Communication Optimization on GPU: A Case Study of Sequence Alignment Algorithms

Jie Wang
University of California, Los Angeles
Los Angeles, USA

Xinfeng Xie
Peking University
Beijing, China

Jason Cong
University of California, Los Angeles
Los Angeles, USA

Abstract—**Data movement is increasingly becoming the bottleneck of both performance and energy efficiency in modern computation. Until recently, it was the case that there is limited freedom for communication optimization on GPUs, as conventional GPUs only provide two types of methods for inter-thread communication: using shared memory or global memory. However, a new warp shuffle instruction has been introduced since the Kepler architecture on Nvidia GPUs, which enables threads within the same warp to directly exchange data in registers. This brought new performance optimization opportunities for algorithms with intensive inter-thread communication. In this work, we deploy register shuffle in the application domain of sequence alignment (or similarly, string matching), and conduct a quantitative analysis of the opportunities and limitations of using register shuffle. We select two sequence alignment algorithms, Smith-Waterman (SW) and Pairwise-Hidden-Markov-Model (PairHMM), from the widely used Genome Analysis Toolkit (GATK) as case studies. Compared to implementations using shared memory, we obtain a significant speed-up of 1.2× and 2.1× by using shuffle instructions for SW and PairHMM. Furthermore, we develop a performance model for analyzing the kernel performance based on the measured shuffle latency from a suite of microbenchmarks. Our model provides valuable insights for CUDA programmers into how to best use shuffle instructions for performance optimization.**

I. INTRODUCTION

The graphics processing unit (GPU) is a widely used heterogeneous platform that is equipped with a massive number of threads, offering high performance for many applications. It has become an integral part of today's computing systems.

Communication is an important factor for both performance and energy efficiency on GPUs. Table I shows the computation throughput, shared memory and global memory bandwidth (BW) on two Nvidia GPUs. As we can see, there is a huge gap between computation and memory systems, making it usually unrealistic to fully exploit the rich computation resources on GPUs. This has spawned an active research community for communication optimization on GPUs [1]–[3].

Communication among threads (inter-thread communication) takes up a large portion of the total communication on GPU, considering that the GPU uses a large number of threads to explore the parallelism in applications. For applications with intensive inter-thread communication, the performance will be limited by the efficiency of inter-thread communication methods. There are two conventional types of methods for GPU threads to communicate with each other: shared memory and global memory. Threads within the same thread block can communicate through shared memory, which is basically a scratchpad memory that can offer high bandwidth. Threads in different thread blocks can communicate via global memory. For both methods, we need to set explicit synchronization barriers for potential data hazards.

Table I: Overview of the gap between computation and memory systems on modern GPUs.

	Nvidia K1200	Nvidia Titan X
GFLOPs	1,057	6,611
shared memory BW(GB/s)	550	3,302
global memory BW(GB/s)	80	336

Starting from the Kepler architecture, threads within the same warp can communicate with each other using a new instruction called SHFL, or "shuffle" [4]. Shared memory can be saved by using shuffle instructions, which may help improve the occupancy, and synchronization overheads are eliminated because shuffle is used by threads within one single warp with implicit synchronization.

This new instruction provides a unique opportunity for optimizing communication in applications with intensive inter-thread communication. However, shuffle has its limitations, as it can be used only for threads within the same warp, and it will increase the register usage which may become the new limiter for occupancy. Trade-offs between the benefits and limitations of shuffle need to be considered before deploying shuffle in kernels. Previous works [5]–[7] using shuffle instructions to obtain performance gains did not investigate such trade-offs and there is a lack of quantitative and systematic approaches for analyzing the impacts of shuffle instructions on kernel performance. This motivates us to conduct a detailed analysis of shuffle instructions.

In this work we select two algorithms, Smith-Waterman (SW) and Pairwise-Hidden-Markov-Model (PairHMM), from a widely used genomic application (GATK) [8] as case studies for analyzing the effectiveness of shuffle instructions on communication optimization. For each algorithm, we implement two designs using either shared memory or shuffle for inter-thread communication. Furthermore, we develop a suite of microbenchmarks to evaluate the latency of shuffle and other instructions in detail. A performance model to analyze the kernel performance is built based on the measured latency from these microbenchmarks. We conduct a detailed analysis of shuffle instructions in two kernels with the help of the performance model and microbenchmarks.

We summarize the contributions of our work as follows.

- We conduct a quantitative and systematic analysis of the impact of shuffle instructions on communication optimization. A suite of microbenchmarks is developed for measuring the latency of shuffle instructions. Furthermore, a performance model for analyzing the performance of sequence alignment algorithms is developed. This model helps validate the results from microbenchmarks and estimates the performance gains of using shuffle instructions, taking all trade-offs into consideration.
- We use shuffle instructions to optimize the inter-thread communication for two sequence alignment algorithms, SW and PairHMM. Compared to designs using shared memory, we achieve speedups of 1.2× and 2.1× for SW and PairHMM, respectively, using shuffle instructions.
- We conclude the trade-offs of using shuffle instructions from the case studies of two sequence alignment algorithms. This work provides valuable insights for CUDA programmers to use shuffle instructions for communication optimization in a wider class of applications.

The remainder of this paper is organized as follows. In Section II we present details of shuffle instructions. Microbenchmarks for testing the shuffle latency are discussed. Section III describes the algorithms of SW and PairHMM. In Section IV we discuss the general design methodology of using shared memory or shuffle instructions for these algorithms, and optimization techniques to further improve the performance. Then we present the performance model for analyzing and estimating the performance of these designs. Experimental results are presented in Section V. We discuss the overall performance of our designs and conduct a detailed analysis on trade-offs of using shuffle instructions. Section VI summarizes prior research. Finally, we conclude our work in Section VII.

II. Understanding Shuffle Instructions

In this section we will first introduce the shuffle instructions. Then we will present the microbenchmarks for testing the latency of shuffle and several related instructions.

A. *Shuffle Instruction*

Shuffle allows threads within a warp to directly share data in registers. It can be used only for threads within a single warp, and all the threads involved in the shuffle instruction need to be active during the execution time. No explicit synchronization is needed for shuffle as it is executed within the single warp.

Figure 1: Variants of shuffle instructions.

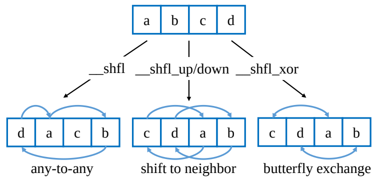
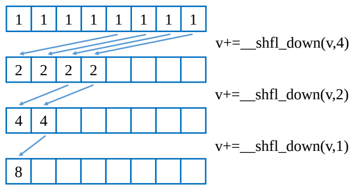


Figure 2: An example of using shuffle instructions for reduction.



There are four variants of shuffle instructions, as depicted in Figure 1. *__shfl* can directly copy data from any indexed lane. *__shfl_up* and *__shfl_down* copy data from a lane with either a lower or a higher ID relative to the caller. The last instruction *__shfl_xor* copies data from a lane based on bitwise XOR of its own lane ID.

Figure 2 depicts an example using shuffle for reduction. Without shuffle, we need to store the intermediate data at each reduction stage back to either shared or global memory. The benefits of using shuffle are summarized below [9].

- Save shared memory usage. Using register shuffle can free up shared memory. The shared memory can be either used for other data or to increase the occupancy.
- Reduce instruction count. As for read-after-write access (e.g., Figure 2), when using shared memory, three instructions (write, synchronize, and read) are needed. Shuffle can finish the same work with only one single instruction.
- Eliminate synchronization. Explicit synchronization is not required since threads within the same warp are implicitly synchronized due to the single-thread-multiple-thread (SIMT) execution model on GPU.

Nevertheless, we find that with the exception of the third point above, the benefits are not obvious performance-wise. Although using register shuffle can save shared memory, the register usage will increase, which may become the new limitation factor for occupancy, and thus may affect performance. As for the second point on the reduced instruction count, the latency of shuffle instructions is not disclosed publicly, which makes it difficult to estimate the performance gains using shuffle. These observations motivate us to study the shuffle instructions in a quantitative and systematic approach, using sequence alignment algorithms as case studies.

```
__global__ void reg(float* in, float* out) {
  float a = in[threadIdx.x];
  for (int i=0; i<ITERATIONS; i++)
    a *= a;
  out[threadIdx.x] = a;
}
__global__ void shuffle(float* in, float* out) {
  float a = in[threadIdx.x];
  for (int i=0; i<ITERATIONS; i++)
    a *= __shfl(a, src_thread);
  out[threadIdx.x] = a;
}
__global__ void sharedMem(float* in, float* out) {
  __shared__ float buf[32];
  for (int i=0; i<32; i++)
    buf[i] = in[i];
  int ind = buf[0];
  float a = 1.0;
  for (int i=0; i<ITERATIONS; i++) {
    ind = buf[ind];
    a *= ind;
  }
  out[0] = a;
}
__global__ void sharedMemSync(float* in, float* out) {
  __shared__ float buf[32];
  for (int i=0; i<32; i++)
    buf[i] = in[i];
  int ind = buf[0];
  float a = 1.0;
  for (int i=0; i<ITERATIONS; i++) {
    ind = buf[ind];
    a *= ind;
    __syncthreads();
  }
  out[0] = a;
}
```

Listing 1: CUDA code for testing the instruction latency.

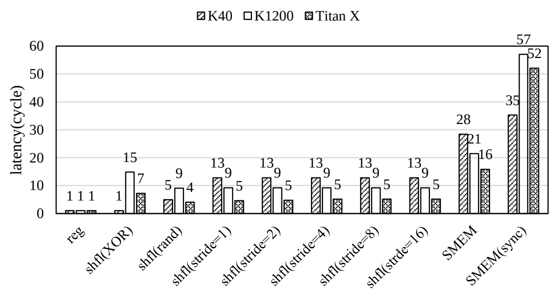
B. *Microbenchmark for Testing the Shuffle Latency*

The shuffle instructions are first introduced on the Kepler architecture. However, latency of these instructions is not disclosed and therefore remains unclear to CUDA programmers. Such information is critical to performance estimation. Therefore, we develop a suite of microbenchmarks to estimate the latency of shuffle and several other related instructions. This benchmark is conducted on several GPUs with different architectures to further evaluate the performance across different GPU generations.

This microbenchmark covers all variants of shuffle instructions, as shown in Figure 1 in Section II. To provide performance comparison, the microbenchmark also covers the shared memory access and synchronization using *_syncthreads()*.

The major code used in the microbenchmark is shown in Listing 1. For the kernel *shuffle()*, each thread in the warp loads one item from the global memory and updates it using data from other threads for several iterations. Considering the RAW dependency of variable a across different

Figure 3: Test results of microbenchmarks.



iterations, the elapsed time of the kernel can be calculated as below.

$$t_{shuffle} = \#iteration \times (latency_{shuffle} + \alpha) + \beta \quad (1)$$

The factor α refers to the sum of latency of all the remaining instructions (e.g., multiplication). The factor β covers overheads outside the loop.

The kernel *register()* uses only register access. Similarly, the elapsed time for this kernel is calculated by the equation below.

$$t_{reg} = \#iteration \times (latency_{reg} + \alpha) + \beta \quad (2)$$

Therefore, by conducting multiple runs with different numbers of iterations, we can use a linear regression model to obtain the slope factors $k_{shuffle} = latency_{shuffle} + \alpha$ and $k_{reg} = latency_{reg} + \alpha$. The latency of the *shfl()* instruction is therefore derived as $latency_{reg} + k_{shuffle} - k_{reg}$.

To avoid the effects of warp scheduling among different thread blocks, we will launch only one block with 32 threads. We apply this approach to test all shuffle instructions.

The kernel *sharedMem()* tests the shared memory access latency. The goal of this test is to evaluate the latency instead of throughput; therefore, we launch only one thread block with one thread inside for pointer chasing in the shared memory. The kernel *sharedMemSync()* is used for evaluating the latency of *__syncthreads()*. We add *__syncthreads()* after each iteration in the code. With the same methodology, we can derive the latency of shared memory access and synchronization as shown below.

$$latency_{sharedMem} = latency_{reg} + k_{sharedMem} - k_{reg} \quad (3)$$

$$latency_{sync} = latency_{reg} + k_{sync} - k_{reg} - latency_{sharedMem} \quad (4)$$

On GPUs, register access takes one cycle to finish, therefore, $latency_{reg} = 1$ in all the equations. In the test, we will conduct ten runs with different values of *ITERATIONS*. The results are shown in Figure 3. We test *__shfl_up/down* with different strides and *__shfl* with randomly generated lane IDs.

As we can see, on average the latency of shuffle is in-between that of shared memory access and register access.

Besides, the latency of different types of shuffle instructions varies. For example, on K1200, *shfl_xor* takes longer latency than any other shuffle instruction. Such results indicate that the underlying mechanisms might be different for different shuffle instructions.

The latency pattern of shuffle is consistent on GPUs with the same architecture (K1200 and Titan X, using Maxwell architecture). However, we notice that such latency varies across different architectures. As we can see, on K40 (Kepler architecture), *shfl_xor* is the instruction with the lowest latency among all shuffle instructions, while it is the instruction with the highest latency on Maxwell architecture. It shows that the underlying architecture for shuffle is also modified across different GPU generations.

The results from this microbenchmark help clear up previous confusion regarding shuffle instructions. Clearly, the shuffle instruction is not as fast as direct register access, but it is still faster than shared memory access. The latency also varies across different types of instructions and GPU architectures.

III. Overview of Sequence Alignment Algorithms

In this section we will first introduce the general idea of sequence alignment algorithms, and then describe the details of SW and PairHMM.

A. Algorithm Overview

Sequence alignment algorithms, e.g., Smith-Waterman, Needleman-Wunsch, and PairHMM, have been employed in many application domains such as bioinformatics, finance, language processing, *etc*. The principle of these algorithms is to traverse possible alignments between two sequences and select the alignment with the best score according to certain criteria using a dynamic programming approach. This is done by updating a matrix with the size of $M \times N$, in which M, N are the lengths of two sequences. The computation complexity of these algorithms is $O(MN)$. Each entry in the matrix depends on its neighbors from certain directions. Figure 4 shows the common dependency graph of these algorithms. Each entry depends on its left, up, and left-up neighbors.

To accelerate the application, programmers can explore two kinds of parallelisms: 1) inter-task parallelism and 2) intra-task parallelism. Inter-task parallelism refers to the parallelism among different alignment tasks which are independent of each other while intra-task parallelism refers to the parallelism among different cells on the same anti-diagonal. Figure 4 shows one example of intra-task parallelism. Many previous works [10]–[13] use either one or both of these two kinds of parallelisms to accelerate the computation.

Exploiting intra-task parallelism in these algorithms will introduce intensive inter-thread communication. Therefore, we choose these algorithms as our application drivers to justify the effectiveness of shuffle instructions.

Figure 4: Dependence graph for sequence alignment algorithms.

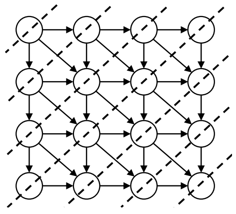
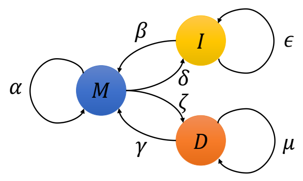


Figure 5: PairHMM model.



B. Algorithm Details

The SW and PairHMM algorithms in this paper are extracted from the HaplotypeCaller [14] of GATK. These two algorithms are used to align DNA sequences for discovering variants in human genes.

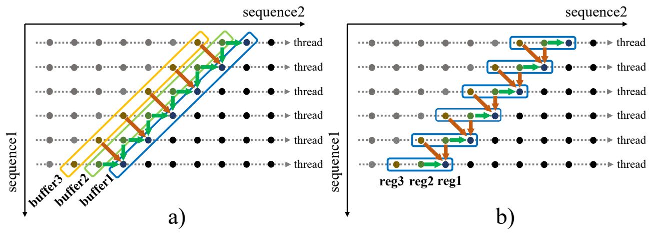
SW [15] is a well-known algorithm for sequence alignment. It is exploited to identify the optimal local alignment between two sequences by means of dynamic programming. Given two sequences s_1 and s_2 of length M and N, it computes the score matrix H as follows.

$$H_{i,j} = \max \begin{cases} 0 \\ H_{i-1,j-1} + s(a_i, b_j) \\ \max_{k \geq 1}\{H_{i-k,j} + W_k\} \\ \max_{l \geq 1}\{H_{i,j-l} + W_l\} \end{cases} \tag{5}$$

where $1 \leq i \leq M$ and $1 \leq j \leq N$. a_i and b_j are i-th and j-th characters in sequence s_1 and s_2, respectively. $s(a, b)$ is a similarity function for two characters, and W_k and W_l are the gap-scoring scheme. For the latter two cases in Equation 5, we maintain two buffers holding the local maximum along each direction so that each time only the left and up neighbors need to be accessed. Meanwhile, a back-tracing matrix $btrack$ recording the paths chosen for each cell is updated. After the computation, we will locate the cell with the maximal value in the last row and column of the score matrix H, and retrieve the optimal alignment of two sequences back from this position using the matrix $btrack$. Note that the conventional SW will find the maximum throughout the whole score matrix, and the algorithm has been modified to adapt to the needs of HaplotypeCaller.

PairHMM [16] is a variant of SW. However, there are several fundamental differences between the two algorithms. PairHMM uses the hidden Markov model to align two sequences and will generate a probability score, measuring the similarity of two sequences. Figure 5 shows the HMM

Figure 6: GPU designs for sequence alignment algorithms with different types of inter-thread communication: a) design A: using shared memory, b) design B: using shuffle.



used for this algorithm. There are three states in total: *match*, *insertion*, and *deletion*. $\alpha, \beta, \gamma, \delta, \epsilon, \zeta, \mu$ are transition probabilities among different states.

Different from SW, in PairHMM, we will compute three score matrices ($match(M)$, $insertion(I)$, and $deletion(D)$) using the following equations.

$$\begin{cases} M_{i,j} = P_{i,j}(\alpha M_{i-1,j-1} + \beta I_{i-1,j-1} + \gamma D_{i-1,j-1}) \\ I_{i,j} = \delta M_{i-1,j} + \epsilon I_{i-1,j} \\ D_{i,j} = \zeta M_{i,j-1} + \mu D_{i,j-1} \end{cases} \tag{6}$$

where $P_{i,j}$ is the prior probability of emitting two characters (a_i, b_j) in the two sequences s_1 and s_2. The sum of all the cells in the last row of matrix I and D is the probability that measures the similarity of two sequences.

In conclusion, compared to SW, there are three matrices (M,I, and D) to update instead of only one score matrix H. Also, there is no back-tracing phase in PairHMM, and the output of the PairHMM will be one single number measuring the similarities between two sequences.

IV. Implementation Details

In this section we will first discuss the general methodology of implementing sequence alignment algorithms in Section III. Then, optimization techniques for both kernels are described. In the end, we present the performance model for analyzing the performance of our designs.

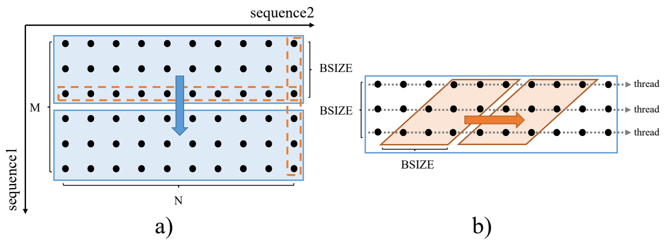
A. Design Methodology

In this section we discuss the general design methodology for algorithms with dependence graph as shown in Figure 4 by exploiting the intra-task parallelism. Shared memory and shuffle are used as two alternatives for inter-thread communication.

Respecting the dependence order between two adjacent anti-diagonals, we will iterate through all the anti-diagonals one by one. Each thread will be assigned one cell on the anti-diagonal.

The inter-thread communication occurs when loading and writing data among threads. This can be implemented either using shared memory or shuffle. Figure 6 shows the designs using shared memory or shuffle (denoted by design A and B). Listing 2 shows the major CUDA code for two designs.

Figure 7: Two-level tiling scheme for SW: a) coarse-grained tiling, b) fine-grained tiling. In a), cells on the boundaries that are surrounded by dashed boxes need to be stored back to the global memory. The data on the horizontal boundaries will be retrieved and consumed by the next block, and data on the vertical boundaries are used for finding the maximal value in the last column of the score matrix.



In design A, we use shared memory to store the data on the anti-diagonals. Data on the same anti-diagonal are stored in one line buffer. This enables coalesced shared memory access from different threads on the same anti-diagonal. From the dependency graph, three line buffers are sufficient for these algorithms, as shown in Figure 6a). After the computation on each anti-diagonal has finished, we will rotate three line buffers and synchronize all the threads before the next iteration.

In design B, data in the anti-diagonals are stored in local registers, and the three line buffers using shared memory in design A are freed up. Each thread will hold three registers, as denoted by *reg1*, *reg2*, *reg3*. These registers store the three cells calculated or to be calculated by the current thread. As shown in Figure 6b), in order to calculate a cell, it will load its left neighbor from *reg2* locally, and its up or left-up neighbors from *reg2* or *reg3* of the adjacent thread, respectively, using shuffle instructions. There is no explicit synchronization at the end of each iteration, because the threads within the warps are implicitly synchronized.

The characteristics of the algorithms and the instructions (shared access vs. shuffle) will bring unique design challenges and opportunities for each kernel. In the following sections, we will describe the techniques we employ to tackle those obstacles and help further improve the performance.

B. Design Optimization of SW

1) Shared Memory: The major challenge for SW is that we need to store the whole back-tracing matrix $btrack$ for retrieving the optimal alignment later. Storing the whole $btrack$ matrix in the shared memory is impossible due to the limited shared memory resource. Therefore, we apply a two-level tiling method to mitigate the problem, which is depicted in Figure 7. We first tile the matrix at the row dimension. The whole matrix will be tiled into blocks of size *BSIZE*$\times$*N*. Cells lying in the boundaries of the block will be stored back to the global memory, as the horizontal boundary

```
__shared__ data_t buf1[BUF_SIZE];
__shared__ data_t buf2[BUF_SIZE];
__shared__ data_t buf3[BUF_SIZE];
for (int diag = 0; diag < DIAG_NUM; diag++) {
        // LOAD
        data_t left = buf2[threadIdx.x];
        data_t up = buf2[threadIdx.x-1];
        data_t leftup = buf3[threadIdx.x-1];
        data_t cur = compute(left, up, leftup);  // COMPUTE
        buf1[threadIdx.x] = cur;  // WRITE
        rotate(buf1, buf2, buf3);  // ROTATE
        __syncthreads();  // SYNC
}
```

```
data_t reg1, reg2, reg3;
for (int diag = 0; diag < DIAG_NUM; diag+) {
        // LOAD
        data_t left = reg2;
        data_t up = __shfl(reg2, threadIdx.x-1);
        data_t leftup = __shfl(reg3, threadIdx.x-1);
        // COMPUTE
        data_t cur = compute(left, up, leftup);
        // WRITE
        reg1 = cur;
        // ROTATE
        reg3 = reg2; reg2 = reg1;
}
```

Listing 2: CUDA code for implementations of using shared memory (design A) and shuffle (design B).

will be used by the next block, and the vertical boundary will be used for searching for the maximal score later. Inside each block, we further tile it into smaller tiles with the shape of parallelograms. Therefore, each tile covers *BSIZE* anti-diagonals. We choose the shape of parallelograms instead of normal rectangles due to the fact that using rectangular tiles will result in low warp efficiency considering most threads will be wasted at the upper left and lower right corners of the tiles.

In the end, we will have three line buffers of length *BSIZE*, and a matrix of size *BSIZE*×*BSIZE* to store the data in *btrack*. We will assign *BSIZE* threads for the task. Each thread will work on one complete row in the matrix, and data along the row can be reused locally inside each thread. The execution order of tiles is depicted in Figure 7.

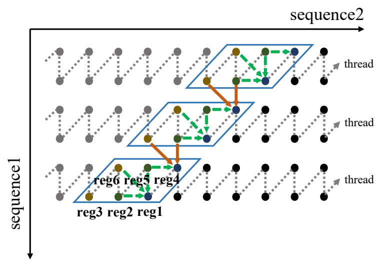
2) Shuffle: The two-level tiling scheme is applied to shuffle as well. Data in the anti-diagonals are stored in local registers, and the previous three line buffers in shared memory are freed up as depicted in Figure 6b).

C. Design Optimization of PairHMM

As discussed in Section III, there are several differences between PairHMM and SW, which may affect the design choices when implementing the kernels. Several architecture modifications are made to adapt to these features.

1) Shared Memory: The implementation for PairHMM is nearly the same as depicted in Figure 6a). The last thread working on the last row will have an additional job as accumulating the results of cells in matrix $match$ and $insertion$. Tiling is no longer needed here, because the shared memory is mostly used by line buffers to hold the intermediate results in anti-diagonals, whose size can be fit on-chip in our experiments. However, based on the observations that sequence lengths vary a lot in our datasets, setting the same line buffer length for all tasks is not efficient. We optimize the performance by duplicating the kernels with several copies, each with different line buffer size. Tasks with different sequence lengths will fall into different kernels at the launch time to efficiently use the shared memory.

Figure 8: Shuffle implementation for PairHMM. In this example, each thread will hold six registers for two cells in total, and compute them one by one on each anti-diagonal. Inter-thread communication only happens between boundary cells.



Similar to SW, each thread will work on one complete row in the matrix, which offers the opportunity of data reuse along the horizontal axis. PairHMM benefits more from this feature, as there are more metadata (e.g., transition probabilities) of sequences used for calculation, and the movement of these data can be saved by data reuse.

2) Shuffle: The implementation that uses the shuffle instruction faces the limitation that this instruction can be only used within one warp. Therefore, if there is more than one warp working on the anti-diagonal, threads in different warps need to communicate with each other using either shared memory or global memory. This will result in branch divergence. Besides, the access of shared/global memory will unfortunately cancel the benefits of using shuffle instructions because every shuffle instruction within the warp is accompanied by one shared/global memory access across the warps. Based on these considerations and experiments, we make the compromise to use 32 threads (one warp) for calculating the whole sequence alignment task. This solution still delivers remarkable performance, as shown in Section V.

Similar to what we have done for SW, we create three registers (*reg1*, *reg2*, *reg3*) for each thread. Threads look up data from neighbors via shuffle instructions. However, using only 32 threads will bring problems for tasks whose sequence lengths are longer than 32. We solve this by assigning multiple cells along the anti-diagonal for each thread, and each thread will compute these cells one by one. We cannot use a fixed number of cells for each thread because this will cause inefficiency across tasks with different sequence lengths. Based on the similar heuristic adopted in the shared memory implementation, we will create subfunctions with different numbers of cells to calculate for each thread, and assign tasks with different subfunctions during the runtime.

Figure 8 depicts one example when each thread needs to compute two cells on the anti-diagonal. This scheme will bring even more benefits for performance. As we can see, inter-thread communication only takes place between boundary cells across different threads. For the rest of the cells, communication can finish by using direct register access, with the lowest access latency among all the data access methods on GPU.

D. Performance Model

In this section we present the performance model which provides a quantitative perspective to analyze the performance of different kernels for accelerating sequence alignment algorithms.

CUPs (cell update per second) is the widely used metric for measuring the performance of sequence alignment algorithms. It measures the number of cells in the matrix that can be computed per second. We adopt this metric as the measurement of kernel performance in this work. Note that for PairHMM, we count three updates in three matrices (M, I and D) as one cell update for simplicity.

The performance model is shown as below.

$$performance(CUPs) = \frac{parallelism \times frequency}{latency} \tag{7}$$

$parallelism$ is defined as the number of cells updated in parallel. If each thread is assigned to update one cell in the matrix, this factor equals the active threads lying in all the SMs on GPU. This factor can be calculated using the equation below.

$$\begin{aligned} parallelism = \#SM \times min\{ & \frac{\#reg/SM}{\#reg/thread}, \\ & \frac{\#sharedMem/SM}{\#sharedMem/threadBlock} \times \frac{thread}{threadBlock}\} \end{aligned} \tag{8}$$

where $\#reg/SM$, $\#sharedMem/SM$, and $\#SM$ are platform-dependent information, $\#reg/thread$, $\#sharedMem/threadBlock$ and $thread/threadBlock$ are kernel characteristics, which can be derived from kernels with the help of the Nvidia nvcc compiler by setting specific compilation flags. Note that this factor is in proportional to *occupancy* as well.

The factor $frequency$ is gathered from hardware specification. $latency$ refers to the average time interval to finish one cell. More specifically, when every cell on the anti-diagonal is assigned to one thread, it refers to the latency to finish the entire anti-diagonal. The latency is dominated by the critical path in each iteration which includes 1) *load* data from neighbor cells, 2) *compute* the current cell, and 3) *write* back the current cell.

This performance model is intuitively simple and can be quite handy when estimating and analyzing the performance of kernels. In our work, this model serves two purposes. First, it helps justify the validity of our microbenchmarks when testing the latency of different instructions. After gathering the performance of kernels and kernel characteristics, we can calculate the $latency$ and compare it to the estimated $latency$ using results from our microbenchmarks. Second, it helps programmers estimate the performance using different communication methods. With estimated $parallelism$ using Equation 8 and $latency$ from our microbenchmarks, CUDA programmers can easily make trade-offs in advance before taking efforts to implement a new kernel using shuffle instructions.

V. Experimental Results

In this section we first introduce the setup for experiments and present the overall performance of our designs on different platforms. Then we use the performance of the designs to validate the microbenchmarks with the help of the performance model. We discuss the trade-offs of using shuffle instructions in the end.

A. Experiment Setup

The kernel performance is evaluated on two Nvidia GPUs—Quadro K1200 and Titan X. Both K1200 and Titan use the Maxwell architecture, while K1200 is a low-end GPU with high energy efficiency, and Titan X is a high-end GPU with high computation capability.

The algorithms of SW and PairHMM are extracted from GATK 3.6. We use the genome sample of a human with breast cancer (HCC1954) as the inputs of HaplotypeCaller and dump out the data as input datasets for two kernels.

All of the GPU kernels are written in CUDA 7.5. The Nvidia nvcc compiler is used to compile the code and get kernel characteristics (e.g., reg and shared memory usage). For the convenience of illustration, in the following sections we will denote SW implementations using shared memory or shuffle as SW1 and SW2, and PairHMM implementations as PH1 and PH2.

B. Performance Overview

The DNA sequence is broken into regions to be analyzed in order in HaplotypeCaller. For each region, HaplotypeCaller will have two intermediate stages, generating several pairs of sequences to align, using SW and PairHMM, respectively. Each pair of sequences is denoted as a *task*. We dump out the tasks for SW, and group them together as batches for each region. The input datasets for PairHMM

Figure 9: Performance overview of GPU implementations: a) SW, b) PairHMM.

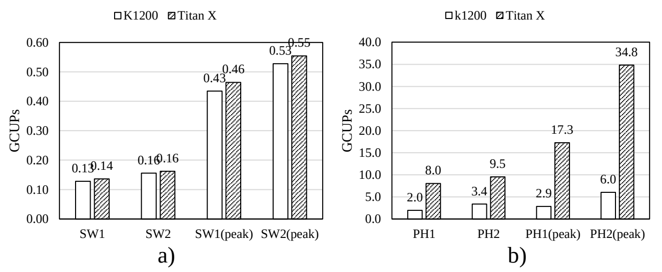
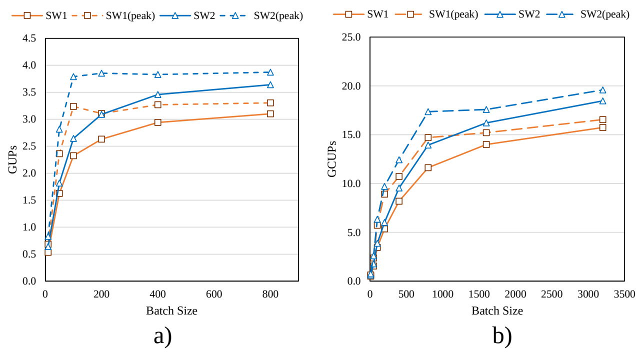


Figure 10: Impacts of re-batching on performance of SW kernels: a) K1200, b) Titan X.



are generated in the same fashion. For each kernel, the number of tasks in each batch varies depending on the DNA clips being analyzed. In our datasets, the average number of tasks per batch for SW is four, whereas the number is 189 for PairHMM. The insufficient tasks for SW will limit the performance, as discussed later.

We measure the GCUPs performance for each batch and take the average. For GPU implementations, each task is assigned to one thread block, and all the tasks within the same batch are launched together as a compute kernel. We set *BSIZE* as 32 for both SW1 and SW2, which offer the best performance from our experiments. We set 128 threads/threadBlock for PH1 because the maximal sequence length is less than 128, and 32 threads/threadBlock for PH2. Note that the GPU performance reported below includes the data transfer time between the host and device.

Figure 9 shows the average and maximal performance of all the kernels. Note that the performance for SW is relatively low, because the GPU performance is severely impacted by the batch size, as mentioned above. Therefore, we break the boundary of regions and re-batch the tasks in different regions together to evaluate SW kernels. Results are shown in Figure 10.

As we can see, our SW kernels deliver significant performance. On Titan X, design SW2 achieves the peak performance of 19.6 GCUPs and an average performance of 18.5 GCUPs when re-batching 3,200 tasks together.

Table II: Detailed information of kernels.

	SW1	SW2	PH1	PH2
GCUPs	3.6	4.3	3.2	6.8
occupancy(%)	32.2	49.7	56.2	29.1
#reg/thread	54	56	56	94
#sharedMem/threadBlock	1868	1472	6764	2644
latency(cycle)	1203	1014	1528	379
reduction(cycle)		189		1149

Table III: Instruction breakdown and analysis for latency reduction.

operation	instruction	#instruction			
		SW1	SW2	PH1	PH2
LOAD	SMEM/(shfl,reg)	3	(2,1)	32	(6,25)
WRITE	SMEM/reg	1	1	12	12
ROTATE	SMEM/reg	2	2	24	24
SYNC		1	0	1	0
	latency	183	22	1485	115
	est. reduction	161(-14.8%)		1370(19.2%)	

As for PairHMM, on Titan X design PH2 can achieve the peak performance of 34.8 GUCPs and an average performance of 6.0 GCUPs. For all the implementations, using shuffle instructions delivers better performance than those using shared memory instead.

C. *Model Validation*

In this section we will use our performance model to validate the results from microbenchmarks. We select the biggest batch among the original datasets (w/o re-batching) as the inputs and K1200 as the target platform. This guarantees that the GPU is fully occupied by tasks so that our analysis will not be affected by factors other than computation itself. We conduct several repeated runs for each kernel and take the average as the kernel performance. This number doesn't include the data transfer because our focus is the computation part of the kernel.

Table II presents the kernel performance and statistics. Using our performance model, we can compute the average latency of each iteration for four kernels.

For SW, using shuffle can cut down the iteration latency by 189 cycles. The reduction comes from: 1) data access latency, and 2) synchronization. Table III shows the detailed breakdown of the latency.

In the shared memory implementation shown in Listing 2a), there are three shared memory loads and one write in each iteration. Two more accesses are required for rotating three line buffers, and one $\_\_syncthreads()$ at the end of each iteration. Based on our microbenchmarks, shared access takes around 21 cycles, and $\_\_sycnthreads()$ takes 57 cycles. We can estimate the latency as $3 \times 21 + 1 \times 21 + 2 \times 21 + 1 \times 57 = 183$ cycles. In SW2, three loads from neighbor cells are replaced by two shuffles and one direct register access, as shown in Listing 2b). All the other instructions can be replaced by direct register accesses; the synchronization is eliminated. The estimated latency will be $2 \times 9 + 4 \times 1 = 22$ cycles. Therefore, the estimated latency reduction is $183 - 22 = 161$ cycles. The relative error of our analysis is -14.8%.

For PairHMM, in each iteration there are three matrices to update (eight loads in total: three in $match$, three in $insertion$, and two in $deletion$). Additionally, 128 threads (4 warps) are used to update one anti-diagonal, which will issue $8 \times 4 = 32$ shared memory instructions each time. As for design PH2, each thread will compute 4 cells in total. Inter-thread communication happens between boundary cells, which brings two shuffle instructions and one register access for each matrix (only two shuffle for $deletion$). For all the remaining cells inside, direct register access suffices. There will be six shuffle instructions and 25 register accesses in total. The analysis for other operations are similar to SW. Based on our estimation, using shuffle instruction helps reduce latency by 1370 cycles. The relative error is 19.2%.

The relative error for prediction for both designs is low, which validates results of the microbenchmarks. This analysis shows a normal flow for CUDA programmers to estimate the performance gains when using shuffle instructions. We can first estimate the new register and shared memory usage when using shuffle instructions and compute the factor $parallelism$. Then, we can calculate the $latency$ based on the computation breakdown and latency of instructions from microbenchmarks. Finally, we can use our model to compute the performance of shuffle designs.

D. Trade-Off Analysis

Using shuffle instructions helps improve performance over designs using shared memory. As shown in Table II, using shuffle instructions provides performance gains of $1.2\times$ and $2.1\times$ for SW and PairHMM, respectively.

For SW, using shuffle instructions helps save shared memory, and therefore increases the occupancy ($parallelism$). Meanwhile, $latency$ is reduced. Both factors contribute to the performance improvement of SW2 over SW1.

As for PairHMM, we observe a drop of occupancy from PH1 to PH2. The reason is that we assign more cells for each thread to calculate, thus significantly increasing the register usage for each thread. The register usage becomes the limiter of occupancy and drags down the occupancy from 56.2% to 29.1%. Meanwhile, inter-thread communication in PairHMM is more intensive than SW, as we will access more data (three matrices vs. one matrix) with more threads (128 threads/threadBlock vs. 32 threads/threadBlock). The reduction of $latency$ from using shuffle instructions is larger than SW, which offsets the decrease of $parallelism$ and improves $performance$ eventually.

Based on the analysis above, we conclude that the trade-offs of using shuffle instructions are as follows:

- Using register shuffle can free up shared memory. However, the impact on occupancy varies among different applications. It is possible that the increased number of registers may become the new limiter of occupancy, which will hurt the $parallelism$ we can obtain.
- In terms of performance, both $parallelism$ and $latency$ matter. For applications with intensive inter-thread parallelism, the reduction of the $latency$ using shuffle instructions plays an important role in overall $performance$, which could even offset the negative impacts of using more registers and bring performance gains in the end.

With the help of the performance model and the microbenchmarks for shuffle instructions, CUDA programmers can easily handle such trade-offs and make design decisions in advance. Note that the root cause of the first point lies in the limitation of shuffle instructions since they can be only used for threads within the same warp. Therefore, for applications like PairHMM, we need to place more cells to calculate for each thread, with more registers to use per thread. This will significantly increase the register usage and affect the occupancy eventually.

VI. Related Work

The shuffle instruction offers a new alternative for inter-thread communication. Previous works [5]–[7] leverage shuffle instructions as a replacement for shared memory operations, and have seen performance gains from such optimization. However, there is no detailed analysis about the root causes for such benefits. Our work extends the shuffle instructions to a new application domain, sequence alignment, and is the first work to conduct a systematic and quantitative analysis of shuffle instructions.

In this paper we pick SW and PairHMM as two application drivers which present dependency patterns of near-neighbor communication. SW is a well-known sequence alignment algorithm, and there are many GPU implementations for different variants of SW. Manavski et al. [11] utilize the inter-task parallelism by assigning each GPU thread one entire alignment task. The sequences need to be sorted in advance so that the task for threads within the same block can be as similar as possible. Liu et al. [10] propose a combined solution which couples CPU and GPU together to accelerate SW protein search. They use the same programming model as Manavski et al. and further utilize the SIMD instructions to improve parallelism. The SW application we adopt in the paper is different from these works in terms of both the algorithm and datasets.

PairHMM is a variant of SW which integrates HMM into the algorithm. There have been several previous works [12], [17] on CPU and FPGA that employ the intra-task parallelism along the anti-diagonals. Intel released Genomics Kernel Library (GKL) [13] which uses AVX intrinsics to accelerate the algorithm. Ito et al. [12] propose a systolic array architecture on FPGA that uses the IBM CAPI interface and delivers the performance of 1.7 GCUPs on the same genome sample that we use in this paper. Our implementation in this work outperforms all the previous works on PairHMM.

VII. Conclusion

Data movement is one of the critical limiting factors of performance and energy efficiency. In this work we look into the communication optimization methodology for applications with intensive inter-thread communication. The shuffle instruction offers new alternatives in addition to conventional methods for using shared memory and global memory, and brings trade-offs to be considered at the same time. In this work we conduct a quantitative analysis on shuffle instructions, using two sequence alignment algorithms (SW and PairHMM) as case studies.

A suite of microbenchmarks is developed for measuring the latency of shuffle and several other instructions. We find that the latency of shuffle is in-between that of register and shared memory access, and it varies across different types of shuffle instructions and architectures (Kepler vs. Maxwell).

We implement two algorithms using either shared memory or register shuffle for inter-thread communication. Using shuffle instructions instead of shared memory has brought significant performance gains of 1.2× and 2.1× for SW and PairHMM, respectively. This demonstrates the optimization opportunities for deploying shuffle instructions for applications with intensive inter-thread communication.

We are proposing a performance model that takes kernel characteristics and the instruction latency into consideration, and helps analyze and estimate the design performance for such algorithms. With the help of this performance model and microbenchmarks, we conduct a detailed analysis on the performance impacts of shuffle instructions.

This work provides valuable insights for CUDA programmers for making trade-offs when using shuffle instructions in a wider class of applications.

VIII. Acknowledgement

The authors would like to thank Muhuan Huang and Janice Martin-Wheeler for editing the paper. This work was supported in part by C-FAR, one of the six SRC STARnet Centers, sponsored by MARCO and DARPA, NSF/Intel Innovation Transition Grant (CCF-1436827) awarded to the Center for Domain-Specific Computing, and contributions from Fujitsu Laboratories.

References

[1] W.-m. Hwu, "What is ahead for parallel computing," *Journal of Parallel and Distributed Computing*, vol. 74, no. 7, pp. 2574–2581, 2014.

[2] S. Xiao and W. c. Feng, "Inter-block gpu communication via fast barrier synchronization," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, April 2010, pp. 1–12.

[3] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August, "Automatic cpu-gpu communication management and optimization," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 142–151.

[4] Nvidia. (2016) Cuda c programming guide. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz4NVazmUbb

[5] Y. Hanada, S. Kitaoka, and Y. Xinhua, "Optimizing particle simulation for kepler gpu," *Procedia Engineering*, vol. 61, pp. 376 – 380, 2013.

[6] D. Bakunas-Milanowski, V. Rego, J. Sang, and C. Yu, "A fast parallel selection algorithm on gpus," in *2015 International Conference on Computational Science and Computational Intelligence (CSCI)*. IEEE, 2015, pp. 609–614.

[7] H. Jiang and N. Ganesan, "Fine-grained acceleration of hmmer 3.0 via architecture-aware optimization on massively parallel processors," in *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*. IEEE, 2015, pp. 375–383.

[8] B. Institute. (2016) Genome analysis toolkit. [Online]. Available: https://software.broadinstitute.org/gatk/

[9] N. Mark Harris, "Cuda pro tip: Do the kepler shuffle," 2014. [Online]. Available: https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-kepler-shuffle/

[10] G. Lu and J. Ni, "Highlighting computations in bioscience and bioinformatics: review of the symposium of computations in bioinformatics and bioscience (scbb07)," *BMC Bioinformatics*, vol. 9, no. 6, p. S1, 2008.

[11] S. A. Manavski and G. Valle, "Cuda compatible gpu cards as efficient hardware accelerators for smith-waterman sequence alignment," *BMC Bioinformatics*, vol. 9, no. 2, p. S10, 2008.

[12] M. Ito and M. Ohara, "A power-efficient fpga accelerator systolic array with cache-coherent interface for pair-hmm algorithm," in *2016 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS XIX)*. IEEE, 2016, pp. 1–3.

[13] Intel. (2016) Genomics kernel library (gkl). [Online]. Available: https://github.com/Intel-HLS/GKL

[14] B. Institute, "Haplotypecaller," 2016. [Online]. Available: https://software.broadinstitute.org/gatk/gatkdocs/org_broadinstitute_gatk_tools_walkers_haplotypecaller_HaplotypeCaller.php

[15] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of molecular biology*, vol. 147, no. 1, pp. 195–197, 1981.

[16] R. Durbin, S. R. Eddy, A. Krogh, and G. Mitchison, *Biological sequence analysis: probabilistic models of proteins and nucleic acids*. Cambridge university press, 1998.

[17] S. Ren, V. M. Sima, and Z. Al-Ars, "Fpga acceleration of the pair-hmms forward algorithm for dna sequence analysis," in *Bioinformatics and Biomedicine (BIBM), 2015 IEEE International Conference on*, Nov 2015, pp. 1465–1470.