CPU vs. GPU illustrated through Mandelbrot set computing

Emmanuel DUPUIS (emmanuel.dupuis@undecentum.com) — January 2018

Comparing CPU and GPU computing capabilities is a popular issue since more than a decade (see for instance **"Debunking the 100x GPU vs. CPU myth"**). The idea that the comparison result would lead to choose one or the other is usually a non-sense. Designing a software application is, and has always been, a search for a compromise between different resource uses (Memory vs. Processing, Resident memory vs. Storage on disk, CPU vs. GPU, Energy consumption vs. speed …). And since that compromise rarely excludes any resource the matter, is in fact, how to optimise cooperation between them.

Nevertheless, it is important in that perspective to be aware of the strengths and weaknesses of each resource and to get inspiration from GPU native parallelism efficiency to modify the way programs are executed on multi-cores CPUs.

We use a basic and well-known example to illustrate that brief study, the Mandelbort Set (set of points of the complex plane for which the module of the sequence $z_{n+1}=z_n^2+c$ and $z_0=0$ is bounded).

The tests are performed for 3 different modes, sequential on CPU, multithreaded dispatcher on CPU (see details at the end of this article) and CUDA on GPU. It's important to note that the CPU, as well as the GPU, is not running other tasks at the same time (OS tasks on CPU are not significant or neutralised by the test protocol). In the context of a whole and composite application, different tasks may be distributed and executed on both resources. In this case a more complex load balance management might be needed.

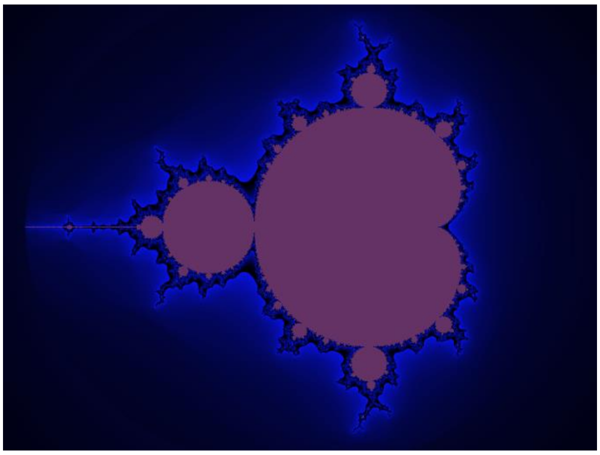

Figure 1: Mandelbot set (purple zones) and its exterior (blue zones)

Optimised compilation on CPU leads to performances almost twice better

This example is part of CUDA sample set for which it implements a benchmark comparison between GPU and CPU performances. It's very interesting to notice that the Visual Studio Solution defines a single configuration, the debug one. No release configuration is provided and if it's on purpose is certainly because compilers (for Intel chipset in our case) have powerful optimisation capabilities that may tremendously change the performances.

As you can see below, optimisation on CPU leads to performances almost twice better or even more. Whereas CUDA compiler optimisation does not produce any significant change.

Elapsed time in ms		CPU Sequential	CPU Dispatcher	GPU
Not Optimised		174.5	33.4	2.180
Optimised	CPU: Maximize Speed /O2 GPU: -O3 -use_fast_math	91.5	14.8	2.207
Performance ratio		1.91	2.26	0.99

Figure 2: Performance improvement using optimised compilation

For some complex mathematical algorithms the advantage of GPU might be lower

If you have studied algorithms colouring the Mandelbrot Set, you may notice that this CUDA sample use a rather simple solution for the exterior of the set. The usual technic to produce continuous colouring is to introduce a distance function, which is evaluated using the logarithm function (two calls for each single point).

This remark would be useless if this extra computing were another floating-point arithmetic operation. But as you see below, the relative cost of this mathematical subroutine is highly different for CPU and GPU. In the first case, it's representing around 10% of the total, whereas in the latter it's 60%. ALU of CPU are obviously more complex chipset than the ALU of GPU, they integrate dedicated circuits for specific mathematic functions.

Elapsed time in ms		CPU		GPU
		Sequential	Dispatcher	
With	logarithms	91.5	14.8	2.18
Without		80.2	13	0.87
Performance ratio		1.14	1.14	2.51

Figure 3: Performance improvement removing calls to logarithm

As a consequence, for some complex mathematical algorithms the advantage of GPU might be lower. In our case, the elapsed time for logarithm computation is about 1.8ms for CPU and 1.3ms for GPU (773,820 log function calls) and we might imagine a different load balance: use GPU to compute the sequence and CPU to compute the colour. In this configuration the CPU computation would be parallel to the GPU computation (of the next image). But this solution is beneficial only if the added stream of data between CPU and GPU has not a greater cost than the computing gain. Indeed, here the GPU computation result is aimed to be displayed and thus sent to the video memory.

GPU is 5 times slower in double precision

The last factor, we'll consider in this illustration is the floating-point precision. For this global image of the Mandelbrot set we don't need a high numerical precision of the sequence. But, if we zoom a part of the image and have a small distance between two pixels, the need of precision is major to get an accurate result (and the length of the sequence to determine the bounding property has also to be higher but that affects both CPU and GPU). As you can see, if CPU chipset has, as expected, the same performances for simple and double precision, the GPU is 5 times slower in double precision. CUDA sample use an emulation of double precision, called double simple precision (see MPFUN90 package), that gives better results in term of speed but still with a 4 or 5 to 1 ratio.

Elapsed time in ms	CPU		GPU
	Sequential	Dispatcher	
Simple precision	89.8	14.9	2.18
Double precision	91.5	14.8	10.97
Performance ratio	1.02	0.99	5.03

Figure 4: Performance impact of floating precision

CUDA: the number of registers and the segmentation of the grid are critical

They are numerous factors that impact the performances of CPU and GPU. And whatever the resource use scheme is, many parameters have to be tuned for both CPU and GPU. Regarding GPU and CUDA, the impact of the number of registers used might be critical since one of the most constraining limitation is often the number of threads per SM. The segmentation of the grid is also critical and the choice for CUDA sample (i.e. 5 blocks from the number of SM) is not the most efficient. A block for 16 pixels and a single kernel for each pixel lead to almost a half duration.

CPU Multithreading leads to a more than 6 times better performance

The « GPU revolution » has surely changed the way parallelism is integrated to software application design. The sequential mode can't be considered anymore as a default mode. The question to use parallelism should apply to any operation. In this computing configuration, the question has a straightforward answer: yes. The cost of multithreading integration is low compared to the performance gain.

The solution we choose is to use a task dispatcher. A certain number of customer threads (a single one in our configuration) sends requests to the dispatcher (tasks to be performed, here a sub part of the plane to be treated). These requests are stored within a thread-safe queue. And a pool of provider threads picks the requests and performed the associated task.

The tests show without any doubt that it's more efficient to leave the OS to assign the different threads (Th), customers and providers, to the different logical cores (LC) rather than specifying affinity (for instance 8 provider threads with specified affinity to LC1, LC2, …, LC8).

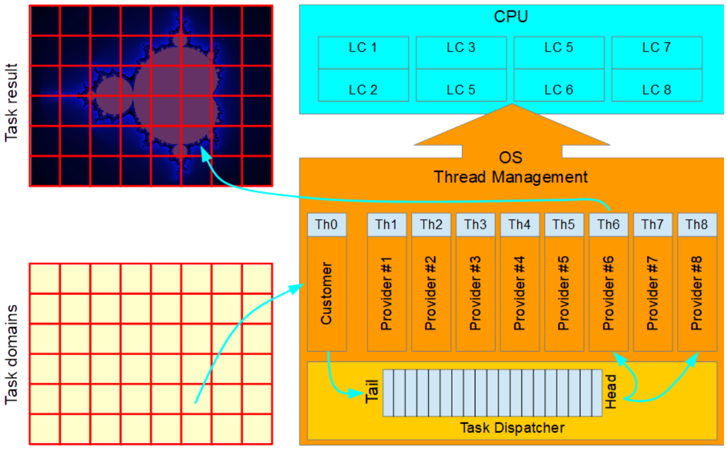

Figure 5: Dispatcher of subtasks using a thread-safe queue and a pool of threads

The performance gain is really significant since the best configuration leads to a more than 6 times better performance and the duration ratio GPU vs. CPU moves approximately from 1:42 to 1:7. This 6 factor has to be compared to the number of logical processing units (logical cores), 8.

Multithreaded features, like the CUDA grid, need "dimension tuning". Here, we might especially set different number of providers. The number of logical cores is naturally a minimum and the tests, which results are represented by *figure 6*, show that a greater number does not give any benefit. The degree of segmentation is another important parameter. The same results show that we should avoid a too low segmentation (less than 200 here) and a too high (more than 1200), the extreme parameterisation, one pixel per thread, leading to a twice slower result.

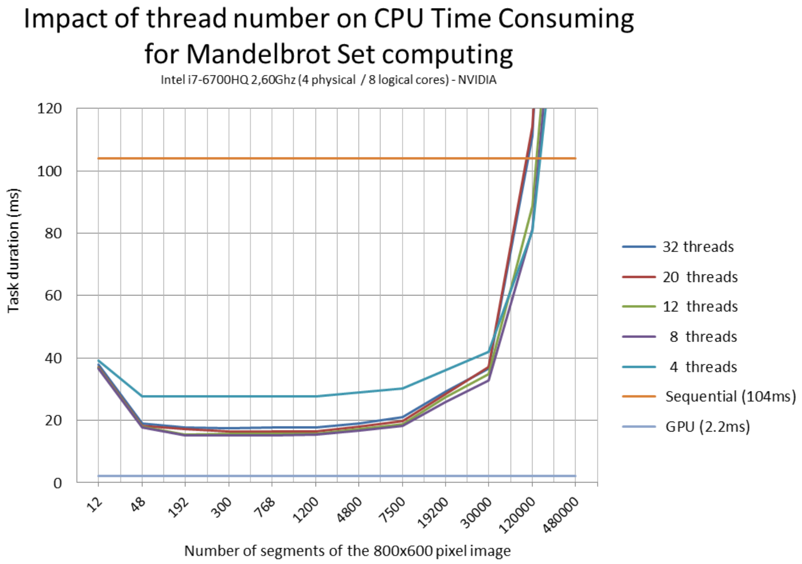

Figure 6