GPU-accelerated Fast Fourier Transform for Interactive Visualization of Audio Signals

Michele Rullo

25/02/2013

Abstract

Our modern fast-paced world is mostly being driven by the new computer technologies. With the advent of internet, the most brilliant human artifact of our times, and with the wide distribution of personal computers, Computer Science is becoming increasingly important. This thesis aims to be a "gate" toward the CUDA technology, presenting an experimental project which will introduce the reader to *Parallel Computing Programming* philosophy.
This work aims to progressively discover the potential of parallel computing through concrete examples, quoting official documents and presenting the software project with an algorithm-oriented explanation, guiding the reader to the final practical goal: **draw a 3D image of an audio stream, using CUDA technology**.

Acknowledgements

To my brother, Pierpaolo, who is the center of my universe.
To my parents, who constantly believe in me.
To my relatives, especially Franco, who have always supported me.
To Paolo, Enrico, Andrea and Pasqualino who have completely changed my life.
I wish to acknowledge the professor Marco Fratarcangeli, who gave me many reason to make my dream become a concrete objective.

Contents

Chapter 1

Introduction

1.1 Cuda & General Purpose Parallel Programming

"Because of various fundamental limitations in the fabrication of integrated circuits, it is no longer feasible to rely on upward-spiraling processor clock speeds as a means for extracting additional power from existing architectures. Because of power and heat restrictions as well as a rapidly approaching physical limit to transistor size, researchers and manufacturers have begun to look elsewhere." [2].

CUDA is the answer from NVIDIA to face the new era of computer's technology. Since we have reaching the physical limits of processors, the new technologies are focusing on **Parallelism** among different **Cores**. The reason is simple: **if we can't rely anymore on the processing velocity of a single entity, why don't we use more entities and make them work in cooperation?** It is now common to see how we can find Multi-Core processors and/or GPU with thousands of cores built inside in a personal computer. With no doubts we can tell how parallelism will be play an important role in future.

1.1.1 GPU's and General Purpose Parallel Programming

In order to understand what is the revolution driven by NVIDIA, it is necessary to talk about the early days of gpu's. With the evolution of Personal

Computers, softwares began to require always more resources, especially in the videogame industry:

> *"By the mid-1990s, the demand for consumer applications employing 3D graphics had escalated rapidly, setting the stage for two fairly significant developments. First, the release of immersive, first-person games such as Doom, Duke Nukem 3D, and Quake helped ignite a quest to create progressively more realistic 3D environments for PC gaming."* [2]

In order to improve the graphics capability of a personal computer, the necessity was to develop a new technology which allowed to separate data-processing from graphical information-processing: it was the dawn of Graphical Processing Units.
Mixing the capabilities of a CPU, which can only process single instructions at a time (*serial*), and of a GPU, which can process multiple instructions at a time (*parallelism*), PC could finally overcome important limits. This hardware "separations" allowed software houses to develop new softwares and videogames with better performance, extending, in a wide way, their horizons.
Later, GPU's parallelization capability began to seize the interest of researchers, who started to assess the real potential of this technology, looking for a way to exploit it for general purpose computation, that is, **the idea of using GPU's capabilities in order to accomplish every kind of programming tasks, not only for graphical computation**. The basic idea was to extend the original API's (Application Programming Interface) for graphics device. The original API's, in fact, were very limited, since they were constrained to work only for graphical purposes.

A first "trick" adopted to overcome these limits was to make GPU performing non-rendering tasks by making those tasks appear as if they were a standard rendering. This type of gpu-programming was a smart idea, but it showed its limits very soon, as it was too restrictive for programmers.

1.1.2 What is CUDA

> *"In November 2006, NVIDIA unveiled the industry's first DirectX 10 GPU, the GeForce 8800 GTX. The GeForce 8800 GTX was also the first GPU to be built with NVIDIA's CUDA Architecture. This architecture included several new components designed strictly for GPU computing and aimed to alleviate many of the*

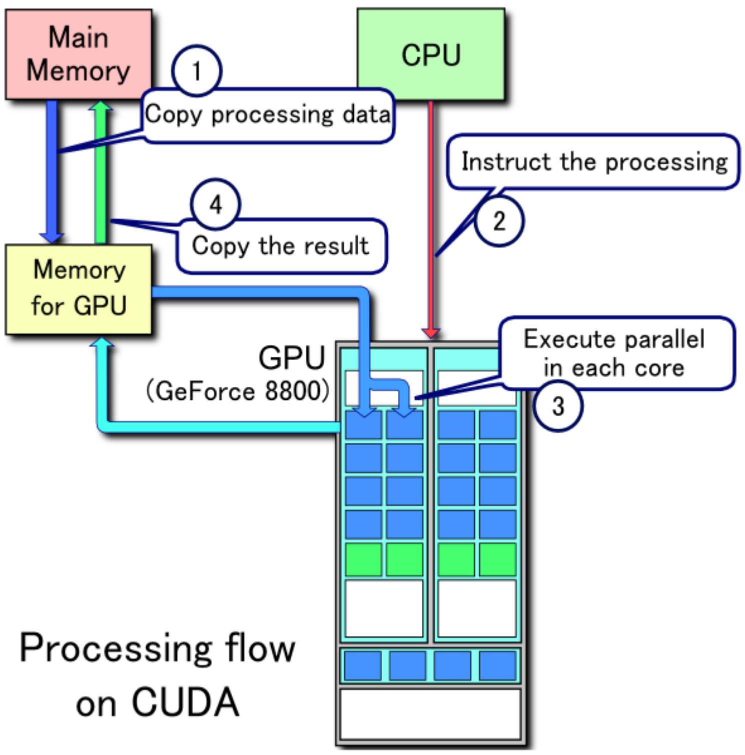

Figure 1.1: CUDA Processing Flow

limitations that prevented precious graphics processors from being legitimately useful for general-purpose computation." [2]

CUDA stands for **Compute Unified Device Architecture**, is a parallel computing platform and programming model created by NVIDIA and implemented by the Graphics Processing Units that they produce. Programmers can now write code in C/C++ in order to accomplish general purpose tasks without use API's like OpenGl or DirectX.

Looking at the figure 1.1 we can see a simple scheme which explain how it basically works, underlining the co-operation between CPU and GPU.

The process is divided in four phases:

1. The initial data are copied from the main memory (outside GPU) to GPU's memory, using CUDAmalloc() system call families.

2. The CPU instruct the GPU, preparing it to perform the desidered task,

launching the so-called CUDA Kernels, which contains code to execute in every single core of the GPU.

3. Every CUDA Kernel execute its task.

4. The output is copied back to the main memory, ready to be visualized or processed again.

This is the main pattern to perform tasks using CUDA.

1.1.3 Why CUDA

> *The modern GPU is a highly data-parallel processor. The GPU features many lightweight closely-coupled thread processors that run in parallel. While the performance of each thread processor is modest, by effectively using many thread processors in parallel, GPUs can deliver performance that substantially outpaces a CPU.* [1]

As we mentioned above, CUDA is a step ahead for computer technology, it allows programmers to significantly extend their possibilities. Nowadays we can easily program a GeForce GTX 560 Ti, using its 384 cores, at a very reasonable price. It is now possible to build gpu-accelerated softwares in order to significantly increase the overall performances. Another advantage is the possibilty to make the CUDA cores "communicate", using a fast shared memory regions. Such a CUDA architecture can be used to perform a wide set of operations, like *scientific (and engineering) computations*, *physics simulations* (like PhysX or Bullet) and so on, but it can also be useful in fields like *cryptography, computational biology* ecc..

We are discussing soon how CUDA technology affects an incredibly wide range of fields.

1.1.4 Trade-Off

Unfortunately, as always in engineering fields, there are several limitations using CUDA technology. Firstly, the communication between CPU and GPU. In fact we saw how (almost) every CUDA task begins and ends moving data between these two hardware components, this could actually be a bottleneck for performances, due to system bus latency and their relatives "low" bandwith. Programmers should be aware of this, limiting the data-transfer between the two devices.

Latter, complexity. Taking advantage of parallelism requires to formulate a different schematization of a given problem, therefore such a *parallel* formulation could even shows worse performances than a *serial* one. We always should remember to use CPU and GPU in a "complementary" way, avoiding, as much as possible, exclusive approaches.

1.2 Concrete Applications of CUDA technology

Here we have an example proposed by NVIDIA which shows the enormous potential of parallel computing, and how CUDA technology is affecting our modern society, not only in scientific fields

Medical Imaging
Since general-purpose parallel programming is thinked to overcome the computation limitations of modern computers, we can easily see how CUDA could solves many concrete problems. Here we have an example of medical diagnosis system, TechniScan, which can be finally implemented thanks to CUDA Architecture.

"TechniScan has developed a promising, three-dimensional, ultrasound imaging method, but its solution had not been put into practice for a very simple reason: computation limitations. [...] The introduction of NVIDIA's first GPU based on the CUDA Architecture along with its CUDA C programming language provided a platform on wich TechniScan could convert the dreams of its founders into reality. [...] Thanks to the computational horsepower of the Tesla C1060, within 20 minutes the doctor can manipulate a highly detailed, three-dimensional image of the woman's breast." [2]

1.3 Example: Summing Vectors

Now, we will have an example of how CUDA works. Given two vectors we will write code to perform sum between them using parallelization.

The basic idea is to separate the sum between single elements. So, according to the Figure 1.2, we will have four CUDA Kernels:

1. Kernel A: will perform $1 + 8 = 9$
2. Kernel B: will perform $3 + 3 = 6$

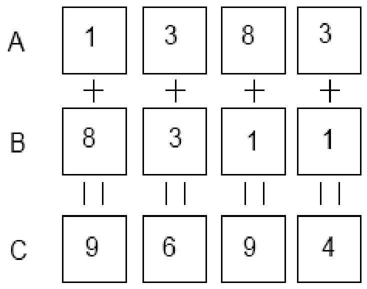

Figure 1.2: Summing two vectors

3. Kernel C: will perform $8 + 1 = 9$.
4. Kernel D: will perform $3 + 1 = 4$

Code: "main" method - CPU code

```
#define N 4

int main()
{
  int a[N], b[N], c[N]; //declare arrays
  int *dev_a, *dev_b, *dev_c; //declare pointers to gpu-arrays

  //Allocate on-gpu memory with cudaMalloc() for device arrays
  HANDLE_ERROR(cudaMalloc((void**)&dev_a, N * sizeof(int)));
  HANDLE_ERROR(cudaMalloc((void**)&dev_b, N * sizeof(int)));
  HANDLE_ERROR(cudaMalloc((void**)&dev_c, N * sizeof(int)));

  //Fill arrays
  a[0] = {1, 3, 8, 3};
  b[0] = {8, 3, 1, 1};

  //Phase 1: copying data from CPU to GPU
  HANDLE_ERROR(cudaMemcpy(dev_a, a, N * sizeof(int),
               cudaMemcpyHostToDevice));
  HANDLE_ERROR(cudaMemcpy(dev_b, b, N * sizeof(int),
               cudaMemcpyHostToDevice));

  //Phase 2: CPU instruct the GPU to perform tasks
  add<<<N,1>>>(dev_a, dev_b, dev_c);

  //Phase 4: copying the results back to CPU memory
  HANDLE_ERROR(cudaMemcpy(c, dev_c, N * sizeof(int),
               cudaMemcpyDeviceToHost));

  //Print results
  for (int i = 0; i < N; i++)
  {
    printf("%d+%d␣=␣%d\n", a[i], b[i], c[i]);
  }

  cudaFree(dev_a);
  cudaFree(dev_b);
  cudaFree(dev_c);

  return 0;
}
```

As we can see, code is clean and straightforward. We've implemented three of the four phases we mentioned above. The only particularity are the triple brackets on phase 2:

```
add<<<N,1>>>(dev_a, dev_b, dev_c);
```

We want to call "add" function on gpu with N (= 4) Kernels, passing "dev_a", "dev_b" and "dev_c" as parameters. Now, talking about the "1" inside the triple brackets: CUDA technology allows us to execute code on parallel Kernels, but every Kernel, in turn, can execute multiple instances of code using one or more Threads. In our case, one Thread per each Kernels is enough.

N.B.: using a combination of Kernels and Threads we can talk about "grids", where every cell is identified by a Kernel-Thread pair.

Now we can procede to phase 3: the "heart" of CUDA, the Kernel Code.

Code: Kernel Cuda - GPU code

```
//Phase 3: Execute CUDA Kernels
__global__ void add(int *a, int *b, int *c)
{
        int tid = blockIdx.x;
        if (tid < N)
                c[tid] = a[tid] + b[tid];
}
```

With only two lines, we can perform parallel computation of a sum of two vectors.

Firstly, we used the "__global__" qualifier in the method signature, it specifies the visibility of the Kernel code. Using "__global__" we can call the method "add" from CPU code.

The explanation of this instructions resides in the general idea of parallel computation: we have to assign every single sum to a Kernel. So we would to have, for instance:

Kernel 0 : c[0] = a[0] + b[0]
Kernel 1 : c[1] = a[1] + b[1]
Kernel 2 : c[2] = a[2] + b[2]

and so on. We can accomplish this using the built-in variable "blockIdx.x", which returns an unique identifier for the Kernel which is performing that specific task.

So this is the first line:

```
int tid = blockIdx.x;
```

Since we always would have a stable and bugs-free code, we control if the variable "tid" is less than the "N" value. Of course, in this example, we would not need to check this, but it is a good practice to never forget it as our code starts to become more complicated.

```
if (tid < N)
        c[tid] = a[tid] + b[tid];
```

The last line is simply our goal: put the result in the "c" vector.

Chapter 2

Thesis Argument

After this important introduction, which should us "guide" through CUDA and parallel computation philosophy, I will introduce you to my thesis topic.

2.1 Introduction

2.1.1 Reference Example

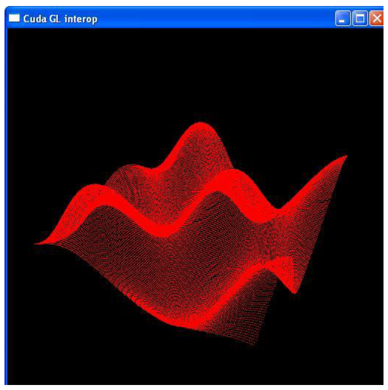

Figure 2.1: Reference Example

I began developing my thesis starting from the **"SimpleGL"** example provided by NVIDIA, which simply draws a sinusoidal 3D wave using CUDA. This is one of several open-source examples included in the CUDA SDK, officially released by NVIDIA.

The figure 2.1 show the sinusoidal 3D pattern drawn by SimpleGL. I have modified the original code without intervene on the rendering part, which is written by NVIDIA using OpenGL libraries.

2.1.2 Tools Used

Programming Languages and IDE's
All my project is developed using C++ Language and Visual Studio 2010.

Extern Libraries
In order to elaborate the **audio stream** I've used SndLib: http://www.mega-nerd.com/libsndfile/.
The **Fast Fourier Transform** is computed with FFTW: http://www.fftw.org/index.html.

2.1.3 The Algorithm

The goal of my experimental project is to extend the said NVIDIA example, drawing a dinamic three-dimensional image taking an audio stream as input, calculating its Fast Fourier Transform. In this case, every CUDA Kernel will draw a pixel on the screen, using the FFT data previously calculated. The algorithm consists of three phases: Preliminary phase, CUDA phase and OpenGL drawing phase.

Preliminary Phase:

1. Acquire stream data from the audio file.
2. Calculate initial DFT data.

CUDA Phase:

1. Copy DFT data from CPU memory to GPU memory.
2. Launch CUDA Kernels from CPU.
3. Execute Kernels.
4. Copy results back to CPU memory.

The OpenGL phase is already implemented by Nvidia.

After this last step, the loop will be restarted, until there will be no more data to compute.

2.1.4 Code Organization

The project is composed of the following files:

- *simpleGl.cpp*: CPU Code which implements the Preliminary Phase and the OpenGL drawing phase.
- *rendercheck_gl.cpp*: Code written by NVIDIA, for drawing purposes.
- *simpleGL_kernel.cu*: GPU Code which implements the CUDA phase.

2.2 The Code

Now, I will present the code, following the organization that we've mentioned before.

2.2.1 Preliminary Phase

In order to accomplish the Preliminary Phase I have written two methods, that I report on this document:

"init()" method: this is the initialization method, it verifies if the input audio stream is legal, reads first data from it and initialize the fftw data structures.

"next_data()" method: To compute the DFT, there are two possible approaches: calculate it at the beginning or calculate a partition of it time after time. I personally tested that the first approach is a bottle-neck for performances, since it requires a significant startup time and a big amount of memory. So I opted for the second method, to compute, step-by-step, subsets of data ready to be processed.

Acquire stream data from the audio file

Using the sndlib library open a .wav file is straightforward:

```
static SNDFILE* fIn;
fIn = sf_open( pszInputFile, SFM_READ, &info_in );
```

So, "fIn" is a pointer to a SNDFILE object, which will contains the audio data.

Fast Fourier Transform

The purpose of this project is to draw a 3D image starting from an audio stream frequency spectrum; in order to perform Fourier analysis we need to use the Discrete Fourier Transform (DFT). Calculating the DFT from its definition inevitably requires a O(n^2) algorithm. Using a Fast Fourier Transform (FFT) algorithm we can compute the DFT in only O($nlogn$) operations; with no doubts a significant improvement.

Here's the generic definition of Discrete Fourier Transform:

"The discrete Fourier transform (DFT) converts a finite list of equally-spaced samples of a function into the list of coefficients of a finite combination of complex sinusoid, ordered by their frequencies, that has those same sample values. It can be said to convert the sampled function from its original domain (often time or position along a line) to the frequency domain." (Wikipedia)

Mathematically:

The sequence of N complex numbers $x_0, ..., x_{N-1}$ is trasformed into an N-periodic sequence of complex numbers according to the DFT formula:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi kn/N}$$

Since each X_k is a complex number, we will extract its amplitude:

$$|X_k| = \sqrt{Re(X_k)^2 + Im(X_k)^2}$$

So, we would like to use these elements as inputs for the second phase. In order to accomplish that, here I show the code for calculating the DFT, using FFTW library:

First, we have to initialize all the data structures needed for FFTW:

```
snd_plan = fftw_plan_dft_r2c_1d(
                        nDftSamples,
                        fftw_in, fftw_out,
                        FFTW_ESTIMATE );
```

We've passed four parameters:

- *nDftSamples*: Number of samples.
- *fftw_ in*: The input data, i.e., the audio data.
- *fftw_ out*: Array for the output.
- *FFTW_ ESTIMATE*: flag.

The next phase is to execute the "plan" we've initialized in the previous step:

```
fftw_execute(snd_plan);
```

N.B.: the code will compute a total of nDftsamples every time the nextData() method is called.

Once we have ready data to process, we can successfully pass to the second phase.

2.2.2 CUDA Phase

Some pages before we've described the common pattern of a cooperation between CPU and GPU, this project it is not an exception. This "GPU" phase take as input the output produced in the previous phase. So, we will have a "step by step" explanation of the algorithm with the relative code, which will help us along to fully understand how the final output is generated.

First, I will introduce the two methods which I used to perform the CUDA phase:

File: *"simpleGL_ kernel.cu"*

CUDA Kernel

```
__global__ void kernel(float4* pos, unsigned int width, unsigned int height,
                          float time, fftw_complex* data)
{
    unsigned int x = blockIdx.x*blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y*blockDim.y + threadIdx.y;

    //calculate u-v coordinates
    float u = x / (float) width;
    float v = y / (float) height;

    float w = imabs(data[(int)(u * 220)]) / 2.0;

    u = u*2.0f - 1.0f;
    v = v*2.0f - 1.0f;

    //write output vertex
    int position = y*width+x;
    pos[position] = make_float4(u, w, v, 1.0f);
}
```

Kernel Launcher

```
extern "C" void launch_kernel(float4* pos, unsigned int mesh_width,
              unsigned int mesh_height, float time, fftw_complex* data)
{
    fftw_complex* cudaData;

    cudaMalloc((void**)&cudaData, 220 * sizeof(fftw_complex));
    cudaMemcpy(cudaData, data, 220 * sizeof(fftw_complex),
                                   cudaMemcpyHostToDevice);

    // execute the kernel
    dim3 block(8, 8, 1);
    dim3 grid(mesh_width / block.x, mesh_height / block.y, 1);

    kernel<<< grid, block>>>(pos, mesh_width, mesh_height, time, cudaData);

    cudaMemcpy(data, cudaData, 220 * sizeof(fftw_complex),
                                   cudaMemcpyDeviceToHost);
}
```

The first method is our "core", the code executed by every CUDA Kernel (more precisely, by every CUDA Thread), which will instruct the OpenGL phase to correctly draw the pixels on the screen.
The second method purpose is to prepare data to be processed by the first method, allocating memory on the GPU and instructing it to perform its Kernels.

Both methods take as inputs the same parameters:

- **float4* pos**: the VBO (Vertex Buffer Object) array, which will collect data to draw pixels on the screen (used by the OpenGL phase provided by NVIDIA)
- **unsigned int mesh_width** & **unsigned int mesh_height**: mesh dimensions.
- **float time**: current milliseconds past from the beginning (for debug purposes).
- **fftw_complex* data**: pointer to DFT data computed in the preliminary phase.

Now we can procede to explain, step by step, how these two method do their work.

Step 1: Copy DFT data from CPU memory to GPU memory

```
    fftw_complex* cudaData;

    cudaMalloc((void**)&cudaData, 220 * sizeof(fftw_complex));
    cudaMemcpy(cudaData, data, 220 * sizeof(fftw_complex),
                                   cudaMemcpyHostToDevice);
```

The first step is to initialize a pointer to "fftw_complex" objects:

```
    fftw_complex* cudaData;
```

Then we allocate memory on GPU using cudaMalloc() facility:

```
    cudaMalloc((void**)&cudaData, 220 * sizeof(fftw_complex));
```

So we allocate enough memory to store 220 *fftw_ complex* objects in the memory pointed by *cudaData*. We allocate 220 "blocks" because we only need to store DFT data from a range of 0 Hz to 20KHz.

Then we finally copy data from *data* array (passed as parameter) to *cudaData* array using cudaMemcpy():

```
    cudaMemcpy(cudaData, data, 220 * sizeof(fftw_complex),
                                   cudaMemcpyHostToDevice);
```

Notice how the last parameters specify the copy direction, from **Host** (CPU memory) to **Device** (GPU memory).

Step 2: Launch CUDA Kernels from CPU

```
dim3 block(8, 8, 1);
dim3 grid(mesh_width / block.x, mesh_height / block.y, 1);
kernel<<< grid, block>>>(pos, mesh_width, mesh_height, time, cudaData);
```

Starting from the first line:

```
dim3 block(8, 8, 1);
```

As we said before, in this phase we create a virtual grid where every cell is identified by a Kernel-Thread pair. In this case we instantiate a "dim3" object (that we've called "block") passing 3 parameters: *x, y, z*. Since we would like to create a bi-dimensional grid we set the z parameter to 1. In this way we will have $8 \times 8 = 64$ threads per blocks.

```
dim3 grid(mesh_width / block.x, mesh_height / block.y, 1);
```

Here we define the "grid" object, specifying its dimensions: we want our grid to have $(mesh\_width/block.x) \times (mesh\_height/block.y)$ blocks. So, for instance, if our image should be 512 pixels wide and 512 pixels high we would have $(512/8 = 64) \times (512/8 = 64)$ blocks. We will have as much pixels as threads.

```
kernel<<<grid, block>>>(pos, mesh_width, mesh_height, time, cudaData);
```

Now we simply launch the Kernel, passing the *grid* and *block* parameters we've just defined.

Step 3: Execute Kernels
We can now examine the CUDA Kernel.

Here's the first two lines:

```
unsigned int x = blockIdx.x*blockDim.x + threadIdx.x;
unsigned int y = blockIdx.y*blockDim.y + threadIdx.y;
```

Our objective is to write a single source code in order to perform parallel computing, so we need to associate every vertex to its thread. In order to accomplish that we use these two lines, which instantiate two unique variables *x, y*. Their uniqueness is guaranteed by the built-in *blockDim.x, blockDim.y, threadIdx.x, threadIdx.y* variables. In fact it's easy to notice how we use these variables so that they identify every single cell of the grid we've defined before.

```
//calculate u-v coordinates
float u = x / (float) width;
float v = y / (float) height;
```

Once we have successfully calculated the two identifiers (x and y) we now calculate the u and v variables, which represents the effective position of the vertices in the final image. Since we need to specify three coordinates for every vertices (we want to draw a 3D image) we should define another variable, which we will call w variable. It will depends by the DFT data we've previously calculated:

```
float w = imabs(data[(int)(u * 220)]) / COMPRESSION;
```

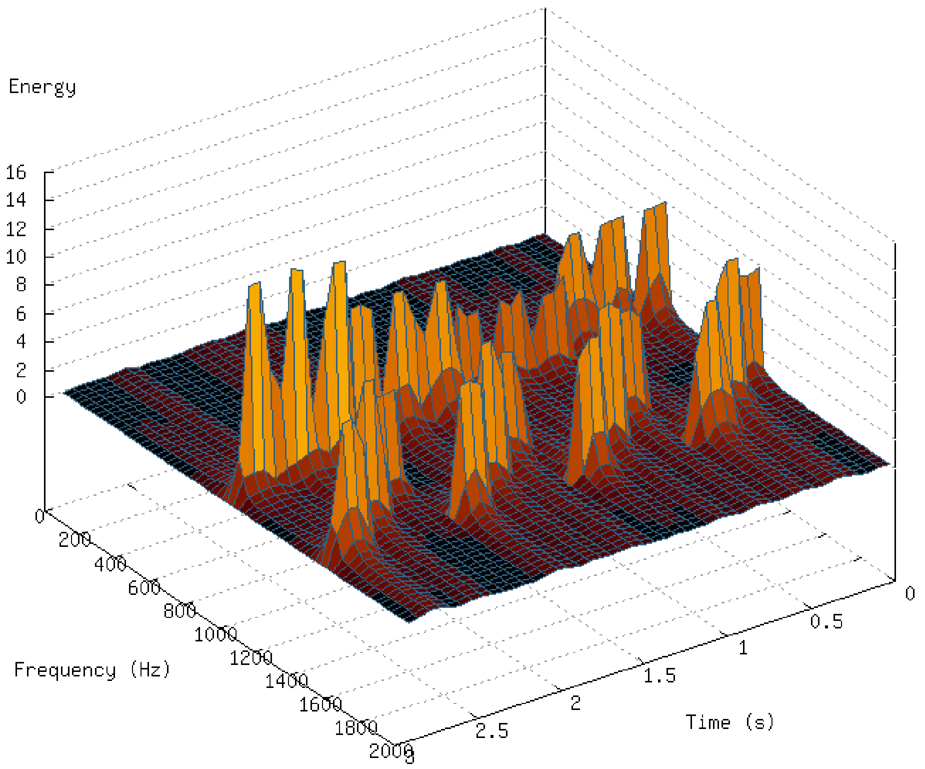

Figure 2.2: Reference System of the output

We have introduced mathematically the *Discrete Fourier Transform* some paragraphs before, in particular we said that every X_n is a complex number. So the first thing we want to do is to extract the magnitude of every X_n (using *imabs* method) and use it to calculate the w variable. It is necessary to make some clarifications about the parameter passed to *imabs* method. The final image will have the same reference system of the Figure 2.2, so the u variable will identify the **Frequencies** coordinate, the v variable will identify the **Time** coordinate and the w variable will identify the **magnitude** (**Energy**) coordinate. According to this we will use the u variable to index the right data associated to the current thread, after multiplying it per 220 (the range is between 0 and 220 KHz).

```
u = u*2.0f - 1.0f;
v = v*2.0f - 1.0f;
```

This is a simple correction in order to draw a more compact image (inherited by the original example).

```
int position = y*width + x;
pos[position] = make_float4(u, w, v, 1.0f);
```

The last two lines simply write the calculated vertex into the VBO (the *"pos"* vector) at the index specified by *"position"* variable.
N.B.: the *"pos"* variable is a pointer to an array which resides in the GPU memory! We will copy the results back to the CPU in the next step.

Step 4: Copy results back to CPU memory
We arrived to the last step of our project: copying back to CPU the computed vertices.
So, returning to the *"launch_kernel"* method we have:

```
cudaMemcpy(data, cudaData, 220 * sizeof(fftw_complex),
                                cudaMemcpyDeviceToHost);
```

As we can see this line is similar to the one we've used in the step 1; the difference is the copy direction, witnessed by the last parameter: *"cudaMemcpyDeviceToHost"*, which confirm that we are copying data from **Device** (GPU memory) to **Host** (CPU memory).

2.3 Demonstration

In this section I will provide some examples, showing the inputs characteristics, its frequencies spectrum and finally the 3D image produced. I will use as inputs simple audio tests, so it will be possible to easily "predict" the final output behaviour.

2.3.1 Low Frequencies Sample

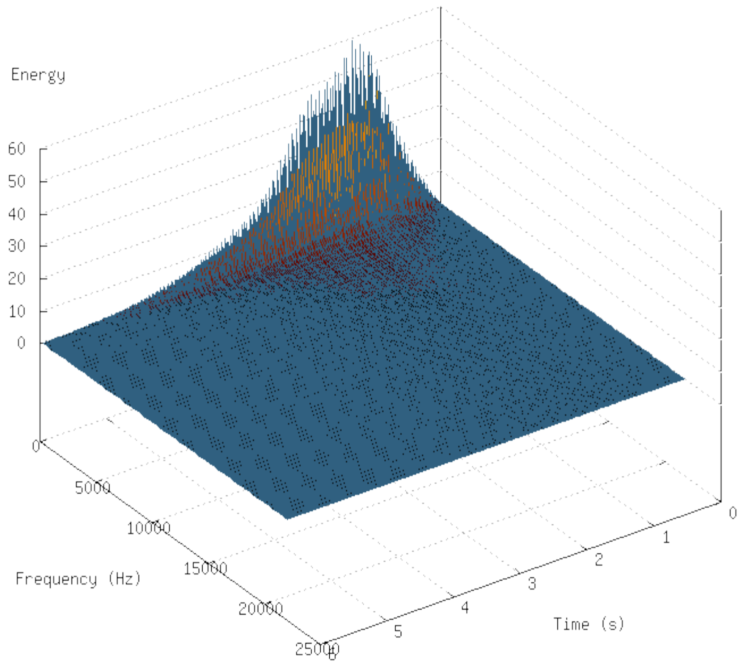

Figure 2.3: Low-Frequencies Audio Stream

The Figure 2.3 represents the frequencies spectrum of this first example, which is an audio stream characterized by low frequencies. As we can see, it starts with a **magnitude** equals to zero in $t_0 = 0$, grow almost constantly reaching its maximum **magnitude** value in $t_2 = 1$ and constantly returns to zero.

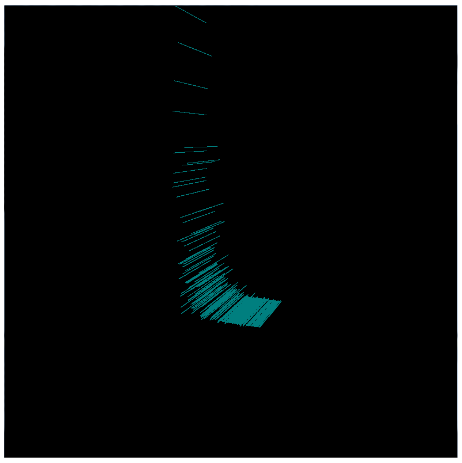

Figure 2.4: Low-Frequencies Audio Stream: CUDA Computation

After passing the sample as inputs in my project, in Figure 2.4 we have a screenshot from the final computation in a certain instant of time. We can see how the high frequencies are close to zero while the low frequencies magnitude is very high.

2.3.2 Constant Frequencies Spectrum Sample

The second sample is a 500 Hz constant tone. As we can see in Figure 2.5 there is a peak in correspondence of 550 Hz.

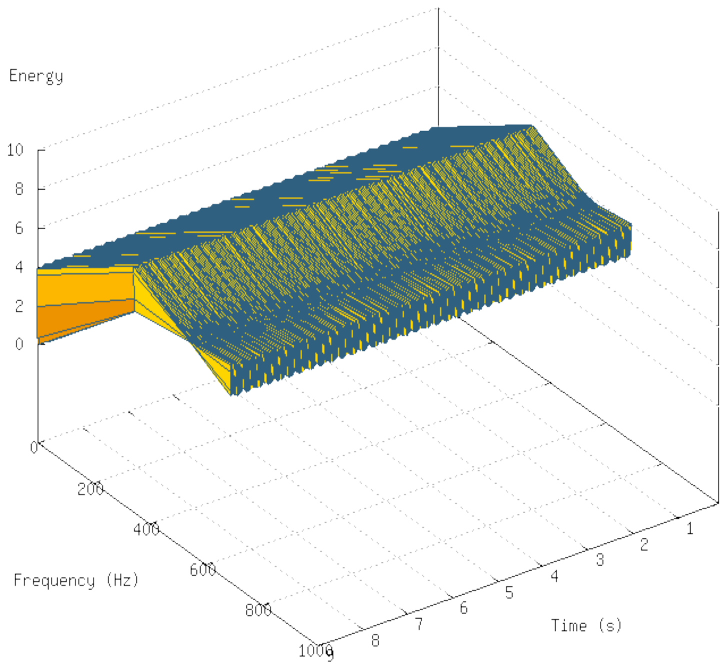

Figure 2.5: 500 Hz constant sound

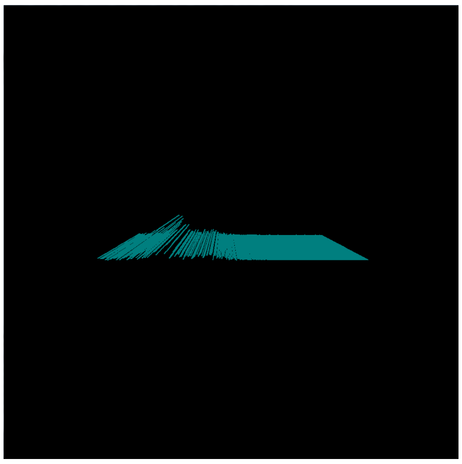

Figure 2.6: Constant Frequencies Spectrum Sample: CUDA Computation

The Figure 2.4 shows the output image generated by our program. In correspondence of 500 Hz there is the expected peak. Moving away from this point we notice how the magnitude progressively approaches to zero.

2.3.3 Siren Audio Sample

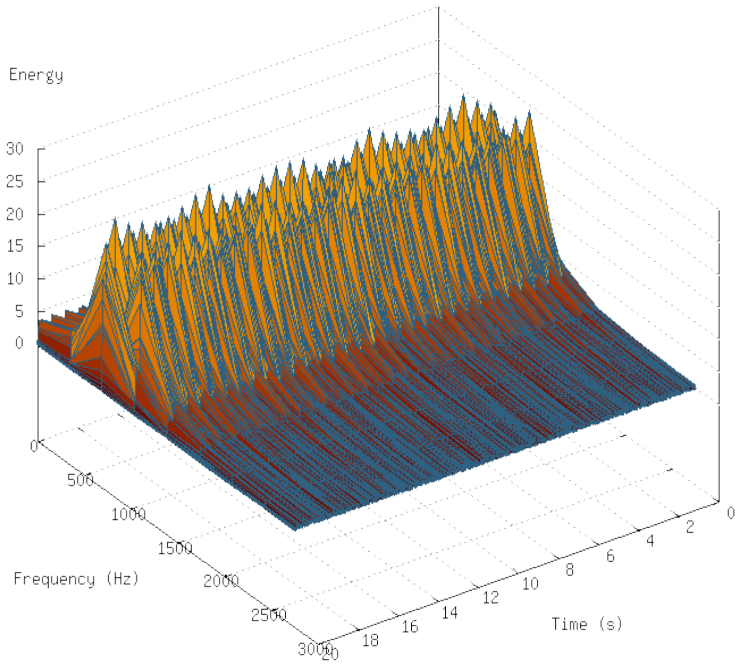

Figure 2.7: Oscillating Siren

This last sample shows the frequencies spectrum of an ambulance siren. The Figure 2.7 shows how the signal oscillates between two frequencies.

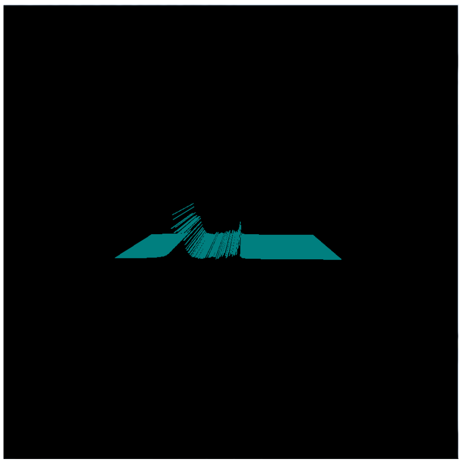

Figure 2.8: Siren Audio Sample: First Oscillation

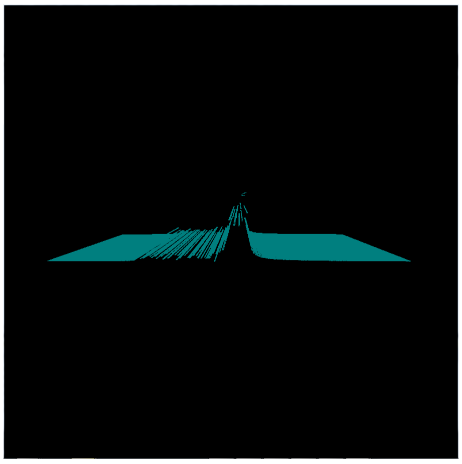

Figure 2.9: Siren Audio Sample: Second Oscillation

The Figure 2.8 shows the first frequency while the Figure 2.9 shows the second frequency. The dynamic image generated will oscillate between this two frequencies.

Chapter 3

Practical Applications

Such a project like this can easily find many kind of concrete applications. Now, I will show a practical example.

3.1 Stage Lighting System

Figure 3.1: Complex Stage Lighting System

The sound engineering has made many steps ahead in the last years, helping artists and musicians to greatly increase the quality of their performances. One of the most important element which make the difference on a stage, are lights. In a concert contest, it is a common practice to make lights follow the music rhythm. In order to accomplish this, a technician will take care of it using an appropriate controller. What about if we design an **automated** lighting system?
The idea is to convert my application in order to accomplish this task using CUDA technology.

The Figure 3.1 shows an example of a complex stage lighting system; it is easy to understand how it could be difficult to manage such a system like this. Our purpose is to design a software which will control the hardware, that is, the lights. Recording the audio stream from the live performance we would like to activate the lights in such a way they follow the music rhythm and intensity helping along to make the show more engaging.

Following the same idea we have used before, we will design a **Preliminary Phase** and a **Cuda Phase**.

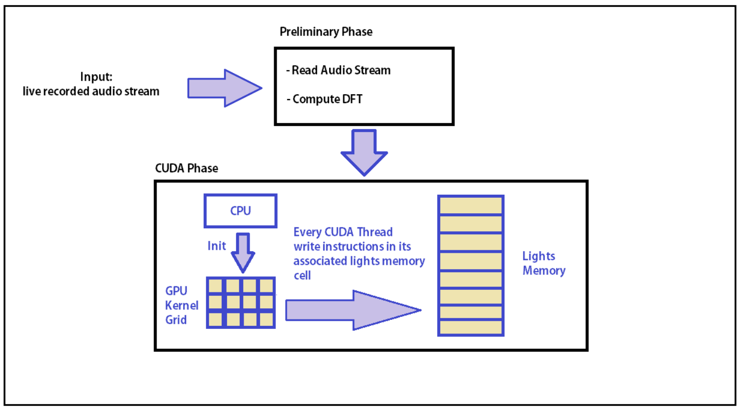

Figure 3.2: Automated Lighting System Scheme

Preliminary Phase
While musicians are playing, the audio stream that they produce, properly recorded with microphones and digital recorders, will be the input of the preliminary phase. As we did before, the preliminary phase simply consists of two steps: acquire audio data and calculate the Discrete Fourier Transform using the FFT algorithm.
Since the algorithm complexity is $O(nlogn)$ and we sample every time small "bursts" of DFT data, we can assume that this phase will be executed very fast. In this way the resultant latency will be unnoticeable.

CUDA Phase
Instead of drawing a 3D image now we should pilot all the lighting system using CUDA technology.
The idea behind is to **assign a set of lights to a set of CUDA threads**. To accomplish this we can assume that every lights set (which should contains one or more lights) have their own associated memory, where they can

read instructions (such as movement directions, light color, intensity, intermittence ecc..) which are given by the relative CUDA thread.
Each thread will instruct its associated lights set with a custom algorithm which will take as input the DFT computed in the preliminary phase.

Thanks to the powerful CUDA technology write such a code is straightforward and it guarantees a high-performance computation only using a personal computer equipped with an NVIDIA GPU. The overall effect given by the lighting system is only up to our creativity.

Bibliography

[1] Dan Geiger Anjul Patney Mark Silberstein, Assaf Schuster and John D. Owens. Efficient computation of sum-products on gpus through software-managed cache.

[2] Jason Sanders and Jack Dongarra. *CUDA By Example.* Addison Wesley, 2010.