ADVANCED MATHEMATICAL ON LINE ANALYSIS IN NUCLEAR EXPERIMENTS

Usage of parallel computing CUDA routines in standard root analysis

Andrzej Grzeszczuk, Seweryn Kowalski
Nuclear Phys. Dept., Institute of Physics, University of Silesia, Katowice, Poland
E-mail to author: andrzej.grzeszczuk@us.edu.pl

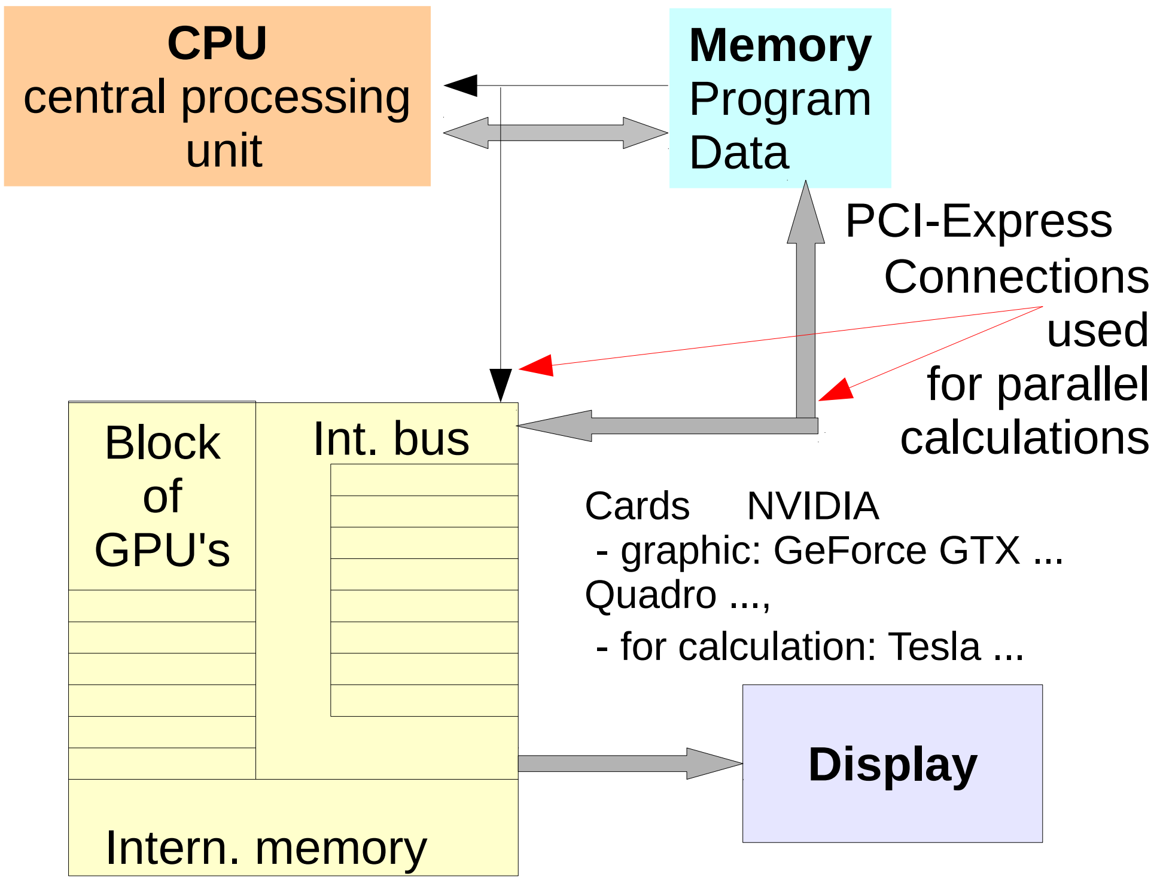
for IWM-2014, Catania, May 2014

Base:
Huge calculation power of present graphic carts needed for fast real time conversion video information used by games and 3D movies – realized by high level of parallel computing of drawing objects elements on screen

And:
Its calculation power can be used “mainly too” for advanced high level parallel calculation.

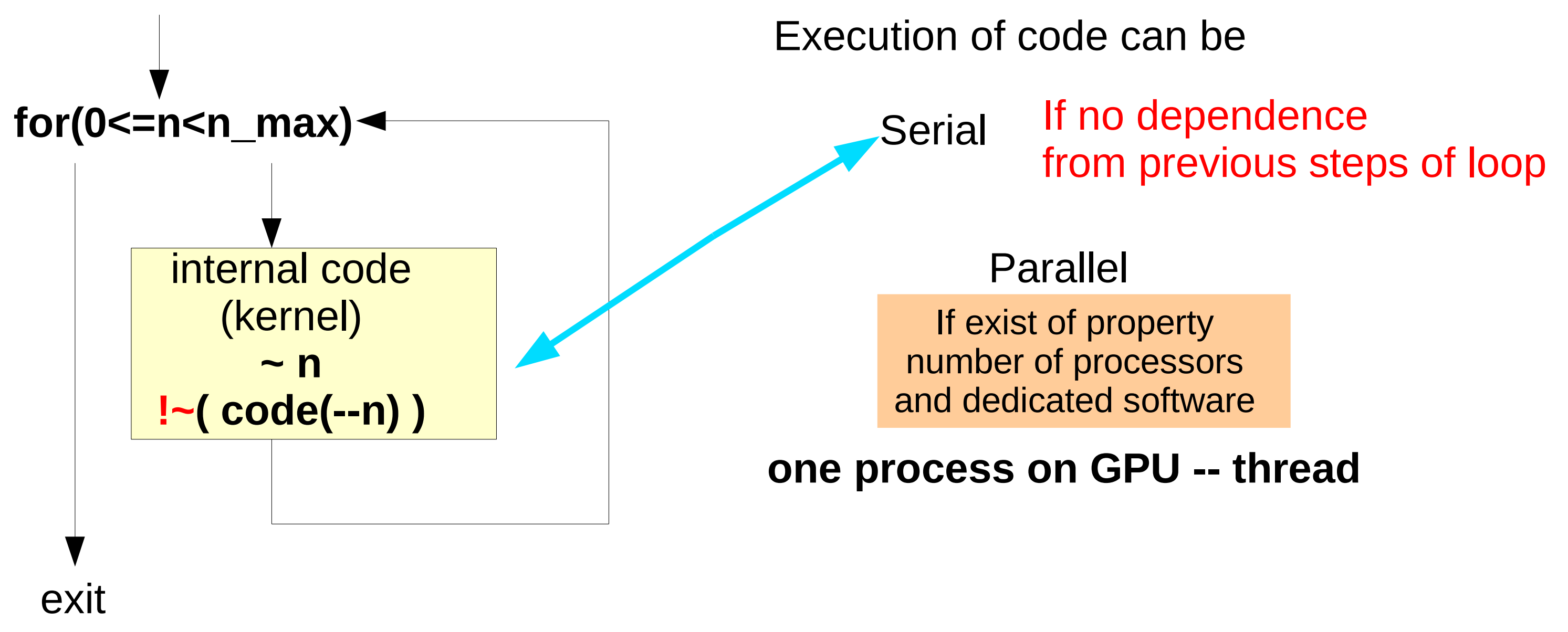
Supercomputer on desktop
Nvidia -- Tesla cards

Relations
CPU - central processing unit
GPU - graphics processing unit



Cards NVIDIA
- graphic: GeForce GTX ... Quadro ...,
- for calculation: Tesla ...

Example of parallelism:
loop
for(0 <= n < n_max)



Execution of code can be

Serial — If no dependence from previous steps of loop

Parallel — If exist of property number of processors and dedicated software

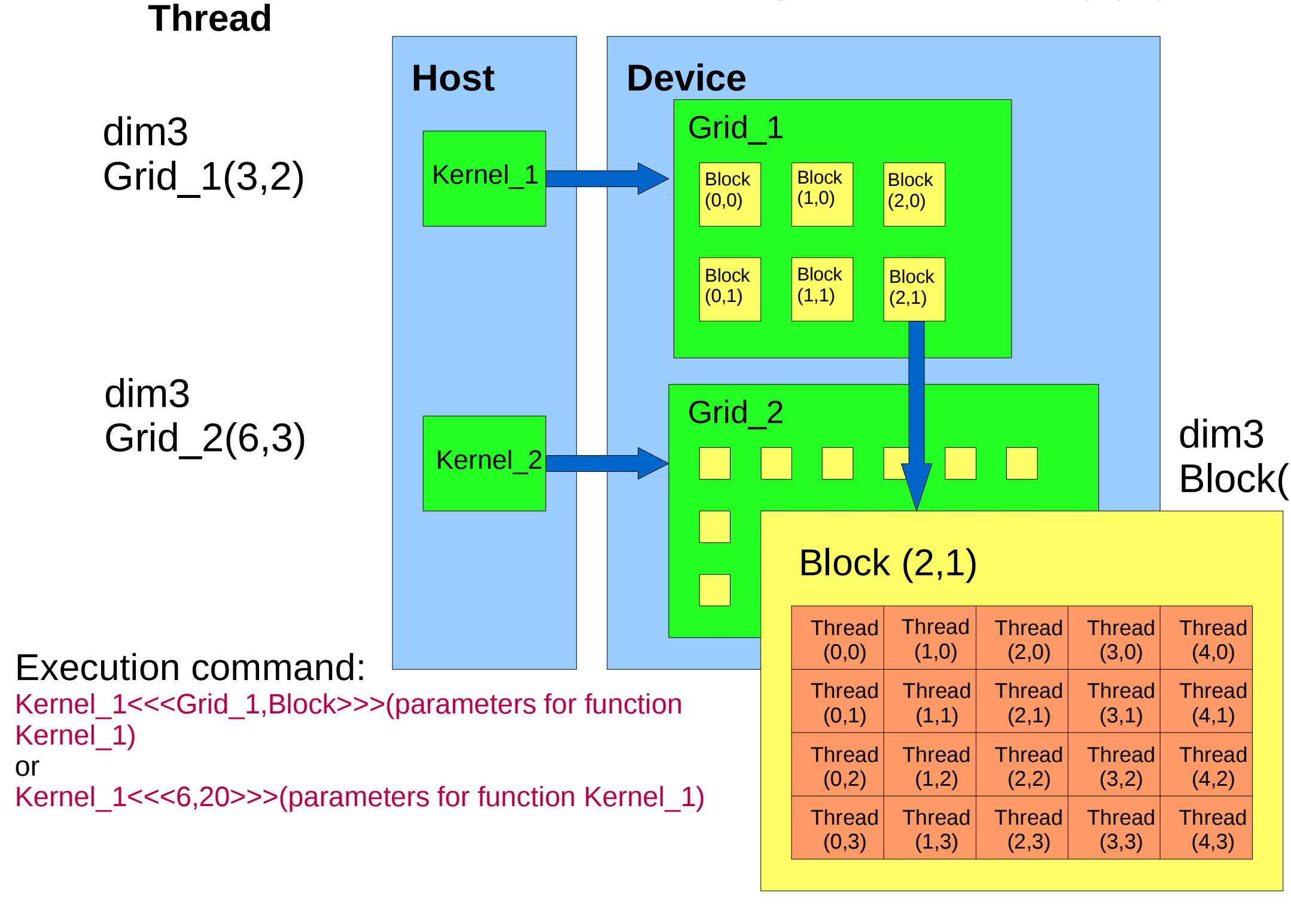
one process on GPU -- thread

Compute Unified Device Architecture (CUDA) is a parallel computing architecture developed by Nvidia for graphics processing.

System of: -threads

Device (GPU's card)
Grid (3 - dim)
Block (3 - dim)
Thread

Multidimensional access to blocks and threads by vector dim3 - example: dim3 dimBlock(x,y,z)



Execution command:
Kernel_1<<<Grid_1,Block>>>(parameters for function Kernel_1)
or
Kernel_1<<<6,20>>>(parameters for function Kernel_1)

-memory

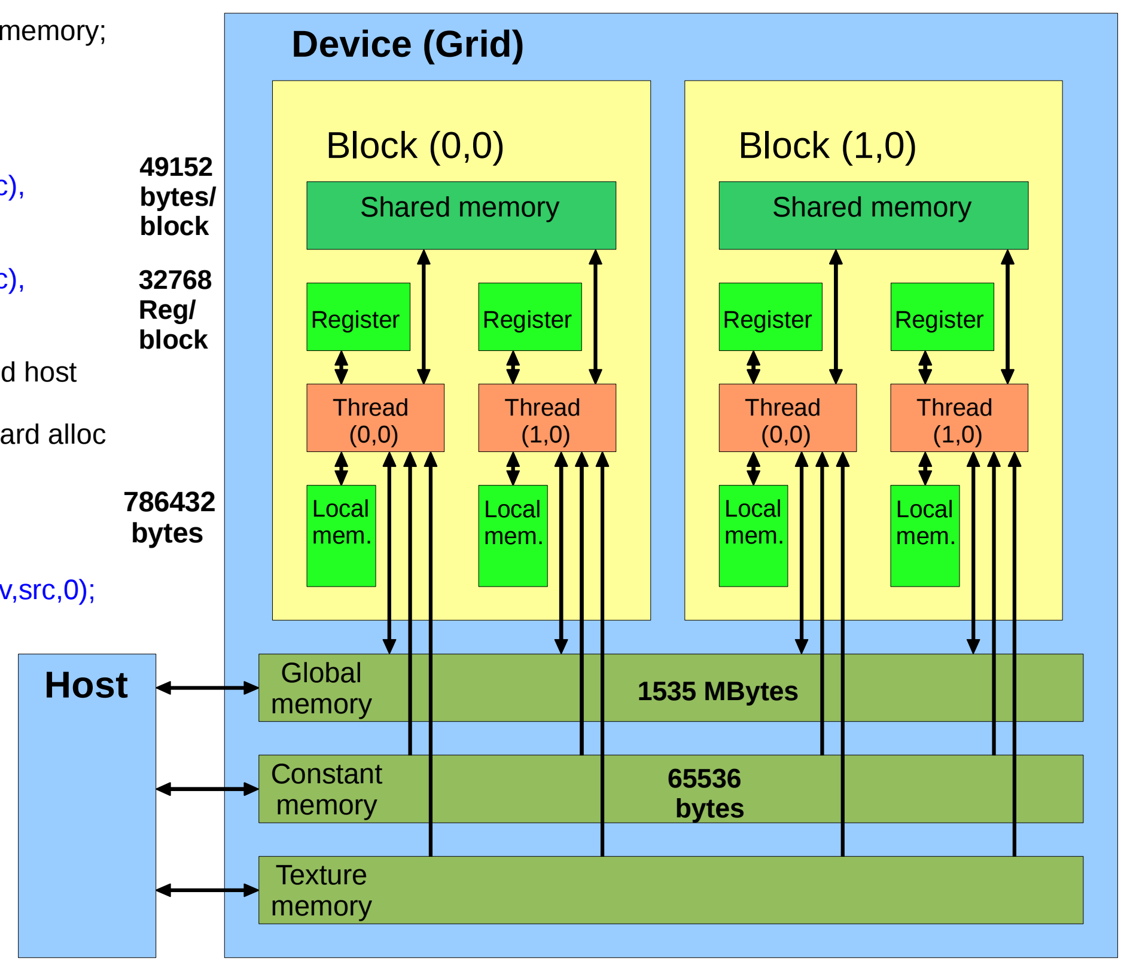
Management of memory:
- CUDA architecture accepts data in one dimensional linear form only
all other forms have to be converted to these shape of arrays

Controll function for usage of global memory;

```
cudaMalloc(&src_dev, sizeof(src));
cudaMalloc(&dst_dev, sizeof(dst));

cudaMemcpy(src_dev, src, sizeof(src),
      cudaMemcpyHostToDevice);

cudaMemcpy(dst, dst_dev, sizeof(src),
      cudaMemcpyDeviceToHost);
```

Controll function for usage of mapped host memory;
void *src=malloc(sizeof(src)); //standard alloc

```
cudaHostAlloc(&src,sizeoff(src),
      CudaHostAllocMapped);

cudaHostGetDevicePointer(&src_dev,src,0);
```



```
./deviceQuery Starting...
CUDA Device Query (Runtime API) version (CUDART static linking)
Found 1 CUDA Capable device(s)
Device 0: "GeForce GTX 580"
CUDA Driver Version / Runtime Version          4.10 / 4.0
CUDA Capability Major/Minor version number:    2.0
Total amount of global memory:        1535 MBytes (1609760768 bytes)
(16) Multiprocessors x (32) CUDA Cores/MP:     512 CUDA Cores
        GPU Clock Speed:                            1.54 GHz
Memory Clock rate:                          2004.00 Mhz
Memory Bus Width:                            384-bit
L2 Cache Size:                             786432 bytes
Max Texture Dimension Size (x,y,z)
                1D=(65536), 2D=(65536,65535), 3D=(2048,2048,2048)
Max Layered Texture Size (dim) x layers
                1D=(16384) x 2048, 2D=(16384,16384) x 2048
Total amount of constant memory:              65536 bytes
Total amount of shared memory per block:       49152 bytes
Total number of registers available per block: 32768
Warp size:                                 32
Maximum number of threads per block:          1024
Maximum sizes of each dimension of a block:    1024 x 1024 x 64
Maximum sizes of each dimension of a grid:     65535 x 65535 x 65535
Maximum memory pitch:                        2147483647 bytes
Texture alignment:                          512 bytes
Concurrent copy and execution:                Yes with 1 copy engine(s)
Run time limit on kernels:                  No
Integrated GPU sharing Host Memory:           No
Support host page-locked memory mapping:       Yes
Concurrent kernel execution:                 Yes
Alignment requirement for Surfaces:           Yes
Device has ECC support enabled:              No
Device is using TCC driver mode:             No
Device supports Unified Addressing (UVA):      No
Device PCI Bus ID / PCI location ID:          3 / 0
Compute Mode:
           < Default (multiple host threads can use
           ::cudaSetDevice() with device simultaneously) >
deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 4.10,
CUDA Runtime Version = 4.0, NumDevs = 1,
Device = GeForce GTX 580
```

Nvidia CUDA Tools:

Release: 4.2 / 12 April 2012, Operating system: Windows >=XP, Linux, Mac Os X, License: Freeware, Webside: http://www.nvidia.com/object/cuda_home_new.html

Cuda kit last release (for linux) contains:
-Drivers (ver 295.41),
-SDK NVIDIA GPU Computing Software Development Kit
group of tests and benchmarks for presented graphics and calculation software

-Toolkit binaries
--nvcc -nvidia compiler, based on nvopencc
-nvidia version of Open64 SGI compiler
--ptxas -gpu assembler
--nvvp -nvidia new profiler
--cuda-gdb -debuger

--cuda-memcheck
--bin2c, --cuobjdump -disassembler
--cudafe, --cudafe++, --fatbin,
--fatbinary, --filehash -programs in compilation phase

-Toolkit libraries
--libcudart.so -runtime
--libOpenCL.so -opencl
--libcufft.so -fft
--libcublas.so -blas
--libcusparse.so -sparse matrix
--libcurand,so -random numb. gener.
--libnpp.so -performance primitives

Programming - Simple Examples

Main.cu

```
#include <stdio.h>
int src[ ] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
int dst[10] = {0};

__device__ int foo(int aa, int bb)
{
   return aa*bb;
}

__global__ void kernel(int * src, int * dst)
{
   dst[threadIdx.x] = src[threadIdx.x] * 2;
   dst[threadIdx.x] = foo(dst[threadIdx.x],10);
}

int main()
{
   int * src_dev, * dst_dev;

   cudaMalloc(&src_dev, sizeof(src));
   cudaMalloc(&dst_dev, sizeof(dst));
   cudaMemcpy(src_dev, src, sizeof(src), cudaMemcpyHostToDevice);

   kernel<<<1, 10>>>(src_dev, dst_dev);

   cudaMemcpy(dst, dst_dev, sizeof(src), cudaMemcpyDeviceToHost);

   for(int i = 0; i < 10; i++) printf("%d\t", dst[i]);
   printf("\n");
}
```

“Classic” serial loop

```
for(int n=0; n<10; n++){
   dst[n]=dst[n]*2;
   dst[n]=foo(dst[n],10);
}
```

n <=> threadIdx.x

Compilation:

```
nvcc -o test main.cu
or
nvcc -c main.cu
g++ -o test main.o -lcudart
```

rtcuda.cu

rtcuda – example of interface between host and device units
In host part , the ROOT's library functions can be included

```
#include <stdio.h>
#include "rtcuda.h"

int src[ ] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
int dst[10] = {0};

__device__ int foo(int aa, int bb)
{
   return aa*bb;
}

__global__ void kernel(int * src, int * dst)
{
   dst[threadIdx.x] = src[threadIdx.x] * 2;
   dst[threadIdx.x] = foo(dst[threadIdx.x],10);
}

int rtc::rtcuda()
{
   int * src_dev, * dst_dev;
   cudaMalloc(&src_dev, sizeof(src));
   cudaMalloc(&dst_dev, sizeof(dst));
   cudaMemcpy(src_dev, src, sizeof(src), cudaMemcpyHostToDevice);
   kernel<<<1, 10>>>(src_dev, dst_dev);
   cudaMemcpy(dst, dst_dev, sizeof(src), cudaMemcpyDeviceToHost);
   for(int i = 0; i < 10; i++) printf("%d\t", dst[i]);
   printf("\n");
   return 0;
}
```

rtcuda.h

```
#ifndef _rtc_
#define _rtc_

class rtc
{
public:
static int rtcuda();
};
#endif
```

rtclinkdef.h

```
#ifdef __CINT__
#pragma link C++ defined_in rtcuda.h;
#endif
```

```
nvcc -c rtcuda.cu
rootcint -f rtc_Dict.C -c -I/usr/include/cudart rtcuda.h rtclinkdef.h
g++ -c -I/$ROOTSYS/include rtc_Dict.C
g++ -shared -o rtc.so rtcuda.o rtc_Dict.o -lcudart  -lcrypt -pthread -lnsl -lm -ldl -rdynamic
```

Cula library benchmarks

```
Initializing CULA...
Initializing MKL...

Benchmarking the following functions:
-------------------------------------
           SGEQRF
           SGETRF
           SGELS
           SGGLSE
           SGESV
           SGESVD
-------------------------------------


    -- SGEQRF Benchmark  --

 Size   CULA (s)    MKL (s)   Speedup
------ ---------- ---------- ---------
 4096       1.07       8.89    8.2869
 5120       1.32      16.80   12.7744
 6144       1.95      28.99   14.8788
 7168       2.61      45.32   17.3399
 8192       3.45      68.02   19.7218

    -- SGETRF Benchmark  --

 Size   CULA (s)    MKL (s)   Speedup
------ ---------- ---------- ---------
 4096       0.59       5.76    9.7261
 5120       0.88       9.17   10.4011
 6144       1.26      14.35   11.4296
 7168       1.71      22.38   13.1139
 8192       2.30      34.30   14.9174

    -- SGELS Benchmark  --

 Size   CULA (s)    MKL (s)   Speedup
------ ---------- ---------- ---------
 4096       1.16       9.04    7.7727
 5120       1.79      16.85    9.4241
 6144       2.56      28.32   11.0471
 7168       3.54      44.33   12.5315
 8192       4.74      65.29   13.7863

    -- SGGLSE Benchmark  --

 Size   CULA (s)    MKL (s)   Speedup
------ ---------- ---------- ---------
 4096       1.29      27.49   21.3076
 5120       1.91      43.64   22.8623
 6144       2.81      68.82   24.4671
 7168       3.79      99.38   26.2445
 8192       5.01     141.75   28.2978

    -- SGESV Benchmark  --

 Size   CULA (s)    MKL (s)   Speedup
------ ---------- ---------- ---------
 4096       0.61       5.83    9.6138
 5120       0.91       9.28   10.2493
 6144       1.33      14.52   10.9380
 7168       1.78      22.60   12.6829
 8192       2.37      34.56   14.5763

    -- SGESVD Benchmark  --

 Size   CULA (s)    MKL (s)   Speedup
------ ---------- ---------- ---------
 4096      20.33    1598.62   78.6496
 5120      32.49    2190.45   67.4264
 6144      50.61    3644.20   72.0099
 7168      70.81    5609.31   79.2148
 8192     103.14    7379.31   71.5432
```

Conclusions – problems - future :

- Fast parallel computing gives unique possibility to apply more complicated calculation in on-line analysis of experiment. It can be useful when shape of pulses should be saved Parallel methods are able to increase speed of on-line data compression or reduce of amount data by usage advanced systems of limitation – on-line comparison real and based shapes of pulses.
- Presented solution is based on NVIDIA architecture Fermi (2010) – (Geforce 580); next generations are: Kepler (2012), Maxwell(2014), and future Volta (2015-2016)
- High progress in architecture - Fermi is characterized by application FP64 arithmetics, Kepler by including Dynamic Parallelism, Maxwell by Unified Virtual Memory and Volta by (will be) Stacked DRAM (Data from Nvidia Public GPU Road map)
- Constant problems coupled with GPU systems:
  - -relative slow transport of data between CUDA card and memory reduce real speed of calculations – effective speed is higher when “amount of calculations” for one thread is higher - big progress on this area: in benchmark on my computer Bandwidth ~ 730MB/s on present solutions ~5000MB/s,
  - -high power consumption – is reduced from ~200 wtts with 512 procs (Fermi) to ~120 wtts for ~1500 procs, (Maxwell)
- Most important problem: – Parallelism in mentality – People solve problems by understanding series of events (Me too :))