GenDP: A Framework of Dynamic Programming Acceleration for Genome Sequencing Analysis

Yufeng Gu
University of Michigan
Ann Arbor, MI, USA
yufenggu@umich.edu

Arun Subramaniyan
Illumina Inc.
San Diego, CA, USA
asubramaniyan1@illumina.com

Tim Dunn
University of Michigan
Ann Arbor, MI, USA
timdunn@umich.edu

Alireza Khadem
University of Michigan
Ann Arbor, MI, USA
arkhadem@umich.edu

Kuan-Yu Chen
University of Michigan
Ann Arbor, MI, USA
knyuchen@umich.edu

Somnath Paul
Intel Corporation
Hilsboro, OR, USA
somnath.paul@intel.com

Md Vasimuddin
Intel Corporation
Bangalore, KA, India
vasimuddin.md@intel.com

Sanchit Misra
Intel Corporation
Bangalore, KA, India
sanchit.misra@intel.com

David Blaauw
University of Michigan
Ann Arbor, MI, USA
blaauw@umich.edu

Satish Narayanasamy
University of Michigan
Ann Arbor, MI, USA
nsatish@umich.edu

Reetuparna Das
University of Michigan
Ann Arbor, MI, USA
reetudas@umich.edu

ABSTRACT
Genomics is playing an important role in transforming healthcare. Genetic data, however, is being produced at a rate that far outpaces Moore’s Law. Many efforts have been made to accelerate genomics kernels on modern commodity hardware such as CPUs and GPUs, as well as custom accelerators (ASICs) for specific genomics kernels. While ASICs provide higher performance and energy efficiency than general-purpose hardware, they incur a high hardware design cost. Moreover, in order to extract the best performance, ASICs tend to have significantly different architectures for different kernels. The divergence of ASIC designs makes it difficult to run commonly used modern sequencing analysis pipelines due to software integration and programming challenges.

With the observation that many genomics kernels are dominated by dynamic programming (DP) algorithms, this paper presents GenDP, a framework of dynamic programming acceleration including DPAx, a DP accelerator, and DPMap, a graph partitioning algorithm that maps DP objective functions to the accelerator. DPAx supports DP kernels with various dependency patterns, such as 1D and 2D DP tables and long-range dependencies in the graph structure. DPAx also supports different DP objective functions and precisions required for genomics applications. GenDP is evaluated on genomics kernels in both short-read and long-read analysis pipelines, achieving 157.8× throughput/mm² over GPU baselines and 132.0× throughput/mm² over CPU baselines.

CCS CONCEPTS
• Computer systems organization → Special purpose systems;
• Applied computing → Genomics.

KEYWORDS
Computer Architecture, Hardware accelerators, Reconfigurable architectures, Genomics, Bioinformatics

ACM Reference Format:

1 INTRODUCTION
Genome sequencing, a key component of precision health, is necessary for early detection of cancer [71], autism [40], infectious diseases (such as COVID-19 [2]) and genetic diseases [78]. Genomics is a wide space and there are diverse applications within it, such as whole genome sequencing [43] and pathogen detection [50]. With the innovation in genome sequencing technologies over the past decade, sequencing data is being produced cheaper and faster, increasing at a rate that far outpaces Moore’s Law. The cost to sequence a human genome has dropped from $100 million at the beginning of this century to less than $1000 nowadays [74]. The total
amount of sequencing data has been doubling approximately every seven months and projections indicate that 100 million genomes will be sequenced by 2030 [7].

This large volume of sequencing data poses significant computational challenges and requires novel computing solutions which can keep pace. Recent works explore architecture-aware optimization on commodity hardware such as leveraging SIMD hardware on CPUs [34, 73] and thread-level parallelism on GPUs [9, 10, 26, 28, 57, 61–63]. Custom accelerators, however, achieve much better performance and are more area and power efficient than CPUs and GPUs [14, 23–25, 32, 70, 77]. These accelerators gain orders of magnitude speedups over general-purpose hardware, but at a high cost of hardware design. Specific genomics “kernels” (algorithms) do not have a market large enough to justify a custom chip. This makes it difficult to design an accelerator for the particular implementation of a single kernel, since the state-of-the-art implementation may change significantly over the next few years. For example, Smith-Waterman (SW), the basic approximate string matching algorithm, was optimized from the original SW [66] to a banded SW [17], and further to an adaptive banded SW [44] and a wavefront version [48]. Therefore, the high cost for designing custom accelerators and frequent kernel developments motivate a generic domain-specific accelerator for genome sequencing analysis.

In the commonly used genome sequencing pipelines, dynamic programming (DP) algorithms are widely used, including read alignment and variant calling in reference-guided alignment, layout and polishing in de-novo assembly, as well as abundance estimation in metagenomics classification [68]. Matrix multiplications are the heart of machine learning applications, which motivates the design of Tensor Processing Units (TPU) [33]. Similarly, DP algorithms are adopted by many genomics kernels and account for large amounts of time in mainstream sequencing pipelines, which provides the opportunity for a dynamic programming accelerator which supports both existing and future DP kernels.

Dynamic programming simplifies a complicated problem by breaking it down into sub-problems which can be solved recursively. However, accelerating a general-purpose DP algorithm comes with several challenges. First, DP kernels in common sequencing pipelines have different dependency patterns, including both 1-Dimension and 2-Dimension DP tables. Some kernels have long-range dependencies in the graph structure, where cells in the DP table not only depend on the neighboring cells, but also depend on cells far away. Second, DP kernels have different objective functions which include multiple operators. For instance, approximate string matching, an algorithm applied in DNA, RNA, and protein sequence alignment, has three modes: local (Smith-Waterman), global (Needleman-Wunsch) and semi-global alignment (overlap), as well as three methods for scoring insertions and deletions: linear, affine, and convex [72]. Each mode or method above requires a unique objective function. Third, DP kernels have different precision requirements. It is challenging to support multiple precision arithmetic while neither losing efficiency for low-precision computation nor compromising accuracy for high-precision computation.

To address these challenges, we propose GenDP, a framework of dynamic programming acceleration for genome sequencing analysis, which supports multiple DP kernels. First, we present DPAX, a DP accelerator capable of solving multiple dependency patterns by providing flexible interconnections between processing elements (PEs) in the systolic array. The systolic array helps exploit the wavefront parallelism in the DP table and provides better spatial locality for DP dataflow. DPAX decouples the control and compute instructions in the systolic array. Second, we present DPMap, a graph partitioning algorithm which maps the data-flow graph of the objective function to compute units in the DPAX accelerator. DPAX supports different objective functions and multiple precision arithmetic by programmable compute units.

We evaluate the GenDP framework on four DP kernels: Banded Smith-Waterman (BSW) [73], Chain [38, 39], Pairwise Hidden Markov Model (PairHMM) [58] and Partial Order Alignment (POA) [72]. We also demonstrate generality of the proposed framework by extending to other dynamic programming algorithms such as Dynamic Time Warping (DTW) which is commonly used for speech detection [12], as well as the Bellman-Ford (BF) algorithm for shortest path search in robotic motion planning tasks [51].

In summary, this paper makes the following contributions:

- We propose GenDP, a general-purpose acceleration framework for dynamic programming algorithms.
- We design DPAX, a DP accelerator with programmable compute units, specialized dataflow, and flexible PE interconnections. DPAX supports multiple dependency patterns, objective functions, and multi-precision arithmetic.
- We describe DPMap, a graph partitioning algorithm, to map the data-flow graph of DP objective functions to the compute units in DPAX.
- We synthesize the design of DPAX in a TSMC 28nm process. DPAX achieves 157.8× throughput per unit area and 15.1× throughput/Watt compared to GPU, and 132.0× throughput per unit area over CPU baselines.

2 BACKGROUND

2.1 Common Genomics Pipelines

Genome sequencing starts with raw data from the sequencer. The raw signals are interpreted to derive reads (short sequences of base pairs). This process is named basecalling. Next-generation sequencing (NGS) technologies produce short reads with ~ 100–150 base pairs (bp) [49], while third-generation technologies produce much longer reads (> 10,000 bp) [13]. After obtaining reads from raw data, there are two important analysis pipelines: reference-guided assembly and “de novo” assembly (without using a reference genome).

In reference-guided assembly, the sample genome is reconstructed by aligning reads to an existing reference genome. Read alignment can be abstracted to an approximate string matching problem, where dynamic programming algorithms [42] are used to estimate the pairwise similarity between the read and the reference sequence. After the alignment, small variants (mutations) still exist in aligned reads. A Hidden Markov Model (HMM) [58] or machine learning model [46] is then applied to detect such mutations, in a step known as variant calling.

If there is no reference genome available for alignment, the genome sequence needs to be constructed with reads from scratch, which is referred to as “de novo” assembly. Reads with overlapping regions can be chained to build an overlap graph and are then
We introduce four important and time-consuming DP kernels from genomics classification aligning input microbial reads to a reference pan-genome (consisting of different species) and then estimates the proportion of different microbes in the sample.

### 2.2 Dynamic Programming

Dynamic programming [11] simplifies a problem by breaking it down to subproblems. Following the Bellman equation [37] which describes the objective function, the subproblems can be solved recursively from the initial conditions. Longest common subsequence (LCS) [31] is a classic DP algorithm that involves looking for the LCS of two known sequences $X_m = \{x_0, x_1, \ldots, x_{m-1}\}$ and $Y_n = \{y_0, y_1, \ldots, y_{n-1}\}$. First, looking for the LCS between $X_m$ and $Y_n$ can be simplified by looking for LCSs between $X_{m-1}$ and $Y_{n-1}$, as well as $X_m$ and $Y_{n-1}$, as well as $X_{m-1}$ and $Y_n$. Each of these two subproblems can be further broken down into computing the results for LCSs between $X_{m-1}$ and $Y_{n-1}$. If we define $c[i, j]$ to be the length of an LCS between the sequence $X_i$ and $Y_j$, the objective function can be represented as shown in Equation 1:

$$
c[i, j] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
\min \left( c[i-1, j-1] + 1, \max(c[i, j-1], c[i-1, j]) \right) & \text{if } i, j > 0 \text{ and } x_i = y_j \end{cases}
$$

Second, a DP table can be constructed based on the sequence $X_m$ and $Y_n$ to memorize the subproblem results, as shown in Figure 1. $c[i, j]$ is calculated based on its upper, left, and diagonal neighbors $c[i-1, j]$, $c[i, j-1]$, and $c[i-1, j-1]$. The first row and first column of the DP table are filled with 0. Based on this initial condition, the cells in the whole DP table can be filled out. Finally, the largest value in the table is the length of longest common subsequence and the corresponding subsequence can be found by the trace back step, as shown in the orange block chain in Figure 1.

### 2.3 DP Kernels in Genomics Pipelines

We introduce four important and time-consuming DP kernels from commonly used genomics pipelines, as shown in Figure 2. Banded Smith-Waterman (BSW) is applied in read alignment, and variants of BSW are also used for RNA and protein alignment. Pairwise Hidden Markov Model (PairHMM) is used in post-alignment variant calling. Partial Order Alignment (POA) is applied in the polishing step of assembly. Chain is used in both alignment and assembly of long read sequencing, as well as metagenomics classification. These four kernels spend 31%, 70%, 47% and 75% of time in corresponding sequencing pipeline stages respectively [68]. The details of these algorithms are explained as follows:

**Banded Smith-Waterman (BSW)** is the banded version of the Smith-Waterman [66] algorithm, which estimates the pairwise similarity between the query and reference sequences. The similarity score for a given DNA sequence is typically computed with affine-gap [52] penalties, identifying short insertions and deletions in pairwise alignments. The objective function is shown in Figure 2a, which computes three matrices $H, E$ and $F$, corresponding to three edit types: match, insertion and deletion. $S$ is a similarity score between the base $X(i)$ and $Y(j)$. $H(i, j)$ refers to the similarity score for the substring $X(\theta, i)$ and $Y(\theta, j)$. The banded version of Smith-Waterman is applied with a maximum of w insertions or deletions, illustrated as the region between black cells in Figure 2a. BSW can be computed using 8-bit or 16-bit integer arithmetic depending on the sequence length [24].

**Pairwise Hidden Markov Model (PairHMM)** reads aligns to candidate haplotypes identified by the De-Bruijn graph traversal. The most likely haplotype supported by the reads is identified from the pairwise alignment, which is performed by a Hidden Markov Model (HMM). The likelihood score is computed by the formula shown in Figure 2b, where $f^M, f^I$ and $f^D$ represent match, insertion and deletion probabilities for aligning read substring $X(\theta, i)$ to haplotype substring $Y(\theta, j)$. The weights $\alpha$ are different transition and emission parameters of the HMM. $\rho$ is the prior probability of emitting bases $X(\theta)$ and $Y(\theta)$. The computation in PairHMM uses floating-point arithmetic [77].

**Partial Order Alignment (POA)**: In the assembly polishing step, multiple read sequences are used to construct a partial-order graph and the consensus sequence is then generated from the graph. Each unaligned sequence is aligned to the existing graph, as shown in Figure 2c. The nodes in the partial-order graph represent bases in the read sequence, and the weighted edges denote the times that the edges appear in different reads. Each cell not only depends on the upper and diagonal cells in the previous row, but also depends on earlier rows if there is an edge connecting that row with the current row in the graph. The objective function is similar to that used in BSW.

**Chain**: Given a set of seed pairs (anchors) shared between a pair of reads, Chain aims to group a set of collinear seed pairs into a single overlap region, as shown in Figure 2d(i). In the 1-Dimension DP table (Figure 2d(ii)), each anchor is compared with N previous anchors (default setting N=25) to determine the best parent. However, the dependency between neighboring anchors poses difficulties for parallelism. The reordered Chain algorithm [28] compares each anchor with N subsequent anchors and updates the scores each time for them (Figure 2d(iii)).

Table 1 summarizes the characteristics of four DP kernels above, including the dimension and size of DP tables, dependency patterns and arithmetic precision. BSW and PairHMM are used for short read pipelines, while POA and Chain are used in long read pipelines with larger DP tables. The first three kernels have 2-Dimension DP tables, whereas Chain has a 1-Dimension DP table. Each of these four kernels have different precision requirements, as shown in the last column.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Dimension</th>
<th>Size (k)</th>
<th>Dependency Pattern</th>
<th>Arithmetic Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Banded Smith-Waterman</td>
<td>2</td>
<td>8064</td>
<td>Linear</td>
<td>Floating-point</td>
</tr>
<tr>
<td>Pairwise Hidden Markov Model</td>
<td>2</td>
<td>8064</td>
<td>Linear</td>
<td>Floating-point</td>
</tr>
<tr>
<td>Partial Order Alignment</td>
<td>1</td>
<td>1024</td>
<td>Linear</td>
<td>Floating-point</td>
</tr>
<tr>
<td>Chain</td>
<td>1</td>
<td>1024</td>
<td>Linear</td>
<td>Floating-point</td>
</tr>
</tbody>
</table>
Figure 2: Black cells in BSW show the bands that limit the computing regions. Grey cells in all figures show previously computed elements and green ones on the wavefront are cells being computed in parallel. White cells show the untouched cells. Long arrows pointing to green cells show the direction that cells are being computed in different processing units in parallel. Short arrows pointing to blue circles show the dependency patterns. Cells in POA may also include long dependencies from rows other than the previous row shown by orange arrows.

Table 1: Characteristics of DP kernels

<table>
<thead>
<tr>
<th>Kernels</th>
<th>Dimension</th>
<th>Dependency</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSW</td>
<td>2D Table</td>
<td>~ 100 × 60</td>
<td>Last 2 Wavefronts</td>
</tr>
<tr>
<td>PairHMM</td>
<td>2D Table</td>
<td>~ 100 × 60</td>
<td>Last 2 Wavefronts</td>
</tr>
<tr>
<td>POA</td>
<td>Graph structure</td>
<td>~ 1000 × 500</td>
<td>Long-range dependency</td>
</tr>
<tr>
<td>Chain</td>
<td>1D Table</td>
<td>~ 20000</td>
<td>Last N (~ 25) Anchors</td>
</tr>
</tbody>
</table>

3 GENDP FRAMEWORK

Figure 3: GenDP Framework

3.1 Inter-Cell Dependency Pattern Supports

Figure 4: Overview of the DPax Architecture
of the DP kernel. The last PE in the PE array can either be connected to the first PE in the next PE array or to the data buffer directly. The 16 integer PE arrays can be concatenated and make up a large systolic array consisting of 64 PEs. Figures 5b and 5d show PE arrays of size 4 and 8 respectively.

The 16 integer PE arrays can be concatenated and make up a large systolic array consisting of 64 PEs. Figures 5b and 5d show PE arrays of size 4 and 8 respectively.

### 4 DPAX Architecture

#### 4.1 Processing Element Array

An overview of the DPAX architecture is shown in Figure 4 and discussed in the previous section. Figure 6 shows the architecture of the PE array and PE. The systolic array architecture simplifies control by allowing data-flow between neighboring PEs. However, the systolic data path alone cannot satisfy the requirements of various DP kernels. For example, POA has long-range dependencies and its dependency pattern is determined by the graph structure. The movement for such dependency requires branch instructions. Therefore, DPAX decouples the computation and control architecture to provide flexible data movements similar to decoupled access execute architectures [65]. Meanwhile, the parallelism and the massive reduction tree pattern observed in genomics DP kernels (Section 4.3) motivate the VLIW architecture.

The PE array consists of the input and output data buffers, control instruction buffer, decoder, first-in-first-out (FIFO) buffer and four PEs. PEs are connected as a systolic array and data can be passed from one PE to the next. The FIFO connects the last and first PEs. The first PE is connected to the input data buffer to receive the input sequences. The last PE stores the results to the output data buffer. The last PE in the PE array is also equipped with a dedicated port connected to the first PE in the next PE array to build a larger PE group. In Figure 6, blue solid and orange dotted lines show the data and control flow respectively.

#### 4.2 Processing Element

Each processing element (PE) is capable of running a control and a compute thread in parallel. Control and compute instructions are stored in two instruction buffers and decoded separately. Each PE contains a register file (RF) and a scratchpad memory (SPM) that store the short and long range dependencies respectively. Load and store ports are connected to the previous and next PEs. The first and last PEs are also connected to the FIFO and Input/Output Data Buffers.

Each PE is a 2-way VLIW compute unit array that can execute two independent compute instructions in parallel to exploit the instruction-level parallelism (ILP). Every PE contains two 32-bit compute units which execute the VLIW instructions. Each compute unit (CU) can either execute operations on 32-bit or four concurrent 8-bit groups of operands as a SIMD unit to make use of data-level parallelism (DLP). The SIMD unit improves the performance of low-precision kernels, e.g., BSW, where four DP tables are mapped to four SIMD lanes. The floating point PE array and PE architecture is similar to the integer one, but only supports 32-bit FP operands.

#### 4.3 Compute Unit Design Choice

We observe that genomics DP kernels have a common reduction tree pattern as shown in Figure 7 (a) and (b). Thus, we propose a
Figure 6: PE Array Architecture

reduction tree architecture for the compute unit (CU). The outputs of the first-level ALUs are used as inputs to the next-level ALUs. The CU also contains a multiplication module for the weight calculation in the Chain kernel. Since multiplication increases the length of the CU critical path, we design it as a separate unit from the ALU reduction tree.

Figure 7: Reduction Tree Pattern in (a) BSW and (b) PairHMM Kernels. (c) (d) (e) Compute Unit Design Choices. GenDP uses the two-level reduction tree.

Figure 7 (c) (d) (e) shows three possible choices for the ALU reduction tree. Compared to a 1-level reduction tree, the 2-level design requires fewer register file accesses and has better balances of critical path and area; the more levels the ALU reduction tree has, the fewer times CUs need to access the register file.

Table 2 compares ALU reduction trees of 1, 2, and 3 levels, which come with 1, 3, and 7 ALUs respectively. "RF Accesses" shows the number of accesses to each RF in a single cell of the DP table. "CU Utilization" is calculated as the percentage of cycles during which each ALU is utilized in the single-cell computation. The 3-level ALU reduction tree best reduces register file accesses, but lowers the CU utilization as well. It uses more than twice as many ALUs as the 2-level tree, but hardly reduces the number of RF accesses. Thus, we pick a 2-level reduction tree for the CU design.

4.4 Execution Model and GenDP ISA

We adopt the following execution model for GenDP. Instructions are preloaded to the accelerator before starting a DP kernel. Each PE array runs one thread of execution, controlling the data movement between data buffers and PEs, as well as the start of the execution for each PE. Upon receiving the start flag from the PE array, each PE runs two threads of execution: control and compute. The control thread manages data movement between the SPM, register file, and the systolic data-flow between neighboring PEs. This thread also controls the start of the compute thread. The compute thread executes a compute instruction by decoding instructions, loading the data from the register file, executing computations in the CU array, and finally writing results back to the register file.

Table 3: Control Instruction Set Architecture

The control ISA is shown in Table 3, which is applied to the control instructions in both the PE array and PEs.
movements in both PEs are considered into the critical path and the movement can be done in one cycle. The control instructions are generated manually in this work.

The DP objective function is represented as a data-flow graph (DFG).

4-input ALUs and multipliers, because a CU supports at most one access to the register file and are removed by DPMap in three edges within the subgraphs represent the data movements within the parent of node \( v \). An edge shows the dependency between\( v \) presents an operator, while an edge shows the dependency between\( v \). Operations in the compute instructions are generated by DPMap algorithm in Section 5.

The 2-way VLIW compute instructions are executed by two compute units, each of them containing 3 operations (for 3 ALUs of the top left ALU and 2 for the right one). Operations in the compute instructions are listed in Table 4. The compute instructions are generated manually in this work.

The 2-way VLIW compute instructions are executed by two compute units, each of them containing 3 operations (for 3 ALUs of the top left ALU and 2 for the right one). Operations in the compute instructions are listed in Table 4. The compute instructions are generated manually in this work.

Table 4: Compute instruction operation

<table>
<thead>
<tr>
<th>Operation</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>\textit{out} = \textit{in}[0] + \textit{in}[1]</td>
</tr>
<tr>
<td>Subtraction</td>
<td>\textit{out} = \textit{in}[0] - \textit{in}[1]</td>
</tr>
<tr>
<td>Multiplication</td>
<td>\textit{out} = \textit{in}[0] \times \textit{in}[1]</td>
</tr>
<tr>
<td>Carry</td>
<td>\textit{out} = \text{carry}(\textit{in}[0], \textit{in}[1])</td>
</tr>
<tr>
<td>Borrow</td>
<td>\textit{out} = \text{max}(\textit{in}[0], \textit{in}[1])</td>
</tr>
<tr>
<td>Maximum</td>
<td>\textit{out} = \text{min}(\textit{in}[0], \textit{in}[1])</td>
</tr>
<tr>
<td>Left-shift 16-bit</td>
<td>\textit{out} = \textit{in}[0] \ll 16</td>
</tr>
<tr>
<td>Right-shift 16-bit</td>
<td>\textit{out} = \textit{in}[0] \gg 16</td>
</tr>
<tr>
<td>Copy</td>
<td>\textit{out} = \textit{in}[0]</td>
</tr>
<tr>
<td>Match Score</td>
<td>\textit{out} = \text{scoretable}(\textit{in}[0], \textit{in}[1])</td>
</tr>
<tr>
<td>Log2 LUT</td>
<td>\textit{out} = \text{log2}(\textit{in}[0]) \gg 1</td>
</tr>
<tr>
<td>Log_sum LUT</td>
<td>\textit{out} = \text{log_sum}(\textit{in}[0])</td>
</tr>
<tr>
<td>Comparison &gt;</td>
<td>\textit{out} = \text{in}[0] &gt; \text{in}[1] \text{?} \text{in}[2] : \text{in}[3]</td>
</tr>
<tr>
<td>Comparison ==</td>
<td>\textit{out} = \text{in}[0] == \text{in}[1] \text{?} \text{in}[2] : \text{in}[3]</td>
</tr>
<tr>
<td>No-op</td>
<td>Invalid</td>
</tr>
<tr>
<td>Halt</td>
<td>Stop Computation</td>
</tr>
</tbody>
</table>

5 DPMAP ALGORITHM

The DP objective function is represented as a data-flow graph (DFG). The DPMAP algorithm generates compute instructions by mapping the DFG to compute units in the PE. In the DFG, a node represents an operator, while an edge shows the dependency between operators. The DFG has \(|V|\) nodes \( V = \{v_0, \ldots, v_{|V|−1}\} \) and \(|E|\) edges \( E = \{e_0, \ldots, e_{|E|−1}\} \). In edge \( e_i = (v_m, v_n) \), the operator in node \( v_n \) takes the result of \( v_m \) as an operand. We define node \( v_m \) as the parent of node \( v_n \), and \( v_n \) as the child of \( v_m \). Figure 9(a) shows the DFG of the BSW kernel.

DPMAP breaks the entire graph into subgraphs that contain either one multiplication or three ALU nodes (Figure 7(d)). The edges within the subgraphs represent the data movements within the compute units (CU). The edges between subgraphs represent accesses to the register file and are removed by DPMAP in three steps. First, Partitioning extracts nodes that will be mapped to 4-input ALUs and multipliers, because a CU supports at most one such operation. Second, Seeding looks for nodes that could be mapped to the second level of the ALU reduction tree. Nodes with more than one parent or more than one child are selected as seeds. The seed and its parents are mapped to a CU together. After seeding, the remaining nodes have a single parent or a single child. Third, Refinement maps every two remaining nodes to the 2-level ALU tree in a CU. Figure 9 shows an example of the DPMAP algorithm. Four subfigures represent the original graph and the three steps in DPMAP separately. Dashed blocks represent final subgraphs.

Algorithm 1 Partitioning

1. for \( v_i \in V \) do // Traverse the DFG
2. if \( \text{opcode}[v_i] = \text{Multiplication} \) then
3. Remove input and output edges of node \( v_i \)
4. end if
5. if \( \text{opcode}[v_i] = \text{Comparison/MatchScore} \) then
6. Remove input edges of node \( v_i \)
7. if node \( v_i \) has more than one child then
8. for \( v_j \in \text{children of node } v_i \) do
9. if \( \text{opcode}[v_j] = \text{Subtraction} \) then
10. Remove output edge of node \( v_i \)
11. else
12. Replicate node \( v_i \)
13. end if
14. end if
15. end if
16. end if
17. end for

Partitioning: Algorithm 1 breaks both input and output edges connected to nodes of 4-input ALUs and Multipliers. All parent and child edges of the multiplication nodes are removed (lines 2-4). DPMAP also removes the parent edges of 4-input operations (line 6). For a 4-input node that has two children, we replicate it if the operations of its children are commutative (except Subtraction) in order to decrease register file accesses (lines 8-14). After partitioning, all nodes have at most two parents.

Algorithm 2 Seeding

1. for \( v_i \in V \) do // Traverse the DFG
2. if node \( v_i \) (seed) has two parent nodes then
3. Remove output edges of node \( v_i \)
4. for \( v_j \in \text{parents of node } v_i \) do
5. Remove input edges of node \( v_j \)
6. end for
7. end if
8. if node \( v_i \) (seed) has more than one child then
9. Remove output edges of node \( v_i \)
10. end if
11. end for

Seeding: In Algorithm 2, we look for nodes that are suitable for the second level of the ALU reduction tree and name them seeds. Nodes that have two parent nodes are located to fit the structure of the ALU reduction tree (line 2). The output edges of seeds are removed because the output of this operator will be stored to the register file (line 3). In addition, since input operands of the seed’s parents must be fetched from the register file, DPMAP also removes the input edges of the seed’s parent nodes (lines 4-7). Finally, we
remove the output edges of all nodes with more than one child as its outputs have to be stored in the register file (lines 9-11). At this point, all nodes have at most one parent or one child.

**Refinement:** Algorithm 3 traverses the DFG in reverse order. If the node has a grandparent (line 4), the edge connecting its parent and grandparent is removed to group every two nodes (line 5). In the end, all subgraphs are able to be mapped to compute units in the PE.

**Algorithm 3 Refinement**

```plaintext
1: for v_i \in \{v_j|v_j \ldots v_0\} do // Traverse in reverse order
2: for v_j \in \text{parents of node } v_i \text{ do}
3: if node v_j has a parent node then
4: Remove input edge of node v_j
5: end if
6: end for
7: end for
```

### 6 EVALUATION METHODOLOGY

We synthesize the DPax accelerator using the Synopsys Design Compiler in a TSMC 28nm process. We use a cycle-accurate simulator to measure the throughput of DPax accelerator on the 4 DP kernels introduced in Section 2.3. The BSW, PairHMM and POA simulations show same results as CPU baselines. The Chain simulation implements the reordered algorithm and its accuracy is measured. The accuracy is compared with CPU baseline in Table 6. We use Ramulator [36] to generate DRAM configurations and use DRAMPower [4] to measure the power by DRAM access traces. The baseline CPU and GPU configurations are shown in Table 5. All CPU baselines utilize SIMD optimizations with AVX512. The CPU die area is estimated around 600 mm² [6]. We evaluate the GPU baselines on the Google Cloud Platform. The benchmark configuration for each DP kernel is detailed as follows.

**Banded Smith-Waterman (BSW):** BSW is evaluated on two million seed extension pairs with four 8-bit SIMD lanes on the DPax accelerator. The dataset is obtained from the inputs to the Smith-Waterman function in BWA-MEM2 [73] using reads from the NA12878 human genome sample ERR194147, an Illumina genomics dataset consisting of short reads of 101 bp. We choose the 8-bit optimized SIMD implementation in BWA-MEM2 as the CPU baseline, and BSW implementation in GASAL2 [9] as the GPU baseline. We also compare GenDP with GenAx [24], an ASIC baseline.

**Pairwise Hidden Markov Model (PairHMM):** We evaluate read-haplotyping as input pairs obtained from the calcLike11hoodScore function in GATK Haplotype Caller, with BWA-MEM aligned reads for human chromosome 22 as inputs. The CPU baseline is the optimized SIMD implementation in GATK Haplotype Caller [58]. We choose the implementation in [16] as the GPU baseline. A pruning-based implementation [77] is used for both the ASIC baseline and GenDP. GenDP evaluates the scan phase in a pruning-based implementation which accounts for 97.7% of the workload. The other 2.3% of the workload is a re-computation step which is performed on the CPU. The measured performance results include time spent in re-computation on the CPU host.

**Partial Order Alignment (POA):** POA is evaluated by 6217 consensus tasks obtained when polishing the Flye-assembled Staphylococcus aureus genome with Minimap2-aligned ONT long reads [75]. The CPU baseline is the SIMD accelerated implementation in Racon [72] and the GPU baseline is in [5]. GenDP supports long-range dependencies of at most 128 cells away in each row of the DP table. A few ultra-long dependency (>128) cases (caused by a very long

---

**Table 5: Baseline CPU and GPU Configurations**

<table>
<thead>
<tr>
<th></th>
<th>Intel® Xeon® Platinum 8380</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base Frequency</td>
<td>2.3 GHz</td>
</tr>
<tr>
<td>Cores(Threads)</td>
<td>40(80)</td>
</tr>
<tr>
<td>Process</td>
<td>10nm</td>
</tr>
<tr>
<td>TDP</td>
<td>270W</td>
</tr>
<tr>
<td>Cache</td>
<td>L1 D&amp;I 40x48KB, 40x32KB</td>
</tr>
<tr>
<td>Memory</td>
<td>512GB DRAM</td>
</tr>
<tr>
<td>Die Area</td>
<td>600mm²</td>
</tr>
</tbody>
</table>

**GPU**

<table>
<thead>
<tr>
<th></th>
<th>Nvidia A100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boost Frequency</td>
<td>1.4 GHz</td>
</tr>
<tr>
<td>CUDA Cores</td>
<td>6912</td>
</tr>
<tr>
<td>Process</td>
<td>7nm</td>
</tr>
<tr>
<td>TDP</td>
<td>300W</td>
</tr>
<tr>
<td>Cache</td>
<td>L2 40MB</td>
</tr>
<tr>
<td>Memory</td>
<td>80GB DRAM, HBM2e</td>
</tr>
<tr>
<td>Die Area</td>
<td>826mm²</td>
</tr>
</tbody>
</table>
deletion in the last few input reads in a read group) account for
2.4% of the workload, and are performed on the host CPU.

**Chain**: We evaluate the Chain kernel using 10K reads from PacBio
SMRT sequencing data of the *C. elegans* worm [3, 28] when com-
puting overlaps with itself. We choose the SIMD optimized imple-
mentation in [35] as the CPU baseline and implementation in [28]
as the GPU baseline. The GPU baseline and GenDP both apply the
reordered Chain (Section 2.3) with N=64 in order to best utilize the
parallelism and avoid large overhead of branches, thus computing
3.72× more cells than the CPU baseline. We penalize the measured
GPU and GenDP throughput results by 3.72× to normalize them
with the original CPU implementation. Our profiling results show
that the reordered Chain has comparable accuracy with original
Minimap2 when mapping PBSIM2 [55] simulated long reads to
human genome reference T2T-CHM13 [54], as shown in Table 6.

### Table 6: Chain Accuracy Comparison

<table>
<thead>
<tr>
<th>Minimap2</th>
<th>Reordered Chain (N=64)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map failure or error</td>
<td>0.2476%</td>
</tr>
<tr>
<td>Phred quality score of low quality map Q &lt; 10</td>
<td>54.36</td>
</tr>
</tbody>
</table>

7 RESULTS

7.1 DPax Area and Power

Table 7 shows the breakdown of area and power for the DPax
ASIC under a TSMC 28nm process. DPax consumes 5.4mm² in
area. Within a PE, 30% of the area is taken by the register file, 22%
is taken by the compute unit array, and 16% is taken by the
two decoders. The other 32% of total area is consumed by SRAM,
including instruction buffers and SPM. Table 8 shows the power
breakdown of DPax and DRAM in 28nm. DRAM power is averaged
across the 4 kernels and DPax power is the peak power of the ASIC.

### Table 7: Breakdown of Area and Power of DPax ASIC

<table>
<thead>
<tr>
<th>Components</th>
<th>Area (mm²)</th>
<th>Power (W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Unit Array</td>
<td>0.012</td>
<td>0.007</td>
</tr>
<tr>
<td>Decoder</td>
<td>0.008</td>
<td>0.004</td>
</tr>
<tr>
<td>Register File</td>
<td>0.015</td>
<td>0.009</td>
</tr>
<tr>
<td>Integer PE</td>
<td>0.035</td>
<td>0.020</td>
</tr>
<tr>
<td>1×4 Integer PE Array</td>
<td>0.149</td>
<td>0.081</td>
</tr>
<tr>
<td>16×4 Integer PE Array</td>
<td>2.381</td>
<td>1.307</td>
</tr>
<tr>
<td>Floating Point (FP) PE</td>
<td>0.047</td>
<td>0.019</td>
</tr>
<tr>
<td>1×4 FP PE Array</td>
<td>0.196</td>
<td>0.080</td>
</tr>
<tr>
<td>Sub Total</td>
<td>2.577</td>
<td>1.387</td>
</tr>
</tbody>
</table>

### Memory

<table>
<thead>
<tr>
<th>Components</th>
<th>Area (mm²)</th>
<th>Power (W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Buffer (200KB)</td>
<td>0.424</td>
<td>0.273</td>
</tr>
<tr>
<td>Instruction Buffer (208KB)</td>
<td>1.222</td>
<td>1.385</td>
</tr>
<tr>
<td>Scratchpad (156KB)</td>
<td>0.351</td>
<td>0.217</td>
</tr>
<tr>
<td>FIFO (276KB)</td>
<td>0.819</td>
<td>0.306</td>
</tr>
<tr>
<td>Sub Total</td>
<td>2.845</td>
<td>2.182</td>
</tr>
<tr>
<td>Total</td>
<td>5.391</td>
<td>3.569</td>
</tr>
</tbody>
</table>

### Table 8: Breakdown of DPax Power

<table>
<thead>
<tr>
<th>Components</th>
<th>Static (W)</th>
<th>Dynamic (W)</th>
<th>Total (W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPax</td>
<td>1.456</td>
<td>2.113</td>
<td>3.569</td>
</tr>
<tr>
<td>DRAM</td>
<td>0.446</td>
<td>0.645</td>
<td>1.091</td>
</tr>
<tr>
<td>Total</td>
<td>1.902</td>
<td>2.758</td>
<td>4.660</td>
</tr>
</tbody>
</table>

7.2 GenDP Performance

We use throughput per unit area measured in Million Cell Up-
dates per Second/mm² (MCUPS/mm²) as a metric for performance.
The area and power of CPU, GenDP and custom accelerators are
scaled [67] to a 7nm process for fair comparison with GPU. GenDP
is expected to run at 2GHz. Figure 10(a) shows throughput/mm²
comparisons across four DP benchmarks. Overall, GenDP achieves
132.0× speedup over CPU and 157.8× over GPU. Figure 10(b) shows
the throughput/Watt comparison between GenDP and GPU. The
large speedup can be attributed to the GenDP ISA, the special-
dataflow and on-chip memory hierarchy tailored for dynamic
programming.

Both the BWA-MEM2 CPU baselines and GenDP benefit from
8-bit SIMD optimizations for the BSW kernel. With AVX512, BWA-
MEM2 has 64 SIMD lanes, and GenDP has 4 SIMD lanes. The
PairHMM baseline applies floating point, whereas GenDP applies
the pruned-based implementation using logarithm and fixed point
numbers to approximate the computation and reduce complexity.
The bottleneck of POA performance on GenDP is the memory ac-
cesses. First, it has the graph dependency pattern, which is more
complex than other kernels. The dependency information needs to
be loaded from the input data buffer to each PE. Second, downstream
trace-back functions in POA need the move directions on the DP
table for each cell, which requires 8-byte outputs to be written to
the output data buffer from each cell. Both the input of the depend-
cency information and the output of the move directions consume
extra data movement instructions that limit POA performance on
GenDP. In the Chain kernel, both performances of GPU and GenDP
are penalized by 3.72× for the extra cell computation.

7.3 Comparison with Accelerators

The GenDP framework’s goal is to build a versatile dynamic pro-
gramming accelerator to support a wide range of genomics appli-
cations. Thus it sacrifices performance for programmability and
supporting a broader set of kernels. A key research question is
how much performance is sacrificed for generality. Figure 10(c)
shows the performance of GenDP compared to available custom
genomics ASIC accelerators, GenAx [24] accelerator for BSW, and
pruning-based PairHMM accelerator [77]. We observe a geomean
of 2.8× slowdown. This can be attributed largely to area overheads,
custom datapaths for cell score computation, custom data-flow, and
custom precision. For example, 37.5% of the register and 40% of the
SRAM are only utilized by POA but idle in other kernels, because
POA is significantly more complex than the other three. A custom
data-flow could specify the data bus width between neighboring
PEs and propagate all the data in a single cycle, whereas GenDP
needs control instructions to move data between neighboring PEs
because of various data movement requirements. An accelerator
for one specific kernel can implement one appropriate precision to
save area. For instance, the pruning-based PairHMM ASIC utilizes
20-bit fixed-point data which satisfies the compute requirements,
but GenDP has no such custom precision choice.

In addition to custom genomics ASIC accelerators, we also com-
pare GenDP with other data-flow and spatial architectures. Soft-
Brain [53] is a stream data-flow accelerator, which utilizes a data-
flow graph for repeated and pipelined computation, as well as
7.4 ISA Analysis

GenDP has a more efficient ISA for DP algorithms than general-purpose processors. We compare the number of compute instructions required per cell update in GenDP ISA to riscv64 and x86-64 ISA. The riscv64 and x86-64 instruction counts are obtained using \texttt{riscv64-unknown-elf-g++} and \texttt{g++} compilers respectively. Among four kernels, the instruction counts on GenDP are reduced by 8.1× and 4.0× on average when compared with riscv64 and x86-64, shown in Figure 10(d). The efficiency of GenDP instructions is affected by compute unit utilization, as shown in Table 2.

Several advantages of GenDP ISA are shown as follows: First, GenDP applies the VLIW architecture, where one instruction contains opcodes for 6 ALUs in the compute unit (CU) array. The ALU reduction tree in the CU fits the compute characteristics of DP kernels well. GenDP has an average 48% VLIW utilization among 64, shown in Figure 10(d). The efficiency of GenDP instructions is impacted by compute unit utilization, as shown in Table 2.

7.5 Scalability

With 8-channel DDR4-2400 DRAM (153.2 GB/s peak bandwidth), GenDP could scale up to 64 DPAx tiles and achieve 6.17× raw performance speedup over the GPU baseline, shown in Table 12. The area of GenDP is scaled [67] to 7nm to make a fair comparison with the GPU baseline.

7.6 Generality and Limitation

In addition to these four DP kernels within the commonly used sequencing pipelines, the GenDP framework also supports other
DP algorithms in either genomics or broader fields. This section discusses the generality and limitation of GenDP.

7.6.1 Dependency range. DP algorithms could be categorized into near-range (e.g., neighboring dependency pattern), limited long-range (e.g., dependency distance within 128) and ultra long-range (e.g., dependency distance longer than 128). GenDP could efficiently support near-range and long-range dependencies by fine-grained spatial locality design, such as the systolic array and scratchpad memory in the PE. GenDP also supports ultra long-range dependencies but needs to access these data through DRAM because the on-chip buffer is not large enough. However, the ultra long-range dependencies are usually rare, for example, POA only has 2.4% workload with dependency distances longer than 128, which are performed on the host CPU in simulation.

7.6.2 Active region. GenDP requires to specify the active regions in the DP table before the computation starts. For example, GenDP supports the static band choice in the DP table but does not support adaptive or dynamic band choice. In these cases, GenDP could choose a larger tiled static region that covers the adaptive bands but will sacrifice some performance.

7.6.3 Objective function. GenDP ISA supports most computations in the commonly used genomics pipeline, including local, global and semi-global approximate string matching as well as linear, affine, and convex scoring modes mentioned in Section 1. It also supports DP algorithms in other fields such as speech detection and robot motion planning.

7.6.4 Multi-precision arithmetic. DP kernels utilize computations of different precisions. For example, BSW can be computed using 8-bit or 16-bit precision depending on the sequence length. Computations in POA and Chain are in 32-bit integer format and PairHMM requires both integer and floating-point computation. DPax has both integer and floating-point PEs. The integer PEs support 32-bit and 8-bit integer arithmetic, and also support 64-bit and 16-bit basic operations such as addition, subtraction, and multiplication by using two parallel compute units.

![Figure 11: GenDP Instruction and Performance on DTW and BF benchmarks](image)

7.6.5 Broader field. Dynamic Time Warping (DTW) measures the similarity between two temporal sequences, which could be utilized for nanopore signal basecalling [23] and speech detection [12]. DTW has near-range dependency pattern similar to Smith-Waterman. Bellman-Ford (BF), a shortest path search algorithm, is commonly used for robotic motion planning applications. BF has a graph-based dependency pattern where the long-range dependencies within a certain distance could be efficiently supported by GenDP and the ultra-long range dependencies need accesses through DRAM. GenDP supports both objective functions of DTW and BF. Their performance and instruction comparisons with GPU [1, 64] are shown in Figure 11.

8 RELATED WORK

Dynamic Programming Accelerators in Genomics: Many custom genomics accelerators have been proposed to boost the performance of DP kernels in genomics pipelines, which significantly improve the performance over commodity hardware. However, these accelerators only support a single genomics kernel and must be customized and combined to support different stages of genomics pipelines. This increases both design cost and complexity. For example, [18–20, 24, 25, 45, 47, 70] are customized for read alignment and SquiggleFilter [23] is optimized for basecalling. GenASM [14] converts the DP objective function into bit-wise operations such as AND, OR, and SHIFT. Although GenASM partially supports the affine gap penalty model [52], bit-wise operations inherently fail to implement all the complex objective functions needed in different stages of genomics pipelines. SeGraM [15] extends GenASM to sequence-to-graph mapping and supports seeding, but supports limited DP objective functions as well. Race Logic [47] utilizes the race conditions in the circuit to accelerate the edit distance function in the bioinformatics field such as DNA sequence alignment and protein string comparison. However, other DP kernels such as PairHMM and Chain are not edit distance problems and have more complicated objective functions and higher numeric precision requirements. It is challenging to map such kernels to the Race Logic accelerator. GenDP aims to fill this gap and proposes a generalized acceleration framework that can be applied to accelerate the various flavors of dynamic programming common in genomics pipelines.

Besides genomics applications, dynamic programming algorithms are also accelerated in other domains, such as shortest path in robot motion planning [51]. There has also been industry interest in DP acceleration. For example, NVIDIA recently announced the dynamic programming instructions (DPX) in the Hopper architecture [8], but the corresponding products and CUDA library have not been released yet.

Domain Specific Accelerators in Genomics: Several domain specific accelerators have been explored for databases [76], machine learning [21, 33] and graph processing [22, 30]. However, there has been little work on domain specific accelerators for genomics. Based on the insight that data manipulation operations are also common in genomics pipelines, Genesis [29] proposes a domain-specific acceleration framework customized for data manipulation operations in genome sequencing analysis. Genesis uses an extended SQL as a domain specific language and provides the relevant hardware libraries. The framework is evaluated on AWS cloud FPGA and achieves up to 19.3x speedup over the 16-thread CPU baseline. However, the hardware modules only cover database operations for data manipulation and users need to add custom modules for different genomics pipelines. In contrast, different from Genesis’s
target, GenDP focuses on dynamic programming acceleration in genomics pipelines.

**General Accelerators:** There are also several works that have identified common compute and memory access patterns across both regular and irregular workloads and proposed reconfigurable, spacial and dataflow architectures for these patterns [27, 53, 56, 59]. But these accelerators are mostly optimized for data-parallel or data-intensive applications and not suitable for dynamic programming kernels. Plasticine [59] is a spatially reconfigurable architecture for parallel patterns, which supports broad applications, but has lower functional unit utilization on data-dependent applications such as PageRank (3.9%) and BFS (3.1%) than data-parallel applications (~50%). SoftBrain [53] and TIA [56] are discussed in Section 7.3.

9 CONCLUSION

In order to support general-purpose acceleration for genomics kernels in the commonly used sequencing pipelines, this work presented GenDP, a programmable dynamic programming acceleration framework, including DPax, a systolic array-based DP accelerator, and DPMap, a graph partitioning algorithm to map DP kernels to the accelerator. DPax supports multiple dependency patterns through flexible PE interconnections and different DP objective functions using programmable compute units. GenDP is evaluated on four important genomics kernels, achieving 157.8X throughput/mm² and 5.1X throughput/Watt compared to GPU, and 132.0X throughput/mm² over CPU baselines, and is also extended to DP algorithms in broader fields.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their suggestions which helped improve this paper. This work was supported in part by the NSF under the CAREER-1652294 and NSF-1908601 awards, and the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

A ARTIFACT APPENDIX

A.1 Abstract

This document briefly describes how to reproduce the main performance results of this paper in Figure 10 (a) and (c). The instructions in this document include 1) how to download the datasets, 2) how to run CPU/GPU baselines, 3) how to run GenDP simulations. The source code and instructions are accessible from GitHub. The expected results are shown in Table 13, 14 and 15.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Banded Smith-Waterman (BSW), Chain, Pairwise Hidden Markov Model (PairHMM), Partial Order Alignment (POA).
- **Program:** C++ and Python
- **Compilation:** g++ 8.3.1 and Intel® oneAPI DPC++/C++ Compiler 2021.8.0
- **Data sets:** Illumina NA12878 human genome sample ERR194147 (BSW), PacBio SMRT sequencing data of the C.elegans worm (Chain), human chromosome 22 (PairHMM), Flye-assembled Staphylococcus aureus genome (POA).
- **Hardware:** Intel CPU with >= 16G memory and >= 40G storage, and NVIDIA GPU
- **Execution:** Bash script for compilation and execution

- **Metrics:** Throughput: cell updates per second
- **Output:** CPU/GPU runtime and GenDP throughput
- **Experiments:** CPU/GPU baselines and GenDP simulation for 4 benchmarks (BSW, Chain, PairHMM and POA)
- How much disk space required (approximately)?: 40G
- How much time is needed to prepare workflow (approximately)?: ~ 1 hour
- How much time is needed to complete experiments (approximately)?: ~ 24 hours
- Publicly available?: Yes.
- Archived (provide DOI)?: https://doi.org/10.5281/zenodo.7792246

A.3 Description

A.3.1 How to access. The artifact could be accessed from GitHub and Zenodo.

A.3.2 Hardware dependencies.

1. Intel CPU and NVIDIA GPU
2. 16G memory and 40G storage

A.3.3 Software dependencies.

1. Linux OS
2. gcc >= 8.3.1
3. cmake >= 3.16.0
4. OpenMP >= 201511
5. Intel® DPC++/C++ Compiler >= 2021.8.0
6. ZLIB >= 1.2.8
7. CUDA >= 10.0
8. Python >= 3.7.9
9. numactl >= 2.0.0

A.3.4 Data sets. The list below shows the details of datasets and the table shows the approximate simulation time and corresponding input size. BSW simulation is fast and the default setting is entire dataset.

- BSW: Illumina NA12878 human genome sample ERR194147 (1932254 short reads with length <= 128)
- Chain: PacBio SMRT sequencing data of the C.elegans worm (10000 long reads)
- PairHMM: Human chromosome 22 (1420266 short reads)
- POA: Flye-assembled Staphylococcus aureus genome (6216 consensuses, each including 10 ~ 100 long reads)

A.4 Installation

Download the code base from GitHub and install Intel DPC++/C++ Compiler (ICX).

A.5 Experiment workflow

Please follow the instructions on GitHub.

Step 1: Check System Requirements
Step 2: Download Repository and Data sets (~ 10 min)
Step 3: Run CPU Baselines (~ 10 min)
Step 4: Run GPU Baselines (~ 10 min)
Step 5: Run GenDP Simulation (~ 24 hours)

Table 16 shows the relationship between data sets size and simulation time. We recommend to run scripts for ~ 6 hours or ~ 24 hours.
Table 13: CPU Baselines

<table>
<thead>
<tr>
<th>CPU</th>
<th>Operating System</th>
<th>SIMD Flag</th>
<th>Threads</th>
<th>BSF</th>
<th>Chain</th>
<th>PairHMM</th>
<th>POA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel® Xeon® Platinum 8380</td>
<td>CentOS Linux 7 (CORE)</td>
<td>AVX512</td>
<td>80</td>
<td>0.0504</td>
<td>0.306</td>
<td>0.587</td>
<td>16.6</td>
</tr>
<tr>
<td>Intel® Xeon® Gold 6136</td>
<td>Ubuntu 20.04.5 LTS</td>
<td>AVX512</td>
<td>32</td>
<td>0.0984</td>
<td>0.473</td>
<td>0.792</td>
<td>34.3</td>
</tr>
<tr>
<td>Intel® Xeon® E5-2697 v3</td>
<td>CentOS Linux 7 (CORE)</td>
<td>AVX2</td>
<td>28</td>
<td>0.196</td>
<td>2.35</td>
<td>2.13</td>
<td>41.7</td>
</tr>
<tr>
<td>12th Gen Intel® Core ™ i5-12600</td>
<td>Ubuntu 22.04.2 LTS</td>
<td>AVX2</td>
<td>12</td>
<td>0.140</td>
<td>2.21</td>
<td>1.71</td>
<td>36.6</td>
</tr>
<tr>
<td>Intel® Core ™ i7-7700</td>
<td>Ubuntu 20.04.5 LTS</td>
<td>AVX2</td>
<td>8</td>
<td>0.29</td>
<td>4.79</td>
<td>4.51</td>
<td>98.5</td>
</tr>
</tbody>
</table>

Table 14: GPU Baselines

<table>
<thead>
<tr>
<th>GPU</th>
<th>Arch Code</th>
<th>CUDA Version</th>
<th>BSF</th>
<th>Chain</th>
<th>PairHMM</th>
<th>POA</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVIDIA A100</td>
<td>sm_80</td>
<td>11.2</td>
<td>0.012</td>
<td>0.155</td>
<td>0.597</td>
<td>2.53</td>
</tr>
<tr>
<td>NVIDIA RTX A6000</td>
<td>sm_86</td>
<td>12.0</td>
<td>0.012</td>
<td>0.339</td>
<td>0.572</td>
<td>3.70</td>
</tr>
<tr>
<td>NVIDIA TITAN Xp</td>
<td>sm_61</td>
<td>10.2</td>
<td>0.020</td>
<td>0.747</td>
<td>0.915</td>
<td>11.2</td>
</tr>
</tbody>
</table>

Table 15: GenDP Speedup over CPU and GPU Baselines

<table>
<thead>
<tr>
<th></th>
<th>BSW</th>
<th>Chain</th>
<th>PairHMM</th>
<th>POA</th>
</tr>
</thead>
<tbody>
<tr>
<td>GenDP Normalized</td>
<td>47,574</td>
<td>3,626</td>
<td>17,681</td>
<td>2,965</td>
</tr>
<tr>
<td>GenDP Speedup over CPU</td>
<td>365.1x</td>
<td>63.7x</td>
<td>185.3x</td>
<td>70.4x</td>
</tr>
<tr>
<td>GenDP Speedup over GPU</td>
<td>198.9x</td>
<td>281.4x</td>
<td>440.8x</td>
<td>25.1x</td>
</tr>
</tbody>
</table>

Table 16: Data Sets Size and Approximate Simulation Time

<table>
<thead>
<tr>
<th>Simulation time</th>
<th>BSW</th>
<th>Chain</th>
<th>PairHMM</th>
<th>POA</th>
</tr>
</thead>
<tbody>
<tr>
<td>~ 6 hours</td>
<td>1,932,254</td>
<td>100</td>
<td>100,000</td>
<td>100</td>
</tr>
<tr>
<td>~ 24 hours</td>
<td>1,932,254</td>
<td>1,000</td>
<td>500,000</td>
<td>200</td>
</tr>
<tr>
<td>~ 250 hours</td>
<td>1,932,254</td>
<td>10,000</td>
<td>1,420,254</td>
<td>6,216</td>
</tr>
</tbody>
</table>

A.6 Evaluation and expected results

- The CPU and GPU baselines are machine-dependent. Some reference results on different platforms are listed in Table 13 and Table 14.
- GenDP normalized throughputs are comparable to reported results. See Row 9 in Table 15. The CPU and GPU baselines shown in the table above are obtained from Xeon Platinum 8380 and NVIDIA A100 separately. The simulation results with entire datasets could reproduce the results but it may take ~ 250 hours and require ~ 2 TB storage space. We recommend to run scripts for ~ 6 hours or ~ 24 hours. The simulation results with limited input size could be different but comparable to the reported table above.

REFERENCES
