SEARCHING POSITIVE SELECTION USING GPU AND BENCHMARKING GPU BASED SHORT READ ALIGNERS

Enrollment Number: 121503

Name of Student: Siddharth Singh Tomar

Name of Supervisor: Dr. Tiratha Raj Singh

*Submitted in partial fulfillment of the Degree of
Bachelor of Technology*

DEPARTMENT OF BIOTECHNOLOGY AND BIOINFORMATICS
JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY,
WAKNAGHAT

Contents

Certificate iii

Acknowledgements iv

Summary v

List of Figures vii

List of Tables viii

List of Abbreviations ix

1 Introduction 1
- 1.1 Introduction . . . 1
  - 1.1.1 Problem Statement . . . 2
  - 1.1.2 Objectives . . . 2
  - 1.1.3 Existing Tools . . . 2
- 1.2 Development Strategy . . . 3
  - 1.2.1 Phase I . . . 3
  - 1.2.2 Phase II . . . 3
  - 1.2.3 Phase III . . . 3
- 1.3 Intro to CUDA/OpenCL . . . 4
  - 1.3.1 CUDA . . . 4
  - 1.3.2 OpenCL . . . 5

2 Technical Specifications 8
- 2.1 Overview . . . 8
  - 2.1.1 Host and Kernels . . . 8
- 2.2 FastCodeML to GCodeML . . . 9
  - 2.2.1 Libraries . . . 10
- 2.3 Option 2 . . . 10
- 2.4 Option 3 . . . 11

3 Analysis 17
- 3.1 Drawbacks in FastCodeML . . . 17
  - 3.1.1 OpenMPI . . . 17
  - 3.1.2 Coding . . . 17
- 3.2 CUDA and OpenCL . . . 18
  - 3.2.1 CUDA framework and restrictions . . . 18

3.2.2 Hardware restrictions . . . 18
3.2.3 Caching and Optimization of probabilities . . . 18
3.2.4 OpenCL . . . 19
3.3 OpenACC . . . 19
3.4 Optimal Algorithm . . . 19

4 Benchmarking-Short Read Aligners **20**
4.1 Introduction . . . 20
4.2 Method . . . 21
4.2.1 Read Generation . . . 22
4.2.2 Benchmark . . . 22

5 Benchmarking-Results **27**
5.1 Numbers . . . 27
5.1.1 Speed . . . 27
5.1.2 Performance . . . 30
5.1.3 Discussion . . . 30

6 Conclusion **32**
6.0.1 Conclusion: I . . . 32
6.0.2 Conclusion: II . . . 32
6.0.3 Limitations of this study . . . 33
6.0.4 Future possibilities . . . 33

Bibliography **34**

Certificate

This is to certify that the work titled **"Searching Positive Selection using GPU and Benchmarking GPU based Short Read Aligners"** submitted by Siddharth Singh Tomar in partial fulfillment for the award of degree of Bachelor of Technology in Bioinformatics of Jaypee University of Information Technology, Waknaghat has been carried out under my supervision. This work has not been submitted partially or wholly to any other University or Institute for the award of this or any other degree or diploma.

Signature of Supervisor: ________________________

Name of Supervisor: Dr. Tiratha Raj Singh

Designation of Supervisor: Assistant Professor

Date: ______________

Acknowledgements

First and foremost, I would like express my deepest gratitude to my supervisor, Dr. Tiratha Raj Singh, who guided me and helped me in every possible way during the course of this thesis. His continued support and guidance enabled me to pursue study in a relatively new field.

I express my sincere thanks to Dr. Rajinder S. Chauhan (Head: Dept. of Biotechnology and Bioinformatics) This study would not have been possible without his support and mandate.

My sincere thanks also goes to Dr. Ivo Kwee, for offering me the summer internship opportunity in his group at IOR and letting me explore new prospects in computational biology.

I would also like to thank Nvidia Corporation for their hardware grant consisting of a Tesla K40 for this study.

Last but not the least, I would like to thank my family: my parents and to my brother for supporting me and my decisions in life.

Signature of Student: ________________________

Name of Student: Siddharth Singh Tomar

Date: ______________

Summary

This study and thesis is divided into two parts. The first part of this thesis pertains to development of an algorithm for searching positive selection(specifically by implementing Branch Site Model) using GPU and ascertain the feasibility of such implementation. This include changes in the underlying algorithm of preexisting tools to accommodate GPU and HPC acceleration paradigms. Along with implementation, this study also identified potential drawbacks of such implementation, and alternative strategies for possible program.

The second part of this thesis deals with performance profiling of short read aligners (using Nvidia CUDA framework) for testing scalability in GPGPU environment. We studied the impact of GPU based aligners on NGS analysis pipeline and included a comparison with CPU based counterparts. The main aim of this study was to identify the possible gains by using GPU in NGS analysis within a similar price bracket and to study the implementation of such aligners in GPU. The performance was measured by running alignments on simulated Illumina reads on human genome.

__________	__________
Signature of Student	Signature of Supervisor
Siddharth Singh Tomar	Dr. Tiratha Raj Singh
Date: ______________	Date: ______________

List of Figures

1.1 Phases 7

2.1 Overview 13
2.2 Caching 14
2.3 Block management 15
2.4 Caching 16

4.1 Basic IO diagram between CPU and GPU. Notice the lack of direct interface between system memory and GPU. 24
4.2 Approach followed by GPU based aligners 25
4.3 A simplified representation of short read aligners 26

List of Tables

5.1 Numerical performance across parameters 28
5.2 Average time in seconds 30

List of Abbreviations

GPU	**G**raphic **P**rocessing **U**nit
GPP	**G**eneral **P**urpose **P**rocessor
CPU	**C**entral **P**rocessing **U**nit
CPV	**C**onditional **P**robability **V**ector
API	**A**pplication **P**rogram **I**nterface
ML	**M**aximum **L**ikelihood
MCMC	**M**arkov **C**hain **M**onte **C**arlo

1 Introduction

This thesis sis divided into two major parts. Chapter 1, 2, 3 are related to a development perspective, whereas chapter 4, 5 are related to evaluation of existing programs. Chapter 6 serves as conclusion of both parts.

1.1 Introduction

Detecting positive Darwinian selection at the DNA sequence level has been a subject of considerable interest. However, positive selection is difficult to detect because it often operates episodically on a few amino acid sites, and the signal may be masked by negative selection. The detection of positive selection is widely used to study gene and genome evolution, but its application remains limited by the high computational cost of existing implementations. The non-synonymous (amino acid altering) to synonymous (silent) substitution rate ratio (d_N/d_S) provides a measure of natural selection at the protein level, with $\omega = 1, \omega < 1, \omega > 1$ indicating neutral evolution, purifying selection, and positive selection, respectively. In biology, phylogenetic trees are used to represent the evolutionary relationships among groups of organisms or among a family of related nucleic acid or protein sequences based upon their similarities and differences. They are helpful in inferring the history of organism lineages as they evolve over time. Since phylogenetic relationships among species also can help determine which ones might have similar functions, it is widely used in medical research for predicting potential hazards from species with rapidly changing pathogens, for example to identify lethal variants of HIV and SARS viruses. The operation of phylogenetic analysis aims to investigate the evolution and relationships among species. It is widely used in the fields of system biology and comparative genomics.

Phylogenetic analysis aims to infer the evolutionary order of a set of species or genes. It estimates the relationship by comparing homologous sites between species on different branches so that evolutionary scores can be assigned to given phylogenetic trees. Since biological sequences normally have some dissimilarities, sequences under investigation need to be multiply-aligned so that homologous sites can be discovered to form columns in the alignment. These alignments are then used to construct phylogenetic trees. A phylogenetic tree is a diagram which depicts the relationships of species in a tree format. Phylogenetic trees can be either rooted or unrooted. Phylogenetic trees are normally branching diagrams, where, leaves in trees represent species, and species are joint together to compose internal nodes. Internal nodes in trees represent the inferred most recent common ancestors of their descendants. The Maximum Likelihood (ML) algorithm can be used to infer a phylogenetic tree or evolutionary tree, as it searches for the tree with the highest probability or the maximum likelihood from an existing tree. For this, pairs of biological sequences need to be multiply-aligned in advance. There are many tools for designed for performing multiple sequence alignment like PAUP, ClustalW, etc. This multiple sequence alignment file is to be used as input for our GPU based implementation of positive selection with codon models.

1.1.1 Problem Statement

The primary task for the project is to improve the computation of positive selection using codon models and especially the Branch- Site model. While this model has advantages of biological realism and of robustness, the main implementation is computationally intensive . The scanning for positive selection in gene evolution is CPU limited. This is because CPUs do not optimize for throughput, rather they optimize for latency thus restricting the CPUs from handling computationally intensive data. We will be modifying the same algorithm for GPUs and will try to analyze the performance gain which might be possible from a GPU based implementation, and any shortcomings for the given approach.

1.1.2 Objectives

These are the primary objectives:

1. To devise an algorithm for computing positive selection with codon models on GPUs.
2. Improving computational performance for positive selection search.
3. To create a framework for a tool like PAML [1] by using OpenCL/CUDA for parallel programming.
4. To analyze shortcomings of FastCodeML.

1.1.3 Existing Tools

These are the tools which we will use as base for constructing our algorithm:

1. PAML (Phylogenetic Analysis By Maximum Likelihood) [1] : PAML is a package of programs for phylogenetic analysis of DNA or protein sequences using maximum likelihood. It is maintained and distributed for academic use free of charge by Ziheng Yang. ANSI C source codes are distributed for UNIX/Linux/Mac OSX, and executables are provided for MS Windows. PAML is not good for tree making. It may be used to estimate parameters and test hypotheses to study the evolutionary process.
2. BEAGLE API (Broad-Platform Evolutionary Analysis General Likelihood Evaluator) [2] : An application programming interface (API) and high-performance computing library for statistical phylogenetics. Emphasises on evaluating phylogenetic likelihoods of biomolecular sequence evolution and aims to provide high performance evaluation 'services' to a wide range of phylogenetic software, both

Bayesian samplers and maximum likelihood optimizers allowing phylogenetic software using the library to make use of optimized hardware such as GPUs.

3. PhyML[3] : Is a tool for fast and accurate estimation of phylogenies by maximum likelihood.

1.2 Development Strategy

We planned the execution of project in three primary phases (Figure 1.1 on page 7):

1. Phase I: Preparation
2. Phase II: Development
3. Phase III: Extension

1.2.1 Phase I

Completed During this phase, we collected all the relevant literature, linked them to our problem and determined best course of action. Initial review clearly illustrated the magnitude of problem, while the problem itself is solvable, it will take considerable efforts. We decided to create a WBS (Work Breakdown Structure) to divide different parts of problem to different members of group. Also, we extracted the relevant information from literature and condensed them to form a overview to create a structure which we can implement.

1.2.2 Phase II

Completed During this phase, we started primary development and analyzed most of the code for a proof of concept algorithm. During this phase, we also plan to modify the algorithm to suit our requirements, introducing features as required. One of the observation is that most of the algorithm inherently is recursive in nature, which is unsuitable for GPUs, therefore we plan an iterative and flat conversion of such routines. Other complexities are discussed in technical section. As of now, we have successfully completed implementation of optimized tree traversal on GPU, which we will use for creating multiple MC-MC simulations. We are currently working on understanding code of FastCodeML, and devising ways to replace function calls to BLAS with MAGMA functions. We have already identified R subroutines which we can use in conjunction in MAGMA for matrix manipulation in GPU.

1.2.3 Phase III

Completed We tried to analyze the best possible algorithm and implementation for the problem. The results and technical peculiarities are discussed in later chapters.

1.3 Intro to CUDA/OpenCL

In this section, we try to explain the basic working behind CUDA and OpenCL based programs and the difference between excution on GPP and GPU. GPU programming is inherintly different from general programming used for x86/POWER based processors, and ths requires some explaination before we move to techincal section of this project.

1.3.1 CUDA

The CUDA programming model is a heterogeneous(i.e. using both GPP and GPU) programing paradigm. Here host is GPP and the associated main system memory(i.e. RAM/cache) and device is the GPU and the video buffer/memory it contains. GPP manages the device and its memory, simultaneously also launching kernels which are functions executed on the device. The kernels are actual programming "objects" which are used parallelized via threading on GPU.

A typical sequence of operations for a CUDA C program is:

1. Declare and allocate host and device memory.
2. Initialize host data.
3. Transfer data from the host to the device.
4. Execute one or more kernels.
5. Transfer results from the device to the host.

A sample CUDA code representing Single Precision X*Y+A program:

```
#include <stdio.h>

__global__
void saxpy(int n, float a, float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

int main(void)
{
  int N = 1<<20;
  float *x, *y, *d_x, *d_y;
  x = (float*)malloc(N*sizeof(float));
  y = (float*)malloc(N*sizeof(float));

  cudaMalloc(&d_x, N*sizeof(float));
  cudaMalloc(&d_y, N*sizeof(float));
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);

  // Perform SAXPY on 1M elements
  saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);

  cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);

  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = max(maxError, abs(y[i]-4.0f));
  printf("Max error: %fn", maxError);
}
```

1.3.2 OpenCL

Similarly, OpenCL also follows a similar paradigm for program execution, but with a main difference that it is a cross-platform C99 implementation specifically made for parallel computing. It can practically work with GPUs and CPUs from all major vendors and is more or less a subset of C language, unlike CUDA which extensively modifies C language for its own execution. The same Single Precision X*Y+A program in OpenCL looks like:

```
#include <iostream>
#include <algorithm>

#include "include/CLUtility.h"
#include "include/CLEvent.h"
#include "include/CLContext.h"
#include "include/CLBuffer.h"
#include "include/CLProgram.h"

int main()
{
   // create a context for the second GPU
   clp::Context context(CL_DEVICE_TYPE_GPU, 1, 1);

   // create and build a program
   clp::Program program(context);
   program.setSource(
   "kernel void saxpy(\n"
   "   global float *x, global float *y, float a\n"
   ")\n"
   "{\n"
   "   const uint index = get_global_id(0);\n"
   "   x[index] += a*y[index];\n"
   "}\n"
   );
   program.build();

   // obtain a kernel object
   clp::Kernel<void(float*,float*,float)> saxpy =
      program.getKernel<void(float*,float*,float)>("saxpy");

   // create device buffers
   clp::Buffer<float> x(context, 1024);
   clp::Buffer<float> y(context, 1024);

   // map the buffers
   clp::Event xevent = x.map();
   clp::Event yevent = y.map();

   // fill them with data and unmap
   xevent.wait();
   std::fill(x.begin(), x.end(), 45);
   x.unmap();

   yevent.wait();
   std::fill(y.begin(), y.end(), 3);
   y.unmap();

   // execute kernel
   saxpy(clp::Worksize(1024,256), x, y, 13);

   return 0;
}
```

It should be noted that this OpenCL code is specifically made for GPU only. An optimal OpenCL implementation will be too large for even a simple problem, mainly because of optimizations and accessory code required for each architecture. There is a third option too, which will be discussed in next chapter.

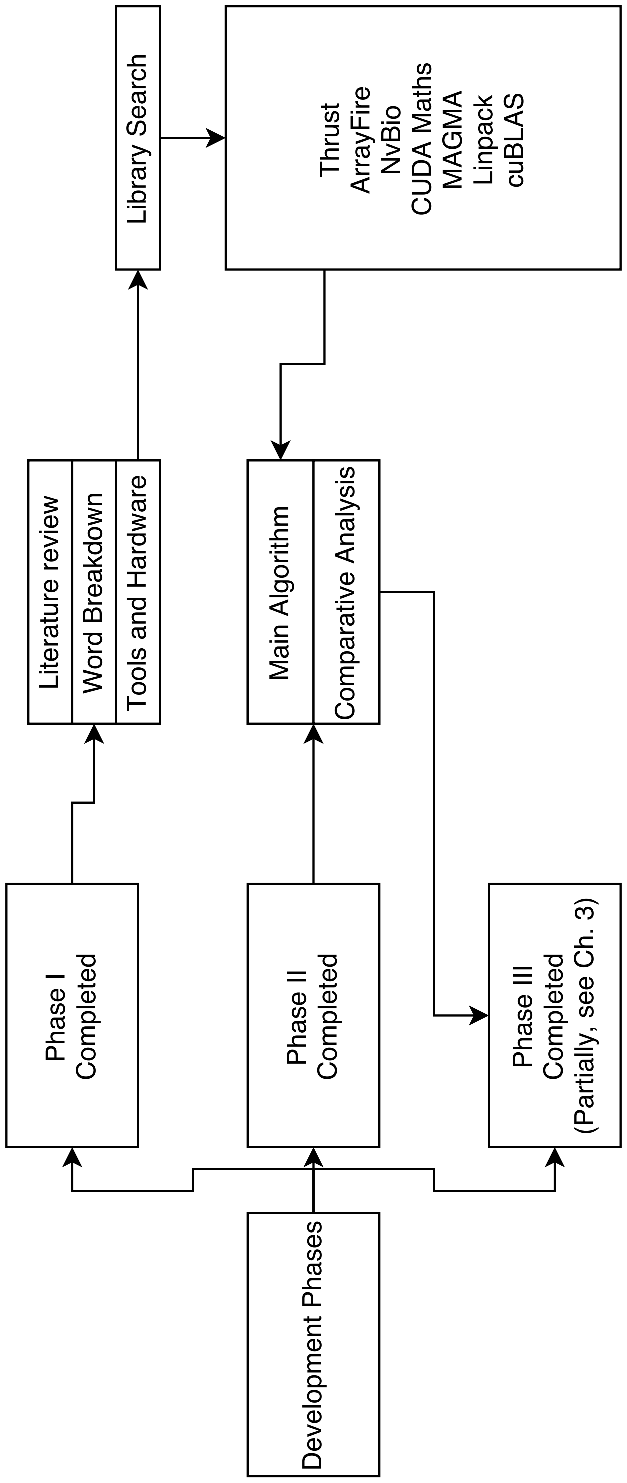

FIGURE 1.1: Development phases

2 Technical Specifications

2.1 Overview

Driver-Kernel relationship Initially, we planned to use both, GPP(General Purpose Processor, CPU) and GPU for calculation, whereby CPU will calculate the transition probabilities locally, and transfer the result to GPU global memory, but ultimately decided to use GPU only for all calculations, including input preprocessing [if required, although nvbio library supports only pairwise alignment processing and thus is not suitable for such tasks. Multiple Sequence Alignment is possible on GPU, but the tools [4], [5] are not as accurate as latest version of ClustalW (Clustal Omega)]. Main issue with this approach was added latency cost and further complications in cache management. We are designing algorithm for a single node-single card setup, thus there is no scope of incorporating techniques like multi-GPU and cluster based work sharing. One interesting prospect in future can be to use IBM Power8 based servers and GPU based cluster to bifurcate single dataset and compute the parts separately(for example, reducing state space on GPU based cluster and performing final reduction on Power8), merging the results in the end.

2.1.1 Host and Kernels

The host program, which will primarily use CPU will act as a simple driver, which will call the kernels and manage the threads and block. It will not contribute in actual calculations, and will be completely separate from problem. The kernels are divided into three major categories, likelihood processors, transition probability/ state space processor and matrix processor. Likelihood processors will work in a fashion similar to previous implementations[6], [7]. We prefer the given implementation[6] because of the way threads and blocks are managed. It should be noted that the GPU we are using allow a higher number of concurrent threads and better management of resources at hardware level, thus the algorithm requires extreme modification. Another implementation which piqued our interest [7] uses a slightly different approach, which although is efficient but needs extensive modification to support expanded feature set of Maxwell generation (Figure 2.1 on page 13 explains basic overview of program).

Transition ratios(Transversion, Transition, Synonymous/Non Synonymous ratios etc) will be managed independently, but concurrently. This is another area of discussion between group, they can be concurrently processed and stored(using x% of total processing blocks), and cached for likelihood calculations in real time. Same kernel will also manage state space, to reduce the number of calls between two operations.

2.2 FastCodeML to GCodeML

Then comes the actual porting of FastCodeML on GPU, for which we studied the primary implementation elucidated in the publication[8]. Most of the implementation makes no sense on GPU, since there is substantial amount of modifications required. Load balancing paradigm needs to be rewritten from scratch. In actual sense, load balancing on GPU works a little differently and in case of FastCodeML, it is just efficient caching of previous results. The problem in GPU is that while threads in same block can utilize the common values, threads between different blocks would need to access L2 cache or Global memory, which would incur significant cost in terms of data transfer and associated latency. CPVs (Conditional Probability Vectors) are too large to be stored on L1 cache or 96KB shared memory per block. Ranking and grouping, although is possible on GPU, is not a logical solution since it would too require constant exchange of information between global and local memory(each transition cost 2 to 3 milliseconds, a considerable amount if summed up for 3000+ cores and many more threads).

One extreme solution would be to use minimum possible cores to calculate CPVs for a subset, store them according to subtree topology and for next iteration, try a local search using stored values if subtree topology matches with node values. Simultaneously, the unused topologies(and associated node values) will be replaced by most frequent ones. This method is completely naive and random, but is easily implementable on GPU, allowing us to reduce to access time in average case (rather optimistic scenario, Figure 2.2 on page 14 and Figure 2.3 on page 15).

There is another area where GPU based implementation becomes complex, we are constrained by the amount of global memory(12GB at max, out of which roughly 10GB is usable). We are currently thinking about ways to manage the global memory without any compression or encoding involved(many short read aligners[9], [10] use encoding schemes for efficient search and storage on GPU). If the need arises to use an encoding scheme for data management, we would then also group subtrees according to similarity. This runs contrary to our initial proposal to use a naive method; since we are already programming a CPU dependent encoding scheme, it wouldn't create much problem if we do the sorting on GPP itself, violating the GPU only nature of program.

2.2.1 Libraries

As for the linear algebra and associated libraries for matrix operations, the solution is rather simple. Both BLAS and LAPACK can be replaced by MAGMA[11] and cuBLAS (MAGMA can substitute cuBLAS too, unless we decide to reuse the code from Fast-CodeML). If instructed and required, cuBLAS can be replaced by cuBLAS XT, allowing us to use multiple GPUs for matrix operations. Initially we also considered CULA-Tools, but the licensing was restrictive in nature and was available with reduced set of operations, we decided to use MAGMA. ArrayFire is another alternative which we are considering for future version. CUDA Maths will replace the general mathematics library for simple functions, and Thrust is already implemented on GPU.

2.3 Option 2

We have also created another possible option, by using the OpenCL programming framework and libraries. The resultant algorithm and program thus will be architecture independent, and thus will be able to run on both, GPP and GPU. This strategy specifically requires high amount of optimization, which we were not able to archive in given time frame. Provided that an OpenCL based implementation is able to run on products supplied by AMD, this is an interesting prospect given that the required level of optimization is reached. AMD has high memory GPUs in market which have an advantage over Nvidia based on the criteria of compute power. OpenCL implementation also enables a seamless hybrid computation strategy, in which the resulting program can use both GPU and GPP for calculating parameters better suited for each respective type of processors. Like the CUDA based implementation, even this implementation has its own drawbacks, many of which are shared between two.

In OpenCL based implementation, we decided to create a substructure which uses the libraries which are compatible with both, GPP and GPU. Specifically LAPACK is available as an library for OpenCL based implementations also, although it will help only with matrix operation mostly. The problem of efficient caching still remains. Considering that we still have to manage CPV values, we are considering a way to store them using a tree like structure in GPU main memory, which is possible since the GPP can be used fairly easily with OpenCL based implementation to manage the trees and transfer them to GPU after a specific number of calculations have been reached, unlike CUDA which requires a complete memory refresh to work with arrays.

2.4 Option 3

The last and most radical option is to use a new compiler based acceleration framework known as OpenACC. OpenACC is meant to augment OpenMP and MPI based codes by providing line to line replacement for OpenMP based complier directives meant for parallel acceleration on CPU. Ideally, OpenACC based code can run on both GPU and CPU without any modification, although it can not utilize both optimally for a same subroutine. OpenACC is based on compiler intelligence to parallelize code, which means that the programming effort require to solve a program using parallelization is minimum. This also means that the solution given by OpenACC compiler might not be optimal, or even worse than a serial implementation. This option enables us to implement FastCodeML without any substantial modification to measure the performance on GPU using their existing algorithm, although given that the algorithm itself is largely unsuitable for such architecture, it might be possible to use OpenACC to create a proof of concept for testing the caching strategy. The same Single Precision X*Y+A program in OpenACC looks like:

```
#include <stdio.h>
#include <openacc.h>
#include <stdlib.h>

void saxpy_openacc(float *restrict vecY,
float *vecX, float alpha, int n)
{
     int i;
#pragma acc kernels
     for (i = 0; i < n; i++)
           vecY[i] = alpha * vecX[i] + vecY[i];
}

void saxpy_cpu(float *vecY,
float*vecX, float alpha, int n)
{
     int i;

     for (i = 0; i < n; i++)
           vecY[i] = alpha * vecX[i] + vecY[i];
}

int main(int argc, char *argv[])
{
   float *x_host, *y_host;
   float *x_dev, *y_dev;
   float *y_shadow;

   int n = 32*1024;
   float alpha = 0.5f;
   int nerror;

   size_t memsize;
   int i, blockSize, nBlocks;

   memsize = n * sizeof(float);

   x_host = (float *) malloc(memsize);
   y_host = (float *)malloc(memsize);
   y_shadow = (float *) malloc(memsize);

    for ( i = 0; i < n; i++)
   {
      x_host[i] = rand() / (float)RAND_MAX;
      y_host[i] = rand() / (float)RAND_MAX;
      y_shadow[i]=y_host[i];
   }


   saxpy_openacc(y_shadow, x_host, alpha, n);

   free(x_host);
   free(y_host);
   free(y_shadow);

   return 0;
}
```

A flowchart illustrating OpenACC based approach for computing positive selection is mentioned in Figure 2.4 on page 16. Next chapter will illustrate various drawbacks of each strategy along with possible performance implications.

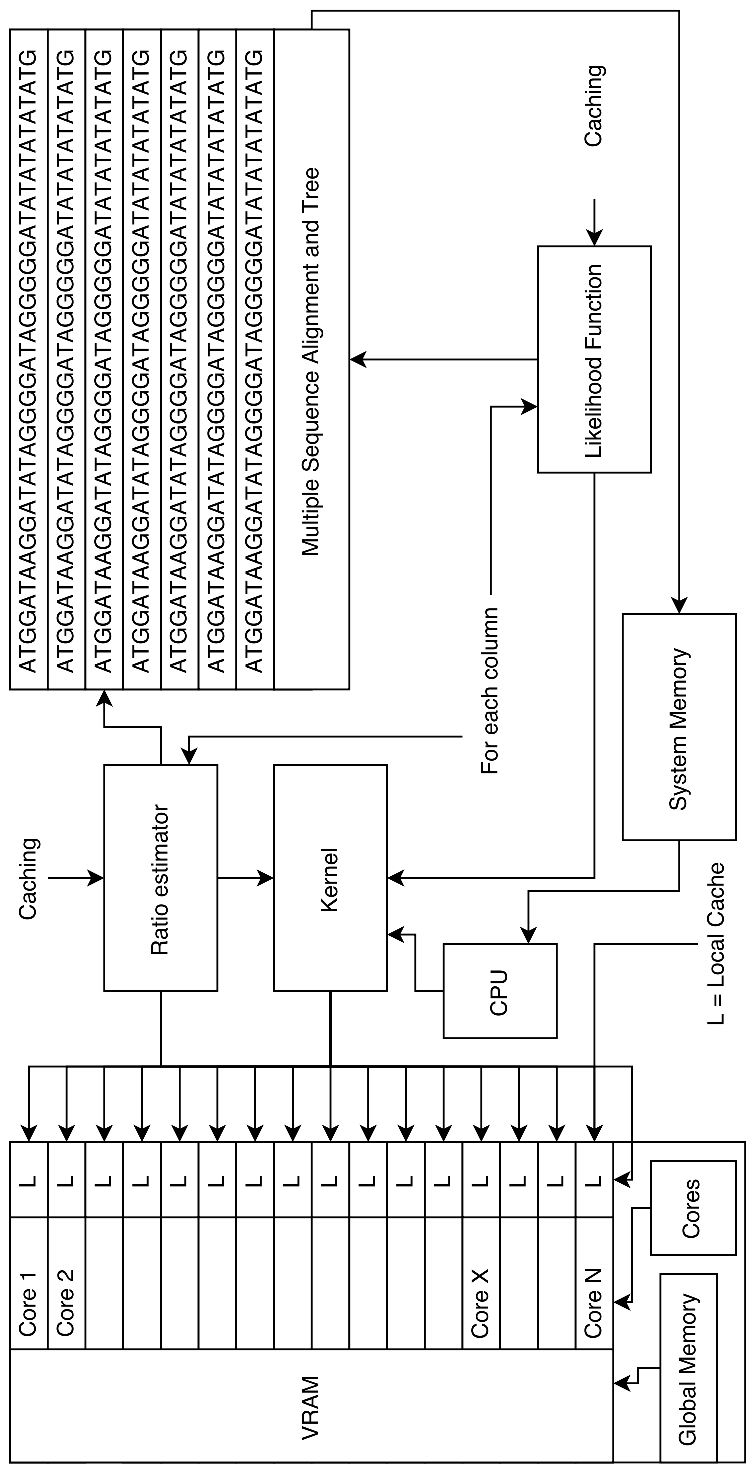
ATGGATAAGGATATAGGGGATAGGGGGATATATATATATG
ATGGATAAGGATATAGGGGATAGGGGGATATATATATATG
ATGGATAAGGATATAGGGGATAGGGGGATATATATATATG
ATGGATAAGGATATAGGGGATAGGGGGATATATATATATG
ATGGATAAGGATATAGGGGATAGGGGGATATATATATATG
ATGGATAAGGATATAGGGGATAGGGGGATATATATATATG
ATGGATAAGGATATAGGGGATAGGGGGATATATATATATG
Multiple Sequence Alignment and Tree
Caching
Likelihood Function
For each column
System Memory
Caching
Ratio estimator
Kernel
CPU
L = Local Cache
L
Core 1
Core 2
Core X
Core N
VRAM
Cores
Global Memory

FIGURE 2.1: Program Overview

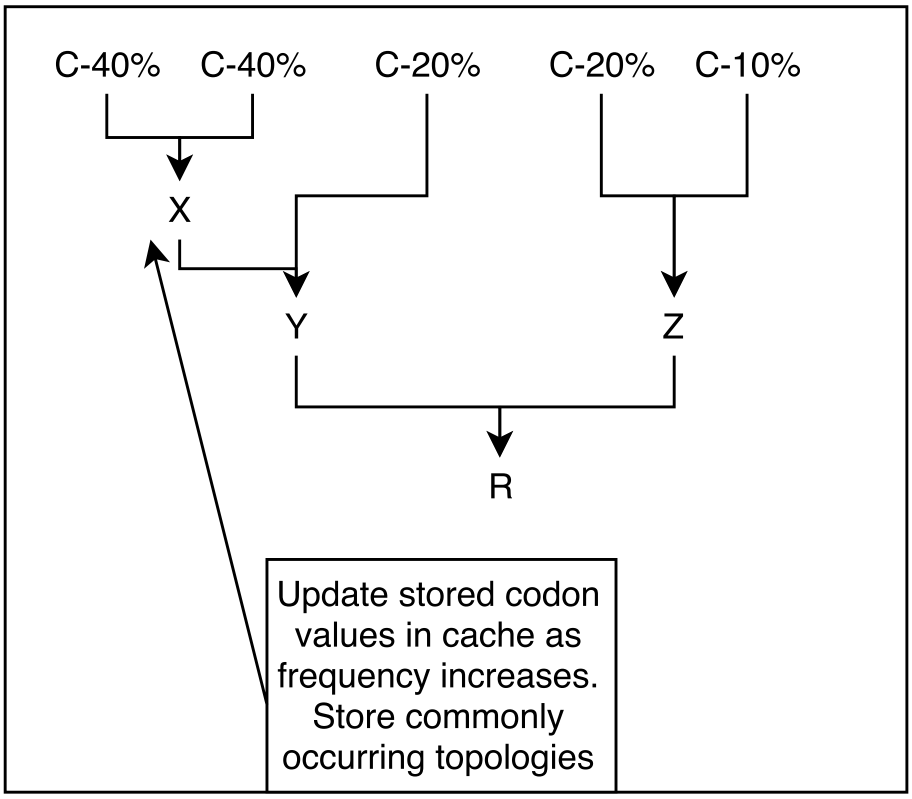
C-40%
C-40%
C-20%
C-20%
C-10%
X
Y
Z
R
Update stored codon values in cache as frequency increases. Store commonly occurring topologies

FIGURE 2.2: Naive approach for caching

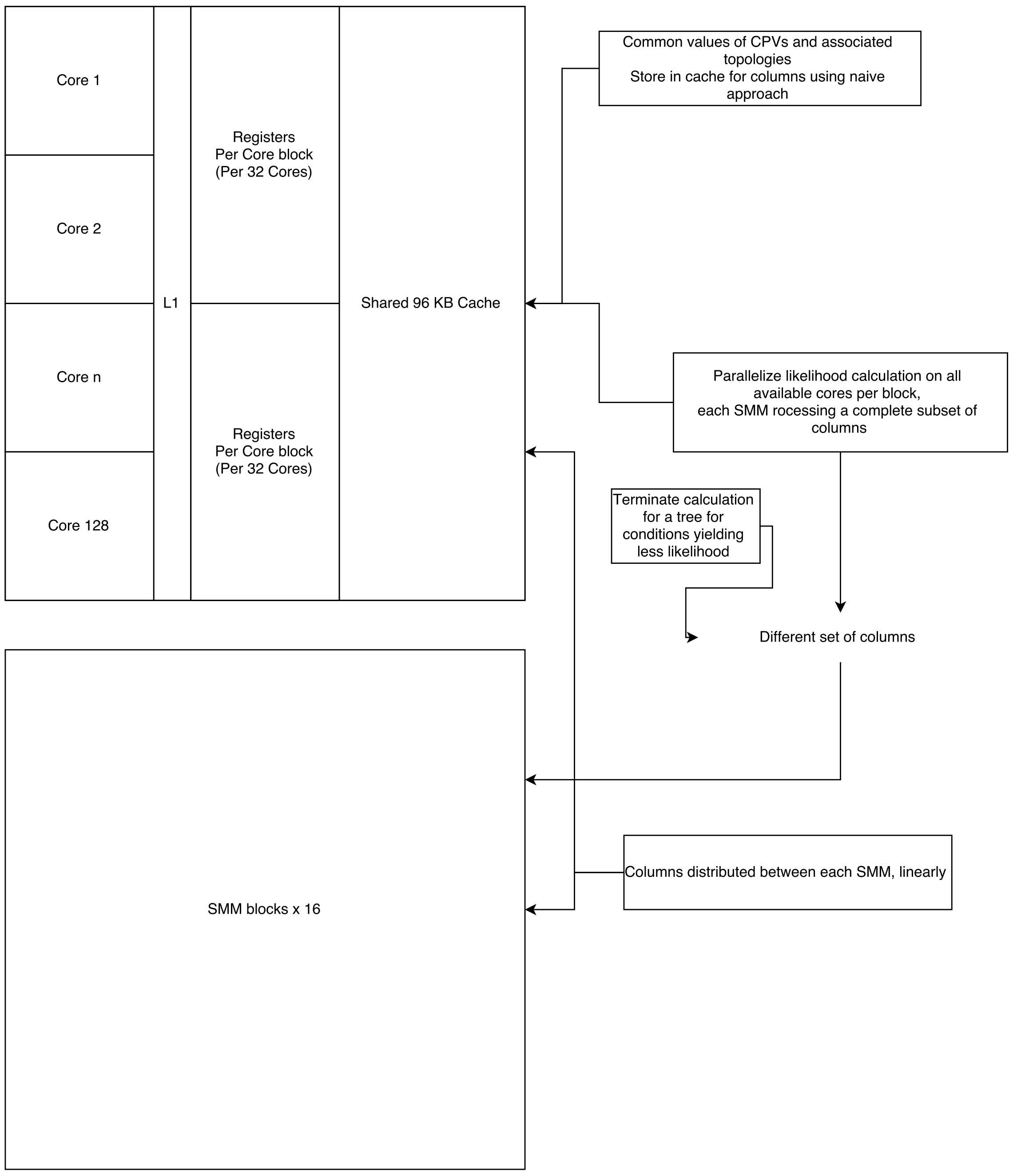
Core 1
Core 2
Core n
Core 128
L1
Registers Per Core block (Per 32 Cores)
Registers Per Core block (Per 32 Cores)
Shared 96 KB Cache
Common values of CPVs and associated topologies Store in cache for columns using naive approach
Parallelize likelihood calculation on all available cores per block, each SMM rocessing a complete subset of columns
Terminate calculation for a tree for conditions yielding less likelihood
Different set of columns
Columns distributed between each SMM, linearly
SMM blocks x 16

FIGURE 2.3: Implementation per SMM

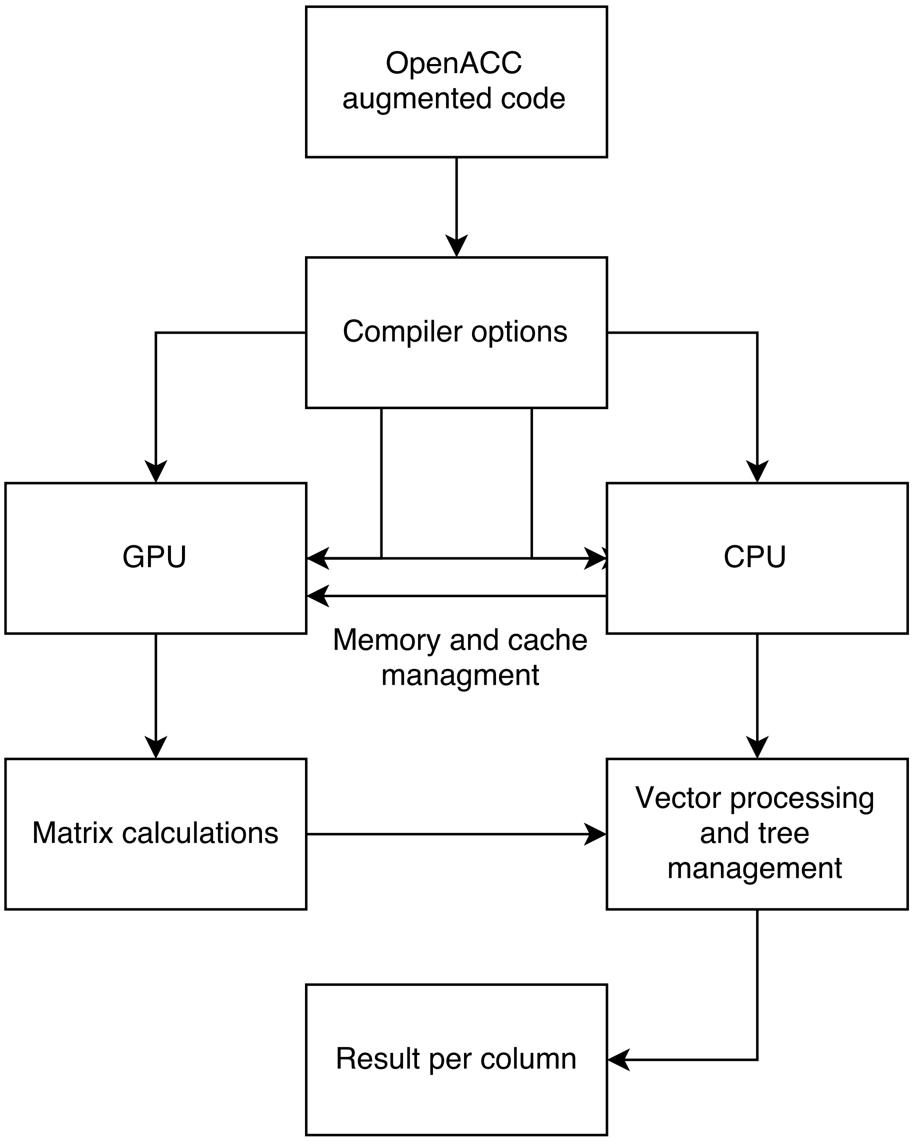

FIGURE 2.4: OpenACC based approach

3 Analysis

3.1 Drawbacks in FastCodeML

3.1.1 OpenMPI

FastCodeML uses OpenMPI as the main driver behind the parallelization done for iterative codes and calculations related to matrices. While OpenMPI is good for a single node, the target system for FastCodeML is based on a clustered with distributed, not shared memory. Thus an OpenMPI based program will not work with same efficiency as a MPI based program, even though it is a modified extension of MPI API itself. This is the main reason why FastCodeML exhibited a strange behavior while being executed on a large cluster, specifically due to inefficient use of threads and issues in memory management.

3.1.2 Coding

The program itself derives its acceleration and edge over PAML by using the highly optimized BLAS and LAPACK libraries and to some certain extent, the caching of previous results in a similar fashion to generic dynamic programming paradigms. While the tree pruning itself generate large amount of "results", managing the results alone would not result in considerable speedup. While the actual time on a system might look small compared to PAML, other factors influence the actual time visible to user, which gives impression of highly unstable behavior of FastCodeML, unlike PAML, which is uniform across all parameters(CPU and user time). Moreover, FastCodeML suffers from deficiencies in optimization of code itself. It is highly depended on the compiler with too many options for the end user related to optimization which might create unstable output, meaning the program can behave unpredictably with different type of compiler optimizations. Moreover, a direct comparison between existing GPU based API (MrBayes) and FastCodeML is not possible, mainly because FastCodeML is based on ML, whereas MrBayes is based on MCMC exploratory search.

3.2 CUDA and OpenCL

3.2.1 CUDA framework and restrictions

One inherent disadvantage of CUDA is that it works only on Nvidia based GPUs, and is not platform independent. Moreover, the rapid rate at which CUDA is evolving makes it difficult to create a code which is compatible with the latest hardware and runtime. Since the start of this project, there has been two major revisions in CUDA, meaning the compiler specification, depreciation of components and addition of newer components have made it difficult to analyze and work with MrBayes. MrBayes was initially designed for Kepler generation GPUs, and the succeeding Maxwell architecture is considerably different from Kepler, with a different framework for managing threads. This requires a complete optimisation of CUDA code to work with newer hardware, which is again supplemented by Pascal architecture, differing considerably from Maxwell. This makes the thread management component of algorithm and program incompatible with newer architecture, thus requiring reanalysis and changes in code.

3.2.2 Hardware restrictions

CUDA based programming is not completely heterogeneous in nature. Instead, it depends on host to manage it the local memory stored in the device, and the device itself cannot access the main memory directly, thus preventing any direct sharing of resources and variables between GPU and GPP. This introduces additional latency required for memory transfer. Moreover, exceptions handling is not explicitly available for CUDA, meaning that a portion of code can crash the complete program under certain conditions. This also makes the CUDA code harder to debug.

3.2.3 Caching and Optimization of probabilities

This by far is the most difficult aspect of both CUDA and OpenCL based approaches, whereby optimization of the previously calculated result is problematic. Since the Branch-site model itself works on ML, there is a large amount of data collected over each node, i.e. the likelihood for the nodes originating from it and preceding it. Storing this data in local blocks is difficult since the local cache is considerably small(usually 2megabytes). Thus it is essential to recycle the values without any significant cost of memory access and transfer, which is not possible in this case. This by far is the biggest drawback we have identified with a GPU only implementation.

3.2.4 OpenCL

OpenCL shares many inherent limitations from CUDA, except for platform dependence. On a hardware level, CUDA and OpenCL share same strategy, thus caching on GPU still creates a problem. Further, a properly optimized OpenCL code is nearly two to three time larger than CUDA code, specially since OpenCL based code need specific instruction for each type of architecture to work optimally. This increases the development time considerably. Apart from this, OpenCL also supports a limited subset of C++ functionality, and some of the most commonly used functions are not implemented in OpenCL, or in CUDA. This creates a problem when converting a tightly integrated program like FastCodeML to a GPU based program. This ultimately reduces the portability of the package, restricting it to a certain environment.

3.3 OpenACC

OpenACC itself has some severe drawbacks compared to other options. Function calls are not supported in parallel regions, which is a major bottleneck, specially since many subroutines in maximum likelihood calculation are situated in the parallel region. Moreover nested parallelism is not supported, meaning low level parallelism of matrix operations is not possible without manual intervention(i.e. specifying the parallel blocks manually using OpenMPI,CUDA,OpenCL).

3.4 Optimal Algorithm

The most optimal algorithm for the implementation would be a modification of FastCodeML, along with OpenACC, as illustrated in Figure 2.4 on page 16. Since the code is already optimized for parallel execution on a multi-threaded system, it is easier to replace the OpenMPI directives with OpenACC directives. The CPVs can be calculated on GPU and can be sorted on CPU for creating an optimal tree based cache, which the the GPU can utilize. This "blocks" of code can coexist on same system and can run parallel on both GPU and CPU, with GPU being the bottleneck. The caching mechanism built into FastCodeML needs to substantially modified to accommodate this conversion, moreover there is a need to reduce the latency between memory transfer between main memory and device memory, via prefetching techniques. Use of OpenACC will make the code compatible with OpenMPI and CUDA based devices, and to some extent with OpenCL based paradigms, making it easier to create a proof of concept program and a benchmark to asses the performance of each individual strategy, and then optimize the program using dedicated strategy like CUDA or OpenCL.

4 Benchmarking-Short Read Aligners

4.1 Introduction

High throughput sequencing is becoming cheaper gradually and as a result the data generated by it is increasing at an exponential rate. A typical Illumina Hi-Seq 2500 run would result in 2 billion to 3 billion short reads ranging anywhere between 100 to 150 bps typically. Processing such large amount of read data requires increased computational resources and traditional computational component, a typical x86 based processor is mostly not sufficient for such tasks in batches. Although a high-end CPU (or Dual/Quad CPU setups involving Intel Xeons) might be enough for a single instance-single study purpose, they may or may not scale well with increase of data in terms of performance. GPUs on the other hand present a completely different scenario as they are highly scalable but programming or even porting something on GPU is difficult, given that a problem needs to be broken into various small parts, and there is requirement of a generalized "solver" which can work on each part independently. Sensitivity is a measure of underlying algorithm used in tools, and thus cannot be directly mapped to the architecture used for running the algorithm, unless a program uses hardware specific instructions(like Intel AVX, SEE etc).

Initial attempts to use GPU for general purpose computation were rather arcane in nature (in Bioinformatics), requiring use of OpenGL or some other third party graphic API to translate data to a graphical (i.e. pixels and vector) instruction format, communicate with subunits and then applying calculations on this temporary floating dataset[12]. Then with the emergence of unified shader architecture from Nvidia and ATI, it was possible to address a single shader, or technically now a computing unit individually. Also, with the launch of Tesla architecture, Nvidia launched a programming interface for their cards. This interface, named as CUDA, essentially allowed a reduced subset of C/C++ to run on GPU. This converted the GPU into a general purpose compute device, albeit with the inherent limitations. Manavski and his group were first to utilize CUDA for the purpose of sequence alignment, by porting Smith-Waterman algorithm to GPU[13]. Initial CUDA implementations were crippled by limited local memory to store a reference sequence, where the memory was in ranges of megabytes. MUMmerGPU was one of the first aligners to efficiently break the reference into smaller pieces for further processing(partially assisted by the tree style data structure used to process alignments in algorithm)[14]. The scenario now is completely different, with high end Tesla GPUs containing 24GB of frame buffer and consumer grade Titan X having 12GB of memory, enough for holding the reference or most of it in the local memory. High throughput sequencing technologies were also being developed in parallel, and data generated by them was large in quantity, but each individual point in dataset was small, creating a large homogeneous dataset. GPUs are designed to work on such datasets, and thus there is a recent trend of porting or developing aligners for such short reads in CUDA (OpenCL and other equivalent implementations exist, and will be discussed later in report). The biggest perceived advantage of a GPU based implementation for short read alignment is speed. While the sensitivity and specificity measurements are comparable to the CPU based counterparts, claims of GPU based implementation to be 5-10x times faster are usually present in the respective publications from authors[10][9].

A simplified algorithm followed by GPUs(included in benchmark) for alignment can be condensed as:

- Build an index for storing reference in a fast access format/GPU compatible format. -Uses CPU[9]
- Seed and find approximate positions
- Align
- Pass additional alignments to CPU.[10]
- Condense the results and convert them to BAM/SAM format.

4.2 Method

We selected following aligners for benchmark:

- BARRACUDA: GPU –v0.7.107a
- SOAP3-DP: GPU/CPU –Source taken from latest release as on 1-August-2015
- Bowtie: CPU –v1.1.1
- STAR: CPU/Splice Aware –v2.4.1d
- HISAT: CPU/Splice Aware –v0.1.6 Beta

Specification of machine:

- CPU: Intel Xeon E5-2640 v3
- RAM: 128GB DDR4
- GPU: 3x Nvidia Titan X (All cards at PCIe 16x electrical lanes-Full Bandwidth)
- Accelerator Card: Nvidia Tesla K40 (At PCIe 16x electrical lanes-Full Bandwidth)
- Chipset: X99
- Graphic Driver Version: 346.59
- OS: Ubuntu 15.04

- Storage: 512GB PCIe 4x SSD
- CUDA Toolkit Version : 7.0

We used the provided binaries wherever possible, to make sure that any of the compiler or OS dependent parameters don't affect performance. For CPU based aligners, we used all 16 threads available for computation. As for GPU aligners, we used Nvidia Titan X as the working device. Although there was a scope of using multiple GPUs by dividing the input, we decided to asses single GPU performance to be more fair against a single CPU, provided that they both fall in similar 1000$ price bracket. No other non-essential tasks were running on the server while benchmarking was done.

4.2.1 Read Generation

We used ART[15] for generating the simulated reads. The reference genome hg19 was downloaded from UCSC Genome browser in condensed format, instead of chromosome by chromosome basis. We used four different read datasets for evaluation, with difference in error rate and sequencer in question(HiSeq 2000 and 2500). First set of reads were generated using default parameters, with following command:

```
art_illumina -sam -i reference.fa -l 100
-ss 25 -c 100000 -o default
```

Second set of reads with a static insertion/deletion rate of 0.0002 were generated using following command:

```
art_illumina -insRate 0.0002 -dr 0.0002
-sam -i reference.fa -l 100 -ss 25 -c
100000 -o twenty
```

Similarly third set was generated by altering the insertion and deletion rate to 0.0004. For the forth set, the substitution rate was 1/10th of default using following command:

```
art_illumina -i reference.fa -qs 10 -qs2
10 -l 100 -c 100000 -s 10 -sam -o
suppressed_er
```

Why ? We used a small number of reads to assure that computations are completed in small period of time, at the same time we also took 1.4 billion reads to measure the speed performance of aligners, but with default parameters from ART. A typical run from a sequencer would ideally result in more than 3 billion reads, but since performance figures(True positive rate etc) are more or less independent of the input data size (the number of reads were variable in all the input, due to tendency of ART to simulate only a certain subset of mappable reads, and the performance figures were still same).

4.2.2 Benchmark

We used Rabema[16] for benchmarking the resultant alignments. First step was to use perfect alignment generated by ART and generate a "Gold Standard" for comparison.

Following commands were used for conversion and sorting perfect alignment using Samtools package[17]:

```
samtools view -Sb sim.sam >sim.bam
samtools sort sim.bam coord
```

And following command was used to generate gold standard using Rabema Oracle mode:

```
rabema_build_gold_standard -oracle-mode
-o sim.gsi -r reference.fa -b coord.bam
```

This step was repeated for all four datasets to create four different gold standards. For actual evaluation, following command was used:

```
rabema_evaluate -r genome.fa -g sim.gsi
-b results.bam
```

Further, no logging was enabled and variant calling performance was not measured. The biggest bottleneck was construction of gold standard, which took considerable amount of time with given amount of reads, and was the main reason behind low number of reads.

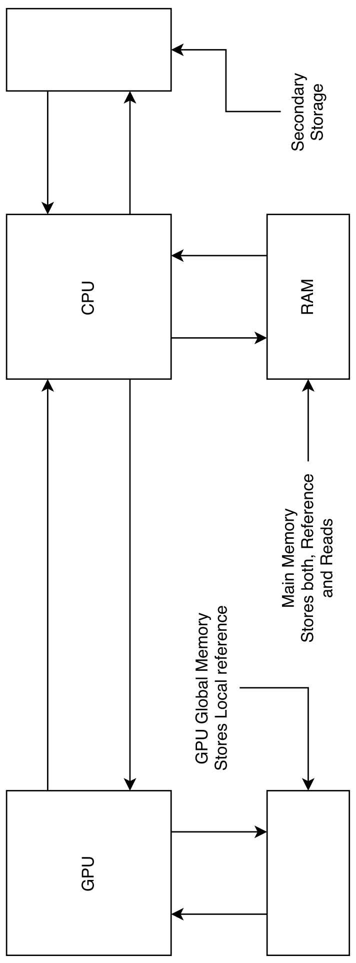

FIGURE 4.1: Basic IO diagram between CPU and GPU. Notice the lack of direct interface between system memory and GPU.

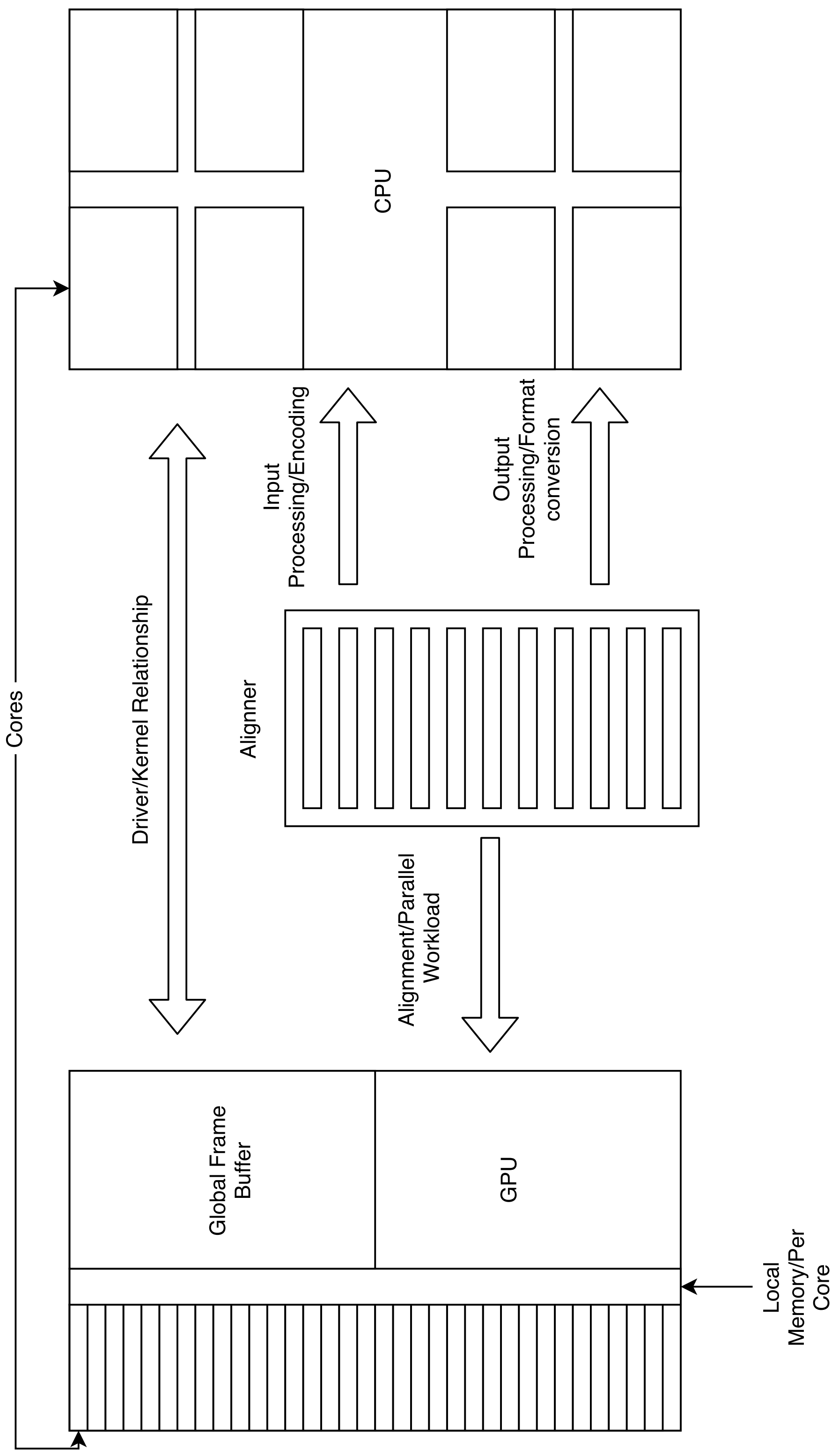
CPU
Input Processing/Encoding
Output Processing/Format conversion
Driver/Kernel Relationship
Cores
Alignner
Alignment/Parallel Workload
Global Frame Buffer
GPU
Local Memory/Per Core

FIGURE 4.2: Approach followed by GPU based aligners

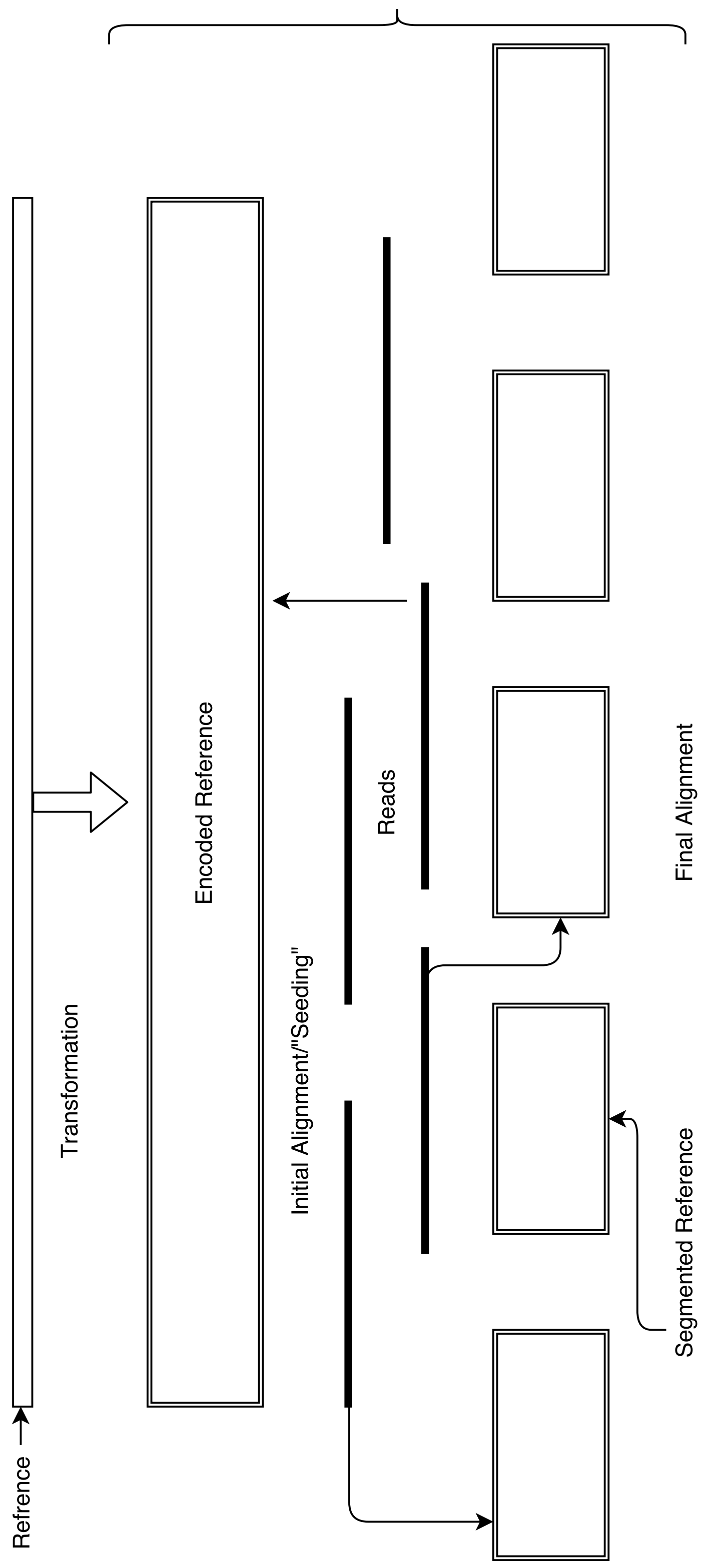

FIGURE 4.3: A simplified representation of short read aligners

5 Benchmarking-Results

5.1 Numbers

After running the benchmarks, it is clearly evident (results in table 5.1) that all three BWA based DNA-seq short read aligners[10][9][18] have quite similar performance statistics. While it is assumed that additional cores in a GPU can give benefit by utilizing multiple instances of same read on multiple core and map with more confidence, the results don't illustrate so. Although Bowtie is oldest of the three, it still performs quite well, at the expense of speed. Mapping performance to number of cores, Bowtie can perform as good as a GPU based aligners in terms of speed on a 14 core system(28 threads), maybe even faster. There was less than 1% difference in mapping performance in every benchmark instance, which is quite interesting since SOAP3-DP claims to use dynamic programming for reads which are difficult to map. More detailed analysis is done in respective subsection:

5.1.1 Speed

The major advantage of using a GPU over CPU is the sheer amount of speed potential it gains from additional cores. While this might be true in applications[19] which use GPU for visualization or empirical calculations, same might not be true in case of sequence alignment. Text processing problems constitute a considerably large subset of algorithms intended to give "exact" and with large datasets "heuristic" based results, and short read sequence alignment falls in between two such sets. While opting for dynamic programming is highly inefficient in terms of speed and resource usage, it can give highest sensitivity, and at the same time heuristic or randomized algorithms can give a speed increase, they tend to be struck on localized result rather than the global result. The three DNA-seq aligners which we tested used a compression algorithm as a mean to store data and perform alignment, by analyzing the seeds in reverse direction. BARRACUDA is a direct implementation of of BW Aligner[20] on GPU, while SOAP3-DP and Bowtie have custom implementation of the indexing algorithm. It is interesting to draw conclusions on the basis of speed. While we didn't use a very large dataset(due to time restrictions), the results were no less exciting with a smaller dataset. Roughly, SOAP3-DP is twice as fast as Bowtie, and both can analyze a typical sequencing run from a Illumina HiSeq 2500 under five hours(3 billion reads of 100bp reads in average). In a typical single run/single study scenario, both aligners are good enough and will not create a bottleneck in downstream analysis, but in a scenario where data is processed in large batches(like in 1000 Genomes project), SOAP3-DP might prove more efficient and will result in considerable time saving. This also holds true where cloud computing instances are used for analysis

Aligner	Mapped-Total	Mapped-Mappable	Mapped Reads	Normalized-Reads	Invalid
SOAP	61.3024	70.5752	5471108	5471110	1181899
BARRACUDA	61.9401	71.3093	5528022	5528022	1164811
STAR	85.0033	97.8612	7586367	7586367	2355542
HISAT	80.1001	92.2163	7148765	7148765	1759303
BOWTIE	61.8914	71.2533	5523675	5523675	1167950
10% of Default Profile					
SOAP	65.5621	70.6531	5851009	5851010	644553
BARRACUDA	66.1899	71.3298	5907041	5907040	639414
STAR	90.8174	97.8696	8104893	8104890	1340261
HISAT	85.5818	92.2275	7637647	7637650	976293
BOWTIE	66.127	71.2619	5901425	5901420	640213
Insertion/Deletion Rate: .0002					
SOAP	61.2994	70.5918	5470648	5470650	1174358
BARRACUDA	61.9263	71.3138	5526601	5526600	1166150
STAR	84.9757	97.8572	7583633	7583630	2356786
HISAT	80.0441	92.1781	7143517	7143520	1765900
BOWTIE	61.8584	71.2355	5520534	5520530	1167608
Insertion/Deletion Rate: .0004					
SOAP	61.2876	70.6303	5469388	5469390	1179408
BARRACUDA	61.8975	71.3333	5523823	5523820	1169078
STAR	84.9194	97.8646	7578328	7578330	2365425
HISAT	80.0044	92.2003	7139702	7139700	1770081
BOWTIE	61.8348	71.261	5518225	5523820	1163466

TABLE 5.1: Numerical performance across parameters

pipeline, whereby the restrictions are on basis of time, not hardware. In such scenario GPU based aligners might be more advantageous.

Considering the cost of hardware, the GPU which we used was selling for $990 per unit at the time of writing, while the CPU was selling at $980 per unit. In a lab where datasets are usually small or downstream analysis in limited to low number of samples(10-15 on average on whole genome), a CPU would be beneficial, mainly because of the versatility it can provide as it can run many downstream analysis softwares which are yet not available for GPU(for variant analysis etc). GPUs are currently restricted to limited availability of softwares specifically made to utilize the built in wide SIMD implementation. This scenario might change in future, due to aggressive efforts by Nvidia to introduce CUDA based hardware and development tools to researchers. Recently, Intel also introduced MIC(Many Integrated Core) architecture, which is essentially a set of multiple x86 processors on a fast topology to transfer data between them. The biggest advantage of such architecture is that many of the preexisting libraries and programming tools are optimized for x86 architecture, thus it requires considerably less effort to parallelize code on Xeon Phi(the commercial name of architecture). Researchers have used Xeon phi to create extremely fast short read aligner(named MICA), but the results are quite new and the tool is still not validated by any third party study[21]. The paper clearly illustrates scalability of a traditional architecture, but unlike CUDA, which is already widespread now, Xeon Phi and related development ecosystem will still take time to mature. We can safely assume that in future we may see more short read aligners implemented on MIC. The biggest problem with GPUs for general purpose computing is that they can implement only a limited subset of commonly used languages like C++, and the memory management in GPUs is drastically different than CPU. There are various levels of memory in a GPU, and while the local memory is now large enough to hold whole reference genome, the local cache and memory available to per core can limit the performance with larger reads(BW Transformation was mainly used in context of limited memory). Even after initial alignment is done, the output needs post processing to convert it to a more standard format like BAM/SAM. In case of a traditional CPU, even though the number of cores are in range of 6-14, each core can access the main system RAM directly, and can performance complex operations without any explicit intermediate process. These give CPU a distinctive advantage over GPU. While GPUs can parallelize tasks, the amount of optimization required is high and the increase in performance is sometimes restricted by the algorithm in question.

We also tested STAR and HISAT, just to check their speed an performance in DNA seq data. The results clearly illustrated the tendency of a splice aware aligner to align everything which is possible. In terms of speed, they don't have any contemporary rivals from DNA-seq aligners, regardless if they are either GPU or CPU based. STAR was fastest, taking 36 seconds to process nearly 800000 reads, while HISAT was close enough with 48 seconds. This should potentially allow use of these aligners in a batch scenario, even in standalone machines with decent specifications. But researchers should use them on DNA-seq experiments cautiously, mainly because of their high invalid alignment rate.

5.1.2 Performance

There is not much to discuss in terms of performance, as mentioned above, the performance of the GPU based aligners and Bowtie is nearly similar. We didn't test the overlap of alignments in result, but that can elucidate algorithmic differences. Variant calling performance was not measured due to time restrictions, but there exist a complete benchmarking pipeline for that purpose[22] which I plan to use later to access the sensitivity. Splice aware aligners cannot be used for variant calling, mainly because of high rate of invalid alignment. Barracuda performed better, if we take a precision till three decimals which can be significant if there is a very large dataset but for smaller datasets(i.e. $<$ 1 billion reads), the performance in quite similar. STAR and HISAT, both have a very high sensitivity and mapped more than 80% of reads in nearly every case, with STAR generally mapping 5% more reads. This was at the expense of high number of invalid reads, sometimes more than 15% of total dataset, which is not good enough for variant calling. Biologists can use such tools just for primitive mapping for finding approximate locations of reads on reference genome, and then can use dynamic programming based algorithms for making those mappings exact. Invalid alignments are those which show deviation of more than 4% from the gold standard. HISAT presents an interesting case, as it show a decreased number of mapped reads, but a considerable decrease (8% or more) in invalid alignments too, and therefor can be used to map DNA-seq reads better, unlike STAR, which is suitable only for RNA(HISAT is built on top of Bowtie2, and hence better performance in terms of DNA-seq reads).

TABLE 5.2: Average time in seconds

aligner	time
BARRCUDA	183.73
SOAP3-DP	91.32
BOWTIE	234
STAR	36
HISAT	48

5.1.3 Discussion

The results clearly show that GPU based aligners are faster than CPU based aligner, giving a speed advantage of nearly 2x(SOAP3-DP), and a better accuracy(Barracuda). But it is also worth noting that the performance figures let alone cannot account of blind use of GPUs in this scenario. The prices of high end consumer GPUs are equal to a good server grade processor, and dedicated accelerator cards like Tesla cost 3 to 4 times more than their consumer counterparts. This may hinder large scale use of GPUs in labs where small scale studies are performed. Also, the versatility of a CPU allows researchers to run nearly all of the downstream analysis programs on a single machine, and massive parallelization may give speed boosts equivalent or better than GPUs. Also, recent advances in secondary analysis may increase decrease the gap in performance, given that now the core problem is to assure that the resources are utilized properly. Recently, a study sponsored by Intel illustrated the importance of preprocessing data into format which can remove bottleneck related with amount data available to each processing node[23]. In the end, we can't really say from the tests about the "best" aligner, but it is quite evident from the results that GPU aligners tend to be faster than the CPU aligners, even when they share some underlying process. In year 2016 Nvidia launched their own version of "Heterogeneous" GPU (already launched by AMD in 2015), which can work in tandem with CPU to accelerate parallel tasks automatically, without explicit programming constructs from developers.

6 Conclusion

This thesis was done in two parts. The seventh semester was primarily dedicated to applications of GPU based computing in field of Bioinformatics, specifically Short read aligners. This was primarily done to analyze the actual efficacy of GPUs and to asses if GPUs can replace traditional CPUs in high performance task. The eighth semester was dedicated to development of a GPU based algorithm in field of Bioinformatics. We tried to develop an algorithm for searching positive selection using GPU by implementing Branch-Site model. In due course, we learned more about the peculiarities of development related with GPU, and learned implementation of a computationally intensive problem on a parallel architecture.

6.0.1 Conclusion: I

As for the conclusion to first part, we identified the main obstacle in implementation of a fairly straight forward mathematical model. The reason behind lack of a GPU based implementation for searching positive selection is mainly due to the inefficiency of caching local variables. Although it might be possible, we were unable to find a solution in stipulated time frame. The problem was made more complex by the fact that tree pruning generates considerable amount of data, even if for short duration which requires sizable working memory, something which is not possible for each individual processing unit in a GPU. Considering the advent of new programming tools like OpenACC, it is possible to convert the preexisting program for a highly parallel architecture like a GPU, but as of now the technology is still not suitable for direct replacement. The basic structure of problem prohibits use of OpenACC for complete acceleration. Still, given time and resources, it can be used to identify best programming paradigm for the problem, either CUDA or OpenCL and then the developer can utilize any one of them as the main parallel directive(depended on the hardware).

6.0.2 Conclusion: II

For the second part of thesis, we believe that GPUs, if used efficiently can provide considerable speedup. Although the nature of problem differed considerably(Alignment versus Prediction), even sequence alignment was considered to be unachievable on GPU before. Our results show how much the advanced the GPUs have become. There was a performance increase of at least 2x for even worst case, validating our believe that GPUs are better suited for parallel computations than CPUs.

6.0.3 Limitations of this study

The primary limitation of this study is that we were not able to provide a proof of concept for the first part of thesis. Given that we misjudged the complexity of problems and this lead to a large gap in the end. While we conceptualized an algorithm and program in theory, we still can't say with absolute faith that a GPU based solution in current condition would work best for a problem related to phylogenetics. Calculations alone cannot validate an theory without an experiment to back it. As for the second part, we believe that the study was rounded up quite quickly, and still stands unique since there is no review or subjective comparison of GPU based tools used in NGS pipeline. Also, the results were validated multiple times to ensure that there are no discrepancies. Per se, there are no limitations but there are future possibilities, which will be discussed in next subsection.

6.0.4 Future possibilities

Given that we were not able to create a proof of concept program for first part, and there exists no program on GPU for searhing positive selection using Branch-Site Model, it is safe to assume that the outlines can be enough to provide a quick start for the tool itself from scratch. And as for the second part, we didn't measure variant calling performance, which can be measured to identify the aligners which can be best suited for areas where mutation study is essential, like oncology. Moreover, it is possible to combine the existing tools to create a NGS data processing pipeline completely based on GPU, which can compete with existing pipelines like Casava.

I would like to conclude the study hoping that the future batches would find it useful, and will be interested in pursuing the cutting edge technology which has a potential to change the field of computational biology drastically.

Bibliography

[1] Z. Yang, “PAML 4: phylogenetic analysis by maximum likelihood”, *Mol. biol. evol.*, vol. 24, no. 8, pp. 1586–1591, 2007.

[2] D. L. Ayres, A. Darling, D. J. Zwickl, P. Beerli, M. T. Holder, P. O. Lewis, J. P. Huelsenbeck, F. Ronquist, D. L. Swofford, M. P. Cummings, A. Rambaut, and M. A. Suchard, “BEAGLE: an application programming interface and high-performance computing library for statistical phylogenetics”, *Syst. biol.*, vol. 61, no. 1, pp. 170–173, 2012.

[3] S. Guindon, J. F. Dufayard, V. Lefort, M. Anisimova, W. Hordijk, and O. Gascuel, “New algorithms and methods to estimate maximum-likelihood phylogenies: assessing the performance of PhyML 3.0”, *Syst. biol.*, vol. 59, no. 3, pp. 307–321, 2010.

[4] C. L. Hung, Y. S. Lin, C. Y. Lin, Y. C. Chung, and Y. F. Chung, “CUDA ClustalW: An efficient parallel algorithm for progressive multiple sequence alignment on Multi-GPUs”, *Comput biol chem*, vol. 58, pp. 62–68, 2015.

[5] A. Gudy and S. Deorowicz, “QuickProbs–a fast multiple sequence alignment algorithm designed for graphics processors”, *Plos one*, vol. 9, no. 2, e88901, 2014.

[6] C. Ling, K. Benkrid, and T. Hamada, “High performance phylogenetic analysis on cuda-compatible gpus”, *Sigarch comput. archit. news*, vol. 40, no. 5, pp. 52–57, Mar. 2012, ISSN: 0163-5964. DOI: 10.1145/2460216.2460226. [Online]. Available: http://doi.acm.org/10.1145/2460216.2460226.

[7] M. A. Suchard and A. Rambaut, “Many-core algorithms for statistical phylogenetics”, *Bioinformatics*, vol. 25, no. 11, pp. 1370–1376, 2009.

[8] M. Valle, H. Schabauer, C. Pacher, H. Stockinger, A. Stamatakis, M. Robinson-Rechavi, and N. Salamin, “Optimization strategies for fast detection of positive selection on phylogenetic trees”, *Bioinformatics*, 2014.

[9] P. Klus, S. Lam, D. Lyberg, M. S. Cheung, G. Pullan, I. McFarlane, G. S. h. Yeo, and B. Y. Lam, “BarraCUDA - a fast short read sequence aligner using graphics processing units”, *Bmc res notes*, vol. 5, p. 27, 2012.

[10] R. Luo, T. Wong, J. Zhu, C. M. Liu, X. Zhu, E. Wu, L. K. Lee, H. Lin, W. Zhu, D. W. Cheung, H. F. Ting, S. M. Yiu, S. Peng, C. Yu, Y. Li, R. Li, and T. W. Lam, “SOAP3-dp: fast, accurate and sensitive GPU-based short read aligner”, *Plos one*, vol. 8, no. 5, e65632, 2013.

[11] K. Kabir, A. Haidar, S. Tomov, and J. Dongarra, "On the design, development, and analysis of optimized matrix-vector multiplication routines for coprocessors", English, Lecture Notes in Computer Science, vol. 9137, J. M. Kunkel and T. Ludwig, Eds., pp. 58–73, 2015. DOI: 10.1007/978-3-319-20119-1_5. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-20119-1_5.

[12] W. Liu, B. Schmidt, G. Voss, and W. Muller-Wittig, "Streaming algorithms for biological sequence alignment on gpus", *Parallel and distributed systems, ieee transactions on*, vol. 18, no. 9, pp. 1270–1281, 2007, ISSN: 1045-9219. DOI: 10.1109/TPDS.2007.1069.

[13] S. A. Manavski and G. Valle, "CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment", *Bmc bioinformatics*, vol. 9 Suppl 2, S10, 2008.

[14] M. C. Schatz, C. Trapnell, A. L. Delcher, and A. Varshney, "High-throughput sequence alignment using Graphics Processing Units", *Bmc bioinformatics*, vol. 8, p. 474, 2007.

[15] W. Huang, L. Li, J. R. Myers, and G. T. Marth, "ART: a next-generation sequencing read simulator", *Bioinformatics*, vol. 28, no. 4, pp. 593–594, 2012.

[16] M. Holtgrewe, A. K. Emde, D. Weese, and K. Reinert, "A novel and well-defined benchmarking method for second generation read mapping", *Bmc bioinformatics*, vol. 12, p. 210, 2011.

[17] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, and R. Durbin, "The Sequence Alignment/Map format and SAMtools", *Bioinformatics*, vol. 25, no. 16, pp. 2078–2079, 2009.

[18] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg, "Ultrafast and memory-efficient alignment of short DNA sequences to the human genome", *Genome biol.*, vol. 10, no. 3, R25, 2009.

[19] D. A. Case, T. E. Cheatham, T. Darden, H. Gohlke, R. Luo, K. M. Merz, A. Onufriev, C. Simmerling, B. Wang, and R. J. Woods, "The Amber biomolecular simulation programs", *J comput chem*, vol. 26, no. 16, pp. 1668–1688, 2005.

[20] H. Li and R. Durbin, "Fast and accurate short read alignment with Burrows-Wheeler transform", *Bioinformatics*, vol. 25, no. 14, pp. 1754–1760, 2009.

[21] R. Luo, J. Cheung, E. Wu, H. Wang, S. H. Chan, W. C. Law, G. He, C. Yu, C. M. Liu, D. Zhou, Y. Li, R. Li, J. Wang, X. Zhu, S. Peng, and T. W. Lam, "MICA: A fast short-read aligner that takes full advantage of Many Integrated Core Architecture (MIC)", *Bmc bioinformatics*, vol. 16 Suppl 7, S10, 2015.

[22] J. C. Mu, M. Mohiyuddin, J. Li, N. Bani Asadi, M. B. Gerstein, A. Abyzov, W. H. Wong, and H. Y. Lam, "VarSim: a high-fidelity simulation and validation framework for high-throughput genome sequencing with cancer applications", *Bioinformatics*, vol. 31, no. 9, pp. 1469–1471, 2015.

[23] B. J. Kelly, J. R. Fitch, Y. Hu, D. J. Corsmeier, H. Zhong, A. N. Wetzel, R. D. Nordquist, D. L. Newsom, and P. White, “Churchill: an ultra-fast, deterministic, highly scalable and balanced parallelization strategy for the discovery of human genetic variation in clinical and population-scale genomics”, *Genome biol.*, vol. 16, p. 6, 2015.