

# GPGPU Programming

Instigate CJSC  
www.instigatedesign.com

July 2009

## Abstract

*GPGPU - stands for General Purpose computing on Graphics Processing Units. The idea is to use GPU's tremendous processing power in non graphical computations. But global switching to parallel programming can not be performed fast and easily due to a huge obstacle in the form of unusual and complicated programming model compared to the conventional ideas of sequential programming. The intent of this document is to help with classification of problems, which can profit from parallelization, to define some base approaches for implementing parallel algorithms and parallelizing the existing sequential ones.*

## Introduction

GPU architectures, intended to effectively cope with graphics processing, can efficiently deal with non graphical parallel computations. Sequential algorithms, that contain independent logical portions can benefit significantly from parallel implementation. In fact, for some problems, GPU can accelerate by over an order of magnitude over the CPU.

There are several GPGPU technologies available and some are expected to come in the near future. Though the stable version of OpenCL was released at the end of 2008, its first implementation is expected to be available at the end of 2009, with the release of Mac OS X 'Snow leopard'. Microsoft also claims to provide its own GPU programming language with their new version of OS and DirectX. Thus, so far we have two main GPGPU programming environments - NVIDIA CUDA and ATI Stream. While presenting different programming languages, both exploit the same basic concepts. Therefore, the main ideas of GPGPU programming methodology can be defined regardless of the API.

Attempting to provide some general knowledge, the following topics are covered in the subsequent sections: *Abstract introduction of GPGPU architectures, Classification of the problems from the point of parallelization, Some general approaches to implement parallel algorithms.*

This information is based not only on available public documentations and results, but on Instigates in-house parallel programming experience, which goes beyond solely GPGPU programming.

## 1. GPGPU ARCHITECTURES

Even without aiming at GPGPU, video cards were designed as parallel devices from the very beginning, as processing of each pixel in a 3D scene can be performed independently

from the others, it was reasonable to add more blocks with the same functionality instead of increasing the frequency of existing ones. Adding programmable stages and increased precision arithmetics was enough to turn graphic cards into powerful parallel programming platforms. Current GPUs contain up to 800 identical processing elements, and this count is far from satisfying the growing needs of computer graphics and parallel programming.

Originally designed for graphics processing, GPUs are limited in terms of operations and programming, and are only effective in *stream processing*. GPUs are stream processors - processors that can operate in parallel by running a single *kernel* on many records in a stream at once (kernels are the functions that run in parallel on each element of the stream), to put it differently we deal with data parallel programming model. GPUs also provide *barrier synchronisation* as it is the most convenient model for data parallel programming. On the whole, *functional parallelism* and even synchronisation techniques can be achieved on GPUs, but that is not their function. Thus we can achieve significant acceleration in the tasks, which belong to the class of problems, suitable for the data parallel programming model. So, the ability to analyze the problem in point of parallelism should be considered as the most important skill in the parallel programming.

## 2. CLASSIFYING THE PROBLEMS

Before experimenting and getting real results, it is reasonable to understand the nature of the problem. This will help to get some idea of the probable benefit from parallel implementation.

Below three criteria are mentioned, that need to be paid attention to:

### 1. Data parallelism

This expresses the level of the parallelism that is present in the problem, i.e. the parts where the same functionality is applied to the different groups of data, which can be processed in parallel. This can be illustrated better with an example.

Vector addition is one of the simplest and most obvious instances to reflect data parallelism. In fact, each element of the resulting vector can be computed independently and if we have  $M$  identical CPUs, which can do elementary addition, then it is possible to compute the sum of two  $N$  element vectors with about  $N / M$  steps, while we would need  $N$  steps for sequential implementation. In the most cases the parts that can be detached and parallelized are not so evident, but separating these parts should always form the basis of parallel algorithm implementation.

## II. Compute intensity

This criterion shows the ratio of computations to I/O and memory access. For example, in the above adverted vector addition we have one memory write and two read operations per single addition operation. The more computations are per I/O and memory access, the better, as in general, memory frequency and latency are the hardware bottlenecks. Speaking of memory, it should be mentioned that now-days video memory has much higher bandwidth than the main memory, but also has higher latency, which eliminates the advantage of the bandwidth.

## III. Data locality

Coming back to already noted memory access, it is necessary to note, that this problem is concealed with some success by faster local memory usage. All modern GPGPU hardwares hold notable amount of local memory. If the data to process should be read at the beginning of the algorithm and be written only in the final stage, then this data can be once read into local memory and whole the further accesses performed on it. Arranging memory accesses algorithm this way will bring to perceptible advantage.

## 3. THINKING PARALLEL

Taking into account the above mentioned criteria during algorithm creation already makes up the first steps of *Thinking parallel*. As it is said "*it is the first step that costs*", especially when switching from the habitual sequential programming. It is conclusive that sequential programming is more intuitive and susceptible for our thinking, but after acquiring some techniques and experience, the achieved benefits will be greatly evident.

For clarity let's consider the same vector addition example and write it's pseudo code implementation in data parallel manner. This is how it would look like in a *C* style implementation:

```
void vector_add(int* input_1, int* input_2,
               int* output, int size)
{
    for (int i = 0; i < size; ++i) {
        output[i] = input_1[i] + input_2[i];
    }
}
```

If we manage to launch the same function in parallel for all the elements of the array, the only additional information needed to proceed with a single element addition is a unique *ID* for each call instance, to specify the element of the vector to deal with. And the code will be something similar to this:

```
void parallel_vector_add(int* input_1,
                       int* input_2, int* output)
{
    output[ID] = input_1[ID] + input_2[ID];
}
```

## 4. SUMMARY AND CONCLUSIONS

One thing can be stated for sure - possessing good skills, one can utilize the processing power of GPUs in full measure. Understanding the paradigm of parallel programming and Stream Processing in particular, and combining this knowledge with the comprehension of a particular GPGPU programming platform will lead to success. Different GPGPU

platforms have different architectures, memory structure and programming specifics. These should always be taken into account when optimizing the already implemented algorithm. The following approach for developing parallel algorithms justifies itself by success in parallelization of different types of problems - following this order of steps during planning and development will be very helpful:

1. when developing an algorithm consider the specifics of the target platform, but don't concentrate on them.
2. do not apply the whole parallelism to the implementation at once. Do it step-by-step, to have the reference to compare the result and performance with.
3. and do device specific optimizations, which may include using less expensive operations for the particular hardware, managing the memory allocation in the appropriate way, etc.

Good luck!