Shepard: A Fast Exact Match Short Read Aligner

Chad Nelson, Kevin Townsend, Bhavani Satyanarayana Rao, Phillip Jones, Joseph Zambreno
Department of Electrical and Computer Engineering
Iowa State University
Ames, IA, USA
{cnel711, ktown, bhavani, phjones, zambreno}@iastate.edu

***Abstract*—The mapping of many short sequences of DNA, called reads, to a long reference genome is an common task in molecular biology. The task amounts to a simple string search, allowing for a few mismatches due to mutations and inexact read quality. While existing solutions attempt to align a high percentage of the reads using small memory footprints, Shepard is concerned with only exact matches and speed. Using the human genome, Shepard is on the order of hundreds of thousands of times faster than current software implementations such as SOAP2 or Bowtie, and about 60 times faster than GPU implementations such as SOAP3.**

Shepard contains two components: a software program to preprocess a reference genome into a hash table, and a hardware pipeline for performing fast lookups. The hash table has one entry for each unique 100 base pair sequence that occurs in the reference genome, and contains the index of last occurrence and the number of occurrences. To reduce the hash table size, a minimal perfect hash table is used. The hardware pipeline was designed to perform hash table lookups very quickly, on the order of 600 million lookups per second, and was implemented on a Convey HC-1 high performance reconfigurable computing system. Shepard streams all of the short reads through a custom hardware pipeline and writes the alignment data (index of last occurrence and number of occurrences) to a binary results array.

***Keywords*-FPGA-based Short Read Aligner; Minimal Perfect Hash; Hardware Hash Function**

I. Introduction

We are in a golden era of DNA research. After the first human genome was sequenced in 2003 [1], the next generation of sequencing technologies were developed with higher throughput and lower costs. The machines operate by breaking a person's genome into smaller fragments, or reads. Since humans share 99.9% of their DNA, a reference genome is used to help align the reads in the correct order. Many reads, copies of the same genome, are aligned to improve the quality of the process. These techniques have produced a massive amount of data needing alignment.

The 2012 MEMOCODE design contest [2] was focused on the exact matching of 100 base pair short reads to a 3.1 billion base pair human genome. The contest made some simplifications, presumably to allow teams to focus on creating a fast solution. Such simplifications included:

- Performing only exact matches
- Ignoring quality data
- Using a packed-binary representation of the reads
- Short reads always had the same length (in this case, 100 base pairs)

Teams were provided with approximately one month to develop solutions and were encouraged to use both field programmable gate arrays (FPGAs) and GPUs. The primary and secondary contest categories were speed and cost-performance. Our first place solution optimized for memory bandwidth; we chose hash table lookups because they minimized overheads in memory bandwidth and we chose an FPGA implementation using the Convey HC-1 because of its large available memory bandwidth of 80 GB/sec.

The remainder of this paper is organized as follows. Section II discusses prior work on short read alignment programs. Section III discusses our approach of using minimal perfect hashing and how we preprocess the reference genome. Section IV discusses the actual implementation of the hardware pipeline on the Convey HC-1. Section V shows our results.

II. Related Work

Early solutions for DNA sequence alignment initially used a database of common sequences or a hash table to help align new data. BLAST [3] was one of the first tools available. MAQ [4] was an implementation that provided quality data along with its alignment results. Unfortunately, MAQ become obsolete because of its slow alignment speed when the new generation of sequencing machines became available.

A newer approach, used in both SOAP2 [5] and Bowtie [6], is to index a reference genome using an FM Index [7] and compress using a Bowler-Wheeler transformation [8]. This scheme allows the genome to be compressed, reducing the memory footprint, and allowing use of commodity hardware. Unlike a hash table approach, indexing in this way creates a solution where getting the alignment data is the result of a pointer-based tree transversal. In cases of a mismatch, time-consuming backtracking is used to find segments that may match with high probability. Applications like SOAP2 and Bowtie differ slightly in the way that they construct their index of the reference genome, but are both limited by memory bandwidth because of the tree traversal. SOAP3 [9] uses the same approach, but uses the increased memory bandwidth of GPUs to increase speed.

Our approach differs from prior implementations in a number of important ways. Shepard ignores considerations such as memory usage and alignment quality. By preprocessing the reference genome into a hash table, alignment of arbitrary reads is a simple hash table lookup.

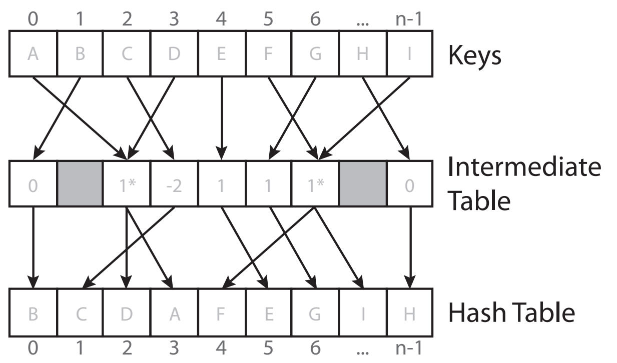

Fig. 1. An example of a Minimal Perfect Hash (MPH) table. Each key is initially hashed once to retrieve a few bits of information stored in an intermediate table. The key is then either rehashed with the new seed from the intermediate table, or the offset from the intermediate table is added to the initial index. The MPH is created by choosing values in the intermediate table so that a given set of keys will not collide.

III. Approach

The problem with a hash table approach is the size of the table. The table size is proportional to the size of the reference genome since there would be one entry per unique 100 base pair sequence in the reference. We used a reference genome from the 1000 genomes project [10] which contained about 2.8 billion unique 100 base pair segments. This means that every byte of data per entry increases the size of the hash table by 2.8 GB. If we were to store the key (the short read) in the hash table, we would need about 32 bytes per entry, or a table size of over 80 GB. In addition, any collisions would necessitate a linked list type of structure for collision handling, resulting in more increases in the size of the table.

A. Minimal Perfect Hash Function

Shepard's solution to the large size of hash tables is to use a minimal perfect hash. A minimal perfect hash is one with no collisions and no empty slots, but requires a fixed set of keys known in advance. Shepard avoids the need for storing any collision handling in the hash table by using a minimal perfect hash to avoid collisions.

A minimal perfect hash requires storing a few bits per entry. To briefly summarize its use, the minimal perfect hash function Shepard uses requires first looking up a seed or offset value in an intermediate table, then using this seed or offset to compute the actual index into the hash table. This is displayed in Figure 1. The construction of the minimal perfect hash table is discussed briefly below.

The first step is to find all the entries that need to be stored in the hash table. This can be done by hashing each 100 base pair word in the reference genome in a number of rounds, throwing away duplicates and storing collisions for processing in the next round.

With a set of known keys, we can construct a minimal perfect hash using a generalized method called hash and displace. More details can be found in both [11] and [12]. First, all the keys are hashed into buckets. If a key collides with another key, the bucket will contain multiple keys. Buckets are then sorted based on how many keys they contain. Starting with the largest buckets, the keys are reseeded such that they find an empty spot in the hash table. The reseed value is stored in an intermediate table. Once all buckets containing two or more keys have been reseeded, buckets containing just one key are placed into open spaces in the hash table. The offset from their original location is recorded in the intermediate table.

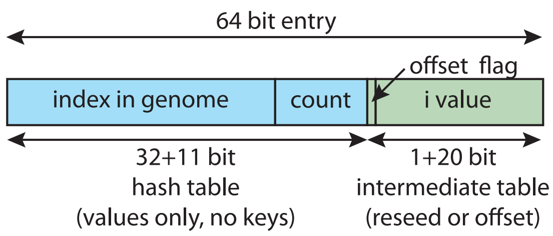

Fig. 2. An entry in the Minimal Perfect Hash table. The table is an array with one 8 byte entry per key. The intermediate table is stored in the 21 least significant bits and is used to find a unique index into the table for each key. The first 43 bits contain the value associated with the key.

The intermediate table provides a minimal perfect hash for the given key set, and can now be used to add values to the table to complete the key-value association.

It should be noted that any short read will give a hash into a minimal perfect hash table, whether it is a member of the hash table or not. A check for existence is needed. Conveniently, Shepard uses the index of last occurrence to check the binary packed form of the reference genome to see if the short read is an actual match to the reference genome. Thus, the need to store the keys in the hash table is unnecessary.

Using a minimal perfect hash function with the reference genome [10] created a 22.5 GB hash table containing 2.8 billion 8-byte entries. If less memory is required, a possible option is the create hash tables for each chromosome rather than the entire reference genome. The layout of each entry is described in Figure 2.

IV. Implementation

The hardware pipeline aligns reads in five stages:

Algorithm 1 Pseudocode of the Shepard Pipeline

```
1: procedure ALIGN($reads, genome, hashtable, results$)
2:     for ($i = 0; i < length(reads); i++$) do
3:         $r \leftarrow reads[i]$;                                ▷ Stage 1
4:         $h \leftarrow hash(r, seed = 0)$;                       ▷ Stage 2
5:         $ivalue \leftarrow intermediateTable[h]$;
6:         if ($ivalue$ is an offset) then                         ▷ Stage 3
7:             $index \leftarrow hash(r, seed = 0) + ivalue$;
8:         else
9:             $index \leftarrow hash(r, seed = ivalue)$;
10:        end if
11:        $entry \leftarrow hashtable[index]$;
12:        $check \leftarrow genome[entry.index]$;                 ▷ Stage 4
13:        if ($r == check$) then                                  ▷ Stage 5
14:            $results[i] \leftarrow entry$;
15:        else
16:            $results[i] \leftarrow NULL$;
17:        end if
18:    end for
19: end procedure
```

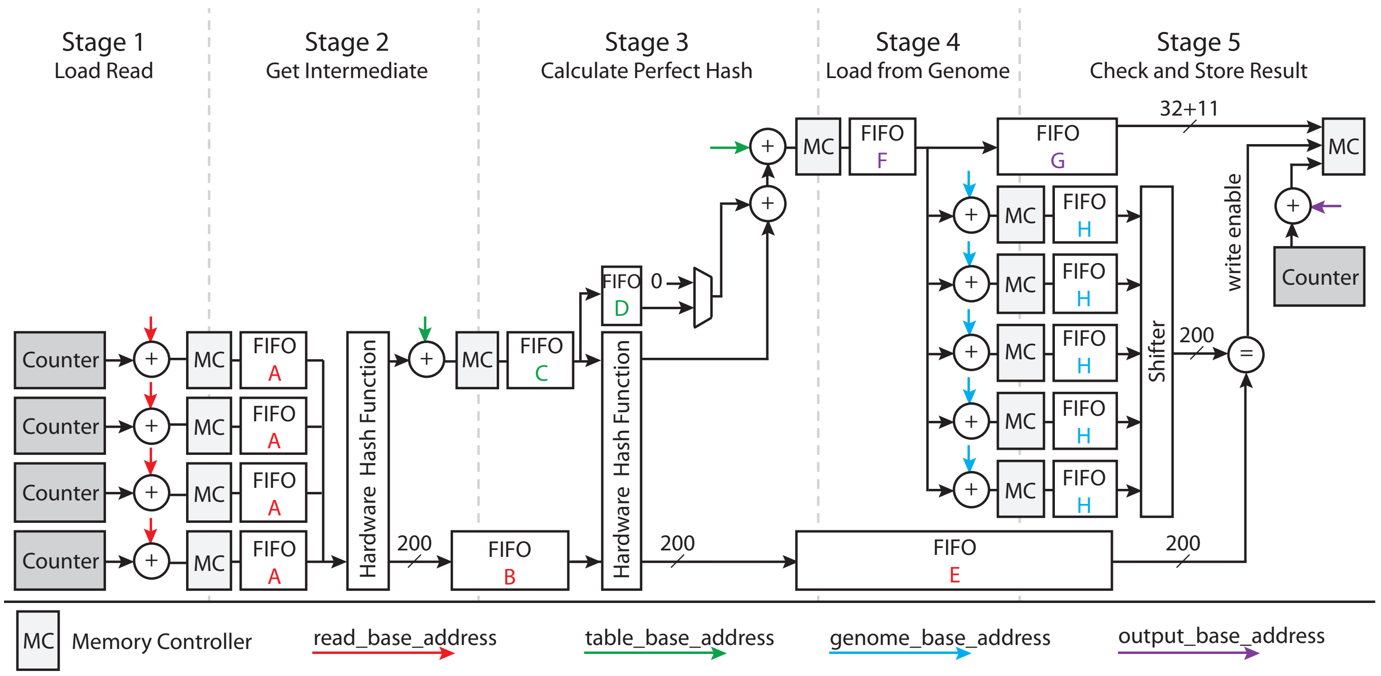

Fig. 3. Shepard's hardware hash table lookup pipeline is broken into five stages, shown here flowing left to right. In stage 1, a bank of four counters and the base address of the read data is used to calculate the address for a given read. These addresses are used to load short reads from four memory controller ports. After a memory latency of about 100 cycles, stage 2 hashes a short read to obtain an index for retrieving a value from the intermediate table. In stage 3, the value from intermediate table allows Shepard to compute an index that is guaranteed not to collide with other entries in the table (see Section III on minimal perfect hashes). The value is either a new seed value for the hash function, or an offset. The unique index is calculated and used to load the value from the hash table. In stage 4, the index in the genome loaded from the hash table is used to load the corresponding 100 base pairs from the reference genome. In stage 5, the reference genome is compared to the read. If they match, the index of occurrence in the genome and number of occurrences are recorded in an output table in memory.

Most of the computation of the algorithm occurs in the hash function. Jenkin's Spooky Hash [13] was chosen because it is both fast in software and easy to implement in hardware due to its reliance on only shifts, adds, and XOR operations. Shepard uses a slimmed down version Jenkins' Spooky Hash that has been stripped to work with only keys of size 25 bytes (100 base pairs). By stripping unnecessary branches and instructions, each hash requires 23 rotations, 23 XORs, 27 additions, and 1 mod operation. By grouping operations together, a 34 stage pipeline hasher was created.

A. *Convey HC-1*

Shepard's hardware pipeline was built for use with the Convey HC-1 platform containing a minimum of 32GB of coprocessor memory. The Convey platform is a hybrid computer, containing a regular motherboard and a coprocessor board that contains is a set of 14 FPGAs. Eight FPGAs are wired as memory controllers (MCs), two are used as an Application Engine Hub (AEH), and the remaining four are programmable and called Application Engines (AEs). The host (x86) processor can send the AEH custom instructions, which will then load a custom bitfile onto the AEs and execute the instruction. The coprocessor contains its own memory, though the host processor and coprocessor can share all memory in a cache coherent manner. More information can be found in Convey's documentation.

The distinct competitive advantage that the Convey HC-1 provides is its actual 80 GB/s of memory bandwidth. In addition, the Convey system has "scatter-gather" DIMMs, allowing random access to memory locations with speed on par with sequential access to memory. Convey provides a hardware interface for accessing memory. Each AE is given access to 16 MC ports, which are multiplexed to the eight MCs. The AE's operate at a clock frequency of 150 MHz, allowing each memory controller port to make 150 million memory requests per second. Using these memory controllers, along with Convey's provided reorder queue and crossbar switch, greatly simplified the hardware design by allowing each MC port to access any address and making the data to flow in order.

V. Shepard Architecture

The five stages of the hardware pipeline are shown and described in Figure 3. The set of reads is split into four pieces, and each AE contains an identical pipeline to process its subset of the reads.

A. *Software Wrapper*

To simplify the hardware design process, the host processor (the x86 processor) does some marshaling data from the hard disk to the coprocessor memory and printing the data in a nice format. The reads do go through the host processor since the coprocessor does not have direct access to the hard drive.

After the coprocessor processes the reads, the result array is sent back to the host processor memory. The host processor converts this binary array into the human readable format specified by the Memocode competition. Aligning the data took the least amount of time when compared with these other disk operations. The following lists in order from fastest to slowest:

1) Alignment (coprocessor)
2) Moving data form and to host and coprocessor memory
3) Output formatting (i.e. sprintf)
4) Disk I/O

VI. Results

Our solution's runtime is 895 milliseconds for processing all 284,881,619 short read sequences in hardware. The timer starts before the reads begin streaming to the hardware pipeline, and ends after the last read's index and count are written to the result array. Using alignment speeds from [14] and [9], we can compare our runtime with other short read aligners:

TABLE I
Performance comparison of popular short read aligners.

Tool	Platform	Speed (reads/s)	% Aligned	Memory (GB)
MAQ [4]	CPU	50	93.2	1.2
SOAP	CPU	70	93.8	14.7
SOAP2 [5]	CPU	2,000	93.6	5.4
Bowtie [6]	CPU	2,500	91.7	2.3
SOAP3 [9]	GPU	6,000,000	96.8	3.2
Shepard	FPGA	350,000,000	25.2	23.3

The cost of the HC-1 is $67,100. Shepard's runtime omits the one time costs associated with creating and loading a 22.5 GB hash table into memory and loading a 780 MB reference genome to memory. In addition, it does not include the time it takes to load the reads from disk into memory. On a commodity server with 48 GB of available memory, the creation of the hash table took 147 minutes.

Another consideration was Shepard's use of available hardware resources on the Xilinx Virtex 5 LX330 [15]:

TABLE II
Hardware Resource Usage of Each FPGA

Resource	Amount Used	Available	% Used
Block RAM	24 blocks	288 blocks	8.3%
LUT	14,900	207,306	7.2%
Flip-Flops	15.5 Kb	3,420 Kb	0.5%

Convey's hardware interfaces to the memory controllers and application engine hub use some resources, but the majority of the reconfigurable hardware is left unused. However, Shepard's runtime is memory bound. The current design uses 12 out of the 16 available memory controller ports. Putting multiple pipelines on a single AE in order to use all available memory bandwidth would increase complexity and lead to only a slight improvement of performance.

VII. Conclusions

Currently, most mismatches between the reference genome and read sequences are from imperfect read quality, not due to genetic differences. The DNA of two individuals differs by roughly 0.1%, about one base pair out of a thousand. As DNA sequencing quality improves, so will the usefulness of exact match alignment tools such as Shepard. In fact, the alignment part of the problem disappears from the runtime when using approaches like Shepard. Indeed, future work will be needed to pull reads from disk or the network at speeds near 20 GB/s in order to keep such a hardware pipeline busy.

Additionally, one could increase the speed of the preprocessing step using the Convey HC-1. Instead of using a generic human reference genome, people may be able to use the DNA sequence of a blood family member as a reference in order to increase the percentage of exact matches.

Finally, there are other application domains in which the Shepherd architecture could be used with minimal modification. In 2011, Google developed their own hash function, CityHash [16], to improve the speed of their hash table lookups at their data centers, but Shepard would have been orders of magnitude faster. Other examples include internet data mining, social graphs, and search optimization.

References

[1] S. Levy et al., "The diploid genome sequence of an individual human," *PLoS Biol*, vol. 5, no. 10, 2007.

[2] S. A. Edwards, "MEMOCODE 2012 hardware/software codesign contest: DNA sequence aligner," Mar 2012.

[3] 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.

[4] H. Li, J. Ruan, and R. Durbin, "Mapping short DNA sequencing reads and calling variants using mapping quality scores." *Genome Research*, vol. 18, no. 11, pp. 1851–1858, 2008.

[5] R. Li, C. Yu, Y. Li, T.-W. Lam, S.-M. Yiu, K. Kristiansen, and J. Wang, "SOAP2: an improved ultrafast tool for short read alignment." *Bioinformatics*, vol. 25, no. 15, pp. 1966–1967, 2009.

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

[7] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, ser. FOCS '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 390–.

[8] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," Tech. Rep., 1994.

[9] SOAP3 alignment time. [Online]. Available: http://www.cs.hku.hk/2bwt-tools/soap3-dp/

[10] 1000 genomes project: Reference genome. [Online]. Available: ftp://ftp-trace.ncbi.nih.gov/1000genomes

[11] D. Belazzougui, F. C. Botelho, and M. Dietzfelbinger, "Hash, displace, and compress," in *Proceedings of the European Symposium on Algorithms (ESA)*, 2009, pp. 682–693.

[12] S. Hanov. (2011, Mar) Throw away the keys: Easy, minimal perfect hashing. [Online]. Available: http://stevehanov.ca/blog/index.php?id=119

[13] B. Jenkins. (2012, March) Spookyhash: a 128-bit noncryptographic hash. [Online]. Available: http://burtleburtle.net/bob/hash/spooky.html

[14] O. Knodel, T. Preusser, and R. Spallek, "Next-generation massively parallel short-read mapping on FPGAs," in *Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, Sept 2011, pp. 195–201.

[15] Xilinx. (2009, February) Virtex 5 family overview. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf

[16] G. Pike and J. Alakuijala. (2011, April) Introducing cityhash. [Online]. Available: http://google-opensource.blogspot.com/2011/04/introducing-cityhash.html