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. Although coding a program that gives the expected result is an obstacle, another problem arises when performance is to be maximized. Once this is done, optimizing the written code might take a lot of time, depending on whether 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 of understanding the code, then optimizing it sequentially and finally trying to improve it by parallelizing through vectorization or offloading the calculation to an accelerator (GPU).

[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.

1.2 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

[1]With a low complexity.

[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.

The code was compiled on a Linux-Fedora 18 64bits station and is to run on a Linux-Ubuntu 12.04 64bits station.

1.3 Hardware Considerations

Table 1 and 2 show respectively the hardware characteristics of the compiling machine and the running machine. The compiling machine is much slower than the running machine but is able to generate code optimized for the running machine.

Model Name	Intel Core i7-QM720
Clock Speed	1.6 GHz
Max Turbo Frequency	2.8 GHz
Cache line size and alignment	64 B
CPU cores	4
CPU Threads	8
Integrated GPU	No
Memory Channels	2
Max Memory Bandwith	21 GB/s
Flags	fpu, sse, sse2, sse3, ssse3, sse4_1, sse4_2

Table 1. CPU Specifications for the compiling station.

Model Name	Intel Core i7-2600
Clock Speed	3.4 GHz
Max Turbo Frequency	3.8 GHz
Cache line size and alignment	64 B
CPU cores	4
CPU Threads	8
Integrated GPU	Intel HD Graphics 2000
Memory Channels	2
Max Memory Bandwith	21 GB/s
Flags	fpu, sse, sse2, sse3, ssse3, sse4_1, sse4_2, avx

Table 2. CPU Specifications for the running station.

Figure 1 and 2 gives the hardware topology of the compile machine the run machine. Both used an intel i7 on one socket with almost the same topology; the only difference being the size of the L3 cache of 8M on the running machine compared to 6M on the compile machine. The other main advantage of the Imperial College computer over the laptop is presence of AVX instructions. Therefore all the optimizations to tune the code on the laptop should be efficient on the Imperial College computer, except that the laptop should generate an AVX code. As a result the code could not run on the laptop which has generated the code, but will be more efficient on the Imperial College computer.

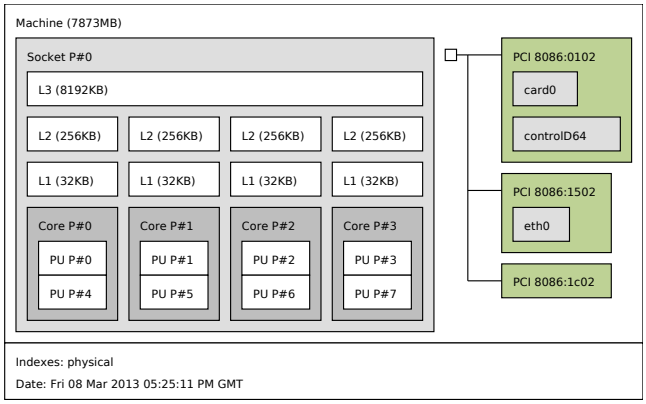

Figure 1. Topology of the running station.

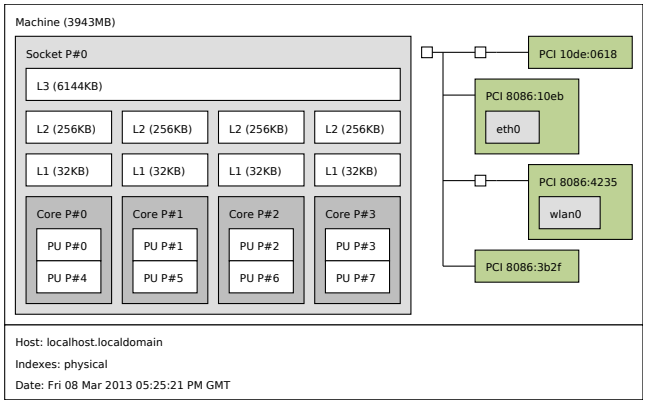

Figure 2. Topology of the compiling station.

2. The Sequential Issue

First the code was compiled with the g++ compiler optimised at level 3, and analyzed with Intel VTunes profiler to identify the bottlenecks. The initial speed for was 6.91 the small size graph, 54.0s for the medium and 510s for the large one. It appears that the initial program spend about 80% of its time in the function `mesh_quality()`. 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.

2.1 Code Style

Few changes in the style of the code might help the compiler to optimise `mesh_quality()`. First of all, the code has been rewritten in order to be more oriented object than before. Basically all the attributes of the `Mesh` class have been set to private and the `smooth` function is now a method of this class. This did not really increase the performance but improved the readability. Additionally increase the modularity of the code allows the compiler to do more optimizations as all the classes or function are not spread across multiple files. It might also help the compiler to improve its compiled code as it is used to read object oriented code.

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.

2.2 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 Cramer's rule is as following:

Theorem 1. Given an equations system $Ax = b$, where A is a squared matrix of size n, and x and b two n-vertical vectors.

If $\det A \neq 0$ then the system has exactly one solution. Its solution is:

$$x_i = \frac{\det A_i}{\det A}$$

where $x = (x_1, \ldots, x_i)^T$ and A_i is the matrix formed by replacing the i-th column of A by the column vector b.

In this case (2×2), the result is:

$$x_1 = \frac{(b_1 \times a_{2,2}) - (b_2 \times a_{1,2})}{(a_{1,1} \times a_{2,2}) - (a_{2,1} \times a_{1,2})}$$

$$x_2 = \frac{(a_{1,1} \times b_2) - (a_{2,1} \times b_1)}{(a_{1,1} \times a_{2,2}) - (a_{2,1} \times a_{1,2})}$$

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`.

2.3 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.

Finally the vectors `normals` and `ENList` are not anymore vectors. The Blitz library[3] provides useful arrays' types with a speed comparable to Fortran implementations. After testing it on the different vectors in the program, it appears that the type `Array` of this library was more efficient for vectors that are linearly accessed. `normals` and `ENList` have this property. So they are now implemented thanks to `Array` type.

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

[3] `http://blitz.sourceforge.net/`

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 ⩽ number of node do
    C[i] ← get_colors(G[i]);
end
return C;
```

Algorithm 1: The `coloring` Algorithm.

```
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 ⩽ 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.

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.

4. Results

4.1 Sequential Improvements

4.2 Parallelization Performance