Production Systems and Information Engineering
Volume 6 (2013), pp. 27-40

USING GRAPHICAL PROCESSING UNITS FOR DETERMINISTIC SINGLE MACHINE SCHEDULING PROBLEMS

KRISZTIÁN MIHÁLY
SAP Hungary Ltd., Hungary
krisztian.mihaly@sap.com

OLIVÉR HORNYÁK
University of Miskolc, Hungary
Department of Information Engineering
hornyak@ait.iit.uni-miskolc.hu

[Received January 2012 and accepted September 2012]

Abstract. This paper gives an introduction to how graphical processing units can be used in non-graphical related problems or tasks. First a history of GPU is provided. The next part focuses on GPU programming. A brief description is given about the available hardware facilities and the available programming languages. As an initial result of the project an easy and well-known scheduling algorithm was implemented for deterministic, single machine models. To check the performance achievement both the CPU and GPU code were implemented. Finally, some of the performance measurements are presented.

Keywords: GPGpu, OpenCL, CUDA, scheduling problems

1. Aims and scope of the paper

This paper aims to give an overview of the history of graphical processing units (GPUs) and how they can be used in non-graphical related tasks. After a short historical introduction it presents a short discussion on two programming languages and shows an easy example, where next to the CPU the GPU is used for solving deterministic, single machine scheduling problems.

2. Historical overview of GPUs

As Sanders put it "the state of graphics processing underwent a dramatic revolution. In the late 1980s and 1990s, the growth in popularity of graphically driven operating systems such as Microsoft Windows helped create a market for a new type of processors. In the early 1990s, users began purchasing 2D display accelerators for their personal computer. These display accelerators offered

hardware-assisted bitmap operations to assist in the display and usability of graphical operating systems" [1]. This accelerator approach was the first step to separate the graphic related tasks from the CPU. In 1992, Silicon Graphics opened the programming interface to its hardware by releasing the OpenGL library. It was a standardized, platform-independent method for developing 3D applications. At that time the computing of rendering was running in the CPU. The demand of the market was strong to have some kind of hardware support for 3D calculations to improve the application speed. The major companies were NVIDIA Corp., ATI Technologies and 3dfx Interactive.

"August 31, 1999 marked the introduction of the graphics processing unit (GPU) for the PC industry, the NVIDIA GeForce 256." [2] First the graphical hardware took care of the calculation of transformation and lighting computations, so these operations could run on the hard-wired graphical processors and the CPU could be used for other tasks. The next breakthrough in parallel-computing was the first graphic card, which supported the Microsoft's DirectX 8.0 standard. This standard introduced the programmable vertex and pixel shaders. This was the first point when the developers were able to influence the control of GPU programs.

Usually the developers used the GPU for graphics related tasks, but there were some developers who wanted to use the calculation power of the GPU cards. Since 2010 there has been a greater focus on the GPU's massive parallel computing capacity. Figure 1 shows how the calculation power of GPUs is increasing.

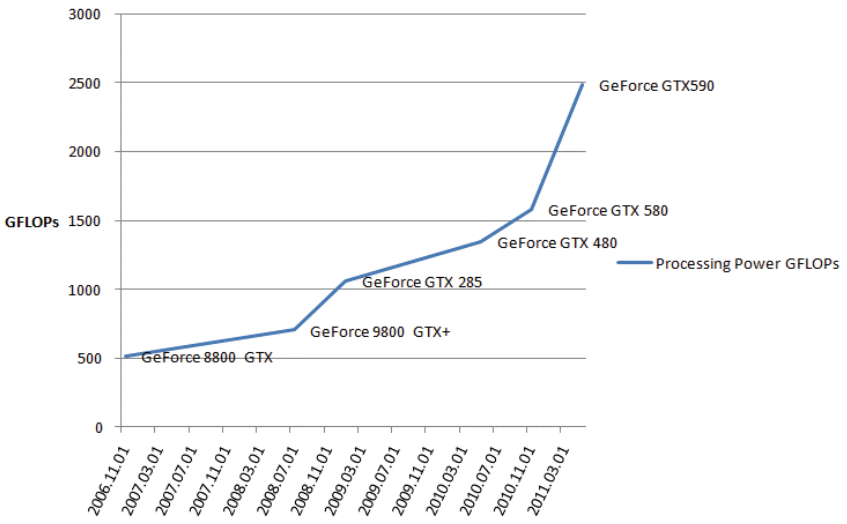

Figure 1. NVidia GPU card GFLOPs evolution

2.1 Tasks of the GPU and the programming environments

Let us have a look at what a GPU card should do. The application runs on the CPU and computes the 3D geometry. The geometry is loaded to the GPU and there is a transformation from 3D to 2D. After the transformation the card creates fragments and composes the image.

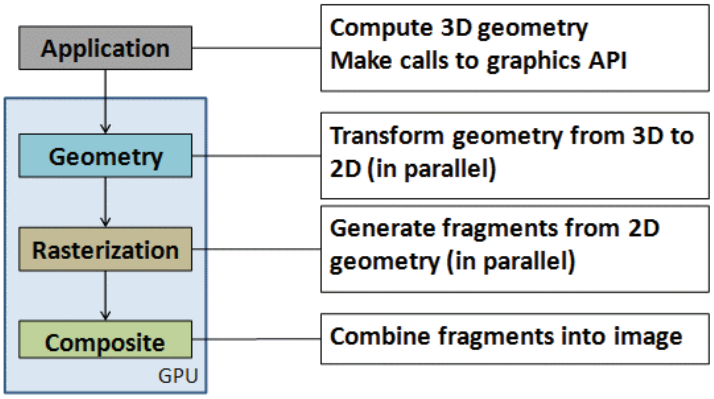

Figure 2. The rendering pipeline of the GPU

As described in the previous section, from the Microsoft's DirectX 8.0 standard on, the developer was able to influence the behaviour of the GPU through the so-called shaders. The geometry transformation can be changed through the vertex shader, the rasterization through the pixel shader.

There were disadvantages of programming GPUs through these shaders. The provided APIs were designed to support graphical APIs and the programmers needed a very deep knowledge of the graphical platform. There were resource constraints; data could be loaded as picture and texture and retrieved data was another picture. So if the algorithm required accessing a memory area to write, it could not run on GPU. It was nearly impossible to predict how your particular GPU can deal with floating-point data. There did not exist any good methods to debug implementations in GPU.

3. New GPU programming languages

Two main programming languages will be described briefly: they are used to develop applications running on GPU, without having any knowledge of the graphical API.

Open Computing Language (OpenCL)

OpenCL is a multivendor open standard for general-purpose parallel programming of heterogeneous systems that include CPUs, GPUs and other processors. OpenCL provides a uniform programming environment for software developers to write efficient, portable code for high-performance computer servers, desktop computer systems and handheld devices. It is managed by the Khronos Group [4]. This standard is applied by industrial companies and academics as well.

OpenCL has four main models

- Platform model

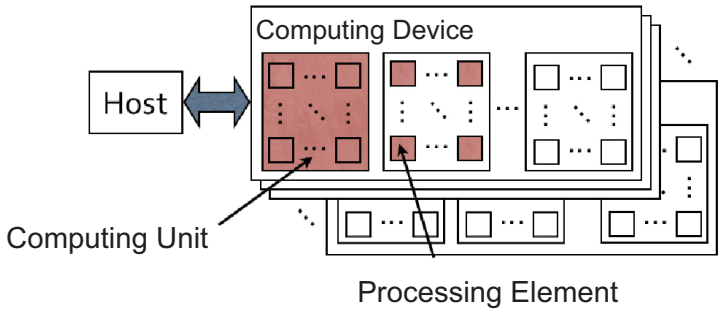

Figure 3. OpenCL platform model [12]

The host coordinates execution and data loading from computing devices. Each computing device has one or more computing units, where one or more processing elements take place.

- Execution model

The host role is context management and controlling of processes. The kernel takes care of controlling the computing units. The kernel program runs on the processing elements, achieves an index range and is grouped into work groups.

- Memory model

There are four main types of memory. The *global memory* is readable/writable from every processing element. Every processing element can access it. The *constant memory* is readable from the processing elements, the host allocates it and fills it with values. The *local memory* is accessed from computing units. Every process element can read/write it within the computing unit. The host cannot access it. The *private memory* is assigned to the processing element and only the assigned process element can read/write it.

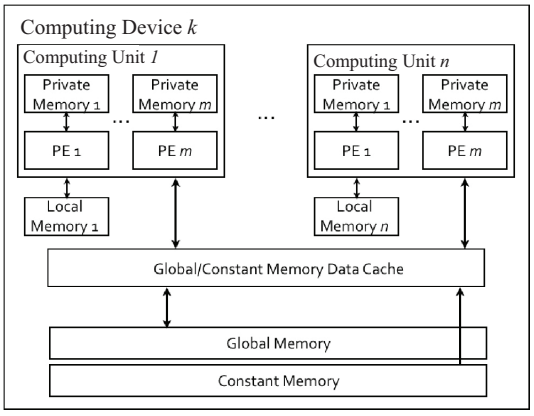

Figure 4. OpenCL memory model [12]

- Program model

  The division of a kernel into work-items and work-groups supports data-parallelism, but OpenCL supports another kind of parallelism as well, called task-parallelism.

OpenCL includes a language for writing so-called kernels. OpenCL provides parallel computing using task-based and data-based parallelism.

CUDA

CUDA was developed by NVidia Inc. and its architecture has two main parts: one is the hardware which supports the CUDA programming and can be called the device and the programming language to be able to create programs using the device capacity. The programming language is based on industry standard C and adds a relatively small number of keywords in order to harness some of the special features of CUDA architecture. There is a public compiler for this language, CUDA C. For further information you can refer to the CUDA Programming Guide by NVidia [5].
As mentioned before, CUDA is an extension of C language and includes GPU relevant APIs and interfaces. There are three main tasks, which are the tasks of the developers, such as thread hierarchy, memory access and synchronisation [6].

Thread hierarchy

To perform computation with CUDA, programmers have to define a special C function, the so-called kernel. This function can be run in a thread using a specified number of lightweight threads in GPU. The kernel is loaded to the device from the

host, where the normal code is running. Threads are grouped into blocks, and blocks form a grid. Threads can communicate through the memory area assigned to the block. The blocks run independently and their behaviour cannot be affected by the programmer.

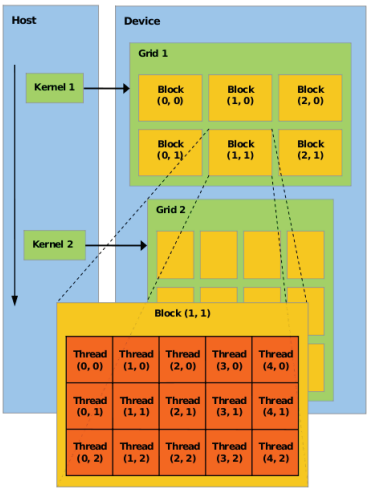

Figure 5. CUDA Thread Hierarchy [7]

Memory hierarchy

There are a great number of types of memory areas, such as global memory, shared memory, constant memory, registers and local memory. The differences between the memory types are the size, the accessing time and the caching property. For more details refer to [5] or to [6].

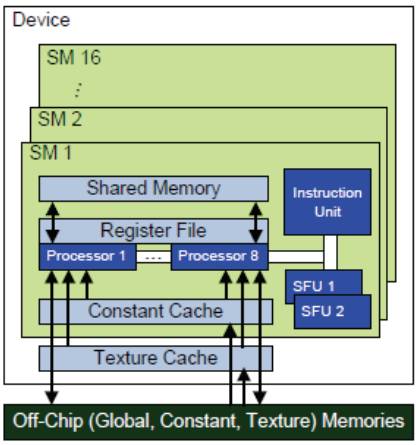

Figure 6. Basic organization of the GeForce 8800 [8]

Synchronisation

Sometimes algorithm behaviour requires a consistent state of the threads within the same block. For example when the shared memory is used by the threads, it could be necessary to have a consistent state in case after writing a thread. The CUDA language provides a __synctrheads( ) method, which defines a waiting point in the kernel code. The processing of a thread will be continued only if each thread reaches this point.

4. Single machine scheduling problems with objective function total weighted completion time

Single machine models are very simple and well-known models in the literature [9] [10]. Only the definition of this model is given here.

There are two main types of objects, the machine and the job. The machine (M) is the processing unit, which is capable of completing the jobs (J). The machine can process only one job at one time and job execution cannot be interrupted. Each job has a processing time (p) on the machine. Each job can have weight (w), which is a priority factor, denoting the importance of the job related to other jobs. Each job can have a completion time (C), which describes the point when the job has been finished. Processing time, weight and completion time will be indexed with j (p_j, w_j, C_j).

Each model can have one or more objective functions, which are used to compare feasible schedulings. This model uses the total weighted completion time as an objective function. This is the summary of the completion time weighted by the priority of a job.

$$TWCT = \sum_{j=1}^{n} w_j c_j \tag{4.1}$$

If there are two feasible schedules, the schedule with the lower TWCT has to be used.

5. First example of implementation

There is a proven theorem for this model [11]: The Weighted Shortest Processing Time First (WSPT) rule is optimal for single machine models, where the objective function is the total weighted completion time. This paper aims at comparing the following implementations for this problem:

- CPU
- GPU with CUDA

Each implementation is based on the following object model. This model is used and generalized later for other problem types.

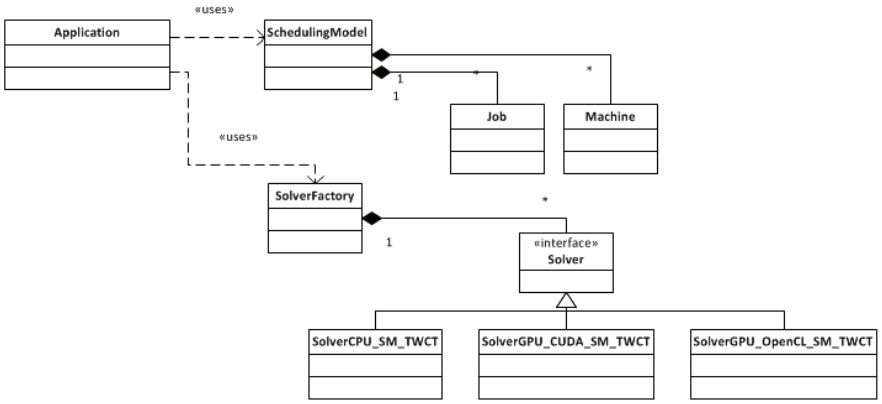

Figure 7. Application object model

Application builds up the scheduling model from a txt file, where the jobs, the job processing time and the job weight are stored. The main flow is described in Figure 8.

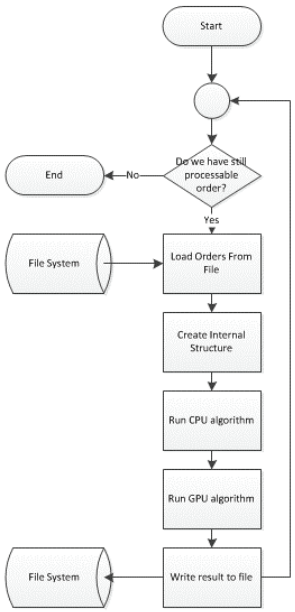

Figure 8. Application main flow

First the file is loaded from the file system and is mapped to the internal structures. In our implementation we always should have enough free memory to store our models. From this general model each algorithm should pre-fill its own data model. During performance measurements this is measured as well. After a running of the algorithms has been finished, the result and the performance result are stored in the file system.

The CPU algorithm

The CPU algorithm uses a vector where the element structure is:

- job identifier
- job processing time
- job weight
- job weight / job processing time calculation result.

The values are calculated in a loop. During one loop phase only one item in the vector can be processed.

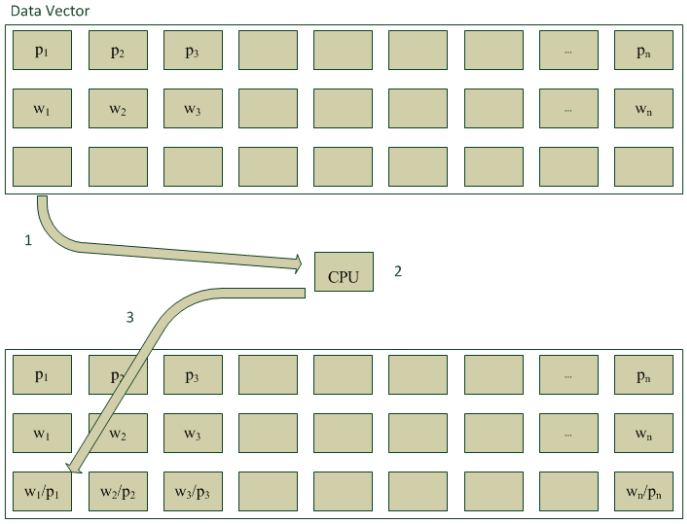

Figure 9. Value calculation flow with CPU

The implementation algorithm is very simple. The data vector is pre-filled with p_j and w_j values of the jobs. There is a loop where an index is used.

Step 1: Values from the index are accessed.

Step 2: The result value is calculated.

Step 3: The result is stored in the memory, the accessing index is increased and the next item will be processed (GoTo Step1) if there exists one.

As can be seen in Figure 9, only one w_j/p_j value is calculated at one time.

The GPU algorithm

The GPU is capable of running the same kernel parallel in several GPU cores. The vector is split into the available blocks and the data are loaded from the host to the device. On the device each processing unit uses the same kernel function and accesses the same memory.

In the single machine model the w_j/p_j values are independent of each other. That is why several independent processing units can be used.

Because the algorithm runs on the GPU, first data have to be loaded from the host (CPU) to the device. Next, memory is allocated to the device, then data are copied from the host. Vectors are used and to each vector item a separated core is assigned. It works only if the number of vector elements is smaller than 65 535. If it is not true, the vector is split into several fragments and the GPU kernel is called more times.

If data is loaded to the device, we start so many parallel kernels as possible to get the highest efficiency. Each processing unit uses the same kernel code.

```
__global__ void gpuCalculateWSPT( int *a, int *b, float *c){
      int tid = blockIdx.x;
      c[tid] = a[tid] / b[tid];
}
```

Step 1: During runtime each GPU core accesses one piece of the data. The indexes come from the CUDA platform and they are called block indexes.

Step 2: The result value is calculated.

Step 3: The result is stored in the device memory.

After the calculations the data have to be copied from the device to the host.

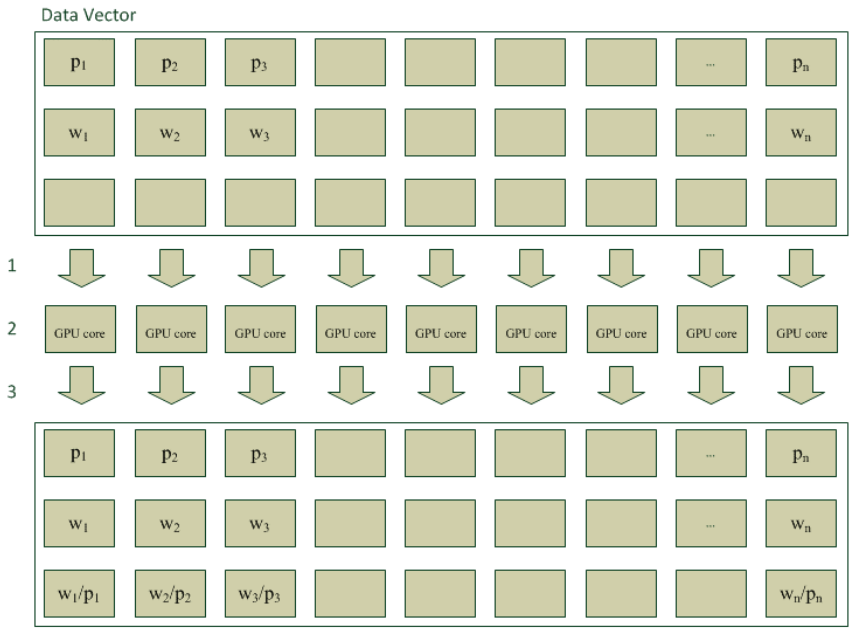

Figure 10. Value calculation flow with CPU

Performance measurements

In order to be able to run performance measurements, a test data set was generated. The main program loads the available test data file from the file system, starts the CPU and GPU solver, measures the processing time and stores the result file back to the file system.

To execute the performance measurements, the following PC configuration was used:

- CPU : Intel Core i3 – 2310M
- GPU: NVIDIA GeForce GT 520M
- Memory: DDRIII 2GB

The test program was developed in C++. The GPU programming model uses the NVida CUDA version 1.1.

The measurements are depicted in the following diagramme.

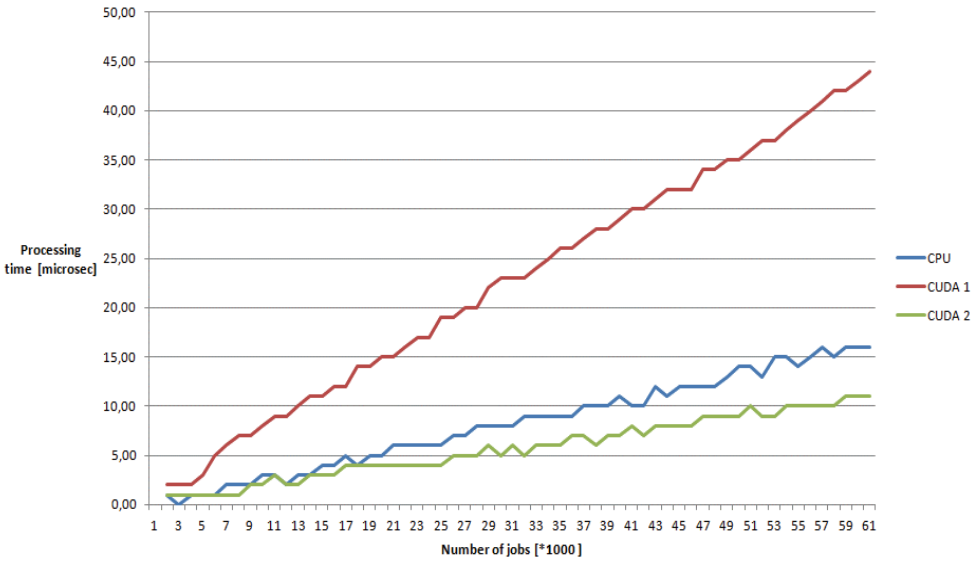

Figure 11. Performance measurements

The axis X shows the number of generated jobs (divided by 1000). The axis Y shows the processing time in microsecunds.

The CPU line shows the processing time of the CPU algorithm.

CUDA 1 line shows the GPU algorithm processing time when the internal data mapping, data copying and the calculation were measured.

CUDA 2 line shows the GPU algorithm processing time when internal data mapping was not necessary. In this case only the processing time of the calculation result and the data copying from the host to the device and vica versa were measured.

Conclusion

As can be seen from the performance results, the solution has O(n) complexity in each case. The CUDA 2 measurement contains the necessary mapping between the structures, the CUDA 1 measurement contains only the calculation of the w_j/p_j rate. As shown, the GPU algorithm has a better performance if the data are structured and stored as appropriate for the GPU.

Acknowledgements

This research was carried out as part of the TAMOP-4.2.1.B-10/2/KONV-2010-0001 project with support by the European Union, co-financed by the European Social Fund.

REFERENCES

[1] SANDERS, J., KANDROT, E.: *CUDA by example,* Addison-Wesley, 2010, p. 3

[2] INTERNET: http://www.nvidia.com/page/geforce256.html, (2012.01.05)

[3] INTERNET: http://www.hardwareinsight.com/nvidia-cuda/, (2012. 01.05)

[4] INTERNET: http://www.khronos.org/opencl/

[5] INTERNET: NVidia Programming Guide 4.0, NVidia Developer Zone http://developer.download.nvidia.com/compute/cuda/4_0_rc2/toolkit/docs/CUDA_C_Programming_Guide.pdf (2011. 08. 25)

[6] CZAPISNKY, M., BARNES, S.: Tabu search with two approaches to parallel flowshop evaluation on CUDA platform, J. Parallel Distrib. Comput. (2011)

[7] INTERNET: http://mohamedfahmed.wordpress.com/2010/05/03/cuda-computer-unified-device-architecture/ (2011. 08. 25)

[8] RYOO, S., RODRIGES, C. I., BAGHSORKHI, S. S., STONE, S. S.: Optimization Principles and Application Performance Evaluation of Multithread GPU Using CUDA

[9] BLAZEVICZ, J., ECKER, K.H., PESCH, E., SCHMIDT, G., WEGLARZ, J.: Scheduling Computer and Manufacturing Processes, Springer

[10] CHEN, B., POTTS, C.N., AND WOEGINGER, G.J.: A review of machine scheduling: Complexity, algorithms and approximability. Handbook of Combinatorial Optimization (Volume 3) (Editors: D.-Z. Du and P. Pardalos), 1998, Kluwer Academic Publishers. 21-169.

[11] PINEDO, M. L.: Scheduling; Theory, Algorithms, and Systems, Springer, 2008, p. 36.

[12] INTERNET: http://developer.amd.com/documentation/articles/pages/opencl-and-the-amd-app-sdk.aspx