# Advanced Computer Architecture: The “Smooth” Challenge

Romain Brault[1], Alexandre Camus[2], Giorgos Flourentzos[3]

**Abstract**
Coding is a question of how to compute things, but also how to compute them the fastest. These are often two questions that can't be resolved at once. First, coding a program that gives the expected result, is the first obstacle to face. But it is not the biggest. Once this is done, optimizing the written code might take a lot of time, depending on if the hardware on which it is running, is taken into account. Here, given a correct program, the aim was to optimize it, given a chosen architecture. This consisted in understanding the code, then optimizing it sequentially and finally trying to improve it by parallelizing over the CPU.

[1]RB812 [2]AC5612 [3]GF210

## Contents

## 1. Introduction

### 1.1 Context and Objectives

Simulation in computer science is usually computationally intensive. Although an algorithm can be theoretically efficient[1], a naive implementation, without any hardware consideration reveal to be slower than expected.

This paper start with a basic implementation of a curve smoothing algorithm where the curve – a mesh – is represented by a graph. This paper present a various methods to reduce the computation time of the smoothing algorithm on a specific machine, according to the hardware specification. The list of optimizations presented is non exhaustive, however the considered approach reduced the computation time by approximately 1300% on the given architecture.

The method used to achieved this speed-up is the following:

- first optimize on one CPU core,
- then parallelize over one node (here a single computer),
- eventually used hardware accelerator such as GPU.

[1]With a low complexity.

### 1.2 Software Considerations

The hardware considered is one of the Imperial College computing laboratory. All these computer are equipped with an Intel CPU, thus the best performances were obtained using the Intel Compiler[2]. However the Imperial College computer do not have the latest version of the Intel Compiler installed on, providing some optimizations and and the use of the new C++ standard (C++11). To obtained the maximum throughput additional library, not installed on Imperial College computer, such as blitz++, were used. A solution to have the best of the latest compiler version, library flexibility and the performances of the Imperial College computers is to cross-compile the code.

Cross-compiling is the action of creating executable code for a platform other than the one on which the compiler is running. This is challenging because the compiler usually tune the code to be as fast as possible on the machine where the code is compiled lowering the

[2]The speed-up gained by switching from g++ (GNU) to icpc (Intel) is presented section **??**.

performances on the target machine. Additionally, extra care must be payed to the portability of the code.

### 1.3 Hardware Considerations

The machine used for this coursework runs a Fedora 18 distribution. This is the exact information:

| | |
|---|---|
| Model Name | Intel Core i7 CPU Q720 |
| Clock Speed | 1.597 GHz |
| Max Turbo Frequency | 2.8 GHz |
| Cache size | 6144 KB |
| Cache line size and alignment | 64 B |
| CPU cores | 4 |
| CPU Threads | 8 |
| Integrated GPU | No |
| Memory Channels | 2 |
| Max Memory Band-with | 21 GB/s |
| Flags | fpu, sse, sse2, sse3, ssse3, sse4_1, sse4_2 |

**Table 1.** CPU Specifications

| | |
|---|---|
| Model Name | nVidia GeForce GTX 260M |
| Clock Speed | 1.375 GHz |
| Multiprocessors | 14 |
| Total CUDA cores | 112 |
| Allocated Memory | 1 GB |
| Memory Frequency | 950 MHz |

**Table 2.** GPU Specifications

To summarize, the figure 1 gives a quick overview of the hardware topology. The GPU is connected on the PCI port `10de:0618`.

## 2. The Sequential Issue

After running for the first time the program, it appears that the method `mesh_quality()` is the most called and so is the most expensive in computation time. Hence its code has been considered as a potential source of optimization.

Reading the code, some sequential issues were found. There are three kinds of problems: the algorithms chosen, the coding style and the data's representation.

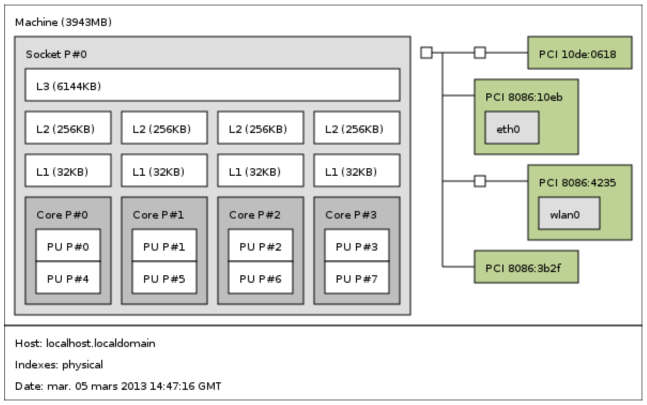

**Figure 1.** Topology

### 2.1 Algorithm

The method used to solve an equations' system was correct but two general to be efficient. The program needs only to solve systems of two equations in two unknowns. The Cramer's rule based on determinants is far more efficient.

The use of the method `pow()` seems a little too heavy in the method `element_quality()`. As the number of multiplication is known, the use of simple multiplications is more efficient here, e.g. `x*x*x`.

### 2.2 Code Style

Few changes in the style of the code might help the compiler. Some of these changes have been done to speed up the program.

Inline functions do often help the compiler to optimize the translated assembly language. That is why functions like `isSurfaceNode()` or `isCornerNode()` or `element_quality()` or `svd_solve_2x2()`.

In loops changes have been made to avoid recomputation of the invariant (e.g. calling `.size()` in a loop). Instead, this invariant is now stored in a local variable.

### 2.3 Data's Representation

The data structure used is fine and one of the most efficient to represent a graph. However, the C++ structures used seem to be a little too big in this case. So instead of using vectors of sets, the program is now using vectors of vectors. This increases a little the global performance.

Similarly, the number of node does not exceed $10^5$ which is less than the maximum integer represented thanks to the `uint32_t`. This type is faster to manipulate than the classic `int` type. Hence `uint32_t` has been preferred.

In the `element_quality()` method, the call of an element in a matrix during its multiplication, is very expensive in memory access. So instead of calling the element at each step of the loop, it is better to use an intermediate variable that is assigned into the matrix element at the end.

## 3. CPU Parallelization

### 3.1 Analysis

Parallelization can't be done easily. The program needs to be slightly modified in order to cut the graph in groups of independent nodes. Independent nodes are nodes that can be inspected at the same time while running the program. To group nodes in such groups the graph must be colored. Then each color represents nodes that can be inspected at the same time because they are not adjacent.

But the coloring algorithm might be very expensive in computation time. This depends on the number of colors used and if this number is specified or not. In this very case, the goal of such an algorithm is to minimize the colors used in order to maximize parallel computations. Hence a powerful algorithm is needed.

### 3.2 Optimization

A first try was realized with a very complex algorithm that colors a graph with minimum of colors needed. But the simple fact of coloring the graph was itself longer than the original program (hundreds of minutes). This confirmed that the graph is complex and parallelizing the program will surely reduce the run time.

So the coloring algorithm finally chosen was very basic. It just tries to color the graph without minimizing the number of colors used. It succeeds in coloring the graphs with 4 or 5 colors. Algorithms 1 and 2 are its implementation in pseudocode.

```
input  : the graph $G$
output : $C$ the array containing the colors

creating $C$, the output of size: number of node;
intializing each element of $C$ to $-1$;
for $i \leqslant$ number of node do
    $C[i] \leftarrow get\_colors(G[i])$;
end
return C;
```

**Algorithm 1:** The `coloring` Algorithm

Then each color represents a set of nodes that are computable in parallel. Actually this is simply done by adding a `for` loop on the colors before the loop that inspects all the nodes of the graph. The color of each node is compared to the current color. If they are different the node is skipped. The parallelization is then simply done on the loop that inspects all the nodes.

```
input  : the array $A$ of all neighbours of one node
output : return the color $C$ which is not used by
         other neighbours

$C := 0$;
while do
    for $i \leqslant$ number of neighbours do
        if $C = A[i]$ then
            $C := C + 1$;
            break;
        end
    end
    if $i =$ number of neighbours then
        return C;
    end
end
```

**Algorithm 2:** The `get_colors` Algorithm

## 4. Results

### 4.1 Sequential Improvements

### 4.2 Parallelization Performance