EAI Endorsed Transactions
on Internet of Things
Research Article
EAI.EU

A Comparative Study of the Implementation of SJF and SRT Algorithms on the GPU Processor Using CUDA

Youness Rtal [1,*], Abdelkader Hadjoudja [1]

[1] Department of Physics, Laboratory of Electronic Systems, Information Processing, Mechanics and Energy, Faculty of Sciences, Ibn Tofail University, Kenitra, Morocco

Abstract

GPU (Graphical Processing Units) have become in a few years very powerful tools for parallel computing. They are currently used in several fields such as image processing, bioinformatics, medical applications and numerical computation...etc. Their advantages are faster processing and lower power consumption compared to CPU power. It is simple to program a GPU processor using the CUDA C language to perform tasks that are typically computed in parallel. But you need to understand the different architectural aspects of the GPU. In this paper, we will define and implement the two operating system algorithms the SJF (Shortest Job First) algorithm and the SRT (Shortest Remaining Time) algorithm in a single-wire CPU environment using the C language, and then the same algorithms will be implemented on the GPU using the CUDA C language, in order to compare the different performances of the implementation of the two algorithms on GPU and CPU processors and to verify the efficiency of this study.

Keywords: CUDA, GPU, CPU, SRT, SJF, thread.

Received on 07 January 2021, accepted on 28 January 2021, published on 08 February 2021

doi: 10.4108/eai.8-2-2021.168689

1. Introduction

With the emergence of high-level programming languages for graphics processing units (GPUs), GPUs have become more attractive to speed up tasks that are typically performed in parallel. Despite these new languages, it is difficult to use these complex architectures effectively. Indeed, graphics cards are evolving rapidly, with each generation bringing its own features dedicated to accelerating graphics routines or high-performance computing. The architectural details of these architectures remain largely secret, as manufacturers are reluctant to disclose the implementations used. These new features added to GPUs are the result of the simulation of different architectural solutions carried out by manufacturers to determine their validity and performance. The complexity and performance of today's GPUs present significant challenges when exploring new architectural solutions or refining certain parts of the processor.

GPU computing needs are increasing exponentially such as physical simulation [3], risk calculation for financial institutions, weather forecasting, video and audio encoding [4]. So, GPU computing has brought a huge advantage over the CPU in terms of performance (speed and energy efficiency). It is therefore one of the most interesting areas of research and development in modern computing. The GPU is a graphics processing unit that mainly allows us to run high quality graphics, which is the essential demand of the modern computing world. The main task of the GPU is to calculate 3D functions, this type of calculations is very difficult to do on the CPU (central processing unit), the GPU can help us to

*Corresponding author: youness.pc4@gmail.com

work more efficiently because the evolution of the GPU over the years has been oriented towards better floating point performance. In 2006, NVIDIA introduced its massively parallel architecture called "CUDA" and changed the whole perspective of GPU programming. The CUDA architecture consists of several processor cores that work together to consume all the data provided in the application. The use of the GPU to process non-graphical objects is known as the general-purpose graphics processing unit or GPGPU, which is used to perform very complex mathematical operations in parallel to achieve low temporal complexity. The arithmetic power of the GPGPU is the result of its highly specialized computing architecture. [8-9]

A computer necessarily has several processes competing for time processor, this situation occurs when 2 or more processes are in a ready state simultaneously. The Scheduler (scheduler) is the part (a program) of the operating system. responsible for adjusting the status of the processes (Loan, Asset, etc.) and managing the transitions between these processes. states; it is the allocator of the processor to the different processes; it allocates the processor to the process by head of Loans. In this paper, we will implement the two operating system algorithms SJR (Shortest Job First) and SRT (Shortest Remaining Time) on GPU and CPU processors using the CUDA C programming language in order to compare the different performances of the resulting implementation; the rest of our paper is organized as follows: in section 2, we present the CUDA architecture and the hardware used. In section 3, we list the operating system objectives and programming criteria. In section 4, we present the types of planning. Section 5 presents the SRT and SJF algorithms and their advantages and disadvantages. In section 6, we explain the performance of the scheduling algorithms. Section 7 presents the experimental design and the implementation steps. In the last section, we review the results of the implementation of the SRT and SJF algorithms on the GPU and CPU.

2. CUDA architecture and the hardware used

The CUDA environment is a parallel computing platform and programming model invented by NVIDIA [1]. 1] It allows to significantly increase computing performance by exploiting the power of the graphics processing unit (GPU). CUDA C or C ++ is an extension of the C or C ++ programming languages for general computing. CUDA is well adapted and useful for highly parallel algorithms. It is necessary to have multiple threads in order to increase the performance of the algorithms while running on the GPU. Normally, the higher the number of threads, the better the performance. The main idea of CUDA is to have thousands of threads running in parallel. All these threads execute the same code, called kernel. All these threads are executed using the same instructions and different data. Each thread knows its own ID address and based on this ID address it determines the data elements it has to work on. [2]A CUDA program consists of a few steps that are executed on the host (CPU) or on a GPU device. In the host code, no data parallelism phases are executed. In some cases, data parallelism is weak in the host code. In the device code, phases with high data parallelism are executed. A CUDA program is a unified source code that includes both the host and device code. The host code is simple C code compiled using only the standard C compiler. It can be said to be an ordinary CPU process. The device code is written using CUDA keywords for parallel tasks, called kernels and their associated data structures. In some cases, kernels can be run on the CPU if no GPU device is available, but this functionality is provided using an emulation function. The CUDA SDK provides these features. The CUDA architecture consists of three essential parts, which help the programmer to efficiently use all the computing capabilities of the graphics card on the system in question. The CUDA architecture divides the GPU device into grids, blocks and threads in a hierarchical structure, as shown in Figure 1. Since there are several threads in a block and several blocks in a grid and several grids in a single GPU, the parallelism that is achieved using such a hierarchical architecture is very important. [7.12]

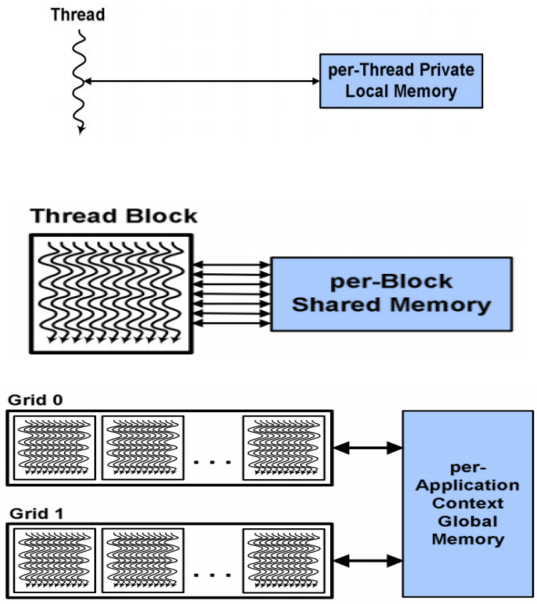

Figure 1. Architecture of the CUDA program and these memories

A grid is a group of many threads running the same kernel. These threads are not synchronized. Each call to CUDA from the CPU is made through a single grid. On multi-GPU systems, grids cannot be shared between different GPUs as they use many grids for maximum efficiency. The grids are made up of many blocks. Each block is a logical unit containing several coordination threads and a certain amount of shared memory. The blocks are no longer shared between the multiprocessors. Each block in a grid uses the same program. A built-in variable "blockIdx" can be used to

identify the current block. Blocks themselves are made up of many threads that run on the individual cores of multiprocessors, but unlike grids and blocks, they are not limited to a single core, there are around 65,535 blocks in a GPU. Like blocks, each thread has its own ID called "threadIdx". Thread IDs can be 1D, 2D, or 3D depending on the block dimensions. The thread ID is relative to the block in which it is located. Threads have a certain amount of register memory. [5, 10] Usually there can be 512 threads per block.

The platform used in this study is a conventional computer, dedicated to video games and equipped with an Intel Core 2 Duo E6750 processor and an NVIDIA GeForce 8500 GT graphics card. All specifications for both platforms are available in [14,15].

The processor is a dual core, clocked at 2.66 GHz and considered entry level in 2007.

The graphics card has 16 streaming processors running at 450MHz and was also considered entry-level in 2007.

In terms of memory, the host has 2GB, while the device has only 512MB.

3. Objectives and Scheduling Criteria

3.1. Objectives of a Scheduler

The main objectives of a Schedule are:

- Maximize CPU and GPU usage
- Present an acceptable response time
- Respect the equity between the processes according to the scheduling criteria used.

3.2. Scheduling criteria

The goal of a scheduling algorithm is to identify the process that will lead to the best possible system performance. Of course, this is a subjective assessment in which various criteria of varying relative importance are taken into account. The scheduling policy determines the importance of each criterion. Several algorithms have been proven in the implementation of a scheduling policy. [13]

The following list reviews frequently used scheduling criteria:

- CPU Usage: Percentage of time the CPU is running a process. The importance of this criterion generally varies depending on the degree of sharing of the system.
- Distributed use: Percentage of time during which all resources are used (in addition to CPU, memory, I / O device, etc.)
- Throughput: Number of processes that can be executed by the system over a given period.
- Turnaround time: The average length of time it takes for a process to run. The turnaround time of a process includes all the time it spends in the system. It is inversely proportional to the flow.
- Waiting time: The average time a process spends waiting. Measuring performance by turnaround time has one drawback: The production time of the process increases the turnaround time; The wait time is therefore a more precise measure of performance.
- Response Time: The average time it takes for the system to start responding to user input.
- Fairness: degree to which all processes are given an equal opportunity to perform.
- Priorities: Gives preferential treatment to processes with a higher priority level.

4. Types of Scheduling

There are 2 types of scheduling [13]:

Preemptive Scheduling: With requisition where the Scheduler can interrupt a running process if a new higher priority process is inserted in the Loans queue.

Non-preemptive scheduling: Scheduling until completion: the elected process retains control until the time allotted to it is used up even if higher priority processes have reached the Loans list

5. The SJR and SRT algorithm

We are going, according to the problems which we have just exposed and the partial solutions which have been envisaged, to present here some realizations of schedulers while showing for each one of them the advantages and the disadvantages towards the system and towards the users. [13]

5.1. Shorter Work First (SJF)

The Shorter First technique (SJF for Shortest Job First) is still a scheduling scheme without requisition (therefore unusable in time-sharing) where the process with the lowest estimated time of execution until completion takes priority. It is therefore a technique that was created to partially overcome the disadvantage of First In First Out which allowed the execution of very long works before less important works, only their order of arrival being considered.

SJF therefore favours short work to the detriment of the most important. As a result, it leads to a much greater variance than the First In First Out algorithm, in particular for long jobs. SJF works so that the next run can complete (and therefore exit the system) as soon as possible. This technique therefore tends to reduce the number of pending jobs, which has the consequence of reducing the average process waiting times.

The main disadvantage of SJF is that it requires precise knowledge of the execution time, a value that is usually not possible to determine. The only way is to trust an estimate given by the users themselves. This estimate may be good in production environments where the same jobs are submitted regularly, but it is rarely possible in development environments.

Knowledge of this scheduling scheme might tempt some to intentionally underestimate execution time in order to take advantage of undue priority. In order to avoid this kind of "dishonesty", the user is warned that his work will be abandoned if it is exceeded. This has two drawbacks:

- Obligation for users to increase the estimates.
- Poor profitability of the processor (the time spent on abandoned jobs quickly decreases performance).

A second possibility is therefore offered: to continue the execution of the work during the estimated time increased, if necessary, by a certain percentage (generally low) then to "put aside" in the state in which it is. to resume execution later. Of course, the user will be penalized by this wait but also by an additional billing.

A third possibility is not to "set aside" the work, but to continue its execution until completion by charging for the excess time at a much higher rate. This solution is ultimately better accepted because the supplement effectively corresponds to better service.

5.2. The Shortest Remaining Time algorithm (SRT)

The Shortest Remaining Time strategy (SRT for Shortest Remaining Time) is the version with requisition of SJF (therefore usable in timesharing) where, again, priority is always given to the process with the lowest remaining execution time (in considering new arrivals).

In SRT, an active process can therefore be interrupted in favour of a new process having an estimated execution time shorter than the time required for the completion of the first. Here again, and more particularly because of the requisition, the "designer" must provide a deterrent about "clever" people who know the scheduling strategy.

The cost of SRT is higher than that of SJF: it must take into account the time already allocated to the running processes, perform the switches on each arrival of a short job which will be executed immediately before resuming the interrupted process (unless even shorter labor occurs). Long jobs experience a longer average wait and a larger variance than in SJF.

In theory SRT should offer a minimum waiting time, but due to its own cost of operation, it is possible that in certain situations, SJF is more efficient. In order to reduce this cost, one can consider several refinements avoiding requisition in borderline cases:

- suppose the current process is almost complete and a job with a very low estimated execution time arrives. Should there be requisition? In these cases, it is possible to guarantee that a process in progress whose remaining execution time is less than a threshold is completed regardless of the arrivals;
- another example: the active process has an execution time remaining slightly higher than the estimated time of an incoming job. Here again, if SRT is applied "to the letter", there is requisition. But if the cost of this requisition is greater than the difference between the two estimated times, this decision becomes absurd!

The conclusion of all this is that the "designers" of systems must carefully assess the costs generated by sophisticated mechanisms because they can in many cases defeat the desired goal: saving time.

6. Performance of Scheduling Algorithms

The performance of an algorithm for a given set of processes can be analysed if the appropriate process information is provided. For example, data on the arrival of the process and the time of execution of that process are needed to evaluate these algorithms. We give some parameters to calculate for the implementation of the two algorithms:

- Bust Time (BT) and Arrival Time (AT).
- Waiting time = start time – arrival time
- Turnaround Time = Waiting Time + Burst Time for all processors.
- Average Waiting Time = Sum Waiting Time / number of processors.
- Average Turnaround Time = Sum Turnaround Time / number of processors.
- the acceleration is given by: $\text{Speedup} = T_S/T_P$, Where, T_S is the time required to run the sequential algorithm on CPU and T_P is the time required to run the parallel algorithm on GPU.

7. Experimental Setup and Implementation

The Software Development Kit or the SDK may be a good way to learn a few about a CUDA, anyone will compile the examples and can learn how the toolkit works. The SDK is available at the NVIDA's website and can becdownloaded by any aspiring programmers who wants to learn about a CUDA programming. Anyone who has some basic knowledge about C programming can begin a CUDA programming very quickly. No prior knowledge of graphics programming is required to write CUDA codes. CUDA is derived from C with some modifications that mode it works on the GPU. CUDA is a C for GPU.

The main objective of the work is to analyse some operating system scheduling algorithms on GPU and CPU. The first criterion for evaluating the programming of the central units is the waiting time and the burst time of the processes which are in the same conditions. This paper implemented some scheduling algorithms, namely Shortest-Job-First (SJF) scheduling, Shortest Remaining Time (SRT) scheduling, first on a single-threaded CPU environment and calculated the execution time of each algorithm, then, the same algorithms are implemented with NVIDIA's GPU programming environment, CUDA v10.2. Then by comparing the performance of these algorithms on both CPU and GPU platforms using the CUDA language.

The steps to implement the CUDA code are as follows:

- Install visual studios as an environment for CUDA programming.
- Install specific NVIDIA GPU drivers according to GPU model and install the CUDA SDK.
- Write a program code according to a normal C / C ++ programming construct.

- Change the written program into the CUDA parallel code by using the library functions provided by the SDK. The library functions are used to copy data from host to device, change execution from CPU to GPU and Vice versa, copy data from device to host.
- Allocate CPU memory.
- Allocate same amount of GPU memory using library function "CudaMalloc".
- Take data input in CPU memory.
- Copy data into GPU memory using library function CudaMemCpy with parameter as (CudaMemcpyHostToDevice).
- Perform processing in GPU memory using kernel calls. Kernel calls are a way to transfer control from CPU to GPU; they also specify the number of grids, blocks and threads i.e. Parallelism is required for your program.
- Copy final data in CPU memory using library function CudaMemCpy with parameter as (CudaMemcpyDeviceToHost).
- Free the GPU memory or other threads using library function Cudafree.

Setting up the environment and writing programs in CUDA is a straightforward task. But it requires a deep knowledge of architecture and knowledge of writing parallel codes. The most important part of programming in CUDA is the kernel calls in which the programmer must determine the parallelism required by the program. Dividing data into the appropriate number of threads is the main area that defines successful code. [10]

8. Results and Discussions

In order to analyse the performance of the implemented SRT et SJF algorithms the speedup achieved on the execution with respect to time was evaluated for all the test results. All the tests on the algorithms were performed with the similar number of processing nodes or processors and therefore, the speedup in execution is not evaluated based on the number of processors used but by analysing the speedup in execution time because of the change in parallelizing approach taken up in the program [9.11]
this implementation is based on the algorithm:

```
forever
    p: = priority;
    if (state (p) ≠ ready) do
        p: = following (p);
    end − to − do
    restore the context of (p);
    to give a hand to (p);
end-for
```

In this program, we have assumed the list ordered by decreasing priority starting from the entry <<priority>>, and moreover looped. The results of the implementation of the SJF and SRT algorithms are grouped together in table 1 and the two figure 2 and 3:

Table 1. The different performances of the implementation of the SRT and SJF algorithms on GPU and CPU

Algorithm Used	Process	Burst Time (BT)	Arrival Time (AT)	Waiting Time (WT)	Turnaround Time	Average Waiting Time		Average Turnaround Time		CPU Time T_S in (ms)	GPU Time T_P in (ms)
						CPU	**GPU**	**CPU**	**GPU**		
Shortest Job First (SJF)	P2	12	0	0	12						
	P3	8	3	19	27						
	P4	4	4	7	11	10	10	18	18	0,992	0,0681
	P1	10	8	20	30						
	P5	6	10	4	10						
Shortest Remaining Time (SRT)	P2	12	0	18	30						
	P3	8	3	4	12						
	P4	4	5	0	4	9	9	17	17	0,879	0,054
	P1	10	10	20	30						
	P5	6	12	3	9						

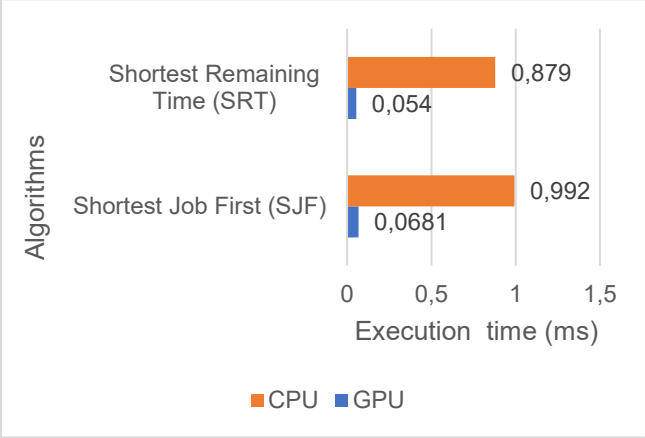

Figure 2. The execution time obtained by implementing two algorithms on GPU and CPU

The results of Table 1 and Figure 2 show that for the same data from the processors of two algorithms, SRT and SJF, we have:

- GPU and CPU having the same average wait time and the same average execution time.
- The average wait time executed on the GPU of the SRT algorithm is higher than that of the SJF algorithm.
- Average execution time calculated by the GPU of the SRT algorithm smaller than the SJF algorithm.
- The execution time on the GPU of both algorithms is about 14 to 17 times faster than the CPU.
- The GPU execution time of the SRT algorithm is smaller than that of the SJF algorithm.

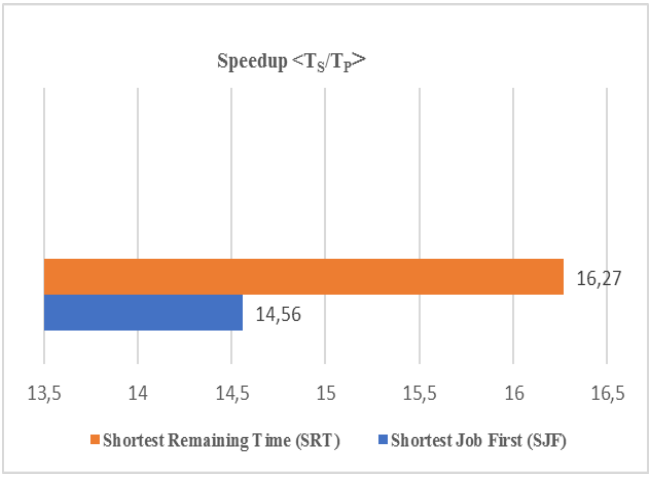

Figure 3. represents the Speedup of two algorithms SRT and SJF

Figure 3: shows that the acceleration of SRT algorithms is higher than that of the SJF algorithm, implying that the acceleration factor of preemptive planning algorithms is faster than that of non-preemptive planning algorithms. The implementation results shown in Table 1 and Figures 2 and 3 explain that CPU processors process data sequentially (task by task), while GPUs process data in parallel (several tasks simultaneously), implying that the performance of the implementation of two algorithms on GPU is very high compared to the implementation on CPU.

9. Conclusion

In this paper, we have successfully presented a comparative study of the implementation of SRT and SJF algorithms on GPUs and CPUs using the CUDA C language, the latest features of the NVIDIA CUDA SDK 10. 2, the results of the implementation show that the execution time achieved on the GPU for both algorithms is faster than on the CPU (about 14 to 17 times) and the acceleration factor of the SRT algorithm is higher than that of the SJF algorithm, showing that the performance of SRT (Preemptive) algorithms is more efficient than that of SJF (Non-Preemptive) algorithms. This shows the efficiency of using GPUs for parallel computing and obtaining the best performance. Despite this implementation by CUDA C, Nvidia still faces many challenges to keep CUDA C faithful to parallel task programming on GPUs, the main task being to convince programmers that it is a credible platform. GPUs are gaining popularity in the scientific computing community due to their high processing power and easy availability and are becoming the preferred choice of programmers due to the support offered to programmers by models such as CUDA.

Abbreviations

The following abbreviations are used in this manuscript:

GPU	Graphical Processing Unit.
CPU	Central Processing Unit.
CUDA	Compute Unified Device Architecture.
SJF	Shorter Work First.
SRT	Shortest Remaining Time.

References

[1] NVIDIA. NVIDIA CUDA Compute Unified Device Architecture Programming Guide, Version 2.0, 2008.
[2] Wikipedia- http://en.wikipedia.org/wiki/CUDA.
[3] CalleLedjfors, “High Level GPU Programming”, Department of Computer Science Lund University.2008.
[4] “CUDA C programming guide version 6.5”, NVDIA Corporation, August 2014.
[5] Anthony Lippert – “NVIDIA GPU Architecture for General Purpose Computing”.
[6] Danilo De Donno et al., “Introduction to GPU Computing and CUDA Programming: A Case Study on FDTD,” IEEE Antennas and Propagation Magazine, June 2010.
[7] Manish Arora, “The Architecture and Evolution of CPU-GPU Systems for General Purpose-Computing “.

[8] David Tarditi, Sidd Puri, Jose Oglesby, “Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses”, October 2006.
[9] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Kevin Skadron, “A Performance Study of General-Purpose Applications on Graphics Processors Using-CUDA”.
[10] Yadav K., Mittal A., Ansari M. A., Vishwarup V., “Parallel Implementation of Similarity Measures on GPU Architecture using CUDA”.
[11] Maria Andreina F. Rodriguez, “CUDA: Speeding Up Parallel Computing”.
[12] Jayshree Ghorpade, Jitendra Parande, Madhura Kulkarni, Amit Bawaskar, “GPGPU PROCESSING IN CUDA ARCHITECTURE” Advanced 12 Computing: An International Journal (ACIJ), Vol.3, No.1, January 2012.
[13] M Merci - bibliothequer.com.
[14] http://ark.intel.com/Product.aspx?id=30784.
[15] http://www.nvidia.com/object/geforce_8500.html.