EFDA–JET–CP(10)04/01

J. Nieto, G. de Arcas, J. Vega, M. Ruiz, J.M López, E. Barrera, A. Murari
A. Fonseca and JET EFDA contributors

Exploiting Graphic Processing Units Parallelism to Improve Intelligent Data Acquisition System Performance in JET's Correlation Reflectometer

“This document is intended for publication in the open literature. It is made available on the understanding that it may not be further circulated and extracts or references may not be published prior to publication of the original when applicable, or without the consent of the Publications Officer, EFDA, Culham Science Centre, Abingdon, Oxon, OX14 3DB, UK.”

“Enquiries about Copyright and reproduction should be addressed to the Publications Officer, EFDA, Culham Science Centre, Abingdon, Oxon, OX14 3DB, UK.”

The contents of this preprint and all other JET EFDA Preprints and Conference Papers are available to view online free at www.iop.org/Jet. This site has full search facilities and e-mail alert options. The diagrams contained within the PDFs on this site are hyperlinked from the year 1996 onwards.

Exploiting Graphic Processing Units Parallelism to Improve Intelligent Data Acquisition System Performance in JET's Correlation Reflectometer

J. Nieto[1], G. de Arcas[1], J. Vega[2], M. Ruiz[1], J.M López[1], E. Barrera[1], A. Murari[3]
A. Fonseca[4] and JET EFDA contributors*

JET-EFDA, Culham Science Centre, OX14 3DB, Abingdon, UK

[1]Universidad Politecnica de Madrid, Spain
[2]Asociación EURATOM-CIEMAT, Avenida Complutense 22, E-28040 Madrid, Spain
[3]Associazione EURATOM-ENEA sulla Fusione, Consorzio RFX Padova, Italy
[4]Associação EURATOM / IST, Instituto de Plasmas e Fusão Nuclear, Lisbon, Portugal
** See annex of F. Romanelli et al, "Overview of JET Results",*
(Proc. 22 nd IAEA Fusion Energy Conference, Geneva, Switzerland (2008)).

Preprint of Paper to be submitted for publication in Proceedings of the
17th IEEE NPSS Real Time Conference, Lisbon, Portugal.
(24th May 2010 - 28th May 2010)

ABSTRACT.
The performance of intelligent data acquisition systems relies heavily on their processing capabilities and local bus bandwidth, especially in applications with high sample rates or high number of channels. This is the case of the self adaptive sampling rate data acquisition system installed as a pilot experiment in KG8B correlation reflectometer at JET. The system, which is based on the ITMS platform, continuously adapts the sample rate during the acquisition depending on the signal bandwidth. In order to do so it must transfer acquired data to a memory buffer in the host processor and run heavy computational algorithms for each data block. The processing capabilities of the host CPU and the bandwidth of the PXI bus limit the maximum sample rate that can be achieved, therefore limiting the maximum bandwidth of the phenomena that can be studied.

Graphic processing Inits (GPU) are becoming an alternative for speeding up compute intensive kernels of scientific, imaging and simulation applications. However, integrating this technology into data acquisition systems is not a straight forward step, not to mention exploiting their parallelism efficiently.

This paper discusses the use of GPUs with new high speed data bus interfaces to improve the performance of the self adaptive sampling rate data acquisition system installed on JET. Integration issues are discussed and performance evaluations are presented.–

I. INTRODUCTION

The pursuit of more performing plasma scenarios in reactor class devices, with high energy content and long pulses, has motivated the measurements of more quantities, with higher spatial and time resolution, leading to an exponential increase in the amount of data to be analyzed, which cannot be handled with traditional methods [1]. The traditional paradigm of analyzing the data after the shot must be abandoned, and approaches such as the one proposed through the Intelligent Test and Measurement System (ITMS) platform [2] must be adopted to provide more information during the shot. The pilot data acquisition system developed for JET's KG8B correlation reflectometry [3] is an example of such a system that is being used to analyze potential issues and candidate solutions of this new approach. The system implements a self adaptive sampling data acquisition mechanism, which consists of acquiring data with a variable sampling rate which is continuously adapted depending on the bandwidth of the input signals, therefore optimizing the volume of data generated without loosing any information. In order to achieve this goal the system must process considerable amount of data in real time to compute the bandwidth of the input signals. This is one of the typical problems of these systems which leads to the need of multi-parallel scalable processing capabilities in the data acquisition architectures.

ITMS provides a framework to develop such applications using high level development tools such as LabVIEW [4], and offers the programmer several mechanisms to increase the processing capabilities of the system, from additional processing units (PCPUs) to FPGA-based cards . In this paper, the possible use of GPUs in this architecture as an alternative to icrease its processing capabilities is analyzed. GPU are being used mainly to speed up offline data processing codes, such

as simulations or data analysis codes, but might be used to analyze the data during data acquisition under certain circumstances. The objective of this work is to discuss the easiness of integration of these new tools in existing architectures and to evaluate their performance in a generic signal processing application. Therefore first the development process is discussed to determine the possibility of using them from high level tools, and then the performance improvement obtained in a real application is analyzed.

2. SYSTEM DESCRIPTION

The test system has been developed with a commercial Workstation Hewlett-Packard model Z600, that hosts two Xeon X5550 QuadCore processors at 2.66Ghz with 4 Gbytes DDR3 RAM, and a NVIDIA TESLA C1060 processor board with 4 Gbytes GDDR3 RAM with 240 streams processors running at 1.3Ghz. The system setup was chosen to maximize the data transfer rate between both processors by using a PCI-Express Gen 2 Bus with 16 data lines, which allows a maximum data transfer rate of 4Gb/s in each direction. The software running in the host processor has been developed with National Instrument's LabVIEW 8.6.1 under Microsoft Windows XP-32 bit operating system. And the software running in the GPU (TESLA board), has been developed using the drivers and the runtime system of NVIDIA's Computed Unified Device Architecture (CUDA) library v2.3, and Microsoft Visual Studio 2008.

3. DEVELOPMENT METHODOLOGY

The main goal of integrating GPUs into the ITMS platform is to evaluate their performance for processing data in real time, and at the same time, to analyze the development process in order to determine the possibility of using them from high level applications. Taking into account the internal architecture of GPUs, the algorithms to be used during the processing must have a high degree of paralelization in order to achieve the first goal. This parallelization degree can be inherent to the algorithm, due to its nature (eg. Fast Fourier Transform); or it can be forced applying several techniques, such as data parallelization.

The algorithm used for bandwidth estimation in KG8B's adaptive sampling rate DAQ system (NOCOFE) was chosen as a good example of a generic signal processing algorithm as it contains both natively (FFT) and non-native (filters, searches) parallel routines.

Regarding the development methodology, LabVIEW was chosen as the development environment, as it the base for the ITMS architecture. The process relies on the use of the LabVIEW GPU Computing Toolkit, which permits to use the CUDA library from LabVIEW. This toolkit consists of a set of functions as shown in Fig.1, named VIs, which permit to make low level calls to the CUDA runtime to use the hardware resources.

It is important to note that calls to the code running in the GPU are called asynchronously by threads running on the host under the LabVIEW environment. Therefore it is possible to parallelize the code running in the GPU, and that running in the host, but synchronization mechanisms must be taken into account in order to develop a consistent application

The work flow for the execution of an application in the GPU from LabVIEW is as follows:

- Create a context in a CUDA-enabled device.
- Determine which memory resources will need the GPU and reserve them from the host.
- Transfer the data from the host memory to the GPU memory.
- Launch the CUDA function to run in the GPU. This must be embedded in a DLL containing calls to the kernel.
- Transfer the results from the GPU memory to the host memory.
- Free memory resources from the GPU and close the context.

The LabVIEW code implementing the abovementioned process is shown in the Fig.2.

All functions calls are performed synchronously, except that to the processing routine in the GPU. The problem is solved by the function moving the data back to the host memory, which waits until the operation in the GPU is completed. In order to obtain the expected performance data must be transferred between the host and the GPU memory at high speeds. In the system tested this is guaranteed by the PCI-Express bus, providing sustained data transfer capabilities at 4GBytes/s

The difficult part of the process comes when porting the algorithm to GPU code as it is not possible to write this code using high level programming languages, such as LabVIEW. The structure of the NOCOFE algorithm for bandwidth estimation is shown in Fig. 3, so a function must be implemented for each step of the algorithm. The code must be written in C language and encapsulated in a DLL that will be called from LabVIEW. When writing the code, it is important to look for the highest degree of parallelization, so each operation must be analyzed in detailed

3.1. POWERSPECTRUM

The first step to compute the power spectrum is the Fast Fourier Transform. Since the FFT has a high inherent degree of parallelization, its implementation has been straight forward as NVIDIA provides an optimized implementation for its CUDA enabled devices though a library named *cuFFT*. The function implements a parallel algorithm and it also allows for data parallelization by passing the input data in a certain way, so the first step is quite straightforward. Then, the single sided power spectrum must be computed according to the definition:

$$Sxx = \frac{F\{x\}*F^*\{x\}}{n^2} \tag{1}$$

where $F\{x\}$ is the FFT of the signal, $F^*\{x\}$ its conjugate, and n number of elements in x. This operation must be repeated for each output value of the FFT as the *cuFFT* library only returns the spectrum components between 0 and π.

To complete this calculation a kernel function that maximizes the use of existing TESLA processor board has been developed. Parallelization has been achieved by developing a kernel function which implements the operation for one sample, and launching this function in parallel as many times as input samples. So there is a kernel function which computes the addition of the squared real part and imaginary parts of each spectral component, divided by the square of the number of existing

components. This function is called from the DLL library in the host processor by launching as many threads as number of FFT output values.

3.2. NORMALIZE-DB SCALE

This task consist of normalizing each sample of the power spectrum calculated above and converting it to decibels (dB). As there is no optimized function available from NVIDIA for this operation, a kernel function has been developed using the same strategy as in the previous task. Fig. 4 shows the kernel function code(GPU) and how this function is called in parallel from the DLL in the host (CPU), as many times as the number of input values to analyze. The expression <<<*Blocks*,*ThreadsPerBlocks*>>> launches as many *Blocks* times as *ThreadsPerBlocks* threads, each of them running the kernel function called. The parameter *ThreadsPerBlocks* is chosen depending on the maximum number of threads that a GPU processor can run (e.g. 512 in this case), and the parameter *Blocks* is chosen depending on the number of input samples. An expression must be used to synchronize all threads before proceeding to the next step. Regarding the implementation of the kernel function, a second degree of optimization is achieved by using the primitive functions included in the NVIDIA library.

3.3. FILTER

The type of filter chosen has a direct impact on the performance of the algorithm, as some implementations lead to poor parallelization, whereas others are just the opposite. It is important to test several alternatives to find the best compromise between functionality and performance. In this case the filter is used to smooth the output of the Power Spectrum function so the search process that comes afterwards produces more consistent results when analyzing similar power spectrums. Therefore a FIR filter with 30 taps has been chosen. FIR filters have a high degree of parallelization since their output values only depend on the values of the input and the coefficients. This is not the case for IIR filters, where it is necessary to accumulate several previous input and output values in order to compute each output value. Again there is no optimized function in the NVIDIA libraries to perform this operation so it has been implemented in C following the same strategy as before. Parallelization has been achieved by developing a kernel function which implements the filter equation, and launching one thread per output sample to call that function in parallel as many times as needed.

3.4. SEARCH

In order to compute the bandwidth of the signal, the point where the filtered power spectrum meets the noise level, corner frequency must be computed. This is done in two steps. First, the noise level is obtained as the mean plus the standard deviation of the last 20% of the power spectrum. Both operations have been implemented as kernel functions, and parallelization as been achieved by dividing the input data block in several sub-blocks in order to run them in parallel.

Then, the corner frequency is obtained by searching in the filtered power spectrum array starting at its maximum value. Searching is a linear operation, so it is not possible to parallelize it directly. Therefore, the only possibility is data parallelization (e.g. running one search per acquired input channel).

3.5. DECIMATE

The last step is to decimate the input signals depending on the maximum bandwidth value of the acquired channels. This function has been fully parallelized by following the same approach as in the previous examples.

4. EXPERIMENTAL RESULTS

In order to evaluate the performance of the GPU, execution times must be measured. There are two possibilities: to use the signal clock of the CPU, which has a resolution of milliseconds; or to use the timers in the GPU, which have a resolution of half a microsecond. In the first case the time measurements are taken from LabVIEW, in the second they are taken from the DLL by using calls to primitive functions from the CUDA library. The later has been chosen as it has a much better resolution.

First, the algorithm execution time has been measured for different input block sizes. The execution time of each step of the algorithm has been determined to locate the weakest points of the process. Table I shows how this results, in milliseconds, for each step of the algorithm depending on the data block size used to acquire each channel. The *Search* process is computationally more expensive because it has the lowest parallelization level.

It is also clear from the results that when the user must parallelized an algorithm it leads to poorer performance than when there is an optimized library available, as the performance of all tasks that have been parallelize have a higher dependency on the data block size than that of the FFT. This is shown for clarity in Table II, where the increase of execution time with respect to the increase in the data block size is displayed for each algorithm step. As the data block size doubles from one row to the other an increase of 100% would be expected from one row to the other. Therefore the difference between the displayed increased execution time and the expected one is a measure of the optimization degree of the code. Anyhow, one must also consider that the FFT is the task with the highest native parallelization degree, so it not fare to do a straight comparison with the rest of the functions. Further efforts could be made to improve the parallelization of the other tasks, such as playing with the memory allocation of the data in the GPU among the different types of memories defined in the CUDA Memory Model, but this was not the objective of this work.

In order to analyze the performance improvement obtained by using GPUs a version of the same algorithm was also implemented in LabVIEW and run in the host processor. Table III shows the system improvement obtained depending on the data block size of each acquired channel. It must be considered that the execution time of the GPU includes the data transfer between the host and the CUDA-enabled device memory as data acquisition control is still reserved to the host processor. Therefore the possible benefits of the GPU rely in a sense on the availability of high speed data buses.

The processing time of a data block limits the maximum sample rate that can be used to acquired that data in real time. Table IV shows the improvement obtained by using the GPUs in terms of the maximum sample rate that can be achieved. This a crucial parameter of the system, as it limits the spectral bandwidth of the system, and therefore the sources of information that can be analyzed.

It must be reminded that the data block size, together with the sample rate, determine the available processing time, and the time resolution of the system. Therefore, small block sizes are interesting from the point of view of the time resolution; while bigger ones might provide better performance as it the case in the GPU.

It has been demonstrated that developing efficient code for GPUs requires a high degree of knowledge of the processor architecture and some degree of expertise in algorithm parallelization. Software development tools, e.g. compilers and high level tools, are evolving much slower than microelectronics, creating a gap that complicates the widespread use of these technologies. Anyhow, it is also true that the massive processing capabilities of GPUs, counteract against this problem by compensating the lack of optimization with more processing power. Therefore, it has been proved that it is possible to integrate GPUs in DAQ applications from high level programming languages, although it is not a straightforward process. Another question to solve is the availability of GPU based solutions for industrial environments, e.g. PXI based boards, or the interconnection between existing 1U systems and these systems.

ACKNOWLEDGEMENTS

This work, supported by the European Communities under the contract of Association between EURATOM/CIEMAT, was carried out within the framework of the European Fusion Development Agreement. Financial support was also received from the Spanish Ministry of Science and Innovation under the research project ENE2009-10280. The views and opinions expressed herein do not necessarily reflect those of the European Commission.

REFERENCES

[1] J.Vega, A. et al., “New Developments at JET in Diagnostics, Real Time Control, Data Acquisition and Information Retrieval on the route to ITER”. *Fusion Engineering and Design*, vol **84**, issue 12., p.p. 2136-2144. December 2009.

[2] Barrera E. et al.,

[3] G. de Arcas, et al “Self-adaptive sampling rate data acquisition in JET’s correlation reflectometer”.*Review of Scientific Instrum*ents, vol.**79**, issue 10, p.p. 10F336-10F336-4. 2008

[4] J. González et al. “Configuration and supervision of advanced distributed data acquisition and processing systems for long pulse experiments using JINI technology”. *Fusion Engineering and Design,* vol **84**, issue 2-6., p.p. 832-836. June 2009

Data block size/ch	Power Spectrum	Normalize dB scale	Filter	Search	Decimate
16sS	0.936	0.615	0.110	1.988	0.31
32kS	0.945	0.677	0.176	3.438	0.62
64kS	0.958	0.776	0.320	6.227	1.20
128kS	1.196	0.974	0.620	11.715	2.39

Table I. Execution time in ms for each task

Data block size/ch	Power Spectrum	Normalize dB scale	Filter	Search	Decimate
16kS	–	–	–	–	–
32kS	1%	10%	60%	73%	99%
64kS	1%	15%	82%	81%	94%
128kS	25%	26%	94%	88%	99%

Table II. Increased execution time depending on the data block size

Data block size/ch	LabVIEW exec. time(ms)	GPU Exec. time (ms)	Performance improvement
16kS	6	4	33.3%
32kS	14	6	57.1%
64kS	26	10	61.5%
128kS	51	18	64.7%

Table III. Compared performance of the GPU and the host

Data block size/ch	LabVIEW	Data transfer + processing	Performance improvement
16kS	2.73	4.09	50.0%
32kS	2.34	5.46	133.3%
64kS	2.52	6.55	160.0%
128kS	2.57	7.28	183.3%

Table IV. Maximum sample rates (MS/s)

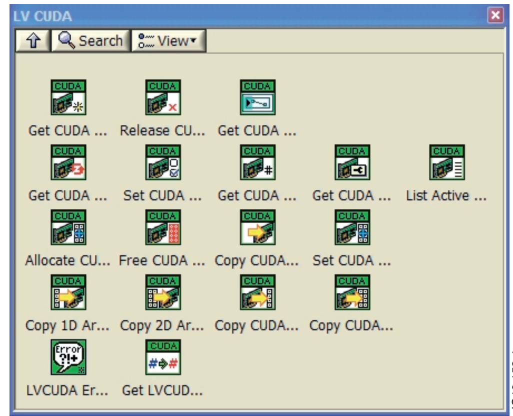

Figure 1: LabVIEW GPU Computing Toolkit Palette.

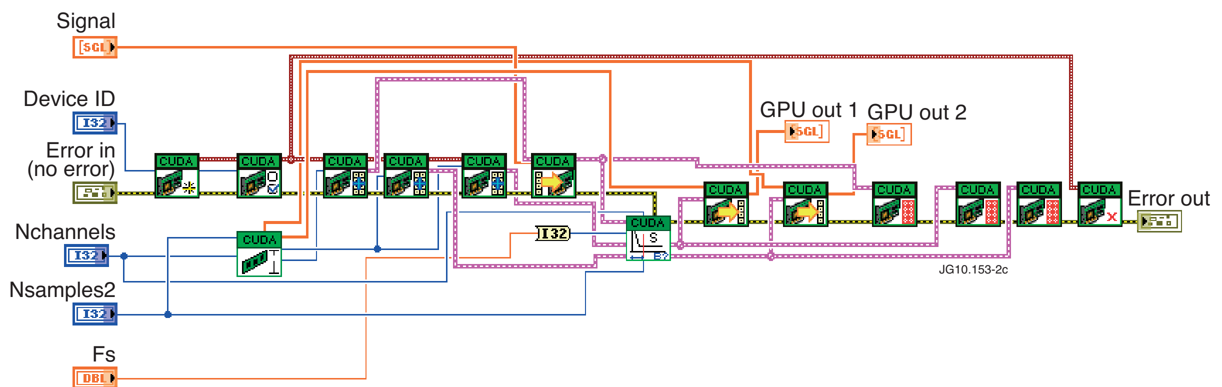

Figure 2: LabVIEW G-code.

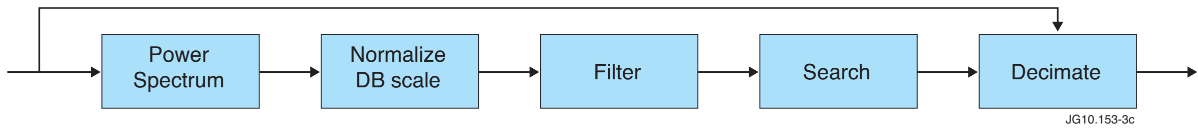

Figure 3: Bandwidth estimation algorithm steps.

```
INLINE DEVICE void normalizespectrum_D(float *pIN, float *maximum)
{
        int q;
        int block=0;
        float temp1,temp2;

        q = blockIdx.x*blockDim.x+threadIdx.x;

        block = q/channelsize;
        temp2  = pIN[q]/maximum[block];
        temp1  = log10f(temp2);
        temp1  = temp1*10.0;
        pIN[q] = temp1;
}
```

```
MaximumValue_phase1_H<<<Blocks_a,ThreadsPerBlocks_a>>>(Results,maxtemp);
cudaThreadSynchronize();

MaximumValue_phase2_H<<<Blocks_b,ThreadsPerBlocks_b>>>(maximum,index);
cudaThreadSynchronize();

normalizespectrum_H<<<Blocks,ThreadsPerBlocks>>>(Results,maximum);
cudaThreadSynchronize();

FirFilter_H<<<Blocks_f,ThreadsPerBlocks_f>>>(Results, Data_Out);
cudaThreadSynchronize();

Mean_H<<<Blocks_m,ThreadsPerBlocks_m>>>(Data_Out, means);
cudaThreadSynchronize();

Desviation_H<<<Blocks_d,ThreadsPerBlocks_d>>>(Data_Out,Results);
cudaThreadSynchronize();
```

JG10.153-4c

Figure 4: C code runs into the GPU and the CPU.