Assessment of General Purpose GPU systems in real-time control

Tautvydas J. Maceina, Gabriele Manduchi

Abstract—**The recent advance of GPU technology is offering great prospects in computation. Originally developed for graphical applications, general purpose GPUs (GPGPU) have been extensively used for massively parallel computation. The penetration of the GPU technology in real-time control has been somewhat limited due to two main reasons:**

1) **Control algorithms for real-time applications involving highly parallel computation are not very common in practical applications**
2) **The excellent performance in computation of GPUs is paid for a penalty in memory transfer. As a consequence, GPU applications for real-time controls suffer from an often unacceptable latency.**

There are in any case some real-time applications in fusion research that may take benefit from the usage of GPGPUs such as state space-based control for a very large number of states. The excellent performance of GPUs in term of throughput of computation is however counterbalanced by a poor performance in memory transfer, leading to an increase in latency in the typical cycle in real-time control involving data sample acquisition; elaboration; transfer of the resulting data to actuators. A precise assessment of latency vs throughput represents therefore a very useful information when designing real-time control systems for potentially parallel applications, especially when facing the option for GPUs or multi-threaded CPU applications. We designed a code (for GPU & CPU) to test latency and jitter. Operations that we used as a test load were dense matrix-vector multiplications and memory transfer in order to mimic a large state space based control algorithm. We compared obtained results to see where GPU computation excels and where it falls behind in order to give useful hints to designers facing the option of using either a multi-threaded, multicore CPU application or a GPGPU.

I. Introduction

GPGPU has firmly earned its reputation in HPC as hardware for massively parallel computation. Yet GPU application in Real-Time problem solving is just starting to develop. GPGPU is steadily pushing its way into fusion scientific community as a new tool for Real-Time applications. We as members of fusion community would like to give a comprehensive study on applicability and prospects of GPU. Particularly we hope we will succeed to indicate points, when it is advantageous migrating your code to GPU and when CPU is just sufficient.

II. GPU architecture

The CPU directs processing tasks to the GPU via a stream through a hardware graphics pipeline. A GPU device has streaming multiprocessors (SM) each of which contains a fixed set of processing cores. This streaming architecture executes single instruction multiple data (SIMD) in parallel where each SM is computationally independent from any other SM, making it ideal for problems requiring large data sets processing. The Compute Unified Device Architecture (CUDA) provides the API to submit tasks to and receive results from the graphics processor. The computations are performed by calling a method from the CPU that hosts the GPU device known as a kernel function. Processing threads are created and grouped together in blocks. The number of threads and blocks are parameters of the kernel function. A block is executed by the GPU scheduler using a set of 32 parallel threads (known as a warp). While executing, each block and thread maintain unique indices `blockIdx` and `threadIdx`.

III. GPU memory

GPU has 3 types of physical memory: register memory, shared memory and global memory. Register memory (256 kB) and shared memory (48 kB) both reside on chip, thus are the fastest in access. Global memory is the bank usually of several gigabytes, which resides off chip and is the slowest in access. Each of them has a special purpose in the GPU. Register memory (also called private or local) is the memory available to each thread and cannot be accessed by any other thread. It stores variables, which are specific to that thread, such as thread ID, block ID and any other private variables significant for the thread. Shared memory is designed to store data that can be shared among threads belonging to the same block. It enables thread communication and is extremely useful in calculations based on mathematical reduction, such as matrix operations. Global memory is accessible by all threads and host. It is used to copy data from/to host and usually stores the user data of input and output of the application. The fact, that global memory and shared memory have difference in speed of access by several orders, many levels of optimization can be achieved by careful tuning of parameters such as amount of shared memory, number of threads/block and number of streams. For example maximizing accesses of shared memory over the global memory leads to significant performance gains, since access to the shared memory is about 100 times faster than the global memory. Our tests were mainly comprised of matrix-vector multiplication $A_{m\times n}\cdot\vec{x}_{n\times 1}=\vec{b}_{m\times 1}$ kernel (1), where we are able to change the size of the problem and explore different modes of operation by choosing various types of global memory allocation and having shared memory as an

Tautvydas J. Maceina is with the Università degli Studi di Padova, Padova, Italy, (e-mail: tautvydas.maceina@igi.cnr.it).

Gabriele Manduchi is with the Consorzio RFX, Padova, Italy, (e-mail: gabriele.manduchi@igi.cnr.it).

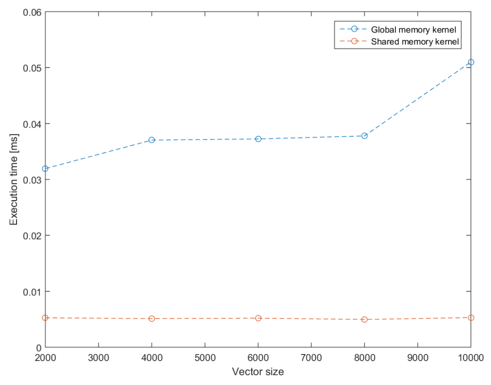

Fig. 1: Matrix-vector multiplication performance: Shared memory vs Global memory

option.

$$\begin{bmatrix} A_{1,1} & \cdots & A_{1,n} \\ \vdots & \ddots & \vdots \\ A_{m,1} & \cdots & A_{m,n} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_m \end{bmatrix}, \tag{1}$$

where vector $\vec{x}$ is accessed multiple times during operation, which signifies it can be pre-loaded from Global memory to much faster Shared memory and therefore execution time is reduced. Our first test was conducted in order to estimate performance gains, when Shared memory is involved into computation (Fig. 1). Another optimization can be achieved

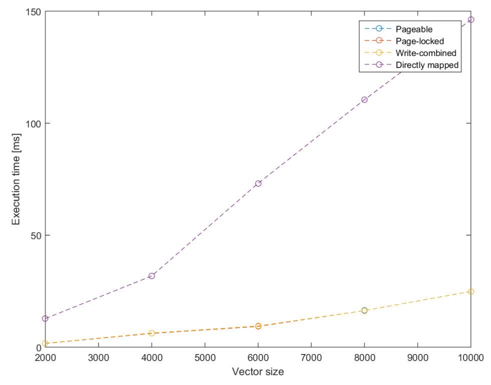

Fig. 2: Matrix-vector multiplication performance using various types of memory allocation

by balancing the amount of overhead spent on transferring information back and forth between the GPU global memory and CPU RAM. By varying the type of GPU global memory allocation (pageable, page-locked, write-combined, directly mapped) we compared the performances of the same matrix-vector multiplication kernel (Fig. 2).

Pageable allocation is the default type of allocation and the most commonly used. It is the type of allocation that is freely managed by the system, i.e. it can be moved and reallocated by internal system processes, which are transparent to the user. Page-locked allocation does not allow the system to page the memory, also called "pinned" allocation, i.e. it is fixed to a physical memory address that does not change. In most systems page-locked allocation results in speed up of applications, since system does not perform paging on that memory and data can copied faster via DMA circuit without involvement of CPU. Write-combined allocation is special type of allocation, which is a good option for one-way transfer from CPU to GPU. Directly mapped allocation is a type of allocation, which allows to tie CPU RAM and GPU RAM directly. In this way the programmer does not have to manually code the memory transfer from host to GPU anymore. System will manage copying automatically, i.e. whatever gets written in CPU RAM will automatically appear on GPU RAM. In our test the directly mapped allocation performed the worst, while other allocations perform much better and identically between each other. We think that directly mapped allocation results in multiple calls of internal copy operations of small size, which builds up the latency. The other three allocations can only perform in such similar manner when the memory paging is not present in the system. We believe it is the case.

IV. MARTe framework and GPU

The Multi-threaded Application Real-Time executor (MARTe) is a C++ framework that provides a development environment for the design and deployment of real-time applications, e.g. control systems. The kernel of MARTe comprises a set of data-driven independent blocks, connected using a shared bus. A MARTe application is designed by configuring and connecting a series of blocks named Generic Application Modules (GAM). MARTe and GPU have a compatibility issue. Firstly MARTe does not yet provide interface to access GPU resources and secondly CUDA code can only be compiled with a proprietary compiler `nvcc` from Nvidia. However these issues can be overcome by external compilation. We managed to wrap CUDA commands into C++ functions and compile it externally with `nvcc` to an object. Then the object was linked to a MARTe GAM during compilation of the GAM. In such a way GAM is able to call wrapper functions, which contain CUDA code that was compiled externally by `nvcc`.

V. Hardware

Nvidia Tesla K40 was used for running performance tests (Fig. 1, 2, 5, 6, 7). Tesla K40 is one of the latest Nvidia products for HPC built on Kepler architecture and released in 2014. It runs on 745 MHz clock with 12 GB of RAM and 48 kB of shared memory.

Nvidia GeForce GTX 480 was used for real-time tomography application (Fig. 3). GeForce GTX 480 is a graphics card originally designed for gaming purposes built on Fermi architecture and released in 2010. It runs 700 MHz clock with 1.5 GB of RAM and 48 kB of shared memory.

VI. Real-Time GPU applications

There are a number of real-time domains where GPUs may be applied. GPU can efficiently carry out digital signal processing operations and matrix operations of large size. These operations, coupled with other GPU-efficient algorithms, can

be used in medical imaging, video processing and data processing.
As members of fusion community hereby we present a real-time application being developed for plasma control purposes on the basis of GPU computation. The application runs an algorithm of plasma tomography on ISTTOK tokamak (Fig. 3). Signals are produced by 3 detectors with linear arrays of 16 sensors each sensitive to the intensity of plasma emission. Each cycle a signal of 48 channels (3x16) arrives to the data acquisition system.
The signal $\vec{f}$ is received by MARTe framework and immediately copied to the GPU, where multiplied with so-called pseudo-inverse contribution matrix C^+ produces tomographic coefficient vector $\vec{a}$.

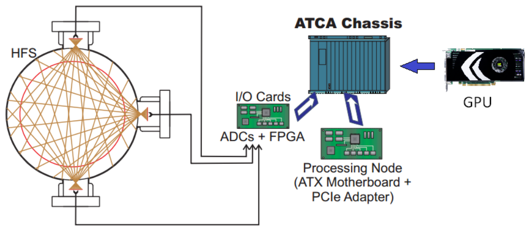

Fig. 3: Tomography data acquisition ATCA system installed in ISTTOK

$$\vec{a} = C^+ \cdot \vec{f} \tag{2}$$

Then the so-called *synthetic* signals are calculated by the original un-inversed contribution matrix C.

$$\vec{\tilde{f}} = C \cdot a \tag{3}$$

Our goal is to choose the best-fitting subset of coefficient vector $\vec{a}$. It is performed by comparing signal data $\vec{f}$ with all sets of the synthetic signals $\vec{\tilde{f}}$ in term of statistical error χ^2.

$$\chi^2 = \sum_i^S \left(\frac{f_i - \tilde{f}_i}{\sigma} \right)^2, \tag{4}$$

where S is the number of sensors in the system (dimension of signal vector $\vec{f}$) and σ is the standart deviation of the signal, which relates to noise level present in the system. According to minimum of χ^2 a corresponding subset of $\vec{a}$ is selected for the reconstruction. The selected subset of $\vec{a}$ is then multiplied with so-called basis matrix B, which results in a reconstruction image $\vec{g}$ (5). Image $\vec{g}$ then is copied back to the host.

$$\vec{g} = B \cdot \vec{a}^*, \tag{5}$$

$\vec{a}^*$ is a subset of $\vec{a}$. Contribution matrix C, its pseudo-inverse C^+ and basis matrix B are pre-loaded to GPU before the real-time phase and do not need updating during execution of the algorithm. This algorithm contains three kernels based on matrix-vector multiplication and two data copy operations (one for arriving $\vec{f}$ and one for exporting $\vec{g}$).
We would like to demonstrate that such applications have an inherent memory bottleneck due to data transfer between CPU and GPU in real time. We separately measured the durations

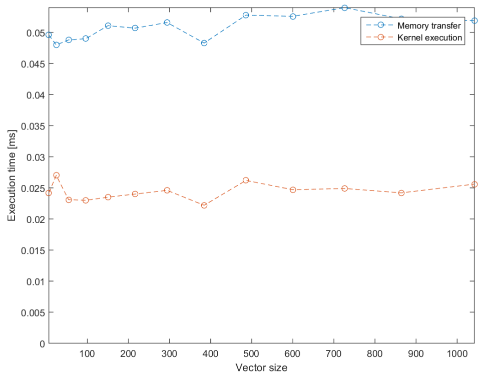

Fig. 4: Demonstration of inherent bottleneck in real-time reconstruction algorithm, where 3 kernels perform faster than 2 memory transfers

of kernels and memory transfer (Fig. 4). Our application demonstrates a common nature of GPU applications based on matrix-vector multiplication, where processing is faster than data transfer. We would like to extend this knowledge to more general class of algorithms called *state-space based control algorithms*.

VII. State-space based control algorithms

State-space based control algorithms are the most used type of control algorithms in fusion. They usually are described by a generic matrix and an input vector. The product of them gives the feedback (control actuation) to the system. We would like to discuss the general applicability of these algorithms to CPU and GPU. As already discussed GPU based real-time applications usually have an inherent memory bottleneck. Anyway GPU kernels alone offer great processing speeds because of their massively parallel nature. On the other hand CPU virtually has no memory bottleneck, yet a single CPU is not capable to process big data fast enough. As real-time application developers we would like to explore, how multi-threaded CPU would perform under same load versus GPU code. Here we present a comparison between single-threaded CPU, multi-threaded CPU and GPU performances (Fig. 5). Obviously single-threaded CPU falls far behind, but its sole purpose is to give reference. We can observe an interesting feature of GPU and of multi-threaded CPU performances. They are practically similar along the whole range of input vector dimension. This fact implies that matrix-vector multiplication is a border line application, where CPUs and GPUs are equally computationally capable. GPU would still be a favorable choice for an extremely high degree of computation, because the number of threads that GPU can have is virtually unlimited, while CPUs have limited amount of threads by design. However real-time applications may not require such a high throughput, therefore multithreaded CPU can be sufficient in terms of computation and absence of memory bottleneck. An as well a developer would escape the need to learn GPU programming (which is quite different from CPU programming).

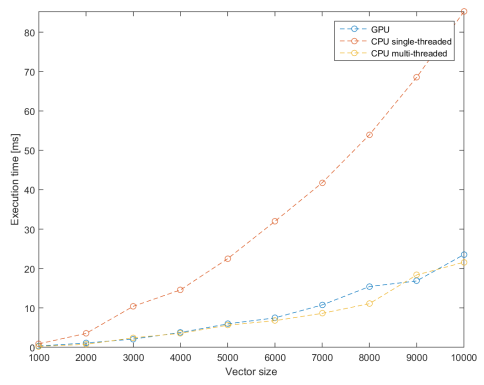

Fig. 5: Comparison of performance of GPU and multithreaded CPU

VIII. Jitter

Jitter is another very important parameter describing performance of a real-time system. Jitter is the variation of latency. A short latency does not yet guarantee reliable performance of the system, unless jitter is contained in certain limits. As an example we demonstrate jitter of a test kernel running matrix-vector multiplication with shared GPU memory versus global GPU memory <u>without</u> memory transfer (Fig. 6). Not only

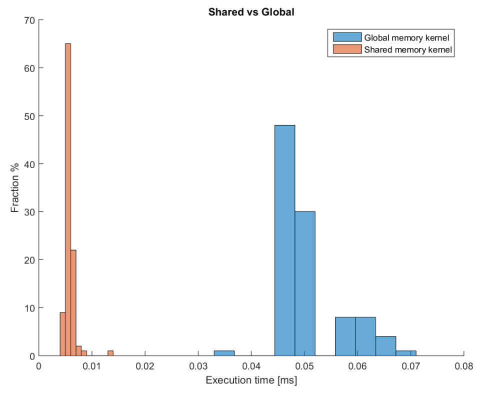

Fig. 6: Jitter of matrix-vector multiplication using Shared memory and Global memory <u>without</u> memory transfer

usage of shared memory results in faster execution (Fig. 1), but it also reduces jitter of the overall performance (Fig. 6). However if we include memory transfer in the measurements, we observe much more variance in the jitter and much slower executions overall (Fig. 7).

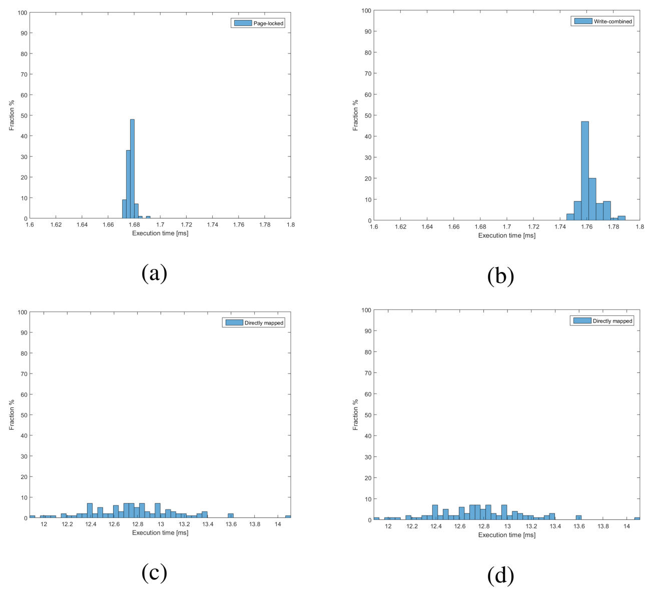

Fig. 7: Jitter of matrix-vector multiplication kernel using various types of Global memory allocation <u>with</u> memory transfer included: a) Pageable b) Page-locked c) Write-combined d) Directly mapped

IX. Conclusion

We hope we provided some valuable insights and discussions regarding nature of GPU applications in real-time. The results of this work can be concluded to:

- CPU and GPU can be equally exploitable for a matrix-vector multiplication based algorithms, unless required throughput is extremely high. Then GPU is more favourable option.
- We presented probably the first real-time GPU application in fusion community.
- Advantages of GPU computing in real-time are only available for a trade-off of a memory bottleneck.
- We presented an important insight into applicability of state-space based algorithms to CPU and GPU, which can be useful for developer when choosing between CPU and GPU as the computation platform.