# GPU-Based Digital Hologram Reconstruction and Particle Detection

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Daniel Benjamin Taylor

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

John Sartori

December, 2014

© Daniel Benjamin Taylor 2014
ALL RIGHTS RESERVED

# Acknowledgements

I would like to acknowledge a number of people for their contributions to my experience and success in graduate school:

Professors John Sartori (Electrical Engineering) and Jiarong Hong (Mechanical Engineering), for both giving me the opportunity to work on this project and for the expertise required to complete it.

Mostafa Toloui, for providing a sound basis for this research and for his help in developing the algorithms presented.

Professor Brent Hecht (Computer Science), for pushing me to do more than I wanted to and for providing feedback and insight of great value.

# Dedication

I would like to dedicate this work to my family, for providing the means and motivation to do great things:

My parents, for letting me drill holes in the walls, build siege engines in the basement, get aluminum chips everywhere, and generally encouraging doing things of real value.

My eldest brother, for setting a standard to live up to.

And my #1 fan, for keeping me from slacking off.

## Abstract

Digital holograms, when combined with tracer particles, can be used for examining otherwise-invisible fluid flows. These holograms can be captured with standard digital imaging equipment, however processing them to extract tracer or particle locations is computationally expensive. Exacerbating the issue is that hundreds or thousands of holograms must be reconstructed to analyze a single flow.

Presented here is a hologram reconstruction and particle extraction system exploiting the massive parallelism of graphics processing units (GPUs) to reduce the time required to locate the particles in 3D space by orders of magnitude. This system requires no expensive proprietary hardware and runs on standard computers, with the only special requirement being an off-the-shelf Nvidia GPU.

# Contents

**Acknowledgements** i

**Dedication** ii

**Abstract** iii

**List of Tables** vi

**List of Figures** vii

**1 Introduction** 1

**2 Background and Motivation** 4

2.1 Hologram Reconstruction . . . . . . . . . . . . . . . . . . . . . . . . . . 4

2.2 CUDA architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

2.2.1 Host and device code . . . . . . . . . . . . . . . . . . . . . . . . 7

2.2.2 Kernels, threads, warps, blocks, and grids . . . . . . . . . . . . . 7

2.2.3 CUDA memory model . . . . . . . . . . . . . . . . . . . . . . . . 8

**3 Related Work** 10

**4 Implementation** 13

4.1 Stage 1: Reconstruction . . . . . . . . . . . . . . . . . . . . . . . . . . 13

4.2 Stage 2-3: Thresholding . . . . . . . . . . . . . . . . . . . . . . . . . . 15

4.2.1 Tenengrad map formation . . . . . . . . . . . . . . . . . . . . . . 16

4.2.2 Combined image, max tenengrad . . . . . . . . . . . . . . . . . . 16

4.2.3 Threshold calculation . . . . . . . . . . 18
4.3 Stage 4-6: Centroid Calculation . . . . . . . . . . 18
4.3.1 2D CCL . . . . . . . . . . 18
4.3.2 3D connected-component labeling (CCL) . . . . . . . . . . 22
4.4 Host code . . . . . . . . . . 24

**5 Results and Discussion** **28**
5.1 Test setup . . . . . . . . . . 28
5.2 Figures of merit . . . . . . . . . . 29
5.3 Test results . . . . . . . . . . 29
5.4 Discussion . . . . . . . . . . 31
5.4.1 Detection Rate and False-Positives . . . . . . . . . . 31
5.4.2 XY Error . . . . . . . . . . 32
5.4.3 Z Error . . . . . . . . . . 32
5.4.4 Execution Time . . . . . . . . . . 32

**6 Conclusion** **35**

**References** **36**

**Appendix A. Glossary and Acronyms** **38**
A.1 Glossary . . . . . . . . . . 38
A.2 Acronyms . . . . . . . . . . 39

# List of Tables

5.1 3 $\mu m$ performance . . . . . 30
A.1 Acronyms . . . . . 39

# List of Figures


2.1 Example reconstructed planes . . . . . . . . . . . . . . . . . . . . . . . . 6
2.2 CUDA thread organization . . . . . . . . . . . . . . . . . . . . . . . . 8
2.3 CUDA memory organization . . . . . . . . . . . . . . . . . . . . . . . 9
4.1 Pseudo-code: Stage 1 zKernel . . . . . . . . . . . . . . . . . . . . . . . 14
4.2 Pseudo-code: Stage 1 memcpyKernel . . . . . . . . . . . . . . . . . . . 14
4.3 Pseudo-code: Stage 1 operation . . . . . . . . . . . . . . . . . . . . . . 15
4.4 Pseudo-code: Tenengrad calculation . . . . . . . . . . . . . . . . . . . 15
4.5 Pseudo-code: Gradient . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
4.6 Pseudo-code: Tenengrad map calculation . . . . . . . . . . . . . . . . . 17
4.7 Pseudo-code: Combined image, max tenengrad . . . . . . . . . . . . . . 17
4.8 Pseudo-code: Threshold calculation . . . . . . . . . . . . . . . . . . . . 19
4.9 Particle YZ cross-section . . . . . . . . . . . . . . . . . . . . . . . . . . 20
4.10 Pseudo-code: 2D CCL, host code . . . . . . . . . . . . . . . . . . . . . 20
4.11 Pseudo-code: 2D CCL, device code . . . . . . . . . . . . . . . . . . . . 21
4.12 Pseudo-code: Centroid $x, y$ approximation . . . . . . . . . . . . . . . 22
4.13 Pseudo-code: 2D CCL centroid formation . . . . . . . . . . . . . . . . . 23
4.14 Pseudo-code: 3D CCL, device code . . . . . . . . . . . . . . . . . . . . 24
4.15 Pseudo-code: 3D CCL centroid formation . . . . . . . . . . . . . . . . . 25
4.16 Pseudo-code: Centroid merging . . . . . . . . . . . . . . . . . . . . . . 26
4.17 Pseudo-code: False-positive removal . . . . . . . . . . . . . . . . . . . 26
4.18 Pseudo-code: Host code overall operation . . . . . . . . . . . . . . . . 27
5.1 CUDA execution time by stage . . . . . . . . . . . . . . . . . . . . . . 31
5.2 MATLAB and CUDA Implementation $Z$ Error . . . . . . . . . . . . . . 33
5.3 MATLAB and CUDA Implementation Execution Times . . . . . . . . . 34

# Chapter 1

# Introduction

In recent years, GPUs have become an important tool for high-powered computing, especially as the performance gains of single-threaded processors slows [1]. GPUs overcome this limitation by providing immense computational capabilities through massively parallel processing of data, using the so-called SIMD (single instruction, multiple data) architecture, in which identical instructions are carried out upon differing pieces of data in parallel.

This architecture, while powerful, only lends itself well to a subset of problems. These problems typically exhibit two characteristics: independence of data, and low data transfers.

In order to reap a benefit from parallel processing, there must be work available for each processor. Forced serialization- where calculations on one piece of data require a previous piece of data to have been processed- can quickly negate the benefits of having multiple processors available. In the most extreme case, where no calculation can be performed before the previous one has completed, no benefit can be had from parallel processing, as only one processor could be active at a time. Thus workloads with a large number amount of data with no interdependencies will maximally benefit from parallelization.

Ken Batcher is often attributed with saying *"A supercomputer is a device for turning compute-bound problems into I/O-bound problems."* GPUs exhibit similar behavior. Because they can process large amounts of data quickly, getting data into and out of the GPU can quickly the performance bottleneck instead of computational capabilities.

The ideal GPU-computing problem is computationally intensive, but requires relatively small datasets as input, and generates relatively little data as output.

The problem considered in this paper is that of digital hologram reconstruction and particle detection, especially as applicable to tracer-particle detection for identifying fluid flows. It exhibits both of the characteristics noted above. Firstly, holograms can be reconstructed by using the Fresnel-Kirchhoff Diffraction Formula for each pixel of a given plane, independent of other pixels. Similarly, the resulting reconstructed images can be processed pixel-by-pixel to identify particles, operations which can be performed independently and summed together at the end.

Additionally, hologram reconstruction and particle detection requires minimal data input and produces minimal output. The only input required is a single image that makes up the hologram, and the output produced consists of a list of particles' $x, y, z$ locations.

Because of these characteristics, hologram reconstruction and particle extraction is theoretically an ideal problem for GPU acceleration. This paper shows this to be true in practice, with runtimes orders of magnitude lower than a traditional CPU implementation.

This paper makes two main contributions to the existing GPU-accelerated holography work:

- An approach to hologram reconstruction and particle extraction that runs entirely on the GPU, excepting some minor CPU-based post-processing
- Massive improvements in reconstruction speed with accuracy comparable to existing sequential techniques, achieved through parallel adaptations of the sequential algorithms

The remainder of the paper is organized as follows:

- Chapter 2 gives an overview of the problem, as well as the motivation for this work
- Chapter 3 covers related work in the field
- Chapter 4 describes in detail how the system was implemented in a parallel manner using the CUDA GPU-computing toolkit

- Chapter 5 presents and discusses the speed and accuracy of the implementation, both in absolute terms and as compared to a single-threaded reference implementation
- Chapter 6 provides a summary of the work presented and comments on future work.

# Chapter 2

# Background and Motivation

This chapter seeks to provide an overview of how hologram reconstruction works and why it's a good match for a massively parallel SIMD architecture. It further outlines the basic workings of CUDA, required to follow the details of the implementation presented in chapter 4.

## 2.1 Hologram Reconstruction

The problem considered in this paper is that of reconstructing digital holograms and extracting particle locations from those holograms. The general approach for doing so is as follows:

1. **Obtain a hologram:** This is outside the scope of this paper, but a hologram is generally generated by illuminating the area of interest with a laser. Any particles in the area will create refraction patterns, and a camera is used to record those into a hologram. Thus a hologram consists of a single monochrome image containing the sum of the refraction patterns from all of the particles in the area of interest.

   Simulated holograms can also be generated (including on GPUs, as in [2], [3]).

2. **Reconstruct focused depth planes:** Using the Fresnel-Kirchhoff Diffraction Formula, it's possible to re-create how the diffraction patterns would appear in a plane some distance from the camera along the direction of the light's propagation. By doing so for a multitude of distances from the camera, an approximation of

the diffraction patterns in 3D space can be created. An example of the results of this process can be seen in figure 2.1.

3. **Threshold particles:** In order to locate particles in space, they must somehow be separated from the background. Because the pixels contributing to particles have a different intensity than those forming the background, a single value can be used to separate the two. This step is used to find the value that maximizes the particle-contributing pixels detected while still ignoring the background.

4. **Calculate centroids:** Once the particles have been separated from the background, their location in 3D space must be calculated. This is done by scanning across the reconstructed planes and connecting pixels that are adjacent (and fall within the threshold) into individual centroids, each representing a particle. The average (x,y,z) location of all of the pixels belonging to each centroid is used to calculate the location of that particle.

5. **Post-processing:** Centroids may not be detected perfectly. For example, the refraction patterns may constructively interfere to create pixels within the particle threshold, leading to the detection of a "particle" that doesn't exist. Similarly, actual particles may register as multiple particles due to gaps in their thresholded pixels. Because of this, post-processing of the detected centroids is required to filter out these false-postives and to merge parts of the same centroid into a single particle.

On a standard CPU, these steps must all be done sequentially. Ignoring the acquisition step, the CPU must first iteratively reconstruct each focused Z plane. From there, it must calculate the ideal threshold value, then use that threshold to identify each centroid and add up the pixels that comprise it. Finally, it must perform any post-processing on the centroids calculated. As shown in chapter 5, this is a lengthy process, even with modern high-end CPUs. Analysis of a turbulent fluid flow may require processing hundreds or thousands of holograms, greatly exacerbating the issue.

All but one of these tasks has a large amount of independent work- each Z plane that must be reconstructed has no dependence on any other Z plane, and thus each could be done in parallel. Planes can similarly be thresholded at the same time. Finally,

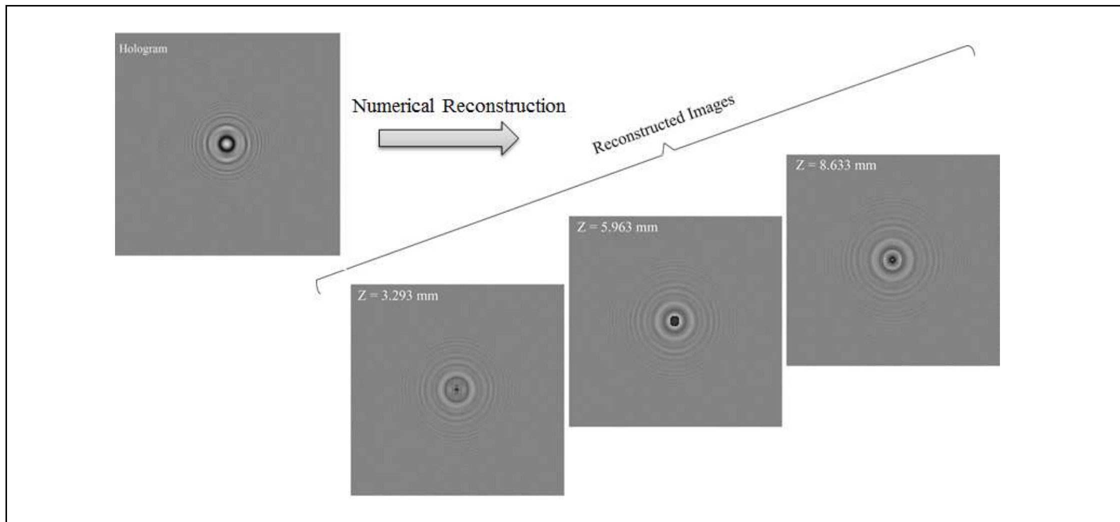


**Figure 2.1:** Example reconstructed planes from a simulated hologram (upper left). The planes are reconstructed at (from lower-left to upper-right) 3.293, 5.963, and 8.633 mm from the camera long the $z$ axis. The black spots indicate the particle being in focus at that depth.

identifying centroids is a largely independent task, as each centroid can be calculated at the same time as the others.

Post-processing is the only step that's largely sequential, due in large part to the fact that pieces of particles may be merged together during the process. This creates data dependencies that severely limit the applicability of parallelism.

For the other steps, however, the parallelism isn't limited to just the planes or the particles. Multiple threads can work together to reconstruct a single plane, as the pixels within a plane are *also* independent of one another. Similarly, multiple threads can count the pixels comprising a single centroid, then merge their sums at the end to get the average location. Because the number of potential parallel tasks is so large, this problem benefits enormously from the massive amount of parallelism available in modern GPUs.

## 2.2 CUDA architecture

For the parallel implementation of hologram reconstruction, CUDA was used. CUDA is Nvidia's GPU computing toolkit, enabling general computation to be done on their

GPUs. Discussed in this section is an overview of CUDA's execution and memory models, intended to provide a background for understanding the implementation presented in chapter 4.

### 2.2.1 Host and device code

Each CUDA program consists of two parts: the host code, and the device code. The host code runs on the CPU of the computer and is, for our intents and purposes, identical to any other C program. It can use all of the usual libraries, constructs, etc. that any other program has access to, and (generally) runs purely sequentially. It additionally has the job of controlling the GPU, which mostly entails moving data into or out of the GPU's memory and commanding the GPU to perform computations.

The device code executes on the GPU itself, and consists of kernels and other device helper functions. A kernel is basically the unit of execution in CUDA- a function that is executed in parallel by a number of threads. When a kernel is launched by the host code, the host specifies the block size and the grid size of the kernel launch, in addition to the parameters passed to the function itself.

### 2.2.2 Kernels, threads, warps, blocks, and grids

A CUDA kernel is essentially a function executed on the GPU. A kernel is launched with a specific grid. This grid is is comprised of thread blocks, each of which contains a number of threads. Grids and blocks may be 1-, 2-, or 3-dimensional. Figure 2.2 illustrates this with a 2D grid containing 2D blocks. The blocks in a grid execute largely independently, while constructs exist for threads within the same block to synchronize and quickly share access to data.

Threads in CUDA are further grouped into warps. A warp is a group of threads that is executed in strict lockstep- each thread in a warp will execute the same instruction at the same time. Unlike grids, blocks, and threads, warps are handled internally by the GPU, and not specified by the programmer.

When a kernel is launched, it has a specified block and grid size. Each thread can identify its location within the block by the parameters threadIdx.x, threadIdx.y, and threadIdx.z. Position within in the grid can similarly be obtained with the blockIdx

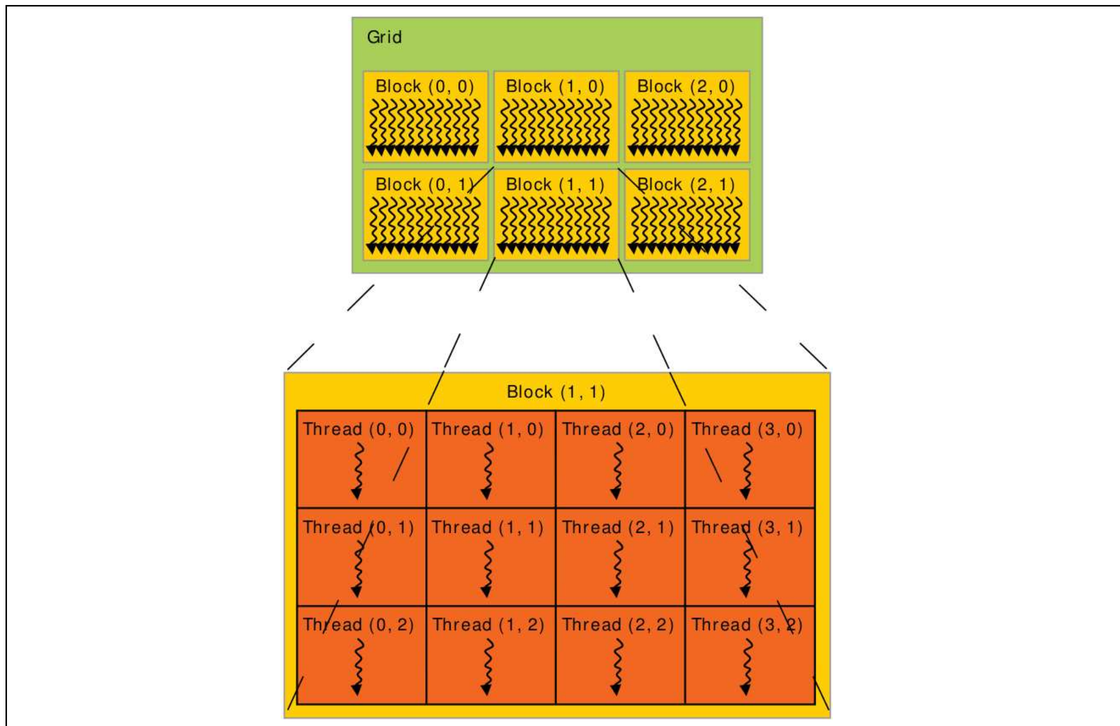


**Figure 2.2:** Visualization of CUDA thread organization (figure 6 of the CUDA C Programming Guide [4])

structure. Using this information, each thread can select a separate piece of data to compute on.

### 2.2.3 CUDA memory model

In addition to the parallel computation model, CUDA has a memory model that differs slightly from standard CPU. CUDA still has registers and main memory (RAM, referred to as "global memory" in CUDA parlance) on the GPU, as a standard CPU does, as well as cache. There are two important distinctions from a standard CPU, however.

Similar to a traditional system, memory accesses in CUDA are significantly slower than the processor frequency. In CUDA, however, many threads are trying to access memory at once, as opposed to (ideally) a single thread on a CPU. Because threads execute in a SIMD fashion, all of the threads in a warp *must* complete their memory access before *any* of them can continue. Thus a single memory access can have the same effect on speed as many consecutive accesses on a CPU.

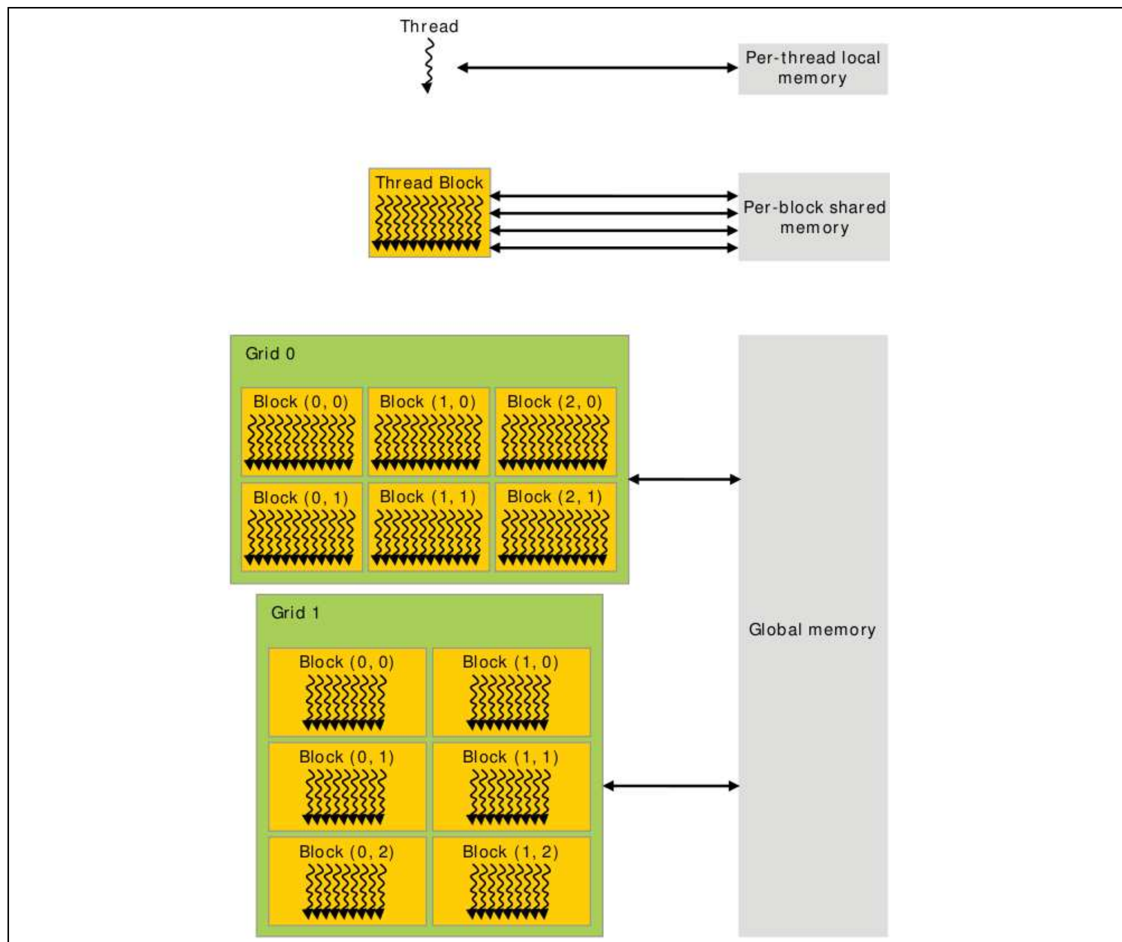


**Figure 2.3:** Visualization of CUDA memory organization (figure 7 of the CUDA C Programming Guide [4])

In part to work around that, CUDA implements a split implicit/explicit cache, with the explicit cache being referred to as ”shared memory”. As its name implies, shared memory is shared amongst a block of threads, and may be used to either share data among them, or to cache data that each thread would otherwise individually fetch from main memory. Because CUDA can rapidly become IO-bound, this can be a great advantage, as an access to shared memory is significantly faster than global memory [4].

This memory configuration is shown in figure 2.3.

# Chapter 3

# Related Work

Digital holography work exists outside of the realm of GPUs; many of the ideas are re-used here.

[5], [6] present techniques for reconstruction very similar to those utilized in this paper.

[7] utilizes a component segmentation approach like that used in the 3D CCL approach used here.

[8] presents the hybrid tenegrad approach for improved depth accuracy in processing holograms, also utilized in this work.

Similarly, using GPUs in digital holography is similarly not a new idea. A body of work exists to either render images from captured holograms or to generate holograms in various capacities, using GPUs for acceleration.

[9] shows work to render detailed images from captured holograms. CUDA was used to greatly reduce the reconstruction time when compared to a CPU implementation, and a speedup of greater than 100 times was shown. Similar to the work presented here, cuFFT was used for FFT operations.

[10] similarly examines capturing and manipulating holograms using CUDA. They further extend the work to high-resolution (4k and 8x) imagery and real-time rendering of the captured scenes in 3D. The work focuses on the detailed rendering 3D imagery, as opposed to extracting information from these images.

[3] further explores reconstructing images, and also presents a technique for creating them in the first place. While GPUs are used in the work, it was based around OpenCL

and an AMD GPU, not Nvidia's CUDA. Again, no attempt was made to detect particles.

[2] similarly demonstrates generating holograms, but further shows the utilization of multiple GPUs to do so. A speedup of approximately 100 times over a CPU implementation was shown using a single GPU, with an additional factor of 2.5 being achieved when multiple GPUs were used.

[11] uses holograms to manipulate tiny particles using laser beams. The work presented uses GPUs to solve for the phase modulation required to deliver energy from a laser array to the target focal point.

Prior efforts to utilize parallel processing not only to recreate or generate holograms, but also to extract information from them also exist.

[12] presents an implementation of reconstructing digital holograms in real-time (24 frames per second), using 512-pixel-square holograms. This was implemented using CUDA and the cuFFT library, and included parallel capture (with the CPU) and processing (with the GPU) of frames. No attempt was made, however, at detecting particles in the reconstructed images. Furthermore, each frame reconstructed is similar in work to a single plane reconstructed in this work presented in this paper, and thus the performance presented here is significantly higher. This is likely due to advances in GPU technology, as well as improvements in the cuFFT library.

[13] presents a parallel particle-detection approach using computer clusters. The technique presented is shown to function over a range of particle sizes by using different algorithms based on the particle size. The algorithms generally utilize peak-detection, with the value being detected being changed based on particle size (possibilities presented include pixel intensity and Sobel gradient).

The implementation presented, in addition to not being on a GPU, is also done in an "embarrassingly parallel fashion", where a single core handles a single hologram. Thus a speedup in the processing of multiple holograms is demonstrated, but not for any single hologram.

[14] also demonstrates hologram particle extraction, but uses a different technique and GPUs. Planes are reconstructed, then approximate particle locations are detected based on the focus at a given plane. The most intense particles are then segmented, with their refraction patterns removed from the planes to minimize noise for consecutive particle extractions

Despite the use of GPUs, only a 10x speedup over a CPU implementation was found. Additionally, some processing was found to require either double-precision calculations on the GPU or additional CPU processing.

No hard performance numbers were given to enable a performance comparison.

Finally, [15] presents work similar to that presented in this paper, showing particle extraction from holograms (and even particle tracking, albeit using existing libraries), implemented in the interpreted IDL programming language. A different approach is taken to find particles' locations as well, approximating the $x, y$ centers by taking multiple gradients along the refraction patterns, then applying a best-fit approach to match the patterns to predictions made by the Lorenz-Mie theory to find $z$.

Particle extraction was done at 8 frames per second with images 200 pixels square, using a combination of multi-threaded CPU processing and a GPU. The GPU-CPU hybrid implementation was found to be 20x faster than that of the CPU-only version.

The work presented in this paper improves upon this performance by using CUDA directly from a compiled C program. Additionally, it shows an implementation almost entirely on the GPU, maximizing the benefits of parallelism and identifying particles in half the computation time, adjusted for image size. Finally, this paper presents an alternative method for locating particles in space.

# Chapter 4

# Implementation

This chapter describes the implementation of the hologram reconstruction algorithm in CUDA. The implementation is broken up into "stages", roughly corresponding to the steps outlined in chapter 2. One section is devoted to each stage, including an overview of what functionality the stage provides, as well as the implementation thereof and a discussion of why each stage was implemented in the manner indicated.

The numbering irregularities of the stages should also be addressed. A reference implementation in MATLAB (developed by Mostafa Toloui and Jiarong Hong, currently unpublished) was used as a guide for the CUDA implementation. The MATLAB implementation consisted of six discrete steps in extracting particle centroids. Due to architectural differences between CUDA and MATLAB, it was efficient to combine some stages into a single stage in CUDA. Thus the numbering of CUDA stages reflects the MATLAB stage functionality that each CUDA stage implements.

## 4.1 Stage 1: Reconstruction

Reconstructing Z planes- recreating the refraction patterns at a distance from the camera- is performed in as straightforward a manner as possible, using previously-established techniques [5], [6].

First, the hologram is converted into the frequency domain using a FFT. CUDA provides a fast, parallel implementation of this in their cuFFT package. To avoid reinventing the wheel, and to leverage the knowledge the Nvidia developers have of

their product, this was used for all FFT operations.

Once the hologram is in the frequency domain, three steps are performed for each plane to be reconstructed. First, a kernel is launched to calculate the value of each pixel. This calculation is shown in figure 4.1.

**Require:** $w, h$, the width and height of the hologram
**Require:** $hz$, the complex output array
**Require:** $x, y, z$, the location to reconstruct
**Require:** $\lambda$, the wavelength of illuminating light
**Require:** $reso$, the image resolution in micrometers per pixel
**Require:** $fftHolo$, the complex hologram array in the frequency domain
$f_x = (x + \frac{w}{2}) \text{ mod } w + \frac{1}{2} - \frac{1}{w}$
$f_y = (y + \frac{h}{2}) \text{ mod } h + \frac{1}{2} - \frac{1}{h}$
$\phi = \frac{\pi\lambda}{reso^2}$
$\theta = z\phi(f_x^2 + f_y^2)$
$hz(x, y) = fftHolo(x, y)\angle\theta$

**Figure 4.1:** Stage 1 calculation of pixel values in the complex frequency domain.

Because each pixel's value is independent of the adjacent pixels' values, this calculation can be done highly efficiently in parallel; each thread performs the calculation shown in figure 4.1 on its own $x, y$ location in parallel. The host code iterates over $z$, as seen in figure 4.3.

Once each pixel in the plane has been calculated, an inverse FFT is performed (again using cuFFT) to find the complex value of the refractions at that plane.

Finally, a third kernel is launched to calculate the magnitude of the refraction at each pixel and copy the result into the destination array for use by later kernels. Again, each value is independent of the others, so calculations can be done efficiently. This operation is shown in figure 4.2.

**Require:** $planeIn$, the 3D complex-valued inverse FFT data array
**Require:** $planeOut$, the 2D real-valued output data array
**Require:** $x, y, z$, the pixel location to copy
$planeOut(x, y) = \text{abs}(planeIn(x, y, z)$

**Figure 4.2:** Device algorithm for copying the inverse FFT results into the output array

It should be noted that this approach appears to introduce an additional memory copy, which were previously noted to be expensive. Because of the way cuFFT handles

complex-to-real transforms, additional memory juggling would be required even if this approach was used. As such, the more human-readable approach was taken.

The overall stage 1 code is shown in figure 4.3.

```
Require: holoIn, the real-valued input hologram
  fftHolo = cuFFT(holoIn, forward)
  for all z do
    Launch zKernel (figure 4.1)
    Perform inverse FFT on the result using cuFFT
    Launch memcpyKernel (figure 4.2)
  end for
```

**Figure 4.3:** Algorithm for completing the reconstruction step.

## 4.2 Stage 2-3: Thresholding

In order to detect a threshold for separating the particles from the background, some metric for evaluating the quality of any given threshold is required. The metric chosen was the sharpness of the minimum-intensity image- that is, the image generated by condensing all of the reconstructed planes into a single image by taking the minimum pixel intensity value at each $x, y$ location. The sharpness of a given configuration was calculated as the average tenengrad value of pixels with non-zero tenengrads, where the tenengrad of a pixel at position $x, y$ is shown in figure 4.4. This approach was shown by [8] to be highly accurate in detecting the locations of particles along the axis of reconstruction.

```
Require: pixel(x, y) is the magnitude of the pixel at location x, y of the input
Require: pixel(x ± [0,1], y ± [0,1]) = pixel(x, y) when x ± [0,1], y ± [0,1] is out of
  bounds
  l = (pixel(x − 1, y − 1) + 2pixel(x − 1, y) + pixel(x − 1, y + 1))/8
  r = −(pixel(x + 1, y − 1) + 2pixel(x + 1, y) + pixel(x + 1, y + 1))/8
  t = (pixel(x − 1, y − 1) + 2pixel(x, y − 1) + pixel(x + 1, y − 1))/8
  b = −(pixel(x − 1, y + 1) + 2pixel(x, y + 1) + pixel(x + 1, y + 1))/8
  return (l + r)^2 + (t + b)^2
```

**Figure 4.4:** Pseudo-code for calculating the tenengrad of a pixel at location $x, y$.

In order to accomplish this, a large number of steps were used. It seems likely that

they could be simplified somewhat, though that's left as an area of future research. As an alternative, an option was provided in the code to manually enter a pixel threshold based on prior knowledge of the holograms, bypassing these lengthy calculations.

### 4.2.1 Tenengrad map formation

First, the tenengrad of each pixel in each reconstructed plane is calculated, along with the average tenengrad of each of those planes. This is accomplished with a single CUDA kernel consisting of a single block for each reconstructed Z plane. The threads iterate over the plane, calculating tenengrads for individual pixels as they do so, also keeping track of the total calculated tenengrad.

Once the tenengrad for each pixel of a plane has been calculated, the block utilizes shared memory to find the sum of all tenengrads in the plane. From this the average tenengrad is found for the plane.

With the average tenengrad known, the threads re-iterate over each pixel in the plane, this time calculating the gradient (figure 4.5). For pixels with gradients greater than the average, the gradient is stored into the tenengrad map, while a zero value is stored for those under the average. The result of this kernel is the tenengrad map for the plane- an image roughly corresponding to the edges detected in the plane, weighted by the sharpness of the edge.

This procedure is shown in figure 4.6

**Require:** $x, y$, the location to calculate the gradient for
  $gradLR$ =abs($pix(x-1,y) - pix(x+1,y)$)
  $gradUD$ =abs($pix(x,y-1) - pix(x,y+1)$)
  **return** $0.5\times$max($gradLR, gradUD$)

**Figure 4.5:** Gradient calculation function. For this application, the magnitude of the greater of the vertical and horizontal gradients is taken. The result is divided by two, as the calculated gradient is taken across two pixels.

### 4.2.2 Combined image, max tenengrad

Next, a second kernel calculates two additional images: the combined image, and the maximum tenengrad image. The former consists of the minimum-intensity (lowest-valued) pixel in each $x, y$ location for any depth $z$, while the latter holds the highest

**Require:** $w, h$ are the width and height of the hologram
**Require:** *tenengrad_map* is an output array with dimensions equal to $w, h$
$x = threadIdx.x$;
$y = threadIdx.y$;
$xstep = blockDim.x$;
$ystep = blockDim.y$;
**for all** $i, px = x + i \times xstep < w$ **do**
**for all** $j, py = y + j \times ystep < h$ **do**
$tenengrad_map(px, py) =$tenengrad$(px, py)$
**end for**
**end for**

**Figure 4.6:** Pseudo-code for calculating the tenengrad map of an image. This would be run by each thread in a block, with one block per reconstructed plane

tenengrad value at each $x, y$ location. This kernel is configured as a 2D grid, where each thread handles a single pixel, and iterates along the z dimension. Taken together, these are used to form a heuristic for the sharpness at a given threshold value in the next step.

The calculation of the combined image and maximum tenengrad is shown in figure 4.7.

**Require:** $tenengrads[z](x, y)$ corresponds to the tenengrad calculated for pixel $x, y$ on plane $z$
**Require:** $pixel[z](x, y)$ corresponds to the value of the pixel $x, y$ of the reconstructed plane $z$
**Require:** *combined* and *max_tenengrad* are output arrays
$x = blockIdx.x \times blockDim.x + threadIdx.x$;
$y = blockIdx.y \times blockDim.y + threadIdx.y$;
**for all** $z$ **do**
$combined(x, y) =$min$(pixel[z](x, y), combined(x, y))$
$max_tenengrad(x, y) =$max$(tenengrads[z](x, y), max_tenengrad(x, y))$
**end for**

**Figure 4.7:** Pseudo-code for calculating the combined image and maximum tenengrad. Each thread on the GPU runs this code, with one thread per pixel of the image.

### 4.2.3 Threshold calculation

The last kernel is used to calculate the actual best threshold. First, the combined image is masked with a potential threshold value- each pixel under the threshold is left as-is, while those above the threshold are set to 0. This seemingly-backwards approach is due to pixels corresponding to particles being darker than the background, and hence having a low pixel value. The resulting values are pairwise-multiplied by the maximum tenengrad image, and added to the total sharpness of the threshold.

Once the total sharpness for a threshold value has been calculated, the average is found by dividing by the number of non-zero elements that added to the total.

The possible threshold values are calculated in parallel by the GPU, one threshold value per block. From these values, the host selects the to be used in later calculations.

This algorithm is described in figure 4.8.

## 4.3 Stage 4-6: Centroid Calculation

The final GPU step of extracting particles from the hologram is calculating centroids. This involves taking the reconstructed planes, examining the pixels below the calculated threshold, deciding which belong to what particle, and finally calculating the average location of each particle based on the pixels that compose it.

An averaging approach is desirable because the particles become highly elongated along the axis of reconstruction (figure 4.9). Thus finding one point where the particle is most in focus is less than guaranteed to reflect the particle's true location.

Two approaches to solving this problem were tested, presented in the following sections.

### 4.3.1 2D CCL

This approach is significantly simpler than 3D CCL (section 4.3.2), and is based on the premise that with the combined image known in advance, the approximate $x, y$ locations of the particles in the image are also known. From there, the search space for particle centroids can be greatly reduced, resulting in a faster runtime.

This makes use of a few assumptions. Firstly, it assumes that particles are well-spaced- that is, that each particle will form a distinct shape in the combined image, and

**Require:** $w, h$ are the width and height of the hologram
**Require:** $threshold$ is the block's threshold to calculate on
**Require:** $tenengrad_map$ is the previously-calculated array of tenengrads
**Require:** $cmb_masked$ is the previously-calculated minimum-intensity image, but with any $cmb_masked(x, y) < threshold$ being 0
$x = threadIdx.x;$
$y = threadIdx.y;$
$xstep = blockDim.x;$
$ystep = blockDim.y;$
$sharpness = 0$
$nonzero = 0$
**for all** $i, px = x + i \times xstep < w$ **do**
  **for all** $j, py = y + j \times ystep < h$ **do**
    $val = tenengrad_map(px, py) \times \text{tenengrad}(cmb_masked(px, py))$
    $sharpness += \text{abs}(val)$
    **if** $val \neq 0$ **then**
      $nonzero += 1$
    **end if**
  **end for**
**end for**
$nonzero = \text{sum}(nonzero)$
$sharpness = \text{sum}(sharpness)$
**return** $nonzero, sharpness$

**Figure 4.8:** Threshold sharpness calculation GPU code. Each block calculates the sharpness for a different threshold; the host code than selects the threshold value with the highest sharpness of those returned. The "sum" function simply merges the partial results from all of the threads in the block into the final value.

not overlap with other nearby particles. This is a safe assumption for the use case of tracking fluid flows, as tracer particles that are that nearby can be treated as a single particle anyway. Furthermore, if they're separated sufficiently along the Z axis, they can be differentiated anyway.

Second, the particles are assumed to be of a uniform size. As explained later, the search space for each centroid is based upon the likely size of the particle, in this case calculated as the average of all particles present. While this simplifies the algorithm somewhat, it could likely be worked around in future work. For the purposes of tracking fluid flows, where tracers are of uniform size, this assumption is justified.

Finally, the particles are assumed to be roughly spherical, which is again the case

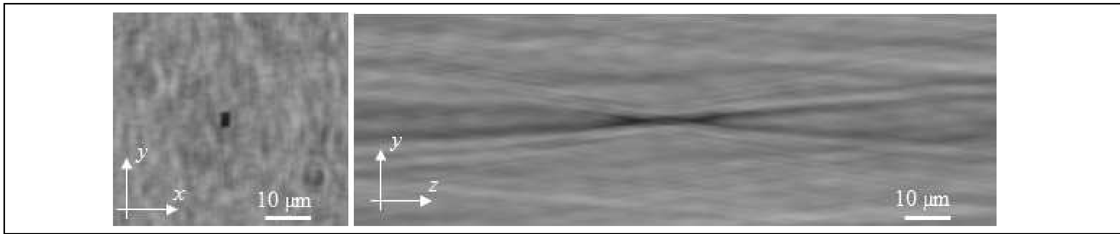


**Figure 4.9:** A spherical particle's cross-section along the axis of reconstruction.

with tracers. This is important for identifying the center of the search space for the centroid of each particle- an elongated particle may present an inaccurate $x, y$ center, causing the end result to be skewed.

### 2D CCL

Before performing localized searches for particles, the approximate locations of them must be known. The first part of this endeavor is performing a 2-dimensional CCL to form the groups of pixels denoting the centers of the particles.

Each pixel is initially labeled with either its row-major index ($w \times y + x$) if its combined-image value is less than the threshold, or -1 (unsigned, the largest possible unsigned value), if it's above.

From there, each CUDA thread repeatedly examines the pixels surrounding the pixel it corresponds to, and assigns its pixel the lowest-valued label of itself and any of its neighbors, as seen in figure 4.11. As soon as no thread changes its pixel's value for an iteration, the labels are globally consistent, and the components have been formed. This procedure is shown in figure 4.10.

```
Require: labels is the array containing the component labels for each pixel
  changed = true
  while changed do
    changed =launch_ccl_kernels(labels)
  end while
```

**Figure 4.10:** 2D CCL algorithm host code. It simply re-launches the kernels until none of them adjust the labels.

$x = blockIdx.x \times blockDim.x + threadIdx.x;$
$y = blockIdx.y \times blockDim.y + threadIdx.y;$
$globalChanged = false$
$changed = true$
**while** $changed$ **do**
  $changed = false$
  $oldlabel = pix(x, y)$
  $label(x, y) = \min(label(x, y), label(x-1, y), label(x+1, y), label(x, y-1), label(x, y+1))$
  **if** $label(x, y) \neq oldlabel$ **then**
    $globalChanged = true$
  **end if**
  $changed =$ __syncthreads_or($changed$)
**end while**
**return** $globalChanged$

**Figure 4.11:** 2D CCL algorithm executed on the GPU. The line $changed =$ __syncthreads_or($changed$) is used to keep individual blocks iterating until the block converges by instructing each block to re-iterate if any thread updated a pixel value.

#### Centroid $x, y$ approximation

Once the components have been labeled, the approximate $x, y$ positions of each particle must be found. This operation is identical to finding a centroid in 2D.

A global list of centroids is created, ordered by label. A unique center label can be identified by any pixel having a label matching its index.

Each thread then examines a single pixel. If that pixel has a valid label, it adds the $x, y$ values to the global total of the center with the corresponding label, as well as incrementing the pixel count for that center. Once all threads have finished this task, the totals are divided by the pixel counts for each center to find the average $x, y$ position for that particle. This pixel count information is further used to calculate the average size of the particles.

This procedure is shown in figure 4.12. The host code handles the task of dividing the $x, y$ sums by the number of pixels in the center to get the average $x, y$ location.

```
if label(x, y) is valid then
   center = findCenterByLabel(label(x, y))
   center.x+ = x
   center.y+ = y
   center.pixel_count+ = 1
end if
```

**Figure 4.12:** Algorithm for finding the centers of potential particles in the combined image.

#### Centroid formation

Once the approximate centers of the particles have been detected, forming the centroids is relatively straightforward. A kernel is launched with one block for each particle detected, with the block being sized proportionately to the average particle cross-section calculated previously. Each block is centered on one of the detected particles, and essentially iterates along the Z dimension, adding any pixels under the threshold to the total contributing to the location.

When two consecutive planes are found that contain no pixels under the threshold, the centroid is considered finished, and no further pixels are added to it. If a plane further on contains pixels below the threshold, a new centroid is formed.

This algorithm is shown in figure 4.13.

### 4.3.2 3D connected-component labeling (CCL)

The second approach considered is full 3D CCL. This attempts to create components in 3D space, instead of attempting to find just the $x, y$ centers. It uses much the same approach as the 2D CCL, but also propagates labels along the Z axis. This approach is largely identical to that detailed in [7].

#### 3D CCL

The first step is the actual CCL approach. As mentioned previously, this is largely the same as the 2D version (figure 4.11), but iterates over the Z dimension as well. At each iteration, it takes the lowest label of any pixel adjacent to that $x, y, z$ location, assuming the location has a label to begin with. The kernel is configured such that one thread handles a given $x, y$ location for all $z$. This is shown in figure 4.14. The host code is

**Require:** $x, y$, center location of this block
$cx = x - blockDim.x/2 + threadIdx.x$;
$cy = y - blockDim.y/2 + threadIdx.y$;
$centroid$ = empty centroid
**for all** $z$ **do**
 **if** No pixels below the threshold **then**
  **if** Was forming a centroid and this is the second such plane **then**
   merge($centroid$)
   globalSave($centroid$)
   $centroid$ = empty centroid
  **end if**
  Go to next iteration
 **end if**
 $centroid.x+ = cx$
 $centroid.y+ = cy$
 $centroid.z+ = z$
 $centroid.pixel_count+ = 1$
**end for**

**Figure 4.13:** Algorithm for finding centroids using 2D CCL. The function merge(*centroid*) combines the partial centroids being calculated by each thread in the block; globalSave(*centroid*) saves the calculated centroid to global memory for the host to access. The host code later divides the centroid's $x, y, z$ values by its pixel count to get the actual $x, y, z$ location of the centroid.

identical to the 2D CCL case (figure 4.10), except that the *labels* array is 3D instead of 2D.

Once this operation has completed, the *label* array contains elements either with value $INT_MAX$ (if they don't contain a particle pixel), or of an integer representing which centroid they belong to.

**Centroid formation**

Forming centroids using the 3D CCL approach is done by simply examining each pixel and, if it has a label corresponding to a centroid, adding that pixel's $x, y, z$ location to the centroid's total. After this has been completed, the host code divides the $x, y, z$ totals by the number of pixels that contributed to the centroid to get the average location.

The GPU code implements this in a straightforward manner. One thread is assigned to each $x, y$ location, and the kernel then iterates over $z$. When a thread finds a pixel

```
x = blockIdx.x × blockDim.x + threadIdx.x;
y = blockIdx.y × blockDim.y + threadIdx.y;
minlabel = INT_MAX
changed = true
globalchanged = false
while changed do
  changed = false
  for all z do
    if label(x, y, z) == INT_MAX then
      minlabel = INT_MAX
    else
      minlabel = min(minlabel, label(x, y, z), label(x + 1, y, z), label(x − 1, y, z), label(x, y + 1, z), label(x, y − 1, z))
      if minlabel ≠ label(x, y, z) then
        changed = true
        globalchanged = true
      end if
    end if
    label(x, y, z) = minlabel
  end for
  changed = __syncthreads_or(changed)
end while
return globalchanged
```

**Figure 4.14:** 3D CCL algorithm executed on the GPU. The line *changed* = __syncthreads_or(*changed*) is used to keep individual blocks iterating until the block converges by instructing each block to re-iterate if any thread updated a pixel value. $INT_MAX$ is used to denote pixels over the threshold.

that's labeled with a centroid, it looks up the centroid by it's label, then adds the $x, y, z$ location of the pixel to it and also increments the pixel count.

This is illustrated in figure 4.15.

## 4.4 Host code

The host code in this implementation has fairly minimal functionality. It largely exists to manage passing data between the different CUDA stages, and also handles the usual configuration duties- configuring variables, reading input and writing output.

The host code does, however, perform some relevant computational duties. Because

$x = blockIdx.x \times blockDim.x + threadIdx.x;$
$y = blockIdx.y \times blockDim.y + threadIdx.y;$
**for all** $z$ **do**
  **if** $pix(x, y, z) \neq INT_MAX$ **then**
    $centroid$ =getCentroid($pix(x, y, z)$)
    $centroid.x+ = x$
    $centroid.y+ = y$
    $centroid.z+ = z$
    $centroid.pixel_count+ = 1$
  **end if**
**end for**

**Figure 4.15:** 3D centroid-formation algorithm. The function getCentroid($pix(x, y, z)$) gets a reference to the global centroid array, looking up the correct centroid by index.

the GPU memory is limited, it can't reconstruct and process an arbitrarily large number of Z planes at a time- the memory required scales with the size of the hologram and reconstruction resolution. As such, it must reconstruct and process sets of planes iteratively. The host code manages this process by iterating over the Z plane sets and launching the kernels appropriately.

Additionally, each iteration creates as output a complete set of detected centroids. Because the Z plane processing commences iteratively, it's possible that a particle is detected in two such sets. It's also possible that a particle becomes over-segmented- that is, multiple centroids are registered for a single particle because the refraction pattern for that particle is composed of two or more disjoint pixel groups. The host code must correct for these errors.

Fortunately, both can be solved in one pass: In both cases, centroids are simply merged together based on proximity- if two centroids are close enough in $x, y$, and $z$, it can reasonably be assumed that they belong to the same particle. Thus the centroids are combined, forming a new centroid with an $x, y, z$ location given by the average of the two contributing centroids, weighted by the number of pixels comprising each.

While this is a computationally-expensive process ($O(n^2)$ in the number of centroids), it maps poorly to the GPU due to the high number of data dependencies.

The actual algorithm for merging centroids is shown below in figure 4.16.

Finally, the host code must do some filtering. Particles' refraction patterns may interfere constructively or destructively with each other to create pixels that register as

```
Require: mergeRadius, a configurable parameter
  for all c_i ∈ centroids do
    for all c_j ∈ centroids, c_j ≠ c_i do
      if distance(c_i, c_j) < mergeRadius then
        merge(c_i, c_j)
      end if
    end for
  end for
```

**Figure 4.16:** Host code algorithm for merging fragmented centroids.

being part of a particle, but aren't. The centroids corresponding to these false-positives should then ideally be detected and discarded.

The refraction pattern for each particle changes with each Z plane. Because of this, any interference significant enough to trigger a false-positive reading is likely to be short-lived. As such, the centroid resulting from any such interference will have relatively few pixels contributing to it when compared to a centroid created by a real particle. Thus a simple filter to discard centroids smaller than a fixed number of pixels is effective in greatly reducing these false-positive results.

This approach is shown in figure 4.17.

```
Require: minCentroidSize, a configurable parameter
  for all centroid c_i do
    if sizeof(c_i) < minCentroidSize then
      remove c_i
    end if
  end for
```

**Figure 4.17:** Host code algorithm for reducing false-positives reported.

Finally, the high-level host code operation is shown in figure 4.18.

**Require:** *hologram*, an input hologram image
**Require:** $k$, the number of planes that can be handled at once in GPU memory
**Require:** $nz$, the number of planes to reconstruct

```
Allocate memory on GPU
for all i, k × i < nz do
   Launch stage 1-2 kernels
   Launch stage 3 kernels
   Launch stage 4-6 kernels
   Copy centroids to host from GPU
end for
Merge centroids
Remove false-positives
return centroids
```

**Figure 4.18:** Host code algorithm for controlling the entire hologram reconstruction and particle extraction.

# Chapter 5

# Results and Discussion

## 5.1 Test setup

Due to difficulties in getting the existing MATLAB code to run in order to benchmark it, the MATLAB and CUDA experiments were run on different machines.

The GPU tests were run on a Linux machine containing two Intel Xeon E5-2650 processors at 2.0 GHz, 64 GB of RAM, and three Nvidia GPUs was used (though only a single Tesla K20c GPU was used by the application). The software used was CUDA 6.5.

The MATLAB tests were run on a Windows 7 machine containing an Intel Core i7-4770 CPU at 3.4 GHz with 16 GB of RAM. MATLAB 2013 was used to perform the tests.

While this is less than ideal for making a fair comparison, the MATLAB performance is, if anything, overstated, as the CPU used for the MATLAB computations has almost twice the single-thread performance as that used for the GPU experiments [16].

For both the CUDA and MATLAB configurations, the inputs provided consisted of the input hologram, as well as parameters concerning the hologram, such as the wavelength of the illuminating light, the resolution of the image (pixels/m), and the reconstruction depths, which theoretically correspond to the bounding near and far planes of the particles contained in the hologram.

As outputs, each algorithm provided a list $x, y, z$ positions, one per particle detected.

To allow the accuracy of each approach to be tested, simulated holograms were

generated and used as inputs, with each hologram tested by both the CUDA and MATLAB implementations. Because the holograms were simulated, the "real" locations of all particle contained within them could be known. This information was compared to the particles detected by each algorithm, and used to determine the accuracy of each.

## 5.2 Figures of merit

Before comparing the GPU and reference implementations, figures of merit must be decided upon. In this instance, there are essentially two points on which a comparison can be made.

Firstly, speed. Because the driving force for this research is to greatly reduce the time overhead involved in analyzing fluid flows, the time required to reconstruct and analyze a hologram is a key point of comparison. As mentioned previously, memory operations (allocating memory and moving data) are relatively expensive in CUDA. As such, it becomes relevant to not only measure the time required to reconstruct and analyze a single hologram, but also the time required to process multiple holograms, where the memory allocation cost is amortized across multiple images.

The second figure of merit for hologram reconstruction is accuracy. Being able to reconstruct holograms and detect particles arbitrarily fast is only useful if the results are useful, after all. With holograms, accuracy can be compared in a number of ways- the position of detected particles in three dimensions compared to the physical location of particles in space, and the number of particles detected. The latter point encompasses detecting all the particles that exist, as well as the existence of false-positives- reporting a particle where none exists.

## 5.3 Test results

The performance of both the MATLAB and CUDA implementations are shown in table 5.1. Figure 5.3 graphically represents execution, and the relative time required for each CUDA stage is shown in figure 5.1. Figure 5.2 further highlights the accuracy of the CUDA and MATLAB approaches as a function of the particle field density.

| Technique | Number of particles | Particles Detected | False Positives | Average XY error ($\mu m$) | Average Z error ($\mu m$) | Std. Z error ($\mu m$) | Total runtime (s) | Calculation time (s) |
|---|---|---|---|---|---|---|---|---|
| 3D CCL | 100 | 100 | 0 | 0 | 0.280 | 0.4 | 5.596 | 1.779 |
| 2D CCL | | 100 | 0 | 0 | 0.270 | 0.5 | 5.409 | 1.355 |
| MATLAB | | 100 | 0 | 0.039 | 0.984 | 0.3 | 2091 | — |
| 3D CCL | 467 | 467 | 0 | 0.001 | 0.308 | 0.5 | 5.579 | 1.817 |
| 2D CCL | | 467 | 0 | 0.005 | 0.291 | 0.5 | 5.426 | 1.380 |
| MATLAB | | 467 | 0 | 0.090 | 1.022 | 0.4 | 2127 | — |
| 3D CCL | 1030 | 1016 | 4 | 0.047 | 0.764 | 2.5 | 5.743 | 1.932 |
| 2D CCL | | 1017 | 4 | 0.110 | 0.668 | 1.9 | 5.118 | 1.401 |
| MATLAB | | 1021 | 0 | 0.136 | 1.033 | 0.5 | 2131 | — |
| 3D CCL | 1417 | 1402 | 6 | 0.344 | 1.333 | 7.2 | 5.846 | 2.049 |
| 2D CCL | | 1402 | 6 | 0.385 | 1.255 | 7.1 | 5.273 | 1.445 |
| MATLAB | | 1379 | 3 | 0.163 | 1.029 | 0.5 | 2121 | — |
| 3D CCL | 1877 | 1829 | 7 | 0.253 | 1.448 | 5.1 | 6.044 | 2.253 |
| 2D CCL | | 1826 | 8 | 0.321 | 1.403 | 5.0 | 5.475 | 1.577 |
| MATLAB | | 1795 | 6 | 0.187 | 1.075 | 0.6 | 2516 | — |
| 3D CCL | 2229 | 2180 | 25 | 0.648 | 2.413 | 10.5 | 6.301 | 2.471 |
| 2D CCL | | 2186 | 27 | 0.662 | 2.376 | 10.6 | 5.541 | 1.695 |
| MATLAB | | 2094 | 17 | 0.116 | 1.102 | 0.6 | 2298 | — |

Table 5.1: Performance comparison between CUDA and MATLAB code. All holograms were $1024 \times 1024$ pixels, with particles 3 $\mu m$ in diameter, and a camera resolution of 0.75 $\mu m$/pixel. 1101 planes were reconstructed in 1 $\mu m$ increments (z = 0 to 1100 $\mu m$). MATLAB doesn't have fixed memory allocation costs as CUDA does, and thus does not have a separate 'calculation' time.

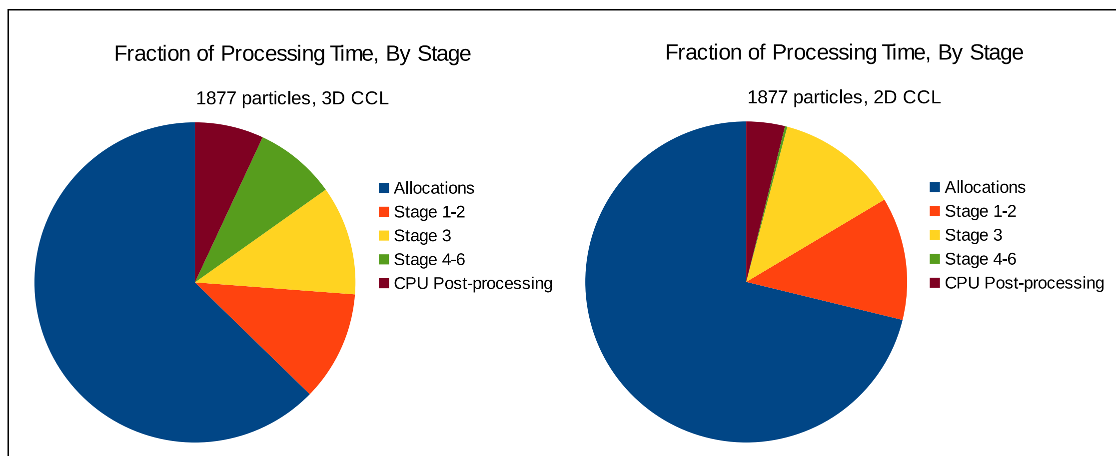


**Figure 5.1:** Relative execution time of each CUDA stage. Memory transfers for each stage are rolled into each stage's execution time.

## 5.4 Discussion

### 5.4.1 Detection Rate and False-Positives

The first metric to compare the CUDA and MATLAB implementations is how many particles were correctly detected, and how many false-positives were reported. From table 5.1, it can be seen that both implementations behave identically up to 467 particles. At 1030 particles, the MATLAB implementation performs slightly better, detecting a few more particles and reporting no false positives (as compared to 4 false positives for the CUDA code).

For greater particle densities (1415-2229 particles), the CUDA version detects more of the particles, at the expense of more false-positives. Both the 2D and 3D segmentation approaches used by the CUDA algorithm perform similarly.

The false-positive rate should be considered carefully, however, as the CUDA and MATLAB codes perform different post-processing steps on the detected particles. Because this work focused on the parallelizable portions of the algorithm, the post-processing code has room for improvement, which could well improve the false-positive rate to a level on par with the existing MATLAB code.

Despite this, both MATLAB and CUDA perform well by this metric, having around a false positive rate of under 2% and detecting well over 90% of the particles present in the hologram.

### 5.4.2 XY Error

As shown in table 5.1, the accuracy of detected particle locations in the XY plane was found to be superb for all particle densities and both CUDA and MATLAB approaches. Average error in all was well under one particle diameter.

While the accuracy was found to be good, it did show a decrease as the particle density increased. This is due in part to the higher numbers of false-positives, which generally appeared near, but not exactly on top of, the locations of real particles, and thus contributed to larger average error values. Because the false-positives were reported as particles by the algorithm, their error was still counted in calculating the overall accuracy of the implementation.

### 5.4.3 Z Error

The MATLAB and CUDA implementations diverge when it comes to the accuracy of located particles along the Z axis, as shown in figure 5.2. For up to 1030 particles in the field, the different approaches behaved similarly, with the CUDA implementation demonstrating slightly more accuracy at lower densities.

At higher densities (1417-2229 particles), however, the CUDA implementation proved to be less reliable. While the average detection error remained under the diameter of a particle, the standard deviation increased enormously. This is again partially attributable to the false-positive issue noted previously, where each false-positive contributed significantly more error than the real particles detected.

With the values measured, the CUDA implementation likely would not be useful for real applications with high particle densities, though future work to implement more advanced post-processing steps could likely change that. This is further supported by the CUDA implementation actually performing better at lower field densities.

### 5.4.4 Execution Time

Before comparing the execution times of the MATLAB and CUDA implementations, it should be noted that, despite running on a faster CPU, the MATLAB implementation is inherently handicapped. Because MATLAB code is interpreted, it runs slower than a

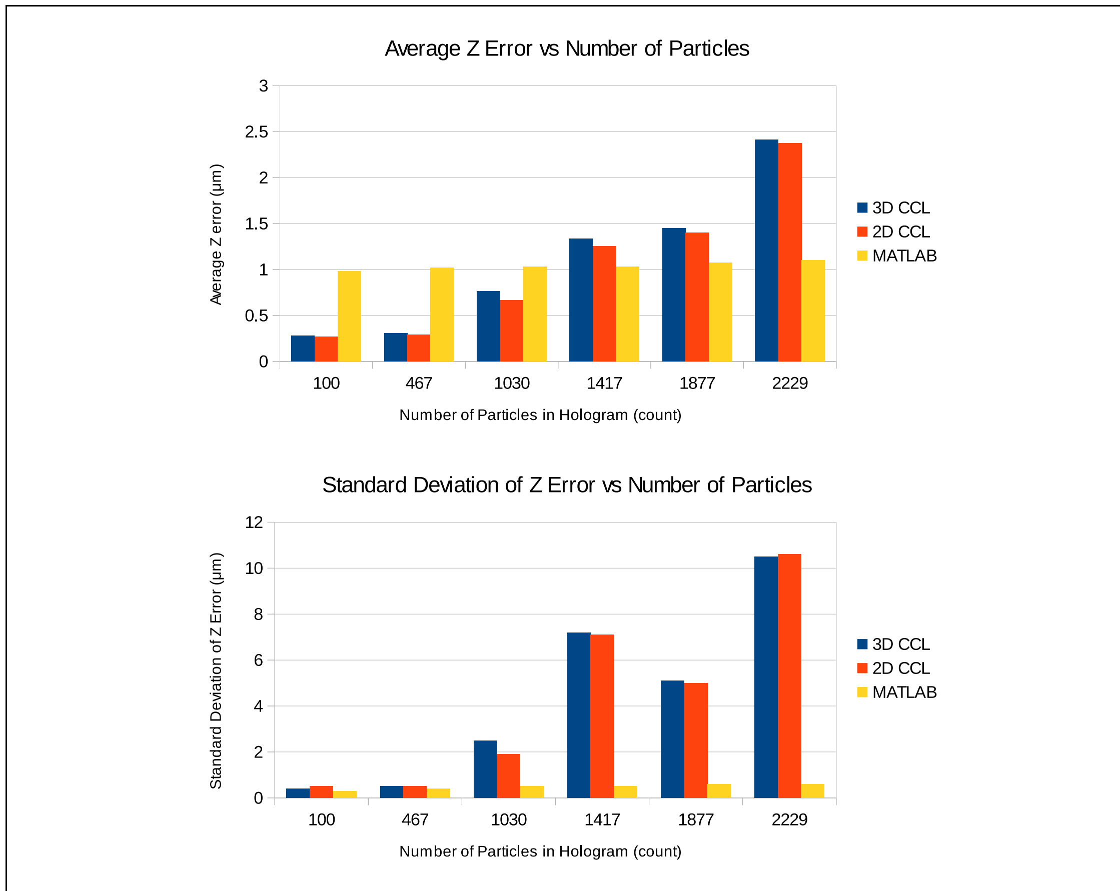


**Figure 5.2:** Plots of CUDA (2D and 3D CCL) and MATLAB average Z error (top) and standard deviation (bottom), as a function of particle density.

compiled C program[1] , such as the CUDA implementation. As such, the performance gains demonstrated are not entirely due to the parallelization of the algorithms, but also in part to the fact that they're compiled into code that can run directly on the CPU and GPU without the interpreter's overhead.

Regardless of the reason, the CUDA implementation clearly demonstrates a performance advantage. As shown in figure 5.3, the MATLAB implementation takes significantly longer- by almost three orders of magnitude- to perform the same workload as the CUDA version. This difference is especially notable when only the CUDA calculation time is taken into account, as one-time memory allocations and other setup account for

[1] This likely is the same reason the work presented here shows significant performance improvements over that presented in [15].

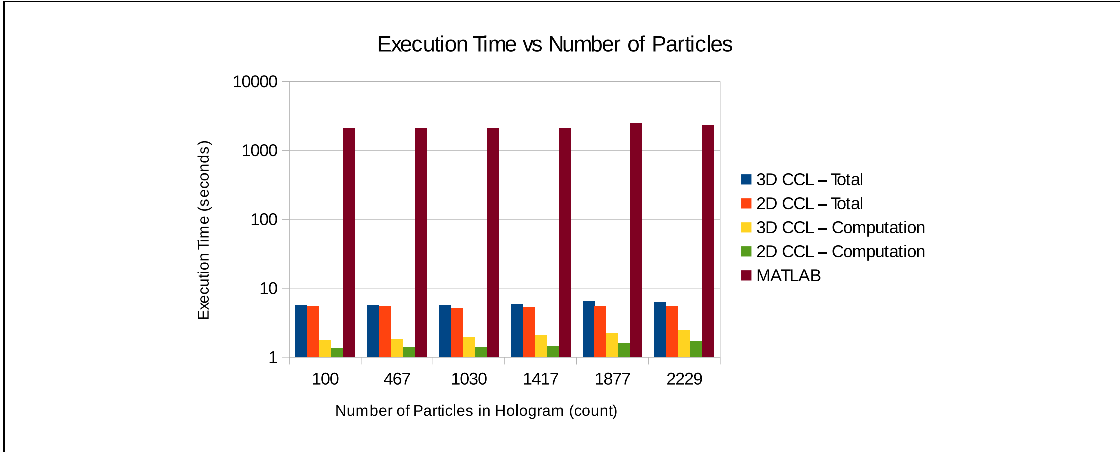


**Figure 5.3:** Plot of MATLAB and CUDA execution times for various particle field densities.

over half of the CUDA runtime (table 5.1 and figure 5.1).

One other point of interest is the dependence on the particle field density. The MATLAB version seems to have negligible dependence on the field density- likely because reconstructing the planes and calculating tenengrad maps is the majority of the execution time, and thus any small additional amount of work is less than obvious.

The CUDA version clearly has some dependence on the number of particles. Because some post-processing is done on the CPU after the GPU has finished, and because the GPU is significantly faster than the CPU, the extra time is highly visible. Furthermore, the post-processing steps in the CUDA version are $O(n^2)$, creating a larger dependence on particle count. Figure 5.1 shows that this processing time is non-negligible for larger particle counts.

Figure 5.1 contains a couple other details of interest. Firstly, it highlights the large one-time setup overhead CUDA has: over half of the execution time. Batch processing of holograms thus stands to show further performance gains over a single hologram.

Furthermore, it can be seen that, especially for the 3D CCL approach, each CUDA stage contributes enough execution time to benefit from parallelization. This is likely a factor in the performance improvement seen over [15], which only parallelized part of the algorithm. The 2D CCL approach shows the near-negligible stage 4-6 computation time, and also the lower relative effect of the CPU-based post-processing. Both of these point to it being more conducive to a parallel approach than 3D CCL.

# Chapter 6

# Conclusion

GPU computing is a highly effective solution to a class of problems with low data transfer and high computation requirements. This paper presents a GPU-accelerated implementation of one such problem: that of reconstructing digital holograms and extracting 3D particle locations from them.

It further demonstrates that the parallel implementation of the procedure produces similar results as the sequential version (for relatively sparse particle fields), while taking almost three orders of magnitude less time to execute.

Some accuracy issues remain at higher particle field densities. These could likely be resolved with more sophisticated post-processing techniques of the detected particles, such as those implemented in the MATLAB code. This is one area of future research.

The implementation presented also has possibilities for further performance enhancements. Because groups of reconstructed planes are processed iteratively, independently of the other groups, the possibility exists to process them in parallel using multiple GPUs. A performance increase roughly linear in the number of GPUs used should be possible with this technique.

Despite these opportunities for improvement, the ability to extract process sufficient holograms for flow analysis in a matter of a few hours instead of a few weeks is a huge step forward.

# References

[1] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs journal*, 30(3):202–210, 2005.

[2] Hoonjong Kang, Fahri Yaraş, and Levent Onural. Graphics processing unit accelerated computation of digital holograms. *Applied optics*, 48(34):H137–H143, 2009.

[3] Tomoyoshi Shimobaba, Tomoyoshi Ito, Nobuyuki Masuda, Yasuyuki Ichihashi, and Naoki Takada. Fast calculation of computer-generated-hologram on amd hd5000 series gpu and opencl. *Optics express*, 18(10):9955–9960, 2010.

[4] NVIDIA Corporation. *CUDA C Programming Guide*, July 2013.

[5] Jerome H Milgram and Weichang Li. Computational reconstruction of images from holograms. *Applied optics*, 41(5):853–864, 2002.

[6] Joseph Katz and Jian Sheng. Applications of holography in fluid mechanics and particle dynamics. *Annual Review of Fluid Mechanics*, 42:531–555, 2010.

[7] Jian Sheng, Edwin Malkiel, and Joseph Katz. Single beam two-views holographic particle image velocimetry. *Applied optics*, 42(2):235–250, 2003.

[8] Daniel R Guildenbecher, Jian Gao, Phillip L Reu, and Jun Chen. Digital holography reconstruction algorithms to estimate the morphology and depth of nonspherical absorbing particles. In *SPIE Optical Engineering+ Applications*, pages 849303–849303. International Society for Optics and Photonics, 2012.

[9] Lukas Ahrenberg, Andrew J Page, Bryan M Hennelly, John B McDonald, and Thomas J Naughton. Using commodity graphics hardware for real-time digital hologram view-reconstruction. *Display Technology, Journal of*, 5(4):111–119, 2009.

[10] Yasuyuki Ichihashi, Ryutaro Oi, Takanori Senoh, Kenji Yamamoto, and Taiichiro Kurita. Real-time capture and reconstruction system with multiple gpus for a 3d live scene by a generation from 4k ip images to 8k holograms. *Optics express*, 20(19):21645–21655, 2012.

[11] S Bianchi and R Di Leonardo. Real-time optical micro-manipulation using optimized holograms generated on the gpu. *Computer Physics Communications*, 181(8):1444–1448, 2010.

[12] Tomoyoshi Shimobaba, Yoshikuni Sato, Junya Miura, Mai Takenouchi, and Tomoyoshi Ito. Real-time digital holographic microscopy using the graphic processing unit. *Optics express*, 16(16):11776–11781, 2008.

[13] Jacob P Fugal, Timothy J Schulz, and Raymond A Shaw. Practical methods for automated reconstruction and characterization of particles in digital in-line holograms. *Measurement Science and Technology*, 20(7):075501, 2009.

[14] László Orzó, Z Gorocs, István Szatmári, and Szabolcs Tokés. Gpu implementation of volume reconstruction and object detection in digital holographic microscopy. In *Cellular Nanoscale Networks and Their Applications (CNNA), 2010 12th International Workshop on*, pages 1–4. IEEE, 2010.

[15] Fook Chiong Cheong, Bhaskar Jyoti Krishnatreya, and David G Grier. Strategies for three-dimensional particle tracking with holographic video microscopy. *Optics express*, 18(13):13563–13573, 2010.

[16] PassMark Software, 2014.

# Appendix A

# Glossary and Acronyms

## A.1 Glossary

- **Block** – A collection of CUDA threads. May be 1-, 2-, or 3-dimensional.
- **Compute Unified Device Architecture** (**CUDA**) – Nvidia Corporation's GPU computing platform, allowing their graphics processors to perform general computation.
- **Connected-Component Labelling** (**CCL**) – The process of examining the pixels in an image and assinging those belonging to a single component (eg, a particle) the same label or number.
- **False-positive** – A particle reported by the algorithm which doesn't correspond to any physical particle present in the hologram.
- **Global memory** – The on-GPU RAM of a CUDA GPU. Equivalent to the "main memory" of a CPU.
- **Graphics Processing Unit** (**GPU**) – A device in modern computers that handles rendering graphics on the display. Can also be used for general-purpose processing, operating in a massively parallel computing paradigm.
- **Grid** – A collection of blocks in CUDA. May be 1-, 2-, or 3-dimensional. Each grid corresponds to a single kernel launch.

- **Minimum-Intensity Image** – A projection of a reconstructed hologram onto a 2D plane. Each $x, y$ location contains the lowest value of any pixel in that $x, y$ location across all $z$ planes.

- **Shared memory** – Fast, explicit cache on CUDA GPUs, available to programmers to minimize accesses to global memory and maximize performance.

- **Tenengrad** – Essentially the gradient of a pixel, calculated using all 8 pixels surrounding the pixel of interest.

- **Tenengrad Map** – An image with pixels containing the tenengrad of each pixel of an original image; similar to a gradient map.

- **Thread** – A single process operating on a CPU or GPU.

- **Warp** – The unit of execution for a CUDA GPU. The threads in a warp are executed in lockstep by the GPU. Warps are transparent to the programmer, and managed by the GPU.

## A.2 Acronyms

Table A.1: Acronyms

| Acronym | Meaning |
|---|---|
| CCL | Connected-Component labelling |
| CUDA | Compute Unified Device Architecture |
| FFT | Fast Fourier Transform |
| GPU | Graphics Processing Unit |
| SIMD | Single Instruction, Multiple Data |