Parallel short read mapping using CUDA

Joe Zhou

December 15, 2016

1 Introduction

Next-generation sequencing yields a massive amount of short sequences data to be analyzed. The first step of the analysis is usually mapping those short reads against a reference genome. This process involves comparing each read to the entire reference genome while tolerating minimal mismatching. Because of the massive number of reads to align, mapping is intensive in terms of memory and time. Numerous algorithms have been implemented to map reads as accurately and fast as possible while minimize the memory required. There are many aspects of the analysis that can be paralleled, and they have been exploited and studied by researchers in the past decade. In this project, I will implement a parallel short reads mapping using NVIDIA CUDA platform. In this proposal, I will first explain the problem, show the implementation, explain the experimental design, discuss the results, and finally discuss the challenges and future directions for further improvements.

2 The Problem

The computational problem arises with the huge number of short reads data. The brute-force approach to genome alignment is simply align each read to the genome one by one, but its time complexity space and time complexity of is too costly in practice, $O(\frac{R!}{(R-M)!(M)!})$where R is the length of the reference genome (3.2 billion base pairs in human) and M is the length of each read (typically 70-80 base pairs) and N is number of reads (varies on the order of millions). The complexity increases to if mismatches (due to sequencing error and background noise) between reference and reads and gaps are tolerated. Fortunately, there are many ways to improve the algorithm so that it finishes in reasonable time.

One of the most obvious solution to the large time and space complexity is parallelization of the mapping algorithm. Since the FASTA file that contains read data is large (typically 5-20 gigabytes) but the reference genome is small (30-70 megabytes), we can distribute the read data among nodes (machines) and run the mapping program independently on each node. Gene expression profile can be calculated independently on each process, and the final result is the sum of profiles of all the nodes. Within each node are many processers, and we can

utilize them to speed up the program even further by having multiple threads can map reads in parallel. Theoretically, this approach can speed up the runtime and reduce the memory required roughly by the number of total processers provided. Other tricks to speed up the program include grouping identical reads before mapping, seed and extend to save memory space at the cost of time, and transforming reference genome into suffix trees to reduce time complexity of searching for a sequence to. More sophisticated alignment program relies on pre-computating methods to reduce space and time complexity, such as producing consensus sequence using Burrows-Wheeler transformation (such as Bowtie). This task can also be parallelized, having multiple threads assembling short reads simultaneously. Recent development of BlastReduce, a high performance short read mapping with MapReduce dramatically reduced run time by 250 times.

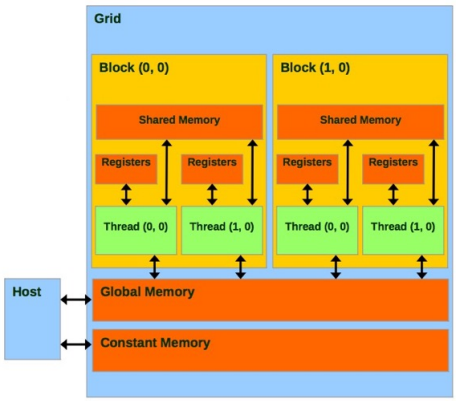

Figure 1: Memory hierarchy of a CUDA-enabled GPU

3 Implementation

The goal of the program is to map short reads to the reference genome as fast as possible by utilizing as many threads as avilable to us. I implemented the program using NVIDIA's CUDA platform. The memory organization of a CUDA-enabled GPU is shown in figure 1.

In a CUDA-enabled GPU, many threads are organized into blocks, and many blocks are organized into grids. All blocks in a grid have the same dimension. Blocks can be imagined to have three dimensions, `BlockDim.x, BlockDim.y, BlockDim.z`. Every thread in a block has an unique ThreadID, and every block in a grid has an unique BlockID. Using BlockID and ThreadID, individual threads in a grid can be identified. All threads in a grid execute the same kernel function.

Figure 2 shows the general design of the implemented mapping program.

First, reference genome sequence is passed to the host CPU. The host first separate the reference sequence into b partitions equally, where b is the number of blocks to be used, specified by the user. The goal of partitioning the reference sequence is to 1) speed up the time it takes to build a the suffix array, and 2) share the workload among multiple blocks, thus parallelizing the mapping algorithm. Then, the host constructs suffix arrays using a O(n2Log(n)) algorithm. The file that contains a large number of short reads are first processed in the Host CPU to be separated into smaller partitions equally for faster mapping on multiple threads in parallel.

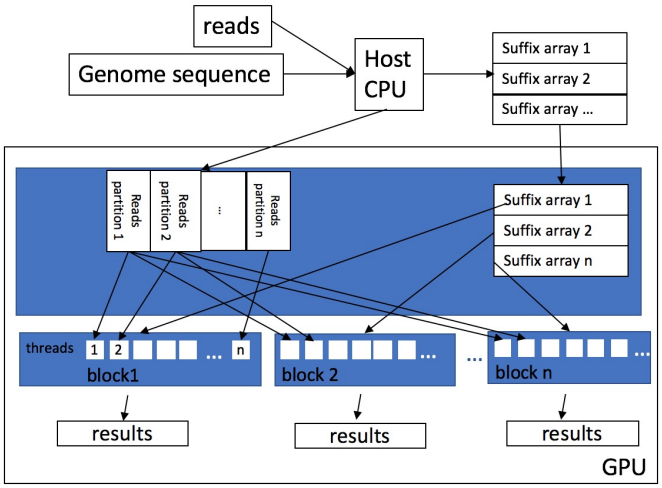

Figure 2: General design of the mapping program

Upon completion, CPU passes the partitioned suffix array (as an array of suffix arrays) and the partitioned read files (as an array of array of strings) to the CPU main memory. The CUDA syntax for this operation is first allocate space on GPU memory using `cudaMalloc(pointer, size)`, followed by `cudaMemCpy(*dest, *source, size, direction)` call, where the paramemter `direction` is `cudaMemcpyHostToDevice`. Once data were transferred from CPU memory to GPU memory, the kernel is called to be executed on all threads on the same grid (Figure 4). The syntax for calling a kernel is `search<<<numBlocks, numThreads>>>(arguments);`

A kernel is executed on every threads at the same time. A CUDA kernel is identified by the keyword `__global__`. Variables `blockIdx.x, blcokDim.x, threadIdx.x` are defined by the CUDA runtime An unique threadID is calculated for each threads. Then, each thread obtains a partition of the reads file from GPU's main memory. Finally, each thread finds the given read in a already reduced reference genome (in the form of a suffix array) using a O(m*Log(n)) search algorithm. Because of the massive amount of threads in each GPU grid, the number of search operations executed on each thread is relatively small com-

pared to the workload of each CPU thread. This massive parallel advantage is the core of the anticipated performance speedup.

In device, kernel cannot call host function. However, the `search()` kernel requires `strncmp()`, which was not available in CUDA's library. This is a common problem for many functions found in c++ standard library. Because of the fundamental architecture difference, host functions need to be tailored for a CUDA environment, but often it is very hard to achieve, such as like `strncmp`. A primitive `strncmpGPU` function was written (Figure 3). Note that in `strncmpGPU()`, `threadID` were not utilized, which means that multiple threads are performing the same task, while only one thread is sufficient. This is an example of the limited control and flexibility with GPGPU.

Finally, each block outputs an int array of size n, where n is the length of the partitioned reference genome relevant. The results array is transfered back to CPU main memory and examined.

4 Experimental Design

First, reads are generated from the reference file, and the correct starting position of the read is stored in a file that will be compared with results of the program. GPU mapping program was executed on a cluster BigRed2 at Indiana University (description of this system can be found at https://kb.iu.edu/d/bcqt). CPU mapping program was executed on a personal laptop with 4 intel i5 cores. Each trail was repeated 2 or 3 times, and the average times are represented in clock ticks. Number of reads and number of threads were the independent variable in the experiment.

5 Results and Discussion

The relationship between number of reads and clock ticks seems to be perfectly linear, both on CPU and GPU. This is expected as number of reads directly affect the amount of workload. The program was executed on 30 threads in GPU, but it only achieved 7.5 time speedup compare to executed on CPUs. Nonetheless, if a large number of threads are used in the program, the speedup can be significant.

Mapping results were originally designed to be compared with the correct key. However, the `saxpy` program (a sort of "hello world" program in CUDA, more details at https://devblogs.nvidia.com/parallelforall/easy-introduction-cuda-c-and-c/) was unable to give correct result. I researched this problem and it seemed like the specific GPU I was using required binaries to be built with CUDA 8.0. Unfortunately, the system only had CUDA 7.5 installed. Consequently, I was not able to generate result from mapping on GPUs.

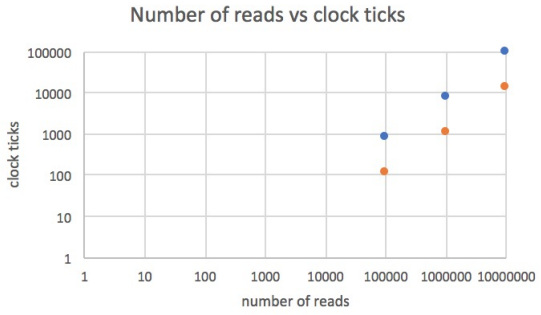

Figure 3: Blue points indicate CPU program. Orange points indicate GPU program. 30 threads were used in GPU.

6 Challenges and future directions

One challenge is the flexibility of working in a CUDA environment. While GPUs give the users tremendous amounts of computing power, but the trade-off is that it is not early as flexible or compatible. Many useful functions in the c++ standard library are simply not found in CUDA library, and thus needs to be implemented by the users. Even with self-implemented kernel, the ability to access memory is limited.

Another challenge was the hardship with organizing and storing a massive amount of data, especially the read file. I encountered a mysterious segment fault error in reading and storing the reads into an array of string array, only in more than read file that contains more than 1E7 reads. The error did not affect the timing of the program, so I just left it unfixed.

Further improvements include reduce the alphabet A,T,C,G to be represented by 2 bits, utilizing compiler directives, and many optimized libraries.

7 Conclusion

In conclusion, I demonstrated the possibility that the short read mapping program can achieve significant speedup by utilizing a large amount of threads on GPUs using CUDA. Throughout the process, the advantages and disadvantages of GPGPU was demonstrated. GPGPU can harvest a large amount of computation power, especially in applications that are embarrassingly parallel, but the trade-off is the limited control over threads and flexibility. Nonetheless, it is a powerful tool to be used in bioinformatics, where sequencing data is exploding in the era of cheap and massively-parallel sequencing technologies.

```
__device__
void strncmpGPU(const char *s1, const char *s2, size_t n,
        int* result){
    int threadID = blockIdx.x * blockDim.x + threadIdx.x;
    for( ; n>0;s1++, s2++, --n)
        if(*s1 != *s2)
            *result = ((*(unsigned char *)s1 <
                *(unsigned char *)s2) ? -1 : +1);
        else if (*s1 == '\0')
            *result = 0;
    *result = 0;
}

__global__
void search(char*** readsGroups, int numReads, char* ref,
        int* sa, int* result, int m, int n){
    int threadID = blockIdx.x * blockDim.x + threadIdx.x;
    //each thread reads a subset of all the reads
    char** reads = readsGroups[threadID];

    for(int i=0; i< numReads; i++){
        char* pat = reads[i];
        int l = 0, r = n-1;

        while (l <= r)
        {
            int mid = l + (r - l)/2;
            int *res = new int;
            strncmpGPU(pat, ref+sa[mid], m, res);

            if (*res == 0)
                result[sa[mid]] = result[sa[mid]] + 1;
            if (*res < 0)
                r = mid - 1;
            else
                l = mid + 1;
        }
    }
}
```

Figure 4: the string searching CUDA kernal