Automatic Code-Generation for Accelerating Structured-Mesh-Based Explicit Numerical Solvers on FPGAs

Beniel Thileepan
Department of Computer Science
University of Warwick
Coventry, United Kingdom
beniel.thileepan@warwick.ac.uk

Suhaib A. Fahmy
King Abdullah University of Science and Technology (KAUST)
Thuwal, Saudi Arabia
suhaib.fahmy@kaust.edu.sa

Gihan R. Mudalige
Department of Computer Science
University of Warwick
Coventry, United Kingdom
g.mudalige@warwick.ac.uk

***Abstract*—Structured-mesh-based stencil computations are a common motif in many numerical algorithms, such as for solving PDEs. Recent work has shown promising runtime performance and energy efficiency when mapping these applications to FPGAs. However, this requires significant manual effort with hardware-specific optimizations and customization. We present a new code-generation framework that automates the mapping of stencil applications to FPGAs. From a domain-specific declaration, the framework applies radical, ad-hoc, and dynamic optimizations, including a novel window-buffer chaining scheme, and an on-chip loopback pipeline approach that minimizes memory transaction overhead. We show that the generated code is able to match or exceed the performance of hand-tuned state-of-the-art designs. A range of stencil solvers benchmarked, including non-trivial multi-stencil solvers, on two FPGAs, demonstrating performance-portability. Comparisons to best-in-class solvers on an Nvidia H100 GPU indicate competitive performance with 2-19× energy savings.**

***Index Terms*—Stencil Applications, Field Programmable Gate Arrays, Automatic Code Generation, Domain Specific Languages**

I. Introduction

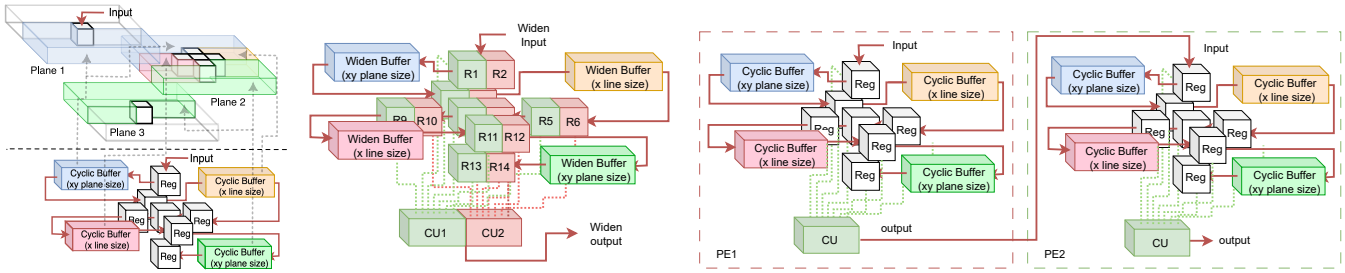
Recent work has demonstrated the significant utility of Field Programmable Gate Arrays (FPGAs) for numerical simulations in scientific and high-performance computing (HPC). Their energy efficiency is of particular note due to their slower clock frequencies and spatial on-chip communication. Modern FPGAs also demonstrate enhanced performance for floating point computations through more capable DSP blocks and spatial processor arrays [1]. Work has been shown in the solution of partial differential equations (PDEs) [2]–[4], Multilevel Monte Carlo Methods (MLMC) [5], [6] and for more specific applications such as weather simulation [7], [8] and financial computing [9]–[12]. However, the dataflow model necessary to build FPGA accelerators and the low-level hardware expertise required have limited their wider adoption. The introduction of high-level synthesis (HLS) tools that can translate programs written in standard high-level languages such as C/C++ or SYCL has somewhat lowered the barrier to entry. However, achieving high performance still requires significant manual low-level design iteration through a long-winded development process.

One solution is to exploit the key characteristics of a class of applications, to develop domain-specific abstractions which can be used by developers to describe a specific problem, and apply automated translation and compilation of optimized parallel implementations. Such a strategy targeting a specific class of applications, for instance, using Domain Specific Languages (DSLs), has long been effective in HPC. In this paper, we investigate and develop such automatic translation techniques for a stencil DSL and demonstrate effective code generation for FPGAs.

We focus on the structured-mesh-based application class, characterized by stencil computations. This parallel pattern is a widely used motif, particularly in solvers for PDEs. The main feature is looping over a rectangular multi-dimensional set of mesh points using one or more *stencils* to process data. In explicit solvers, one or more of these stencil loops are wrapped within an iterative time-marching loop, leading to what are called *iterative stencil loops* (ISLs). Indeed, a number of previous works have explored automatic code-generation for FPGAs targeting these applications [7], [8], [13]–[16]. However, specialized optimizations applied by these frameworks are limited, resulting in reduced performance and restricting them to simpler stencil applications.

We present an alternative code-generation strategy that builds upon these works, applying previously demonstrated as well as novel optimizations in a flow that uses the OPS structured mesh DSL [17] as its front-end. Specifically, we make the following contributions:

1) We develop a new code generator for OPS, based on LLVM/Clang LibTooling and dynamic skeleton templates to produce FPGA HLS code. It allows the modularization and reuse of optimized code components with dynamic customization and incorporates progressive optimizations in an ad-hoc manner based on the target.
2) Two novel optimization approaches are applied: (1) a new *Window Buffer Chaining Algorithm* automates the creation of optimal memory buffers between stencil points. With



(a) Buffers for 7pt 3D stencil (b) Cell parallelism (c) Step parallelism

Fig. 1: Architecture optimizations for implementing stencil accelerators.

this algorithm, any generic stencils defined can be translated to an effective FPGA stencil computational architecture, and (2) a novel *On-Chip Loopback Pipeline* design that optimizes memory read-write overhead on FPGAs for small and medium mesh sizes.

3) The new code-generator is used to automatically generate FPGA HLS code for a range of applications, including a non-trivial multi-stencil solver, targeting the AMD Alveo U280 and AMD Versal VCK5000 data center FPGA accelerators. Throughput, bandwidth, and energy are compared to state-of-the-art hand-coded and manually tuned solutions from [2], [3]. We further compare FPGA performance to optimized code for the same applications on an Nvidia H100 GPU, the best-in-class traditional architecture for such solvers.

Results indicate that the OPS code-generated solvers closely match or exceed the performance of hand-coded designs and those reported in previous works that automatically generate HLS code for FPGAs, even without applying batching (combining multiple smaller mesh computations together to improve throughput) or spatial-blocking/tiling (computing larger meshes by breaking them into smaller spatial blocks) (see Table I). Our tool can also target different FPGA architecture generations and capacities. We also see comparable throughput to GPU but with over 4× better energy efficiency on the FPGAs.

The design space that needs to be explored to gain even decent performance on FPGAs is vast, much larger than for CPUs/GPUs. Our framework automates this process, eliminating the need for a user to write **any** HLS code, significantly reducing development time, and outperforming the state-of-the-art. Both window-buffer chaining and on-chip loopback are novel optimizations that boost performance. Their theoretical limitations have been analytically modeled in this paper, and their application detailed, with potential impact beyond this domain.

II. Background

Unlike traditional architectures (CPUs/GPUs), FPGAs do not have a fixed architecture but comprise basic circuit elements, such as look-up tables (LUTs), registers, a significant number of digital signal processing (DSP) arithmetic blocks, and small on-chip block memories (BRAM/URAM). An extensive

TABLE I: Comparing our generated designs to previous work. ([h]: handcoded, [a]: automatic/code-gen, [s]: single batch, [m]: multi-batch, [t]: tiled). Platforms - AMD: U280, VCK5000, and ADM-KU3. Intel: GX1150, GX2800

Work	Best Kernel	(GFLOP/s)	Platform
Kamalakkannan et al. [2][h]	Poisson2D[s]	735.00	U280
	Poisson2D[m]	922.00	U280
Waidyasooriya et al. [3][h]	Jacobian2D[t,s]	874.00	GX1150
StencilFlow [7][a,s]	Poisson2D	568.20	GX2800
SODA [15][a,t,s]	Heat3D	134.91	ADM-KU3
	Jacobian2D	90.04	ADM-KU3
	Jacobian3D	83.98	ADM-KU3
Stencil-HMLS [8][a,s]	PW-Advection	38.15	U280
This paper[a]	Poisson2D[s]	984.61	U280
	Jacobian2D[s]	814.20	U280
	Jacobian3D[s]	752.73	U280
	Heat3D[s]	967.37	U280
	PW-Advection[s]	271.46	VCK5000

routing fabric allows these elements to be composed to create spatial datapath accelerators. Contemporary large FPGAs, such as one of the AMD models targeted in this paper, can also comprise multiple in-package dies, called Super Logic Regions (SLRs), connected via a silicon interposer. Data bandwidth between components within a die is extremely high (TB/s), while inter-SLR bandwidth is more limited. This is crucial to consider in larger FPGA accelerator designs.

Accelerators are built by decomposing a complex computation into arithmetic and data-storing primitives, which are physically composed together using the flexible routing architecture, allowing data to move directly between datapath elements with low latency. This lack of reliance on register/-cache/memory for data movement between arithmetic operations is a key factor in spatial architectures' energy efficiency. The datapath is customized for a specific application, resulting in a deep pipeline, which can ideally accept new data every clock cycle and produce results with a latency equal to the pipeline depth.

A. Implementing Stencil Codes on FPGAs

We begin with a specific example of an iterative stencil loop (ISL) and explore its optimal FPGA implementation. Equation (1) details an ISL for the solution of a 2D finite-difference scheme for the explicit solution of a generic PDE:

$$U^{t}_{x,y} = aU^{t-1}_{x-1,y} + bU^{t-1}_{x+1,y} + cU^{t-1}_{x,y-1} + dU^{t-1}_{x,y+1} + eU^{t-1}_{x,y} \quad (1)$$

where U is a 2D discretization of a regular rectangular domain. Each mesh point is updated every time step, t, using the value of its 4 neighbors and itself at time step $t-1$, giving a 5-point stencil. The time-step iteration continues until a steady-state solution is achieved. The scheme extends naturally to 3D or higher dimensions (e.g., see Table II). More complex applications (see Section V) consist of multiple such stencil loops within the time-marching loop. In an explicit scheme such as the above, each mesh point could be computed in parallel, as there are no data dependencies, but time-step iterations must be done in order.

Previous work on mapping stencil applications to FPGAs has identified a number of strategies for optimizing performance; we present a brief overview from Kamalakkannan et al. [2], which summarizes these for both 2D and 3D, including multidimensional mesh elements and multiple stencil loops. The key baseline optimizations required to obtain high performance are (1) transformations enabling pipelining and (2) unrolling loops by replicating computational units (CUs). These lead to the following specific transformations to the stencil code:

1) Unrolling multi-dimensional nested stencil loops, creating longer pipelines. Each stage of unrolling carries out the computation for one of the x, y, and z dimensions of a mesh point (Fig. 1a).
2) Replicating multiple pipelines for the same computation (i.e., loop body or kernel)–called the *cell-parallel* method [4]–allows computation of the stencil on multiple mesh points simultaneously. Each replica CU, when executed in parallel, leads to a vectorized operation (Fig. 1b).
3) Unrolling the outer, iterative time-marching loop—called the *step-parallel* method [4], allows results from a previous iteration to be fed to the next iteration through fast on-chip memory, without writing back to slower external memory (Fig. 1c).

Transformation 1 enables the optimal data reuse path. A chain of on-chip FIFO buffers is used to cache data for seamless streaming of mesh elements. The amount of buffering depends on the order and dimensionality of the stencil. In a 2D application, a D-order stencil requires buffering D *rows*, while in a 3D application, the D-order stencil necessitates buffering D *planes*, and each plane requires buffering of D *rows*. In this technique, referred to as *window buffering* in [2], the total number of mesh elements that must be buffered is determined by the maximum number of mesh elements between any two stencil points. An illustration of this perfect data reuse path for a 3D, 2nd order stencil is shown in Fig. 1.

Fig. 1b shows an implementation of Transformation 2 with a factor of 2, where the vectorization factor indicates the number of mesh points updated simultaneously. However, the resource capacity in an FPGA can restrict the number of feasible parallel units.

Fig. 1c illustrates unrolling the outer iterative loop leading to 2 "compute modules" (Transformation 3). This technique boosts throughput without needing extra external memory bandwidth, but the unrolling factor is again limited by FPGA resources and on-chip memory capacity. Reduced external memory access also improves power efficiency, but the longer computational pipeline can significantly affect performance for small meshes due to under-utilization of resources in the deep pipeline [2]. The novel loopback design proposed in this paper addresses this issue (as discussed in Section IV-B).

The runtime (latency in clock cycles) of the above designs for 2D and 3D stencil applications for n_{iter} iterations can be modeled, based on the mesh size ($m \times n$ or $m \times n \times l$), vectorization factor (V) and iterative loop unroll factor (p) as detailed in [2]:

$$Latency_{2D} = (n_{iter}/p) \cdot \lceil m/V \rceil \cdot (n + pD/2) \quad (2)$$

$$Latency_{3D} = (n_{iter}/p) \cdot \lceil m/V \rceil \cdot n \times (l + pD/2) \quad (3)$$

Considering the 2D case, $\lceil m/V \rceil$ is the total latency taken to process a row from a mesh sized $m \times n$ (assuming each row is padded to be a multiple of V if required). If the order of the stencil is D, the compute pipeline will process $n + D/2$ rows, as there are $D/2$ different rows between the current stencil update mesh point and the farthest mesh point required for the stencil computation. In FPGA design, initiation interval (II) = 1 is ideal, allowing the CU to accept an input every cycle. Higher II values, caused by PE complexity, reduce throughput by accepting inputs only every N cycles. Our model assumes II=1. It similarly extends to 3D. We generalize this model further in Section IV-B.

Furthermore, the model incorporates limitations on the vectorization factor (V), and the unroll factor p. Vectorization must satisfy the following condition:

$$BW_{channel} \geq 2Vf\texttt{sizeof}(t) \quad (4)$$

where the bandwidth of the data channel, i.e., port connected to the device memory, is $BW_{channel}$, the operating frequency of the FPGA is f and the size in bytes of a mesh element is $\texttt{sizeof}(t)$. The maximum p that can be chosen is limited by the minimum of p_{dsp} or p_{mem}, defined as:

$$p_{dsp} = FPGA_{dsp}/(VG_{dsp}) \quad (5)$$

$$p_{mem} = FPGA_{mem}/(\texttt{sizeof}(t)D(\prod_{i=1}^{N} m_i)) \quad (6)$$

Where G_{dsp} is the number of DSP blocks required for a single mesh element computation, $FPGA_{dsp}$ is the total number of DSP blocks available in the FPGA, D is the order of the stencil, $FPGA_{mem}$ is the total on-chip FPGA memory (BRAMs and URAMs) and m_i is the mesh size in i-th dimension.

In addition to the above, spatial blocking optimizations to implement solutions for larger meshes and batching of multiple meshes to increase throughput of the pipeline are discussed in [2], enabling more complex applications to be developed while retaining high performance. Our objective in this paper is to generate the above designs automatically for non-trivial applications through code generation.

B. Related Work

Previous research on automatic code-generation targeting FPGAs includes work on image processing, such as Hipacc [18] and HeteroCL [19]. Several domain specific language frameworks have also been proposed: SDSLc [13], SODA [15], StencilFlow [7] and Stencil-HMLS [8] for stencil codes. SDSLc uses PolyOpt/HLS [14] as the optimization backend and a later implementation by Natale et al. [20] introduces polyhedral optimizations to generate FPGA code from high-level syntax to support multi-FPGA deep pipeline kernel solutions. However, both these tools do not achieve high performance; SDSLc under-performs hand-written code by $11\times$ in some cases [13].

SODA [15] develops a DSL with source-to-source translation providing a custom compiler front-end. It addresses both spatial and temporal blocking optimization of a single deep pipeline of kernels as well as iterative stencil loops (ISLs). HeteroCL [19], an image processing DSL leverages the SODA backend for image processing-related stencil computations, which are single deep pipelines of kernels. SODA proposes an optimal microarchitecture generated from the DSL. However, it underperforms due to significant routing congestion arising from the design of a single deeply pipelined processing element (PE). This single PE design fails to support more modern FPGAs with multiple SLR regions, where inter-SLR communication limits scalability.

StencilFlow [7] is a declarative DSL built on top of the DaCe [16] backend framework that focuses on dataflow optimizations. Stencilflow demonstrates promising performance from code generated targeting a single FPGA and for multiple FPGAs. It is designed to support both ISL and non-iterative deep pipelines and performs heuristic dataflow optimizations. Though StencilFlow is versatile enough to support non-traditional multi-stencil systems with good performance in general, the design itself does not incorporate more novel optimizations relevant for ISL applications and thus comparatively underperforms hand-coded designs such as those in [2], [3]. Additionally, StencilFlow does not incorporate the outer loop in the generated FPGA architecture for ISL settings which leads to multiple kernel calls and blocking buffer copies for an ISL application with a large number of iterations ($p < n_{iter}$), due to kernel call overheads.

More recently, Stencil-HMLS [8] uses MLIR-based code translation via a custom HLS dialect to optimize a weather application. The HLS dialect captures hardware optimizations that are provided by Vitis HLS and thus can apply better optimizations. Stencil-HMLS claims 14–100$\times$ improved performance over the DaCe implementation of a weather application (PW-Advection). However, the authors indicate it as a work in progress pending further benchmarking.

In contrast to the above works, in this paper, an ISL is declared using the OPS DSL. OPS (Oxford Parallel library for Structured mesh solvers) [21] is a high-level embedded domain specific language for writing multi-block structured mesh algorithms, and the corresponding software library and code translation tools to enable automatic parallelization on multi-core and many-core architectures. We develop a code-generator based on LLVM/Clang LibTooling to parse/analyze source code, extract problem parameters, and automatically generate FPGA high-level synthesis code. The generated code incorporates best-in-class optimizations that have been encoded in dynamic skeleton templates. The resulting deep pipelined loopback circuit design allows small/medium-sized meshes to execute with high throughput. As we will show in the rest of this paper, it produces HLS code for a specified target FPGA, resulting in better throughput in ISL applications, compared to the state-of-the-art. A summary comparison is given in Table I. Poisson2D, Laplace2D, Jacobian2D, and Heat3D in this table are specified in detail in Table II. We also implemented PW-Advection with OPS using the specification in [8] and compared performance.

Listing 1 ISL of eq (1) declared with the OPS API

```
1  void sten5pt(const ACC<float> &u0, ACC<float> &u1) {
2      u1(0,0) = a*u0(-1,0) + b*u0(1,0) + c*u0(0,-1) + d*u(0,1) +
       ↪  e*u(0,0);}
3  void copy(const ACC<float> &u0, ACC<float> &u1) {
4      u0(0,0) = u1(0,0);}
5
6  void main(){
7  ops_block block = ops_decl_block(2,"block0");
8  int size[] = {size_x, size_y}; int halo_neg = {-1,-1};
9  int halo_pos = {1,1}; int base = {0,0};
10
11 float* data0, data1;
12 ops_dat dat0 = ops_decl_dat(block, 1, size, base, halo_neg, halo_pos,
   ↪  data0, "float", "dat0");
13 ops_dat dat1 = ops_decl_dat(block, 1, size, base, halo_neg, halo_pos,
   ↪  data0, "float", "dat1");
14
15 int s2D_00[] = {0,0};
16 ops_stencil S2D_00 = ops_decl_stencil(2,1,s2D_00,"00");
17 int s2D_5pt[] = {0,-1, -1,0, 0,0, 1,0, 1,1};
18 ops_stencil S2D_5pt = ops_decl_stencil(2,5,s2D_5pt,"5pt");
19
20 unsigned int N = 100;
21 int range[] = {0, size_x, 0, size_y};
22 #pragma ISL "isl_0" N
23 for (int iter = 0; iter < N; iter++) {
24     // 5-point stencil loop
25     ops_par_loop(stencil5pt, "5pt-sten", block, 1, range,
26         ops_arg_dat(dat0, 1, S2D_5pt, "float", OPS_READ),
27         ops_arg_dat(dat1, 1, S2D_00, "float", OPS_WRITE));
28     // copy updates
29     ops_par_loop(copy, "copy", block, 1, range,
30         ops_arg_dat(dat1, 1, S2D_00, "float", OPS_READ),
31         ops_arg_dat(dat0, 1, S2D_00, "float", OPS_WRITE));
32 }}
```

III. A Code-Generator for FPGAs

An ISL can be declared using OPS with its C/C++ API. For example, the stencil computation in Eq (1) can be declared within an ISL as detailed in Listing 1. Here, two stencils are declared (lines 15–18), `S2D_00` and `S2D_5pt`, a "self" stencil and a 5-point stencil, respectively, which are used to access several 2D data meshes holding FP32 (`float`) data, `dat0`, `dat1`, etc. The loops over the mesh are declared using the `ops_par_loop` call (lines 25–27 and lines 29–31). The computation for Eq (1) is declared as an elemental kernel `sten5pt` (lines 1–2) and used as function pointer argument in `ops_par_loop` (lines 25–27). Similarly, the copy operation is defined in `copy` (lines 3–4). The OPS API was first developed for declaring block-structured mesh

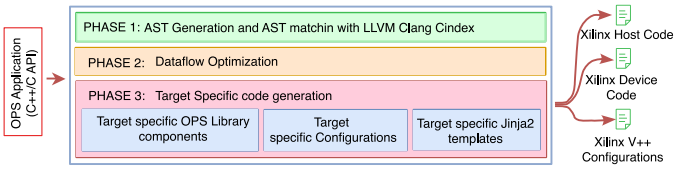


Fig. 2: High level view of OPS code generator for FPGAs.

Listing 2 Sample AST of the ISL region in Listing 1.

```
|- CursorKind.FOR_STMT
| |- CursorKind.DECL_STMT
| | `- CursorKind.VAR_DECL iter
| |    `- CursorKind.INTEGER_LITERAL  "int" 0
  ..................
| `- CursorKind.COMPOUND_STMT
  ..................
|    | | `- CursorKind.OVERLOADED_DECL_REF ops_par_loop
|    | |- CursorKind.DECL_REF_EXPR stencil5pt
  ..................
|      | `- CursorKind.OVERLOADED_DECL_REF ops_par_loop
|      |- CursorKind.DECL_REF_EXPR copy
```

applications in [21], [22]. As these works demonstrate, the API is sufficient for a loop-by-loop translation to generate parallel code for multi-core, many-core, and cluster systems using a range of parallel programming languages (CUDA, HIP, MPI, etc.). However, the main challenge of targeting FPGAs, in contrast to traditional CPU and GPU systems, is the creation of a dataflow architecture, requiring cross-loop analysis of the ISL. To support this, we introduce a `#pragma ISL` (line 22) "marker" to delineate ISLs of interest. Then, for FPGA targets, we parse the ISL region and reason about the dataflow of the `ops_par_loops` chain. The optimized dataflow representation can then be used to generate FPGA HLS code. The high-level overview of these steps is detailed in Fig. 2.

A. AST Generation and Matching

The OPS code-generator's parser supports the extraction of individual stencil loop parameters, their order, and the data used in the loops by inspecting the abstract syntax tree (AST) generated by the `Clang` compiler using LLVM's tooling library (libtooling) API. The process, based on techniques developed in [23] involves traversing the AST, checking if AST nodes match OPS API nodes. For a matched node, we extract the node's information and that of its children. For our example in Listing 1, the portion of the AST for the `stencil5pt ops_par_loop` and its related data is illustrated in Listing 2. `ops_decl_block` and `ops_decl_dat` etc., declarations are matched by searching for `FUNCTION_DECL` nodes. The `#pragma ISL` region is matched by searching for `FOR_STMT` nodes, within which ops_par_loop will be matched with `OVERLORDED_DEC_REFs`. Similarly, the arguments within an `ops_par_loop`, can be matched, but additionally, type checking is done for `dats` used in the loop (wrapped as `ops_arg_dats`) by searching for their declarations which appear as `FUNCTION_DECLs` inside the scope. The stencil used and access mode `OPS_READ` etc., is similarly extracted. Finally, one of the crucial components of the parallel loop, the `range`, is passed as a `DECL_REF_EXPR` where the AST matcher searches for the original `VAR_DECL` AST to verify the array.

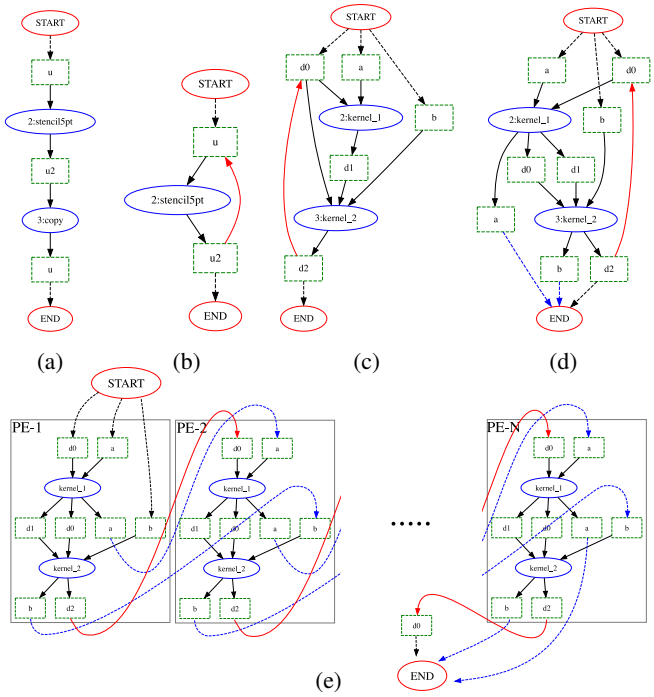


Fig. 3: Dataflow Optimizations (a), (b) Before and after copy detection. (c), (d) Before and after buffer propagation, (e) chained PEs dataflow after buffer propagation.

B. Dataflow Optimizations

The extracted information can now be used to build a dataflow graph of the ISL region, which is suitable for FPGA synthesis. This is an extra step that is critical for obtaining an optimized FPGA implementation compared to other platforms. We use the Rustworkx Python API [24] to implement these transformations. An ISL region can be represented as a dataflow graph with `ops_par_loops` as vertices and `ops_dats` as edges, which act as buffers through which data is communicated from one loop to another. For example, the dataflow graph for Listing 1 is given in Fig. 3a.

The dataflow graph is synthesized as a single PE where each `ops_par_loop` vertex is a dataflow task within the PE. In FPGA canonical dataflow regions–FPGA sub-regions that adhere to the principles of dataflow computing, buffers are converted to streams, where data flows continuously from a single producer (source of data) to a single consumer (processing or sink element). The solid black edges in the graph represent these streams. Multiple PEs can be chained together (implementing step-parallelism) by connecting the output buffers of one PE to the input buffers of the next, creating dataflow between PEs. We apply several optimizations over the initial vanilla dataflow graph before synthesis:

Copy kernel detection: A copy kernel simply loops over each mesh point in an `ops_dat` and copies it to another `ops_dat` (e.g., `copy` kernel in Listing 1). On an FPGA, it is optimal to map the input buffer to the output buffer directly and remove the kernel. Such kernels can be easily detected by analyzing the data access patterns of the `ops_par_loop`'s kernel. The dataflow graph after eliminating the copy kernel in Listing 1

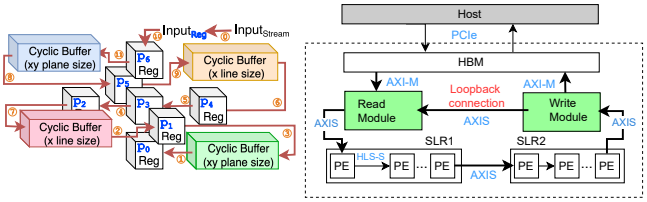
Listing 3 Consecutive kernels for buffer propagation.

```
void kernel_1(const ACC<float>& a, const ACC<float> &d0, ACC<float> &d1)
↪  { d1(0,0) = a(0,0) * d0(0,0); }
void kernel_2(const ACC<float>& b, const ACC<float> &d0, const
↪  ACC<float> &d1, ACC<float> &d2) {
    d2(0,0) = b(0,0) * d1(0,0) + d0(0,0); }
void copy(const ACC<float> &d0, ACC<float> &d2) { d0(0,0) = d2(0,0);}
```

is given in Fig. 3b. Here, the edge connection in red indicates an internal memory copy.

Buffer propagation: The input and output buffers of each vertex must be analyzed and the edges of the dataflow graph must be reconfigured to support optimal global memory movement. For example, consider the three consecutive kernels in Listing 3. Its single PE dataflow graph is given in Fig. 3c. However, this graph cannot be synthesized as multiple PEs due to (1) `d0` being accessed by both kernels (i.e. multiple accesses from external global memory within the same ISL region) and (2) `a` and `b` also being accessed directly from global memory. In the first case, it is optimal to read `d0` once, input it to `kernel_1`, and forward the same `d0` to `kernel_2`. This reconfiguration is illustrated in Fig. 3d. In the second case, to chain multiple PEs, `a` and `b` should be read once into the first PE, and this read-only data should be passed on to the following PEs. Fig. 3d again shows the reconfiguration needed in blue, and multiple PE chaining can be seen in Fig. 3e.

Mesh boundary data propagation with dependency analysis: Recall that an `ops_dat` can have a halo around its boundary (e.g. see `halo_pos` and `halo_neg` in Listing 1). When `ops_dats` are directly connected as memory streams, the values in the halos could get overwritten by garbage/uninitialized values from a different `ops_dat`. For example, the `copy` kernel in Listing 3 directly connects `d2` to `d0`, where `d2` has been updated in the previous kernel (`kernel_2`). However, `kernel_2` does not read `d0`, and only the internal (non-halo) mesh points in `d2` are updated. Thus, uninitialized values in the halo regions of `d2` overwrite the original values of `d0` in the `copy` kernel. With the above example, one solution would be to propagate the halo data of `d0` via an additional stream connection from `kernel_1` to `kernel_2` to `copy` kernel. However, this would consume more FPGA resources and could cause performance and routing issues. Our approach utilizes existing intermediate stream connections to propagate halo data with internal mesh point values. The dataflow layer performs a dependency analysis to generate a dependency graph. With the knowledge from dependency analysis, in this example, we can propagate the `d0` halo via either `d0(start)`→`d1`→`d2` or `d0(kernel_1)`→`d2`. In a practical problem, the cardinality of non-intermediate buffers, $|B'_{inter}|$ (such as `d0` and `d2`) will be much larger, and finding an appropriate path becomes a complex task compounded by the need to find a suitable mutually exclusive path for each non-intermediate buffer. However, read-only non-intermediate buffers B_{r-only} do not need to retain the halo transferred via other intermediate buffers. Theoretically, if we can show each non-intermediate-non-read-only buffer b can have a mutually exclusive path $P_b(S,E)$ from start(S) to end(E) then the halo propagation can be done without needing any additional



(a) (b)

Fig. 4: (a) Window buffer chaining for a 7pt 3D stencil. (b) End-to-end solver pipeline for loopback FPGA design.

Listing 4 Generated HLS snippet for an `ops_par_loop`.

```
1 //elemental kernel
2 inline void k1_core(const int* cst0, float reg0, ...){...}
3 class Stencil_k1 : public ops::hls::StencilCoreV2<...>{
4   void stencilRun(...){ ...; k1_core(...); /*elemental kernel call*/}
    ↪  };
5 void kernel1_PE(...){ Stencil_k1 stk1; stk1.stencilRun(...); }
```

stream connection to pass the halo (Eq 7).

$$\forall b \in (B'_{inter} \cap B'_{r-only}), \exists P(S,E) \text{ such that } \forall b' \in (B'_{inter} \cap B'_{r-only}) \setminus \{b\}, P_b(S,E) \cap P'_{b'}(S,E) = \emptyset. \quad (7)$$

However, this problem is a variation of the edge-disjoint shortest path problem, an NP-hard problem to solve. Therefore, we select two heuristic approaches to select suitable paths: (1) Dijkstra's shortest paths algorithm to find the shortest path between the buffer-pair. For our example scenario, we can decide to go with `d0(kernel_1)`→`d2` based on the shortest path. (2) Jaro-Winkler string comparison metric [25] to select between paths with the same path size, assuming that the users will name buffers meaningfully. If a suitable path cannot be found heuristically, the OPS translator can add a dedicated stream for halo propagation.

C. Target Specific Code-Generation

For generating multiple parallelizations for traditional targets, OPS uses the ideas developed in [23] where each `ops_par_loop`'s information (including the elemental kernel function) is used to populate a template or a "skeleton" that encodes the optimized state-of-the-art implementation for a given architecture and a parallel programming model. For example, to generate CUDA target code for NVIDIA GPUs, skeleton code written in CUDA is used with placeholders to expand/replace the text based on the elemental kernel, number of arguments, their access mode, dimensions, etc., of a candidate `ops_par_loop`. Essentially, the skeleton provides boilerplate for the best CUDA parallelization for implementing a structured-mesh/stencil loop. In [23], skeletons were written in C++ and expanded/replaced using Clang Libtooling's refactoring tool, which can apply replacements to the source code based on the AST of the skeleton. However, to simplify maintenance and re-usability, we use skeletons written in Jinja2 and refactor them with Python.

Extending the above for generating HLS targeting FPGAs is considerably more involved, requiring multiple skeletons for

Listing 5 Generated HLS snippet for a PE.

```
1  // Fused loops for a PE
2  void PE(..., <inter_PE_stream_connection_args>){
3    #pragma HLS DATAFLOW
4    ::hls::stream<..> k1_k2_arg1_interconnect;
5    ...
6    kernel1_PE(..., k1_k2_arg1_interconnect, ...);
7    kernel2_PE(k1_k2_arg1_interconnect, ...);
8    ...
9    kernelN_PE(...);
10 }
11 // PE outer loop unrolling - step parallel method
12 void PE_chain(<inter_PE_connection_args>){
13   for (int i = 0; i < iter_par_factor; i++) {
14   #pragma HLS UNROLL factor=iter_par_factor
15     PE(arg0_streams[i], arg0_streams[i+1], ...);
16   }
17 }
18 // PE chain top function with axis connections
19 void PE_func_top(const int num_iter,<const args...>,
   ↪  <config_args...>, <AXIS_stream_args..>){
20   ...
21   for (int i = 0; i < num_iter; i++){
22     ops::hls::axis2stream<...>(...); ...
23     PE_chain(..);
24     ops::hls::stream2axis<...>(...); ...
25   }
26 }
```

Listing 6 Host wrapper and CPU host function.

```
1  class Wrapper_ISL: public ops::hls::Kernel{
2    public:Wrapper_ISL():Kernel("ISL"),
3    m_kernelName("PE_func_top"), m_rwName("PE_rw_func_top") {
4      OCL_CHECK(err, m_kernel = cl::Kernel(m_fpga->getProgram(),
       ↪  m_kernelName.c_str(), &err));
5      ...
6      OCL_CHECK(err, m_rw = cl::Kernel(m_fpga->getProgram(),
       ↪  m_rwName.c_str(), &err));
7    }
8    void run(...){
9      // passing arguments from host to device via OCL
10     OCL_CHECK(err, err = m_kernel.setArg(<arg_id>, ...)
11     ...
12     OCL_CHECK(err, err = m_rw.setArg(<arg_id>, ...);
13     ...
14     //enqueue task to the execution queue
15     OCL_CHECK(err, err = m_fpga->getCommandQueue().
16     enqueueTask(m_rw, ...));
17     OCL_CHECK(err, err = m_fpga->getCommandQueue().
18     enqueueTask(m_kernel, ...));
19     ...
20     //synchronize
21     m_fpga->getCommandQueue().wait();
22   }
23 }
24 // CPU host function for ISL
25 void ops_iter_par_loop(int num_iter, int* range,
26   ops_arg(...), ...){
27   Wrapper_ISL() wrap_instance;
28   wrap_instance.run(range, num_itr, <ops_args...>);
29 }
```

different components – `ops_par_loop` kernels, PEs, memory read/write modules, host driver code, and configurations to achieve valid synthesis. More crucially, it requires the fine handling of stream connections between each of these components, including between PEs that may be synthesized in different SLRs on larger FPGAs.

The OPS code-generator builds up the HLS code by first producing Vitis C++ for each of the `ops_par_loops` in the ISL region by instantiating and expanding on a loop skeleton, which specifies the creation of the circuit components for each candidate loop in a PE. Listing 4 illustrates the basic structure. `k1_core` (line 2) is the outlined elemental kernel. However, the dataflow arrangement requires the loops to be fused as established in the dataflow graph from Phase 2. A skeleton for fused loops is used, which utilizes the per loop code generated together with specifying stream connectivity between loops. Listing 5, lines 1–10, illustrate the key structure of the generated HLS code.

For the next level, a skeleton is used to generate HLS for PEs. At this level, the stream connectivity between both inter- and intra-SLR PEs is specified (see Listing 5 lines 12–17). Additionally, read/write modules are generated (not shown here) with two special skeletons and their streams connected to the PE chain. The PE chain and the read-write module code are wrapped under a host-side visible function for the full ISL region (see Listing 6 lines 1–23). The kernel (`m_kernelName`) and data-mover module (`m_rwName`) (see line 3) are set. These are enqueued to run by passing arguments from the host via OCL (lines 9–21). A skeleton for the wrapper enables this code generation. The outer loop with `#pragma ISL` in the high-level program is replaced by a call to the wrapper function detailed in lines 24–29.

Finally the host (CPU) code is generated, consisting of the original high-level OPS API code (such as in Listing 1) but with modifications to call the wrapper code generated above. The changes to the code are done by direct text replacement of AST nodes similar to [23].

IV. FPGA Accelerator Design

A. Window-Buffer Chaining

The FPGA stencil accelerator design in Kamalavasan et.al [2] uses multiple cyclic buffers as illustrated in Fig. 1a. This "window buffer" technique boosts the pipeline performance of the stencil computation by storing data in on-chip memory more efficiently. The buffer sizes are allocated based on the maximum mesh sizes for a problem. However, in [2], the connectivity between buffers and the registers in the datapath pipeline was manually chained, requiring careful handling depending on the complexity of the stencil for each FPGA hardware platform. An example linking of buffers to registers is illustrated in Fig. 4a where the red arrows show the chaining of the components together. In our framework, we automate this window buffer chaining step with a new algorithm outlined in Alg. 1. It takes as input the sorted mesh points of the stencil. For example, the sorted ordering of points in the 7-point stencil in Fig. 4a is given by the subscripts of $p_0, p_1, ..., p_6$. The algorithm loops over each point and checks whether the next point is streaming from a cyclic buffer or is shifted from a register. If the next point (given by p_{+1} in Alg. 1) is on the same row of the mesh, then it is linked to a register (lines 10, 11). If the next point is in the adjacent row but on the same plane, then a buffer with a size equal to the maximum row size times the distance between the points in the y-dimension is created and added (lines 15, 16). If the next point is in a different plane then the buffer size is multiplied further by the number of planes between the points in the z-dimension (lines 13, 14). For HLS synthesis, chaining links must be in the correct order to satisfy dataflow. Therefore, the algorithm not only defines the connection between registers/buffers but also

Algorithm 1 Window Buffer Chaining Algorithm

```
INPUT: Sorted mesh-points (sp)
chain.add(reg_in, stream_in)
prev_buf ← NULL, prev_reg ← NULL
for all p ∈ sp do
    if p ≡ last_point then
        chain.add(p, reg_in)
        if prev_buf ≠ NULL then
            chain.add(prev_buf.pop(), prev_reg.pop())
        end if
    else if p_{+1}.row ≡ p.row then
        chain.add(p, p_{+1})
    else
        if _{+1}.plane ≡ p.plane then
            curr_buf ← rowbuf[sizeof((p.y − p_{+1}.y) × y_size))]
        else
            curr_buf← planebuf[sizeof((p.z − p_{+1}.z) × z_size))]
        end if
        chain.add(p, curr_buf)
        if prev_buf ≠ NULL then
            chain.add(prev_buf.pop(),prev_reg.pop())
        end if
        prev_reg.push(p_{+1})
        prev_buf.push(curr_buf)
    end if
end for
```

produces link pairs in the correct order (starting from the final register) as illustrated in the numbers in orange in Fig. 4a. The algorithm is novel compared to previous work [3], [4], [15], requiring no programmer intervention to create buffers from any OPS stencil definition.

B. Loopback Pipeline and Memory Model

The baseline latency models (2) and (3) developed in [2] and measured in clock cycles can be generalized for a mesh with N dimensions:

$$Latency_{ND} = \frac{n_{iter}}{p}\left(\left\lceil\frac{m_1}{V}\right\rceil \cdot \prod_{i=2}^{N-1} m_i \cdot (m_N + D \times p)\right) + L \tag{8}$$

Here, m_i is the size of the mesh in dimension i with a stencil of order D. Assuming initiation interval $II = 1$ and the latency of a single PE is L_{PE} then the total latency of a deep pipeline of p chained PEs is given by $L = pL_{PE}$ (neglecting small stream connection latency).

However, the end-to-end ($Latency_{tot}$) solution time also includes latency due to data movement to and from external (DDR or HBM) memory at the start and end of the PE chain (see Fig. 4b):

$$Latency_{tot\_cmp} = Latency_{ND} + (n_{iter}/p)Latency_{mem\_rw} \tag{9}$$

This memory overhead becomes more significant for small meshes. This was mitigated in [2] by batching multiple independent meshes for processing. In our framework, we introduce an additional optimization that allows the output of a PE chain to be looped back to the start of the chain within the FPGA, thereby avoiding costly external memory access. This has the advantage of increasing performance, even when processing a single mesh.

The key idea for the loopback pipeline is to retain the full mesh within the memory elements of the deep pipeline, consisting of the stream connections, registers, and on-chip memory Fig. 4b). Vitis HLS implements AXI-Stream (AXIS) buffers for memory access at the top level of the design and HLS-stream buffers for more lightweight streaming connections between components. Within the deep pipeline, each PE holds a fraction of data in registers and window buffers composed from on-chip memory. For loopback to work, the total number of vectorized mesh elements, $E = (\prod_{i=1}^{N} m_i)/V$ must fit entirely in on-chip memory.

Each PE can be considered as a buffer as it can hold a number of vectorized elements. The buffer depth is equal to the latency of the PE (assuming II=1), and the total depth contribution from all PEs is then $Len_{PE} = pL_{PE}$. Within an SLR, each PE is connected to others via HLS-stream connections that have total buffer length $Len_{interPE}$, which is dependent on the depth of the HLS-streams connecting PEs, l_{hls}, the number of PEs in each SLR, p_{SLR}, and the total number of SLRs used, N_{SLR}. Additionally, across SLRs, PEs are connected via AXIS buffers, contributing a total buffer length of $Len_{interSLR}$, which is proportional to the AXIS interconnect buffer size, l_{axis}, and the number of SLRs. If the FPGA platform is a single SLR, then this component is not included. As shown in Fig. 4b, the PE chain connected to data read-write modules via AXIS, contributes Len_{RW} to the total buffer length. Then the total length (Len_T) of the loopback pipeline is given by:

$$\begin{aligned} &Len_T = Len_{RW} + Len_{interSLR} + Len_{PE} + Len_{interPE}, \\ &\text{where } Len_{interSLR} = (N_{SLR} - 1)l_{axis} \\ &Len_{interPE} = N_{SLR}(p_{SLR} - 1)l_{hls} \end{aligned} \tag{10}$$

Users can control the length of stream buffers, l_{hls} and l_{axis} by overriding default values in the OPS configurations. With the maximum grid size given, whether to use the loopback design or not can be determined by the conditions:

$$Len_T > E_{max},\ E_{max} = \frac{\prod_{i=1}^{N} m_{i_{max}}}{V} \tag{11}$$

$$0.85 \times Len_T > E_{max} \tag{12}$$

where $m_{i_{max}}$ is the maximum mesh size in the ith dimension and E_{max} is the total vector elements in the maximum grid that need to be supported. For a practical synthesized design, a good rule-of-thumb is not to exceed approximately 85% of this limit (Eq 12), to allow for the AXI/HLS stream connections space to stall and starve. The AXI-stream interconnections and HLS-stream buffer sizes can be defined as an OPS-translator configuration.

V. Evaluation

We evaluate the performance of our code-generated designs on AMD Alveo U280 and Versal VCK5000 FPGA cards.

TABLE II: Stencil benchmarks overview.

Benchmark	Computation	FLOPs/cell
Black-Scholes 1D [9] [26]	$k1_i \cdot U^t_{i-1} + k2_i \cdot U^t_i + k3_i \cdot U^t_{i+1}$	5
Poisson 2D [2]	$\frac{1}{8} \cdot (U^t_{i,j-1} + U^t_{i-1,j} + U^t_{i+1,j} + U^t_{i,j+1}) + 0.5 \cdot U^t_{i,j}$	6
Laplace 2D [3]	$\frac{1}{4} \cdot (U^t_{i,j-1} + U^t_{i-1,j} + U^t_{i+1,j} + U^t_{i,j+1})$	4
Jacobian 9pt 2D [3]	$k1 \cdot U^t_{i-1,j-1} + k2 \cdot U^t_{i,j-1} + k3 \cdot U^t_{i+1,j-1} + k4 \cdot U^t_{i-1,j} + k5 \cdot U^t_{i,j} + k6 \cdot U^t_{i+1,j} + k7 \cdot U^t_{i-1,j+1} + k8 \cdot U^t_{i,j+1} + k9 \cdot U^t_{i+1,j+1}$	17
Diffusion 3D [27]	$k1 \cdot U^t_{i,j,k} + k2 \cdot (U^t_{i,j,k-1} + U^t_{i,j-1,k} + U^t_{i-1,j,k} + U^t_{i+1,j,k} + U^t_{i,j+1,k} + U^t_{i,j,k+1})$	10
Jacobian 7pt 3D [2]	$k1 \cdot U^t_{i,j,k} + k2 \cdot U^t_{i,j,k-1} + k3 \cdot U^t_{i,j-1,k} + k4 \cdot U^t_{i-1,j,k} + k5 \cdot U^t_{i+1,j,k} + k6 \cdot U^t_{i,j+1,k} + k7 \cdot U^t_{i,j,k+1}$	13

TABLE III: System specifications.

FPGA	**Alveo U280** [28]	**Versal VCK5000** [29]
Technology	16nm	7nm
SLRs	3	1
DSP Blocks	9024	1968
LUTs	1,304K	899.84K
Registers	2,607K	1,779K
BRAMs	2016 (9MB)	967 (4.35MB)
URAMs	960 (34.5MB)	463 (16.67MB)
HBM	8GB, 32ch, 460GB/s	N/A
DDR	32GB, 38GB/s	16GB, 102.4GB/s
Platform	Vitis 2022.2	Vitis 2022.1
Host	AMD EPYC 7763 (64 cores), 514 RAM	
GPU	**Nvidia H100 PCIe** [30]	
Global Mem.	80GB HBM2e, 2.0TB/s	
Host	2×AMD EPYC 9334 (32 cores), 384GB RAM	
Compilers, OS	CUDA 12.3, NVHPC 24.3 SDK, Debian GNU/Linux 12	

The VCK5000 has faster floating point capable DSP blocks. Performance is compared to optimized CUDA code, generated by OPS for an NVIDIA H100 GPU. Hardware specifications are noted in Table III. Eight stencil codes are benchmarked, including the multi-stencil applications. Computations are summarized for all applications in Table II and Alg 2. Model parameters for the FPGA designs are summarized in Table IV. Our application development workflow with automatic code-generation consists of the following process: (a) Determine whether to use the loopback pipeline based on the condition in (12), which is the case for all 1D and 2D applications used. (b) Setting the vectorization factor (V) to 8 (or a reduced number for large applications) based on (4). (c) Using software emulation-based resource estimates to set p and adjust it until the generated design reaches the DSP block or on-chip memory capacity (5,6), or in some cases, routing congestion prevents hardware implementation. (d) Fine-tuning the hardware design if it does not reach the target frequency by adjusting l_{axis}, l_{hls}. All above workflow tuning was done via configurations passed to the code generator. The end-to-end code generation and optimization passes are fully automated.

For GPU implementations, higher batch sizes were also tested, indicated by 10B/100B in results, to achieve higher GPU utilization, as in [2]. We evaluate throughput based on FLOPS/s, a metric used in previous works [4], [7], [15], but also compute achieved bandwidth on the FPGAs and the GPU similar to [2], [11], where we consider data movement via streams in computing FPGA bandwidth. We also measured the energy draw of the platforms using the Xilinx `xbutil` tool for the FPGAs, and the `nvidia-smi` CLI tool for the H100 GPU.

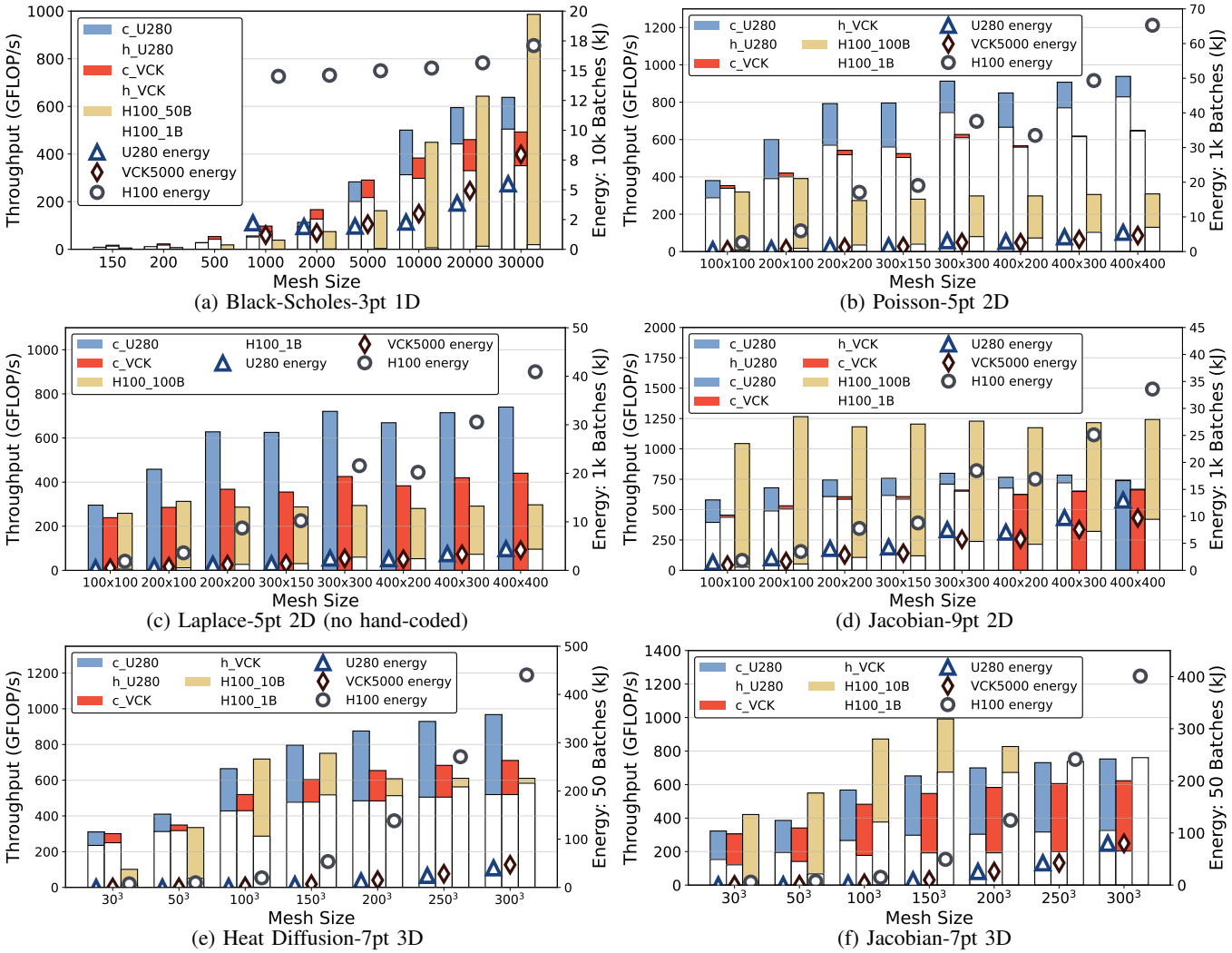
TABLE IV: Baseline model parameters. U: U280, V: VCK5000, H: handcoded, C: code-generated, p superscripts: D: DSP bounded, B: Memory bounded.

Application		**Dependent**			**User Defined**		
		Freq	G_{dsp}	L_{PE}	p	l_{axis}	l_{hls}
Black-Scholes	U-H	276	13	21	66^D	-	-
	U-C	282	13	21	66^D	1024	10
	V-H	300	3	11	48^B	—	—
	V-C	300	3	11	48^B	16384	10
Poisson2D	U-H	257	8	28	78	—	—
	U-C	284	8	28	81	8192	10
	V-H	300	3	10	54^B	—	—
	V-C	300	3	10	55^D	16384	10
Laplace2D	U-C	300	9	24	90^D	4096	50
	V-C	300	4	10	56^D	16384	10
Jacobian2D	U-H	300	43	37	21^D	—	—
	U-C	300	43	37	21^D	4096	360
	V-H	300	12	14	18^D	—	—
	V-C	300	12	16	18^D	16384	10
Diffusion3D	U-H	253	33	30	24^B	-	-
	U-C	300	15	40	48^B	1024	10
	V-H	300	9	16	24^B	—	—
	V-C	300	6	14	34^D	2048	10
Jacobian3D	U-H	251	14.5	30	27^B	—	—
	U-C	300	33	33	27^D	1024	10
	V-H	300	5.4	14	14^B	—	—
	V-C	300	9	19	22^D	8192	10
PW-Adv3D	V-C	300	48	23	4^D	16	10

A. 1-Dimensional Black-Scholes-3pt

The Black-Scholes European option pricing iterative solver uses a 1D 3-point 2nd-order stencil based on the implementation in [9]. The VCK5000 had lower G_{dsp} and L_{Pe} due to the more capable DSP blocks. Generated designs matched or outperformed the hand-coded baselines.

Fig. 5a compares throughput, showing the code-generated designs outperforming the hand-coded versions, with the highest throughput on the U280, of 637.94 GFLOP/s achieved for a mesh size of 30000. The H100 (`50B`) outperforms the U280 only for mesh sizes above 10000, reaching a maximum bandwidth of 138.53 GB/s (4.1% of peak). This is due to the low arithmetic intensity of this application. In practice, a 5k mesh size is beyond the practical size for iterative Black-Scholes [9], [31]. The U280 is also shown to be 6× as energy efficient as H100 (`50B`).



(a) Black-Scholes-3pt 1D

(b) Poisson-5pt 2D

(c) Laplace-5pt 2D (no hand-coded)

(d) Jacobian-9pt 2D

(e) Heat Diffusion-7pt 3D

(f) Jacobian-7pt 3D

Fig. 5: Throughput (y-axis, higher is better) and Energy (y_2-axis, lower is better) for different applications with increasing mesh sizes – h_: hand-coded, c_: code-generated.

B. 2-Dimensional Stencils

We implemented three 2D applications with loopback enabled: Poisson-5pt, Laplace-5pt, and Jacobian-9pt. Poisson-5pt and Jacobian-9pt are compared to hand-coded versions from [2], [4], while Laplace-5pt was only code-generated.

Poisson-5pt, consists of a 5-point 2nd-order cross stencil. Code-generation packed higher p on the U280 than the hand-coded version due to better balanced resource usage, though limited by routing congestion. With more PEs, the loopback design maintained 31% higher throughput on average. On the VCK5000, the code-generated design marginally outperforms the hand-coded design due to better on-chip memory use. The U280 can leverage higher bandwidth HBM for loopback, thereby outperforming the VCK5000 (Fig. 5(b)). On the U280, 985 GFLOP/s throughput was achieved for a 300×575 mesh, outperforming the H100 (`100B`), on average by 155% while being, on average, 19× as energy efficient. The H100 achieved 1039 GB/s bandwidth (31% of peak).

Laplace-5pt is a 5-point 2nd-order stencil with lower FLOPS per cell than Poisson-5pt. However, it consumes more DSPs on both platforms, likely to maintaining II=1. The initial performance of the generated code was lower when the stencil was expressed as a compound operation as in Table II, but splitting it into two simpler expressions resulted in a 20% boost (L_{PE} = 24) on the U280. DSP usage limits p on both platforms.

The H100, limited by memory bandwidth due to low compute intensity, peaked at 312.9 GFLOP/s, reaching a bandwidth of 729 GB/s (21.1% of peak). In contrast, the U280 achieved 773 GFLOP/s for a 300×520 mesh with $l_{axis} = 4096$, reaching 1023.35 GB/s bandwidth, outperforming the GPU by 109% on average with 7.3× the energy efficiency.

Jacobian-9pt, a more compute-intensive 9-point 2nd-order stencil [4], is ideal for GPU-FPGA comparison. On the U280, the code-generated design achieves the same G_{dsp} and L_{pe} as the hand-coded design and packs similar p. On the VC5K000, the code-generated version achieves almost identical performance to the hand-coded version. On the U280, with loopback,

Algorithm 2 RTM Forward-Pass [2], [33]

for all i = 0, i < n_{iter}, i++ **do** do
  $K_1 = f_{pml}(Y_{25pt}, \rho, \mu) \times dt; T = Y + K_1/2$
  $K_2 = f_{pml}(T_{25pt}, \rho, \mu) \times dt; T = Y + K_2/2$
  $K_3 = f_{pml}(T_{25pt}, \rho, \mu) \times dt; T = Y + K_3$
  $K_4 = f_{pml}(T_{25pt}, \rho, \mu) \times dt$
  $Y = Y + K_1/6 + K_2/3 + K_3/3 + K_4/6$
end for

$l_{axis} = 4096$ and $l_{hls} = 360$, without any frequency loss, performing 20% better on average than the hand-coded version. Code-generation achieves 798 GFLOP/s for a 300×300 mesh and 375.94 GB/s bandwidth (see Fig. 5(d)). The H100 GPU (`100B`) outperforms the U280 by 64% in terms of throughput, achieving 1242.5 GFLOP/s with 892 GB/s bandwidth (26.7% of peak), but the U280 and VCK5000 are 2.1× and 3.1× as energy efficient, respectively.

C. 3-Dimensional Stencils

We implemented two 3D benchmarks and compared them to manual versions from [27] and [2].

Heat Diffusion-7pt solves the classic heat equation using a 7-point stencil. The hand-coded version, based on [2], took several weeks to implement and optimize to achieve the parameters in Table IV. The initial code-generated version, completed in under a week, had lower G_{dsp}(17.375) and higher L_{pe}(53) compared to the hand-coded version. Breaking the kernel into multiple simpler terms, improved G_{dsp} and L_{pe} by 13.7% and 24.5% respectively, allowing higher p, boosting performance by 50–120%. On the U280, the code-generated design achieves 31–86% better performance compared to hand-coded, which could not be improved despite diligent coding. This highlights the advantage of code-generation, which avoids design pitfalls. On the U280, the generated version outperforms the H100 (`10B`) due to GPU memory bottlenecks (see Fig. 5(e)), and delivers 16× the energy efficiency. The H100 reached a maximum of 1029 GB/s bandwidth (30.7% of peak).

Jacobian-7pt is another 3D 7-point solver. The code-generated design uses more DSPs than the hand-coded designs from [2], [32]. However, the hand-coded designs only achieve II=2 on both platforms, demonstrating sensitivity to toolchain changes. The code-generated designs achieve II=1 at 300 MHz (both U280 and VCK5000), trading higher G_{dsp} and L_{pe} for higher performance, with identical p on U280 but higher p on VCK5000 due to balanced resource use. The code-generated design outperforms the hand-coded designs by 119% on the U280 and 181% on the VCK5000. Conversely, the H100 (`10B`) outperformed the U280 by an average of 19.2%, especially for mesh sizes between $100^3 - 150^3$. Like Jacobian-2D, this app exploits the GPU better due to higher computational intensity. The H100 reaches 1013.83 GFLOP/s throughput with 1068 GB/s bandwidth (31.9% of peak). Still, the U280 is 9× as energy efficient.

D. Multi-Stencil Applications

We implemented two multi-stencil applications, which are more industrially relevant and representative.

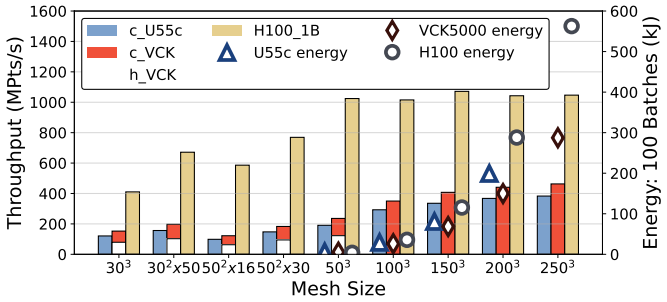


Fig. 6: RTM-FP : Throughput (y-axis, higher is better) and Energy (y_2-axis, lower is better) – `h_`: hand-coded, `c_`: code-generated.

RTM-FP (Reverse Time Migration–Forward Pass) is an industry-level application based on the algorithm in [33] modified as Alg. 2, that uses 3D meshes (Y, T, $K1..K4$), each with six-dimensional elements. Except Y, all are intermediate meshes updated via the $fpml$ using a 25-point, 4th-order 3D stencil. Scalars ρ and μ are also 3D. We used OPS with multidimensional meshes (multidim-2). Due to tool issues, only AMD Vitis 2022.1 would build this application, and we had to target the AMD Alveo U55c instead of the U280 (identical as a device, save for the 16GB HBM and no DDR memory). The hand-coded designs from [2], [32] would only build for the VCK5000. Like Jacobian-3D, the hand-coded version only reached II=2 due to tool issues. The code-generated design outperformed the hand-coded design by 48% on the VCK5000 (Fig. 6), peaking at 463 MPts/s throughput with 224 GB/s bandwidth. Note the VCK5000 does not have HBM. The H100 achieved 2× the throughput (1072 MPts/s) with 491 GB/s bandwidth (14.6% of peak). The VCK5000 is more energy efficient than the H100.

PW-Advection, a 27-point 2nd-order multi-stencil solver was implemented solely with code-generation on the VCK5000 for comparison with previous work [8]. We were able to pack $p = 4$, limited by DSP resources. the generated design achieves 271.46 GFLOP/s (4762.43 MPts/s) for a 250^3 mesh-over 7× as fast as the design in [8](see Table I). The H100 achieved 1036.41 GFLOP/s throughput with 787.06 GB/s bandwidth (23.5% of peak), due to the application's high compute intensity (57 FLOPs per mesh point).

E. Observations and Design-Space Exploration

Across applications, the code-generated FPGA designs achieve comparable or better performance than hand-coded designs, and outperform state-of-the-art published work, as detailed in Table I. Code-generation was also shown to be effective in porting to the newer VCK5000 architecture. For 3D stencils, the gains are higher, as the hand-coded versions struggle to achieve II=1. Manual tuning for each hand-coded application took 2 to 4 days, for example, when targeting the VCK5000 instead of the U280. Code generation only requires a configuration file change to switch the target platform, resulting in a suitable design within a few hours. Furthermore, manual tuning of hand-coded designs sometimes fails to achieve balanced resource usage, resulting in increased area or a subsequent build failure. Systematic dataflow optimizations through

TABLE V: Resource Usage % Heatmap. Absolute values reported in the artifacts

	U280							VCK5000						
Application	**LUT**	**LUTRAM**	**FF**	**BRAM**	**URAM**	**DSP**	**Acv Frq**	**LUT**	**LUTRAM**	**FF**	**BRAM**	**URAM**	**DSP**	**Acv Frq**
BlackScholes H	56.1	26.8	45.3	13.2	8.3	76.3	276	15.9	1.0	20.8	6.5	82.9	79.3	300
BlackScholes C	60.8	34.6	42.0	18.8	0.0	76.1	282	18.4	21.6	21.4	7.2	31.1	58.5	300
Poisson-2D H	70.1	15.5	53.7	11.3	65.0	55.6	257	17.5	1.0	23.1	6.5	93.3	89.0	300
Poisson-2D C	73.2	18.1	54.8	56.9	0.0	59.3	284	21.0	3.0	25.7	50.4	7.8	95.1	300
Laplace-2D C	67.5	21.6	48.7	28.4	39.6	73.9	300	14.5	6.0	16.7	51.2	7.8	96.8	300
Jacobian-2D H	46.1	4.0	37.4	11.3	17.5	80.4	300	7.1	3.2	10.9	10.2	31.1	89.0	300
Jacobian-2D C	46.3	4.2	36.7	19.5	10.8	80.6	300	8.1	3.4	14.9	19.8	7.8	89.7	300
Diffusion-3D H	50.7	9.4	36.2	15.4	80.0	70.9	256	12.9	6.8	14.7	100.0	0.0	88.8	300
Diffusion-3D C	65.5	20.4	47.9	51.2	0.0	67.1	300	14.0	7.1	15.4	82.2	0.0	94.0	300
Jacobian-3D H	38.1	4.4	27.8	22.0	67.5	35.1	300	7.4	1.7	8.9	18.1	90.7	31.3	300
Jacobian-3D C	46.4	9.2	36.3	23.8	22.5	80.9	300	11.9	5.5	19.2	55.5	3.9	87.9	300

code-generation show significant benefits for the multi-loop application, enabling fast design space exploration. Overall, the loopback pipeline is effective for smaller mesh sizes; however, it can adversely affect frequency. The U280 generally outperforms the VCK5000 due to its much larger resource capacity and HBM. Still, for applications with higher compute intensity, the VCK5000 matches or even exceeds the U280's performance due to the denser DSP block capabilities. On a larger Versal architecture device, code-generated designs are expected to compete more strongly with the GPU due to the more capable DSP blocks enabling higher p in the same area, compared to the U280.

Though the automated Domain Space Exploration (DSE) is out of the scope of this work, our systematic approach to fine-tuning on top of the initial estimation of a single p achieved balanced resource utilization with better performance. The resource usage results indicate that DSPs are the most critical, followed by on-chip memory, as reflected in both platforms (see Table V). One notable resource imbalance in code-generated designs is low usage of URAM. However, this can be fixed simply by the user through the setting `"supported_internal_storage" : ["uram", ... ]` in the configuration JSON file. Since thecode-generated designs were not limited by internal memory in this case, this was not significant. The hand-coded Jacobi-3D app used significantly fewer DSP blocks due to not meeting II=1.

VI. Conclusion

In this paper, we presented a new DSL-based framework that automates the generation of high-level synthesis code for structured-mesh-based stencil applications targeting modern FPGAs. HLS code for FPGAs is generated from a high-level declaration in the OPS DSL, combining state-of-the-art techniques and new optimizations. Key features include (1) the use of a new LLVM/Clang LibTooling-based source-to-source translation process to generate optimized FPGA source with radical, ad-hoc, and dynamic optimizations, (2) a new *window-buffer chaining* algorithm to automate the creation of optimal memory buffers between stencil points, and (3) a novel *deep-pipeline loopback* design to reduce HBM read/write overhead, hence improving throughput. The framework was used to generate code for several applications, including non-trivial multi-stencil solvers, targeting AMD Alveo U280 (and a U55c for RTM-FP), and Versal VCK5000 FPGAs. Performance of the generated designs was compared with hand-tuned versions as well as with optimized GPU implementations on an Nvidia H100 GPU. Across all applications, code-generated designs outperformed hand-coded designs by 15–56% on the U280 and 3–60% on the VCK5000. In most cases, code-generated designs outperform the H100 GPU in lower compute-intensity settings. For applications where the H100 performs better, generated designs on the FPGA are more energy-efficient: minimum 4× and 3×, average 14× and 10× on the U280 and VCK5000 respectively. The code generation approach enables rapid design space exploration with different resources and configurations that would otherwise consume significant time if performed manually. Future work will investigate incorporating batching, tiling (for solving large mesh sizes), and application to implicit solvers. The FPGA and GPU source code developed in this paper are available as open-source software at [34].

Acknowledgments

We are grateful to Kamalavasan Kamalakkannan for his advice in evaluating and comparing the hand-coded designs.

References

[1] M. Bouaziz and S. A. Fahmy, "Benchmarking floating point performance of massively parallel dataflow overlays on AMD Versal compute primitives," in *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2025, pp. 99–103.

[2] K. Kamalakkannan, G. R. Mudalige, I. Z. Reguly, and S. A. Fahmy, "High-level fpga accelerator design for structured-mesh-based explicit numerical solvers," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 1087–1096.

[3] H. M. Waidyasooriya and M. Hariyama, "Multi-fpga accelerator architecture for stencil computation exploiting spacial and temporal scalability," *IEEE Access*, vol. 7, pp. 53 188–53 201, 2019.

[4] H. M. Waidyasooriya, Y. Takei, S. Tatsumi, and M. Hariyama, "OpenCL-based FPGA-platform for stencil computation and its optimization methodology," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 5, pp. 1390–1402, 2017.

[5] C. Brugger, C. de Schryver, N. Wehn, S. Omland, M. Hefter, K. Ritter, A. Kostiuk, and R. Korn, "Mixed precision multilevel monte carlo on hybrid computing systems," in *IEEE Conference on Computational Intelligence for Financial Engineering Economics (CIFEr)*, 2014, pp. 215–222.

[6] S. Omland, M. Hefter, K. Ritter, C. Brugger, C. De Schryver, N. Wehn, and A. Kostiuk, *Exploiting Mixed-Precision Arithmetics in a Multilevel Monte Carlo Approach on FPGAs*, 2015, pp. 191–220.

[7] J. De Fine Licht, A. Kuster, T. De Matteis, T. Ben-Nun, D. Hofer, and T. Hoefler, "StencilFlow: Mapping large stencil programs to distributed spatial computing systems," in *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Feb. 2021, pp. 315–326.

[8] G. Rodriguez-Canal, N. Brown, M. Jamieson, E. Bauer, A. Lydike, and T. Grosser, "Stencil-HMLS: A multi-layered approach to the automatic optimisation of stencil codes on FPGA," in *Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W)*, 2023, p. 556–565.

[9] E. László, Z. Nagy, M. B. Giles, I. Reguly, J. Appleyard, and P. Szolgay, "Analysis of parallel processor architectures for the solution of the Black-Scholes PDE," in *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2015, pp. 1977–1980.

[10] T. Becker, O. Mencer, S. Weston, and G. Gaydadjiev, *Maxeler Data-Flow in Computational Finance*. Springer International Publishing, 2015, pp. 243–266.

[11] K. Kamalakkannan, G. R. Mudalige, I. Z. Reguly, and S. A. Fahmy, "High throughput multidimensional tridiagonal system solvers on FPGAs," in *International Conference on Supercomputing (ICS)*, Jun. 2022.

[12] M. Bouaziz, M. Samet, and S. A. Fahmy, "A dataflow overlay for Monte Carlo multi-asset option pricing on AMD Versal AI Engines," in *ISC High Performance Research Paper Proceedings*, 2025.

[13] P. Rawat, M. Kong, T. Henretty, J. Holewinski, K. Stock, L.-N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan, "SDSLc: a multi-target domain-specific compiler for stencil computations," in *International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing(WOLFHPC)*, 2015.

[14] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong, "Polyhedral-based data reuse optimization for configurable computing," in *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, 2013, p. 29–38.

[15] Y. Chi, J. Cong, P. Wei, and P. Zhou, "SODA: Stencil with optimized dataflow architecture," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018, pp. 1–8.

[16] T. Ben-Nun, J. de Fine Licht, A. N. Ziogas, T. Schneider, and T. Hoefler, "Stateful dataflow multigraphs: a data-centric model for performance portability on heterogeneous architectures," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2019.

[17] I. Z. Reguly, G. R. Mudalige, M. B. Giles, D. Curran, and S. McIntosh-Smith, "The ops domain specific abstraction for multi-block structured grid computations," in *Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*, 2014, pp. 58–67.

[18] O. Reiche, M. A. Özkan, R. Membarth, J. Teich, and F. Hannig, "Generating fpga-based image processing accelerators with hipacc: (invited paper)," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2017, pp. 1026–1033.

[19] 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 *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FGPA)*, 2019, p. 242–251.

[20] G. Natale, G. Stramondo, P. Bressana, R. Cattaneo, D. Sciuto, and M. D. Santambrogio, "A polyhedral model-based framework for dataflow implementation on FPGA devices of iterative stencil loops," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2016, pp. 1–8.

[21] OPS DSL Documentation (Latest). Accessed: 2024-09-13. [Online]. Available: https://ops-dsl.readthedocs.io/en/latest/

[22] I. Z. Reguly, G. R. Mudalige, and M. B. Giles, "Loop tiling in large-scale stencil codes at run-time with ops," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 4, pp. 873–886, 2018.

[23] G. Balogh, G. Mudalige, I. Reguly, S. Antao, and C. Bertolli, "Op2-clang: A source-to-source translator using clang/llvm libtooling," in *IEEE/ACM Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, 2018, pp. 59–70.

[24] M. Treinish, I. Carvalho, G. Tsilimigkounakis, and N. Sá, "rustworkx: A high-performance graph library for python," *Journal of Open Source Software*, vol. 7, no. 79, p. 3968, 2022.

[25] W. E. Winkler, "String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage," in *Proceedings of the Section on Survey Research, American Statistical Association*, 1990.

[26] G. Dura and A.-M. Moşneagu, "Numerical approximation of black-scholes equation," *Analele ştiinţifice ale Universităţii "Alexandru Ioan Cuza" din Iaşi. Matematică (Serie nouă)*, vol. 56, pp. 39–64, 2010.

[27] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Implicit and explicit optimizations for stencil computations," in *Workshop on Memory System Performance and Correctness (MSPC)*, 2006, p. 51–60.

[28] (2023, June) Alveo U280 Data Center Accelerator Card Data Sheet (DS963) (v1.7). [Accessed 07-06-2024]. [Online]. Available: https://docs.amd.com/r/en-US/ds963-u280/Alveo-Product-Details

[29] (2024) VCK5000 Versal Development Card. [Accessed 13-01-2025]. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/vck5000.html

[30] (2023) NVIDIA H100 Tensor Core GPU. [Accessed 13-01-2025]. [Online]. Available: https://www.nvidia.com/en-gb/data-center/h100/

[31] M. Nurul Anwar and L. Sazzad Andallah, "A Study on Numerical Solution of Black-Scholes Model," *Journal of Mathematical Finance*, vol. 08, no. 02, pp. 372–381, 2018.

[32] (2023) StencilsOnFPGA: Modularised HLS(Vivado C/C++) based implementation of contrasting stencil applications targeting Xilinx FPGAs. Accessed: 2025-02-18. [Online]. Available: https://github.com/OP-DSL/StencilsOnFPGA

[33] R. Clayton and B. Engquist, "Absorbing boundary conditions for acoustic and elastic wave equations," *Bulletin of the Seismological Society of America*, vol. 67, pp. 1529–1540, Dec. 1977.

[34] (2025) OPS-HLS repository. [Online]. Available: https://github.com/benielT/OPS

APPENDIX

A. Abstract

This artifact evaluation appendix contains details of the workflow, artifacts, and the details of the result data artifacts. Evaluators require an Nvidia H100 GPU and one of the U280 or VCK5000 hardware to verify the results.

B. Artifact check-list (meta-information)

- **Algorithm:** OPS - DSL for generating code for both GPU, FPGA platforms, containing AST generation, AST matching, 'Window buffer chaining algorithm', optimization iterations, and an automated end-to-end build workflow for the user applications.
- **Compilation:** Python 3 $>$ 3.8. **For H100** - Require NVHPC 23.7 toolkit, GNU Make, and OPS-DSL (NOTE: OPS and OPS_batched are packed as sub-modules in the artifacts).
  For FPGAs (U280, VCK5000) - Require Vitis 2022.2 or Vitis 2022.1 with compatible runtime, GNU C++ 9.4(or compatible GNU C++)
- **Transformations:** Artifact contains OPS, OPS_batched (only used for GPU batched applications) translator. Installation guidelines provided. OPS translators require Python, and setup instructions and a script are given.
- **Binary: H100 Binaries** - Require Compatible runtime with CUDA 12.6/NVHPC 23.7. Results produced on a Debian 6 environment with NVidia Driver 560.35.03 (binaries should be cross executable in a compatible Linux environment).
  FPGA Binaries - Apps contains prebuilt XCLBIN files for both code-generated designs and hand-coded designs, and host-side binaries. XCLBINs are specific for specific targets. Details can be found in codegen_apps/README.md.
- **Run-time environment:** set OPS_HLS_ARTIFACT_DIR as the extracted artifacts directory. **For H100** - NVHPC-23.7 toolkit (or compatible NVHPC toolkit). Linux-based OS (tested on Debian 6 amd_64).
  For FPGA - Vitis 2022.1 or 2022.2 environment with Xilinx license. Specific platforms xilinx_u280_gen3x16_xdma_1_202211_1 for U280 and xilinx_vck5000_gen4x8_xdma_2_202210_1. (Optional) xilinx_u55c_gen3x16_xdma_3_202210_1 for U55c, which is used only for RTM.
- **Hardware:** NVidia H100 required. Either AMD(Xilinx) Alveo U280 or AMD(Xilinx) Versal VCK5000 is required (require both platforms for full evaluations). AMD(Xilinx) Alveo U55c - Optional.
- **Execution:** Strongly recommend running the applications as an exclusive user of the hardware to evaluate profile data. FPGA environment, the user needs exclusive access to the hardware. NOTE: No need for sudo privilege to run applications.
- **Metrics:** Runtime(us) and energy usage(kJ). Expected values are available in each app under the directory profile_data. Makefiles have options to switch the runtime, power profiling.
- **Output:** CSV files with runtime and power usage will be generated inside the profile_data directory.
- **Experiments:** Experiment flow is detailed in the README.md files. Make sure the generated FPGA design achieves similar clock performance.
- **How much disk space is required:** 2 GB for artifacts. Building an FPGA design using Vitis requires 100GB+.
- **How much time is needed to prepare workflow (approximately):** If hardware, runtime requirements are met, (10 minutes) for modifying the environment source scripts.
- **How much time is needed to complete experiments: GPU-H100** - requires 5-10 minutes for each app to run. Build instantly. **FPGAs** require 5-10 minutes for each app to run. Build takes 5-10 hours each app.
- **Publicly available:** Yes. Github: https://github.com/benielT/ops-hls-pact25-artifact Zenodo: https://doi.org/10.5281/zenodo.16785478
- **Code licenses:** Artifact licensed under MIT License and underlying OPS, OPS_batched are licensed under BSD 3-Clause.
- **Workflow automation:** Makefile together with run_scripts.sh connected with other bash scripts provides automated experimentation and result generation for multiple parameters.
- **Archived:** https://doi.org/10.5281/zenodo.16785478

C. Description

1) How to access:

- Download the archived artifact. The estimated size is around 900 MB.
- Clone git sub-modules inside the artifact folder (check README.md). - Detailed instructions are available.
- If the required hardware is not available, you can try requesting access from AMD Heterogeneous Accelerated Compute Clusters (link: https://www.amd-haccs.io/).

2) Hardware dependencies: NVIDIA H100 is required for GPU apps. AMD (Xilinx) Alveo U280, AMD (Xilinx) Versal VCK5000 are required for FPGA designs. AMD(Xilinx) Alveo U55c is optional.

3) Software dependencies: python3 ($>$3.8), GNU Make. Additional for H100: NVHPC 23.7 toolkit and supported runtime, drivers. Additional for FPGAs: Vitis 2022.1 or 2022.2 toolkit, GNU C++ 9.4 compiler (or compatible GNU compiler).

D. Installation

Detailed instructions are available in README.md. Additional setup for GPU apps in gpu_apps/README.md and for FPGA apps in codegen_apps/README.md.

E. Experiment workflow

Check app execution workflow in the main README.md file for details.

F. Evaluation and expected results

Each app has already generated runtime and power profiles under the profile_data directory. Additionally unified_data_artifacts folder contains data used in the publication. Check the generated results after the experiment run and compare with the results initially given in the artifacts. NOTE: 'make clean' will remove existing results. Therefore, make sure you have a copy before 'clean'

G. Experiment customization

Each app has a run_script.sh or run_hls_script.sh, which will be working with Makefile. You can customize parameters there for experiment customization.

H. Methodology

Submission, reviewing and badging methodology:

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