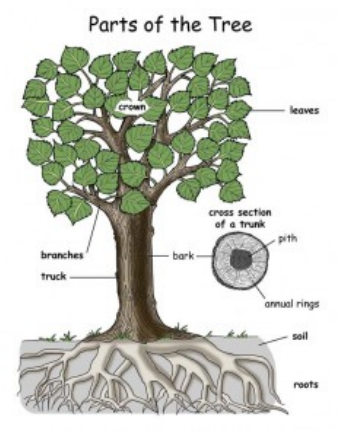
Separation of Compute and Transfer from the rest of the code.



What if trees had the roots, trunk and crown were mixed up? Would it still have the advantage over other plants?

In the beginning of 2012 I spoke with Patrick Viry, former CEO of Ateji - now out-of-business. We shared ideas on GPGPU, OpenCL and programming in general. While talking about the strengths of his product, he came with a remark which I found important and interesting: separation of transfer. This triggered me to think further - those were the times when you could not read on modern computing, but had to define it yourself.

Separation of focus-areas are known to increase effectiveness, but are said to be for experts only. I disagree completely - the big languages just don't have good support for defining the separations of concerns.

For example, the concepts of loops is well-known to all programmers, but OpenCL and CUDA have broken with that. Instead of using huge loops, those languages describe what has to be done at one location in the data and what the data is to be processed. From what I see, this new type of loop is getting abandoned in higher level languages, while it is a good design pattern.

I would like to discuss separation of compute and transfer from the rest of the code, to show that this will improve the quality of code.
Separation of compute from repetition
In the concept of looping, one can choose to loop over (i, j) of the input-data or (k, l) of the result-data, in OpenCL one is forced to do the computations seen from each (k, l) of the result-data (with some exceptions). An important difference is that you cannot have temporary-answers and use them in the next iteration. If iterations are needed, then data can be computated in several steps.

By forcing the programmer to think this way through the separation of the single computation from the repetition, the code can be optimized and scaled easier. And exactly this is what is abandoned in high-level GPGPU-languages. You see the low-level languages are all on the left, and most of the higher-level on the right. Many libraries solve this very elegantly. They require two inputs: functionality (the function being called) and the data to operate on.

Node-wise
Functional
Iterative & directives

OpenCL
CUDA
DirectCompute
Library calls
C++ AMP
HMPP
Aparapi
OpenACC

Node-wise (each group of data-elements on which computed is on) has the advantage of scaling, but takes some time to implement by programmers who are used to loops. The functional approach could be a good direction, when applicable.

The old-fashioned way (directives on iterations) is like using plasters on a bad design. I get into more details in a follow-up post on 'loops'.

Separation of transfer from 'malloc'

When using GPUs, and also when reading a file from disk, part of the time that the whole operation takes is transferring data. This needs scheduling and most programmers don't think about this. Explicit scheduling data-transfer has been a very important part of the host-code in OpenCL since 1.0.

Explicit in host-code

Scheduled by compiler

OpenCL
CUDA
DirectCompute

C++ AMP
HMPP
Aparapi
OpenACCSVM (in OpenCL 2 and CUDA)

Forcing programmers to explicitly defining transfers, makes him/her think about it - limiting transfers results in a good speed-up.

In a future post I will discuss HSA. This new technology solves many problems around memory sharing and is worth to take a good look at.

To separate or not to separate?

Even if the old ways of programming have shown their scaling limits, most higher level languages which aim to replace OpenCL and CUDA, focus on trying to trust the old paradigms instead of copying the separation of concerns. It is not parallelism itself but

these separations that could help us all advancing in the multi-core era.

I invite language and compiler designers to focus on that, instead of making the new way of programming not new at all and with that making it actually harder to program parallel computers.

What do you think? More or less separation to cope with scaling on multi-core processors?