17th International Conference
ENGINEERING MECHANICS 2011
Svratka, Czech Republic, 9 – 12 May 2011

GPU COMPUTATION IN ENGINEERING PRACTICE

C. Fischer[*]

Abstract: *In many research fields the numerical problems demand extremely large computational power. As a consequence, researchers are constantly demanding new and more powerful computers. Recently, the graphics chip producer NVIDIA launched the CUDA parallel computing architecture, which enables scientists to utilize the extreme power available on modern and relatively cheap Graphical Processor Units (GPUs). Necessity to change the common approach together with significant difference in the GPU architecture prevents the wide usage of the graphics hardware. A flexible programming interface to GPU has been recently implemented in numerical packages like Mathematica™ or Matlab®. Such an interface does not hide the differences completely, but greatly simplifies the development of the optimized code. The contribution presents simple examples of GPU utilization in a real engineering environment.*

Keywords: ***Numerical computation, GPU, CUDA.***

1. Introduction

It is widely respected fact, that the progress in computer hardware development is driven mainly by entertainment industry. Unceasing demand of better and more realistic games and wide spread of the personal computers makes the powerful hardware cheaper and cheaper. It is no doubt, that developing of the specialized computational equipment only for the scientific purposes would be very expensive. It has already appeared in the past, that the scientist have adopted devices developed for home entertainment and used them in their serious work, like powerful clusters based on cheap XBOXes (Microsoft) or PS3 (Sony) game consoles (see [PS3]). Recently, similar effect came to light with GPU computation. However, there is one significant difference between approaches of companies producing game consoles and GPU producers. While Microsoft and Sony tried hard to disable any other usage of their products than pure entertainment, ATI (AMD) and NVIDIA support the great success of the general purpose computing on their graphics hardware.

Is GPU computing really profitable or is it only hype? As everything in the real world, it depends on the circumstances. The modern GPU serve as a powerful graphics engine thanks to its highly parallel programmable processor. As a parallel device it features peak arithmetic and memory bandwidth that substantially outpaces its CPU counterpart. Contemporary graphics processors thus operate as co-processors within the host computer. This means that each GPU is considered to have its own memory and processing elements that are separate from the host computer. In contrast to contemporary CPUs, which have 1-8 fast computational cores, the recent GPUs are equipped by several hundreds of simple yet powerful cores. From that fact one can deduce the typical field of application of the GPUs. The appropriate task should be able to exploit a high number of available cores. Typical examples are various simulation procedures (chemistry, electro, biology, etc.), CFD, Monte Carlo methods and others.

The acronym CUDA (see [CUDA]) stands for Compute Unified Device Architecture. Being developed, maintained and distributed by NVIDIA, it offers unified programming approach to all recent NVIDIA GPUs. Access to the ATI's GPUs is provided using the OpenCL standard (see [OpenCL]). Open Computing Language (OpenCL) is a framework for writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs, and other processors. CUDA, as well as OpenCL use C language as the main programming tool. Although OpenCL standard can be used for NVIDIA devices too, we will restrict ourselves to the CUDA programming interface in this paper.

[*] RNDr. Cyril Fischer, PhD.: Institute of Theoretical and Applied Mechanics AS CR, v.v.i., Prosecká 76, Prague 9, Czech Republic, e-mail FischerC@itam.cas.cz

Tab. 1: Selected representatives of the NVIDIA graphics cards.

Graphics Card Product	Cores	Memory	Price
Quadro 6000 (Fermi)	448	6.0GB ECC, 384-bit, 144GB/s	3 200 €
Quadro 5000 (Fermi)	352	2.5GB ECC, 320-bit, 120GB/s	1 800 €
Quadro 4000 (Fermi)	256	2.0GB, 256-bit, 89.6GB/s	800 €
Quadro 2000 (Fermi)	192	1.0GB, 128-bit, 41.6GB/s	500 €
Quadro 600 (Fermi)	96	1.0GB, 128-bit, 25.6GB/s	180 €
GeForce GT 430	96	1.0GB, 128-bit, 25.6GB/s	70 €
GeForce GT 330M (mobile)	48	1.0GB, 128-bit, ??GB/s	

2. Main differences

Although the presented contribution attempts to show that the usage of graphics hardware can be straightforward, it is necessary to point out the main differences between CPU and GPU programming. These differences could be named as “dedicated and complicated memory”, “compute capabilities”, “parallel algorithms”.

- Dedicated memory. The graphics cards are usually equipped with a certain amount of a very fast graphics memory, connected to the main memory of the computer via relatively slow bus (PCIe). All data have to pass this bottleneck.
- Complicated memory. Memory architecture of the graphics devices is rather complicated. However, the high level interface unifies (up to certain limit) memory access. For details see e.g. [CUDA2].
- Compute capabilities and precision. Although CUDA recognizes several categories of the GPUs, the end-user interested in scientific computation distinguishes only two categories: Prior version 1.3, which does not use double precision floating point numbers, and version 1.3 and newer, which is able to perform double precision arithmetic. From this point of view, situation is quite similar to the early computational era, when the cost of double precision operations was twice the single precision. Performance of the recent GPUs, which are capable double precision operations, suffers similar handicap. In contradistinction to the Intel based FPU, which uses internally 80bit precision and several types of rounding, the double precision computations of NVIDIA GPUs use 64bit even for the intermediate results. There are some additional deviations from the IEEE 754 standard, see [CUDA] and (Goldberg, 1991). The trigonometric functions implemented in the hardware are not very accurate, (with error up to several “ULP, unit in the last place”), but CUDA programming toolkit provides less powerful but accurate math library.
- Parallel algorithms. Graphics processors are built as SMID (single instruction, multiple data) devices. This means, that the code (instruction) is always preformed on a predefined set of data. The minimum size of the data processed in SIMD fashion by a CUDA multiprocessor is a group of 32 threads (so called warp). However, instead of manipulating warps directly, programmers work with blocks that can contain 64 to 1024 threads.

3. Programming examples

As we are working with graphics cards, let us start with a famous graphics example, Mandelbrot set. It is defined by the relation:

$\mathbb{M} = \{c \in \mathbb{C}: \exists s \in \mathbb{R}, \forall n \in \mathbb{N}, |P_c^n(0)| \leq s\}$ where $P_c: z \longmapsto z^2 + c$.

The sample code is listed in Fig. 1. Arguments of the functions are as follows: Scalar return vector ANS, dimension of the desired image, complex constant `c`, scale of the image `r` and maximal allowed number of iterations. The code looks out tangled,

```
__global__ void mandelset(int * ANS, int n,
Real_t c0RE,Real_t c0IM,Real_t r,int lim) {
int index=threadIdx.x+blockIdx.x*blockDim.x;
if (index<n*n){
 Real_t dx = r/(n - 1), dy = r/(n - 1);
 Real_t zRE, zIM, cRE, cIM, tmp;
 int i=(int)(index/n), j=(index%n), k=0;
 zRE = 0; cRE = c0RE + (2*i - 1 - n)*dx;
 zIM = 0; cIM = c0IM + (2*j - 1 - n)*dy;
 while (hypot (zRE, zIM) < 2 && k++ < lim){
   tmp = 2*zRE*zIM + cIM;
   zRE = zRE*zRE - zIM*zIM + cRE;
   zIM = tmp;}
 ANS[index]=k;
 }
```

Fig. 1: C code for generating Mandelbrot Set.

but it is only due to lack of complex arithmetic in C language. Having the code ready (e.g. in the form of text string `src`), a Mathematica function can be prepared using a simple command:

```
Needs["CUDALink`"];
CudaMandel = CUDAFunctionLoad[src, "mandelset", {{_Integer, _, "InputOutput"},     (1)
                             _Integer, _Real, _Real, _Real, _Integer}, blockdim];
```

Resulting function can be used in the same manner like any other standard or user defined function.

```
n = 999; zzz = ConstantArray[1, {n*n}];
ans = CudaMandel[zzz, n, .37, .1, 0.005, 255];
```

There are several constructions in the example, which is worth to emphasize. First, type specification `Real_T` stands for a general floating point type. Compiler substitutes single or double precision type according to available hardware. Second, let us look at the 3rd line of code in Fig. 1:

```
int index=threadIdx.x+blockIdx.x*blockDim.x;
```

Variables `threadIdx`, `blockIdx`, and `blockDim` are part of the standard CUDA toolkit, see [CUDA3]. They identify individual threads and help to access the corresponding data for a particular part of computation. Third, see the line 4 in the code. As the program is executed in wraps, each thread has to check if it operates on the valid data. The last remark belongs so the parameter `blockdim` in (1). This value depends not only on the particular hardware, but also on memory demands of the code. It is bounded by hardware parameters: maximum block dimension, maximum threads per block and maximum number of registers per block. Typical values of the mentioned parameters can be 512, 512 and 8096 respectively. Especially number of registers can be limiting for a complicated code.

As a second example, let try us check precision of the build-in trigonometric functions. To show the basic access to the memory on the graphics card, the procedure will be following:

1. Allocate input and output arrays on the graphics card, lines 15 – 17, Fig. 2.
2. Fill the input array ε*□ for integers i: $0 \leq i < N$ (function fill, lines 5 – 9), ε is ULP or the machine epsilon – line 18. It is computed using the local procedure (resident on GPU), lines 1 – 4.
3. For each value x from the input array call the built-in instruction __sinf(x), line 19, function is defined 20 – 14.
4. Read and check the result, lines 21 – 24.

Due size limitation of a single block of GPU memory, the interval $(0,2\pi)$ has to be divided to several parts of length N and computation has to repeat. It appears, that the greatest error is about 10ε for arguments close to 2π.

```
src=″
1 __device__ float gpuEps(void) {
2 float x=2.0; int k=0,lim=10000;
3   while ((1.0+x)>1.0 && k++<(lim))x=x/2.0;
4 return x;}
5 __global__ void fill(Real_t * A, int length) {
6 int index=threadIdx.x + blockIdx.x*blockDim.x;
7   if (index < length) A[index]=index*gpuEps();
9 }″;
filleps = CUDAFunctionLoad[src, "fill",
                        {{_Real}, _Integer}, 512];
src = "
10 __global__ void ts(float *A, float *B, int N){
11 int index=threadIdx.x + blockIdx.x*blockDim.x;
13   if (index < N) A[index]=__sinf(B[index]);
14 }";
testsin = CUDAFunctionLoad[src, "ts",
               {{_Real}, {_Real}, _Integer}, 512];
15 n = 10000000;
16 mem1 = CUDAMemoryAllocate[Real, {n}];
17 mem2 = CUDAMemoryAllocate[Real, {n}];
18 filleps[mem1, n];
19 testsin[mem2, mem1, n];
20 var1= CUDAMemoryGet[mem1];
21 var2= CUDAMemoryGet[mem2];
22 GPUEPS = var1[[2]];
23 Maximum=Max[Abs[var2 – Sin[var1]]/GPUEPS
```

Fig. 2: Program for checking the accuracy of __sinf.

As the last example of usage GPU computing, let us try to apply the described approach to the analysis of experimental data, obtained in UTAM in 2010. The resonance properties of the pendulum have been measured. Pendulum was supported by Cardan joint and excited in one direction only by a shaker, see Pospíšil (2010). Movement of the pendulum was recorded in the both directions by a pair of identical high-speed rotary magnetic encoders. They have a frictionless design and provide 8192 counts per revolution. The natural frequency of the pendulum was about 0.6 Hz, sampling frequency of the data acquisition setup was 200 samples per second. Length of the records varied, starting from 6 minutes up to several hours. It has appeared that the instantaneous frequency of the response was noticeably dependent on the actual amplitude, which has varied quite quickly in the resonance region. Moreover, as the resonance frequency interval for this setup was rather narrow, less than one Hertz, the data analysis had to be performed with exceptional care. To obtain accurate values of the instantaneous frequency and amplitude, the response had to be divided into individual periods (response was almost harmonic). Each section of data was then approximated by a trigonometric function, whose parameters provided most accurate data.

Performing this computation using just simple Mathematica’s function FindFit took for a single data almost an hour. On the other hand, fitting a sinus function through almost harmonic data, especially when the very good initial approximation from FFT analysis is available is a task, which can be easily programmed and then parallelized massively on the GPU.

```
src=″ ...
1 __global__ void fpars(Real_t *ANS, int nans,
2 Real_t *data, Real_t *t, int ndata) {
3 int index=threadIdx.x + blockIdx.x*blockDim.x;
4 int n=3;
5 Real_t tolx=1e-5,tolf=1e-5;
6 if (index<nans)
7   newt(&(ANS[index*n]),n,tolx,tolf,
        &(data[index*ndata]),
        &(t[index*ndata]),ndata);
}“;

Needs["CUDALink`"]
CudaNewt=CUDAFunctionLoad[src,
 "fpars",{{_Real,_,"InputOutput"},_Integer,
       {_Real, _,"Input"},{_Real,_,"Input"},
       _Integer}, 128];
```

Fig. 3: Driver for the procedure `newt`.

Data fitting consists in minimization of a function $Z(a,b,c) = \sum_{i=1}^{n}(data_i - a\sin(b\,t_i + c))^2$. If the gradient vector and Hessian matrix of this function are available, the standard Newton method can be used to find a zero point of the grad Z. Implementation of the Newton method was taken from Numerical Recipes (Press et al., 1992). The main part of the CUDA program is shown in the listing in Fig. 3. To understand it, let us denote `nans` number of analyzed periods and `ndata` number of samples used in each period of data. Vector `ANS` has dimension (3 `nans`) — 3 parameters for each period. Vectors `t` and `data` have dimension (`ndata*nans`). Initial approximation is stored in triples in `ANS` on input; resulting values are in the same places of `ANS` on output. From the code in Fig. 3 let us emphasize the line 7, where the input/output data vectors are distributed according to thread index. To each instance of `newt` procedure, the addresses of the appropriate parts of the arrays are passed.

4. Conclusions

It is always difficult to reasonably balance the time, which is necessary for certain task, between long computation and tedious programming. Great success of high level programming packages like Matlab and Mathematica is based on ease of use, which is redeemed by reduced computation speed. We have shown, that at little extra effort certain numerical problems can be speeded up noticeably.

Acknowledgement

The support of the Grant Agency of the AS CR No. IAA200710805 and AV0Z20710524 research plan are gratefully acknowledged.

References

Goldberg, D. (1991) What every scientist should know about floating-point arithmetic, Journal ACM Computing Surveys (CSUR), Volume 23 Issue 1.

Pospíšil S., Fischer C., Náprstek J. (2010) Experimental and theoretical stability analysis of damped auto-parametric pendulum, In Engineering Mechanics 2010. Book of extended abstracts. Praha, Institute of Thermomechanics AS CR, v. v. i., pp. 111-123.

Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P. (1992) Numerical Recipes in C, second edition, Cambridge University Press.

[CUDA] http://en.wikipedia.org/wiki/CUDA.

[CUDA2] http://ixbtlabs.com/articles3/video/cuda-1-p5.html.

[CUDA3] http://developer.download.nvidia.com/compute/cuda/1_0/
NVIDIA_CUDA_Programming_Guide_1.0.pdf.

[OpenCL] http://en.wikipedia.org/wiki/OpenCL.

[PS3] http://www.ps3cluster.umassd.edu/.