A Massively Parallel Algorithm for Cell Classification Using CUDA

by

Samuel Schmidt

A Thesis Submitted to the Graduate School of the University of Cincinnati

in partial fulfillment of the requirements for the degree of

Master of Science (Computer Science)

in the Department of Electrical Engineering and Computing Systems of the

College of Engineering and Applied Science

Committee Chair:

Fred Annexstein, Ph.D.

Abstract

In Bioinformatics, Cell classification is the act of separating human cells into different groups based on their RNA-seq expression levels. These data can be quite large, as there are about 20,000 known human genes. Even relatively small datasets (¡1000 cell samples) will contains millions of values. Computations and classifications on this data force a choice: speed or thoroughness? Many scientists today choose speed - this means they must reduce the dataset through feature selection or some similar approach. Others may use complex computing systems to achieve thoroughness, but the cost of these systems is high. NVIDIA CUDA allows the average researcher to perform cell classification without making this choice. CUDA, running on specialized GPUs, gives desktop computers the power of a large computing system and eliminates the need to reduce the complexity of the dataset.

Acknowledgements

Jeremy Cox, project creator.

David Moneysmith, team member during our Parallel Computing class.

Dr. Fred Annexstein, Parallel Computing professor and thesis advisor.

Contents

Chapter 1

Overview

The cell is the basic unit that makes up all living things. Biologists find it helpful to classify cells by form and function using various criteria. However, cell behavior is highly flexible and adaptive; even two cells of the same type may perform different tasks. Similarly, consider two construction workers: one may be pouring concrete while the other does carpentry. Still, they are of the same type [1].

According to the central dogma of molecular biology, DNA is copied to produce RNA, which in turn is used to produce proteins. A cell's behavior can be predicted by the proteins it produces. However, these proteins cannot be measured directly. Instead, we measure the amount of mRNA in a given cell [10]. This is referred to as the expression level for a given gene and is used as an indicator of the amount of a given protein being produced by a cell. Therefore, by using expression levels as a stand in for protein production we

can determine a cell's behavior and classify its type.

The genomics revolution has dramatically increased our ability to measure cell behavior, but processing these measurements quickly continues to be a challenge. In this paper we demonstrate how a parallel computing paradigm called NVIDIA CUDA can to bring the necessary computing power to classify cells based on RNA-expression level to a desktop system.

Chapter 2

Introduction

The current estimate of the number of genes in the human genome is about 20,000 [8]. This means that genomic data sets can easily have millions of data points. The hardware on a modern desktop computer cannot process this high amount of data quickly, even when using efficient algorithms. A common approach to solving this problem is to use feature selection to eliminate the bulk of the data [2, 17]. For example, a researcher might select the 500 most important genes. However, by cutting out over 97% of available information, computations and analyses can become limited. Another possible approach is to use a dedicated computing cluster, but these may be difficult to set up, maintain, and schedule.

One barrier to scientists have to using all 20,000 genes is that they (just like most people) tend to develop habits. If a researcher cannot get a reasonable result in an adequately short time on a desktop, they will use a less

computationally-intensive approach as their method of habit. This tends to be a feature selection technique, as described above, to reduce data dimensionality and compute time. In essence, we would like to provide an approach that can study more data while still remaining convenient to the average researcher.

We want to enable exploratory, high risk research to do high dimensionality computations in a short time (1-3 minutes) on a desktop. NVIDIA CUDA provides this platform. The only hardware upgrade the typical desktop computer will need is a new graphics card (sometimes referred to as a Graphics Processing Unit, or GPU). This graphics card can come at a much more affordable cost and should avoid many of the negatives that typically come with traditional computing clusters.

Our approach to this project is to first code a serial implementation of a cell classification solution. Then, we will use CUDA to produce a parallel approach. Finally, we will compare the two methods and show that a CUDA algorithm is useful and practical for use in a research environment.

Our work began in Autumn 2014 as part of Dr. Fred Annexstein's Parallel Computing class. Professor Annexstein's lectures and assignments gave us the background in CUDA programming we needed for this project. We coded the serial parts of our project during this time, as well as the initial version of our parallel code. Samuel Schmidt continued the parallel work by himself through the spring semester and into the summer months. He presented this work as part of a poster session at the 2015 GLBIO conference at Purdue

University in West Lafayette, Indiana. After GLBIO, Samuel improved on the CUDA algorithm's efficiency and wrote this final thesis.

Chapter 3

Design

Our project consists of two separate code modules: a serial program, used for baseline control measurements, and a parallel CUDA program. We will use our parallel code to show that it is practical to use CUDA in a research setting. Both modules are written entirely in C++. This section elucidates the design and implementation details of the parallel code.

3.1 CUDA Background

CUDA is a parallel C/C++ (and Fortran, though we did not use this capability) computing platform and programming model invented by NVIDIA [5]. CUDA allows the user to send C/C++ code to an NVIDIA GPU. The best problems to solve with CUDA are those with a few simple tasks that need to be repeated many times. It's also best if the developer can express their

solutions in terms of simple array operations, as these are most naturally implemented with CUDA.

These operations are translated by the programmer into C/C++ functions called "kernels". These functions perform array operations in massive parallel. CUDA sends the proper data arrays and code to the GPU, which can run the code much faster than a normal CPU.

As shown below, the CUDA paradigm matches well with our Cell Classification problem because it involves large arrays of data and can be encapsulated into a few small blocks of code. These code blocks will become our CUDA kernels.

[3] lists some commonly used CUDA algorithms. These are typically array-based computations that can be written in a thread-safe manner. One of these algorithms,"reduce", is a key component of our CUDA code. In order to provide a little context into how CUDA works, reduce is described below.

3.1.1 The Reduce Algorithm

Reduce allows the programmer to apply a binary associative operator to an array of values. This definition has two parts: the operation must take two inputs (binary) and the order of the inputs must not matter (associativity). For example, the addition operator in basic algebra is binary and associative. It takes two inputs (the addends), and the order of the inputs do not matter (we will leave the proof of the associativity of addition up to the reader!).

Other examples of binary associative operators are multiplication, logical OR, and the finding of the maximum of two numbers.

Reduce applies whatever binary associative operator is specified to an array of values. Assume an input array of size n. Then reduce uses $n/2$ threads to carry out the reduce iteratively. In the first iteration, thread 0 applies the operator to the array positions 0 and $n/2 - 1 + 0$ and stores the result in position 0. Thread 1 applies the operator to positions 1 and $n/2 - 1 + 1$ and stores the result in position 1. That is, Thread x applies the operator to positions x and $n/2 - 1 + x$ and stores the result in position x. This method allows the reduce to be threadsafe. All of the operations in a single iteration can happy simultaneously.

The second iteration then proceeds with an array that is half the size of the original. Now the array length is n/2, and so the number of threads is also reduced by a factor of 2. This continues until there is only a single value in the array at position 0, which is the result.

3.2 Statistical Background

Cell classification is a multi-class problem. For our solution, we decided upon a Naive Bayes classifier with a Gaussian distribution. This classifier aligns well with our CUDA goals, as it requires simple calculations to be made many times. We can write just a few blocks of code and run them in parallel to finish the classification quickly. Additionally, Bayesian approaches have

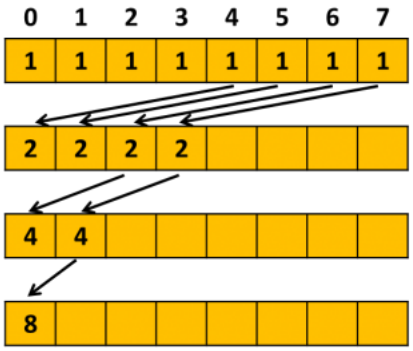

Figure 3.1: A visual representation of additive reduce on a small array.

been applied successfully to cell classification in the past [2, 17].

The Gaussian classifier has the equation

$$p(x = v|c) = \frac{1}{\sigma_c\sqrt{2\pi}} \exp(\frac{-1}{2}(\frac{v - \mu_c}{\sigma_c})^2)$$

where v is the current sample, c is the class, σ_c is the standard deviation, and μ_c is the mean. [11] describes it as a good general-purpose classifier.

Classification takes place in two asynchronous steps, training and testing. The training step is completed once, and then can be used for all future tests/classifications. The computational requirements of the two steps are considered separately below.

3.2.1 Training

The training step builds a mathematical model based on the NB classifier. Training can be accomplished in two steps: mean calculation and variance calculation. These two steps can be described in two separate functions, or CUDA kernels.

3.2.2 Testing

The testing step uses the mathematical model from the training step to select a class for each testing sample. To classify each sample, calculations are performed for each class-feature pair. These values are then reduced to a fuzzy set of C values, where C is the number of classes. In a fuzzy classifier, the result can be seen as existing in multiple classes at once, but in this problem one class probability is typically many orders of magnitude larger than the others, and so we place the sample into this class.

3.3 CUDA Design

Our project consist of four main CUDA kernels named after their tasks: mean calculation, variance calculation, classification, and reduction (for CUDA reduce). The mean and variance kernels are similar, and will be discussed together. The classification kernel uses the Gaussian distribution to provide statistical outputs. And, the reduce kernel uses the CUDA reduce algorithm to quickly process these outputs and simplify them to single values.

There is also some ancillary code. This will be described for completeness.

3.3.1 Mean and Variance Kernels

These kernels are simple functions that calculate the mean and variance for subsets of the data. They each take the entire input set as arguments. Their output is an array of length $N_f * N_c$ where N_f is the number of features in each sample and N_c is the number of classes. Each value is the mean or variance of a certain feature f across all samples that are defined as class c. The value for a specific f and c can be found at array position $f * N_c + c$.

3.3.2 Classification Kernel

This CUDA function uses the Gaussian distribution to calculate the likelihood that the input sample belongs to each class. To be more precise, this function outputs $N_f * N_c$ values, where the likelihood of the input sample belonging to class k is

$$\prod_{i=k*N_f}^{i*N_f+N_c} output[i]$$

3.3.3 Reduce Kernel

As discussed above, reduce applies a binary associative operator to an array of data. Our operator is multiplication. This kernel applies the above product calculation to the output from the classification kernel.

3.3.4 Utility Functions

Other CUDA code was written to supplement the main functions' work. For instance, there is a function that calculates the max of an array, which is used to decide which class a sample belongs to. There is also some initialization and comparison code for various purposes.

3.4 Input/Output

The input consists of training and testing data. Each input training sample must be accompanied by its class number.

Output consists of command line text that describes each classification.

The input and output are yet to be finalized. The end user will be able to configure the program to their needs.

3.5 Optimization

Various speedups can be achieved from standard CUDA reduce code. For those more familiar with CUDA, see slides 1-19 in [7] for information on the optimizations we achieved in our code.

Chapter 4

Results/Performance Analysis

The results in this section were procured using a typical RNA-seq dataset. Our dataset represents 479 samples and 39,328 features. There are about 40,000 features (rather than 20,000, the approximate number of human genes) because our dataset is actually the result of two measurements on the same samples.

The preliminary results Samuel presented at the GLBIO conference in May 2015 are included here for completeness and comparison. Various improvements and optimizations were made after this presentation.

4.1 Preliminary Results (May 2015)

At this point, the parallel code had been fully rewritten for correctness and organization. However, significant inefficiencies had yet to be discovered and

corrected. As seen in Figure 4.1, the parallel code was performing about twice as fast as the serial implementation.

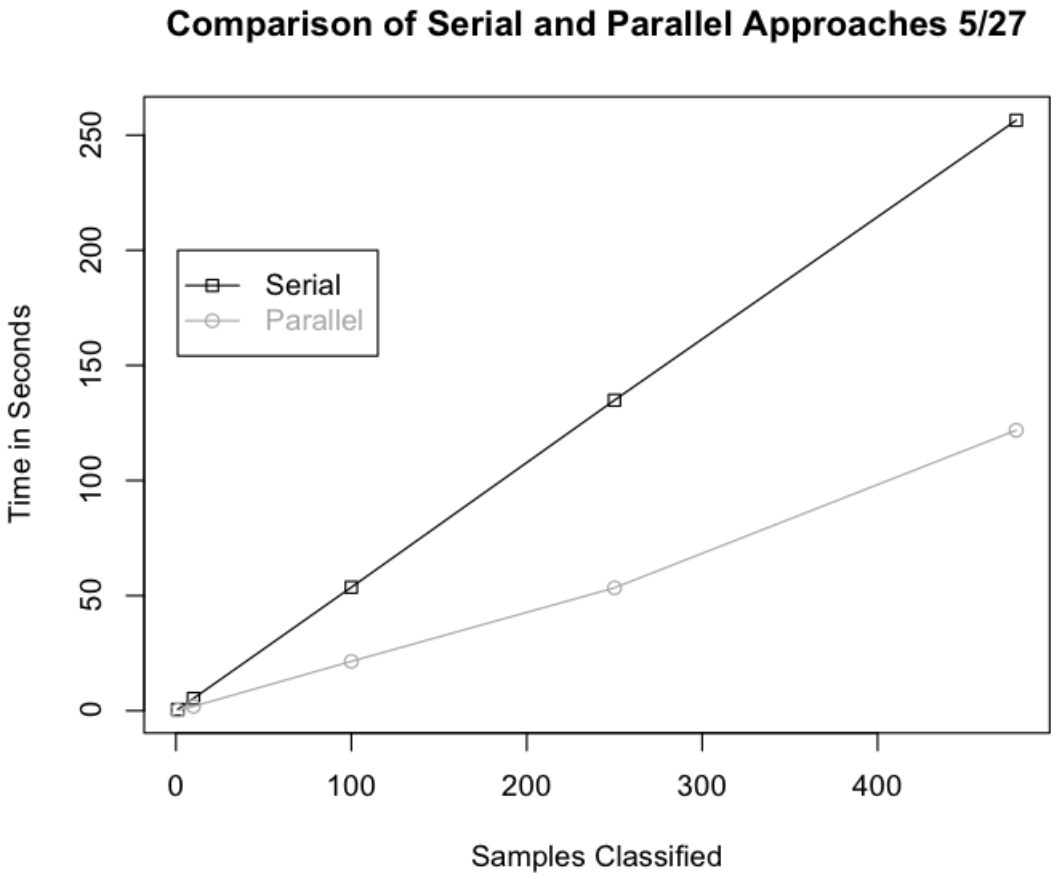

Figure 4.1: Comparison of Serial vs. Parallel code from May 2015

The parallel implementation ran in about half the time of our baseline serial code. Both algorithms have practically linear efficiency. As the number of samples is increased, the run time increases constantly. The full dataset

was classified in under 2 minutes. This was significantly faster than the lethargic serial algorithm, but still creates a relatively long wait time for the user. We were not happy with this result, as we feel it did not create enough incentive for a user to adopt a CUDA approach. The startup costs would likely be deterrent as the benefit was not nearly great enough.

4.2 Final Results (August 2015)

In the months after Samuel's GLBIO presentation, optimizations were made to the CUDA code. For an in-depth look at optimization techniques for CUDA kernels, see [7] The results from this optimizations are seen below in Figure 4.2.

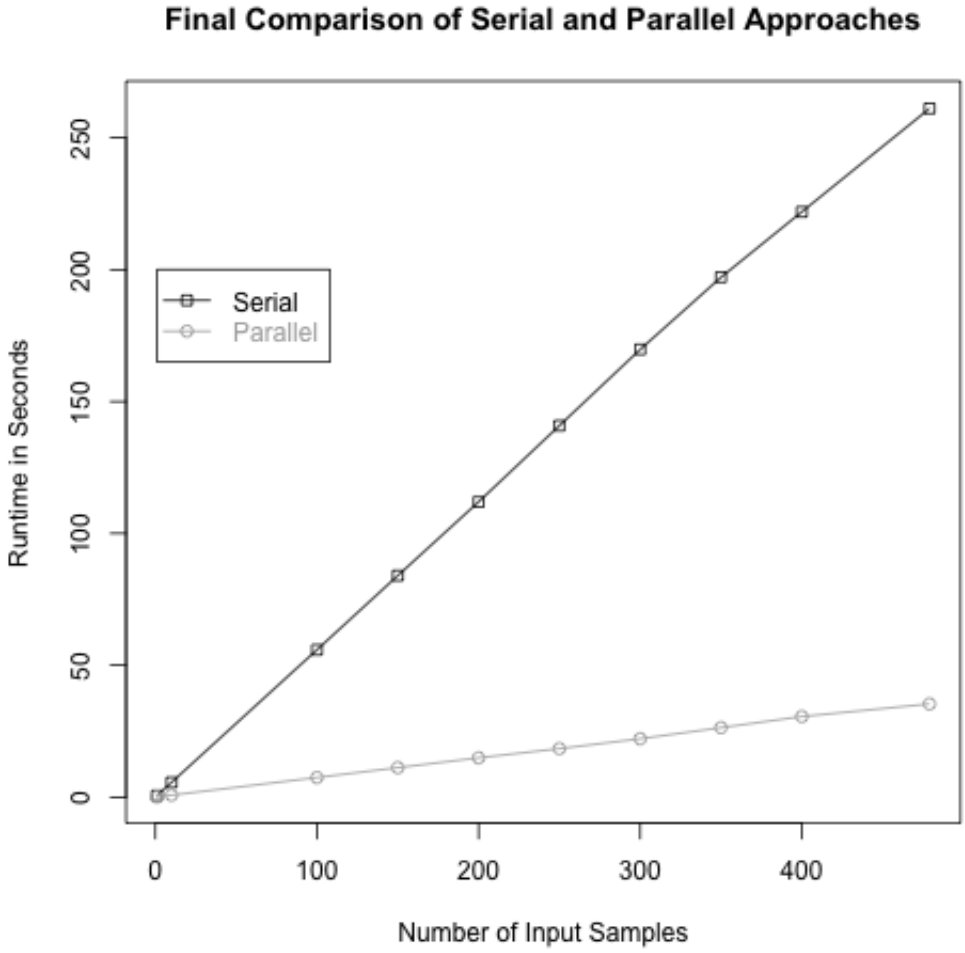

Figure 4.2: Final Comparison of Serial vs. Parallel code

Each data point in this graph represents the average of 10 runs of our serial or parallel code. Each algorithm displays a linear efficiency as the number of input samples is increased.

Using various speedup techniques, we were able to significantly improve

the efficiency of our parallel algorithm. The grey line now represents an approximate 7x speedup over the serial program. This shows that our work is useful and practical. On average, testing almost 500 samples against our trained data was completed in about 35 seconds. This is a huge speedup of 7.4x over the 261 seconds the serial approach took to complete.

Figure 4.3 shows the speedup efficiency achieved for each input size.

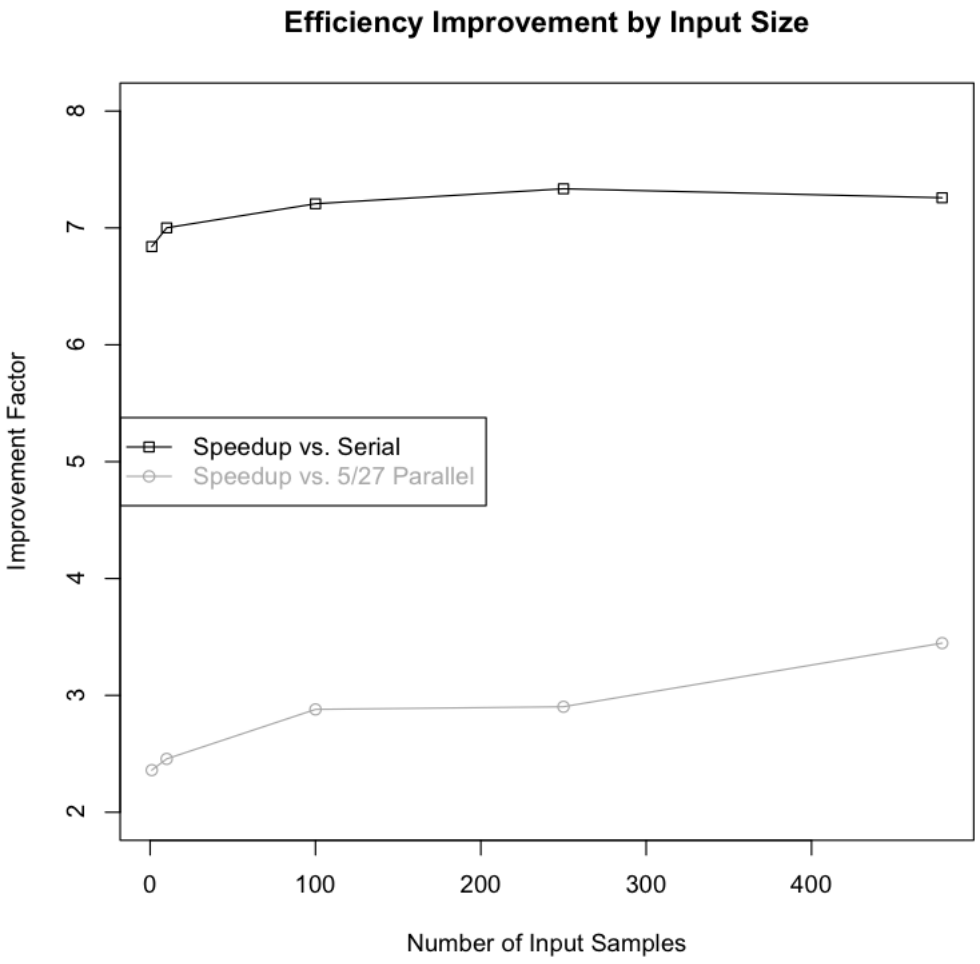

Figure 4.3: Speedup Achieved Versus Serial and Initial Parallel Code

The black line represents the speedup achieved versus the serial code. The line is nearly constant, showing our achievement of an approximate 7x speedup. The grey line shows the improvement from the preliminary results from May 2015. We improved our results by a factor of about 3 since then.

4.3 Classification Accuracy

The accuracy of the parallel algorithm was comparable to that of the serial approach. On some runs it was a few percentage points lower, but this can be attributed to rounding errors. Because CUDA is running in parallel, calculations may occur in a different order between separate runs of our algorithm. This means that roundings may cause slight aberrations in results.

Chapter 5

Challenges

We experienced a few different challenges throughout our work. These are listed here, along with a few issues that may arise in any continuing work

5.1 Development Environment

When we first started in Fall 2014, we used the development environments immediately available to us. Our serial work was done in Xcode, but our parallel coding was done in simple text editors on Linux. This provided a huge challenge, as CUDA doesn't provide much native debugging help. CUDA kernels often feel like mysterious "black boxes". Interpreting debug output is impossible, as threads are running concurrently, and may finish in any order. This prevented us from completing all that we wanted to accomplish for Dr. Annexstein's class.

Fortunately, we discovered a relatively simple solution in January 2015. Microsoft Visual Studio provides a CUDA add-on called NSight [6]. This extension provides a robust environment where which can help more precisely and efficiently develop CUDA kernels. Jeremy and Samuel configured this environment in January 2015, and Samuel used it for development throughout 2015.

5.2 Overfitting

Using all the available data for cell classification sounds like a great idea at first. It seems intuitive that removing a portion of the data will cause less accurate results and using all of the data will provide the best results possible. However, those using entire datasets will need to consider the possibility of overfitting. Some feature selection may be necessary to prune the data of biases. It is likely that a combined approach of feature selection and CUDA will produce even better results than either would on their own.

5.3 Opportunity Costs

Potential users will have to view our approach from their specific situation. In a high-end lab that already has a computing cluster, CUDA may relatively useless. However, CUDA could be perfect for a smaller research group with a restrictive budget. They would only need a single NVIDIA graphics card

to get started. We will explore CUDA's hardware costs in the next section.

Users should also consider the level of sophistication they will need. Right now, our approach only uses a single (relatively straightforward) classifier. Further work will need to be done in order to make our CUDA methods more nuanced and usable. If a researcher needs a more subtle or varied approach, they may need to do some work of their own, or consider other options. Our CUDA algorithm is very good at what it does, but its nascence is a weakness. Possibilities for future work are discussed below.

Chapter 6

CUDA as a Solution

6.1 Use Cases

Our intention is to help researchers that currently use 2 approaches to cell classification: feature selection and high cost computing. Many scientists will use low-percentage feature selection on their dataset in order to bring it to a manageable size. This may require pruning of 95% of the data in extreme cases. Of course, feature selection has some other helpful qualities; it can help prevent overfitting, for example. But, there is a good chance that this feature selection may be causing more harm than good. Our algorithm is "feature selection-agnostic". That is, a user may input their entire dataset (without fear of the program running endlessly) or they may perform some helpful, limited feature selection in order to gain the benefits of both approaches.

The second set of users we intend to help are those that are leery of high-

cost, high-performance computing. Configuring, maintaining, and scheduling a high-performance system (such as a cluster, etc.) can be expensive and time-consuming. We would like to help enable valuable and complex research in even small labs that cannot afford these systems. In addition, we may able to assist labs with these systems to reduce their load by moving some work onto desktop machines with CUDA.

6.2 CUDA GPUs

For a complete list of CUDA-compatible GPUs, see [4]. NVIDIA has a wide variety of solutions, but we will focus on those that we think might be used for our application. Of course, a potential user may want to do further research in order to decide what's best for them.

6.2.1 GeForce GPUs

The GPU used in our project is the GeForce GTX 770 [9]. This card can be bought for around $250 [12]. Similar cards (also in the GTX 700 series) can be had for even less. Some of these cost about $100. Anyone can achieve performance increases like we have seen with just a few hundred dollars and a straightforward installation.

Some of the latest and more powerful GeForce GPUs cost a bit more. Those in the GTX 900 and GTX TITAN lines can be a bit pricier. However, some of these exist in the same range as the 700 series.

6.2.2 Tesla GPUs

These cards are higher-end versions that can run in a server environment [13]. They are not our main focus, but are worth mentioning. For about $3000, a researcher can buy a Tesla K20 or K40 card [14, 15]. The newer K80 card will cost about $5000 [16].

Chapter 7

Future Work

This section outlines idea for work that may be done in the future. Readers that may be interested in CUDA and cell classification should read this section and contact the authors with any questions or thoughts.

7.1 Other Classifiers

The most obvious approach may be to add a few new CUDA kernels to run a different classifier on the data. Our framework should allow any future programmers simple access to their data, as we have done the basic I/O and data manipulation steps. The trickiest part of this is to find a classifier that fits well with the CUDA programming paradigm. The classifier should facilitate work with arrays. It should also require simple operations to be performed many times. Algorithms such as SVMs, Hidden Markov Models,

and other machine learning techniques may be great places to start.

7.2 Further Optimizations

We have done a few different levels of optimizations that have improved our algorithm's speed greatly. In spite of this, there may be further speedups possible. This will be more specialized work, and may be impractical. However, we have greatly improved the project's efficiency, so attempting to eke out a few more percentage points of speed is likely not the best place to start.

Along the same lines, some rounding errors seemed to cause slight problems in classification accuracies between runs. Further work may need to be done in this area to fine-tune the way CUDA handles different inputs and outputs in order to remove these problems.

Chapter 8

Conclusion

We were able to produce a CUDA-parallel version of a simple classifier that can be used in Cell Classification. Researchers doing this type of analysis should be able to increase their speed and throughput without sacrificing the quality of their work. We are confident that any researcher that uses our tool will adopt it as habit.

Say where the code is and how to download a runnable version. (Make a runnable version)

Our code is hosted here 093ur09j if you would like to fork it and/or make modifications. It is open source under the MIT license.

If you have any questions, comments, or suggestions, please contact Samuel Schmidt at CSTSchmidt@gmail.com.

Bibliography

[1] B. Alberts, *et al.*, *Molecular Biology of the Cell*, 4th Edition, 2014.

[2] H. Chen, "Evaluating statistical learning methods for cell type classification and feature selection using RNA-seq data", in *UT KBRIN Bioinformatics Summit*, Cadiz, KY., 2014

[3] (2008). *NVIDIA CUDA SDK - Data-Parallel Algorithms* [Online]. Available: `http://www.nvidia.com/content/cudazone/cuda_sdk/Data-Parallel_Algorithms.html`

[4] (2015). *CUDA GPUs* [Online]. Available: https://developer.nvidia.com/cuda-gpus

[5] *NVIDIA CUDA* [Online]. Available: `http://www.nvidia.com/object/cuda_home_new.html`

[6] (2015). *NVIDIA Nsight Visual Studio Edition* [Online]. Available: `https://developer.nvidia.com/nvidia-nsight-visual-studio-edition`

[7] M. Harris. *Optimizing Parallel Reduction in CUDA* [Online]. Available: http://www.cuvilib.com/Reduction.pdf

[8] I. Ezkurdia, *et al.* "Multiple evidence strands suggest that there may be as few as 19 000 human protein-coding genes", *Human Molecular Genetics*, vol. 23, no. 22, pp. 5866-5878, Jun. 2014.

[9] (2015). *GeForce GTX 770M Specifications* [Online]. Available: http://www.geforce.com/hardware/notebook-gpus/geforce-gtx-770m/specifications

[10] D. A. Jaitin, *et al.*, "Massively Parallel Single-Cell RNA-Seq for Marker-Free Decomposition of Tissues into Cell Types", *Science*, vol. 343, no. 6172, pp. 776-779, Feb. 2014.

[11] (2015). *Naive Bayes Classifier* [Online]. Available: http://www.statsoft.com/textbook/naive-bayes-classifier

[12] (2015). *newegg* [Online]. Available: http://www.newegg.com

[13] (2015). *TESLA GPU Accelerators for workstations* [Online]. Available: http://www.nvidia.com/object/tesla-workstations.html

[14] (2015). *NVIDIA TESLA K20 (900-22081-2220-000) GK110 5GB 320-bit GDDR5 PCI Express 2.0 x16 3.52 Tflops Workstation Video Card - OEM* [Online]. Available: http://www.newegg.com/Product/Product.aspx?Item=N82E16814132008

[15] (2015). *NVIDIA Tesla K40 Graphic Card - 1 GPUs - 745 MHz Core - 12 GB GDDR5 SDRAM 900-22081-2250-000* [Online]. Available: http://www.amazon.com/NVIDIA-Tesla-Graphic-Card-900-22081-2250-000/dp/B00KDRRTB8

[16] J. K. Author. (2015). *NVIDIA Tesla K80* [Online]. Available: http://www.sharbor.com/nvidia-tesla-k80.html?gclid=COHQ5e2ipscCFQiNaQodJiUG-Q

[17] K.Y. Yeeung, *et al.* "Bayesian model averaging: development of an improved multi-class, gene selection and classification tool for microarray data", *Bioinformatics*, vol. 21, no. 10, pp. 2394-2402, Feb. 2005.