```

Algorithm 13: Dimension augmentation + loop tiling with size of 5461

```
const int tile_size = 5461;
... same as before
```

IV. Experimental Results

Experimental setup. The execution timing is obtained by using the OpenMP primitives [14], an example is shown in Algorithm 14. The OpenMP settings are shown in Algorithm 15. The experiment is executed on two matrices, A and B (initialized with the Algorithm 16), with a size of constant **N = 32768** and on a PC with current specifications:

- **CPU**
  - **Model name**: Intel(R) Core(TM) i5-9600K CPU @ 3.70GHz
  - **CPU(s)**: 6
  - **Socket(s)**: 1
  - **Core(s) per socket**: 6
  - **Thread(s) per core**: 1
  - **Cache L1d**: 192 KiB (6 instances)
  - **Cache L1i**: 192 KiB (6 instances)
  - **Cache L2**: 1.5 MiB (6 instances)
  - **Cache L3**: 9 MiB (1 instance)
  - **Intel Hyper-Threading Technology**: No
- **Memory**
  - **Size**: 26164116 kB
- **Compiler**: clang 14.0.5 with OpenMP 5.0

Algorithm 14: An example of code for obtaining the execution timing

```
double t_init = omp_get_wtime()
// Your code's here
printf ("\%0.15f", omp_get_wtime() - t_init);
```

Algorithm 15: OpenMP Configs

```
int n_threads = omp_get_max_threads();
omp_set_max_active_levels(1);
omp_set_num_threads(n_threads);
omp_set_dynamic(0);
```

Algorithm 16: A and B matrices initialization

```
float **A = (float **)malloc(N * sizeof(float *));
float **B = (float **)malloc(N * sizeof(float *));
for (x = 0; x < N; x++)
{
    A[x] = (float *)malloc(N * sizeof(float));
    B[x] = (float *)malloc(N * sizeof(float));
    for (y = 0; y < N; y++)
    {
        A[x][y] = 1.0f;
        B[x][y] = 0.0f;
    }
}
```

Results. Here the following are represented the execution timings for every experiment that has been done.

- **Loop interchange** computation time of:
  - the Algorithm 1 is: 5.0533 (s)
  - the Algorithm 2 is: 55.8343 (s)

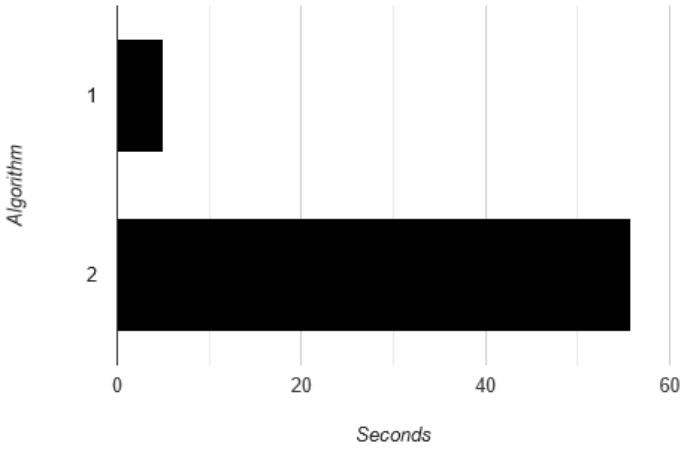

Fig. 1: Loop interchange computation timings

- **Optimization flags**:
  - the Algorithm 1 by compiling with the command 1 is: 5.0493 (s)
  - the Algorithm 1 by compiling with the command 2 is: 1.3734 (s)
  - the Algorithm 1 by compiling with the command 3 is: 0.5931 (s)
  - the Algorithm 1 by compiling with the command 4 is: 0.5904 (s)

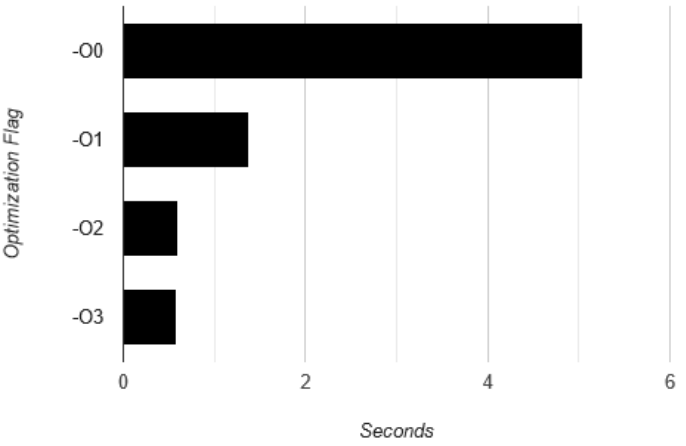

Fig. 2: Optimization flags computation timings

- **Parallel loops**:
  - the Algorithm 3 is: 0.4585 (s)
  - the Algorithm 4 is: 0.4760 (s)
  - the Algorithm 5 is: 0.7419 (s)

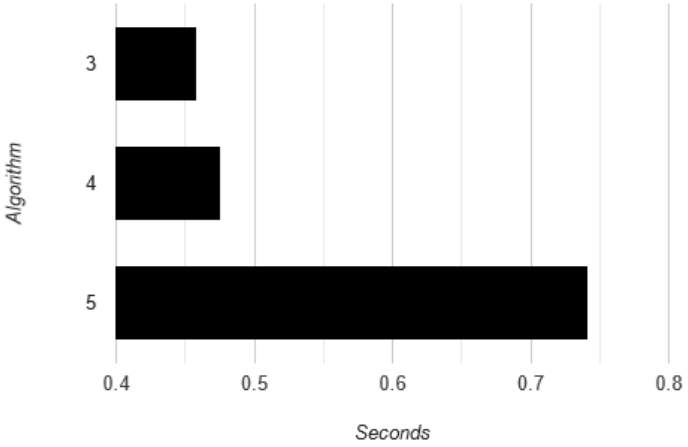

Fig. 3: Parallel loops computation timings

- **Loop tiling**:
  - the Algorithm 6 is: 0.4451 (s)
  - the Algorithm 7 is: 0.4599 (s)
  - the Algorithm 8 is: 0.4422 (s)
  - the Algorithm 9 is: 0.4537 (s)

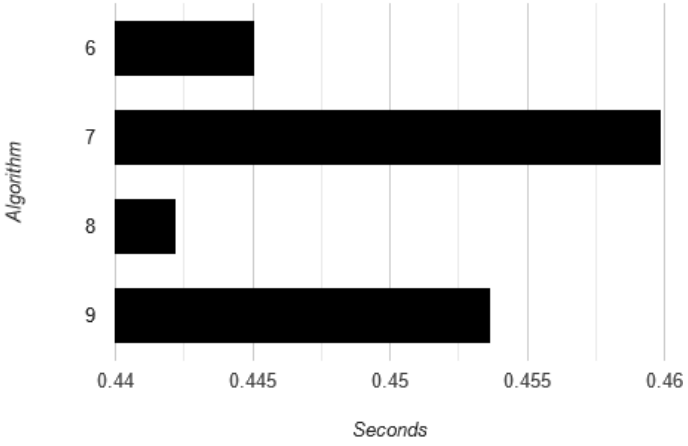

Fig. 4: Loop tiling computation timings

- **Dimension augmentation**:
  - the Algorithm 10 is: 0.2649 (s)
  - the Algorithm 11 is: 0.2911 (s)
  - the Algorithm 12 is: 0.2645 (s)
  - the Algorithm 13 is: 0.2676 (s)

V. Conclusions

This article presents a number of different approaches to optimize the above-detailed stencil computations algorithm on multi-core architectures, based on a range of micro benchmarks. For the **loop interchange** optimization, it indicates that the standard loop *x-y* is faster than the *y-x* loop, this is favored by the C programming language memory layout. For the **optimization flags** optimization, it indicates that the best flag

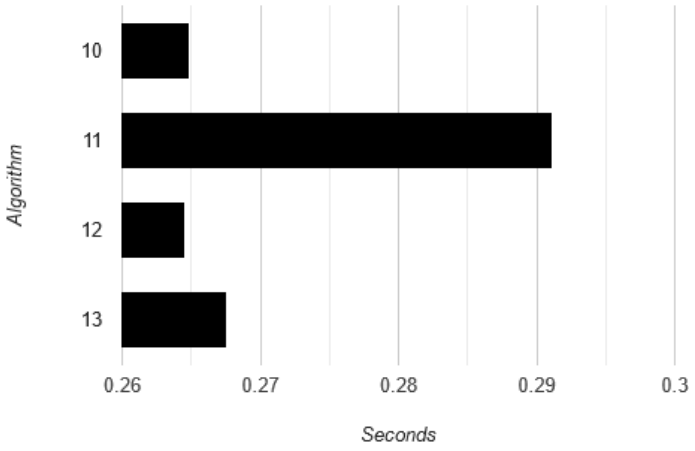

Fig. 5: Dimension augmentation computation timings

to use for compiling the code is the -O3. For the **parallel loop** optimization, it indicates that having only one parallel loop on the external loop is faster. For the **loop tiling** optimization, it indicates that having a *tile_size* of *8192* is faster. About the **dimension** of the Stencil computation algorithm, it shows that by augmenting the number of dimensions of two, the computation is faster with **loop tiling** optimization than having a lower number of dimensions.

In future work, would be interesting to try to implement what is the maximum number of dimension augmentation that offers a lower computation timing.

REFERENCES

[1] A. Taflove. Computational electrodynamics: The finite-difference time-domain method. 1995.
[2] G. Smith. Numerical Solution of Partial Differential Equations: Finite Difference Methods. Oxford University Press, 2004.
[3] J. Cong, M. Huang, and Y. Zou. Accelerating fluid registration algorithm on multi-FPGA platforms. FPL, 2011.
[4] On the Transformation Optimization for Stencil Computation
[5] Huayou Su *, Kaifang Zhang and Songzhu Mei. On the Transformation Optimization for Stencil Computation
[6] Cruz, R.D.L.; Araya-Polo, M. Algorithm 942: Semi-Stencil. ACM Trans. Math. Softw. 2014, 40, 1–39.
[7] Youcef Barigou. Acceleration of real-life stencil codes on GPUs. Hardware Architecture [cs.AR]. 2011.
[8] Justin Holewinski, Louis-Noël Pouchet, P. Sadayappan. High-Performance Code Generation for Stencil Computations on GPU Architectures.
[9] Multi-core
[10] OpenMP
[11] Loop interchange
[12] Options That Control Optimization
[13] João M.P.Cardoso, José Gabriel F. Coutinho, Pedro C. Diniz. Chapter 5 - Source code transformations and optimizations. Embedded Computing for High Performance
[14] omp_get_wtime() function
[15] OpenMP API 5 Specification. OMP Parallel.
[16] OpenMP API 5 Specification. OMP For.
[17] OpenMP API 5 Specification. Collapse.