Realtime scheduling using GPUs - proof of feasibility

PETER FODREK*, ĽUDOVÍT FARKAS*, TOMÁŠ MURGAŠ**
*Institute of Control and Industrial Informatics
Faculty of Electrical Engineering and Information Technology
Slovak University of Technology
Ilkovičova 3, 812 19 Bratislava
SLOVAK REPUBLIC

**RT Systems s.r.o.
Kopčianska 14, 851 01 Bratislava
SLOVAK REPUBLIC
peter.fodrek@stuba.sk, ludovit.farkas@stuba.sk, tomas.murgas@rtsystems.sk

Abstract: - This paper will report our evaluation to use openCL as a platform for hard realtime scheduling. Specifically, we have evaluated which types of tasks are faster on GPGPU than on CPU. We have investigated computational tasks, memory intensive tasks (especially tasks using low latency GDDR memory) and disk intensive tasks. This study is the first part of a larger research program to design an innovative Linux scheduler subsystem that runs on GPGPU and schedules tasks running on GPGPU as well as on CPU. Based on the results obtained from benchmarking various types of tasks, we found out that some of them are faster on GPGPU than on CPU and therefore should preferably be executed on GPGPU. Preliminary data suggest that we can expect a speed up of up to 10-fold with respect to execution time and latency.

Key-Words: - OpenCL, task scheduling, GPGPU, proof of feasibility, scheduling performance

1 Introduction

There are several algorithms to achieve low latency, which is a requirement for hard real-time operating systems for automation. But there is not a known algorithm that is designed to run on General Purpose Graphics Processing Units (GPGPUs).

MAC OS X 10.6 and 10.7 and their Darwin named kernels are known for GPGPUs usage to speed-up kernel tasks performance. One project named KGPU that allows GPGPU performance for Linux kernel is known. KGPU uses nVidia only CUDA toolkit but there is a task to allow the running of KGPU on AMD GPUs.

There is also another scheduler intended for soft-realtime environment presented in [1]. It is called Timegraph and it was evaluated on NVIDIA drivers.

AMD improved their drivers and switched Ati to more universal and open standard named OpenCL (Open Computing Language) and AMD added support for openCL to AMD CPU device drivers. Therefore programs written for openCL are portable and can run on AMD CPUs and GPUs and nVidia and Intel GPUs. This is a reason to do research using openCL for the scheduling and other kernel tasks.

The use of GPGPUs for hard realtime scheduling would be an important contribution to hard realtime operating systems that are used for realtime control. This is a new topic in the research, so there cannot be found much literature about it. This paper is one of the first papers dealing with this issue. We can state this according to the e-mail communication in which our workgroup is described as the first in Europe dealing with realtime scheduling using GPGPUs.

In the rest of the paper we will show a basic methodology for benchmarking the speed of application execution and a method for comparing the execution times between the applications that run on the GPGPU and the applications that run on the CPU.

2 Prerequisities

In this section we divided kernel tasks to several types as shown in the subsections of this section.

2.1 Computing intensive tasks

Computing intensive tasks are tasks where the dominant load is based on computational performance. In the kernel tasks there are tasks that

ISBN: 978-1-61804-056-5

are computing intensive. There are at least these types of them:

- scheduling algorithms
  - on-line
  - off-line
- inter-process communications
  - message queues
  - signal handling a dispatch
- resource management tasks to achieve maximum real-time performance with minimal energy cost

2.2 Memory intensive tasks

There are two types of memory intensive tasks used in the kernel space using GPGPU. Let us call and classify them as of what type of memory they use:

- local - to store and transfer data using only one part of the system
  - for CPU using CPU memory - (RAM/DDR)
  - for GPU using GPU/GPGPU memory - (VRAM/GDDR)
- global - to transfer data between GPU and CPU and vice versa

3 Previous GPGPU solutions in the Linux kernel

There is a project named KGPU. The main idea of KGPU is a GPU computing framework for the Linux kernel. It allows Linux kernel to call CUDA programs running on GPUs directly. The motivation is to augment operating systems with GPUs so that not only user-space applications but also the operating system itself can benefit from GPU acceleration. It can also free the CPU from some computation intensive work by enabling the GPU as an extra computing device [2, 3]. This project was started on March 28th, 2011 by Weibin Sun. The main disadvantage of this project is its linkage only to nVidia based GPUs because of using CUDA library only.

This may be the problem of KGPU project. It is not designed as universal framework from the beginning. This may cause KGPU troubles for future implementations.

4 Benchmarks

In this section we are going to show the testing scripts for evaluation of the possibility to use GPGPU in the schedulers of the kernel.

4.1 Testing script for time measurement

For the measurement of the program running-time we used a listed script because the shell built-in *time* command was unable to achieve output to file. This is the negative side of that command because it measures time in range of milliseconds not tenths of milliseconds as shown in the script

```
#!/bin/bash
rm $2
for i in `seq 1 100`;
do
   echo $i
   echo $i >>$2
   sleep 1
   /usr/bin/time -p --output=$2 --append  $1>/dev/null
   echo >>$2
   sleep 1
done
```

The listed script runs the test sequence 100 times to achieve the desired quality and relevancy of the measurement. We used the mean values and the standard deviation to conclude the usability of such technique.

The format of the output was not well formatted for computer post-processing such as standard deviation, average and median count. A more suitable data format for statistical processing is the format named Comma separated values (csv). To produce output in this format we have created another script listed below

```
#!/bin/bash
rm $2
echo "Total, Userspace, kernelspace" >>$2
for i in `seq 1 100`;
do
   echo $i
   #echo $i >>$2
   sleep 1
   /usr/bin/time -f "%e, %U, %S" --output=$2 --
append  $1>/dev/null
   #echo >>$2
   sleep 1
done
```

4.2 Application for running OpenCL codes

For the compilation and running of our OpenCL testing codes we used a program heavily based on [4]. This program is suitable for speed tests. The code contains some bugs like not checking response from the function calls. The condition for its use is that the user has to be sure that the program is running. Sometimes we had to use a smaller count of the parallel processing OpenCL kernels because

when using maximum count of them, the OpenCL program could not run. We recommend the use of half of the total count of the vector elements or vector dimensions for parallel OpenCL kernel. This issue will be one of the goals of our future research. The program has to set the same number of the kernels and vector elements.

4.3 OpenCL kernels

We have written three types of OpenCL codes to test three different criterions for evaluating the use of GPGPU. The first criterion has been the execution of mathematical tasks, the second has been the videoRAM transfer and the third has been the file read. We have compared the evaluation times with comparable functions written in standard C codes. The OpenCL codes are listed below:

For mathematical tasks

```
__kernel void vector_add(__global const int *A, __global
const int *B, __global int *C)
{
  // Get the index of the current element to be processed
  int i = get_global_id(0);

  // Do the operation
  C[i] = A[i] + B[i];
}
```

For VideoRAM transfer

```
__kernel void vector_add(__global  int *A, __global  int
*B, __global int *C)
{
  // Get the index of the current element to be processed
  int i = get_global_id(0);

  // Do the operation
  C[i] =  B[i];
  B[i]=A[i];
  A[i]=C[i];
}
```

For file read

```
__kernel void file_add(__global  const int *A, __global
const int *B, __global int *C)
{
  // Get the index of the current element to
//be processed
  int i = get_global_id(0);
  int fp;

  // Do the operation
  fp=open("./dataFile.dat",O_CREATE|O_RDONLY);
  C[i]=fp;
  close(fp);
}
```

5 Benchmark results

In this section we are going to show the results of our benchmarks by comparing the measured running time data of the programs executed by the GPU and the standard programs written in C executed by the CPU. The results are in the form of histograms of the program execution times.

Fig. 1 shows the execution times of a program with 1000000 computations and 1 data transfer between the CPU and the GPU. We have made the same experiment with 100 times more computations per experiment and again 1 data transfer as seen in Fig. 2. The mean value is 55.24 seconds and the standard deviation is 10.29 seconds. The median value is 56.31 seconds. This is a deviation of about 5 % from the mean value.

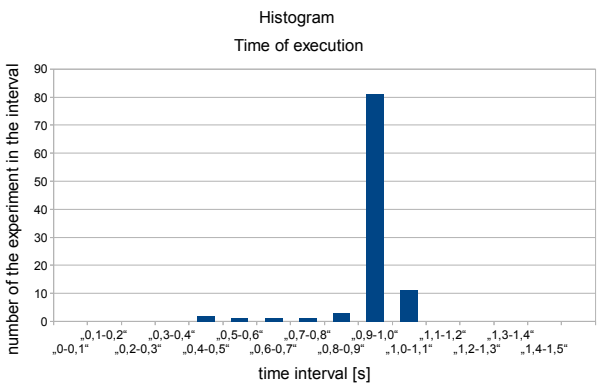

Fig. 1: Execution time intervals for 1 data transfer between the CPU and the GPU and 1000000 computations.

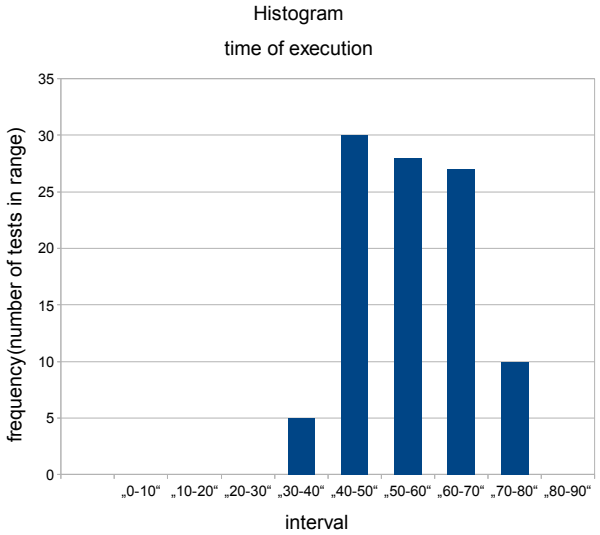

Fig. 2: Execution time intervals for 1 data transfer between the CPU and the GPU and 100 times more computations per experiment.

This shows us, that the execution times are random variables using standard distribution.

ISBN: 978-1-61804-056-5

Fig. 3 shows the results of experiments with not just 1 data transfer per experiment, but with a data transfer before and a data transfer after a computation. The number of computations per experiment has been 1000000.

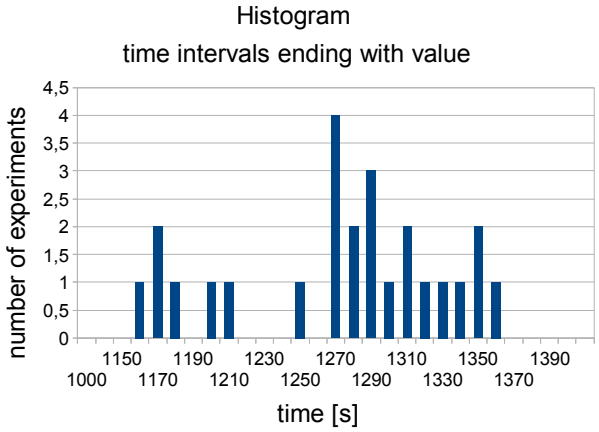

Fig. 3: Execution time intervals for programs with a data transfer before and after each computation

Because the experiments have been too long, we have made only 25 experiments instead of 100. The mean value has been 1268.06 seconds with a standard deviation of 59.47 seconds and the median has been 1276.82 seconds. The addition of data transfer between the computations raised the execution interval by 1358 times. It seems that the data transfer between the RAM and the VRAM is too time-consuming.

We have compared the execution times of the OpenCL programs with comparable programs written in C. Execution times for pure C code without memory transfers are shown in Fig. 4.

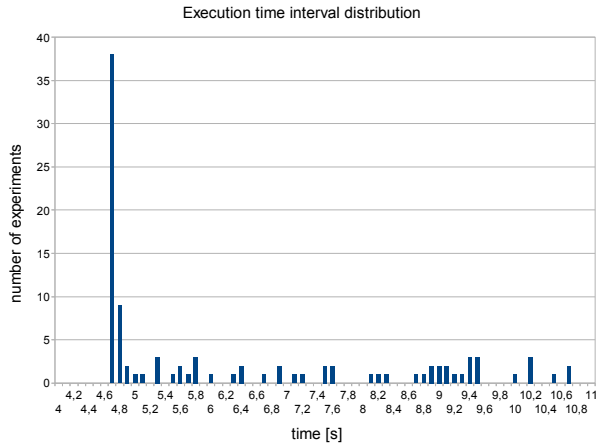

Fig. 4: Execution time intervals for pure C code without memory transfer

The mean value of the execution times of pure C code has been 6.271 seconds with a high standard deviation of 2 seconds what is about 1/3 of the mean value. The median has been 5.015 seconds what is about 20 percent difference to the mean value. These values are too high to say that the execution time of the standard C code is random with a standard distribution. It also cannot be stated that the C code execution is deterministic in standard Linux kernel.

Fig. 5 shows the next experiment, which has been a standard C code with additional data transfers between RAM elements. The mean value has been 14.53 seconds with a very high standard deviation of 3.662 seconds what is about ¼ of the mean value. The median is 12.18 seconds what is about 16-17 % difference to the mean value. This difference is extremely high to see a predictability of the execution interval length.

It means that the OpenCL code running time constraint on a GPGPU is more predictable than the C language based code running on the CPU.

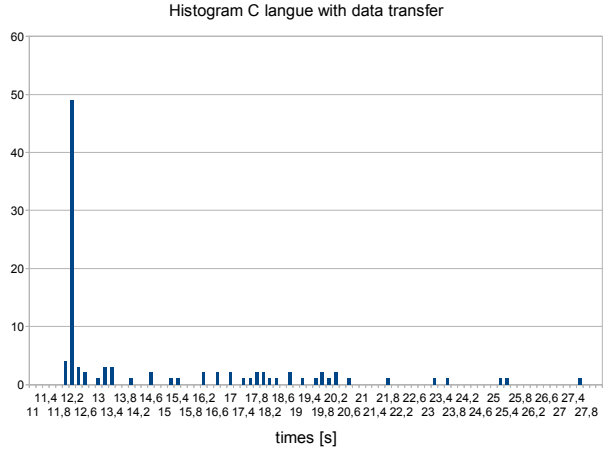

Fig. 5: Execution time intervals for pure C code with additional memory transfers

6 Summary of the experiments

All of the proposed experiments are summarized in Tab. 1.

Experiment type	Mean value [s]	Standard deviation [s]	Median [s]
C code without data transfer	6.271	2.000	5.015
C code with data transfer	14.530	3.662	12.180
OpenCL code without data transfer	0.940	0.100	0.940
OpenCL code with data transfer	1268.060	59.470	1276.820

Tab. 1: Summary of the execution times

ISBN: 978-1-61804-056-5

As seen in Tab. 1, the OpenCL application is much faster than a standard application written in C. This is known by pure FLOPS performance difference between the GPGPU and the CPU. In our scenario, the Geforce 310M GPGPU's performance is 6 times higher than the Core I5 520M CPU in a Dell Vostro 3300n laptop.

Based on our research we are able to state the next recommendations:

Recommendation 1: *Do not use any value based argument in real-time or kernel code. Use only reference based arguments for any C function you ever use. This is a must for any hard real-time code and for kernel code as well.*

This is a standard in system programming but not a standard in real-time programming.

Recommendation 2: *When using GPGPU for executing applications, minimize data transfers between the CPU and the GPGPU.*

This was never mentioned before so far we know.

For the scheduling using GPGPU we are able to define this recommendation:

Recommendation 3: *When you are trying to do scheduling using GPGPU performance, then you are not able to use on-line scheduling, but you have to use off-line scheduling algorithms like Computing Schedules for Time-Triggered Control using Genetic Algorithms [5].*

7 Conclusion

We have developed a basic methodology for benchmarking the speed of application execution and a method for comparing the execution times between the applications that run on the GPGPU and the applications that run on the CPU. We have found out that the data transfer is the main blocking issue to use the GPGPU in on-line scheduling.

There are off-line algorithms [5] that are suitable to be used on GPGPU effectively for hard real-time scheduling.

The main problem is that an off-line algorithm must assume the number of scheduling time slots for tasks starting in the future before a re-run of the scheduling algorithm occurs.

One of the main advantages of our approach is that using GPGPU is more flexible then using pure hardware co-processor based scheduling as mentioned in [6].

Our research is just in the beginning, but we hope it will be a notable contribution to the use of GPGPUs in hard realtime operating systems.

8 Acknowledgments

This work was supported by the Slovak Research and Development Agency under contract No. VMSP-II-0034-09. It has been supported by the project Req-00048-001 too.

References:

[1] S. Kato, K. Lakshaman, R. Rajkumar, Y. Yshikawa, TimeGraph: GPU Scheduling for Real-Time Multi-Tasking Environments, In *2011 USENIX Annual Technical Conference Proceedings*, June 15-17, 2011, Portland, OR USA, pp. 17 – 30.

[2] W. Sun, R. Ricci, X. Lin, Kgpu augmenting linux with the gpu, 2011. http://code.google.com/p/kgpu/.

[3] W. Sun, R. Ricci, Augmenting Operating Systems With The Gpu, http://www.cs.utah.edu/~wbsun/kgpu.pdf

[4] E. Smistad, Getting started with opencl and gpu computing, 2011. http://www.thebigblob.com/getting-startedwith-opencl-and-gpu-computing/.

[5] T. Nghiem, G. E. Fainekos, Computing schedules for time-triggered control using genetic algorithms. In *Preprints of the 18th IFAC World Congress,Milano (Italy) August 28 – September 2, 2011 (2011),* International Federation of Automatic Control, pp. 794–799.

[6] V. Domen, Co-processor for microkernel os services. *In Preprints of the 18th IFAC World Congress,Milano (Italy) August 28 – September 2, 2011 (2011)*, International Federation of Automatic Control, pp. 1946–1951.