Selecting Applications Suitable for Porting to the GPU

Assessing software is never comparing apples to apples

The goal of this writing is to explain which applications are suitable to be ported to OpenCL and run on GPU (or multiple GPUs). It is done by showing the main differences between GPU and CPU, and by listing features and characteristics of problems and algorithms, which can make use of highly parallel architecture of GPU and simply run faster on graphic cards. Additionally, there is a list of issues that can decrease potential speed-up.

It does not try to be complete, but tries to focus on the most essential parts of assessing if code is a good candidate for porting to the GPU.

GPU vs CPU

The biggest difference between a GPU and a CPU is how they process tasks, due to different purposes. A CPU has a few (usually 4 or 8, but up to 32) ?fat? cores optimized for sequential serial processing like running an operating system, Microsoft Word, a web browser etc, while a GPU has a thousands of ?thin? cores designed to be very efficient when running hundreds of thousands of alike tasks simultaneously.

A CPU is very good at multi-tasking, whereas a GPU is very good at repetitive tasks. GPUs offer much more raw computational power compared to CPUs, but they would completely fail to run an operating system. Compare this to 4 motor cycles (CPU) of 1 truck (GPU) delivering goods ? when the goods have to be delivered to customers throughout the city the motor cycles win, when all goods have to be delivered to a few supermarkets the truck wins.

Most problems need both processors to deliver the best value of system performance, price, and power. The GPU does the heavy lifting (truck brings goods to distribution centers) and the CPU does the flexible part of the job (motor cycles distributing doing deliveries).

Assessing software for GPU-porting fitness

Software that does not meet the performance requirement (time taken / time available), is always a potential candidate for being ported to a GPU.

The data and algorithms in the software can be assessed for GPU-porting fitness using the following list of checks, in order of importance:

- data volume,
- data dependency,
- data processing order,
- data type,
- code/algorithmic complexity, and

- computational intensity.

Do note that the data is more important than the algorithm ? it's therefore better to talk about *High Throughput Data* than *High Performance Computing*.

The more checks of the above list disappoint, the lower the chance it can be ported to the GPU. Do keep in mind that software with a low score, could still be ported by replacing the type of algorithm.

Data Volume

The input data has to be transferred from a main computer memory (RAM) to a memory of a GPU. It should therefore be large enough to hide the time of copying it to a GPU.

Each processor of the GPU should have multiple tasks to process in order to hide scheduling-overhead. Data-size should therefore be multiple megabytes at minimum. The larger the data, the bigger GPU can be utilized. See also ?Computational Intensity? below.

A general rule-of-thumb is that if the data is high and the current throughput is less than 1 GB/s, there is plenty of room for improvement.

Data Dependency

The input data should be structured and processed in such a way, that processing a single data point should not depend on previous data-points. If this happens, the algorithm needs being reworked.

If an algorithm has several rounds, each round should be scored on itself.

Data Processing Order

The data should be read in a continuous order. Data should not be written or read in random order for maximum utilization of a GPU. If this happens, the algorithm needs being reworked.

Data Type

Most GPUs are optimized for arrays of 32-bit floating-points (float), and 16 or 32-bit integer values (unsigned int, int). Operations on strings are slow unless they can be treated as numbers.

Several recent GPUs have support for 16-bit or even 8-bit floats, giving added performance when using low-precision data. Professional GPUs often have good support for 64-bit. When 16-bit and 64-bit floats are to be processed, performance varies a lot per GPU.

Code/Algorithmic Complexity

A good hint that the code can be ported to GPU is its simplicity. Deeply branched code and while-loops perform less on a GPU. This is due to GPUs having less cache and less complex schedulers than CPUs.

Recursive functions have to be rewritten ? it's impossible to run recursive functions on GPUs unless very small.

There are two blogs interesting for further reading. ?*The 13 application areas where OpenCL can be used*? gives an overview of types of algorithms that can be ported: https://streamhpc.com/blog/2013-06-03/the-application-areas-opencl-can-be-used/

?*Lots of loops*? shows what types of loops should be rewritten when performance is to be increased:
https://streamhpc.com/blog/2012-04-08/lots-of-loops/

Computational Intensity

Accessing memory takes more time than computations. Depending on a GPU loading four bytes of memory (one int) can be 15 - 30

times slower than a single addition operation.

GPUs are designed to be able to hide costs of accessing memory when the algorithm is compute-intensive. This means there should a lot processing compared to loading and storing data ? described as ?FLOPS per Byte?. When the data-size is small, the FLOPS-per-Byte should be high.

A low FLOPS-per-Byte hits the memory-wall, a high FLOPS-per-Byte hits the frequency-wall. Most algorithms hit the memory-wall.

Phases of optimizations

Scoring algorithms and data

Using profiling tools, the hot spots can be found where most of the time is used. The algorithms and data need to be scored, as described in the previous chapter.

The result is a list of observations what are the possible hurdles to overcome. An experienced GPU-developer can now tell how big is the chance the software can be ported to a GPU.

Preparing CPU-code for the GPU

CPU-code is simply different from GPU-code. Therefore most code must be cleaned and simplified first. For example advanced C++ features should be rewritten in C-like functions, and flexible Python-code should be rewritten to a single function.

The code where the computation is happening often has to be mostly rewritten. This requires the original code to be well-documented, such that the transition is painless in terms of correctness and understanding. To have the transition under good control, Stream prefers to base the work on the original papers describing the mathematics and algorithms.

The end-result is new code which has the most-important problems eliminated.

Researching porting options

Given the scores and the simplified code, the targeted algorithm has to be isolated and get multiple variations of the implementation.

The result is a rough implementation of the algorithm that can be used in the final port, and ? most importantly ? a performance number.

Actual GPU-coding

Applying the rules of good software engineering takes testability, documentation and robustness into account. The rough implementation is embedded in the existing code-base, or ? in case of new software ? put into a well-tested framework.

Will it work for you?

We can tell you! We have built GPU-software for 8 years. Many of the software was based on test-code or research-code written in a language like Matlab, Python, C/C++ or Java/.NET. We then focused on correct implementation and performance.

Using this experience, we can assist you in making that assessment quickly. We often only need 2 weeks for validation. Feel free to discuss by phone or email.