Image processing algorithm optimization with CUDA for Pure Data

Ir. Rudi GIOT
Laras - ISIB
150, Rue Royale
Bruxelles, Belgique, 1000
giot@isib.be

Ing. Abilio RODRIGUES E SOUSA, Msc
IRISIB
150, Rue Royale
Bruxelles, Belgique, 1000
rodrigues@isib.be

Abstract

Production chains featuring industrial vision are becoming more and more widespread. Those processes are often heavy and applied to high-definition images with important frame rate. Powerful calculators are thus needed to follow the ever growing production rate.

NVIDIA[1] is currently designing interfaces providing a CUDA (Compute Unified Device Architecture)[2] architecture allowing parallel data computation. This could increase the performance of every operating system using graphical processing units (GPU).

Pure Data, thanks to its graphical modular development environment, allows fast prototype developments.

Those factors led us to start a research program dedicated to the realizations of image processing modules for Pure Data written in CUDA. First results are encouraging.

Keywords

- Image Processing
- GEM
- Cuda
- Pure Data
- Algorithm
- Optimization

1 Introduction

This project is set up with financial support of the Walloon Region[3].

The objective of this project is to develop and implement CUDA-written modules in a visual development environment, Pure Data.

The purpose is to accelerate the already existing image processing modules in Pure Data (the GEM[4] library – Graphics Environment for Multimedia) by dispatching a part of the process to the GPU.

Integrating CUDA blocks inside Pure Data will facilitate and accelerate the integration of prototypes from various kinds. The CUDA-blocks can be applied to any field requiring a high frame rate, a high resolution, an important amount of operations or computation-greedy processes. This application can be used also for industrial, medical or artistic purposes.

2 CUDA

The CUDA multi-core architecture allows parallel data computing. It can improve performances by using the NVIDIA graphics cards.

CUDA requires a driver that uses a streaming technique for communication between the GPU, called *device*, and the CPU (Central Processing Unit), called the *host*.

Every program written in CUDA follows four important steps. First, data to be processed by the GPU are copied in its memory (1). Then the CPU transmits the instructions (2) that will be executed on the graphics card (3). Once these previous steps have been accomplished, the results of those operations are copied in a local memory (4).

1 http://www.nvidia.com/

2 http://www.nvidia.com/object/cuda_home_new.html

3 http://www.wallonie.be/

4 http://gem.iem.at/

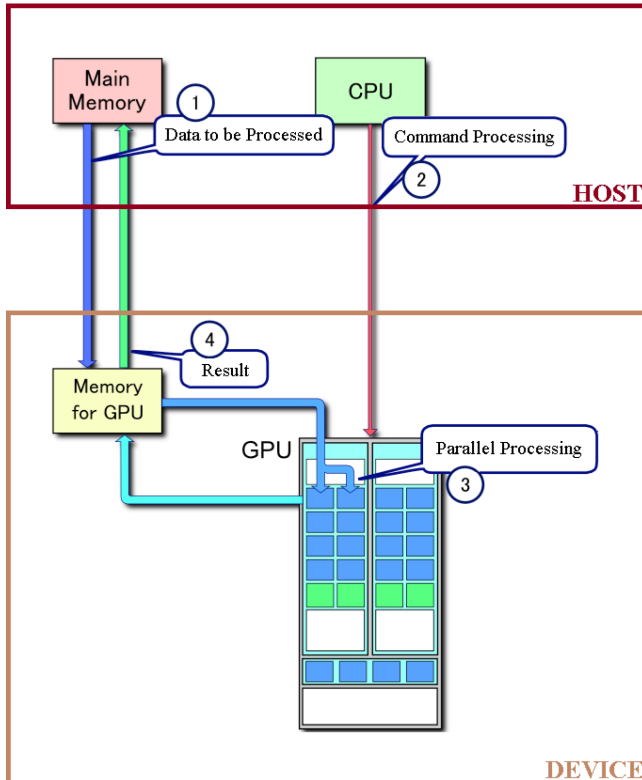

Figure 1: Programming steps in CUDA

In order to obtain good performances, massively parallelizable phases must be executed on the *device.*

In image processing, the CPU and GPU have a distinct role to play:

- **CPU** : Capturing the picture (camera) ;
- **GPU** : Processing ;
- **CPU** : Action to be done depending on the processing result ;
- **CPU** : Memory cleaning

2.1 Architecture

A GPU witch counts a strict minimum of 32 cores (average is 128) is called the *grid.*

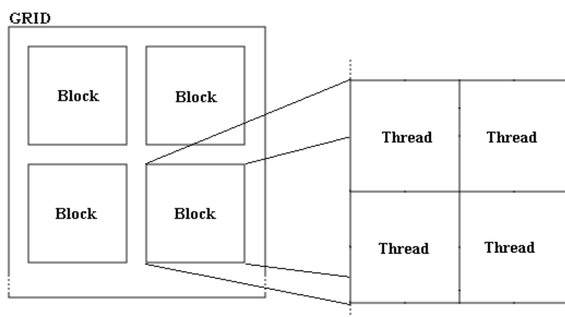

Figure 2: Grid subdivision in threads

This grid is divided into a given number of *blocks*. This number is defined according to the task to be undertaken at programming time. Each of those *blocks* contains multiple *threads* once the code has been written and taking into account the hardware (GPU cores number). A *thread* is the smallest subdivision of a task.

Blocks are independent. In other words they don't depend on the results coming from other *blocks*. *Threads* running inside a particular block can thus only communicate with *threads* from this *block*.

2.2 Programming

As already said, a CUDA program consists of two parts : one part is running on the *host* and the other is running on the *device*. The complete code can fit in one file (with the ".cu" extension), describing those two parts.

CUDA extends the C language by bringing 9 new key words, 24 new types and 62 new functions.

CUDA software architecture, suitable for running the CUDA program, consists of three parts:

- a **driver** responsible of transmitting calculation from the application to the GPU
- a **runtime** offering an interface between the GPU and the application
- A bundle of **libraries**

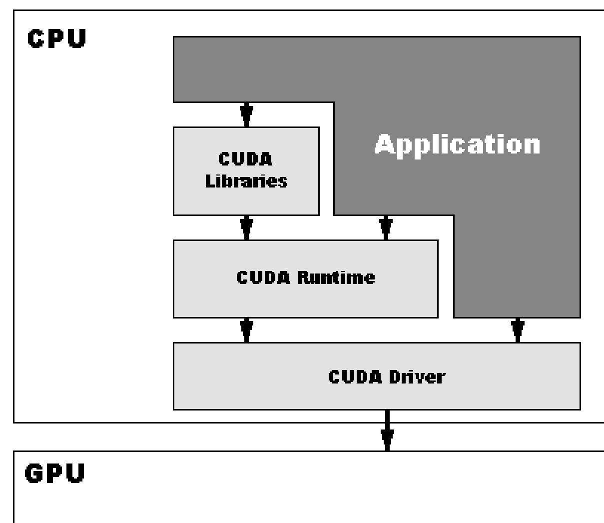

Figure 3: Software architecture of CUDA

The heart of CUDA is its *kernels. A kernel* is a portion of code to be executed in parallel on the *device*. Each instance of a *kernel* is itself a *thread*.

Let's take as an example a function that computes a threshold on a 640x480p grayscale picture (8 bits):

On a CPU, a threshold function may be implemented as this (function is included in GEM library):

```
void pix_threshold :: processGrayImage(imageStruct &image)
{
   int datasize = image.xsize * image.ysize;
   unsigned char *base = image.data;
   while(datasize--)
   {
         if (base[chGray] < m_thresh[chRed]) base[chGray] = 0;
         base++;
   }
}
```

Code 1 : Threshold function in GEM

On GPU, things are a bit more complicated. As it is showed in figure 1, GPU programming requires memory transfers CPU-GPU (steps 1-4) and function calls (step 2) that are executed on the GPU (step 3).

```
int main(int argc, char* argv[])
{
   ...
   cudaMalloc((void**) &imgGPU, imgSize * sizeof(uint8_t));   (a)
   cudaMemcpy(imgGPU, imgCPU, imgSize * sizeof(uint8_t),
                  cudaMemcpyHostToDevice);                     (1)
   Threshold <<< imgSize/512 , 512 >>> (imgGPU, m_thresh);    (2)
   cudaMemcpy(imgCPU, imgGPU, imgSize * sizeof(uint8_t),
   cudaMemcpyDeviceToHost);                                    (4)
   cudaFree(imgGPU);                                           (b)
   ...
}
```

Code 2 : Modified threshold function in GEM

```
__global__ static void Threshold( uint8_t *img, int m_thresh)
(3)
{
   int idx = threadIdx.x + blockIdx.x * blockDim.x;          (c)
   if ( img[idx] > seuil )      img[idx] = 255;               (d)
   else                         img[idx] = 0;                 (d)
}
```

Code 3 : Kernel of a threshold

The "main" function (code 2) is executed on the CPU, while the "threshold" (code 3) is executed on the GPU (called by the CPU).

GPU can only gain access of data in its memory. Therefore, the CUDA API offers some functions to handle the GPU memory: cudaMalloc(), cudaFree() and cudaMemcpy(). Those functions are respectively used for allocating memory on the GPU (a), freeing memory on the GPU (b) and executing CPU-GPU transfers (1 and 4).

Once data copied on the GPU, the CPU makes a *kernel call* (2), specifying in the right order both parameters "nBlock" and "threadBlock" (between triple chevrons which are preceded by "kernel" parameters).

The "nBlock" parameter defines the number of *blocks* inside the *grid*, while the parameter "threadBlock" indicates the number of *threads* to be simultaneously executed by each *block*.

In the previous example, the 640x480 pixels image, is divided into 600 (imgSize/512) *blocks*, each containing 512 pixels. Each block will execute 512 *threads*, one for each pixel.

As it can be seen in code 3, each *kernel* is preceded by a qualifier. In the example, "__global__" it means that the function is called by the CPU and executed on the GPU. For functions both called and executed on the GPU, the "__device__" qualifier must be used.

In the example, for each image, one *thread* is created for each pixel. To know which pixel a *thread* is working on, each kernel has three read-only variables:

- **blockIdx** : *block* index in the *grid* ;
- **threadIdx** : *thread* index in the *block* ;
- **blockDim** : number of *threads* by *blocks*.

The identification of a *thread* (c) is done by multiplying the *block* size by its index (blockIdx.x*blockDim.x) and by adding the *thread* index (+threadIdx) to it. Therefore, every *thread* is able to identify the pixel to be processed (d).

3 CUDA integration inside Pure Data

3.1 GEM Library

The image processing library of GEM is one of the most frequently used in Pure Data. The Gem project on Visual Studio has a modular architecture. An image processing always begins by "pix_", following by the name of the process itself. For instance, "pix_threshold" as showed in code 1.

3.2 CUDA integration inside GEM

In the example, the integration is done by replacing every parallelizable portion of the GEM code.

The "pix_threshold" function (code 1) is replaced by the "main" function (code 2) by adding some lines for variables initialization. Meanwhile, the *kernel* is replaced by a ".cu" file.

In order to compile the modified GEM project, Visual Studio needs an additional rules file for CUDA (cuda.rules) to be placed inside the « VCProjectDefaults » folder.

Visual Studio uses two compilers:

The Visual Studio compiler for the code executed on the *host* (standard C ANSI/ISO).

The NVCC[5] compiler, published by NVIDIA for the code executed on the *device* (standard C ANSI/ISO with some CUDA extensions).

Once the "gem.dll" file obtained, it is still to be placed in the "extra" folder of Pure Data, so the modifications may be taken into account by the software.

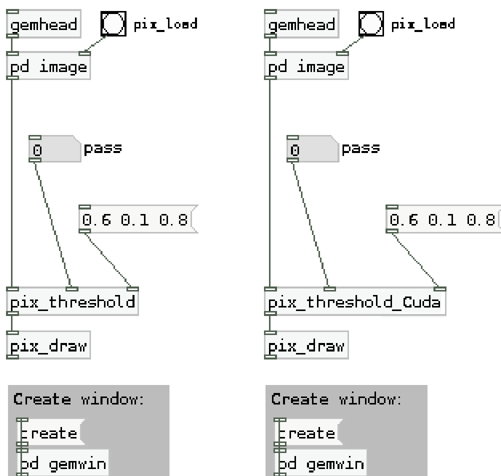

Figure 4: GEM original VS GEM modified by CUDA

Figure 4 presents an utilization of the new CUDA block, compared to the old version.

Regarding the performances, the processing on the GPU is nearly thirty times faster (table 1) than the CPU. However, a significant time is lost when transferring from CPU to GPU and from GPU to CPU.

The block is divided into three parts. First one to allow the transfer from CPU to GPU. Second one to transfer the calculation to be made. Third one to transfer the result from GPU to CPU.

In Pure Data, the patch becomes like figure 5.

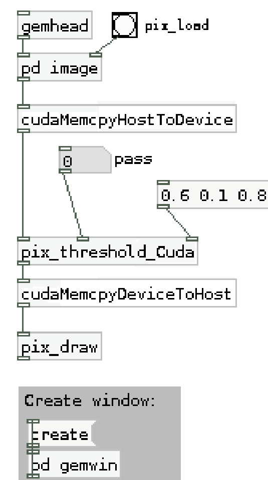

Figure 5: Blocs de transfert des données

This way, when several processes are chained up, transfer times are penalizing only once.

For better performances, data transferred to the second additional bock may only contain position coordinates.

---

5 http://developer.nvidia.com/search/node/nvcc

4 Results and performances

Execution time for the threshold on a 1024x768p grayscale picture was measured. First measurement corresponds to the original function (CPU) and the second one to the modified CUDA function (GPU).

Threshold monochrom picture 1024 x 768	
CPU (ms)	2,68
GPU (ms)	0,098
Accelerating factor	**27,34**

Tableau 1: Computation time for threshold by CPU and GPU

Processing via the graphics card obtains a very good accelerating factor (27,34). This result is interesting if not taking into account the CPU-GPU transfer, which has an average of 8,4 ms for a 1024x768p picture.

Figure 6 shows the efficiency of a chain up of three CUDA processing blocks in Pure Data

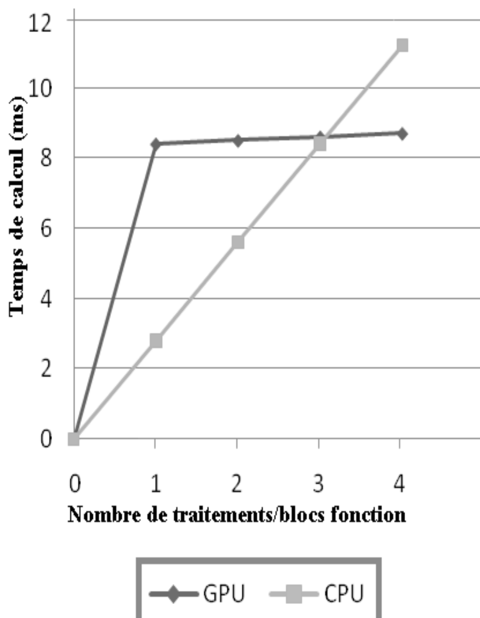

Figure 6: Computation time comparison between CPU and GPU

By executing one or two processes, the overall computation time on GPU combined with the time for data (picture) transfer is superior to the computation time with CPU only (8,5 against 5,8 ms).

When the the number of processes are by the number of three, computation time with both methods is slightly similar (8,4 ms).

It requires at least four processes to gain benefit from the GPU computation. (8,4 against 11,4 ms)

5 Conclusion

At this stage of the project, several image processing algorithms have been coded CUDA for Pure Data: RGB to grayscale transformation, thresholds, convolution, etc.

It was noted that in order to get advantage of the GPU, several processes are to be processed and/or the calculations are to be complicated enough.

So, by using the presented solution, for four or more function blocks (or complicated functions), the gain in computation time is real.

For the following of the First program, efforts will be provided for developments of new CUDA blocks, including in particular a “pix_draw_cuda” function. The purpose of it is to display processing results directly from the GPU, avoiding the transfer from GPU to CPU. This will improve performance.

6 Acknowledgements

Our thanks go to the Walloon Region Belgium for giving has the opportunity developing this challenging project.

Références

[1] J. Sanders and E. Kandrot: *CUDA by exemple – An Introduction to General-Purpose GPU Programming,* Addison-Wesley, Michigan, October 2010.

[2] D. B. Kirk and W. W. Hwu: *Programming Massively Parallel Processors – A Hands-on Approach*, Morgan Kaufman, 2009.

[3] J. Kreidler: *Jloadbang – Programming Electronic Music in Pure Data*,Wolke Verlag, Hofheim, 2009.