EPJ Web of Conferences **88**, 01007 (2015)
DOI: 10.1051/epjconf/ 20158801007

Advanced mathematical on-line analysis in nuclear experiments. Usage of parallel computing CUDA routines in standard root analysis

A. Grzeszczuk, S. Kowalski

Nuclear Phys. Dept., Institute of Physics, University of Silesia, Katowice, Poland

Abstract

Compute Unified Device Architecture (CUDA) is a parallel computing platform developed by Nvidia for increase speed of graphics by usage of parallel mode for processes calculation. The success of this solution has opened technology General-Purpose Graphic Processor Units (GPGPUs) for applications not coupled with graphics. The GPGPUs system can be applying as effective tool for reducing huge number of data for pulse shape analysis measures, by on-line recalculation or by very quick system of compression. The simplified structure of CUDA system and model of programming based on example Nvidia GForce GTX580 card are presented by our poster contribution in stand-alone version and as ROOT application.

The requirement of high speed for visualization of computer data - needed for present general application - forces construction changes of graphic processors. One of possible solutions is including parallelism in computing process for graphic devices, due to independence of parts of display process in different segments of screen. Parallel mode of computing was previous used by supercomputers for advanced calculation of complexed problems. Nvidia company implemented that idea in constructions of graphic cards. Simultaneously the other project Tesla was opened as application of devices with like structure, strictly for parallel calculations. The same architecture of these devices enables simple adapt the graphic card as independent strong parallel computer unit. These cards are existing in architecture of old version families, and next in solutions from 2011 year named Fermi series 500, Kepler 2012/600, Maxwell 2013/700 and Volta in next future.

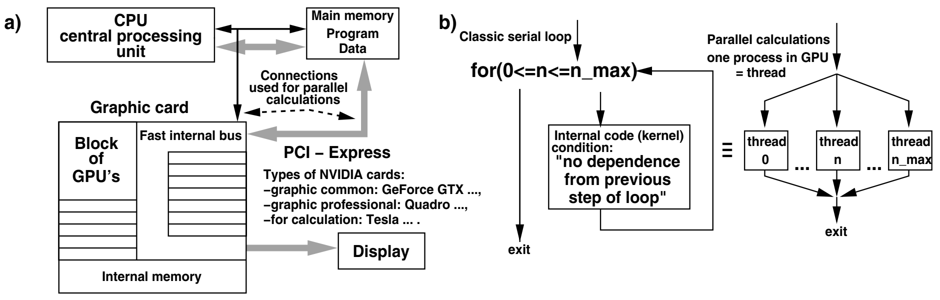

a)
CPU central processing unit
Main memory
Program
Data
Connections used for parallel calculations
Graphic card
Block of GPU's
Fast internal bus
PCI – Express
Types of NVIDIA cards:
–graphic common: GeForce GTX ...,
–graphic professional: Quadro ...,
–for calculation: Tesla ... .
Display
Internal memory
b)
Classic serial loop
for(0<=n<=n_max)
Internal code (kernel) condition: "no dependence from previous step of loop"
exit
≡
Parallel calculations one process in GPU = thread
thread 0
thread n
thread n_max
...
exit

Figure 1: Relations between central (CPU) and graphic (GPU) processing units (a), example for understanding of parallelism in execution of loop (b).

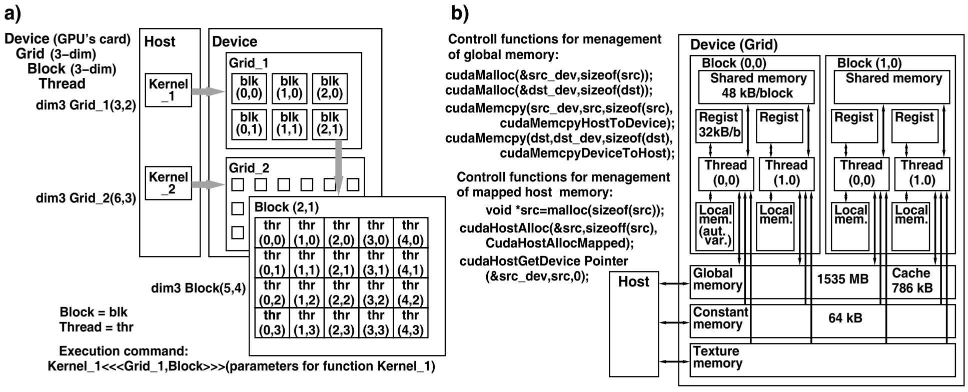
Figure 1 a) presents simplified construction of the card and system of communication between central unit, main memory and graphic units, b) shows simple example of application of parallel computing as substitution of classical loop.

The Nvidia developed standard - Compute Unified Device Architecture (CUDA) as frame for programming GPGPU devices. It is important to present that this system is coupled with calculations and graphics while other company solutions connected strictly with graphics. Nvidia provides free software CUDA support for Windows, Linux and Mac OSX systems for 32 and 64 bit versions. The support contains drivers for cards and tool-kit with free C++ compiler and system of libraries. The actual lasts version of CUDA software are: driver 331.79 from May 2014 and CUDA 6 production release.

The web page `https://developer.nvidia.com/cuda-zone` is devoted for description CUDA systems. It is good source for information and examples for all classes of users.

Figure 2 a) shows the hierarchy of CUDA threads. On level "Device" (particular board) different procedures "Kernel..." are executed in 3-dimensional Grids (one kernel for one grid) on separated Blocks. Each Block describes 3-dimensional structure of threads. On example the first procedure Kernel_1 is executed on 6(3*2*1)blocks * 20(5*4*1)threads = 120 threads. The example form of execution call command is also shown.

Figure 2 b) presents systems of CUDA memory and accessing commands. The one of main problems in case of CUDA system usage is relative law speed



Execution command:

```
Kernel_1<<<Grid_1,Block>>>(parameters for function Kernel_1)
```

Controll functions for menagement 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(dst),
        cudaMemcpyDeviceToHost);
```

Controll functions for menagement of mapped host memory:

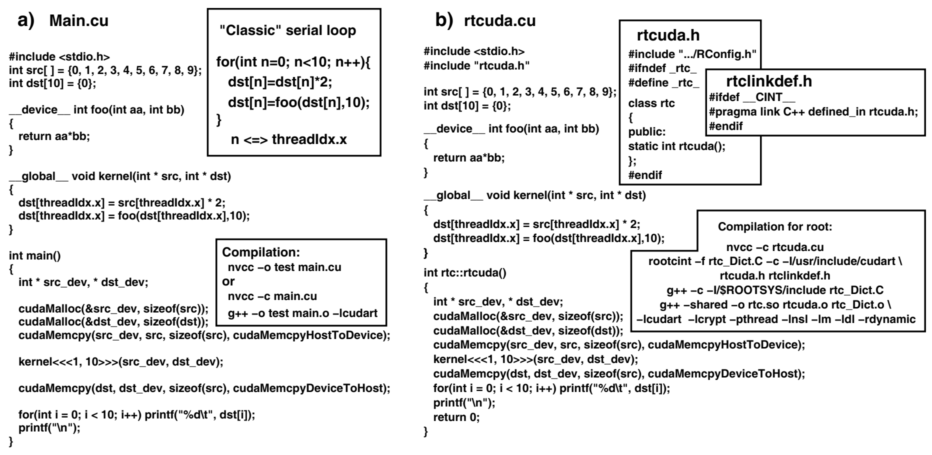
```
    void *src=malloc(sizeof(src));
cudaHostAlloc(&src,sizeoff(src),
    CudaHostAllocMapped);
cudaHostGetDevice Pointer
    (&src_dev,src,0);
```

Figure 2: System of threads management (a), and scheme of memories with access to them (b).

of connection between main memory of computer and internal memory of CUDA system. It depends on construction of computers and increases from old to new solutions but dramatic reduce efficiency of parallel calculation when "amount of mathematics" is low in thread. CUDA system gives two options to avoid this problems. In first, declared area exist in global memory of the card. The blocks of input data from main computer memory have to be copy to declared area and output data in opposite side. In other solution declared part of main memory works parallel works for operations with CPU and CUDA card with two independent addressing systems using different mappings. The efficiency of calculation on CUDA system can be increased after including different types of memory on card, due to differences in time of access and optimization for each of them.

The simple system for include parallel CUDA procedures to ROOT Data analysis framework is presented on last picture. Figure 3 a) shows example of simple loop and its parallel implementation *Main.cu* on ten GPU units in one block in one dimensional form ($<<< 1, 10 >>>$). Figure 3 b) presents main source *rtcuda.cu*, header *rtcuda.h* and *rtclinkdef.h* help header files needed for compilation for ROOT framework. The declaration *__global__* means that this routine is "kernel" procedure for GPU execution and the label *__device__* points that it is subroutine for kernel also for GPU. It should be emphasized that object files and libraries produced by Nvidia nvcc compiler can be linked together with modules prepared by GNU system.

The parallelism expands models of programming in last years on two ways. The first one is inclusion parallel methods in automatic compilation and optimization system. The second is using of parallel algorithms for so-



a) 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
```

b) rtcuda.cu

```
#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

```
#include ".../RConfig.h"
#ifndef _rtc_
#define _rtc_

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

rtclinkdef.h

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

Compilation for root:

```
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
```

Figure 3: The example of simple loop solving for illustration of parallel programming as stand-alone program (a) and as ROOT application (b).

lution of particular problems. The Cetus (`http://cetus.ecn.purdue.edu`) [1] project from Purdue University is example of translation system for paralleling of loops in source of program. The other example is CULA (`http://www.culatools.com`) [2] library containing parallel versions part of linear algebra LAPACK routines. The CUDA system can be used for increase of speed of on-line data compression in acquisition - algorithm references from Mast. Thes. A. Nicolaisen TU of Danemark (`http://www2.imm.dtu.dk/pubdb/views/edoc_download.php/6642/pdf/imm6642.pdf`) [3].

The our idea is connection CUDA architecture to on-line simplified analysis acquisition for reducing amount of data, especial for many channels pulse shape analysis. The gauss code for fitting shapes of pulses for on-line GPU analysis was adapted and is testing under ROOT system.

References

[1] Cetus - A Source-to-Source Compiler Infrastructure for C Programs, Purdue University 2011

[2] CULA - GPU-accelerated linear algebra libraries, EM Photonics 2013

[3] A. L. V. Nicolaisen, Algorithms for Compression on GPU, Mast. Thes. DTU Compute, Kongens Lyngby, August 2013