

A Stencil computation algorithm optimization

Ivan Buccella e Matteo Maiorano
Department of Computer Science
University of Salerno
Italy

Abstract—The ReLeaSE (Reinforcement Learning for Structural Evolution) method is a new computational strategy for de novo design of molecules with desired properties. This strategy integrates two deep neural networks, one generative and one predictive, which are trained separately but used jointly to generate new targeted chemical libraries. ReLeaSE uses a simple representation of molecules only through SMILES (simplified molecular-input line-entry system) strings. The generative models are trained with a stack-augmented memory network to produce chemically feasible SMILES strings, and predictive models are derived to predict the desired properties of the generated de novo compounds. In the first stage of the method, the generative and predictive models are trained separately with a supervised learning algorithm. In the second stage, both models are jointly trained with the RL approach to guide the generation of new chemical structures toward those with the desired physical and/or biological properties. The ReLeaSE method was used to design chemical libraries with a bias toward structural complexity or toward compounds with maximal, minimal or specific physical properties, such as melting point or hydrophobicity, or toward compounds with inhibitory activity toward Janus protein kinase 2. Based on these explanations, the main work focuses on the execution of the method by training a recurrent neural network to predict logP from SMILES; in addition, several optimizations to the code are applied in order to create a platform-independent execution using modern technologies like Docker and Docker Compose.

I. INTRODUCTION

The World Economic Forum has called the combination of big data and artificial intelligence (AI) the fourth industrial revolution, capable of radically transforming science. AI is revolutionizing several fields of medicine, such as radiology, pathology and other specialties, and has begun to be used in drug discovery through deep learning (DL) technologies. These technologies are being applied in different areas, such as molecular docking, transcriptomics, understanding reaction mechanisms, and molecular energy prediction. In drug discovery, a crucial step is to formulate a well-reasoned hypothesis on how to synthesize new compounds or select them from available chemical libraries based on structure-activity relationship (SAR) data. Automated design approaches to create compounds with specific properties have been the subject of research for the past 15 years. Although there are many synthetically feasible chemicals that could be considered as possible drug-like molecules, their huge number makes it impossible to systematically examine all of them and verify them through the construction and evaluation of each individual compound. In this document we present an execution of a novel method to generate de novo chemical

compounds with desired physical, chemical and/or bioactive properties based on deep reinforcement learning (RL). RL, a subfield of artificial intelligence, is used to solve dynamic decision problems and involves analyzing possible actions, estimating the statistical relationship between actions and their possible outcomes, and determining a treatment regime that seeks to achieve the most desirable outcome. The integration of RL and neural networks was developed in the 1990s, but with the recent advancement of deep learning, which benefits from big data, new and powerful algorithmic approaches are emerging. RL, especially when combined with deep neural networks, is currently experiencing a renaissance and has recently enabled superior human performance. The execution of the method, called ReLeaSE (Reinforcement Learning for Structural Evolution), applies to the problem of designing chemical libraries with desired properties and proves to be a viable solution to this problem. This method is able to predict different properties like logP, Tm, and pIC50. In the field of machine learning, the partition coefficient (logP) can be used as a property of a chemical compound to predict its toxicity or biological activity. For example, in some reinforcement learning applications, logP can be used as a feature to help the model predict the response of the biological system to a given chemical compound. In this way, the ReLeaSE model can learn to select the most promising molecules for further study or new drug development. For example, the model could be trained to select molecules with a high logP because they have a higher probability of being biologically active. Conversely, the model could be trained to exclude molecules with too high a logP because they might be too toxic. In addition, logP can be used to predict the distribution of the chemical compound in the body and thus to determine its bioavailability. For example, a compound with a high logP might have higher bioavailability because it has a greater tendency to accumulate in lipid membranes and thus be absorbed by the body. Conversely, a compound with a low logP might have lower bioavailability because it has a lower tendency to accumulate in fats. In some cases, the model can be used to predict the melting point (Tm) of a substance from its chemical structure; it can be useful in identifying new molecules that may have desirable properties. Tm (melting point) is the temperature at which a solid changes to a liquid state. In chemistry and medicinal chemistry, the melting point of a substance is often used as an identifying property because each substance has a specific melting point. In other cases, the model can be used to predict the activity of a series of chemical compounds on a particular enzyme,

like the pIC50 values of the compounds which can be used as a metric to assess how accurately the model can predict the activity of each compound. The Inhibitory concentration (IC) is a measure of the efficacy of a chemical compound in inhibiting the activity of a particular biological target, such as an enzyme or protein. The pIC50 is a specific type of IC that represents the concentration of a compound that is required to inhibit the activity of the target by 50. In the following sections we will look in detail at the use of ReLeaSe.

Related work. The generative model in references (23, 37) is a “vanilla” RNN without augmented memory stack, which does not have the capacity to count and infer algorithmic patterns (34).

II. THE METHOD

The ReLeaSe method implements a deep RL approach for de novo design of novel chemical compounds with desired properties. ReLeaSe is distinguished from other similar approaches by its simple representation of molecules through the molecular string input system (SMILES) only during the generation and prediction phases of the method and the integration of these phases into a single workflow that also includes an RL module. The simplified molecular-input line-entry system (SMILES) is a specification in the form of a line notation for describing the structure of chemical species using short ASCII strings [19]. The project makes use of QSPR (Quantitative Structure-Property Relationship) a technique that uses mathematical models to predict the properties of a chemical compound based on its chemical structure. QSPR models are based on the assumption that there is a quantitative relationship between the structure of a chemical compound and its properties. To build a QSPR model, data on the structure and properties of a large number of known chemical compounds (training set) are used, which are then used to train the model. Once the model has been trained, it can be used to predict the properties of unknown chemical compounds (test set). QSPR models are very useful because they make it possible to predict the properties of a chemical compound without having to synthesize or test it experimentally. The training process consists of two stages: in the first stage the generative model is built. Since Regular RNNs such as long short-term memory (LSTM) (31) and gated recurrent unit (GRU) (32) are unable to solve the sequence prediction problems because of their inability to count, for the generative model, a special type of stack-augmented RNN (Stack-RNN) (30) has been used; that has found success in inferring algorithmic patterns. This Stack-RNN defines a new neuron or cell structure on top of the standard gated recurrent unit [20] cell (see Fig. 1 - A). It has two additional multiplicative gates referred to as the memory stack, which allow the Stack-RNN to learn meaningful long-range interdependencies. Stack memory is a differentiable structure onto and from which continuous vectors are inserted and removed. The objective of the Stack-RNN then is to learn hidden rules of forming sequences of letters that correspond to legitimate SMILES strings (SMILES strings as sentences composed of characters used in SMILES

notation). The generative model has two modes of processing sequences—training and generating. At each time step, during the training mode, the generative network takes a current prefix of the training object and predicts the probability distribution of the next character (Fig. 1 - A).

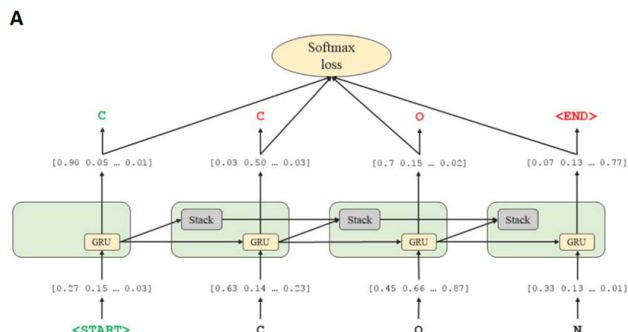


Fig. 1: A

Then, the next character is sampled from this predicted probability distribution and is compared to the ground truth. Afterward, on the basis of this comparison, the cross-entropy loss function is calculated, and parameters of the model are updated. At each time step, in generating mode, the generative network takes a prefix of already generated sequences and then, like in the training mode, predicts the probability distribution of the next character and samples it from this predicted distribution (Fig. 2 - B).

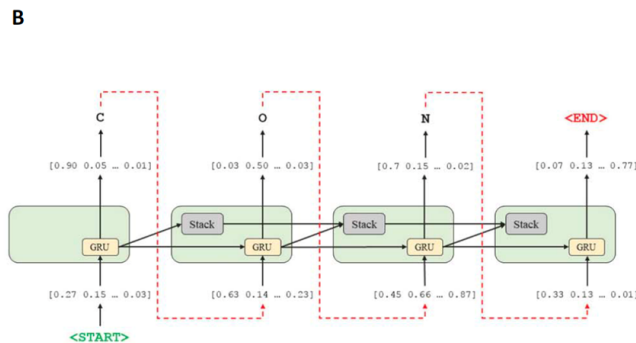


Fig. 2: B

In the generative model, the model parameters are not updated. At the second stage, both generative and predictive models are combined into one RL system, as shown in figure 3.

In this system, the generative model G (Fig. 3) plays the role of an agent, whose action space is represented by the SMILES notation alphabet, and state space is represented by all possible strings in this alphabet. The predictive model plays the role of a critic estimating the agent’s behavior by assigning a numerical reward to every generated molecule (that is, SMILES string). The predictive model P (Fig. 3)

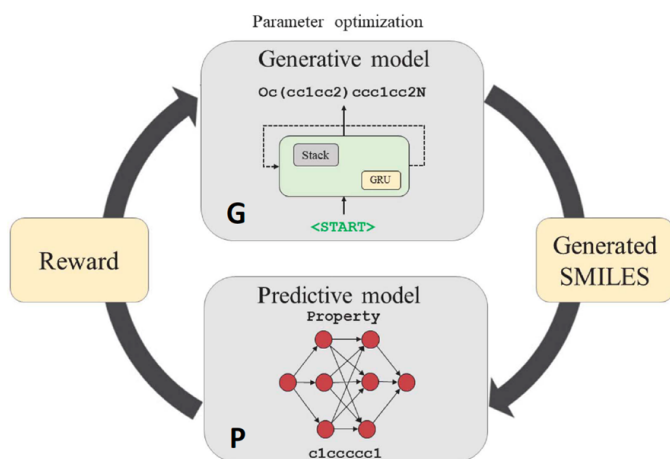


Fig. 3: generative and predictive models

is a model for estimating physical, chemical, or biological properties of molecules. This property prediction model is a deep neural network, which consists of an embedding layer, an LSTM layer, and two dense layers. This network is designed to calculate user-specified property (activity) of the molecule taking a SMILES string as an input data vector. This model takes a SMILES string as an input and provides one real number, which is an estimated property value, as an output (Fig. 4). In a practical sense, this learning step is analogous to traditional quantitative structure–activity relationships (QSAR) models. However, unlike conventional QSAR, no numerical descriptors are needed, as the model distinctly learns directly from the SMILES notation as to how to relate the comparison between SMILES strings to that between target properties. The predictor uses, as shown in the figure 4, two unidirectional LSTM layers with hidden size of 128 each, with an embedding size of 128. The dense layer is a Multi-layer Perceptron with hidden size of 128 and ReLU activation function.

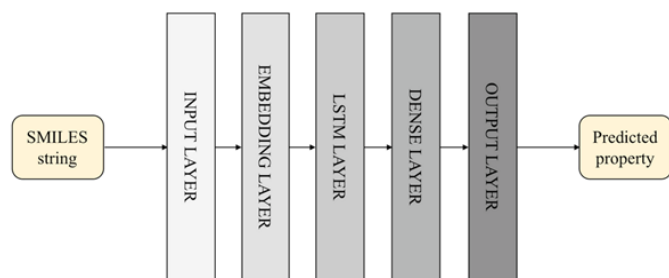


Fig. 4: SMILES strings to predicted property

The reward (Fig. 3) is a function of the numerical property calculated by the predictive model. At this stage, the generative model is trained to maximize the expected reward. The absolute majority of compounds generated de novo by the ReLeaSE method are novel structures as compared to the data sets used to train generative models, and any traditional quan-

titative structure–activity relationships model could be used to evaluate their properties. As a proof of principle, the method has been tested on three diverse types of endpoints: physical properties, biological activity, and chemical substructure bias. The use of a flexible reward function enables different library optimization strategies where one can minimize, maximize, or impose a desired range to a property of interest in the generated compound libraries.

III. ESEMPIO DI UTILIZZO DI ReLeaSE

In this experiment we will optimize parameters of pretrained generative RNN to produce molecules with values of logP within drug-like region according to Lipinsky rule. We use policy gradient algorithm with custom reward function to bias the properties of generated molecules aka Reinforcement Learning for Structural Evolution (ReLeaSE) as was proposed in Popova, M., Isayev, O., Tropsha, A. (2018). Deep reinforcement learning for de novo drug design. Science advances, 4(7), eaap7885.

The following example uses real SMILES strings, contained in a given input file (we used the ChEMBL database of drug-like compounds).

We will use stack augmented generative GRU as a generator. The model was trained to predict the next symbol from SMILES alphabet using the already generated prefix. Model was trained to minimize the cross-entropy loss between predicted symbol and ground truth symbol.

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 [10].

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 [11].

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 [12].

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) [13].

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 [16].

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 tile's execution is independent, 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) [14].

IV. THE APPLIED METHOD

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

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

```

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” [16][17] 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. To apply the directive on both loops, it has been used the “pragma omp parallel for collapse(2)” [18] 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** matrix into multiple smaller *tiles*; For this a new variable has been introduced into the code “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 *tilted* 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

```

V. EXPERIMENTAL RESULTS

Experimental setup. The execution timing is obtained by using the OpenMP primitives [15], 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 execution timings are represented 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)
- **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)
- **Parallel loops:**
 - the Algorithm 3 is: 0.4585 (s)
 - the Algorithm 4 is: 0.4760 (s)
 - the Algorithm 5 is: 0.7419 (s)

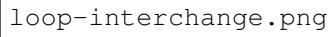
A placeholder for a figure showing loop interchange computation timings.

Fig. 5: Loop interchange computation timings

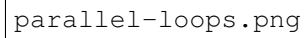
A placeholder for a figure showing parallel loops computation timings.

Fig. 7: 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)

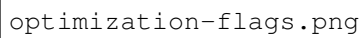
A placeholder for a figure showing optimization flags computation timings.

Fig. 6: Optimization flags computation timings

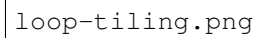
A placeholder for a figure showing loop tiling computation timings.

Fig. 8: 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)

dimension-augmentation.png

Fig. 9: Dimension augmentation computation timings

VI. CONCLUSIONS

This article presents a number of different approaches to optimize the above-detailed Stencil computation's 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 to use to compile 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 the table I is shown a summary with the best running time, relative and absolute speedups for every code optimization.

In future work, it 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.

Implementation	Running time	Rel. speedup	Abs. speedup
Starting algorithm	5.0533	1.00	1.00
+ Loop interchange	5.0533	1.00	1.00
+ Optimization flags	0.5904	8.55	8.55
+ Parallel loops	0.4585	1.28	11.02
+ Loop tiling	0.4422	1.03	11.42
Dimension Augmentation	0.2645	1.67	19.10

Table I: Summary

- [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] Lipinski's rule of five
- [9] Justin Holewinski, Louis-Noël Pouchet, P. Sadayappan. High-Performance Code Generation for Stencil Computations on GPU Architectures.
- [10] Multi-core
- [11] OpenMP
- [12] Loop interchange
- [13] Options That Control Optimization
- [14] João M.P.Cardoso, José Gabriel F. Coutinho, Pedro C. Diniz. Chapter 5 - Source code transformations and optimizations. Embedded Computing for High Performance
- [15] `omp_get_wtime()` function
- [16] OpenMP API 5 Specification. OMP Parallel.
- [17] OpenMP API 5 Specification. OMP For.
- [18] OpenMP API 5 Specification. Collapse.
- [19] Simplified molecular-input line-entry system
- [20] J. Chung, C. Gulcehre, K. Cho, Y. Bengio, Empirical evaluation of gated recurrent neural networks on sequence modeling, (2014).