Advanced Computer Architecture: The “Smooth” Challenge

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

Abstract
Coding is a question of how to compute things, but also how to compute it the fastest. These are often two questions that can’t be resolve 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 or using GPU.

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

Contents

Model Name	Intel Core i7 CPU Q720
Clock Speed	1.597 GHz
Max Turbo Frequency	2.8 GHz
Cache size	6144 KB
CPU cores	4
CPU Threads	8
Integrated GPU	No
Memory Channels	2
Max Memory Bandwith	21 GB/s

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

1. Introduction

1.1 Hardware Considerations

The chosen hardware is not one of the machine from the Lab. They do not have any interesting GPUs. But one of our laptops is very powerful with an interesting GPU. This is the chosen one. The tables 1 and 2 contains the details of this hardware.

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

1.2 Software Considerations

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

```
$ uname -a
Linux localhost.localdomain
   3.8.1-201.fc18.x86_64 #1 SMP Thu
   Feb 28 19:23:08 UTC 2013 x86_64
   x86_64 x86_64 GNU/Linux
```

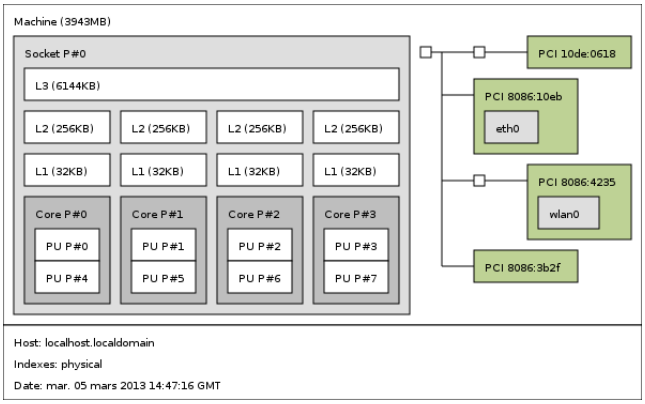

Figure 1. Topology

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.

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 basic algorithm that colors a graph without assuming anything on its structure. But the simple fact of coloring the graph was itself longer than the original program. This confirmed the need of a very powerful algorithm.

4. GPU Acceleration

4.1 Analysis

4.2 Optimization

5. Results

5.1 Sequential Improvements

5.2 Parallelization Performance