



DP-HLS: A High-Level Synthesis Framework for Accelerating Dynamic Programming Algorithms in Bioinformatics

Anshu Gupta^{†*}, Yingqi Cao^{†*}, Jason Liang[†] and Yatish Turakhia[†]
 University of California San Diego[†]
 {ang037, yic033, jsliang, yturakhia}@ucsd.edu

Abstract—Dynamic programming (DP) is a widely used algorithmic paradigm, particularly in bioinformatics, finding applications in a wide spectrum of tasks, including read assembly, homology search, gene annotation, basecalling, and phylogenetic inference. Due to its computationally intensive nature, many ASIC- and FPGA-based accelerators have been proposed in recent years to accelerate specific tasks. However, DP algorithms in bioinformatics can vary considerably, and most existing solutions are customized for a single application, representing just one design point within the broader DP space. These implementations typically rely on low-level hardware description languages (HDLs), often requiring months of manual implementation effort. This paper introduces DP-HLS, a novel framework based on High-Level Synthesis (HLS) that simplifies and accelerates the development of a vast set of bioinformatically relevant 2-D DP algorithms in hardware. DP-HLS achieves this by introducing a new abstraction layer that decouples the front-end specification from predefined HLS-based back-end optimizations, enabling users to efficiently develop new 2-D DP kernels in C++ and deploy them on FPGAs without needing any expertise in hardware design or HLS. In our experience, DP-HLS significantly reduced the development time of new kernels (months to days) and produced designs with comparable resource utilization to open-source hand-coded HDL-based implementations and performance within 7.7–16.8% margin. DP-HLS is compatible with AWS[®] EC2 F1 FPGA instances. To showcase its versatility, we implemented 15 diverse 2-D DP kernels using the DP-HLS framework, achieving 1.38–41× improved cost-efficiency over state-of-the-art GPU and CPU baselines and providing the first open-source FPGA implementation for several of them. The DP-HLS codebase is available freely under the MIT license at <https://github.com/TurakhiaLab/DP-HLS>.

I. INTRODUCTION

Genomic data is one of the fastest-growing data types globally, far outpacing Moore’s law in terms of data generation [1]. To keep pace with the escalating computational demands of this data, numerous recent hardware acceleration efforts - spanning GPUs, FPGAs, and ASICs - have targeted bioinformatics applications in recent years [2]–[25]. Although many of these accelerators deliver significant speedups, they are typically tailored to narrow applications and tend to lack the flexibility needed to generalize across broad tasks in bioinformatics. Despite this limitation, we observe that they frequently share two interesting characteristics, as discussed.

First, many of these solutions use 2-D dynamic programming (DP) algorithm that involves populating a 2-D scoring

matrix based on a recursive formula, optionally followed by a traceback step to get the optimal alignment path [26], [27]. This is unsurprising, as 2-D DP provides an efficient framework for comparing biological sequences - such as DNA, RNA, proteins, or even electrical signals from sequencing instruments - which is fundamental to many bioinformatics tasks. This includes local pairwise alignments [28], multiple sequence alignment [29], [30], homology searches [31], [32], whole-genome alignments [33], basecalling [34], and variant calling [35]. Due to their computational intensity, 2-D DP-based algorithms often dominate the overall runtime of these applications [36]. Even commercial hardware vendors have recognized this fact and begun to optimize for DP—for example, NVIDIA[®] introduced a specialized instruction, DPX, specifically to accelerate DP algorithms on latest GPUs [37]. *Second* key characteristic, particularly in FPGA and ASIC solutions [10]–[25], is the use of the same hardware primitive, i.e., linear systolic array, which has been recognized since the 1980s for its efficiency in accelerating 2-D DP algorithms [38].

Despite sharing a common algorithmic paradigm (dynamic programming) and hardware primitive (linear systolic array), repurposing existing hardware accelerators for new applications remains challenging. This is because most previous solutions focus on a specific 2-D DP algorithm and are designed at the Register Transfer Level (RTL) using low-level Hardware Description Languages (HDLs) like Verilog or VHDL. As a result, supporting new 2-D DP variants - of which there is a rich and complex variety in bioinformatics (see Section II-B) - would necessitate complex modifications at the signal level, as well as verification, optimization, and hardware resynthesis - a process that is not only time-consuming and error-prone, but also demands significant hardware expertise. This level of development is also largely out of reach for most bioinformaticians, who, despite being skilled in algorithms and software, typically lack the specialized training required for low-level hardware design.

To address these challenges, we present *DP-HLS*, a novel framework based on High-Level Synthesis (HLS) for accelerating broad kernels from the 2-D DP paradigm (Section II-A) on FPGAs. DP-HLS builds on the key insight that when a broad class of algorithms can be mapped to a shared hardware primitive, the hardware-specific HLS directives can be encapsulated within a unified *back-end*, allowing the *front-*

* These authors contributed equally.

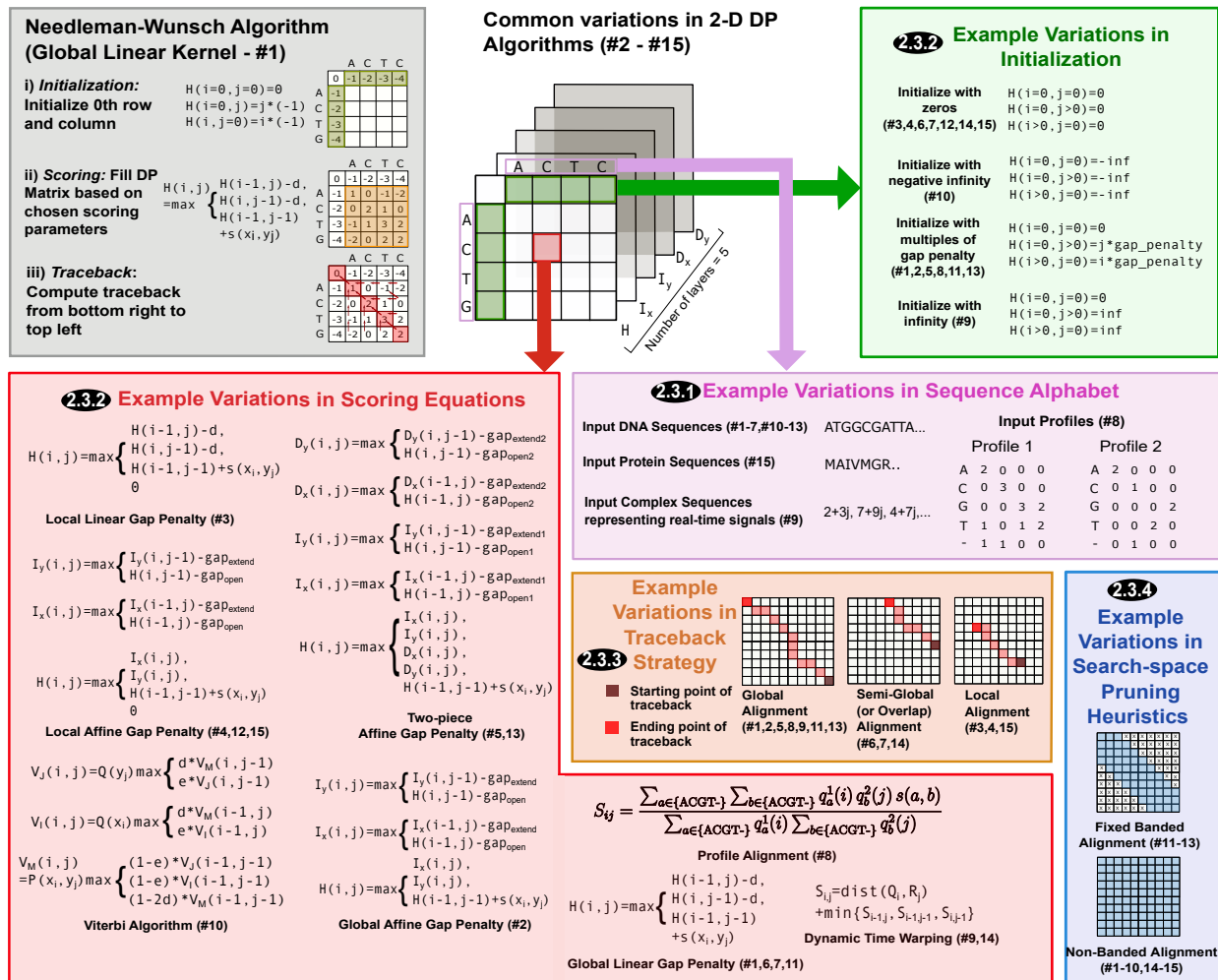


Fig. 1: Top-left: The steps of the Needleman–Wunsch algorithm, a canonical example of a simple 2-D DP algorithm. Center-right: Common variations in 2-D DP algorithms in Bioinformatics. Kernels are indexed using #'s based on Table I.

end to focus solely on the algorithmic specification. We primarily target FPGAs in this work as they have not only been successfully adopted in commercial bioinformatics applications [39]–[41], but they are also readily accessible to the bioinformatics community through off-the-shelf products and cloud providers. This makes them both easy and cost-effective to deploy, even for niche applications. DP-HLS makes the following key contributions:

- 1) We present DP-HLS, an HLS framework tailored for 2-D DP paradigm, that introduces a layer of abstraction in the HLS flow by separating the front-end and back-end to enable flexible kernel generation with improved productivity. Existing HLS tools and frameworks [42]–[48], rely on a fixed set of pre-defined kernels that can only be tuned at the parameter level and require substantial hardware expertise to achieve RTL-level performance. In contrast, DP-HLS front-end allows users to define new 2-D DP kernels at the algorithmic level in C++ without HLS or hardware design expertise. At the back-end, DP-HLS generates an optimized linear systolic array architecture

using a fixed set of HLS directives, each customized with unique functionality, operations, and dataflow patterns to match the specific needs of its target algorithm.

- 2) To showcase its versatility, we implemented 15 diverse, bioinformatically relevant 2-D DP kernels on FPGAs using the DP-HLS framework (Table II). For most of these, DP-HLS provides the first open-source FPGA implementation. These kernels span a broad spectrum of applications - from basecalling to protein sequence alignment. All kernels are functionally verified and deployed on AWS® EC2 F1 instances for broad accessibility.
- 3) We performed extensive evaluations and observed that DP-HLS provides 1.47–41× higher throughput, 1.38–41× more cost-efficiency, and up to 2.27× more energy-efficiency compared to CPU and GPU baselines. Moreover, DP-HLS delivers throughput and resource utilization on par with hand-optimized RTL implementations, while reducing the design implementation time from months to days.
- 4) DP-HLS is built with extensibility in mind, enabling users

to add new 2-D DP kernels beyond the 15 examples (Fig. 1) provided, tailored to their requirements.

- 5) We demonstrate, through an example, that previously proposed tiling heuristics [12], [24] are compatible with DP-HLS and can be used for performing both short and long sequence alignments on the FPGA.

Our broad insight is that while HLS does not necessarily improve productivity over RTL design for a single kernel due to challenges in identifying and tuning a required set of HLS directives (Section VII-F), it proves highly effective when targeting multiple algorithms that share a common pattern. DP-HLS demonstrates how separating front-end and back-end in HLS frameworks improves productivity and efficiency across a broad class of 2-D DP algorithms that map to the same hardware primitive - a strategy that could also be applicable to artificial intelligence, where diverse operations (matrix multiplication, convolution, etc.) map well to 2-D systolic array architectures [49]–[53].

II. BACKGROUND AND MOTIVATION

A. The 2-D Dynamic Programming Paradigm

The 2-D DP paradigm is essential for sequence comparison tasks (identifying similarities and differences between two or more sequences) in bioinformatics. It involves three main steps (Fig. 1): (i) *initialization*, where two biological sequences are placed along the 2-D DP matrix axes and boundary scores are set with predefined values; (ii) *matrix fill*, where remaining cells are scored recursively using neighboring values (from the top, left, and diagonal neighbor cells); and (iii) optional *traceback*, which reconstructs the optimal path through the matrix or returns only the best score. One of the simplest 2-D DP algorithms is the Needleman-Wunsch algorithm [70], illustrated in Fig. 1.

B. Variations in the 2-D DP Paradigm

Bioinformatics applications exhibit a rich variety of data types and algorithmic variations within the 2-D DP paradigm, which cannot be captured by a finite set of design parameters and demand a high level of adaptability, such as that offered by DP-HLS. To highlight this point, Table I provides a selection of 2-D DP kernels (indexed with ‘#’ hereafter) that are commonly used in various bioinformatics applications, widely cited, and in some cases, targeted by hardware accelerators. Here, we highlight how these variations, though diverse, fall into four broad categories: i) *Sequence Alphabet*, ii) *Scoring Equations*, iii) *Traceback Strategy*, and iv) *Search-space Pruning Heuristics* (Fig. 1). Below, we summarize these variations (refer to original sources in Table I for details).

1) *Sequence Alphabet*: A key source of variation in the 2-D DP paradigm is the choice of sequence alphabet, the set of symbols representing input sequences. Most bioinformatics applications use 4 or 20 character alphabets for DNA, RNA, or proteins, though some variations that introduce ambiguous bases, e.g., Ns, also exist [32], [33]. More complex alphabets appear in Profile Alignment (#8), where each position is represented as a tuple of 5 (DNA) or 21 (proteins) integers

reflecting base or amino acid frequencies and gaps [71] (Fig. 1). Dynamic Time Warping (DTW) kernels (#9 and #14) use real or complex numbers to compare raw time-series signals (Fig. 1), such as in nanopore basecalling [67]. In summary, bioinformatics applications exhibit a diverse range of input types - from simple 2-bit integers to complex tuples of multiple floating-point values.

2) *Scoring Equations*: In the 2-D DP paradigm, scoring refers to the recurrence equations used to compute cell values - typically rewarding matches and penalizing mismatches or gaps. A 2-D DP algorithm in bioinformatics may include an arbitrary number of recurrence equations, each with its own specification. We highlight this fact using common variations of scoring equations below:

(a) **Substitution Scores**: Simplest form of scoring systems uses a single value to reward a match or penalize a mismatch, or both. However, in some cases, more complex scoring systems based on large substitution matrices are used to account for varying substitution frequencies (e.g., transitions vs. transversions) [72] (Fig. 1). Substitution scores can also be computed dynamically - for example, in Profile Alignments (#8) and DTW (#9) - using metrics like Sum-of-Pairs scoring [71] and Manhattan/Euclidean distance [67], [73].

(b) **Gap Penalties**: Gap penalties are used to penalize insertions and deletions (Fig. 1). One of the simplest methods is linear gap penalty, which applies a constant penalty for each gap. More biologically realistic models use affine gap penalties, which assign a higher penalty to opening a gap than to extending one [74], and require three score values per cell (H, I, D) to track different alignment states. Minimap2 [58] builds a two-piece affine gap model, computing five values per cell to better distinguish biological gaps from sequencing errors. In general, each additional affine layer introduces two more values per cell, allowing the gap cost to approximate a smooth convex function.

(c) **Initialization**: The scoring equations also define how the initial row and column are scored. Here too, there is broad variation, ranging from constant values (such as 0 or $-\infty$) to using a linear function of the gap penalties (Fig. 1).

(d) **Min/Max Objective Function**: Most 2-D DP formulations (#1-7, #11-13) penalize the gap and find the maximum cell score, whereas DTW aims to find the minimum cell score while the score represents distances between the sequence. In this case, we need to replace the `max` function in recurrence equations with `min` (Fig. 1).

3) *Traceback Strategy*: The traceback step identifies the alignment corresponding to the optimal score. While recurrence equations define valid transitions, the traceback strategy determines where the path *starts* and *ends*. Four strategies are common (but not exhaustive) - global, local, semi-global, and overlap - and they differ in how they influence the recurrence and initialization equations (Fig. 1). *Global* strategy performs end-to-end whole genome sequence comparison, with a traceback path from the bottom-right to the top-left cell of the 2-D DP matrix. *Local* strategy finds similar subsequences and is used to identify conserved motifs or functional regions in

TABLE I: Common bioinformatics kernels using 2-D DP algorithms, along with their associated tools, applications, and modifications relative to the baseline kernel (#1: Global Linear Alignment). Some tools implement multiple kernels. This selection represents a small subset of kernels that can be implemented under the DP-HLS framework.

#	Alphabet	2-D DP Kernel	State-of-the-art Tools	Example Applications	Modifications in DP-HLS (color-coded as per Fig. 1)
1	DNA	Global Linear Alignment (Needleman-Wunsch)	BLAST [32], EMBOSS Stretcher [54]	Similarity Search	N/A
2	DNA	Global Affine Alignment (Gotoh)	BLAST [32], EMBOSS Needle [54]	Accurate Similarity Search	Scoring
3	DNA	Local Linear Alignment (Smith-Waterman)	BLAST [32], FASTA [55], BLAT [56]	Homology Search	Initialization, Scoring, Traceback
4	DNA	Local Affine Alignment (Smith-Waterman-Gotoh)	BLAST [32], LASTZ [57]	Whole Genome Alignment	Scoring, Initialization, Traceback
5	DNA	Global Two-piece Affine Alignment	Minimap2 [58]	Long Read Alignment	Scoring
6	DNA	Overlap Alignment	CANU [59], Flye [60]	Genome Assembly	Initialization, Traceback
7	DNA	Semi-global Alignment	BWA-MEM [61]	Short Read Alignment	Initialization, Traceback
8	Seq. Profiles	Profile Alignment	CLUSTALW [62], MUSCLE [30]	Multiple Sequence Alignment	Sequence Alphabet, Scoring
9	Complex Nos.	Dynamic Time Wrapping Algorithm (DTW)	SquiggleKit [63]	Basecalling	Sequence Alphabet, Initialization, Scoring
10	DNA	Viterbi Algorithm (PairHMM)	HMMER [64], AUGUSTUS [65]	Remote Homology Search, Gene Prediction	Initialization, Scoring (no Traceback)
11	DNA	Banded Global Linear Alignment	BLAST [32], Bowtie [66]	Fast Similarity Search	Banding
12	DNA	Banded Local Affine Alignment	Minimap2 [58]	Long Read Assembly	Initialization, Scoring, Banding (no Traceback)
13	DNA	Banded Global Two-piece Affine Alignment	Minimap2 [58]	Long Read Assembly	Scoring, Banding
14	Integers	Semi-global DTW (sDTW)	SquiggleFilter [67], RawHash [68]	Basecalling	Sequence Alphabet, Initialization, Scoring, Traceback
15	Amino acids	Local Affine Alignment with protein sequences	EMBOSS Water [54], BLASTp [32], DIAMOND [69]	Protein Sequence Alignment	Sequence Alphabet, Initialization, Scoring, Traceback

sequences, with the traceback from the highest-scoring cell to a 0-scoring cell. *Semi-global* strategy matches one sequence end-to-end with a subsequence of the other, with the traceback starting from the bottom row’s highest-scoring cell to the top row. *Overlap* strategy, used in genome assembly, matches sequence ends (prefixes or suffixes) with traceback path from the highest-scoring cell in the rightmost column (bottom row) to the top row (leftmost column) of the 2-D DP matrix.

4) *Search-space Pruning Heuristics*: As 2-D DP matrices grow quadratically with sequence lengths, many algorithms employ heuristics to prune unpromising regions (Fig. 1). *Fixed* banding algorithms compute cells near the main diagonal within a fixed distance [75], while *adaptive* methods like X-Drop [76] adjust band size dynamically based on cell scores.

III. AN OVERVIEW OF THE DP-HLS FRAMEWORK

DP-HLS is a framework that can be used to accelerate a wide variety of 2-D DP algorithms on FPGAs (Fig. 2). It consists of two main components: the *front-end* (Section IV), which enables users even without HLS expertise, to define kernels in C/C++ and supports co-simulation, verification, and FPGA deployment; and the *back-end* (Section V), which is built using AMD® Vitis HLS [77] and applies HLS directives to generate optimized hardware designs. The front-end offers user specification in C/C++ to provide a high degree of flexibility, capable of supporting 2-D DP kernels listed in Table I

and well beyond, while the back-end can translate any such specification into a highly efficient FPGA implementation.

IV. FRONT-END IMPLEMENTATION

The front-end component of the DP-HLS framework is available to users to define new 2-D DP kernels and efficiently deploy them on FPGAs. Fig. 2A shows the design flow, involving kernel configuration, simulation, synthesis, co-simulation, and final implementation. Using examples from the 15 2-D DP kernels in Table I, we outline six key steps required to configure the front-end for a specific kernel (Fig. 2B).

① **Customizing Data Types and Parameters**: The DP-HLS framework supports custom data types of variable precision for scoring, traceback, and logic indices, enabling users to optimize efficiency for their specific kernel requirements. It also allows customization of scoring and input parameters. Primary front-end customizations are enumerated below.

1. **Sequence Alphabets**: As explained in Section II-B1, the sequence alphabet in 2-D DP kernels requires high configurability. DP-HLS addresses this by allowing users to define a custom data type, `char_t`, for the sequence alphabet. Listing 1 depicts how a 2-bit unsigned integer used as `char_t` to represent the four nucleotide bases in DNA and RNA (for kernel #1) can be modified to a 5-bit unsigned integer for representing protein sequence alphabets (for kernel #15). The front-end allows alphabet of more complex data types—for

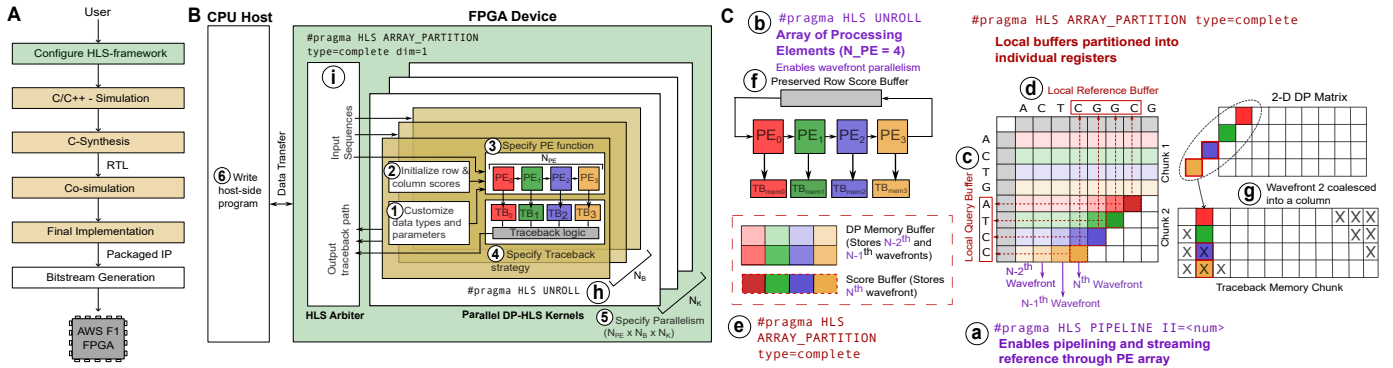


Fig. 2: DP-HLS implementation overview. (A) Basic workflow of DP-HLS kernels from user customization to FPGA deployment. (B) Front-end layout of DP-HLS kernels with customizable modules. (C) Key back-end optimizations with corresponding pragma directives.

example, in the DTW kernel (#9), `char_t` is a struct of two 32-bit fixed-point numbers representing the real and imaginary parts of complex temporal signals being compared.

2. Scoring Layers and Data Types: As described in Section II-B2, some 2-D DP kernels involve multiple recurrence equations, each computing a unique value per cell. DP-HLS front-end provides a variable, called `N_LAYERS`, which configures the number of unique values computed and stored per cell of the 2-D DP matrix. So, for kernels with an affine gap penalty (Kernels #2, #4, #10, #12), `N_LAYERS`=3, with two-piece affine gap penalty (Kernels #5, #13), it is set to 5, and for the remaining kernels, it is set to 1 (Listing 1).

3. Scoring Parameters: DP-HLS allows users to define an arbitrary number of scoring parameters with any data type in a C/C++ struct, called `ScoringParams`, with values set at runtime by the host code. Listing 1 shows an example with three parameters (`match`, `mismatch`, and `linear_gap`) used by the Global Linear kernel (#1). Users can add more scoring parameters for the affine-gap penalty-based equations (for kernel #15).

4. Maximum Sequence Lengths: In DP-HLS, users can set `MAX_REFERENCE_LENGTH` and `MAX_QUERY_LENGTH` to define memory allocation for sequences and traceback pointers on the FPGA device. While the kernel supports fixed maximum lengths, longer sequences can be handled using software tiling approaches on the host-side [12]. We demonstrate this by applying software tiling to kernel #2, #4, and #5 (see Section VI).

5. Traceback Pointer Data Types and States: Traceback pointers in DP-HLS are stored in data type `tb_t` which users may define using arbitrary precision data type provided in Vitis HLS. Listing 1 depicts how it can be defined as `ap_uint<2>` for kernel #1 which can be modified to `ap_uint<4>` for kernel #15, since recurrence equations require a minimum of 2-bits and 4-bits, respectively to represent the traceback pointers.

The traceback logic in the final step of 2-D DP algorithms is equivalent to a finite state machine (FSM) in which the current state and 2-D DP matrix cell score determine the

next state and cell in the score matrix, which is translated to the traceback path. In DP-HLS, users enumerate the possible traceback states in the variable `TB_STATE`. Listing 1 shows that kernel #1 and #15 enumerate three traceback states - MM, INS, and DEL. User can modify this to add extra traceback states (`LONG_INS`, `LONG_DEL`), one for each additional recurrence equation modeling long gap scores (in kernel (#5)). DP-HLS framework also offers a no-rollback option, used by the Viterbi (#10) and Banded Local Affine (#12) kernels, to skip the rollback step.

6. Banding Width: DP-HLS allows users to use a fixed banding search space pruning strategy by setting the macros `BANDING` and `BANDWIDTH` to the desired band size.

(2) Initializing Row and Column Scores: The DP-HLS framework includes two internal 2-D arrays, `init_row_scr` and `init_col_scr`, to store the scores of the initial row and column of the 2-D DP matrix, with dimensions of `MAX_REFERENCE_LENGTH × N_LAYERS` and `MAX_QUERY_LENGTH × N_LAYERS`, respectively. The users can specify the values of these arrays, defining how DP-HLS would initialize them at runtime. Listing 1 shows the initialization step of kernel #1, which has a single scoring layer at index 0 whose first row and column are initialized to account for gaps at the start of the alignment, whereas #15 initializes the rows and columns to 0.

(3) Specifying PE Function: In DP-HLS, all computations involved in the *matrix fill* step of the 2-D DP kernels (Section II-B2) are executed by computing units known as *processing elements* (PEs). Within the DP-HLS front-end component, users only need to specify the recurrence equations in `PE_func` function for computing the score and traceback pointer for a single cell (i, j), located at row i and column j of the 2-D DP matrix. The back-end component manages systolic communication between PEs and the storage of data in memory buffers. Listing 1 demonstrates how the PE functions can be modified from one kernel to another, depending on the use cases.

(4) Specifying Traceback Strategy: An FSM intuitively defines the traceback logic in the final stage of the 2-D DP

```

1 //Sequence Alphabet
2 typedef ap_uint<5> char_t;
3
4 //Scoring Layer
5 N_LAYERS = 3;
6
7 //Scoring Parameters
8 struct ScoringParams {type_t
    gap_open; type_t
    gap_extend; type_t
    mismatch; type_t match;
    type_t linear_gap; }
    params;
9
10 //Traceback States
11 enum TB_STATE {MM, INS, DEL,
12 } tb_next_state, tb_curr_state;
13
14 //Traceback pointer data type
15 typedef ap_uint<4> tbp_t;
16
17 //Initialization
18 type_t gap = scoring_params.
    linear_gap;
19 for (int i = 0; i <
    MAX_REFERENCE_LENGTH; i
    ++){init_row_scr[i][0] =
    0; }
20 for (int i = 0; i <
    MAX_QUERY_LENGTH; i++){
    init_col_scr[i][0] = 0; }
21
22 //PE scoring function
23 type_t insert_open =
    left_prev[1] + params.
    gap_open;
24 type_t insert_extend =
    left_prev[0] + params.
    gap_extend;
25 bool insert_open_b =
    insert_open >
    insert_extend;
26 write_score[0] = insert_open_b
    ? insert_open :
    insert_extend;
27
28 type_t delete_open = up_prev
    [1] + params.gap_open;
29 type_t delete_extend = up_prev
    [2] + params.gap_extend;
30 bool delete_open_b =
    delete_open >
    delete_extend;
31 write_score[2] = delete_open_b
    ? delete_open :
    delete_extend;
32
33 type_t match = dp_mem_diag[0]
    + ((lc_qry_val ==
    lc_ref_val) ? params.
    match : params.
    mismatch);
34
35 type_t max_value = (
    write_score[0] >
    write_score[2]) ?
    write_score[0] :
    write_score[2];
36 write_score[1] = (max_value >
    match) ? max_value :
    match;

```

Listing 1: Front-end code for initialization and matrix-fill specifications for (Left) Global Linear kernel (#1) and (Right) Local Affine kernel with protein sequences (#15). This example helps demonstrate how the baseline kernel (#1) can be transformed into a more complex kernel (#15) with few code changes that do not affect any back-end optimizations.

```

1 //Traceback pointer
2 if (max_value == write_score[0])
3 {dir_tb = TB_LEFT;}
4 else if (max_value == write_score
    [2])
5 {dir_tb = TB_UP;}
6 else if (max_value == write_score
    [1])
7 {dir_tb = TB_DIAG;}
8 else if (max_value == 0){dir_tb =
    TB_END;}
9 wt_tbp = dir_tb + insert_tb +
    delete_tb;
10
11 //Traceback State Transitions
12 if (tb_state == TB_STATE::MM){
13 if (tb_ptr == TB_DIAG){tb_move =
    AL_MMI; }
14 else if (tb_ptr == TB_UP){
    TB_STATE::DEL; tb_move =
    AL_NULL; }
15 else if (tb_ptr == TB_LEFT){
    TB_STATE::INS; tb_move =
    AL_NULL; }
16 else if (tb_ptr == TB_END) {
    tb_move = AL_END;}
17 else {tb_move = AL_END;}
18 tb_state = TB_STATE::MM;}
19 else if (state == TB_STATE::DEL){
20 if (tb_ptr == TB_DIAG or TB_UP)
    continue
21 else {state = TB_STATE::MM;}
22 tb_move = AL_DEL;}
23 else if (state == TB_STATE::INS){
24 if (tb_ptr == TB_DIAG or TB_LEFT)
    continue
25 else {state = TB_STATE::MM;}
26 tb_move = AL_INS;}

```

Listing 2: Front-end code with traceback specifications for (Left) Global Linear kernel (#1) and (Right) Local Affine kernel with protein sequences (#15).

algorithm (Section II-B3). DP-HLS allows for customization of the FSM states and traceback pointers. For multiple scoring matrices, each matrix maps to a state, and transitions represent jumps between matrices. Listing 2 shows an example where an outer if-statement checks the current `tb_state`, assigns the new state, and directs the traceback via `wt_tbp` for INS, DEL, MM, or AL_END (end of traceback). Similar changes needs to be done for additional traceback state transitions, as shown for kernel #15.

⑤ **Specifying Parallelism:** The front-end component provides parameters (N_{PE} , N_B , N_K) that users can adjust to parallelize the synthesized design empirically, without understanding the internal details of the DP-HLS back-end and compiler optimizations. N_{PE} determines the level of *inner-loop parallelism* for a single pair of sequences. DP-HLS also exploits *outer-loop parallelism* across multiple sequence pairs by setting the parameters N_B and N_K . This allows the processing of $N_B \times N_K$ independent sequence pairs, with N_K independent channels to the host CPU (to take advantage of CPU multi-threading, for example), each consisting of N_B blocks sharing a single arbiter on the device, as shown in Fig. 2B. The design allows linking N_K heterogeneous kernels (e.g.,

a mix of global and local aligners) seamlessly in the design, a process that would be quite cumbersome with HDL.

⑥ **Writing Host-Side Program:** After completing the device specification, the user must define the host application to manage pre-processing and transfer of input sequences to the device, invoke device kernels, and receive alignment output from the device and/or perform tiling (for long reads). The host program uses OpenCL syntax [78], and relevant examples from our 15 implemented kernels (Table I) are provided in the DP-HLS codebase. The full end-to-end working code, including compilation and execution on FPGA, is also available in our open-source codebase for reproducibility.

V. BACK-END IMPLEMENTATION

The DP-HLS back-end component performs efficient mapping of 2-D DP algorithms to a linear systolic array, which is customized as per front-end specifications. The back-end is hidden from end users, as it encapsulates various HLS directives that do not require user modifications, thereby helping boost productivity by adding an abstraction layer to the HLS design flow. DP-HLS back-end also imposes design constraints on scoring and traceback stages to ensure optimal performance. Key optimizations (Fig. 2B–C, ①–⑩) guide the HLS compiler to exploit *wavefront (anti-diagonal) parallelism* (①–②) during score computation and inter-task parallelism in traceback (③–⑩), which are detailed below.

A. Scoring Logic Design

Fig. 2C illustrates the scoring operation using an example of a linear systolic array of four PEs ($N_{PE}=4$), which are used to compute the 2-D DP matrix. The rows of the DP matrix are divided into *chunks*, with each chunk containing N_{PE} consecutive rows, each assigned to a different PE (color-coded). During the processing of a query chunk, each PE is initialized with the query base-pair corresponding to its row in the chunk while the reference sequence streams through the array, leveraging wavefront parallelism. Chunk-wise score computation requires a buffer (*Preserved Row Score Buffer*) to store the scores computed by the last PE, which are then used by the first PE of the next query chunk. The buffer size corresponds to the maximum length of the reference sequence in the 2-D DP matrix.

In the back-end component, the 2-D DP matrix is computed using user-defined `PE_func` within nested `for` loops. The *outer loop* divides the matrix into chunks, the *middle loop* iterates through the wavefronts in this chunk, and the *inner loop* unrolls the PE function across PEs. Within the middle loop, the back-end applies optimization ① using `#pragma HLS PIPELINE` directive to enable wavefront pipelining, launching each wavefront every `Initialization Interval` (II) cycles (II=1 preferred). For complex PE functions requiring more than 1 cycle per wavefront, HLS selects the lowest valid II. Optimization ② uses `#pragma HLS UNROLL` in the inner loop to create a linear systolic array of PEs, which requires parallel access to inputs

and outputs. DP-HLS ensures this by fully partitioning local buffers using the `#pragma HLS ARRAY PARTITION variable=<buffer> type=complete` directive (③ and ④), which store the sequence characters at the current wavefront. This directive is also used in the optimization ⑤ for `DP Memory Buffer` and `Score Buffer`, which store the previous two wavefronts and the output scores of the current wavefront.

B. Traceback Logic and Memory Design

The optimal design of traceback logic and memory is essential for efficiency, as it typically consumes the most amount of memory resources in 2-D DP algorithms that require traceback. Below, we describe the two most critical optimizations implemented by the DP-HLS back-end component.

First, the back-end reshapes the 2-D DP matrix so that the first dimension corresponds to N_{PE} , while the second dimension is scaled to accommodate the total pointers for user-specified `MAX_REFERENCE_LENGTH` and `MAX_QUERY_LENGTH`. This ensures that each PE has access to a dedicated memory bank to independently store its traceback (TB) pointers every cycle, thereby minimizing the II of the wavefront loop. The back-end further applies address coalescing, which maps consecutive wavefronts in the DP matrix to consecutive TB memory columns (see ⑥ in Fig. 2). This ensures that all the PEs write their TB pointers to the same address in different memory banks. Additionally, the back-end keeps track of the corresponding addresses each wavefront shall write in the TB memory. This memory coalescing is one of the frequency-related optimizations performed by DP-HLS, since ensuring regular access patterns helped reduce routing complexity and meet timing constraints.

Second, the back-end allows users to configure the start and end conditions of the traceback. For traceback strategies that require locating the maximum scores in the last row or column of the DP matrix, each PE tracks its local maximum of the scores it computes if its coordinate satisfies the requirements (at the last row or column). Then, reduction logic is incorporated to identify the global maximum cell of the entire 2-D DP matrix block within a few cycles by a reduction maximum over each PE's local max.

C. Parallel Execution of Kernels

In DP-HLS, users can specify parallelism using three parameters: N_K , N_B , and N_{PE} (Section IV). The parallelism for N_{PE} is managed by the scoring logic optimizations discussed in Section V-A, while N_K is handled by the linker. To enable parallel processing of N_B blocks within a kernel, DP-HLS uses two main optimizations. *First*, it uses `#pragma HLS UNROLL` to create multiple blocks within a kernel, allowing for the concurrent execution of these blocks (⑦ in Fig. 2). *Second*, it ensures concurrent memory access to the input and output buffers for each block through block-wise partition, using `#pragma HLS ARRAY_PARTITION type=block dim=1`, with the number of partitions set to N_B (⑧ in Fig. 2).

Since optimal N_K , N_B , and N_{PE} values for a kernel depend on several post-synthesis factors, the DP-HLS codebase includes a script that performs an automatic grid search on these parameters and use synthesis results to identify the best configuration.

VI. METHODOLOGY

A. Datasets

DNA Sequences: The DNA sequences were generated using PBSIM2 [79] by simulating 1,000 PacBio [80] reads (10,000 bases, 30% error rate) from the human reference genome (GRCh38). Full reads were used for the tiled version of Kernel #2, #4, and #5, while reads were truncated to 256 bases for short alignment kernels (#1–#7, #10–#13).

Protein Sequences: Protein sequences of 256 residues for Kernel #15 were randomly sampled from UniProtKB (we used Swiss-Prot, version 2024_03, 571k entries) [81].

Complex Number Sequences: For DTW kernel (#9), we generated random complex-valued sequences of length 256.

sDTW Sequences: For the sDTW kernel (#14), we used 256-base sequences from SquiggleFilter dataset [67].

Sequence Profiles: For Kernel #8, profiles were derived from random 256-base pair regions across *Drosophila melanogaster* and *Drosophila simulans* genomes.

B. DP-HLS Framework Evaluation

Implementation: We implemented 15 DP-HLS kernels using C++ (v17) (Table I), verified the correctness using C++-based simulation and synthesized using AMD[®] Vitis HLS 2021.2 tool [77] on AWS[®] FPGA Developer AMI (v1.12.2). The design was then deployed on AWS[®] EC2 F1 instance (f1.2xlarge; with FPGA device XCVU9P-FLGB2104-2-I) using v++ command-line tools and OpenCL-based host code. For Kernel #2, #4, and #5, we applied host-side code changes to support long alignments using a tiling heuristic [11]. All kernels were set to a fixed target frequency before synthesis. Parameters N_{PE} , N_B , and N_K were configured for each kernel to maximize device throughput (Table II). Although our evaluations used the AWS[®] EC2 F1 FPGA, DP-HLS is compatible with all Vitis-supported FPGA devices, which we have confirmed through simulations and bitstream generation.

Throughput: Throughput for each DP-HLS kernel was estimated using the number of clock cycles from the co-simulation step in AMD[®] Vitis HLS, along with the maximum achievable frequency and the degree of parallelism on the FPGA. The co-simulation results accurately reflect the actual FPGA throughput (excluding host-FPGA transfer overhead)—this was confirmed by manually executing the bitstreams of three diverse kernels (#2, #12, and #14) on AWS EC2 F1 FPGAs.

Resource Utilization: Post-routing reports of the bitstream generation step are used to obtain the Block RAMs (**BRAM**), Flip-Flops (**FF**), Lookup Tables (**LUT**), and Digital Signal Processors (**DSP**) utilization on the FPGA device for all kernels reported as a percentage of the total resources available on the AWS[®] EC2 F1 FPGA device.

Scalability: To examine the scalability of the kernels, we chose two diverse kernels, Global Linear (#1) and DTW (#9), and evaluated their resource utilization and throughput values with increasing N_{PE} and N_B and fixed operating frequency of 250 and 200 MHz (Section VII-B).

Accuracy: We validated the accuracy and alignment scores produced by DP-HLS kernels, which were identical to software baseline implementations, as DP-HLS faithfully implements the underlying algorithms without introducing any approximations.

Energy-efficiency: The energy-efficiency of each tool was calculated in terms of alignments/J as the ratio of its throughput to power consumption. We estimated the total FPGA power consumption of DP-HLS kernels using the AMD[®] Power Design Manager tool.

C. Baseline Comparison

Software Baselines: We compared the cost-efficiency (in terms of alignments/\$ - derived from throughput normalized by hourly cost of AWS instances) of DP-HLS Kernels #1-7, #11-12, #15 with state-of-the-art parallel CPU implementations, using SeqAn3 [82] (v3.3.0), a widely-used, multi-threaded bioinformatics library, as the baseline. For kernels #5 and #15, we used Minimap2 [58] (v2.28) and EMBOSS Water [54] (v6.6.0) as our software baselines, tested using g++ (v10.3) and cmake (v3.16.3) on a 36-core CPU-optimized AWS[®] EC2 instance, c4.8xlarge (60 GB memory, \$1.591 per hour), comparable in cost to the AWS[®] EC2 F1 instance, f1.2xlarge (\$1.650 per hour) used for DP-HLS benchmarking. Throughput values for SeqAn3 and Minimap2 were measured with 36 threads and considering the alignment execution time, excluding I/O overhead (keeping it consistent with hardware baselines). Since EMBOSS lacks multi-threading, we measured its throughput of 36 parallel jobs launched with GNU parallel.

Hardware Baselines:

RTL Baselines: We obtained optimized open-source RTL implementations of GACT [11], Banded Smith-Waterman (BSW) [12] (v1), and SquiggleFilter [67] (v1.1.0) accelerators as RTL baselines to compare with Kernels #2, #12, and #14 (Table I), respectively, since all are based on linear systolic array architecture, similar to DP-HLS kernels. The baselines are implemented using AMD[®] Vivado 2021.2 tool on the same AWS[®] EC2 F1 FPGA instance (f1.2xlarge), and resource utilization numbers were collected from the *Implementation* step. Throughput values for the first two were measured via Icarus Verilog simulations and the BSW kernel via Vivado waveform simulations. To demonstrate the scaling effects, DP-HLS's Kernel #2 was compared with GACT with increasing N_{PE} . The match-bonus feature in SquiggleFilter was removed to match Kernel #14's implementation.

GPU Baselines: CUDASW++4.0 [83], which provides a GPU implementation of the Smith-Waterman algorithm for protein sequences, was used as the baseline for DP-HLS Kernel #15. We disabled the traceback step in DP-HLS since it is not performed in the baseline. We used GASAL2 [5],

with alignment type set as LOCAL, GLOBAL, BSW, as baselines for DP-HLS Kernels #4, #2, and #12, respectively.

Both baselines were evaluated on AWS[®] EC2 p3.2xlarge with an NVIDIA[®] Tesla V100 GPU (\$3.06/hour), targeting high-performance deployments, and on AWS[®] EC2 g4dn.2xlarge with an NVIDIA[®] Tesla T4 GPU (\$0.752/hour), targeting cost-effective scenarios. We additionally evaluated CUDASW++4.0 on AWS[®] EC2 p5.4xlarge with an NVIDIA[®] Hopper H100 GPU (costing \$7/hour), supporting DPX instructions, and compared it with DP-HLS Kernel #15. GASAL2 was excluded from H100 evaluation as it is not optimized for the latest DPX instructions. Cost-efficiency was compared by normalizing the throughput to instance cost, and energy-efficiency was measured using GPU power data from `nvidia-smi`.

HLS Baselines: As HLS baselines, we used the AMD[®] Vitis Genomics Library, containing optimized Vitis HLS Libraries [84]. We used the 2021.2 branch, which works with the Vitis HLS 2021.2 version used for DP-HLS implementation. The Smith-Waterman kernel in this library matches with DP-HLS Kernel #3 implementation, so we chose it as our HLS baseline, with $N_{PE}=32$, $N_K=1$, $N_B=32$ and the maximum target clock frequency of 333 MHz.

Comparison with Wavefront Algorithm: We also evaluated the FPGA performance of the Wavefront Alignment (WFA) algorithm [85], which, similar to the DP-HLS Kernel #2, performs global alignment of sequences. WFA, however, uses a wavefront propagation strategy that differs from the classical 2D DP paradigm and is not currently supported in DP-HLS. WFA performance data was taken from [13], and to account for FPGA device differences, we compared its throughput normalized to LUT usage with that of DP-HLS.

VII. RESULTS

A. DP-HLS efficiently implements diverse 2-D DP kernels

Table II summarizes the performance and resource utilization of all 15 DP-HLS kernels, which vary widely in terms of applications and computational patterns (Table I), also evident from the range of hardware resource utilization and throughput values presented in Table II.

For instance, when comparing the resource utilization of a single block of 32 PEs, Profile Alignment (#8) and DTW (#9) kernels have relatively higher DSP consumption. This is expected as these kernels perform multiplications, with Kernel #8 requiring two matrix-vector multiplications in each cell. For most other kernels, the DSP is only used for pre-computing traceback starting addresses.

The BRAM usage is primarily influenced by the complexity of traceback across most kernels. For instance, Kernels #5 and #13, which implement a two-piece affine gap penalty, require more BRAM as they need at least 7 bits to store each pointer, compared to only 2 bits per pointer for kernels with a linear penalty (e.g., Kernels #1 and #3). In Kernels #10 and #12, BRAM usage is minimal since traceback is not involved. Kernel #15, which deals with protein alignment, also consumes more BRAM to store the larger substitution

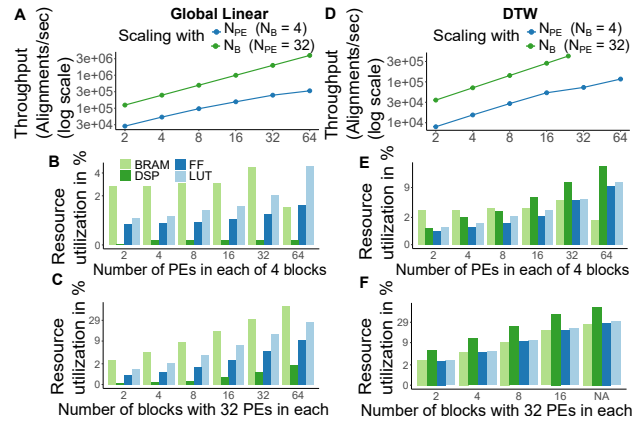


Fig. 3: Scaling results for Global Linear (#1) and DTW (#9) kernels with N_{PE} and N_B . (A, D) Throughput scaling of Global Linear (A) and DTW (D) in \log - \log scale with N_{PE} and N_B ; (B-C) Resource utilization scaling for Global Linear kernel; (E-F) Resource utilization scaling for DTW kernel.

matrix in `ScoringParams` (20×20 for protein sequences, compared to 4×4 for DNA sequences).

LUT and FF usage are mainly influenced by the complexity of the scoring equations. For instance, Kernel #8, with its complex matrix-vector multiplication, shows the highest FF and LUT utilization, followed by the Viterbi kernel (#10). Banded kernels (#11-13) also have slightly elevated logic usage due to extra compute needed to determine the bands.

In terms of throughput, resource-intensive kernels (#8-10) have relatively lower values due to their complex computational patterns. For instance, Kernel #8 requires multiple cycles ($II=4$) to compute a single 2-D DP cell due to matrix-vector multiplications. The complexity of the scoring equations also impacts clock frequency, as seen in the Viterbi (#10) and Banded Global Two-piece Affine (#13) kernels. Overall, the resource utilization, clock frequency, and throughput numbers suggest that all kernels have been efficiently implemented by DP-HLS, which is also confirmed by the baseline comparisons.

B. DP-HLS kernel implementations demonstrate 1-D systolic array behavior

Though the DP-HLS back-end optimizations serve as hints to the HLS compiler to produce N_B 1-D systolic arrays, each with N_{PE} processing elements, the compiler may apply alternative hardware mappings that it may explore and find more efficient. Since HLS-generated code is not human-interpretable, it is difficult to confirm the mapping directly from the code. Hence, we varied N_{PE} and N_B and checked if the throughput and resources scaled according to the expected behavior of 1-D systolic arrays. Fig. 3 presents these results for two diverse kernels, Global Linear (#1) and DTW (#9), but we observed similar patterns across all 15 kernels.

Fig. 3A,D shows that throughput scales nearly perfectly with N_{PE} at lower values for both kernels, but experiences some saturation at higher values. This is expected as the wavefront parallelism exploited by systolic arrays diminishes near

TABLE II: **Performance summary of 15 DP-HLS Kernels.** The kernel numbers correspond to the kernels listed in Table I. Utilization in % of available FPGA resources is shown for a single block for 32 PEs for uniformity. We also show the optimal configuration of (N_{PE}, N_B, N_K) for each kernel that resulted in maximum throughput (alignments/sec), and its corresponding maximum frequency and throughput achieved, along with the total power consumed (watt) and energy-efficiency (alignments/J).

Kernel No.	Resource utilization for 32PE				Optimal (N_{PE}, N_B, N_K)	Max Freq (MHz)	Alignments /sec	Total Power (watt)	Alignments /J
	LUT	FF	BRAM	DSP					
#1	0.72%	0.42%	1.78%	0.029%	64,16,4	250.0	3.51e6	17.6	2.0e5
#2	1.30%	0.517%	1.78%	0.029%	32,16,4	250.0	2.85e6	14	2.03e5
#3	0.95%	0.63%	1.67%	0.014%	32,16,5	250.0	3.43e6	16.4	2.09e5
#4	1.60%	0.75%	1.67%	0.014%	32,16,4	250.0	2.71e6	19.2	1.41e5
#5	2.03%	0.65%	2.67%	0.029%	32,8,5	150.0	1.06e6	10.73	9.87e4
#6	0.98%	0.66%	1.67%	0.014%	32,16,4	250.0	2.73e6	14.9	1.83e5
#7	1.17%	0.67%	0.83%	0.014%	32,16,4	250.0	3.34e6	17.54	1.90e5
#8	3.66%	2.56%	2.56%	28.11%	16,1,5	166.7	3.70e4	9.18	4.02e3
#9	1.62%	1.55%	1.88%	2.84%	64,4,3	200.0	2.31e5	15.47	1.49e4
#10	3.78%	1.69%	1.67%	0.014%	16,4,7	125.0	4.90e5	8.15	6.01e4
#11	1.02%	0.40%	0.94%	0.029%	64,8,7	166.7	2.25e6	11.6	1.93e5
#12	1.44%	0.70%	0.57%	0.014%	16,16,7	200.0	4.77e6	14.4	3.31e5
#13	2.25%	0.69%	1.83%	0.029%	16,8,7	125.0	1.24e6	7.6	1.63e5
#14	1.22%	0.76%	0.57%	0.014%	32,16,5	250.0	5.16e6	17.67	2.92e5
#15	1.47%	0.95%	2.56%	0.014%	32,8,5	200.0	9.33e5	11.96	7.80e4

the edges of the DP matrix, leading to more PEs with more idle cycles. In contrast, the large inter-alignment parallelism exploited by the N_B independent arrays allows throughput to scale almost perfectly with N_B , which is confirmed by the results for both kernels in Fig. 3. For DTW, N_B is capped at 24 as it reached maximum DSP availability.

Also, the percentage utilization of all resource types scales almost perfectly with increasing N_B while keeping N_{PE} constant (Fig. 3C, F). This occurs because each parallel block is identical, leading to a proportional increase in resource usage. When N_B is fixed, LUT and FF utilization scales perfectly with increasing N_{PE} for both kernels (Fig. 3B, E) due to the linear systolic array structure. However, DSP usage varies depending on the kernel’s algorithm. For instance, DSP utilization scales well with increasing N_{PE} in the DTW kernel (Fig. 3E) as DSPs are used by each PE for the scoring logic, whereas for the Global Linear kernel, DSP usage remains constant (Fig. 3B) since DSPs are used for fixed logic outside the PEs to pre-compute the traceback starting address. Additionally, BRAM is primarily used for TB memory, which increases with N_{PE} up to 32 but does not scale proportionally. For higher N_{PE} values (e.g., $N_{PE}=64$), HLS compiler optimizations sometimes convert BRAMs to LUTRAMs to reduce memory access latency, resulting in lower BRAM usage.

C. DP-HLS kernels provide competitive performance to optimized RTL implementations

We compared three DP-HLS kernels with their RTL implementations (Fig. 4A-C), and observed that DP-HLS achieves lower but competitive throughput to the baselines. Specifically, the DP-HLS throughput was within 7.7%, 16.8%, and 8.16% of the baseline for Global Affine, Banded Local Affine, and sDTW kernels, respectively. This is not surprising because, with added flexibility and programming ease, the DP-HLS framework misses out on some optimization opportunities that RTL implementations leverage. For example, all RTL

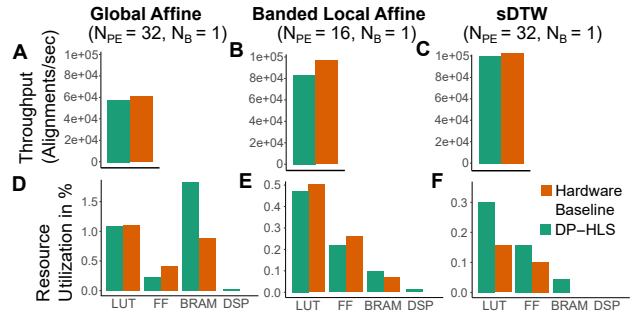


Fig. 4: **Comparison of DP-HLS kernels (#2, #12, #14) with hardware baselines.** (A-C) Throughput, and (D-F) Resource utilization comparison of Kernel #2, #12, #14 with GACT [11], BSW [12], and SquiggleFilter [67], respectively.

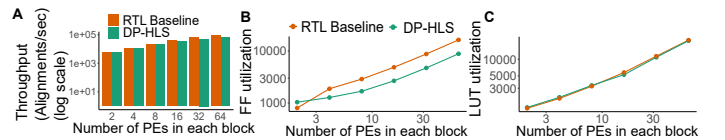


Fig. 5: **Scaling comparison of DP-HLS kernel #2 with hardware baseline (with increasing N_{PE} , $N_B=1$).** (A) Throughput comparison (in log-log scale), and (B-C) FF and LUT scaling of kernel #2 with its hardware baseline, GACT.

implementations overlap query reads and 2-D DP matrix initialization with computation, but these steps are currently performed sequentially in DP-HLS. This overhead is even more apparent in the Banded Local Affine kernel, as it does not employ traceback. The relative throughput of the Global Affine kernel versus GACT remained consistent for long alignments, as both approaches use the same number of tiles.

The resource utilization comparison is more nuanced. For kernel #2, DP-HLS shows comparable LUT and FF usage compared to GACT (Fig. 4D). This is also reflected in the

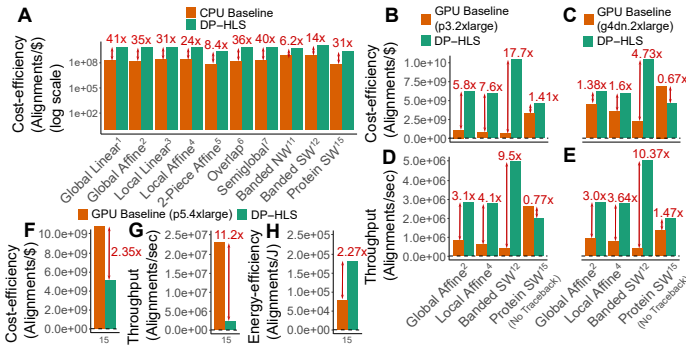


Fig. 6: DP-HLS kernels performance compared to CPU and GPU baselines. (A) CPU cost-efficiency (in \log scale) and (B-C, F) GPU cost-efficiency, (D-E, G) throughput, and (H) energy-efficiency of GPU baselines compared to DP-HLS kernels. Kernels indices follow Table I.

scaling behavior, where throughput remains similar (Fig. 5A), and the resource usage difference stays constant (Fig. 5B-C). DP-HLS also uses DSPs for pre-computing traceback starting addresses, unlike the baselines, but this does not affect scaling with N_{PE} . Despite this, DP-HLS has slightly better LUT and FF utilization on the Banded Local Affine kernel (Fig. 4E).

DP-HLS kernel #2, although faster than the WFA software (Section VII-D), is $10.6\times$ (for high error rates)- $21.3\times$ (for low error rates) less efficient than the WFA FPGA baseline [13] in terms of iso-LUT alignment throughput, since the WFA algorithm is inherently faster than the classical DP, as it skips computing many DP matrix cells while ensuring optimality.

Overall, DP-HLS kernels demonstrate efficient resource scaling and competitive throughput compared to optimized RTL designs, with acceptable trade-offs for each kernel.

D. DP-HLS kernels provide 1.38–41 \times higher cost-efficiency than CPU and GPU baselines

We performed the cost-efficiency comparison of DP-HLS kernels #1-4, #6-7, #11-12 (Table I) and observed that DP-HLS outperforms SeqAn3 by $6.2\times$ to $41\times$ (Fig. 6A). Interestingly, the baseline throughput shows minor variability across these kernels, as SeqAn3 uses the same underlying software implementation across kernels with minor adjustments. In contrast, DP-HLS applies various architectural optimizations tailored to each kernel, maximizing throughput by fitting multiple parallel kernels on the FPGA. For Kernels #5 and #15, which are computationally intensive, DP-HLS achieves higher relative cost-efficiency, $8.4\times$, and $31\times$, respectively (Fig. 6). This boost is likely due to DP-HLS’s use of arbitrary precision values and customized hardware data paths, enabling higher throughput than general CPU-based implementations. Compared to the software implementation of WFA [8], DP-HLS achieves $36\times$ higher throughput for Kernel #2 (Fig. 6A).

Surprisingly, the cost-efficiency of DP-HLS was better even when compared to GPU baselines on AWS EC2 p3.2xlarge instances— $1.41\times$ higher than CUDASW++ and $5.83\text{--}17.72\times$ higher than GASAL2 (Fig. 6B). Since GASAL2

codebase has not been updated recently, it might have lagged behind newer CPU implementations, particularly in SeqAn3. On the more cost-effective g4dn.2xlarge (NVIDIA Tesla T4), DP-HLS remains $1.38\text{--}4.72\times$ more cost-efficient than GASAL2 and CUDASW++, except for #15, where CUDASW++ outperformed DP-HLS by $1.49\times$ (Fig. 6C), but at a 1.47-fold lower raw throughput than DP-HLS. CUDASW++ maintains higher cost-efficiency than DP-HLS’s #15 on p5.4xlarge instance, which includes a modern NVIDIA Hopper H100 GPU with DPX instructions. However, the Hopper GPU consumes significant power (25.5-fold higher than the FPGA), and thus, despite DP-HLS running on an FPGA three generations older in transistor technology than Hopper (14 nm vs. 5 nm), it achieves $2.27\times$ higher energy-efficiency (Fig. 6G-H). These results indicate that despite several optimizations in modern GPUs, FPGAs still offer a superior throughput-efficiency trade-off for 2-D DP algorithms.

E. DP-HLS kernel outperforms a previous HLS baseline

We compared Kernel #3 with the HLS implementation of the Smith-Waterman algorithm and found that DP-HLS achieved 32.6% higher throughput than the HLS baseline. This difference could be explained by two factors. *First*, the HLS baseline uses streaming functions to transfer some data between the host and device, for which the DP-HLS kernels use device memory. *Second*, DP-HLS backend adds more extensive optimization hints to the compiler than the baseline, which results in better throughput than the baseline. We also note that it is significantly more challenging for new users to modify the HLS baseline to implement new 2-D DP kernels, as the HLS compiler directives (*pragmas*) are interleaved with the code that needs to be changed, resulting in a steeper learning curve compared to DP-HLS.

F. DP-HLS can significantly improve the productivity of implementing new 2-D DP kernels

In addition to providing efficient FPGA implementations for 15 bioinformatically relevant 2-D DP kernels, many of which lacked existing hardware implementations, the DP-HLS framework was developed to simplify the creation and deployment of new 2-D DP kernels. In our experience, setting up the initial DP-HLS framework took several months, as matching hand-tuned RTL performance required careful directive placement and navigating HLS tool limitations like restricted memory control, parallelism, and certain non-intuitive, tool-specific optimizations. However, once we encapsulated the required directives in a common back-end, we were able to implement, test, and deploy new 2-D DP kernels on AWS[®] EC2 F1 instances in just 2–4 days using only the front-end. This typically involved modifying only tens of lines of code at the algorithmic level (C/C++) in DP-HLS front-end. For example, modifying the baseline kernel #1 to kernel #4 required just 67 lines of C++ code changes in DP-HLS, whereas an open-source handwritten RTL baseline (GACT [11]) for kernel #4 contains 1057 lines. Moreover, DP-HLS automatically generates the memory interface and host communication code, avoiding the

need for manual implementation and low-level debugging, which can require an additional hundreds of lines of RTL code. Additionally, manual RTL design also necessitates the time-consuming steps of hardware verification, optimization, and synthesis, which often span weeks to months of effort. This stark contrast illustrates how DP-HLS, which separates high-level algorithmic specifications from low-level hardware optimizations, can significantly enhance design productivity.

G. DP-HLS supports tiling for long-sequence alignments

The software-level tiling approach [11] that was used to compare kernel #2 with GACT on long-read alignments can be applied to any DP-HLS kernel to enable long-sequence alignment. In this approach, the DP matrix is split into small overlapping tiles processed sequentially, enabling long-sequence alignment within limited on-chip memory while maintaining similar resource utilization and speedup relative to software as short, fixed-size alignments. To demonstrate generality, we performed long-read alignments for two additional kernels (#4, #5), and observed negligible overhead - resource utilization was comparable to the short-alignment case, and the number of tiles processed per second closely matched the alignment throughput of the corresponding short-alignment kernels.

VIII. LIMITATIONS

DP-HLS focuses on mapping 2-D DP algorithms, one of the most common computational patterns in bioinformatics, to linear systolic arrays. However, some emerging algorithms, such as graph alignments, WFA, and A*PA [86]–[88], involve irregular 2-D DP patterns and even non-2-D DP structures, which DP-HLS does not currently support. Our future work will involve generalizing DP-HLS to support a broader class of DP algorithms and hardware primitives, including those supporting irregular parallelism and memory access patterns.

IX. RELATED WORK

A. Dynamic Programming in Bioinformatics

Dynamic programming (DP), introduced by Bellman in the 1950s [26], is a foundational algorithmic paradigm in bioinformatics. Early applications include the Needleman-Wunsch algorithm for global alignment [70], followed by local pairwise alignments [28], multiple sequence alignment [29], [30], homology searches [31], [32], whole-genome alignments [33], basecalling [34], variant calling [35], and pangenomics [89], [90]. Given its computationally intensive nature and broad applicability, NVIDIA[®] introduced DPX instructions in Hopper GPUs to accelerate DP algorithms [37]. DP-HLS similarly modularizes 2-D DP algorithmic patterns in an HLS framework, allowing users to easily customize FPGA-based kernels for various bioinformatics applications.

B. Systolic Array Accelerators for Bioinformatics Applications

In recent years, many linear systolic array-based FPGA and ASIC accelerators have been proposed for different bioinformatics applications [10]–[25]. Examples include GenASM, which accelerates the approximate string matching problem

using the Bitap algorithm [24], SquiggleFilter, for genomic surveillance application [67], and Darwin-WGA, which accelerates the X-Drop algorithm for whole-genome alignments [12]. Although these accelerators provide huge speedups for specific steps in genomic data analysis, they typically represent a single design point in the 2-D DP space and cannot be easily refactored into other 2-D DP kernels due to their custom low-level HDL implementation. On the other hand, DP-HLS framework aims to accelerate the development of 2-D DP kernels using HLS, while providing competitive performance and similar resources to hand-crafted HDL designs.

A recent work, GenDP [91], shares similarities with the DP-HLS framework, as it also recognizes that many bioinformatics tasks rely on DP algorithms that are well-suited for systolic array implementations and provides a programmable framework to accelerate these tasks. However, GenDP's programmability operates at the software level, using an instruction set and a custom dataflow graph compilation for its PEs. Although GenDP's RTL code is not available for a fair comparison, software programmable solutions typically incur significant overheads in fetching and decoding instructions [92], [93]. In contrast, DP-HLS takes advantage of the hardware re-configurability of FPGAs, providing an optimized hardware implementation for every kernel. DP-HLS kernels can be readily and efficiently deployed on cloud platforms with FPGA support, like AWS[®]. We also note that commercial bioinformatics accelerators, such as the Illumina's DRAGEN[™] Bio-IT Processor[®] [40], are based on FPGAs, where DP-HLS could be used for providing greater flexibility and productivity.

C. The Emergence of HLS Tools and Frameworks

High-Level Synthesis (HLS) allows users to design custom hardware using high-level languages, greatly reducing hardware development time, and thus widely adopted across various domains [94], [95], including machine learning [42], [43], cryptography [44], [45], and bioinformatics [46]–[48], [96]. However, most previous works target a single or a handful of kernels, offering limited flexibility. An exception in bioinformatics is the work of Benkrid et al. [96], who used Handel-C to develop a flexible systolic array for pairwise sequence alignment, though limited to a few alignment types (global, local, and overlap). While we could not directly compare to Benkrid et al. [96] as their codebase is not actively maintained, DP-HLS offers much greater flexibility within the 2-D DP paradigm, supporting customization of scoring schemes, input characters, traceback logic, and more.

Some frameworks, such as ScaleHLS [97], HeteroCL [98], and SuSy [99], add prior high-level abstractions through new domain-specific languages (DSLs) along with custom compilation flows based on multi-level intermediate representation (MLIR) [100]. DP-HLS is fundamentally different, since it adds an abstraction layer using an existing compiler (Vitis HLS) and C/C++, lowering the learning curve and simplifying deployment. Moreover, these frameworks do not support the rich 2-D DP algorithmic variants common in bioinformatics. For example, while SuSy introduces a tensor abstraction for

2-D DP targeting systolic arrays, it does not support traceback, multiple affine layers, complex alphabets, and banding.

X. CONCLUSION AND FUTURE WORK

We introduce DP-HLS, a novel HLS-based framework designed for FPGA acceleration of 2-D DP algorithms that are widely used in bioinformatics applications. DP-HLS provides a software-like design experience for hardware, thus empowering broad users, even those lacking hardware design experience, to quickly and flexibly develop efficient hardware solutions for 2-D DP algorithms. DP-HLS does this by introducing an abstraction layer in HLS design, which leverages the fact that a broad range of 2-D DP algorithms can efficiently map to the same hardware primitive - linear systolic arrays. DP-HLS is deployed on the AWS[®] cloud platform and achieves competitive throughput and similar resource utilization to hand-crafted RTL. In the future, we plan to implement HLS frameworks for other algorithmic classes for AI and signal processing applications, using similar strategies.

ACKNOWLEDGMENTS

We thank the AMD-omics group for helpful feedback. Research reported in this manuscript was supported by an Amazon Research Award (Fall 2022 CFP), AMD AI & HPC Fund, and the Hellman Fellowship.

REFERENCES

- [1] Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, C. Zhai, M. J. Efron, R. Iyer, M. C. Schatz, S. Sinha, and G. E. Robinson, "Big data: astronomical or genomics?" *PLoS biology*, vol. 13, no. 7, p. e1002195, 2015.
- [2] J. Lindegger, D. Senol Cali, M. Alser, J. Gómez-Luna, N. M. Ghiasi, and O. Mutlu, "Scrooge: a fast and memory-frugal genomic sequence aligner for cpus, gpus, and asics," *Bioinformatics*, vol. 39, no. 5, p. btad151, 2023.
- [3] N. Ahmed, T. D. Qiu, K. Bertels, and Z. Al-Ars, "Gpu acceleration of darwin read overlapper for de novo assembly of long dna reads," *BMC bioinformatics*, vol. 21, pp. 1–17, 2020.
- [4] A. Zeni, G. Guidi, M. Ellis, N. Ding, M. D. Santambrogio, S. Hofmeyr, A. Buluç, L. Oliker, and K. Yelick, "Logan: High-performance gpu-based x-drop long-read alignment," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020, pp. 462–471.
- [5] N. Ahmed, J. Lévy, S. Ren, H. Mushtaq, K. Bertels, and Z. Al-Ars, "Gasal2: a gpu accelerated sequence alignment library for high-throughput ngs data," *BMC bioinformatics*, vol. 20, pp. 1–20, 2019.
- [6] S. D. Goenka, Y. Turakhia, B. Paten, and M. Horowitz, "Segalign: A scalable gpu-based whole genome aligner," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–13.
- [7] S. Park, J. Hong, J. Song, H. Kim, Y. Kim, and J. Lee, "Agatha: Fast and efficient gpu acceleration of guided sequence alignment for long read mapping," in *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, 2024, pp. 431–444.
- [8] Q. Aguado-Puig, S. Marco-Sola, J. C. Moure, C. Matzoros, D. Castells-Rufas, A. Espinosa, and M. Moreto, "Wfa-gpu: Gap-affine pairwise alignment using gpus," *bioRxiv*, pp. 2022–04, 2022.
- [9] A. Zeni, S. Onken, M. D. Santambrogio, and M. Samadi, "Leveraging difference recurrence relations for high-performance gpu genome alignment," in *Proceedings of the 2024 International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 133–143. [Online]. Available: <https://doi.org/10.1145/3656019.3676894>

- [10] X. Fei, Z. Dan, L. Lina, M. Xin, and Z. Chunlei, "Fpgasw: accelerating large-scale smith-waterman sequence alignment application with backtracking on fpga linear systolic array," *Interdisciplinary Sciences: Computational Life Sciences*, vol. 10, pp. 176–188, 2018.
- [11] Y. Turakhia, G. Bejerano, and W. J. Dally, "Darwin: A genomics co-processor provides up to 15,000 x acceleration on long read assembly," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 199–213, 2018.
- [12] Y. Turakhia, S. D. Goenka, G. Bejerano, and W. J. Dally, "Darwin-wga: A co-processor provides increased sensitivity in whole genome alignments with high speedup," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 359–372.
- [13] A. Haghi, S. Marco-Sola, L. Alvarez, D. Diamantopoulos, C. Hagleitner, and M. Moreto, "An fpga accelerator of the wavefront algorithm for genomics pairwise alignment," in *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2021, pp. 151–159.
- [14] P. Zhang, G. Tan, and G. R. Gao, "Implementation of the smith-waterman algorithm on a reconfigurable supercomputing platform," in *Proceedings of the 1st international workshop on High-performance reconfigurable computing technology and applications: held in conjunction with SC07*, 2007, pp. 39–48.
- [15] R.-T. Chien, Y.-L. Liao, C.-A. Wang, Y.-C. Li, and Y.-C. 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)*, 2018, pp. 1–5.
- [16] P. Chen, C. Wang, X. Li, and X. Zhou, "Hardware acceleration for the banded smith-waterman algorithm with the cycled systolic array," in *2013 International Conference on Field-Programmable Technology (FPT)*, 2013, pp. 480–481.
- [17] D. Fujiki, S. Wu, N. Ozog, K. Goliya, D. Blaauw, S. Narayanasamy, and R. Das, "Seedex: A genome sequencing accelerator for optimal alignments in subminimal space," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 937–950.
- [18] J.-P. Wu, Y.-C. Lin, Y.-W. Wu, S.-W. Hsieh, C.-H. Tai, and Y.-C. Lu, "A memory-efficient accelerator for dna sequence alignment with two-piece affine gap tracebacks," in *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2021, pp. 1–4.
- [19] M.-J. Lin, Y.-C. Li, and Y.-C. Lu, "Hardware accelerator design for dynamic-programming-based protein sequence alignment with affine gap tracebacks," in *2019 IEEE Biomedical Circuits and Systems Conference (BioCAS)*, 2019, pp. 1–4.
- [20] E. J. Houtgast, V.-M. Sima, K. Bertels, and Z. Al-Ars, "An fpga-based systolic array to accelerate the bwa-mem genomic mapping algorithm," in *2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, 2015, pp. 221–227.
- [21] T. Oliver, B. Schmidt, D. Maskell, D. Nathan, and R. Clemens, "Multiple sequence alignment on an fpga," in *11th International Conference on Parallel and Distributed Systems (ICPADS'05)*, vol. 2, 2005, pp. 326–330.
- [22] S. Huang, G. J. Manikandan, A. Ramachandran, K. Rupnow, W.-m. W. Hwu, and D. Chen, "Hardware acceleration of the pair-hmm algorithm for dna variant calling," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 275–284.
- [23] S. Walia, C. Ye, A. Bera, D. Lodhavia, and Y. Turakhia, "Talco: Tiling genome sequence alignment using convergence of traceback pointers," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2024, pp. 91–107.
- [24] D. S. Cali, G. S. Kalsi, Z. Bingöl, C. Firtina, L. Subramanian, J. S. Kim, R. Ausavarungnirun, M. Alser, J. Gomez-Luna, A. Boroumand, A. Nori, A. Scibisz, S. Subramoney, C. Alkan, S. Ghose, and O. Mutlu, "GenASM: A High-Performance, Low-Power Approximate String Matching Acceleration Framework for Genome Sequence Analysis," Sep. 2020, arXiv:2009.07692 [cs, q-bio]. [Online]. Available: <http://arxiv.org/abs/2009.07692>
- [25] D. S. Cali, K. Kanellopoulos, J. Lindegger, Z. Bingöl, G. S. Kalsi, Z. Zuo, C. Firtina, M. B. Cavlak, J. Kim, N. M. Ghiasi *et al.*, "Segram: A universal hardware accelerator for genomic sequence-to-graph and sequence-to-sequence mapping," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 638–655.

- [26] R. Bellman, "Dynamic programming," *science*, vol. 153, no. 3731, pp. 34–37, 1966.
- [27] S. R. Eddy, "What is dynamic programming?" *Nature biotechnology*, vol. 22, no. 7, pp. 909–910, 2004.
- [28] T. F. Smith, M. S. Waterman *et al.*, "Identification of common molecular subsequences," *Journal of molecular biology*, vol. 147, no. 1, pp. 195–197, 1981.
- [29] F. Sievers, A. Wilm, D. Dineen, T. J. Gibson, K. Karplus, W. Li, R. Lopez, H. McWilliam, M. Remmert, J. Söding *et al.*, "Fast, scalable generation of high-quality protein multiple sequence alignments using clustal omega," *Molecular systems biology*, vol. 7, no. 1, p. 539, 2011.
- [30] R. C. Edgar, "Muscle: a multiple sequence alignment method with reduced time and space complexity," *BMC bioinformatics*, vol. 5, pp. 1–19, 2004.
- [31] S. R. Eddy, "Profile hidden markov models." *Bioinformatics (Oxford, England)*, vol. 14, no. 9, pp. 755–763, 1998.
- [32] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of molecular biology*, vol. 215, no. 3, pp. 403–410, 1990.
- [33] R. S. Harris, *Improved pairwise alignment of genomic DNA*. The Pennsylvania State University, 2007.
- [34] J. T. Simpson, R. E. Workman, P. Zuzarte, M. David, L. Dursi, and W. Timp, "Detecting dna cytosine methylation using nanopore sequencing," *Nature methods*, vol. 14, no. 4, pp. 407–410, 2017.
- [35] R. Nielsen, T. Korneliussen, A. Albrechtsen, Y. Li, and J. Wang, "Snp calling, genotype calling, and sample allele frequency estimation from new-generation sequencing data," 2012.
- [36] A. Subramaniyan, Y. Gu, T. Dunn, S. Paul, M. Vasimuddin, S. Misra, D. Blaauw, S. Narayanasamy, and R. Das, "Genomicsbench: A benchmark suite for genomics," in *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2021, pp. 1–12.
- [37] A. C. Elster and T. A. Haugdahl, "Nvidia hopper gpu and grace cpu highlights," *Computing in Science & Engineering*, vol. 24, no. 2, pp. 95–100, 2022.
- [38] R. J. Lipton and D. Lopresti, "A systolic array for rapid string comparison," in *Proceedings of the Chapel Hill conference on VLSI*. Chapel Hill NC, 1985, pp. 363–376.
- [39] S. Behera, S. Catreux, M. Rossi, S. Truong, Z. Huang, M. Ruehle, A. Visvanath, G. Parnaby, C. Roddey, V. Onuchic *et al.*, "Comprehensive genome analysis and variant detection at scale using dragen," *Nature Biotechnology*, pp. 1–15, 2024.
- [40] A. Goyal, H. J. Kwon, K. Lee, R. Garg, S. Y. Yun, Y. H. Kim, S. Lee, and M. S. Lee, "Ultra-fast next generation human genome sequencing data processing using dragentm bio-it processor for precision medicine," *Open Journal of Genetics*, vol. 7, no. 1, pp. 9–19, 2017.
- [41] TimeLogic® biocomputing solutions. [Online]. Available: <https://www.activemotif.com/catalog/75/timelogic-biocomputing-solutions>
- [42] F. Fahim, B. Hawks, C. Herwig, J. Hirschauer, S. Jindariani, N. Tran, L. P. Carloni, G. Di Guglielmo, P. Harris, J. Krupa, D. Rankin, M. B. Valentin, J. Hester, Y. Luo, J. Mamish, S. Orgrenci-Memik, T. Aarrestad, H. Javed, V. Loncar, M. Pierini, A. A. Pol, S. Summers, J. Duarte, S. Hauck, S.-C. Hsu, J. Ngadiuba, M. Liu, D. Hoang, E. Kreinar, and Z. Wu, "hls4ml: An Open-Source Codesign Workflow to Empower Scientific Low-Power Machine Learning Devices," Mar. 2021, arXiv:2103.05579 [physics]. [Online]. Available: <http://arxiv.org/abs/2103.05579>
- [43] M. Shahshahani, B. Khabbazan, M. Sabri, and D. Bhatia, "A Framework for Modeling, Optimizing, and Implementing DNNs on FPGA Using HLS," in *2020 IEEE 14th Dallas Circuits and Systems Conference (DCAS)*. Dallas, TX, USA: IEEE, Nov. 2020, pp. 1–6.
- [44] A. Barengi, M. Madaschi, N. Mainardi, and G. Pelosi, "OpenCL HLS Based Design of FPGA Accelerators for Cryptographic Primitives," in *2018 International Conference on High Performance Computing & Simulation (HPCS)*. Orleans: IEEE, Jul. 2018, pp. 634–641.
- [45] E. Homsirikamol and K. G. George, "Toward a new HLS-based methodology for FPGA benchmarking of candidates in cryptographic competitions: The CAESAR contest case study," in *2017 International Conference on Field Programmable Technology (ICFPT)*. Melbourne, VIC: IEEE, Dec. 2017, pp. 120–127.
- [46] D. Castells-Rufas, S. Marco-Sola, J. C. Moure, Q. Aguado, and A. Espinosa, "Fpga acceleration of pre-alignment filters for short read mapping with hls," *IEEE Access*, vol. 10, pp. 22 079–22 100, 2022.
- [47] K. Liyanage, H. Gamaarachchi, R. Ragel, and S. Parameswaran, "Cross layer design using hw/sw co-design and hls to accelerate chaining in genomic analysis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 9, pp. 2924–2937, 2023.
- [48] P. Meng, M. Jacobsen, M. Kimura, V. Dergachev, T. Anantharaman, M. Requa, and R. Kastner, "Hardware accelerated novel optical de novo assembly for large-scale genomes," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, 2014, pp. 1–8.
- [49] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-I. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, Z. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snellman, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1–12. [Online]. Available: <https://doi.org/10.1145/3079856.3080246>
- [50] N. Srivastava, H. Jin, S. Smith, H. Rong, D. Albonese, and Z. Zhang, "Tensoraurus: A versatile accelerator for mixed sparse-dense tensor computations," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 689–702.
- [51] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 58–70.
- [52] H. Kung, B. McDanel, and S. Q. Zhang, "Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 821–834. [Online]. Available: <https://doi.org/10.1145/3297858.3304028>
- [53] H. Genc, S. Kim, A. Amid, A. Haj-Ali, V. Iyer, P. Prakash, J. Zhao, D. Grubb, H. Liew, H. Mao, A. Ou, C. Schmidt, S. Steffl, J. Wright, I. Stoica, J. Ragan-Kelley, K. Asanovic, B. Nikolic, and Y. S. Shao, "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 769–774.
- [54] F. Madeira, N. Madhusoodanan, J. Lee, A. Eusebi, A. Niewielska, A. R. N. Tivey, R. Lopez, and S. Butcher, "The embl-ebi job dispatcher sequence analysis tools framework in 2024," *Nucleic acids research*, p. gkae241, April 2024. [Online]. Available: <https://academic.oup.com/nar/advance-article-pdf/doi/10.1093/nar/gkae241/57199791/gkae241.pdf>
- [55] W. R. Pearson, "Using the fasta program to search protein and dna sequence databases," *Computer Analysis of Sequence Data: Part I*, pp. 307–331, 1994.
- [56] W. J. Kent, "Blat—the blast-like alignment tool," *Genome research*, vol. 12, no. 4, pp. 656–664, 2002.
- [57] R. S. Harris, *Improved pairwise alignment of genomic DNA*. The Pennsylvania State University, 2007.
- [58] H. Li, "Minimap2: Pairwise alignment for nucleotide sequences," *Bioinformatics*, vol. 34, no. 18, pp. 3094–3100, Sep. 2018, arXiv: 1708.01492 Publisher: Oxford University Press.
- [59] S. Koren, B. P. Walenz, K. Berlin, J. R. Miller, N. H. Bergman, and A. M. Phillippy, "Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation," *Genome research*, vol. 27, no. 5, pp. 722–736, 2017.
- [60] M. Kolmogorov, J. Yuan, Y. Lin, and P. A. Pevzner, "Assembly of long, error-prone reads using repeat graphs," *Nature biotechnology*, vol. 37, no. 5, pp. 540–546, 2019.

- [61] H. Li, "Aligning sequence reads, clone sequences and assembly contigs with bwa-mem," *arXiv preprint arXiv:1303.3997*, 2013.
- [62] J. D. Thompson, D. G. Higgins, and T. J. Gibson, "CLUSTAL W: Improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice," *Nucleic Acids Research*, vol. 22, no. 22, pp. 4673–4680, Nov. 1994.
- [63] J. M. Ferguson and M. A. Smith, "SquiggleKit: a toolkit for manipulating nanopore signal data," *Bioinformatics*, vol. 35, no. 24, pp. 5372–5373, 07 2019. [Online]. Available: <https://doi.org/10.1093/bioinformatics/btz586>
- [64] R. D. Finn, J. Clements, and S. R. Eddy, "HMMER web server: Interactive sequence similarity searching," *Nucleic Acids Research*, vol. 39, no. Web Server issue, pp. W29–W37, Jul. 2011.
- [65] M. Stanke and B. Morgenstern, "Augustus: a web server for gene prediction in eukaryotes that allows user-defined constraints," *Nucleic acids research*, vol. 33, no. suppl_2, pp. W465–W467, 2005.
- [66] B. Langmead and S. L. Salzberg, "Fast gapped-read alignment with bowtie 2," *Nature methods*, vol. 9, no. 4, pp. 357–359, 2012.
- [67] T. Dunn, H. Sadasivan, J. Wadden, K. Goliya, K.-Y. Chen, D. Blaauw, R. Das, and S. Narayanasamy, "Squigglefilter: An accelerator for portable virus detection," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 535–549.
- [68] C. Firtina, N. Mansouri Ghiasi, J. Lindegger, G. Singh, M. B. Cavlak, H. Mao, and O. Mutlu, "Rawhash: enabling fast and accurate real-time analysis of raw nanopore signals for large genomes," *Bioinformatics*, vol. 39, no. Supplement_1, pp. 297–307, 06 2023.
- [69] B. Buchfink, K. Reuter, and H.-G. Drost, "Sensitive protein alignments at tree-of-life scale using diamond," *Nature methods*, vol. 18, no. 4, pp. 366–368, 2021.
- [70] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of molecular biology*, vol. 48, no. 3, pp. 443–453, 1970.
- [71] G. Wang and R. L. Dunbrack, "Scoring profile-to-profile sequence alignments," *Protein Science : A Publication of the Protein Society*, vol. 13, no. 6, pp. 1612–1626, Jun. 2004.
- [72] S. Henikoff and J. G. Henikoff, "Amino acid substitution matrices from protein blocks." *Proceedings of the National Academy of Sciences of the United States of America*, vol. 89, no. 22, pp. 10915–10919, Nov. 1992. [Online]. Available: <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC50453/>
- [73] H. Skutkova, M. Vitek, P. Babula, R. Kizek, and I. Provaznik, "Classification of genomic signals using dynamic time warping," *BMC Bioinformatics*, vol. 14, no. S10, p. S1, Aug. 2013. [Online]. Available: <https://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-14-S10-S1>
- [74] O. Gotoh, "An improved algorithm for matching biological sequences," *Journal of Molecular Biology*, vol. 162, no. 3, pp. 705–708, Dec. 1982. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/0022283682903989>
- [75] K.-M. Chao, W. R. Pearson, and W. Miller, "Aligning two sequences within a specified diagonal band," *Bioinformatics*, vol. 8, no. 5, pp. 481–487, Oct. 1992.
- [76] Z. Zhang, S. Schwartz, L. Wagner, and W. Miller, "A Greedy Algorithm for Aligning DNA Sequences," *Journal of Computational Biology*, vol. 7, no. 1-2, pp. 203–214, Feb. 2000.
- [77] "Vitis High-Level Synthesis User Guide," 2021.
- [78] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in Science Engineering*, vol. 12, no. 3, pp. 66–73, 2010.
- [79] Y. Ono, K. Asai, and M. Hamada, "PBSIM2: a simulator for long-read sequencers with a novel generative model of quality scores," *Bioinformatics*, vol. 37, no. 5, pp. 589–595, Mar. 2021. [Online]. Available: <https://doi.org/10.1093/bioinformatics/btaa835>
- [80] A. Rhoads and K. F. Au, "PacBio Sequencing and Its Applications," *Genomics, Proteomics & Bioinformatics*, vol. 13, no. 5, pp. 278–289, Oct. 2015.
- [81] The UniProt Consortium, A. Bateman, M.-J. Martin, S. Orchard, M. Magrane, S. Ahmad, E. Alpi, E. H. Bowler-Barnett, R. Britto, H. Bye-A-Jee, A. Cukura, P. Denny, T. Dogan, T. Ebenezer, J. Fan, P. Garmiri, L. J. Da Costa Gonzales, E. Hatton-Ellis, A. Hussein, A. Ignatchenko, G. Insana, R. Ishtiaq, V. Joshi, D. Jyothi, S. Kandasamy, A. Lock, A. Luciani, M. Lugaric, J. Luo, Y. Lussi, A. MacDougall, F. Madeira, M. Mahmoudy, A. Mishra, K. Moulang, A. Nightingale, S. Pundir, G. Qi, S. Raj, P. Raposo, D. L. Rice, R. Saidi, R. Santos, E. Speretta, J. Stephenson, P. Tootoo, E. Turner, N. Tyagi, P. Vasudev, K. Warner, X. Watkins, R. Zaru, H. Zellner, A. J. Bridge, L. Aimo, G. Argoud-Puy, A. H. Auchincloss, K. B. Axelsen, P. Bansal, D. Baratin, T. M. Batista Neto, M.-C. Blatter, J. T. Bolleman, E. Boutet, L. Breuza, B. C. Gil, C. Casals-Casas, K. C. Echionk, E. Coudert, B. Cuche, E. De Castro, A. Estreicher, M. L. Famiglietti, M. Feuermann, E. Gasteiger, P. Gaudet, S. Gehant, V. Gerritsen, A. Gos, N. Gruaz, C. Hulo, N. Hyka-Nouspikel, F. Jungo, A. Kerhornou, P. Le Mercier, D. Lieberherr, P. Masson, A. Morgat, V. Muthukrishnan, S. Paesano, I. Pedruzzi, S. Pilbout, L. Pourcel, S. Poux, M. Pozzato, M. Pruess, N. Redaschi, C. Rivoire, C. J. A. Sigrist, K. Sonesson, S. Sundaram, C. H. Wu, C. N. Arighi, L. Arminski, C. Chen, Y. Chen, H. Huang, K. Laiho, P. McGarvey, D. A. Natale, K. Ross, C. R. Vinayaka, Q. Wang, Y. Wang, and J. Zhang, "UniProt: the Universal Protein Knowledgebase in 2023," *Nucleic Acids Research*, vol. 51, no. D1, pp. D523–D531, Jan. 2023. [Online]. Available: <https://academic.oup.com/nar/article/51/D1/D523/6835362>
- [82] K. Reinert, T. H. Dadi, M. Ehrhardt, S. Hauswedell, S. Mehringer, R. Rahn, J. Kim, C. Pockrandt, J. Winkler, E. Siragusa, G. Urgese, and D. Weese, "The SeqAn C++ template library for efficient sequence analysis: A resource for programmers," *Journal of Biotechnology*, vol. 261, pp. 157–168, Nov. 2017.
- [83] B. Schmidt, F. Kallenborn, A. Chacon, and C. Hundt, "Cudasw++ 4.0: ultra-fast gpu-based smith-waterman protein sequence database search," *bioRxiv*, pp. 2023–10, 2023.
- [84] Xilinx, "Xilinx vitis libraries." [Online]. Available: https://xilinx.github.io/Vitis_Libraries
- [85] S. Marco-Sola, J. C. Moure, M. Moreto, and A. Espinosa, "Fast gap-affine pairwise alignment using the wavefront algorithm," *Bioinformatics*, vol. 37, no. 4, pp. 456–463, 2021.
- [86] C. Lee, C. Grasso, and M. F. Sharlow, "Multiple sequence alignment using partial order graphs," *Bioinformatics*, vol. 18, no. 3, pp. 452–464, 2002.
- [87] S. Marco-Sola, J. C. Moure, M. Moreto, and A. Espinosa, "Fast gap-affine pairwise alignment using the wavefront algorithm," *Bioinformatics*, vol. 37, no. 4, pp. 456–463, 09 2020. [Online]. Available: <https://doi.org/10.1093/bioinformatics/btaa777>
- [88] R. Groot Koerkamp and P. Ivanov, "Exact global alignment using a* with chaining seed heuristic and match pruning," *Bioinformatics*, vol. 40, no. 3, p. btac032, 2024.
- [89] N. Noll, M. Molari, L. P. Shaw, and R. A. Neher, "Pangraph: scalable bacterial pan-genome graph construction," *Microbial Genomics*, vol. 9, no. 6, p. 001034, 2023.
- [90] G. Hickey, D. Heller, J. Monlong, J. A. Sibbesen, J. Sirén, J. Eizenga, E. T. Dawson, E. Garrison, A. M. Novak, and B. Paten, "Genotyping structural variants in pangenome graphs using the vg toolkit," *Genome biology*, vol. 21, pp. 1–17, 2020.
- [91] Y. Gu, A. Subramanian, T. Dunn, A. Khadem, K.-Y. Chen, S. Paul, M. Vasimuddin, S. Misra, D. Blaauw, S. Narayanasamy *et al.*, "Gendp: A framework of dynamic programming acceleration for genome sequencing analysis," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–15.
- [92] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Computing Surveys (csur)*, vol. 34, no. 2, pp. 171–210, 2002.
- [93] W. J. Dally, Y. Turakhia, and S. Han, "Domain-specific hardware accelerators," *Communications of the ACM*, vol. 63, no. 7, pp. 48–57, Jun. 2020. [Online]. Available: <https://dl.acm.org/doi/10.1145/3361682>
- [94] J. Cong, J. Lau, G. Liu, S. Neuendorffer, P. Pan, K. Vissers, and Z. Zhang, "FPGA HLS Today: Successes, Challenges, and Opportunities," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 15, no. 4, pp. 51:1–51:42, Aug. 2022. [Online]. Available: <https://dl.acm.org/doi/10.1145/3530775>
- [95] K. Rupnow, Y. Liang, Y. Li, and D. Chen, "A study of high-level synthesis: Promises and challenges," in *2011 9th IEEE International Conference on ASIC*, Oct. 2011, pp. 1102–1105.
- [96] K. Benkrid, Ying Liu, and A. Benkrid, "A Highly Parameterized and Efficient FPGA-Based Skeleton for Pairwise Biological Sequence Alignment," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 4, pp. 561–570, Apr. 2009. [Online]. Available: <http://ieeexplore.ieee.org/document/4773142/>

- [97] H. Ye, C. Hao, J. Cheng, H. Jeong, J. Huang, S. Neuendorffer, and D. Chen, "Scalehls: A new scalable high-level synthesis framework on multi-level intermediate representation," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 741–755.
- [98] Y.-H. Lai, Y. Chi, Y. Hu, J. Wang, C. H. Yu, Y. Zhou, J. Cong, and Z. Zhang, "Heterocl: A multi-paradigm programming infrastructure for software-defined reconfigurable computing," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 242–251. [Online]. Available: <https://doi.org/10.1145/3289602.3293910>
- [99] Y.-H. Lai, H. Rong, S. Zheng, W. Zhang, X. Cui, Y. Jia, J. Wang, B. Sullivan, Z. Zhang, Y. Liang, Y. Zhang, J. Cong, N. George, J. Alvarez, C. Hughes, and P. Dubey, "Susy: A programming model for productive construction of high-performance systolic arrays on fpgas," in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2020, pp. 1–9.
- [100] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shepsman, N. Vasilache, and O. Zinenko, "Mlir: Scaling compiler infrastructure for domain specific computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2021, pp. 2–14.

APPENDIX

A. Abstract

In this section, we provide the DP-HLS artifact, including instructions on accessing the full codebase, installing software dependencies, configuring, and running our tool using the provided scripts, and evaluating/reproducing the results. This artifact also provides instructions for evaluating GPU and CPU baselines. The DP-HLS codebase is publicly available on GitHub (<https://github.com/TurakhiaLab/DP-HLS>) and Zenodo (DOI: 10.5281/zenodo.17853462).

B. Artifact check-list (meta-information)

- **Program:** DP-HLS codebase, GASAL2, CUDASW++4.0, Minimap2, SeqAn3
- **Compilation:** Vitis HLS 2021.2, gcc-11, g++-11
- **Data set:** Simulated DNA datasets generated using PBSIM2 - consisting of 1000 256-base pair reads, randomly sampled protein sequences of 256 base-pairs each from UniProtKB
- **Run-time environment:** Ubuntu 22.04, Python 3.10, CUDA 12.2 Toolkit with Nsight Systems
- **Hardware:** 36-core x86 CPU machine, GPU machine with NVIDIA Tesla T4 GPU
- **Metrics:** Throughput (number of alignments/sec), cost-efficiency (number of alignments per USD)
- **Output:** Numerical values produced in log files
- **Experiments:** Experiments reproduce 1) FPGA HLS evaluation results (maximum throughput), 2) GPU baseline evaluation (throughput and cost-efficiency), 3) CPU baseline evaluation (cost-efficiency)
- **How much time is needed to prepare workflow (approximately)?:** 1-2 hours
- **How much time is needed to complete experiments (approximately)?:** 2-3 hours
- **Publicly available?:** Yes
- **Code licenses:** MIT License
- **Data licenses:** MIT License
- **Archive DOI:** DOI: 10.5281/zenodo.17853462

C. Description

1) *How to Access:* Users can access our artifacts by cloning the DP-HLS GitHub repository (<https://github.com/TurakhiaLab/DP-HLS>) or downloading

the Zenodo package (DOI: 10.5281/zenodo.17853462). Approximate disk space required after unpacking: 200MB.

2) Hardware Dependencies:

- **DP-HLS:** AMD FPGA instance (e.g., AMD Xilinx UltraScale+) (for bitstream deployment)
- **CPU baseline:** x86 CPU with 36 cores
- **GPU baseline:** NVIDIA Tesla T4 GPU

3) Software Dependencies:

- **DP-HLS:** OS: Ubuntu 24.04, Vitis HLS 2021.2, Python 3.10, gcc-11, g++-11, CMake
- **CPU baseline:** OS: Ubuntu 24.04, gcc-11, g++-11, Python 3.10, Docker, CMake, zlib
- **GPU baseline:** CUDA 12.2 Toolkit with Nsight systems, gcc-11, g++-11, zlib, CMake

4) *Data Sets:* All required benchmarking and synthetic datasets are included in the DP-HLS codebase.

D. Installation

1) **DP-HLS: Step 1:** To run the DP-HLS tool, download the Vitis HLS 2021.2 toolchain from here.

Step 2: Use the following commands to clone the DP-HLS repository and build the HLS kernels:

```
1 git clone https://github.com/TurakhiaLab/DP-HLS
2 cd DP-HLS
3
4 mkdir build
5 cd build
6 cmake ..
```

Step 3: Set the HLS_HOME path (Vitis HLS installation path) inside CMakeLists.txt

Step 4: Run the following command:

```
1 make test_csims_global_affine
```

2) **CPU baseline:** Run the following commands to install the CPU baseline tools (SeqAn3):

```
1 git clone https://github.com/seqan/seqan3.git
2 cd SeqAn3
```

3) **GPU baseline:** Run the following commands to install and build the GPU baseline tool (GASAL2):

```
1 cd https://github.com/nahmedraja/GASAL2.git
2 cd GASAL2
3
4 #configure the project
5 ./configure.sh <CUDA_installation_path>
6
7 #build the GASAL2 library
8 make GPU_SM_ARCH=sm_75 MAX_QUERY_LEN=256 N_CODE=0x4E
   N_PENALTY=1
9
10 #build test program
11 cd test_prog
12 make
13
14 #set dataset path
15 export DATASET_PATH=<add_dataset_path>
```

Run the following commands to build the GPU baseline tool (CUDASW++):

```
1 git clone https://github.com/asbschmidt/CUDASW4.git
2 cd CUDASW4
3
4 #build the project
5 make makedb
6 make align
7
```

```

8 #copy or move your dataset files into the data
  directory
9
10 #generate the database
11 mkdir dbfolder
12
13 ./makedb <dataset_path> dbfolder/reference

```

E. Experiment workflow

- 1) **DP-HLS:** Once Vitis HLS and other software dependencies are installed and DP-HLS kernels are built, please run the following commands to perform simulation, synthesis, and co-simulation of 15 DP-HLS kernels presented in the paper (Table II, Column 8):

```

1 cd DP-HLS
2
3 #run the simulation, synthesis, and cosimulation for
  all 15 kernels (takes around 2-3 hours)
4 ./evaluate_throughput.sh
5
6 #extract the reports for the computed throughputs
7 python3 compute_throughput.py

```

- 2) **CPU baseline:** Run the following commands to get the cost-efficiency of the CPU baseline presented in Fig. 6A:

```

1 #generate the reports for cost-efficiency numbers
2 ./run_seqan3_baseline.sh

```

- 3) **GPU baseline:** Run the following commands to get the throughput and cost-efficiency of the GPU baseline presented in Fig. 6C,E:

```

1 cd GASAL2/test_prog/
2
3 #profile all the sequence alignments and get the kernel
  running time
4 ./profile_test_prog.sh
5
6 cd ..
7
8 #generate reports
9 python3 extract_report.py
10
11 cd ../../
12
13 cd CUDASW4
14
15 #performs benchmarking
16 nsys profile --trace=cuda -o ./report_cudasw4 ./align
  --query <path_of_dataset_in_data_folder> --db
  dbfolder/reference
17
18 #generate reports
19 python3 extract_report.py

```

F. Evaluation and expected results

- 1) **DP-HLS kernels maximum throughput evaluation:** This experiment verifies that DP-HLS achieves high throughput (in terms of alignments per sec) across all 15 kernels reported in the paper (Table II, Column 8). Part I of Section E details the workflow used to reproduce these results.
- 2) **Cost-efficiency comparison with CPU baseline:** This experiment evaluates the cost-efficiency of DP-HLS relative to the CPU baseline presented in the paper (Fig. 6A). Comparison of numbers generated by Part I and Part II of Section E reproduces the cost-efficiency.
- 3) **Throughput and cost-efficiency comparison with GPU baseline:** This experiment evaluates the cost-efficiency

and throughput of DP-HLS relative to the GPU baseline reported in the paper (Fig. 6C,E). Comparison of the numbers generated by Part I and Part III of Section E reproduces the throughput and cost-efficiency.

We expect the reproduced values to fall within the same general range as the numbers reported in the paper. However, because our original evaluation used AWS EC2 FPGA, CPU, and GPU instances, results may vary when running on other hardware platforms.

G. Methodology

Submission, reviewing, and badging methodology:

- <https://www.acm.org/publications/policies/artifact-review-and-badging-current>
- <https://cTuning.org/ae>