Genomics-GPU: A Benchmark Suite for GPU-accelerated Genome Analysis

Zhuren Liu*, Shouzhe Zhang*, Justin Garrigus, Hui Zhao
Department of Computer Science and Engineering, University of North Texas
{zhurenliu, shouzhezhang, justingarrigus}@my.unt.edu, hui.zhao@unt.edu,

***Abstract*—Genomic analysis is the study of genes which includes the identification, measurement, or comparison of genomic features. Genomics research is of great importance to our society because it can be used to detect diseases, create vaccines, and develop drugs and treatments. As a type of general-purpose accelerators with massive parallel processing capability, GPUs have been recently used for genomics analysis. Developing GPU-based hardware and software frameworks for genome analysis is becoming a promising research area. To support this type of research, benchmarks are needed that can feature representative, concurrent, and diverse applications running on GPUs. In this work, we created a benchmark suite called Genomics-GPU, which contains 10 widely-used genomic analysis applications. It covers genome comparison, matching, and clustering for DNAs and RNAs. We also adapted these applications to exploit the CUDA Dynamic Parallelism (CDP), a recent advanced feature supporting dynamic GPU programming, to further improve the performance. Our benchmark suite can serve as a basis for algorithm optimization and also facilitate GPU architecture development for genomics analysis.**

***Index Terms*—genomics, bioinformatics, benchmarking, GPU, accelerated computing, genome analysis, computer architecture.**

I. Introduction

Genome sequence analysis refers to the study of an organism's DNA sequence. This procedure has many important applications, such as pandemic outbreak tracing, early cancer detection [79], drug development [43], and genetic disease identification [87]. To analyze an organism's genome computationally, the DNA molecule must first be converted to digital data in the form of a string of four letters (A, C, T, and G), also known as bases or nucleotides. The process of determining the sequence of the bases is known as genome sequencing [30]. The process of comparing and discovering differences between biological sequences is known as sequence alignment [67]. The last decade has seen exponential growth in genomic databases and a large amount of data needs to be analyzed with the help of computation tools. As a result, several widely-used tools for genome analysis have been developed, such as BLAST [57] and GATK [58].

To improve the performance, SIMD-capable CPUs have been employed by some genome sequencing frameworks, such as Parasail [31] and KSW2 [53]. They take advantage of the parallelism offered by SIMD instructions to execute matrix computations by running the same vector command on multiple operands in parallel. FPGASW [39] uses the large number of execution units in FPGAs to create a linear systolic array. Execution units in the systolic array can compute the value of a single entry of the dynamic programming matrix in the algorithm. ASIC-based hardware platforms have also been developed to explore performance, power, and area tradeoffs, including ASAP [21], and GenAx [41]. However, systems accelerated with FPGAs or ASICs usually require user knowledge of the underlying hardware on how to program them.

In contrast, General Purpose Graphics Processing Units (GPGPUs) are a type of general-purpose accelerators which are very efficient in executing SIMD commands. GPU-based frameworks can significantly reduce the execution time for genomic analysis. For example, the GPU-based GASAL2 [15] can improve the performance by 20× compared to CPU. CUDA Dynamic Parallelism (CDP) [1] was introduced to the CUDA programming model. It enables GPU threads to be launched dynamically, simultaneously, and independently [1], [25], [47], [52], [81], [83], [88].

To steer the genomics research in the right direction, benchmarks are of paramount importance to help draw correct conclusions through rigorous evaluations. There have been several benchmarks created for GPU computing, such as ispass-2009 [7], Parboil [75], Rodinia [27] and etc. These benchmarks target on high-performance computing and do not focus on genomics applications. Benchmarks such as BioPerf [19], BioBench [17], and MineBench [60] were developed for genomics research. However, they were created a decade ago and do not include today's popular sequencing techniques. GenomicsBench [76] is the most recent work on genomics benchmarking. However, GenomicsBench focuses on CPU-based applications and only has three GPU benchmarks. Therefore, there is a dire need to create a GPU-based genomics benchmark covering concurrent techniques to facilitate research in this fast-growing area.

In this paper, we compiled ten representative GPU-based genomics applications. We made adaptations with up-to-date libraries so that they are compatible with today's software and hardware. For each benchmark, we created a second version to support execution with CDP [1], [4]. We ran the benchmarks on an NVIDIA RTX 3070 GPU [6] to collect performance results using hardware performance counters. In order to provide more in-depth analysis at the microarchitectural level, we also ran them with a state-of-the-art GPU simulator (Accel-Sim [50]) to collect data not available in hardware. We performed a comprehensive characterization of

*Both authors contributed equally to this work

the applications and analyzed the performance bottleneck.

This paper makes the following contributions:

- We present the Genomics-GPU benchmark suite consisting of 10 representative kernels spanning the major algorithms in genomics analysis, such as genome comparison, matching, scoring, clustering and variant selection. We also provide input datasets of different sizes.
- For each benchmark, we created an additional version to support the advanced CUDA Dynamic Parallelism (CDP) feature. We provided a comparison between the CDP and non-CDP implementations.
- We performed a detailed evaluation of these benchmarks on both hardware and software simulators. We made adaptations to the source code so that they can be executed with different versions of libraries required by the hardware and simulator.
- We provided in-depth characterization and analysis of the benchmarks at microarchitectural levels, such as memory access pattern, thread scaling behavior, and pipeline stall breakdown.

II. Background

A. GPU architecture

Genomic analysis typically generates a large amount of data per genome. It requires massive computing power to process the data, which poses a challenge to even the fastest CPUs. GPUs have become a popular platform for accelerating parallel executions due to their extraordinary parallel computing power. GPU programming models allow users to create thousands of threads and can significantly improve the system throughput. The high throughput of GPUs is realized by grouping threads into fixed-size SIMD batches known as warps, and then many such warps are concurrently executed on a single GPU core. Figure 1 shows the general architecture of a modern GPU. It consists of multiple Streaming Multiprocessors (SMs) and several layers of memory partitions. Streaming processors are connected by the interconnect network. Each Streaming Multiprocessor contains control units, registers, execution pipelines, scratchpad memory, and caches.

B. Cuda Dynamic Parallelism (CDP)

Cuda Dynamic Parallelism (CDP) [1], [83] is an advanced feature in the Nvidia GPU programming model that was first released with the Kepler GK110 [63] architecture. It enables kernels to start from GPU devices without returning to the host, allowing a user to call a global or host function within a certain global or host function [1], [81], [88]. A parent can be a kernel, block, or thread that initiates the device launch, and a child can be a kernel, block, or thread launched by a parent. `cudaDeviceSynchronize` is a device runtime API that synchronizes between the child and parent. Each launch can be nested, e.g., from parent to child and then from a child to a child within a child, and so on. Synchronization depth is how far nesting can go with explicit synchronization. When an explicit synchronization is called, the parent will have to halt and yield to the child kernels. However, if there is no explicit synchronization, the execution order of the child

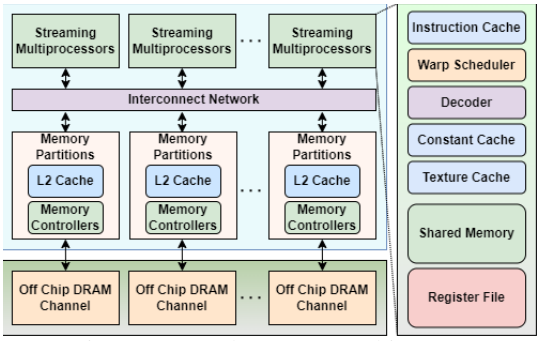


Fig. 1: General GPGPU architecture

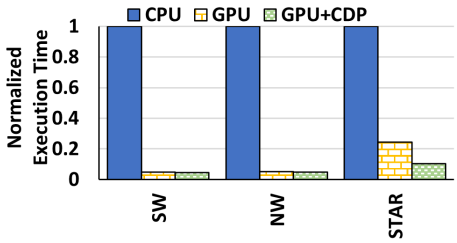


Fig. 2: Performance Comparison between CPU and GPU.

and parent will be random. Although executing each child kernel concurrently may still be possible, there is no guarantee without explicit synchronization. Shared memory and local memory are exclusive for parent and child kernels, while global memory can be shared by both. The overheads for CDP implementations include API calls, kernel parameter parsing, device runtime setups, and the management of child kernels [82], [83]. A bigger input size can alleviate these overheads and result in better performance.

C. Performance comparison between CPUs and GPUs

Many genomic analysis algorithms were developed for CPUs before GPUs were widely used as accelerators. Only in recent years have researchers started to create new algorithms or port previous algorithms to GPUs. Although it might be more convenient to develop CPU-based implementations using conventional programming models, CPUs can have a big performance disadvantage compared with GPUs. We compared the performance for the algorithms of Smith-Waterman (*SW*) [74], Needleman–Wunsch (*NW*) [62], and Center Star Algorithm (*STAR*) [29] using high-performing CPUs and GPUs. The result is shown in Figure 2. For each algorithm, we collected execution time for CPU implementations, GPU implementations, and a GPU implementation with CDP. The experiments were run on Lonestar 6 [8], one of the high-performance computing systems of Texas Advanced Computing Center (TACC) [64]. Same inputs were used for different implementations. The results are normalized to CPU performance. GPUs can lead to a huge performance gain and achieve a speedup as much as 20x compared with CPUs. For *STAR*, CDP can further reduce the execution time by more than half.

III. Methodology

A. Genomics application description

Needleman–Wunsch (NW) [45], [62] is used for sequence alignment and belongs to the global alignment family by

TABLE I: Hardware configuration settings

Configuration	Settings
Shader Cores	**78**
Warp Size	**32**
Constant Cache Size / Core	**64KB** (256-way set assoc. 128B lines LRU)
Texture Cache Size / Core	**128KB** (64-way set assoc. 128B lines LRU)
Number of Registers / Core	16384, 32768, **65536**, 131072, 262144
Number of CTAs / Core	8, 16, **32**, 64, 128
Number of Threads / Core	384, 768, **1536**, 3072, 6144
Shared Memory / Core (KB)	32, 64, **100**, 256, 512
L1 Cache	32KB, **128KB**, 256KB, 512KB, 4MB (256-way set assoc. 128B lines LRU)
L2 Cache	512KB, **4M**, 8MB, 16MB, 128MB (16-way set assoc. 128B lines LRU)
Memory Controller	**out of order(FR-FCFS)**, in order(FIFO)
Scheduler	**LRR**, GTO, OLD

TABLE II: Interconnect Configuration settings

Configuration	Settings
Topology	Mesh, **Local Xbar**, Fat Tree, Butterfly
Routing Mechanism	**Dimension Order**, Destination Tag, Nearest Common Ancestor
Routing delay	**0**
Virtual channels	**2**
Virtual channel buffers	**4**
Flit size (Bytes)	8, 16, 32, **40**
Alloc iters	**1**
VC alloc delay	**1**
Input Speedup	**2**

finding the best match in the entire sequence. This algorithm employs dynamic programming to implement genome alignment and can reduce the number of possibilities while still ensuring the best solution is found [62], [84]. This algorithm is one of the most widely used in genome alignment.

Smith–Waterman (SW) [56], [74] is a sequence alignment algorithm that belongs to local alignment. *SW* is also a dynamic programming algorithm that can provide conserved regions between two sequences and can align two partially overlapping sequences [74], [80], [84]. This algorithm can also align a sequence's subsequence to itself. It is also a widely used genome alignment algorithm.

Center Star Algorithm (STAR) [72], [90] provides a reliable and efficient Multiple Sequences Alignment (MSA) solution for large-scale datasets. This algorithm automatically performs and optimizes multiple sequence alignments for user-submitted sequences without making any assumptions. STAR employs a co-running approach to fully leverage both the CPU and GPU devices. It can quickly and precisely align genomes for mining single nucleotide polymorphisms (SNPs) and copy-number variants (CNV) [72], [89], [90].

GASAL2 [15] is an extension of GASAL [16] that is a CUDA library for DNA/RNA sequence alignment algorithms [15], [59]. *GASAL2* supports four kinds of alignments, including global alignment (**GG**), local alignment (**GL** and **GKSW**), semi-global alignment (**GSG**), and tile-based banded alignment with or without traceback. *GASAL2* outperforms the fastest CPU-optimized SIMD implementations, including SeqAn [66] and Parasail [31], as well as NVIDIA's own GPU-based library, *NVBIO*, with an overall performance speedup of

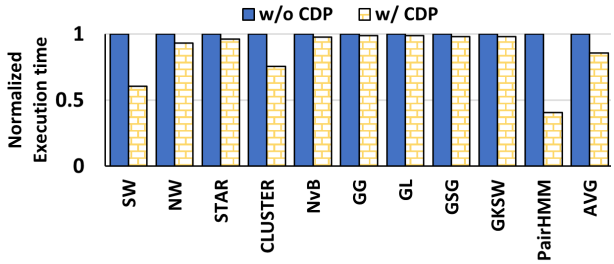


Fig. 3: Performance Comparison With CDP and Without CDP.

2x, 21x and 13x respectively [15]. According to Alser et al. [18], it is one of the most representative genomics algorithms employing GPUs. Our benchmark suite includes *GG, GL, GKSW*, and *GSG* from *GASAL2*.

nGIA (CLUSTER) [48] is a precise and efficient greedy incremental alignment-based (GIA) algorithm and clustering tool [48], [73]. *nGIA* is composed of a pre-filter, a modified short word filter, a new data packing strategy, a modified greedy incremental method, and is GPU parallelized. *nGIA* is much faster compared with other cluster algorithms, such as CD-HIT [40], Vsearch [69], and Uclust [38] [48].

Pair Hidden Markov Model (PairHMM) [68], [71] is a variant of the standard HMM [65]. It is particularly effective for discovering sequence alignments and assessing the relevance of the aligned symbols. In contrast to the traditional HMM, which produces only a single sequence, Pair-HMM produces an aligned pair of sequences [86]. Prior research results show that the GPU-based PairHMM forward algorithm outperforms existing CPU-based implementations by up to 5.47 times [68].

NVBIO (NvB) [10], [46] is a reusable component library created by NVIDIA for bioinformatics applications using CUDA. The program was initially developed to harness the potential of NVIDIA GPUs, and the majority of its components are totally cross-platform and include both host C++ and device CUDA code [46]. We select NvBowtie from the NVBIO library to add to our Genomics-GPU suite. NvBowtie is a CUDA implementation of the Bowtie2 [3], which is an ultrafast and memory-efficient tool for aligning sequencing reads to long reference sequences [51].

B. CUDA Dynamic Parallelism implementation

Cuda Dynamic Parallelism (CDP) [1], [4] is an important design feature that enables further exploration of parallelism and allows the parallelism to be represented more precisely [35], [36]. For genomics applications with massive datasets, CDP offers mechanisms for nested kernel launch and can bring benefits to recursive applications, such as divide-and-conquer [25]. As shown in Figure 3, we observe that, compared to non-CDP implementations, using CDP can improve the kernel execution time by up to 59%, with an average improvement of 14%. In our optimization using CDP, we employed a similar method as Wang et al. [83] by modifying the original kernels and converting them to child kernels, as shown in Listing 1. The key idea is to use the child kernel to replace the original

TABLE III: Benchmark Properties

Benchmark	Abr.	Input	Grid	CTA	Shared Memory?	Constant Memory?	CTA/CORE
Smith-Waterman [56], [74]	**SW**	32K bases with 4 types (A/C/G/T)	(3,1,1)	(64,1,1)	NO	YES	30
Needleman–Wunsch [45], [62]	**NW**	32K bases with 4 types (A/C/G/T)	(500, 1, 1)	(128, 1, 1)	YES	YES	6
Center Star Algorithm [72], [90]	**STAR**	protein.txt [72]	(12, 1, 1)	(256, 1, 1)	NO	YES	4
GASAL2 GLOBAL [15], [59]	**GG**	query_batch.fasta [59]	(40, 1, 1)	(128, 1, 1)	NO	YES	12
GASAL2 LOCAL [15], [59]	**GL**	query_batch.fasta [59]	(40, 1, 1)	(128, 1, 1)	NO	YES	12
GASAL2 KSW [15], [59]	**GKSW**	query_batch.fasta [59]	(40, 1, 1)	(128, 1, 1)	NO	YES	12
GASAL2 SEMI-GLOBAL [15], [59]	**GSG**	query_batch.fasta [59]	(40, 1, 1)	(128, 1, 1)	NO	YES	12
Greedy Incremental Alignment-based [48], [73]	**CLUSTER**	testData.fasta [73]	(128, 1, 1)	(128, 1, 1)	YES	YES	12
Pair Hidden Markov Model [68], [71]	**PairHMM**	Synthetic_data(128_128) [71]	(150, 1, 1)	(128, 1, 1)	YES	YES	10
NVBIO [10], [46]	**NvB**	hg19.fa [78], SRR493095.fastq [22], [61]	(2048,1,1)	(256,1,1)	NO	YES	6

parallel loops in the non-CDP implementation.

```
1 //For each parent kernel execute child kernel
2 argument, ... = kernelOri(parameters, ...);
3 if(condition(argument))
4     KernelCDP <<<BLOCKS, THREADS>>> (argument, ...);
5         KernelOri <<<blocks, threads>>> (parameters,
              ...);
```

Listing 1: Code structure of Cuda Dynamic Parallelism (CDP)

C. Experiment environment

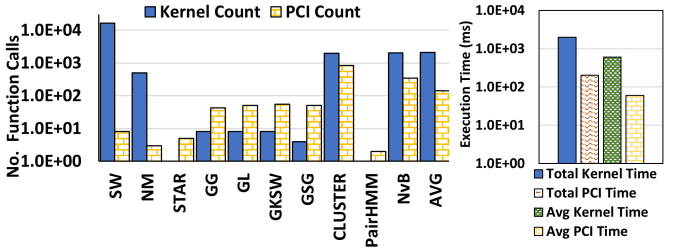
All of our benchmarks are run with both hardware GPU and a software simulator (Accel-Sim 4.0 [50]). We run all benchmarks to completion on both hardware and simulator. Table I lists the configurations we used in our evaluation. Highlighted settings represent configurations used by the hardware GPU which are also the default configurations for simulation. The rest of the configurations were run with the simulator. The hardware GPU is Ampere RTX 3070 [6], with over 5500 CUDA cores running at a 1.50 GHz base clock rate. We ran the Accel-Sim simulator on servers at the Texas Advanced Computing Center (TACC) [64]. Those servers are equipped with two AMD EPYC 7763 64-Core Processor ("Milan") CPUs [2], with a base clock rate of 2.45 GHz. To maintain consistency between the hardware and simulation platforms, we used the same configuration as the baseline for both of them. Table II shows the configurations for the interconnection network. We used a detailed interconnection network simulator called Booksim [32] to collect on-chip network results.

Table III describes the benchmark properties, such as the CTA dimensions, the grid dimensions, memory types, and the input datasets. To collect hardware GPU data, we used NVIDIA's universal GPU profiler called nvprof [12] and the NVIDIA Nsight Systems tool [14]. Our experiments were run with Ubuntu 18.04 and CUDA 11.4.

IV. Results and Analysis

A. Kernel execution pattern

We first measured the number of kernel function calls the CPU has invoked on the GPU. We used NVIDIA profiling tools nvproof [12] and NVIDIA Nsight Systems [14] to collect this information. Figure 4(a) shows the kernel function invocation counts. We are only able to collect data for the non-CDP versions. This is because the NVIDIA profiling tools do not support the tracing of CDP kernels for Volta (compute capability 7.0) and higher GPU architectures [23]. The GPU we used is Ampere RTX 3070, which has a compute capability of 8.6. The "Kernel count" in Figure 4(a) represents the times of each application's kernel function being called during the execution. PCI count represents the number of cudaMemcpy function calls in an application. cudaMemcpy function calls are used to transfer data between the CPU and GPU using PCI, which is the communication interface between them. We can observe that the number of kernel function calls, for most applications, is higher than the number of PCIs. For *SW* and *NW*, kernel function calls greatly outnumbered the PCI calls. This indicates that these applications are more intensive in computation than communication. *GASAL2* applications, on the other hand, have more PCI transactions than kernel function calls. This indicates that there is a large communication overhead when running these applications, and data movement can become a potential performance bottleneck. Figure 4(b)



(a) (b)

Fig. 4: (a) Kernel function invocation count. (b) Average function execution times

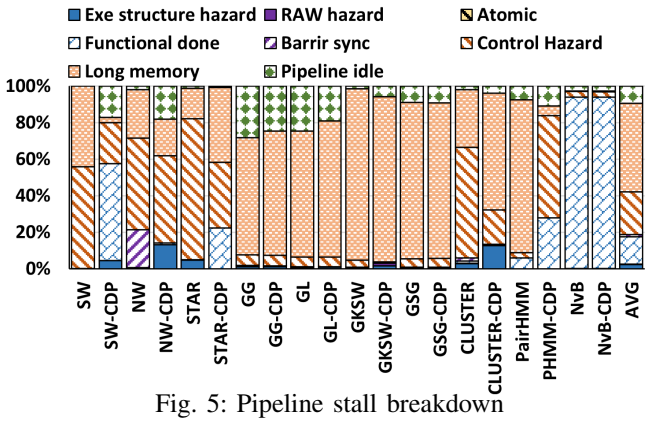


Fig. 5: Pipeline stall breakdown

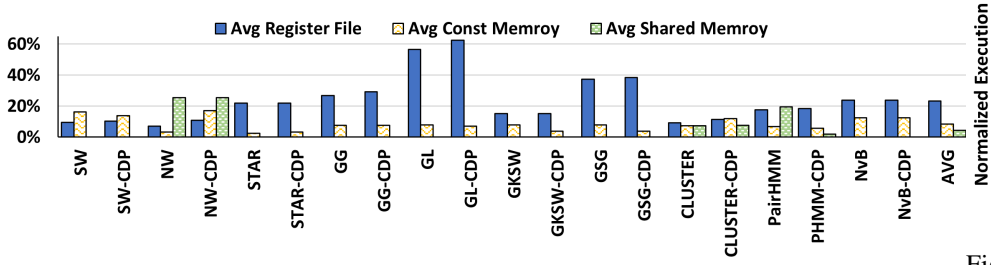


Fig. 6: Utilization of SRAM structures

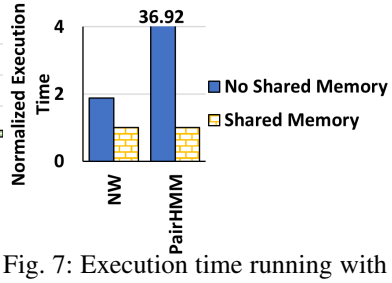


Fig. 7: Execution time running with or without shared memory

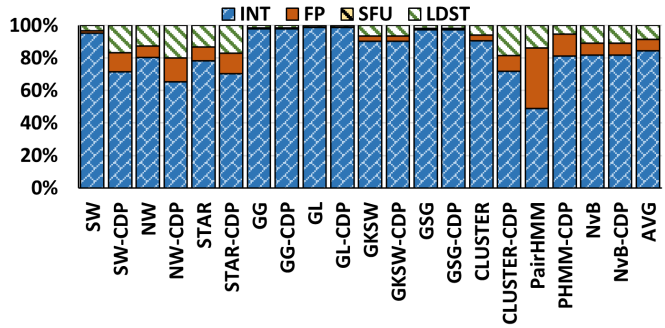


Fig. 8: Distribution of instruction types

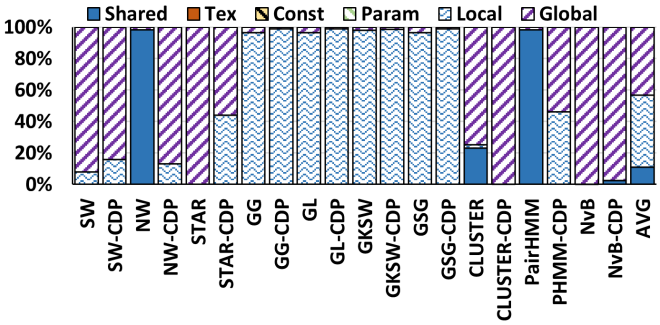


Fig. 9: Distribution of memory instruction types

shows the total and average execution time spent in kernel and PCI invocations. In general, the total and average time spent on PCI is quite significant compared with the time spent in kernel execution, making the data movement a critical performance bottleneck in these genomics applications.

B. Performance bottlenecks

In this section, we evaluate different types of pipeline stalls and their impact on performance. We used Accel-Sim simulator [50] to collect experiment results. Figure 5 shows the breakdown of each pipeline stall cause. It can be observed that the breakdown of pipeline stalls between CDP and non-CDP implementations is similar. The most significant cause of pipeline stalls is long memory latency, accounting for up to 95% of all pipeline stalls. This is because genome applications work with very large datasets and frequent off-chip memory accesses cause pipelines to wait for the data. Control hazards and pipeline idle are the next two most significant causes for pipeline stalls. We can also observe that, for *NvB* and *NvB-CDP*, functional done causes over 90% of all their stalls. Functional done means that GPU cores are waiting to be initialized to run a new kernel. There exist many CUDA API calls in *NvB* and *NvB-CDP*. Therefore, a lot of time is spent on switching between the kernels.

C. SRAM structure utilization

SRAMs are the largest resource in GPU processing cores, including register files, shared memory, and constant memory. We collected the usage of SRAM resources using the RTX 3070 as our experimental platform. RTX 3070 comes with a register file of 64KB, a shared memory of 100KB, and a constant memory of 64 KB. We collected the usage of these SRAM resources through an NVCC [11] compiler option "Xptxas=-v". The total usage of the SRAM resources is calculated using the concurrent number of CTAs per Stream

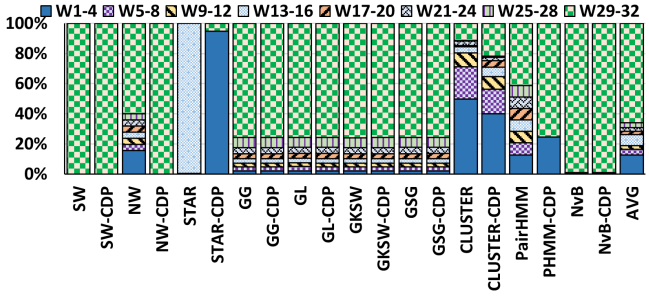


Fig. 10: Warp Occupancy

Multiprocessors (SM) and the number of threads within each CTA. The result is shown in Figure 6.

As can be observed, register file is more utilized among the three types of SRAMs. Constant memory is less used in comparison. Only a few applications use shared memory, such as *NW*, *CLUSTER*, and *PairHMM*. The lack of shared memory usage is related to the programmer's coding style. Programming with shared memory is a unique feature for GPUs, and it takes extra steps rather than using the global memories directly. We observe that appropriate usage of shared memory can greatly benefit performance. We compared the performance between running with and without shared memory for *NW* and *PairHMM* and show the result in Figure 7. Here the execution time is normalized to those using shared memory. Using shared memory can improve performance by 1.88 times for benchmark *NW* while the performance can be improved by 36.92 times for benchmark *PairHMM*. Many genomics programs were originally created for CPUs and then ported to GPUs and shared memory might be neglected during the porting procedure. Our observation suggests that rather than straightforward translation, taking advantage of shared memory can better exploit GPU's computing power.

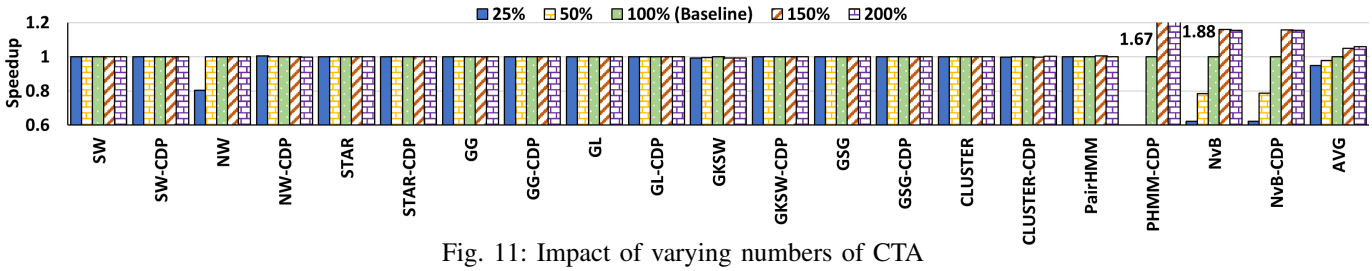


Fig. 11: Impact of varying numbers of CTA

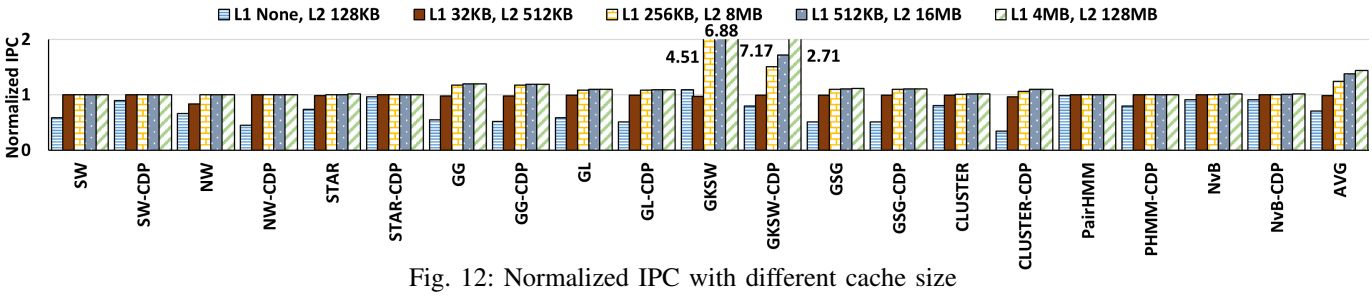


Fig. 12: Normalized IPC with different cache size

D. Instruction types

Figure 8 shows the different instruction types executed when running the non-CDP and CDP variants of the applications. As can be observed, integer instructions are the most commonly executed instructions, followed by load/store instructions and floating point instructions. Integer instructions take up more than 60% of the total instructions executed. Special function instructions are very rare. A similar behavior can be observed in the CDP implementations but with a more even distribution between floating point and load/stall instructions. Figure 9 shows the breakdown of the memory operations. Here we use Shared, Tex, and Const, to represent the number of instructions in shared memory, texture memory, and constant memory, respectively. Param is the number of parameters read instructions executed. Local is the number of local memory accesses, and Global is the number of global memory accesses. The applications' memory types vary from one another. We can see that for *Gasal2* applications (*GG*, *GL*, *GKSW*, *GSG*, and their CDP implementations), local memory accesses are the most dominant. Over 95% of memory accesses are to the shared memory for *NW* and *PairHMM*, whereas the remaining applications access global and local memory more frequently.

E. Branch divergence

Fung et al. [42] observed that branch divergence is a major source of performance loss in multithreaded SIMD architectures. Warp occupancy is defined as the number of active threads in an issued warp, which can be used to measure how efficiently the GPU throughput is utilized. We categorized the occupancy from W1 to W32, with W1 being the least utilized warp with only one active thread and W32 being the most utilized warp with 32 active threads (i.e. all threads in a warp are active).

Figure 10 shows the warp occupancy for both non-CDP and CDP implementation. The number of large warp occupancy (W29-32) is more than 60% of all issued warps in *NW*, *GG*, *GL*, *GKSW*, and *GSG*, showing good parallelism. Most non-CDP implementations share similar distribution of sub-optimal warp occupancy, with the exception of *CLUSTER* and *STAR*. *CLUSTER* is mostly dominated by unfulfilled warps (W1-4), taking up to over 50% of the time, while only half of the number of threads are active in *STAR*. Fully occupied warps take the majority of the time, with some applications reaching 100% occupancy. The CDP implementations of *NW*, *CLUSTER*, and *PairHMM* have better utilization of warps than their non-CDP counterparts, with *NW* having 100% warp occupancy. The CDP implementation of *STAR* is an outlier, with over 80% of its warps having less than 5 active threads. In general, better warp utilization in the CDP versions of the applications proves that CDP can effectively improve throughput by exploiting dynamic parallelism and avoiding stalls caused by branch divergence.

F. Thread count and block size

Usually, increasing the number of simultaneously executing threads will improve performance by concealing memory access latencies better. However, Bakhoda et al. [20] found that doing so may result in more competition for shared resources such as interconnect and memory. We experiment with cooperative thread arrays (CTA) [13], which defines how many threads can execute a kernel concurrently [5]. Note that in order to modify the CTAs per core for the experiment, we also need to modify shared memory, threads, and registers accordingly. Figure 11 shows the performance speedups for CDP and non-CDP when the CTA sizes (and according resources) change to 25%, 50%, 150%, and 200% of the baseline. As shown in Figure 11, most of the applications exhibit no difference between the different CTA counts, with CDP and Non-CDP sharing similar behavior. The main reason is that the resource provided in each configuration is sufficient that it will not impact the performance very much. We observe performance increases for the *PairHMM-CDP*, *NvB*, and *NvB-CDP* as more CTAs per core are used. Except for these applications, we observe no significant changes in varying the number of CTAs for most of the applications.

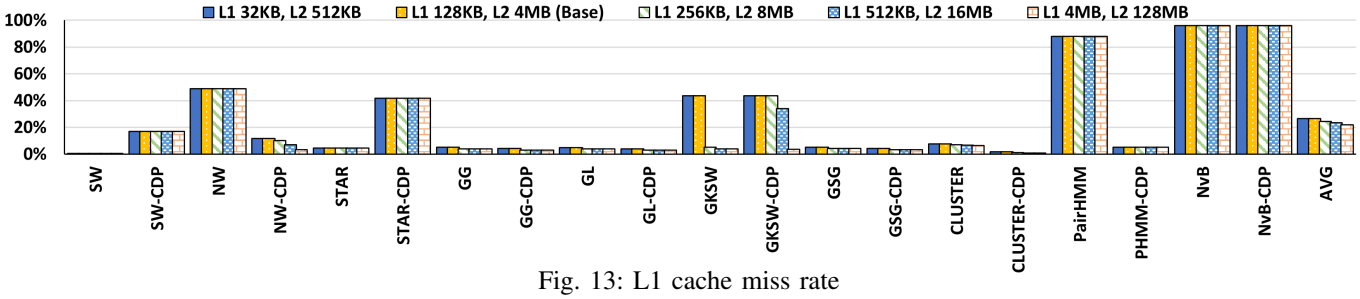


Fig. 13: L1 cache miss rate

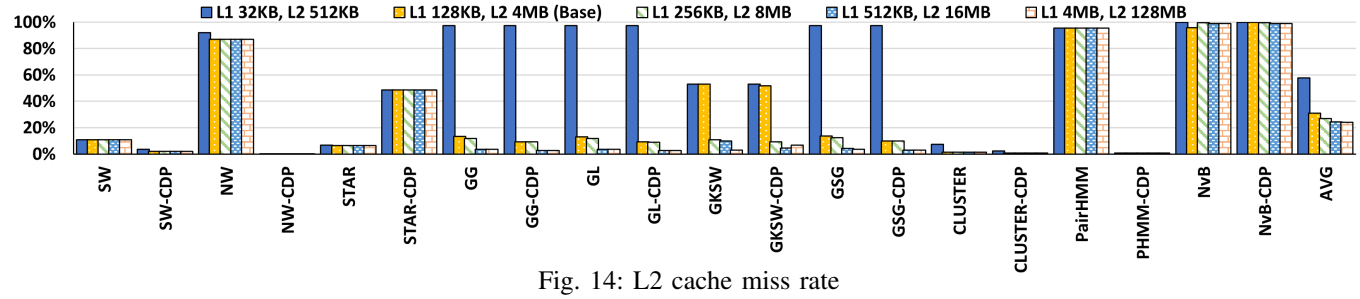


Fig. 14: L2 cache miss rate

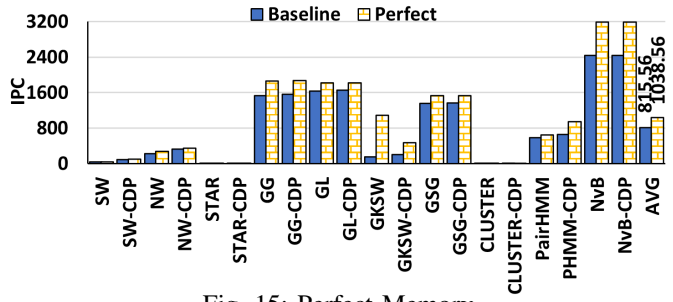


Fig. 15: Perfect Memory

G. Cache impacts

In this section, we explore the effects of varying cache sizes. The baseline configuration for cache size is 128KB for L1 and 4MB for L2. We evaluate more cache configurations including, no L1 +128KB L2, 32KB L1 + 512KB L2, 256KB L1 + 8MB L2, 512KB L1 + 16MB L2, and 4MB L1 + 128MB L2.

Figure 12 shows the effects on performance when we change cache sizes. Speedups are calculated by normalizing the IPC of each cache configuration to the baseline configuration. Performance degrades when the cache size is very small (e.g., 0KB L1 and 128KB L2). When we increase the cache sizes, several benchmarks can improve performance within the range of 10%, and others receive no improvement. *GKSW* and its CDP implementation receive the most improvement with increased cache size. Its Non-CDP implementation can improve by as much as 7 times while its CDP implementation achieves 2.7 times improvement.

Figure 13 and Figure 14 show the L1 and L2 cache miss rates with varied cache sizes. For L1 cache misses, as shown in Figure 13, we observe that most applications' miss rates do not change with increased cache sizes. The average L1 miss rate across all the applications is about 30%, with *SW* and most *GASAL2* applications showing very low miss rates. *NW-CDP, CLUSTER, CLUSTER-CDP*, and miss rates of the *GASAL2* applications slightly reduced when cache size increased. *PairHMM*, *NvB*, and *NvB-CDP* have very high miss rates, regardless of the cache size. We observe that *PairHMM* and *NvB* are not sensitive to L1 sizes. However, Figure 12 also shows a performance decrease when the L1 cache is not used, indicating that the L1 cache is still needed. L2 cache miss rates are similar to that of L1 cache, as shown in Figure 14. *NW*, *PairHMM*, *NvB*, and *NvB-CDP* experience very high L2 miss rates even with large L2 cache sizes. We can also observe that most applications exhibit a higher miss rate with smaller cache sizes, with *GASAL2* reaching up to 95% L2 miss rate. However, the cache miss rates do not change significantly with capacity except *GKSW*. One reason is we use the same cache sizes in the baseline as in RTX 3070, which is big enough for some applications. Another reason why some applications are not sensitive to cache capacity is that the CPU needs to do the memory transfer to GPU memory between two kernel invocations, for example, execute cudaMemcpy after each kernel. Consequently, cache locality will get lost on each kernel invocation [85]. As a result, memory read requests will reload the same data repeatedly and this will compromise the effectiveness of caches [9].

We also experimented with a perfect memory system that has zero memory access latency using the Accel-Sim simulator. Figure 15 shows the performance results. We can observe different reactions to this ideal memory setting. Some applications receive no performance improvement, such as *STAR* and *CLUSTER*. Some applications receive about 25% improvements, such as *GG* and *GL*. *GKSW* can improve performance by as much as 5x. On average, the applications can improve performance by 27% with the perfect memory system.

When comparing the results between CDP and non-CDP implementations, we can observe that for L1 cache misses, *STAR* and *SW*'s CDP implementation has a higher miss rate, and *PairHMM*'s CDP implementation has a lower miss rate. For L2 cache misses, *STAR*'s CDP implementation is

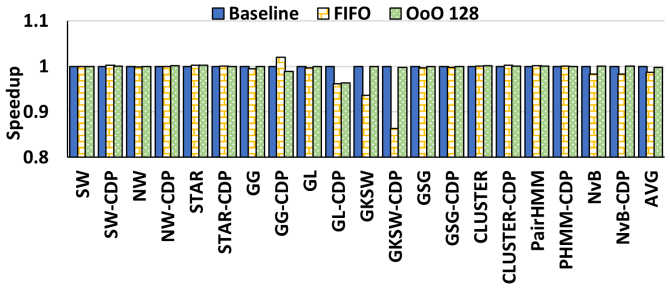


Fig. 16: Impact of DRAM memory controller optimization

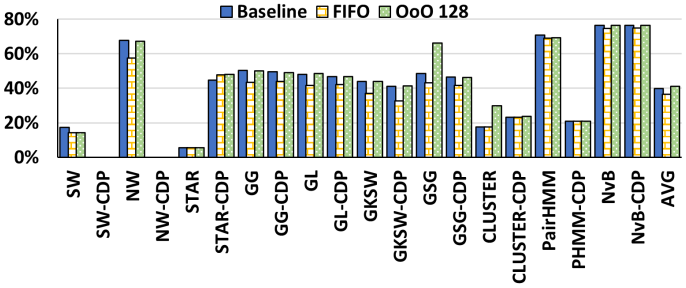


Fig. 17: DRAM efficiency

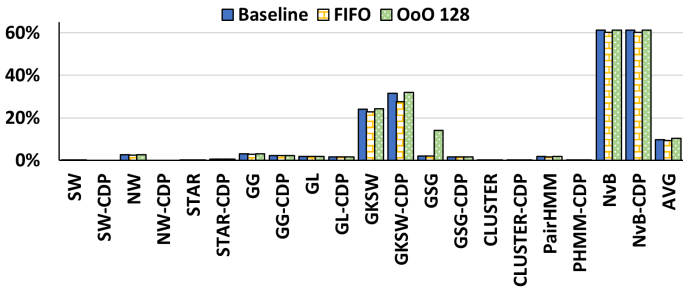


Fig. 18: DRAM utilization

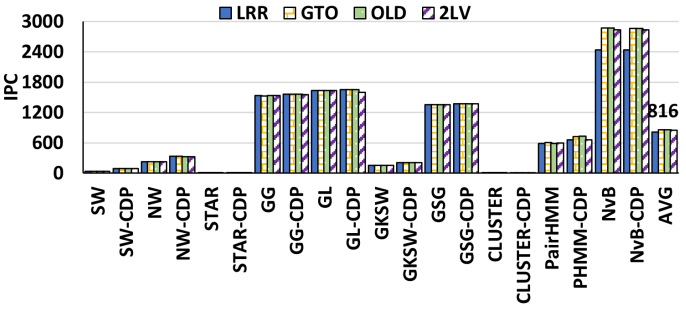


Fig. 19: Scheduler performance

also higher than the non-CDP implementation. *SW*, *NW*, and *PairHMM*'s CDP implementation have lower miss rates than its non-CDP implementations. The remaining applications show little difference between CDP and non-CDP implementations. We attribute the higher L2 miss rates in CDP implementations to interrupted spacial locality because CDP changed loop execution order and cannot access contiguous addresses in different loop iterations like in non-CDP.

H. DRAM utilization

We investigated the impact that different memory controller setups have on the application performance, with Figure 16 showing the results. Our baseline configuration uses an FR-FCFS (First-Row First-Come-First-Serve) memory controller. Requests to an open row in any of the DRAM banks are given higher priority. All requests in the queue are scheduled by the scheduler to open rows first. If no such request exists, a new row will be opened for the oldest request. We compare with a simple FIFO (First In First Out) memory controller and Out-of-Order FR-FCFS controller with a 128-entry buffer (OoO 128). There are no significant changes for both CDP or non-CDP implementations. However, when using FIFO, up to 15% slowdown can be observed for *GL*, *GKSW*, and *GKSW-CDP*.

We also measured DRAM efficiency and DRAM utilization. DRAM efficiency measures the time spent in sending data across the DRAM pins over the time of the memory controller being serviced or having pending memory requests. We observe an average efficiency of 40% across all applications as shown in Figure 17. *NW, PairHMM, NvB*, and *NvB-CDP* have a higher efficiency ranging from 60%-80%. The baseline and the OoO 128 memory controller have almost the same results, whereas FIFO shows a slightly worse efficiency. DRAM utilization measures how much of the total kernel execution time is spent transferring data through the DRAM data pins. The result is shown in Figure 18. Most applications have low DRAM utilization, with the exception of *GKSW*, *GKSW-CDP*, *NvB*, and *NvB-CDP*, meaning that these applications are more memory intensive.

I. Scheduler sensitivity

Figure 19 shows the impact of different warp schedulers on performance. Accel-Sim [50] uses LRR or loose round robin algorithm as default. Greedy-then-oldest (GTO) is a greedy algorithm that schedules instructions from a single warp until it stalls. Oldest-first (OLD) is an algorithm that guarantees the oldest task to use the resource first. Two-level(2LV) scheduler employs two levels of scheduling and groups warps into active warps and pending warps separately. The active warps issue in the way of loose round robin until it encounters long latency instructions. We observe no big differences in performance among these schedulers, with *NvB* and *NvB-CDP* having a slight improvement with the different schedulers over the baseline LRR. GTO and OLD have better performance on the CDP implementation for *PairHMM*. This is because CDP enables more threads to execute in parallel, allowing more opportunities to be explored by the schedulers.

J. Interconnect latency and bandwidth

The default RTX 3070 in Accel-Sim [50] uses a local crossbar as the interconnect backbone. We use it as our baseline. We also evaluated other network topologies, such as fat-tree, butterfly, and mesh. We use destination tag routing for butterfly, dimension-order routing for mesh, and nearest common ancestor routing for the fattree. Figure 20 shows the performance results where all performance is normalized to the baseline local crossbar. For most applications, we observe a slight performance decrease in the other network topologies compared with the baseline. There is a drastic decrease in performance for the CDP variant of *SW* and *NW* using mesh

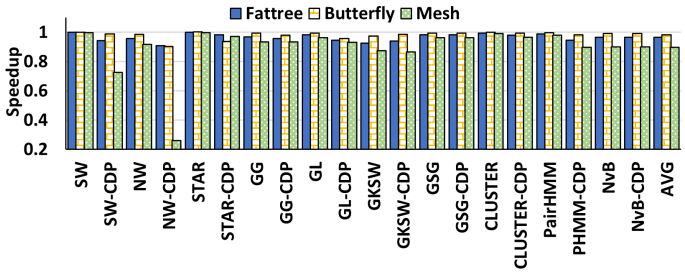


Fig. 20: Interconnect network topology

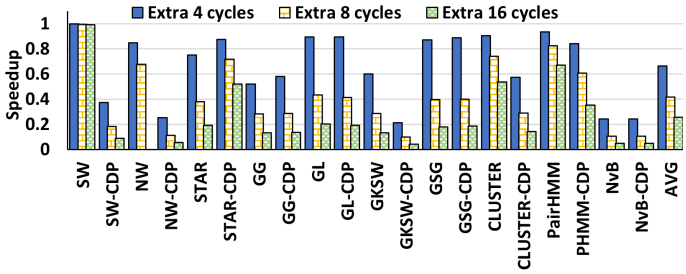


Fig. 21: Interconnect network latency

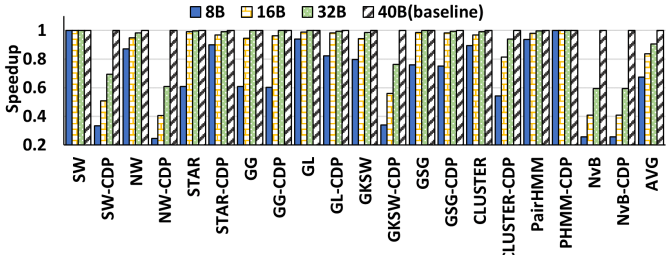


Fig. 22: Interconnect network bandwidth

topology. This is because they are more sensitive to the network performance and a mesh network performs worse due to its large number of hops.

We evaluated how network latency affects performance and the results are shown in Figure 21. We used mesh as the network topology. The baseline is an ideal network with zero router latency. We then add 4, 8, and 16 cycles of delay to the router pipeline. Except for *SW*, most applications suffer significant performance loss due to increased network latency. The average performance degradation is 36%, 60%, and 78% when we increase the latency by 4, 8, and 16 cycles. *SW-CDP*, *NW-CDP*, *GKSW-CDP*, *NvB*, and *NvB-CDP* experience the most significant performance loss. The reason why CDP implementations are more sensitive to network latency is that they have higher parallelism and longer network latency can lead to more performance degradation.

We also experimented with different interconnect bandwidths by varying the channel bandwidth for Mesh topology to 8B, 16B, 32B, and 40B. Figure 22 shows the results where all performance is normalized to 40B. When network bandwidth is reduced to 32B, the average performance degradation is around 10%. However, when we further reduce the bandwidth to 16B and 8B, there is a drastic decrease in performance across all benchmarks. For example, the average performance degraded by 34% when bandwidth is 8B. Similar to latency, we observe that several CDP implementations receive large performance loss. This again shows these types of applications are more sensitive to the network performance.

V. Related Work

There have been many studies on characterization and analysis for GPU applications [20], [26], [85]. Different metrics were used by Kerr et al. [49] to describe PTX kernels while also proposing a methodology for writing effective GPU programs. Burtscher et al. [24] studied the control flow irregularity and memory access irregularity on GPU programs. Some representative benchmarks used in GPU studies also include Parboil [75], Rodinia [28], ispass-2009 [7], and Lonestar [24]. Wang et al. [81] proposed a CDP implementation of graph-based substructure pattern mining. DiMarco et al. [34] evaluated the performance gains of clustering algorithms with dynamic parallelism. Wang et al. [83] also studied the dynamically formed parallelism applications and implemented them with CDP.

Several DNA and protein sequence analysis benchmarks such as BLAST [57] and HMMER [37] were evaluated by the BioPerf [19] benchmark suite while also providing precompiled Alpha binaries to assist simulations. Further characterization for these benchmarks was done by BioBench [17], showing to have a higher ILP (instruction level parallelism). GenomicsBench [76] then improved the coverage of prior benchmarks by including dynamic programming kernels and GPU-optimized dense neural networks. CUDAlign [70] proposed a GPU-accelerated version of the Smith-Waterman(SW) algorithm resulting in a peak of 20.375 GCUPS (Giga Cells Updates per Second). Other researchers also proposed hardware-accelerated GPUs or FPGAs for Smith-Waterman algorithms. [33], [54], [55], [77] Huang et al. [44] proposed the use of GPUs to accelerate the Pair-HMM algorithm resulting in a 487x improvement on a baseline C++ implementation on CPU and a 1.56x improvement to the best hardware implementation at that time. Ren et al. [68] then improved the performance by 5.47x by accelerating Pair-HMMs forward algorithm on GPUs with approaches such as intertask and intratask. Li et al. [52] followed up with a proposed improvement of GPU implementation by optimizing host-to-device communication, task parallelization, and memory management.

VI. Conclusion

In this paper, we present Genomics-GPU, a benchmark suite containing 10 GPU-accelerated genome analysis applications. We also provide an updated version of each application to support CDP. We use hardware GPU and a software simulator to evaluate these applications and provide a detailed characterization and analysis of the results. Genomics-GPU is open source and available on GitHub: https://github.com/Genomics-GPU/Genomics-GPU.

VII. Acknowledgement

We thank the anonymous reviewers for their valuable feedback and suggestions which helped improve this paper. This work was supported in part by NSF grants 2008911, 2046186, and 2051062.

REFERENCES

[1] Adaptive parallel computation with cuda dynamic parallelism — nvidia technical blog. https://developer.nvidia.com/blog/introduction-cuda-dynamic-parallelism/.

[2] Amd epyc™ 7763 — amd. https://www.amd.com/en/products/cpu/amd-epyc-7763.

[3] Bowtie 2: fast and sensitive read alignment. https://bowtie-bio.sourceforge.net/bowtie2/index.shtml. (Accessed on 11/16/2022).

[4] Cuda dynamic parallelism api and principles — nvidia technical blog. https://developer.nvidia.com/blog/cuda-dynamic-parallelism-api-principles/.

[5] Cuda refresher: The cuda programming model — nvidia technical blog. https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model.

[6] Geforce rtx 3070 family — nvidia. https://www.nvidia.com/en-us/geforce/graphics-cards/30-series/rtx-3070-3070ti/.

[7] Github - gpgpu-sim/ispass2009-benchmarks: Benchmarks used in the gpgpu-sim ispass 2009 paper. https://github.com/gpgpu-sim/ispass2009-benchmarks.

[8] Lonestar6 - tacc user portal. https://portal.tacc.utexas.edu/user-guides/lonestar6. (Accessed on 11/22/2022).

[9] Memory and locality. https://www.cs.cornell.edu/courses/cs3110/2012sp/lectures/lec25-locality/lec25.html.

[10] Nvbio — nvidia developer. https://developer.nvidia.com/nvbio.

[11] Nvcc :: Cuda toolkit documentation. https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html.

[12] Profiler :: Cuda toolkit documentation. https://docs.nvidia.com/cuda/profiler-users-guide/index.html.

[13] Ptx isa :: Cuda toolkit documentation. https://docs.nvidia.com/cuda/parallel-thread-execution/.

[14] User guide :: Nsight systems documentation. https://docs.nvidia.com/nsight-systems/UserGuide/index.html.

[15] Nauman Ahmed, Jonathan Lévy, Shanshan Ren, Hamid Mushtaq, Koen Bertels, and Zaid Al-Ars. Gasal2: a gpu accelerated sequence alignment library for high-throughput ngs data. *BMC bioinformatics*, 20(1):1–20, 2019.

[16] Nauman Ahmed, Hamid Mushtaq, Koen Bertels, and Zaid Al-Ars. Gpu accelerated api for alignment of genomics sequencing data. In *2017 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 510–515. IEEE, 2017.

[17] Kursad Albayraktaroglu, Aamer Jaleel, Xue Wu, Manoj Franklin, Bruce Jacob, Chau-Wen Tseng, and Donald Yeung. Biobench: A benchmark suite of bioinformatics applications. In *IEEE International Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005.*, pages 2–9. IEEE, 2005.

[18] Mohammed Alser, Zülal Bingöl, Damla Senol Cali, Jeremie Kim, Saugata Ghose, Can Alkan, and Onur Mutlu. Accelerating genome analysis: A primer on an ongoing journey. *IEEE Micro*, 40(5):65–75, 2020.

[19] David A Bader, Yue Li, Tao Li, and Vipin Sachdeva. Bioperf: A benchmark suite to evaluate high-performance computer architecture on bioinformatics applications. In *IEEE International. 2005 Proceedings of the IEEE Workload Characterization Symposium, 2005.*, pages 163–173. IEEE, 2005.

[20] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *2009 IEEE international symposium on performance analysis of systems and software*, pages 163–174. IEEE, 2009.

[21] Subho Sankar Banerjee, Mohamed El-Hadedy, Jong Bin Lim, Zbigniew T Kalbarczyk, Deming Chen, Steven S Lumetta, and Ravishankar K Iyer. Asap: accelerated short-read alignment on programmable hardware. *IEEE Transactions on Computers*, 68(3):331–346, 2018.

[22] Linda M Boettger, Robert E Handsaker, Michael C Zody, and Steven A McCarroll. Structural haplotypes and recent evolution of the human 17q21. 31 region. *Nature genetics*, 44(8):881–885, 2012.

[23] Behnam Bozorgmehr, Pete Willemsen, Jeremy A Gibbs, Rob Stoll, Jae-Jin Kim, and Eric R Pardyjak. Utilizing dynamic parallelism in cuda to accelerate a 3d red-black successive over relaxation wind-field solver. *Environmental Modelling & Software*, 137:104958, 2021.

[24] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. A quantitative study of irregular programs on gpus. In *2012 IEEE International Symposium on Workload Characterization (IISWC)*, pages 141–151. IEEE, 2012.

[25] Tiago Carneiro Pessoa, Jan Gmys, Francisco Heron de Carvalho Júnior, Nouredine Melab, and Daniel Tuyttens. Gpu-accelerated backtracking using cuda dynamic parallelism. *Concurrency and Computation: Practice and Experience*, 30(9):e4374, 2018.

[26] Shuai Che, Bradford M Beckmann, Steven K Reinhardt, and Kevin Skadron. Pannotia: Understanding irregular gpgpu graph applications. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*, pages 185–195. IEEE, 2013.

[27] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*, pages 44–54. Ieee, 2009.

[28] Shuai Che, Jeremy W Sheaffer, Michael Boyer, Lukasz G Szafaryn, Liang Wang, and Kevin Skadron. A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads. In *IEEE International Symposium on Workload Characterization (IISWC'10)*, pages 1–11. IEEE, 2010.

[29] Xi Chen, Chen Wang, Shanjiang Tang, Ce Yu, and Quan Zou. Cmsa: a heterogeneous cpu/gpu computing system for multiple similar rna/dna sequence alignment. *BMC bioinformatics*, 18(1):1–10, 2017.

[30] George M Church and Walter Gilbert. Genomic sequencing. *Proceedings of the National Academy of Sciences*, 81(7):1991–1995, 1984.

[31] Jeff Daily. Parasail: Simd c library for global, semi-global, and local pairwise sequence alignments. *BMC bioinformatics*, 17(1):1–11, 2016.

[32] William James Dally and Brian Patrick Towles. *Principles and practices of interconnection networks*. Elsevier, 2004.

[33] Lorenzo Di Tucci, Kenneth O'Brien, Michaela Blott, and Marco D Santambrogio. Architectural optimizations for high performance and energy efficient smith-waterman implementation on fpgas using opencl. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 716–721. IEEE, 2017.

[34] Jeffrey DiMarco and Michela Taufer. Performance impact of dynamic parallelism on different clustering algorithms. In *Modeling and Simulation for Defense Systems and Applications VIII*, volume 8752, pages 97–104. SPIE, 2013.

[35] Ke Ding and Ying Tan. Attract-repulse fireworks algorithm and its cuda implementation using dynamic parallelism. *International Journal of Swarm Intelligence Research (IJSIR)*, 6(2):1–31, 2015.

[36] Jianqiang Dong, Fei Wang, and Bo Yuan. Accelerating birch for clustering large scale streaming data using cuda dynamic parallelism. In *International Conference on Intelligent Data Engineering and Automated Learning*, pages 409–416. Springer, 2013.

[37] Sean R. Eddy. Profile hidden markov models. *Bioinformatics (Oxford, England)*, 14(9):755–763, 1998.

[38] Robert C Edgar. Search and clustering orders of magnitude faster than blast. *Bioinformatics*, 26(19):2460–2461, 2010.

[39] Xia Fei, Zou Dan, Lu Lina, Man Xin, and Zhang Chunlei. Fpgasw: accelerating large-scale smith–waterman sequence alignment application with backtracking on fpga linear systolic array. *Interdisciplinary Sciences: Computational Life Sciences*, 10(1):176–188, 2018.

[40] Limin Fu, Beifang Niu, Zhengwei Zhu, Sitao Wu, and Weizhong Li. Cd-hit: accelerated for clustering the next-generation sequencing data. *Bioinformatics*, 28(23):3150–3152, 2012.

[41] Daichi Fujiki, Arun Subramaniyan, Tianjun Zhang, Yu Zeng, Reetuparna Das, David Blaauw, and Satish Narayanasamy. Genax: A genome sequencing accelerator. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 69–82. IEEE, 2018.

[42] Wilson WL Fung, Ivan Sham, George Yuan, and Tor M Aamodt. Dynamic warp formation and scheduling for efficient gpu control flow. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 407–420. IEEE, 2007.

[43] Rama R Gullapalli, Ketaki V Desai, Lucas Santana-Santos, Jeffrey A Kant, and Michael J Becich. Next generation sequencing in clinical medicine: Challenges and lessons for pathology and biomedical informatics. *Journal of pathology informatics*, 3(1):40, 2012.

[44] Sitao Huang, Gowthami Jayashri Manikandan, Anand Ramachandran, Kyle Rupnow, Wen-mei W Hwu, and Deming Chen. Hardware acceleration of the pair-hmm algorithm for dna variant calling. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 275–284, 2017.

[45] imadassir. Github - needleman_wunsch_gpu. https://github.com/imadassir/Needleman_Wunsch_GPU.

[46] Nuno Subtil Jacopo Pantaleoni. Nvbio: Nvbio. https://nvlabs.github.io/nvbio/.

[47] Stephen Jones. Introduction to dynamic parallelism. In *GPU Technology Conference Presentation S*, volume 338, page 2012, 2012.

[48] Zhen Ju, Huiling Zhang, Jintao Meng, Jingjing Zhang, Jianping Fan, Yi Pan, Weiguo Liu, Xuelei Li, and Yanjie Wei. ngia: A novel greedy incremental alignment based algorithm for gene sequence clustering. *Future Generation Computer Systems*, 2022.

[49] Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili. A characterization and analysis of ptx kernels. In *2009 IEEE international symposium on workload characterization (IISWC)*, pages 3–12. IEEE, 2009.

[50] Mahmoud Khairy, Zhesheng Shen, Tor M Aamodt, and Timothy G Rogers. Accel-sim: An extensible simulation framework for validated gpu modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 473–486. IEEE, 2020.

[51] Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L Salzberg. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome biology*, 10(3):1–10, 2009.

[52] Enliang Li, Subho S Banerjee, Sitao Huang, Ravishankar K Iyer, and Deming Chen. Improved gpu implementations of the pair-hmm forward algorithm for dna sequence alignment. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*, pages 299–306. IEEE, 2021.

[53] Heng Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34(18):3094–3100, 2018.

[54] Isaac TS Li, Warren Shum, and Kevin Truong. 160-fold acceleration of the smith-waterman algorithm using a field programmable gate array (fpga). *BMC bioinformatics*, 8(1):1–7, 2007.

[55] Yongchao Liu, Adrianto Wirawan, and Bertil Schmidt. Cudasw++ 3.0: accelerating smith-waterman protein database search by coupling cpu and gpu simd instructions. *BMC bioinformatics*, 14(1):1–10, 2013.

[56] mattlm0831. Github - mattlm0831/localsequencealignment. https://github.com/mattlm0831/LocalSequenceAlignment.

[57] Jo McEntyre and Jim Ostell. The ncbi handbook. *Bethesda (MD): National Center for Biotechnology Information (US)*, 2002.

[58] Aaron McKenna, Matthew Hanna, Eric Banks, Andrey Sivachenko, Kristian Cibulskis, Andrew Kernytsky, Kiran Garimella, David Altshuler, Stacey Gabriel, Mark Daly, et al. The genome analysis toolkit: a mapreduce framework for analyzing next-generation dna sequencing data. *Genome research*, 20(9):1297–1303, 2010.

[59] nahmedraja. Github - nahmedraja/gasal2. https://github.com/nahmedraja/GASAL2.

[60] Ramanathan Narayanan, Berkin Ozisikyilmaz, Joseph Zambreno, Gokhan Memik, and Alok Choudhary. Minebench: A benchmark suite for data mining workloads. In *2006 IEEE International Symposium on Workload Characterization*, pages 182–188. IEEE, 2006.

[61] NCBI. Srr493095 : Run browser : Sra archive : Ncbi. https://trace.ncbi.nlm.nih.gov/Traces/?view=run_browser&acc=SRR493095&display=metadata.

[62] Saul B Needleman and Christian D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970.

[63] NVIDIA. Nvidia-kepler-gk110-gk210-architecture-whitepaper.pdf. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf.

[64] The University of Texas at Austin. Texas advanced computing center (tacc). URL:http://www.tacc.utexas.edu.

[65] Lawrence Rabiner and Biinghwang Juang. An introduction to hidden markov models. *ieee assp magazine*, 3(1):4–16, 1986.

[66] René Rahn, Stefan Budach, Pascal Costanza, Marcel Ehrhardt, Jonny Hancox, and Knut Reinert. Generic accelerated sequence alignment in seqan using vectorization and multi-threading. *Bioinformatics*, 34(20):3437–3445, 2018.

[67] Shoba Ranganathan, Kenta Nakai, and Christian Schonbach. *Encyclopedia of Bioinformatics and Computational Biology: ABC of Bioinformatics*. Elsevier, 2018.

[68] Shanshan Ren, Koen Bertels, and Zaid Al-Ars. Efficient acceleration of the pair-hmms forward algorithm for gatk haplotypecaller on graphics processing units. *Evolutionary Bioinformatics*, 14:1176934318760543, 2018.

[69] Torbjørn Rognes, Tomáš Flouri, Ben Nichols, Christopher Quince, and Frédéric Mahé. Vsearch: a versatile open source tool for metagenomics. *PeerJ*, 4:e2584, 2016.

[70] Edans Flavius O Sandes and Alba Cristina MA de Melo. Cudalign: using gpu to accelerate the comparison of megabase genomic sequences. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 137–146, 2010.

[71] shanshanren33. Github - shanshanren33/pairhmm_gpu_acceleration. https://github.com/shanshanren33/PairHMM_GPU_Acceleration.

[72] ShixiangWan. Github - shixiangwan/cmsa2: Improved center star algorithm for multiple sequences alignment (dna/rna/protein) based on cuda. https://github.com/ShixiangWan/CMSA2.

[73] SIAT-HPCC. Github - siat-hpcc/gene-sequence-clustering: accurate and fast gene greedy clustering tool. https://github.com/SIAT-HPCC/gene-sequence-clustering.

[74] Temple F Smith and Michael S Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

[75] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 127:27, 2012.

[76] Arun Subramaniyan, Yufeng Gu, Timothy Dunn, Somnath Paul, Md Vasimuddin, Sanchit Misra, David Blaauw, Satish Narayanasamy, and Reetuparna Das. Genomicsbench: A benchmark suite for genomics. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 1–12. IEEE, 2021.

[77] Amaro Taylor-Weiner, François Aguet, Nicholas J Haradhvala, Sager Gosai, Shankara Anand, Jaegil Kim, Kristin Ardlie, Eliezer M Van Allen, and Gad Getz. Scaling computational genomics to millions of individuals with gpus. *Genome biology*, 20(1):1–5, 2019.

[78] UCSC.EDU. Index of /goldenpath/hg19/bigzips. http://hgdownload.cse.ucsc.edu/goldenpath/hg19/bigZips/.

[79] MCJ Van Lanschot, LJW Bosch, M De Wit, B Carvalho, and GA Meijer. Early detection: The impact of genomics. *Virchows Archiv*, 471(2):165–173, 2017.

[80] vlab.amrita.edu. Smith-waterman algorithm - local alignment of sequences (theory) : Bioinformatics virtual lab ii : Biotechnology and biomedical engineering : Amrita vishwa vidyapeetham virtual lab. https://vlab.amrita.edu/.

[81] Fei Wang, Jianqiang Dong, and Bo Yuan. Graph-based substructure pattern mining using cuda dynamic parallelism. In *International conference on intelligent data engineering and automated learning*, pages 342–349. Springer, 2013.

[82] Jin Wang, Norm Rubin, Albert Sidelnik, and Sudhakar Yalamanchili. Laperm: Locality aware scheduler for dynamic parallelism on gpus. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 583–595. IEEE, 2016.

[83] Jin Wang and Sudhakar Yalamanchili. Characterization and analysis of dynamic parallelism in unstructured gpu applications. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 51–60. IEEE, 2014.

[84] Zeyu Xia, Yingbo Cui, Ang Zhang, Tao Tang, Lin Peng, Chun Huang, Canqun Yang, and Xiangke Liao. A review of parallel implementations for the smith–waterman algorithm. *Interdisciplinary Sciences: Computational Life Sciences*, pages 1–14, 2021.

[85] Qiumin Xu, Hyeran Jeon, and Murali Annavaram. Graph processing on gpus: Where are the bottlenecks? In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 140–149. IEEE, 2014.

[86] Byung-Jun Yoon. Hidden markov models and their applications in biological sequence analysis. *Current genomics*, 10(6):402–415, 2009.

[87] Eleftheria Zeggini, Anna L Gloyn, Anne C Barton, and Louise V Wain. Translational genomics and precision medicine: Moving from the lab to the clinic. *Science*, 365(6460):1409–1413, 2019.

[88] Peter Zhang, Eric Holk, John Matty, Samantha Misurda, Marcin Zalewski, Jonathan Chu, Scott McMillan, and Andrew Lumsdaine. Dynamic parallelism for simple and efficient gpu graph algorithms. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, pages 1–4, 2015.

[89] Quan Zou, Qinghua Hu, Maozu Guo, and Guohua Wang. Halign: Fast multiple similar dna/rna sequence alignment based on the centre star strategy. *Bioinformatics*, 31(15):2475–2481, 2015.

[90] Quan Zou, Xiao Shan, and Yi Jiang. A novel center star multiple sequence alignment algorithm based on affine gap penalty and k-band. *Physics Procedia*, 33:322–327, 2012.