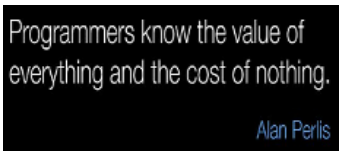
How expensive is an operation on a CPU?



Programmers know the value of everything and the costs of nothing. I saw this quote a while back and loved it immediately. The quote by Alan Perlis is originally about Perl-programmers, but only highly trained HPC-programmers seem to have obtained this basic knowledge well. In an interview with Andrew Richards of Codeplay I heard it from another perspective: software languages were not developed in a time that cache was 100 times faster than memory. He claimed that it should be exposed to the programmer what is expensive and what isn't. I agreed again and hence this post.

I think it is very clear that programming languages (and/or IDEs) need to be redesigned to overcome the hardware-changes of the past 5 years. I talked about that in the article "Separation of compute, control and transfer" and "Lots of loops". But it does not seem to be enough.

So what are the costs of each operation (on CPUs)?

This article is just to help you on your way, and most of all: to make you aware. Note it is incomplete and probably not valid for all kinds of CPUs.
Math
Green: basic operations
The cheapest of the cheapest, in this order:

- addition, subtraction, comparison (1)
- abs (2)
- multiplication (4)

But even if multiplication is 4 times as expensive as the first group, all of them are very fast to compute. Avoiding a multiplication could make it faster in exceptional cases.

If you can multiply integers with 2 then it is easier, as very cheap bit-shifts can be used. For example 6 = 110, 12 = 1100. Same advantage for floats, but then then the exponent just gets increased by one.
Orange: division and modulus
Division and modulus are more than twice as expensive as multiplication (a weight 10). The division by two or a multiple of two is always a trick, but not much more can be done without having side-effects.

If you can replace division by multiplication, you do get a speed-up of more than two. A side-effect is that errors (especially floats) multiply too and you can end up with less precision. Know what you do.
Red: Exp, Sin, Cos, Tan, Sqrt, Pow
On the CPU there is this list:

- exp (50)
- sqrt (??)
- sin, cos, tan (60)
- asin, acos, atan (80)
- pow (100)

The number indicates how many times more expensive they are than an addition (using normal CPU and not the extensions such as

SSE). Here we talk about much more expensive operations than before. These are indications, as for example one power-computation doesn't compare to another. The bad thing is once you can type out the exp or pow, the compiler can do so too.

The square root operation depends so much on the processor, that I could not find any good data on it. You can read this article on sqrt-implementations if you want to learn more.
If you use a lot of trigoniometry, the first you can do is finding replacements in the formula. There are more of these algorithmic tricks, and that is what we mostly do at StreamHPC when we do algorithmic redesign to come up with a much faster variant.

A second option is using fast versions of sqrt and trigonometric functions (the 'f' functions). You get less precision but more speed. Most are implemented in AVX+SSE. See for example this code and read the sheets of this presentation by Sony.

A third option is caching or a form of caching: table look-up, compile-time computing, etc. This is only possible when the number of options are limited, but this works with a lot of slow code.

Want to read more? These two articles are a good start:

http://www.azillionmonkeys.com/qed/optimize.html]
http://www.eventhelix.com/RealtimeMantra/Basics/OptimizingCAndCPPCode.htm]

Memory & data-structures

Memory operations can be very expensive if you focus on the goal and not on the road to it. Here I start with red, as it shows you the way to get a green flag for your code.
Red: unfixed dimension
If you don't know how much data you get and you haven't put efforts in doing batch-like processing to read in data per chunk, then the compiler doesn't miraculously know it either. The whole idea why annotations work so well is that these force the developer to think about what he or she is doing.

If you think in what you need to finish an algorithm successfully, you get implicit operations such as redim. You probably have heard that unfixed arrays (such as vectors) have bad influence on memory-allocation. It is worse, as requesting more memory is expensive. Also, unpredictable memory sizes can be bad for caching-algorithms.
Orange: predictable data-sizes
Tanenbaum wrote in his book "Structured Computer Organization":
Consequently, what most machines do when they hit a conditional branch is predict whether it is going to be taken or not. It would be nice if we could just plug a crystal ball into a free PCI slot to help out with the prediction, but so far this approach has not borne fruit.
In other words: the more the programmer tells, the less prediction-mistakes will be made, and the less expensive the execution is.

Having well-defined arrays and alike, makes it possible for the compiler to see if implicit vectorisation and parallellisation are possible. Especially on modern multi-core CPUs with AVX-extensions. These could make a big difference. In OpenCL you can make them explicit with a little bit more effort.
Green: cached and fixed
If you explicitly localize data before actually using it, the compiler will be happy with you. For example, when doing operations using say 10000 rows in the database, specifying-first is much faster. With a database most people have the feeling that the distance between the software and the data is larger, while it can be as fast as operations on main memory.

We don't live in a time in which you explicitly need to define how memory-operations between the CPU, memory and caches need to be done (as currently on GPUs is still needed). The advantage is much less code, the disadvantage is the idea that it is not needed to show the compiler which data you are going to use.

The killer: repetitions

If you want to run a given extensive algorithm which takes a second, most office-workers don't complain it is too slow. Say this algorithm computes through a portfolio at a bank. And then this bank has increased the number of portfolios to one million over the years. Then one computer would take 277 hour or 12 computers... almost a whole day. The programmer probably had not in mind much more than 10000 to 30000 portfolios would be needed to be computed a day. They would just run these computations during the night. Above tips would work to dramatically increase performance, without even using OpenCL. Guess once where the killer-speedups were made when they ported to OpenCL.

Exactly, by knowing the cost of an operation.

StreamHPC is about scalability and using multi-core processors, but we gain most speed by re-engineering the algorithm.

How to do it yourself: print out the algorithm and takes three markers: green, red and orange. Mark all areas as described above: this is what you should see when programming. What happens when you see this marked code in front of you? Could you have coded your software differently?