CUDAMPF++: A Proactive Resource Exhaustion Scheme for Accelerating Homologous Sequence Search on CUDA-Enabled GPU

Hanyu Jiang, *Student Member, IEEE*, Narayan Ganesan, *Senior Member, IEEE*, and Yu-Dong Yao, *Fellow, IEEE*

Abstract—Biological sequence alignment is an important research topic in bioinformatics and continues to attract significant efforts. As biological data grow exponentially, however, most of alignment methods face challenges due to their huge computational costs. HMMER, a suite of bioinformatics tools, is widely used for the analysis of homologous protein and nucleotide sequences with high sensitivity, based on profile hidden Markov models (HMMs). Its latest version, HMMER3, introduces a heuristic pipeline to accelerate the alignment process, which is carried out on central processing units (CPUs) and highly optimized. Only a few acceleration results are reported on the basis of HMMER3. In this paper, we propose a five-tiered parallel framework, CUDAMPF++, to accelerate the most computationally intensive stages in HMMER3's pipeline, multiple/single segment Viterbi (MSV/SSV), on a single graphics processing unit (GPU) without any loss of accuracy. As an architecture-aware design, the proposed framework aims to fully utilize hardware resources via exploiting finer-grained parallelism (multi-sequence alignment) compared with its predecessor (CUDAMPF). In addition, we propose a novel method that proactively sacrifices L1 Cache Hit Ratio (CHR) to get improved performance and scalability in return. A comprehensive evaluation shows that the proposed framework outperforms all existing work and exhibits good consistency in performance regardless of the variation of query models or sequence datasets. For MSV (SSV) kernels, the peak performance of CUDAMPF++ is 283.9 (471.7) GCUPS on a single K40 GPU, and impressive speedups ranging from 1.8x (1.7x) to 168.3x (160.7x) are achieved over the CPU-based implementation (16 cores, 32 threads).

Index Terms—GPU, CUDA, SIMD, L1 cache, hidden Markov model, HMMER, MSV, SSV, viterbi algorithm

1 Introduction

Typical algorithms and applications in bioinformatics, computational biology and system biology share a common trait that they are computationally challenging and demand more computing power due to the rapid growth of biological data and the need for high fidelity simulations. As one of the most important branches, the biological sequence analysis with various alignment methods scales the abstraction level from atoms to RNA/DNA molecules and even whole genomes, which aims to interpret the similarity and detect homologous domains amongst sequences [1]. For example, the protein motif detection is key to identify conserved protein domains within a known family of proteins. This paper addresses HMMER [2], [3], a widely used toolset designed for the analysis of homologous protein and nucleotide sequences with high sensitivity, which is originally carried out on central processing units (CPUs).

HMMER is built on the basis of probabilistic inference methods with profile hidden Markov models (HMMs) [3]. Particularly, the profile HMM used in HMMER is Plan-7 architecture that consists of five main states (*Match*(*M*), *Insert*(*I*), *Delete*(*D*), *Begin*(*B*) and *End*(*E*)) as well as five special states (*N*, *C*, *J*, *S* and *T*). The *M*, *I* and *D* states, which are in the same position, form a node, and the number of nodes included in a profile HMM indicates its length. The digital number "7" in Plan-7 refers to the total of seven transitions per node, which exist in the architecture and each has a transition probability. In addition, some states also have emission probabilities. This architecture is a little bit different from the original one proposed by Krogh et al. [4] which contains extra *I*-*D* and *D*-*I* transitions.

The profile HMMs employ position-specific *Insert* or *Delete* probabilities rather than gap penalties, which enables HMMER to outperform BLAST [5] on sensitivity [3]. However, the previous version of HMMER, HMMER2, suffers from the computational expense and achieves lower utilization than BLAST. Due to well-designed heuristics, BLAST is in the order of 100x to 1000x faster than HMMER2 [3]. Therefore, numerous acceleration efforts have been made for HMMER2, such as [6], [7], [8], [9], [10]. Most of them employ application accelerators and co-processors, like field-programmable gate array (FPGA), graphics processing unit (GPU) and other parallel infrastructures, which provide good performance improvement. To popularize HMMER for

• *The authors are with the Department of Electrical and Computer Engineering, Stevens Institute of Technology, Hoboken, NJ 07030. E-mail: {hjiang5, nganesan, Yu-Dong.Yao}@stevens.edu.*

Manuscript received 9 July 2017; revised 8 Apr. 2018; accepted 20 Apr. 2018. Date of publication 26 Apr. 2018; date of current version 7 Sept. 2018. (Corresponding author: Hanyu Jiang.) Recommended for acceptance by A. Kalyanaraman. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TPDS.2018.2830393

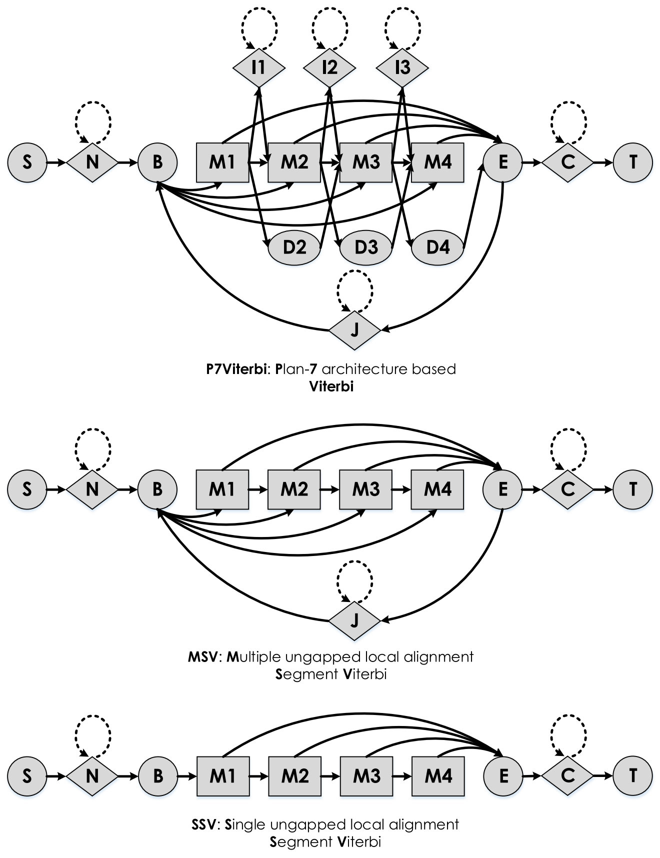

I1
I2
I3
S
N
B
M1
M2
M3
M4
E
C
T
D2
D3
D4
J
P7Viterbi: Plan-7 architecture based Viterbi
MSV: Multiple ungapped local alignment Segment Viterbi
SSV: Single ungapped local alignment Segment Viterbi

Fig. 1. Profiles of P7Viterbi, MSV and SSV models.

standard commodity processors, Eddy et al. propose new versions of HMMER, HMMER3 (v3.0) and its subsequent version (v3.1), which achieve the comparable performance as BLAST [3]. As the main contribution, HMMER3 implements a heuristic pipeline in *hmmsearch* which aligns a query model with the whole sequence dataset to find similar sequence matches. The heuristic acceleration pipeline is highly optimized on CPU-based systems with the support of multi-threading and streaming SIMD extensions (SSE) instructions, and hence only a few acceleration attempts, including [11], [12], [13], [14], [15], [16] and [17], report further speedups.

Our previous work [18], CUDAMPF, proposes a multi-tiered parallel framework to accelerate HMMER3's pipeline on a single GPU, which was shown to exceed the current state-of-the-art. However, the performance evaluation shows that the throughput of the computational kernel depends on the model length, especially for small models, which implies underutilization of the GPU. This inspires us to exploit finer-grained parallelism, compared with the framework presented in [18]. In this paper, we introduce another tier of parallelization that aims to fully take advantage of the hardware resources provided by single GPU. A novel optimization strategy that proactively utilizes the on-chip cache system is proposed to further boost kernel throughput and improve scalability of the framework. A comprehensive evaluation indicates that our method exhibits good consistency of performance regardless of query models and sequence datasets. The limitation and generalization of the proposed framework are also discussed.

The rest of the paper is organized as follows. Section 2 presents background of HMMER3's pipeline, GPU architecture and highlighted CUDA features, followed by a review of CUDAMPF implementation. In Section 3, the in-depth description of our proposed framework is presented. Then, we give comprehensive evaluations and analyses in Section 4. Related work and discussions are presented in Sections 5 and 6, respectively. Finally, we present the conclusion of this paper. Extra experimental results and evaluations can be found in appendixes.

2 Background

In this section, we go through the new heuristic pipeline of HMMER3 and highlight its computationally intensive stages. An overview of GPU architecture and CUDA programming model is also presented. For better understanding of subsequent ideas, we briefly review our previous work, CUDAMPF, at the end of this section.

2.1 Heuristic Pipeline in HMMER3

The main contribution that accelerates the HMMER3 is a new algorithm, multiple segment Viterbi (MSV) [3], which is derived from the standard Viterbi algorithm. The MSV model is a kind of ungapped local alignment model with multiple hits, as shown in Fig. 1, and it is achieved by pruning *Delete* and *Insert* states as well as their transitions in the original profile HMMs. The M-M transitions are also treated as constants of 1. In addition to the MSV algorithm, another simpler algorithm, single segment Viterbi (SSV), is also introduced to boost the overall performance further. Given that the J state is the bridge between two matched alignments, the SSV model assumes that there is rarely a matched alignment with a score that is higher than the cost of going through the J state, and hence it speculatively removes the J state to gain a significant speedup [19]. However, in order to avoid false negatives, the SSV model is followed by regular MSV processing to re-calculate suspected sequences. Fig. 1 illustrates profiles of P7Viterbi, MSV and SSV models with an example of 4 nodes. The solid arrows indicate transitions between different types of states whereas dashed arrows represent the self-increase of a state.

In the pipeline, SSV and MSV models work as heuristic filters (stages) that filter out most of non-homologous sequences. All sequences are scored during SSV and MSV stages, and only about 2.2 percent of sequences are passed to the next stage, given a threshold. The second stage with the P7Viterbi model only allows roughly 0.1 percent of sequences pass, and remaining sequences are then scored by full Forward algorithm [3]. These four stages mentioned above are major components of HMMER3's pipeline. However, SSV and MSV stages consumes more than 70 percent of the overall execution time [17], [18], and hence they are main objectives of acceleration. The theoretical speedup gain for whole application (pipeline), $S_{overall}$, can be estimated by Amdahl's law [20]:

$$S_{overall} = \frac{1}{(1-p) + \frac{p}{S_{part}}}, \tag{1}$$

where p and S_{part} represent the proportion of the overall execution time for target portion and its corresponding speedup rate with GPU acceleration, respectively. Assuming that 10x speedup is achieved for MSV and SSV stages,

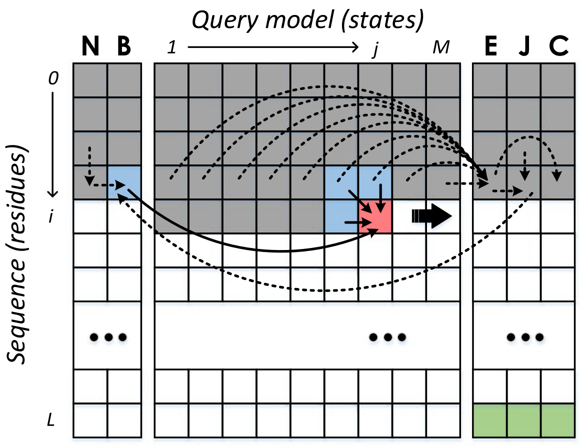


Fig. 2. Dynamic programming matrix of the P7Viterbi stage.

for example, the whole pipeline will gains about 2.7x speedup eventually. Notice that the goal of this paper is to exploit finer-grained parallelism with the architecture-aware design to accelerate specific algorithms (stages) on the GPU. The assembly work for pipeline integration, however, is out of scope and not included in this paper.

Fig. 2 illustrates the dynamic programming (DP) matrix of the P7Viterbi stage, corresponding to Fig. 1. A lattice of the middle region contains three scores for *Match*, *Insert* and *Delete* states, respectively, whereas flanking lattices only have one. To complete the alignment of a sequence, we need to calculate every lattice of the DP matrix starting from the left-top corner to the right-bottom corner (green) in a row-by-row order, which is a computationally intensive process. In the middle region of the DP matrix, each lattice (red) depends on four cells (blue) directly, denoted by solid arrows, which can be formulated as:

$$
\begin{aligned}
V_M[i,j] &= \epsilon_M + \max\begin{cases} V_M[i-1,j-1] + T(M_{j-1}, M_j), \\ V_I[i-1,j-1] + T(I_{j-1}, M_j), \\ V_D[i-1,j-1] + T(D_{j-1}, M_j), \\ B[i-1] + T(B, M_j) \end{cases} \\
V_I[i,j] &= \epsilon_I + \max\begin{cases} V_M[i-1,j] + T(M_j, I_j), \\ V_I[i-1,j] + T(I_j, I_j) \end{cases} \\
V_D[i,j] &= \max\begin{cases} V_M[i,j-1] + T(M_{j-1}, D_j), \\ V_D[i,j-1] + T(D_{j-1}, D_j), \end{cases}
\end{aligned}
\tag{2}
$$

where ϵ denotes emission scores. V and T represent scores of $M/I/D$ states and transitions, respectively. As for MSV and SSV stages, the mathematical formula can be simplified via removing V_I and V_D, which results in moderate dependencies and fewer amount of computation than the P7Viterbi stage. However, in order to exceed the performance of the highly optimized CPU-based implementation with the SIMD vector parallelization, it is imperative to go beyond general methods and exploit more parallelism on other multi/many-core processor architectures.

2.2 GPU Architecture and CUDA Programming Model

As parallel computing engines, CUDA-enabled GPUs are built around a scalable array of multi-threaded streaming multiprocessors (SMs) for large-scale data and task parallelism, which are capable of executing thousands of threads in the *single-instruction multiple-thread* (SIMT) pattern [21]. Each generation of GPU introduces more hardware resources and new features, which aims to deal with the ever-increasing demand for computing power in both industry and academia. In this paper, we implement our design based on Kepler architectures (3rd generation) which equip with multiple powerful streaming multiprocessors, also known as SMXs. For example, a SMX of Kepler GK110 architecture contains 192 single-precision CUDA cores [22]. It also offers another 48KB on-chip read-only texture cache with an independent datapath from the existing L1 and shared memory datapath, and the maximum amount of available registers for each thread is increased to 255 instead of prior 63 per thread. Moreover, a set of Shuffle instructions that enables a warp of threads to share data without going through shared memory are also introduced. This new feature is heavily used in our proposed framework.

The CUDA programming model is designed for NVIDIA GPUs, and it provides users with a development environment to easily leverage horsepower of GPUs. In CUDA, a *kernel* is usually defined as a function that is executed by all CUDA *threads* concurrently. Both *grid* and *block* are virtual units that form a thread hierarchy with some restrictions. Although CUDA allows users to launch thousands of threads, only a warp of threads (32 threads, currently) guarantees that they keep executions in lockstep, which is scheduled by a warp scheduler. Hence, the full efficiency is achieved only if all threads within a warp have the same execution path. Traditionally, in order to make sure that threads keep the same pace, a barrier synchronization has to be called explicitly, which imposes additional overhead.

2.3 CUDAMPF

In [18], we proposed a four-tiered parallel framework, CUDAMPF, implemented on single GPU to accelerate SSV, MSV and P7Viterbi stages of *hmmsearch* pipeline. The framework describes a hierarchical method that parallelizes algorithms and distributes the computational workload considering available hardware resources. CUDAMPF is completely *warp-based* that regards each resident warp as a compute unit to handle the exclusive workload, and hence the explicit thread-synchronization is eliminated. Instead, the built-in warp-synchronism is fully utilized. A warp of threads make the alignment of one sequence one time and then pick up next scheduled sequence. Given that 8-bit or 16-bit values are sufficient to the precision of algorithms, we couple SIMT execution mechanism with SIMD video instructions to achieve 64 and 128-fold parallelism within each warp. In addition, the runtime compilation (NVRTC), first appeared in CUDA v7.0, was also incorporated into the framework, which enabled switchable kernels and innermost loop unrolling to boost the performance further. CUDAMPF yields upto 440, 277 and 14.3 GCUPS (giga cells updates per second) with strong scalability for SSV, MSV and P7Viterbi kernels, respectively.

3 Proposed Framework: CUDAMPF++

This section presents detailed implementations of the proposed framework, CUDAMPF++, that is designed to gain more parallelism based on CUDAMPF. The framework still

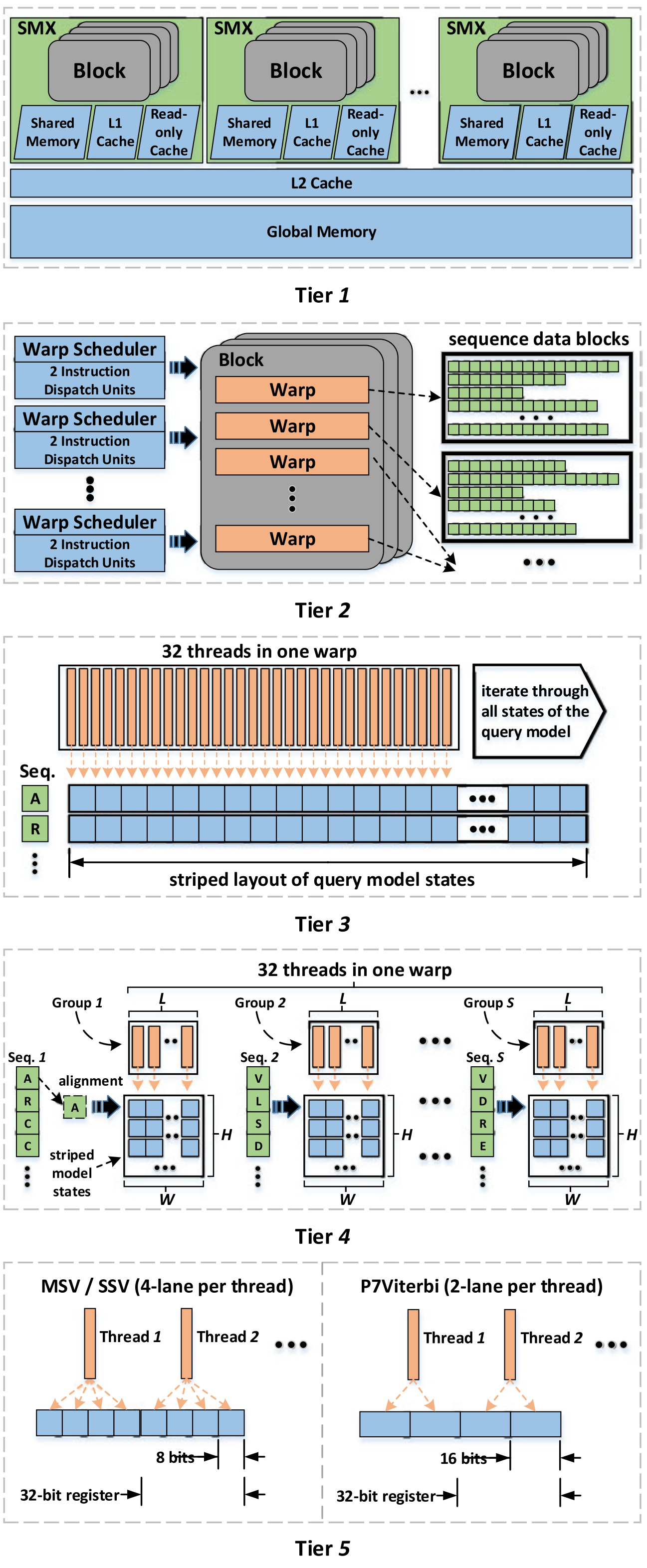


Fig. 3. Five-tiered parallel framework based on the NVIDIA Kepler architecture.

has a clear hierarchy in which each tier performs a different granularity of parallelism. We first introduce a new tier of parallelism followed by a data reformatting scheme, and then in-depth explanations of kernel design are presented. Finally, we discuss advanced kernel optimizations followed by a summary of significant differences between CUDAMPF and the proposed CUDAMPF++.

3.1 Five-Tiered Parallelism

In CUDAMPF, the four-tiered parallel framework is proposed to implement MSV, SSV and P7Viterbi kernels. Although the performance improvement is observed on all accelerated kernels, the speedup on P7Viterbi kernel is very limited whereas MSV/SSV kernel yields significant improvement. Given the profiling information [18], we are able to gain additional insights into the behaviors: (a) L1 Cache Hit Ratio (CHR) of the P7Viterbi kernel degrades rapidly as model size increases, and (b) its register usage always exceeds the maximum pre-allocation for each thread, which indicates the exhaustion of on-chip memory resources and serious register spill. As for MSV/SSV kernels, however, the on-chip memory resources are sufficient. A large amount of low-latency registers, especially when aligning with small models, are underutilized. This can also be proved by performance curves in [18] in which only upward slopes are observed without any flat or downward trends as model size increases from 100 to 2405. The underutilization leaves an opportunity to exploit further parallelism that can fully take advantage of hardware resources on GPUs.

In addition to original four-tiered structure, another tier of parallelism, inserted between the 3rd and 4th tiers of CUDAMPF, is proposed to enable each warp handle multiple alignments with different sequences in parallel while the design of the CUDAMPF only allows single-sequence alignment per warp. This scheme aims to exhaust on-chip memory resources, regardless of the model length, to maximize the throughput of MSV/SSV kernels. Fig. 3 illustrates the five-tiered parallel framework. Notice that Tier 4 of Fig. 3 is that newly added one whereas original Tier 4 of CUDAMPF is marked as Tier 5 now in Fig. 3.

The first tier is based on multiple SMXs that possesses plenty of computing and memory resources individually. Given the non-existence of data and execution dependency between each sequence, it is straightforward to partition whole sequence database into several chunks and distribute them to SMXs, as the most basic data parallelism. Tier 2 describes the parallelism between multiple warps that reside on each SMX. Since our implementation still applies the warp-synchronous execution that all warps are assigned to process different sequences without inter-warp interactions, explicit synchronizations are eliminated completely. Unlike the CUDAMPF in which warps move to their next scheduled task once the sequence at hand is done, the current design allocates a sequence data block to each warp in advance. This change is shown in Tier 2 of Fig. 3. A data block may contain thousands of sequences, less or more, depending on the total size of sequence dataset and available warps. For multi-sequence alignment, all sequences within each data block need to be reformatted as the striped layout, which enables coalesced access to the global memory. Details of data reformatting will be discussed in Section 3.2. The number of resident warps per SMX is still fixed to 32 due to the complexity of MSV and SSV algorithms, which ensures that every thread obtains more registers to handle complex execution dependencies and avoids excessive register spill. Thus, the concept of thread block is completely ignored in our design. In other words, no matter how many thread blocks are launched and assigned to each SMX, the total amount of resident warps per SMX always keep unchanged. Besides, each SMX contains 4 warp schedulers, each with dual instruction dispatch units [22], and hence it is able to dispatch upto 8 independent instructions each cycle. Those hardware resources have the critical influence on the parallelism of Tier 2 and the performance of warp-based CUDA kernels.

Tier 3 is built on the basis of warps. A warp of threads update different model states simultaneously and iterate over remaining model states in batches. The number of iterations depends on the query model size. Once the alignment of an amino-acid residue is done, such as the 'A' and its alignment scores (marked as a row of blue lattices) shown in Fig. 3 (Tier 3), the warp moves to next residue 'R' and starts over the alignment until the end of current sequence. On the basis of Tier 3, Tier 4 illustrates the multi-sequence alignment that a warp of threads are evenly partitioned into several groups, each has L threads, to make alignments with S different sequences in parallel. For example, the first residues of S sequences, like 'A' of Seq. 1, 'V' of Seq. 2 and 'V' of Seq. S, are extracted together for the first-round alignment. Each group of threads update W scores per iteration, and H iterations are required to finish one alignment. The model states are formatted as a rectangle with $W \times H$ lattices. Considering two models, a large model M_l and a small model M_s where the size of M_l is S times larger than the size of M_s, we are able to get $W_s = \frac{1}{S} W_l$ given $H_s = H_l$, which provides S-lane parallelism and roughly keeps register utilization of M_s as same as M_l. The Tier 5 remains unchanged as the fine-grained data parallelism: every thread can operate on four 8-bit values and two 16-bit values simultaneously, using single SIMD video instruction [23], for MSV/SSV and P7Viterbi algorithms, respectively. With the support of Tier 5, the parallelism of Tier 4 is further extended because each thread takes charge of four different sequences at most. The value of S, as the number of sequences processed in parallel by a warp, is defined as below:

$$\{S \mid S = 2^i, i \in \mathbb{Z} \cap [1, \log_2 \frac{\hat{s}\hat{w}_r}{\hat{w}_v}]\}, \tag{3}$$

where $\hat{s}$ is the warp size, $\hat{w}_r$ and $\hat{w}_v$ represent the width of registers and participant values, respectively. With Eq. 3, the rest of values can be also formulated as:

$$\begin{aligned} W &= \frac{\hat{s}\hat{w}_r}{\hat{w}_v S}, \\ L &= \left\lceil \frac{\hat{s}}{S} \right\rceil \\ H &= \max\left\{2, \left\lceil \frac{\hat{m}}{W} \right\rceil \right\}, \end{aligned} \tag{4}$$

where $\hat{m}$ represents the size of query model. Although the proposed framework is more complicated in terms of parallelism, it always yields the same result as MSV/SSV stages of HMMER3's pipeline.

3.2 Warp-Based Sequence Data Blocks

Due to the introduction of the multi-sequence alignment, loading sequence data in sequential layout is highly inefficient. Originally, in [18], residues of each sequence are stored in contiguous memory space, and warps always read 128 residues of one sequence by one coalesced global memory transaction. As for current design, however, the sequential data layout may lead to 128 transactions per memory request while extracting residues from 128 different sequences. It also exacerbates the issue of intra-warp thread idling caused by the huge difference of sequence length. Hence, given that warps have exclusive tasks, we propose a data reformatting scheme that splits whole sequence dataset into many small blocks. Sequences inside are assembled in striped layout to (a) balance workload amongst warps, (b) achieve fully coalesced memory access, and (c) minimize thread idling within each warp. This method aims to boost the advantage of the proposed five-tiered parallel framework.

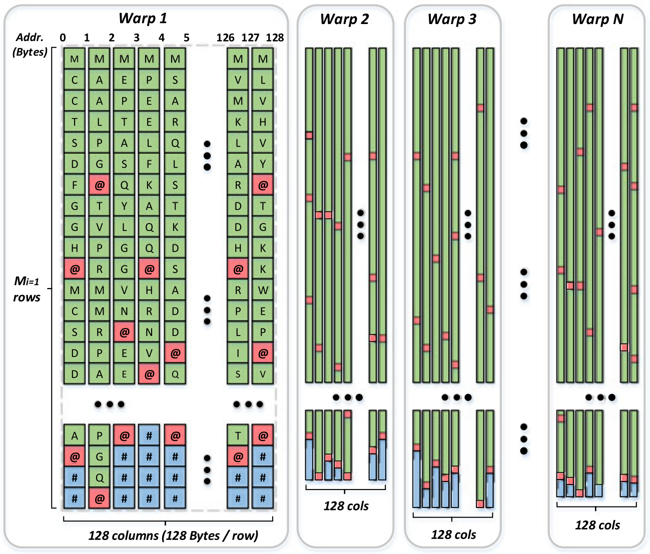


Fig. 4. Warp-based sequence data blocks.

As shown in Fig. 4, all sequences are divided into N blocks, each consists of $M_i \times 128$ residues, where N is the total number of resident warps, and M_i represents the height of each block with $i = [1, 2, 3...N]$. The number 128, as the width of each block, is pre-fixed for two reasons: (a) MSV/SSV kernels only need participant values with the width of $\hat{w}_v = 8$ bits, and hence a warp can handle up to $S = 128$ sequences simultaneously. (b) 128 residues, each occupies 1 byte, achieves aligned memory address for coalescing access. Marked as different colors, residues consist of three types: regular (green), padding (blue) and ending (red). In each block, sequences are concatenated and filled up 128 columns. A *ending* residue '@' is inserted at the end of each sequence, which is used to trigger an intra-warp branch to calculate and record the score of current sequence in the kernel. The value of M_i is equal to the length of the longest column within each block, such as second column of the data block for warp 1. As for the rest of columns whose length is less than M_i, *padding* residue '#'s are attached to make them all aligned. Residues that belong to the same warp are stored together in row-major order.

Algorithm 1 shows the pseudo-code of sequence data reformatting. N is determined by the number of SMXs and resident warps per SMX, denoted by N_{smx} and N_{warp}, respectively (line 1). Given the fixed width of data blocks, $N * 128$ containers C_x are created to hold and shape all sequences (line 3). One container corresponds to one column as shown in Fig. 4. We load first $N * 128$ sequences (line 4), and then the *for* loop (line 6 to 18) iterates over the rest to distribute them across C_x. The Seq with an index (x or y) represents a single sequence. To search a proper container where a sequence is attached, the accumulated length of the container and $maxLen$ are used (line 12). $maxLen$ stores the length of the longest container and is always be checked and updated by $checkMax()$ function once a sequence is attached successfully (line 7), which aims to balance lengths of $lines$ containers. We attach sequences to the last container (line 11) if no position is found. Once all sequences are distributed, $lines$ containers are evenly divided into N blocks, and the length of the longest container within each block M_i will be recorded by $getMax()$ function (line 19). The padding process, as the last step (line 20), shapes warp-based sequence data blocks B_i into rectangles. Implementation and evaluation of this data reformatting approach are presented in Section 4.1 and Appendix B.

Algorithm 1. Sequence Data Reformatting

```
Input: all sequences $Seq$ (sorted in descending order of length),
    the total number of sequences $T$.
Output: sequence data blocks $B$ and their heights $M$.
 1: $N \leftarrow N_{smx} * N_{warp}$
 2: $lines \leftarrow N * 128$
 3: create $lines$ containers $C_x$ where $x \in \mathbb{Z} \cap [0, lines)$.
 4: $C_x.attach(Seq[x])$                         ▷ attach $lines$ sequences
 5: $ptr \leftarrow lines - 1$                   ▷ start from the last container
 6: for all $Seq[y]$ where $y \in \mathbb{Z} \cap [lines, T)$ do
 7:     $maxLen.checkMax()$
 8:     repeat
 9:         if $ptr < 0$ then                    ▷ position is not found
10:             $ptr \leftarrow lines - 1$       ▷ start over
11:             $C_{ptr}.attach(Seq[y])$
12:         else if $Seq[y].len() + C_{ptr}.len() \le maxLen$ then
13:             $C_{ptr}.attach(Seq[y])$
14:         end if
15:         $ptr \leftarrow ptr - 1$             ▷ move to the next container
16:     until $Seq[y]$ is attached.
17:     $ptr \leftarrow lines - 1$ if $ptr < 0$.  ▷ refresh pointer
18: end for                                      ▷ all sequences are attached
19: $M_i \leftarrow C_x.divide(N).getMax(i)$ where $i = [1, \ldots, N]$
20: $B_i \leftarrow C_x.padding(M_i)$
```

3.3 Kernel Design

Since the number of sequences that are processed in parallel is within the range of $\{2, 4, 8, 16, 32, 64, 128\}$, the proposed algorithms are designed to cover all these cases. Therefore, 14 different types of kernels are generated, named as S-lane MSV/SSV kernels, and their implementations slightly vary with value S.

Algorithm 2 outlines the S-lane MSV kernels that are more complex than the implementation of single-sequence alignment in [18]. Some features are inherited, like (a) using local memory to hold intermediate values as well as register spill (line 1), (b) loading scores through read-only cache instead of shared memory to avoid weak scalability with low occupancy (line 15) and (c) fully unrolling the innermost loop for maximizing register usage to reside high frequency values on the on-chip memory (line 13). In order to assign different threads of a warp to work on different sequences without mutual interference, we label group ID gid and the offset in group oig on each thread (line 3 and 4). Threads that work on the same sequence are grouped with a unique gid, and they are assigned to different in-group tasks based on the oig. Inter-thread collaborations are only allowed within each thread group.

Algorithm 2. MSV Kernels with Multi-Sequence Alignment

```
Input: emission score $E$, sequence data block $B$, height of data
    block $M$, sequence length $Len$, offset of sequence $O_{seq}$, off-
    set of sequence length $O_{len}$ and other parameters, such as $L$,
    $W$, $H$, $S$ and $dbias$, etc.
Output: P-values of all sequences $P$.
 1: local memory $\Gamma[H]$
 2: $wid \leftarrow blockIdx.y * blockDim.y + threadIdx.y$
 3: $gid \leftarrow \lfloor threadIdx.x / L \rfloor$          ▷ group id
 4: $oig \leftarrow threadIdx.x \ \% \ L$                        ▷ offset in group
 5: for $k \leftarrow 0$ to $W - 1$ do
 6:     $count \leftarrow 0$
 7:     $mask, sc_E \leftarrow$ 0x00000000
 8:     $I_{seq} \leftarrow O_{seq}[wid] + gid * L + \lfloor k/4 \rfloor$
 9:     $sc_B \leftarrow$ initialize it based on $Len$, $O_{len}[wid]$ and $k$.
10:     while $count < M[wid]$ do
11:         $r \leftarrow extract\_res(B, I_{seq}, k, S)$
12:         $\gamma \leftarrow inter\_or\_intra\_reorder(L, S, v, oig)$
13:         #pragma unroll $H$
14:         for $h \leftarrow 0$ to $H - 1$ do
15:             $\sigma \leftarrow load\_emission\_score(E, S, L, oig, h, r)$
16:             $v \leftarrow \underline{vmaxu4}(\gamma, sc_B)$
17:             $v \leftarrow \underline{vaddus4}(v, dbias)$
18:             $v \leftarrow \underline{vsubus4}(v, \sigma)$
19:             $sc_E \leftarrow \underline{vmaxu4}(sc_E, v)$
20:             $\gamma \leftarrow \Gamma[h]$ & $mask$              ▷ load old scores
21:             $\Gamma[h] \leftarrow v$                          ▷ store new scores
22:         end for
23:         $sc_E \leftarrow max\_reduction(L, S, sc_E)$
24:         $sc_J, sc_B \leftarrow$ update special states, given $sc_E$.
25:         $mask \leftarrow$ 0xffffffff
26:         if $r$ contains ending residue @ then              ▷ branch
27:             $P_i \leftarrow$ calculate P-value of sequence $i$.
28:             $mask \leftarrow$ set affected bits to 0x00.
29:             $sc_J, sc_E, sc_B \leftarrow$ update or reset special states.
30:         end if
31:         $count, I_{seq} \leftarrow$ step up.
32:     end while
33: end for
```

The outer loop (line 5) iterates over columns of the warp-based sequence data block B while the middle loop (line 10) takes charge of each row. The cycle time of outer loop is directly affected by the query model size: the larger model results in the more cycles. This is because on-chip memory resources are limited when making alignment with large models, and it further leads to the kernel selection with small S. For example, a model with length of 45 can be handled by 128-lane kernels whereas a model of 1000-length may only select 4-lane kernels. Given that, in one warp, k is used to index S columns of residues simultaneously during each iteration, and the I_{seq} always points to residues that are being extracted from global memory. The details of residue extraction are shown in Algorithm 3. For S-lane kernels with $S \le 32$, only one 8-bit value (one residue) per thread group is extracted and is ready to make alignment though a warp always have the fully coalesced access to 128 residues assembled in 128-byte memory space. Instead, 64-lane kernels extract two 8-bit values, and 128-lane kernels are able to handle all of them. These residues are then used in the function *load_emission_score* (line 15) to load corresponding emission scores of "Match" states (line 3, 5 and 11 in Algorithm 4). The total number of amino acids is extended to 32, and the extra states are filled with invalid scores, which aim to cover the newly introduced residues (*ending* and *padding*). 64 and 128-lane kernels are treated in a special way as shown in Algorithm 4 (line 4-9 and line 10-15) due to the demand of score assembly. In this case, each thread assembles two or four scores of different residues into a 32-bit register to be ready for subsequent SIMD instructions. All emission scores are loaded through read-only cache to keep shared/L1 cache path from overuse, and the score sharing is done via inter-thread shuffle instructions.

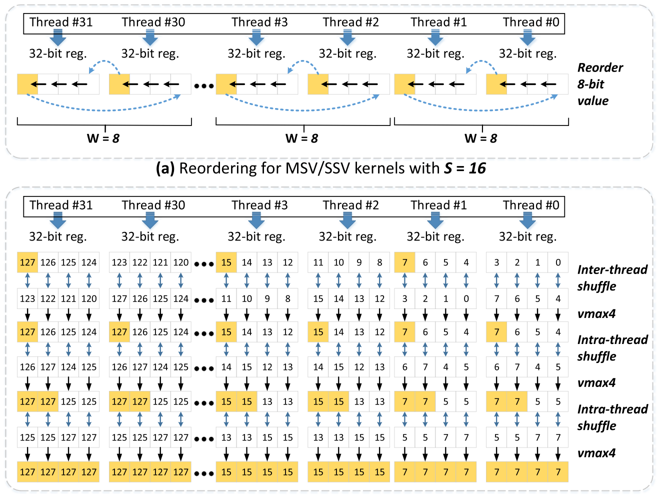
Algorithm 3. Extract Residues from Data Block -*extract_res*

```
Input: $B$, $I_{seq}$, $k$ and $S$.
Output: a 32-bit value $r$ that contains one, two or four residues,
    given $S$.
1: if $S \in \{2, 4, 8, 16, 32\}$ then
2:     $r \leftarrow (B[I_{seq}] >> 8 * (k \% 4))$ & 0x000000ff
3: else if $S = 64$ then
4:     $r \leftarrow (B[I_{seq}] >> 16 * k)$ & 0x0000ffff
5: else if $S = 128$ then
6:     $r \leftarrow B[I_{seq}]$
7: end if
```

Algorithm 4. Get "Match" Scores -*load_emission_score*

```
Input: $E$, $S$, $L$, $oig$, $h$ and $r$.
Output: a 32-bit value $\sigma$ that contains four emission scores,
    each is 8-bit, in striped layout.
 1: $N_a \leftarrow 32$                                          ▷ amino acids
 2: if $S \in \{2, 4, 8, 16, 32\}$ then
 3:     $\sigma \leftarrow E[h * N_a * L + r * L + oig]$                   ▷ ldg
 4: else if $S = 64$ then
 5:     $sc \leftarrow E[h * N_a + threadIdx.x]$ & 0x0000ffff
 6:     $res \leftarrow r$ & 0x000000ff
 7:     $\sigma \leftarrow \sigma \,\|\, (\_\_shfl(sc, res))$                    ▷ assembly
 8:     $res \leftarrow (r >> 8)$ & 0x000000ff
 9:     $\sigma \leftarrow \sigma \,\|\, (\_\_shfl(sc, res) << 16)$              ▷ assembly
10: else if $S = 128$ then
11:     $sc \leftarrow E[h * N_a + threadIdx.x]$ & 0x000000ff
12:     for $bits \in \{0, 8, 16, 24\}$ do
13:         $res \leftarrow (r >> bits)$ & 0x000000ff
14:         $\sigma \leftarrow \sigma \,\|\, (\_\_shfl(sc, res) << bits)$        ▷ assembly
15:     end for
16: end if
```

Algorithms 5 and 6 detail two crucial steps of MSV/SSV kernels, *inter_or_intra_reorder* and *max_reduction*, (line 12 and 23 in Algorithm 2) via the PTX assembly to expose internal mechanisms of massive bitwise operations for the multi-sequence alignment. They aim to reorder 8-bit values and get the maximum value amongst each thread group in parallel, and meanwhile, noninterference between thread groups is guaranteed. Our design still avoids to use shared memory since available L1 cache is the key factor on performance



(a) Reordering for MSV/SSV kernels with ***S* = 16**

(b) Max-reduction for MSV/SSV kernels with ***S* = 16**

Fig. 5. An example of Reordering and Max-reduction for kernels that handle 16 sequences simultaneously.

when unrolling the innermost loop. Therefore, all intermediate or temporary values are held by private memory space of each thread, such as registers or local memory. The shuffle instruction, `shfl`, is employed again to achieve the inter-thread communication but the difference is that a `mask` is specified to split a warp into sub-segments (line 8 in Algorithm 5). Each sub-segment represents a thread group. In Algorithm 6, two reduction phases are required for S-lane kernels with $S \leq 16$. Line 5 to 12 presents inter-thread reductions by using `shfl` and `vmax` to get top four values of 8-bit within each thread group. The following lines are intra-thread reductions which only happen inside each thread and eventually work out the maximum value. As an example, Fig. 5 illustrates the reordering and max-reduction for 16-lane kernels. $W = 8$ is the width of striped model states, and it can also be regarded as the scope of each thread group. For $S = 16$, a warp is partitioned into 16 thread groups, and each handles eight values of 8-bit. Yellow lattices in Fig. 5a are values that need to be exchanged between two threads. Arrows indicate the reordering direction. In Fig. 5b, assuming digital numbers (0 to 127) labeled inside lattices represent the values held by threads, the yellow lattices always track the maximum value of each thread group. Three pairs of shuffle and SIMD instructions are used to calculate the maximum value and broadcast it to all members of the thread group.

The potential divergent execution only happens in the branch for recording P-value of ended sequences, as shown in Algorithm 2, line 26-30. Threads which find the existence of *ending* residues keep active and move into the branch whereas others are inactive during this period. A `mask` is introduced to mark the position of ended sequences within 32-bit memory space and set those affected bits to 0. This is particularly helpful to 64 and 128-lane kernels because it only cleans up corresponding lanes for new sequence while keeping data of other lanes unchanged. Moreover, bitwise operations with `mask` minimize the number of instructions needed inside the innermost loop, which is also beneficial to the overall performance. As for the SSV kernel shown in Algorithm 7, it shares the same framework with the MSV kernel but has less computational workload. Besides, one more mask value is added to reset affected bits since the

`-inf` of SSV kernel is `0x80` rather than `0x00`. $mask1$ cleans up outdated scores as the first step followed by a bitwise disjunction with $mask2$ to reset local memory Γ (line 17).

Algorithm 5. Reorder 8-Bit Value Inter-Thread or Intra-Thread -*inter_or_intra_reorder*

Input: L, S, v and oig.
Output: a 32-bit value γ that contain four 8-bit values.

```
 1: if $S \in \{2,4,8,16\}$ then                    ▷ inter-thread
 2:     $lane \leftarrow (oig + L - 1)\ \%\ L$        ▷ source lane
 3:     asm{shr.u32 x, v, 24}
 4:     asm{mov.b32 mask, 0x1f}
 5:     asm{sub.b32 mask, mask, L(S) − 1}
 6:     asm{shl.b32 mask, mask, 16}
 7:     asm{or.b32 mask, mask, 0x1f}
 8:     asm{shfl.idx.b32 x, x, lane, mask}
 9:     asm{shl.b32 y, v, 8}
10:     $\gamma \leftarrow x \mid y$
11: else if $S = 32$ then                            ▷ intra-thread
12:     asm{shr.u32 x, v, 24}
13:     asm{shl.b32 y, v, 8}
14:     $\gamma \leftarrow x \mid y$
15: else if $S = 64$ then                            ▷ intra-thread
16:     asm{shr.u32 x, v, 8}
17:     asm{and.b32 x, x, 0x00ff00ff}
18:     asm{shl.b32 y, v, 8}
19:     asm{and.b32 y, y, 0xff00ff00}
20:     $\gamma \leftarrow x \mid y$
21: else if $S = 128$ then
22:     $\gamma \leftarrow$ 0x00000000 or 0x80808080    ▷ not reorder
23: end if
```

3.4 Kernel Optimization

The kernel performance of CUDAMPF shows that $H = 19$ is able to cover the longest query model whose length is 2405, for MSV and SSV algorithms [18]. In current design, however, 2-lane kernels can only handle models with the length of 1216 at most, given the same H. In addition, we recall that L1 Cache-Hit-Ratio (CHR) is employed as a metric to evaluate register spill in CUDAMPF, and MSV/SSV kernels with maximum 64 registers per thread have no spill to local memory. This enables us to push up H to hold larger models for the multi-sequence alignment. Two optimization schemes are proposed to improve overall performance and address the concern about scalability.

3.4.1 CHR Sacrificed Kernel

It is well-known that L1 cache shares the same block of on-chip memory with shared memory physically, and it is used to cache accesses to local memory as well as register spill. The L1 cache has low latency on data retrieval and storage with a cache hit, which can be further utilized to increase the throughput of kernels based on the proposed framework. We treat L1 cache as *secondary registers*, and the usage is measured by CHR for local loads and stores. By increasing up H, more model states can reside in registers and cached local memory. The moderate loss of performance due to uncached register spills is acceptable, which is attributed to highly optimized task and data parallelism in the current framework. However, it is impossible to increase H unboundedly due to the limited capacity of L1 cache. Overly large H leads to the severe register spill that causes low CHR and stalls warps significantly. Hence, there is always a trade-off between CHR and H, and the goal is to find a reasonable point where kernel performance (GCUPS) begins fall off. The decline in performance indicates that the latency, caused by excessive communications between on and off-chip memory, starts to overwhelm the benefits of parallelism. The corresponding H at the turning point is considered to be the maximum one, denoted by H_{max}.

Algorithm 6. Get and Broadcast Maximum Value Through Reduction Operations -*max_reduction*

Input: L, S and sc_E.
Output: a 32-bit value sc_E that contains four 8-bit or two 16-bit values.

```
 1: if $S = 128$ then
 2:     do nothing but Return $sc_E$.
 3: else
 4:     $x \leftarrow sc_E,\ y \leftarrow 0,\ z \leftarrow 0$
 5:     if $S \in \{2,4,8,16\}$ then                ▷ inter-thread reduction
 6:         $i \leftarrow \log_2 L - 1$
 7:         for $lm \leftarrow 2^0$ to $2^i$ do
 8:             asm{shfl.bfly.b32 y, x, lm, 0x1f}
 9:             asm{and.b32 m, m, 0x00000000}
10:             asm{vmax4.u32.u32.u32 x, x, y, m}
11:         end for
12:     end if
13:     asm{shr.u32 y, x, 8}
14:     asm{and.b32 y, y, 0x00ff00ff}
15:     asm{shl.b32 z, x, 8}
16:     asm{and.b32 z, z, 0xff00ff00}
17:     asm{or.b32 y, y, z}
18:     asm{and.b32 m, m, 0x00000000}
19:     asm{vmax4.u32.u32.u32 x, x, y, m}
20:     Return $sc_E \leftarrow x$ if $S = 64$.
21:     asm{shr.u32 y, x, 16}
22:     asm{shl.b32 z, x, 16}
23:     asm{or.b32 y, y, z}
24:     asm{and.b32 m, m, 0x00000000}
25:     asm{vmax4.u32.u32.u32 x, x, y, m}
26:     Return $sc_E \leftarrow x$ if $S \in \{1,2,4,8,16,32\}$.
27: end if
```

Table 1 lists a benchmark result that shows the relationship between H, kernel performance and CHR. Starting from $H = 20$ with a step of 5, intuitively, CHR is being consumed after on-chip registers are exhausted, and the kernel performance increases first and falls back eventually as expected. We choose 45 and 50 as the H_{max} for MSV and SSV kernels, respectively. Larger H results in rapid degradation of both performance and L1 CHR. Besides, the difference of H_{max} indicates that MSV kernels have more instructions than SSV kernels within the innermost loop, and hence more registers or local memory are used while unrolling the loop. The H_{max} is therefore algorithm-dependent. Given Eqs. (3) and (4) and H_{max}, we formulate the selection of S as below:

$$\underset{S}{\operatorname{argmax}}\, \boldsymbol{f} = \{S \mid \boldsymbol{f} = W_S H_{max}, \forall \hat{m} \le W_2 H_{max} : \boldsymbol{f} \ge \hat{m}\}, \quad (5)$$

where $\boldsymbol{f}$ is the function of S, and $W_2 H_{max}$ indicates the maximum length of query models that 2-lane kernels can handle.

TABLE 1
Benchmarks of the Maximum H via Innermost Loop Unrolling

H	reg. per thread	stack frame	spill stores	spill loads	GCUPS	L1 CHR (%)
			MSV kernels			
20	63	8	0	0	258.7	99.97
25	63	8	0	0	261.5	99.97
30	64	8	0	0	272.1	99.97
35	64	40	40	44	279.3	75.65
40	64	48	52	56	283.9	75.41
45	**64**	**48**	**56**	**52**	**280.6**	**75.49**
50	64	96	152	96	263.9	25.61
55	64	128	208	140	240.8	14.95
			SSV kernels			
20	62	8	0	0	269.5	99.96
25	62	8	0	0	314.0	99.96
30	62	8	0	0	329.0	99.96
35	61	8	0	0	347.9	99.96
40	64	16	4	4	361.5	99.94
45	64	48	44	40	375.9	80.44
50	**64**	**56**	**64**	**44**	**375.9**	**60.30**
55	64	80	116	72	340.9	17.81

**** Data collections on 32-lane kernels compiled with* nvcc *8.0. Use* env_nr *[24] as the sequence dataset.*

The CUDAMPF implementation will be used instead if any larger model is applicable. Eq. (5) describes a rule of kernel selection that always prefers to use kernels with more lanes if they are able to cover the model length. Once the kernel type (S-lane) is determined, the H of every query model located in the coverage area can be obtained via:

$$\underset{H}{\operatorname{argmin}}\, \mathbf{\Phi} = \left\{ H \,|\, \forall H \in \left[\left[\frac{\hat{m}}{W_S}\right], H_{max}\right] \cap \mathbb{Z} : \mathbf{\Phi} = W_S H, \mathbf{\Phi} \geq \hat{m} \right\}, \tag{6}$$

where $\mathbf{\Phi}$ represents the function of H. Eq. (6) minimizes H to fit query models perfectly, which thereby avoids redundant computation and memory allocation.

In summary, this optimization scheme aims to fully leverage the speedy on-chip memory, including L1 cache via sacrificing CHR proactively, to further boost kernel throughput, and in the meanwhile, it extends coverage of the proposed framework to larger query models.

3.4.2 Performance-Oriented Kernel

Although the proposed framework achieves a significant improvement in performance, it certainly introduces overhead due to the implementation of multi-sequence alignment, compared with CUDAMPF. This downside becomes more apparent as the model length increases (S decreases). Therefore, it is expected that CUDAMPF with single-sequence alignment may exceed CUDAMPF++ for large enough query models. In order to pursue optimal performance consistently, we also merge CUDAMPF implementation into the proposed framework as a special case with $S = 1$. The maximum model length $\hat{m}_{max}$ on which CUDAMPF++ still outperforms is defined as the threshold of kernel switch.

Similar to $\hat{m}_{max}$, another threshold $\hat{m}_{min}$ can also be employed to optimize 128 and 64-lane kernels for small models. We recall that 128 and 64-lane kernels need extra operations to load emission scores in Algorithm 4. Thus, they have more overhead than other kernels within the innermost loop, which may counteract their advantages on the number of parallel lanes. We extend the coverage of 32-lane kernels to handle small models owned by 128 and 64-lane kernels previously, and the evaluation is presented in Section 4.3.

Algorithm 7. SSV Kernel with Multi-Sequence Alignment

Input: emission score E, sequence data block B, height of data block M, sequence length Len, offset of sequence O_{seq}, offset of sequence length O_{len} and other parameters, such as L, W, H, S and $dbias$, etc.
Output: P-values of all sequences P.

```
1:  local memory $\Gamma[H]$
2:  $wid \leftarrow blockIdx.y * blockDim.y + threadIdx.y$
3:  $gid \leftarrow \lfloor threadIdx.x/L \rfloor$                ▷ group id
4:  $oig \leftarrow threadIdx.x \% L$                  ▷ offset in group
5:  for $k \leftarrow 0$ to $W-1$ do
6:      $count \leftarrow 0$
7:      $mask1, mask2, sc_E \leftarrow$ 0x80808080
8:      $I_{seq} \leftarrow O_{seq}[wid] + gid * L + \lfloor k/4 \rfloor$
9:      while $count < M[wid]$ do
10:         $r \leftarrow extract\_res(B, I_{seq}, k, S)$
11:         $\gamma \leftarrow inter\_or\_intra\_reorder(L, S, v, oig)$
12:         #pragma unroll $H$
13:         for $h \leftarrow 0$ to $H-1$ do
14:             $\sigma \leftarrow load\_emission\_score(E, S, L, oig, h, r)$
15:             $v \leftarrow \underline{vsubus4}(v, \sigma)$
16:             $sc_E \leftarrow \underline{vmaxu4}(sc_E, v)$
17:             $\gamma \leftarrow \Gamma[h]$ & $mask1 \parallel mask2$       ▷ load old scores
18:             $\Gamma[h] \leftarrow v$                     ▷ store new scores
19:         end for
20:         $sc_E \leftarrow max\_reduction(L, S, sc_E)$
21:         $mask1 \leftarrow$ 0xffffffff, $mask2 \leftarrow$ 0x00000000
22:         if $r$ contains ending residue @ then            ▷ branch
23:             $P_i \leftarrow$ calculate P-value of sequence $i$.
24:             $mask1 \leftarrow$ set affected bits to 0x00.
25:             $mask2 \leftarrow$ set affected bits to 0x80.
26:             $sc_E \leftarrow$ update or reset special states.
27:         end if
28:         $count, I_{seq} \leftarrow$ step up.
29:     end while
30: end for
```

3.5 Differences Between CUDAMPF++ and CUDAMPF

As an extension work, CUDAMPF++ shares some common traits with CUDAMPF while possessing many exclusive designs. With respect to framework and kernel optimization, significant differences between CUDAMPF and CUDAMPF++ are summarized and listed in Table 2.

4 Experimental Results

In this section, we present several performance evaluations on the proposed CUDAMPF++, such as workload balancing, kernel throughput and scalability. The comparison targets consist of CUDAMPF, CPU-based *hmmsearch* of latest HMMER v3.1b2 and other related acceleration attempts. Both CUDAMPF++ and CUDAMPF are evaluated on a NVIDIA

TABLE 2
Difference Comparison between CUDAMPF and CUDAMPF++

	CUDAMPF	CUDAMPF++
Framework	(a) Four tiers.	(a) Five tiers. A new tier is inserted after Tier 3 to perform multi-sequence alignment within a warp.
	(b) At Tier 2, a warp handles one sequence at a time.	(b) At Tier 2, a warp handles a block of sequences.
	(c) Residues of each sequence are sequentially arranged.	(c) Sequences are assembled as 128-lane data blocks with striped layout.
Kernel Opt.	(a) More on-chip registers are used to reduce latency.	(a) Not only are registers exhausted, but L1 CHR is also proactively sacrificed to boost kernel throughput further.
	(b) NVRTC is used for dynamic kernel compilation.	(b) JIT compilation is used to switch amongst S-lane kernels.

Tesla K40 GPU and compiled with CUDA v8.0 compiler. Tesla K40 is built with the Kepler GK110 architecture that contains 15 SMXs (2880 CUDA cores) and 12 GB off-chip memory [25]. One of NVIDIA profiling tools, *nvprof* [26], is also used to track metrics like L1/tex CHR, register usage and spill. For *hmmsearch*, two types of CPUs are employed to collect performance results: Intel Xeon E5620 (4 physical cores with maximum 8 threads) and dual Intel Xeon E5-2650 (16 physical cores and 32 threads in total). All programs are executed in the 64-bit Linux operating system.

Unlike CUDAMPF implementation, the NVRTC is deprecated in current design due to its unstability in compiling kernels with high usage of on-chip registers. Even with latest compiler *nvcc* v8.0, runtime compilation with the NVRTC library still generates unexpected binary files or report the error of resources exhaustion, especially when unrolling large loops and register spill happens. Thus, we choose the just-in-time (JIT) compilation instead. All kernels are pre-compiled in offline mode and stored as *.ptx* files, each with an unique kernel name inside. Given different query models, the corresponding *.ptx* file is loaded and further compiled to binary code at runtime. The load time and overhead of compilation are negligible.

Two protein/nucleotide sequence datasets [24] are chosen for experiments: (a) *env_nr* (1.9 GB) and (b) *est_human* (5.6 GB). As for query models, we still use Pfam 27.0 [27] that contains 34 thousand HMMs with different sizes ranging from 7 to 2405. The overall performance is measured in kernel throughput (GCUPS) which is directly calculated by the total number of residues contained in each database, model length and kernel execution time.

4.1 Evaluation of Workload Balancing

To avoid time overhead of data reformatting introduced in Section 3.2, we incorporate Redis [28], a high performance in-memory database, into the proposed framework. Redis is written in ANSI C and able to work with CUDA seamlessly. It currently works as an auxiliary component to hold warp-based sequence data blocks and query models in memory, which offers blazing fast speed for data retrieval. Given the single K40 GPU, each sequence dataset is partitioned into 61,440 blocks which are then ingested into Redis database separately as key-value pairs. The quantity of data blocks resided in Redis database should be integral multiple of the number of available warps.

Table 3 summarizes the evaluation result of workload balancing for both sequence datasets. The “avg.” and “sd.” represent average value and standard deviation across all blocks, respectively. We recall that M is the height of data block which serves as the metrics of computational workload for each warp, and the number of *ending* residues is another impact factor of performance because the *ending* residue may lead to thread idling. It is clear to see that both sd. M and sd. *ending* residues are trivial, and the last two columns show that average multiprocessor efficiency approaches 100 percent, which are strong evidences of balanced workload over all warps on GPU. Besides, the Padding-to-Real Ratio (*PRR*) that compares the level of invalid computation to the level of desired computation is investigated to assess the negative effect of padding residues, and it is also proved to be negligible.

4.2 Performance Evaluation and Analysis

In order to demonstrate the outstanding performance of proposed method and its correlation with the utilization of memory resources, we make an in-depth comparison between CUDAMPF++ and CUDAMPF via profiling both MSV and SSV kernels, reported in Tables 4 and 5, respectively. A total of 27 query models are selected to investigate the impact of H on the performance of S-lane kernels. Each kernel type is evaluated with two models that correspond to $H = H_{max}$ and $H = \lceil \frac{H_{max}-1}{2} + 1 \rceil$, except for the 2-lane kernel since the 2405 is the largest model length in [27] with corresponding $H = 38$.

For the MSV kernels, the maximum speedup listed on Table 4 is 17.0x when $\hat{m} = 23$, and the trend of speedup is descending as model length increases. This is because memory resources, like on-chip registers, L1 cache and even local memory, are significantly underutilized in CUDAMPF when making alignment with small models whereas CUDAMPF++ always intends to fully take advantage of them. Given 64 as the maximum number of registers per thread, only about half the amount of registers are occupied in CUDAMPF till $\hat{m} = 735$, and other resources are not

TABLE 3
Evaluation of Workload Balancing for the Warp-Based Sequence Data Blocks

DB name	DB size (GB)	total seq.	total residues	avg. M	sd. M	avg. *ending* residues	sd. *ending* residues	avg. *PRR*	SMX eff. MSV (%)*	SMX eff. SSV (%)*
env_nr	1.9	6,549,721	1,290,247,663	21,109	85	13,645	71	1.14E-4	97.13	96.7
est_human	5.6	8,704,954	4,449,477,543	72,563	174	18,135	60	3.22E-5	97.97	97.76

* *Data collection with 32-lane kernels.*

TABLE 4
Performance Comparison of MSV Kernel between the Proposed CUDAMPF++ and CUDAMPF

						CUDAMPF++ *versus* CUDAMPF					
S-lane kernels	model length $\hat{m}$	acc. ID	H	reg. per thread	stack frame	spill stores	spill loads	L1 CHR (%)	Tex. CHR (%)	GCUPS	speedup
128	23	PF13823.1	23 / 2	63 / 29	24 / 0	0 / 0	0 / 0	99.94 / *unused*	100 / 100	168.6 / 9.9	17.0x
128	45	PF05931.6	45 / 2	64 / 29	48 / 0	48 / 0	24 / 0	51.93 / *unused*	100 / 100	144.4 / 19.4	7.5x
64	46	PF09501.5	23 / 2	64 / 29	16 / 0	0 / 0	0 / 0	99.96 / *unused*	100 / 100	225.7 / 19.8	11.4x
64	90	PF05777.7	45 / 2	64 / 29	48 / 0	64 / 0	32 / 0	50.88 / *unused*	100 / 100	231.3 / 38.7	6.0x
32	92	PF00207.17	23 / 2	64 / 29	8 / 0	0 / 0	0 / 0	99.97 / *unused*	100 / 100	274.1 / 39.6	7.0x
32	180	PF02737.13	45 / 2	64 / 29	48 / 0	56 / 0	52 / 0	75.49 / *unused*	100 / 100	278.8 / 77.4	3.6x
16	184	PF00596.16	23 / 2	63 / 29	8 / 0	0 / 0	0 / 0	99.99 / *unused*	100 / 100	266.7 / 78.9	3.4x
16	360	PF01117.15	45 / 3	64 / 30	56 / 0	60 / 0	60 / 0	80.11 / *unused*	100 / 100	277.0 / 130.9	2.1x
8	368	PF05208.8	23 / 3	63 / 30	8 / 0	0 / 0	0 / 0	99.99 / *unused*	100 / 100	266.7 / 133.6	2.0x
8	720	PB000053	45 / 6	64 / 34	56 / 0	60 / 0	60 / 0	80.10 / *unused*	78.48 / 92.37	271.6 / 183.8	1.5x
4	735	PF03971.9	23 / 6	63 / 34	8 / 0	0 / 0	0 / 0	100 / *unused*	75.3 / 92.37	262.4 / 187.4	1.4x
4	1439	PF12252.3	45 / 12	64 / 51	56 / 0	60 / 0	60 / 0	80.08 / *unused*	67.09 / 72.66	271.5 / 231.6	1.2x
2	1471	PB006678	23 / 12	63 / 51	8 / 0	0 / 0	0 / 0	100 / *unused*	63.38 / 72.6	259.7 / 237.3	1.1x
2	2405	PB003055	38 / 19	64 / 64	40 / 0	32 / 0	44 / 0	100 / *unused*	60.15 / 64.06	264.0 / 271.2	1.0x

**** The* env_nr *[24] is used in data collection.*

utilized at all. In contrast, the CUDAMPF++ not only keeps high usage of registers but also utilizes L1 cache and local memory to hold more data, which results in a near constant performance regardless of the model length. The texture CHR is dominated by model length since we only use texture cache for loading emission scores. Larger model leads to lower texture CHR. Comparing the performance of S-lane kernels in CUDAMPF++, the cases of $H = 45$ outperform the cases of $H = 23$ though more local memory are allocated with register spill. One exception is the 128-lane kernel due to its higher complexity of the innermost loop, which can be optimized via using the 32-lane kernel instead.

As shown in Table 5, SSV kernels have similar evaluation results with MSV kernels but higher throughput. Starting from $\hat{m} = 832$, nevertheless, CUDAMPF outperforms and eventually yields upto 468.9 GCUPS which is 1.5x faster than CUDAMPF++. The case that peak performance of two frameworks are not comparable is due to the overhead of extra instructions introduced for the multi-sequence alignment in CUDAMPF++. The kernel profiling indicates that both MSV and SSV kernels are bounded by computation and memory bandwidth (texture). However, unlike MSV kernels, SSV kernels have fewer operations within the innermost loop, which makes them more “sensitive”. In other words, newly added operations (i.e., bitwise operations for mask) within the innermost loop, compared with CUDAMPF, have more negative effect on SSV kernels than MSV kernels. Therefore, an upto 50 percent performance gap is observed only in SSV kernels.

4.3 Scalability Evaluation

In order to demonstrate the scalability of the proposed framework, a total of 57 query models with different sizes ranging from 10 to 2450 are investigated. The interval of model length is fixed to 50. Figs. 6 and 7 show the performance comparison between CUDAMPF++ and CUDAMPF for MSV and SSV kernels, respectively. The coverage area of model length for each kernel type is highlighted. Right

TABLE 5
Performance Comparison of SSV Kernel between the Proposed CUDAMPF++ and CUDAMPF

						CUDAMPF++ *versus* CUDAMPF					
S-lane kernels	model length	acc. ID	H	reg. per thread	stack frame	spill stores	spill loads	L1 CHR (%)	Tex. CHR (%)	GCUPS	speedup
128	26	PF02822.9	26 / 2	62 / 33	24 / 0	0 / 0	0 / 0	99.93 / *unused*	100 / 100	178.4 / 15.9	11.2x
128	50	PF03869.9	50 / 2	64 / 33	56 / 0	60 / 0	32 / 0	54.29 / *unused*	100 / 100	144.6 / 30.7	4.7x
64	52	PF02770.14	26 / 2	63 / 33	16 / 0	0 / 0	0 / 0	99.96 / *unused*	100 / 100	276.1 / 31.8	8.7x
64	100	PB000229	50 / 2	64 / 33	72 / 0	100 / 0	52 / 0	30.65 / *unused*	100 / 100	239.8 / 61.4	3.9x
32	104	PF14807.1	26 / 2	61 / 33	8 / 0	0 / 0	0 / 0	99.96 / *unused*	100 / 100	307.8 / 63.6	4.8x
32	200	PF13087.1	50 / 2	64 / 33	56 / 0	64 / 0	44 / 0	60.3 / *unused*	100 / 100	365.0 / 122.4	3.0x
16	208	PF15420.1	26 / 2	62 / 33	8 / 0	0 / 0	0 / 0	99.98 / *unused*	100 / 100	305.0 / 127.2	2.4x
16	400	PF13372.1	50 / 4	64 / 37	56 / 0	72 / 0	52 / 0	58.57 / *unused*	95.66 / 100	347.8 / 199.3	1.7x
8	416	PF06808.7	26 / 4	62 / 37	8 / 0	0 / 0	0 / 0	99.99 / *unused*	98.92 / 100	302.6 / 209.1	1.4x
8	800	PF02460.13	50 / 7	64 / 42	56 / 0	72 / 0	52 / 0	58.53 / *unused*	75.79 / 87.24	344.8 / 306.9	1.1x
4	832	PB001474	26 / 7	62 / 42	8 / 0	0 / 0	0 / 0	99.99 / *unused*	76.90 / 87.24	302.1 / 319.1	0.9x
4	1600	PB000744	50 / 13	64 / 56	56 / 0	72 / 0	52 / 0	58.49 / *unused*	65.68 / 70.80	349.3 / 403.5	0.9x
2	1630	PB000663	26 / 13	62 / 56	8 / 0	0 / 0	0 / 0	100 / *unused*	66.57 / 70.81	293.1 / 411.1	0.7x
2	2405	PB003055	38 / 19	64 / 63	16 / 0	0 / 0	0 / 0	100 / *unused*	59.69 / 64.05	318.8 / 468.9	0.7x

**** The* env_nr *[24] is used in data collection.*

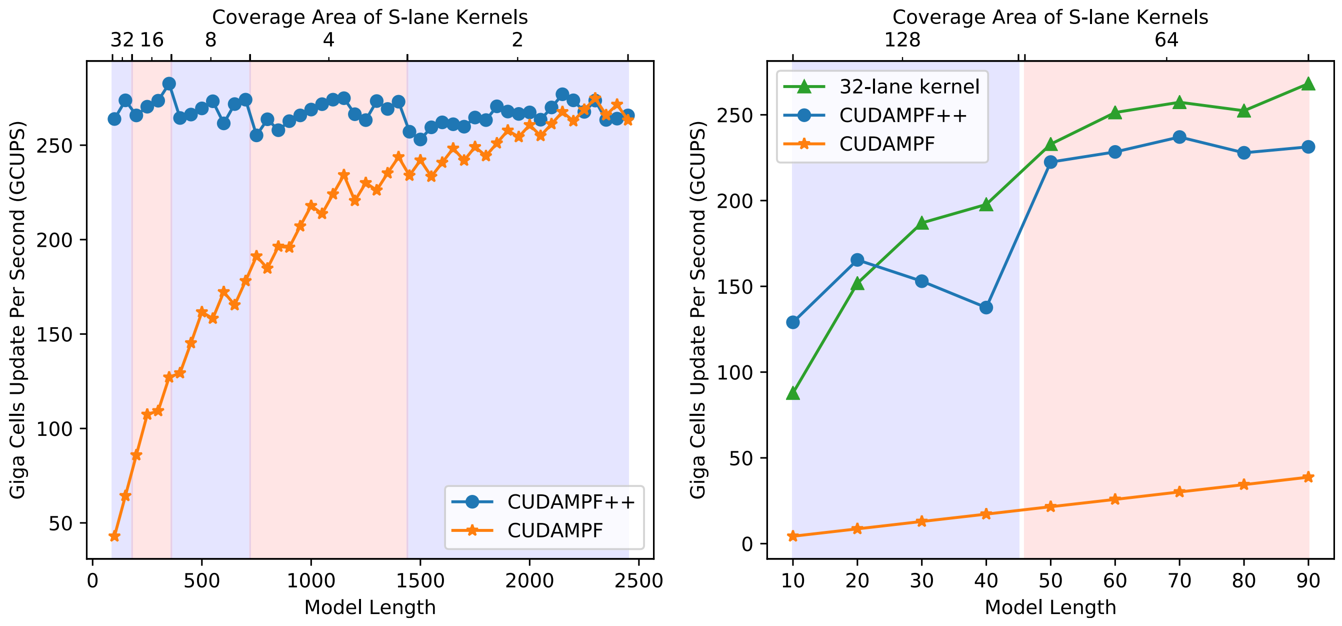


Fig. 6. Performance comparison between CUDAMPF++ and CUDAMPF for the MSV kernel.

subfigure depicts the performance of 128 and 64-lane kernels while others are shown in the left one. Overall, CUDAMPF++ achieves near constant performance and significantly outperform CUDAMPF with small models. The S-lane MSV (SSV) kernel yields the maximum speedup of 30x (23x) with respect to CUDAMPF. It is worth mentioning that, in CUDAMPF++, SSV kernels have larger fluctuation margin of performance than MSV kernels. This is caused by the overhead of using read-only cache (texture cache) to load emission scores. Although both MSV and SSV kernels have only one *__ldg()* function inside innermost loop, the texture cache read in SSV kernels, as one of kernel limits, has a higher proportion of negative effect on performance than that in MSV kernels, which results in such obvious fluctuation. A simple evidence is that the performance curve will be smooth and regular if replacing texture cache reads with a constant value.

Besides, 32-lane kernels are also tested to compare with 128 and 64-lane kernels. By decreasing H, the 32-lane kernel is able to cover smaller query models, and it outperforms 128 and 64-lane kernels until the model length is slightly larger than 20. We simply set $\hat{m}_{min} = 20$ for both MSV and SSV kernels in terms of evaluation results. As for $\hat{m}_{max}$, the model lengths of 2450 and 1000 are selected for MSV and SSV kernels, respectively.

4.4 Performance Comparison: CUDAMPF++ Versus Others

A comprehensive performance comparison is also made between optimized CUDAMPF++ and other implementations.

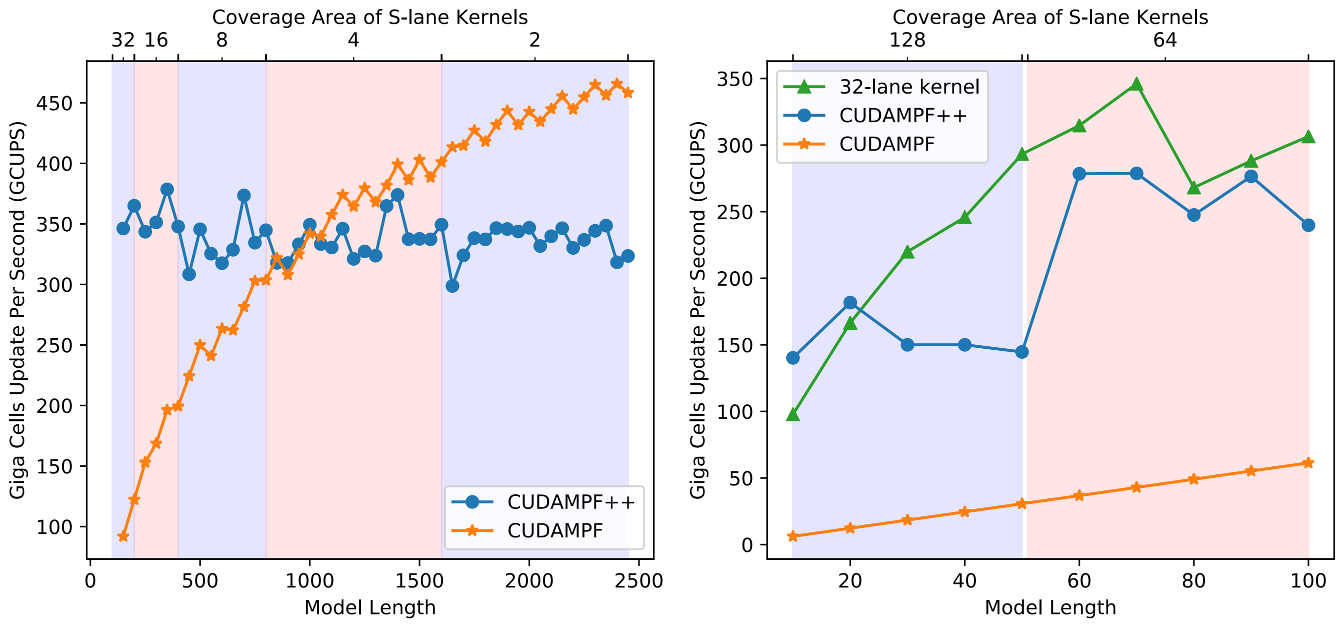


Fig. 7. Performance comparison between CUDAMPF++ and CUDAMPF for the SSV kernel.

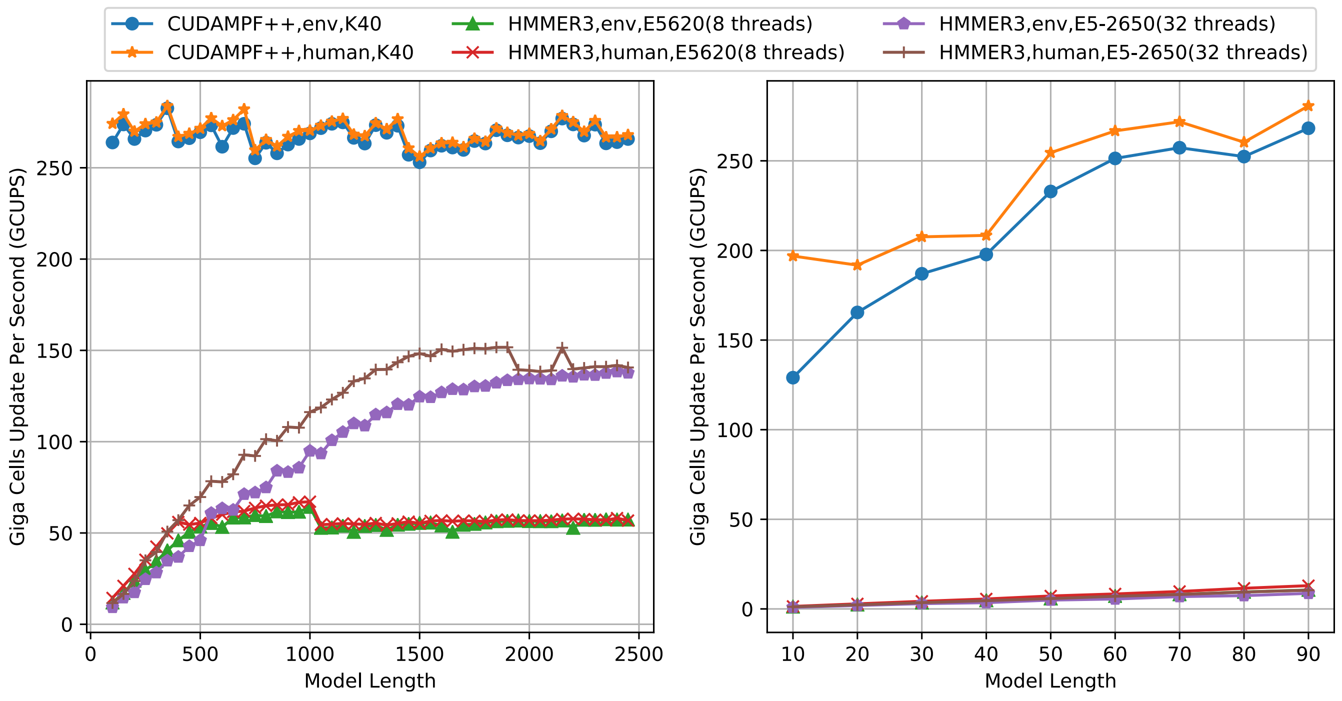


Fig. 8. Performance comparison between CUDAMPF++ and HMMER3's CPU-based implementation for the MSV kernel (stage).

Figs. 8 and 9 present results of comparison between CUDAMPF++ and CPU-based MSV/SSV stages with two datasets. The CUDAMPF++ achieves upto 282.6 (283.9) and 465.7 (471.7) GCUPS for MSV and SSV kernels, respectively, given the *env_nr* (*est_human*) dataset. Compared with the best performance achieved by dual Xeon E5-2650 CPUs, a maximum speedup of 168.3x (160.7x) and a minimum speedup of 1.8x (1.7x) are observed for the MSV (SSV) kernel of CUDAMPF++. Correspondingly, overall speedup for the whole pipeline ranges from 1.5x to 3.3x, which is estimated by using Eq. (1).

In the original HMMER3 paper [3], Eddy reports 12 GCUPS for MSV stage, achieved by a single CPU core. Several acceleration efforts exist and report higher performance: (a) an FPGA-based implementation [11] yields upto 81 GCUPS for MSV stage; (b) Lin [13] inherits and modifies a GPU-based implementation of HMMER2 [6] to accelerate MSV stage of HMMER3, which achieves upto 32.8 GCUPS on a Quadro K4000 GPU; (c) [16] claims the first acceleration work on SSV stage of latest HMMER v3.1b2 and reports the maximum performance of 372.1 GCUPS on a GTX570 GPU. Moreover, performance of all

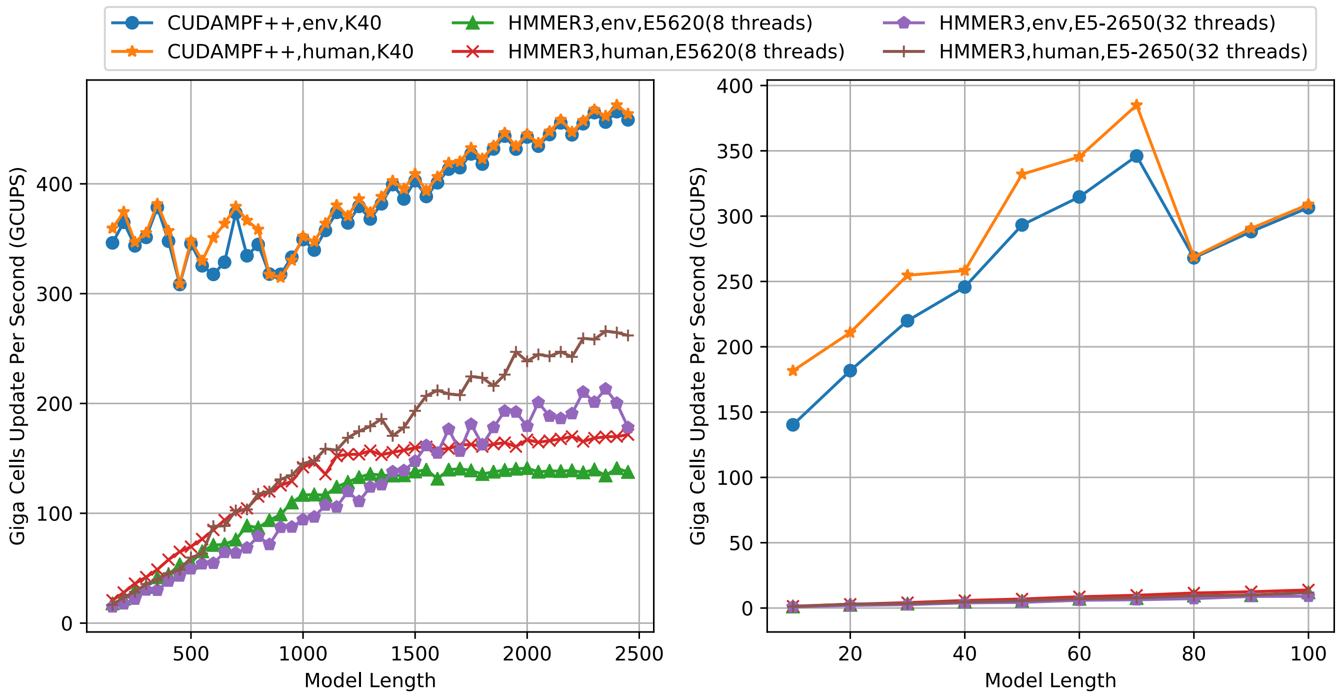


Fig. 9. Performance comparison between CUDAMPF++ and HMMER3's CPU-based implementation for the SSV kernel (stage).

existing work are highly correlated to the length of query models.

To sum up, as shown in Figs. 8 and 9, the proposed framework, CUDAMPF++, exceeds all existing work and exhibits strong consistency in performance regardless of either the model length or the amount of sequences. The reason that performance is not affected by increasing size of the database is because sequence data neither reside in on-chip memory nor consume L1 cache. Instead, they are placed in the outer loop and read once from global memory (off-chip) with the fully coalescing access. This design has been illustrated in Algorithm 2 (line 11), 7 (line 10) and 3.

5 Related Work

As one of the most popular tool for the analysis of homologous protein and nucleotide sequences, HMMER attracts many acceleration attempts. The previous version, HMMER2, is based on Viterbi algorithm that has proved to be the computational bottleneck. The initial effort of GPU-based implementation for HMMER2 is ClawHMMER [29] which introduces a streaming version of Viterbi algorithm for GPUs. They also demonstrate the implementation running on a 16-node GPU cluster, each equipped with a Radeon 9800 Pro GPU. Another early GPU-based implementation is proposed by Walters et al. [6] who properly fit the Viterbi algorithm into the CUDA-enabled GPU with several optimizations, like memory coalescing, proper kernel occupancy and shared/constant memory usage, which outperforms the ClawHMMER substantially. Yao et al. [30] present a CPU-GPU cooperative pattern to accelerate HMMER2. Ganesan et al. [7] re-design the alignment process of a single sequence across multiple threads to partially break the sequential dependency in computation of Viterbi scores. This helps building a hybrid task and data-level parallelism that eliminates the overhead due to unbalanced sequence lengths.

However, with the heuristic pipeline, HMMER3 achieves about 100x to 1000x speedups over its predecessor [3], which hence renders any acceleration effort of HMMER2 obsolete. There are only few existing work that aim to accelerate SSV, MSV and P7Viterbi stages of *hmmsearch* pipeline in HMMER3. Abbas et al. [11] re-writes mathematical formulas of MSV and Viterbi algorithms to expose reduction and prefix scan computation patterns which are fitted into the FPGA architecture. In [12], a speculative method is proposed to reduce the number of global memory access on the GPU, which aims to accelerate the MSV stage. Lin et al. [13], [14] also focus on MSV stage but incorporate SIMD video instructions provided by the CUDA-enabled GPU into their method. Like the strategy of [6], they assign each thread to handle a whole sequence. A CPU-based implementation of P7Viterbi stage is done by Ferreira et al. [15] who propose a cache-oblivious parallel SIMD Viterbi algorithm that offsets cache miss penalties of original HMMER3 work. Neto et al. [16] accelerate the SSV stage via a set of optimizations on the GPU, such as model tiling, outer loop unrolling, coalesced and vectorized memory access.

6 Discussion

While the proposed framework has achieved impressive performance on a single NVIDIA K40 GPU, we believe that using other CUDA-enabled GPUs may also benefits from the design of hierarchical parallelism, provided that compilation rules with excessive register usage are unchanged. It is easy to port the framework to run on advanced GPUs with more available hardware resources (i.e., register, cache and SMs) and gain better performance. Also, the framework naturally has linear scalability when distributing sequences to multiple GPUs. To handle large models that exceed the carrying capability of single GPU, however, one potential solution is the model partitioning that distributes different segments of a query model to different GPUs while introducing inter-device communication (i.e., max-reduction, reordering). The multi-GPU implementation of the proposed framework is being investigated. In addition, for some extreme cases that sequences within a dataset are hard to be balanced well by Algorithm 1 (Section 3.2), a hybrid partitioning approach is needed. For example, the length of few sequences is extremely long whereas the rest are very short, which will still results in unbalanced workloads after the data reformatting. In such cases, we may group or extract sequences based on their length in advance and treat them with different approaches. This scenario will be also addressed in the future work.

As for the general applicability, not only is the framework suitable for accelerating analogous algorithms of biological sequence analysis, other domain-specific applications with some features may also benefit from it. The highlight features, for example, include data irregularity, large-scale working set and complex logic with moderate dependency. Those applications can be accelerated by utilizing the proposed framework that incorporates multiple levels of parallelism. The number of applicable tiers varies with the complexity and precision of algorithms. In the contrary, for some agent-based problems that usually investigate the behavior of millions of individuals, such as molecular dynamics or simulation of spatio-temporal dynamics, our framework may not be the preferred choice. Actually, the key performance factor is the innermost loop, corresponding to 3rd, 4th and 5th tiers of the proposed framework, in which we should only put necessary operations. In general, there are several suggestions for minimizing the costs of the innermost loop if it is a performance limiter of kernels: (a) try to hold repeatedly used values in registers to avoid high-frequency communications between on and off-chip memory; (b) pre-load data needed by the innermost loop in outer loops; (c) use either L1 or texture cache to reduce the overhead of load/store operations; (d) try to use high-throughput arithmetic instructions. (e) use shuffle instructions rather than shared memory, if applicable.

Ultimately, this work sheds light a strategy to amplify the horsepower of individual GPU in an architecture-aware way while other acceleration efforts usually aim to exploit performance scaling with multiple GPUs.

7 Conclusion

In this paper, we propose a five-tiered parallel framework, CUDAMPF++, to accelerate computationally intensive tasks of the homologous sequence search with profile HMMs. This framework is based on CUDA-enabled GPUs, and it aims to fully utilize hardware resources of the GPU via

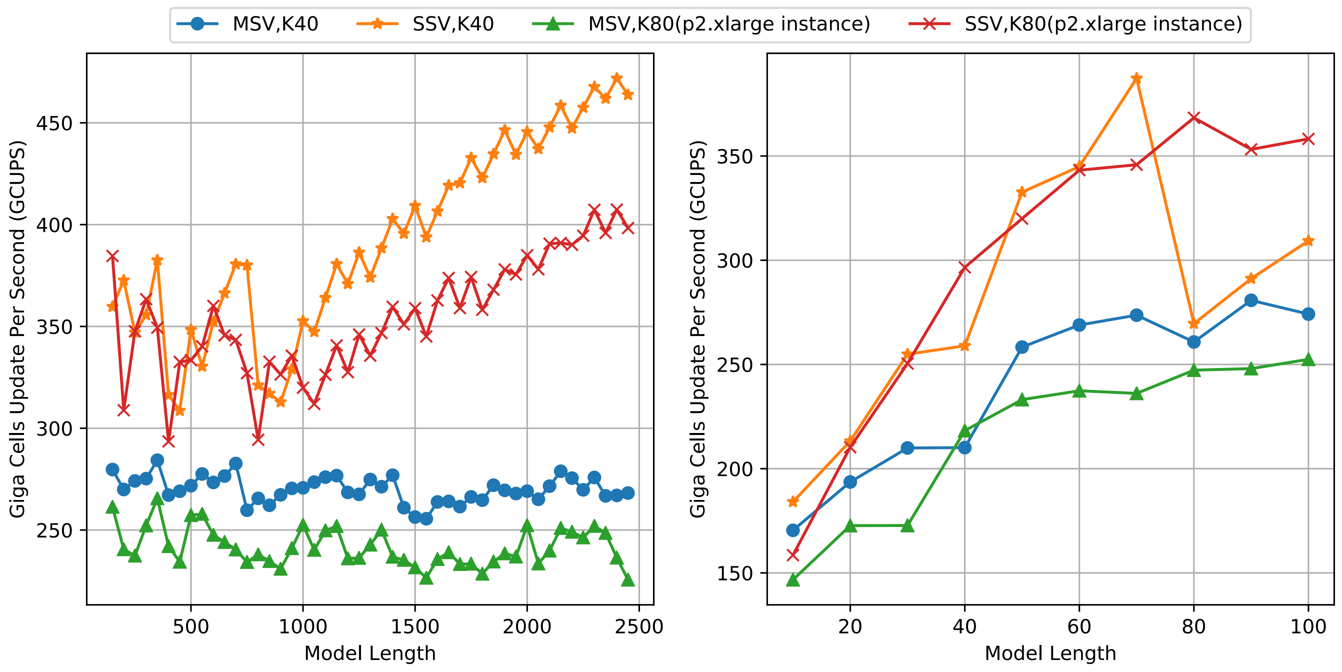


Fig. 10. Performance evaluation of CUDAMPF++ with *est_others* dataset on K40 and K80 (p2.xlarge instance) GPUs.

exploiting finer-grained parallelism (multi-sequence alignment) compared with its predecessor. In addition, we introduce a novel idea that improves the performance and scalability of the proposed framework by sacrificing L1 CHR proactively. As shown in experimental results, the optimized framework outperforms all existing work with 100 percent accuracy, and it exhibits good consistency in performance regardless of the variation of query models or sequence datasets. For MSV (SSV) kernels, the peak performance of the CUDAMPF++ is 283.9 (471.7) GCUPS on a single K40 GPU, and impressive speedups ranging from 1.8x (1.7x) to 168.3x (160.7x) are achieved over the CPU-based implementation (16 cores, 32 threads). Moreover, further generalization of the proposed framework is also discussed.

Appendix A
More Performance Evaluation

To further validate our framework and methods in terms of performance and scalability, we select another large BLASTable sequence dataset, *est_others* (41.3GB) [31], and perform experiments on both Tesla K40 and K80 (single card) GPUs. This K80 GPU is provided by *p2.xlarge* instance from Amazon Web Services [32]. As the Kepler GK210 architecture, K80 GPU consists of two physical cards, each has 13 SMXs with 2496 CUDA cores, and it supports the Compute Capability of 3.7 rather than 3.5 (K40) [33]. As for each SMX, K80 and K40 GPUs have similar amount of on-chip resources. However, on *p2.xlarge* instance, we are only allowed to use one card of K80 GPU, which makes the overall computational power slightly lower than that of a single K40 GPU.

Fig. 10 shows performance evaluation of CUDAMPF++ on both K40 and K80 GPUs with *est_others* dataset. For MSV (SSV) kernels, the average kernel throughput of 264.8 (373.1) and 237.3 (345.2) GCUPS are achieved on K40 and K80 (p2.xlarge) GPUs, respectively. Noticeably, the proposed CUDAMPF++ generally performs better on K40 GPU for MSV stage, which is expected due to the additional CUDA cores. As for SSV stage, however, both K40 and single card of K80 GPUs exhibit similar performance with irregular fluctuation for model lengths, ranging from 10 to 1000, that are carried out by multi-lane kernels ($S \neq 1$). This phenomenon proves, once again, that SSV kernels with multi-sequence alignment are mainly limited by read-only cache. Thus, in such cases, additional CUDA cores of K40 GPU will not result in an advantage in terms of kernel throughput. Overall, not only does the result validate our proposed framework on a different type of GPU, but it also proves that the kernel performance is oblivious to the size of biological sequence databases. The GPU memory capacity, in our case, is not a roadblock when scaling up the amount of sequences since we can easily split large datasets into many small chunks for the asynchronous streaming and batch processing. Some advanced GPUs that support Unified Memory (i.e., devices of Compute Capacity 6.0 and above) [21] can even eliminate the boundary between device (GPU) and host (CPU) memory. Moreover, the host memory capacity only has an effect on the overlap of data transfer and kernel execution. Although sufficient amount of host memory is non-essential for handling large databases, keeping sequence data always in page-locked memory helps hide the overhead of data transfer.

Appendix B
Performance Impact of Warp-based Data Reformatting

As one of key features of CUDAMPF++, warp-based data reformatting scheme is designed to support the proposed five-tiered framework via reducing thread idling and the overhead of memory access. Table 6 lists the performance gain of MSV/SSV kernels after using reformatting data. Columns with title “regular” represent the performance of

TABLE 6
Performance Comparison Between using Sequential Data Layout and Warp-Based Data Blocks in CUDAMPF++

	MSV kernels				SSV kernels			
S-lane kernels	acc. ID	regular (GCUPS)	warp-based (GCUPS)	speedup	acc. ID	regular (GCUPS)	warp-based (GCUPS)	speedup
NVIDIA Tesla K40								
128	PF13823.1	25.5	168.6	6.6x	PF02822.9	30.2	178.4	5.9x
64	PF09051.5	45.5	225.7	5.0x	PF02770.14	56.0	276.1	4.9x
32	PF00207.17	67.1	274.1	4.1x	PF14807.1	79.3	307.8	3.9x
16	PF00596.16	76.7	266.7	3.5x	PF15420.1	90.2	305.0	3.4x
8	PF05208.8	78.4	266.7	3.4x	PF06808.7	90.8	302.6	3.3x
4	PF03971.9	77.3	262.4	3.4x	PB001474	91.5	302.1	3.3x
2	PB006678	77.2	259.7	3.4x	PB000663	90.2	293.1	3.2x
NVIDIA Tesla K80 (p2.xlarge)								
128	PF13823.1	28.4	109.1	3.8x	PF02822.9	34.4	131.0	3.8x
64	PF09051.5	50.2	169.6	3.4x	PF02770.14	62.6	205.8	3.3x
32	PF00207.17	72.2	215.4	3.0x	PF14807.1	86.0	246.2	2.9x
16	PF00596.16	80.6	235.3	2.9x	PF15420.1	95.0	280.3	3.0x
8	PF05208.8	82.4	248.5	3.0x	PF06808.7	95.5	283.8	3.0x
4	PF03971.9	80.5	246.1	3.1x	PB001474	95.9	286.7	3.0x
2	PB006678	80.5	245.9	3.1x	PB000663	94.1	281.9	3.0x

**** The* env_nr *[24] is used in data collection.*

CUDAMPF++ with sequential data layout. Speedups ranging from 2.9x to 6.6x are observed for different query models in the benchmark. Referring to curves in Figs. 6 and 7 (Section 4.3), we find that, without warp-based data reformatting, CUDAMPF++ cannot outperform CUDAMPF in most cases. Thus, reformatting sequence dataset into warp-based data blocks is an essential component in this work to achieve high-throughput processing.

ACKNOWLEDGMENTS

The authors would like to thank the NVIDIA for generous donations in carrying out this research.

REFERENCES

[1] N. Marco S., C. Paolo, T. Andrea, and B. Daniela, “Graphics processing units in bioinformatics, computational biology and systems biology,” *Briefings Bioinf.*, vol. 18, no. 5, pp. 870–885, Sep. 2017.

[2] S. Eddy, “Profile hidden markov models,” *Bioinf.*, vol. 14, pp. 755–763, 1998.

[3] S. Eddy, “Accelerated profile HMM searches,” *PLoS Comput. Biology*, vol. 7, no. 10, 2011, Art. no. e1002195.

[4] K. Anders, B. Michael, M. I. Saira, S. Kiminen, and H. David, “Hidden Markov models in computational biology: Applications to protein modeling,” *J. Molecular Biol.*, vol. 235, pp. 1501–1531, 1994.

[5] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman, “Basic local alignment search tool,” *J. Molecular Biol.*, vol. 215, no. 3, pp. 403–410, 1990.

[6] J. P. Walters, V. Balu, S. Kompalli, and V. Chaudhary, “Evaluating the use of GPUs in liver image segmentation and HMMER database searches,” in *Proc. Int. Symp. Parallel Distrib. Process.*, 2009, pp. 1–12.

[7] N. Ganesan, R. D. Chamberlain, J. Buhler, and M. Taufer, “Accelerating HMMER on GPUs by implementing hybrid data and task parallelism,” in *Proc. 1st ACM Int. Conf. Bioinf. Comput. Biol.*, 2010, pp. 418–421.

[8] R. Maddimsetty, J. Buhler, R. Chamberlain, M. Franklin, and B. Harris, “Accelerator design for protein sequence HMM search,” in *Proc. 20th ACM Int. Conf. Supercomputing*, 2006, pp. 288–296.

[9] T. Oliver, L. Y. Yeow, and B. Schmidt, “Integrating FPGA acceleration into HMMer,” *Parallel Comput.*, vol. 34, no. 11, pp. 681–691, 2008.

[10] T. Takagi and T. Maruyama, “Accelerating HMMER search using FPGA,” in *Int. Conf. Field Programmable Logic Appl.*, 2009, pp. 332–337.

[11] N. Abbas, S. Derrien, S. Rajopadye, and P. Quinton, “Accelerating HMMER on FPGA using parallel prefixes and reductions,” in *Proc. Int. Conf. Field-Programmable Technol.*, 2010, pp. 37–44.

[12] X. Li, W. Han, G. Liu, H. An, M. Xu, W. Zhou, and Q. Li, “A speculative HMMER search implementation on GPU,” in *Proc. 26th IPDPS Workshop PhD Forum*, 2012, pp. 73–74.

[13] L. Cheng and G. Butler, “Implementing and accelerating HMMER3 protein sequence search on CUDA-enabled GPU,” Ph.D. dissertation, Dept. Comput. Sci. Softw. Eng., Concordia Univ., Montral, QC, 2014.

[14] L. Cheng and G. Butler, “Accelerating search of protein sequence databases using CUDA-enabled GPU,” in *Proc. 20th Int. Conf. Database Syst. Adv. Appl.*, 2015, pp. 279–298.

[15] M. Ferreira, N. Roma, and L. Russo, “Cache-oblivious parallel SIMD viterbi decoding for sequence search in HMMER,” *BMC Bioinf.*, vol. 15, 2014, Art. no. 165.

[16] A. C. de Arajo Neto and N. Moreano, “Acceleration of single- and multiple-segment viterbi algorithms for biological sequence-profile comparison on GPU,” in *Proc. 21st Int. Conf. Parallel Distrib. Process. Techn. Appl.*, 2015, pp. 65–71.

[17] H. Jiang and G. Narayan, “Fine-grained acceleration of HMMER 3.0 via architecture-aware optimization on massively parallel processors,” in *Proc. 14th IEEE Int. Workshop High Perform. Comput. Biol.*, 2015, pp. 375–383.

[18] H. Jiang and N. Ganesan, “CUDAMPF: A multi-tiered parallel framework for accelerating protein sequence search in HMMER on CUDA-enabled GPU,” *BMC Bioinf.*, vol. 17, no. 1, 2016, Art. no. 106.

[19] K. Bjarne, “The SSV filter implementation,” *HMMER v3.1b2 Source Code*, Mar. 2015, http://eddylab.org/software/hmmer3/3.1b2/hmmer-3.1b2.tar.gz

[20] G. M. Amdahl, “Validity of the single-processor approach to achieving large-scale computing capabilities,” in *Proc. Sprint Joint Comput. Conf. Amer. Fed. Inf. Process. Societies*, 1967, pp. 483–485.

[21] NVIDIA, “CUDA C programming guide,” Jun. 2017. [Online]. Available: http://docs.nvidia.com/pdf/CUDA_C_Programming_Guide.pdf

[22] NVIDIA, “NVIDIAs next generation CUDA compute architecture: Kepler GK110/210,” nVIDIA Corporation Whitepaper. 2014. [Online]. Available: http://international.download.nvidia.com/pdf/kepler/NVIDIA-Kepler-GK110GK110%-GK210-Architecture-Whitepaper.pdf

[23] NVIDIA, “Parallel thread execution ISA,” Jun. 2017. [Online]. Available: http://docs.nvidia.com/pdf/ptx_isa_5.0.pdf

[24] FASTA sequence databases. Sep. 2014. [Online]. Available: ftp://ftp.ncbi.nlm.nih.gov/blast/db/FASTA/

[25] NVIDIA, "NVIDIA Tesla GPU accelerators," 2013. [Online]. Available: http://www.nvidia.com/content/tesla/pdf/NVIDIA-Tesla-Kepler-Family-Data sheet.pdf

[26] NVIDIA, "Profiler user's guide," Jun. 2017. [Online]. Available: http://docs.nvidia.com/pdf/CUDA_Profiler_Users_Guide.pdf

[27] Pfam: Protein Family Database. May 2013. [Online]. Available: ftp://ftp.ebi.ac.uk/pub/databases/Pfam/releases/Pfam27.0/

[28] S. Karl and N. Perry, "The little redis book," 2012. [Online]. Available: http://openmymind.net/redis.pdf

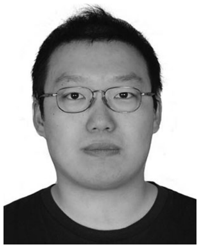
[29] D. Horn, M. Houston, and P. Hanrahan, "ClawHMMER: A streaming HMMer-search implementation," in *Proc. ACM/IEEE Supercomputing Conf.*, 2005, pp. 11–11.

[30] Y. Ping, A. Hong, X. Mu, L. Gu, L. Xiaoqiang, W. Yaobin, and H. Wenting, "CuHMMer: A load-balanced CPU-GPU cooperative bioinformatics application," in *Proc. Int. Conf. High Perform. Comput. Simul.*, 2010, pp. 24–30.

[31] FASTA Sequence Databases. Jan. 2018. [Online]. Available: ftp://ftp.ncbi.nlm.nih.gov/blast/db/FASTA/

[32] J. Varia and S. Mathew, "Overview of amazon web services," Jan. 2014, http://d36cz9buwru1tt.cloudfront.net/AWS_Overview.pdf

[33] NVIDIA, "Tesla K80 GPU accelerator," Jan. 2015. [Online]. Available: https://images.nvidia.com/content/pdf/kepler/Tesla-K80-BoardSpec-07317–001-v05.pdf



Hanyu Jiang received the BS degree in control science and engineering from the Harbin Institute of Technology, Harbin, China, in 2012, and the MEng degree of computer engineering from the Stevens Institute of Technology, in 2014. He is currently working toward the PhD degree in computer engineering in the Stevens Institute of Technology, Hoboken, New Jersey. His current research interests include heterogeneous and parallel computing, multi-core processor architecture, bioinformatics, and artificial intelligence. He is a student member of the IEEE.

Narayan Ganesan received the PhD degree from Washington University, St. Louis, in Dec. 2006. He is an assistant professor of electrical and computer engineering with the Stevens Institute of Technology. He worked on designing novel compute architectures utilizing massively parallel processors and reconfigurable hardware for scientific computing. Prior to joining Stevens in 2011, he was a research associate with the University of Delaware, where he designed algorithms for accelerating problems in computational chemistry. His research interests include designing efficient algorithms and computing architectures for handling big-data and big-computation problems in molecular dynamics, computational biology, bioinformatics, and healthcare notification systems. He is a senior member of the IEEE.

Yu-Dong Yao (S'88-M'88-SM'94-F'11) received the BEng and MEng degrees from Nanjing University of Posts and Telecommunications, Nanjing, China, in 1982 and 1985, respectively, and the PhD degree from Southeast University, Nanjing, in 1988, all in electrical engineering. From 1987 to 1988, he was a visiting student with Carleton University, Ottawa, Canada. From 1989 to 2000, he was with Carleton University, Spar Aerospace Ltd., Montreal, Canada, and Qualcomm Inc., San Diego. Since 2000, he has been with Stevens Institute of Technology, Hoboken, where he is currently a professor and chair of the Department of Electrical and Computer Engineering. He holds one Chinese patent and 13 U.S. patents. His research interests include wireless communications, cognitive radio, and machine learning and deep learning techniques. He served as an associate editor for the *IEEE Communications Letters* (2000 to 2008) and the *IEEE Transactions on Vehicular Technology* (2001 to 2006) and as an editor for the *IEEE Transactions on Wireless Communications* (2001 to 2005). For his contributions to wireless communications systems, he was elected a fellow of IEEE (2011), National Academy of Inventors (2015), and Canadian Academy of Engineering (2017). He is a fellow of the IEEE.