POLITECNICO DI MILANO
MASTER OF SCIENCE COURSE IN
COMPUTER SCIENCE AND ENGINEERING

Optimization of a MATLAB Iterative Kernel: Performance Comparisons Between GPU-based Computing and the Polyhedral Model

High Performance Processors and Systems – Research Project
Prof. Donatella Sciuto and Marco D. Santambrogio, a.y. 2013/2014

a project by

Edoardo Mondoni
(816283)

tutored by

Riccardo Cattaneo,
NECSTLab

Table of contents

Preamble

This is the final report for Edoardo Mondoni's *High Performance Processors and Systems* Research Project. It contains a thorough illustration of the project's theoretical foundations and aims, and explains what it has consisted in in detail and what conclusions could be drawn from the experimental evidence.

The remainder of the document is organized as follows.

- Chapter 1 describes the technologies tackled throughout the project, namely CUDA and the polyhedral model, and provides some context and examples to help understand the reasons behind their birth and success.
- Chapter 2 presents the goals of the project and how they were achieved, featuring detailed analyses of code snippets and of the optimizations they underwent; it also contains a concise rundown on the theoretical work underlying the code on which we worked.
- Finally, Chapter 3 illustrates the results of our efforts and the conclusions we drew from said achievements, sketching out some suggestions for future developments of this work.

1 Technical Overview

Parallelization is one of the hottest keywords driving research and evolution in the field of computer architectures nowadays. Over the last decades, CPUs have become more and more powerful thanks to a variety of enabling factors, ranging from ever-shrinking transistors to smarter and more stratified cache memories. Diminishing returns from these betterments, however, have become evident in recent years, and have pushed the industry towards the development of multi-threaded and multi-core CPUs realizing a *Multiple Instruction Multiple Data (MIMD)* paradigm.

While general-purpose computing can surely benefit from such modern architectures, which usually feature 4 or 8 identical cores on one chip, other applications – e.g. photo and video editing, 3D graphics, simulations, cryptography, etc. – need way more computational horsepower to complete their tasks in sensible amounts of time. More specifically, what these applications all have in common is:

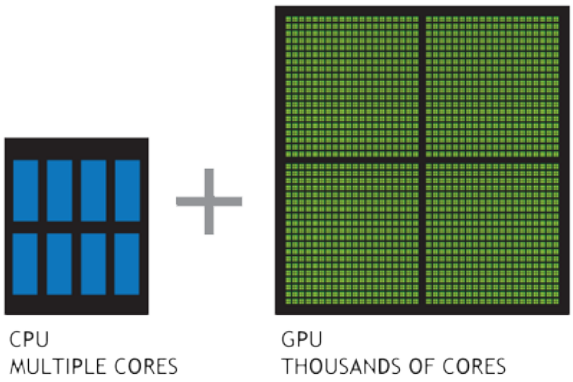
- the **repetitiveness** of their tasks, meaning they mostly consist in executing the same sequence of instructions on multiple pieces of data;
- the relative **simplicity** of the operations performed by these programs, generally featuring a very linear execution flow and a low branching factor;
- the **large amounts of data** to which those operations have to be applied.

The *Single Instruction Multiple Data (SIMD)* computing paradigm therefore looks much more suitable to this kind of applications. Today's most widespread devices implementing this architectural model are undoubtedly Graphics Processing Units.

1.1 GPUs and General-Purpose Computing

GPUs were originally conceived as separate processors, with a dedicate memory hierarchy, taking care of all graphics-related computations, thus allowing the CPU to be relieved of this kind of workload and to focus on the rest. Given the limited variety of tasks they were in charge of, these devices featured much simpler pipelines than those found in CPUs; instead of being devoted to complicated control structures, on-chip surface was allotted to multiple identical *multiprocessors*, each of which contained up to hundreds of *cores* (i.e. processing units). This configuration granted GPUs the ability to execute large numbers of instructions on multiple different data at the same time.

While the structure of a GPU has roughly remained the same, a lot of companies in the industry (including Apple, Nvidia, AMD, Intel, Nokia, and many more) recently steered their attention towards the opportunity to harness the vast computational power of such devices for a broader category of tasks, thus giving birth to *General-Purpose computing on Graphics Processing Unit (GPGPU)*. In layman's words, they explored the possibility to program a GPU to make it perform any kind of operation, rather than just those strictly related to graphics and images. These efforts led to the development of the two most common GPGPU frameworks currently available:



Picture 1: GPUs outperform CPUs when it comes to raw computational horsepower.

- **CUDA** (short for *Compute Unified Device Architecture*), developed by Nvidia, which was first released in 2007 and is the dominant proprietary solution [**1**]; as such, it only works with Nvidia GPUs.
- **OpenCL** (short for *Open Computing Language*), which was released by Apple in 2009 and has been being developed and maintained by the Khronos Group since the following year [**2**]; it is the most prominent GPGPU open framework and supports a broad variety of devices from different vendors.

1.2 CUDA

Nvidia defines CUDA as «a parallel computing platform and programming model». CUDA encompasses a wide range of technologies aimed at making parallel computing straightforward and broadly available. While running CUDA applications only requires a compatible device (i.e. any recent Nvidia-branded GPU) and the installation of a toolkit (freely available for all most popular operating systems such as Windows, OS X and Linux), developing parallel programs is not much harder for a programmer. In fact, the framework offers three distinct approaches to make use of a GPU's computing power, each of which is characterized by an increasing degree of flexibility (and, of course, of difficulty):

- *Optimized parallel libraries* are offered containing parallel versions of common computationally intensive algorithms [**3**]: adopting them is as easy as adapting a function call where needed and linking them instead of their serial counterparts.
- *Compiler directives* enable a developer to parallelize custom code without having to change any of it. The programmer's task just consists in adding said directives to point out parts of the code that are suitable for parallelization; the compiler takes care of translating those chunks into GPU code.
- *Extensions to common programming languages* make it possible to write parallel applications, endowing the programmer with full control over what is executed on the GPU. As of 2014, Nvidia offers extensions to C/C++, FORTRAN and Python: this solution – as opposed to the development of an entirely new programming language –

features much steeper learning curves (familiarity with syntax, libraries, functions, IDEs…) and avoids the need to rewrite the parts of the program that are not going to be executed on the GPU, thus preventing the inevitable introduction of bugs the rewriting process would induce.

Given that the program had to be written from scratch anyway, we chose to adopt the third approach: the software was thus developed in CUDA C/C++.

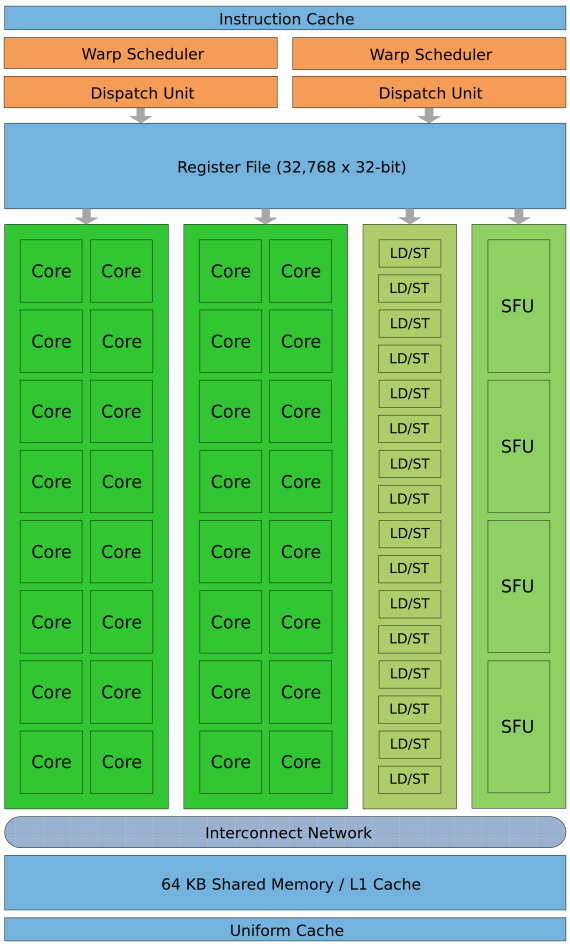
1.2.1 Architectural Outline

The CUDA framework identifies two distinct entities where computation takes place:

- the CPU, or the ***host***, in charge of executing serial pieces of code; and
- the GPU, or the ***device***, to which kernels are dispatched for parallel execution.

It is, of course, up to the programmer to be aware of the difference between them and to write code according to where it will be executed; this distinction imposes a clear-cut separation between the host's and the device's memory spaces – more on that will follow in Subsection 1.2.3.

The physical architecture of a CUDA-capable GPU corresponds to the general model we laid out in Section 1.1: it contains a number of identical *Streaming Multiprocessors (SM)*, each containing large quantities of *CUDA Cores* carrying an FP unit and an Integer unit. A typical CUDA SM is shown in Picture 2.



Picture 2: a Fermi Streaming Multiprocessor containing 32 CUDA Cores.

1.2.2 The Single Instruction Multiple Thread (SIMT) Paradigm

Although CUDA undoubtedly implements a SIMD paradigm, Nvidia further elaborated on the subject, renaming its computational model *Single Instruction Multiple Thread (SIMT)*: understanding the subtlety of this definition requires knowledge of the way the computation is organized inside the GPU.

To a first approximation, the only pieces of code executed on the device are functions marked with the `__global__` or `__device__` qualifiers. Any other instruction is executed on the CPU, none differently from what would happen in a normal, serial program. A function declared with the `__global__` keyword is also referred to as *kernel*: it is called from the host but its execution takes place on the device. Kernels are the core of a parallel program: they typically replace functions consisting in `while` and `for` loops of thousands, millions or even billions of iterations – accordingly named *iterative kernels*. If the iterations of an iterative kernel are mutually independent (i.e. any given iteration does not need the data produced by a previous

iteration, as it happens in Code snippet 1), they can be parallelized and executed at the same time: a GPU is thus an optimal choice because its thousands of cores can simultaneously process thousands of iterations, leading to a dramatic reduction of the execution time.

STANDARD C CODE

```
saxpy_serial(int n, double a,
             double *x, double *y) {
   for (int i = 0; i < n; i++)
      y[i] = a * x[i] + y[i];
}

//Perform saxpy on 1M elements
saxpy_serial(4096*256, a, x, y);
```

PARALLEL C CODE

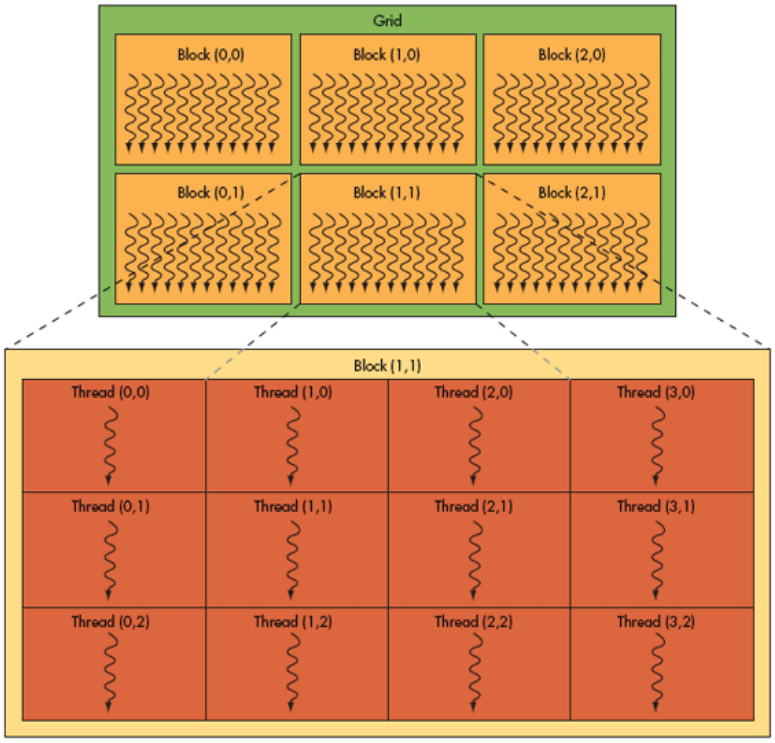
```
saxpy_parallel(int n, double a,
               double *x, double *y) {
   int i = blockIdx.x + blockDim.x *
           threadIdx.x;
   if (i < n)
      y[i] = a * x[i] + y[i];
}

//Perform saxpy on 1M elements
saxpy_parallel<<<4096,256>>>(a, x, y);
```

Code snippet 1: `for` loops are parallelized, and their iterations are assigned to different threads executed on different physical cores.

As a consequence, the iterative kernel is "flattened" and each iteration runs in what is called a ***thread***.

> ***Calling a kernel causes the GPU to launch as many threads as the programmer specifies. Each thread runs the same code (i.e. the body of the kernel), but is aware of its identity thanks to some identifiers (`threadIdx`, `blockIdx`) which are often used to access different locations in memory (e.g. as array indices) and thus to perform the same sequence of operations on different pieces of data.***

The thread is therefore the fundamental unit of computation in the SIMT paradigm. At any given time, a thread runs in one CUDA Core. In order to simplify the manipulation of inherently bi- or tri-dimensional data, threads can be organized in 1-, 2- or 3-D ***thread blocks***; it is essential to understand that the dimensional disposition of the threads does not affect their actions in any way, in that it is just "geometric sugar" aimed at helping the programmer not to get lost with indices and memory references. Furthermore, because a block cannot contain more than 1024 threads (as of CUDA 6.5), it is certainly possible to launch more than one block at a time: the set of blocks dispatched simultaneously to execute a



Picture 3: the kernel in this picture is being concurrently executed by 72 threads, partitioned into a two-dimensional grid of 6 two-dimensional blocks of 12 threads each.

single kernel is called ***block grid***. The grid can, in turn, be one-, two- or three-dimensional according to the programmer's preferences on a case-by-case basis.

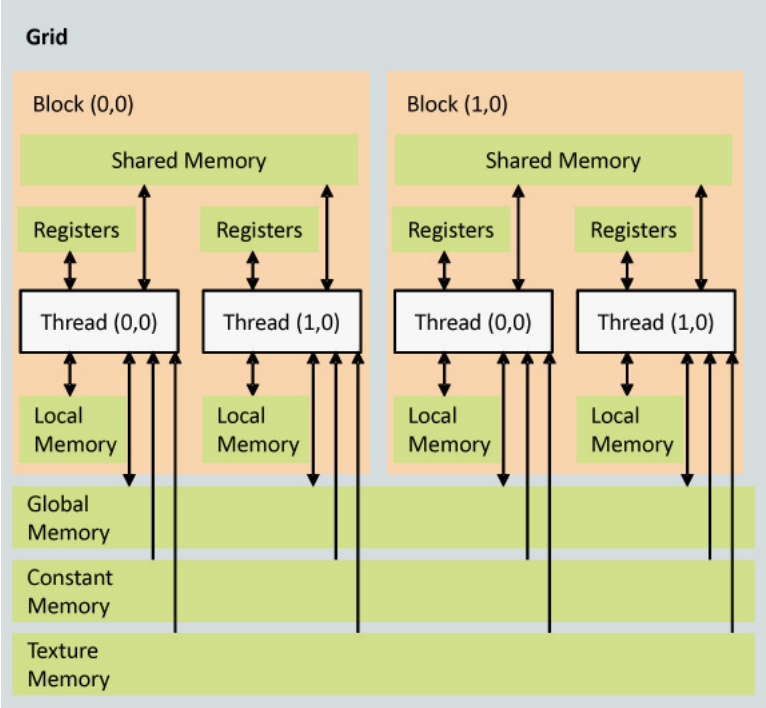
As Code snippet 1 points out, a kernel call must be accompanied by the *triple chevron construct* (`<<<gridSize, blockSize>>>`) informing the GPU of the desired sizes for the grid and the blocks (all blocks pertaining to the same invocation must have the same size). The `gridSize` and `blockSize` parameters can be `int`s (or of any other analogous type) or, more generally, of type `dim3` – a CUDA built-in `struct` composed of three integers (fields `x`, `y`, and `z`), each of which identifies the magnitude of one dimension.

To come full circle, the difference between SIMD and SIMT lies in the way code is written. In CUDA, device code looks like a serial chunk of code: there is no difference between the instructions one would use in a standard C context and those found in a kernel. In other words, **the programmer writes the code for just one thread**; the replication of that code into parallel threads is specified *outside the kernel* (through the `<<<>>>` construct) and is managed by the platform, *not* by the programmer. SIMD architectures (e.g. vector processors), on the contrary, expose the parallelization to the software itself, in that they require the use of inherently vectorial instructions.

1.2.3 Memory Hierarchy

As we've said in Subsection 1.2.1, the separation of host and device induces the disjunction of their respective memory spaces. What's more, mastering the stratified memory model offered by the platform is essential to obtaining the performance bumps that CUDA promises.

For starters, **the host cannot access the device's memory, and vice-versa**. As a consequence, data need to be explicitly moved from one memory space to another through the `cudaMemcpy()` primitives offered by the CUDA Runtime API: kernel invocations are thus commonly preceded by a host-to-device `cudaMemcpy()` – to copy the kernel's input to device memory – and followed by the same call in the opposite direction – to make the results of the computation available to host code (e.g. to write them to a file or to print them). The introduction of Unified Memory Programming [**4**] in April 2014 only *apparently* lifted this principle: host code can access device pointers and vice-versa. In reality, UMP just masks the need for `cudaMemcpy()` calls by performing the copy under the hood: this allows programmers to write



Picture 4: the CUDA memory hierarchy includes five different types of device memory.

more immediate and maintainable code without taking care of moving the data, but upholds CUDA's memory space separation.

While host memory is just the plain old memory we're used to, device memory is partitioned into **five different memory areas**, as shown in Picture 4; the programmer's in charge of allocating the right type of memory for each piece of data according to how often those data will be accessed, how big they are and how the kernel accesses those data (i.e. in a coalesced way, in a strided way, at random…).

- **Global memory** is the default type of memory, allocated by calls to `cudaMalloc()` and by declaring variables with the `__device__` qualifier. *Shared among all threads* in a kernel and by far the *largest* one, global memory is also characterized by relatively *high latencies* (just like main memory with respect to caches in CPUs).
- **Local memory** is the second most used type of memory. It is *private to each thread* and allocated by in-kernel variable declarations; it often gets cached to registers, which makes it the *fastest* and the *smallest* type of memory available on the GPU.
- **Shared memory** is *shared among all threads in the same block*. It can be allocated by declaring an in-kernel variable with the `__shared__` qualifier and is an effective compromise between the low latencies of local memory and the sharing capabilities of global memory.
- **Constant memory** is a small memory space allocated by declaring a variable in host code with the `__constant__` qualifier. Its name refers to the impossibility for threads to modify its content; a `__constant__` variable is, however, not to be confused with a `const` variable, since the former can still be modified at any time from host code with the `cudaMemcpyToSymbol()` primitive. So, provided that a value doesn't need to be changed during the execution of a kernel, it can be stored in constant memory in order to benefit from *register-like access latencies* and *global-memory-like scope*; a variable residing in constant memory yields considerable performance boosts over global memory only if threads access it in lockstep (i.e. at the same point in the code).
- **Texture memory** is a read-only memory space, *shared among all threads*, optimized for 2-D spatial locality. As its name suggests, it was originally conceived to host graphic textures for fast access, but is available to general-purpose computing as well.

1.2.4 Compute Capability

Since CUDA was first introduced in 2007, Nvidia has been keeping on developing additional functionalities and relaxing constraints for the framework. While support on the software part can be easily obtained by updating the CUDA Toolkit to the latest version, many of those features require built-in hardware support and are subsequently not available on older devices.

Each GPU is therefore assigned a *Compute Capability* value, which roughly works as a "hardware version" identifier. **GPUs with the same Compute Capability are guaranteed to support the same set of features**, and only differ from each other in the number of SMs or the amount of memory available – i.e. in the raw power the GPU offers. Nvidia regularly updates a table [**5**] detailing the differences between Compute Capabilities in terms of supported functionalities and technical specifications.

1.3 The Polyhedral Model

Progress in the field of code optimization hasn't only stemmed from parallelization and SIMD architectures, though. Over the last two decades, researchers have also focused on the development of the so-called *polyhedral model* – a mathematical framework aimed at **mapping the iterations of an iterative kernel to a finite set of points** belonging to a multidimensional space. Establishing a parallel between a kernel and geometrical entities – the polyhedra – subsequently enables compilers and other tools to **perform automated manipulations of said kernel**, by simply applying algebraic transformations, to the purpose of improving performance.

The polyhedral representation of code is a breakthrough in that it paves the way to various kinds of code rearrangements, ranging from reordering instructions to unrolling loops, replacing instructions with vector equivalents, and much more, while **ensuring compliance of such manipulations with the data dependences** between different iterations of the kernel.

It is therefore essential to understand that the polyhedral model is nothing more than a set of mathematical procedures, mainly borrowing from vector space theory, that *do not constitute performance boosts on their own*. Rather, software tools can make use of this representation of code to optimize the code according to different goals (e.g. parallelism and/or data locality).

1.3.1 Background

Not all kernels can be represented in the polyhedral model. To a first approximation, we can obtain a polyhedral representation of a piece of code if and only if it is a *static control part (SCoP)*, that is, a **maximal set of consecutive statements with static control in a program** [6], [7]. In turn, a statement is said to have static control if it matches the following conditions ("affine" can be read as "linear" for the sake of simplicity):

- its only control statements are `for` loops with affine bounds and `if` conditionals with affine conditions;
- said bounds and conditions only depend on outer loop counters and constant parameters.

A SCoP is thus a kernel whose loop bounds and conditions do not depend on values that change during their execution. Each statement in a SCoP is executed multiple times, according to the enclosing loops and conditionals, and each one of these iterations can be identified by the values of the loop counters, which together form the *iteration vector* associated to that particular instance of the statement. Consequently, the loops and conditionals enclosing a statement S determine its *iteration domain* $\mathcal{D}_S$:

```
for(i = 0; i <= N; i++) {
  for(j = 1; j <= M-i; j++) {
    S1(i, j);

    if(i >= j)
      S2(i, j);
  }
}
```

Code snippet 2: a static control part (SCoP).

Given a statement S, enclosed in loops and conditionals adding up to n counters, its iteration domain $\mathcal{D}_S$ is the set of all iteration vectors valid for that statement.

$$\mathcal{D}_S = \{(x_1, \dots, x_n) \in \mathbb{Z}^n | (x_1, \dots, x_n) \text{ is an iteration vector for } S\}$$

$$\mathcal{D}_{S1} = \{(i,j) \in \mathbb{Z}^2 \mid i \geq 0 \wedge i \leq N \wedge j \geq 1 \wedge j \leq M - i\}$$
$$= \left\{ (i,j) \in \mathbb{Z}^2 \mid \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ -1 & -1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} \geq \begin{pmatrix} 0 \\ -N \\ 1 \\ -M \end{pmatrix} \right\}$$
$$\mathcal{D}_{S2} = \{(i,j) \in \mathbb{Z}^2 \mid i \geq 0 \wedge i \leq N \wedge j \geq 1 \wedge j \leq M - i \wedge j \leq i\}$$
$$= \left\{ (i,j) \in \mathbb{Z}^2 \mid \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ -1 & -1 \\ 1 & -1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} \geq \begin{pmatrix} 0 \\ -N \\ 1 \\ -M \\ 0 \end{pmatrix} \right\}$$

Picture 5: the iteration domains for statements S1 and S2 from Code snippet 2, in two equivalent forms.

Because a kernel consists in a finite number of iterations, a statement's iteration domain is finite, too. As such, the interpretation of its elements as vectors (or points) generates an n-dimensional polyhedron for each statement. We can thus infer that **the polyhedral representation of a SCoP is a union of polyhedra**, one for each statement it includes. This is the way the polyhedral model maps kernels to geometric entities.

A lot of papers progressively extended and polished the original model, making it more inclusive and enabling its use in a number of situations where it was previously deemed inapplicable: it is the case of `while` loops [**8**], for example.

1.3.2 The Polyhedral Framework

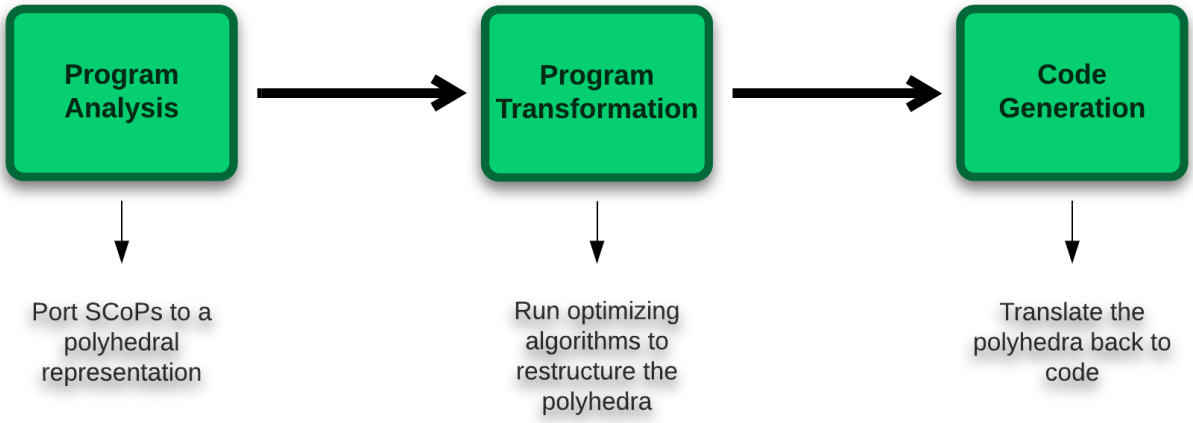
As Benabderrahmane et al. point out [**8**], optimizing programs through the polyhedral model is a **three-step process** (Picture 6):

- PROGRAM ANALYSIS. Ad-hoc algorithms scan the program to detect the SCoPs it contains and translate them to the polyhedral model.
- PROGRAM TRANSFORMATION. Polyhedral optimizers perform affine transformations on the resulting polyhedra in order to optimize the program. According to the optimization goals, different types of transformations are available.
- CODE GENERATION. The optimized polyhedra are ported back to the original language, which might be a high-level programming language (in the case of source-to-source polyhedral compilers like PLuTO [**9**]) as well as a low-level intermediate representation (like Polly, which works with LLVM-IR [**6**], [**10**]). The importance of this step is not to be underestimated: poor code generation algorithms can waste the performance improvements introduced in the second phase.

1.3.3 Program Transformations and Performance Boosts

To complete our overview on the polyhedral model and on how iterative kernels can benefit from it, let's take a look at a handful of transformations that a polyhedral optimizer can apply during the second phase. The following examples take inspiration from [**6**].

Loop blocking, i.e. the composition of strip-mining and interchange, is one of the most common polyhedral transformations. Consider Code snippet 4: the loop completely traverses the `c` array

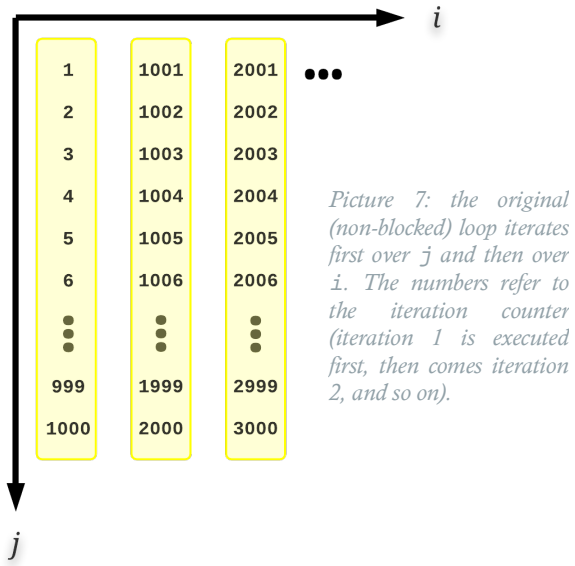


Picture 6: the three-step process of the Polyhedral Framework.

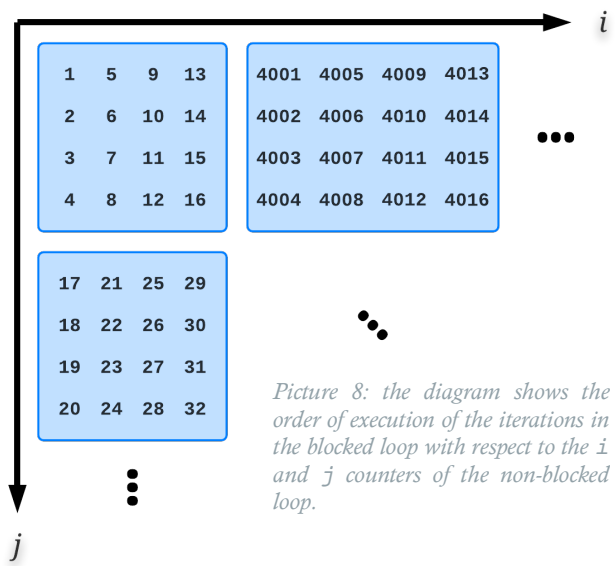
a thousand times. Depending on the size of the cache, the CPU might need to re-fetch the entire array over and over if it doesn't entirely fit into the cache. How many load operations are required to load the entire array depends on the size of the cache block, instead. The point is that improving data locality – i.e. making smarter use of cached data, thus dramatically reducing the number of loads from main memory – is as simple as restructuring the loops enclosing the statements: Code snippet 4 exemplifies this transformation; Picture 7 and Picture 8 further clarify how loop blocking works.

Detection of parallel loops is a technique that polyhedral compilers can apply to determine whether two or more statements under the same loop can be split into two separate loops during the Code Generation phase. If so, the resulting program would feature relevant performance boosts on multi-threaded or multi-core machines, which would execute the split loops in parallel. Code snippets 5 and 6 constitute a clear example of how loop restructuring can make parallelism explicit.

Finally, *loop vectorization* consists in shrinking the iteration domain of a statement by replacing it with its SIMD equivalent. Indeed, this is only possible if the processor's ISA includes vector instructions, hence the code generator must take care of making the appropriate substitutions according to the architecture the program is being compiled for. See Code snippets 7 to 9 for an example.



Picture 7: the original (non-blocked) loop iterates first over `j` and then over `i`. The numbers refer to the iteration counter (iteration 1 is executed first, then comes iteration 2, and so on).



Picture 8: the diagram shows the order of execution of the iterations in the blocked loop with respect to the `i` and `j` counters of the non-blocked loop.

```
for(i = 0; i < 1000; i++)
  for(j = 0; j < 1000; j++)
    A[i][j] = B[i] * C[j];
```

Code snippet 4: loads from the `C` array do not benefit from caching as much as they could in principle, since load operations from the same cell are separated by 1000 other loads from the other cells of the array. Stores to the `A` matrix might also benefit from increased data locality.

```
for(ti = 0; ti < 1000; ti+=4)
  for(tj = 0; tj < 1000; tj+=4)
    for(i = ti; i < ti + 4; i++)
      for(j = tj; j < tj + 4; j++)
        A[ti][tj] = B[ti] * C[tj];
```

Code snippet 4: blocking the loop results in enhanced data locality. The 4-by-4 block size allows us to expect, at worst, one-fourth of the cache misses than those experienced with the non-blocked loop, without considering the impact of blocking on the store operation.

```
for(i = 0; i < M; i++)            (1)
  for(j = 0; j < N; j++)          (2)
    for(k = 0; k < K; k++) {      (3)
      A[i][j] += k;
      B[k] += k;
    }
```

Code snippet 5: there is no possibility to parallelize any of the three loops. Scheduling the iterations of loop (3) in parallel would, in fact, lead to `K` race conditions on every `A[i][j]`. Similarly, parallelizing loops (1) or (2) or both would cause `M`, `N`, or `M*N` race conditions (respectively) on every `B[k]`.

```
for(i = 0; i < M; i++)            (a)
  for(j = 0; j < N; j++)          (b)
    for(k = 0; k < K; k++)        (c)
      A[i][j] += k;

for(i = 0; i < M; i++)            (d)
  for(j = 0; j < N; j++)          (e)
    for(k = 0; k < K; k++)        (f)
      B[k] += k;
```

Code snippet 6: loops (a) and (b) can be parallelized (e.g. by assigning their iterations to different threads and/or cores), as well as loop (f). Simply splitting the loops exposed parallelism that was impossible in the original code.

```
for(i = 0; i < 4096; i++)
  B[i] = A[i];
```

Code snippet 7: the original loop performs a scalar operation 4096 times.

```
for(i = 0; i < 4096; i+=4)
  for(ii = i; ii < i + 4; ii++)
    B[ii] = A[ii];
```

Code snippet 8: the loop is manipulated so as to expose a trivially vectorizable loop (i.e. a parallelizable loop, having a constant and small number of iterations).

```
for(i = 0; i < 4096; i+=4)
    B[i:(i+3)] = A[i:(i+3)];
```

Code snippet 9: the newly created loop is replaced by a vectorial instruction. The resulting loop features one fourth of the iterations (i.e. lower execution time and less control overhead) and takes advantage of the SIMD facilities of the processor.

2 Project Report

2.1 Aim of the Project

The project consisted in **a performance comparison** between three versions of the same program, differing from each other in the language in which they were written:

- the **MATLAB** version, i.e. the original code we were provided;
- the **plain C** version, which was written from scratch and in turn engendered other subversions according to the compilation parameters and the features included;
- the **CUDA C/C++** version, identically developed on our own and forking into a number of other declinations featuring progressive memory optimizations.

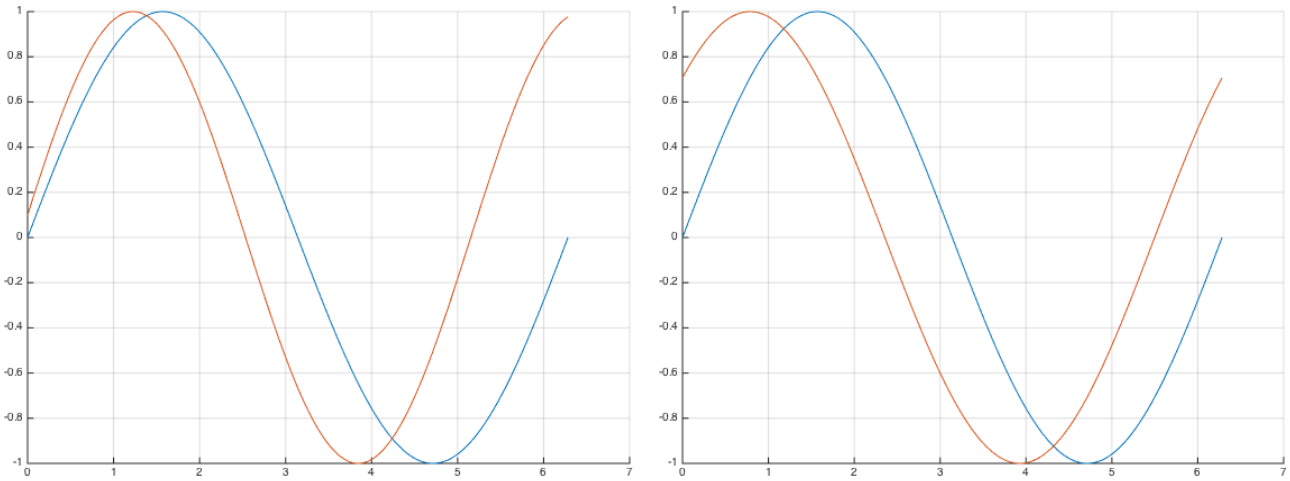
The performance indicator deemed most relevant to achieving our goals is the **execution time** of the programs. In light of the use of lower-precision instructions in optimized code, an assessment of the outputs' precision was conducted as well by contrasting the results produced by the C and CUDA C/C++ binaries and deriving their absolute and relative errors against the original MATLAB program.

2.2 Theoretical Foundations: What Does The Code Do?

As anticipated in the previous Section, we optimized a pre-existing piece of code. More precisely, we were provided with the original MATLAB program by Prof. Paolo Maffezzoni, who wrote it in order to verify and prove the theoretical findings exposed in [**11**]. A smattering of said paper can greatly improve the understanding of the programs we wrote and tested.

The paper deals with the simulation of arrays of interconnected LC oscillators generating multiphase signals. In layman's words, the research proposes a technique to simulate the behaviour of the output waves of such oscillators, which generally feature a sinusoidal shape. The ultimate purpose of this simulation is to identify which combinations of array topology and oscillator parameters are best at **ensuring a stable phase separation between the output signals**. Picture 9 helps to understand the difference between a time-variant phase separation and a stable one.

The interconnections between the oscillators, which cause their outputs to influence one another, are modelled by a *transadmittance matrix* $Y \in \mathbb{R}^{n_{\text{osc}} \times n_{\text{osc}}}$, $n_{\text{osc}} \in \mathbb{N}$ being the number of oscillators in the array. The elements of the transadmittance matrix depend on the topology of



Picture 9: on the left, two signals with a time-variant phase separation. On the right, two waves with a stable phase separation.

the network and on the physical properties of the oscillators themselves; we'll consider them a given.

The output of the k-th oscillator in the system, $V_k(t)$, is a function of both the output of the same oscillator in free running (i.e. without external perturbations), $V_o(t)$, and of the time-dependent time-shift it is subject to due to the other oscillators, $\alpha_k(t)$ [s]:

$$V_k(t) = V_o(t + \alpha_k(t)) \tag{1}$$

The α time-shift, in turn, depends on the external perturbation coming from the system, represented by the current $I_k(t)$ exchanged with the resistive interconnection network; that value is weighted through a periodic sensitivity function, $\Gamma(t)$, equally depending on the oscillators themselves:

$$\dot{\alpha}_k(t) = \Gamma(t + \alpha_k(t)) \cdot I_k(t) \tag{2}$$

Furthermore, the currents can be computed considering that

$$I_k(t) = \sum_{j=1}^{n_{\text{osc}}} y_{kj} \cdot V_j(t). \tag{3}$$

Our final goal is to calculate the total phase $\theta_k(t)$ for each oscillator in the system for a sufficient time frame, in order to determine whether the separation between the phases is constant for $t \to +\infty$ (i.e. the θ_k all proceed in parallel) or not (i.e. their graphs diverge). The θ_k are defined as follows:

$$\theta_k(t) = \omega_k \cdot (t + \alpha_k(t)) \text{ [rad]}, \tag{4}$$

ω_k being the pulsation of the k-th oscillator's output signal. **Obtaining the θ_k values thus requires prior knowledge of the α_k**, which we will compute by plugging (1) into (3) and (3) into (2), and by subsequently **solving the resulting differential equation**:

$$\dot{\alpha}_k(t) = \Gamma(t + \alpha_k(t)) \cdot \sum_{j=1}^{n_{osc}} y_{kj} \cdot V_o(t + \alpha_j(t)). \quad \textbf{(5)}$$

This equation is then easily solved by resorting to the definition of first-order derivative and, of course, discretizing the time dimension. After establishing an integration step $t_{step} = h$, in fact, we can partition the time frame t_{end} of our analysis into $n_{steps} = {}^{t_{end}}/_{t_{step}}$ uniform integration intervals, for each of which we will calculate the values of α_k and θ_k. The discretization allows us to turn the continuous derivative operator into a finite differentiation:

$$\dot{\alpha}_k(t) = \frac{d\alpha_k}{dt} = \lim_{h \to 0} \frac{\alpha_k(t+h) - \alpha_k(t)}{h} \xrightarrow{\text{discretization}} \dot{\alpha}_k(t_i) = \frac{\alpha_k(t_{i+1}) - \alpha_k(t_i)}{t_{step}} \quad \textbf{(6)}$$

To wrap it up, plugging (6) into (5) and operating some algebraic manipulations yields the **equation that the iterative kernel constituting the core of this project will solve**:

$$\alpha_k(t_{i+1}) = \alpha_k(t_i) + t_{step}\left(\Gamma(t_{i+1} + \alpha_k(t_i)) \cdot \sum_{j=1}^{n_{osc}} y_{kj} \cdot V_o(t_{i+1} + \alpha_j(t_i))\right) \quad \textbf{(7)}$$

$$\forall k, i: 1 \leq k \leq n_{osc} \wedge 1 \leq i < n_{steps}$$

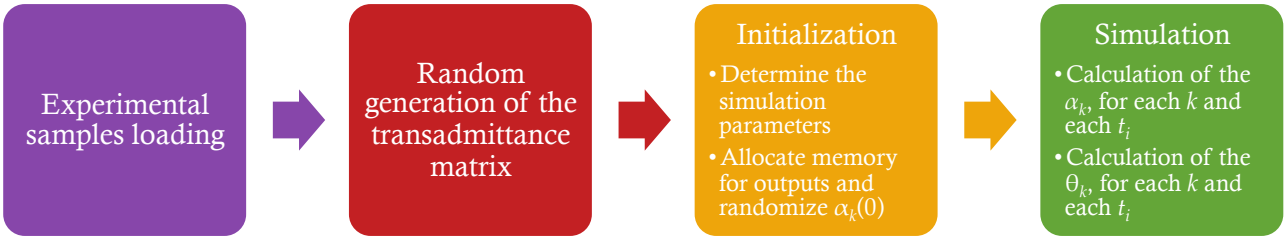
where the $\alpha_k(t_0 = 0)$ are randomly generated in the interval $[0; T_k]$, T_k in turn being the period of the k-th oscillator.

2.3 Program Overview

This Section illustrates the structure of the programs subject to the performance comparisons of this project, along with an analysis of the most noteworthy parts of the code and of the optimizations they underwent.

2.3.1 Structure of the MATLAB Program

The C and CUDA C/C++ translations of the code faithfully reproduce the structure and the execution flow of the original program. It is therefore useful to provide a flow diagram illustrating how the MATLAB code works.



Picture 10: summary of the execution flow in the MATLAB program.

a. *Experimental samples loading.* Γ and V_o are not identified by an analytical expression. Rather, they have been experimentally sampled, and their values stored in a `.mat` file whose contents have to be imported in the MATLAB workspace at the beginning of the program.
b. *Random generation of the transadmittance matrix.* As anticipated, in a real-world scenario, the values of the transadmittance parameters would depend on the way the oscillators are interconnected. In order to prove that the findings of the paper hold in general (i.e. they are not tied to a particular network structure), the Y matrix is randomized. More in detail, it is produced by a deterministic algorithm whose input is a random stencil containing +1's and -1's; said algorithm ensures that the main diagonal is entirely composed of zeros and that the matrix is symmetric.
c. *Initialization – part 1: oscillator array initialization.* Some parameters have to be set before the actual computation starts:
    i. *Frequency randomization.* Each oscillator's output period is randomized. The resulting periods are uniformly distributed in the $[0.9995T; 1.0005T]$ range, T being the period of the oscillators in free-running mode.
    ii. *Calculation of the integration parameters.* The final instant `tstop` is determined in compliance with the `n_per` parameter (which states the extension of the simulation time frame in terms of periods), along with the integration step `tstep` and the instants vector `time`, containing all time instants for which the simulation will be carried out.
    iii. *Noise generation (optional).* If the `IsNoisy` flag is set, this phase generates a random noise that will be added to the simulated α; otherwise, the noise is set to zero.
d. *Initialization – part 2: output initialization.* Memory is preallocated for the `alpha` and `theta` matrices. Then, the $\alpha_k(0)$ are randomly generated.
e. *Simulation.* Here lies the core of the program. An iterative kernel calculates the α_k by simply translating Equation (7) in Section 2.2; next, Equation (4) is applied to determine the total phases θ_k.

2.3.2 Organization of the C and CUDA Programs

The differences between the C and CUDA C/C++ programs are limited to the iterative kernels and to a few other technicalities (for example, the host-device data movements, or the allocation of the simulation parameters in duplicate – both on the host and on the device). For this reason, the architectures of the two applications largely coincide in terms of source files.

Table 1 lays out said architecture, briefly illustrating the contents of each source file.

Source file(s)	Found in Plain C	Found in CUDA	Description
`adm_matrix.h` **`adm_matrix.c(u)`**	✓	✓	Contain functions and macros related to the Y matrix. *Kernels*: `init_matrix` (initializes the matrix to zero), `random_pattern` (generates random pattern), `pattern_to_matrix` (modifies the matrix according to the pattern), `finalize_matrix` (multiplies each element by a magnitude constant).

Source file(s)	Found in Plain C	Found in CUDA	Description
`cuda_settings.h`		✓	Contains macros related to the block and grid sizes for the CUDA kernels.
`err.h`	✓	✓	Contains exit codes for the application.
`hpps_c.h` `hpps_c.c`	✓		Include the `main` function, which merely calls functions from other source files, plus a couple of other facilities.
`hpps_cuda.h` `hpps_c.cu`		✓	These files are the equivalent of the `hpps_c.*` files for the CUDA program.
`init_osc.h` `init_osc.c(u)`	✓	✓	Carry out the initialization of the oscillator array. *Kernels*: `randomize_periods` (randomizes each oscillator's output period), `generate_time_vector` (produces the vector containing the simulation instants).
`init_output.h` `init_output.c(u)`	✓	✓	Perform the output initialization phase. *Kernels*: `randomize_t0` (zeroes out the `alpha` and `theta` matrices and randomly generates the initial time-shifts).
`output_file.h` `output_file.c(u)`	✓	✓	Store the results of the simulation to a `.mat` file using MATLAB's MATFile APIs.
`rng.h` `rng.cu`		✓	Initialize the GPU's random number generator provided by the cuRAND library. Not needed in the Plain C version since the initialization only requires one function call to `srand()`, located in `hpps_c.c`. *Kernels*: `init_rng` (performs the initialization).
`samples.h` `samples.c(u)`	✓	✓	Use the aforementioned MATFile API to read the `.mat` file containing the V_o and Γ samples and import them into main memory.
`sim_parameters.h`	✓	✓	Provide a number of macros and variable declarations regarding the simulation parameters.
`simulate.h` `simulate.c(u)`	✓	✓	Contain the core of the programs, i.e. the kernels performing the actual simulation plus some auxiliary functions. *Kernels*: `simulate` (performs the simulation without noise), `simulate_noisy` (includes noise in the simulation of the time-shifts).

Table 1: structure of the C and CUDA C/C++ programs.

2.4 Code Optimizations

After completing a first working draft of both programs, some parts of their code were rewritten in order to enact some optimizations. The following subsections thoroughly illustrate the kind of betterments that were enacted and the reasons why they worked.

2.4.1 Both Versions: Use of the MATFile API

Using MATLAB's libraries proved very effective from the points of view of code maintainability, program flexibility and storage footprint.

Both versions of the original programs, in fact, had the `vo`, `gamma` and `instants` samples hard-coded into the `samples.c(u)` file – which was bad enough in that it didn't allow the program to simulate the behaviour of different oscillator models without changing the source code to accommodate the appropriate samples. Moreover, because each of them consisted in a 200-element array, that resulted in 600 `double` values to be stored on the stack.

What's more, the results of the simulation were written to three distinct files – `alpha.sim`, `theta.sim` and `time.sim` – in text mode. In light of the 10-digits precision required, each `double` was represented by 16 characters (`n.mmmmmmmmmme±xx`) and thus resulted in **17 bytes** per `double`, including the separator, being printed out to the output files. Using `.mat` files is way more convenient, in that every `double` is represented in its classical binary form and thus only takes up **8 bytes**: results subsequently take up **53% less disk space** than the original solution, besides being ready for a MATLAB import with a single `load` command (instead of three separate and much slower `csvread` commands).

2.4.2 Plain C Version: Polyhedral Optimizations With PLuTO

PLuTO is a source-to-source polyhedral optimizer [**9**]: it parses C source files for SCoPs, ports them to the polyhedral model, optimizes them according to the specified preference flags and outputs an optimized version of the same code. Among the code manipulations it offers, two are the ones we used:

- `--tile` reorganizes the loops that the SCoP consists in in order to improve data locality (as in *loop blocking*, Subsection 1.3.3);
- `--parallel` builds a data dependence graph and looks for the parts of a loop that can be executed in parallel by different threads/cores (as in *detection of parallel loops*, Subsection 1.3.3); in case of success, it adds compiler directives to realize loop parallelization.

```
for (unsigned int t = 1; t < n_steps; t++) {
  for (unsigned int k = 0; k < n_osc; k++) {
    //stuff
    for (unsigned int kk = 0; kk < n_osc; kk++)
      current += matrix[kk + k*n_osc] * vo_lut(…);
    //other stuff
  }
}
```

```
unsigned int t, k, kk;

#pragma scop
for (t = 1; t < n_steps; t++) {
  for (k = 0; k < n_osc; k++) {
    //stuff
    for (kk = 0; kk < n_osc; kk++)
      current += matrix[k][kk] * vo_lut(…);
    //other stuff
  }
}
#pragma endscop
```

Code snippet 10: the `simulate` kernel had to be modified as shown in order for PLuTO to recognize it as a SCoP. Note that the expression used as an array index for `matrix` in the original code was considered non-affine in parameters and loop counters (of the `k*n_osc` product), which required `matrix` to be restructured as a bidimensional array.

PLuTO requires candidate SCoPs to be explicitly marked by the programmer by enclosing them in the `#pragma scop` and `#pragma endscop` directives. Also, PLuTO is very strict for what concerns the definition of a SCoP: for example, it refuses loop counter declarations in loop headers and non-linear expressions in loop bounds and array indices. As a consequence, some kernels (along with other parts of the code) had to be rewritten to comply with PLuTO's constraints as shown in Code snippet 10.

Code snippet 11 contrasts the original `simulate` kernel with its optimized counterpart: the `#pragma omp` directives, which instruct the compiler on what parts of the code should be parallelized, only appear in relation with marginal parts of the code; the kernel's core, highlighted in red, will unfortunately not benefit from any of PLuTO's optimizations and is thus likely to feature minor performance improvements, if any.

```
for (unsigned int t = 1; t < n_steps; t++) {
  for (unsigned int k = 0; k < n_osc; k++) {
    current = 0.0;

    for (unsigned int kk = 0; kk < n_osc; kk++)
      current = current + matrix[kk + k * n_osc] *
                vo_lut(time[t] + alpha_prev[kk], periods[kk]);

    alpha_cur[k] = alpha_prev[k] + tstep *
                   gamma_lut(time[t] + alpha_prev[k], periods[k]) * current;
    theta_cur[k] = omega[k] * (time[t] + alpha_cur[k]);
  }

  alpha_prev = alpha_cur;
  alpha_cur = alpha_cur + n_osc;
  theta_cur = theta_cur + n_osc;
}
```

```
int t1, t2, t3, t4, t5, t6;
int lb, ub, lbp, ubp, lb2, ub2;
register int lbv, ubv;

/* Start of CLooG code */
if (n_osc >= 1) {
  lbp=0;
  ubp=n_osc-1;
  #pragma omp parallel for private(lbv,ubv,t3,t4,t5,t6)
  for (t2=lbp;t2<=ubp;t2++) {
    omega[t2] = 2 * PI / periods[t2];;
  }
}

if (n_osc >= 1) {
  for (t2=1;t2<=n_steps-1;t2++) {
    for (t4=0;t4<=n_osc-1;t4++) {
      current = 0.0;
      for (t6=0;t6<=n_osc-1;t6++) {
        current += matrix[t4][t6] *
                   vo_lut(time[t2] + alpha_prev[t6], periods[t6]);
      }
      alpha_cur[t4] = alpha_prev[t4] + tstep *
                      gamma_lut(time[t2] + alpha_prev[t4], periods[t4]) *
                      current;
    }
    alpha_prev = alpha_cur;
    lbp=0;
    ubp=n_osc-1;
    #pragma omp parallel for private(lbv,ubv,t5,t6)
    for (t4=lbp;t4<=ubp;t4++) {
      theta_cur[t4]= omega[t4] * (time[t2] + alpha_cur[t4]);
    }
    alpha_cur = alpha_cur + n_osc;;
    theta_cur = theta_cur + n_osc;;
  }
}
/* End of CLooG code */
```

Code snippet 11: comparison between the original `simulate` kernel and the source code produced by PLuTO.

2.4.3 CUDA Version: Memory Access Coalescing

One of the crucial factors for a CUDA kernel's performance is the memory access pattern. *Coalescing memory accesses* means to **ensure that adjacent threads access adjacent locations in**

memory when executing the same instruction, thus leading to the minimization of memory transactions and to the maximization of the use of a cache block.

```
for (t = 1; t < d_n_steps; t++)
  if (idx < d_n_osc)
    for (k = 0; k < d_n_osc; k++)
      current += d_matrix[k + idx * d_n_osc] * d_temp_vj[k]; //non-coalesced
```

```
for (t = 1; t < d_n_steps; t++)
  if (idx < d_n_osc)
    for (k = 0; k < d_n_osc; k++)
      current += d_matrix[idx + k * d_n_osc] * d_temp_vj[k]; //coalesced
```

Code snippet 12: the second solution yields better execution times because it coalesces memory accesses. Note: `idx` is the thread identifier. Remember that these pieces of code are executed simultaneously by n_{osc} threads!

The single most executed instruction in the entire CUDA program is located in the inner loop: `current`'s sum-assignment is in fact performed $n_{steps} \cdot n_{osc}^2$ times (i.e. $n_{steps} \cdot n_{osc}$ for each thread). The original code contained a non-coalesced memory access to `matrix`, as shown in Code snippet 12.

Albeit counterintuitive at first blush, accessing memory by columns instead of by rows proved the most effective solution: the reason for this is that what looks like a strided access (the by-column solution) is in fact a coalesced access when considering that multiple threads are reading all the elements of the same row at once (see Tables 2-3 for an example). To put it differently, the optimized code dramatically enhances both temporal and spatial locality, in that adjacent memory locations are accessed at the same time. The non-coalesced solution, on the contrary, requires the GPU to fetch one block of memory at once for each row of the matrix; depending on the size of the matrix (i.e. on how many oscillators are in the system), the cache might not be able to retain all these blocks, thus leading to cache thrashing.

	0	1	2	3	4
0	T_0, t_0	T_0, t_1	T_0, t_2	T_0, t_3	T_0, t_4
1	T_1, t_0	T_1, t_1	T_1, t_2	T_1, t_3	T_1, t_4
2	T_2, t_0	T_2, t_1	T_2, t_2	T_2, t_3	T_2, t_4
3	T_3, t_0	T_3, t_1	T_3, t_2	T_3, t_3	T_3, t_4
4	T_4, t_0	T_4, t_1	T_4, t_2	T_4, t_3	T_4, t_4

Table 2: non-coalesced memory accesses.

T_i, t_j means that thread i accesses the cell at time t_j. Cells of the same color are accessed at the same time.

	0	1	2	3	4
0	T_0, t_0	T_1, t_0	T_2, t_0	T_3, t_0	T_4, t_0
1	T_0, t_1	T_1, t_1	T_2, t_1	T_3, t_1	T_4, t_1
2	T_0, t_2	T_1, t_2	T_2, t_2	T_3, t_2	T_4, t_2
3	T_0, t_3	T_1, t_3	T_2, t_3	T_3, t_3	T_4, t_3
4	T_0, t_4	T_1, t_4	T_2, t_4	T_3, t_4	T_4, t_4

Table 3: coalesced memory accesses.

2.4.4 CUDA Version: Shared Memory Allocation for Temporary Data

As Code snippet 13 points out, the `simulate` kernel allocates an array of `doubles` of size n_{osc}, called `d_temp_vj`. At each iteration of the outer loop, each of the n_{osc} threads temporarily stores the value of the V_o function associated to that thread's oscillator. Said values need to be shared among all threads, in that the inner loop makes use of the entire array in every thread.

The original version of the CUDA program allocated the temporary array in global memory; one of the optimizations we carried out consisted in **moving `d_temp_vj` to each block's shared memory area**, so as to benefit from lower memory latencies.

```
double * d_temp_vj;
cudaMalloc(&d_temp_vj, sizeof(double) * s_block.x * s_grid.x);
simulate<<<s_grid, s_block>>>(…, d_temp_vj);

__global__ void simulate(…, double * d_temp_vj) {
  //stuff
  for (unsigned int t = 1; t < d_n_steps; t++) {
    d_temp_vj[idx] = vo_lut(…);
    //other stuff
  }
}
```

```
simulate<<<s_grid, s_block, sizeof(double) * h_n_osc>>>(…);

__global__ void simulate(…) {
  //stuff
  extern __shared__ double d_temp_vj[];

  for (unsigned int t = 1; t < d_n_steps; t++) {
    d_temp_vj[idx] = vo_lut(…); //used in the same way as before
    // other stuff
  }
}
```

Code snippet 13: comparison between the original program, where `d_temp_vj` was allocated in global memory, and the shared-memory optimization. As is evident, the only differences concern the declaration and allocation of the array, but not its use.

One drawback of this solution is that it **limits the number of oscillators that the program can simulate**: because the amount of shared memory per block is limited to 48 kilobytes, the `d_temp_vj` array cannot contain more than $\frac{48 \cdot 1024\ [\mathrm{B}]}{8\ [\mathrm{B/oscillator}]} = 6144$ oscillators (each oscillator contributing to the array with an 8-byte `double` value).

2.4.5 CUDA Version: Constant Memory Allocation for Samples

Since it was clear that major performance gains could come from memory-related enhancements, the next step in the optimization process was to spot other frequently accessed data that could benefit from a more appropriate memory space location.

It was the case of the samples loaded at the beginning of the program. The `vo`, `gamma` and `instants` arrays are accessed $2 \cdot n_{\mathrm{steps}}$ times per thread ($2 \cdot n_{\mathrm{steps}} \cdot n_{\mathrm{osc}}$ in total); furthermore, the access pattern is inherently random, which makes global memory extremely inefficient at storing these data.

```
__device__ double d_vo[N_SAMPLES+1];
__device__ double d_gamma[N_SAMPLES+1];
__device__ double d_instants[N_SAMPLES];
```

```
__constant__ double d_vo[N_SAMPLES+1];
__constant__ double d_gamma[N_SAMPLES+1];
__constant__ double d_instants[N_SAMPLES];
```

Code snippet 14: moving a variable allocated in global memory is as simple as changing its declaration qualifier.

The choice was therefore made to move the samples to constant memory, since the values of the samples never change throughout the execution of the program (and even more so during the `simulate` kernel) and the 64 kilobytes

offered by the constant memory space comfortably accommodate the 600 `doubles` that the samples consist in. All we had to do was to change the arrays' declaration qualifier in the `samples.cu` file, as proven by Code snippet 14.

2.4.6 Backward Compatibility of the CUDA Code

When developing CUDA code, it is important to keep the need for backward compatibility into account. As discussed in Subsection 1.2.4, in fact, the availability of certain features on a given device varies with its own Compute Capability; the dimensionality and size of the thread blocks, the amount of available shared and global memory, and many other factors must therefore be planned in advance, in order to make sure that the target device(s) are fit to run the final executables.

In order for the products of this project to be compatible with as many CUDA GPUs as possible, the initial intention was to target devices with Compute Capability greater than or equal to 1.3 (dating back to 2008). Receding to previous Compute Capabilities was deemed too limiting, in light of the inability of said devices to handle double-precision floating-point numbers (`doubles` are demoted to `floats` when compiling with the `-arch=compute_11` or `-arch=compute_12` flags).

As soon as the development phase started, however, the convenience of a Compute Capability value of at least 2.0 emerged. It turned out, in fact, that the `nvcc` compiler did not feature a linker for programs compiled for CC 1.3 or earlier. In layman's words, compiling for such legacy architectures would force the programmer to stuff all device code in one single compilation unit (i.e. physical source file), thus compromising the modularization of the source code and its clarity.

In view of these stark limitations, and considering that Fermi-based GPUs (i.e. those featuring CC 2.0) hit the market in March 2010, the decision was made to **leave CC 1.3 (and earlier) devices behind and target CC 2.0**.

3 Results and Conclusions

3.1 Development and Testing Environment

3.1.1 Software

The development phase of the C and CUDA C/C++ programs required the following software:

- **MATLAB R2014b**, in order to run and experiment with the code provided by professor Maffezzoni and to exploit the MATFile APIs as described in Subsection 2.4.1;
- **PLuTO v0.11.2**, in order to carry out polyhedral optimizations on the plain C source code;
- **CUDA Toolkit v6.5**, which comprises the drivers and software facilities needed to develop CUDA-powered applications, including:
  - the *CUDA Samples*, a collection of ready-to-compile sample programs, among which the `deviceQuery` and `bandwidthTest` utilities;
  - *Nsight Eclipse Edition*, a customized version of the Eclipse IDE supporting syntax highlighting and checking for CUDA programs, compilation settings for the `nvcc` compiler toolchain, and much more;
  - `nvprof`, i.e. the CUDA profiler, together with the *Nvidia Visual Profiler*, providing a GUI to `nvprof`.

3.1.2 Hardware

Testing was carried out on a desktop computer, whose hardware configuration is laid out in Table 2. Table 3 and Table 4, in particular, detail the technical specifications of the testing machine's CPU and GPU.

Desktop Specifications	
CPU	Intel® Core™ i7-2600K
Memory	4 x 2 GB, DDR3 @ 1333MHz
Motherboard	Asus® Maximus IV Extreme-Z
GPU	Nvidia® GeForce® GTX 770
Operating System	Ubuntu 14.04.1 LTS

Table 2: testing machine specifications.

Intel® Core™ i7-2600K Specifications [12]	
Physical Cores	4
Total Simultaneous Threads	8 (2 per physical core)
Clock Frequency	3.4 GHz
Maximum Turbo Frequency	3.8 GHz
Maximum Memory Bandwidth	21 GB/s
Caches [13]	32 kB L1 Data Cache *per core* 32 kB L1 Instruction Cache *per core* 256 kB L2 Unified Cache *per core* 8 MB L3 *shared* Unified Cache
Instruction Set Extensions	Intel® Streaming SIMD Extensions 4.1 (SSE 4.1) Intel® Streaming SIMD Extensions 4.2 (SSE 4.2)

Table 3: Intel® Core™ i7-2600K Technical Specifications.

Nvidia® GeForce® GTX 770 Specifications [14]	
General Specifications	
Clock Frequency	1202 MHz[1]
Available Memory	2048 MB GDDR5 @ 3505 MHz
Memory Bus Width	256 bits
Maximum Memory Bandwidth	224.3 GB/s
Caches [15]	16 ÷ 48 kB[2] L1 Data Cache *per SM* 512 kB L2 Data Cache *shared among all SMs*
CUDA Specifications	
Compute Capability	3.0
Streaming Multiprocessors	8
CUDA Cores per SM	192
Total CUDA Cores	1536
Maximum Threads per Block	1024
Maximum Threads per SM	2048
Maximum Block Dimensions	1024 x 1024 x 64 threads
Maximum Grid Dimensions	2147483647 x 65535 x 65535 blocks
Maximum Resident Blocks per SM	16
Constant Memory	64 kB
Shared Memory	16 ÷ 48[2] kB *per block*
Total Registers per Block	65536
Concurrent Copy-Kernel Execution	Yes, 1 Copy Engine

Table 4: Nvidia® GeForce® GTX 770 GPU Technical Specifications.

[1] The value returned by the `deviceQuery` utility differs from Nvidia's official specifications.

[2] The L1 cache resides on the same chip as shared memory: they add up to 64 kB which can be dynamically reassigned to the two kinds of memory depending on the needs of each kernel. The CUDA Runtime API provides three different settings: `cudaFuncCachePreferShared` (allocates 16 kB to L1 cache and 48 KB to shared memory), `cudaFuncCachePreferEqual` (allocates 32 kB to both L1 cache and shared memory), `cudaFuncCachePreferL1` (allocates 48 kB to L1 cache and 16 kB to shared memory), `cudaFuncCachePreferNone` (expresses no preference, i.e. resorts to the last used configuration or to the default one, which is `cudaFuncCachePreferShared`).

3.1.3 *Additional Settings*

The CUDA Drivers include a **run time limit on kernels**, informally called "watchdog timer", preventing single kernel calls from taking more than 5 seconds to run. When a kernel takes longer to complete, the driver simply kills it and aborts the program.

While the watchdog timer is trickier to circumvent on Windows, the Linux drivers dynamically switch the timer on and off according to whether a display manager is running and a monitor is plugged into the GPU; the output of the `deviceQuery` utility confirms the automatic switch.

```
$ ./deviceQuery
[…]
Run time limit on kernels: Yes
```

Printout 1: the watchdog timer is enabled when a monitor is plugged into the GPU and/or a window server (like X) is running.

```
$ ./deviceQuery
[…]
Run time limit on kernels: No
```

Printout 2: unplugging all monitors and switching off X disables the time limit.

Because heavy workloads cause our kernel to run longer than the timer allows, gathering the measurements would have proven very problematic; at the same time, it would have obviously been impossible to operate on a monitorless machine. In order for the run time to be lifted, **the testing phase was therefore entirely conducted via SSH**.

3.2 Performance Measurements

This Section illustrates the results of the testing phase. Several binary files resulted from the various subversions of the code and from the compilation settings used to produce them; Table 5 provides a comprehensive list of the executable files and illustrates their differences.

Executables Prospect

Plain C versions

Executable	Output	Polyhedral	Reference
hpps-c-noMAT_O0 **hpps-c-noMAT_O3**	text	✗	Subsection 2.4.1
hpps-c-MAT_O0 **hpps-c-MAT_O3**	`.mat`	✗	Subsection 2.4.1
hpps-c-pluto	`.mat`	✓	Subsection 2.4.2

CUDA C/C++ versions

Executable	Output	Coalesced	Shared memory	Constant memory	Ref.
hpps-cuda-noMAT_O0 **hpps-cuda-noMAT_O3**	text	✓	✗	✗	Subs. 2.4.1
hpps-cuda-MAT_O0 **hpps-cuda-MAT_O3**	`.mat`	✓	✗	✗	Subs. 2.4.1
hpps-cuda-non-coalesced	`.mat`	✗	✗	✗	Subs. 2.4.3
hpps-cuda-shmem	`.mat`	✓	✓	✗	Subs. 2.4.4
hpps-cuda-constant-samples	`.mat`	✓	✗	✓	Subs. 2.4.5
hpps-cuda-shmem-constant-samples	`.mat`	✓	✓	✓	Subs. 2.4.4 2.4.5

OUTPUT = format of the output file(s). POLYHEDRAL = uses polyhedral optimizations. COALESCED = features coalesced memory accesses. SHARED MEMORY = allocates the `d_temp_vj` array to shared memory. CONSTANT MEMORY = allocates the samples to constant memory.

Table 5: prospect of the executables used for the performance measurement phase.

Some of the subversions listed in Table 5 consist of two binaries, named `*_O0` and `*_O3`, because they were compiled with different sets of compilation settings. Namely:

- the `_O0` binaries have been compiled with the `-O0` flag, meaning that no static optimizations have been performed by the compiler;
- the `_O3` programs, on the other hand, were produced by passing the `-O3 --use_fast_math` flags to the compiler, in order for code transformations to be applied;
- the other executables were compiled with the same settings as the `_O3` binaries.

3.2.1 noMAT vs. MAT: Storage Footprint Comparison

A comparison was made between the sizes of the output files of the `noMAT` versions and those of the `.mat` file generated by the `MAT` executables. The `_O3` CUDA binaries were arbitrarily chosen to produce these data (`_O0` versions, or plain C programs, would yield identical results).

Parameters	noMAT	MAT	MAT/noMAT Gain
60 oscillators, 1000 periods	50.08 MiB	18.94 MiB	**62.2%**
256 oscillators, 1000 periods	212.66 MiB	72.52 MiB	**65.9%**
512 oscillators, 1000 periods	424.96 MiB	104.28 MiB	**75.5%**
1024 oscillators, 1000 periods	849.10 MiB	275.68 MiB	**67.5%**

Table 6: noMAT vs. MAT storage footprint comparison.

Table 6 substantiates the claims made in Subsection 2.4.1: printing the outputs to `.mat` files shrinks their footprint by more than 50%; the observed gain is even better than expected, averaging at 69%. For this reason, the `noMAT` versions were discarded and every subsequent enhancement was made to the MATFile-complying code.

3.2.2 -O0 vs. -O3: Execution Time Comparison

Table 7 and Table 8 contain the execution times of all `noMAT` and `MAT` versions of the programs: a total 8 binaries were therefore tested. Furthermore, the executables were run multiple times in order to evaluate their response to different workload configurations. The speedup granted by `-O3` over `-O0` was then calculated as usual:

$$\text{Speedup} = \frac{\text{Performance with -O3}}{\text{Performance with -O0}} = \frac{1/t_{\text{ex, -O3}}}{1/t_{\text{ex, -O0}}} = \frac{t_{\text{ex, -O0}}}{t_{\text{ex, -O3}}}$$

Parameters		hpps-c-noMAT		-O3/-O0 Speedup	hpps-c-MAT		-O3/-O0 Speedup
n_{osc}	n_{per}	_O0	_O3		_O0	_O3	
60	*1000*	4.413 s	2.731 s	1.62x	3.656 s	1.923 s	1.90x
256	*1000*	59.39 s	29.81 s	1.99x	57.99 s	25.47 s	2.28x
512	*1000*	160.4 s	100.1 s	1.60x	162.7 s	95.79 s	1.70x
1024	*1000*	545.1 s	381.5 s	1.43x	637.7 s	364.3 s	1.75x
Average speedup				***1.50x***			***1.77x***

Table 7: execution times and speedups of the plain C noMAT and MAT versions.

Parameters		hpps-cuda-noMAT		-O3/-O0 Speedup	hpps-cuda-MAT		-O3/-O0 Speedup
n_{osc}	n_{per}	_O0	_O3		_O0	_O3	
60	*1000*	2.978 s	2.948 s	1.01x	2.046 s	2.037 s	1.00x
256	*1000*	9.039 s	9.01 s	1.00x	5.289 s	5.256 s	1.01x
512	*1000*	15.94 s	16.04 s	0.99x	11.00 s	10.68 s	1.03x
1024	*1000*	27.95 s	29.68 s	0.94x	29.04 s	27.39 s	1.06x
Average speedup				***0.97x***			***1.04x***

Table 8: execution times and speedups of the CUDA noMAT and MAT versions.

These data suggest there's a clear-cut difference between the impact of the compilation flags for plain C programs and the influence they have on the CUDA binaries. Table 7, in fact, points out consistent speedups for the plain C versions, while the elaboration of the data from Table 8 reveals a substantial ineffectiveness of the `-O3` optimizations for the CUDA code.

It is easy to glean the reasons behind this behaviour: while `gcc` applies the `-O3` optimizations to the whole program – thus including the iterative kernels it is composed of – the same flag **only applies to host code** when passed to `nvcc`. As a consequence, `–O3` static optimizations have negligible repercussions on CUDA programs because they act on marginal parts of the source code whose execution time share is imperceptible in comparison with that of the kernels.

To wrap it up, `-O3` optimizations **definitely have a positive impact on the performance of plain C code** – with average time gains ranging from 33% to 43% – while they proved **useless on CUDA** (given their modesty, the fluctuations in the average speedups can sensibly be ascribed to physiological variations of the execution time, rather than to the compiler flags).

3.2.3 Plain C Version: PLuTO

Table 9 details the time it took for the `pluto` version to complete with the usual variety of parameters, in comparison with the "baseline" performance, i.e. `hpps-c-MAT_O3`.

Parameters		hpps-c-MAT_O3	hpps-c-pluto	pluto/-O3 Speedup
n_{osc}	n_{per}			
60	*1000*	1.923 s	1.886 s	1.02x
256	*1000*	25.47 s	25.67 s	0.99x
512	*1000*	95.79 s	97.13 s	0.99x
1024	*1000*	364.3 s	402.0 s	0.91x
Average speedup				***0.93x***

Table 9: execution time comparison between the MAT_O3 and pluto versions of the plain C executable.

PLuTO's optimizations had no positive effect on the performance of the program: we saw that coming in Subsection 2.4.2. The negative impact might seem more puzzling, but it is probably due to the parallelization overhead which doesn't match a significant performance increase.

3.2.4 CUDA Version: Coalesced Memory Accesses

The `hpps-cuda-bad` executable, whose performance will act as the baseline for all subsequent performance comparisons, was run with multiple combinations of input parameters and its execution times were then compared to those observed when running `hpps-cuda-MAT_O3`.

Parameters		hpps-cuda-bad		hpps-cuda-MAT_O3		-O3/bad	-O3/bad
n_{osc}	n_{per}	global	kernel	global	kernel	global Speedup	kernel Speedup
60	*1000*	2.631 s	855.78 ms	2.037 s	467.72 ms	1.29x	1.83x
256	*1000*	8.011 s	4.634 s	5.256 s	1.60 s	1.52x	2.90x
512	*1000*	19.24 s	12.35 s	10.68 s	3.888 s	1.80x	3.18x
1024	*1000*	45.95 s	25.35 s	27.39 s	7.685 s	1.68x	3.30x
512	*100*	2.366 s	1.25 s	1.553 s	400.88 ms	1.52x	3.19x
1024	*100*	4.763 s	2.51 s	3.017 s	780.40 ms	1.58x	3.22x
5120	*100*	51.59 s	44.63 s	11.50 s	4.599 s	4.49x	9.70x
10240	*100*	229.8 s	216.0 s	26.53 s	12.91 s	8.66x	16.73x
***Average speedup** (excluding (10240, 1000))*						***2.19x***	***4.72x***

Table 10: `hpps-cuda-bad` execution times and comparison with `hpps-cuda-MAT_O3`. "Global" indicates the time it took for the complete execution, "kernel" reports the execution time of the simulate kernel alone.

Table 10 reports the comparison described above. Although what counts is undoubtedly the global execution time, we also assessed the impact of our enhancements on the `simulate` kernel alone.

The speedups obtained by access coalescing clearly demonstrate that **careful engineering of a program's memory interactions is a key factor for performance**. Execution times for the whole program shrunk by an average 54%; at the same time, the kernel's average execution time gain amounts to a stunning 79%.

Furthermore, the results highlight a **positive trend of the speedups as the number of oscillators in the simulation grows**: the solution with the highest n_{osc} benefited from an 88% (global) and 94% (kernel) reduction of the execution time. To that end, for the sake of comparability, the average speedups were computed without keeping the execution times for the (10240, 100) configuration into account: failing to do so would have made `shmem`'s speedups incomparable with the present ones (`shmem` cannot be run for $n_{osc} > 6144$).

3.2.5 CUDA Version: Access Coalescing + Shared Memory

The same experiments were repeated for the `hpps-cuda-shmem` version.

Parameters		hpps-cuda-bad		hpps-cuda-shmem		shmem/bad	shmem/bad
n_{osc}	n_{per}	global	kernel	global	kernel	global Speedup	kernel Speedup
60	*1000*	2.631 s	855.78 ms	1.92 s	332.81 ms	1.37x	2.57x
256	*1000*	8.011 s	4.634 s	4.748 s	1.041 ms	1.69x	4.45x
512	*1000*	19.24 s	12.35 s	9.428 s	2.59 s	2.04x	4.77x
1024	*1000*	45.95 s	25.35 s	23.00 s	5.154 s	2.00x	4.92x
512	*100*	2.366 s	1.25 s	1.399 s	266.70 ms	1.69x	4.69x
1024	*100*	4.763 s	2.51 s	2.518 s	523.05 ms	1.89x	4.80x
5120	*100*	51.59 s	44.63 s	12.83 s	5.424 s	4.02x	8.23x
10240	*100*	229.8 s	216.0 s	--	--	--	--
Average speedup						***2.41x***	***5.97x***

Table 11: `hpps-cuda-shmem` execution times and comparison with `hpps-cuda-bad`. "Global" indicates the time it took for the complete execution, "kernel" reports the execution time of the simulate kernel alone.

In compliance with the limitations stated in Subsection 2.4.4, it was not possible to run the `shmem` executable with the (10240, 100) configuration.

The combined effect of coalesced accesses and shared memory usage yields average **performance increments 59%**; the **kernel alone benefits from an average 83%** execution time reduction, instead.

Another fact emerging from the gathered data is that **`shmem`'s performance degrades** with respect to `MAT_O3`'s **for** n_{osc} big enough. As n_{osc} increases, so does the number of cells of the `d_temp_vj` that a single thread has to fill (retrieving temporary data is an $O(n_{osc} \cdot n_{steps})$ operation). Because the kernel is executed in blocks of 512 threads each, simulating 1024 oscillators means that each thread fetches and stores 2 values per time step; simulating 5120 implies that every thread has to perform that same job 10 times *per time step*. This does not apply to the `MAT` version, in that each thread only deals with the value pertaining to the oscillator it represents (it is an $O(n_{steps})$ operation).

3.2.6 CUDA Version: Access Coalescing + Constant Memory

Parameters		`hpps-cuda-bad`		`hpps-cuda-const`		`const/bad`	`const/bad`
n_{osc}	n_{per}	global	kernel	global	kernel	global Speedup	kernel Speedup
60	*1000*	2.631 s	855.78 ms	1.977 s	372.03 ms	1.33x	2.30x
256	*1000*	8.011 s	4.634 s	4.974 s	1.119 s	1.61x	4.14x
512	*1000*	19.24 s	12.35 s	9.475 s	2.611 s	2.03x	4.73x
1024	*1000*	45.95 s	25.35 s	26.66 s	5.102 s	1.72x	4.97x
512	*100*	2.366 s	1.25 s	1.374 s	274.44 ms	1.72x	4.55x
1024	*100*	4.763 s	2.51 s	2.762 s	523.52 ms	1.72x	4.79x
5120	*100*	51.59 s	44.63 s	11.09 s	3.759 s	4.65x	11.87x
10240	*100*	229.8 s	216.0 s	26.47 s	12.431 s	8.68x	17.38x
***Average speedup** (excluding (10240, 100))*						***2.31x***	***6.66x***

Table 12: `hpps-cuda-const` execution times and comparison with `hpps-cuda-bad`. "Global" indicates the time it took for the complete execution, "kernel" reports the execution time of the simulate kernel alone.

The experimental findings show that there is **little difference between the speedups obtained with shared memory and those granted by constant memory**. Actually, further analysis of the execution times highlights that `const`'s speedups are consistently higher than `shmem`'s in correspondence with the (5120, 100) configuration, which is perfectly compatible with the considerations we made in the previous Subsection: because `const` manages temporary data inside the kernel in the same way as `MAT_O3`, it doesn't suffer from `shmem`'s setback.

3.2.7 CUDA Version: Access Coalescing + Shared Memory + Constant Memory

The final version of the CUDA program features every optimization we could implement.

Parameters		hpps-cuda-bad		hpps-cuda-shmem-const		shmem-const/bad global Speedup	shmem-const/bad kernel Speedup
n_{osc}	n_{per}	global	kernel	global	kernel		
60	*1000*	2.631 s	855.78 ms	1.917 s	341.23 ms	1.37x	2.51x
256	*1000*	8.011 s	4.634 s	4.691 s	1.043 s	1.71x	4.44x
512	*1000*	19.24 s	12.35 s	9.455 s	2.593 s	2.03x	4.76x
1024	*1000*	45.95 s	25.35 s	24.44 s	5.196 s	1.88x	4.88x
512	*100*	2.366 s	1.25 s	1.363 s	266.12 ms	1.74x	4.70x
1024	*100*	4.763 s	2.51 s	2.482 s	523.98 ms	1.92x	4.79x
5120	*100*	51.59 s	44.63 s	12.85 s	5.494 s	4.01x	8.12x
10240	*100*	229.8 s	216.0 s	--	--	--	--
Average speedup						***2.35x***	***5.92x***

Table 13: `hpps-cuda-shmem-const` execution times and comparison with `hpps-cuda-bad`. "Global" indicates the time it took for the complete execution, "kernel" reports the execution time of the simulate kernel alone.

Combining `shmem`'s and `const`'s enhancements didn't work out as well as expected. **Performance largely remains unchanged** with respect to the `shmem` and `const` versions: evidently, either of the two optimizations is enough to max out the speedup we can obtain from a wise exploitation of the CUDA memory hierarchy. The code has been optimized to the point where **memory is no longer the bottleneck**.

3.2.8 MATLAB vs. Plain C vs. CUDA: Execution Time Wrap-Up Comparison

The following wrap-up comparison justifies the work done for this project. For each major version, the best-performing subversion (i.e. `hpps-c-MAT_O3` and `hpps-cuda-const`) was chosen.

Parameters		MATLAB	hpps-c-MAT_O3	hpps-cuda-const	C/MATLAB Speedup	CUDA/MATLAB Speedup
n_{osc}	n_{per}					
60	*1000*	3.275 s	1.923 s	1.977 s	1.70x	1.66x
256	*1000*	13.35 s	25.47 s	4.974 s	0.52x	2.68x
512	*1000*	27.97 s	95.79 s	9.475 s	0.29x	2.95x
1024	*1000*	77.94 s	364.3 s	26.66 s	0.21x	2.92x
512	*100*	2.837 s	9.543 s	1.374 s	0.29x	2.06x
1024	*100*	7.950 s	36.40 s	2.762 s	0.22x	2.88x
5120	*100*	106.7 s	949.6 s	11.09 s	0.11x	9.62x
10240	*100*	375.0 s	4151 s	26.47 s	0.09x	14.17x
***Average speedup** (including (10240, 100))*					***0.11x***	***7.25x***

Table 14: performance comparison between the MATLAB, plain C, and CUDA versions of the code.

`-O3` optimizations notwithstanding, the **plain C version performs incomparably worse** than the original MATLAB program does. This is probably due to MATLAB's built-in support for vector instructions and thread parallelization, which causes the purely serial C binary to take 809% more time to complete on average (the (10240, 100) configuration took more than *one hour* to complete!).

On the other hand, **the CUDA version yields substantial speedups** against the MATLAB program: while it guarantees very good gains in terms of execution time for $60 \leq n_{osc} \leq 1024$

(40% to 66% less time), even more impressive results emerge for greater quantities of oscillators (where CUDA proves 10 to 14 times faster than MATLAB). A careful analysis of the reasons behind these results is provided in Section 3.3.

3.2.9 MATLAB vs. Plain C vs. CUDA: Output Precision

As anticipated in Section 2.1, we also intended to assess the computational error made by the C and CUDA versions with respect to MATLAB's output. Because of the random number generations involved in some parts of the code, resulting in the three versions simulating different oscillator networks, running the programs with the same input configurations wasn't enough to obtain comparable outputs. The code was therefore slightly modified to replace all calls to RNGs with deterministic expressions.

The three binaries (`MATLAB`, `hpps-c-MAT_O3` and `hpps-cuda-const`) were launched with the (60, 1000) parameter configuration; we then calculated the absolute and relative errors made by the C and CUDA programs with respect to MATLAB.

Output variable	Average absolute error		Average relative error	
	Plain C	CUDA C/C++	Plain C	CUDA C/C++
alpha	$2.043 \cdot 10^{-13}$	$2.043 \cdot 10^{-13}$	0.029%	0.029%
theta	$1.3 \cdot 10^{-3}$	$1.3 \cdot 10^{-3}$	0.00009%	0.00009%
time	0	0	0%	0%

Table 15: absolute and relative errors of the C and CUDA outputs with respect to the results of the MATLAB version.

According to the results, the gap between our programs and the original one is definitely negligible. It is thus safe to state that the **performance improvements** introduced by the above optimizations **did not harm the outputs' precision**.

3.3 Conclusions: What Has Worked and What Hasn't

The figures exposed in the previous Section sparked several observations.

As far as the C program is concerned, we can draw the following conclusions:

- The Plain C version **performed way worse than the MATLAB code**, probably due to MATLAB's native support to vectorization and parallelization.
- The **`-O3` compilation flag is quite effective** on C iterative kernels: the `MAT_O3` version benefited from a 43% average reduction of the execution time in comparison with the `MAT_O0` executable.
- **PLuTO didn't have any positive effect** on performance, as expected: its output (Subsection 2.4.2) was a clear symptom that the `simulate` kernel would not benefit from parallelization or tiling, at least not more than what the `-O3` flag already granted.

The most interesting facts we noticed have to do with CUDA, though:

- The program definitely took advantage of the GPGPU computational paradigm: it proved **up to 93% faster than its MATLAB counterpart**, 86% on average.

- Making **wise use of CUDA's memory hierarchy** – coalescing memory accesses and exploiting shared and constant memory – turned out to be vital to an application's performance, as demonstrated in Subsections 3.2.4, 3.2.5, and 3.2.6.
- **`-O3` compiler optimizations are ineffective** on CUDA, since they only target host code while most of the computation takes place on the device.

3.3.1 Computational Complexity of the Algorithm

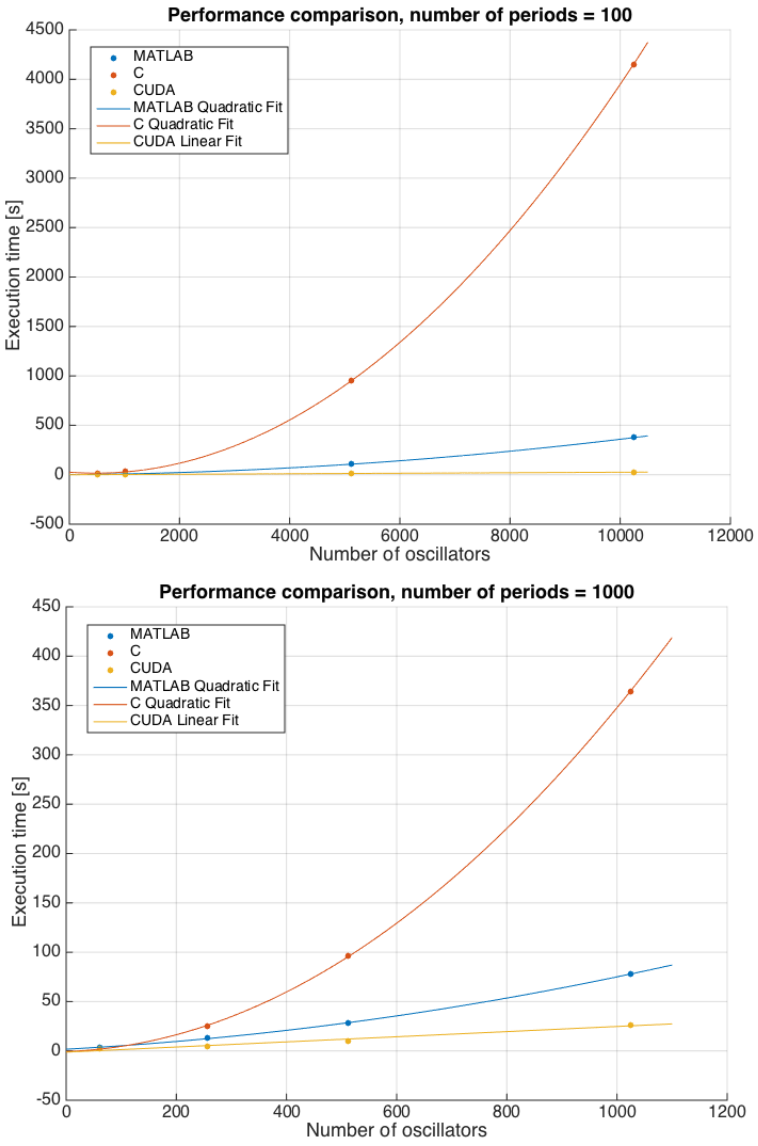
One could argue that the results we obtained are not that impressive, considering that GPGPU speedups lie around the mid-hundreds in some cases. Even though said figures are arguable most of the times [**16**], the algorithm itself also plays a key role: some programs are very prone to parallelization, others feature data dependences preventing disruptive optimizations (or even preventing it at all).

In our case, the `simulate` kernel consists in three nested loops:

- an outer loop iterating $\forall t: 1 \leq t \leq n_{\text{steps}}$;
- a middle loop iterating $\forall k: 0 \leq k < n_{\text{osc}}$;
- an inner loop iterating $\forall kk: 0 \leq kk < n_{\text{osc}}$.

Therefore, as already discussed in Subsection 3.2.5, the kernel's computational complexity in the MATLAB and C versions is $\mathbf{O}(\boldsymbol{n_{\text{steps}} \cdot n_{\text{osc}}^2})$. The reason why **the CUDA version** dramatically reduces the program's execution time is that it **downgrades the computational complexity** of the algorithm: the parallelization of the middle loop, whose function is to perform the simulation for each oscillator in the system, lowers the complexity to $O(n_{\text{steps}} \cdot n_{\text{osc}})$, i.e. **from quadratic (with respect to $\boldsymbol{n_{\text{osc}}}$) to linear**.

Picture 11 proves this hypothesis: the C program definitely shows a quadratic behaviour, while the CUDA program fits a linear interpolant; MATLAB's curve still features a parabolic shape, albeit less immediate since the automatic



Picture 11: plots of the execution time of the various versions of the program, as a function of the number of oscillators to simulate. Markers represent the actual measurements, solid lines constitute the interpolant polynomials.

vectorization and parallelization that take place under the hood reduce the magnitude of the second-order coefficient.

There's room for further improvements, as detailed in Section 3.4, but **the lower bound for this algorithm's complexity is surely $\mathbf{O}(n_{\mathbf{steps}})$**: because each iteration of the outer loop requires all results from the previous one to be available, there's no way that the simulation of different time steps will ever be parallelized.

3.3.2 Hardware Limitations

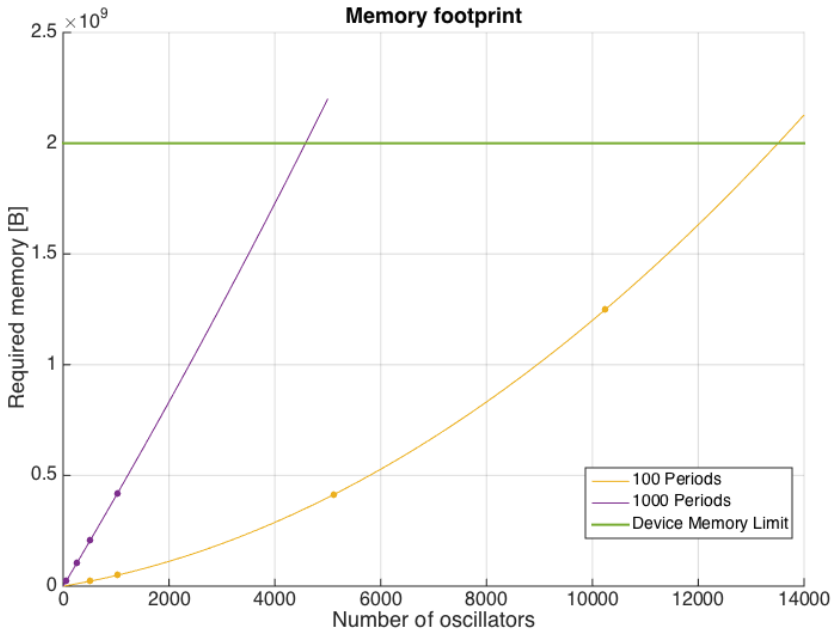
The most interesting conclusions we drew in the previous Sections stemmed from an observation of the trends of the execution time as n_{osc} grew. To that end, the (5120, 100) and (10240, 100) configurations were the most significant. One could therefore wonder why we didn't persist with these measurements and test the programs for even higher values of n_{osc}.

At first, the simulation only consisted in the four (*, 1000) configurations. We were then able to increase n_{osc} at the cost of reducing n_{per} to 100. The limited memory on the device was the main constraint we had to deal with: for any given parameter configuration, the amount of memory required is a function of n_{osc} and n_{per}:

$$\text{Required memory} \sim \text{size}_Y + \text{size}_\alpha + \text{size}_\theta = \text{size}_{\text{double}} \cdot (n_{\text{osc}}^2 + 2 \cdot n_{\text{osc}} \cdot n_{\text{steps}}) \text{ [B]}$$

Since $\text{size}_{\text{double}} = 8$ B and $n_{\text{steps}} = 25 \cdot n_{\text{per}}$, we get:

$$\text{Required memory } (n_{\text{osc}}, n_{\text{per}}) = 8n_{\text{osc}}^2 + 400 n_{\text{osc}} n_{\text{per}} \text{ [B]} \quad \textbf{(8)}$$



Picture 12: memory footprint of the program.

For n_{osc} big enough, the term driving the amount of memory required by the program is definitely $8n_{\text{osc}}^2$: reducing the number of periods is therefore ineffective on the long run. **Device memory constraints** are thus **the only reason why we stopped our simulations at 10240 oscillators** (even though we could have ventured somewhat further, as the graph suggests); Section 3.4 describes one possible way to circumvent this limitation.

3.4 Further Developments

As hypothesized in Subsection 3.2.7, we maxed out the speedups we could obtain by working on memory-related optimizations; plus, as suggested in the previous Subsection, the program is unable to simulate more than some thousands oscillators at the moment. The following paragraphs explain how we could seek additional speed bumps for the CUDA program and how to dismiss the limitations due to the kernel's memory footprint.

3.4.1 CUDA Dynamic Parallelism

Devices of Compute Capability 3.5 or greater feature *Dynamic Parallelism*: this functionality **allows kernels to call other kernels**, i.e. `__global__` functions can call other functions making use of the usual triple chevron construct. Running threads can thus "spawn" other threads to perform parallel work.

Future work on the algorithm would thus consist in implementing the inner loop of the `simulate` kernel in an external `current` kernel: every `simulate` thread would then call `current` instead of looping over the kk index n_{osc} times. Because the inner loop implements the well-known *reduce* parallel pattern (i.e. an array of elements is reduced to a single value by means of a binary operator [**17**], which in our case is the sum operator), the step complexity of this piece of code would decrease from $O(n_{\text{osc}})$ to $O(\log_2 n_{\text{osc}})$, thus **bringing the computational complexity of the whole kernel down to $O(n_{\text{steps}} \cdot \log_2 n_{\text{osc}})$**!

Indeed, **hardware resources would limit the actual speedup** (each `simulate` thread would launch n_{osc} threads, thus resulting in n_{osc}^2 `current` threads running at once), because no device offers the computational resources needed to execute the entirety of those tasks in parallel; yet, this enhancement would undeniably increase parallelism, keeping the GPU's thousands of cores steadily busy during the kernel's execution.

3.4.2 The cuSPARSE Library

Subsection 3.3.2 identified memory availability as the main reason why we couldn't experiment with greater quantities of oscillators: reducing n_{per} proved effective for the sake of performance assessment, but is not a viable option in a real-world scenario (those extra periods are indispensable!).

Considering that the interactions in the interconnection network are often limited to pairs of adjacent oscillators [**11**], it would make sense to **store the Y matrix in a sparse data structure**, thus saving lots of precious device memory: as an example, Y^D_{10240} (the dense matrix of a 10240-oscillator network) takes up 800 MiB of device memory, while its sparse counterpart, Y^S_{10240}, would not exceed a 400 kB footprint – a 99.95% reduction! – under this paragraph's opening assumption.

Another possible development would consequently consist in the **adoption of the cuSPARSE library**, which provides a set of data structures and functions to handle sparse matrices in CUDA applications.

3.4.3 Concurrent Copy and Kernel Execution

Still, representing Y as a sparse matrix would only be a short-term solution. In fact, as equation **(8)** suggests, the simulation results have a very extended memory footprint, and cannot be compressed into a sparse representation: a hypothetical (4096, 1000) configuration would produce approximately 1.6 GiB of output data.

The ultimate workaround for memory-related constraints consists in the exploitation of CUDA's concurrent copy/execution capabilities. In other words, CUDA devices with CC 2.0 or greater allow **data transfers to happen concurrently with kernel execution**. We could thus reengineer the kernel so as to simulate the network's behaviour for finite portions of the simulation time: after computation for a given chunk is over, control returns to the host, which disposes the copy of the results to host memory – and then, eventually, to file – while launching the kernel again to resume the simulation from where it was stopped.

The adoption of this technique, combined with the sparse representation of Y, would allow the simulation to proceed for an **almost unlimited number of oscillators**.

References

[1] NVIDIA Corporation. CUDA Zone. [Online]. https://developer.nvidia.com/cuda-zone

[2] Khronos Group. OpenCL's Homepage. [Online]. https://www.khronos.org/opencl

[3] NVIDIA Corporation. An overview of NVIDIA's ready-to-use CUDA libraries. [Online]. https://developer.nvidia.com/gpu-accelerated-libraries

[4] NVIDIA Corporation. Unified Memory Programming. [Online]. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-unified-memory-programming-hd

[5] NVIDIA Corporation. Feature Support and Technical Specifications for CUDA's different Compute Capabilities. [Online]. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities

[6] Tobias Grosser, Armin Groesslinger, and Christian Lengauer, "Polly – Performing Polyhedral Optimizations on a Low-Level Intermediate Representation," *Parallel Processing Letters*, vol. 22, no. 4, December 2012.

[7] Cédric Bastoul, "Code Generation in the Polyhedral Model Is Easier Than You Think," in *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques*, Washington, DC, USA, 2004, pp. 7-16.

[8] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul, "The Polyhedral Model Is More Widely Applicable Than You Think," *Lecture Notes in Computer Science*, vol. 6011, pp. 283-303, 2010.

[9] Multicore Computing Lab, Indian Institute of Science. PLuTO's Website. [Online]. http://pluto-compiler.sourceforge.net

[10] Raghesh Aloor, Tobias Grosser, Andreas Simbürger, and Hongbin Zheng. Polly's Website. [Online]. http://polly.llvm.org

[11] Paolo Maffezzoni, Bichoy Bahr, Zheng Zhang, and Luca Daniel, "Analysis and Design of Weakly Coupled LC Oscillator Arrays Based on Phase-Domain Macromodels," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 1, pp. 77-85, January 2015.

[12] Intel Corporation. Intel Core i7-2600K Processor Specifications. [Online]. http://ark.intel.com/products/52214/Intel-Core-i7-2600K-Processor-8M-Cache-up-to-3_80-GHz

[13] Intel Corporation. (2013, June) 2nd Generation Intel® Core™ i7 Processor Datasheet, vol. 1. [Online]. http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/2nd-gen-core-desktop-vol-1-datasheet.pdf

[14] Nvidia Corporation. Nvidia® GeForce® GTX 770 GPU Technical Specifications. [Online]. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-770/specifications

[15] Nvidia Corporation. Internal Architecture of GPUs of Compute Capability 3.x. [Online]. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#architecture-3-0

[16] Lawrence Latif, "Nvidia says large GPGPU speed up claims were due to bad original code," *The Inquirer*, November 2012. [Online]. http://www.theinquirer.net/inquirer/news/2227038/nvidia-says-large-gpgpu-speed-up-claims-were-due-to-bad-original-code

[17] Michael McCool. (2009, July) The Reduce Parallel Pattern. [Online]. https://software.intel.com/en-us/blogs/2009/07/23/parallel-pattern-7-reduce