PARALLEL IMPLEMENTATIONS OF HYPERSPECTRAL REMOTE SENSING ALGORITHMS

A Thesis Presented
by
Cory James Crothers Brett
to
The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements
for the degree of

Master of Science
in
Electrical Engineering

Northeastern University
Boston, Massachusetts

May 2014
Parallel Implementations of Hyperspectral Remote Sensing Algorithms

Cory James Crothers Brett

March 26, 2014
Abstract

Remote sensing of materials and chemical clouds using Hyperspectral imaging sensors has many different applications. Some of these applications, such as detecting plumes in the aftermath of natural disasters, are time sensitive. As of now, the detection must be done on an offline system. In this thesis, we consider the matched filter, the normalized matched filter, and sequential maximum angle convex cone detection techniques for parallel implementations. The detectors are mapped to a multicore CPU using multithreading, and efficient data management to achieve a real time result. A Graphics Processing Unit (GPU) is also investigated as a possible architecture for the time sensitive problem. Finally, we assess the performance of the implementations in terms of run time, and conclude how the performance can be improved further.
Acknowledgements

This thesis was done with the help and guidance of many people. First, I would like to name Dimitris G. Manolakis for his immense help and guidance during research, presentation, and tasks of the work. I would also like to thank Professor Vinay K. Ingle for allowing the opportunity to work with him and Dr. Manolakis over the last two years. I would further like to thank Robert DiPietro for his help with understanding these algorithms and the background information necessary to complete this thesis. Finally, I would like to thank Professor Miriam Leeser for her class, High Performance Computing, which taught me different techniques used throughout this thesis, as well as reading this thesis.
## Contents

Abstract 2

Acknowledgements 3

1 Introduction 8
   1.1 Hyperspectral Image ............................................. 8
   1.2 Signal Modeling .................................................. 9
      1.2.1 Signal Detection ............................................ 11

2 Multicore CPU 15
   2.1 Architecture .................................................... 16
      2.1.1 Processor .................................................. 16
      2.1.2 Memory ..................................................... 18
   2.2 Mapping Signal Detection to Multicore CPU ............... 21
      2.2.1 Basic Linear Algebra Routines ............................... 21
      GEMM: Subroutine Optimization ................................ 22
      2.2.2 OpenMP .................................................... 25
      2.2.3 Summary .................................................... 27

3 Graphics Processing Unit (GPU) 28
   3.1 Architecture .................................................... 28
   3.2 Running the GPU ................................................ 31
   3.3 Mapping Signal Detection to GPU ............................... 33
List of Figures

1.1 Example of a graphical approach .............................................. 12
1.2 Example of graphical approach: Demean .................................... 12
1.3 Example of graphical approach: Whitened ................................. 13

2.1 Example of CPU Architecture [1] ............................................. 17
2.2 Memory Pyramid [2] ............................................................. 18
2.3 2-set way associative cache .................................................. 20
2.4 SSE example [3] ................................................................. 25
2.5 Spawning thread in openMP ................................................... 26

3.1 Fermi Architecture: Green is execution units, orange is scheduler and dispatch, dark blue is main memory and light blue is registers L1 cache [4]. . . 29
3.2 Streaming Multiprocessor [4] .................................................. 30
3.3 Thread organization [4] ......................................................... 32
3.4 Coalesce vs. Not coalesce memory accesses ............................... 35
3.5 GEMM GPU breakdown [5] .................................................... 38

4.1 Matched-filter (MF) and Normalized Match-filter (NMF) components . . . . 40
4.2 cuBLAS implementation of Equations 4.1 and 4.2. ......................... 44
4.3 Technique in [6]. One block of many launched ............................ 45
4.4 First Memory Overlap attempt. 256 threads and 160 blocks were spawned . 45
Chapter 1

Introduction

Hyperspectral images (HSI) are similar to a normal camera in that it measures energy off of object illuminated by the sun in spectral bands. The difference is in the wavelength of the bands captured and the amount of spectral bands. For sensing chemical gas plumes, we look at the Long-wave Infrared (LWIR) bands encompassing [8-13]µm. We use the LWIR sensor range because these gas plumes have a unique spectral signatures in this range as opposed to the short-wave infrared (SWIR) sensor range [8]. This technology has a wide range of applications such as measuring gas pollutants in the factory settings, response to natural disasters to see if there are hazardous chemical gases in the air, and the chemical warfare agents in protecting our military. Due to the nature of these applications, implementing them efficiently will cut down on crucial response times.

1.1 Hyperspectral Image

Before we explain how we implement the remote sensing, we need to know how the image is made, and what it comprised of. The sensor is made up of three major components, an HSI spectrometer, an image scanning device, and optical system.
The optical system is what grabs the radiant energy from the objects and atmosphere below it. The image scanning device has different ways of moving the optical system in order to collect the data needed to populate the image. The spectrometer is what takes the energy that was collected, separates it by wavelength, and gives it an intensity of how much energy was collected in each wavelength.

1.2 Signal Modeling

Once we have the energy in the image, we need to know how to mathematically model it. When the image is formed, we need to be able to break it out into sections. For example, if we are detecting some type of chemical plume that is a half mile away from our sensor, there will be distortion from the atmosphere between the plume and the sensor, as well as the background clutter or earth surface. These types of unwanted radiant energy needs to be modeled so that it can be taken away so we can accurately detect the chemical plume. So, we will break the image into these type of components, the first being the background of the image, the second being the plume itself, and the third being atmosphere between the plume and the sensor. We further break it down into parts that contain the plume, and parts that do not, $L_{\text{on}}(\lambda)$ and $L_{\text{off}}(\lambda)$ respectively. The second part is given by

$$L_{\text{off}}(\lambda) = L_{\text{a}}(\lambda) + \tau_{\text{a}}(\lambda)L_{\text{b}}(\lambda) \quad (1.1)$$

where $L_{\text{a}}(\lambda)$ is the atmospheric layer, the atmosphere, $L_{\text{b}}(\lambda)$ is the background, and $\tau_{\text{a}}(\lambda)$ is atmospheric transmittance. Atmospheric transmittance is what happens to the radiant energy when it passes through the atmosphere. It gets affected and manipulated by the atmosphere by the time it reaches the sensor; therefore it must be taken into account when modeling the signal. When there is a plume present, we have
\[ L_{\text{on}}(\lambda) = L_a(\lambda) + \tau_a(\lambda)\tau_p(\lambda)L_b(\lambda) + \tau_a(\lambda)[1 - \tau_p(\lambda)]B(\lambda, T_p) \]  

(1.2)

where similarly \( \tau_p(\lambda) \) is the plumes transmittance, \( T_p \) is the temperature, and \( B(\lambda, T_p) \) is Planck function, which gives a model for the amount of radiant energy the plume gives off at a certain temperature. \( L_{\text{on}}(\lambda) \) and \( L_{\text{off}}(\lambda) \) components give us a total picture of a model of the whole HSI image. Combining these together to get a more useful picture,

\[ L_{\text{on}}(\lambda) = \tau_a(\lambda)[1 - \tau_p(\lambda)](B(\lambda, T_p) - L_b(\lambda)) + L_{\text{off}}(\lambda). \]  

(1.3)

This is the model for the total signal received by the sensor. From here, we need to make some assumptions in preparation for the use of the detection algorithms. The first of them being that the plume is optically thin. What this does is it gives an ability to say that \( \tau_p(\lambda) \) can be linearized. Next, the temperature difference between the background and plume \( (\Delta T = T_p - T_b) \) must be small, so that we can linearize the Planck function. Finally, we use a flat background emissivity approximation. This refers to the ability of the background to radiate energy relative to a blackbody. This approximation will allow us to say that there the emissivity is approximately one. With these approximations taken from [8], we get a more workable signal model,

\[ L_{\text{on}}(\lambda) = \sum_{m=1}^{N_g} (C_b\Delta T \gamma_m)\tau_a(\lambda)\alpha_m(\lambda) + L_{\text{off}}(\lambda) \]  

(1.4)

where \( C_b \) is a constant. Up until this point, the model is the at-sensor model. So now, the sensor needs to process that radiance, which adds some effects and noise, leading to

\[ \tilde{L}_{\text{on}}(\lambda) = L_{\text{on}}(\lambda) \ast R_F(\lambda) + \eta(\lambda) \]  

(1.5)

where \( \eta(\lambda) \) is noise added by the sensor, \( R_F(\lambda) \) is sensors spectral response function, and \( \tilde{L}_{\text{on}} \) is the radiance in the middle of the spectral channel \( \lambda \). The operation \( \ast \) is a convolution.
operation. Renaming these and forming them into vectors,

\[ x = Sg + v, \quad v \sim N(m_b, C_b) \]  

(1.6)

where \( S \) is the gas signatures, \( g \) is linear mixture model coefficients, and \( v \) is the background and noise clutter. This is the final signal model that we will use. Its simplicity is desired because with this, we can now use mathematical techniques to complete our remote sensing [8].

### 1.2.1 Signal Detection

Getting a good model for the image is important for formulating the detection algorithms. Assuming that \( v \sim N(m_b, C_b) \), \( S \) to only have one signature, so \( S = s \), and \( m_b \) and \( C_b \) are known by maximum likelihood test, we get a least-square solution,

\[ g = (s^T C_b^{-1} s) s^T C_b^{-1} x. \]  

(1.7)

The problem with \( g \) is that it is biased. By removing the mean \( m_b \), \( v \) will have new distribution \( v \sim N(0, C_b) \). Using Hypothesis testing,

\[ H_0 : g = 0 \text{ (Plume absent)} \]  

(1.8)

\[ H_1 : g \neq 0 \text{ (Plume present)} \]  

(1.9)

we can use the generalized likelihood ratio test approach [9], and get the MF detection algorithm,

\[ Y_{MF} = \frac{(x^T C_b^{-1} s)^2}{(s^T C_b^{-1} s)^{-1}} \]  

(1.10)

which then leads to the normalized matched filter (NMF)

\[ Y_{NMF} = \frac{Y_{MF}}{x^T C_b^{-1} x}. \]  

(1.11)

The MF and NMF give a way to detect a chemical plume in HSI [8]. Graphical interpretations of these derived detection algorithms can be just as informative and useful as the
mathematical derivations.

Figure 1.1: Example of a graphical approach

Figure 1.1 gives an insight into each pixel model. There are two Gaussian distributions shown, the green being the background clutter, and the red pixels containing plume. Pixels can fall between these if the pixel is made up of more than just plume.

Figure 1.2: Example of graphical approach: Demean

Removing the mean from the image is depicted in 1.2. It moves the background clutter distribution to the origin as expected. It does not move the plumes distribution because the mean is made up primarily of background clutter. In order to completely show how $Y_{MF}$ or $Y_{NMF}$ are shown geometrically, the whitening must be represented in a figure.
Whitened space is a linear transformation converting variance of a distribution to 1. In HSI, this means taking our data matrix $X$ whose variance $C_b$ and transforming it to $\text{var}(X) = AC_bA^T = \sigma^2 I$. To express the derivation

$$C_b = \frac{1}{n_p}X^TX, \quad (1.12)$$

we propose that using $C_b^{-\frac{1}{2}}$ can act as a whitening matrix, hence

$$Y = C_b^{-\frac{1}{2}}X \quad (1.13)$$

$$\text{var}(Y) = E[YY^T] = E[C_b^{-\frac{1}{2}}XX^TC_b^{-\frac{1}{2}}] = C_b^{-\frac{1}{2}}E[XX^T]C_b^{-\frac{1}{2}} = C_b^{-\frac{1}{2}}C_bC_b^{-\frac{1}{2}} = I. \quad (1.14)$$

This shows that $Y$ is a whitened version of $X$. This whole system is not unique to $C_b^{-\frac{1}{2}}$, but any rotation of that will work. For a more efficient way of getting a whitening matrix, we use the Cholesky Decomposition. It breaks up $C_b = LL^T$ where $L$ decomposes $C_b$ into upper and lower triangular matrices, and can be a whitening matrix. Using the cholesky decomposition, we can do the same whitening technique. From here we can see the last and final graphical representation

![Graphical representation](image)

**Figure 1.3:** Example of graphical approach: Whitened
The whitening in Figure 1.3 is seen with the Gaussian distributions. The shape has become more circular which is evident by $C_b = I$. Now we can look at the decision boundaries for the MF and NMF. The MF boundary is a plane that intersects the background and chemical plume. We project each pixel onto the signature vector. If the projection crosses the MF plane, then there is plume in that particular pixel. The NMF looks at the angle between the signature vector and the pixel vectors. If the angle is small enough, then we say that there is plume in that pixel.

Using these remote sensing algorithms, we will be able to detect different chemical gases in an HSI image. We want to make these as fast as we can because if we are able to get the data, process it before getting the next data cube, we could track chemical plumes in real time. To implement this, we need to investigate different parallel computing architectures.
Chapter 2

Multicore CPU

Up until the early 2000s, most processors that were manufactured were single core processors. At this point, processors hit fundamental limits and a redesign in technology was needed.

The first of these limits is the Power limit. As processors were given faster clocks, the power was increasing. The processors were getting too hot, so there needed to be some way to dissipate the power, and fans were not going to keep up. So, the performance of the processor was limited by the amount of power that the computer could dissipate.

Next was the frequency limit. With the increase of instructions in pipelines, trying to increase the functionality and throughput, there needed again to be a faster clock. But this also came to a point where there were diminishing returns. Even if we added more steps to the pipeline to account for different situations, the throughput would not increase because there were too many steps.

Finally and most importantly, there is the memory limit. It has gotten to the point where accessing DRAM memory is taking hundreds of cycles. Instead of the processor computing, it is spending most of the time waiting for data to get to it. In single core processors, it is the dominating factor in the processors performance [2].

Because of these fundamental limits in serial computing, the new design for processors
was a parallel, multicore design. This allowed for dividing tasks, and lowering the clock speed. By lowering the clock speed, the power dissipation could remain maintained by fans. They could now reduce the amount of steps in the pipeline because each core in the processor did not need to work as hard. And each core has its own instruction cache and both can send requests for data, so we will not be waiting as long for memory. The design started with dual core processors, and has quickly escalated to more cores on a single processor chip.

2.1 Architecture

Architecture changes with different multicore processors. Depending on the company that builds it, there will be some different aspects in each. For the most part, companies do not release their actual design and pipelines for the public to see, but there is a general idea of what the pipeline is doing. This along with memory organization will be explained below.

2.1.1 Processor

The processor as explained above has changed considerably since its creation. There has been many advances in processor design, but it is based on the stages that are explained below. For our purposes, explanation of things such as branch prediction or out of order instructions is not necessary. Figure 2.1 is an example of a processor that one might find. Explaining this form left to right, the first thing that should be noticed is the Fetch/Decode control block. This attached to the instruction cache and instruction decoder. This is the first stage of the pipeline; it grabs the next instruction from the instruction cache, and brings it to the instruction decoder.

As shown, the instruction decoder (ID) is the next step. This is not only connected to the instruction fetch (IF) stage, but also is connected to what is called the TLB. This stands for translation look aside buffer (TLB). When an instruction is decoded, it has memory addresses
Figure 2.1: Example of CPU Architecture [1]

that it needs to access, but these addresses are in terms of virtual memory. Virtual memory is used so that the processor thinks there is more memory than there actually is so it can run many processes at once. Because of virtual memory we need to translate the virtual memory address to the physical memory address, which is done on a page table. Page tables will keep track of whether the page is located in the memory or disk as well as change a virtual memory address to a physical one. The TLB is a full associative cache for the page tables, following a least recently used replacement policy. The TLB will give the translation very quickly, where as if the page table needs to be consulted that is not in the TLB will cause a CPU stall. The decoder takes the instruction and turns it into something that the load/store, and execution units can read.

Following the ID stage there are the execution units (EX). The EX stage takes the values from the registers that were decoded in the ID stage, does the instruction to them and writes them to an intermediate register.

Next is the memory stage (MEM), which takes the answer from the EX stage and gets the memory location that it is going to be written to, then sends it to the write back (WB) stage. This stage just writes the memory location to the new value.
Although there seems to be a lot more going on in Figure 2.1, most of the arrows are just how each stage communicates with each other, and some blocks are just registers which hold information and act as an intermediate step. This picture shows a generalized view of what is going on with one core in a processor. So, in our case, we have multiples of these cores running in parallel all with similar architectures shown in Figure 2.1. These cores are hyperthreaded as well. Hyperthreading allows for two threads in each core, so two processes can be running concurrently on one core [2]. All of the cores have separate L1 and L2 data caches, and all are connected through the L3 data cache.

### 2.1.2 Memory

Memory hierarchy is arguably more important than what is going on in the processor. Although it is improved with more cores in a single processor, the memory limit does not go away. Since memory is much slower than processors, a lot of the time processors are just waiting for the data to get to them to complete the task the instruction has given. This still happens with multicore processors, because the memory speed just cannot keep up with the processor. It necessitates that we know how the memory is organized so that full advantage of the processors speed can be taken. First, this is a general memory hierarchy picture,

![Memory Pyramid](image)

**Figure 2.2:** Memory Pyramid [2]

This is just a basic general way to organize memory. First there are registers. These are
by far the fastest part of memory; they are used as intermediate steps in the pipeline, as well as to store values that are used often. It would be advantageous for a programmer to declare variables that are going to be used on a consistent basis as registers. As you can see, it might only take three clock cycles to get from register to processor.

Our next best option is to have it in the L1 cache, L1 standing for level 1. This memory is much bigger than the registers, on the order of 64Kb. It is the next place that a processor will look for data, and if used efficiently, can prevent CPU stalls.

This pyramid in Figure 2.2 continues in this fashion, were the next best place for the data to be is the next biggest memory type. The more amount of memory there is, the longer the data will take to get to the processor. This leads to the first way we can optimize our code. The closer we can keep the data to the processor, the better our code will perform. Even by keeping the operation count the same, the best optimization a programmer can do is data organization.

Now that we have established the hierarchy of the memory, there is some more design organization that happens inside of a cache. There is three basic ways to organize a cache.

- Direct Mapped
- Set-way Associative
- Fully Associative

In direct mapped cache, each spot in the cache is associated with one memory location. It means that only one memory location can exist in that cache location. This is the fastest way and most efficient way to cache data, although it is not used in practice. Directed mapped caches have complicated, expensive hardware, and take up a lot of space. For these reasons, they are not used.

The next cache would be the fully associative cache. This cache is designed so that any memory location can sit anywhere in the cache. A good thing about this is that we can
keep any combination of data in the cache. The problem is that as the cache grows in size, the processor will take more time to find if the data sits in the cache, rendering the cache useless. If a processor looks for a memory location in the cache, and cannot find it, it needs to send a request for the memory location in the next memory stage. If it takes too long to see if the value sits in the lower cache, then it would make more sense to go straight to the higher cache.

Finally there is the Set-way Associative cache. This is by far the most popular design for caches. It is a blend between direct mapped and fully associative cache. It basically means that there are set locations in the cache that only certain memory can access, but a certain memory location can be put in any of those locations.

<table>
<thead>
<tr>
<th>Main Memory Locations</th>
<th>Cache locations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>2</td>
</tr>
<tr>
<td>19</td>
<td>3</td>
</tr>
<tr>
<td>20</td>
<td>4</td>
</tr>
<tr>
<td>21</td>
<td>5</td>
</tr>
<tr>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>23</td>
<td>7</td>
</tr>
<tr>
<td>24</td>
<td>8</td>
</tr>
<tr>
<td>25</td>
<td>9</td>
</tr>
<tr>
<td>26</td>
<td>10</td>
</tr>
<tr>
<td>27</td>
<td>11</td>
</tr>
<tr>
<td>28</td>
<td>12</td>
</tr>
<tr>
<td>29</td>
<td>13</td>
</tr>
<tr>
<td>30</td>
<td>14</td>
</tr>
<tr>
<td>31</td>
<td>15</td>
</tr>
<tr>
<td>32</td>
<td>16</td>
</tr>
</tbody>
</table>

**Figure 2.3:** 2-set way associative cache

Figure 2.3 shows a simplified cache scheme. In this example, there is only one cache and there are 32 memory locations in the main memory. Memory locations 1-4 are red and are associated with cache locations 1 and 2. This means that any of the red blocks can go into
the red cache locations, and nowhere else. So, the processor knows if it is requesting memory locations 1-4, that it will first look in the first 2 cache locations, and if it is not there, it will have to request the data from main memory. This gets the best of both worlds, in that the cache is smaller, less expensive, and we can have combinations of memory in the cache.

2.2 Mapping Signal Detection to Multicore CPU

Hardware architecture is important to know so that when we are designing our code, we can take full advantage of all the processor gives us. The software needs to be designed with the hardware and memory hierarchy in mind, so that we can manipulate where the memory is, and stop as many stalls as we can. It might seem that the software is more complicated, has more for loops, or if statements. Concise code does not equate to performance.

2.2.1 Basic Linear Algebra Routines

Since there is a lot of linear algebra involved in our algorithms, the first thing that we looked into was basic linear algebra routines (BLAS). These routines were developed in the 1980s using FORTRAN in order to make linear algebra techniques available to engineers, which eventually developed into MATLAB. Since then, there have been significant improvements not only in processor power, but sophistication of BLAS routines. This lead to some investigation on which BLAS routines we should use, and why is it the most effective.

It seemed that one BLAS library was beating all else. GotoBLAS is designed for performance and compiled with hardware in mind. It uses all of the hardware mentioned above, as well as some software optimization that makes it perform better. GotoBLAS gets a performance boost when it considers the TLB in its design. By removing stalls from the TLB misses, GotoBLAS can obtain maximum performance [10].
GEMM: Subroutine Optimization

GEMM is a term used in BLAS language. The GE stands for general, and MM is matrix matrix multiply. Optimizing the GEMM routine will lend itself to all other BLAS routines, and most important in HSI detection.

The discussion in this section is done with GotoBLAS design of GEMM. The open source library that is used in this work is called OpenBLAS. OpenBLAS is an extension of GotoBLAS and is an updated for newer CPU architectures. It is found here [11].

GEMM can be decomposed in many different ways. In our case, it is broken down in steps as follows where \( C = AB + C \). Notation is taken from [10].

<table>
<thead>
<tr>
<th>Table 2.1: Dimensions of matrices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>A</td>
</tr>
<tr>
<td>B</td>
</tr>
<tr>
<td>C</td>
</tr>
</tbody>
</table>

Both \( A \) and \( B \) matrices can be decomposed into:

\[
A = \begin{pmatrix} A_0 | A_1 | A_2 | \ldots | A_{k-1} \end{pmatrix}
\]

\[
B = \begin{pmatrix} B_0 \\ B_1 \\ B_2 \\ \vdots \\ B_{n-1} \end{pmatrix}
\]

(2.1)
Consider that $A_0$ is $\mathbb{R}^{m \times k_c}$ and $B_0$ is $\mathbb{R}^{k_c \times n}$. Next, this can be decomposed again

$$A_0 = \begin{pmatrix} A_{00} \\ A_{01} \\ A_{02} \\ \vdots \\ A_{0(m-1)} \end{pmatrix} \hspace{1cm} C = \begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ \vdots \\ C_{m-1} \end{pmatrix} \quad (2.2)$$

So our equation breakdown so far is

$$C_0 = A_{00}B_0 \quad (2.3)$$

Finally, this meets its final smallest component:

$$\begin{pmatrix} C_{00} | C_{01} | \ldots | C_{0(n-1)} \end{pmatrix} = A_{00} \begin{pmatrix} B_{00} | B_{01} | \ldots | B_{0(n-1)} \end{pmatrix} \quad (2.4)$$

Assume that $A_{ij}$ is $\mathbb{R}^{m_c \times k_c}$, $B_{ij}$ is $\mathbb{R}^{k_c \times n_r}$, and $C_{ij}$ is $\mathbb{R}^{m_c \times n_r}$ [10].

The reasoning behind breaking the matrix multiplication down into its smallest part is because we need to find the best dimensions for our processor. Meaning we can now find out the size of $A_{00}$ that will fit inside the cache for fast access. Once $A_{00}$ is no longer needed, it will be copied back to main memory, and replaced by another sub matrix of $A$. We will assume that $A_{ij}$, $B_{ij}$, and $C_{ij}$ will be able to fit inside the cache together, and that $A_{ij}$ will be cached as long as it is needed.

In Section 2.1.2 we explain how the faster the memory, the smaller the amount of memory that is available. So we need to decide where each sub matrix goes. If we count the amount of memory operations, called memops, per floating point operations (FLOPs), we will see that we need $m_c k_c$ memops from $A_{ij}$, $k_c n_r$ memops from $B_{ij}$, $m_c n_r$ memops from $C_{ij}$ reading, $m_c n_r$ memops from $C_{ij}$ writing. So we have $m_c k_c + (2m_c + k_c)n$ memops, and $2m_c k_c n$ FLOPs. Making an assumption $k_c << n$ [10]

$$\frac{2m_c k_c \text{ FLOPs}}{2m_c + k_c \text{ memops}} \quad (2.5)$$
It is important to see because now we know that the bigger $m_c k_c$, the more FLOPs we are getting from the smaller amount of memops. If we look at this in terms of the dimensions of $A_{ij}$, the bigger $A_{ij}$, the better. By putting $A_{ij}$ into the L2 cache, we are freeing space in the L1 cache for $B_{ij}$ and $C_{ij}$, as well as make $A_{ij}$ bigger.

The datas cache position is not the only important factor in organization. How $A_{ij}$, $B_{ij}$, and $C_{ij}$ are organized in their respected cache will also affect performance. We will assume that the data is in column major order. Moving data is generally done by accesses a group of memory locations instead of one at a time. So, by packing $A_{ij}$, $B_{ij}$, and $C_{ij}$ into an array that the processor will access contiguously, we can reduce the total number of memory accesses.

Organizing the data in this way reduces the amount of cache misses, but they still occur. The CPU can use techniques to mitigate these misses. The difference between a cache miss and a TLB miss in terms of CPU response, is TLB misses will always cause a stall. To mitigate TLB misses, pages have to be kept in the TLB for as long as they are needed and then replaced, not only for fast translations [10].

We have found the dimension parameters for the sub matrices that fit into their respective cache. Keep in mind that each core has its own L1 and L2 cache, so multiple parts of the whole matrix can be computed independently. There was consideration put into the TLB and its effect on performance, as well as the way the sub matrix is ordered in its respective cache. The ability to do this data movement will have the biggest effect on overall performance of each BLAS routine.

What is not seen in this memory optimization is the use of pthreads and SSE instructions. pthreads is a primitive, low-level library that allows programmers to use multiple threads. Many functions are built on top of pthreads for higher performance, including openMP, which is used in our implementation. SSE instructions denotes Streaming SIMD Extensions instructions. SIMD stands for Single Instruction Multiple Data. SSE takes multiple data
points, and does the same instruction on them at the same time. This cuts down on the amount of instructions that need to be issued.

![Figure 2.4: SSE example [3]](image)

An example of how SSE instructions are used is shown in Figure 2.4. The registers are actually 128 bits instead of just 64. This means we can fit 2 64 bit words into one register. If we have two double precision numbers that have the same operation needed to be done then we can cut down the time it takes to do both sequentially in half. Since linear algebra is loaded with these types of instructions (multiply and add at many different memory locations) SSE instructions are important to use when optimizing BLAS operations.

Since MF and NMF are highly linear algebra dependent, SSE instruction become a crucial tool in the optimization process. Linear algebra is filled with just multiply and add instructions on many data points, mapping perfectly to SSE.

### 2.2.2 OpenMP

OpenBLAS is not the only open source library that was used in our implementations. Another important one is openMP (open multiprocessor). This is an open source library that allows for the use of multithreading. At openMP’s core, it is a set of library routines that express
shared memory parallelism [12].

A hardware thread is a term used to describe a processor containing two IF and ID stages in their pipeline. Obviously, this is used to fetch and decode instructions simultaneously, note the hardware threads, because they are physically there. This distinction is there because a programmer can spawn as many threads as they feel fit. So even though a processor could only have 12 hardware threads, the programmer could spawn 30 software threads. The processor then needs to decide which ones go when. Although the programmer might think this is a good idea, there starts to be a problem with context switching. This is when the processor has to decide what thread runs where and when/if that stalls, switch to another thread. This is not a free switch; it takes some amount of clock cycles to occur. With too many threads, context switching can become too cumbersome and actually cause the application to slow, instead of speed up. As a general rule, spawning the same amount of software threads as hardware threads is where the best speed is obtained.

![Figure 2.5: Spawning thread in openMP](image)

Figure 2.5 explains the idea of openMP. A program might have a single threaded part, and a multithreaded part. The multithreaded section divides tasks of a function to complete it, each thread running independent of the others. They are allowed to communicate through
data, but communication cuts into performance. Regardless of the hardware, openMP allows to launch as many threads as desired, and the threads will map automatically to hardware. 

openMP is a vital tool in our implementations of HSI algorithms. When appropriate, dividing pixels into independent parts using openMP gives improvement in run time.

2.2.3 Summary

This chapter is really about utilizing all that the CPU gives us. The CPU gives us hyper threading, so we take advantage of that. We get rid of CPU stalls as much as we can so the processor can run unimpeded. We try and fit our matrices into packed forms so we can get the data to the processor as fast as possible. None of this is possible without knowing not only what is available to us in hardware, but the ins and outs of how it works together, and take advantage of it.
Chapter 3

Graphics Processing Unit (GPU)

A GPU is a processor design for throughput and nothing else. It is not designed to have many tasks to be running at once, or allow for internet connections, or run a word processor. It is only there for mathematical operations. It was originally invented to process graphics for video games. As video games became more popular, the GPU evolved for the growing video gaming community. Because of its impressive throughput, it began to get the attention of the scientific community for faster processing.

3.1 Architecture

The architecture of a GPU looks much different than that of a CPU. The design of a GPU is for massive SIMD operations and impressive throughput. Figure 3.1 shows the basic breakdown of the GPU as a whole. There is nothing special about the main memory and L2 cache as they work very similar to the main memory and L2 cache in the CPU. One thing that you can notice from the very beginning is the amount of execution units compared to Figure 2.1, where there is only a couple SSE execution units. The execution units dominate the GPU where as it is a small portion in the CPU. This is because the CPU needs to be
Figure 3.1: Fermi Architecture: Green is execution units, orange is scheduler and dispatch, dark blue is main memory and light blue is registers L1 cache [4].

able to do a lot more in terms of general programming, whereas the GPU has specific tasks. Inside each of the green blocks in Figure 3.1, there is what we see in Figure 3.2. There are 32 Compute Unified Device Architecture (CUDA) cores, 16 load and store units LD/ST, and 4 special function units (SFU), as well as registers caches and schedulers. The first thing is the instruction cache. This is filled by the GigaThread block in Figure 3.1, and caches instructions that the cuda cores will use. Next are the warp schedulers. A warp in GPU terms is a group of 32 parallel threads. These work together with the dispatch units to send the instructions to each cuda core. It takes two instructions from each warp (warps are completely independent) and distributes them to the CUDA cores, the LD/ST units or the SFUs. These instructions can be issued as any combination, so we can issue a mixture of
integer, floating point, load, store, and SFU instructions at the same time. So in theory, the SM can be computing something on each of these hardware entities [4].

After the warp scheduler and dispatch unit there is the register file. This is just memory locations that can be access by the CUDA cores. Threads are allotted a certain amount of registers, until the register file is full, at which point it will spill into local memory/L1 cache. Depending on the amount of threads spawned, there will be a certain amount per thread. This is a different design than we see in the CPU, were the registers are the smallest memory. Because each core needs to be able to access fast close memory, the register is actually one of the bigger amounts of memory.

The CUDA core is where most of the calculations will occur. Each CUDA core has a
floating point (FP) unit and integer (INT) unit, as well as another dispatch port. Each CUDA core can run up to 48 threads at once, so each SM can handle 1536 threads. These CUDA cores only have one task, and that is to compute the answer to whatever instruction is given. Unlike the CPU, whose cores have to be able to do a multitude of tasks that general computing will come up with, a CUDA core is doing multiplies and adds. An important piece of information about each CUDA core is it contains a fused multiply add (FMA) instruction. This means it will do $c = a \times b + d$ instructions without any rounding in between. This gives a more accurate answer to the same equation on a CPU. Since linear algebra is loaded with FMA instructions, we might get a more accurate answer than the CPU. So even though we are doing the same calculation of both types of devices, we might get different answers, even though both follow the IEEE standard. Finally there are the LD/ST units, which allow for memory addresses to calculate for 16 threads in one clock cycle.

There is also another memory organization after the execution units seen in Figure 3.1. This is a user configurable shared memory/L1 cache of 64Kb. The smallest each cache can be is 16Kb. The L1 cache works exactly as the L1 cache explained in the CPU, and shared memory will be explained in the next section.

There are 14 SMs in our Fermi GPU. As shown in Figures 3.2 and 3.1, the GPU is not terribly complicated. It is evident that the GPU is designed with one thing in mind, and that is throughput. To solve problems on a GPU, it is more efficient to use brute force parallelism than it is to become clever in function design.

### 3.2 Running the GPU

The CPU and GPU are connected via PCI bus. In order for the GPU to run, the CPU needs to send a command through its connection to start working. First thing that needs to happen is to send the data over to the GPU. The PCIe2.0 supports up to 16Gb/s transfer,
which would be running at full capacity. So the CPU calls a memory transfer over to the GPU, and this would be considered the first communication.

The memory transfer from CPU to GPU must happen, and needs to be factored in when using the GPU. One thing that can be used to our advantage is the fact that the CPU and GPU run asynchronously. This means that once the CPU calls a function to be run on the GPU, it moves on to the next instruction. This will become an important in Section 3.3.

Although running asynchronously is the natural way the CPU and GPU work, we can also change that to working synchronously. Function calls like `cudaDeviceSynchronize()` and `cudaMemcpy()` will stall the CPU until the GPU completes that function. For our purposes, having the GPU and CPU in sync does not help us because we cannot queue instructions, as explained in Section 3.3.

Once the data is on the GPU we can start to use its functionality. It is hard to explain the memory system and thread system separately so they will be explained together here.

![Thread organization](image)

**Figure 3.3:** Thread organization [4]
When a kernel starts to run, it has threads, blocks, and grids. A grid is a group of blocks that share the same kernel. A block is a group of threads up to 1024 threads. The significance of blocks is that each thread can communicate with each other in the block through shared memory. Shared memory acts as a cache for a block. For example, suppose we have a vector $z$ of size $n$, and we run a kernel with $n$ threads. During the kernel, every thread needs to access every value in $z$. This would mean $n^2$ main memory accesses. In the beginning of the kernel, if we put all of $z$ in shared memory, all of the thread could access every element in $z$ for a fraction of the memory cost of main memory. The second significant part of blocks is that they run asynchronously. We do not have any control of what block is running when on the GPU, so all blocks must be completely independent of all other blocks. Finally, a thread can be thought of as an instance of a kernel. Each thread will do each instruction in that kernel, similar to the threads on a CPU.

### 3.3 Mapping Signal Detection to GPU

The GPU has almost infinite amount of implementations of a single application. It can be launched with many threads, small amount of threads, different amount of blocks in different kernels, all run in the same kernel. There is a combination of these types of things that will prove to be the best implementation of these algorithms. It is hard to find, and there must be a lot of trial and error in order to find the best way to map our applications to the GPU.

First thing is to know what the software allows the programmer to utilize. When it comes to the transfer of data, there isn’t much that we can do, but what we can is important because this is just overhead. There is no calculation occurring and we are just trying to get the data from the CPU to the GPU, which can turn out to be a high percentage of the actual run time. To get the data from CPU to GPU, there are two different function calls; one of them being `cudaMemcpy()` and the second one being `cudaMemcpyAsync()`. During `cudaMemcpy()`, the
GPU will stall the CPU until the memory transfer is over. In `cudaMemcpyAsync()`, the GPU will allow the CPU to move on to other calculations. In order to use `cudaMemcpyAsync()`, we need to introduce streams. Streams are a sequence of operations that execute in issue-order on the GPU [13]. The Fermi architecture allows for one compute stream, and two memory transfer streams. It allows two memory transfer streams to allow for memory to be copied to the GPU, while memory is being copied to the CPU. So, while we are copying some of our data over to the GPU, we can start the detection process on previously transferred data.

Another memory optimization that we can take advantage of is called pinned memory. When allocating memory on the CPU, we can flag it as pinned memory, which keeps the pages for virtual to physical memory translation in the TLB. When this happens, the CPU can access the data faster, and therefore transfer it over faster to the GPU.

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Time (ms)</th>
<th>Speed up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not pinned</td>
<td>33</td>
<td>–</td>
</tr>
<tr>
<td>Pinned</td>
<td>20.7</td>
<td>1.6x</td>
</tr>
</tbody>
</table>

*Table 3.1:* Overhead speed up with 640,000 x 42 matrix

In Table 3.1 there is a gain of ≈ 13 milliseconds just from pinning the memory and using a different memory copy command. Since the overhead of getting the data is an extra cost that the CPU does not have, making it as small as possible is advantageous. The main memory transfer that we are concern is the data matrix transfer, not the signature vector transfer. The signature vector can be transferred while we are computing, so we can mask the time that takes.
In Figure 3.4 we see the difference between coalescing our memory reads and not coalescing our memory reads. The left most part of Figure 3.4 is coalescing and the right is the opposite. The way that memory is accessed on a GPU is in chunks. For example, if a thread wants to access a global memory location called $q$ for discussion purposes, a group of memory locations around $q$ are brought with it. So if we coalesce our memory reads we will minimize the amount of global memory accesses since the locations around that one accessed are already cached. If we do not coalesce our memory reads, each thread will need to grab a chunk of memory, since it cannot rely on the fact that the thread next to it had already grabbed its memory location. Accessing global memory can be expensive in terms of clock cycles, so by minimizing these accesses, we will speed up our kernels. To Illustrate this, there is a study done by NVIDIA whereby the make a kernel to just read floating point numbers. One is done by with coalesce memory reads, and the other is done without them. The results are in [13].
Table 3.2: NIVIDA study by Paulius Micikevicius

<table>
<thead>
<tr>
<th>Coalesced</th>
<th>Timing (µs)</th>
<th>Speed up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>356</td>
<td>–</td>
</tr>
<tr>
<td>No</td>
<td>3,494</td>
<td>≈ 10x</td>
</tr>
</tbody>
</table>

Just by changing the way you access memory on the GPU can change the timing for such a simple operation. This study was done with 12 Mb of data, where as we have on the order of 100 Mb of data. Since we have more data, coalesce reads becomes more important.

3.3.1 cuBLAS and MAGMA

cuBLAS and Matrix Algebra on GPU and Multicore Architectures (MAGMA) are the basis libraries for linear algebra on the GPU. We need both because cuBLAS does not contain some important linpack matrix decompositions that we need for detection. What cuBLAS does not have MAGMA makes up for. The highest FLOPs that cuBLAS gets is with square matrices, where as in MAGMA, there is multiple designs for each routine. Since there is so many ways to optimize GPU code, different thread and block and even different kernel design can make a big difference in performance. By using both of these libraries, we can tune our application to the fastest code, depending on the dimensions of our cube.

For example, lets look at the matrix-vector multiplication on a GPU. We will look at four different situations that will necessitate different threads and blocks, and what each thread is doing. First, we will look at ”tall and skinny” matrices. These types of matrices will benefit from one thread per row implementation of matrix-vectors multiplication. Since there are 14 SMs in a Fermi GPU card, and each SM can handle 1536 active threads, so that is 1536 · 14 = 21504 rows of a matrix that can be multiplied by a vector at once. Since every thread in each block needs to access the vector \( \mathbf{x} \) in \( \mathbf{A} \mathbf{x} = \mathbf{y} \) we will put it in
shared memory for each block so every thread can access it quickly. We will accumulate the products in a register for each thread because this is the fastest type of memory. Once we have the dot product, we will put the register in the correct memory location of \(y\). Next, for more square matrices, the best performance comes from a several threads per row design. This breaks the blocks into two-dimensions, meaning each thread has an \(x\) and \(y\) direction identification. Each block will serve a sub matrix of \(A\) and a sub vector of \(x\). Again we will put the sub vector into shared memory for easy access. We will now need to introduce the \texttt{atomicAdd()} functionality of the GPU. Atomic functions allow all threads to sum a value in global memory. This can occur with threads from different blocks. For the sum to be correct, the GPU only allows one thread to access the memory location at once. Once it is completed another thread can access it and so on. Although this will cause threads to be waiting for the opportunity to access the memory location, it is a better option than making another kernel to sum these partial values, which is the only other option. We can then have several rows per thread. If a matrix is very tall and skinny, on the order of hundreds of thousands of rows and small amount of columns, this will be the fastest implementation. It will minimize the shared memory accesses since we will be holding a lot of data in registers.

There is one more but it does not need to be discussed for our applications. It is clear that because of our situation with our data cubes in general, we will benefit from the several rows per thread optimization [14].

We have looked at the macro side of the matrix vector case; we can now switch gears and look at the inner workings of a general matrix multiplication GEMM. GEMM is the biggest operation that we have in our application, so in the same situation as the CPU, optimizing this is our biggest priority.

The GPU works much better with independent calculations. The blocking system has been designed in GEMM so that each block does not need to talk to another block and therefore speed up our application. So if we can manipulate \(C = AB + C\), into independent
tiles, then we can have a fast GEMM. For example, if the size of the calculations is $M \times N \times K$, we can break up each block into $M_{\text{block}} \times K$ of $A$ and $K \times N_{\text{block}}$ of $B$, which would produce a $M_{\text{block}} \times N_{\text{block}}$ tile of $C$ [5]. We have now broken up into something that looks like this,

![Figure 3.5: GEMM GPU breakdown [5].](image)

Figure 3.5 shows a breakdown of the matrix in blocks, and validates the size of the blocks explained in the previous paragraph. The red block is acts as a slide that is sliding over the gray area. It represents the size of the block that is spawned when the kernel is called. So the slide is a for loop in the kernel, and the red is what is being calculated right now. The red block of size $M_{\text{block}} \times N_{\text{block}}$ in $C$ is what is calculated from the gray area of $A$ and $B$. The first thing that is done is put the sub matrices of $A$ and $B$ into shared memory. Since all of the threads will need to access them, it will be the quickest way to get the data to them. Then for each block, we take certain parts of the sub matrices and put them into register for each thread. Then when these are multiplied, it is accumulated into a register belonging to the same thread. Once all the calculations are done, the $C$ sub parts located in thread registers, are written to device memory [5]. The more advanced algorithms, like the one used in MAGMA, use texture memory. Texture memory is an available type of memory on the GPU, that allows for different memory access that are fast. So in the case of GEMM, we need to access one matrix in non-contiguous order, banding the memory to texture, can speed up that process. Instead of grabbing the data from device memory, it is grabbed from texture memory, but this does not change how the algorithm is run. In order to choose
what the dimensions of the sub matrices will be, $M_{\text{block}}$, $N_{\text{block}}$, and $K_{\text{block}}$, we need to take in consider some GPU hardware specifications.

1. warp size
2. threads per block
3. registers per thread
4. shared memory size.

We care about the warp size because this will define how many threads can be scheduled at once. And keeping the amount of threads in a block as a multiple of warp size can improve performance. The amount of threads per block is also important because it will help define the size of the sub matrices of $A$ and $B$. We care about how many registers per thread because this will be a big limiting factor in the size that $M_{\text{block}}$ and $N_{\text{block}}$ can be. If we use too many registers, they spill into what is called local memory. The name is deceiving because local memory resides in global memory, the slowest of all memories. Although this is mitigated by the emergence of L1 cache in Fermi architecture, it is bad GPU coding practice to spill registers. Using these guidelines, the GEMM function is tuned to our GPU, and performed at its highest performance.

### 3.4 Summary

Using these techniques, libraries, and hardware, we can create high throughput algorithms on the GPU. Although the GPU and CPU are different in architecture, algorithm design, fundamental optimizations, the idea is still the same; take what the device give you. Design the application around the hardware, and taking everything into account. The more that you can consider and weave into the application design, the faster the application will be. There needs to be a fundamental understanding on what is going on in the hardware, as much as software says you do not, in order to create optimized code.
Chapter 4

Matched Filter

This chapter will explain how we break down the MF algorithm for an efficient implementation. As explained in the Section 1.2, we get the algorithm from assumptions that are used to make this problem doable. We will define the algorithm itself, how it is done on the CPU and GPU, and where the designs change due to hardware. We will also go into what else could be done to improve the performance.

4.1 Algorithm

The Matched filter, Equation 1.10 can be broken down into five different components. Although these components vary in number of operations and complexity, any naïvely implemented component will greatly diminish the overall performance.

Figure 4.1: Matched-filter (MF) and Normalized Match-filter (NMF) components

Figure 4.1 shows a block diagram for both the MF and NMF. For this explanation, the reader needs to know that the data matrix $X_b$ is $\mathbb{R}^{n_p \times n_b}$ where $n_b$ is number of bands, $n_p$ is...
number of pixels, and \( \mathbf{u} = \begin{bmatrix} 1 & \ldots & 1 \end{bmatrix}^T \) and is contained by \( \mathbb{R}^{n_p \times 1} \)

\[
\mathbf{m}_b = \frac{1}{n_p} \cdot \mathbf{X}_b^T \mathbf{u} \tag{4.1}
\]

First we must find the mean of each band so that we can demean our data.

\[
\mathbf{X}_0 = \mathbf{X}_b - \mathbf{u} \mathbf{m}_b^T \tag{4.2}
\]

Now that the data is demeaned, we can calculate the covariance,

\[
\Sigma = \frac{1}{n_p} \cdot \mathbf{X}_0^T \mathbf{X}_0. \tag{4.3}
\]

As seen in Figure 4.1 the next step will be to whiten the data. Whitening of data can be performed by and any matrix \( \mathbf{A} \) such that \( \mathbf{A} \Sigma \mathbf{A}^T = \sigma^2 \mathbf{I} \) where \( \sigma^2 \) is arbitrary [15]. Usually, this is done by \( \Sigma^{-\frac{1}{2}} \), but this is not necessary. Since inverting a whole matrix is very expensive, we use the Cholesky decomposition which is discussed in detail in Section 1.2.1,

\[
\Sigma = \mathbf{L} \mathbf{L}^T \tag{4.4}
\]

where \( \mathbf{L}^{-1} \) is a whitening matrix.

\[
\mathbf{X}_w = \mathbf{X}_0 \mathbf{L}^{-1} \quad \mathbf{s}_w = \mathbf{L}^{-1} \mathbf{s} \tag{4.5}
\]

Since we have whitened data, the equation 1.10 can reduce to,

\[
y_{mf} = \frac{1}{\mathbf{s}_w^T \mathbf{s}_w} \cdot (\mathbf{X}_w \mathbf{s}_w) \ast (\mathbf{X}_w \mathbf{s}_w). \tag{4.6}
\]

This is the extent of the algorithm when it is broken down into parts. At first glance, the thought is that we will only need BLAS for the covariance computation, the Cholesky decomposition and the whitening steps. This is where the most computation occurs in MF so reducing this time will give us the best run time. We can also get some more performance out of some not so obvious tasks. Equation 4.1 and 4.2 are set up in a way that leads to explore BLAS routines. To give a summary of the algorithm floating point operations in the
same type of break down,

Table 4.1: gflows per instruction

<table>
<thead>
<tr>
<th>Function</th>
<th>gFLOP count CPU</th>
<th>gFLOP count GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>.01064</td>
<td>.01064</td>
</tr>
<tr>
<td>Mean Subtract</td>
<td>.01056</td>
<td>.01056</td>
</tr>
<tr>
<td>Covariance</td>
<td>.6868</td>
<td>1.363</td>
</tr>
<tr>
<td>Whitening</td>
<td>.00075</td>
<td>.00075</td>
</tr>
<tr>
<td>Matched Filter Scores</td>
<td>.01069</td>
<td>.0169</td>
</tr>
<tr>
<td>Total gFLOP</td>
<td>.719564</td>
<td>1.3959</td>
</tr>
</tbody>
</table>

These FLOP counts in Table 4.1 are found in [16]. The FLOP count for the GPU and the CPU are different because we use different BLAS functions for the covariance computation. The way to get performance on a GPU and a CPU are different. On a CPU, by minimizing the FLOP count and having an intelligent function design will yield the best implementation. On a GPU, by using more brute force meaning every thread doing the same thing, and every keeping functions simple yield the best performance. This is evident by the FLOP count on the CPU and GPU. Instead of having threads turning on and off to reduce the FLOP count on a GPU, we have all threads doing the same calculation. The real difference is in the covariance computation. On the CPU, we only calculate half of the covariance since it is a semi-definite symmetric matrix, whereas on the GPU, we calculate the whole covariance.

4.2 Results

The results of the MF algorithm were performed on both a multicore CPU and a GPU. The CPU is an Intel(R) Xeon(R) X5650 processor. It runs at 2.66 GHz, has six cores, and twelve
hardware threads. The total amount of on chip cache is 12 Mb.

The GPU used is a Tesla C2050 graphics card. It contains 3Gb of main memory, configurable 64Kb of shared/L1 cache memory. It has 448 CUDA cores each running at 1.15 GHz.

All results are optimized for a sensor that gives images of size $n_p = 40960$ and $n_b = 129$. All GPU functions are optimized for this configuration, and will need to be changed for different sizes.

### 4.2.1 Mean and Mean Subtraction

Mean and mean subtraction might seem like simple calculations. They do not require many computations and can be implemented easily with for loops. To illustrate the importance of speeding even the smallest components up, we can look at the three implementations shown below.

<table>
<thead>
<tr>
<th>Method</th>
<th>Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>For Loop</td>
<td>5.8ms</td>
<td>1.0x</td>
</tr>
<tr>
<td>openMP</td>
<td>4.1ms</td>
<td>1.4x</td>
</tr>
<tr>
<td>OpenBLAS</td>
<td>1.1ms</td>
<td>5.2x</td>
</tr>
</tbody>
</table>

You can see that we can save 4.7 ms by changing the mean calculation from a naive for loop to using OpenBLAS. **openMP** also gives some type of speed up but not to the extent of OpenBLAS which uses all of the optimizations explained in 2.1. This shows the power of data organization. There is not much of a difference in the number of FLOPs, but the data management, multithreading and SSE instructions allow for 5.2x speed up.

The same thing can be said about the 4.2 equation. By changing it to a matrix rank-1
update, where $\mathbf{u}$ and $\mathbf{m}_b$ create a matrix, which is then subtracted from $\mathbf{X}_b$, save us more time. It solidifies the fact that it is not the speed of the processor, it is the optimization techniques discussed in 2.1.

On the GPU, the process of working to get efficient implementations of 4.1 and 4.2 is different. When looking at these equations, set up in linear algebra fashion, our initial thought is to send the data over to the processor, and then do two BLAS routines, `cublasSgemv()` and `cublasSger()`. These are a matrix vector multiplication and a rank-1 update of the data matrix. Looking at the timeline produced by Nsight, a GPU profiler,

![Figure 4.2: cuBLAS implementation of Equations 4.1 and 4.2.](image)

In Figure 4.2, we can see the breakdown of timing with the memory transfer, `cublasSgemv()` and `cublasSger()`. Knowing about streams from Section 3.3 we can see that we are not utilizing the compute stream while data is being transferred. Since computing the mean for each band is completely independent, we can send over bands separately. This would allow us to overlap the memory transfer and computations. So we decided to stream some of the data, start calculating the mean, while more of the data is being transferred over. The first thought was to transfer it in a band at a time, compute the mean of that band using shared memory with a decomposition technique in [6]. This summation technique is graphically explained below.
The vectors are contained in shared memory. Each iteration we are cutting the number of calculations in half. To map this to our data, we use the $n_p$ threads. We cannot put that many threads into one block, so multiple blocks are also used. We put a section of the spectral band into each blocks shared memory, and sum that section with design ?? Once the sum is calculated, each block uses the `atomicAdd()` function to sum and get the mean of the spectral band. This was run with 8 streams, sending one spectral band at a time.

![Diagram showing memory overlap attempt]
This did not work very well. We were able to overlap some of the transfers, but it seems to be very scattered. Also the kernel itself is taking longer than the memory copy. To be completely efficient we want the kernel to be at most as big as every transfer, so that we can completely overlap them without any extra time. So, we decided to look at the kernel. With this kernel design, we need to rely heavily on shared memory. This is not necessarily a bad thing, but it somehow we could get it into registers, the faster the kernel would run. So instead of using \( n_p \) threads, we use less threads. This allows us to use coalesce memory reads and make partial sums in each thread. From there, each thread puts its partial sum into shared memory. Now, we have a vector of partial sums in shared memory. From here, we use the same summation technique seen in Figure ??.

![Figure 4.5: Second memory overlap attempt. 256 threads and 20 blocks were spawned](image)

One thing to notice is that we are using 140 less blocks, so \( 140 \times 256 = 35840 \) threads that we are not using anymore, but the speed of the kernel is faster. It is just another example on the GPU that we do not want to just have as many threads as we can, and putting some more thought into the memory systems, the faster the kernel can be. We again can optimize this even more. Notice that in Stream 12, Stream 13, and Stream 19, we do not have overlapping. Consulting [13], the ordering of when the kernels are called matters.
We want to fill the compute and transfer stream queues before moving on to the next one respectively. So, instead of calling a `memcpyAsync()` followed by `computeMean()` kernel, we called all the `memcpyAsync()`s followed by all of the `computeMean()` kernels.

![Figure 4.6](image)

**Figure 4.6:** Third memory overlap attempt. 256 threads and 20 blocks were spawned.

Now, looking at Figure 4.6 we can see we are overlapping to the best of our ability. This is after an initial startup, and more toward the middle of the for loop. This still was not as good as the BLAS operations, and we have not implemented the mean subtraction yet, but it seems that this is the way to go. Since there are many ways to complete this task, we decided to look at a different approach. This was to send over many bands at a time, to get a bigger transfer, and then spawning a block per band, and using coalesce memory reads, and getting rid of the shared memory aspect of our kernel. Now we just accumulate pixels in threads, and then use atomic functions to add all of those values. The block spawned take care of the spectral bands sent over depicted below.
Each block is assigned to one spectral band, and threads accumulate pixels in a coalesce fashion. Once all of the pixels have been added to their respected thread in a block, then we use atomic functions to sum all partial sums that each thread contains. So we send the data over in thirds, and add in the mean subtract.

Figure 4.8: Fourth memory overlap attempt. 256 threads and 43 blocks were spawned

Finally putting everything together, meaning adding some more streams, and making the kernels overlap as best as possible.
Figure 4.9: Fifth memory overlap attempt. 256 threads and 16-17 blocks were spawned.

Table 4.3: Timing attempts for initial data copy

<table>
<thead>
<tr>
<th>Method</th>
<th>Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>cuBLAS</td>
<td>6.38 ms</td>
<td>–</td>
</tr>
<tr>
<td>Attempt 1</td>
<td>10.25 ms</td>
<td>.62x</td>
</tr>
<tr>
<td>Attempt 2</td>
<td>6.07 ms</td>
<td>1.05x</td>
</tr>
<tr>
<td>Attempt 3</td>
<td>5.68 ms</td>
<td>1.12x</td>
</tr>
<tr>
<td>Attempt 4</td>
<td>5.63 ms</td>
<td>1.13x</td>
</tr>
<tr>
<td>Attempt 5</td>
<td>5.18 ms</td>
<td>1.23x</td>
</tr>
</tbody>
</table>

Our final implementation has saved us \( \approx 1 \text{ ms} \) from our total GPU implementation. When trying to squeeze out all of the performance that we can, this is the type of exploration that is needed.

4.2.2 Covariance

The covariance computation is what dominates all of the calculations for the MF. It is a third order BLAS operation with the highest FLOP count. It was very clear from the beginning
that BLAS was going to be needed for this calculation because of the amount of optimization that comes with this popular routine.

First, on the CPU, we had OpenBLAS use `cblas_sgemm` which is the generic matrix multiply. We see a timing result of 19.3 ms for the computation. This is computing the whole covariance matrix, where we only need an upper or lower triangular version of the covariance. This is because covariance matrices are conjugate symmetric. Since we only need an upper triangular matrix, we changed the function call to `cblas_ssyrk()`. This `cblas_ssyrk()` BLAS function which does $C = \alpha \cdot A^T A$ seems to be better in our case because we can now put more of $A$ in closer caches. By putting $A$ closer to the processor, we can now compute the covariance faster. So, not only does this function cut the FLOP count in half, but it also allows for better memory use. When computing using `cblas_ssyrk` we get 8.9 ms which is $\approx 2.5x$ better than the `cblas_sgemm()` call.

For the GPU we have a different story on how to get the covariance matrix. Since there is such a difference in performance with minor changes to an algorithm, we need more trial and error. We looked at three different implementations of the covariance computation.

<table>
<thead>
<tr>
<th>Method</th>
<th>FLOP count</th>
<th>Run time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cublasSsyrk</td>
<td>.686</td>
<td>31.917</td>
</tr>
<tr>
<td>cublasSgemm</td>
<td>1.36</td>
<td>13.302</td>
</tr>
<tr>
<td>magmablas_sgemm_fermi64</td>
<td>1.36</td>
<td>11.48</td>
</tr>
</tbody>
</table>

Table 4.4 shows us that FLOP count matters little in this situation. What matters more is the blocks, threads, and algorithm design. `cublasSsyrk()` does not have a good design, so even though the FLOP count is half, the run time is over double what the `cublasSgemm` and `magmablas_sgemm_fermi64()` functions run times are. Since they are calculating the
same thing, we looked into the design of each of these to see why there is a discrepancy.

Table 4.5: Functions Investigated

<table>
<thead>
<tr>
<th>Function</th>
<th>cublasSsyrk</th>
<th>cublasSgemm</th>
<th>magma_sgemm_fermi64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Kernels</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Grid Size</td>
<td>[1,1,1]</td>
<td>[3,3,1]</td>
<td>[3,3,1]</td>
</tr>
<tr>
<td>Block Size</td>
<td>[64,4,1]</td>
<td>[8,16,1]</td>
<td>[64,4,1]</td>
</tr>
<tr>
<td>Registers per Thread</td>
<td>63</td>
<td>63</td>
<td>50</td>
</tr>
<tr>
<td>Shared Mem. Used</td>
<td>12.4Kb</td>
<td>8.5Kb</td>
<td>8.3Kb</td>
</tr>
<tr>
<td>DRAM Utilization</td>
<td>1.4 Gb/s</td>
<td>3.03 Gb/s</td>
<td>3.47Gb/s</td>
</tr>
</tbody>
</table>

To preface this discussion, in the `cublasSsyrk()` function call, the information is based on the longest kernel counting for 21ms of the 31ms of the total run time. The first red flag that goes off with this is the in the `cublasSsyrk()` function, there is three different kernels called. Right here we know that `cublasSsyrk()` has to reload all registers and shared memory for each kernel. This amount of memory movement does not produce efficient run times. We also see that in `cublasSsyrk()` function that we are using far less registers than in the `cublasSgemm()` and `magma_sgemm_fermi64()` which causes it to use more shared memory. Shared memory is slower than local registers available to threads. The design of `cublasSsyrk()` indicates from this information that threads need to communicate more than the others, which again is why the run time is much slower. It is evident here that this is a memory bound function, and needs better memory organization.

As for the other two functions, the design is very similar. We are using the same amount of threads per block, but organized in a different manner. When we apply our data to these functions, which is a matrix of size $n_p \times n_b$, where $n_p >> n_b$ it becomes evident why `magma_sgemm_fermi64()` function out performs the `cublasSgemm()` function. Since we
are launching $[64,4,1]$ threads per block, and the nature of our data being a tall skinny matrix, the threads map better for memory access in the $[64,4,1]$ pattern than the $[8,16,1]$ pattern.

One thing to notice is that the GPU gets out performed by the CPU in terms of the Covariance computation. It was an interesting result because with all of the extra execution units that the GPU has, at first thought we would think that this would give us a huge advantage. The reality is that there is not, and most of the performance is the memory organization and how fast we can get the data to the processors, making this function memory bound. Since the L1 cache is relatively small in the GPU, we cannot stream enough data into the register/shared memory/L1 cache that we can keep our execution units busy.

### 4.2.3 Whitening

For whitening stage, equations 4.4 and 4.5, need to be looked at. For the matched filter, the numerator is $s^T \Sigma^{-1} X$, and the denominator is $\sqrt{s^T \Sigma^{-1} s}$. If we were just to do this outright without any thought, we would have a couple matrix multiplications, which as explained above, is an expensive operation to do. We know that we are going to do the Cholesky decomposition, so the numerator become $s^T L^{-1} L^{-T} X$. Instead of doing matrix multiplications, first we can solve $Lb^T = s^T$ which solves the left side, and then solving $L^T h = b$. Then we can multiply $hX$. Instead of doing matrix multiplications, we only do matrix vector multiplications, and triangular solves for inverses. This will cut down on algorithm complexity and improve run time. Since the reduction in calculations here is great, we used this on both the CPU and GPU. It also means we are not exactly whitening the data and signature, but since we are still getting the correct answer, and the point is to be faster, this procedure makes more sense. We also see that $h$ is involved in the denominator. The denominator becomes $\sqrt{h^T h}$. So, instead of doing matrix-vector operations, we can now do a dot product, as well as using the result to be put into $\alpha$ in BLAS operations, meaning
it is divided through without an explicit function.

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>.21 ms</td>
</tr>
<tr>
<td>GPU</td>
<td>3.9 ms</td>
</tr>
</tbody>
</table>

Table 4.6: Timing results for inverse

In Table 4.6 the CPU out performs the GPU by over 10x. This is because to do matrix decompositions, there needs to be a lot of thread communication. Since the GPU is not design for this, it needs to break the Cholesky decomposition into multiple kernels. As explained in the \texttt{cublasSsyrk()}, functions that have multiple kernels add unnecessary run time. Since the Covariance is so small, a $\text{bands} \times \text{bands}$ matrix, a memory transfer to the CPU to compute the Cholesky decomposition and find $\mathbf{h}$ is more efficient. So we came to the conclusion that doing a memory transfer, solving the system and then copying the data back would be the best for the GPU, meaning the CPU does get involved with the GPU version.

4.2.4 Complete Results

There is just now, a dot product, a matrix vector multiply, and element by element square operation. The dot product and squaring are negligible parts of the algorithm in terms of timing, and the matrix vector multiplication has been discussed above. So our final timing results for both the CPU and GPU are as follows.
Table 4.7: Timing results for matched filter

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Time</th>
<th>Speed up</th>
<th>gFLOPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>13.2 ms</td>
<td>–</td>
<td>61.04</td>
</tr>
<tr>
<td>GPU</td>
<td>18.68 ms</td>
<td>.7x</td>
<td>74.68</td>
</tr>
</tbody>
</table>

As we can see from this Table 4.7, the GPU is not faster than the CPU in calculating the matched filter. There are a couple reasons for this. One of them being the memory transfers adding time into the equation, especially with such highly optimized algorithms. It is hard to make up this transfer time. Also the GPU is a fairly new architecture in the scientific community, and is not fully as understood as well as the CPU. The CPU has been optimized over the last 40 years, and getting perfect algorithms and compilers to fit to certain CPUs is a clear advantage. This is also not a sequential to parallel comparison. Our CPU version is not only using multiple threads, but using multiple cores as well to complete these task, and is able to run efficiently. If we were to compare each of these to their MATLAB implementations, using the parallel toolbox for the GPU calculations, we can see that we beat them by a large margin.

Table 4.8: Timing Results of CPU and GPU version vs. MATLAB

<table>
<thead>
<tr>
<th>Software</th>
<th>Timing</th>
<th>Speed Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>MATLAB CPU</td>
<td>138 ms</td>
<td>–</td>
</tr>
<tr>
<td>C++ CPU</td>
<td>13.2 ms</td>
<td>10x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Software</th>
<th>Timing</th>
<th>Speed Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>MATLAB GPU</td>
<td>95 ms</td>
<td>–</td>
</tr>
<tr>
<td>CUDA/C++ GPU</td>
<td>18.68 ms</td>
<td>5x</td>
</tr>
</tbody>
</table>

Table 4.8 shows MATLAB vs. our implementations. As you can see, we achieve a much higher performance in lower level coding than MATLAB, even though MATLAB is using a BLAS implementation for both the CPU and GPU. The most dramatic improvement is on the CPU, with a 10x speed up. By allowing more diverse coding practices than MATLAB
allows, and utilizing all that the processor can give us, we are able to gain squeeze out more performance with the same architecture.
Chapter 5

Normalized Matched Filter

The normalized matched filter is just some added steps to the matched filter, as shown in 4.1. There is an extra step by doing the mahalanobis distance calculation and normalizing each matched filter score. Although this is only an additional step, we need to change some of the algorithm to get the most efficient implementation.

5.1 Algorithm

Up until the whitening step, we have an identical beginning in both the MF and NMF. Although there is only one added step in the NMF case, it will complicate the implementation of the algorithm discussed below.

\[ y_{\text{NMF}} = \frac{y_{\text{MF}}}{x_w^T x_w} \]  \hspace{1cm} (5.1)

where \( x_w \) is a whitening pixels contained in \( X_w \). We need to change the way the whitening step is now done because we cannot get away with just matrix vector operations. The Mahalanobis distance makes it impossible to implicitly calculate \( X_w \) as done in Section 4.2.3. This in turn causes us to have a higher FLOP count.
Table 5.1: gFLOP count per instruction

<table>
<thead>
<tr>
<th>Function</th>
<th>gFLOP count CPU</th>
<th>gFLOP count GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>.01064</td>
<td>.01064</td>
</tr>
<tr>
<td>Mean Subtract</td>
<td>.01056</td>
<td>.01056</td>
</tr>
<tr>
<td>Covariance</td>
<td>.6868</td>
<td>1.363</td>
</tr>
<tr>
<td>Whitening</td>
<td>.6835</td>
<td>.6835</td>
</tr>
<tr>
<td>Matched Filter Scores</td>
<td>.01069</td>
<td>.0169</td>
</tr>
<tr>
<td>Mahalanobis Distance</td>
<td>.0159</td>
<td>.0159</td>
</tr>
<tr>
<td>Normalized Matched Filter Scores</td>
<td>.000081</td>
<td>.000081</td>
</tr>
<tr>
<td>Total gflop</td>
<td>1.417</td>
<td>2.0941</td>
</tr>
</tbody>
</table>

Table 5.1 shows the difference in CPU and GPU FLOP count. The reason the GPU and CPU have different FLOP count is discussed in Section 4.1.

5.2 Whitening

As discussed above, the NMF changes from the MF starting at the whitening step. We have two choices in completing this step. The first of these choices is using solving algorithms for both the data matrix and the signature. We know from the MF, that using the solving system will be better because we are only solving for a vector. In the data matrix case, we are solving for a matrix, therefore each equation has multiple unknowns. We need to check which way will be easier for each hardware. Solving will work well on the CPU because it gets rid of the hassle of computing an inverse, and the numerical considerations that come along with doing an inverse. We would also not have to do another matrix multiply to whiten the data. On the GPU, it seems that it would not be a good idea to do a solving algorithm.
because of the amount of communication needed. In fact, it takes much longer on the GPU to do these solves, than just taking the inverse and then whitening the data and signature. We check each way of doing the whitening step with the GPU and CPU.

**Table 5.2: Timing Results Whitening versions**

<table>
<thead>
<tr>
<th>Whitening</th>
<th>Timing</th>
<th>Whitening</th>
<th>Timing</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU W/ Solving</td>
<td>5.1 ms</td>
<td>GPU W/ Solving</td>
<td>22.4 ms</td>
</tr>
<tr>
<td>CPU W/ Inverse</td>
<td>4.3 ms</td>
<td>GPU W/ Inverse</td>
<td>8.5 ms</td>
</tr>
</tbody>
</table>

In Table 5.2 we see that there is a difference from CPU to GPU. The GPU does not have a great capacity to have many threads communicate with each other. On the CPU, we can have all of our threads communicate well because a CPU is designed differently. The GPU is not made for communication, and does much better when things are independent of each other. This is why we see the GPU having a big difference between the solving and the matrix operations. We also see that the multicore CPU beats the GPU in both aspects. This is in part because of the memory bound matrix operations on the GPU.

### 5.3 Mahalanobis Distance and NMF scores

These final two steps in NMF are done differently on the CPU and on the GPU. On the CPU, we do not want to do \( n_p \) dot products. This would not utilize our processor efficiently. To design the Mahalanobis distance, we first use **openMP** to square each element in \( X_w \). Then making a similar vector to \( u \), \( r = [1 \ldots 1] \) where \( r \) is of size \( \mathbb{R}^{1 \times nb} \). Having both \( r \) and \( T = X_w \ast X_w \), we can set up a matrix vector equation that computes all mahalanobis distances. After the Mahalanobis distance is computed, we need to compute the NMF scores. Since it is just a divide between \( y_{MF} \) and \( x_w^T x_w \), we can use **openMP** to break up the divisions.

58
Each thread is given an equal section of the divisions to do. This will complete the normalized matched filter.

On the GPU, we are able to the Mahalanobis distance and NMF scores in one kernel. For the Mahalanobis distance, we need to do a dot product of every pixel. This means that every pixel is independent of every other pixel. If we use \( n_p \) threads, all of the dot products can be computed simultaneously. We do not need to Mahalanobis distance specifically, we just need to it scale the \( y_{MF} \) scores. By keeping the mahalanobis distance in a register of each thread, we can easily do the division in the same kernel. These are the types of functions that map well to the GPU because it has many independent parts.

5.4 Complete Results

Taking most of the MF results and using these different steps, we are able to get the NMF scores for our data with a certain signature. These results are similar to the MF because of the fact that we are just adding some more functionality to the MF.

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Time</th>
<th>Speed up</th>
<th>gFLOPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>22.6 ms</td>
<td>–</td>
<td>62.73</td>
</tr>
<tr>
<td>GPU</td>
<td>28.5 ms</td>
<td>.8x</td>
<td>73.4</td>
</tr>
</tbody>
</table>

The timing results are very similar to the results shown in Table 4.7, in that the CPU beats the GPU. One reason this could be the case is that there is not enough work for the GPU to do. The run time on these algorithms are small, so memory transfers take up a good percentage of run time on the GPU. When there is more work, the GPU could in fact beat the multicore CPU.
Figure 5.1: Timing breakdown CPU vs. GPU

Figure 5.1 shows the timing breakdown of each step comparing all of the steps in NMF calculations. We are heavily dominated by the covariance and whitening steps of the algorithm. These are the two steps that contain the highest complexity of functions. We also notice that these are the two steps that the GPU loses to the multicore CPU, so to continue this work, we would focus our attention on the whitening and covariance computations. This also does not include the memory transfers to get the data to the GPU because we do not have anything to compare it to. The compute mean and mean subtract timing for the GPU are the effective timing results, since we do some of the computations while the data is being brought to the GPU, it is a masked computation.

Our implementations relay heavily on open source BLAS libraries, which is great for the CPU because it is able to accommodate most matrices. On the GPU on the other hand, most of the BLAS libraries are optimized for square matrices. Since we are far from a square matrix, we are losing some performance due to the nature of our data. This was a bit of
a surprise when we saw the results, the multicore CPU does a better job performance wise than the GPU. It did show us that the way memory is moved has a clear importance over the computations themselves. By not reducing the FLOP count by very much and doing some better data movement, we can greatly affect our performance, especially with linear algebra.
Chapter 6

Sequential Maximum Angle Convex Cone: Endmember Extraction

The next algorithm that we have looked at is the Sequential Maximum Angle Convex Cone (SMACC). SMACC is a different exploitation of HSI because it tries to find materials with the data itself, and no external library. It was chosen for investigation because it does not include a covariance computation, which was the bottle neck with MF and NMF. In order to find materials in SMACC, we find what are called end members. An end member is defined as a pixel that is made up by only one material. For example, there is a building surrounded by grass. The pixels that are on the edge of the building and grass with contain radiant energy from the grass and from the building. This would not be considered an end member. A pixel that is located in the middle of the building would be considered an end member.

Figure 6.1 shows a graphical interpretation of how SMACC is decomposing the pixels in our image. Each of the green pixels can be made up as a linear combination of the two end members, $s_1$ and $s_2$. Another thing to notice is that we constrain our problem. First of them being the non negativity constraint. This is used because we cannot have a negative combination of $s_1$ and $s_2$ to make up pixels. The red pixel could then become an end
Figure 6.1: SMACC End members

member as SMACC continues to iterate. The next constraint is the convex hull constraint. This would mean that when we make up a pixel in terms of a linear combination of end member, we make the combination coefficients add to one. We do not use this constraint because we need to account for some error in our image.

6.1 Algorithm

As a prerequisite to doing SMACC, we need to model the pixel. Note that for this discussion, the HSI image $X$ discussed earlier will be $X^T$. By modelling our pixels, we can show some insight to how SMACC from [7] works. First we model the HSI image by pixels

$$x_i = Sa_i + e_i, \; i = 1, \ldots, N$$  \hspace{1cm} (6.1)

where $x_i$ is a pixel in our image. $S$ is the spectral end members found by SMACC. $e_i$ is the error matrix of the left over energy. $n$ is the number of pixels and $a_i$ is the abundance value of each end member in $S$. In other words, $a_i$ are the fractions of each end member that is in $x_i$. Due to the nature of SMACC, there needs to be a way to index pixels, bands, and
iteration, so we introduce

\[ x_{i,j}^{(k)} \]  

(6.2)

where \( i \) indexes pixels, \( j \) spectral band index, and \( k \) is the iteration number. We will define \( N \) as the total number of pixels, \( M \) is the total iterations, and \( b \) is the number of bands. When we reference just one of those symbols, it represents a vector. For example,

\[ x_i \text{ or } x_j. \]  

(6.3)

where \( x_i \) is explained above, and \( x_j \) references a spectral band. To begin SMACC we find the euclidean norm value for each pixel

\[ \| x_i \|^2, \ i = 1, \ldots, N \]  

(6.4)

Next, we find the max index of these norms to choose our endmember.

\[ I = \max_i (\| x_i \|^2) \]  

(6.5)

\[ w = x_I \]  

(6.6)

In our first iteration, we find the maximum norm value because this pixel contains the most energy. Since it contains the most energy, it has the best chance of containing only one material. We then project each pixel onto \( w \)

\[ z_i = w^T x_i \]  

(6.7)

This projection is shown below.

![Projection of pixels](image)

**Figure 6.2:** Projection of pixels
We project all of our pixels onto the newly selected end member so that we can subtract the energy $w$ has. Before we can subtract the energy, we need to check a constraint that the abundances $A$ have to be non-negative. When we have this constraint, we need to introduce an oblique projection. Oblique projections is used in the constrained case of SMACC [7].

![Oblique projection illustration](image)

**Figure 6.3: Oblique projection illustration [7]**

The oblique projection is shown in green in Figure 6.3. This differs from an orthogonal projection (red) in that it is used to make sure that the abundance for the pixel $x_i$ will always be positive. To decide if the oblique projection will be needed, we compute,

$$v_{i,j} = \frac{a_{i,j}}{a_{I,j}z_i}$$

where $i = 1, \ldots, N$ and $j = 1, \ldots, k - 1$ and $I$ is the pixel index of the newest endmember. Since SMACC is an iterative algorithm which finds new endmembers each iteration, the abundance value from previous endmembers must be updated to accommodate new end members. We constrain our problem to have all positive abundance, so we need to make sure that when previously found abundances are updated, they do not become negative. We use oblique projections to ensure this constraint. In order to see if the constraint is active, we calculate,

$$y_i = \min(v_i)$$
and define $\alpha$ as

$$\alpha = y_j : \text{if } y_j < 1$$

(6.10)

$$\alpha = 1 : \text{if } y_j > 1.$$  

(6.11)

$\alpha$ tells us whether the non negativity constraint is active or not. We can now get our abundance values for the newest endmember,

$$a_{i,j} = \alpha z_i$$

(6.12)

where $i = 1, \ldots, N$ and $j = k$. Once we have our new abundances, we can subtract the energy contained in all the pixels from $w$ and update the previous abundances accordingly.

$$x_i = x_i - a_{i,k}w$$

(6.13)

$$a_{i,j} = a_{i,j} - a_{I,j}a_{i,k}$$

(6.14)

where $j = 1, \ldots, k - 1$ and $i = 1, \ldots, N$. By completing SMACC, we are given end members in an HSI image that can be thought of as materials. We can then use these end members to run detection algorithms, such as MF and NMF to see where they are in the image. Combining SMACC and a detection algorithm can lead to a fully autonomous detection system.

### 6.2 Results

This is a longer algorithm in terms of run time than MF and NMF algorithms looked at in previous chapters. This bodes well for the GPU because there is inherently more work to be done. When SMACC is looked at carefully, it can be broken down into $n_p$ independent parts, which again bodes well for the GPU. The data cube that is used for these SMACC simulations was one of size 640,000 pixels and 42 spectral bands.
6.2.1 CPU

We have already explained what OpenBLAS and openMP are able to do, we decided to use these tools in our design for SMACC. Table 6.1 shows that either openMP or openBLAS was used in every step in SMACC. The first step of SMACC is the Equation 6.4. This step is exactly the same as the mahalanobis distance in Chapter 5, so we used the same approach. The next step is Equation 6.2. Although this equation is shown as separate pixels, we can use a matrix vector multiplication to compute them all together using cblas_sgemv() in openBLAS.

The following step is computing the matrix $V$. Since each $v$ vectors are independent computationally, using openMP would be the best option. We break up our data into equal sections with each section being given to a thread that openMP opens. The problem is that when we open threads, there is some overhead. We do not want to keep opening and closing them, so it is better to keep them open as long as possible. Instead of computing $V$, then updating $X$ and $A$, we compute $V$ and update $A$ in the same openMP function. This means we can keep threads open and avoid the overhead. The problem is that now we need to keep track of not only $A$, but also the previous $A$ matrix. It is a small price to pay for the improved performance.

The design is that each thread is assigned a certain number of coefficients that it is assigned to. Each thread works independently of the others, computing its section of $V$, and updating its section of $A$. It is a tiling scheme where the tiles are divided evenly.

### Table 6.1: CPU SMACC breakdown

<table>
<thead>
<tr>
<th>Step</th>
<th>Library Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Norms</td>
<td>openMP</td>
</tr>
<tr>
<td>Projection</td>
<td>openBLAS</td>
</tr>
<tr>
<td>Compute $v_j$</td>
<td>openMP</td>
</tr>
<tr>
<td>Update Pixels</td>
<td>openBLAS</td>
</tr>
<tr>
<td>Update $A$</td>
<td>openMP</td>
</tr>
</tbody>
</table>
The only part left is to update the pixels abundances to the the new ones as shown in equation 6.13. This is the same type of update used in MF and NMF to subtract the mean from \( X \). So we decided to use the same OpenBLAS technique as before.

As you can see there is a good mix between vectorized openMP and using BLAS operations. SMACC is not a very linear algebra intensive algorithm, but can be parallelized with openMP. When we implement it in MATLAB and on the CPU with optimizations:

<table>
<thead>
<tr>
<th>Software</th>
<th>Time</th>
<th>Speed up</th>
</tr>
</thead>
<tbody>
<tr>
<td>MATLAB</td>
<td>25.26 s</td>
<td>–</td>
</tr>
<tr>
<td>optimized C++</td>
<td>2.403 s</td>
<td>10x</td>
</tr>
</tbody>
</table>

Table 6.2: Timing results for SMACC finding 20 endmembers

By adding multithreading, and BLAS optimizations we are able to greatly improve our performance in terms of run time. SMACC is affected by parallelism because of its nature of having independent parts. It can map to multiple independent sections easily which gives it good performance with openMP.

### 6.2.2 GPU

SMACC is an algorithm that fits very well to the model of the GPU. We are able to break it up into pixels that are independent of each other. With the massive amount of execution throughput the GPU allows, we can fill all of the SMs with work, and get a better throughput. When doing these calculations, we found it useful to use \( n_p \) threads. This is not only for simplicity of algorithm development, but we found that using registers and no shared memory in the areas where we needed to sum values or compare values is more efficient.
Figure 6.4 shows a representation of how we compute the norms for each pixel. With this implementation, we can bypass shared memory and keep every thread completely separate, which is the best case for the GPU. After finding the norm of each pixel, we need to search for the maximum of each norm. This is done using cuBLAS, the function call `cublasIsaMax()`. It is a tricky thing to do on a GPU because it takes a lot of communication and comparing values. The projection step, equation 6.7 is just a matrix vector multiplication that is explained in the Section 6.2.1. The next step is to find $\mathbf{V}$. Similarly to the CPU, on the GPU, we compute $\mathbf{v}_j$ and $\mathbf{A}$ in the same kernel to make sure we do not launch many kernels. Our design of updating $\mathbf{A}$ is as follows. First we need to find $y$, so we need to calculate all of the $v_k$s. We open $n_p$ threads, each thread contains a vector $v$ with $k$ elements and calculates all of the $v$ values. Once $v_{\text{min}}$ is found, we can get the newest $\mathbf{a}_i$ vector as well as update the previous $\mathbf{a}_i$s quickly in the same kernel. Then all we need to do is update the $\mathbf{X}$ matrix, which is a rank-1 update, and done with a `cublasSger()` function call. All of this leads to a performance time against the CPU,
Table 6.3: Timing results for SMACC finding 20 endmembers CPU vs. GPU.

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Time</th>
<th>Speed up</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>2.403 s</td>
<td>–</td>
</tr>
<tr>
<td>GPU</td>
<td>0.400 s</td>
<td>6x</td>
</tr>
</tbody>
</table>

Table 6.3 shows because the run time is longer, and there is more work to do, the GPU performs better. With the amount of computing power the GPU processes, it easily scales to this magnitude of a problem. The CPU however, cannot just use as many threads as it wants to improve performance. Each thread is given a lot more work to be done which leads to longer run times. The timing result includes the time it takes for the memory transfer from CPU to GPU to occur.

In SMACC, we have not relied so heavily on the open source BLAS libraries that we had for MF and NMF. This was more of defining finding out how the data needs to be manipulated, and In SMACC, we have not relied so heavily on the open source BLAS libraries as we had for MF and NMF. This was more of finding out how the data needs to be manipulated, and going about it in the best way. This was much more of a singular pixel manipulation than seen in MF and NMF. A way that these can be used together would be to find the end members associated with an image, find out what material they were by comparing them to the library of signatures, and running the NMF or MF algorithms to detect where in the image that the material is located. SMACC also shows us that the GPU, if given the correct type of algorithm, can outperform a multicore CPU by a large margin.
Chapter 7

Conclusions

The subset of applications that find remote sensing of chemical clouds useful will grow as hyperspectral imaging sensors advance technologically. The particular application that this thesis is concerned with is a system where chemical clouds can be detected tracked in real time. The results presented here show these real time systems can be a part of the useful application subset, and that it can be done using commercially available processors.

7.1 Summary

In this thesis, we go over how to efficiently implement hyperspectral detection algorithms using a multicore CPU and a GPU. Although both have parallel architectures, there are differences which affect their performance in the algorithms investigated.

The first two of these HSI detection algorithms investigated were the matched filter and the normalized matched filter. We found that the multicore CPUs performance is better than the GPUs performance despite the extra computing power the GPU has. This is because it is not only about the amount of computing power that a processor has but more about how the memory is organized, and how fast we can get the data to the processor.
We were surprised that the CPU beat the GPU on both the matched filter and normalized matched filter in terms of run time. Our initial thought was since the GPU has much more computing power, and our problem has lots of computations, our GPU would achieve a shorter run time. We quickly found that almost all functions on GPUs are memory bound. This means that the computations do not slow the GPU, but the memory slows the GPU. We also found that there are almost infinite ways to implement the same function onto a GPU. We found that with different block and thread combinations, kernels that compute the same thing could have vastly different run times. Optimizations like coalesce memory reads, and texture memory can have the biggest impact on a kernels performance. In the GPUs covariance computation we found that even adding unnecessary computations can improve run time on a GPU, as long as it keeps threads independent.

For the final HSI algorithm investigated, SMACC, we observed the opposite in terms of run time. The GPU beat the CPU by a large margin. The biggest different between matched filter detection and SMACC is that there is no covariance computation. In SMACC, each pixel is independent of every other pixel. This is where we found that keeping threads on a GPU independent is the way to achieve the best performance. We learned that the best way to access the GPUs computing power is to have many independent parts. No matter how many parts there are, the GPU can answer the problem with more computational threads.

7.2 Future Work

We do not believe that we achieved the best possible performance on the GPU for any of the algorithms due to the infinite amount ways to implement them. Some things that can still be investigated would be a QR matrix decomposition approach to the matched filter detectors. This would not be operationally efficient for the GPU, but could reduce the run time. Also, if this were to be implemented for a real time system there would be a stream of
data cubes coming from a sensor. Using the data streams, the initial memory transfer from CPU to GPU could be masked, which could lead to practical use of a GPU. We had also mentioned that the more work the GPU is given the better it would perform when compared to the CPU, so we could scale this problem and look for multiple gases in one cube. Putting this all together could lead to a better implementation than the one presented in this thesis.
Bibliography


