UNIVERSITY OF CALIFORNIA

SANTA CRUZ

SMARTSSD FOR GENOMICS

A Thesis submitted in partial satisfaction of the

requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE AND ENGINEERING

by

Sachet Mittal

September 2020

The Thesis of Sachet Mittal
is approved:

Professor Heiner Litz, Chair

Professor Scott Beamer

Professor Benedict Paten

Quentin Williams
Interim Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

SmartSSD for Genomics

by

Sachet Mittal

In this thesis, I introduce the SmartSSD architecture and list the hardware and software requirements for the system. Then I give a brief overview of the domain of genomics and discuss suitability for this architecture. I then evaluate two workloads and profile them for amenability for this architecture. In this process, I provide key insights that aid in the process and develop a methodology for the using and developing on this system.

Acknowledgments

I would like to thank my advisor Professor Heiner Litz for guiding me throughout this project, Western Digital, for providing support for this research project, Genomics Institute, for providing me with relevant domain expertise and application-specific issues. And my reading committee Professor Heiner Litz, Professor Scott Beamer, and Professor Benedict Paten.

Part I

Background

Chapter 1

Introduction

SmartSSD, respectively *Computational Storage*, is a hardware architecture that processes data at the storage layer rather than the compute layer[8]. This architecture places a compute unit on the storage device, near the storage device, or between the CPU and the storage.

The benefits of this architecture are increased parallelism, reduced data movement, and lower energy costs on the server. In this thesis, I explore these benefits for applications in Genomics. The research for the work is done in collaboration with Western Digital.

Genomics is transforming humanity's approach to health-care [15] [11]. This transformation has lead to the development of more accurate and efficient genomics algorithms. Many of the critical applications in Genomics have a significantly high data footprint that leads to a lot of data traffic to and from the storage, which impacts the overall performance and energy costs of the data center. Thus, there is a need for a hardware implementation that addresses this. SmartSSD is an architecture

that helps alleviate this by offloading a part of the algorithm to be performed at the storage layer. In this thesis, I discuss two critical algorithms used in Genomics and profile them to accelerate them by using a SmartSSD. A novel workflow and methodology are developed for these specific examples that subsequently is generalized so that it can be applied to any other workload for SmartSSD acceleration.

In this thesis, I provide an overview of the SmartSSD architecture and describe the properties that an application exhibits that make it suitable for SmartSSD. Then I explain why Genomics is a domain that is well suited for this acceleration. I describe the generic properties that most applications in genomics exhibit, I evaluate two specific applications and examine properties and discuss suitability as SmartSSD workloads.

After the analysis mentioned above, I present a methodology that will aid future work for other applications. This methodology helps decide if an application is a candidate for SmartSSD, evaluates if a specific part of the algorithm can be offloaded to the storage layer (henceforth called *kernel*), and finally the steps to follow for offloading a computation kernel. The goal of thesis is to develop a methodology that aids analyzing workloads and not design the actual hardware.

The applications that I chose are MarginPhase [13] and minimap2 [19]. The chapters that follow describe the details of these applications.

1.1 SmartSSD

A SmartSSD enables high-performance accelerated computing closer to flash storage, bypassing CPU and memory limitations. SmartSSDs, when powered by FP-

GAs, increase performance and efficiency while lowering operating costs by pushing intelligence to where data resides.

The SmartSSD architecture couples SSD drives with FPGA accelerators to speed up data processing and analytics. Data is processed directly on the SmartSSD, before it reaches the host CPU, avoiding significant data movement between the storage and the CPU and speeding up the application and reducing costs associated with this data movement.

In industry, Samsung and Bittware (a Molex Company) are the major players in the field. Samsung's SmartSSD Computation Storage Drive ® [6] is commercially available, and the various domains that it can be used are life sciences, data science, financial services, video processing, image analysis, log processing, big data processing, and machine learning. Bittware's Computational Storage Service ® (CSS) is another product available. The focus areas of CSS research are database acceleration, big data processing, content delivery, and machine learning. Figure 1.1 is the architecture of CSS.

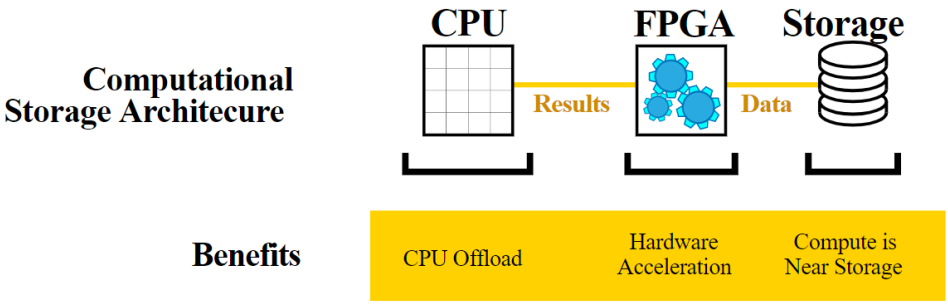

Figure 1.1: Bittware's SmartSSD Block diagram [2]

The benefits of this architecture are:

1. **Improved parallelism**

   As work is offloaded to the Storage Layer, other threads can be scheduled on the CPU to be executed.

2. **Less data movement**

   The data is available at the storage layer, and the results of the computation or the processed data are sent to the CPU. Communicating the compute operation generally reduces the communication relative to transferring the actual data. This is especially relevant when the SmartSSD is accessed over the network.

3. **Power savings**

   There is a cost associated with the movement of data. As the amount of data that is transferred from storage to the CPU is reduced, the CPU consumes less power. SmartSSD enables reducing the overall power consumption of storage applications.

4. **Better utilization of cache for small data tasks**

   Perform parts of the program that operate on smaller data (smaller than typical processor cache size) and offload more data-intensive tasks.

5. **Accelerated processing of data**

   As computation is offloaded, a specialized accelerator for the workload can be designed using the FPGA.

1.2 Genome sequencing

This section introduces the domain in which I discuss SmartSSD acceleration. In this thesis, the workloads I selected belong to Genomics domain so a brief understanding on the domain is called for.

The Genome Sequencer cannot read the whole genome sequence contiguously in one go. So it collects multiple reads that are fragments of the original genome. The process of defragmenting these reads to form a *chain* that represents the whole genome is called Genome Sequencing. The resultant chain is used for further analysis. The details of some of these analysis steps are explained in the following sections.

The Genome Sequence Pipeline involves collecting the genomic reads using a sequencer. These reads are to computational data (Base-calling). As the resultant set of reads have a large data footprint, they are uploaded to the Cloud for storage. The next steps are cleaning up the reads to form the whole-genome (Assembly and alignment) and performing the operations on this whole-genome to infer various genomic properties. Some of these properties include classifying inherited characteristics and genealogy.

In the next section, I discuss phases in the pipeline that I focused my work on in detail.

1.2.1 Sequence Alignment

The process of sequence alignment can be visualized as shredding multiple copies of a book and throwing away a chunk of the shredded pieces and then recreating the original contents of the text from the shreds.

Step	**Input**	**Input Size**	**Output**
Sample preparation	Biological sample	10+ uL	single-strand DNA
Nanopore recording	single-strand DNA	10+ uL	Signal (FAST5)
Base-calling	Signal (FAST5)	500 GB	Reads (FASTQ)
Upload data to Cloud	Reads (FASTQ)	500 GB	Reads (FASTQ)
Assembly	Reads (FASTQ)	200 GB	Reference Sequences (FASTA)
Alignment	Reads, Reference	200GB	Aligned reads (BAM)
Polishing	Aligned reads, Reference	200GB	Reference Sequences (FASTA)
Phasing	Aligned reads, Reference	100GB	Haplotypes, Variants (VCF)
Variant Calling	Haplotypes	100GB	Variants (VCF)

Table 1.1: steps in Genome sequencing Pipeline along with the input data size and format

If the original book is available for reference, then it is called *reference guided genome inference*, and if not, then it is called *De Novo assembly.*

Figure 1.2 [25] shows the reference guided genome inference process. Reference

Genomes are standardized for a species. The set of reads are collected using the biological method (the exact nature of this is out of the scope of this work). Using the Reference Genome, the reads are mapped using an alignment algorithm. A standard alignment algorithm is *Smith-Waterman* [10].

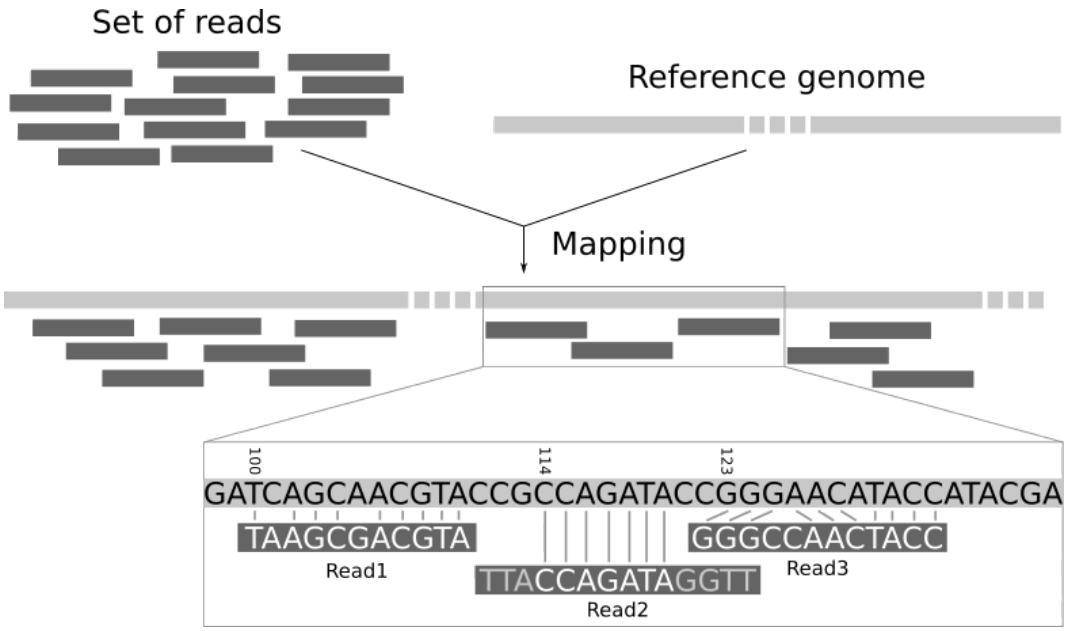

Figure 1.2: Illustration of the mapping process. The input consists of a set of reads and a reference genome. In the middle, it gives the results of mapping: the locations of the reads on the reference genome[25]

1.2.2 Alignment Matrix

The reads that were aligned to a reference genome in a previous step are given as input in the form of an alignment matrix (see Figure 1.3). The rows are individual reads, and the columns are the genetic sites.

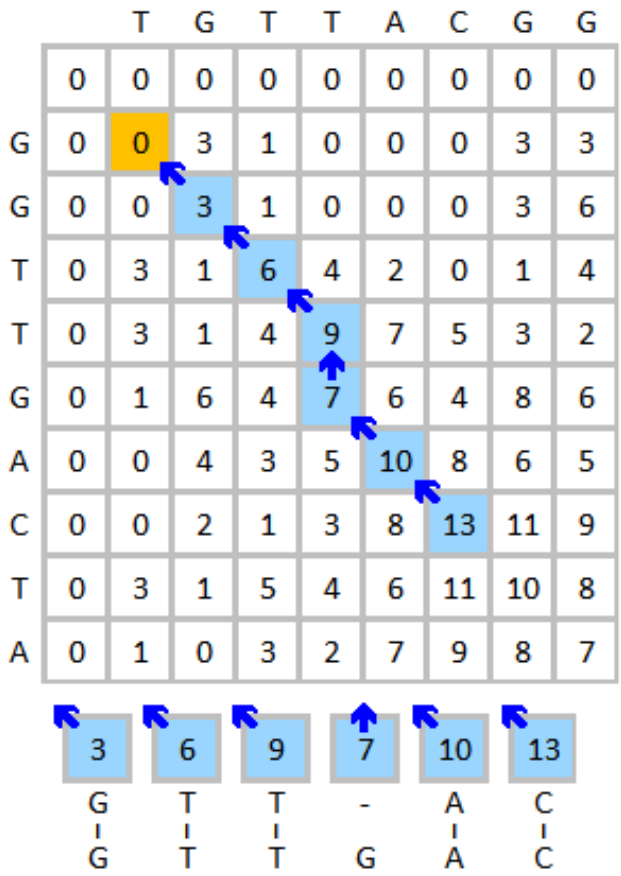

Figure 1.3: Example of an alignment matrix. Together the reads can be used to provide the best guess for the full sequence [1]

1.2.3 Genotyping

Genotyping is the process of determining differences in genetic makeup (genotype) of an individual to another sample or a reference sequence. Genotyping is also used to reveal personality traits inherited from parents.

1.2.4 Haplotyping

A haplotype is a particular sequential pattern on a single chromosome. Haplotyping (or phasing) is the process of determining haploid DNA sequences (haplotypes) from unordered (unphased) genotype data.[14]

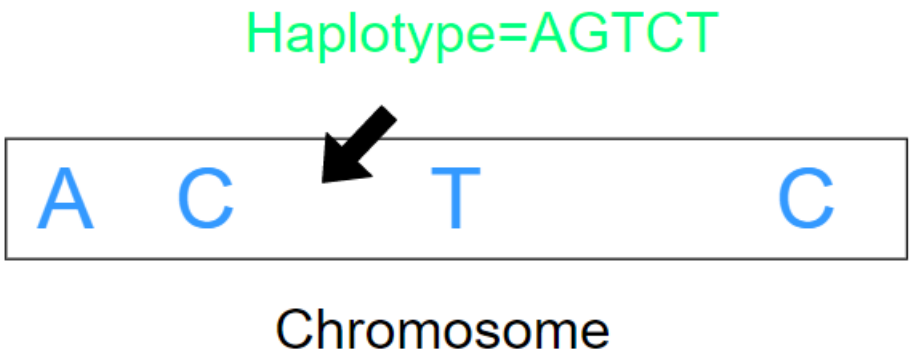

Figure 1.4: Example haplotype

1.3 Tools Used

To determine if SmartSSDs can accelerate the workloads, I used the below-mentioned tools for profiling. The following sections provide an introduction to the various tools I used in this work.

1.3.1 Perf

Perf is a performance analysis tool in Linux. Perf profiles the code in both user and kernel space. It provides stats for various hardware performance counters, software performance counters, tracepoints, and dynamic probes. It is one of the most commonly used profiling tool [5]

1.3.2 Valgrind and Callgrind

Valgrind is programming that was initially developed for memory debugging, memory leak detection, and profiling. It has since evolved into a generic framework for creating dynamic analysis tools. In this work, I use Callgrind, which is a tool in Valgrind.

Callgrind is a profiling tool that records the call history among functions in a program's run as a call-graph. By default, the collected data consists of the number of instructions executed, their relationship to source lines, the caller/callee relationship between functions, and the number of such calls.

1.3.3 pyCparser

pyCparser is a parser for the C language, written in pure Python. It is a module designed to be easily integrated into applications that need to parse C source code. I use pyCparser for parsing the source code of the application and extract the effective lines of code for the kernel that includes function calls in the given computation kernel and lines of code in the called function.

Chapter 2

SmartSSD architecture

This chapter describes the basic architectural and software interface requirements that a SmartSSD architecture should satisfy. These requirements are generic and not limited to the problem domain of Genomics.

2.1 Architectural Requirements

1. An FPGA is connected to the NVMe [16] port of the disk that can intercept the IO requests.

2. Storage (preferably SRAM/DRAM) to store offloaded instructions in the SmartSSD. This storage is being referred to as *Code Blocks*, where the execution logic can be written into and modified. Each of these *Blocks* is identified by a unique number from 1 to N, where N is the number of Code Blocks.

3. The FPGA has shared memory that is accessible from all the Code Blocks. This needs to be shared so that multiple data transformations to the data can

be applied in-place. An example of this is compressing and encrypting the data.

4. The component that is responsible for intercepting the IO requests as per requirement 1 is called the interceptor

5. The interceptor will allow standard and native IO requests to pass through to the SSD disk

6. The computation IO request payload contains the Code Block identifier. These requests are routed to the specified code block.

7. The communication workflow of this architecture is shown in Figure 2.2

8. Each Code Block has a job queue. As soon as the job is placed on this queue, the IO request returns. The application can poll the status of the job.

9. After the job is completed, the job moves to the completion buffer. When this job is polled, the results are returned.

10. The architecture needs to support logging and monitor the system's health. These are out of the scope of this thesis.

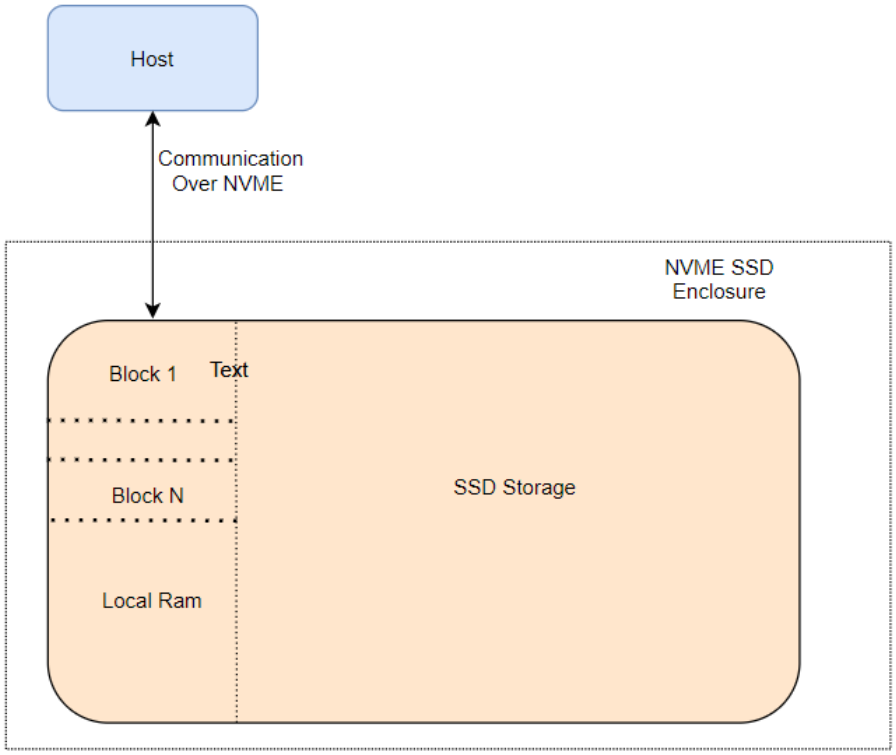

Figure 2.1: The block diagram for the SmartSSD

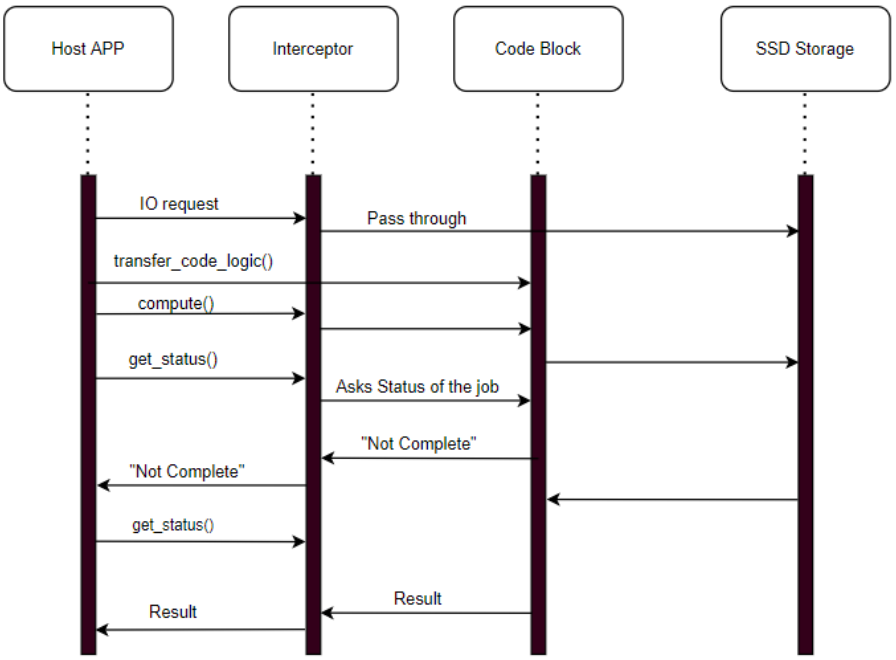

Figure 2.2: Communication Workflow for the architecture

2.2 Software Interface

This section describes the API that the SmartSSD needs to support. The API is divided into two types:

1. Management API

   - `int get_block_information()`

     Return the total number of empty blocks of the SmartSSD

   - `int get_empty_block()`

     Returns the first available empty block if any else returns 0

   - `int transfer_code_logic(int block_number, void *kernel_logic)`

Transfers the logic specified by the *kernel_logic* data structure to the block specified by *block_number*.

2. IO API

- `int write(int block, int offset, void *data)`

  Writes data to logical block number plus offset. For more details on these, refer [3] and [4].

- `void * read(int block, int offset)`

  Returns from logical block and offset

- `int  compute(int block_number, int logical_block, int offset)`

  Place the request on the job queue and returns the job id

- `void get_status(int job_id, void *result)`

  Get the status of the job or the results if the job is completed.

Chapter 3

Determining the Suitability of a Workload

This chapter describes the properties of a suitable workload for this architecture. The properties are segregated into compute and data-structure properties. Native Linux tools and pyCparser are used to determine the value of these properties. The specifics of which tool to use is described in Table 3.1 and Table 3.2. At a high level, as the goal of the architecture is to speed up the storage layer, an amenable application spends significant time at the disk and spends a majority of time data processing.

Property	**Desired Value**	**Reason**	**Tool**
Computationally expensive kernel	>10% of the pipeline's computation time	According to Amdahl's law, the speedup is limited by the acceleration factor, so there needs to be an expensive kernel to get a speedup	Perf
IO Bound	>10 GB/s, >50% time spent in kernel/filesystem/-driver	The speedup in this accelerator is achieved by reducing the amount of IO	Perf
Compute to space ratio	>50 % cache miss rate	Allows precomputation, redundancy, compute ->space overheads	Perf
Parallelism	100+ threads	This architecture adds more cores to the system, so for optimal utilization, the application needs to be multithreaded	htop or psutils
Dynamic Instructions per SLOC	1M (10^6 ins/-line)	Due to the trade-off of porting efforts with performance gains	pyCparser + gcc

Latency	Insensitive, tolerate 10-100μs per access	some of IO wait time can overlap with some compute	Valgrind
kernel compute complexity	Single kernel >1ms computation time	The kernel should perform a significant compute in each call	Perf
High Loop Count in kernel	>1,000 iterations	The kernel should perform simple operation a large number of times	Valgrind + pyCparser
Recursion Free	The repeated computation comes from a loop and not due to recursion	To reduce the efforts in Verilog synthesis of the kernel	Code examination

Table 3.1: Compute properties of an amenable workload

Property	**Desired Value**	**Reason**	**Tool**
Large data footprint	1 TB+ input data size	The data footprint needs to be larger than standard RAM sizes to ensure most of the data reside on the disk	free or htop
Partitionable Data Structures	NA	Partitionable data can be operated on in parallel	Code Examination
Streaming data access	L3 MPKI of 5+, 20GB/s+ memory bandwidth	Contiguous data greater in size than cache	Code Examination + Perf
Data structure ownership	Single writer, multiple readers	To reduce the thread wait time to access shared data	Code Examination
Sparsity	Flat, single-level (not maps of maps of maps), deep-copy-friendly	The data will be serialized and deserialized between memory and disk	Code Examination
Complexity	Common+simple (matrices, arrays, maps, trees)	The data will be serialized and deserialized between memory and disk	Code Examination

Synchronization	Coarse grain, $<$ 1 sync per ms	To ensure sync overhead due to accessing IO device is < 1 %	Code Examination + Perf
Data transfer	$\leq$ 2 data transfers per data item between host and accelerator	To ensure bandwidth of kernel offloading is not a bottleneck	Valgrind
No Dynamic objects	NA	Programming for storage needs data size to be explicit, so the size should be static and determined at compile time.	Code Examination

Table 3.2: Data Structure properties of an amenable workload

Part II

Evaluating Workloads

Chapter 4

MarginPhase

4.1 Introduction

Current genotyping approaches for single nucleotide variations (SNVs) rely on relatively accurate short reads. Currently, new long-read technology is becoming more prevalent with the advent of companies such as Oxford Nanopore and Pacific Biosciences. These reads come with significant (about 15%) sequencing error [22], making them unsuited for current genotyping algorithms. However, long reads make a longer strand of genome unambiguously mappable and provide more linkage information between neighboring variants. The spatial locality information for these different genes is known as the haplotype.

One algorithm created for both genotyping and haplotyping (known together as diplotyping) is presented by Ebler et al. [13]. They consider bi-partitions of the sequencing reads, corresponding to the two haplotypes. They formalize the computational problem in terms of a Hidden Markov Model (HMM) and compute posterior genotype probabilities using the forward-backward algorithm. Genotype predictions

are then made, picking the most likely genotype at each site. Ebler et al. implement their algorithm in a C program referred to as MarginPhase.

MarginPhase, however, takes 9 hours to run on the smallest single human chromosome with an Intel Xeon, with an input alignment file of size 1.8 GB. Larger chromosomes can easily be 5x the size, providing a runtime of about 45 hours. This long run time presents an opportunity to accelerate the algorithm and substantially decrease the overall time for the genomics pipeline. In this thesis, the features of the MarginPhase workload are characterized to gain insight into bottlenecks and its suitability for SmartSSD acceleration.

MarginPhase is a C implementation of the algorithm described by Ebler et al. [13]. The input to the program is BAM files [7], which provide the aligned reads and can be up to 9 GB in size. MarginPhase returns the most likely genotype and haplotype.

4.2 Algorithm

Since a genome has two of each chromosome, to determine the haplotype, each read must not only be aligned but also assigned to the correct chromosome. Consider reads belonging to the first chromosome as haplotype1 (H1) and the second as haplotype1 H2. Therefore the problem to solve is: assign reads from Alignment Matrix (M) to either H1 or H2, maximizing the probability of H1 and H2 given M.

The problem is formalized with a Hidden Markov Model applied to a graph representation of the bipartition of M. The details of the algorithm are out of scope for this thesis. However I bring attention to the most time-consuming step of the program: computing the HMM from the alignment matrix. The algorithm is shown

in Figure 4.1. Subgraphs of the HMM are computed and recursively merged to form the full HMM graph.

Algorithm 1

procedure COMPUTEPRUNEDHMM($\mathbf{M}$)
  if maxCov $\geq t$ **then**
    Divide $\mathbf{M}$ in half to create two matrices, $\mathbf{M_1}$ and $\mathbf{M_2}$, such that $\mathbf{M_1}$ is the first $\frac{n}{2}$ rows of $\mathbf{M}$ and $\mathbf{M_2}$ is the remaining rows of $\mathbf{M}$.
    $\mathbf{HMM_1} \leftarrow$ computePrunedHMM($\mathbf{M_1}$)
    $\mathbf{HMM_2} \leftarrow$ computePrunedHMM($\mathbf{M_2}$)
    $\mathbf{HMM} \leftarrow$ mergeHMMs($\mathbf{HMM_1}, \mathbf{HMM_2}$)
  else
    Let **HMM** be the read partitioning HMM for $\mathbf{M}$.
  return subgraph of **HMM** including visited states and transitions each with posterior probability of being visited $\geq v$, and which are on a path from the start to end nodes.

Figure 4.1: The main algorithm implemented in MarginPhase. Note the recursive calls to ComputePrunnedHMM

4.3 Why is it suitable for acceleration?

The primary requirement of an amenable workload for this architecture is that the application has a large data footprint and a computationally expensive kernel that is IO-bound. MarginPhase has a large data footprint, and the application partitions the data for processing. It also has a relatively small computation kernel that takes up a significant amount of runtime.

Knowing the algorithm and various workload properties, MarginPhase was a good

candidate for future analysis. Section 4.5 details the profiling experiments performed and properties extracted from them. These properties will help determine if SmartSSD can accelerate the workload. If the application can be accelerated, then proceed to porting the kernel to SmartSSD compute hardware.

4.4 Workload Properties

4.4.1 Parallelizability

In the current implementation of MarginPhase, a majority of the time is spent in the merge step (see Figure 4.2). By analyzing the code and the algorithm manually, I found that merge operations can be parallelized. This is possible because the data is partitionable. Due to the recursive nature of the current implementation, the merge step needs to be modified to an iterative version to allow parallelization

4.4.2 Partionablity

Partitionability is defined as the ability of the memory accesses of the workload to be separated with few transfers between them, a concept similar to the coarse-grained spatial locality. The kernel that is under examination is *mergeTilingPaths*. The tiling path refers to the pruned HMM mentioned in the algorithm described in Figure 4.1. Upon profiling these data structures, I found them to be a flat data structure that only has a single writer. Moreover, the tiling paths are extensive in size, they don't fit in the cache or the main memory, so there are significant cycles spent on just data movement.

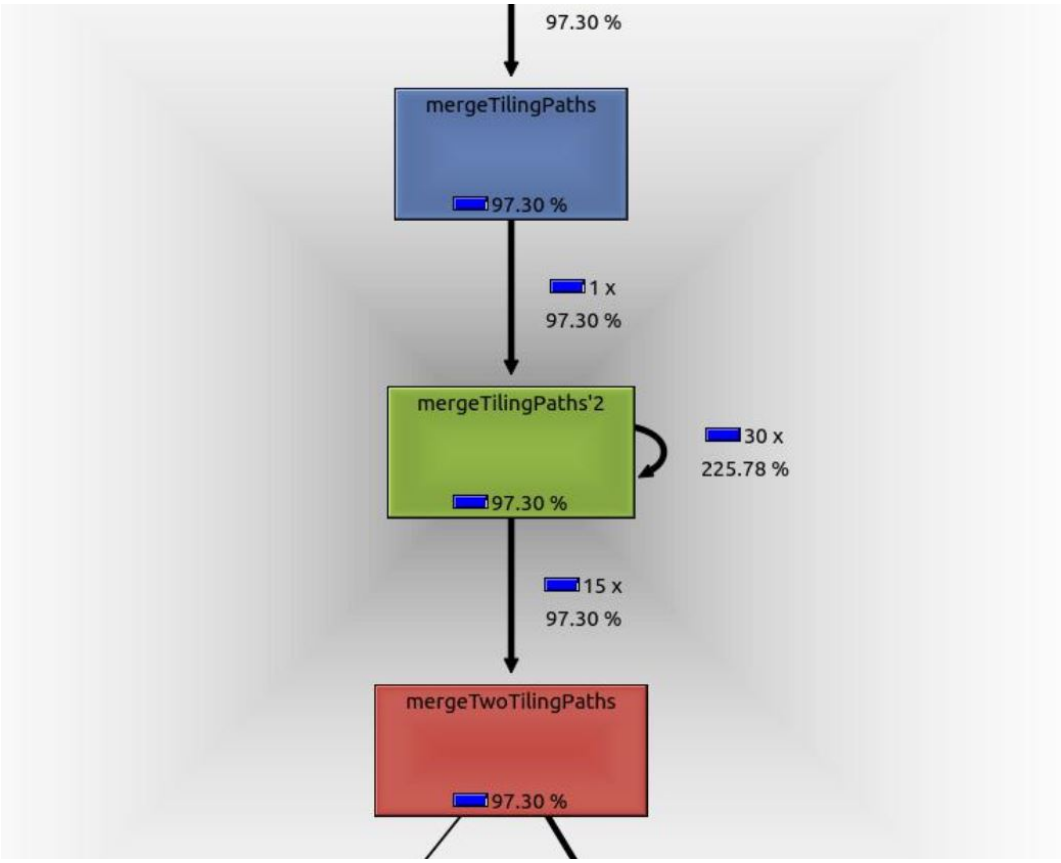

Figure 4.2: A portion of the call graph for MarginPhase generated from Callgrind, showing recursion in the merging step of the algorithm. The merge needs to be iterative to allow partitionability and parallelism.

4.4.3 High Data Use and Low Reuse

MarginPhase must-read in the entire input dataset, then in each step as it merges subHMMs, it reads in the sub-HMMs and writes out the new merged HMM. This processes large amount of data with very little reuse (low temporal locality).

The program is bound to a single core, and perf top-down is used with an interval of 1K to understand how cycles are used during execution. The results are given in Table 4.1.

Even in a single-threaded sequential implementation, 27% of cycles are spent waiting for memory or computation. Upon exploring a little deeper, it can be seen that

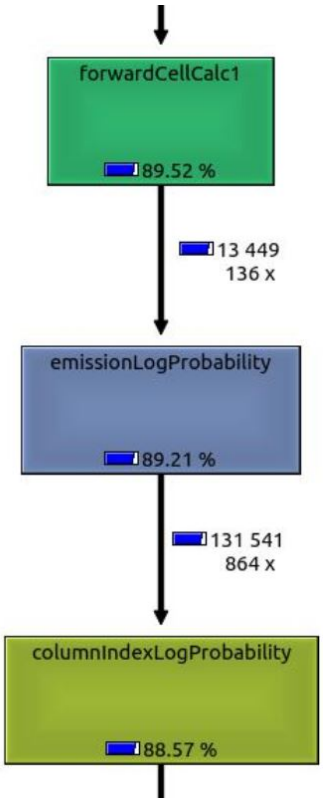

Figure 4.3: Call graph of a portion of MarginPhase that consumes the majority of runtime. These functions memory intensive and perform simple computations. This is a later portion of the same call graph as in Figure 4.2.

Retiring	Bad Speculation	Frontend Bound	Backend Bound
44.4%	5.8%	22.5%	27.3%

Table 4.1: Perf Topdown

the cache miss rate is unacceptably high at 64%. Caches are being heavily underutilized and wasting significant energy. A compute unit close to data, e.g., inside the same enclosure as the SSD, avoid the unnecessary overhead of caches, bringing improvement in both performance and energy.

According to Table 4.1, a moderate number of frontend stalls are also observed possibly caused by complex instructions or branch mispredictions. I leave the detailed

analysis to future work and focus on the larger problem of backend stalls.

Miss Rate	Miss Count	Reference Count
64.4%	1,633,498,853	2.536,992,733

Table 4.2: Cache Accesses

4.5 Methodology

I profiled the code to find the hot spots in the code using Valgrind, more specifically the Callgrind tool, and generated a graph from the result. Figure 4.2 and Figure 4.3 are snippets from Callgrind's graph. Displaying the full graph here would take up much space and would not have provided much more information. As can be seen from Figure 4.2, mergeTilingPaths is a function where the majority of the time is spent. On inspection of the code and the inputs to this function, I determined that if the recursive calls are modified to a loop, the data will become partitionable. This is because the merge operation of 2 tiling paths (MergeTwoTilingPaths) is independent of other tiling paths.

However, as this is a recursive call, the function calls at the top-level hold the reference to original data in the call stack, adding data dependency. After transforming this function from recursive to iterative, apply merge operations on data items that are independent. Now, it is possible to parallelize the procedure and, in turn, offload it to a co-processor (SmartSSD). It is preferable to offload the kernel to a SmartSSD instead of a GPU as offloading to GPUs involves large data movements, which are precisely the problem that this architecture addresses. The snippet shown in Figure 4.3 is a little lower in the call-stack of the mergeTwoTilingPath function. These

functions are minimal in size (15 lines of code). However, they account for a significant portion of the run time. This is because these functions are called many times. To give an idea of this, for our testing, I used a tiny subsection of the chromosomes, and marginPhase takes less than 2 minutes to complete the run. Even in this run, these functions are called millions of times. So instead of performing these operations in software, developing specialized hardware can improve efficiency. An issue in offloading all these functions is that because some of these functions have pointers as inputs, serializing/deserializing becomes an issue if the pointers point to very complex data structures (such as an array of pointers which point to Hidden Markov Models).

4.6 Results

Property Name	**Satisfied (Yes/No)**
Computationally expensive	Yes
IO Bound	Yes
Compute to space ratio	NA
Parallelism	No (A complete redesign can fix this)
Instructions per SLOC	Yes
Latency	NA
kernel compute complexity	No
High Loop Count in kernel	No (A complete redesign can fix this)
Recursion Free	No (A complete redesign can fix this)

Table 4.3: Compute properties of MarginPhase

Property	**Satisfied (Yes/No)**
Large data footprint	Yes
Partitionable	No
Streaming data access	No
Data structure ownership	Yes
Sparsity	No
Complexity	NA
Synchronization	NA
Data transfer	Yes
No Dynamic allocation	NA

Table 4.4: Data Structure properties of MarginPhase

4.7 Verdict

MarginPhase seemed like a good candidate for smartSSD acceleration and displayed properties that were ideal for SmartSSD. However, the current implementation is highly optimized to run on a general server, and it is not possible to extract a kernel to offload to the SSD. Thus, MarginPhase is not a candidate for SmartSSD acceleration.

The main issues with the MarginPhase code are:

1. Sparse data structures with pointer chasing.

2. Recursion

3. Tightly coupled code that makes it difficult to extract a function without

making significant changes to other parts of the code.

4. Shared data structures referenced by multiple threads

4.8 Learnings

Although MarginPhase didn't turn out to be a good candidate, genomics as a domain has properties that make it a promising avenue for more research. There exists a large number of algorithms that need to be re-architected for hardware-assisted acceleration. I will discuss one such algorithm in the next chapter.

MarginPhase was an excellent first application in this endeavor. This is mainly because the implementation had so many optimizations that make it run fast on a general linux machine and traditional architecture that makes it difficult for SmartSSD. This improved the understanding of the whole group and helped steer us in the right direction. This application enforced our belief in the importance of profiling and enhanced our knowledge of SmartSSD architecture.

During the process of profiling MarginPhase, I also developed some tools to aid in deciding if an application is a good fit. These tools significantly helped in selecting the next application for our work.

Chapter 5

Minimap2

5.1 Introduction

Minimap2 is a sequence alignment program that aligns DNA or mRNA sequences against a large reference database. It can be used for the following. [19]

- Mapping genomic reads to the human genome
- Finding overlaps between long reads with error rate up to 15%
- Splice-aware alignment of PacBio Iso-Seq or Nanopore cDNA or Direct RNA reads against a reference genome
- Aligning Illumina single- or paired-end reads
- Assembly-to-assembly alignment
- Full-genome alignment between two closely related species

For long-read sequences, minimap2 is orders of magnitude faster[19] than other long read mappers such as BLASR [12], BWA-MEM [18], NGMLR [20], and GMAP[26].

In this work I am using minimap2 to map genomic read sequences to reference genome strings using global alignment. Global Alignment [21] performs the alignment over the enitre read sequence.

5.2 Algorithm

The goal of this algorithm is to map the genome query sequences to a reference string. The output of the algorithm is **P**. P is a collection of *chains* and each chain is a mapping of a section of a query sequence to the reference string.

1. Initialize **P** as an empty set. Values are added to it in step 9.

2. Read **I** reference bases and index them into a hash table called a *reference index*.

3. Read **K** query sequences. These sequences form the whole human genome sequence. For each query sequence, do step 4 through 9.

4. Partition the query sequence into subsequences called *minimizers*.

5. For each minimizer in the query sequence, check if there is a partial match in the reference index. Filter out the minimizers that are not found. The remaining minimizers are called *seeds*. This step reduces the number of alignment operations required in step 6.

6. Perform an alignment operation on each seed to determine the location in the reference genome where the seed is most likely to map to.

7. The result of the alignment is an alignment string (refer bottom part of Figure

1.3) called a *chain* and the score for the alignment is called a *chaining score*.

8. Sort the chains according to the chaining score in decreasing order. This is to ensure that P contains chains that have maximum chaining scores as chains with lower scores will be dropped.

9. For each chain, if more than a certain threshold value of the section of reference genome that this chain maps to is covered by other chains in P then drop this chain. Else add it to P.

10. Return P.

5.3 Why is it suitable for acceleration?

As described in Section 1.2.1, algorithms used in alignment are substring matching algorithms. The Smith-Waterman algorithm is commonly used for this. As there are a large number of reads to be processed, running this algorithm over all the reads takes unacceptably long runtime. This issue is handled by splitting alignment into two steps: an approximate match to filter out some overlapping reads (this step uses some hash-based processing) and an exact match to align the filtered reads (by Smith-Waterman or some other dynamic programming algorithm).

Among the two steps, the second step is the performance bottleneck, so we will focus our efforts on optimizing the second step. As with the Smith-Waterman algorithm, minimap2 has a computationally expensive kernel, which is executed a large number of times.

The data footprint of this algorithm depends on the size of the reference and genome

read strings. And as these tend to be huge in size(upto 500GB), the footprint of this algorithm is large.

Due to these factors and other properties that will be described in detail in the next section, minimap2 is the right candidate for acceleration for SmartSSD acceleration.

5.4 Workload Properties

5.4.1 Partionablity

The input data for minimap2 a Fasta file which consists of a set of reads. The program preprocesses the reads to filter out the reads which cover the overlapping region in the reference genome. The remaining reads are then processed (aligned) independently. This shows that the minimap2 data exhibits data partionablity.

5.4.2 Parallelizability

As discussed above, the processing of each read is independent of other reads. For the alignment, the data structures that represent the reference genome and reads are read-only. Due to these reasons, the main alignment kernel is highly parallel.

5.4.3 Large Data Footprint

The inputs for minimap2 are genome reads and the reference genome. Both of these are several gigabytes in size. A massive data footprint is not an issue if the CPU uses only part of the data at a time. For minimap2 to take advantage of all the parallelization opportunities in a traditional architecture, all the genome strings need to be in cache at the same time.

5.4.4 A Computationally Expensive Kernel

Using perf and pyCparser, I determined that in minimap2, the function *ksw_extd2_sse41* takes up a significant portion of the runtime. This is the function that performs the alignment task. It takes the offset of the reference genome and the offset of the genome string to perform the alignment operation on. There are 495 effective lines of code in this function, which implies that the function is relatively simple and can be mapped to hardware.

5.4.5 Pointer Arithmetic/ Pointer Chasing

From static analysis of the source code, I found that the data structures used in minimap2 are arrays, and the structures used for the kernel do not have pointer chasing. The details of the data used in minimap2 kernel are described in details in Section 5.7

5.4.6 Dynamic Allocation

The data structure that the function ksw_extd2_sse41 works with is dynamically allocated, and the memory is reallocated for the alignment string during the processing. To address this issue, I allocated the maximum possible memory used by the string statically, and it didn't have any problems, nor did it have any impact on the performance.

5.5 Methodology

I profiled the code to find the hot spots in the code using Valgrind, more specifically the Callgrind tool, and generated a graph from the result. Figure 5.1 is the snippet from Callgrind's graph. Displaying the full graph here would take up much space and probably would not have added any more useful information. As can be seen from Figure 5.1, ksw_extd2_sse41 contributes the most to runtime as compared to other functions. The higher-level functions are not contributing to the runtime, so for this research, I focus on offloading this function.

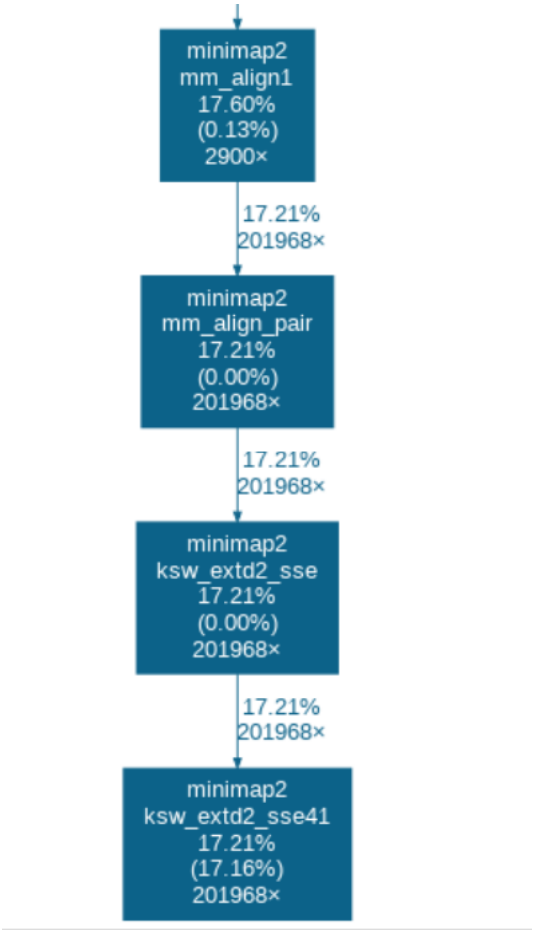

Figure 5.1: Snippet of Minimap2 algorithm call graph

On inspection of the code and the inputs for this function, I found that the arguments

to the function and the returned values (genome read, offset in the reference genome string, scoring matrix, and initially empty alignment string) are serializable with minimal effort. This enables easy movement of data to and from the disk. The operations that this function performs are easy to map to hardware. The only issue with the implementation is memory allocation. The memory for the data members is dynamic and changes at run time. Programming for the storage layer requires that the space needed is static. To overcome this, I modified the code so that the data members take up the maximum required space statically. This leads to an increase in the total data footprint of the program, but as this is to offload to the disk, the amount of storage used is not a concern. These modifications in the code do not affect the overall performance of the code. However, these modifications make the kernel amenable for offload.

Now, as per the APIs described in Section 2.2, the data needed for the kernel is sent to the SmartSSD as soon as the data structure for them is populated. This step is an asynchronous operation and needs to be started before the execution fans out to multiple threads so each thread can perform the alignment as soon as the data is available on the disk.

5.6 Results

Property Name	**Satisfied (Yes/No)**
Computationally expensive kernel	Yes
IO Bound	Yes
Compute to space ratio	Unknown
Parallelism	Yes
Instructions per SLOC	Yes
Latency	NA
kernel compute complexity	Yes
High Loop Count in kernel	Yes
Recursion Free	Yes

Table 5.1: Compute properties of minimap2

Property	**Satisfied (Yes/No)**
Large data footprint	Yes
Partitionable	Yes
Streaming data access	Yes
Data structure ownership	Yes
Sparsity	Yes
Complexity	Yes
Synchronization	Unknown
Data transfer	Yes
No Dynamic allocation	No (but can be modified)

Table 5.2: Data Structure properties of minimap2

5.7 Shared data analysis

In this section, I will list the parameters of the kernel and various details that aid in porting the application.

Name	**Description**	**type**	**Size**	**Mode (On Accelerator)**	**Transfer**
query	query string	array of uint8_t	280 GB	Read Only	Host to Accelerator before the first kernel invocation
target	reference string	array of uint8_t	500 GB	Read Only	
m	Scoring Matrix	array of int8_t	5 B	Read Only	
e	alignment parameter	int8_t	1 B	Read Only	Host to Accelerator for each Kernel Invocation
q	alignment parameter	int8_t	1 B	Read Only	
e2	alignment parameter	int8_t	1 B	Read Only	
q2	alignment parameter	int8_t	1 B	Read Only	
w	alignment parameter	int8_t	1 B	Read Only	
zdrop	overlap parameter	int	4 B	Read Only	
end_bonus	overlap parameter	int	4 B	Read Only	

cigar	Mapping of query to reference	array of uint8_t	32.53 KB	Read Write	Accelerator to Host after the last kernel invocation

Table 5.3: Shared data members

5.8 Verdict

Minimap2 is an application amenable for SmartSSD acceleration. As can be seen in Table 5.1 and Table 5.2, minimap2 does not satisfy all the properties, but it has enough to make it a good candidate for acceleration.

5.9 Learnings

Minimap2 has proved to be amenable to this architecture. The current implementation of minimap2 uses SIMD libraries to make operations faster. Writing Verilog code for this adds complexity to the offloading task. Offloading minimap2 has two issues that make the process difficult.

1. Minimap2 uses an in-built memory allocation module, which, as per my understanding, provides no benefit over standard malloc() and free() C commands. Moreover, as the data is moved to the storage layer, the same logic would also have to be implemented at storage to understand the new memory allocation

module.

2. Minimap2 built a complicated threading library that provides no benefit other than slightly easier thread spawning functionality but reduces the code readability and maintainability.

The first issue can be eliminated by replacing the special memory allocation and deallocation commands with standard C commands for the same. This did not hamper the performance. The second turned out not to be an issue for me as the offloaded kernel does not spawn new threads.

5.10 Acceleration Opportunities

1. Reduce data movement by storing the reference genome and reads on disk

   The kernel ksw_extd2_sse41 performs the string matching operations on genome reads, and reference strings based on the scoring matrix and creates an alignment string. Offloading this operation to the SmartSSD requires the strings to be present on the disk. So the data structures (arrays) used to store these strings can be deep copied to the disk.

2. Moving the parameters in the preprocessing phase

   Minimap2 is a multithreaded application, and it has a sequential phase (preprocessing) followed by a parallel alignment step for each read sequence. The data structures (arrays of strings) used to store these reads can be deep copied to the disk in the preprocessing phase of the execution. As the preprocessing is a single-threaded operation, making this transfer asynchronous provides even

more performance improvement. This requires that instead of passing pointers to the kernel function, offsets from the start be function parameters.

Part III

Learnings

Chapter 6

Methodology

With the experience of working with the two workloads mentioned above, one of which is a good candidate and the other, which did not turn out to be a good candidate, I have come up with a methodology that aids in identifying properties that make an application amenable to SmartSSD acceleration, identifying kernel that can be offloaded to SmartSSD, and doing the actual port of the application. This methodology is described below.

Given an application, use tools such as perf and Valgrind to analyze the code base and find out if there is a kernel that takes up a significant runtime as described in earlier chapters. Note that this kernel is ideally near the leaf in the call graph. From the list of candidate kernels functions, filter out candidates for which the high computation time is not due to a high loop count. The loop may not be in the candidate function but a higher-level function, which calls this function repeatedly.

From the remaining candidates, if a function is called from different places, it is likely not a good candidate. Furthermore, if the function is part of a recursive call or if it

contains a recursive call, then either avoid this function or modify the source code to make it an iterative call (as recursion is complicated to implement in hardware). In my experience, I only found one or two candidate functions, and they were part of the same call stack. In this case, it is preferable to offload the higher-level function as more work offloaded to the accelerator means more time the host CPU is free to do some other work.

After a function is finalized, determine the data that is to be moved to and from the host and the target. This data is generally just the data structures local to the function and global variables if any. The requirements for these variables is that they not be dynamically allocated (if they are, then modify the source code to allocate them statically) and the data structures are flat. In case it is not possible to flatten the data structures, then the data structures need to be serializable [9] by other means, such as writing a custom serialization library. Duplicate the global constants within the accelerator. At the time of writing this thesis, I did not have access to the real hardware. In the absence of hardware, a simulator can be developed. The API that the simulator supports needs to align with the API described in section 2.2. Extract the function body along with any global dependent variables and copy the code in a separate binary than the workload.

On the original codebase, develop a strategy to serialize/deserialize the data structures of the application. This is needed as the accelerator will behave as a different process, and hence the virtual memory of the application will not be shared with the accelerator. The best way to emulate this process of data movement is to serialize the data at the host and deserialize at the target. Now, as the original function body is empty, fill with serialization library calls to data that needs to be moved

from host to target, and a function call to invoke the kernel function is a different binary and then deserialize the results from the function.

At this point, the kernel can be offloaded to the simulator. Now benchmark and perform code improvements to reduce performance bottlenecks.

Chapter 7

Related Work

Amazon Aurora [23] is an example of a system that disaggregates storage from the compute layer. The compute layer writes data to local storage, and the required storage replication, garbage collection, and indexing are performed asynchronously at the storage layer. However, computational storage offloads the actual computation to the storage device to accelerate the workload.

There has been a recent focus in the industry on performing computations at the storage layer. Samsung has developed SmartSSD [6]. The focus of work done in Samsung is finance, data science, and data processing. Another startup that is working on SmartSSDs is Bittware(a Molex company). Its area of focus is database acceleration and big data processing. Neither of these companies is focussed on the genomic domain.

Skyhook[17] is another implementation of SmartSSD that is developed using Ceph[24]. But it differs from the architecture described here, and other companies mentioned previously in that Skyhook is an object store. The benefit of the object store is

that the module that is responsible for translating object details to the location at storage can be extended to create an interceptor discussed in Section 2.1.

Chapter 8

Conclusion

In this thesis, I explored the suitability of Genomics as a domain for SmartSSD acceleration by examining two applications – marginPhase and minimap2. The significant properties an amenable application should exhibit are:

- A large data footprint
- IO bound
- A simple but computationally expensive kernel (high runtime comes from a high loop count)
- Flat data structures in the kernel

The primary issue that rendered MarginPhase unsuitable for SmartSSD acceleration is the data structures used. The shared data structures are sparse with a lot of pointer chasing. On the other hand, minimap2 has a function that takes up a significant runtime that has a high loop count. The parameters to these functions are serializable so they can be transferred to and from the storage layer. So minimap2

is a good candidate for SmartSSD acceleration.

I also developed a strategy described in Chapter 6 that aids profiling a workload to decide suitability for SmartSSD acceleration, determine the kernel to be offloaded to SmartSSD, and outlines steps to port the application for this architecture.

Chapter 9

Future Work

The next steps in this research area are to develop a working simulator that supports the APIs as described in Section 2.2 and designs a hardware implementation of the proposed architecture. A simulator can be developed and used to benchmark and identify the potential benefits of an application before rewriting it for this architecture.

The methodology and tools developed as part of this research project can be used to analyze any other domain that deals with processing large amounts of data. Some of the other fields that can be explored are Finance, Data Management, Data Security.

Bibliography

[1] Alignment matrix. https://en.wikipedia.org/wiki/Smith-Waterman_algorithm.

[2] Bittware architecture. https://www.bittware.com/fpga/storage/.

[3] Disk allocation strategy. http://www2.cs.uregina.ca/~hamilton/courses/330/notes/allocate/allocate.html.

[4] Logical block addressing. https://en.wikipedia.org/wiki/Logical_block_addressing.

[5] Perf repo. https://github.com/torvalds/linux/tree/master/tools/perf.

[6] Samsung smartssd. https://samsungsemiconductor-us.com/smartssd/,.

[7] What are bam files? https://software.broadinstitute.org/software/igv/BAM.

[8] What is computational storage? https://www.snia.org/education/what-is-computational-storage.

[9] What is serialization. https://isocpp.org/wiki/faq/serialization.

[10] Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195 – 197, 1981.

[11] A. D. Baxevanis. Transforming medicine: Genomics, bioinformatics, and human health. In *2007 IEEE 7th International Symposium on BioInformatics and BioEngineering*, pages 1449–1449, 2007.

[12] Mark J Chaisson and Glenn Tesler. Mapping single molecule sequencing reads using basic local alignment with successive refinement (blasr): application and theory. *BMC bioinformatics*, 13(1):238, 2012.

[13] Jana Ebler, Marina Haukness, Trevor Pesout, Tobias Marschall, and Benedict Paten. Haplotype-aware genotyping from noisy long reads. *bioRxiv*, 2018.

[14] Gustavo Glusman, Hannah C. Cox, and Jared C. Roach. Whole-genome haplotyping approaches and genomic medicine. *Genome Medicine*, 6(9):73, Sep 2014.

[15] Alan E Guttmacher, Mary E Porteous, and Joseph D McInerney. Educating health-care professionals about genetics and genomics. *Nature Reviews Genetics*, 8(2):151–157, 2007.

[16] A. Huffman and D. Juenemann. The nonvolatile memory transformation of client storage. *Computer*, 46(8):38–44, 2013.

[17] Jeff LeFevre and Noah Watkins. Skyhook: Programmable storage for databases. Boston, MA, February 2019. USENIX Association.

[18] Heng Li. Aligning sequence reads, clone sequences and assembly contigs with bwa-mem. *arXiv preprint arXiv:1303.3997*, 2013.

[19] Heng Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34(18):3094–3100, 05 2018.

[20] Fritz J Sedlazeck, Philipp Rescheneder, Moritz Smolka, Han Fang, Maria Nattestad, Arndt Von Haeseler, and Michael C Schatz. Accurate detection of complex structural variations using single-molecule sequencing. *Nature methods*, 15(6):461–468, 2018.

[21] Temple F Smith, Michael S Waterman, et al. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

[22] Yatish Turakhia, Gill Bejerano, and William J. Dally. Darwin: A genomics co-processor provides up to 15,000x acceleration on long read assembly. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 199–213, New York, NY, USA, 2018. ACM.

[23] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 1041–1052, New York, NY, USA, 2017. Association for Computing Machinery.

[24] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320, 2006.

[25] Joachim Wolff, Bérénice Batut, and Helena Rasche. Mapping (galaxy training materials), 03 2020.

[26] Thomas D Wu and Colin K Watanabe. Gmap: a genomic mapping and alignment program for mrna and est sequences. *Bioinformatics*, 21(9):1859–1875, 2005.