Lab Vision Systems – Learning Vision Systems on Graphics Cards (MA-INF 4308)
Assignment 1 Issues: 29.10.2015 WS1516

Lab policies

The room I.28 has 10 *cuda* and 3 *bigcuda* computers. Students can use cudas directly in I.28 or everywhere in the Informatik network. I.28 will be opened always for students. (If not, come to my office, I.09). If you want to access from the outside of Informatik network, first connect to one of the bigcudas and locally connect to a cuda using SSH.

Between 15:00 and 18:00 on every Tuesday (open hours), students can take a place in the lab and discuss with other students including me. This is not compulsory but recommended to communicate with you. During the open hours, I will keep my position in the room I.09 unless a special notice is given before.

Each assignment will be issued on once a week or two weeks depending on the level of the assignment. I'd like to get your feedbacks in any ways. The best way is to send me your source code and results when you finish the given tasks. Even if not completed, please let me take a look at your code and give you advice. Request on model code without any of your efforts is not allowed.

How to access to cudas using SSH

You can do your work in the room I.28 or at home by connecting the cuda systems as below.

- To *bigcuda* from inside Informatik network.

```
ssh -X yourid@bigcuda1
ssh -X yourid@bigcuda3
ssh -X yourid@bigcuda4
```

- To *bigcuda* from outside Informatik network.

```
ssh -X yourid@131.220.7.90          (bigcuda1)
ssh -X yourid@131.220.7.92          (bigcuda3)
ssh -X yourid@131.220.7.93          (bigcuda4)
```

- To *cuda* only from inside Informatik network.

```
ssh -X yourid@cuda1
ssh -X yourid@cuda2
...
ssh -X yourid@cuda8
```

Currently, 7 cudas(*cuda4, cuda5, cuda6, cuda9, cuda10, cuda11, cuda12*) and 2 bigcudas(*bigcuda1 and bigcuda3*) are alive.

Bigcudas are usually busy for research purpos. Please use cudas to do your assignments. If you need more computation power to do your work, e.g. final assignment, check its work schedule first before using them.

Background of CUDA C with an example

In the first assignment, we will learn how to use CUDA C codes in your project, and examine its computation performance according to the size of blocks and threads. Let's start with a simple linear vector operation, axpy: $z = a x + y$ to understand how CUDA works. You can download the example source code here:

http://ais.uni-bonn.de/WS1516/LabVision/ass_1_cudatest.zip

This project, `cudatest`, is written in C++ and with CMake. You can check its structure in the `CMakeLists.txt`

```
cmake_minimum_required(VERSION 2.4.6)

project (cudatest)

## CUDA
FIND_PACKAGE(CUDA REQUIRED)
INCLUDE(FindCUDA)

set (SOURCES
     main.cpp
     axpygpu.cu
)

set  (HEADERS
     axpygpu.h
)

CUDA_ADD_LIBRARY (axpygpulib
     axpygpu.cu
     axpygpu.h
)

CUDA_ADD_EXECUTABLE(cudatest axpygpu.cu)
ADD_EXECUTABLE  (cudatest ${SOURCES} ${HEADERS})
TARGET_LINK_LIBRARIES (cudatest)
```

The project includes three files: `main.cpp, axpygpu.cu, axpygpu.h`. As you see in the link `CUDA_ADD_LIBRARY,` the two later files compose a library `libaxpygpulib.a` which will use a GPU device. To do that, CUDA toolkit should be installed in the system so that CMake finds related packages and definitions. All cuda computers of course have the toolkit, but if you want to install CUDA toolkit on your system, refer to the nvidia document and follow step by step.

http://docs.nvidia.com/cuda/index.html#axzz3XH6ZQWvK
https://developer.nvidia.com/cuda-downloads

Let's look at `main.cpp.` First you can determine how many blocks and threads you want to use. Here, simply the size of vectors (`n`) is selected by the multiplication of the two values.

```
   int n_block = 10;
   int n_thread = 5;
   int a = 2.0;
```

With the constant parameter (`a`), AXPYGPU class is initialized and ready to operate.

```
   int a = 2.0;
   AXPYGPU axpy_gpu(n_block, n_thread, a);
```

Let's generate two random input vectors, x, y from 0 to 1 as an example.

```
for(int i=0;i<n;i++){
    x[i] = (float) rand() / RAND_MAX;
    y[i] = (float) rand() / RAND_MAX;
}
```

With axpy_gpu operator, the result z will be calculated in parallel with GPU.

```
// z = a*x+y
axpy_gpu.compute(x, y, z);
```

With axpy_gpu operator, the result z will be calculated in parallel with GPU.

```
// z = a*x+y
axpy_gpu.compute(x, y, z);
```

Now let's look at inside of AXPYGPU class. You can find several unfamiliar CUDA C commands in `axpygpu.cu`. First, the core operation function (called kernel) should be defined globally with `__global__` command. This function calculates $z = ax + y$ each elements in the vectors in each thread in GPU. You can also find three predifined variables, `blockIdx.x, blockDim.x, threadIdx.x,` which indicate indexes of blocks and threads. Current index of the vector is calculated by using the three variables.

```
__global__ void axpy(float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    y[i] = a*x[i] + y[i];
}
```

This is the initialization step of global memory in GPU that can be shared between the host and the device. `CudaMalloc` function creates the global memory as similar to `malloc`.

```
AXPYGPU::AXPYGPU(int n_block_, int n_thread_, float a_)
    :n_block(n_block_), n_thread(n_thread_), a(a_)
{
    n = n_block * n_thread;
    cudaMalloc((void **) &x, n*sizeof(float));
    cudaMalloc((void **) &y, n*sizeof(float));
}
```

When the axpygpu get the input vectors, it copies the data from the host to the global memory in the device. `cudaMemcpy` function do the work with `cudaMemcpyHostToDevice` command.

```
void AXPYGPU::compute(float* x_, float* y_, float* z_)
{
    cudaMemcpy(x, x_, n*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(y, y_, n*sizeof(float), cudaMemcpyHostToDevice);

    axpy<<<n_block,n_thread>>>(a,x,y);

    cudaMemcpy(z_, y, n*sizeof(float), cudaMemcpyDeviceToHost);

    cudaFree(x);
    cudaFree(y);
}
```

After executing kernel operation by `axpy<<<n_block,n_thread>>>(a,x,y),` the result in the global memory is copied to the host memory by using `cudaMemcpy` again.

Task 1

- Compile and run `cudatest` project using CMake.
- Test how many blocks and threads you can use with GPU. Compare its computation performance with different settings of blocks and threads. (Threads are faster than blocks)
- In order to compare its computation time with CPU, make AXPYCPU class with for-loop. Compare their computation times according to the size of vectors

Task 2

- Let $N=200, K=500, M=400$, Given three random matrixes, $A \in \mathbb{R}^{N \times K}, B \in \mathbb{R}^{K \times M}, C \in \mathbb{R}^{N \times M}$, and one random vector, $E \in \mathbb{R}^{N}$, determine the result of the vector, $D \in \mathbb{R}^{M}$ following equation efficiently with CUDA:

$$d_j = \sum_{i=0}^{N} \left(\sum_{k=0}^{K} a_{ik} b_{kj} + c_{ij} + e_i \right)$$

- Implement the same function with loops in CPU, and compare computation time difference and ensure that the two results should be the same.