

Acceleration of real-life stencil codes on GPUs

Youcef Barigou

▶ **To cite this version:**

Youcef Barigou. Acceleration of real-life stencil codes on GPUs. Hardware Architecture [cs.AR]. 2011. dumas-00636254

HAL Id: dumas-00636254
https://dumas.ccsd.cnrs.fr/dumas-00636254

Submitted on 27 Oct 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Acceleration of real-life stencil codes on GPUs

Youcef BARIGOU

Under the supervision of:

Steven DERRIEN (University of Rennes 1)

Sanjay RAJOPADHYE (Colorado State University)

June 2, 2011

1 Introduction

Until recently, the CPU has performed most important computations, even if they are compute intensive like physics simulation, off-line movie rendering, weather forecasting, encoding video and sound files, etc. CPUs became more and more faster away as their frequency increased. However, the more their frequency increased, the more they heated, until a cooling limit was reached. Consequently, CPUs could not go faster any longer, and the only way to increase the computing power was to use parallel processing units. As a result, parallel architectures were, and are presently developed. Fortunately, most of the compute intensive applications previously mentioned are easily parallelizable and can therefore run faster on parallel architectures. However, most parallel architectures are expensive and designed for a niche market. This, until the GPU (Graphical Processing Unit) imposed itself in parallel computing. The GPU is a mainstream product with a wide distribution through the outlets of video games, thereby reducing it costs compared to a very specialized architecture. The GPU is a relatively cheap architecture, designed for parallel computing and the performance of which sometimes exceeds 20x a last generation high-end CPU.

Among these compute intensive applications, the FDTD (Finite-Difference-Time-Domain) [5] allows to solve time-dependent differential equations. This method is commonly used in electromagnetism to find solutions to Maxwell equations. Although considered as powerful and flexible, the FDTD algorithm has the disadvantage of having a huge numerical dispersion due to nested loops, which are loops inside other loops in the same program. Therefore, the execution time required for such methods is often prohibitive, particularly when the number of loop iterations is large, or when coupled with global optimization algorithms such as genetic algorithms [8].

Nevertheless, there exist some program transformations techniques, such as loop tiling, that can be applied on code, precisely on nested loops, in order to improve its execution time on parallel architectures. This is done by dividing computations into independent blocks, then assigning each block to a processor within a parallel machine. Loop tiling transformation is particularly appropriate for GPUs, because they are massively parallel and powerful. During our internship, we have focused on how to optimize the execution of a FDTD-based application developed by Rolland et al. [1] on GPUs, by applying such transformation.

In this report, Section 2 describes the context of the internship by presenting, in general, our target application and reviews the CUDA programming model and loop tiling transformation. Section 3 deals with the recent works related to FDTD on GPUs. Section 4 describes in detail the FDTD algorithm of the target application, and presents the different approaches of loop tiling transformation that can be applied on it.

2 Context of the Internship

In this section, we first describe the application developed at IETR by Rolland et al. [1], on which we have worked during the internship. Then we present what the stencil codes consist of. Then, the loop tiling transformation is depicted. Finally, we present the basics of GPU architecture and CUDA programming language.

2.1 The target application

The application developed at IETR is an optimization tool that computes and analyses a set of antennae represented according to a numerical model. The goal is to find a best antenna which evinces some interesting characteristics expressed, for example, in a form of cost function. The whole process (depicted below) of the application is a genetic algorithm [8].

The genetic algorithm can be decomposed in several steps :

1. Generating antennae : before the simulation starts, the antennae data are generated by a pre-processing module built-up with Matlab functions that enable an easy design of complex-geometric antennae.

2. Electromagnetic excitations simulation : based on the FDTD method, the simulation (implemented in C) carry out the application of electromagnetic excitations on the generated antennae. Results can be : radiation diagrams, temporal and frequential cartographies, etc.

3. Elitism : the results of the FDTD simulation are used to select the antennae that, once coupled, may produce a better set of new antennae. The selection is performed with the use of a fitness criteria and the tournament method [10].

4. Coupling : the chosen antennae are then "coupled" (or combined). The result is the new generation of antennae.

5. The steps from 2 to 4 are repeated until a break condition is satisfied. For example, when an antenna that minimizes a the cost function is found among the current generation. Or when the number of evaluated antennae is greater than a certain predefined threshold.

The Figure 1 illustrates the different steps of the genetic algorithm.

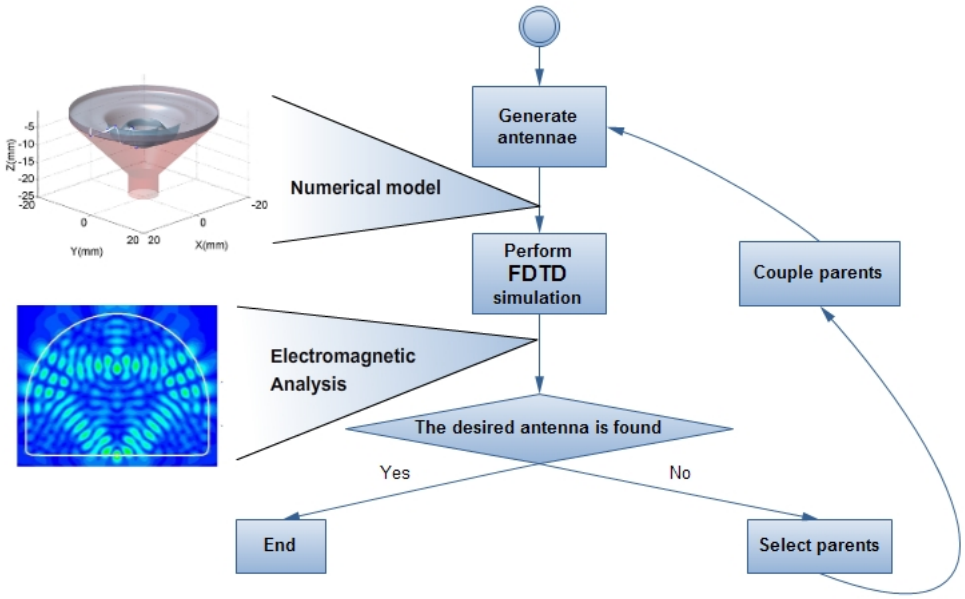


Figure 1: Global structure of the genetic algorithm

The FDTD simulator at step 2 is very compute-intensive, and a simulation may run over 1 minute for a simple antenna. Since the simulation is performed for many antenna within the genetic algorithm, the goal of our internship is to parallelize and optimize it's execution on GPU so that it runs faster. The FDTD algorithm developed at IETR belongs to the category of 2D Stencil Codes.

2.2 Stencil Codes

Stencil codes are commonly used in scientific computing, notably in solving partial differential equations, image processing, etc. Jacobi kernels and Gauss-Seidel kernels are examples of stencil codes.

Stencil codes [4] perform a sequence of sweeps that corresponds to time-iterations through the arrays they update. In each scan, some or all of the array elements are updated. The update of an element usually comprises the accesses of some values of it's neighbours. The access pattern of an element remains the same in all the iterations. The dimension of a stencil code is the dimension of the array it updates regardless of the time dimension. For instance, the 2D Jacobi stencil code (Algorithm 1) shown in Figure 2 has a 4-accesses

layout, where each element (except in the boundaries, which are not computed) is computed using its nearest neighbours in the cardinal directions. The 1D Jacobi stencil code computes one dimensional array, where each element is computed using its nearest right and left elements.

Algorithm 1 2D Jacobi stencil code

for $t = 1 \rightarrow T_{MAX}$ **do**
  $t1 \leftarrow t \ mod \ 2$
  $t2 \leftarrow (t+1) \ mod \ 2$
  for $i = 1 \rightarrow Width - 1$ **do**
    for $j = 1 \rightarrow Height - 1$ **do**
      $H\,[t1]\,[i]\,[j] \leftarrow H\,[t2]\,[i-1]\,[j] + H\,[t2]\,[i+1]\,[j] + H\,[t2]\,[i]\,[j-1] + H\,[t2]\,[i]\,[j+1]$
      $j \leftarrow j + 1$
    end for
    $i \leftarrow i + 1$
  end for
  $t \leftarrow t + 1$
end for

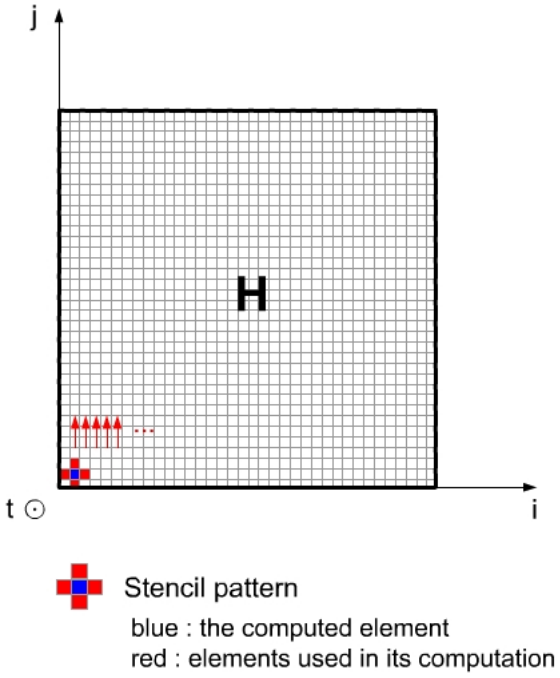


Figure 2: 2D Jacobi stencil pattern

The advantage of stencil codes is their regularity, ie. the fact that the data needed for a element computation are the same in all the time iterations gives the ability to distinguish between different independent computations. One can therefore execute them in parallel. For example, in the 2D Jacobi stencil code, the loops with I and J as indices can be executed in parallel. So stencil codes are propitious for parallel-computing.

2.3 Loop tiling transformation

Before touching on the loop tiling transformation, it is necessary to understand some basics of the polyhedral model.

2.3.1 The polyhedral model

The Polyhedral Model [12] [13] is a mathematical framework for loop nest optimization in compiler theory. The polyhedral method models operations within nested manifest loops (see Definition 1 below) as mathematical objects called polyhedra, performs affine transformations on the polyhedra, and then converts the transformed polyhedra into equivalent, but more efficient, loop nests. The polyhedral model consists of constructing an iteration domain for each statement in a given loop nest. An iteration domain is a multi-dimensional space where each dimension corresponds to a loop index. A statement is then represented by a set of instances identified by their positions in the iteration domain. In the following, we introduce the basic concepts and some mathematical notations that allow to represent a program in a polyhedral form.

Definition 1 (Loop nest) *A* loop nest *is a set of* for loops $(L_1, L_2, ..., L_n)$ *where* $n \geq 1$ *and each loop* L_i *confines directly* L_{i+1}. L_n *may contain another* for *loop.*

Definition 2 (SCoP) *A* SCoP *(for Static Control Part) is a program that can be represented using the polyhedral model. A SCoP is a loop nest with the following characteristics :*

- *No* while *loop.*
- *Loop bounds, conditions and array access functions must be* ***affine*** *functions of the loop indices and program parameters.*
- *The loop step must be equal to 1 (unitary).*
- *The iteration domain (see below) must be convex.*

Even if a program is not a SCoP, it is sometimes possible to modify it so that it becomes a SCoP. For example, if the program exhibit a non-convex iteration domain, it can be separated into two equivalent SCoPs (by splitting the non-convex polyhedron into two convex ones). Stencil codes are SCoPs in general, we can therefore construct their polyhedral models in order to apply program transformations (such as loop tiling) on them.

Algorithm 2 below is a SCoP :

Algorithm 2 Example of SCoP

```
for i = 1 → n do
  r[i] ← 0 {R}
  for j = 1 → n do
    s[i][j] ← s[max(1, i − 1)][j] + a[i][j] ∗ x[j] {S}
    j ← j + 1
  end for
  i ← i + 1
end for
```

Here, n is a parameter which is not modified during the execution of the loop nest. But since it is known at compile time, the loops are said to be *parametric.*

Definition 3 (Statement Instance) *A* statement instance *is* ***one*** *execution of a statement. So a statement which is inside a loop may have many instances, whereas as statement outside a loop has only one instance. Each instance can be referred through it's out loop counters value.*

Definition 4 (Iteration Vector) *The* iteration vector $\vec{X}_I$ *of a statement* I*, is the ordered list of loop counters (from the outermost to the innermost) that encompasses* I*. For instance, the iteration vectors of* R *and* S *in Algorithm 2 are :*

$$\vec{X}_R = (i) \ ; \ \vec{X}_S = \begin{pmatrix} i \\ j \end{pmatrix}.$$

We note $\vec{x}_I$ *an occurrence of the iteration vector* $\vec{X}_I$*, which is exactly an instance of* I*. For example,* $\vec{x}_S \begin{pmatrix} 1 \\ 2 \end{pmatrix}$ *is an instance of the statement* S *in Algorithm 2, and an occurrence of its iteration vector that is* $\vec{X}_S = \begin{pmatrix} i \\ j \end{pmatrix}$.

Definition 5 (Iteration Domain) *The* iteration domain D_I *of a statement* I*, is the set of all possible values of its iteration vector, which is equivalent to the set of all possible instances of* I*. A point in the iteration domain represents a* ***unique*** *instance of* I*. For example, the iteration domains of the instructions* R *and* S *in Algorithm 2 are :*
$D_R = \{i \in \mathbb{Z} | 1 \leq i \leq n\}$ *;* $D_S = \{(i,j) \in \mathbb{Z}^2 | 1 \leq i \leq n, 1 \leq j \leq n\}$

If we consider the statement S, its iteration domain is a part of $\mathbb{Z}^2$ which is called the *iteration space*.

Here the iteration domain is expressed as a set of constraints. When these constraints are affine and depend only on the outer loop counters and some parameters, the set of constraints defines a polyhedron. This set of constraints can be expressed in a matrix representation. For instance, the matrix representations of D_S and D_R when $n = 3$ are :

$$D_R : \begin{bmatrix} 1 & 0 & -1 \\ -1 & 0 & 3 \end{bmatrix} \begin{pmatrix} i \\ n \\ 1 \end{pmatrix} \geq \vec{0} \ ; \ D_S : \begin{bmatrix} 1 & 0 & 0 & -1 \\ -1 & 0 & 0 & 3 \\ 0 & 1 & 0 & -1 \\ 0 & -1 & 0 & 3 \end{bmatrix} \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix} \geq \vec{0}$$

Figure 3 illustrates the iteration domain of S (D_S) with $n = 3$.

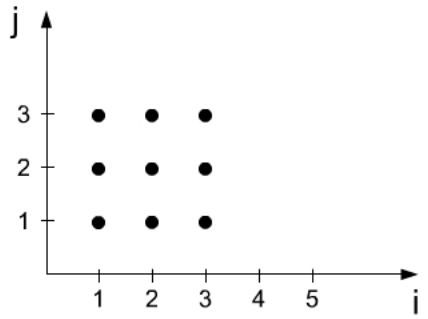


Figure 3: Iteration domain of S (D_S) with $n = 3$

Remark 1 *The iteration domain of a stencil code is equivalent to the set of array points it updates* $*$*(cartesian product) the set of time steps, ie. each iteration point corresponds to a unique array element in a unique time-step and vice versa.*

Definition 6 (Instances dependency) *Let $\vec{x}_I$ and $\vec{x'}_I$ be two instances of a statement I. $\vec{x'}_I$ depends on $\vec{x}_I$ if $\vec{x'}_I$ uses a value computed by $\vec{x}_I$. An instance dependency can be represented in the iteration domain by an arrow that goes, in this case, from $\vec{x'}_I$ to $\vec{x}_I$.*

Figure 4 illustrates the instance dependencies of S.

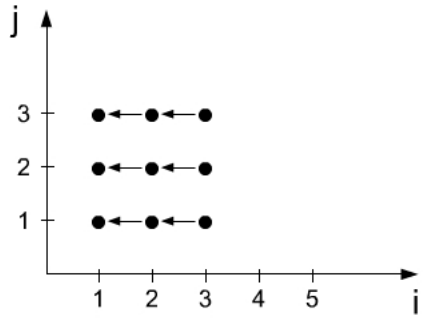


Figure 4: Instances dependency of S with $n = 3$

A schedule of a program is a function which associates a logical date (a time stamp) to each instance of each statement in terms of its iteration vector. A transformation in the polyhedral model is thus a schedule that modifies the order of execution of the different statement instances, consequently, it modifies the iteration domains of the corresponding statement. But not all the schedules enforce the data dependence of the program. So one have to perform an exact dependence analysis to build the set of all possible schedules. The polyhedral model allows to model the different legal schedules in only one convex space that represents all the distinct possible ways to legally reschedule the program, using arbitrarily complex sequences of transformations.

2.3.2 Loop tiling transformation

Loop tiling transformation (or simply tiling) consists of partitioning an iteration domain into many regular and uniform tiles. 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 (ie. being executed at the same moment).

According to Remark 1, there is a direct correspondence between the iteration domain of a stencil code and its array (the array it updates) expanded in the time-dimension. Thus, we can illustrate the tiling of a stencil code directly on array. For instance, Figure 5 shows an example of tiling on a 2D Jacobi stencil code. To simplify, we suppose that the tiles contains only one time-step. In each tile, the elements inside the green rectangle are computed and used in other computations, whereas the elements in the halos are only used (they are already computed by the neighbouring tiles in the previous time-step). Thus, the tiles in a same time-iteration can be executed in parallel.

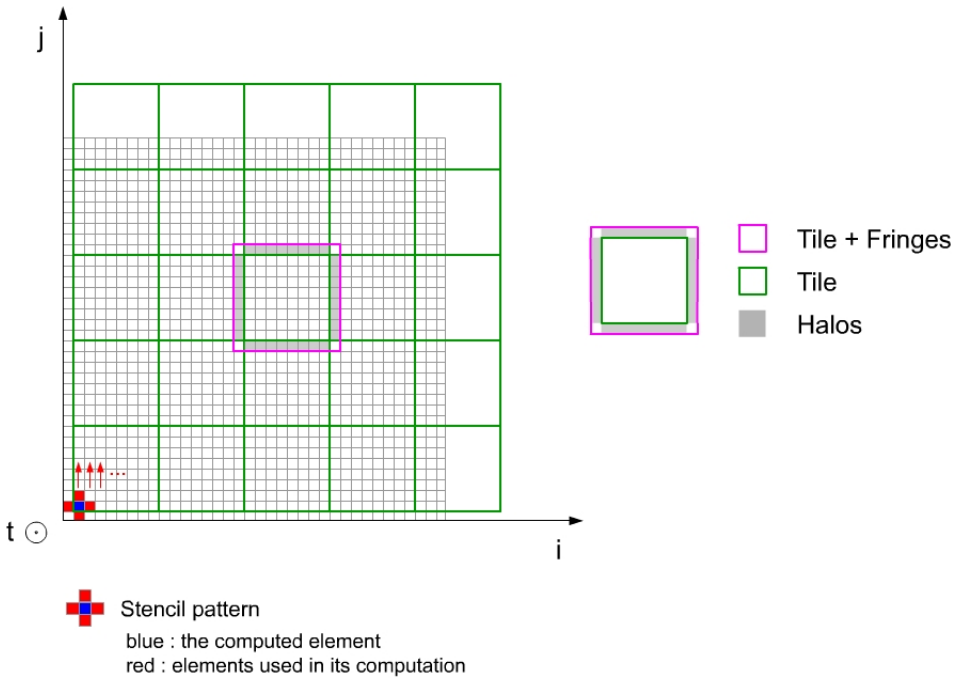


Figure 5: Tiled 2D Jacobi stencil code

The tile size is equal to the number of maximum iteration points it may contain, and the tile shape may be rectangular, parallelepiped, triangular, even a trapeze, etc. The question now is how to determine the size and shape of the tiles (which are the main features to determine) for a optimal and legal tiling transformation ?

Tiling is usually performed according to the architecture that executes the corresponding program. If we take as an example a multiprocessors architecture where each processor possesses a cache, the tile size can be chosen such that the tile's data can fit in the cache, which allow to assign one(or many) tile(s) to one processor. Hierarchical tiling is also conceivable if multiple memory levels are used, this by defining each tiling level so that its extent fits in the corresponding memory level.

Remark 2 *It is very important to precise the amount of data used per tile before deciding which tile size to choose. These amounts are:*

- *Input data: raw data needed before the tile execution starts.*
- *Intermediate results: results to keep in memory for further computations inside the tile.*
- *Output data: can be intermediate results to share with other tiles (called* ***tile footprint****), or final results. In general, the output data are stored in the input data memory locations, because in the case of stencil codes, the later are not needed any more after having been used.*

All these data have to fit in the multiprocessor's shared memory.

Since stencil codes are, in general, compute-intensive and may deal with huge grids, one needs to tile them in order to: do parallel execution, and capitalize on data locality by improving the data reuse within tiles (intra-tile communication) which reduces the communication between tiles (extra-tile communication). It can be done by using either small or large tiles:

- With small tiles : Since tiles are more when they are smaller, there may be more tiles likely to be executed in parallel. Which promote the parallelism. Using small tiles increases parallelism, but at the same time reduces data reuse because less elements are processed per tile. Consequently, small tiles increase extra-tile communication.

- With large tiles : Using larger tiles has the advantage of providing better data reuse, but reduces the parallelism.

So there is a trade-off between parallelism and data locality. Tiles must be not very small which impairs data locality, and not very large which impairs parallelism. Figure 6 shows a tiled 2D Jacobi stencil code with smaller tiles compared to the example in Figure 5. We suppose also in this example that the tiles performs only one time-step.

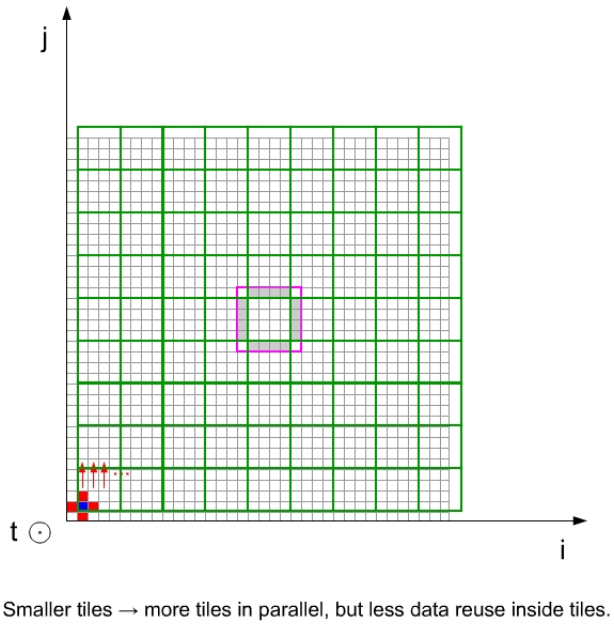


Figure 6: Tiled 2D Jacobi stencil code, with smaller tiles

We have seen that defining an adequate tile size is crucial for the performance of tiling. Now, we are going to show the importance of the tile shape.

Definition 7 (Tile dependency) *A tile T_2 depends on a tile T_1 if there exist an instance dependency between two iteration points $\vec{x}_I$ and $\vec{x'}_I$ such that $\vec{x}_I \in T_1$ and $\vec{x'}_I \in T_2$ and $\vec{x'}_I$ depends on $\vec{x}_I$. A tile dependency can be represented in the iteration domain by an arrow that goes, in this case, from T_2 to T_1.*

2.3.3 Tiling Legality

A tiling to be legal must not contain a tile dependency cycles. In other terms, a legal tiling must not contain **conflicting tiles**. Two tiles T_1 and T_n ($n \in N$) are in conflict iff T_n depends on T_1 and there exist a sequence of tiles $T_2, ..., T_{n-1}$ such that: T_1 depends on T_2, T_2 depends on T_3, ..., and T_{n-1} depends on T_n. Figure 7 shows an example of two conflicting tiles for a 1D Jacobi stencil code.

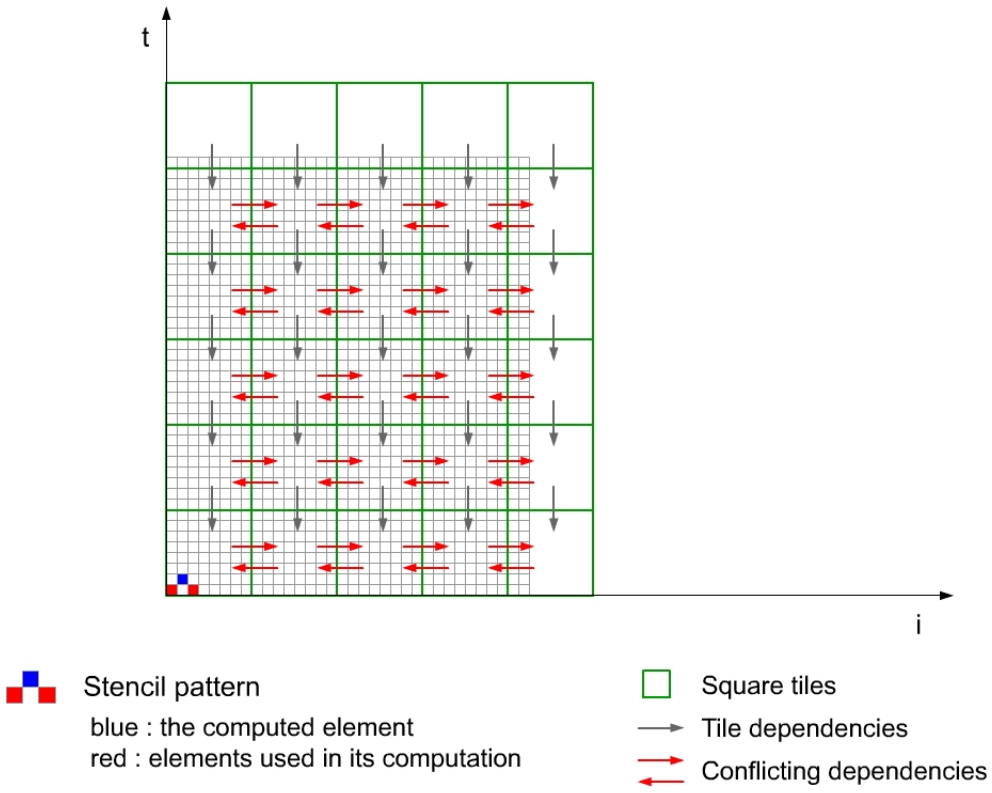


Figure 7: Illegal tiling because of conflicting square tiles (dependency cycles)

The tile shape must avoid conflicts. For example, if we use oblique tiles instead of square ones in the same example of Figure 7, then the tile conflicts are avoided by breaking the dependency cycles. This is illustrated in Figure 8.

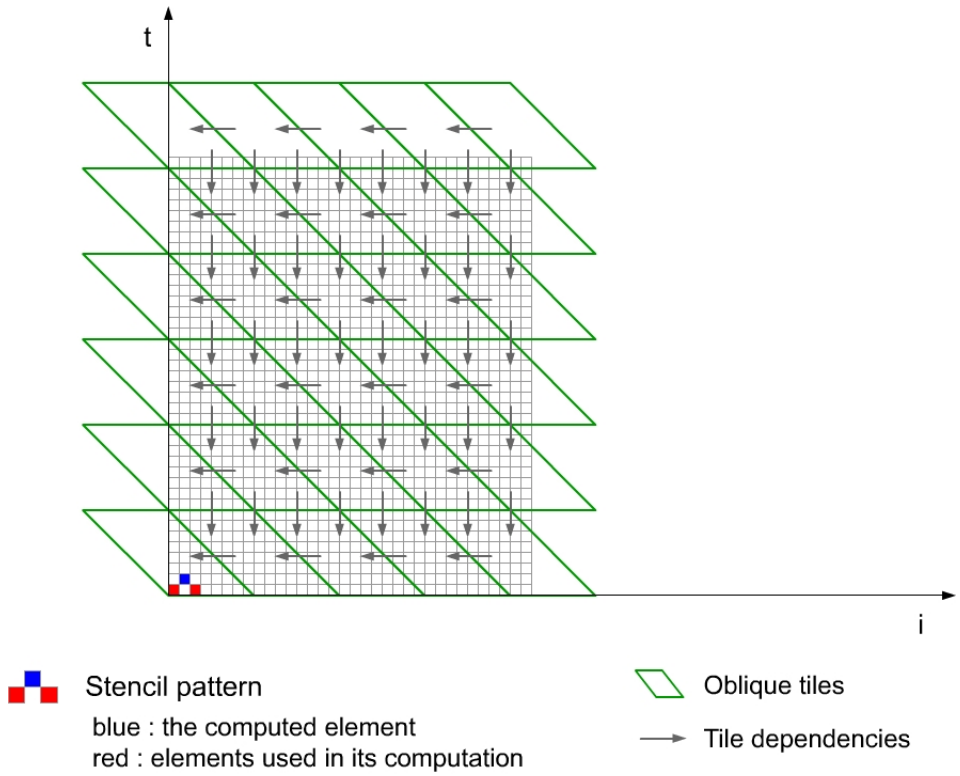


Figure 8: Conflicts solved by using a different tile shape (oblique tiles)

Therefore, we have to pay attention to the conflicts that can be generated when choosing a tile shape. We have seen previously that polyhedral model allows to model the different legal transformations. But in the remainder of this report, we show that our target application does not have a complicated stencil pattern, and its therefore easy for us to eliminate conflicts only by using basic tile shapes. Thus, we prefer not to detail polyhedral tools which are hard to understand and long to explain in this report. However, the reader can find more details in [2] and [7].

Generating tiled code involves the creation of a new external loop for each existing loop, which is called **Strip-Mining**. The newly created loops are used for enumerating tiles and have therefore to be permuted. In addition, the prior loop bounds must to be redefined so that they iterate inside tiles. For example, Algorithm 3 represents the SCoP of Algorithm 2 after tiling.

2.4 GPU and GPGPU

The Graphical Processing Unit, or GPU (as depicted in Figure 9) is a set of multiprocessors (or streaming multiprocessors (SM), or SIMD Cores), each with it's own stream processors (or ALUs), shared memory (SMEM, is a user-managed cache), and register file. The shared memory has a very low latency (almost as fast as registers) but a restricted capacity and small bus. The registre file has in general the same size as shared memory. The stream processors are fully capable of executing integer and single precision floating point arithmetic, with additional cores used for double-precision. All multiprocessors have access to global device memory, which is not cached by the hardware, and hence much slower than shared memory. The global Memory has large capacity and large bus. Register file and shared memory can be accessed only by the stream processors of the owner multiprocessor.

Algorithm 3 SCoP of Algorithm 2 after tiling

```
for jj = 1 → n do
  for ii = 1 → n do
    for i = ii → min(n, ii + tile_dim_i − 1) do
      r[i] ← 0 {R}
      for j = jj → min(n, jj + tile_dim_j − 1) do
        s[i][j] ← s[max(1, i − 1)][j] + a[i][j] ∗ x[j] {S}
        j ← 1
      end for
      i ← 1
    end for
    ii ← tile_dim_i
  end for
  jj ← tile_dim_j
end for
```

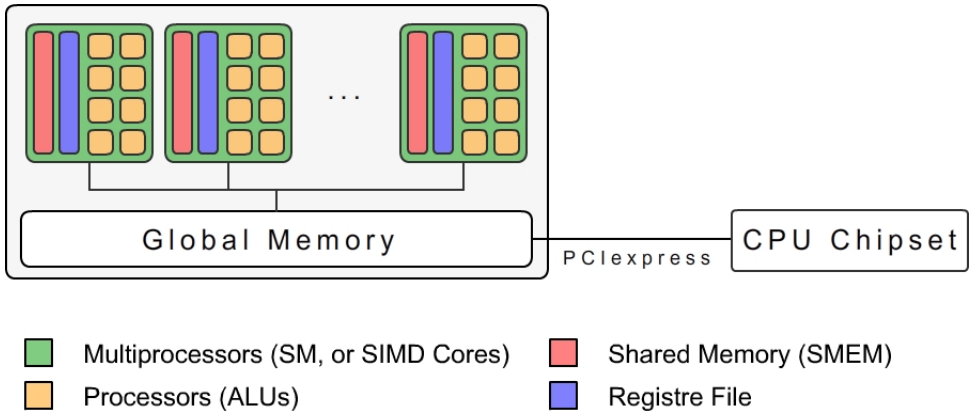


Figure 9: A high level view of GPU architecture

There are two major differences between CPU and GPU threads. First, context switching between GPU threads is essentially free, state does not have to be stored/restored because GPU resources are partitioned. Second, while CPUs execute efficiently when the number of threads per core is small (often one or two), GPUs achieve (in general) high performance when thousands of threads execute concurrently (cf. 2.5). Figure 10 show the computation power evolution in GFLOPS of GPU versus CPU, in terms of months from January 2003.

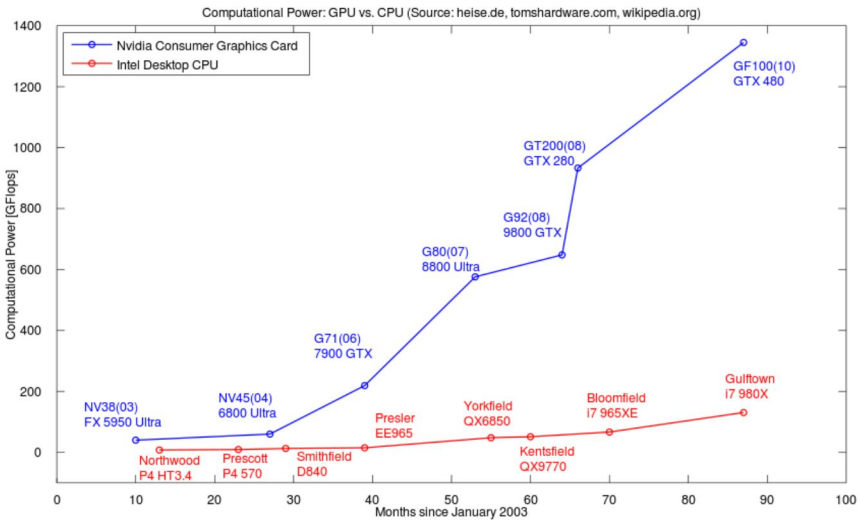


Figure 10: Computational power progression of GPU versus CPU

Besides that, the GPU architectures have the following advantages :

- GPUs are massively parallel (over 30 multiprocessors in Tesla 10-series).
- Very large global memory bandwidth (140 GiB/s).
- 32 bits floating point computations (IEEE 754, since NVIDIA G200).
- Support of high level programming APIs (NVIDIA CUDA - AMD Brook+).
- GPUs are mainstream products, thus of reduced costs.
- Theoretical energy efficiency (more than 6 GFLOPS/Watts compared to 1.7 GFLOPS/Watts of the IBM Blue Gene/Q).

The the technique of using a GPU on non-graphics data is called GPGPU (General-purpose computing on graphics processing units). The GPU typically handles graphic computations. Nevertheless, thanks to high level programming APIs such as CUDA and Brook+, the GPU can be used to accelerate applications traditionally handled by CPU and/or parallel machines.

2.5 CUDA

CUDA (Compute Unified Device Architecture) [3] allows programming GPUs in C. CUDA technology was developed by NVIDIA for GeForce graphic cards, and uses a unified driver to perform stream processing on GPUs, regardless of the hardware layer. It is done by arranging threads into threadblocks, which themselves are arranged into a grid. Each threablock is assigned to one multiprocessor, and each thread in a threadblock is assigned to one processor. Threads are grouped in units (of 32 threads in general) called Warps. Threads in the same warp progress simultaneously, and can read and write any shared memory location assigned to their threadblock. Consequently, threads within a threadblock can communicate via shared memory, or use shared memory as a user-managed cache since shared memory latency is two orders of magnitude lower than that of global memory. If a thread within a warp accesses to global memory, all the warp execution is suspended until the data are supplied, which is very costly (latency of 400 cycles in GeFore GTX-280). Meanwhile, the multiprocessor switch to another warp of the threadblock. Thus, it is recommended by NVIDIA to use as many threads/threadblock as possible in order to hide memory latency (we will discuss it later in Section 3), and also to to coalesce the data to load, because the communication between device

global memory and multiprocessors is performed by bus (of 512 octets in GeForce GTX-280). A barrier primitive __synchthreads() is provided so that all threads in a threadblock can synchronize their execution.

CUDA is in some way a Hardware Abstraction Layer of GPU. It is structured as the following:

- A Grid: composed of threadblocks and structured in 1D or 2D. Grid dimensions are stored in gridDim.x and gridDim.y.
- Threadblocks (TBs): composed of threads, and structured in 1D, 2D or 3D. The coordinates of a threadblock are retrieved through blockIdx.x and blockIdx.y. Threadblock dimensions are stored in blockDim.x, blockDim.y and blockDim.z. The number of threadblocks can be very large (up to 65535 in the GeFroce GTX-280).
- Warps: concurrent groups of simultaneously progressing threads. A warp cannot be referenced, because warps are managed by the GPU scheduler and not by the user.
- Threads: smallest processing unit. A thread can be referenced by threadIdx.x, threadIdx.y and threadIdx.z. The number of threads per threadblock is limited. For example, in the GeForce GTX-280 graphic card, the number of threads/TB is limited to 512, which implies a maximum number of warps/TB equal to 16 (= 512/32).

Figure 11 illustrates the structure of CUDA architecture.

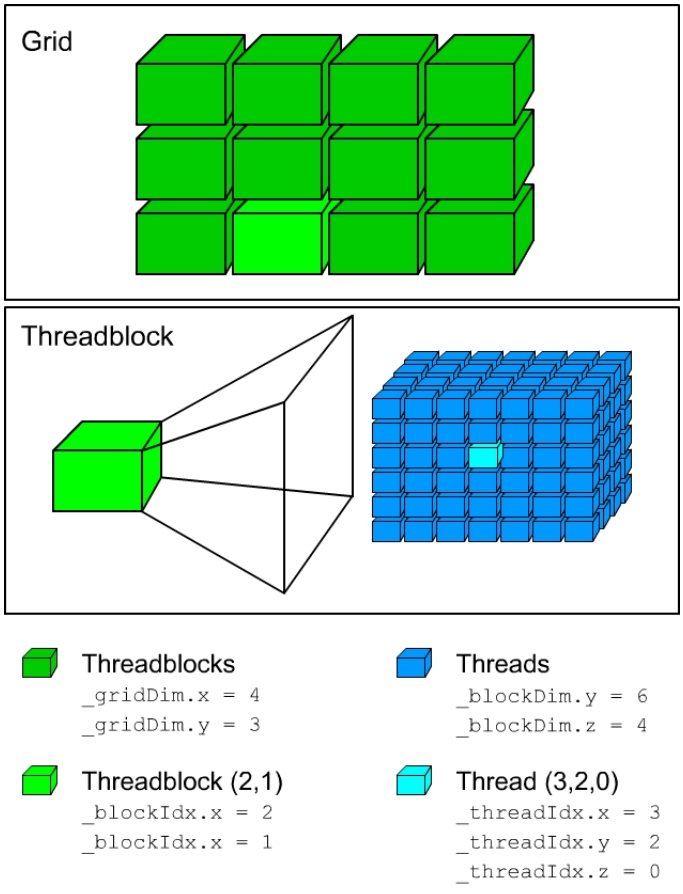


Figure 11: CUDA architecture (with arbitrary values as an example)

Simple CUDA programs have the following basic flow:

1. The kernel to execute on the CUDA device is declared first.
2. The host initializes the data to process on the CUDA device.
3. The memory needed to execute the kernel on the CUDA device is allocated on it.
4. The array is copied from the host to the memory allocated on the CUDA device.
5. The CUDA device operates on the data in the array by executing the kernel.
6. The results are copied on the host.

The CUDA program shown in Figure 12 follows this flow. It takes an array and squares each element.

```
__global__ void square_array(float *a, int N)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;   (1)
    if (idx<N) a[idx] = a[idx] * a[idx];
}

int main(void)
{
    float *a_h, *a_d;
    const int N = 10;
    size_t size = N * sizeof(float);

    a_h = (float *)malloc(size);
    for (int i=0; i<N; i++) a_h[i] = (float)i;   (2)

    cudaMalloc((void **) &a_d, size);   (3)
    cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);   (4)

    int block_size = 4;
    int n_blocks = N/block_size + (N%block_size == 0 ? 0:1);
    square_array <<< n_blocks, block_size >>> (a_d, N);   (5)

    cudaMemcpy(a_h, a_d, sizeof(float)*N, cudaMemcpyDeviceToHost);   (6)
    ...
}
```

Figure 12: Basic CUDA flow

2.6 Conclusion

As the number of tiles may be very large, the number of threadblocks can also be very large; the number of iteration points in a tile is restricted, the number of threads per threadblock is also limited (to 512 in GeForce GTX-280); tiles may be concurrent, threadblocks are concurrent; tile execution is atomic (cf. 2.3.2) which means that there is no communication between two concurrent tiles, synchronizing concurrent threadblocks is not recommended which means that concurrent threadblocks should not communicate; a tile must be assigned to a single threadblock.

We have seen in this section that the FDTD algorithm is very compute-intensive, it therefore needs to be optimized. The FDTD algorithm belongs to a particular category of codes that is Stencil Codes, which is favorable for parallel-computing. We have also depicted the tiling transformation and how GPU performs parallel computations, and concluded finally that tiling is very adequate for CUDA architecture.

3 Some related research work

In this section, we present some of the resent research works related to tiling transformation and FDTD computation on GPUs, and GPGPU in general.

In Micikevicius et al. [11], the author proposes a method to compute a 3D stencil by reading the 3D grid only once (instead of twice in a more naive approach). The two-pass approach consists simply of computing a 2d stencil through all the volume first, than computing the remaining dimension in the second pass. The trick of the one-pass approach is to store the values that were read in the second pass of the two-pass approach in registers. The data are shifted when the tile crosses the stencil. Therefore, the access redundancy decreases by half. This is the main and key idea of the article on optimizing the bandwidth use. Even if the stencils discussed in this article are 3D, we can deduce the importance of the use of registers for temporary storage (buffers), which may well compensate for the lack of shared memory (due to its small size). This can be very helpful for us, because the stencil code we handle is very demanding in terms of local memory (cf. 4).

In Rivera et al. [6], the author also deals with 3D stencils in his article, and shows that the tiling is not necessary in the case of 2D stencils. This by taking as an example the 2D Jacobi stencil, where to compute an element, we need only 4 values (see Algorithm 1). In our case, we need sometimes up to 134 values to calculate a single FDTD cell (element of the FDTD grid). In this case, tiling is essential to reduce the local memory (shared memory + registers) requirements. The author, nonetheless, offers an interesting method to overcome the memory bank conflicts. This problem occurs when two accesses to the same memory bank occur simultaneously in a tile, causing a latency. The solution proposed by the author is to apply Padding, a method of offsetting the memory access layout so that accesses in conflict are shifted and separated.

While the most previous works on GPGPU recommend the use of Thread-Level Parallelism (TLP), ie. to use as many threads per threadblock as possible simply because memory latency is hidden by executing thousands of threads concurrently; Volcov et al. [14] performed a benchmarking showing that it is better to use Instruction-Level Parallelism (ILP) within threads as much as possible, instead of TLP. That is to carry out more independent instructions inside threads, even if it involves fewer threads per threadblock (which is the lower occupancy). In our case, the fact that the FDTD algorithm is greedy regarding the local memory, limits sufficiently the number of threads per threadblock (cf. 4), the occupancy is therefore already low. Moreover, most of the calculations of a FDTD cell are independent, so it would be quite interesting for us to attribute all the calculations of one cell to a single thread, which improves ILP.

4 Tiling the FDTD code

In this section, we first present the FDTD code developed by Rolland et al. IETR [1]. Then we exhibit the similitude between the Jacobi 2D stencil pattern and the FDTD stencil pattern. Next, we apply, analyse and compare our tiling approaches on the Jacobi 2D stencil code. After that, we try to answer the following question: which approach is best to apply, not only on the Jacobi 2D stencil code, but also on the FDTD stencil code ?

4.1 Description of the FDTD stencil code

The FDTD 2D grid, shown in Figure 13, is composed of 6 regions: PML, non-PML, DFT, Boundaries, Axis-symmetry and Excitation. Each cell of these regions contains two types of variables: E-field variables and H-field variables. Each variable in each region is updated at least once in each time iteration, except: the E-field in the Axis-symmetry region and both E-field and H-field in the Boundaries; which are not updated at all (both are initialized to 0). The cells having the same spatial coordinates share a certain number of time-independent coefficients (ie. they remain constant), that carries their variables' update. The number of variables and coefficients depends on the region where the cell is. Figure 14 displays the composition in terms of variables and coefficients of each type of cell.

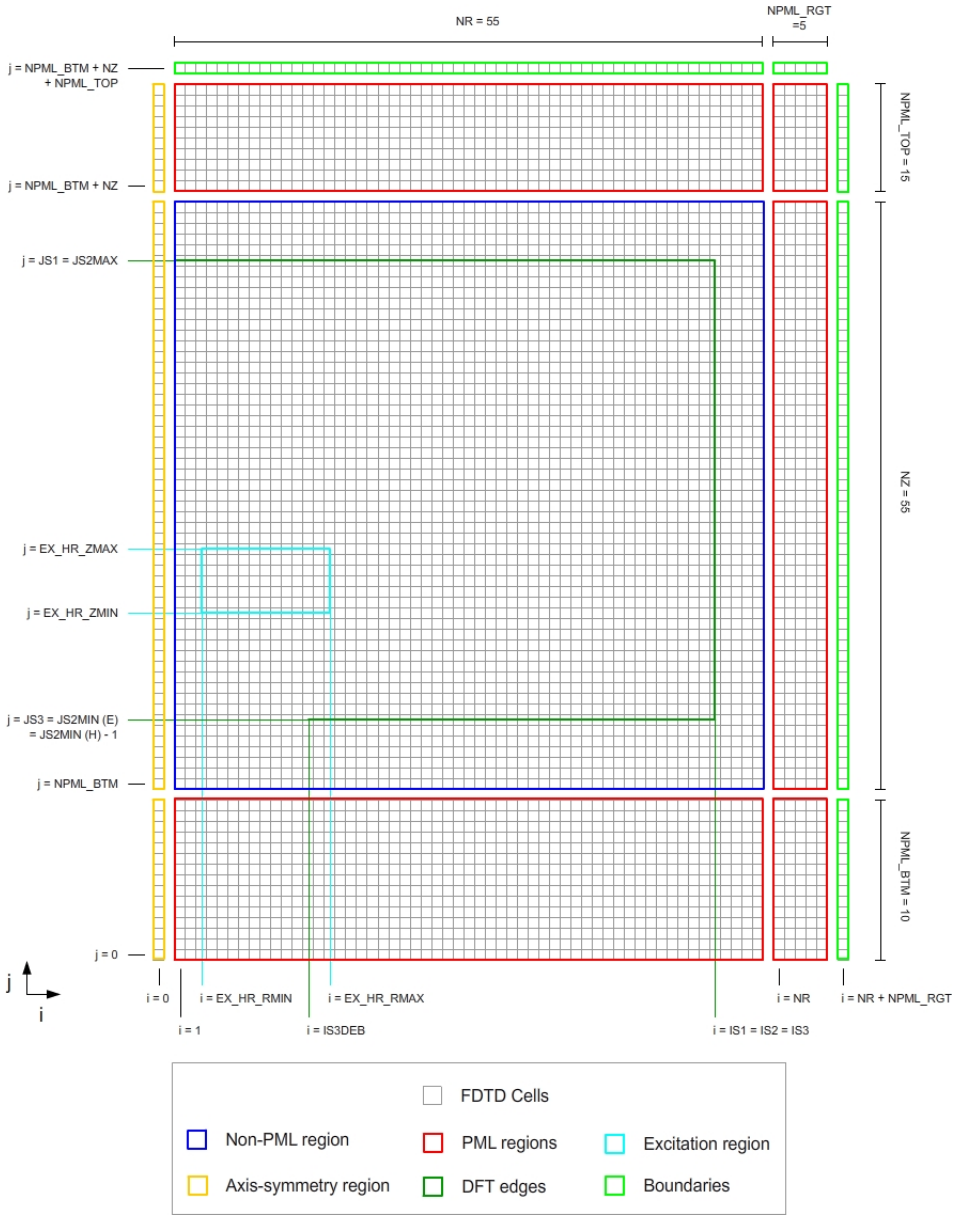


Figure 13: Composition of the FDTD 2D grid (with arbitrary values as an example)

Remark 3 *The FDTD grid height and width are on the order of few hundreds (100 to 300, even 400). But* T_{MAX} *is on the order of several thousands (5000 to 15000 in general).*

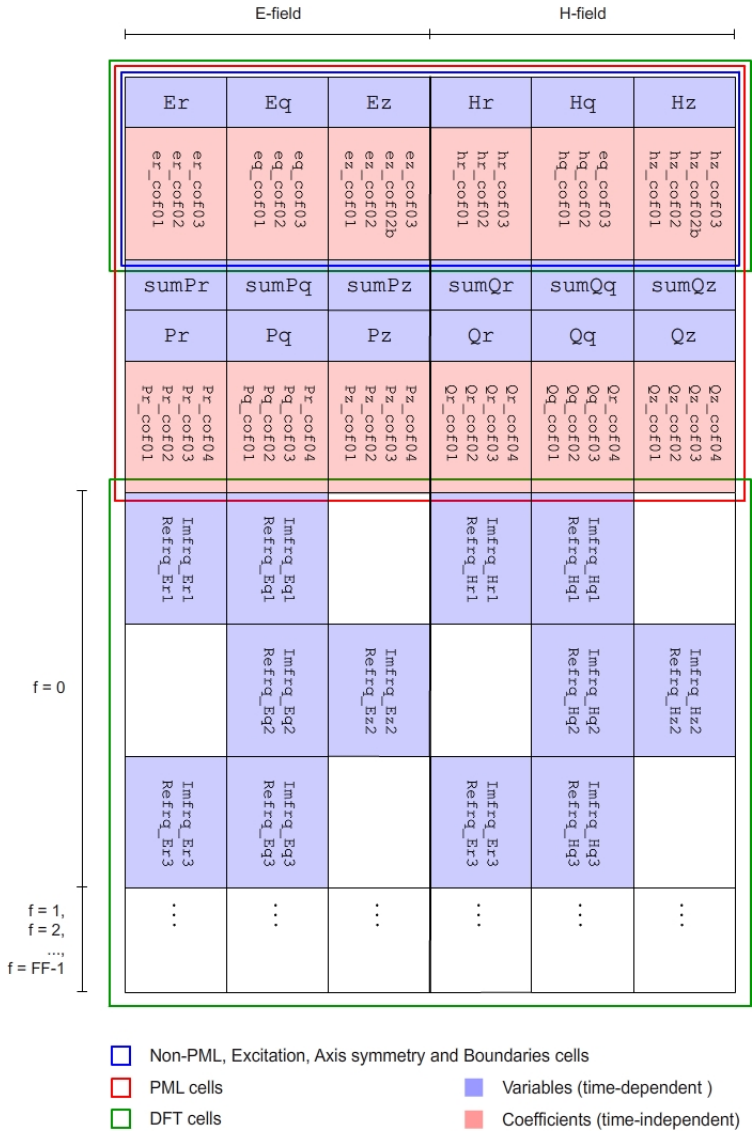


Figure 14: Composition of each type of cell

Figure 15 shows the number of variables, of coefficients and the proportion of each type of cell. FF is the number of frequencies computed in the DFT cells.

	Var.	Coef.	Proportion
Non-PML + Excitation + Axis symmetry + Boundaries	6	20	~84%
PML	18	44	~14%
DFT	6 + 24 * FF	20	~2%

Figure 15: N. of variables, n.of coefficients and proportion of each type of cell

Remark 4 *According to the intra-cell data flow graph (data flow of computations inside a cell) of the FDTD stencil code which we cannot display in this report because it is very huge, computations within cells are often independent of each other, which increases considerably the Instruction Level Parallelism inside threads.*

The stencil patterns with the inter-cells data flow (data flow between cells) in one iteration of all the regions are depicted in Figure 16.

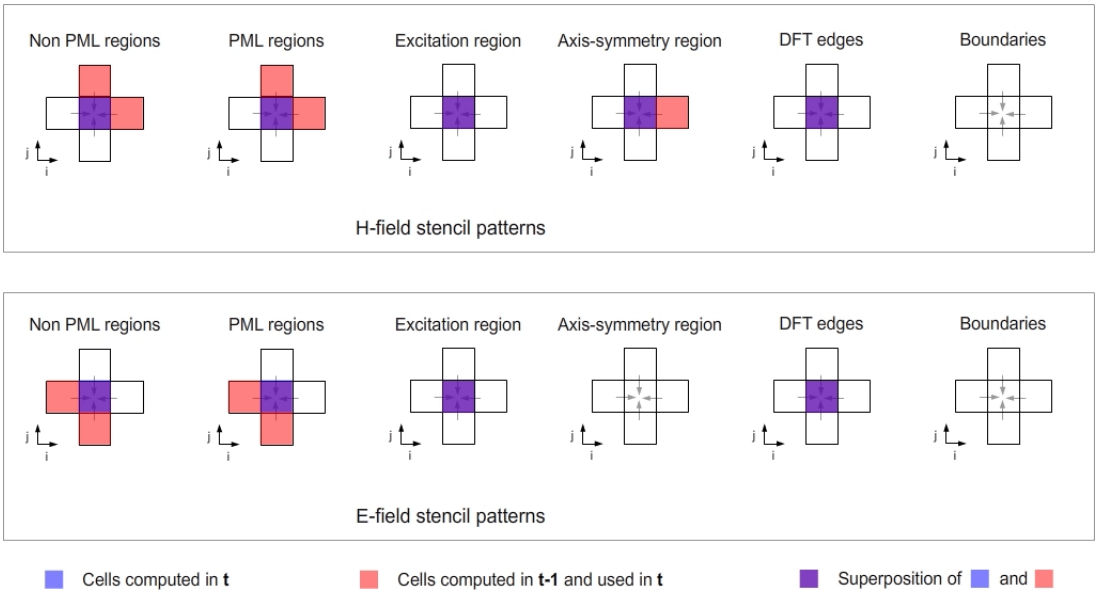


Figure 16: Stencil patterns of each region

The stencil pattern of the FDTD code is therefore the union of all the previous patterns. It is illustrated in Figure 17.

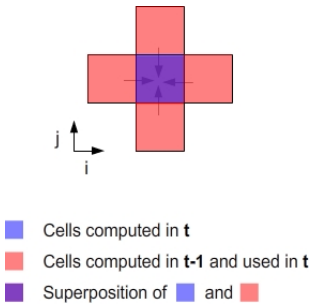


Figure 17: Stencil pattern of the FDTD code

The FDTD stencil pattern is almost the same as 2D Jacobi, the only difference is that, when the 2D Jacobi stencil code updates an element, it does not use its previous value (the value at the previous time step). This is not the case in the FDTD stencil code where, to update a cell, we use the previous values of its variables. Yet, we can just add the missing access in the 2D Jacobi stencil code, in order to obtain a pseudo 2D Jacobi stencil (Algorithm 4) code that has the same stencil pattern (Figure 18) as the FDTD code. The pseudo 2D Jacobi stencil code thus have the same dependency layout as the FDTD application. We can therefore study our different tiling approaches on the pseudo 2D Jacobi stencil code, this is because it is much complicated to deal directly with the FDTD code.

Algorithm 4 Pseudo 2D Jacobi stencil code

for $t = 1 \rightarrow T_{MAX}$ **do**
  $t1 \leftarrow t \ mod \ 2$
  $t2 \leftarrow (t+1) \ mod \ 2$
  for $i = 1 \rightarrow Width - 1$ **do**
    for $j = 1 \rightarrow Height - 1$ **do**
      $H\,[t1]\,[i]\,[j] \leftarrow H\,[t2]\,[i]\,[j] + H\,[t2]\,[i-1]\,[j] + H\,[t2]\,[i+1]\,[j] + H\,[t2]\,[i]\,[j-1] + H\,[t2]\,[i]\,[j+1]$
      $j \leftarrow j + 1$
    end for
    $i \leftarrow i + 1$
  end for
  $t \leftarrow t + 1$
end for

Remark 5 *The data at $t = 0$ (H[0][i][j]) are initialized before the stencil loops are reached.*

For more details on the 2D Jacobi stencil code, see 2.2.

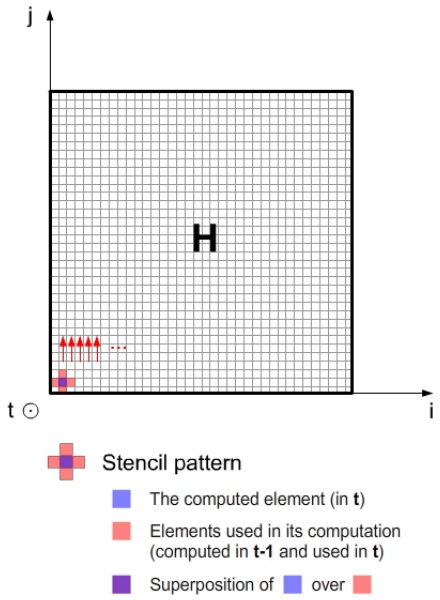


Figure 18: Pseudo 2D Jacobi stencil pattern

We experiment the different approaches using grids with dimensions on the order of those depicted in Remark 3. T_{MAX} is also chosen according to the same remark. The hardware device is GeForce GTX-280 with the following properties:

- Number of multiprocessors: 30
- Number of cores: 240
- Maximum number of threads per block: 512
- Maximum sizes of each dimension of a block: 512 x 512 x 64
- Maximum sizes of each dimension of a grid: 65535 x 65535 x 1
- Warp size: 32
- Total amount of global memory: 1 Gigabyte
- Global memory access latency: 400 cycles
- Global memory bus size: 512 bytes
- Total number of registers available per multiprocessor: 16384
- Total amount of shared memory per multiprocessor: 16384 bytes
- Shared memory access latency: 4 cycles
- Shared memory bus size: 16*4 bytes = 16 floats
- Clock rate: 1.30 GHz
- Copy and execution are concurrent

4.2 Tiling approaches

The purpose of this subsection is to determine the possible choices of tile shapes and iteration space transformations that can be used to tile on the pseudo 2D Jacobi stencil code. In the following, we present 5 approaches, in each one of them we either use a new tile shape or transform the iteration space. The approaches are presented from the most naive to the most tricky one. The tile size is not discussed in this subsection. However, we sometimes expose experimental results in some approaches, with arbitrary tile sizes so as to give an idea on the performances that can be reached.

4.2.1 Approach 1: 2D tiles

This approach consists of tiling the grid H into several 2D tiles (Figure 19), each of them performs only one time step. At the beginning of the execution of the whole application, the data of the grid H are copied from PC memory to GPU global memory in an array T_1 as shown in Figure 20. Then, the CUDA kernel is executed multiple times. At each time, the threadblocks copy the data they have to process from the global memory (ie. from T_1) into their shared memory then perform computations. Each thread inside a threablock computes a single element (or cell). After that, the threadblocks having finished their computations write their results in global memory, but in a different array T_2. At that moment, all the H grid has been computed. The pointers of T_1 and T_2 are then swapped before the next kernel execution. The whole process is repeated $T_{MAX} - 1$ times.

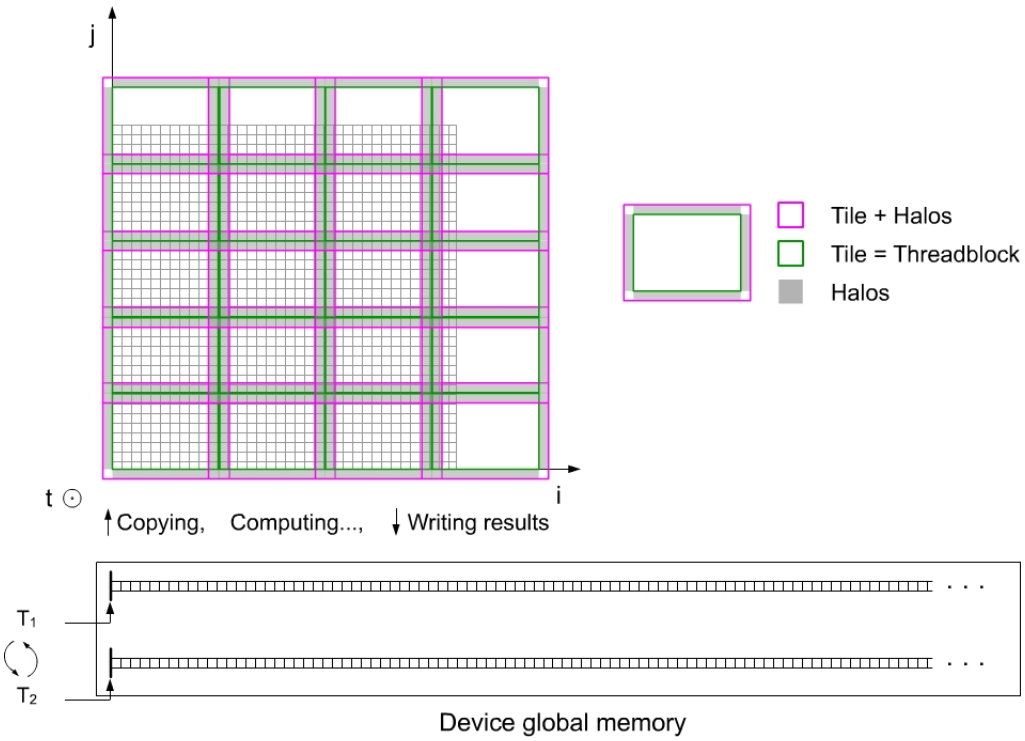


Figure 19: Tiling the pseudo 2D Jacobi stencil code using 2D tiles

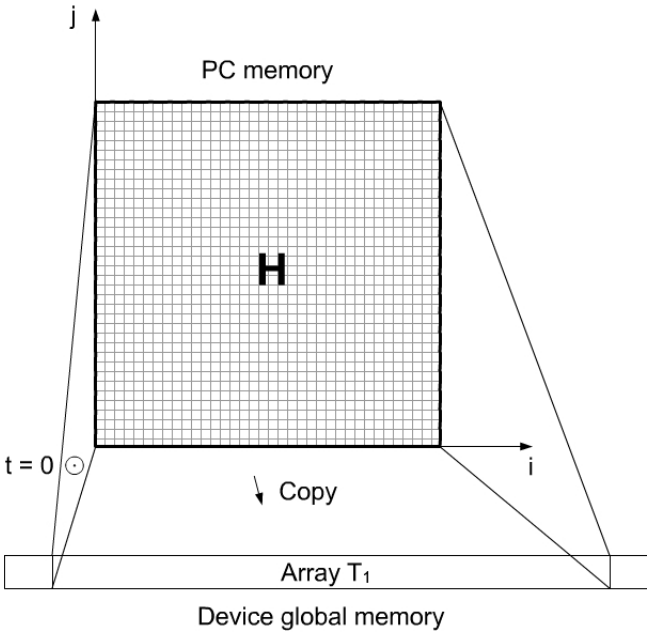


Figure 20: Copy of H from PC memory to GPU global memory

The main advantage of this approach is that it works even if H is very large, and is easy to implement in CUDA. However, the global memory is often accessed in this approach, because the threadblocks performs only one time step before they write their results in global memory. Consequently, the latency is not hidden and the memory hierarchy is badly used because of a weak temporal reuse inside threadblocks.

Experimental results with arbitrary tile size of $16 * 16 * 1$ (1 because tiles are 2D) are shown in Figure 21. We have chosen this arbitrary tile size, which is not necessary the optimal one, in order to give an idea on the performances that can be reached using this approach.

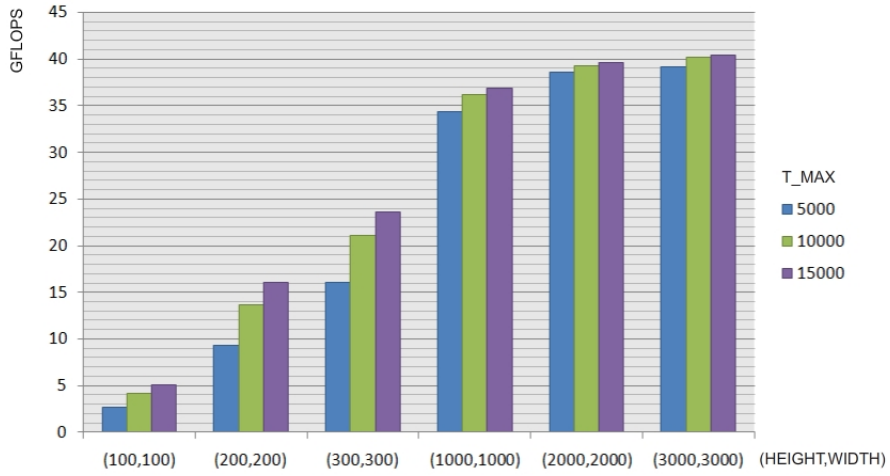


Figure 21: Experimental results of approach 1 (tile size = $16 * 16 * 1$)

Idle time is spent:

- At the beginning of the execution of the whole application (only once): when H data are copied to

global memory, so increasing $Height$ and $Width$ augments this useless time. But increasing T_{MAX} reduces it effect upon the whole execution time.

- Synchronization cost: after each iteration: the next kernel execution needs all the threadblocks to be synchronized (ie. having completed their computations). Thus, some multiprocessors may stay idle for a while. Consequently, increasing T_{MAX} augments this useless time, but increasing $Height$ and $Width$ reduces its impact.

Now, we use lower occupancy that consists of using fewer threads per threadblock in order to increase the number of operations done per thread, therefore the ILP is increased. The results in Figure 22 show the performances using the same tile size as previously $(16 * 16 * 1)$ and 32 threads, which represents 1 warp. In this case, instead that a thread computes a single element, it computes $16 * 16/32 = 8$ elements. The results are not as good as without lower occupancy, even with other tile sizes, number of threads per threadblock and memory layouts. This is due to the fact that the threads have to perform a loop to load their data and to write their results in each kernel execution, which is very costly compared to the first case where each thread loads and writes only one element. In other words, since the threads in this approach load a 2D space (because the grid is 2D) and compute only a 2D space (because the tiles are 2D), the loading and writing loops performed by each thread hide the benefit of ILP. In approach 2, lower occupancy gives better results because the tiles are pseudo 3D (see 4.2.2). We can therefore deduce that lower occupancy is useful when the tiles are of at least one dimension more than the input and output space dimensions (cf. Remark 2).

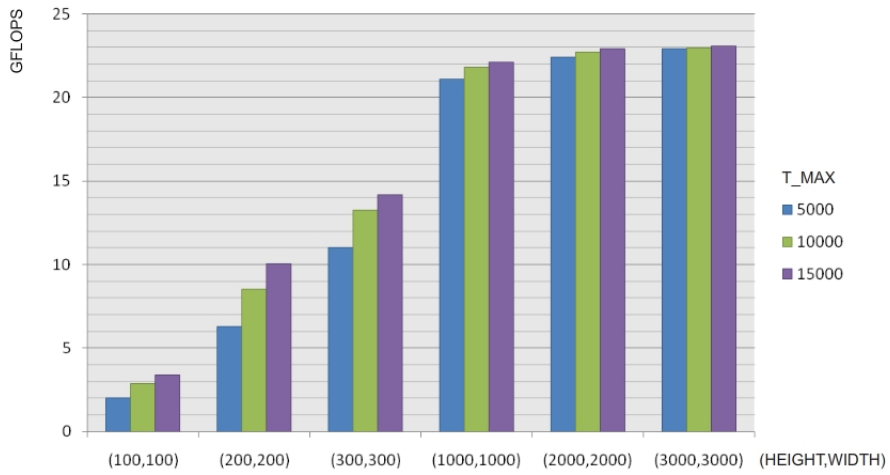


Figure 22: Experimental results of approach 1 at lower occupancy (tile size $= 16 * 16 * 1$)

This approach is naive in the sense that tiles perform only one time step before writing their results in global memory. In the next approach, we try to reuse the data already present in shared memory, and avoid the synchronization cost due to multiple kernel executions.

4.2.2 Approach 2: Synchronized threadblocks

In this approach, we try to avoid multiple kernel executions by using pseudo 3D tiles (pseudo 3D because the tiles are dependent and their execution is not atomic) and reuse the data in shared memory. The global memory is therefore less used and the latency is better hidden than in approach 1. Two main memory management layouts can be considered: row by row, or column by column (Figure 23). Because of the threadblocks perform their computations column by column (Figure 24, they have to exchange columns of data in the boundaries. Thus, the column by column memory management layout provides a better spacial locality by coalescing the data in global memory. This approach does not respect the atomicity of tile execution because tiles are dependent of each other (they have to exchange data before each time step).

Therefore, a deadlock may occurs if there are more tiles than there are multiprocessors, because all the current tiles may be waiting, directly or transitively, for some tile which may not be currently processed. Consequently, this approach works only if there are as many or less tiles as there are multiprocessors. This is also the reason why tile execution must be atomic, ie. currently executed tiles should not communicate (cf. 2.3.2). Figure 25 shows the performances of this approach with 512 threads/threadblock.

The main drawback of this method is, since the data are kept in shared memory and the number of threadblocks is limited to the number of multiprocessors, the grid H cannot be larger than approximately $400 * 400$ elements because the shared memory may not support the amount of data used in t and reused in $t + 1$. Besides that, we have to take into account hardware features (number of multiprocessors) beyond of the CUDA architecture, which is contrary to the GPGPU programming paradigm (cf. 2.5).

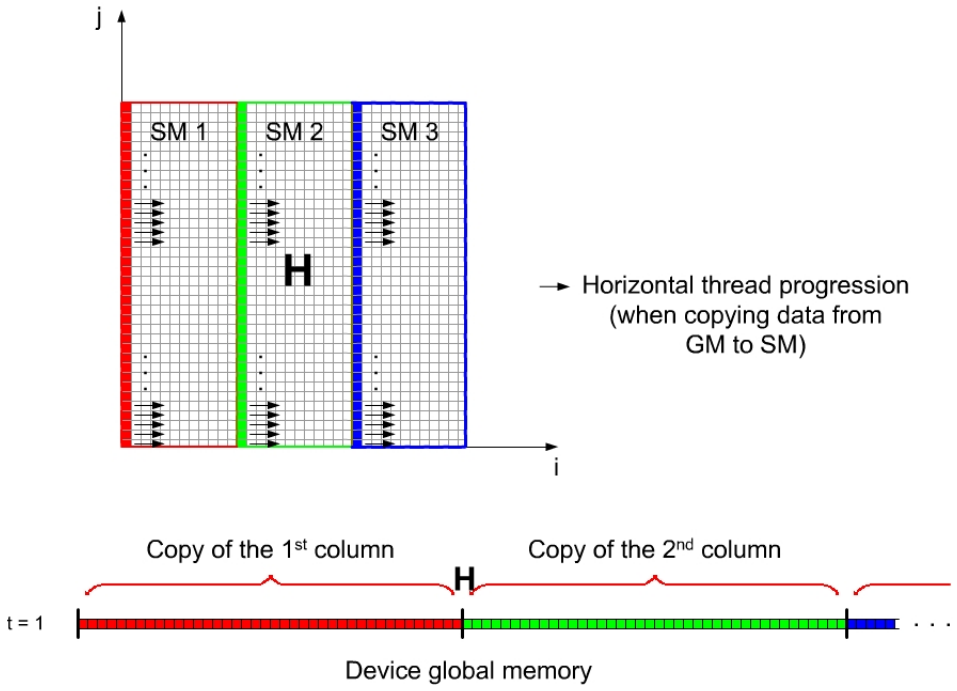


Figure 23: Column by column layout memory management layout

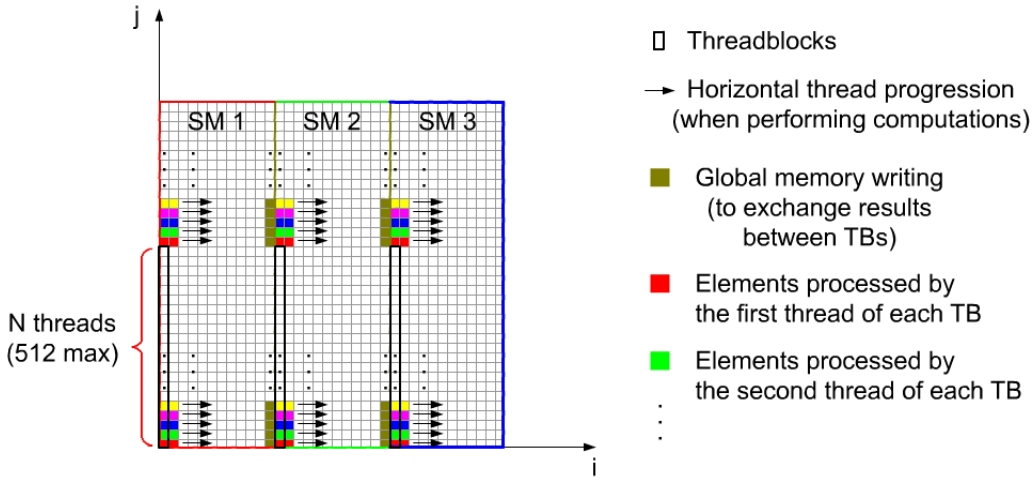


Figure 24: Computational layout

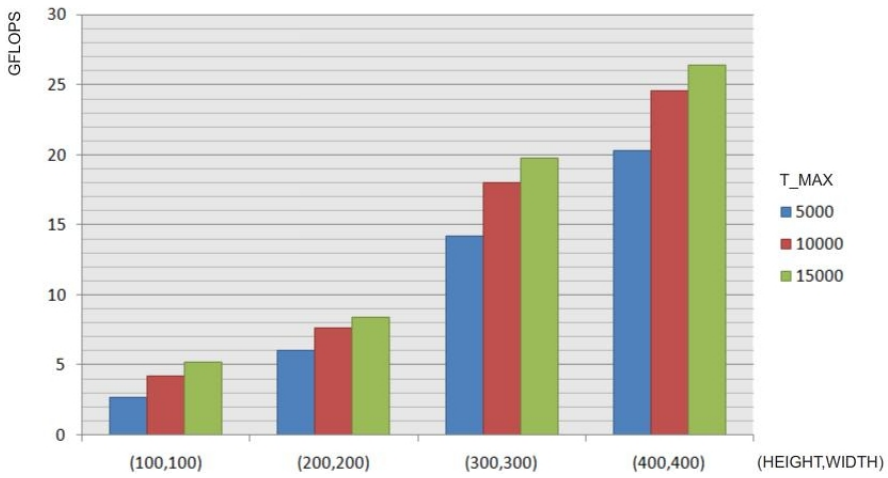


Figure 25: Experimental results of approach 2 with $N = 512$ threads/TB

At lower occupancy, this approach shows better results than approach 1. This can be done by decreasing the number of threads per threadblock. The plot in Figure 26 shows the performance results according to the number of threads per threadblock.

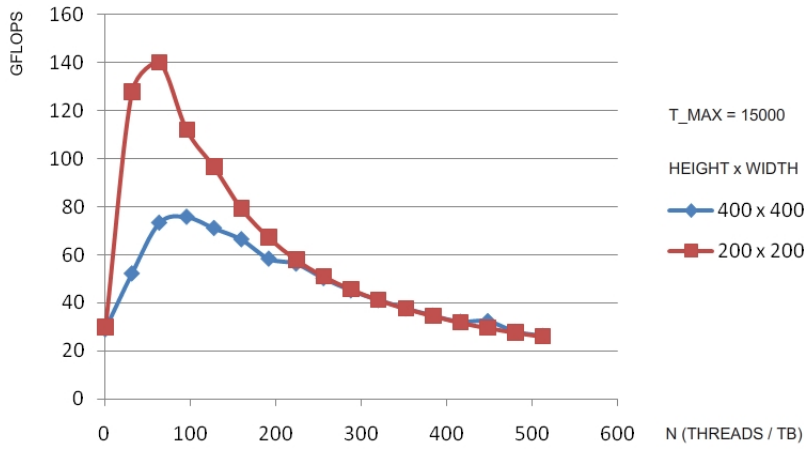


Figure 26: Performance results with different numbers of threads/threadblock

Threadblocks synchronization:

Before exchanging the data, the threadblocks have to be synchronized first. This can be done using a mutex variable located in global memory. This variable is incremented each time a threadblock finishes its computations. The incrementation is performed with the use of __atomicAdd() primitive that manages concurrent accesses. Figure 27 illustrate the threadblocks synchronization process.

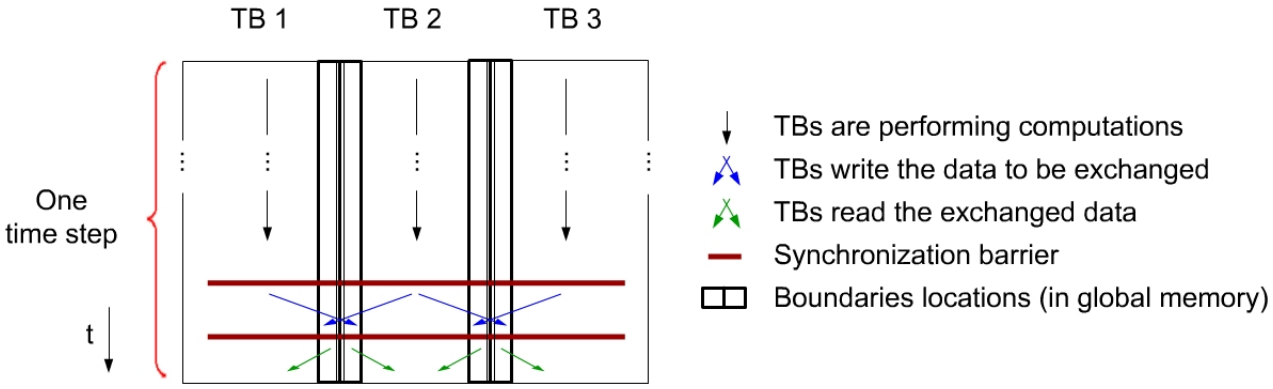


Figure 27: Spatio-temporal representation of one time step behaviour

However, even if it is very unlikely, writing operations performed before synchronization may be not completed when the GPU leaves the synchronization barriers (Figure 28). The solution is to use the costly __threadfence() that holds up the current thread until all writes to shared and global memories are visible to all other threads (including the threads of the other threadblocks). This is considered as bad CUDA programming practice and is not officially supported by NVIDIA.

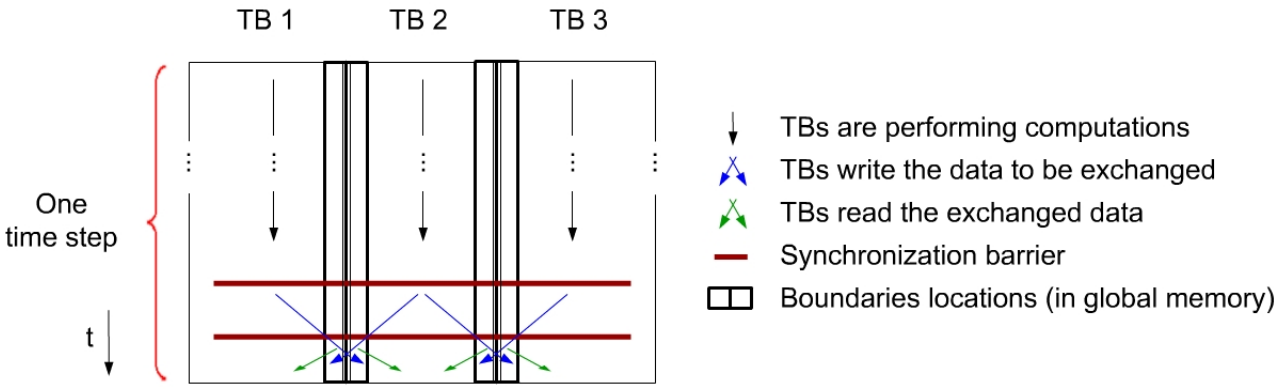


Figure 28: Writing operations not completed even after threadblocks synchronization

This approach does not work for large grids, and the number of tiles is limited to the number of multiprocessors. Besides that, the threadblocks synchronization in a single kernel execution is not recommended by NVIDIA. Thus, we have to think on how to transform the tile shape, or the iteration space in order to execute multiple time steps inside tiles atomically (ie. without synchronizing currently executing threadblocks).

4.2.3 Approach 3: Time-skewing (without time-tiling)

In this approach, we extend the tiles of approach 1 into 3D tiles that perform all the time iterations at one go. We have therefore to solve conflicting tiles other than by synchronizing threadblocks like in approach 2. The idea is to time-skew the volume (ie. skewing the volume throughout (t)) as shown in Figure 29 where we have two conflicting accesses in (i) axis. By skewing in the direction of (j) axis, more conflicts can be avoided as shown in Figure 30. 3D view of time-skewing over both (i) and (j) axes is displayed in Figure 31. Thus, smaller tiles can be used which favors parallelism. However, this approach does not scale with large values of T_{MAX}. It is proved further in the subsection.

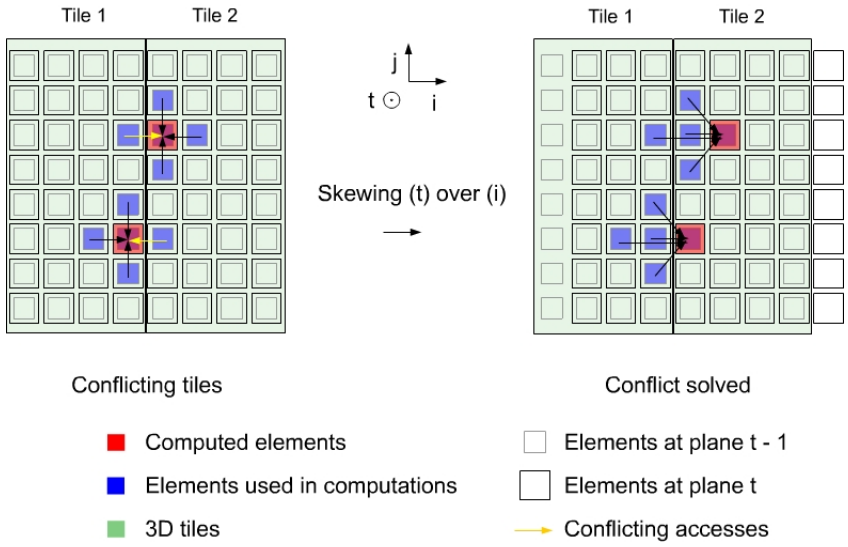


Figure 29: Conflicting tiles solved by time-skewing the volume in the direction of (i) axis

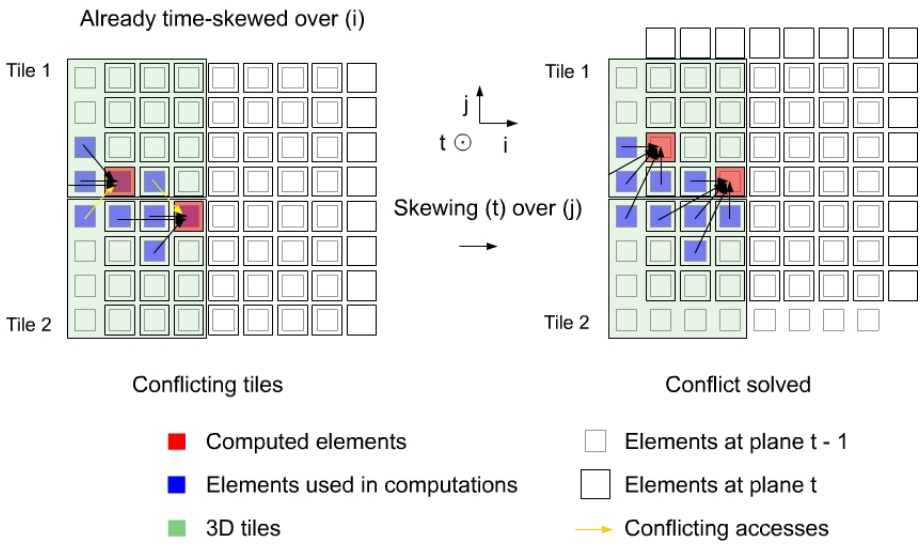


Figure 30: 2D view of time-skewing in the direction of both (i) and (j) axes

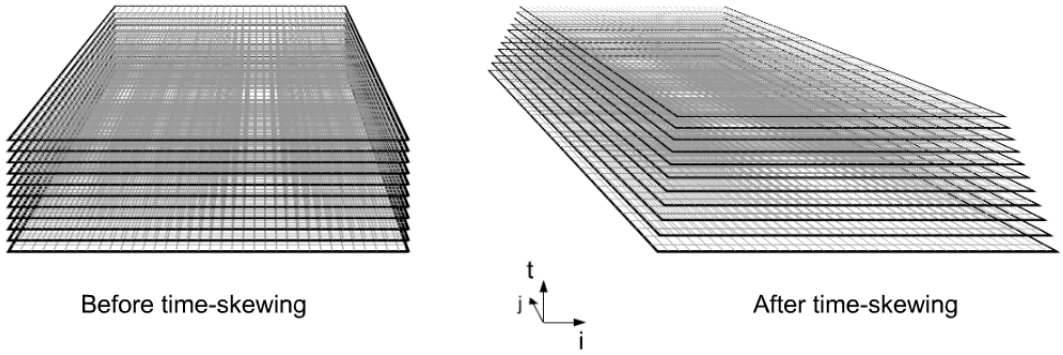


Figure 31: 3D view of time-skewing in the direction of both (i) and (j) axes

Figure 32 below displays the execution of time-skewing applied on the pseudo 2D Jacobi stencil code. Only the first, second, third and last tiles are depicted.

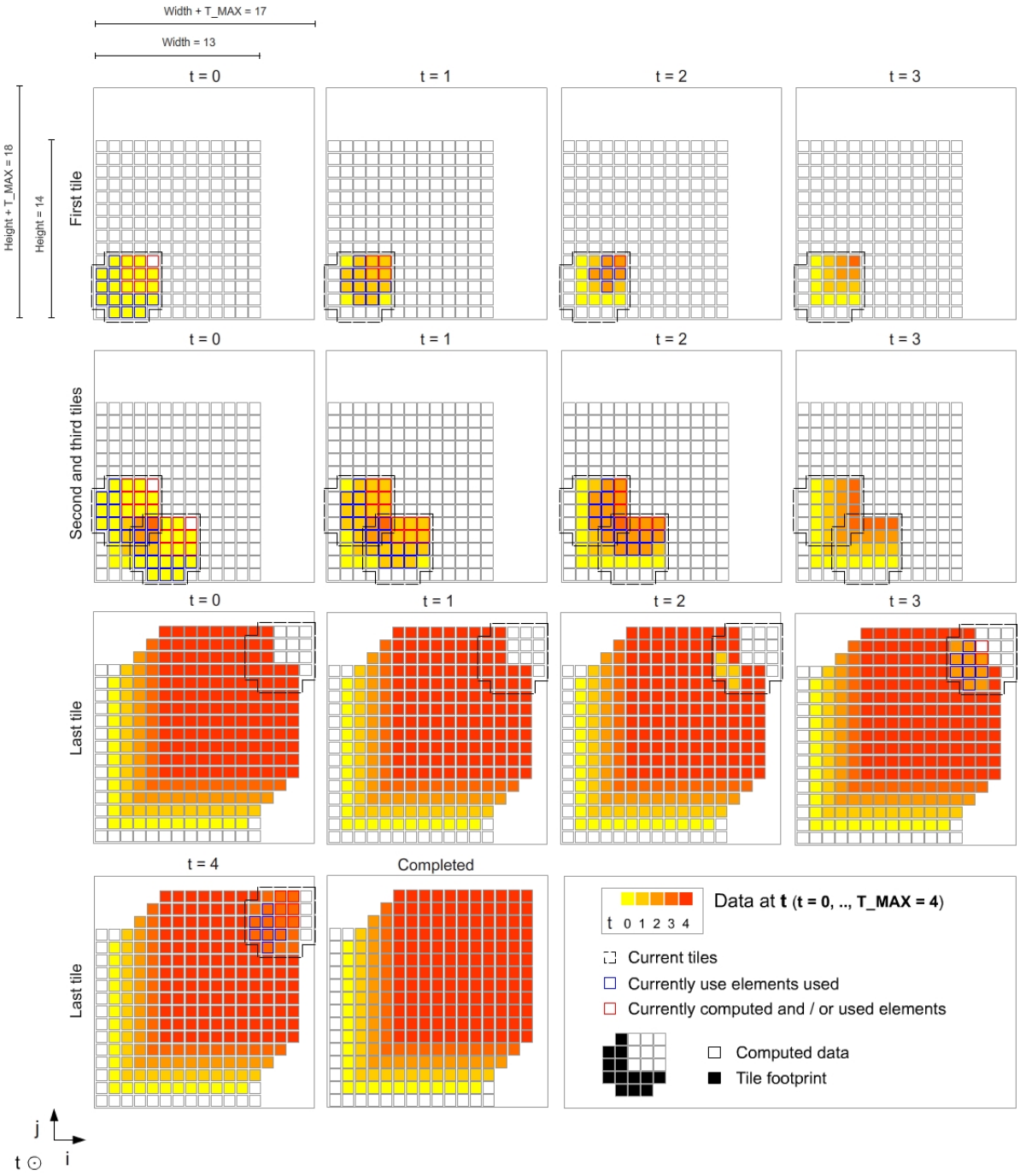


Figure 32: Example of time-skewing applied on the pseudo 2D Jacobi stencil code

Now, we are going to demonstrate why this approach cannot scale with large values of T_{MAX}.

For a given T_{MAX}, T_H (tile height) and T_W (tile width), let's suppose (for simplicity) $T_{MAX} \leq Height$ and $T_{MAX} \leq Width$, then according to Remark 2, the amount of shared memory (in floats) needed per tile (depicted in Figure 33) is given by the formula:

$$[2 * (T_H + T_W) + 1] * T_{MAX} + T_H * T_W - 1$$

This amount must be less than or equal to the total amount of shared memory/threadblock available

in GeForce GTX-280, which is of 16384 bytes, ie. only 4076 floats. For instance, if $T_H = T_W = 3$ (which represents a very small tile), we obtain the constraint:

$$13 * T_{MAX} + 8 \leq 4076$$

which implies that T_{MAX} cannot be greater than 312 !

So, we can deduce that this approach does not scale with a large T_{MAX}.

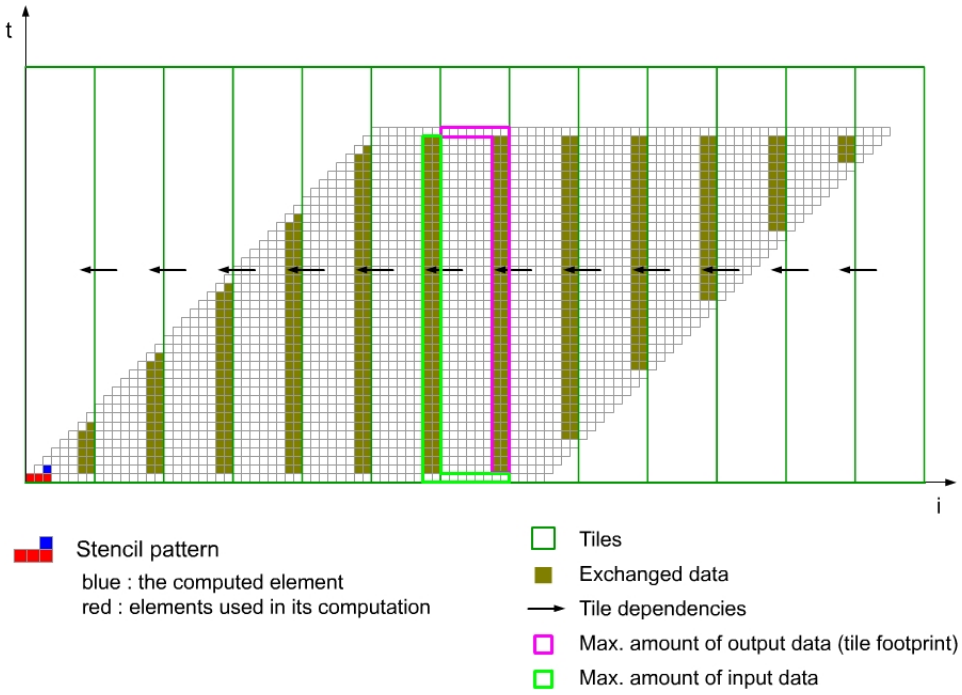


Figure 33: The different types of data within a tile for a similar case of 1D stencil (for simplicity)

In order to fix the amount of memory needed per tile, we can, for example, fix the number of time iterations performed inside the tiles, which corresponds to a time-tiling. This is done in the next approach.

4.2.4 Approach 4: Time-skewing + time-tiling

Here, we use the same idea as in approach 3, but we perform time-tiling as shown in Figure 34 (case of 1D stencil for simplicity). Tiling the (t) axis limits the required amount of shared memory per tile independently of T_{MAX}. Hence, T_{MAX} ban be larger. However, this approach carries a latency when at the beginning of the execution. This is due to the fact that concurrent tiles progress diagonally as shown in Figure 34. Thus, when the execution starts (in the 3D case), only the tile $(0, 0, 0)$ is executed. When its execution is finished, the tiles $(0, 0, 1)$, $(0, 1, 0)$, $(1, 0, 1)$ and $(1, 1, 0)$ that form a diagonal plane are executed, and so on... In addition, it is very difficult to implement this approach with CUDA, because the memory layout is highly complex due to the diagonal tiles progression, and the data (input and output) locality inside tiles is hard to manage. Moreover, using different tile sizes in order to perform load balancing is even more difficult because of time-skewing.

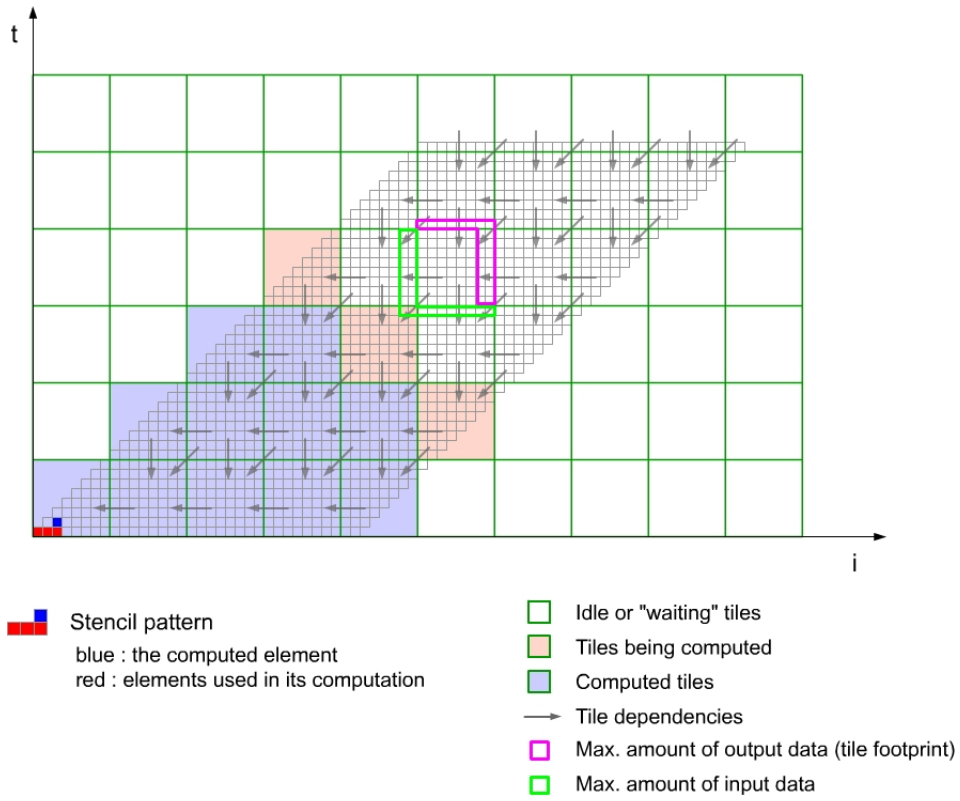


Figure 34: Time-skewing + time-tiling

Even if this approach allows to use 3D tiles without the problems previously encountered with approach 2 (synchronized treadblocks) and approach 3 (time-skewing without time-tiling), it is very difficult to implement. Nonetheless, the next approach extends the 2D tiles of approach 1 to 3D tiles, and remains easy to implement. It solves conflicts between tiles without skewing the iteration domain (as seen in approaches 3 and 4), nor synchronizing threadblocks (as seen in approach 2).

4.2.5 Approach 5: Redundant computations

In this approach, we extend the tiles of approach 1 into 3D tiles then perform time-tiling. Here also we have to solve conflicting tilesbut, without synchronizing threadblocks. The idea is to perform some redundant computations inside tiles as shown in Figure 35. Instead of exchanging data between tiles, each tile performs some extra computations in order to recover the data it needed from other tiles before. In the case of the pseudo 2D Jacobi stencil, this approach gives a trapezoidal tile shape with expanded halos.

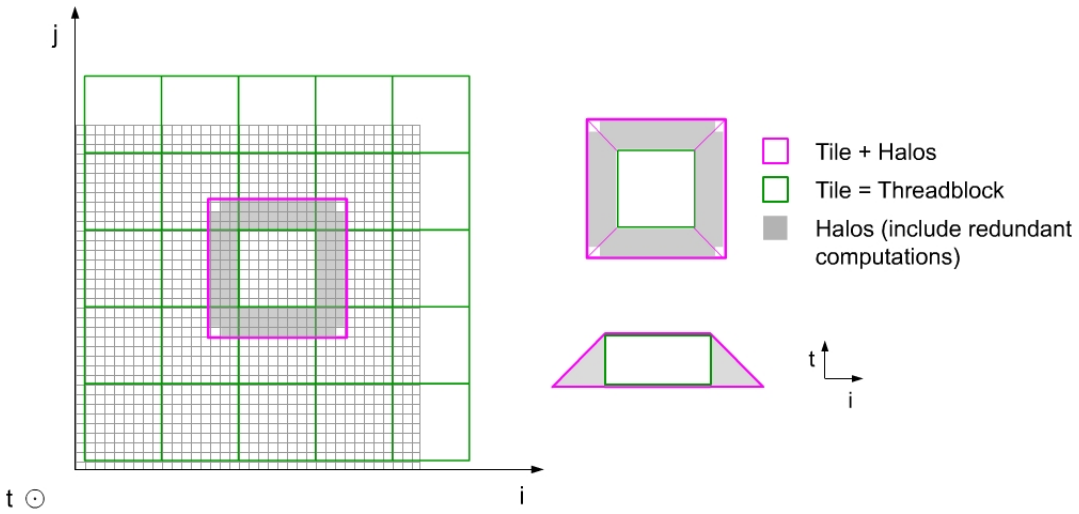


Figure 35: Redundant computations with trapezoidal tile shape

The figure below depicts the similar case of 1D stencil. We can remark from the tile dependencies that the tiles situated at the same temporal level are independent from each other. They can therefore be executed concurrently, which reduces execution latency. Moreover, this approach is easy to implement. It can be done just by applying some modifications on approach 1 in order to extend the halos and increase the number of time steps within tiles. Furthermore, it is easier to operate load balancing by using different tile sizes in entire raws of columns, which can be particularly useful in our case where the computations in FDTD grid is not homogeneous (see Figure 4.1).

Figure 36 depicts the similar case of 1D stencil. We can remark from the tile dependencies that the tiles situated at the same time level are independent from each other. They can therefore be executed concurrently, which prevents execution latency. Moreover, this approach is easy to implement. It can be done just by applying some modifications on approach 1 that extend the halos and increase the number of time iterations within tiles. Furthermore, it is easier to operate load balancing by using different tile sizes in entire raw of columns, which can be particularly useful in our case where the FDTD grid is not homogeneous.

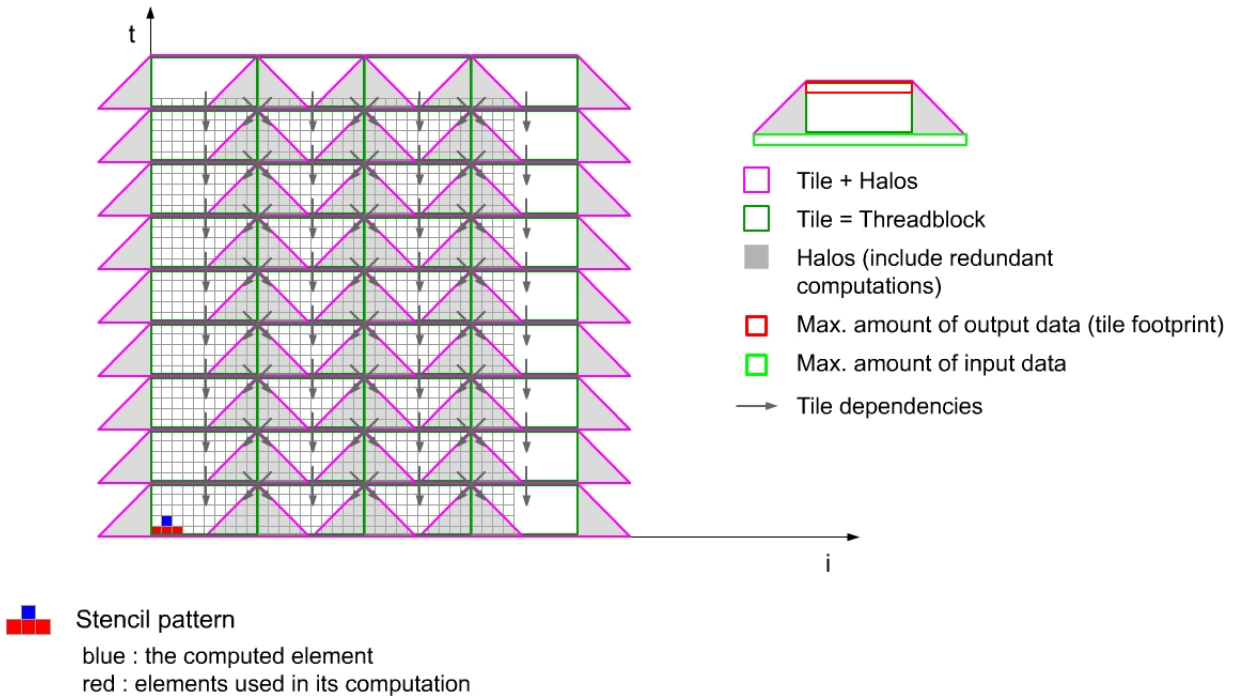


Figure 36: Redundant computations in the case of 1D stencil

4.3 Determining the best approach + the best tile size

We will focus in this subsection on how to determine the best tile size for the different tiling approaches, according to both the pseudo 2D Jacobi and the FDTD stencil codes, then we exhibit the best approach for the two stencils. To do this, we are going to analyse how the shared and global memories are used. Since approach 2 (synchronized threadblocks) and approach 3 (time-skewing only without time-tiling) do not work with large $Height$, $Width$ and T_{MAX} values, we will focus on examining the remaining approaches that are: approach 1 (2D tiles), approach 4 (time-skewing + time-tiling) and approach 5 (redundant computations).

At first, we construct for each approach the function $mem\,(T_H, T_W, T_T)$ that returns the amount (in floats) of shared memory needed to execute a tile, with T_H: the tile height, T_W: the tile width and T_T: the number of time steps performed inside tile, V_C: the number of variables per computed cell, V_L: the number of variables per used only cell (loaded but not computed), C: the number of coefficients per cell. For the 2D Jacobi stencil code, we have $V_C = 1$, $V_L = 1$ and $= C = 0$. For the FDTD stencil code, V_C and C depend on the FDTD grid region that is analysed (see figure 15), whereas V_L can be derived from the intra-cell data flow graph of the FDTD stencil code, it is equal to 2 for all the regions, which means that only two variables are used from any used only cell.

- (Approach 1) $mem_1\,(T_H, T_W, T_T) = V_L * (T_H + T_W) * 2 + V_C * (T_H * T_W) + C * T_H * T_W$
- (Approach 4) $mem_4\,(T_H, T_W, T_T) = V_L * [(T_H + T_W + 2) * T_T + (T_H - 1) + (T_W - 1)] + V_C * [(T_H + T_W - 1) * T_T + (T_H - 1) * (T_W - 1)] + C * [T_H * T_W + (T_T - 1) * (T_H + T_W - 1)]$
- (Approach 5) $mem_5\,(T_H, T_W, T_T) = V_L * [[[T_H + 2 * (T_T - 1)] + [T_W + 2 * (T_T - 1)]] * 2] + V_C * [(T_H + 2 * [T_T - 1)] * [T_W + 2 * (T_T - 1)]] + C * [[T_H + 2 * (T_T - 1)] * [T_W + 2 * (T_T - 1)]]$

Next, we construct the function $comm\,(T_H, T_W, T_T)$ that returns the number of global memory accesses within a tile. The function is expressed in this form: [number of load accesses] + [number of write accesses].

- (Approach 1) $comm_1\,(T_H, T_W, T_T) = V_L * (T_H + T_W) * 2 + V_C * (T_H * T_W) + C * T_H * T_W + V_C * (T_H * T_W)$

- (Approach 4) $comm_4(T_H, T_W, T_T) = V_L * [(T_H + T_W + 2) * T_T + (T_H - 1) + (T_W - 1)] + V_C * [(T_H + T_W - 1) * T_T + (T_H - 1) * (T_W - 1)] + C * [T_H * T_W + (T_T - 1) * (T_H + T_W - 1)] + V_C * [[2 * (T_H + T_W) - 4] * (T_T - 1) + T_H * T_W]$
- (Approach 5) $comm_5(T_H, T_W, T_T) = V_L * [[[T_H + 2 * (T_T - 1)] + [T_W + 2 * (T_T - 1)]] * 2] + V_C * [[T_H + 2 * (T_T - 1)] * [T_W + 2 * (T_T - 1)]] + C * [[T_H + 2 * (T_T - 1)] * [T_W + 2 * (T_T - 1)]] + V_C * (T_H * T_W)$

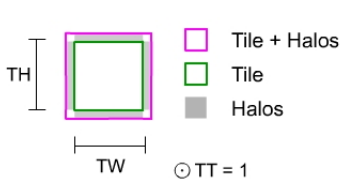
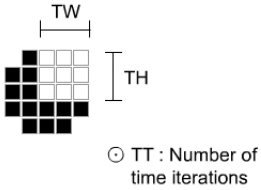
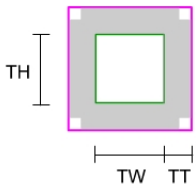
After that, we construct the function $comp(T_H, T_W, T_T)$ that returns the number of floats computed by a tile.

- (Approach 1) $comp_1(T_H, T_W, T_T) = V_C * T_H * T_W$
- (Approach 4) $comp_4(T_H, T_W, T_T) = V_C * T_H * T_W * T_T$
- (Approach 5) $comp_5(T_H, T_W, T_T) = V_C * T_H * T_W * T_T$

Remark 6 *Because approach 1 is a particular case of the two others with $T_T = 1$ that represents the execution of only one time step (approach 1 uses 2D tiles), we have: $mem_1(T_H, T_W, 1) = mem_4(T_H, T_W, 1) = mem_5(T_H, T_W, 1)$, $comm_1(T_H, T_W, 1) = comm_4(T_H, T_W, 1) = comm_5(T_H, T_W, 1)$, and $comp_1(T_H, T_W, 1) = comp_4(T_H, T_W, 1) = comp_5(T_H, T_W, 1)$.*

The best approach is the one that maximizes the ratio of the number of computed floats (which expresses the useful results) to the number of global memory accesses (which expresses the latency), that is $comp(T_H, T_W, T_T) / comm_1(T_H, T_W, T_T)$ under the constraint $mem(T_H, T_W, T_T) \leq M$, where M is the size (in floats) of the GPU shared memory. We have therefore to solve a non-linear integer programming system where the integer variables are: T_H, T_W and T_T (we have seen previously that V_C, V_L and C are already defined). To solve the system, we can perform an exhaustive enumeration for the 3 variables that goes from 1 to $\lfloor M/(C+V) \rfloor$. $\lfloor M/(C+V) \rfloor$ is the maximum value that can have each of the variables when the two other ones are equal to 1.

After solving the system according to the different FDTD regions (cf. 4.1), we have obtained the results in Figure 4.1 with $M = 4076$ floats for the GeForce GTX-280 graphic card, and $M = 16K$ floats for the GeForce GTX-580 graphic card:

	Approach 1: 2D tiles	Approach 4: Time-skewing + time-tiling	Approach 5: Redundant computations
			
PML	GTX-280 Best ratio comp/com = **0.22** Corresponding tile size: **(8, 8, 1)** GTX-580 Best ratio comp/com = **0.22** Corresponding tile size: **(16, 16, 1)**	GTX-280 Best ratio comp/com = **0.41** Corresponding tile size: **(5, 5, 5)** GTX-580 Best ratio comp/com = **0.70** Corresponding tile size: **(10, 10, 9)**	GTX-280 Best ratio comp/com = **0.28** Corresponding tile size: **(6, 6, 2)** GTX-580 Best ratio comp/com = **0.42** Corresponding tile size: **(12, 12, 3)**
Non PML	GTX-280 Best ratio comp/com = **0.18** Corresponding tile size: **(10, 15, 1)** GTX-580 Best ratio comp/com = **0.19** Corresponding tile size: **(22, 28, 1)**	GTX-280 Best ratio comp/com = **0.45** Corresponding tile size: **(7, 8, 7)** GTX-580 Best ratio comp/com = **0.83** Corresponding tile size: **(14, 18, 12)**	GTX-280 Best ratio comp/com = **0.27** Corresponding tile size: **(8, 8, 3)** GTX-580 Best ratio comp/com = **0.47** Corresponding tile size: **(14, 20, 5)**
DFT (FF = 5)	GTX-280 Best ratio comp/com = **0.04** Corresponding tile size: **(5, 6, 1)** GTX-580 Best ratio comp/com = **0.05** Corresponding tile size: **(11, 11, 1)**	GTX-280 Best ratio comp/com = **0.07** Corresponding tile size: **(4, 4, 3)** GTX-580 Best ratio comp/com = **0.12** Corresponding tile size: **(7, 7, 7)**	GTX-280 Best ratio comp/com = **0.04** Corresponding tile size: **(5, 6, 1)** GTX-580 Best ratio comp/com = **0.06** Corresponding tile size: **(9, 9, 2)**
Pseudo 2D Jacobi	GTX-280 Best ratio comp/com = **0.48** Corresponding tile size: **(58, 66, 1)** GTX-580 Best ratio comp/com = **0.49** Corresponding tile size: **(125, 126, 1)**	GTX-280 Best ratio comp/com = **3.13** Corresponding tile size: **(37, 37, 18)** GTX-580 Best ratio comp/com = **6.21** Corresponding tile size: **(75, 76, 35)**	GTX-280 Best ratio comp/com = **3.37** Corresponding tile size: **(34, 42, 13)** GTX-580 Best ratio comp/com = **6.75** Corresponding tile size: **(75, 76, 26)**

Ratio Time-skewing + time-tiling / Redundant computations :

PML :	Non PML :	DFT (FF = 5) :	P. 2D Jacobi :
GTX-280 : **1.47**	GTX-280 : **1.67**	GTX-280 : **1.57**	GTX-280 : **0.93**
GTX-580 : **1.68**	GTX-580 : **1.78**	GTX-580 : **1.92**	GTX-580 : **0.92**

Max shared memory amount of GTX-280 = **4076 floats**
Max shared memory amount of GTX-580 = **16304 floats**

Figure 37: Results of the non linear integer programming system

The results shows that time-skewing+time-tiling approach gives the best ratio for the FDTD stencil code. Whereas, redundant computations approach is the best one for the pseudo 2D Jacobi stencil code.

5 Conclusion

In this report, we have briefly described our target application and presented the Stencil Codes, the category to which it belongs. Then, we have detailed what the loop tiling transformation consists of. We have also presented the CUDA programming model and GPU architecture. After that, we have presented a brief overview of the recent works related to FDTD parallelization on GPUs.

During this internship, we have deeply explored our target FDTD application and exhibited its stencil pattern. We then noticed the similitude between it and the one of 2D Jacobi. Next, we have explored the possible tiling approaches that can be applied on such stencil patterns. We have found 3 interesting approaches: the first one consists of applying 2D tiles. The second one consists of applying time-skewing+time-tiling transformation, whereas the third one consists of performing redundant computations inside tiles.

The time-skewing+time-tiling approach gave the best ratio of computed floats to global memory accesses for the FDTD stencil code. However, the redundant computations approach is not so distanced. Moreover, the time-skewing+time-tiling approach produces execution latency caused by an elapsed delay before getting a maximum number of currently executed tiles, contrary to redundant computations approach that does not carry execution latency. Furthermore, redundant computations approach is much easier to implement than the time-skewing+time-tiling one.

As future work, we will try to implement the FDTD code with the different tiling approaches in order to get a exact comparison between them.

References

[1] A. ROLLAND. Traitement du Signal et Télécommunications. Phd thesis. IETR, groupe Antennes et Hyperfréquences, UMR CNRS 6164, Campus de Beaulieu, Janvier 2009.

[2] C. Bastoul. Improving Data Locality In Static Control Programs. Phd thesis. University of Paris 6 Pierre et Marie Curie, December 2004.

[3] CUDA Programming Guide, 2.1, NVIDIA. `http://developer.download.nvidia.com/compute/cuda/2\_1/toolkit/docs/NVIDIA\_CUDA\_Programming\_Guide\_2.1.pdf`

[4] Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., and Yelick, K. 2008. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures.In Proceedings of the 2008 ACM/IEEE Conference on Supercomputing.

[5] G. Godi, R. Sauleau, "FDTD analysis of reduced size substrate lens antennas", IEEE Antennas and Propagation Society International Symposium, Monterey, California, US, vol. 1, pp. 667-670, 20-26 Juin 2004.

[6] Gabriel Rivera, Chau-Wen Tseng. Tiling Optimizations for 3D Scientific Computations. In Proceedings of SC'00, November 2000, Dallas, TX.

[7] Louis-Noël Pouchet. Iterative Optimization in the Polyhedral Model. *Ph.D (University of Paris-Sud XI)*, Orsay, France, January 2010.

[8] M. G. M. V. Silveirinha, C. A. Fernandes, "Design of a non-homogeneous wire media lens using genetic algorithm", IEEE AP-S International Symposium, San Antonio, Texas, vol. 1, pp. 730-733, 16-21 Juin 2002.

[9] M. Lam, E.Rothberg, andM. E.Wolf. The cacheper-formance and optimizations of blocked algorithms. In Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV), Santa Clara, CA, April 1991.

[10] NADARAJAH Saralees. An Explicit Selection Intensity of Tournament Selection-Based Genetic Algorithms. IEEE transactions on evolutionary computation, 2008, vol. 12, no3, pp. 389-391

[11] Paulius Micikevicius. 3D Finite Difference Computation on GPUs using CUDA. NVIDIA. GPGPU2, March 8, 2009, Washington D.C., US.

[12] P. Feautrier. Some efficient solutions to the affine scheduling problem : one dimensional time. International Journal of Parallel Programming, 21(5):313-348, october 1992.

[13] P. Feautrier. Some efficient solutions to the affine scheduling problem, part II : multidimensional time. International Journal of Parallel Programming, 21(5):389-420, december 1992.

[14] Volkov, V. 2010. *Better performance at lower occupancy*, GPU Technology Conference 2010 (GTC 2010).