GATEKEEPER-GPU: ACCELERATED PRE-ALIGNMENT FILTERING IN SHORT READ MAPPING

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
THE DEGREE OF
MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

By
Zülal Bingöl
August 2020

GateKeeper-GPU: Accelerated Pre-Alignment Filtering in Short Read Mapping
By Zülal Bingöl
August 2020

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Can Alkan(Advisor)

Özcan Öztürk(Co-Advisor)

Eray Tüzün

Tolga Can

Approved for the Graduate School of Engineering and Science:

Ezhan Karaşan
Director of the Graduate School

ABSTRACT

GATEKEEPER-GPU: ACCELERATED PRE-ALIGNMENT FILTERING IN SHORT READ MAPPING

Zülal Bingöl
M.S. in Computer Engineering
Advisor: Can Alkan
Co-Advisor: Özcan Öztürk
August 2020

Recent advances in high throughput sequencing (HTS) facilitate fast production of short DNA fragments (reads) in numerous amounts. Although the production is becoming inexpensive everyday, processing the present data for sequence alignment as a whole procedure is still computationally expensive. As the last step of alignment, the candidate locations of short reads on the reference genome are verified in accordance with their difference from the corresponding reference segment with the least possible error. In this sense, comparison of reads and reference segments requires approximate string matching techniques which traditionally inherit dynamic programming algorithms. Performing dynamic programming for each of the read and reference segment pair makes alignment, a computationally-costly stage for mapping process. So, accelerating this stage is expected to improve alignment performance in terms execution time. Here, we propose, GateKeeper-GPU, a fast pre-alignment filter to be performed before verification to get rid of the sequence pairs, which exceed a predefined error threshold, for reducing the computational load on the dynamic programming. We choose GateKeeper as the filtration algorithm, we improve and implement it on a GPGPU platform with CUDA framework to obtain benefit from performing compute-intensive work with highly parallel and independent millions of threads for boosting performance. GateKeeper-GPU can accelerate verification stage by up to 2.9× and provide up to 1.4× speedup for overall read alignment procedure when integrated with mrFAST, while producing up to 52× less number of false accept pairs than original GateKeeper work.

Keywords: read mapping, pre-alignment filtering, general purpose graphical processing unit (GPGPU), alignment acceleration.

ÖZET

GATEKEEPER-GPU: KISA OKUMA HARİTALAMASI İÇİN HIZLANDIRILMIŞ ÖN-HİZALAMA FİLTRESİ

Zülal Bingöl
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: Can Alkan
İkinci Tez Danışmanı: Özcan Öztürk
Ağustos 2020

Yüksek verimli dizilemede (HTS) son zamanlarda elde edilen gelişmeler çok sayıda DNA parçasının (okumanın) hızlı üretimini kolaylaştırmaktadır. Her geçen gün, bu bağlamda veri üretimi daha az masraflı hale gelmesine rağmen mevcut veriyi bir bütün olarak hizalama programında işlemek hala bilgisayımsal açıdan pahalıdır. Hizalamanın son evresinde, kısa okumaların referans genomu üzerindeki aday pozisyonları, ilgili referans kısmıyla aralarındaki farka bağlı olarak hata payını en aza indirecek şekilde doğrulanmaktadır. Bu bakımdan, okumalar ve referans genomu kısımları arasında karşılaştırma yapmak, geleneksel olarak devirgen programlama algoritmaları içeren yaklaşık karakter dizgisi eşleştirme tekniklerini gerektirmektedir. Herbir okuma ve referans kısmı için devirgen programlama uygulamak, hizalamayı, haritalama işleminin bilgisayımsal olarak pahalı aşaması haline getirmektedir. Bu yüzden, bu aşamayı hızlandırmanın bütün olarak hizalama performansını işletim zamanı açısından iyileştirmesi beklenmektedir. Bu tezde, doğrulama öncesi, devirgen programlama üzerindeki bilgisayımsal yükü azaltmak için belirli bir hata eşiğinin üzerinde olan dizi çiftlerini eleyen bir ön-hizalama filtresi olan GateKeeper-GPU'yu sunuyoruz. Filtreleme için GateKeeper algoritmasını seçiyoruz. GateKeeper'ı geliştiriyoruz ve onu, bilgisayımsal açıdan ağır işi, yüksek oranda paralel milyonlarca izlekle yapmaktan kaynaklanan fayda ile performansı artırmak için CUDA çatısı ile GPGPU (genel kullanım grafik işleme ünitesi) platformuna adapte ediyoruz. GateKeeper-GPU, mrFAST ile entegre edildiğinde doğrulama aşamasına 2.9 kata kadar ve hizalama işleminin bütününe 1.4 kata kadar hızlandırma sağlarken asıl GateKeeper'a göre 52 kata kadar daha az yanlış kabul edilen dizi çifti üretmektedir.

Anahtar sözcükler: Okuma Haritalama, Ön-Hizalama Filtresi, Genel Kullanım Grafik İşleme Ünitesi, Hizalama Hızlandırması.

Acknowledgement

I would like to express my sincere gratitude to my advisor, Asst. Prof. Can Alkan, for giving me the opportunity to combine my major with my childhood passion, genomics, to work in bioinformatics. His patience and generous support enabled me to have a strong start at the graduate study and complete this thesis. I am also grateful to my co-advisor, Prof. Özcan Öztürk, for his valuable guidance towards excellence in research. I would like to thank my other committee members, Asst. Prof. Eray Tüzün and Prof. Tolga Can, for spending time to review this thesis. Also, I would like to thank EMBO grant (IG 2521) and Bilkent University for supporting this work.

I would like to extend my special thanks to Dr. Mohammed Alser for his incredible mentorship and guidance throughout this study. With his highly practical and thought-provoking pieces of advice and support, this work became much easier to accomplish. Thanks to Fatma Kahveci for sharing her valuable experiences in both academic and personal life for the guidance. Thanks to Balanur İçen, for being a wonderful coffee mate, with whom I have never felt awkward for drinking too much coffee while working. Thanks to Halil İbrahim Özercan and Ezgi Ebren for both exchanging informative conversations in bioinformatics and sharing enjoyable time for escaping stress. I am grateful to other AlkanLab members and alumni, especially Fatih Karaoğlanoğlu, Can Fırtına and Alim Gökkaya, for their fellowship. I would like to thank Safari Research Group members for fruitful discussions and suggestions during regular meetings, with special thanks to Damla Şenol Çalı for being an amazing long-distance colleague and friend. I am also thankful to old and current EA-507 office mates, even though we could not spend our last semester together due to pandemic.

I would like to express heartfelt thanks to my friends Pınar Göktepe, Eda Demir, Edanur Atlı and Hatice Selcen Dumlu for their continuous encouragement and support in every aspect of life; and many others in Ankara, İstanbul and Erzurum for their valuable friendship throughout graduate student life and

beyond.

Last but not least, I would like express my deepest gratitude to my parents, Halim and Selma Bingöl, and my brother Selim Bingöl, for believing in me at all times with endless support and love. They give me continuous courage to persevere in every work I start. I also would like to thank to my grandmother for sending delicious meals to Ankara; to my uncle and aunt for making Ankara feel like home.

In Memory of Thesis in Times of Corona
August 2020, Ankara

Contents

List of Figures

List of Tables

Chapter 1

Introduction

High-throughput sequencing (HTS) has created a new era for biological research based on genomic data analysis. Application of HTS is now the standard for deep and detailed studies in many areas such as forensics and medicine. As the vision of predictive, preventive, personalized and participatory (P4) medicine [2] rapidly becomes more prevalent, sequencing technologies stand out to provide vital information as a base for further investigation. Sequencing data generated by HTS constitutes a reliable source for biological research pipelines.

With the recent advances in sequencing technology, HTS produces sequencing data in shorter amount of time with a gradually decreasing cost [3]. As a result of massively parallel sequencing, tremendous amount of data can now be generated in a single run [4, 5], making genomics one of the largest sources of big data today and in the future [6]. On the other hand, the presence of enormous data creates a computational challenge for analysis in terms of both run time and memory, putting an emphasis on the importance of developing efficient methods and solutions.

First main step in the analyses on genome data is either *de novo assembly* or *read mapping*, to combine the sequenced data as streaks of DNA (i.e., reads) to form the whole genome for different purposes leading to further study. In

de novo genome assembly, the reads are compared against each other without a template genome to find the overlap regions which are connected later to build larger fragments (i.e., contigs) [7]. *De novo* assembly is necessary when a species is being sequenced for the first time, therefore the reference genome is not readily available. For *read mapping*, the reference genome of the species already exists and stands as a template [8]. Newly sequenced fragments of a genome are compared to it for understanding the genetic material of the individual sample. Read mapping consists of two main stages which are *mapping* and *verification* (i.e., alignment). By read mapping, the possible locations of reads on the reference genome are found based on the string similarity between the reads and corresponding reference segments. Reads generally have more than one possible mapping position on reference genome due to three main reasons. First, the proportion of reference DNA sequence length over read length is very high, such that human genome is 3.2 gigabase-pairs long [4] whereas short reads vary between 64 base-pairs to 300 base-pairs in length. Second, each DNA is unique in its entirety for different individual organisms but has common repetitive regions and segmental duplications ($> 5\%$ of its content) [4] within itself. Third, reads have differences at base level which either signal short mutations (i.e., *insertion / deletion (indel) and substitution*) or errors introduced by sequencing technique [9]. Although the sequencing error rate is very low for short reads, for instance Illumina data has an error rate of lower than 0.4% [10], unfortunately sequenced data is not completely error-free yet. Bearing these factors in mind, one of the approaches to find the optimal location which fits the read on reference genome is *seed-and-extend* method. This approach is rooted in the idea that two similar sequences share exactly or approximately matching substrings (i.e., kmers) [11, 12]. Consequently, once the matching locations of the shorter substrings (i.e., seeds, kmers), which are extracted from the read, are found on the reference genome, the seeds can then be *extended* to form the candidate reference segment (Figure 1.1). Since kmers are shorter than reads, the seeds may map to various locations.

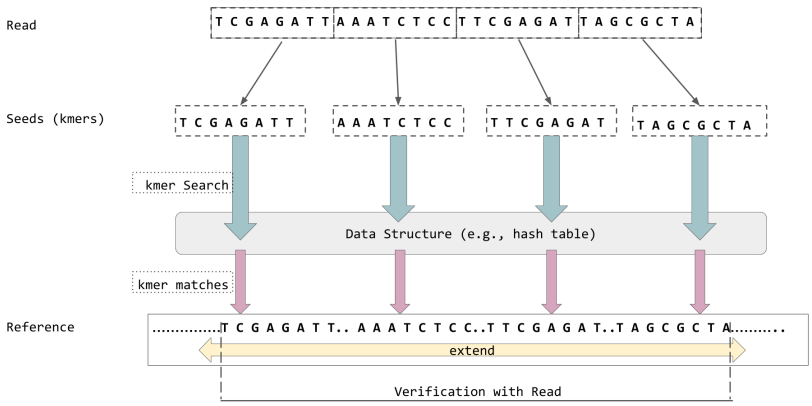


Figure 1.1: Seed-and-Extend Paradigm

1.1 Research Problem

The presence of possible sequencing errors and base mutations decrease the capability of mapping with exact string matching. Therefore, verifying the actual locations requires *approximate (i.e., fuzzy) string matching* techniques on strings with a predefined error threshold. Once kmer search is done and all of the candidate locations are found, read's most accurate location on genome among all its candidates is identified by verification. Current common practices generally lean towards using dynamic programming algorithms such as Needleman-Wunsch [13], Smith-Waterman [14] and Levenstein distance [15] to confirm that the distance between extended sequence segment from reference and the read is within the limits of an error threshold. However, dynamic programming solutions are computationally expensive, requiring $O(mn)$ time and space where m is the read length and n is the reference segment length. Since mapping produces many candidate reference locations because of aforementioned reasons, performing dynamic programming algorithm for each and every pair becomes even more exhaustive of resources. Thus, choosing the best location that fits the read among all of its candidate reference segments at this stage is a computationally-costly problem.

1.2 Motivation

Due to compute-intensive nature of dynamic programming solutions, verification step creates a bottleneck for the entire read alignment procedure. Therefore, efficient improvements targeting this stage are expected to enhance the entire process. The conspicuous approaches to address this problem would be either improving the algorithms in terms of time and space complexities or reducing the workload on verification so that the pipeline visits verification as rarely as possible. In this work, we adopt the second approach and aim to decrease the number of candidate pairs to a feasible level. By achieving this goal, we intend to effectively eliminate the candidate locations that exceed a predefined error threshold before verification stage for better performance.

1.3 Thesis Statement and Contributions

Our thesis statement for this work is *verification stage of short read mapping can be accelerated with the help of a pre-alignment filter designed for GPU, by reducing the number of candidate mappings with fast filtration.* Following this statement, we propose GateKeeper-GPU, a general purpose graphical processing unit (GPGPU) pre-alignment filter for short read mapping to be performed prior to verification. GateKeeper-GPU's main filtering algorithm is built upon GateKeeper [1] and it introduces some modifications for better performance and accuracy. Since GateKeeper algorithm has high accuracy with low resource usage per one filtering operation, it is one of the first efficient pre-alignment filters that can be adapted to hardware platforms for massive parallel execution.

GateKeeper's original code base is developed for field programmable gate array (FPGA) devices. We opt for implementing it on general purpose GPUs with CUDA framework for several reasons. First, GPU has a large of number of specialized cores which have smaller caches when compared to CPU cores, thus its

cores require lower frequency. This makes GPU a powerful candidate for processing the work for which the throughput is crucial [16]. Since pre-alignment filtering requires high throughput in order to complete as many comparisons as possible before verification, GPU is well-suited for this work. Second, GPU is more widely preferred than FPGA because it is easier to install and use, therefore CPU-GPU combinations are a lot more common than CPU-FPGA combinations in heterogeneous systems . Third, once installed, GPU can be used in wide range of applications without extra configuration effort, which makes GPU more economically viable than FPGA. Fourth, GPU code is flexible in terms of programmability to adapt the code for a desirable change or prototyping. On the other hand, FPGA code base is usually target-specific, any change in the algorithm requires more effort to reflect on the code if even possible. Therefore, we aim to provide easy usage and support for wide range of platforms with our design.

Our main contributions with GateKeeper-GPU are:

- We improve GateKeeper algorithm and adapt it to GPU with CUDA framework to facilitate wider range of platform usage with GPGPUs.
- We integrate GateKeeper-GPU with mrFAST to show the its performance gain in a short read mapper with full workflow. This will hopefully serve as an example to embed GateKeeper-GPU to any short read mapping tool easily.
- We provide comprehensive analyses using two different device settings. We show that GateKeeper-GPU can accelerate verification stage by up to 2.9× and provide up to 1.4× speedup for overall read alignment procedure when integrated with mrFAST [17], in opposition to having no pre-alignment filter. Compared to original GateKeeper [1], GateKeeper-GPU produces up to 52× less number of false accepts in candidate mappings.

It should be emphasized that GateKeeper-GPU does not perform seeding and it is not a kmer filter. It relies on the candidate reference locations reported by the mapper and performs filtration only on the read and reference segments

pairs which are expected to match with mapper's high confidence. Further, GateKeeper-GPU does not calculate but *approximates* the edit distance between pairs for fast filtration. Exact edit distance calculation is performed by verification procedure and GateKeeper-GPU acts as an intermediate step in preparation to verification.

Chapter 2

Background

2.1 GateKeeper Filtering Algorithm

GateKeeper [1] is the first pre-alignment filter that was designed for acceleration on hardware systems. The underlying logic is inspired from Shifted Hamming Distance (SHD) [18]. Its simple and bitwise nature makes it applicable for hardware acceleration. GateKeeper [1] improves the core SHD algorithm and offers a hardware-friendly design in FPGA for better performance. Since the algorithm mainly consists of bitwise AND, XOR and shift operations, it is a good fit for FPGA's bitwise functional units in design with logic gates.

Figure 2.1 depicts an example of GateKeeper algorithm's workflow. GateKeeper [1] starts by encoding the read and its candidate reference segment in 2-bits (i.e., "A" = "00", "C" = "01", "G" = "10", "T" = "11") to prepare the bitwise representations of the strings. Then, it performs XOR operation between bitvectors of read and reference segment to prepare the Hamming mask for exact match detection (box 1 in Figure 2.1). Differentiating characters which depict mismatches are indicated as *'1'* on the resulting bitvector and matching characters are shown with *'0'*. Depending on the error threshold E being more than 0, approximate matching phase begins. With current error tolerance value e,

insertion and deletion bitmasks are produced in a loop of $e = 1,...,E$. In each iteration, first, read bitvector is shifted by e bits to right for deletion and to left for insertion tolerance. Then, the shifted bitvector undergoes XOR operation with reference segment bitvector to detect possible errors, so every iteration produces two masks: one for deletion and one for insertion (box 2 & 3 in Figure 2.1). The resultant bitvectors are represented with 'H' in Figure 2.1. Since XOR operation signifies the difference in 2-bits per encoded character comparison, every two bit is combined with bitwise OR to simplify the differences on individual bitvectors and reduce resource usage.

The final stage of approximate matching is an AND operation on all $2k+1$ masks (box 4 in Figure 2.1). Nevertheless, if one of the bitvectors has a 0-bit at a particular position, it dominates and AND will yield a 0-bit indicating a match even if all of the other bitvectors have 1-bit value, which signals a mismatch, at that position. To compensate for this issue, the bitvectors are *amended* before AND to turn short streaks of 0s into 1s considering that these 0s are useless and do not represent an informative part in Hamming masks. The amendment procedure is one of GateKeeper's major contributions over SHD. After the AND operation, the errors are counted by following a windowed approach with a look-up table (box 4 in Figure 2.1). The comparison pair is rejected if the number of $1s$ as errors in final bitvector exceeds the error threshold, and accepted if otherwise.

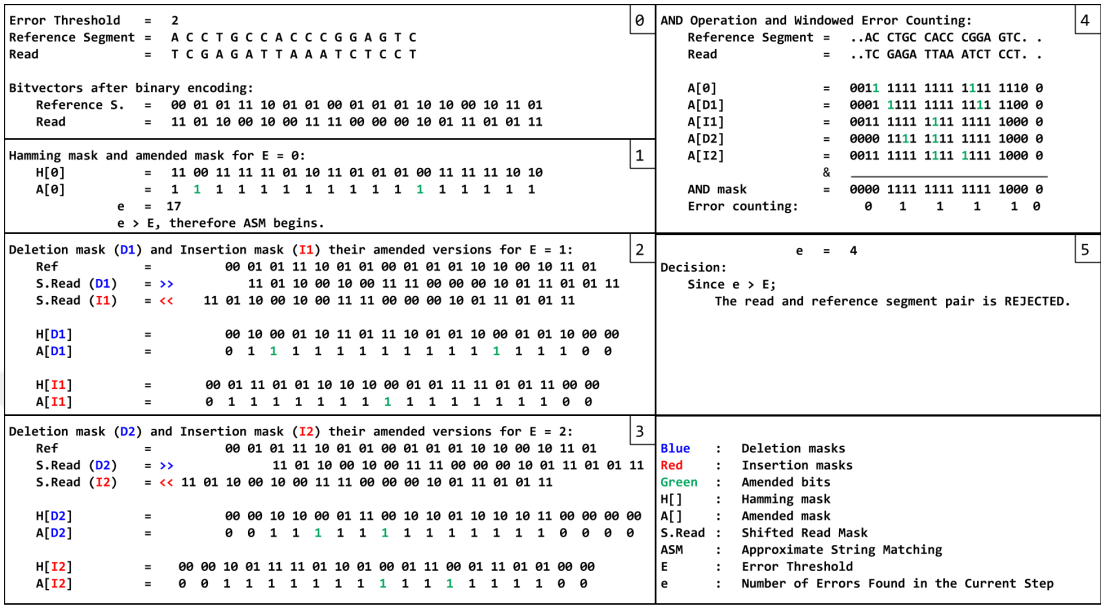


Figure 2.1: GateKeeper workflow for error threshold 2

2.2 Unified Memory Architecture

GPU provides massive amount of parallelism for acceleration of compute intensive work which otherwise takes longer time with CPU. The specialized cores of GPU process data physically residing on device, so utilizing GPU as an accelerator inevitably requires moving the data from CPU to GPU memory via PCI-e bus. Because this creates an extra task for GPU when compared to CPU-only execution, adopting an effective memory management and data movement strategy plays a critical role. GPU cannot have direct access to either CPU memory which is typically pageable or CPU page table [19] during execution. Thus, it needs to convert the data into a page-locked (i.e., pinned) format, then copy to device memory space upon data movement. There are two methods the CUDA driver offers to optimize this process. First method involves allocating CPU data in pinned format at the first place, so that GPU can skip page table management step and directly starts by copying data. Second method is using unified memory for creating a virtual space to which GPU and CPU have access with a single pointer [20].

Unified memory does not remove the necessity of data movement altogether, but maintains an automatic migration of data on demand providing data locality [20]. Conceptually, unified memory model fits GateKeeper-GPU in many aspects. For instance, reference genome's designated segments are requested only when the mapper signals a potential match for the specific read in seeding, which creates an on-demand access situation. Likewise, it is sufficient to copy a single read only once to GPU memory for its multiple candidate reference segments. In our experience with different memory allocation methods at the development stage of GateKeeper-GPU, we also achieved more efficient overall execution with unified memory than pinned memory format. Therefore, we adopted unified memory model in GateKeeper-GPU.

Along with unified memory abstraction, CUDA framework introduces *memory advises* and *asynchronous memory prefetching* features for an optimized data migration during execution. Data access patterns of an object, such as preferred location, can be specified by memory advises so that the processor favors for the optimal place of the data for migration decisions. In addition, asynchronous data prefetching initiates the data transfer to device before the data is being used for minimizing the overhead caused by page faults in runtime [21]. Prefetching is supported by Pascal and later architectures with CUDA 8 in GPU.

2.3 Related Work

Finding optimal solutions for approximate string matching problem has been on the focus of genome sequence studies due to the nature of present data and its tremendous size. One of tools built for this purpose, Edlib [22], is a CPU framework that calculates edit distance (also known as Levenstein Distance [15]) by utilizing Myers's bitvector algorithm [23] for exact pairwise sequence. Although Edlib is mainly curated for bioinformatics tools, it can work on any strings. Since Edlib finds the exact edit distance, we hold Eblib's global alignment results as the ground truth for our accuracy analysis.

Verification stage of read mapping tools determines the exact similarity between the given two sequences as read and candidate reference segment, therefore, the efforts on pre-alignment filtering prioritize quickly eliminating pairs with apparent dissimilarity. Shifted Hamming Distance [18] (SHD) filters out highly erroneous sequence pairs with a bit-parallel and SIMD-parallel algorithm. It prepares the Hamming Masks of strings; applies bitwise shifting, AND, and XOR operations for error toleration on the masks. SHD's algorithm also sets up the foundation for GateKeeper [1]. MAGNET [24] addresses some of SHD's shortcomings, such as not considering leading and trailing zeros in bitmasks, and consecutive bit counting for edit calculation, which are the main sources of high false accept rate. With a new filtering strategy, it improves the filtering accuracy up to two orders of magnitude. Shouji [25], on the other hand, takes a different approach on pre-alignment filtering. It starts by preparing a neighborhood map for identifying the common substrings between two sequences within an error threshold. Then, it finds non-overlapping common subsequences with sliding window approach and decides on accepting or rejecting the pair according to the length of common subsequences. With an FPGA design, Shouji can reduce sequence alignment duration up to 18.8× with two orders of magnitude higher accuracy compared to GateKeeper and SHD. Lastly, SneakySnake [26] solves approximate string matching by converting it to single net routing problem [27]. This enables SneakySnake provide acceleration with all hardware architectures and without hardware support. SneakySnake also improves accuracy of SHD, GateKeeper and Shouji.

Chapter 3

Methods

GateKeeper-GPU is designed to require the least amount of user interaction and effort. Because of the disadvantages on dynamic kernel memory allocation of CUDA, which will be discussed in detail in Section 3.1, read length and error threshold should be specified at compile time. GateKeeper-GPU adjusts itself according to these variables and calculates every other internal variable during execution via system configuration, which will be explained in Section 3.1. Then, reads and reference segments are prepared for bitwise algorithm in preprocessing (Section 3.3) and filtration is applied on each pair in GateKeeper-GPU kernel as described in Section 3.4. In order to observe the real performance of GateKeeper-GPU in a mapper's full pipeline, we also embed GateKeeper-GPU into mrFAST [17, 4]. The details of this design can be found in Section 3.5.

3.1 System Configuration

GateKeeper-GPU begins with configuring the system's properties such as device compute compatibility and architecture since some of the features, like data prefetching, is limited to compute capability of the system. Therefore,

GateKeeper-GPU recognizes system specifications beforehand to allocate memory wisely and enable the extra features accordingly.

Applying the GateKeeper algorithm for a read-reference segment pair is called a *filtration*. Since CUDA provides many massively parallel threads, each filtration is performed by a single CUDA thread in order to have the least possible dependency between the threads for high filtering throughput. Each thread uses the reserved stack frame for temporary data storage such as bitmasks. Before the filtering step, GateKeeper-GPU calculates approximate memory load of a filtration on a thread, *thread load*, by using the compilation variables of read length and error threshold. It retrieves the free global memory size of the GPU along with some other GPU parameters such as maximum number of threads per block, calculates the number of CUDA thread blocks and the number of filtrations possible for one GPU kernel call as *batch size* to fully utilize GPU for boosting performance. Since data transfers between the device and host are expensive, the number of transfers should be kept minimal. Therefore, configuration step ensures that the batch size is maximized. In multi-GPU model, batch size is equal for all devices to ensure fair workload. By this way, we automatically adjust the kernel parameters of efficient GPU usage for device model and memory status without user's concern.

3.2 Resource Allocation

The main algorithm consists of simple bitwise operations. Dividing the workload of one filtration between multiple threads introduces overheads, which emerge from inter-thread dependencies, rather than speeding up the process. Therefore, each thread runs kernel function for a single filtration with least dependency possible.

GateKeeper [1] produces $2e + 1$ Hamming masks to store the state bitvectors for indels where e is the error threshold. Since CUDA dynamic memory allocation inside device functions is restricted and slow, we use fixed-size unsigned integer

arrays for bitmasks in kernel. For this reason, GateKeeper-GPU requires read length and error threshold at compile time. Each thread uses the reserved stack frame in thread local memory for bitmasks which is cached in unified L1/Texture and L2 cache for Pascal architecture; in L1 and L2 cache for Kepler architecture [28]. All of the constant variables and look-up table (LUT) for amending procedure are stored in constant memory.

We utilize unified memory for read buffer and reference for providing simplicity and on-demand data locality. As the number of filtration operations per batch are determined in system configuration stage, read buffer is created in unified memory together with candidate reference indices corresponding to the reads. Since the number of candidate reference locations is unknown until seeding, batch size does not limit candidate reference location count, so there is no predefined value of reference segment count per read.

3.3 Preprocessing

Once the system properties are recognized and parameters are adjusted accordingly, GateKeeper-GPU starts to prepare the reads and reference for filtration. The main portion of preprocessing involves encoding the strings to 2-bit representations for bitwise operations. Since each character takes up two bits after encoding, a 16-nucleotide window is encoded into an unsigned integer (i.e., a word), so a read of length 100bp is represented as 7 words in the system.

GateKeeper is designed for DNA strings, thus it only recognizes the characters *'A', 'C', 'G'* and *'T'*. Occasionally, reads or reference may contain character *'N'* that signals for an unknown base call at that specific location of the sequence. Because GateKeeper does not recognize the character *'N'*, GateKeeper-GPU directly passes those sequence pairs from the filter without applying filtration steps as a design choice for two reasons; first, supporting an extra character requires to

expand the encoding to 3-bit representations which unnecessarily increases the complexity and memory footprint for this rare occasion; second, since these unknown bases add extra uncertainty, leaving the decision to verification increases credibility.

Assigning the encoding job to either host (i.e., CPU) or device (i.e., GPU) creates advantages and disadvantages from different perspectives. Encoding in host and copying the encoded strings to device is cost-effective in data transfer since encoding compresses the strings into smaller units. However, processing in host can be stagnant even if it is multi-threaded when compared to massively parallel processing in device. Conversely, allowing each device to encode the strings for their own operations adds extra parallelism to the whole execution while reducing the efficiency in transfer time. We provide GateKeeper-GPU in both versions and analyze the effect of processor in encoding on overall performance, which will be discussed further in the Evaluation section.

In the workflow of most of the read mappers, first, the possible reference locations are found, the read is verified, and then the same operations are carried out for the next read. Multi-threaded mappers allow processing several reads at a time. Nevertheless, for an effective GateKeeper-GPU execution, it is necessary to utilize the maximum number of threads possible for a kernel call in order to fully employ the GPU resources. Therefore, many reads and their candidate reference segments are prepared and stored for a single kernel call before verification. As the last step of preprocessing, the reads and reference segments are batched in host before GPU process depending on the batch size which is calculated in system configuration.

3.4 GateKeeper-GPU Kernel

When the buffers are ready for execution, we set the usage patterns of the buffers in terms of caller frequency about host or device by CUDA memory advice API. Since mostly kernel uses buffers until the next batch, the preferred location of the

data is set to be the GPU device for the input buffers. According to the assigned memory advice for each buffer, the data arrays are prefetched asynchronously ahead of the kernel function. This provides an early start for the migration of data from host to device and mapping data to the device's page tables before kernel accesses [28]. Since there is more than one buffer for prefetching, each buffer is submitted to a different stream in asynchronous fashion. Memory advise mechanism and prefetching are supported for compute capability 6.× and later, so these actions are skipped for lower CUDA compute capabilities.

After prefetching, kernel function is executed with the number of blocks and threads parameters which are previously set in configuration stage. Kernel performs the complete set of operations for a single filtration of GateKeeper algorithm, starting with encoding the sequences if they are not encoded in preprocessing stage.

Due to architectural differences between FPGA and GPU, some alterations are applied to maintain GateKeeper algorithm in C-derived device functions in GPU. In contrast to FPGA's specialty in bitwise operations, GPU does not support bitvectors in arbitrary sizes. An encoded read of length 100bp can be represented as a 200-bits long register in FPGA whereas GPU allocations are limited with word size of the system, so the encoded read becomes an array of 7 words. Additionally, logical shift operations produce incorrect bits in between the elements of the array. For correcting these bits, we apply *carry-bit* transfers to most significant bits of an element by looking at the least significant bits of the previous one for logical right shift, and vice versa for logical left shift. The correction procedure must be done for each and every bitwise shift, such that there are $2e$ shifts and $2e$ carry-bit (insertion and deletion masks each require e operations) operations in a filtration with an error threshold of e.

Bitwise shift operations leave most significant or least significant bits vacant, depending on and opposite to the shift direction. Even though these bits should be $1s$ on the final bitvector signalling for errors, final AND operation on all masks hide these errors since the corresponding bits are 0 in the shifted masks. This issue was previously addressed by MAGNET [24], too, and it was solved in a different

way with a combination of steps. In order to uncover these possible errors on leading and trailing parts, we add an extra OR operation to turn the excess bits into 1s after preparing amended masks. By this way, we ensure that the leading and trailing bits are 1 even if XOR operation for Hamming mask converts them into 0. In Figure 3.1, we can see how the new amended mask (lower amended mask in **blue**) covers the leading and trailing 0 which is missed by GateKeeper (upper amended mask in **orange**). In the figure, 'Ref' and 'S.Read' stand for reference bitvector and shifted read bitvector, respectively; 'H' and 'A' illustrate Hamming mask and amended mask.

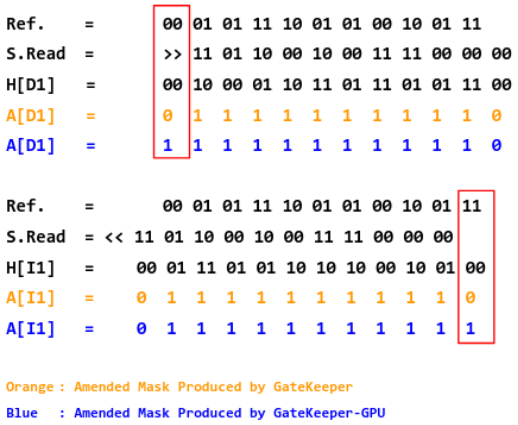


Figure 3.1: Improving Leading and Trailing 0-bit Strategy

As a result of this simple modification on the algorithm, GateKeeper-GPU can discard some of the read and reference segment pairs, that exceed error threshold, which GateKeeper accepts. Figure 3.2 shows the amended masks produced by GateKeeper and GateKeeper-GPU for the same sequence pair for error threshold 2. This improvement patch on GateKeeper-GPU enables it to give the correct decision for rejecting the pair whereas GateKeeper falsely accepts the pair.

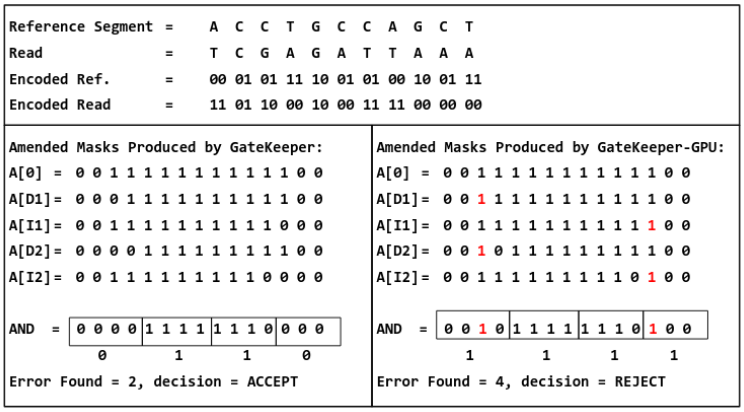


Figure 3.2: Amended Masks produced by GateKeeper [1] and GateKeeper-GPU

3.5 Adaptation to mrFAST Workflow

In order to evaluate the whole genome performance of GateKeeper-GPU, we integrate it with mrFAST [4]. GateKeeper-GPU can be adapted to any short read mapping tool that uses seed extension and we hope that this part will serve as an example for adjustment. We begin with encoding and loading reference into unified memory using multithreading with OpenMP, depicted as purple box in Figure 3.3. While encoding, the locations of *'N'* bases on reference genome are also designated since the segments containing this character will not be evaluated in filtering stage.

In original workflow of mrFAST, candidate locations of a single read are found, they are verified, mapping information of the read is recorded, then the next read is processed. Since GateKeeper-GPU requires batching, we fill the buffers with multiple reads and their candidate location indices with partial multithreaded support. The number of candidate locations of a particular read cannot be anticipated before seeding, therefore there are two factors which dynamically control the size of buffers to ensure that GPU utilization is optimized without exhausting the resources: number of filtration pairs that is calculated by system configuration unit as explained in Section 3.1 and number of reads allowed per batch. The number of reads is predetermined and in our experience with different values, as it can be seen in Table 3.1, we find that $100,000$ reads yield the best results for

mrFAST. We observe that using $100,000$ reads per batch decreases overall time spent, kernel and filter durations since the number of transfers between host and device becomes minimal. Yet, maximum number of reads is a parameter for execution and can be modified easily according to the desired mapper or situation.

Table 3.1: Effect of maximum number of reads processed per batch on time

	Encoding in Host				Encoding in Device			
Max. # Reads	**Overall**	**Encode-Copy**	**Kernel**	**Filter**	**Overall**	**Encode-Copy**	**Kernel**	**Filter**
100	3,041.52	109.54	102.55	212.17	2,944.59	100.19	105.39	187.58
1000	1,446.58	105.99	92.72	114.61	1,335.20	77.55	97.20	116.29
10,000	1,325.95	109.14	80.37	92.99	1,322.96	84.45	83.22	92.29
100,000	1,275.66	103.13	77.45	88.96	1,215.25	75.19	82.37	91.17

Effect of maximum number of reads processed per batch on time (seconds) in different stages of alignment with GateKeeper-GPU. These measurements were recorded during the mapping of chromosome 1 (CHR1). Kernel: Total filtering time measured by CUDA API, Filter: Total filtering time measured from host side.

Once the data transfer buffers are filled, kernel function is called with previously calculated number of blocks and threads parameters. Each thread executes a single comparison starting with extracting the relevant reference segment based on the index. The result of filtering decision as '1' (*accept*) or '0' (*reject*) and approximated edit distance are written back on the result buffers in unified memory.

The only synchronization point for threads in the entire execution is after the completion of filtering step for one batch. Since host and device use a common single pointer for buffers in unified memory model, the threads' job needs to be completed in order for verification to obtain filtering results. The pairs that pass the filter are verified and mrFAST continues with further steps to report the mapping information. Figure 3.3 demonstrates the workflow of mrFAST with GateKeeper-GPU. Pink arrows show the data transfers between device and host through unified memory.

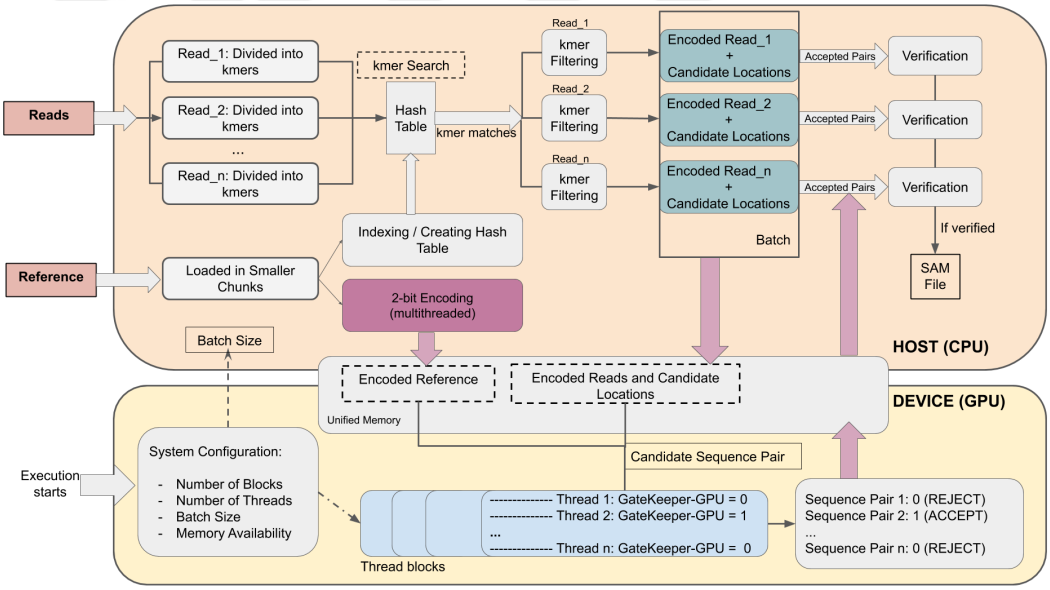


Figure 3.3: Workflow of mrFAST with GateKeeper-GPU

Chapter 4

Experimental Methodology

4.1 Datasets

In order to have a fair comparison for accuracy, we use the same datasets from original GateKeeper [1] for the most part. We run whole genome tests on mrFAST with one real and two simulated datasets. The real data is Illumina HiSeq dataset *ERR240727_1* from 1000 Genomes Project Phase I [29] which contains $4,081,242$ 100bp short reads. One of the simulated datasets was produced by running Mason [30] sequence simulator for GateKeeper's [1] evaluation and has $100,000$ deletion-rich reads with 300bp length (i.e., dataset_1). We create another simulated dataset with Mason using reference the humanG1Kv37 [31]. This new dataset (i.e., dataset_2) contains $1,000,000$ 150bp short reads which have low-indel profile. In all of our tests, we use humanG1Kv37 reference produced by 1000 Genomes Project [31].

For filtering throughput and accuracy analyses, original GateKeeper work has tests with datasets containing $30,000,000$ read and candidate reference segment pairs extracted from mrFAST workflow using *ERR240727_1* (100bp), *SRR826460_1* (150bp) and *SRR826471_1* (250bp) real short read sets (previously downloaded from www.ebi.ac.uk/ena) with different error threshold values.

We, also, run our throughput and accuracy tests on these sets in order to have consistency in comparison between the original work and GateKeeper-GPU. Additionally, we generate read and candidate reference segment pairs by one of the state-of-the-art mapping tools, Minimap2 [32], for accuracy analyses by using *ERR240727_1* (100bp) dataset. For Minimap2, we extract the pairs just before the first chaining function (*mm_chain_dp*) since this is the first dynamic programming part and gather the samples in 11 different sets for error threshold from 0 to 10, each containing $30,000,000$ pairs. By using Edlib's global alignment mode, we prepare filtering status of Minimap2 datasets with the intention of maintaining consistency on ground truth. We also record Minimap2's decisions as accepting or rejecting the pairs after *mm_chain_dp*, for comparison. In all of the experiments, the maximum error threshold is 10% of the sequence length.

4.2 Device Specifications

We run our performance experiments in two different settings. *Setting1* includes a 2.30GHz Intel Xeon Gold 6140 CPU with 754G RAM. There are in total of 8 NVIDIA GeForce GTX 1080 Ti GPUs (Pascal architecture), each with 10GB global memory connected to this processor. CUDA compute capability of the devices is 6.1 with CUDA driver v10.1 installed. Data transfer between the host and devices is managed by PCIe generation 3 with 16 lanes. In *Setting2*, we use a 3.30GHz Intel Xeon CPU E5-2643 0 processor with 256G RAM. 4 NVIDIA Tesla K20X GPGPUs are connected to this host. Each of these devices have 5GB global memory and the data transfer is maintained via PCIe generation 2 with 16 lanes. CUDA compute capability of the devices is 3.5 with CUDA driver 10.2. Tesla K20X has Kepler architecture, therefore data prefetching is not supported in Setting2. In all of our tests, we enable persistence mode in GPU to keep the devices initialized.

4.3 Filtering Throughput Analysis

We run our throughput analysis on the datasets containing read and reference segment pairs, thoroughly explained in Section 4.1. Each of the dataset curated for this purpose include 30 million pairs in total. We calculate the time taken for the filtration of a single pair out of 30 million, then we determine the total number of pairs that can be filtered in 40 minutes in order to have fair throughput comparison with the other tools. We report two different time measurements for throughput analysis: *kernel time* and *filter time*. *Kernel time* denotes the time taken only by GPU devices and we record this time by using CUDA Event API. Since GateKeeper-GPU uses batched kernel calls, we add all kernel times in an execution and report the sum. Kernel time is measured in miliseconds (ms), we convert it to seconds (s) to have consistency in units among all other timing measurements. *Filter time* represents the total time spent for filtering, including host operations such as data transfer and encoding the sequences. Therefore, we measure filter time from host's perspective. For both of these measurements, we also provide different results for encoding the pairs by host (CPU) and device (GPU). In multi-GPU throughput analysis, kernel time represents the time of the device which take the longest time to complete among all other active devices. In all of our timing experiments, we run the tests 10 times and report the arithmetic mean to minimize the effect of random experimental errors on results.

We compare GateKeeper-GPU's filtering throughput with its CPU version, denoted as *GateKeeper-CPU* in tables. In order to maintain the fairness as much as possible, we implement GateKeeper-CPU in multicore fashion and report results of 12 cores.

4.4 Accuracy Analysis

In our accuracy analyses, we hold Edlib's [22] results as the ground truth and generate the edit distance of the read and reference segment pairs by using Edlib's

global alignment. According to the edit distance threshold, we produce filtering status as accept or reject for the pairs. The reason why we prefer global alignment results instead of semiglobal alignment is that GateKeeper-GPU applies filtering for the pairs which are highly expected to match. Throughout accuracy experiments, we analyze *false accept* and *false reject* counts. A false accept represents a read and reference segment pair which is rejected by Edlib because of exceeding error threshold, but is accepted by the filter. On the contrary, false reject case is a valid pair which has less errors than the threshold, but is rejected by the filter.

We have two approaches for testing the accuracy of GateKeeper-GPU. First, we record the filtering status for the sets described in Section 4.1 and compare the results with Edlib. Since GateKeeper-GPU gives direct pass to the pairs that contain unknown base call character (*'N'*), in order to be able to observe the actual accept and reject counts, we exclude these pairs from the tests and report the comparison with Edlib accordingly. For the sake of simplicity, we will call these pairs as *undefined* for the rest of the work.

Second, we compare GateKeeper-GPU with original GateKeeper implementation [1], SHD [18], MAGNET [24], Shouji [25] and SneakySnake [26] in terms of false accept and false reject counts. We denote original GateKeeper implementation as 'GateKeeper-FPGA' for differentiation. For these tests, we choose highest-edit and lowest-edit profile datasets of three different read lengths. For 100bp, lowest-edit containing dataset is prepared by mrFAST's seeding error threshold 2 and highest-edit dataset is curated by error threshold 40. Likewise, mrFAST error thresholds for lowest-edit and highest-edit profile datasets are 4 and 70 for 150bp; 8 and 100 for 250bp, respectively. Because the other tools do not have distinguishing mechanisms for undefined pairs, in order to maintain a fair comparison in this test series, we include these pairs in GateKeeper-GPU's results and report the numbers accordingly.

4.5 Whole Genome Performance & Accuracy Analysis

In order to evaluate GateKeeper-GPU's performance and accuracy in the full workflow of a read mapping tool, we integrate GateKeeper-GPU into mrFAST [4] and perform experiments. GateKeeper [1] is also merged into mrFAST and mrFAST inherently includes Adjacency Filter as a CPU pre-alignment filter. Therefore, testing GateKeeper-GPU as a part of mrFAST enables us to have a brief comparison with the original work and Adjacency Filter. We use the same datasets that GateKeeper [1] uses in whole genome comparison experiments: one real 100bp dataset and one simulated 300bp dataset (i.e., dataset_1). We also report GateKeeper-GPU's results with another simulated 150bp dataset (i.e., dataset_2). All of these datasets are described in Section 4.1 in details. We collect the following metrics from alignment for evaluation: number of mappings, number of mapped reads, total number of candidate mappings, total number candidate mappings which enter verification, time spent for verification, time spent for preprocessing before pre-alignment filtering, and total kernel time spent for running GateKeeper-GPU. We record all timing measurements in seconds, then convert them into hours for easy comprehension. We use single GPU in all our experiments and provide results for encoding in both device and host design.

4.6 Resource Utilization and Power Analysis

Warp occupancy is one of the important indicators when evaluating the kernel performance of a CUDA application. A *warp* is a group of 32 adjacent threads in a block and Streaming Multiprocessor (i.e., SM) schedules the same instruction for the threads in a warp [33]. Warp occupancy is the ratio of number of active warps over maximum supported number of warps for SM. Theoretical occupancy is determined by the numbers of warps, blocks and registers per SM, and shared memory. With these conditions, the maximum number of registers for 100% occupancy while using all threads is 32 per thread. GateKeeper-GPU does not utilize

shared memory; and we opt for maximizing the number of warps and blocks in order to maintain high throughput. By using CUDA occupancy calculator provided by NVIDIA, we calculate GateKeeper-GPU's theoretical warp occupancy and provide a comparison with achieved occupancy value.

We perform profiling experiments for GateKeeper-GPU on the 100bp and 250bp datasets with error thresholds of 4 and 10, respectively, by using CUDA command-line profiler nvprof [34]. By carrying out system profiling and metrics analyses, we provide information about power consumption status, warp execution efficiency and multiprocessor activity.

Chapter 5

Evaluation

5.1 Accuracy Analysis

5.1.1 Accuracy of GateKeeper-GPU with respect to Edlib

In the first phase of accuracy analyses, we evaluate the accuracy of GateKeeper-GPU against Edlib using datasets with three different read lengths described in Section 4.1. We exclude the undefined pairs for both of the tools in this series of tests with the purpose of indicating the actual numbers of accepts and rejects without skipping filtration. We perform the experiments on the datasets that include the pairs with 5% of their length error allowance by mrFAST. With this rule being set; *ERR240727_1_E5* (100bp), *SRR826460_1_E6* (150bp) and *SRR826471_1_E12* (250bp) datasets contain reads and mrFAST's candidates for error thresholds respectively 5, 6 and 12. We perform experiments on these datasets with filtering error threshold from 0% to 10% of their corresponding read length. Additionally, we carry out accuracy tests with potential mappings of Minimap2 for *ERR240727_1* (100bp), as described in detail in Section 4.1. Different from mrFAST-generated dataset experiments, we use the same error threshold, that is set in Minimap2 to produce candidate pair, for filtering. The detailed results of these experiments are placed in Appendix A.1.

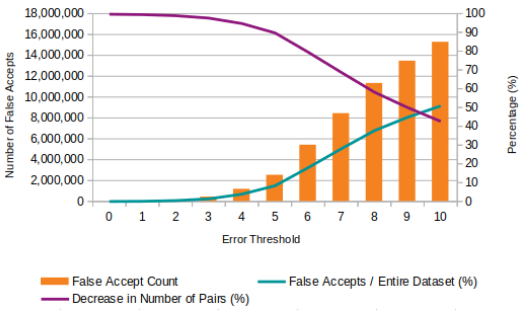


Figure 5.1: False Accept Analysis - 100bp

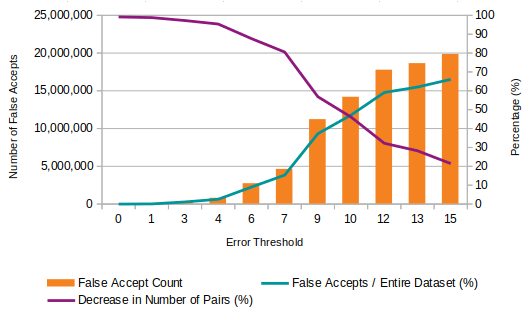


Figure 5.2: False Accept Analysis - 150bp

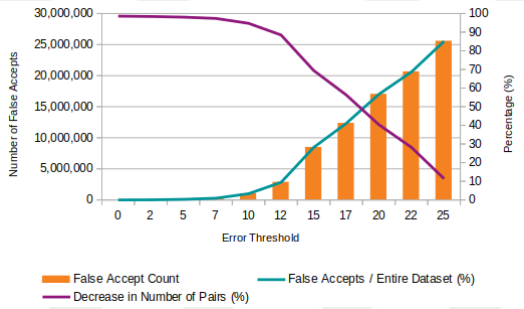


Figure 5.3: False Accept Analysis - 250bp

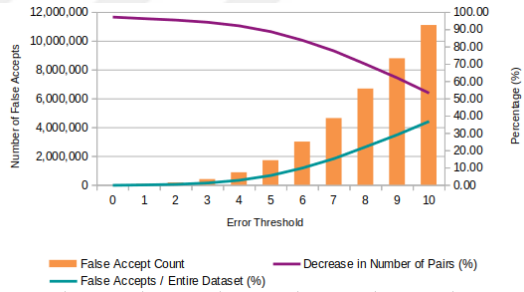


Figure 5.4: False Accept Analysis - Minimap2 dataset 100bp

According to comparison with Edlib's global alignment results, GateKeeper-GPU's false reject count is always 0 for all of the datasets, indicating that it never rejects a true extendable read and reference segment pair. For mrFAST's potential mappings, Figures 5.1, Figure 5.2 and Figure 5.3 illustrate false accept count, the percentage of false accepts over total number of pairs (30 million) and the percentage of decrease in the total number of pairs by rejecting candidate mappings, with respect to the error threshold. Figure 5.4 depicts the same information for Minimap2 datasets. This enables us to have a comparison for GateKeeper-GPU's reduction in potential mappings produced by different tools.

Considering the results of these experiments, we make the following observations regarding the accuracy of GateKeeper-GPU: 1) Up to $\sim$ 3% error thresholds of all read lengths, GateKeeper-GPU can correctly reject more than 90% of the mappings with less than 10% of false accept ratio and with 0 false reject. 2) Even though the efficiency of filtering decrements when the error threshold increases, filtering still continues to serve without a steep drop in efficiency, for the largest error threshold allowed (10%). 3) With an increase in read length, the

increment in false accept rate and decrease in filtering efficiency become sharper. Therefore, we make a conclusion that the maximum error threshold, for which GateKeeper-GPU keeps its filtering efficiency, corresponds to the point where false accept ratio and decrease in number of pairs graphs meet. For instance, the maximum error thresholds are 9, 10 and 19 for read lengths 100bp, 150bp and 250bp, respectively.

5.1.2 Comparison with Other Pre-alignment Filters

We compare GateKeeper-GPU's results with 5 other filtering tools; its original FPGA GateKeeper implementation [1], SHD [18], MAGNET [24], Shouji [25] and SneakySnake [26] with respect to Edlib's ground truth in the second phase of accuracy analyses. For this purpose, we use low-edit profile and high-edit profile datasets, previously described in Section 4.4 in details. Low-edit profile datasets are *ERR240727_1_E2* (mrFAST error threshold = 2, 100bp), *SRR826460_1_E4* (mrFAST error threshold = 4, 150bp) and *SRR826471_1_E8* (mrFAST error threshold = 8, 250bp); high-edit profile datasets are *ERR240727_1_E40* (mrFAST error threshold = 40, 100bp), *SRR826460_1_E70* (mrFAST error threshold = 70, 150bp) and *SRR826471_1_E100* (mrFAST error threshold = 100, 250bp). We perform experiments on these datasets with filtering error threshold from 0% to 10% of their corresponding read length. We retrieved false accept and false reject counts of other tools from supplementary material of Shouji [25] work. We call the sequence pairs that contain the unknown base call character '*N*', *undefined* pairs. In order to maintain fair comparison, we include undefined pairs in this test series and mark these pairs as false accept in GateKeeper-GPU's results, where necessary, since it skips filtering these potential mappings. The detailed results of these tests are placed in Appendix A.2.

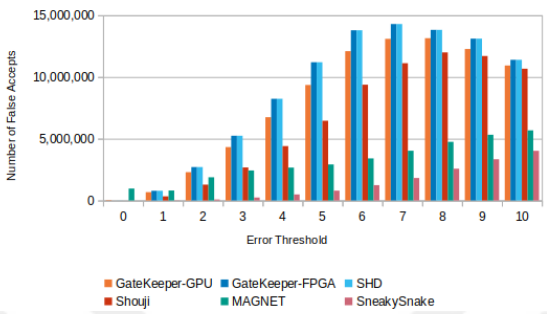


Figure 5.5: False Accept Comparison for Low-Edit Profile in Read Length 100bp, number of undefined pairs = 28,009

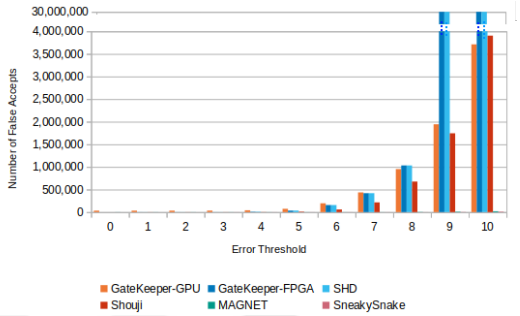


Figure 5.6: False Accept Comparison for High-Edit Profile in Read Length 100bp, number of undefined pairs = 31,487

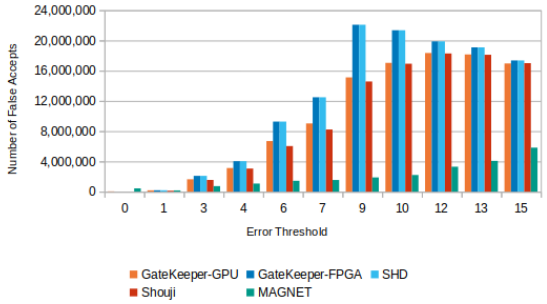


Figure 5.7: False Accept Comparison for Low-Edit Profile in Read Length 150bp, number of undefined pairs = 30,142

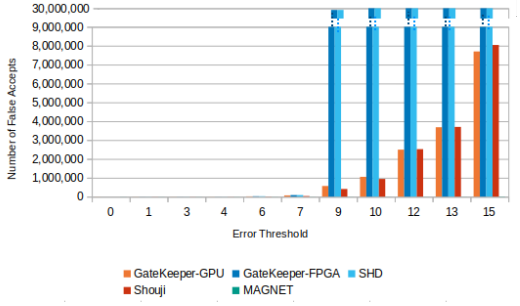


Figure 5.8: False Accept Comparison for High-Edit Profile in Read Length 150bp, number of undefined pairs = 309

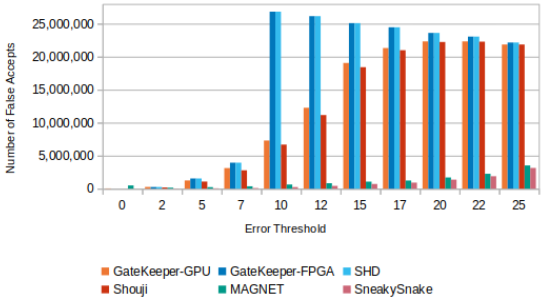


Figure 5.9: False Accept Comparison for Low-Edit Profile in Read Length 250bp, number of undefined pairs = 35,072

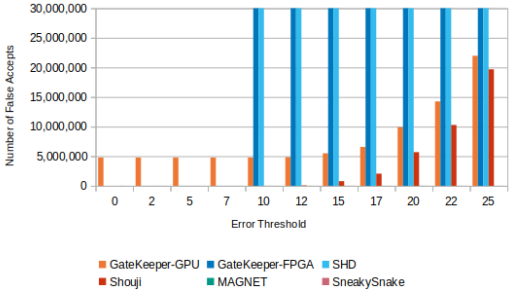


Figure 5.10: False Accept Comparison for High-Edit Profile in Read Length 250bp, number of undefined pairs = 4,763,682

Figures from 5.5 to 5.10 demonstrate the the number of false accept rates by different tools across same datasets with varying error threshold. Even though GateKeeper-GPU has undefined pairs in its false accept count, we see that it has less false accept rate in most of the cases and can produce up to 52× less number of false accepts (Figure 5.8, on 150bp dataset with error threshold 9) compared to original GateKeeper-FPGA and SHD. In Figure 5.10, the reason behind GateKeeper-GPU's high false accept count in low error thresholds is the presence of high amount of undefined pairs. Apart from undefined pairs, it produces much less false accepts. We observe that both GateKeeper-FPGA and SHD completely stop filtering in high error thresholds of high-edit profile datasets and accept all pairs (30 million). In contrast to these, GateKeeper-GPU still functions in those cases and continues to correctly decrease the number of potential mappings. This suggests, the small modifications made on GateKeeper algorithm [1] for leading and trailing parts of the bitvectors, explained in detail in Section 3.4, improved the accuracy and helped GateKeeper-GPU to keep its consistency without completely letting loose of filtering in high error thresholds.

In all of the tests, SneakySnake and MAGNET have the lowest numbers of false accepts. However, we notice that MAGNET produces some false accepts for exact matching when error threshold is 0 as can be seen in Figure 5.5 and Figure 5.7, and generates false rejects in some cases where GateKeeper-GPU and other filters do not have false rejects. GateKeeper-GPU very rarely produces false accepts in exact matching and the number of false accepts in these cases are always lower than MAGNET. Finally, for datasets with read lengths 150bp and 250bp, GateKeeper-GPU and Shouji have similar false accept rates but Shouji's false accept rate is less than GateKeeper-GPU, especially in high error thresholds.

5.2 Filtering Throughput Analysis

In order to assess GateKeeper-GPU's filtering throughput, we perform experiments on the datasets *ERR240727_1_E5* (mrFAST error threshold = 5, 100bp), *SRR826460_1_E10* (mrFAST error threshold = 10, 150bp) and

SRR826471_1_E15 (mrFAST error threshold = 15, 250bp). With respect to kernel time (kt) and filter time (ft), we calculate the throughput of GateKeeper-GPU. We report the results with filtering error thresholds 2 and 5, 4 and 10, 6 and 10 for the datasets with read lengths 100bp, 150bp and 250bp respectively. Tables from 5.1 to 5.6 contain the filtering throughput of GateKeeper-CPU and GateKeeper-GPU with different values for encoding in device and host designs, in terms of billions of filtrations in 40 minutes. In the case of GateKeeper-CPU, kernel time represents the time exclusively spent by the function that contains the GateKeeper algorithm. We provide GateKeeper-CPU's results for both with single core and 12-core experiments, but we make comparisons based on 12-core results in order to be as fair as possible. Please refer to Appendix A.3 for the detailed results of these experiments.

Considering filter time results in Setting1, GateKeeper-GPU can filter up to 3× and 20× more pairs than 12-core GateKeeper-CPU, with single and 8 GPUs, respectively (device_encoded, 250bp, error threshold = 10). In terms of only kernel time, GateKeeper-GPU's filtering throughput can reach up to 72× and 456× more than 12-core GateKeeper-CPU, with single and 8 GPUs, respectively (host_encoded, 250bp, error threshold = 6). In Setting2 with single GPU, GateKeeper-GPU can filter up to 2× and 16× more pairs than 12-core GateKeeper-CPU with respect to filter time and kernel time (device_encoded, 250bp, error threshold = 10). In Table 5.5 and Table 5.6, we see that 12-core GateKeeper-CPU's performance is slightly better than single device GateKeeper-GPU with encoding in host.

Table 5.1: 100bp Filtering Throughput in Setting1

		GateKeeper-CPU		**Device-encoded**		**Host-Encoded**	
	Error Threshold	**1 Core**	**12 Cores**	**1 GPU**	**8 GPUs**	**1 GPU**	**8 GPUs**
kt	**2**	0.7	7.2	244.8	1,189.8	476.8	**3,198.4**
	5	0.4	3.9	150.8	1,041.4	249.3	**1,684.7**
ft	**2**	0.6	6.5	7.7	**39.2**	3.0	14.4
	5	0.4	3.7	7.6	**37.8**	2.9	14.2

Filtering Throughput is calculated wrt. kernel time (kt) and filter time (ft), in terms of billions of pairs in 40 minutes. Highest filtering thoughput within the row is in bold font.

Table 5.2: 150bp Filtering Throughput in Setting1

		GateKeeper-CPU		Device-encoded		Host-Encoded	
	Error Threshold	**1 Core**	**12 Cores**	**1 GPU**	**8 GPUs**	**1 GPU**	**8 GPUs**
kt	**4**	0.3	2.9	89.0	496.5	205.2	**1,022.0**
	10	0.1	1.4	61.9	406.8	98.0	**582.7**
ft	**4**	0.2	2.7	5.2	**25.5**	1.6	9.3
	10	0.1	1.3	5.1	**26.5**	1.6	9.2

Filtering Throughput is calculated wrt. kernel time (kt) and filter time (ft), in terms of billions of pairs in 40 minutes. Highest filtering thoughput within the row is in bold font.

Table 5.3: 250bp Filtering Throughput in Setting1

		GateKeeper-CPU		Device-encoded		Host-Encoded	
	Error Threshold	**1 Core**	**12 Cores**	**1 GPU**	**8 GPUs**	**1 GPU**	**8 GPUs**
kt	**6**	0.1	1.3	45.7	271.8	97.4	**611.6**
	10	0.1	0.9	38.4	248.6	61.8	**331.4**
ft	**6**	0.1	1.3	3.3	**17.0**	1.0	6.5
	10	0.1	0.9	3.3	**17.5**	1.0	6.4

Filtering Throughput is calculated wrt. kernel time (kt) and filter time (ft), in terms of billions of pairs in 40 minutes. Highest filtering thoughput within the row is in bold font.

Table 5.4: 100bp Filtering Throughput in Setting2

		GateKeeper-CPU		Device-encoded	Host-Encoded
	Error Threshold	**1 Core**	**12 Cores**	**Single GPU**	**Single GPU**
kt	**2**	0.7	5.5	41.1	**72.2**
	5	0.3	3.0	29.1	**42.0**
ft	**2**	0.6	4.9	**6.1**	2.7
	5	0.3	2.8	**5.7**	2.7

Filtering Throughput is calculated wrt. kernel time (kt) and filter time (ft), in terms of billions of pairs in 40 minutes. Highest filtering thoughput within the row is in bold font.

Table 5.5: 150bp Filtering Throughput in Setting2

		GateKeeper-CPU		Device-encoded	Host-Encoded
	Error Threshold	**1 Core**	**12 Cores**	**Single GPU**	**Single GPU**
kt	**4**	0.3	2.5	18.8	**32.5**
	10	0.1	1.2	12.0	**16.5**
ft	**4**	0.3	**2.3**	**2.3**	1.7
	10	0.1	**1.2**	**1.2**	1.6

Filtering Throughput is calculated wrt. kernel time (kt) and filter time (ft), in terms of billions of pairs in 40 minutes. Highest filtering thoughput within the row is in bold font.

Table 5.6: 250bp Filtering Throughput in Setting2

		GateKeeper-CPU		Device-encoded	Host-Encoded
	Error Threshold	**1 Core**	**12 Cores**	**Single GPU**	**Single GPU**
kt	**6**	0.1	1.1	**15.8**	**15.8**
	10	0.1	0.7	**12.0**	10.7
ft	**6**	0.1	**1.1**	**1.1**	1.0
	10	0.1	**0.7**	**0.7**	1.0

Filtering Throughput is calculated wrt. kernel time (kt) and filter time (ft), in terms of billions of pairs in 40 minutes. Highest filtering thoughput within the row is in bold font.

Even though in some cases 12-core GateKeeper-CPU performs slightly better than single device GateKeeper-GPU with encoding in host option in terms of filter time, the biggest advantage of GPU implementation over CPU implementation is having a constant performance when error threshold increases. Figure 5.11 takes a close picture of the trends that 12-core GateKeeper-CPU and single device GateKeeper-GPU implementations follow in increasing error threshold, in sequence length 250bp, experimented with both Setting1 and Setting2. While the filter time remains constant in both device-encoded and host-encoded GateKeeper-GPU, the growth in GateKeeper-CPU's filter time is almost linear with increasing error threshold. Hence, GateKeeper-GPU performs better with high error thresholds.

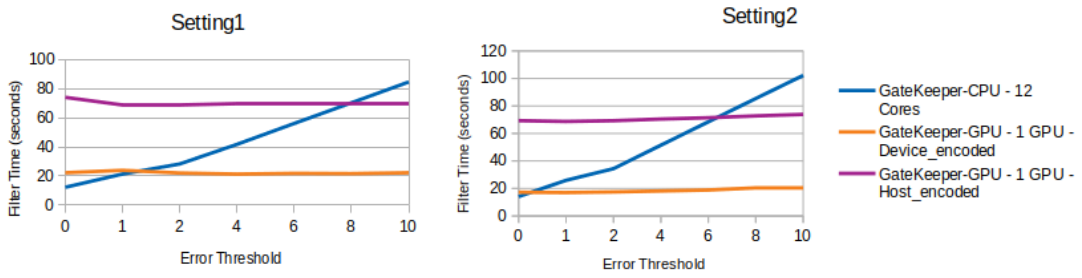


Figure 5.11: Effect of increasing error threshold on the performance of GateKeeper-CPU and GateKeeper-GPU by means of filter time (seconds)

Up to now, in filtering throughput analyses, we briefly touched upon the effect of encoding actor, host or device, on GateKeeper-GPU's performance. For the purpose of focusing on this issue, Figures from 5.12 to 5.14 provide detailed information. We notice that in all three different sequence lengths, encoding in host leads to a higher throughput when we consider kernel time (depicted by bars in figures). Especially in lower error thresholds, the difference between

host-encoded and device-encoded throughput values becomes huge. On the other hand, it turns into the opposite situation when filter time (depicted by lines in figures) is taken into consideration. From these observations, we conclude that encoding procedure creates a bottleneck for the workflow of GateKeeper-GPU and the encoding actor plays a critical role for efficiency. Since GateKeeper-GPU is an intermediate tool, choosing the right encoding actor in different scenarios can lead to best results with GateKeeper-GPU.

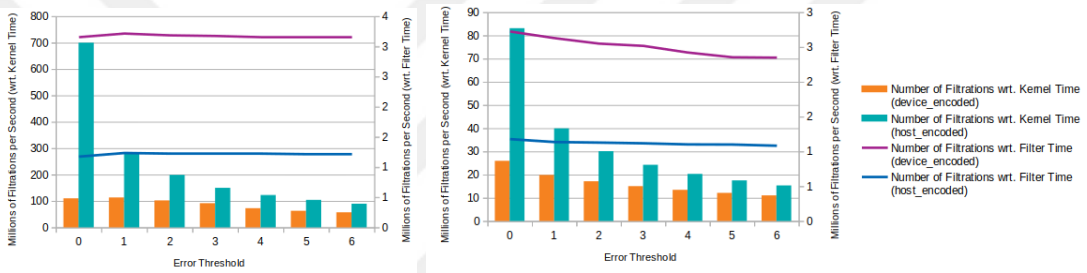


Figure 5.12: Effect of Encoding Actor (device or host) on Filtering Throughput of single-GPU GateKeeper-GPU with Increasing Error Threshold for Read Length 100bp in Setting1 and Setting2, respectively.

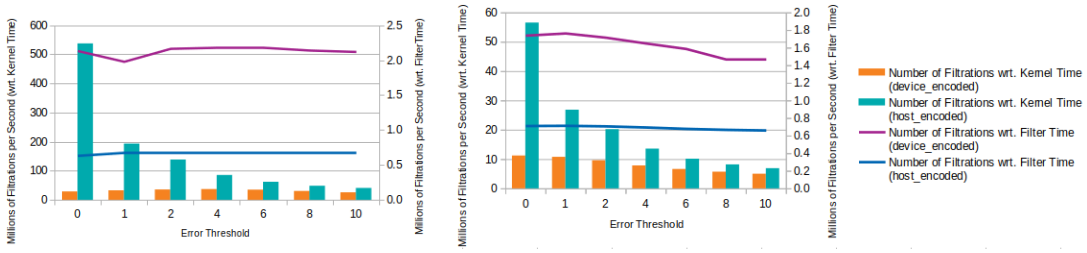


Figure 5.13: Effect of encoding actor (device or host) on filtering throughput of single-GPU GateKeeper-GPU with increasing error threshold for read length 150bp in Setting1 and Setting2, respectively

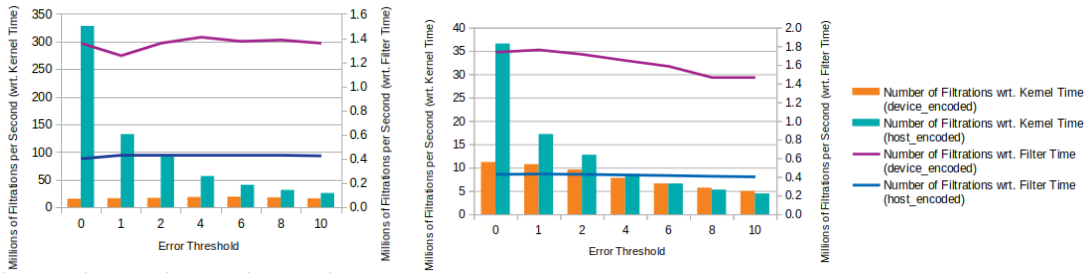


Figure 5.14: Effect of encoding actor (device or host) on filtering throughput of single-GPU GateKeeper-GPU with increasing error threshold for read length 250bp in Setting1 and Setting2, respectively

We previously stated that error threshold has a negligible effect on GateKeeper-GPU's performance and it can yield a constant efficiency with increasing error threshold. Apart from the encoding actor, another factor which has an influence on GateKeeper-GPU's filtering throughput is the sequence length. In longer sequences, GateKeeper-GPU tends to filter with lower throughput rate, as confirmed in Figure 5.15.

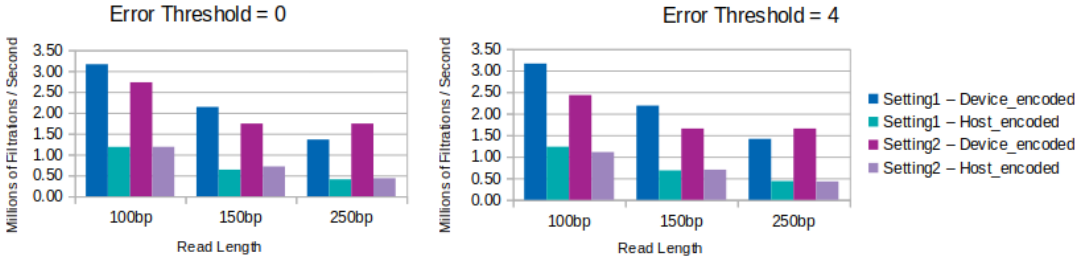
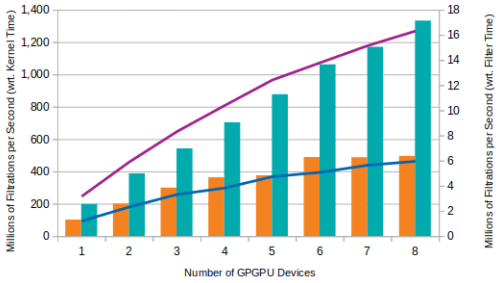


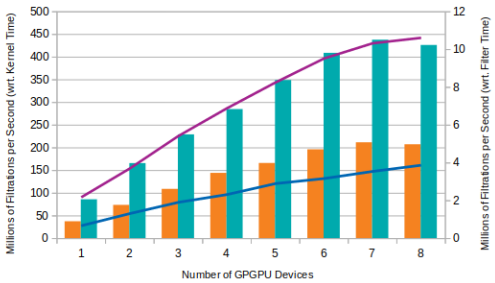
Figure 5.15: Effect of sequence length on single-GPU GateKeeper-GPU's filtering throughput in terms of millions / second, with error thresholds 0 and 4. Filtering throughput is calculated with respect to filter time.

In order to see how GateKeeper-GPU's performance scales with increasing the number of GPGPU devices, we perform tests with 8 GPUs and report the results in Figure 5.16. With respect to filter time, we find that GateKeeper-GPU with encoding in device (shown with **violet** lines on figures) experiences a steeper increase in performance as the number of devices increases. On the other hand, regarding the kernel time, encoding in device (depicted as **orange** bars in figures) makes GateKeeper-GPU show a slower growth in performance with more devices when compared to encoding in host. In some cases with respect to kernel time, we observe that GateKeeper-GPU's throughput almost doubles when encoding is

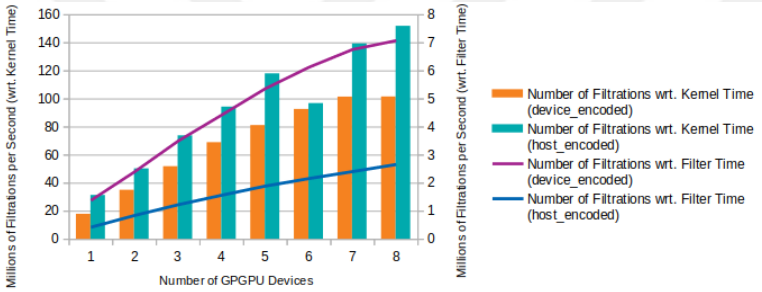
done in host by adding one more device to the system, therefore GateKeeper-GPU is more well-adapted to multi-GPU environment with host-encoded fashion.



(a) 100bp, Error Threshold = 2



(b) 150bp, Error Threshold = 4



(c) 250bp, Error Threshold = 8

Figure 5.16: Multi-GPU filtering throughput of GateKeeper-GPU in Setting1

We observe that in general GateKeeper-GPU's throughput is lower in Setting2 than in Setting1 using single GPU. Setting2 does not support data prefetching and has smaller global memory, both of which are crucial for GateKeeper-GPU's performance, therefore it tends to produce less filtrations than Setting1.

Due to platform differences, we believe that it would not be completely fair to make a comparison between GateKeeper-GPU and other filtering tools in terms of filtering throughput. To create a point of reference, we can briefly express the following observations. GateKeeper-FPGA [1] and SHD [18] can filter up to respectively $\sim$ 4 trillion and 86 billion potential mappings in 40 minutes. Regarding kernel time, GateKeeper-GPU can filter more than 7 trillion pairs within 40 minutes with 8-GPUs in host-encoding fashion, in Setting1 on sequence length of 100bp with error threshold 0. However, when we consider filter time, the highest number of filtration pairs is $\sim$ 40 billion within 40 minutes, on the same dataset conditions with device-encoding option on 8 GPUs.

Considering all of the observations on the filtering throughput of GateKeeper-GPU, we conclude that when the other variables are kept constant, read length and encoding actor (host or device) are critical variables that directly affect the performance whereas error threshold has a negligible effect. Data prefetching also plays an important role as a platform-dependent factor.

5.3 Whole Genome Accuracy & Performance Analysis

We evaluate the accuracy and performance of GateKeeper-GPU in full read mapping workflow using one real *ERR240727_1* (100bp) and two simulated (dataset 1: 300bp and dataset 2: 150bp) datasets, which contain $4,081,242$, $100,000$, and $1,000,000$ reads, respectively.

Table 5.7: Whole genome mapping information with pre-alignment filtering on real dataset

mrFAST w/	-e	Mappings	Mapped Reads	Verification Pairs	Rejected Pairs (Reduction)
No Filter	**0**	13,800,412	3,052,036	257,927,779	NA
	5	639,841,922	3,887,943	45,664,847,515	NA
Adjacency Filter	**0**	13,800,412	3,052,036	208,182,528	49,744,921 (19 %)
	5	639,841,922	3,887,943	45,664,847,515	0 (0 %)
GateKeeper-GPU	**0**	13,800,412	3,052,036	13,824,296	244,103,483 (94 %)
	5	639,820,825	3,887,939	4,289,442,302	41,375,405,213 (90 %)

Mapping information by running mrFAST on ERR240727_1 (100bp) dataset with error thresholds 0 and 5. The entries represent the number of corresponding metric. -e is the error threshold for filtering and mrFAST's edit distance threshold.

Table 5.8: Whole genome mapping information with pre-alignment filtering on simulated dataset_1 (300bp)

mrFAST w/	Mappings	Mapped Reads	Verification Pairs	Rejected Pairs (Reduction)
No Filter	670	626	365,478,108	NA
Adjacency Filter	670	626	365,478,108	0 (0%)
GateKeeper-GPU	670	626	10,305,218	355,172,890 (97%)

Mapping information by running mrFAST on the simulated dataset_1(300bp) with error threshold 15. The entries represent the number of corresponding metric.

Table 5.9: Whole genome mapping information with pre-alignment filtering on simulated dataset_2 (150bp)

mrFAST w/	Mappings	Mapped Reads	Verification Pairs	Rejected Pairs (Reduction)
No Filter	34,677,103	918,403	10,379,001,396	NA
Adjacency Filter	34,677,103	918,403	10,379,001,396	0 (0%)
GateKeeper-GPU	34,677,011	918,403	962,733,131	9,416,268,265 (90%)

Mapping information by running mrFAST on the simulated dataset_2 (150bp) with error threshold 8. The entries represent the number of corresponding metric.

For the accuracy tests, we carry out experiments with error thresholds 0 and 5 for 100bp real dataset; with error thresholds 15 for 300bp simulated dataset_1 and 8 for 150bp simulated dataset_2. Table 5.7, Table 5.8, and Table 5.9 contain mapping information of mrFAST regarding the number of metrics with GateKeeper-GPU and Adjacency Filter. We observe that GateKeeper-GPU can successfully decrease the number of potential mappings by $\sim 90\%$. On the other hand, Adjacency Filter fails to reduce the number of candidate pairs when error threshold is 5, 8 and 15. When error threshold is 0, GateKeeper-GPU can filter out 94% of the faulty mappings with high accuracy, whereas Adjacency Filter can decrease this count by 19%.

We notice that there is a small discrepancy between the total number of mappings and mapped reads produced by mrFAST with and without GateKeeper-GPU when error threshold is 5 on real dataset, as shown in Table 5.7. In order to make a deeper analysis on these potential false rejects, we collect some information about the candidate pairs which GateKeeper-GPU rejected and verification accepted, with random sampling. We run Edlib's global alignment on these samples and find that all of these pairs exceed the error threshold, supporting GateKeeper-GPU's decision on rejection. Furhermore, we never experience false rejects in our deep accuracy analysis, as explained in Section 5.1. Therefore, we make two observations on this issue: 1) by rejecting these mappings, GateKeeper-GPU actually *increases* the accuracy of the alignment of mrFAST, or, 2) there are some cases in which GateKeeper-GPU produces false rejects, even though these cases are extremely rare.

Table 5.10: Theoretical Speedup vs Achieved Speedup in Verification

	Theoretical DP Time / Speedup		Achieved DP Time / Speedup	
mrFAST w/	**Setting1**	**Setting2**	**Setting1**	**Setting2**
No Filter	NA	NA	3.99h	4.66h
GateKeeper-GPU (d)	0.37h / 10.6×	0.44h / 10.6×	1.08h / 3.7×	1.22h / 3.8×
GateKeeper-GPU (h)			1.10h / 3.6×	1.24h / 3.7×

Theoretical Speedup vs Achieved Speedup in Verification using mrFAST with GateKeeper-GPU with single GPU (encoding in d: device, h: host) on *ERR240727_1* (100bp) dataset with error threshold 5. GateKeeper-GPU provides 90% reduction in number of potential mappings. DP: verification.

With respect to the amount of reduction GateKeeper-GPU provides in the number of candidate mappings that enter verification stage, we calculate the theoretical verification time and speedup by direct proportion. Table 5.10 shows the comparison between theoretical speedup and achieved speedup for only verification stage with GateKeeper-GPU on 100bp dataset with error threshold 5. Based on our calculations, we expect 10.6× speedup on verification and in practice, verification can achieve up to 3.8× speedup.

For evaluating how much impact the reduction in the number of candidate mappings that enter verification has on whole genome alignment speedup in a large scale, we construct Table 5.11 to 5.13. We sum up the time spent for filtering and verification in comparison to the verification time of mrFAST when no pre-alignment filtering is used. For filtering time, we consider the kernel time for GateKeeper-GPU. We report only the speedup values of GateKeeper-FPGA since it requires a different platform and the time measurements cannot be scaled for our setting. Although GateKeeper-GPU has better accuracy than GateKeeper-FPGA [1] and can filter out more candidate mappings, GateKeeper-FPGA performs better than both GateKeeper-GPU and Adjacency Filter in terms of accelerating the whole alignment process.

Table 5.11: Speedup comparison between mrFAST's performance with different pre-alignment filters on real dataset

	Filtering + DP Time / Speedup		Overall Time / Speedup	
mrFAST w/	**Setting1**	**Setting2**	**Setting1**	**Setting2**
No Filter	3.99h / NA	4.66h / NA	6.85h / NA	7.81h / NA
Adjacency Filter	5.27h / -	5.61h / -	8.62h / -	9.12h / -
GateKeeper-GPU (d)	1.38h / 2.9×	2.85h / 1.6×	4.79h / 1.4×	6.63h / 1.2×
GateKeeper-GPU (h)	1.38h / 2.9×	2.77h / 1.7×	5.26h / 1.3×	6.82h / 1.2×
GateKeeper-FPGA		*41×		*9.7×

Speedup comparison between mrFAST's performance with using Adjacency Filter, GateKeeper-GPU with single GPU (encoding in d: device, h: host) and GateKeeper-FPGA on ERR240727_1 (100bp) dataset with error threshold 5. DP: verification. *values were retrieved from GateKeeper [1] article.

Table 5.12: Speedup comparison between mrFAST's performance with different pre-alignment filters on simulated dataset_1

	Filtering + DP Time / Speedup		Overall Time / Speedup	
mrFAST w/	**Setting1**	**Setting2**	**Setting1**	**Setting2**
No Filter	0.02h / NA	0.05h / NA	0.12h / NA	0.12h / NA
Adjacency Filter	0.05h / -	0.05h / -	0.14h / -	0.15h / -
GateKeeper-GPU (d)	0.13h / -	0.12h / -	0.31h / -	1.10h / -
GateKeeper-GPU (h)	0.12h / -	0.12h / -	0.31h / -	1.06h / -
GateKeeper-FPGA		*279×		*11×

Speedup comparison between mrFAST's performance with using Adjacency Filter, GateKeeper-GPU with single GPU (encoding in d: device, h: host) and GateKeeper-FPGA on the dataset_1 generated with mason (300bp) with error threshold 15. DP: verification. *values were retrieved from GateKeeper [1] article.

Table 5.13: Speedup comparison between mrFAST's performance with different pre-alignment filters on simulated dataset_2

	Filtering + DP Time / Speedup		Overall Time / Speedup	
mrFAST w/	**Setting1**	**Setting2**	**Setting1**	**Setting2**
No Filter	1.15h / NA	1.17h / NA	2.07h / NA	2.13h / NA
Adjacency Filter	1.39h / -	1.37h / -	2.42h / -	2.39h / -
GateKeeper-GPU (d)	0.38h / 3.0×	0.97h / 1.2×	1.96h / 1.1×	2.23h / -
GateKeeper-GPU (h)	0.33h / 3.4×	0.93h / 1.2×	1.49h / 1.4×	2.19h / -

Speedup comparison between mrFAST's performance with using Adjacency Filter, GateKeeper-GPU with single GPU (encoding in d: device, h: host) on the dataset_2 generated with Mason (150bp) with error threshold 8. DP: verification.

For 100bp reads, GateKeeper-GPU can accelerate verification stage up to 2.9× and 1.7× with Setting1 and Setting2, when filtering and verification is combined. We directly observe the benefit of data prefetching on Setting1 since it creates a larger speedup when compared to Setting2. These speedup values reflect to overall speedup as up to 1.4× and 1.2×, respectively. For this dataset, GateKeeper-GPU outperforms Adjacency Filter. On the other hand, we find that neither Adjacency Filter nor GateKeeper-GPU provides speedup for 300bp simulated dataset_1.

Even though GateKeeper-GPU can correctly discard 97% of candidate mappings (Table 5.8), we notice that the kernel time is longer for 300bp and additional operations, such as preparing buffers and data transfer, which are inserted into mrFAST's workflow and required by pre-alignment filtering with GPU, dominate over the time gained from verification stage. Also, the dataset size is small. Due to the fact that GateKeeper-GPU exhibits its performance with large batch sizes better, the size of the dataset can be another factor for the absence of speedup. On simulated dataset_2 (150bp), GateKeeper-GPU can achieve speedup for both verification and overall time in Setting1. Setting2 is deprived of data prefetching and has smaller global memory, thus the acceleration on verification time is not enough to reflect the impact on overall mapping duration. We, once again, observe the effect of these factors on unified memory usage with the difference created between these two device settings.

5.4 Resource Utilization and Power Analysis

5.4.1 Resource Utilization

The smallest number of registers that GateKeeper-GPU kernel uses for fully completing its functionalities is 40 registers. In different cases, the register count can increase up to 48 registers per thread. The maximum theoretical occupancy that can be reached with 48 registers per thread is 63%, but the number of threads per block should be at most 256 which is quarter of the maximum number of threads per block. Decreasing the number of threads decreases the batch size for one transfer between host and device; and eventually increases the number of transfers. In our experience, we observe that most of the time spent for filtration is not the kernel time, but the time spent in preparation for kernel. Hence, having the smallest number of transfers is optimal in our scenario and we opt for setting maximum number of threads for maximized batch size. Following this way, GateKeeper-GPU's theoretical warp occupancy is 50%. In Setting1 with encoding in device option, average achieved occupancy is 48.5% and 49.2% for 100bp and 250bp sequence sets, respectively. When sequences are encoded in the host, the average occupancy becomes 47.5% and 48.9%. Likewise in Setting2 with encoding in device option, average achieved occupancy is 46.8% and 48.7% for 100bp and 250bp sequence sets. When sequences are encoded in host, the average occupancy becomes 44.6% and 47.8%, respectively.

GateKeeper-GPU's achieved occupancy is very close to theoretical warp occupancy. This suggests that warp scheduler can issue new instructions with negligible or no stalls, and theoretical number of warps are almost reached while SM is active. Therefore, within this limit of register count, the workload within and between the blocks is balanced. On the other hand, because the occupancy is half of its maximum value, average warp execution efficiency is 79.1% and 74.5% in Setting1; 80.2% and 76% in Setting2 for 100bp dataset with encoding on device and host, respectively. For 250bp dataset in both of the device settings, warp execution efficiency is always above 98% on average. This difference created by increasing sequence length can indicate that in longer sequences, the occupancy

is already at the optimum level for hiding latency.

In order to achieve highest throughput possible, it is GateKeeper-GPU's priority to effectively and fully utilize the computing resources. We observe that SM efficiency is always above 98% on average and never goes below 95% regardless of sequence length and encoding actor in both of the device settings. This high multiprocessor activity shows that SM(s) are almost never idle during execution.

5.4.2 Power Consumption Analysis

The power consumption of a single GPGPU device for running GateKeeper-GPU is portrayed in Table 5.14 and Table 5.15. For 100bp and 250bp datasets, we run with error threshold 4 and 10, respectively, in order to evaluate the average energy use of kernel. The first observation is the vague effect of encoding actor on kernel power consumption when sequence length is 100bp. Results show that encoding the sequences on device or host does not create a huge difference. Even though encoding in device puts more workload on device, the energy consumption raise is 2986 mW on average as a result of effective parallelism in Setting2. In general frame, what creates a difference in power consumption is the increase the sequence length. The kernel tends to use more power in longer sequences due to an increase in memory usage and eventually processing more words.

Table 5.14: Power Consumption of GateKeeper-GPU in Setting1

	Device-encoded		Host-encoded	
Power (mW)	**100bp**	**250bp**	**100bp**	**250bp**
min	8,901	8,702	8,803	8,628
max	113,218	238,701	157,730	260,681
average	61,868	89,023	61,881	77,109

Power Consumption (milliwatt) for single GPGPU in Setting1. The values were obtained by running CUDA command-line profiler nvprof.

Table 5.15: Power Consumption of GateKeeper-GPU in Setting2

	Device-encoded		Host-encoded	
Power (mW)	**100bp**	**250bp**	**100bp**	**250bp**
min	30,193	30,097	30,194	30,097
max	107,756	123,026	125,849	129,799
average	77,699	85,532	74,713	77,684

Power Consumption (milliwatt) for single GPGPU in Setting2. The values were obtained by running CUDA command-line profiler nvprof.

Chapter 6

Discussion & Future Work

Having compute-intensive nature and heavy workload of data make verification stage a bottleneck for entire read mapping process. Pre-alignment filters are designed to facilitate verification, by means of reducing the workload, as accurate and fast as possible. Different techniques and hardware platforms are utilized for a quick filtering experience. In that sense, GateKeeper-GPU positions itself at a middle level of being the most accurate and fastest pre-alignment filtering tool. Compared to the original GateKeeper [1] which was implemented in FPGA, GateKeeper-GPU is more accurate but its benefit on the acceleration of entire read alignment process is smaller. On the other hand, being a GPGPU tool makes it more preferable than an FPGA tool. Since it is implemented on GPU, it is also a lot more promising for further improvements.

We believe that, although GateKeeper-GPU still brings benefits, it is more advantageous to consider GateKeeper-GPU when a new short read alignment tool is constructed with a verification-aware design rather adapting it to an existing read alignment workflow, since it requires extra steps for filtration such as encoding. With a well-developed hardware-software co-design of read mapping, it can have a powerful impact on the entire mapping procedure. For that reason, we provide GateKeeper-GPU with two different modes in encoding, both of which can be desirable in different scenarios. If the read aligner is designed

to process encoded sequences in its original workflow, then, GateKeeper-GPU's encoding can be skipped and host-encoded version can be utilized. In other scenarios where the time spent for encoding can be hidden during kernel execution, device-encoded version can be useful.

The actual reasons which degrade GateKeeper-GPU's overall performance are tied to preprocessing work that has to be carried out for kernel execution. As a future work, we intend to solve these problems and improve the extra time spent for preparation in order to bring out GateKeeper-GPU's best performance. In addition, we notice that GateKeeper-GPU mainly utilizes L2 cache with an average hit rate of 86.2% rather than unified/texture L1 cache. The hit rate of unified/texture L1 cache is 31.2% on average, which is low. Improving cache utilization is another aim of ours for encouraging more efficient kernel activity.

GateKeeper-GPU brings many benefits over its original work even though the acceleration is less. It also has comparable results with some of the peer tools. Having the issues resolved, the outcomes can be more profitable. Therefore, GateKeeper-GPU is a promising pre-alignment tool with a wide range of platform support owing to its simple design and GPGPU code base.

Bibliography

[1] M. Alser, H. Hassan, H. Xin, O. Ergin, O. Mutlu, and C. Alkan, "GateKeeper: a new hardware architecture for accelerating pre-alignment in DNA short read mapping," *Bioinformatics*, vol. 33, pp. 3355–3363, Nov. 2017.

[2] M. Flores, G. Glusman, K. Brogaard, N. D. Price, and L. Hood, "P4 medicine: how systems medicine will transform the healthcare sector and society," *Personalized medicine*, vol. 10, no. 6, pp. 565–576, 2013.

[3] W. W. Soon, M. Hariharan, and M. P. Snyder, "High-throughput sequencing for biology and medicine," *Molecular systems biology*, vol. 9, no. 1, 2013.

[4] H. Xin, D. Lee, F. Hormozdiari, S. Yedkar, O. Mutlu, and C. Alkan, "Accelerating read mapping with FastHASH," in *BMC genomics*, vol. 14, p. S13, BioMed Central, 2013.

[5] J. Arram, T. Kaplan, W. Luk, and P. Jiang, "Leveraging FPGAs for accelerating short read alignment," *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, vol. 14, no. 3, pp. 668–677, 2017.

[6] Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, C. Zhai, M. J. Efron, R. Iyer, M. C. Schatz, S. Sinha, and G. E. Robinson, "Big data: astronomical or genomical?," *PLoS biology*, vol. 13, no. 7, p. e1002195, 2015.

[7] C. Firtina, J. S. Kim, M. Alser, D. S. Cali, A. E. Cicek, C. Alkan, and O. Mutlu, "Apollo: A Sequencing-Technology-Independent, Scalable, and Accurate Assembly Polishing Algorithm," *arXiv preprint arXiv:1902.04341*, 2019.

[8] M. Ruffalo, T. LaFramboise, and M. Koyutürk, "Comparative analysis of algorithms for next-generation sequencing read alignment," *Bioinformatics*, vol. 27, no. 20, pp. 2790–2796, 2011.

[9] E. J. Fox, K. S. Reid-Bayliss, M. J. Emond, and L. A. Loeb, "Accuracy of next generation sequencing platforms," *Next generation, sequencing & applications*, vol. 1, 2014.

[10] M. A. Quail, M. Smith, P. Coupland, T. D. Otto, S. R. Harris, T. R. Connor, A. Bertoni, H. P. Swerdlow, and Y. Gu, "A tale of three next generation sequencing platforms: comparison of Ion Torrent, Pacific Biosciences and Illumina MiSeq sequencers," *BMC genomics*, vol. 13, no. 1, p. 341, 2012.

[11] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of molecular biology*, vol. 215, no. 3, pp. 403–410, 1990.

[12] N. Ahmed, K. Bertels, and Z. Al-Ars, "A comparison of seed-and-extend techniques in modern DNA read alignment algorithms," in *2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pp. 1421–1428, IEEE, 2016.

[13] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of molecular biology*, vol. 48, no. 3, pp. 443–453, 1970.

[14] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *J Mol Biol*, vol. 147, pp. 195–197, Mar 1981.

[15] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, pp. 707–710, 1966.

[16] S. Mittal and J. S. Vetter, "A survey of CPU-GPU heterogeneous computing techniques," *ACM Computing Surveys (CSUR)*, vol. 47, no. 4, pp. 1–35, 2015.

[17] C. Alkan, J. M. Kidd, T. Marques-Bonet, G. Aksay, F. Antonacci, F. Hormozdiari, J. O. Kitzman, C. Baker, M. Malig, O. Mutlu, *et al.*, "Personalized copy number and segmental duplication maps using next-generation sequencing," *Nature genetics*, vol. 41, no. 10, p. 1061, 2009.

[18] H. Xin, J. Greth, J. Emmons, G. Pekhimenko, C. Kingsford, C. Alkan, and O. Mutlu, "Shifted Hamming distance: a fast and accurate SIMD-friendly filter to accelerate alignment verification in read mapping," *Bioinformatics*, vol. 31, no. 10, pp. 1553–1560, 2015.

[19] M. Harris, "How to Optimize Data Transfers in CUDA C/C ." https://devblogs.nvidia.com/how-optimize-data-transfers-cuda-cc/, May 2018.

[20] M. Harris, "Unified Memory in CUDA 6." https://devblogs.nvidia.com/unified-memory-in-cuda-6/, Apr 2018.

[21] S. Chien, I. Peng, and S. Markidis, "Performance Evaluation of Advanced Features in CUDA Unified Memory," in *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, pp. 50–57, IEEE, 2019.

[22] M. Šošić and M. Šikić, "Edlib: a C/C++ library for fast, exact sequence alignment using edit distance," *Bioinformatics*, vol. 33, no. 9, pp. 1394–1395, 2017.

[23] G. Myers, "A fast bit-vector algorithm for approximate string matching based on dynamic programming," *Journal of the ACM (JACM)*, vol. 46, no. 3, pp. 395–415, 1999.

[24] M. Alser, O. Mutlu, and C. Alkan, "MAGNET: understanding and improving the accuracy of genome pre-alignment filtering," *arXiv preprint arXiv:1707.01631*, 2017.

[25] M. Alser, H. Hassan, A. Kumar, O. Mutlu, and C. Alkan, "Shouji: a fast and efficient pre-alignment filter for sequence alignment," *Bioinformatics*, vol. 35, no. 21, pp. 4255–4263, 2019.

[26] M. Alser, T. Shahroodi, J. Gomez-Luna, C. Alkan, and O. Mutlu, "SneakySnake: A Fast and Accurate Universal Genome Pre-Alignment Filter for CPUs, GPUs, and FPGAs," *arXiv preprint arXiv:1910.09020*, 2019.

[27] J. Lee, N. Bose, and F. Hwang, "Use of Steiner's problem in suboptimal routing in rectilinear metric," *IEEE Transactions on Circuits and Systems*, vol. 23, no. 7, pp. 470–476, 1976.

[28] NVIDIA, "CUDA C Programming Guide." https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2020.

[29] 1000 Genomes Project Consortium, "An integrated map of genetic variation from 1,092 human genomes," *Nature*, vol. 491, no. 7422, pp. 56–65, 2012.

[30] M. Holtgrewe, "Mason: a read simulator for second generation sequencing data." http://packages.seqan.de/mason/, 2010.

[31] 1000 Genomes Project Consortium, "A global reference for human genetic variation," *Nature*, vol. 526, no. 7571, pp. 68–74, 2015.

[32] H. Li, "Minimap2: pairwise alignment for nucleotide sequences," *Bioinformatics*, vol. 34, no. 18, pp. 3094–3100, 2018.

[33] NVIDIA, "Achieved Occupancy." https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm, 2015.

[34] NVIDIA, "NVIDIA: nvprof." https://docs.nvidia.com/cuda/profiler-users-guide/index.html, 2019.

Appendix A

Data

A.1 Accuracy of GateKeeper-GPU with respect to Edlib

Tables A.1 to A.4 are represented by Figure 5.1 to Figure 5.4. They show the false accept analysis of GateKeeper-GPU with respect to Edlib's [22] global alignment. Since GateKeeper-GPU directly accepts the sequence pairs that contain the unknown base call character *'N'* (we denote these pairs as undefined), in order to observe the real false accept count, undefined pairs are also made accepted by Edlib in all of the tests in this section. The datasets consist of 30 million read and candidate reference segment pairs. Decrease in number of pairs is the percentage of rejected pairs over entire dataset.

Table A.1: False Accept Analysis Table (100bp) of Figure 5.1

	Edlib		GateKeper-GPU				
Error Threshold	**Accepted**	**Rejected**	**Accepted**	**Rejected**	**False Accept Count**	**False Accepts / Entire Dataset (%)**	**Decrease in Number of Pairs (%)**
0	104,403	29,895,597	104,403	29,895,597	0	0.00	99.65
1	136,979	29,863,021	165,125	29,834,875	28,146	0.09	99.45
2	201,392	29,798,608	336,884	29,663,116	135,492	0.45	98.88
3	299,313	29,700,687	719,228	29,280,772	419,915	1.40	97.60
4	427,108	29,572,892	1,589,521	28,410,479	1,162,413	3.87	94.70
5	583,032	29,416,968	3,091,304	26,908,696	2,508,272	8.36	89.70
6	767,631	29,232,369	6,158,981	23,841,019	5,391,350	17.97	79.47
7	983,575	29,016,425	9,392,090	20,607,910	8,408,515	28.03	68.69
8	1,243,411	28,756,589	12,547,826	17,452,174	11,304,415	37.68	58.17
9	1,561,830	28,438,170	15,002,035	14,997,965	13,440,205	44.80	49.99
10	1,960,522	28,039,478	17,210,612	12,789,388	15,250,090	50.83	42.63

False accept analysis of GateKeeper-GPU with respect to Edlib in *ERR240727_1_E5_30million* (100bp) dataset. The dataset contains mrFAST's candidate pairs for error threshold 5. Undefined pairs (92,414) are included in the number of accepted pairs for both Edlib and GateKeeper-GPU.

Table A.2: False Accept Analysis Table (150bp) of Figure 5.2

	Edlib		GateKeper-GPU				
Error Threshold	**Accepted**	**Rejected**	**Accepted**	**Rejected**	**False Accept Count**	**False Accepts / Entire Dataset (%)**	**Decrease in Number of Pairs (%)**
0	264,061	29,735,939	264,061	29,735,939	0	0.00	99.12
1	339,183	29,660,817	365,369	29,634,631	26,186	0.09	98.78
3	496,836	29,503,164	832,921	29,167,079	336,085	1.12	97.22
4	627,847	29,372,153	1,391,221	28,608,779	763,374	2.54	95.36
6	1,006,666	28,993,334	3,705,826	26,294,174	2,699,160	9.00	87.65
7	1,241,736	28,758,264	5,837,388	24,162,612	4,595,652	15.32	80.54
9	1,755,063	28,244,937	12,940,109	17,059,891	11,185,046	37.28	56.87
10	2,024,813	27,975,187	16,172,680	13,827,320	14,147,867	47.16	46.09
12	2,606,248	27,393,752	20,339,185	9,660,815	17,732,937	59.11	32.20
13	2,938,643	27,061,357	21,536,343	8,463,657	18,597,700	61.99	28.21
15	3,745,005	26,254,995	23,563,237	6,436,763	19,818,232	66.06	21.46

False accept analysis of GateKeeper-GPU with respect to Edlib in *SRR826460_1_E6_30million* (150bp) dataset. The dataset contains mrFAST's candidate pairs for error threshold 6. Undefined pairs (15,141) are included in the number of accepted pairs for both Edlib and GateKeeper-GPU.

Table A.3: False Accept Analysis Table (250bp) of Figure 5.3

	Edlib		GateKeper-GPU				
Error Threshold	**Accepted**	**Rejected**	**Accepted**	**Rejected**	**False Accept Count**	**False Accepts / Entire Dataset (%)**	**Decrease in Number of Pairs (%)**
0	422,817	29,577,183	422,827	29,577,173	10	0.00	98.59
2	467,342	29,532,658	479,872	29,520,128	12,530	0.04	98.40
5	498,223	29,501,777	587,918	29,412,082	89,695	0.30	98.04
7	524,364	29,475,636	793,255	29,206,745	268,891	0.90	97.36
10	584,475	29,415,525	1,581,379	28,418,621	996,904	3.32	94.73
12	636,191	29,363,809	3,464,025	26,535,975	2,827,834	9.43	88.45
15	725,455	29,274,545	9,177,446	20,822,554	8,451,991	28.17	69.41
17	788,472	29,211,528	13,113,437	16,886,563	12,324,965	41.08	56.29
20	885,488	29,114,512	17,887,612	12,112,388	17,002,124	56.67	40.37
22	950,913	29,049,087	21,548,632	8,451,368	20,597,719	68.66	28.17
25	1,051,100	28,948,900	26,581,310	3,418,690	25,530,210	85.10	11.40

False accept analysis of GateKeeper-GPU with respect to Edlib in *SRR826471_1_E12_30million* (250bp) dataset. The dataset contains mrFAST's candidate pairs for error threshold 12. Undefined pairs (379,262) are included in the number of accepted pairs for both Edlib and GateKeeper-GPU.

Table A.4: False Accept Analysis Table (100bp) of Figure 5.4

	Edlib		GateKeper-GPU				
Error Threshold	**Accepted**	**Rejected**	**Accepted**	**Rejected**	**False Accept Count**	**False Accepts / Entire Dataset (%)**	**Decrease in Number of Pairs (%)**
0	813,961	29,186,039	813,961	29,186,039	0	0.00	97.29
1	1,020,992	28,979,008	1,080,900	28,919,100	60,351	0.20	96.40
2	1,167,931	28,832,069	1,333,047	28,666,953	165,328	0.55	95.56
3	1,309,257	28,690,743	1,709,395	28,290,605	398,621	1.33	94.30
4	1,463,733	28,536,267	2,333,802	27,666,198	869,491	2.90	92.22
5	1,638,308	28,361,692	3,346,596	26,653,404	1,705,665	5.69	88.84
6	1,836,191	28,163,809	4,844,286	25,155,714	2,999,806	10.00	83.85
7	2,057,082	27,942,918	6,703,603	23,296,397	4,635,784	15.45	77.65
8	2,303,071	27,696,929	9,005,034	20,994,966	6,682,829	22.28	69.98
9	2,572,631	27,427,369	11,382,399	18,617,601	8,784,715	29.28	62.06
10	2,873,114	27,126,886	14,001,878	15,998,122	11,090,571	36.97	53.33

False accept analysis of GateKeeper-GPU with respect to Edlib in *minimap2_ERR240727_1_e* (100bp) dataset. The dataset contains minimap2's candidate pairs for error threshold e by processing *ERR240727_1* reads. Undefined pairs (26,759) are included in the number of accepted pairs for both Edlib and GateKeeper-GPU.

A.2 Comparison with Other Pre-alignment Filters

Tables A.5 to A.10 show the false accept comparison of GateKeeper-GPU against GateKeeper [1] (denoted as 'GateKeeper-FPGA'), SHD [18], MAGNET [24], Shouji [25] and SneakySnake [26] with respect to Edlib's [22] global alignment, represented by Figure 5.5 to Figure 5.10. Since the other filters do not distinguish the sequence pairs that contain the unknown base call character *'N'* (we denote these pairs as undefined), in order to make a fair comparison, we include undefined pairs in GateKeeper-GPU's false accept count. The datasets consist of 30 million read and candidate reference segment pairs.

Table A.5: False Accept Comparison Table (100bp) of Figure 5.5

Error Threshold	GateKeeper-GPU	GateKeeper-FPGA	SHD	Shouji	MAGNET	SneakySnake
0	28,009 (0)	0	10	0	963,941	0
1	672,164 (644,200)	783,185	783,185	333,320	800,099	12,473
2	2,290,693 (2,263,036)	2,704,128	2,704,128	1,283,004	1,876,518	77,165
3	4,324,420 (4,297,528)	5,237,529	5,237,529	2,674,876	2,428,301	234,003
4	6,744,070 (6,718,357)	8,231,507	8,231,507	4,399,886	2,662,902	484,179
5	9,354,269 (9,330,017)	11,195,124	11,195,124	6,452,280	2,916,838	795,582
6	12,092,022 (12,069,082)	13,781,651	13,781,651	9,373,309	3,406,303	1,240,276
7	13,085,652 (13,064,066)	14,283,519	14,283,519	11,113,616	4,026,433	1,815,478
8	13,139,626 (13,119,339)	13,814,295	13,814,295	11,990,529	4,745,672	2,567,290
9	12,264,194 (12,245,362)	13,105,305	13,105,305	11,693,396	5,319,627	3,331,944
10	10,929,703 (10,912,255)	11,389,103	11,389,103	10,664,722	5,673,172	4,020,164

False accept comparison between pre-alignment tools for dataset *ERR240727_1_E2_30million* (100bp). The dataset contains mrFAST's candidate pairs for error threshold 2 by processing *ERR240727_1* reads, with 28,009 undefined pairs. GateKeeper-GPU values show false accepts including undefined pairs outside of parenthesis and excluding undefined pairs inside of parenthesis. The figure (Figure 5.5) was drawn with false accept counts including undefined pairs that are outside of parenthesis. The values for other filters were retrieved from Shouji's supplementary material.

Table A.6: False Accept Comparison Table (100bp) of Figure 5.6

Error Threshold	GateKeeper-GPU	GateKeeper-FPGA	SHD	Shouji	MAGNET	SneakySnake
0	31,487 (0)	0	0	0	7	0
1	31,501 (14)	14	14	2	5	0
2	31,767 (280)	155	155	15	2	0
3	32,689 (1,202)	1,196	1,196	216	4	1
4	40,692 (9,205)	7,436	7,436	1,986	13	3
5	71,158 (39,671)	32,792	32,792	10,551	82	13
6	193,539 (162,052)	155,134	155,134	57,258	298	69
7	435,611 (404,124)	417,444	417,444	214,005	1,030	289
8	951,114 (919,627)	1,031,480	1,031,480	675,029	3,129	1,081
9	1,943,019 (1,911,532)	29,997,022	29,997,022	1,742,476	8,234	3,563
10	3,710,604 (3,679,117)	29,998,373	29,998,373	3,902,535	19,013	9,698

False accept comparison between pre-alignment tools for dataset *ERR240727_1_E40_30million* (100bp). The dataset contains mrFAST's candidate pairs for error threshold 40 by processing *ERR240727_1* reads, with 31,487 undefined pairs. GateKeeper-GPU values show false accepts including undefined pairs outside of parenthesis and excluding undefined pairs inside of parenthesis. The figure (Figure 5.6) was drawn with false accept counts including undefined pairs that are outside of parenthesis. The values for other filters were retrieved from Shouji's supplementary material.

Table A.7: False Accept Comparison Table (150bp) of Figure 5.7

Error Threshold	GateKeeper-GPU	GateKeeper-FPGA	SHD	Shouji	MAGNET
0	30,142 (0)	0	0	0	428,412
1	171,256 (141,961)	173,573	173,573	113,519	156,891
3	1,632,544 (1,603,900)	2,080,279	2,080,279	1,539,365	725,873
4	3,118,355 (3,090,152)	4,023,762	4,023,762	3,042,831	1,064,344
6	6,681,929 (6,654,933)	9,258,602	9,258,602	6,025,592	1,430,272
7	9,016,979 (8,990,561)	12,481,853	12,481,853	8,219,336	1,532,024
9	15,109,160 (15,083,838)	22,076,837	22,076,837	14,568,337	1,874,734
10	17,023,658 (16,998,826)	21,341,979	21,341,979	16,920,389	2,194,275
12	18,335,496 (18,311,754)	19,868,151	19,868,151	18,270,597	3,294,672
13	18,145,432 (18,122,415)	19,082,528	19,082,528	18,095,207	4,066,617
15	16,953,324 (16,932,083)	17,353,835	17,353,835	16,993,568	5,810,797

False accept comparison between pre-alignment tools for dataset *SRR826460_1_E4_30million* (150bp). The dataset contains mrFAST's candidate pairs for error threshold 4 by processing *SRR826460_1* reads, with 30,142 undefined pairs. GateKeeper-GPU values show false accepts including undefined pairs outside of parenthesis and excluding undefined pairs inside of parenthesis. The figure (Figure 5.7) was drawn with false accept counts including undefined pairs that are outside of parenthesis. The values for other filters were retrieved from Shouji's supplementary material.

Table A.8: False Accept Comparison Table (150bp) of Figure 5.8

Error Threshold	GateKeeper-GPU	GateKeeper-FPGA	SHD	Shouji	MAGNET
0	309 (0)	0	0	0	126
1	365 (58)	58	58	43	42
3	407 (100)	90	90	83	35
4	573 (266)	267	267	137	28
6	13,606 (13,299)	18,110	18,110	6,259	25
7	64,840 (64,533)	79,418	79,418	27,092	27
9	564,241 (563,934)	29,698,666	29,698,666	404,742	108
10	1,049,599 (1,049,292)	29,999,388	29,999,388	935,486	231
12	2,490,712 (2,490,405)	29,999,290	29,999,290	2,514,950	965
13	3,677,914 (3,677,607)	29,999,204	29,999,204	3,693,298	2,018
15	7,692,574 (7,692,267)	29,998,847	29,998,847	8,034,737	8,448

False accept comparison between pre-alignment tools for dataset *SRR826460_1_E70_30million* (150bp). The dataset contains mrFAST's candidate pairs for error threshold 70 by processing *SRR826460_1* reads, with 309 undefined pairs. GateKeeper-GPU values show false accepts including undefined pairs outside of parenthesis and excluding undefined pairs inside of parenthesis. The figure (Figure 5.8) was drawn with false accept counts including undefined pairs that are outside of parenthesis. The values for other filters were retrieved from Shouji's supplementary material.

Table A.9: False Accept Comparison Table (250bp) of Figure 5.9

Error Threshold	GateKeeper-GPU	GateKeeper-FPGA	SHD	Shouji	MAGNET	SneakySnake
0	35,075 (3)	0	0	0	479,104	0
2	250,322 (215,613)	238,368	238,368	174,366	143,066	12,319
5	1,242,873 (1,208,633)	1,546,126	1,546,126	1,071,218	226,864	38,814
7	3,113,200 (3,079,257)	3,933,916	3,933,916	2,775,419	347,819	79,246
10	7,283,863 (7,250,529)	26,816,729	26,816,729	6,669,084	624,927	235,689
12	12,260,108 (12,227,208)	26,137,224	26,137,224	11,147,373	825,468	407,799
15	19,039,913 (19,007,867)	25,084,654	25,084,654	18,406,823	1,066,633	705,904
17	21,308,177 (21,276,706)	24,449,131	24,449,131	20,971,826	1,235,999	914,730
20	22,311,079 (22,280,323)	23,595,168	23,595,168	22,223,170	1,695,351	1,364,891
22	22,311,569 (22,281,259)	23,040,384	23,040,384	22,271,215	2,241,984	1,879,428
25	21,843,548 (21,813,824)	22,142,250	22,142,250	21,849,454	3,514,515	3,134,474

False accept comparison between pre-alignment tools for dataset *SRR826471_1_E8_30million* (250bp). The dataset contains mrFAST's candidate pairs for error threshold 8 by processing *SRR826471_1* reads, with 35,072 undefined pairs. GateKeeper-GPU values show false accepts including undefined pairs outside of parenthesis and excluding undefined pairs inside of parenthesis. The figure (Figure 5.9) was drawn with false accept counts including undefined pairs that are outside of parenthesis. The values for other filters were retrieved from Shouji's supplementary material.

Table A.10: False Accept Comparison Table (250bp) of Figure 5.10

Error Threshold	GateKeeper-GPU	GateKeeper-FPGA	SHD	Shouji	MAGNET	SneakySnake
0	4,763,683 (1)	0	0	0	53	0
2	4,763,696 (49)	71	71	55	44	2
5	4,763,688 (102)	249	249	161	49	6
7	4,763,704 (152)	698	698	212	48	6
10	4,771,455 (7,953)	29,999,528	29,999,528	5,627	42	14
12	4,839,211 (75,739)	29,999,480	29,999,480	64,225	45	22
15	5,481,110 (717,669)	29,999,425	29,999,425	775,314	82	47
17	6,545,084 (1,781,675)	29,999,377	29,999,377	2,052,498	175	106
20	9,894,411 (5,131,063)	29,999,282	29,999,282	5,679,869	417	326
22	14,252,812 (9,489,566)	29,999,158	29,999,158	10,277,297	593	495
25	21,963,183 (17,200,145)	29,998,867	29,998,867	19,676,652	1,174	955

False accept comparison between pre-alignment tools for dataset *SRR826471_1_E100_30million* (250bp). The dataset contains mrFAST's candidate pairs for error threshold 100 by processing *SRR826471_1* reads, with 4,763,682 undefined pairs. GateKeeper-GPU values show false accepts including undefined pairs outside of parenthesis and excluding undefined pairs inside of parenthesis. The figure (Figure 5.10) was drawn with false accept counts including undefined pairs that are outside of parenthesis. The values for other filters were retrieved from Shouji's supplementary material.

A.3 Filtering Throughput Analysis

For calculating the filtering thoughput of GateKeeper-GPU in terms of number of pairs in 40 minutes, we first record the kernel time and filter time taken for filtering of 30,000,000 pairs and report them in the tables from A.11 to A.16 as unprocessed data. Then, we calculate the number of pairs filtered per second and in 40 minutes using these values.

Table A.11: 100bp Kernel and Filter Time for Table 5.1

		GateKeeper-CPU		Device-encoded		Host-Encoded	
	Error Threshold	**1 Core**	**12 Cores**	**1 GPU**	**8 GPUs**	**1 GPU**	**8 GPUs**
kt	**2**	102.52	10.04	0.29	0.06	0.15	0.02
	5	194.13	18.49	0.48	0.07	0.29	0.04
ft	**2**	117.14	11.09	9.40	1.83	24.36	5.02
	5	204.35	19.55	9.50	1.90	24.56	5.06

Kernel time (kt) and filter time (kt) in seconds for filtering 30,000,000 pairs in Setting1. Multi-GPU values represent the time of the device that has longest value.

Table A.12: 150bp Kernel and Filter Time for Table 5.2

		GateKeeper-CPU		Device-encoded		Host-Encoded	
	Error Threshold	**1 Core**	**12 Cores**	**1 GPU**	**8 GPUs**	**1 GPU**	**8 GPUs**
kt	**4**	274.39	24.90	0.81	0.15	0.35	0.07
	10	577.29	52.54	1.16	0.18	0.73	0.12
ft	**4**	289.96	26.37	13.74	2.82	44.14	7.73
	10	592.86	54.05	14.12	2.72	44.27	7.86

Kernel time (kt) and filter time (kt) in seconds for filtering 30,000,000 pairs in Setting1. Multi-GPU values represent the time of the device that has longest value.

Table A.13: 250bp Kernel and Filter Time for Table 5.3

		GateKeeper-CPU		Device-encoded		Host-Encoded	
	Error Threshold	**1 Core**	**12 Cores**	**1 GPU**	**8 GPUs**	**1 GPU**	**8 GPUs**
kt	**6**	575.92	53.72	1.57	0.27	0.74	0.12
	10	885.18	82.17	1.87	0.29	1.17	0.22
ft	**6**	601.32	56.06	21.78	4.24	69.43	11.16
	10	910.18	84.54	22.06	4.11	69.97	11.33

Kernel time (kt) and filter time (kt) in seconds for filtering 30,000,000 pairs in Setting1. Multi-GPU values represent the time of the device that has longest value.

Table A.14: 100bp Kernel and Filter Time for Table 5.4

		GateKeeper-CPU		Device-encoded	Host-Encoded
	Error Threshold	**1 Core**	**12 Cores**	**Single GPU**	**Single GPU**
kt	**2**	110.70	13.21	1.75	1.00
	5	211.67	24.39	2.47	1.72
ft	**2**	121.47	14.73	11.74	26.49
	5	222.43	25.89	12.72	27.15

Kernel time (kt) and filter time (kt) in seconds for filtering 30,000,000 pairs in Setting2.

Table A.15: 150bp Kernel and Filter Time for Table 5.5

		GateKeeper-CPU		Device-encoded	Host-Encoded
	Error Threshold	**1 Core**	**12 Cores**	**Single GPU**	**Single GPU**
kt	**4**	262.07	28.49	3.84	2.21
	10	552.53	59.80	6.01	4.36
ft	**4**	278.62	30.71	18.16	43.04
	10	569.17	62.00	20.41	45.23

Kernel time (kt) and filter time (kt) in seconds for filtering 30,000,000 pairs in Setting2.

Table A.16: 250bp Kernel and Filter Time for Table 5.6

		GateKeeper-CPU		Device-encoded	Host-Encoded
	Error Threshold	**1 Core**	**12 Cores**	**Single GPU**	**Single GPU**
kt	**6**	558.09	64.87	4.55	4.55
	10	862.35	98.61	6.01	6.74
ft	**6**	583.10	68.46	18.87	71.42
	10	887.43	102.23	20.41	73.86

Kernel time (kt) and filter time (kt) in seconds for filtering 30,000,000 pairs in Setting2.

A.3.1 Effect of Error Threshold on Filter Time

Table A.17 contains the data for evaluating the effect of increasing error threshold on the performance of single GPU GateKeeper-GPU and 12-core GateKeeper-CPU. The data is illustrated by Figure 5.11.

Table A.17: Effect of Increasing Error Threshold for Figure 5.11

	Setting1			Setting2		
Error Threshold	**12-core CPU**	**Device-encoded GPU**	**Host-encoded GPU**	**12-core CPU**	**Device-encoded GPU**	**Host-encoded GPU**
0	12.18	22.10	73.99	14.09	17.23	69.29
1	21.32	23.84	68.85	25.89	16.99	68.74
2	28.22	22.03	68.77	34.39	17.46	69.28
4	41.72	21.27	69.31	51.46	18.16	70.49
6	56.06	21.78	69.43	68.46	18.87	71.42
8	70.25	21.61	69.59	85.42	20.40	72.76
10	84.54	22.06	69.97	102.23	20.41	73.86

Effect of increasing error threshold in multi-core GateKeeper-CPU and single GPU GateKeeper-GPU in terms of filter time (seconds). 250bp pairs were used for the tests.

A.3.2 Effect of Encoding Actor on Filtering Throughput

Tables from A.18 to A.20 contain the data represented by Figure 5.12 to Figure 5.14 for evaluating the effect of encoding actor on the performance of GateKeeper-GPU.

Table A.18: Effect of Encoding Actor on Filtering Throughput in 100bp for Figure 5.13

	Setting1				Setting2			
	Device-encoded		**Host-encoded**		**Device-encoded**		**Host-encoded**	
Error Threshold	**Kernel**	**Filter**	**Kernel**	**Filter**	**Kernel**	**Filter**	**Kernel**	**Filter**
0	110.1	3.2	699.7	1.2	25.9	2.7	83.1	1.2
1	113.2	3.2	282.6	1.2	19.9	2.6	39.9	1.1
2	102.0	3.2	198.7	1.2	17.1	2.6	30.1	1.1
3	91.6	3.2	149.7	1.2	15.0	2.5	24.2	1.1
4	72.5	3.2	122.5	1.2	13.4	2.4	20.3	1.1
5	62.8	3.2	103.9	1.2	12.1	2.4	17.5	1.1
6	57.0	3.2	89.7	1.2	11.0	2.4	15.3	1.1

Effect of encoding actor (device or host) on filtering throughput of single-GPU GateKeeper-GPU with increasing error threshold for read length 100bp in Setting1 and Setting2. The values represent the number of filtrations in terms of millions per second, produced by GateKeeper-GPU with respect to kernel or filter time.

Table A.19: Effect of Encoding Actor on Filtering Throughput in 150bp for Figure 5.13

	Setting1				Setting2			
	Device-encoded		Host-encoded		Device-encoded		Host-encoded	
Error Threshold	**Kernel**	**Filter**	**Kernel**	**Filter**	**Kernel**	**Filter**	**Kernel**	**Filter**
0	28.9	2.1	537.7	0.6	11.2	1.7	56.5	0.7
1	32.8	2.0	193.5	0.7	10.7	1.8	26.8	0.7
2	35.6	2.2	138.6	0.7	9.6	1.7	20.2	0.7
4	37.1	2.2	85.5	0.7	7.8	1.7	13.6	0.7
6	34.9	2.2	62.1	0.7	6.6	1.6	10.1	0.7
8	30.5	2.1	48.4	0.7	5.7	1.5	8.2	0.7
10	25.8	2.1	40.8	0.7	5.0	1.5	6.9	0.7

Effect of encoding actor (device or host) on filtering throughput of single-GPU GateKeeper-GPU with increasing error threshold for read length 150bp in Setting1 and Setting2. The values represent the number of filtrations in terms of millions per second, produced by GateKeeper-GPU with respect to kernel or filter time.

Table A.20: Effect of Encoding Actor on Filtering Throughput in 250bp for Figure 5.14

	Setting1				Setting2			
	Device-encoded		Host-encoded		Device-encoded		Host-encoded	
Error Threshold	**Kernel**	**Filter**	**Kernel**	**Filter**	**Kernel**	**Filter**	**Kernel**	**Filter**
0	15.3	1.4	328.2	0.4	11.2	1.7	36.6	0.4
1	16.3	1.3	132.2	0.4	10.7	1.8	17.2	0.4
2	17.0	1.4	92.0	0.4	9.6	1.7	12.8	0.4
4	18.5	1.4	56.5	0.4	7.8	1.7	8.6	0.4
6	19.1	1.4	40.6	0.4	6.6	1.6	6.6	0.4
8	17.8	1.4	31.2	0.4	5.7	1.5	5.3	0.4
10	16.0	1.4	25.7	0.4	5.0	1.5	4.5	0.4

Effect of encoding actor (device or host) on filtering throughput of single-GPU GateKeeper-GPU with increasing error threshold for read length 250bp in Setting1 and Setting2. The values represent the number of filtrations in terms of millions per second, produced by GateKeeper-GPU with respect to kernel or filter time.

A.3.3 Effect of Read Length on Filtering Throughput

Table A.21 represents the data depicted by Figure 5.15 for evaluating the effect of different sequence lengths on the performance of GateKeeper-GPU.

Table A.21: Effect of Read Length on Filtering Throughput for Figure 5.15

		Setting1		Setting2	
E	**Read Length**	**Device-encoded**	**Host-encoded**	**Device-encoded**	**Host-encoded**
0	**100bp**	3.16	1.18	2.73	1.18
	150bp	2.14	0.64	1.74	0.71
	250bp	1.36	0.41	1.74	0.43
4	**100bp**	3.16	1.23	2.43	1.11
	150bp	2.18	0.68	1.65	0.70
	250bp	1.41	0.43	1.65	0.43

Effect of sequence length on single-GPU GateKeeper-GPU's filtering throughput in terms of millions of filtrations per second, with error thresholds (E) 0 and 4. The values were calculated with respect to filter time.

A.3.4 Multi-GPU Filtering Throughput Analysis

In order to see how GateKeeper-GPU's performance scales with increasing the number of GPGPU devices, we perform tests with 8 GPUs in Setting1 and report the results in Table A.22 to A.24 which are illustrated by Figure 5.16.

Table A.22: Multi-GPU Filtering Throughput Analysis on 100bp for Figure 5.16 (a)

wrt.	**Kernel Time**		**Filter Time**	
Number of GPU Devices	**Device-encoded**	**Host-encoded**	**Device-encoded**	**Host-encoded**
1	102	199	3	1
2	201	388	6	2
3	300	542	8	3
4	364	704	10	4
5	376	877	12	5
6	488	1,062	14	5
7	487	1,171	15	6
8	496	1,333	16	6

Multi-GPU filtering throughput of GateKeeper-GPU in terms of millions of filtrations per second, with respect to filter time and kernel time, in Setting1. The error threshold is 2.

Table A.23: Multi-GPU Filtering Throughput Analysis on 150bp for Figure 5.16 (b)

wrt.	Kernel Time		Filter Time	
Number of GPU Devices	**Device-encoded**	**Host-encoded**	**Device-encoded**	**Host-encoded**
1	37	85	2	1
2	73	165	4	1
3	109	229	5	2
4	144	284	7	2
5	166	349	8	3
6	196	408	10	3
7	211	437	10	4
8	207	426	11	4

Multi-GPU filtering throughput of GateKeeper-GPU in terms of millions of filtrations per second, with respect to filter time and kernel time, in Setting1. The error threshold is 4.

Table A.24: Multi-GPU Filtering Throughput Analysis on 250bp for Figure 5.16 (c)

wrt.	Kernel Time		Filter Time	
Number of GPU Devices	**Device-encoded**	**Host-encoded**	**Device-encoded**	**Host-encoded**
1	18	31	1	0
2	35	50	2	1
3	52	74	4	1
4	69	94	4	2
5	81	118	5	2
6	93	97	6	2
7	101	139	7	2
8	101	152	7	3

Multi-GPU filtering throughput of GateKeeper-GPU in terms of millions of filtrations per second, with respect to filter time and kernel time, in Setting1. The error threshold is 8.