ECE 508 Final Project Report
CRISPR in CUDA

Dylan Huang
https://github.com/dphuang2/crispr-cuda

May 2019

1 Introduction

The CRISPR System [2] is a tool that allows biologists to reliably edit genes. To create the system, biologists must engineer a "guide" consisting of 20 nucleotides (A, C, T, or Gs) that can match a target gene at a low false positive rate. A low false positive rate is important to avoid unintended edits of a gene. To reduce the false positive rate, we must search for every match in the genome and verify that each match is correct. Typically a biologist will try about 200 guides on a single genome of 3 billion nucleotides. A match will occur at every position where a guide has an edit distance (without indels) of less than or equal to 4. By nature this problem is easily parallelizable due to the independence across calculation. For my project, I implemented the naive solution to this problem in CUDA and attempted to optimize it further by using thread coarsening, warp queues, shared memory, and register cache. After designing the final algorithm, I analyze the performance across various combinations of optimizations.

2 Design Overview and Implementation

2.1 Problem Formulation

Let's define a function match(guide, genomeregion) which returns *true* if guide and genomeregion are close enough that Cas9 would successfully cut at the genomeregion. In other words, it will return true if they are 1) the same length, and 2) differ in at most 4 characters.

```
1 match('AAAAAAAAAAAAAAAAAAAA', 'AAAAAAAAAAAAAAAATTTT') // True
2 match('AAAAAAAAAAAAAAAAAAAA', 'AAAAAAAAAAAAAAATTTTT') // False
```

Given a genome G and a set of guides S (all with the same length), we would like to find all locations in G that match at least one guide in S. This problem is different to sequence alignment and string searching algorithms in that we want to find **every** location that has an edit distance (without indels) of 4 or less rather than a **single** best alignment/exact match for a given query. Our edit distance problem is effectively O(1) as our guide size of 20 is known beforehand.

2.2 Naive Solution

Our naive CPU solution is a simple set of for loops to check for every starting index against every guide.

```
set<int64_t, int64_t> naive_cpu_guide_matching(
    char * genome,
    int64_t genome_length,
    char * guides,
    int num_guides)
{
    set<int64_t, int64_t> results;
    for (int64_t i = 0; i < genome_length; i++) {
        for (int j = 0; j < num_guides; j++) {
            char * guide = guides + (j * GUIDE_BUFFER_SIZE);

            int mismatches = 0;
            for (int k = 0; k < GUIDE_SIZE; k++)
                mismatches += genome[i + k] != guide[k];

            if (mismatches <= EDIT_DISTANCE_THRESHOLD)
                results.insert(make_tuple(j, i));
        }
    }
    return results;
}
```

Figure 1: C++ implementation of the naive CPU algorithm

The naive CPU solution has a final complexity of $O(G * S)$ where G denotes the size of the genome and S is the number of guides. Our naive GPU solution is not far from the CPU solution. We simply replace each starting index in the genome with a thread and process each starting index independently.

```
__global__ void naive_gpu_kernel(
        char * genome,
        int64_t genome_length,
        char * guides,
        int num_guides,
        int64_t * results,
        int * numResults,
        int64_t sizeOfResults)
{
    int64_t tid = blockDim.x * blockIdx.x + threadIdx.x;
    for (int64_t i = tid; i < genome_length - GUIDE_SIZE - 1; i += gridDim.x
        * blockDim.x) {
        for (int j = 0; j < num_guides; j++) {
            char * guide = guides + (j * GUIDE_BUFFER_SIZE);

            int mismatches = 0;
            for (int k = 0; k < GUIDE_SIZE; k++)
                mismatches += genome[i + k] != guide[k];

            if (mismatches <= EDIT_DISTANCE_THRESHOLD) {
                insert_into_queue(
                    results,
                    sizeOfResults,
                    numResults, j, i); // Uses atomicAdd
            }
        }
    }
}
```

Figure 2: CUDA implementation of naive GPU solution

2.3 Optimizations

It is immediately obvious that our naive GPU solution has a lot of overlapping memory accesses that can be optimized.

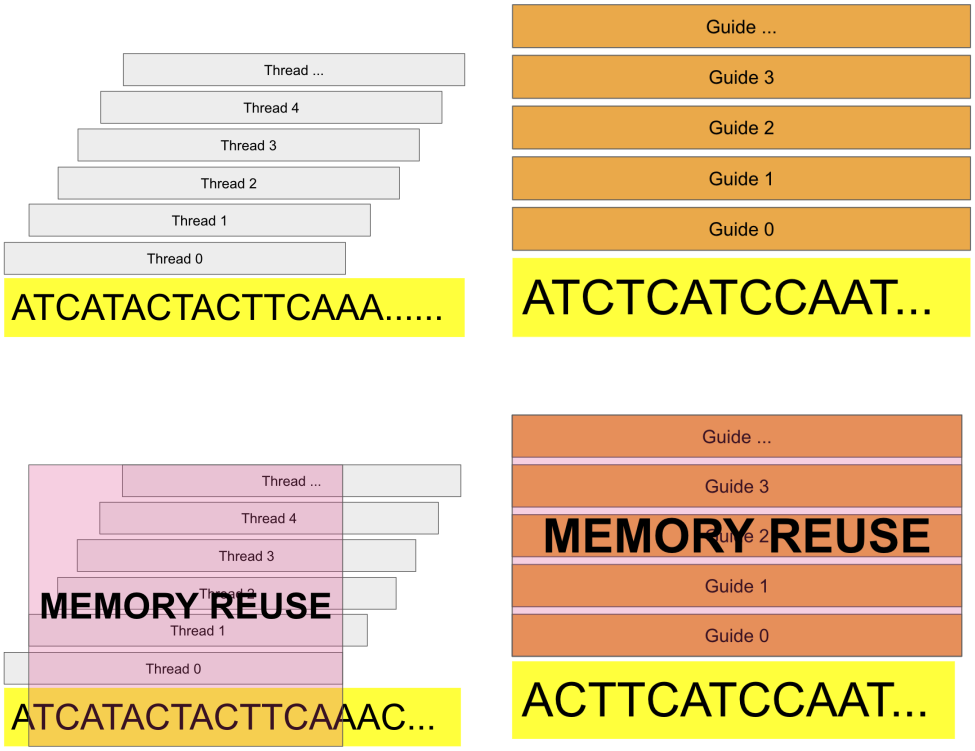


Figure 3: Illustration of how threads access and compare the genome to the guides. Threads have overlapping memory accesses when accessing the genome **(left)** and guides **(right)** from global memory.

Figure 3 illustrates the redundant memory reuse pattern of our naive GPU kernel. For a guide length of 20, each thread shares $20 - 2 * x$ memory accesses with a thread that is x distance away (ie. Thread 0 shares 18 accesses with Thread 1). The total number of redundant memory accesses for a single thread is $\sum_{x=1}^{9} 20 - 2 * x = 90$. This means there are **270 billion redundant memory accesses** For a genome of length 3 billion. Assuming a guide length of 20, number of guides of 200, and 1024 threads in a block there are $1024 * 200 * 20 =$ **4.096 million redundant memory accesses per block**. To reduce these redundant memory accesses we will perform collaborated shared memory stores from the genome buffer and guides buffer. Our thread blocks now have all the necessary genome and guide memory stored in shared memory. We can do another reduction of memory accesses by coarsening our threads such that each thread processes more than one starting index. This helps us reduce the number of overlapping memory accesses between adjacent threads. Another optimization can be made during the insertion of our results into our queue on line 23 of figure 2. Since we are using atomic operations to update the index of our queue, we can reduce memory update conflicts by utilizing warp queues.

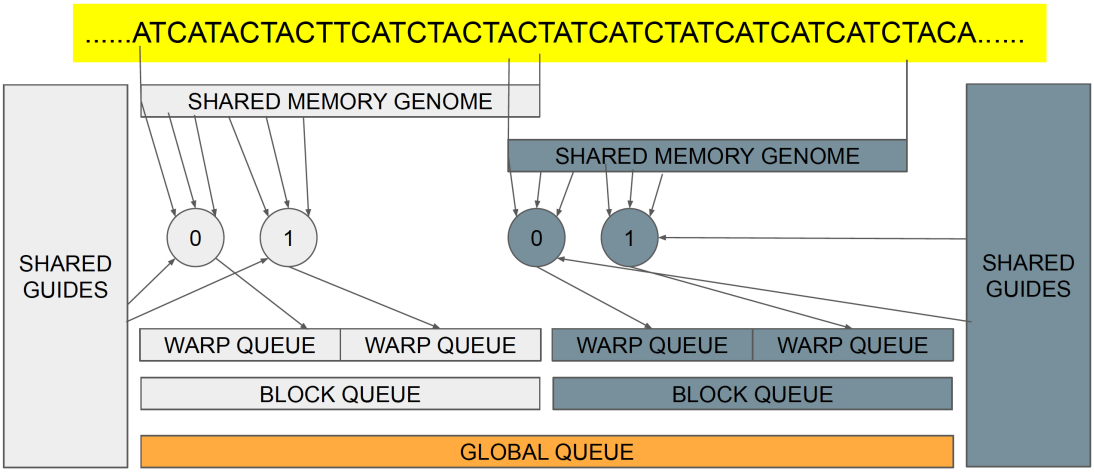


Figure 4: Illustration of data flow in the final algorithm. Shows two consecutive blocks **(differentiated by color)** along with two local threads **(circles)** pulling multiple starting indexes from the genome and inserting results into the warp queue.

Figure 4 illustrates our final algorithm using shared memory for genome and guides, thread coarsening for reduced memory access overlap, and warp queues for reduced atomic overhead.

3 Performance and Results Analysis

For analysis, I implemented a combination of optimizations and tested their kernel times across various genome lengths. The following is a summary of kernels that I implemented and tested:

3.1 Naive

Same as figure 2

3.2 Shared Memory

Uses collaborated stores to shared memory for the genome and guides.

3.3 Shared + Constant Memory

Uses collaborated stores to shared memory for the genome and constant memory for the guides.

3.4 Coarsened Shared Memory with Register Cache

Uses collaborated stores to shared memory for the genome and guides. Coarsens threads by processing multiple adjacent genome indexes per thread. Loads register cache from shared

memory.

3.5 Register Cache

Each thread loads their portion of the genome (20 nucleotides) into a register cache. Genome data is fetched from this register cache when processing.

3.6 Coarsened Register Cache

Threads are coarsened to reduce overlapped memory access and register cache is loaded from global genome buffer. Guides are simply pulled from global memory.

3.7 Coarsened Register Cache + Shared Guides

Threads are coarsened to reduce overlapped memory access and a register cache is used for genome data. Uses collaborated stores to shared memory for guides and loads register cache from global memory.

3.8 Coarsened Register Cache with Shuffling + Shared Guides

Same as section 3.7 but uses register shuffling with warp-centric programming and intra-warp communication as proposed by Ben-Sasson et al. [1]. Each register holds partial data in a round-robin fashion (ie. Thread 0 holds genome[0], genome[32], etc. Thread 1 holds genome[1], genome[33] etc.). In each iteration of the guide comparison for loop, each thread makes a call to __shfl_sync to share and retrieve data from their respective warp. After each publish, read, and computation, each thread calculates the next piece of data to publish without branch divergence on line 11 of figure 5. This was implemented after it was suggested during my presentation.

```
1     char rc[LOCAL_REGISTER_SIZE];
2     // Store genome data in round-robin fashion into register cache
3     // Calculate active thread mask
4
5     for (n = 0; n < OUTPUT_PER_THREAD_SHUF; n++) {
6         char to_share = rc[n];
7         mismatches = 0;
8         for (k = 0; k < GUIDE_SIZE; k++) {
9             // "& (WARP_SIZE - 1)" is equal to modulo WARP_SIZE
10            mismatches += __shfl_sync(mask, to_share, (local_id + k) &
                  (WARP_SIZE - 1)) != guide[k];
11            to_share += (k == local_id) * (rc[n + 1] - rc[n]);
12        }
13        if (mismatches <= EDIT_DISTANCE_THRESHOLD) {
14            insert_into_queue(results,
15                sizeOfResults,
16                numResults,
17                j,
18                global_id + n * WARP_SIZE);
19        }
20    }
```

Figure 5: Shuffle code using the __shfl_sync function. Does not include setup for register cache and mask calculation

3.9 Coarsened Register Cache + Shared Guides + Warp Queue

Same as section 3.7 with the addition of a warp queue for insertion of results. This is the proposed final solution illustrated in figure 4.

3.10 Results

To benchmark each method, I tested genome lengths from 100 million nucleotides to 3 billion nucleotides in 100 million nucleotide increments over 4 trials.

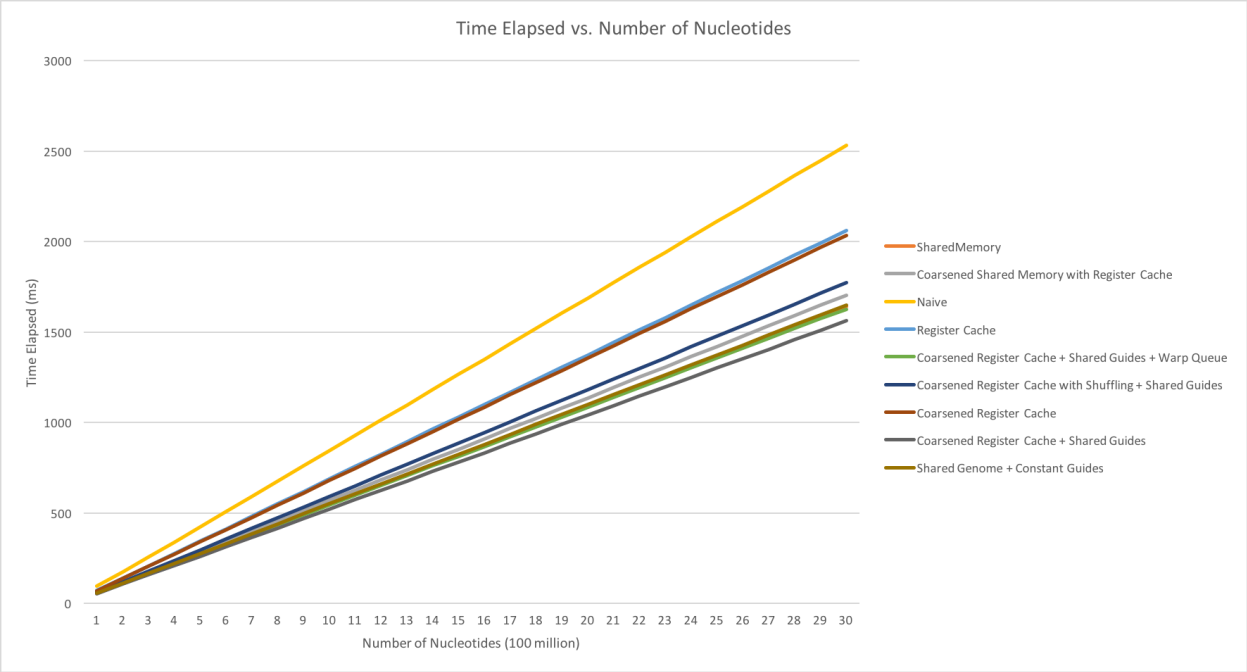


Figure 6: Graph of time elapsed (ms) versus number of nucleotides. Time elapsed only counts kernel time and not time to allocate and copy memory to the device.

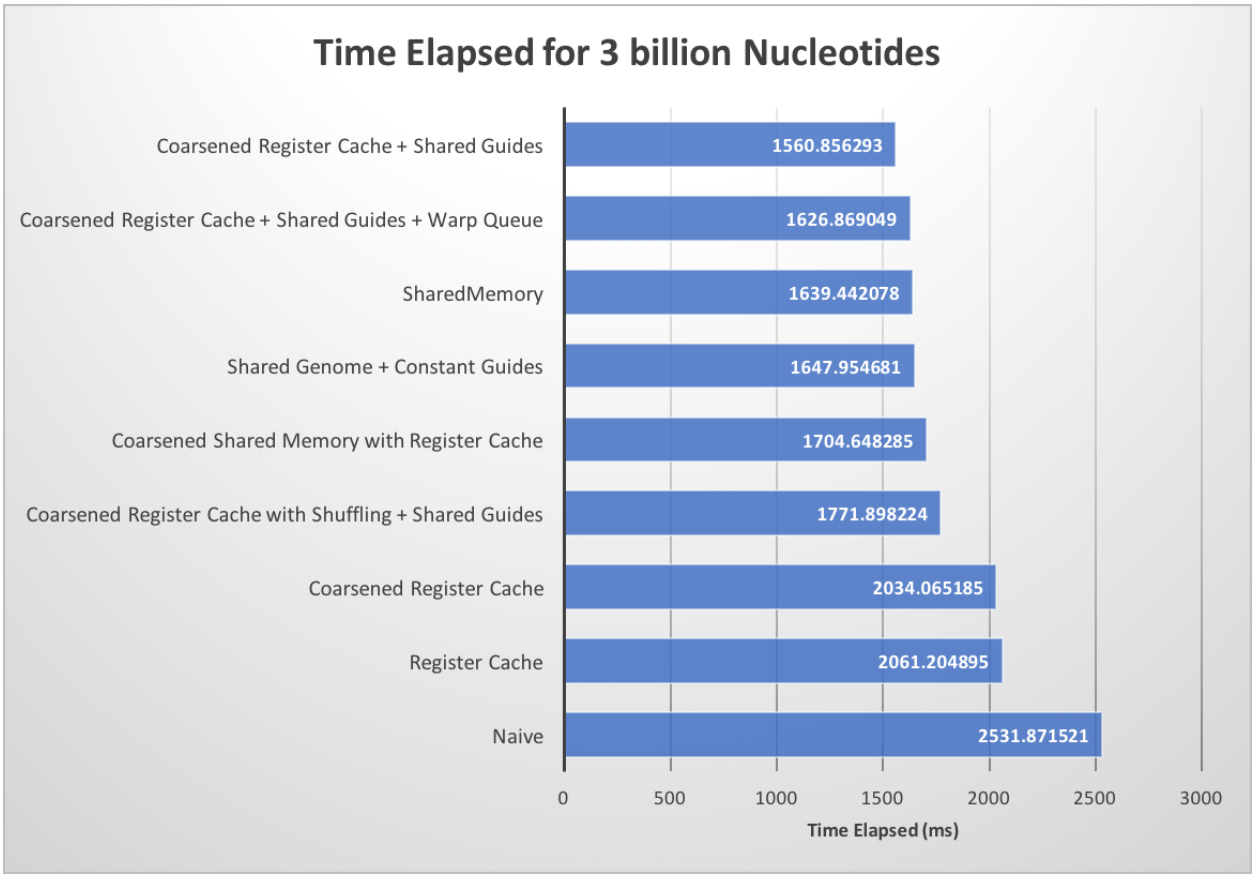
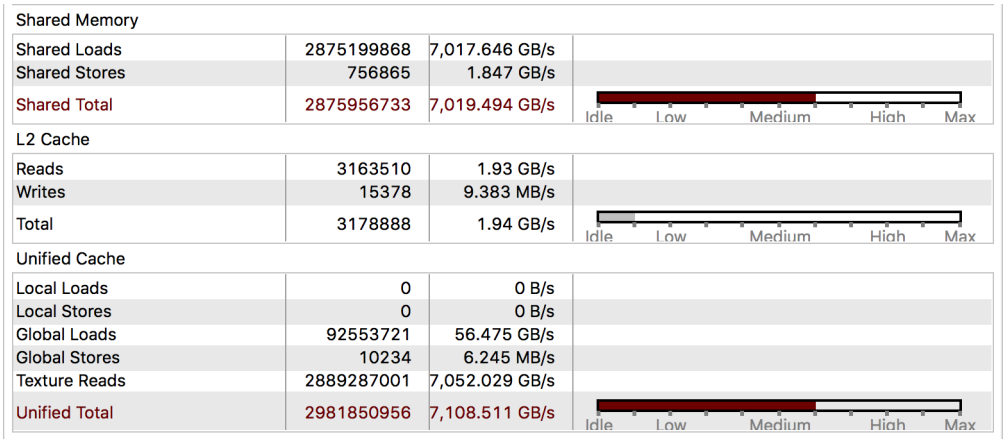


Figure 7: Graph of time elapsed (ms) for 3 billion nucleotides. Time elapsed only counts kernel time and not time to allocate and copy memory to the device.

Figures 6 and 7 show the time elapsed as the number of nucleotides increases and a zoomed in view at 3 billion nucleotides. All methods demonstrate the exact same linear complexity. The slowest being the naive implementation and the fastest being Coarsened Register Cache + Shared Guides. I assumed that my final solution was the most optimized out of all the implementations but it did not yield the best performance. It was surprising to see that not putting the genome into shared memory was faster. It was also interesting to see that adding warp queues decreased the performance. I suspect that simply loading from global memory was faster due to the memory access pattern taking advantage of DRAM bursting and warp queues decreased performance by incurring more overhead than speedup. I was also surprised to see register cache with shuffling as one of the slowest implementations.

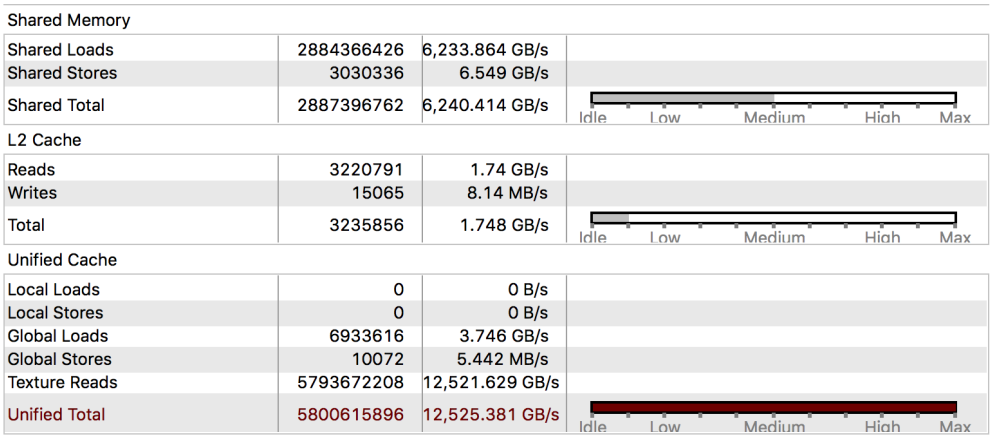
3.11 Analysis

For analysis I ran nvprof with Coarsened Register Cache + Shared Guides (3.7), Coarsened register Cache with Shuffling + Shared Guides (3.8), and Coarsened Register Cache + Shared Guides + Warp Queue (3.9) because methods (3.7) and (3.9) are my fastest implementations and method (3.8) was proposed to me during my presentation.



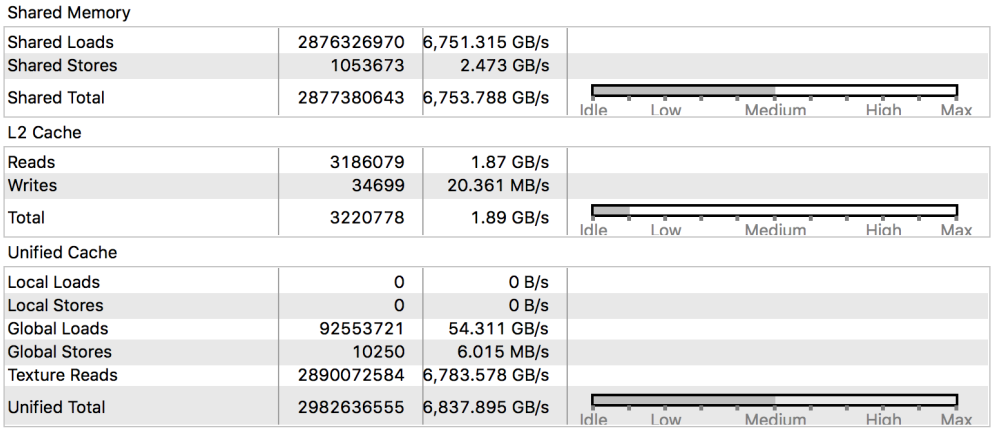
Shared Memory			
Shared Loads	2875199868	7,017.646 GB/s	
Shared Stores	756865	1.847 GB/s	
Shared Total	2875956733	7,019.494 GB/s	Idle Low Medium High Max
L2 Cache			
Reads	3163510	1.93 GB/s	
Writes	15378	9.383 MB/s	
Total	3178888	1.94 GB/s	Idle Low Medium High Max
Unified Cache			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Global Loads	92553721	56.475 GB/s	
Global Stores	10234	6.245 MB/s	
Texture Reads	2889287001	7,052.029 GB/s	
Unified Total	2981850956	7,108.511 GB/s	Idle Low Medium High Max

(a) Coarsened Register Cache + Shared Guides 3.7



Shared Memory			
Shared Loads	2884366426	6,233.864 GB/s	
Shared Stores	3030336	6.549 GB/s	
Shared Total	2887396762	6,240.414 GB/s	Idle Low Medium High Max
L2 Cache			
Reads	3220791	1.74 GB/s	
Writes	15065	8.14 MB/s	
Total	3235856	1.748 GB/s	Idle Low Medium High Max
Unified Cache			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Global Loads	6933616	3.746 GB/s	
Global Stores	10072	5.442 MB/s	
Texture Reads	5793672208	12,521.629 GB/s	
Unified Total	5800615896	12,525.381 GB/s	Idle Low Medium High Max

(b) Coarsened Register Cache w/ Shuffling + Shared Guides 3.8



Shared Memory			
Shared Loads	2876326970	6,751.315 GB/s	
Shared Stores	1053673	2.473 GB/s	
Shared Total	2877380643	6,753.788 GB/s	Idle Low Medium High Max
L2 Cache			
Reads	3186079	1.87 GB/s	
Writes	34699	20.361 MB/s	
Total	3220778	1.89 GB/s	Idle Low Medium High Max
Unified Cache			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Global Loads	92553721	54.311 GB/s	
Global Stores	10250	6.015 MB/s	
Texture Reads	2890072584	6,783.578 GB/s	
Unified Total	2982636555	6,837.895 GB/s	Idle Low Medium High Max

(c) Coarsened Register Cache + Shared Guides + Warp Queue 3.9

Figure 8: Memory Utilization Analysis of analyzed methods with transactions, bandwidth, and utilization shown respectively.

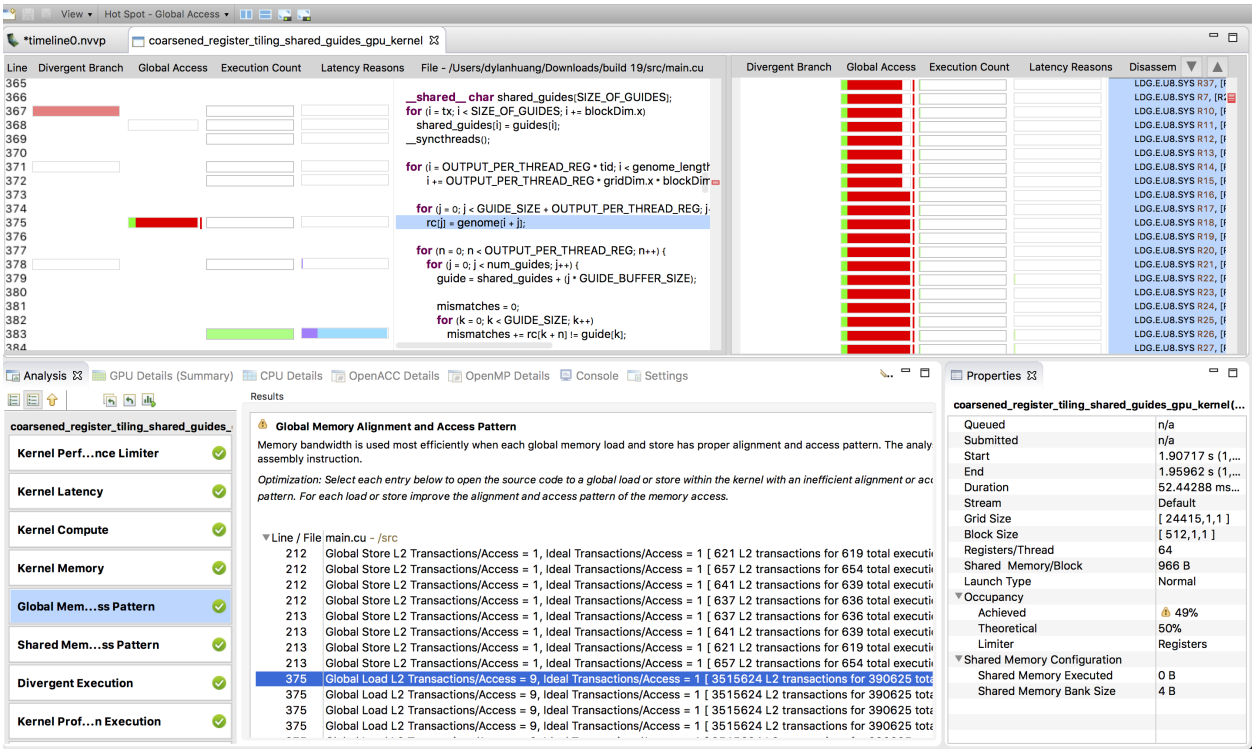


Figure 9: nvvp output showing global memory accesses for method (3.7). Picture shows line info **(left)** and assembly line info **(right)**

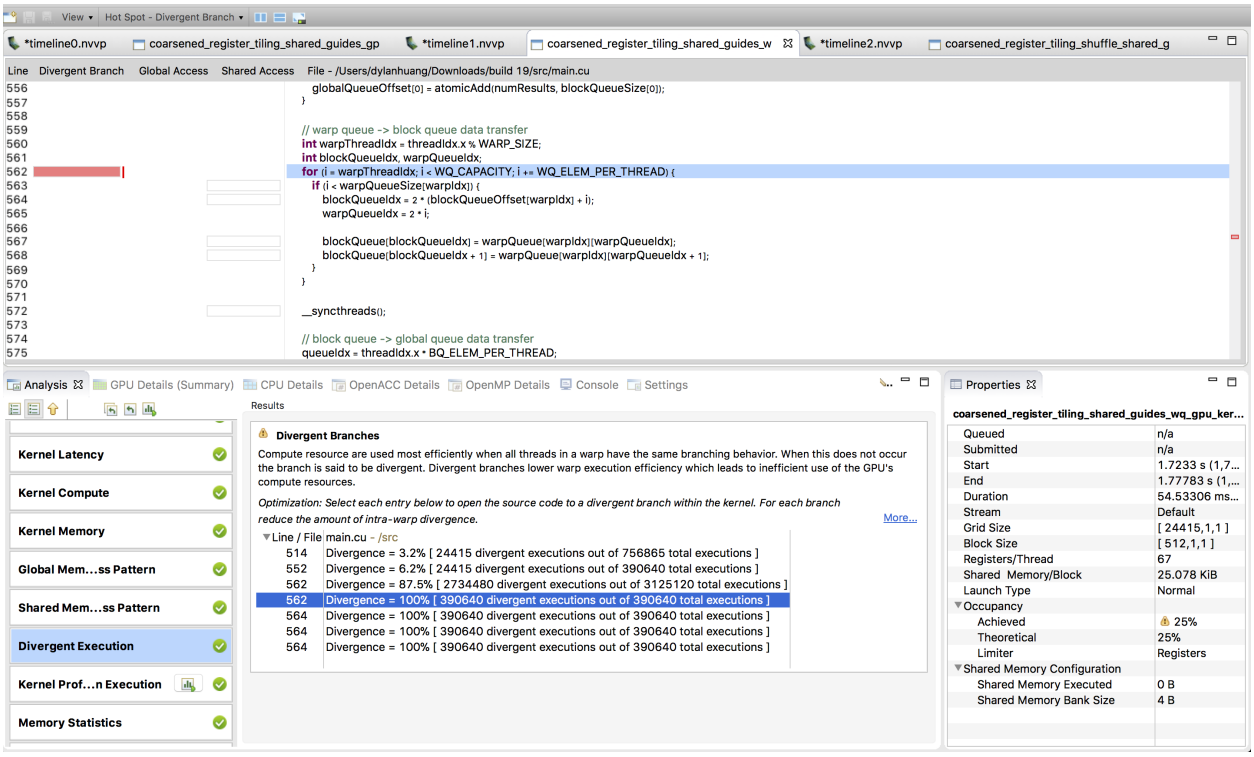


Figure 10: nvvp output showing divergence for method (3.9). Pictures shows line info **(top)** and divergence percentages (bottom)

Methods (3.7) and (3.8) had similar shared memory utilization while method (3.8) had a higher Unified Cache memory utilization. This explains why method (3.7) and method (3.8) are faster than method (3.9).

Figure 9 shows lots of global memory transactions per access for method 3.7 when accessing the genome. We can also see lots of 8 bit assembly load instructions on the right side of the picture. This is because of the overlap of memory accesses between adjacent threads. From figure 8 we do not see L2 Cache utilization in method (3.7) so it does not seem like there is any sort of caching being performed here. The 8 bit load instructions could be alleviated in future implementations by using vector loads [3].

From figure 10 we can see 100% divergence happening while transferring data from warp queue to block queue. This is because of the nature of the algorithm and how the warps are divided. I tried changing the way the threads were distributed among the warp queues but this did not change performance or divergence. This could be the cause of slowdown from method (3.7) to method (3.9).

3.11.1 Analysis Conclusion

Overall the results were surprising. I thought that 3.9 would be the fastest but it turned out that using global memory instead of shared memory for genome data was faster. It also turned out that register shuffling and warp queues did not help performance as much as expected.

4 Conclusion

The optimizations that I implemented show considerable speedup compared to the naive GPU algorithm. Utilizing shared memory and intra-warp communication with warp shuffling were not as fruitful as I had hoped. For the future there are other optimizations that can be explored such as overlapping memory transfer and kernel execution, faster paralellizable serial algorithms, and dynamic parallelism for processing portions of the genome./

References

[1] Eli Ben-Sasson, Matan Hamilis, Mark Silberstein, and Eran Tromer. Fast multiplication in binary fields on gpus via register cache. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, pages 35:1–35:12, New York, NY, USA, 2016. ACM.

[2] Le Cong, F. Ann Ran, David Cox, Shuailiang Lin, Robert Barretto, Naomi Habib, Patrick D. Hsu, Xuebing Wu, Wenyan Jiang, Luciano A. Marraffini, and Feng Zhang. Multiplex genome engineering using crispr/cas systems. *Science*, 339(6121):819–823, 2013.

[3] Justin Luitjens. Cuda pro tip: Increase performance with vectorized memory access. `https://devblogs.nvidia.com/cuda-pro-tip-increase-performance-with-vectorized-memory-access/`.