July 9, 2018

University of Cassino and Southern Lazio

GPU-Accelerated CT Reconstruction

Parallel Processing Systems

MAIA 2017/2018

Author (s):
Ali Berrada
Ama Katseena Yawson

Supervisor (s):
Ing. Saverio De Vito

Abstract

For many years, X-ray Computed Tomography (CT) scanning has been successfully used as an essential medical imaging technique for diagnosing several diseases and anomalies. It allows physicians to see the “inside” of patients in a non-invasive manner by generating high-resolution, cross-sectional images thanks to the emanation of a series of X-ray beams at multiple angles. This process requires heavy computations to obtain high quality images in a very short period which conventional CPUs may not be able to achieve. In this work, we present an accelerated CT reconstruction implementation by exploiting the power of GPUs’ parallelization capabilities. Specifically, we optimized the filtering of the sinogram and the computation of back projection image. Huge improvements over the serial implementation were obtained which advocates for the continual and increasing use of GPUs in the field of medial imaging. The implementation was done using C++ and CUDA.

Table of Contents

1. Introduction

1.1. Overview

Computed Tomography abbreviated as CT is widely used in diagnostic imaging to aid Radiologists in an early detection of disease. This imaging modality utilizes computer-based combinations of many X-ray projections taken at different angles to produce cross-sectional (tomographic) images of specific areas of a scanned body. The intensity of X-ray emitted from the CT device is attenuated by the object following a decreasing exponential law. The extent of attenuation is dependent on the attenuation coefficient and the thickness of the object. Mathematically, the measured projection by the CT device is an integral of the attenuation coefficient. Since the attenuation coefficient depends on the density and characteristics of the material, reconstruction based on attenuation coefficient implies obtaining the information about the different characteristics of the objects in the X-ray line. Hence, it is possible to obtain an image of the body's slice by reconstructing the attenuation coefficient function from the projections and then imaging the attenuation coefficient [1].

1.2. Problem Statement

During CT reconstruction, contrast agents are injected into the patient’s body to target the area to be viewed. This contrast agent increases the density of the target of object and hence it appears very bright in X-ray images. The capturing time elapse for about 20 seconds depending on the number of images the scanner can capture. Immediately after this procedure, the scanned images are processed on a computer network which extrapolates three-dimensional geometry based on two-dimensional images. This process is known as back projection. It takes approximately 97% of the total time to reconstruct an image [2]. The overall execution time could be increased using GPU based computers.

1.3. Objective

The goal of this project is to maximize the speed of the CT reconstruction process by exploiting the GPU’s parallel computing capabilities.

2. Implementation

2.1. Algorithm

Before diving into the means of parallelizing the CT reconstruction process, we need describe the main mechanism of how this reconstruction works from an algorithmic perspective.

The input to the algorithm is the sinogram image and its output is the reconstructed image. The reconstruction is done pixel by pixel, i.e. the back-projection matrix is declared with its dimensions and then traversed element by element and the total back projection value for each of these elements is computed. However, to compute the value for any pixel (or matrix element), the sinogram matrix is scanned column by column as each sinogram column is a slice in the form of 1D column array and one element in this array contributes to the output pixel. To find which element in the sinogram slice contributes to the value of a certain output pixel, we rely on geometry. First, the output is a matrix (or image) whose coordinates system starts from top-left corner as [0][0] (following the convention of [row][column]). We need a mapping from this system to the Cartesian coordinates system – where the origin (0,0) is at the center – to apply geometry as we know it. Then, we can apply the formula $x * \cos(\theta) + y * \sin(\theta)$ to find where the projection of the pixel (x,y) ended up in the sinogram's slice taken at angle θ. The result obtained is of course in Cartesian system and it represents the distance from the origin along the x-axis only. The origin here is nothing but the center of the sinogram; therefore, when we map the location in the slice from Cartesian coordinates back to the array's indexing system, we need to add the index of the slice's middle element. When all slices have been traversed, the final value of the output pixel has been computed.

Figure 1 - Phantom image

Figure 2 – Back projection

This is the general idea of the back projection computation. However, the result suffers from a blurring effect as seen in Figure 2. Three possible ways to resolve this issue are: applying the central slice theorem; 2D FFT on the back-projection image; 1D filtering in spatial domain.

The approach adopted is to filter the sinogram in spatial domain. Specifically, we build a Shepp-Logan 1D filter and the filtered sinogram is obtained by the central part of the convolution between

every slice of the sinogram and the filter. The filtered sinogram is used instead of the original to perform the back-projection in the same manner described above.

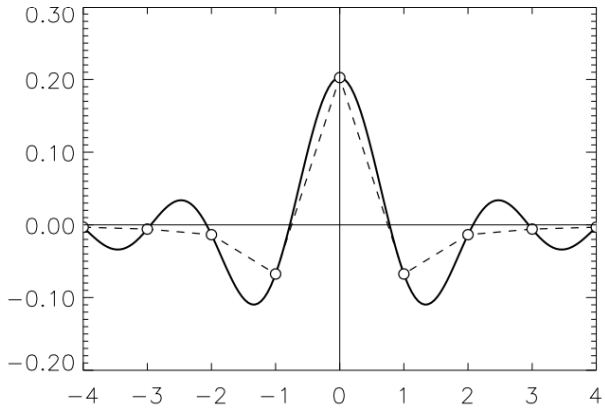


Figure 3 - Shepp-Logan filter (spatial domain) [3]

Figure 4 - Filtered back projection

2.2. Parallelization Strategy

Two main parts of our algorithm are parallelized: filtering the sinogram and computing the back projection. Therefore, 2 kernels were devised.

The first kernel is about filtering the sinogram which is a prerequisite for the back projection to obtain a reconstruction without the blurring effect. Each column of the sinogram is a 1D slice that needs to be convolved with the high pass filter. For this, we create multiple blocks each having as much threads as the size of a slice (i.e. the sinogram's height) where each block is charged to perform the convolution of its assigned slice with the filter. Since accessing the slice and filter data is very frequent and common for all threads within a block, they are loaded into shared memory. The filter is about twice the size of the slice so there is enough space. Once the convolution is done, each thread writes back the computed value to the corresponding location in global memory.

Now that the sinogram has been filtered, we can perform the back projection. However, we know that for each pixel to be reconstructed all the slices are needed which makes it impractical to load the whole sinogram into shared memory and for each block. In addition, each pixel to be reconstructed needs at most only one element from each slice. This is forcing the sinogram to be kept in global memory. However, we can do better by using texture memory where data is on-chip cached for optimized 2D spatial locality. This will be beneficial by improving the memory bandwidth since the sinogram exhibits important aspects of spatial locality.

In the second kernel, which performs the back projection, each thread computes its local and global index. The local index helps to locate the thread within the block while the global will be used to save the computed back projection value in the correct place in global memory. Also, this kernel computes the angles at which the slices were taken. Since these values will be frequently accessible by all threads, they are stored in shared memory. Each thread creates a register variable initialized

with 0 and then iteratively update it when parsing the sinogram slice by slice. When the location index within a slice is calculated, the value of the slice element at that index is retrieved from the texture-bounded sinogram.

3. Parallelization Results

3.1. Efficiency

The table below lists the execution time of the 2 main computation components of the algorithm – filtering the sinogram and computing the back projection – in both serial[1] and parallel[2] implementation.

Sinogram	**Sinogram Filtering**			**Back projection**		
	Serial	**Parallel**		**Serial**	**Parallel**	
		no texture	**with texture**		**no texture**	**with texture**
367 x 180	79 ms	0.06 ms	0.04 ms	2060 ms	0.92 ms	0.61 ms
729 x 180	285 ms	0.15 ms	0.16 ms	8284 ms	2.31 ms	2.06 ms

We can notice that when the size of the sinogram doubles, the time needed for completing a filtered back projection increases by a factor of 4. A sinogram of 729x180 (i.e. 729 parallel X-ray beams emitted at 180 different angles) is a small fraction of a real sinogram; yet, the reconstruction took more than 8s. When using the GPU, we found out that the sinogram filtering was at least 1000 times faster in parallel than in serial while the back projection computation was more than 3000 times faster.

3.2. Occupancy

Considering a sinogram of size 729x180 (i.e. 180 slices of 729 projections each), we evaluated the GPU occupancy as follows:

- Sinogram filtering kernel:

Configuration	
Compute Capability version	2.0
Threads per block	729
Registers per thread	4
Shared memory per block (Bytes)	8752
GPU Occupancy Data	
Active Threads per Multiprocessor	1458
Active Warps per Multiprocessor	46
Active Thread Blocks per Multiprocessor	2
Occupancy of each Multiprocessor	95.83 %

[1] *Tested on Intel® Core™ i5-3230M @ 2.60GHz Processor with 12GB DDR3*

[2] *Tested with NVIDIA Titan Xp 12GB GDDR5X*

- Back projection kernel:

Configuration	
Compute Capability version	2.0
Threads per block	768
Registers per thread	10
Shared memory per block (Bytes)	732
GPU Occupancy Data	
Active Threads per Multiprocessor	1536
Active Warps per Multiprocessor	48
Active Thread Blocks per Multiprocessor	2
Occupancy of each Multiprocessor	100 %

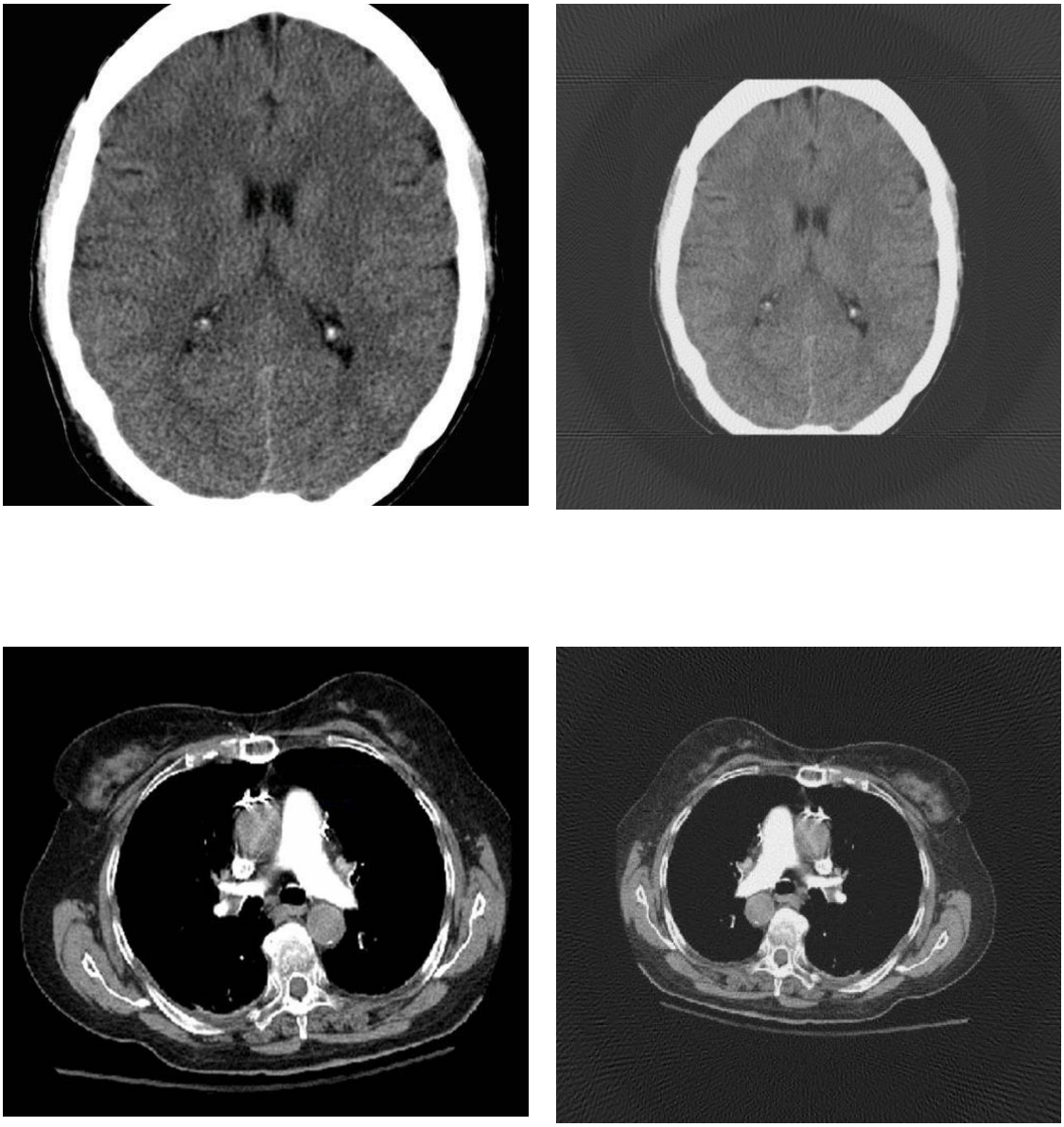
Since the kernel for back projection does not make much use of the shared memory, maximizing the occupancy and number of active threads were then prioritized.

4. Conclusion

This project sought to solve the CT reconstruction problem by exploiting the GPU's parallel computing power. The intended objective was reached as the reconstruction time was dramatically reduced by parallelizing the computation of the filtered sinogram and of the back projection. Furthermore, this project helped in putting into practice the theory learnt during the course and even new concepts were discovered such as the usage of texture memory. Future works can be done such as optimizing the configuration of the kernels to handle larger sinograms, removing the extra padding around the reconstructed image, and experimenting with other filtering methods.

Appendix A – Sample Outputs

Below are examples of the program's output (right original, left reconstructed).



Appendix B – Code

For complete code (and other implementations), visit [4].

```
#define PI 3.14159265358979323846

const int sinogramHeight = 729;
const int sinogramWidth = 180;
const int filterLength = (sinogramHeight%2 == 0) ? (sinogramHeight+1)*2 + 1 : sinogramHeight*2 + 1;

const int bpkernel_blocks = 692;
const int bpkernel_threads = 768;

// Texture reference for 2D float texture
texture<float, 2, cudaReadModeElementType> tex;

/************************** Methods declarations ** **********************/

/* Read file and load to 2D array */
template<typename T, size_t cols>
void loadFromFile(std::string filename, T mat[][cols], size_t rows);

/* Save 2D array to file */
template<typename T, size_t cols>
void saveToFile(std::string filename, T mat[][cols], size_t rows);

/* build a Shepp Logan 1D filter */
void getFitler(float* a, int s);

/***************** CUDA kernels ** ***************/

// Backprojection kernel
__global__ void k_backprojection(float *backProjection) {
	// thread's local index
	int tx = threadIdx.x;

	// thread's global index
	int bpIdx = blockIdx.x * blockDim.x + tx;

	const int bpWidth = sinogramHeight; // size of the back projection image
	const int totalSlices = sinogramWidth;

	// variables shared within each block
	__shared__ int midindex;
	__shared__ float angles[totalSlices]; // the angles at which the slices were taken (rads)

	// only 1 thread per block should initialize shared variables
	if (tx == 0) {
		// find middle index of the projections
		midindex = int(bpWidth / 2);
	}
	// meanwile other threads will compute the angles
	else if ((tx-1) < totalSlices) {
		angles[(tx-1)] = (tx-1) * PI / totalSlices;
```

```
	}

	// ensure all inits completed before continuing
	__syncthreads();

	// equivalent of the index in row-column
	int r = int(bpIdx/bpWidth);
	int c = bpIdx - (r * bpWidth);

	// local accumulator for the backprojection
	float val = 0;

	for (int sliceIdx = 0; sliceIdx<totalSlices; sliceIdx++) {
		// cartesian coordinates (x,y) for current pixel (cause some geometry is needed)
		int x = c - midindex;
		int y = -(r - midindex);

		// find where the projection of this pixel ended up in this sinogram's slice
		int distance = (int)round(x*cos(angles[sliceIdx]) + y * sin(angles[sliceIdx]));

		// convert to array index system
		int projectionIdx = midindex + distance;

		// check if we are in boundaries
		if ((projectionIdx > -1) && (projectionIdx < bpWidth)) {
			val += tex2D(tex, sliceIdx+0.5f, projectionIdx+0.5f) / totalSlices;
		}
	}
	// update in global memory
	backProjection[bpIdx] = val;
}

// Sinogram filtering kernel
__global__ void k_filtersinogram(float *sinogram, float *fltr) {
	// thread's local index
	int tx = threadIdx.x;

	// block index (corresponding to the ith slice)
	int bx = blockIdx.x;

	const int sliceLength = sinogramHeight;

	__shared__ float slice[sliceLength];
	__shared__ float filter[filterLength];

	// load slice to shared memory
	slice[tx] = sinogram[tx * sinogramWidth + bx];

	// load filter to shared memory
	filter[tx] = fltr[tx];
	filter[tx+sliceLength+1] = fltr[tx+sliceLength+1]; // each thread handles 2 elements
	if (tx == 0) {
		filter[sliceLength] = fltr[sliceLength]; // handle the middle element with thread#0
	}
	// wait for all threads to load data
	__syncthreads();
```

```
	// convolve slice with filter (centered convolution, eq. to matlab's conv(a,b,'same'))
	float val = 0;
	for (int j=0; j<sliceLength; j++) {
		val +=  slice[(sliceLength-1)-j] * filter[tx+j+1];
	}
	// update in global memory
	sinogram[tx * sinogramWidth + bx] = val;
}

/**************************
* Program starting point *
**************************/
cudaEvent_t start, stop;
float duration1 = 0, duration2 = 0;

int main(int argc, char ** argv) {
	// where to read and write data
	std::string inputFile = "sinogram.txt", outputFile = "reconstructed.txt";

	// load image in 2D array
	float h_sinogram[sinogramHeight][sinogramWidth]; // input
	loadFromFile(inputFile, h_sinogram, sinogramHeight);

	// build the filter
	float h_filter[filterLength];
	getFitler(h_filter, filterLength);

	// back-projection matrix
	const int bpWidth = sinogramHeight; // number of parallel x-ray emitters/receivers
	float h_backProjection[bpWidth][bpWidth];

	/********************** Start CUDA stuff  **********************/
	cudaEventCreate(&start);
	cudaEventCreate(&stop);

	// total bytes needed for each matrix
	size_t FILTER_BYTES = sizeof(float) * filterLength;
	size_t SINOGRAM_BYTES = sizeof(float) * sinogramHeight * sinogramWidth;
	size_t BACKPROJECTION_BYTES = sizeof(float) * bpWidth * bpWidth;

	// declare GPU memory pointers
	float *d_filter;
	float *d_sinogram;
	float *d_backProjection;

	// allocate GPU memory
	cudaMalloc((void**)&d_filter, FILTER_BYTES);
	cudaMalloc((void**)&d_sinogram, SINOGRAM_BYTES);
	cudaMalloc((void**)&d_backProjection, BACKPROJECTION_BYTES);

	// copy the inputs (filter and sinogram) to GPU memory
	cudaMemcpy(d_filter, h_filter, FILTER_BYTES, cudaMemcpyHostToDevice);
	cudaMemcpy(d_sinogram, h_sinogram, SINOGRAM_BYTES, cudaMemcpyHostToDevice);

	// launch the filtering kernel
```

```
	cudaEventRecord(start);

	k_filtersinogram << < sinogramWidth, sinogramHeight >> >(d_sinogram, d_filter);

	cudaEventRecord(stop); cudaEventSynchronize(stop);
	cudaEventElapsedTime(&duration1, start, stop);

	//******************** Set up texture memory ***************************

	// allocate array
	cudaArray *cuArray;
	cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc<float>();
	cudaMallocArray(&cuArray, &channelDesc, sinogramWidth, sinogramHeight);

	// copy image data
	cudaMemcpyToArray(cuArray,0,0,d_sinogram,SINOGRAM_BYTES,cudaMemcpyDeviceToDevice );

	// bind the array to the texture
	cudaBindTextureToArray(tex, cuArray, channelDesc);

	//*************************************************************************

	// launch the backprojection kernel
	cudaEventRecord(start);

	k_backprojection << < bpkernel_blocks, bpkernel_threads >> >(d_backProjection);

	cudaEventRecord(stop);

	// copy back the result to CPU memory
	cudaMemcpy(h_backProjection, d_backProjection, BACKPROJECTION_BYTES,
                                                        cudaMemcpyDeviceToHost);

	cudaEventSynchronize(stop); cudaEventElapsedTime(&duration2, start, stop);

	// free device memory
	cudaFree(d_filter); cudaFree(d_sinogram); cudaFree(d_backProjection); cudaFree(cuArray);

	// unbind texture
	cudaUnbindTexture(tex);

	/******************** End CUDA stuff ********************/

	// save to file
	saveToFile(outputFile, h_backProjection, bpWidth);

	printf("Filtering kernel time = %f ms\n", duration1);
	printf("Backprojection kernel time = %f ms\n", duration2);

	return 0;
}
```

Appendix C – Run CUDA programs without GPU

If your machine is not equipped with a dedicated NVIDIA graphics, we propose you a few options you might consider. It is important to search for more details to understand better the pros/cons and limitations/precautions if intending to pursue any of the below propositions.

Paying

The first thing that comes to mind is to buy a new machine with NVIDIA card. However, external GPUs exist and may be compatible to operate with your current machine. The best case is when your machine supports Thunderbolt, then the eGPU is just a matter of "plug-and-play". Some hacks are available in the Internet that shows how to connect an eGPU to a laptop through the Wi-Fi slot.

Figure 5 - External GPU box connected to a MacBook [4]

Other paying options is to use online cloud services like Amazon Web Services and Microsoft Azure where you create and connect to GPU-enabled virtual machines.

Almost Free

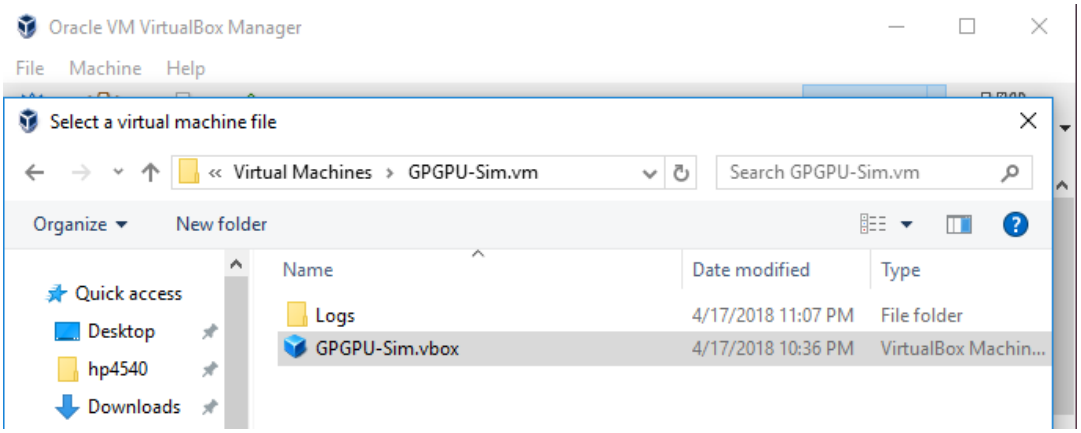
Certain cloud providers such as Google offer some free credits upon subscription which can cover several hours of usage. For more details, refer to [5].

Free

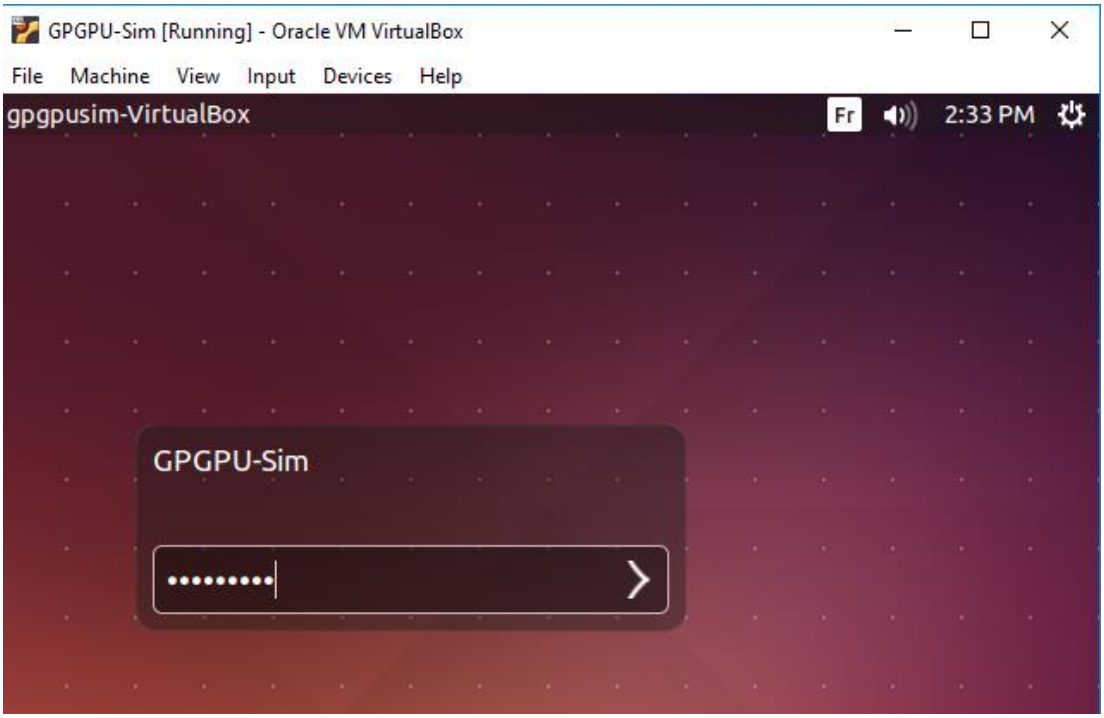
We describe here a no-cost option that is basically running a GPU simulation model. Programs will take longer time to complete (even long than a serial version of the program) but it should do the job as this project was tested and run successfully using this method.

1. Download VirtualBoxhttps://www.virtualbox.org/
2. Download the GPGPU-SIM virtual machine from http://www.gpgpu-sim.org/
3. Unzip the downloaded file gpgpu-sim.vm.tar.gz somewhere

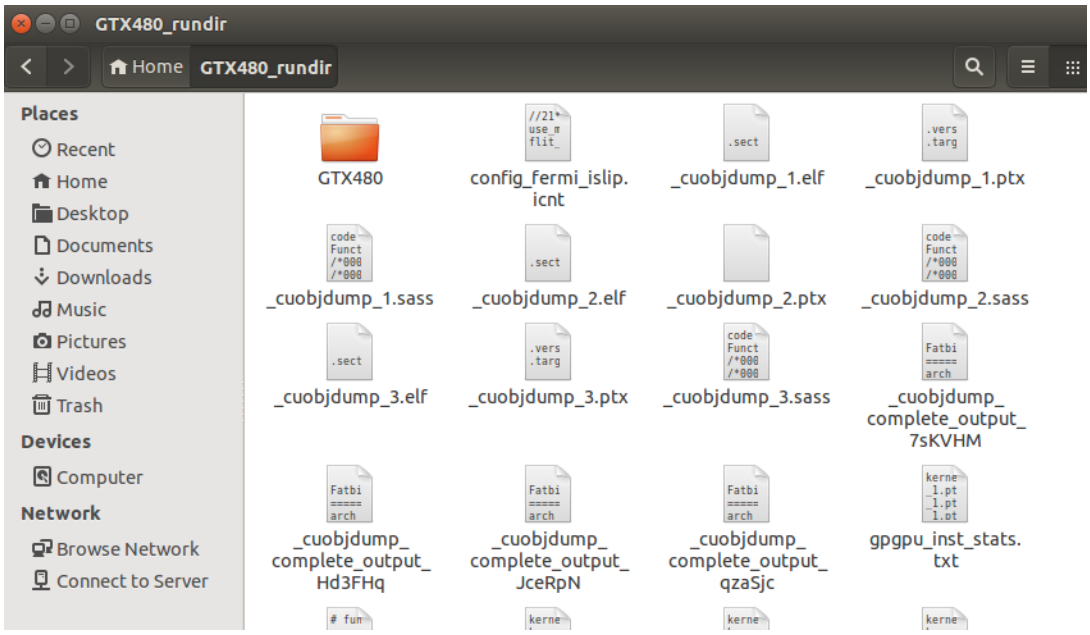
4. Open VirtualBox
5. In the top menu bar, click “Machine” and then “Add”, browse to the virtual machine folder and select GPGPU-Sim.vbox and click “Open”



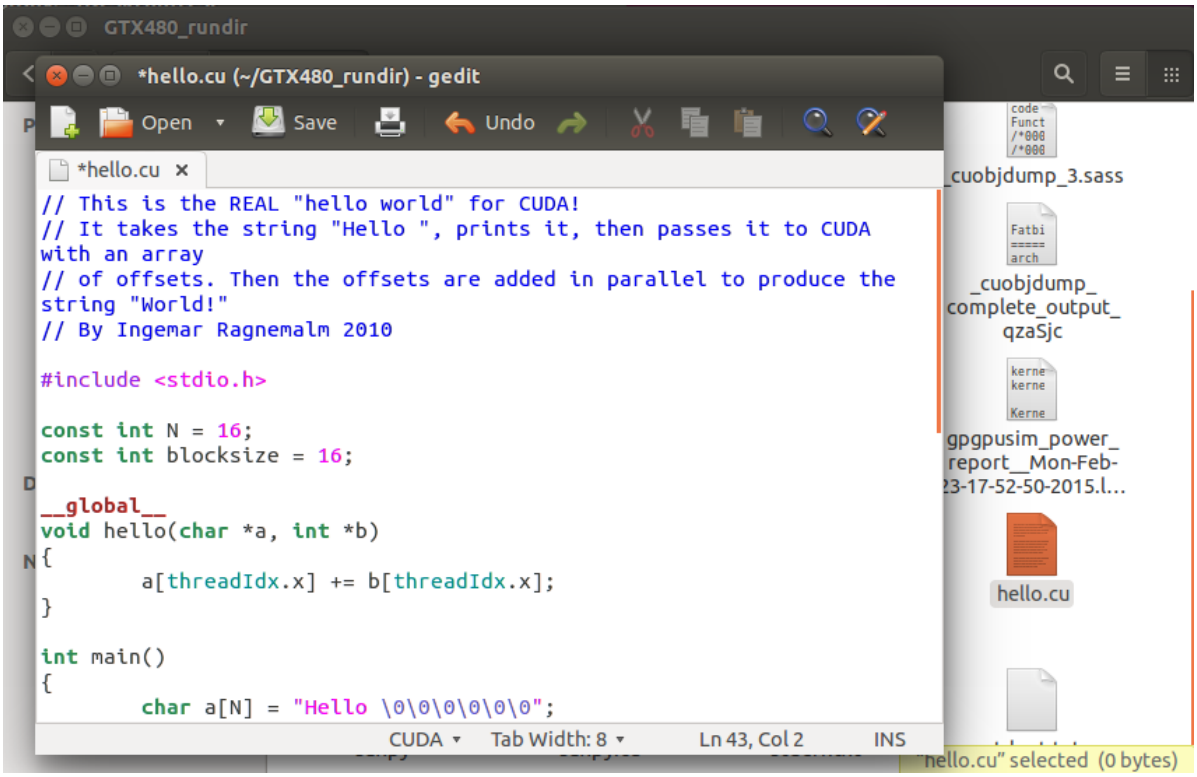
6. The virtual machine launches. Password for the account is: gpgpu-sim



7. Browse to the folder GTX480_rundir under Home

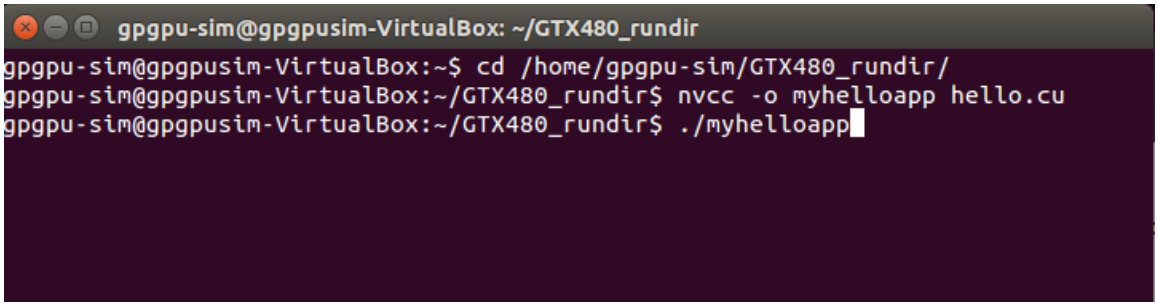


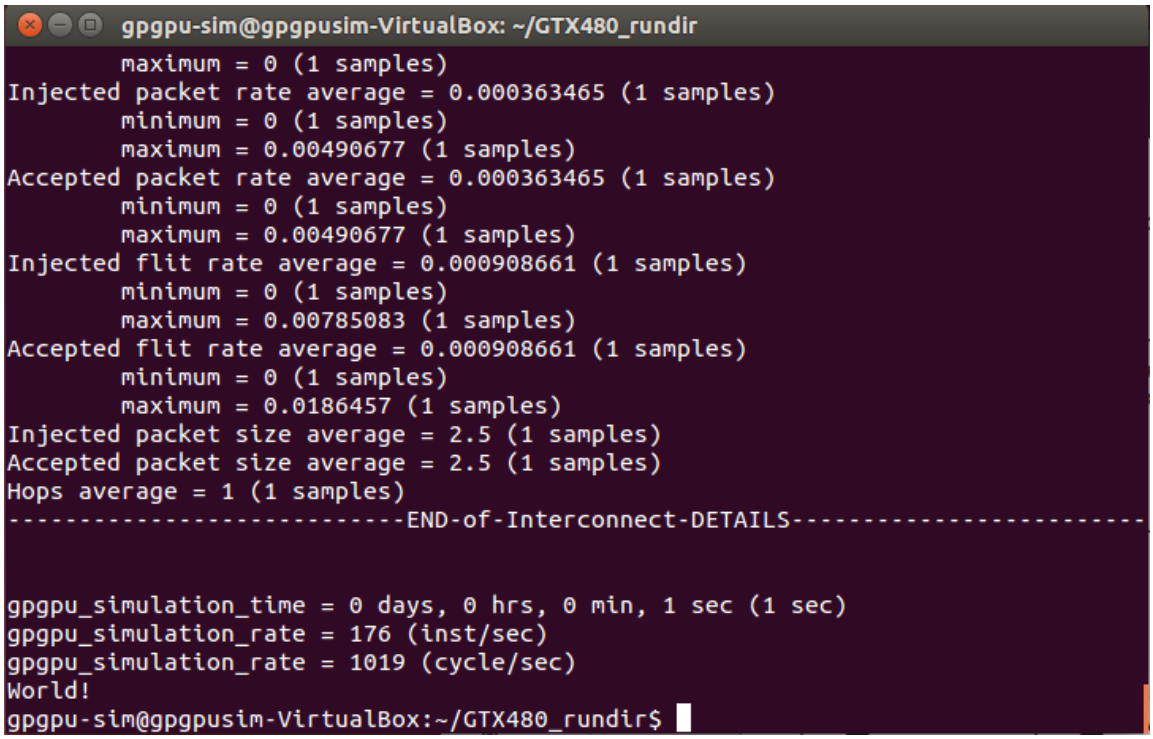
8. Create a new document hello.cu (Right Click > New Document > Empty Document)
9. Edit hello.cu (Right Click on hello.cu > Open With gedit) and write a simple CUDA program, then save



10. Open a terminal window
11. Change directory to the code's file location
    cd /home/gpgpu-sim/GTX480_rundir/

12. Compile with CUDA
    nvcc -o myhelloapp hello.cu
13. Run the program
    ./myhelloapp





References

[1] Liubov A Flores, Vicent Vidal, Patricia Mayo, Francisco Rodenas, and Gumersindo Verd_u. Ct image reconstruction based on gpus. Procedia Computer Science, 18:1412{1420, 2013.

[2] Drew Maier. GPUCT: A GPU Accelerated CT Reconstruction System. PhD thesis, BS Thesis University of Virginia, 2007.

[3] https://gray.mgh.harvard.edu/attachments/article/166/166_HST_S14_lect1_v2.pdf

[4] https://github.com/ali-yar/maia-parallelprocessing

[5] https://bizon-tech.com/

[6] https://medium.com/@jamsawamsa/running-a-google-cloud-gpu-for-fast-ai-for-free-5f89c707bae6