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× $throughput/mm^2$ over GPU baselines and 132.0× $throughput/mm^2$ over CPU baselines.

CCS CONCEPTS

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

KEYWORDS

Computer Architecture, Hardware accelerators, Reconfigurable architectures, Genomics, Bioinfomatics

ACM Reference Format:
Yufeng Gu, Arun Subramaniyan, Tim Dunn, Alireza Khadem, Kuan-Yu Chen, Somnath Paul, Md Vasimuddin, Sanchit Misra, David Blaauw, Satish Narayanasamy, and Reetuparna Das. 2023. GenDP: A Framework of Dynamic Programming Acceleration for Genome Sequencing Analysis. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23), June 17–21, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3579371.3589060

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

ISCA '23, June 17–21, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0095-8/23/06…$15.00
https://doi.org/10.1145/3579371.3589060

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 optimizations 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 $\sim 100-150$ *base pairs (bp)* [49], while third-generation technologies produce much longer reads (> 10,000 bps) [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 further extended into larger contiguous regions. Finally, assembly errors are corrected in a graph-based dynamic programming polishing step [72].

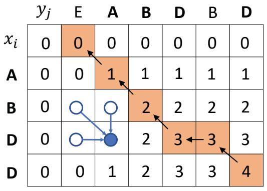
In addition to the two analysis pipelines above, metagenomics classification, another pipeline, is used for real-time pathogen detection [60] and microbial abundance estimation [41]. Metagenomics classification aligns input microbial reads to a reference pan-genome (consisting of different species) and 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 $X_m = \{x_0, x_1 ... x_{m-1}\}$ and $Y_n = \{y_0, y_1 ... y_{n-1}\}$. *First*, looking for the LCS between X_m and Y_n can be simplified by looking for LCSs between 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:

$$c[i,j] = \begin{cases} 0 & if\ i = 0\ or\ j = 0 \\ c[i-1, j-1] + 1 & if\ i, j > 0\ and\ x_{i-1} = y_{j-1} \\ max(c[i, j-1], c[i-1, j]) & if\ i, j > 0\ and\ x_{i-1} \neq y_{j-1} \end{cases} \tag{1}$$

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 ste, as shown in the orange block chain in Figure 1.



	y_j	E	**A**	**B**	**D**	B	**D**
x_i	0	0	0	0	0	0	0
A	0	0	1	1	1	1	1
B	0			2	2	2	2
D	0			2	3	3	3
D	0	0	1	2	3	3	4

Figure 1: DP Table for Longest Common Subsequence

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:

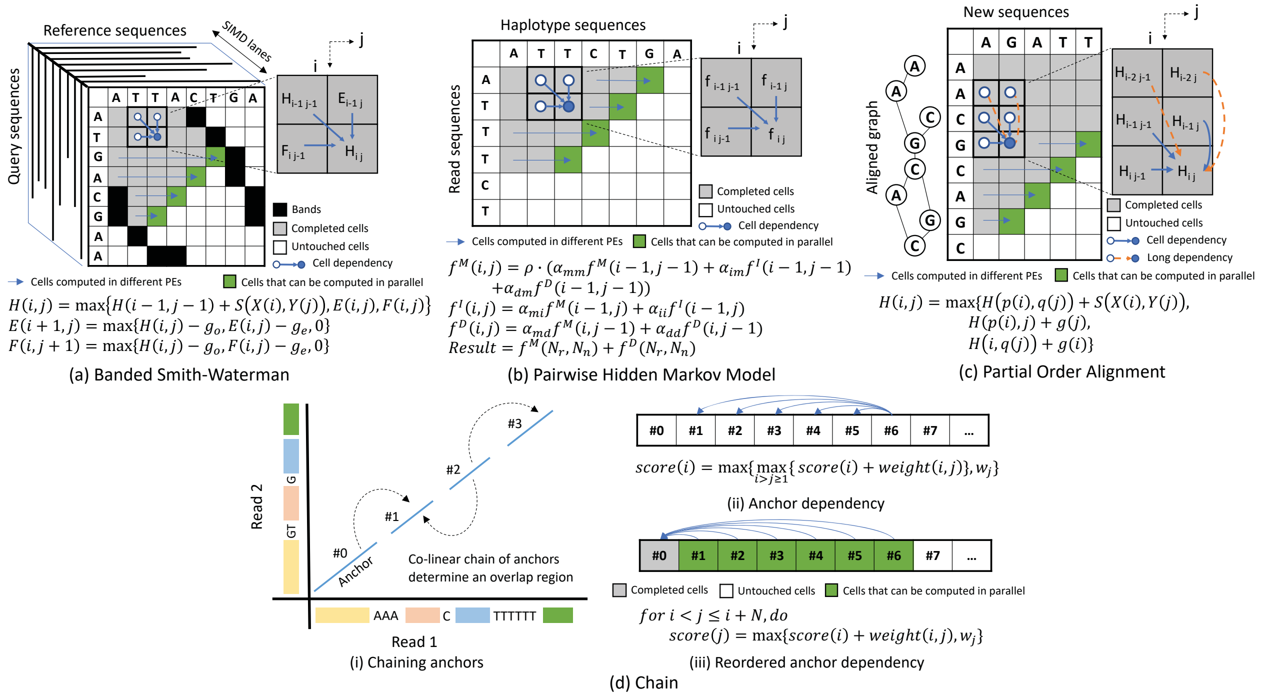
Banded Smith-Waterman (BSW) is the banded version of the Smith-Waterman [66] algorithm, which estimates the pairwise similarity between the query and reference sequences. The similarity score for a given DNA sequence is typically computed with affine-gap [52] penalties, identifying short insertions and 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(0,i)` and `Y(0,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) aligns reads 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(0,i)` to haplotype substring `Y(0,j)`. The weights α are different transition and emission parameters of the HMM. ρ is the prior probability of emitting bases `X(i)` and `Y(j)`. 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.



$H(i,j) = \max\{H(i-1,j-1) + S(X(i),Y(j)), E(i,j), F(i,j)\}$
$E(i+1,j) = \max\{H(i,j) - g_o, E(i,j) - g_e, 0\}$
$F(i,j+1) = \max\{H(i,j) - g_o, F(i,j) - g_e, 0\}$

(a) Banded Smith-Waterman

$f^M(i,j) = \rho \cdot (\alpha_{mm} f^M(i-1,j-1) + \alpha_{im} f^I(i-1,j-1) + \alpha_{dm} f^D(i-1,j-1))$
$f^I(i,j) = \alpha_{mi} f^M(i-1,j) + \alpha_{ii} f^I(i-1,j)$
$f^D(i,j) = \alpha_{md} f^M(i,j-1) + \alpha_{dd} f^D(i,j-1)$
$Result = f^M(N_r,N_n) + f^D(N_r,N_n)$

(b) Pairwise Hidden Markov Model

$H(i,j) = \max\{H(p(i),q(j)) + S(X(i),Y(j)),$
$H(p(i),j) + g(j),$
$H(i,q(j)) + g(i)\}$

(c) Partial Order Alignment

(i) Chaining anchors

$score(i) = \max\{\max_{i>j\geq 1}\{score(i) + weight(i,j)\}, w_j\}$

(ii) Anchor dependency

$for\ i < j \leq i+N, do$
$score(j) = \max\{score(i) + weight(i,j), w_j\}$

(iii) Reordered anchor dependency

(d) Chain

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

Kernels	Dimension	Dependency	Precision
BSW	2D Table $\sim 100 \times 60$	Last 2 Wavefronts	8-bit/16-bit Integer
PairHMM	2D Table $\sim 100 \times 60$	Last 2 Wavefronts	Floating-point
POA	2D Table $\sim 1000 \times 500$	Graph structure Long-range dependency	32-bit Integer
Chain	1D Table ~ 20000	Last $N (\sim 25)$ Anchors	32-bit Integer Floating-point

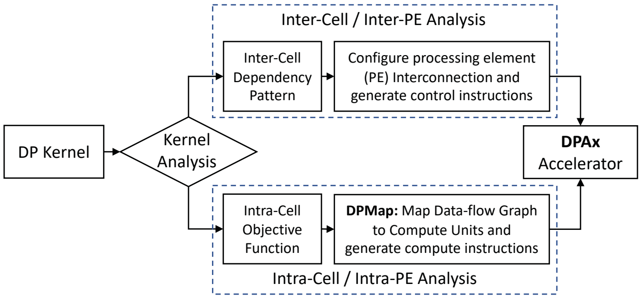

Inter-Cell / Inter-PE Analysis
Inter-Cell Dependency Pattern
Configure processing element (PE) Interconnection and generate control instructions
DP Kernel
Kernel Analysis
DPAx Accelerator
Intra-Cell Objective Function
DPMap: Map Data-flow Graph to Compute Units and generate compute instructions
Intra-Cell / Intra-PE Analysis

Figure 3: GenDP Framework

3 GENDP FRAMEWORK

Figure 3 demonstrates the structure of the GenDP framework. For each new DP kernel, we analyze the inter-cell dependency pattern and the intra-cell objective function, as shown in the top and bottom blocks respectively in Figure 3. *First*, the inter-cell dependency patterns are determined by the recursion rule in the DP kernel. Based on the dependency pattern, we configure the processing element (PE) interconnection and generate the control instructions. *Second*, the intra-cell data-flow graph for the objective function is mapped to compute units based on the DPMap algorithm. The compute instructions are generated based on the mapping results.

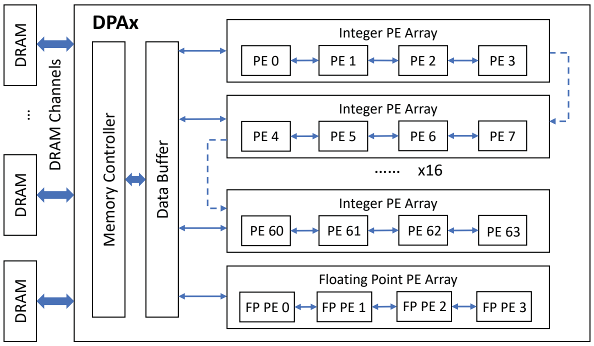
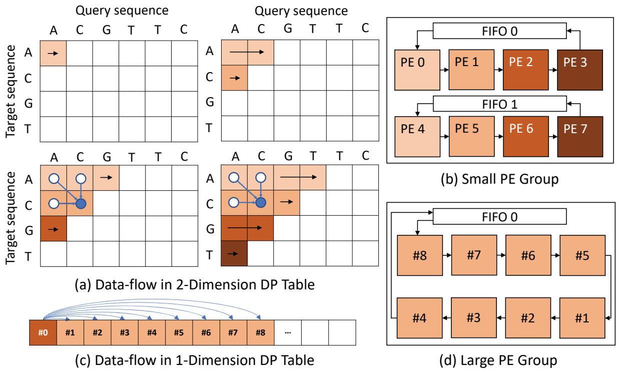


Figure 4: Overview of the DPAx Architecture

3.1 Inter-Cell Dependency Pattern Supports

An overview of the DPAx architecture is described in Figure 4, including 16 integer PE arrays and one floating point PE array. Each PE array contains four PEs that are connected as a 1-Dimension systolic array, in which each PE can receive data from the previous PE and pass data to the next one. The interconnections of the 16 integer PE arrays are configured based on the dependency pattern 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.



(a) Data-flow in 2-Dimension DP Table

(b) Small PE Group

(c) Data-flow in 1-Dimension DP Table

(d) Large PE Group

Figure 5: Example for DP Tables and PE Interconnections

For kernels with a *2D DP Table*, cells within a row are executed in the same PE. For example, rows with different colors in Figure 5 (a) are executed in PEs with corresponding colors in Figure 5 (b). Each element in the target sequence is stored in a PE statically, while elements in the query sequence are streamed through PEs in the same array. Each cell depends on results from three neighbors, the result from the left cell in the same row is stored inside the PE, while results from upper and diagonal cells in the previous row are transferred from the previous PE. First-in-First-out (FIFO) buffers connect the last and the first PEs in the array, from which the first PE (the first row in next 4-row group) acquires the dependent data from the last PE (the last row in the current 4-row group). PE array size determines how many rows can be executed in parallel.

For kernels with a *1D DP Table*, cells are mapped to a large PE array as shown in Figure 5 (d). When multiple small PE arrays are connected together to form a large PE array, only the FIFO in the first PE array is utilized. For example, in the first time step, cells #1-8 in Figure 5 (c) are mapped to PEs as shown in Figure 5 (d). Cells #1-8 all depend on cell #0. Therefore, the value of cell #0 is loaded from the FIFO to each PE sequentially. During the next time step, cells #2-8 move forward to their neighboring PEs. Cell #9 is sent from the input buffer to the first PE, while cell #1 is moved out from the last PE. Meanwhile, cell #1 is loaded from the FIFO to each PE because cells #2-9 all depend on cell #1.

Long-range dependencies in the graph structure are supported by scratchpad memories (SPM) inside each PE. In a 2D DP table, usually only the result of left cell is stored in the PE. However, if a cell depends on other cells in the same row but far away, all of the candidate dependent cells need to be stored in the PE. In kernels with long-range dependencies, the result for each cell is not only stored in registers for reuse by the next cell, but also stored in SPM for potential reuse by later cells.

3.2 Intra-Cell Objective Function Mapping

In order to support DP objective functions efficiently in the PE, we design a multi-level ALU array in the compute unit. This multi-level ALU array is consistent with common compute patterns in genomics DP kernels, and reduces the register access pressure as well. The computations in the objective function are represented by data-flow graphs. DPMap partitions the data-flow graph into multiple subgraphs which can be further mapped to the ALU arrays in compute units. The partition rules, constrained by the structure of the compute unit, will be illustrated in Section 5.

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

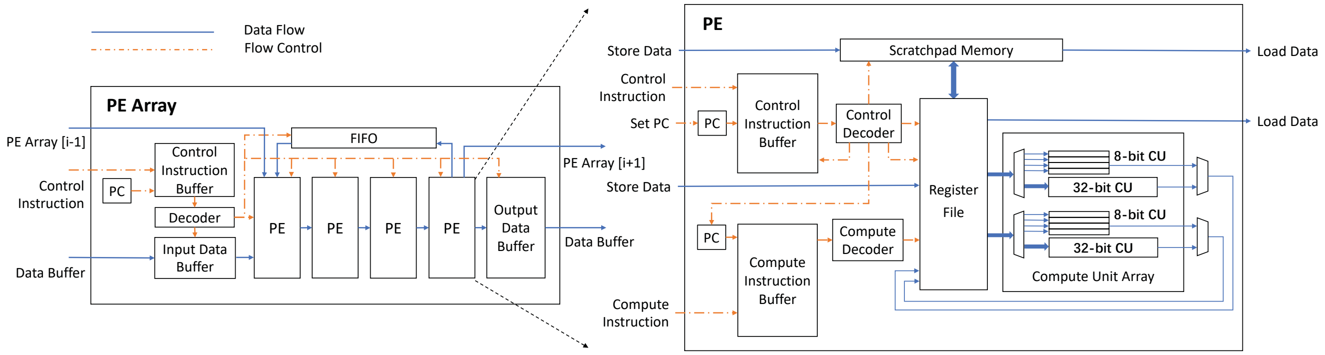
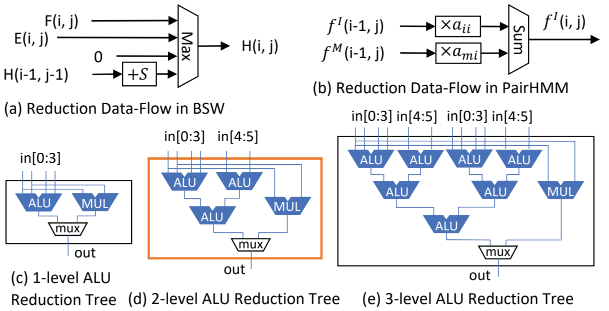


Figure 6: PE Array Architecture

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.



(a) Reduction Data-Flow in BSW

(b) Reduction Data-Flow in PairHMM

(c) 1-level ALU Reduction Tree

(d) 2-level ALU Reduction Tree

(e) 3-level 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.

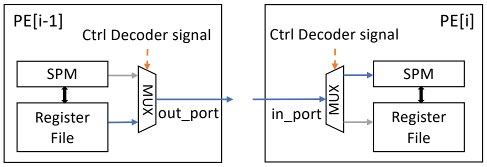
Table 2: ALU Reduction Trees with Different Levels

Kernel	Level of ALU Reduction Tree	RF Accesses	CU Utilization
BSW	1	20	100%
	2	11	60.6%
	3	10	28.6%
PairHMM	1	32	96.9%
	2	16	64.6%
	3	11	40.3%
POA	1	56	85.7%
	2	56	28.5%
	3	54	12.7%
Chain	1	24	95.8%
	2	20	38.3%
	3	20	16.4%

Table 3: Control Instruction Set Architecture

Type	Instruction	Assembly
Arithmetic	add	add rd rs1 rs2
	addi	addi rd rs1 #imm
Move	li	li [dest addr] #imm
	mv	mv [dest addr] [src addr]
Branch	beq/bne/bge/blt	branch rs1 rs2 offset
Other	set	set PC
	no-op	no operation
	halt	halt

The control ISA is shown in Table 3, which is applied to the control instructions in both the PE array and PEs. Arithmetic instructions manipulate the *address registers* within the decoders. mv instructions either load immediate values or move data between memory components. branch instruction enables the loop iteration. set instructions control the start of execution for a subsidiary component (e.g. PE arrays control PEs and PEs control CUs). Figure 8 shows an example for data movement, where two control instructions (one in each PE) are executed. PE[i-1] moves data from 0x00ff in the register file to its out port, while PE[i] moves data from its in port to 0x00ff in the scratchpad memory. The delay for 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.



(a) mv [out_port null] [reg 0x00ff] (b) mv [SPM 0x00ff] [in_port null]

Figure 8: Example for Data Movement

The 2-way VLIW compute instructions are executed by two compute units, each of them containing 3 operations (for 3 ALUs of the 2-level reduction tree in Figure 7 (d)) and 6 operands (4 for 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 by DPMap algorithm in Section 5.

Table 4: Compute instruction operation

Operation	Action
Addition	$out = in[0] + in[1]$
Subtraction	$out = in[0] - in[1]$
Multiplication	$out = in[0] \times in[1]$
Carry	$out = carry(in[0], in[1])$
Borrow	$out = in[0] < in[1]\ ?\ 1 : 0$
Maximum	$out = max(in[0], in[1])$
Minimum	$out = min(in[0], in[1])$
Left-shift 16-bit	$out = in[0] \ll 16$
Right-shift 16-bit	$out = in[0] \gg 16$
Copy	$out = in[0]$
Match Score	$out = scoretable(in[0], in[1])$
Log2 LUT	$out = log2(in[0]) \gg 1$
Log_sum LUT	$out = log\_sum(in[0])$
Comparison >	$out = in[0] > in[1]\ ?\ in[2] : in[3]$
Comparison ==	$out = in[0] == in[1]\ ?\ in[2] : in[3]$
No-op	Invalid
Halt	Stop Computation

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 $V = \{v_0 ... v_{|V|-1}\}$ and $|E|$ edges $E = \{e_0 ... e_{|E|-1}\}$. 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

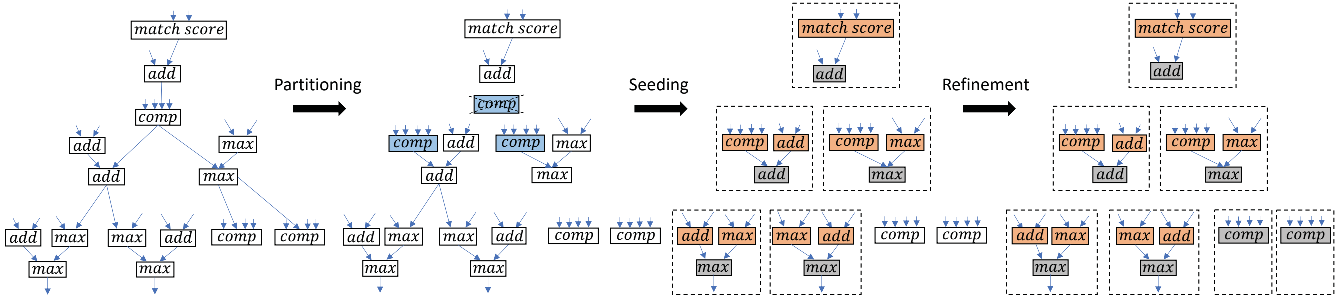
```
 1: for $v_i \in V$ do                                   // Traverse the DFG
 2:     if $opcode[v_i] = Multiplication$ then
 3:         Remove input and output edges of node $v_i$
 4:     end if
 5:     if $opcode[v_i] = Comparison/MatchScore$ then
 6:         Remove input edges of node $v_i$
 7: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



(a) Data-Flow Graph (b) Partitioning (c) Seeding (d) Refinement

Figure 9: Example for the DPMap algorithm in BSW. In (b) Partitioning, input edges of node `comp` are removed and a `comp` node is replicated to two `comp` nodes, because their children both have another parent and contain commutative operations. In (c) Seeding, nodes with two parents or multiple children are located, and are grouped into subgraphs with their parents. In (d) Refinement, the remaining two `comp` nodes are grouped into two subgraphs in the end.

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

```
1: for $v_i \in \{v_{|V|-1} ... v_0\}$ do        // Traverse in reverse order
2:     for $v_j \in$ parents of node $v_i$ 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 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 mm^2 [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.

Table 5: Baseline CPU and GPU Configurations

CPU	Intel® Xeon® Platinum 8380
Base Frequency	2.3 GHz
Cores(Threads)	40(80)
Process	10nm
TDP	270W
Cache	L1 D&I 40×48KB, 40×32KB L2 40×1MB Shared L3 60MB
Memory	512GB DRAM
Die Area	600mm^2
GPU	**Nvidia A100**
Boost Frequency	1.4 GHz
CUDA Cores	6912
Process	7nm
TDP	300W
Cache	L2 40MB
Memory	80GB DRAM, HBM2e
Die Area	826mm^2

Pairwise Hidden Markov Model (PairHMM): We evaluate read-haplotype pair inputs obtained from the `calcLikelihoodScore` 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 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 computing overlaps with itself. We choose the SIMD optimized implementation 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

	Minimap2	Reordered Chain (N=64)
Map failure or error	0.2476%	0.2479%
Phred quality score of low quality map $Q < 10$	54.36	54.14

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 5.4mm^2 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

	Components	Area (mm^2)	Power (W)
Logic	Compute Unit Array	0.012	0.007
	Decoder	0.008	0.004
	Register File	0.015	0.009
	Integer PE	0.035	0.020
	1×4 Integer PE Array	0.149	0.081
	16×4 Integer PE Array	2.381	1.307
	Floating Point (FP) PE	0.047	0.019
	1×4 FP PE Array	0.196	0.080
	Sub Total	2.577	1.387
Memory	Data Buffer (200KB)	0.424	0.273
	Instruction Buffer (208KB)	1.222	1.385
	Scratchpad (136KB)	0.351	0.217
	FIFO (276KB)	0.819	0.306
	Sub Total	2.845	2.182
Total		5.391	3.569

Table 8: Breakdown of DPAx Power

	Static (W)	Dynamic (W)	Total (W)
DPAx	1.456	2.113	3.569
DRAM	0.446	0.645	1.091
Total	1.902	2.758	4.660

7.2 GenDP Performance

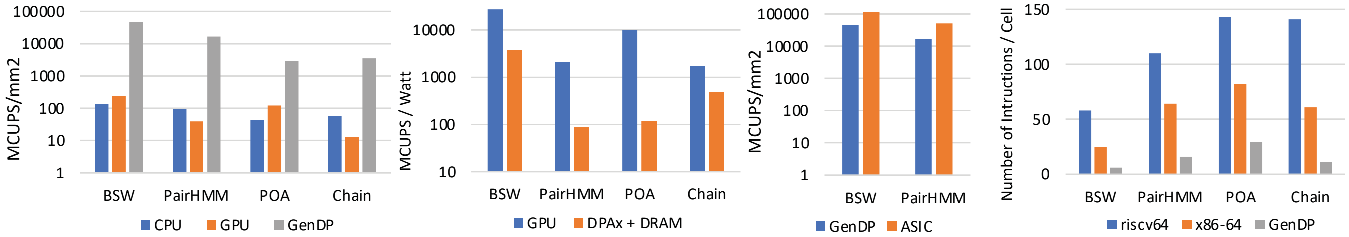
We use throughput per unit area measured in Million Cell Updates per Second/mm^2 (MCUPS/mm^2) as a metric for performance. The area and power of CPU, GenDP and custom accelerators are scaled [67] to a 7nm process for fair comparison with GPU. GenDP is expected to run at 2GHz. Figure 10(a) shows $throughput/mm^2$ 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 specialized 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 accesses. *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 dependency 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 programming accelerator to support a wide range of genomics applications. 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 compare GenDP with other data-flow and spatial architectures. SoftBrain [53] is a stream data-flow accelerator, which utilizes a dataflow graph for repeated and pipelined computation, as well as



(a) Throughput/mm^2 Comparison (b) Throughput/Watt Comparison (c) GenDP Perf. Comparison with ASIC (d) GenDP Instruction Trace Analysis

Figure 10: GenDP Power and Performance Comparison with Baselines

stream-based commands for efficient data movements. For the 4 DP kernels discussed in Section 6, GenDP is more area-efficient and has 2.12× area-normalized speedup over SoftBrain. Table 9 shows the padding overhead and SIMD utilization that limit its performance. In SoftBrain, we introduce padding to remove data hazards between pipeline stages in kernels that use 2D DP tables. SIMD utilization in DP kernels depends on both the number of SIMD lanes as well as the length of input sequences. In addition, kernels with a graph structure like POA gain little from the SIMD parallelism because the number of edges connected to each node varies in the graph. Intra-cell pipelining is not well suited for POA because the intra-cell iterative blocks are sequentially computed and dependent on each other, leading to intra-cell pipeline hazards. Inter-cell pipelining also provides limited benefits because there is a variable number of block iterations within each cell (determined by the number of edges connected to the current node).

Table 9: Benchmark implementation on SoftBrain

	Dimension	Pipe. Stages	Padding Overhead	SIMD Lanes(Util.)	GenDP Speedup
BSW	2D	3	9.9%	8(42.2%)	2.24x
PairHMM	2D	4	15.7%	2(95.9%)	1.13x
POA	Graph	1	0	1(100%)	10.74x
Chain	1D	10	0	2(73%)	0.75x

Triggered instruction architecture (TIA) [56] eliminates the program counter and branch instructions by predication, and exploits the locality in the spatial algorithms by using a PE array with mesh topology. However, the area and timing complexity of the trigger scheduler imposes a restriction on the number of triggered instructions (TI). For example, an implementation of edit distance scoring-based DP on top of TIA requires two PEs for 11 TIs [69]. A similar mapping strategy used on the DP kernels in Section 6 requires multiple PEs to compute the objective function in a single DP cell, limiting the benefits obtained from a spatial architecture like TIA. The number of TIs and PEs required by objective functions in each DP kernel is listed in Table 10.

Table 10: Triggered Instruction (TI) Required on TIA

Kernel	BSW	PairHMM	POA	Chain
Number of TIs required	30	45	90	47
Number of PEs required	5	8	16	8

In summary, GenDP balances well the specialization and generality trade-off for dynamic programming acceleration. Custom ASIC accelerators achieve better performance but are less programmable. SoftBrain, with reconfigurable networks, cannot efficiently support the DP kernels described in this work because of parallelism overhead and pipeline hazards. TIA reduces control instructions and exploits the spatial locality but is not well suited for compute-intensive DP kernels with complex scoring schemes.

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 `riscv64-unknown-elf-g++` and `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 4 kernels, shown in Table 11. The multiplication and conditional operations in Chain and POA could only be mapped to 4-input ALUs in DPAx, which limits the VLIW utilization. Meanwhile, During the execution in POA, 14.3% of the CUs are idle because of the complex dependency pattern of POA. *Second*, GenDP's ISA includes several custom operations such as Comparison, Max/Min and Lookup Table (LUT). In Chain, GenDP uses a special instruction for the LUT. In comparison, riscv64 and x86-64 need 14 and 7 instructions respectively for this LUT implementation. *Third*, the systolic array architecture provides spatial locality and saves many register-to-register operations. Meanwhile, DPAx has a memory hierarchy design with FIFO and scratchpad memory, which cache intermediate values for inter-cell communication and reduce the memory access to DRAM.

Table 11: VLIW Utilization

Kernel	BSW	PairHMM	Chain	POA
Utilization	60.6%	64.6%	38.3%	28.5%

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

Table 12: GenDP and GPU Raw Performance Comparison

	Area(mm^2)	Raw Perf.(GCUPS)	Speedup
NVIDIA A100 GPU	826.0	48.3	1
GenDP (64 tiles)	44.3	297.5	6.17x

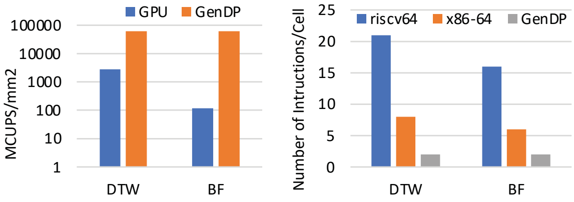
DP algorithms in either genomics or broader fields. This section discusses the generality and limitation of GenDP.

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.



(a) GenDP Perf. Comparison with GPU (b) Instruction Trace Analysis

Figure 11: GenDP Instruction and Performance on DTW and BF benchmarks

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 19.3× 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 157.8× $throughput/mm^2$ and 5.1× $throughput/Watt$ compared to GPU, and 132.0× $throughput/mm^2$ 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

CPU	Operating System	SIMD Flag	Threads	BSW	Chain	PairHMM	POA
Intel® Xeon® Platinum 8380	CentOS Linux 7 (CORE)	AVX512	80	0.0504	0.306	0.587	16.6
Intel® Xeon® Gold 6326	Ubuntu 20.04.5 LTS	AVX512	32	0.0984	0.473	0.792	34.3
Intel® Xeon® E5-2697 v3	CentOS Linux 7 (CORE)	AVX2	28	0.196	2.35	2.13	41.7
12th Gen Intel® Core™ i5-12600	Ubuntu 22.04.2 LTS	AVX2	12	0.140	2.21	1.71	36.6
Intel® Core™ i7-7700	Ubuntu 20.04.5 LTS	AVX2	8	0.29	4.79	4.51	98.5

Table 14: GPU Baselines

GPU	Arch Code	CUDA Version	BSW	Chain	PairHMM	POA
NVIDIA A100	sm_80	11.2	0.012	0.155	0.597	2.53
NVIDIA RTX A6000	sm_86	12.0	0.012	0.339	0.572	3.70
NVIDIA TITAN Xp	sm_61	10.2	0.020	0.747	0.915	11.2

Table 15: GenDP Speedup over CPU and GPU Baselines

	BSW	Chain	PairHMM	POA
Total Cell Updates	2,431,855,834	20,736,142,007	258,363,282,803	6,448,581,509
CPU Runtime (seconds)	0.0504	0.306	0.587	16.6
CPU GCUPS	44.91	19.61	32.88	14.51
CPU Normalized $MCUPS/mm^2$	130.29	56.89	95.41	42.11
GPU Runtime (seconds)	0.012	0.155	0.597	2.53
GPU GCUPS	192.92	10.40	32.35	95.13
GPU $MCUPS/mm^2$	239.16	12.89	40.11	117.94
ASIC Normalized $MCUPS/mm^2$	118,950	-	51,867	-
GenDP Normalized $MCUPS/mm^2$	47,574	3,626	17,681	2,965
GenDP Speedup over CPU	365.1x	63.7x	185.3x	70.4x
GenDP Speedup over GPU	198.9x	281.4x	440.8x	25.1x

Table 16: Data Sets Size and Approximate Simulation Time

Simulation time	BSW	Chain	PairHMM	POA
$\sim$ 6 hours	1,932,254	100	100,000	100
$\sim$ 24 hours	1,932,254	1,000	500,000	200
$\sim$ 250 hours	1,932,254	10,000	1,420,266	6,216

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 $\sim$ 250 hours and require $\sim$ 2 TB storage space. We recommend to run scripts for $\sim$ 6 hours or $\sim$ 24 hours. The simulation results with limited input size could be different but comparable to the reported table above.

REFERENCES

[1] Accelerating Bellman-Ford Single Source Shortest Path Algorithm on GPU using CUDA. https://github.com/sengorajkumar/gpu_graph_algorithms.

[2] Artic Network: real-time molecular epidemiology for outbreak response. https://artic.network/

[3] Caenorhabditis Elegans 40x Coverage Dataset, Pacific Biosciences. http://datasets.pacb.com.s3.amazonaws.com/2014/c_elegans/list.html.

[4] DRAMPower: Open-source DRAM Power and Energy Estimation Tool. https://github.com/tukl-msd/DRAMPower.

[5] A GPU-accelerated implementation of the Partial Order Alignment algorithm. https://github.com/clara-parabricks/GenomeWorks/blob/dev/cudapoa.

[6] Intel Ice Lake Xeon Platinum 8380 Review. https://www.tomshardware.com/news/intel-ice-lake-xeon-platinum-8380-review-10nm-debuts-for-the-data-center

[7] National Genomic Data Initiatives Review. https://www.ga4gh.org/news/ga4gh-publishes-review-of-national-genomic-data-initiatives/

[8] NVIDIA Hopper GPU Architecture Accelerates Dynamic Programming Up to 40x Using New DPX Instructions. https://blogs.nvidia.com/blog/2022/03/22/nvidia-hopper-accelerates-dynamic-programming-using-dpx-instructions/

[9] 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 (2019), 1–20. https://link.springer.com/article/10.1186/s12859-019-3086-9

[10] Richard Barnes. A Review of the Smith-Waterman GPU Landscape. *Electrical Engineering and Computer Sciences University of California at Berkeley. Retrieved from https://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-152.html* (2020).

[11] Richard Bellman. Dynamic programming. *Science* 153, 3731 (1966), 34–37. https://doi.org/10.1126/science.153.3731.34

[12] Donald J. Berndt and James Clifford. Using Dynamic Time Warping to Find Patterns in Time Series. In *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining* (Seattle, WA) *(AAAIWS'94)*. AAAI Press, 359–370. https://doi.org/10.5555/3000850.3000887

[13] Christoph Bleidorn. Third generation sequencing: technology and its potential impact on evolutionary biodiversity research. *Systematics and biodiversity* 14, 1 (2016), 1–8. https://www.tandfonline.com/doi/abs/10.1080/14772000.2015.1099575

[14] Damla Senol Cali, Gurpreet S. Kalsi, Zülal Bingöl, Can Firtina, Lavanya Subramanian, Jeremie S. Kim, Rachata Ausavarungnirun, Mohammed Alser, Juan Gomez-Luna, Amirali Boroumand, Anant Norion, Allison Scibisz, Sreenivas Subramoneyon, Can Alkan, Saugata Ghose, and Onur Mutlu. GenASM: A High-Performance, Low-Power Approximate String Matching Acceleration Framework for Genome Sequence Analysis. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 951–966. https://doi.org/10.1109/MICRO50266.2020.00081

[15] Damla Senol Cali, Konstantinos Kanellopoulos, Joël Lindegger, Zülal Bingöl, Gurpreet S. Kalsi, Ziyi Zuo, Can Firtina, Meryem Banu Cavlak, Jeremie Kim, Nika Mansouri Ghiasi, Gagandeep Singh, Juan Gómez-Luna, Nour Almadhoun Alserr, Mohammed Alser, Sreenivas Subramoney, Can Alkan, Saugata Ghose, and Onur Mutlu. SeGraM: A Universal Hardware Accelerator for Genomic Sequence-to-Graph and Sequence-to-Sequence Mapping. (2022), 638–655. https://doi.org/10.1145/3470496.3527436

[16] Biagioli E et al. Carneiro M, Poplin R. Enabling high throughput haplotype analysis through hardware acceleration. https://github.com/MauricioCarneiro/PairHMM/tree/master/doc.

[17] Kun-Mao Chao, William R Pearson, and Webb Miller. Aligning two sequences within a specified diagonal band. *Bioinformatics* 8, 5 (1992), 481–487. https://academic.oup.com/bioinformatics/article-abstract/8/5/481/213891

[18] Peng Chen, Chao Wang, Xi Li, and Xuehai Zhou. Hardware acceleration for the banded Smith-Waterman algorithm with the cycled systolic array. In *2013 International Conference on Field-Programmable Technology (FPT)*. 480–481. https://doi.org/10.1109/FPT.2013.6718421

[19] Ruei-Ting Chien, Yi-Lun Liao, Chien-An Wang, Yu-Cheng Li, and Yi-Chang Lu. Three-Dimensional Dynamic Programming Accelerator for Multiple Sequence Alignment. In *2018 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*. 1–5. https://doi.org/10.1109/NORCHIP.2018.8573523

[20] Ruei-Ting Chien, Yi-Lun Liao, Chien-An Wang, Yu-Cheng Li, and Yi-Chang Lu. Three-Dimensional Dynamic Programming Accelerator for Multiple Sequence Alignment. In *2018 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*. 1–5. https://doi.org/10.1109/NORCHIP.2018.8573523

[21] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Maleen Abeydeera, Logan Adams, Hari Angepat, Christian Boehn, Derek Chiou, Oren Firestein, Alessandro Forin, Kang Su Gatlin, Mahdi Ghandi, Stephen Heil, Kyle Holohan, Ahmad El Husseini, Tamas Juhasz, Kara Kagi, Ratna K. Kovvuri, Sitaram Lanka, Friedel van Megen, Dima Mukhortov, Prerak Patel, Brandon Perez, Amanda Rapsang, Steven Reinhardt, Bita Rouhani, Adam Sapek, Raja Seera, Sangeetha Shekar, Balaji Sridharan, Gabriel Weisz, Lisa Woods, Phillip Yi Xiao, Dan Zhang, Ritchie Zhao, and Doug Burger. Serving DNNs in Real Time at Datacenter Scale with Project Brainwave. *IEEE Micro* 38, 2 (2018), 8–20. https://doi.org/10.1109/MM.2018.022071131

[22] Vidushi Dadu, Sihao Liu, and Tony Nowatzki. PolyGraph: Exposing the Value of Flexibility for Graph Processing Accelerators. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 595–608. https://doi.org/10.1109/ISCA52012.2021.00053

[23] Tim Dunn, Harisankar Sadasivan, Jack Wadden, Kush Goliya, Kuan-Yu Chen, David Blaauw, Reetuparna Das, and Satish Narayanasamy. SquiggleFilter: An Accelerator for Portable Virus Detection. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 535–549. https://doi.org/10.1145/3466752.3480117

[24] 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)*. 69–82. https://doi.org/10.1109/ISCA.2018.00017

[25] Daichi Fujiki, Shunhao Wu, Nathan Ozog, Kush Goliya, David Blaauw, Satish Narayanasamy, and Reetuparna Das. SeedEx: A Genome Sequencing Accelerator for Optimal Alignments in Subminimal Space. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 937–950. https://doi.org/10.1109/MICRO50266.2020.00080

[26] Hasindu Gamaarachchi, Chun Wai Lam, Gihan Jayatilaka, Hiruna Samarakoon, Jared T Simpson, Martin A Smith, and Sri Parameswaran. GPU accelerated adaptive banded event alignment for rapid comparative nanopore signal analysis. *BMC bioinformatics* 21 (2020), 1–13. https://link.springer.com/article/10.1186/s12859-020-03697-x

[27] Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. DySER: Unifying Functionality and Parallelism Specialization for Energy-Efficient Computing. *IEEE Micro* 32, 5 (2012), 38–51. https://doi.org/10.1109/MM.2012.51

[28] Licheng Guo, Jason Lau, Zhenyuan Ruan, Peng Wei, and Jason Cong. Hardware Acceleration of Long Read Pairwise Overlapping in Genome Sequencing: A Race Between FPGA and GPU. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 127–135. https://doi.org/10.1109/FCCM.2019.00027

[29] Tae Jun Ham, David Bruns-Smith, Brendan Sweeney, Yejin Lee, Seong Hoon Seo, U Gyeong Song, Young H. Oh, Krste Asanovic, Jae W. Lee, and Lisa Wu Wills. Genesis: A Hardware Acceleration Framework for Genomic Data Analysis. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 254–267. https://doi.org/10.1109/ISCA45697.2020.00031

[30] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13. https://doi.org/10.1109/MICRO.2016.7783759

[31] Daniel S. Hirschberg. Algorithms for the Longest Common Subsequence Problem. *J. ACM* 24, 4 (oct 1977), 664–675. https://doi.org/10.1145/322033.322044

[32] Lei Jiang and Farzaneh Zokaee. EXMA: A Genomics Accelerator for Exact-Matching. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 399–411. https://doi.org/10.1109/HPCA51647.2021.00041

[33] Norman P. Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B. Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, Thomas Norrie, Nishant Patil, Sushma Prasad, Cliff Young, Zongwei Zhou, and David Patterson. Ten Lessons From Three Generations Shaped Google's TPUv4i : Industrial Product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 1–14. https://doi.org/10.1109/ISCA52012.2021.00010

[34] Saurabh Kalikar, Chirag Jain, Vasimuddin Md, and Sanchit Misra. Accelerating long-read analysis on modern CPUs. *bioRxiv* (2021). https://www.biorxiv.org/content/10.1101/2021.07.21.453294.abstract

[35] Saurabh Kalikar, Chirag Jain, Md Vasimuddin, and Sanchit Misra. Accelerating minimap2 for long-read sequencing applications on modern CPUs. *Nature Computational Science* 2, 2 (2022), 78–83. https://www.nature.com/articles/s43588-022-00201-8

[36] Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Computer Architecture Letters* 15, 1 (2016), 45–49. https://doi.org/10.1109/LCA.2015.2414456

[37] Donald E Kirk. *Optimal control theory: an introduction*. Courier Corporation.

[38] Mikhail Kolmogorov, Jeffrey Yuan, Yu Lin, and Pavel A Pevzner. Assembly of long, error-prone reads using repeat graphs. *Nature biotechnology* 37, 5 (2019), 540–546. https://www.nature.com/articles/s41587-019-0072-8

[39] Sergey Koren, Brian P Walenz, Konstantin Berlin, Jason R Miller, Nicholas H Bergman, and Adam M Phillippy. Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation. *Genome research* 27, 5 (2017), 722–736. https://genome.cshlp.org/content/27/5/722.short

[40] Niklas Krumm, Tychele N Turner, Carl Baker, Laura Vives, Kiana Mohajeri, Kali Witherspoon, Archana Raja, Bradley P Coe, Holly A Stessman, Zong-Xiao He, et al. Excess of rare, inherited truncating mutations in autism. *Nature genetics* 47, 6 (2015), 582–588. https://www.nature.com/articles/ng.3303

[41] Chenhao Li, Kern Rei Chng, Esther Jia Hui Boey, Amanda Hui Qi Ng, Andreas Wilm, and Niranjan Nagarajan. INC-Seq: accurate single molecule reads using nanopore sequencing. *GigaScience* 5, 1 (08 2016). https://doi.org/10.1186/s13742-016-0140-7 s13742-016-0140-7.

[42] Heng Li. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. *arXiv* (2013). https://arxiv.org/abs/1303.3997

[43] Heng Li and Richard Durbin. Inference of human population history from individual whole-genome sequences. *Nature* 475, 7357 (2011), 493–496. https://www.nature.com/articles/nature10231

[44] Yi-Lun Liao, Yu-Cheng Li, Nae-Chyun Chen, and Yi-Chang Lu. Adaptively Banded Smith-Waterman Algorithm for Long Reads and Its Hardware Accelerator. In *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 1–9. https://doi.org/10.1109/ASAP.2018.8445105

[45] Mao-Jan Lin, Yu-Cheng Li, and Yi-Chang Lu. Hardware Accelerator Design for Dynamic-Programming-Based Protein Sequence Alignment with Affine Gap Tracebacks. In *2019 IEEE Biomedical Circuits and Systems Conference (BioCAS)*. 1–4. https://doi.org/10.1109/BIOCAS.2019.8919080

[46] Ruibang Luo, Chak-Lim Wong, Yat-Sing Wong, Chi-Ian Tang, Chi-Man Liu, Chi-Ming Leung, and Tak-Wah Lam. Exploring the limit of using a deep neural network on pileup data for germline variant calling. *Nature Machine Intelligence* 2, 4 (2020), 220–227. https://www.nature.com/articles/s42256-020-0167-4

[47] Advait Madhavan, Timothy Sherwood, and Dmitri Strukov. Race Logic: A hardware acceleration for dynamic programming algorithms. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 517–528. https://doi.org/10.1109/ISCA.2014.6853226

[48] Santiago Marco-Sola, Juan Carlos Moure, Miquel Moreto, and Antonio Espinosa. Fast gap-affine pairwise alignment using the wavefront algorithm. *Bioinformatics* 37, 4 (2021), 456–463. https://academic.oup.com/bioinformatics/article-abstract/37/4/456/5904262

[49] W Richard McCombie, John D McPherson, and Elaine R Mardis. Next-generation sequencing technologies. *Cold Spring Harbor perspectives in medicine* 9, 11 (2019), a036798. http://perspectivesinmedicine.cshlp.org/content/9/11/a036798.short

[50] Ruth R Miller, Vincent Montoya, Jennifer L Gardy, David M Patrick, and Patrick Tang. Metagenomics for pathogen detection in public health. *Genome medicine* 5, 9 (2013), 1–14. https://link.springer.com/article/10.1186/gm485

[51] Sean Murray, Will Floyd-Jones, George Konidaris, and Daniel J. Sorin. A Programmable Architecture for Robot Motion Planning Acceleration. In *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Vol. 2160-052X. 185–188. https://doi.org/10.1109/ASAP.2019.000-4

[52] Eugene W Myers and Webb Miller. Optimal alignments in linear space. *Bioinformatics* 4, 1 (1988), 11–17. https://academic.oup.com/bioinformatics/article/4/1/

11/205106

[53] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. Stream-dataflow acceleration. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 416–429. https://doi.org/10.1145/3079856.3080255

[54] Sergey Nurk, Sergey Koren, Arang Rhie, Mikko Rautiainen, Andrey V Bzikadze, Alla Mikheenko, Mitchell R Vollger, Nicolas Altemose, Lev Uralsky, Ariel Gershman, et al. The complete sequence of a human genome. *Science* 376, 6588 (2022), 44–53. https://www.science.org/doi/abs/10.1126/science.abj6987

[55] Yukiteru Ono, Kiyoshi Asai, and Michiaki Hamada. PBSIM2: a simulator for long-read sequencers with a novel generative model of quality scores. *Bioinformatics* 37, 5 (2021), 589–595. https://academic.oup.com/bioinformatics/article-abstract/37/5/589/5911629

[56] Angshuman Parashar, Michael Pellauer, Michael Adler, Bushra Ahsan, Neal Crago, Daniel Lustig, Vladimir Pavlov, Antonia Zhai, Mohit Gambhir, Aamer Jaleel, Randy Allmon, Rachid Rayess, Stephen Maresh, and Joel Emer. Triggered Instructions: A Control Paradigm for Spatially-Programmed Architectures. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (Tel-Aviv, Israel) *(ISCA '13)*. Association for Computing Machinery, New York, NY, USA, 142–153. https://doi.org/10.1145/2485922.2485935

[57] Francesco Peverelli, Lorenzo Di Tucci, Marco D. Santambrogio, Nan Ding, Steven Hofmeyr, Aydin Buluç, Leonid Oliker, and Katherine Yelick. GPU accelerated partial order multiple sequence alignment for long reads self-correction. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1–9. https://doi.org/10.1109/IPDPSW50202.2020.00039

[58] Ryan Poplin, Valentin Ruano-Rubio, Mark A DePristo, Tim J Fennell, Mauricio O Carneiro, Geraldine A Van der Auwera, David E Kling, Laura D Gauthier, Ami Levy-Moonshine, David Roazen, et al. Scaling accurate genetic variant discovery to tens of thousands of samples. *BioRxiv* (2017), 201178. https://www.biorxiv.org/content/10.1101/201178.abstract

[59] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Plasticine: A reconfigurable architecture for parallel patterns. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 389–402. https://doi.org/10.1145/3079856.3080256

[60] Joshua Quick, Nathan D Grubaugh, Steven T Pullan, Ingra M Claro, Andrew D Smith, Karthik Gangavarapu, Glenn Oliveira, Refugio Robles-Sikisaka, Thomas F Rogers, Nathan A Beutler, et al. Multiplex PCR method for MinION and Illumina sequencing of Zika and other virus genomes directly from clinical samples. *Nature protocols* 12, 6 (2017), 1261–1276. https://www.nature.com/articles/nprot.2017.066

[61] 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 (2018), 1176934318760543. https://journals.sagepub.com/doi/pdf/10.1177/1176934318760543

[62] Harisankar Sadasivan, Milos Maric, Eric Dawson, Vishanth Iyer, Johnny Israeli, and Satish Narayanasamy. Accelerating Minimap2 for accurate long read alignment on GPUs. *J Biotechnol Biomed* 6, 1 (2023), 13–23. https://doi.org/10.1109/NORCHIP.2018.8573523

[63] Harisankar Sadasivan, Daniel Stiffler, Ajay Tirumala, Johnny Israeli, and Satish Narayanasamy. GPU-accelerated Dynamic Time Warping for Selective Nanopore Sequencing. *bioRxiv* (2023), 2023–03.

[64] Bertil Schmidt and Christian Hundt. cuDTW++: Ultra-Fast Dynamic Time Warping on CUDA-Enabled GPUs. In *European Conference on Parallel Processing*. Springer, 597–612. https://link.springer.com/chapter/10.1007/978-3-030-57675-2_37

[65] James E. Smith. Decoupled Access/Execute Computer Architectures. (1982), 112–119.

[66] Temple F Smith, Michael S Waterman, et al. Identification of common molecular subsequences. *Journal of molecular biology* 147, 1 (1981), 195–197. https://doi.org/10.1016/0022-2836(81)90087-5

[67] Aaron Stillmaker and Bevan Baas. Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm. *Integration* 58 (2017), 74–81. https://www.sciencedirect.com/science/article/pii/S0167926017300755

[68] 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)*. 1–12. https://doi.org/10.1109/ISPASS51385.2021.00012

[69] Jesmin Jahan Tithi, Neal C. Crago, and Joel S. Emer. Exploiting spatial architectures for edit distance algorithms. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 23–34. https://doi.org/10.1109/ISPASS.2014.6844458

[70] Yatish Turakhia, Gill Bejerano, and William J. Dally. Darwin: A Genomics Co-Processor Provides up to 15,000X Acceleration on Long Read Assembly. (2018), 199–213. https://doi.org/10.1145/3173162.3173193

[71] MCJ van Lanschot, LJW Bosch, M de Wit, B Carvalho, and GA Meijer. Early detection: The impact of genomics. *Virchows Archiv* 471, 2 (2017), 165–173. https://link.springer.com/article/10.1007/s00428-017-2159-2

[72] Robert Vaser, Ivan Sović, Niranjan Nagarajan, and Mile Šikić. Fast and accurate de novo genome assembly from long uncorrected reads. *Genome research* 27, 5 (2017), 737–746. https://genome.cshlp.org/content/27/5/737.short

[73] Md. Vasimuddin, Sanchit Misra, Heng Li, and Srinivas Aluru. Efficient Architecture-Aware Acceleration of BWA-MEM for Multicore Systems. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 314–324. https://doi.org/10.1109/IPDPS.2019.00041

[74] Kris A. Wetterstrand. DNA sequencing costs: Data. https://www.genome.gov/about-genomics/fact-sheets/DNA-Sequencing-Costs-Data

[75] Ryan R Wick, Louise M Judd, and Kathryn E Holt. Performance of neural network basecalling tools for Oxford Nanopore sequencing. *Genome biology* 20, 1 (2019), 1–10. https://link.springer.com/article/10.1186/s13059-019-1727-y

[76] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth A. Ross. Q100: The Architecture and Design of a Database Processing Unit. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake City, Utah, USA) *(ASPLOS '14)*. Association for Computing Machinery, New York, NY, USA, 255–268. https://doi.org/10.1145/2541940.2541961

[77] Xiao Wu, Arun Subramaniyan, Zhehong Wang, Satish Narayanasamy, Reetuparna Das, and David Blaauw. A High-Throughput Pruning-Based Pair-Hidden-Markov-Model Hardware Accelerator for Next-Generation DNA Sequencing. *IEEE Solid-State Circuits Letters* 4 (2021), 31–35. https://doi.org/10.1109/LSSC.2020.3045148

[78] 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 (2019), 1409–1413. https://doi.org/10.1126/science.aax4588

Optimization Notice: Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information go to http://www.intel.com/performance.

Intel, Xeon, and Intel Xeon Phi are trademarks of Intel Corporation in the U.S. and/or other countries.

Received 21 Nov. 2022; Revised 20 Feb. 2023; Accepted 9 Mar. 2023