Stream Swizzling: Extending HLS for Efficient Non-Affine Stream Permutations

Chengyue Wang
Computer Science
University of California, Los Angeles
Los Angeles, California, USA
chengyue@ucla.edu

Jonathan Xue
Electrical and Computer Engineering
University of California, Los Angeles
Los Angeles, California, USA
jonathanxue@g.ucla.edu

Lance Giang
Computer Science
University of California, Los Angeles
Los Angeles, California, USA
ltgiang994@g.ucla.edu

Shinju Ju
Electrical and Computer Engineering
University of California, Los Angeles
Los Angeles, California, USA
shinju0916@g.ucla.edu

Yingquan Wu
MBZUAI Institute of Foundation Models
Sunnyvale, California, USA
icodywu@gmail.com

Jason Cong
Computer Science
University of California, Los Angeles
Los Angeles, California, USA
cong@cs.ucla.edu

Abstract

High-Level Synthesis (HLS) substantially improves hardware design productivity but is at a disadvantage when precise control over cycle-level hardware behavior is required. Streaming FFTs expose this limitation: they contain a sequence of non-integer-affine stream permutations that can introduce pipeline bubbles when expressed in conventional HLS. To address this gap, we present HSPL (Hardware Stream-Permutation Language), a core intermediate language that extends HLS with explicit stream-permutation interfaces. Given high-level stream-permutation specifications, our xDSL/MLIR-based compiler realizes each permutation by *stream swizzling*: it derives XOR-based bank and slot address maps under which the permuted stream is bank-conflict-free, together with safe early-start offsets. HSPL preserves the cycle-accurate semantics of these interfaces, which conventional HLS cannot explicitly express or convey to downstream RTL-generation backends. To enable these semantics in the closed-source HLS tool, we introduce an agent-assisted post-HLS RTL refinement layer that provides the missing backend support.

This work makes the following main contributions. First, HSPL introduces a stream-permutation abstraction for module-level pipeline parallelism, providing an intuitive but powerful primitive that future HLS tools can support natively. Second, this work provides HP-FFT, to our knowledge the first high-performance HLS-based streaming FFT generator whose generated designs achieve lower latency than vendor RTL IP in most cases. Third, we present the first agent-guided post-HLS RTL refinement flow to implement features not natively supported by the HLS tool, expanding the practical reach of HLS for cycle-critical designs.

CCS Concepts

• **Hardware → High-level and register-transfer level synthesis**; *Hardware description languages and compilation.*



This work is licensed under a Creative Commons Attribution 4.0 International License.
PACT '26, Chicago, IL, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2915-7/2026/10
https://doi.org/10.1145/3838684.3846858

Keywords

High-Level Synthesis, Stream Permutation, Swizzle, Layout, FFT, MLIR, xDSL, Agentic Compiler, LLM Agent

ACM Reference Format:
Chengyue Wang, Jonathan Xue, Lance Giang, Shinju Ju, Yingquan Wu, and Jason Cong. 2026. Stream Swizzling: Extending HLS for Efficient Non-Affine Stream Permutations. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '26), October 19–22, 2026, Chicago, IL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3838684.3846858

1 Introduction

High-Level Synthesis (HLS) improves hardware design productivity by raising the design entry point from cycle-level RTL to software programs [23]. However, it gives programmers less direct control over cycle-by-cycle hardware behavior than RTL. This long-standing limitation is particularly important for streaming accelerators, where sustaining throughput requires precise coordination between neighboring modules. To compensate for this limitation, Vitis HLS augments C++ with hardware primitives and directives: `DATAFLOW` enables module-level pipeline parallelism, while FIFO and ping-pong (PIPO) channels coordinate communication. These primitives, however, do not directly express the access order and timing required by non-affine stream permutations such as FFT bit reversal. For example, FIFOs preserve arrival order, while PIPOs impose full-buffer handoff [2]. Implementing these permutations through manual indexing and separately handshaken processes introduces pipeline stalls and additional latency.

On the other hand, SPIRAL and SGen map SPL formulas to streaming architectures [22, 32], and show how to derive conflict-free memory layouts for stream permutations [29]. However, the RTL-based generator relies on predefined RTL templates, which can lead to suboptimal latency and resource utilization. To address the limitations of existing HLS interfaces and template-based RTL generators, we propose an approach that combines their strengths: stream-permutation methods provide conflict-free layouts, while an HLS backend jointly schedules fused arithmetic regions and maps their operations to target-device resources.

We present HSPL, a Hardware Stream-Permutation Language for static stream permutations expressed as invertible linear maps

over index bits in $\mathbb{F}_2$. These maps are generally non-affine in integer index space. HSPL also takes a stream width specifying the target throughput in elements per cycle. From these inputs, our xDSL/MLIR-based compiler derives conflict-free bank-address mappings and safe early-start offsets for the resulting access schedules. Our compiler implements these interfaces through *stream swizzling*: the write and read address maps stream the permutation without bank conflicts using XOR-based mappings, much as CuTe swizzles XOR shared-memory address bits. HSPL preserves these interface semantics throughout lowering. We also introduce an `agent` dialect for constrained post-HLS RTL refinement to realize interface behavior beyond Vitis HLS's native support without modifying its closed-source backend.

This work makes four contributions:

- We introduce HSPL, a missing HLS interface primitive for non-affine stream permutations, including XOR-based index maps, with explicit producer–consumer synchronization beyond native FIFO and PIPO semantics.
- We introduce an `agent` dialect for post-HLS RTL refinement, treating generated RTL as an editable compiler-backend artifact to implement interface features beyond native HLS support for cycle-critical streaming designs.
- We build HP-FFT, an xDSL/MLIR-based streaming FFT compiler that reduces latency in cycles by up to 3.42× and 3.28× over the state-of-the-art RTL generators SPIRAL and SGen, respectively. Compared with AMD's HLS IP, Vitis SSR [4], HP-FFT reduces latency by up to 4.27× and increases throughput by up to 14.4×, while using fewer DSPs and FFs. Compared with AMD's RTL IP, LogiCore [3], it reduces latency in cycles by up to 1.27× and achieves a post-route $F_{\max}$ of up to 516.5 MHz, 2.18× LogiCore's best $F_{\max}$ in our sweep.
- We evaluate physical address maps from three CUTLASS-style XOR-based swizzles [26] as stream-permutation microbenchmarks, achieving ideal frame II and lower frame latency than affine compilers CODO [42] and Stream-HLS [9].

2 Background: Permutations in Streaming FFTs

2.1 The FFT and its SPL Representation

We use the FFT as the running example throughout this section. The N-point Discrete Fourier Transform (DFT) is the linear map $X = \mathrm{DFT}_N\, x$, where $(\mathrm{DFT}_N)_{kn} = e^{-j2\pi kn/N}$. The Fast Fourier Transform (FFT) [13] reduces the $O(N^2)$ complexity of the DFT to $O(N \log N)$ by recursively factoring DFT_N into a product of structured and sparse matrices. These matrix factorizations can be expressed through the Signal Processing Language (SPL) introduced by SPIRAL [38]. SPL represents a transform as an algebraic formula built from three primitive matrices and two types of operations.

Primitive Matrices. Each SPL formula is composed of three kinds of primitive matrix. **Base transforms** (e.g. DFT_r) are small, dense arithmetic kernels, used to represent a butterfly computation in FFT. **Diagonal matrices** (T) encode point-wise multiplications, used to represent twiddle factors; the stage-specific twiddle E_ℓ used below is an instance of T. **Permutation matrices** (L, R, …) reorder elements without performing any arithmetic; they are SPL's representation of data movement and the focus of this paper. The stage-local permutation Q_ℓ used below is a composition of stride permutations L (Eq. (2)).

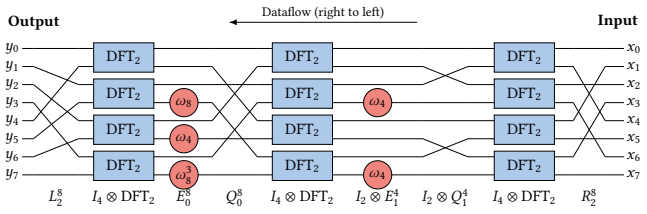


Figure 1: SPL maps directly to a fully unrolled datapath, with $\omega_m = e^{-j2\pi/m}$.

Matrix operations. Primitive matrices are combined by two operations. **Matrix multiplication** ($A \cdot B$) composes operations right-to-left. **Tensor (Kronecker) product** ($A \otimes B$) represents the parallelism found in the FFT: $I_n \otimes B$ represents the creation of n independent copies of B applied to disjoint slices of the input.

For an FFT size $N = r^t$, the iterative radix-r Cooley–Tukey algorithm [13] is represented in SPL for streaming FFTs as [21]:

$$\mathrm{DFT}_{r^t} = L_r^{r^t} \left(\prod_{\ell=0}^{t-1} (I_{r^{t-1}} \otimes \mathrm{DFT}_r) \left(I_{r^\ell} \otimes (E_\ell^{r^{t-\ell}} \cdot Q_\ell^{r^{t-\ell}}) \right) \right) R_r^{r^t}, \tag{1}$$

When read right-to-left, Equation (1) defines a dataflow pipeline through which the input vector flows:

- $R_r^{r^t}$ – **digit-reversal permutation.** An N-point permutation that reorders the input by reversing the base-r digits of each index. For radix 2 this is the traditional bit-reversal.
- $L_r^{r^t}$ – **stride permutation.** An N-point stride permutation; throughout, the subscript denotes the stride and the superscript the matrix size. Thus L_m^{mn} transposes a conceptual $n \times m$ matrix laid out in row-major order.
- $I_{r^{t-1}} \otimes \mathrm{DFT}_r$ – **butterfly operations.** An r-point DFT (the "butterfly") applied to every group of r consecutive samples in $r^{t-1} = N/r$ independent parallel copies.
- $I_{r^\ell} \otimes (E_\ell^{r^{t-\ell}} \cdot Q_\ell^{r^{t-\ell}})$ – **per-stage operator.** Within each stage, $E \cdot Q$ is applied to r^ℓ blocks of size $r^{t-\ell}$. $E_\ell^{r^{t-\ell}}$ is the diagonal matrix of twiddle factors that perform element-wise multiplications between the input vector and the corresponding twiddle factor. A twiddle factor is a unit-magnitude complex coefficient $\omega_m^k = e^{-j2\pi k/m}$ that applies a phase rotation required by the FFT decomposition. For $0 \le \ell < t-1$, the permutation matrix $Q_\ell^{r^{t-\ell}}$ is

$$Q_\ell^{r^{t-\ell}} = L_{r^{t-\ell-1}}^{r^{t-\ell}} \cdot \left(I_r \otimes L_r^{r^{t-\ell-1}} \right). \tag{2}$$

For $\ell = t-1$, $E_{t-1}^r = Q_{t-1}^r = I_r$.

2.2 From SPL to Streaming Datapath

As an example, for an 8-point radix-2 FFT, $N = 8$, $r = 2$, $t = 3$, Eq. (1) instantiates as

$$\begin{aligned}\mathrm{DFT}_8 &= L_2^8 \left(\prod_{\ell=0}^{2} (I_4 \otimes \mathrm{DFT}_2) \left(I_{2^\ell} \otimes (E_\ell^{2^{3-\ell}} \cdot Q_\ell^{2^{3-\ell}}) \right) \right) R_2^8 \\ &\overset{\text{expand}}{=} L_2^8 \cdot (I_4 \otimes \mathrm{DFT}_2) \left(E_0^8 \cdot Q_0^8 \right) \\ &\quad \cdot (I_4 \otimes \mathrm{DFT}_2) \left(I_2 \otimes (E_1^4 \cdot Q_1^4) \right) \\ &\quad \cdot (I_4 \otimes \mathrm{DFT}_2) \left(I_4 \otimes (E_2^2 \cdot Q_2^2) \right) \cdot R_2^8 \\ &\overset{\text{simplify}}{=} L_2^8 \cdot (I_4 \otimes \mathrm{DFT}_2) \cdot E_0^8 \cdot Q_0^8 \\ &\quad \cdot (I_4 \otimes \mathrm{DFT}_2) \cdot (I_2 \otimes E_1^4) \cdot (I_2 \otimes Q_1^4) \\ &\quad \cdot (I_4 \otimes \mathrm{DFT}_2) \cdot R_2^8 .\end{aligned} \tag{3}$$

Every factor of this chained product maps directly to a circuit component, as shown in Figure 1. Read right-to-left, the simplified expansion describes the sequence of operations applied to the 8-element input vector: the input bit-reversal permutation R_2^8, three butterfly columns $I_4 \otimes \mathrm{DFT}_2$ separated by stage-dependent permutations Q_ℓ and twiddles E_ℓ, and the output stride permutation L_2^8. In the figure, every permutation (R, Q, L) is realized as fixed wiring, while the DFT_2 butterflies and the diagonal E matrices perform the actual arithmetic.

For large transform sizes, this fully unrolled mapping is impractical because the circuit becomes enormous, so the datapath must be folded and reused over multiple cycles. Folding turns matrix-vector operations into temporal ones: the bit-reversal R_2^N, for example, now receives its input over many cycles and must buffer the samples in memory before reading them out in bit-reversed order.

Folding therefore raises the bank-conflict problem: data must be laid across memory banks so every permutation reads and writes its parallel elements per cycle without collision. For the $N = 8$ example over 2 cycles, R_2^8 buffers elements $\{e_0, e_1, e_2, e_3\}$ then $\{e_4, e_5, e_6, e_7\}$ across 4 banks and must read them back in bit-reversed order $\{e_0, e_4, e_2, e_6\}, \{e_1, e_5, e_3, e_7\}$ collision-free to sustain the 2-cycle-per-batch throughput.

This bit reversal is not a single affine function of the integer index: $f(0) = 0$ and $f(1) = 4$ would force $f(i) = 4i$, contradicting $f(2) = 2$. Nevertheless, it is linear over $\mathbb{F}_2$ when expressed in terms of index bits.

Püschel et al. [29] address this problem for *linear permutations*: permutations of $N = 2^n$ elements whose action on the n-bit index is an invertible linear map over $\mathbb{F}_2$. For $N = 8$, the bit-reversal R_2^8 becomes the 3×3 binary permutation matrix

$$P = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}, \tag{4}$$

where P maps an input address $x = (x_2, x_1, x_0)^T$ to the output address $y = (y_2, y_1, y_0)^T = Px = (x_0, x_1, x_2)^T$, exactly the bit-reversed index. With permutations expressed in this form, the bank-assignment and slot-assignment problems both reduce to choosing complementary $\mathbb{F}_2$ projections of the index, which is what makes a systematic conflict-free layout resolution possible, as detailed in the next section.

B_0	B_1	B_2	B_3	
e_0	e_4	e_2	e_6	← cyc 0 read
e_5	e_1	e_7	e_3	← cyc 1 read

Figure 2: Conflict-free banked layout for R_2^8 at $SW = 4$ after the helper-matrix construction of Step 2. Each cycle reads one slot across four distinct banks; an output crossbar handles within-cycle reordering.

2.3 Bank Conflict-Free Permutation

Following Püschel et al. [29], we split each address into two parts: a *spatial* part that indexes the parallel lanes within a single cycle, and a *temporal* part that indexes which cycle the element belongs to. In addition to the input and output coordinates x and y, we introduce a memory access coordinate $z = (z_{\mathrm{addr}}, z_{\mathrm{id}})^T$ that addresses memory: z_{id} is the spatial part identifying which of the parallel banks is touched, and z_{addr} is the temporal part identifying the slot inside that bank. The permutation matrix P factorizes into a *write matrix* $\mathbf{M}$ and a *read matrix* $\mathbf{N}$,

$$z = \mathbf{M}\,x = \mathbf{N}\,y, \qquad P = \mathbf{N}^{-1}\,\mathbf{M}, \tag{5}$$

where $\mathbf{M}$ realizes the input crossbar plus the bank write addressing, and $\mathbf{N}$ realizes the bank read addressing plus the output crossbar.

For the $N = 8$ bit-reversal at streaming width 4 over 2 cycles, the memory has 4 banks of 2 slots each, so z, x, and y partition into a 1-bit temporal part (top) and a 2-bit spatial part (bottom) as

$$\begin{aligned} z &= \begin{pmatrix} z_{\mathrm{addr}} \\ \hline z_{\mathrm{id}} \end{pmatrix} = \begin{pmatrix} z_2 \\ \hline z_1 \\ z_0 \end{pmatrix}, \quad x = \begin{pmatrix} x_{\mathrm{cycle}} \\ \hline x_{\mathrm{lane}} \end{pmatrix} = \begin{pmatrix} x_2 \\ \hline x_1 \\ x_0 \end{pmatrix}, \\ y &= \begin{pmatrix} y_{\mathrm{cycle}} \\ \hline y_{\mathrm{lane}} \end{pmatrix} = \begin{pmatrix} y_2 \\ \hline y_1 \\ y_0 \end{pmatrix}. \end{aligned} \tag{6}$$

Bank conflict-free condition. Splitting $\mathbf{M}$ along the same cycle/port split gives the block form

$$\mathbf{M} = \begin{pmatrix} \mathbf{M}_4 & \mathbf{M}_3 \\ \mathbf{M}_2 & \mathbf{M}_1 \end{pmatrix}, \qquad \begin{aligned} z_{\mathrm{addr}} &= \mathbf{M}_4\,x_{\mathrm{cycle}} + \mathbf{M}_3\,x_{\mathrm{lane}}, \\ z_{\mathrm{id}} &= \mathbf{M}_2\,x_{\mathrm{cycle}} + \mathbf{M}_1\,x_{\mathrm{lane}}. \end{aligned} \tag{7}$$

Within a single cycle, x_{cycle} is fixed across all lanes, so $z_{\mathrm{id}} = c + \mathbf{M}_1\,x_{\mathrm{lane}}$ for some constant c. For the SW parallel writes in that cycle to land on SW distinct banks, the map $x_{\mathrm{lane}} \mapsto z_{\mathrm{id}}$ must be a bijection, which is equivalent to $\mathbf{M}_1$ being invertible over $\mathbb{F}_2$. The same argument on $\mathbf{N}$ gives the symmetric requirement that $\mathbf{N}_1$ is invertible.

Step 1: try $\mathbf{M} = P$, $\mathbf{N} = I$. The natural choice satisfies $P = \mathbf{N}^{-1}\mathbf{M}$ trivially, but does not always satisfy the invertibility condition. For the bit-reversal R_2^8 at $SW = 4$ (one cycle bit, two port bits), partitioning P from (4) yields

$$\mathbf{M}_4 = (0), \quad \mathbf{M}_3 = (0\ \ 1), \quad \mathbf{M}_2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \quad \mathbf{M}_1 = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix},$$

so $\mathrm{rank}(\mathbf{M}_1) = 1$: two of the four lanes map to the same bank within a single cycle, a write conflict.

Step 2: introduce a helper matrix H. Setting $\mathbf{M} = HP$ and $\mathbf{N} = H$ preserves $P = \mathbf{N}^{-1}\mathbf{M}$ while leaving H as a free parameter that we choose to make $\mathbf{M}_1$ invertible. For our R_2^8 example, taking $H = I + E(2,0)$ (the identity with an extra 1 at position $(2,0)$) implements an XOR-based address swizzle and yields

$$\mathbf{M}_1^{\text{new}} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = I_2,$$

which is invertible as required. Because the added 1 sits in the bottom-left block of H, the bottom-right block H_1 stays the identity, so $\mathbf{N}_1 = H_1 = I$ is invertible too, completing the conflict-free factorization. Figure 2 shows the resulting bank/slot layout.

General algorithm and hardware decomposition. The same construction generalizes to larger transforms: such an H always exists for any invertible binary index matrix P [29, 30]. The resulting $\mathbf{M}$ and $\mathbf{N}$ still mix spatial routing with temporal addressing; a final decomposition factors each into a spatial factor that maps to a crossbar and a temporal factor that maps to RAM addressing logic, which is what makes the $\mathbb{F}_2$ layout deployable on real hardware. The construction is not specific to FFT permutations: any invertible P over $\mathbb{F}_2$ qualifies. The same algebraic representation underlies Triton's Linear Layouts, which model GPU tensor layouts as linear maps over $\mathbb{F}_2$ [43]. A CuTe `Swizzle<B,M,S>` [26], for instance, XORs one B-bit address field into another, i.e., $P = I \oplus E$ for a single off-diagonal block E; Section 5.7 applies the same factorization to such swizzles.

3 Hardware Stream Permutation

3.1 Interface Abstractions for Module-Level Pipeline Parallelism

In Vitis HLS, a non-inlined C++ function represents a hardware module and its arguments become module ports. C++ has an inherent limitation in expressing module-level pipeline parallelism: its semantics are sequential, so a function accessing an array must complete all reads and writes before it returns, and neighboring modules cannot naturally overlap. Vitis HLS recovers module-level pipeline parallelism through the `#pragma HLS DATAFLOW` pragma, which analyzes dataflow between successive functions or loops and lets producer and consumer tasks run in parallel [2]. Two channel primitives carry data through module interfaces: `hls::stream` FIFOs and ping-pong (PIPO) buffers. Figure 3(a)(b) summarizes both, and Figures 4 and 5 show the corresponding C++. The listings use `dtype = float` and `vec32 = hls::vector<dtype, 32>`.

A FIFO preserves only arrival order, so it cannot express any permutation, such as bit reversal, where the consumer reads in a different order than the producer wrote. A PIPO can store data and is the physical basis for temporal reordering, but it has three limitations: (i) *hard to write*: realizing even a bit reversal forces the designer to hand-write the crossbar, banked-RAM address functions, and pragmas (Figure 5); (ii) *scattered semantics*: the permutation's access order and early-start offset are implicit in the indexing and pragmas across producer and consumer loops. HLS coordinates the DATAFLOW processes through inferred channel protocols, whose handshakes can introduce pipeline bubbles through interfacing overhead even when each process is pipelined; and (iii) *full-buffer protocol*: the PIPO swaps buffers at full-buffer granularity, blocking the consumer until the producer fills a whole buffer even when the early-start offset is much smaller, which inflates end-to-end latency beyond what the permutation alone requires [2].

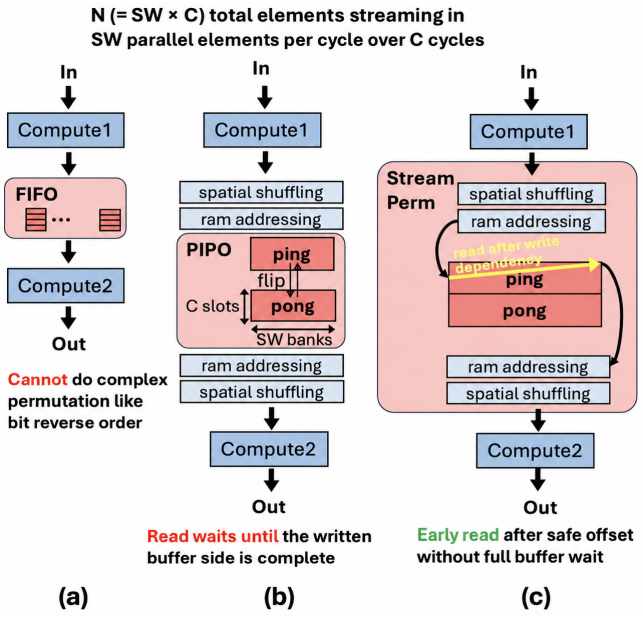


Figure 3: Interface abstractions for streaming permutations. (a) HLS FIFO: ordered-token handoff. (b) HLS PIPO: full-buffer handoff with complex manual indexing and reduced throughput due to pipeline bubbles. (c) HSPL Stream Permutation: early-start handoff with lower latency, a simple API, and sustained target throughput.

HSPL therefore introduces a third interface abstraction, an explicit *stream-permutation interface* (Figure 3(c)). Given a logical permutation P and a stream width, it produces the RAM read/write address orders, the spatial crossbar interconnects, and an early-start offset, and passes that information to interface synthesis. Because Vitis HLS is a commercial tool whose RTL generation we cannot modify, we apply post-csynth RTL patching to realize these semantics. The primitive gives three benefits: (i) *lower latency*: the consumer starts after the early-start offset rather than waiting for a full PIPO frame; (ii) *bubble-free pipelining*: in our generated FFT designs, RTL refinement restores the target II and eliminates variation in frame II and latency (Table 4); and (iii) *a clean abstraction at both source and IR levels*: at the source level, the user supplies only P and the stream width (Figure 6); inside the compiler, an MLIR dialect carries the same interface specification as a first-class IR for analysis and lowering.

3.2 HSPL Dialect: Typed Interface IR

HSPL is an xDSL/MLIR dialect [15, 19] that represents stream-permutation interfaces as first-class typed objects. A *leaf* stream module processes one SW-wide stream value per iteration under HLS pipelining; a *composite* connects leaves through ordered FIFOs or RAM-based PIPOs for temporal reordering.

Figure 4: The FIFO channel preserves producer order and cannot express a bit-reversal interface.

```
void func1_fifo(hls::stream<vec32>&in,hls::stream<vec32>&fifo){
  for (int i = 0; i < 8; i++) {
    #pragma HLS PIPELINE II=1
    fifo.write(compute1(in.read()));
  }
}
void func2_fifo(hls::stream<vec32>&fifo,hls::stream<vec32>&out){
  for (int i = 0; i < 8; i++) {
    #pragma HLS PIPELINE II=1
    out.write(compute2(fifo.read()));
  }
}
void top_fifo(hls::stream<vec32>&in,hls::stream<vec32>&out){
  #pragma HLS DATAFLOW
  hls::stream<vec32> stream_channel;
  func1_fifo(in, stream_channel);
  func2_fifo(stream_channel, out);
}
```

Figure 5: Manual HLS rewrite (pseudocode). Interface semantics are scattered across lane routing, RAM addressing, banked storage, and pragmas. Remaining index bits and consumer-side routing/address logic are omitted.

```
void func1_manual(hls::stream<vec32>& in, dtype ram[32][8]) {
  for (int i = 0; i < 8; i++) {
    #pragma HLS PIPELINE II=1
    vec32 data = compute1(in.read());
    ap_uint<3> cycle = i;
    for (int p = 0; p < 32; p++) {
      #pragma HLS UNROLL
      ap_uint<5> bank = p, src = bank;
      ap_uint<3> slot;
      src[0] = bank[0] ^ cycle[2]; // Lane routing
      src[1] = bank[1] ^ cycle[1];
      ...
      slot[0] = bank[2] ^ cycle[0]; // RAM address
      ...
      ram[bank][slot] = data[src];
    }
  }
}
// func2_manual: analogous RAM read and lane routing (omitted).
void top_manual(hls::stream<vec32>& in, hls::stream<vec32>& out) {
  #pragma HLS DATAFLOW
  dtype pingpong_buf[32][8];
  #pragma HLS ARRAY_PARTITION variable=pingpong_buf complete dim=1
  func1_manual(in, pingpong_buf);
  func2_manual(pingpong_buf, out);
}
```

Each interface's consumer carries a `consume_pattern` attribute recording its access order, stream width, and early-start offset from the producer's first write to the consumer's first read. Using the $\mathbb{F}_2$ factorization of Section 2.3, the compiler derives each PIPO's conflict-free bank/slot layout, per-cycle read/write addresses, and input/output crossbars. For these schedules, the compiler compares each element's write and read cycles and chooses an offset that places every read after its corresponding write becomes visible, including the RAM's write-to-read visibility delay. A fusion pass merges adjacent leaves with compatible shape and width into HLS scheduling regions, retaining separate modules only where a channel protocol is required.

Table 1: HSPL interface types.

HSPL type	Meaning
`!hspl.stream_fifo<N,T>`	Ordered stream of N elements of type T.
`!hspl.stream_pipo<F, !hspl.ram<B,S,T>>`	Stream of F banked-RAM buffers.
`!hspl.wire<SW,T>`	SW elements passed between two ops, within same-cycle, zero-latency.
`!hspl.ram<B,S,T>`	Logical RAM with B banks and S slots.
`!hspl.ram_ref<B,S,T>`	Reference to one RAM object.

Figure 6: HSPL stream-permutation interface. The channel declares the logical permutation and stream width; the compiler derives the bubble-free realization.

```
#include <hspl/stream_perm.hpp>
using vec32 = hls::vector<float, 32>;
using PermMatrix = hspl::gf2::perm_matrix<8,
    0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80>;
void func1_hspl(hls::stream<vec32>& in,
    hspl::stream_perm<vec32, 32, PermMatrix>& ch) {
  for (int i = 0; i < 8; i++) {
    #pragma HLS PIPELINE II=1
    ch.write(compute1(in.read()));
  }
}
void func2_hspl(hspl::stream_perm<vec32, 32, PermMatrix>& ch,
    hls::stream<vec32>& out) {
  for (int i = 0; i < 8; i++) {
    #pragma HLS PIPELINE II=1
    out.write(compute2(ch.read()));
  }
}
void top_hspl(hls::stream<vec32>& in, hls::stream<vec32>& out) {
  #pragma HLS DATAFLOW
  hspl::stream_perm<vec32, 32, PermMatrix> channel;
  func1_hspl(in, channel);
  func2_hspl(channel, out);
}
```

Leaf bodies contain routing, RAM access, and compute operations, including opaque computations. HSPL assumes each leaf achieves its target II under HLS pipelining; local scheduling, register insertion, and operator binding remain HLS's responsibility. Table 1 distinguishes logical streams, same-cycle values, and RAM storage.

3.3 Source-Level HSPL API

The source-level C++ API in `<hspl/stream_perm.hpp>` represents a stream-permutation interface as a channel object, which users can declare without writing compiler IR:

```
hspl::stream_perm<StreamValue, SW, PermMatrix> channel;
```

`StreamValue` is the complete C++ type transferred by each read or write, such as `hls::vector<float, 32>`. Both endpoints must use this type; the current implementation requires FP32 lanes accessible through `operator[]`. `SW` sets the parallel lane count; the current implementation requires a power of two satisfying $2 \le SW \le N$. `PermMatrix` encodes an invertible $\mathbb{F}_2$ matrix for bit reversal, stride permutations, or XOR swizzles. The frontend lowers this interface specification through the compiler's permutation pipeline to derive the HSPL representation, bank layout, early-start

offset, and HLS implementation. Each channel has one producer and one consumer; multiple independent channels are supported.

As an example, consider $N = 16$ samples with $SW = 4$. The matrix below describes four-bit bit reversal:

```
using vec4 = hls::vector<float, 4>;
using PermMatrix = hspl::gf2::perm_matrix<4,
    0b0001, 0b0010, 0b0100, 0b1000>;     // 4x4 bit-reversal
hspl::stream_perm<vec4, 4, PermMatrix> channel;
```

Figure 6 replaces the manual implementation in Figure 5 with a single typed channel declaration. The compiler derives the datapath and handshake requirements from this specification, preserving interface semantics through fusion and RTL refinement.

For FP32 samples $x_0, \ldots, x_{15}$, the producer writes explicitly typed stream values. In a separate task, the consumer reads the bit-reversed sequence shown in the comments:

```
channel.write(vec4{x0,  x1,  x2,  x3});
channel.write(vec4{x4,  x5,  x6,  x7});
channel.write(vec4{x8,  x9,  x10, x11});
channel.write(vec4{x12, x13, x14, x15});

vec4 y0 = channel.read(); // {x0, x8, x4, x12}
vec4 y1 = channel.read(); // {x2, x10, x6, x14}
vec4 y2 = channel.read(); // {x1, x9, x5, x13}
vec4 y3 = channel.read(); // {x3, x11, x7, x15}
```

For a pure permutation between existing native streams, `hspl::connect(in, channel)` and `hspl::connect(channel, out)` declare the two endpoints directly. The compiler connects the generated permutation to those streams without extra pass-through tasks or interface FIFOs. Producers and consumers that perform computation retain the `write/read` form in Figure 6; either endpoint may also use a direct connection.

4 HSPL Compilation and RTL Refinement

4.1 Frontend Paths and HLS Generation

This section presents our xDSL-based HSPL compilation and RTL refinement flow, including HP-FFT for streaming FFT accelerators. Figure 7 summarizes the overall flow. HP-FFT's main frontend starts from an FFT configuration, lowers through the `hpfft`, `spl`, and `hspl` dialects, and then to structural and HLS dialects before emitting Vitis HLS C++. A second source-level entry parses HLS C++ with `hspl::stream_perm` declarations into the same `hspl` dialect; this path targets HLS users who want to use the stream-permutation layer outside the context of HP-FFT, and it is the entry used for the swizzle units in Section 5.7.

HP-FFT consists of the following stages:

(1) **FFT configuration.** The input configuration specifies the transform size, stream width, datatype, and target platform. The compiler first represents these parameters as a high-level FFT transform.

(2) **SPL dialect.** HP-FFT reimplements an SPL-style (discussed in Section 2.1) symbolic representation for FFTs. This stage expresses the FFT pipeline with mathematical operators. A streaming FFT can be represented as a composition of local DFT/butterfly computation, stage permutations, and twiddle diagonals.

(3) **HSPL dialect.** The compiler lowers FFT formulas into HSPL operations for local computation, spatial routing, and temporal reordering, connected by typed FIFO and PIPO interfaces. RAM-based PIPO interfaces additionally carry their bank/slot shape, write and read access functions, and early-start offsets. The compiler then runs the HSPL fusion pass (described in Section 3.2) so that only interfaces requiring a channel protocol survive as DATAFLOW edges. Unlike the manual decomposition in Figure 5, this lowering retains the permutation's access order and early-start offsets while forming HLS scheduling regions, so datapath fusion and handshake refinement use the same interface specification.

(4) **HLS and agent layer.** HSPL primitives are converted into `hlslib` calls that the emitter can directly print, including crossbars, RAM reads and writes, ROMs for twiddle factors, compressed octant ROMs, and different butterfly variants. A Vitis-oriented IR (`hls_vitis`) further represents the top-level FFT function and HLS functions that do not map to an `hlslib` primitive. MLIR structural dialects (`scf`, `func`,

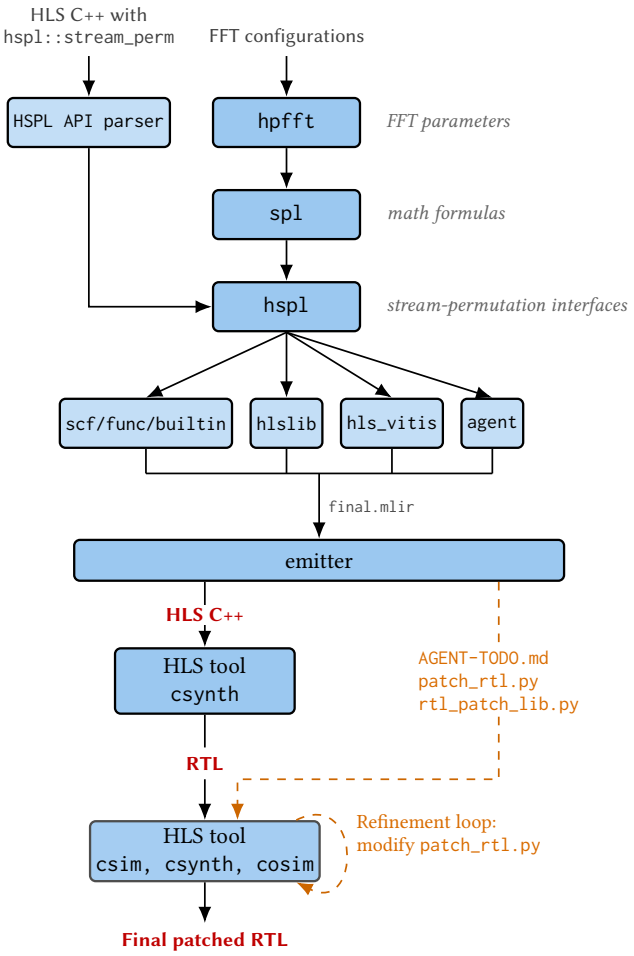


Figure 7: HP-FFT compiler flow: two frontends feed `hspl`; the emitter produces a Vitis HLS C++ project and an RTL refinement contract that drives a constrained agent-guided refinement loop.

builtin) provide control flow and module structure. Alongside this HLS representation, an agent dialect carries the RTL refinement contract for backend behavior that cannot be conveyed through HLS C++ alone.

(5) **HLS C++ emission.** The emitter reads the final IR and generates an HLS C++ project, including the kernel file, testbench, Tcl scripts, and agent-guided refinement artifacts.

4.2 Agent IR and Refinement Loop

Fusion eliminates unnecessary interfacing between processes, but the remaining RAM-based interfaces still inherit Vitis HLS's PIPO handshakes. The refinement contract carries their early-start offsets into the RTL stage: vetted primitives remove stalls between successive frames and enable consumers to read before a complete frame has been written. Vitis HLS RTL generation is a black box: it renames variables and instantiates many modules per design. Across our fused configurations, the patches modify up to 14 Verilog files and up to 72 module instances (Table 3). The agent's role is to identify *where* in the generated RTL to patch; *what* and *how* to patch is fixed by compiler-emitted primitives. A regex-based fallback is available for users who cannot run an agent, but it is tied to the RTL patterns it was developed against and does not generalize to unseen designs (Section 5.4). The agent IR carries four operations, summarized in Table 2; Figure 8 shows an instance for the fused FP32 $N = 256$, $SW = 2$ design together with its rendered AGENT-TODO.md.

Table 2: Four operations in the agent dialect.

Op	Role
agent.context	Loads project context: FFT description, problem size, stream width, target II, pre-/post-patch latency estimates.
agent.goal	A goal the design must reach (e.g., constant II/latency, an II target, a latency-reduction target).
agent.action	A backend step the agent may execute (e.g., run Vitis, apply an RTL patch). Carries a checks region that holds the constraints to verify after the step and any nested follow-up actions.
agent.constraint	A check of the enclosing action's result (e.g., correctness, II, latency, latency-reduction), with a severity and an on_fail policy.

The prototype emits this contract as three files:

(1) **AGENT-TODO.md** – the task document. Renders the context, goals, action and constraint definitions, and the ordered action sequence; each step is parameterized by compiler-produced facts (e.g. early-start offset lists, and total writes per patchable banks_*).

(2) **patch_rtl.py** – the refinement plan, the only artifact the loop lets the agent modify. Restricted to a flat sequence of lib.<primitive>(literal args) calls; no free-form RTL edits, to prevent unreviewed Verilog rewrites.

(3) **rtl_patch_lib.py** – a frozen RTL patch library: discovery helpers, predefined RTL mutation primitives, and a self-check that rejects loops, custom helpers, and any non-whitelisted call in patch_rtl.py.

Together, these artifacts constrain the refinement action space by construction: every refinement plan consists of calls to predefined compiler-emitted primitives. The agent selects primitives and literal arguments but cannot edit Verilog directly or introduce new mutation logic. The functional behavior of the resulting RTL is checked through C/RTL co-simulation.

Figure 8: Schematic agent IR (top) and rendered AGENT-TODO.md (bottom) for the fused FP32 $N = 256$, $SW = 2$ design. The latency values follow the measured Patch 1 to Patch 2 reduction, 579 to 477 cycles (Table 4); instance-specific scheduling parameters are omitted.

```
// -- agent IR --
"agent.context"() {id="fft",
  params={N=256, SW=2, ii_target=128,
          latency_pre=579, latency_reduction=102}
}
"agent.goal"() {id="latency_reduction", params={target=102}}
"agent.action"() ({"agent.constraint"() {id="latency_reduced",
                   severity="error", on_fail=["iterate", 10]}}) {
  id="patch_rtl_pingpong_early_start",
  params={ram_s2p_pipos=[...]}
}

// -- rendered AGENT-TODO.md --
# Context
  This design is a streaming radix-2 Cooley-Tukey
  FFT in Vitis HLS C++ for Versal Premium xcvp1802
  at 250 MHz. N=256, SW=2, II=128,
  latency_pre=579, latency_reduction=102.
# Goals
## goal:latency_reduction  target=102, scope=@FFT_TOP
# Library
## action:patch_rtl_pingpong_early_start
  Replace PIPO full-frame barrier with dual-port early-start FSM.
## constraint:latency_reduced
  Check: post-patch latency drops by goal:latency_reduction.target.
# Sequence
## Step 3. action:patch_rtl_pingpong_early_start
  Per-PIPO variable, wtr, ar, and total_writes:
  [instance-specific scheduling parameters omitted]
  - [ ] constraint:latency_reduced  on_fail=iterate x 10
```

5 Evaluation

5.1 Experimental Setup

Our FFT implementation is the full version of HP-FFT, previously presented as work in progress [34]. Our experiments use Vitis HLS 2025.1 and Vivado 2025.1. We target a Versal Premium xcvp1802 at a 250 MHz target. FFT first-to-last latency (first input to last output) and initiation interval (II) are collected from C/RTL co-simulation over 16 input batches. The main FFT comparison reports resource usage after RTL synthesis for HP-FFT and all four baselines: SPIRAL [22], SGen [32], LogiCore FFT [3], and Vitis SSR FFT [4]. Post-route resource and timing results are reported in the frequency-scaling experiments (Section 5.6). The swizzle comparison in Section 5.7 instead reports HLS synthesis estimates, taken before RTL refinement for HSPL. Fixed-point correctness is checked against SPIRAL on identical input streams: across all configurations, the maximum per-output deviation between our generated design and the SPIRAL reference is at most 4 least-significant bits in RTL simulation. The FFT refinement experiments use Claude Opus 4.7 [8] for agent-assisted post-HLS RTL refinement.

5.2 Agentic Patch Scope

Table 3 reports the scope of the final agentic patch script (`patch_rtl.py`) for each FFT configuration, measured by the number of distinct Verilog source files modified and the number of module instances rewritten. The floating-point and fixed-point variants have identical touched-file and modified-instance counts in the fused designs. The refinement modifies between 2 and 14 Verilog files and between 2 and 72 module instances per configuration. Fusion changes the number of remaining stream interfaces, so the patch scope need not grow monotonically with streaming width. These results show that our contract-guided refinement can perform nontrivial edits on HLS-generated RTL. All evaluated configurations completed the refinement flow successfully under compiler-emitted contracts and a frozen patch library.

Table 3: RTL patch scope per FFT configuration: the number of distinct Verilog source files (`.v`) modified and the number of instances modified by the agent.

N	SW	#.v Files	#instances (Fixed16)	#instances (FP32)
256	2	8	9	9
256	4	7	10	10
256	8	6	13	13
256	16	4	4	4
256	32	3	3	3
256	64	2	2	2
1,024	2	10	11	11
1,024	4	9	12	12
1,024	8	8	15	15
1,024	16	7	22	22
1,024	32	5	5	5
1,024	64	4	4	4
16,384	2	14	15	15
16,384	4	13	16	16
16,384	8	12	19	19
16,384	16	11	26	26
16,384	32	10	41	41
16,384	64	9	72	72

5.3 Agent RTL Refinement Effect

Table 4 reports the FP32 II and latency for each (N, SW) configuration at three refinement levels: no patch, Patch 1, and Patch 2. *No patch* is the unmodified Vitis HLS-generated RTL. *Patch 1* fixes the producer/consumer handshake on every ping-pong buffer so that adjacent frames no longer stall each other. *Patch 2* additionally lets each consumer stage begin as soon as enough elements are written, instead of waiting for the entire buffer to be filled.

Without refinement, back-pressure between dataflow stages produces batch-to-batch variation in II and latency in all 18 FP32 configurations. Patch 1 reaches the ideal $II = N/SW$ in every configuration and stabilizes latency across batches. Patch 2 retains the ideal II and yields a further latency speedup of 1.00–1.23× over Patch 1; the two patches have equal latency at $N = 256$, $SW = 64$. Relative to the maximum unpatched latency, their combined speedup is 1.14–1.50×.

5.4 Why an Agent: Scripts and Model Sensitivity

A natural alternative to the agent is a deterministic post-processing script. Our deterministic comparator uses regex patterns inferred from previously generated RTL to locate signals and modules, whose names can change across designs through HLS naming, mangling, and truncation. To measure how far such patterns generalize, we had an LLM develop a regex-based patch script from three FP32 FFTs ($N = 256$; $SW = 8/16/32$) and applied it to 15 configurations not used during development: only 3 of 15 passed RTL co-simulation. After extending the script to cover every FFT variant, none of the three CuTe swizzle units of Section 5.7 passed RTL co-simulation (0 of 3). After extending it once more to cover those units, we asked an adversarial LLM to break it through semantics-preserving variable renaming; across five rounds of attack and repair, every round produced a new renaming that defeated the updated script. By contrast, our `agent` dialect uses LLM reasoning to analyze each newly generated design and closes the loop with RTL simulation feedback, revising its patch whenever a new RTL pattern appears, thereby making the shipped tool more robust to previously unseen designs.

Table 4: Initiation interval and first-to-last latency (cycles) under three refinement levels for FP32. The no-patch column reports min / avg / max across batches, with averages at integer-cycle precision. Patch 1 and Patch 2 report a single II and constant latency. Parenthesized speedups use the maximum no-patch latency.

N	SW	No patch (min/avg/max)	Patch 1	Patch 2
Initiation interval (cycles)				
256	2	128 / 155 / 160	128	128
256	4	64 / 77 / 80	64	64
256	8	32 / 38 / 40	32	32
256	16	16 / 19 / 20	16	16
256	32	8 / 9 / 11	8	8
256	64	4 / 4 / 6	4	4
1,024	2	512 / 622 / 640	512	512
1,024	4	256 / 311 / 320	256	256
1,024	8	128 / 155 / 160	128	128
1,024	16	64 / 77 / 80	64	64
1,024	32	32 / 38 / 40	32	32
1,024	64	16 / 18 / 20	16	16
16,384	2	8,192 / 9,966 / 10,240	8,192	8,192
16,384	4	4,096 / 4,983 / 5,120	4,096	4,096
16,384	8	2,048 / 2,491 / 2,560	2,048	2,048
16,384	16	1,024 / 1,245 / 1,280	1,024	1,024
16,384	32	512 / 622 / 640	512	512
16,384	64	256 / 310 / 320	256	256
First-to-last latency (cycles)				
256	2	626 / 683 / 691 (1x)	579	477 (1.45x)
256	4	346 / 376 / 380 (1x)	323	276 (1.38x)
256	8	200 / 217 / 220 (1x)	189	169 (1.30x)
256	16	129 / 140 / 143 (1x)	122	115 (1.24x)
256	32	94 / 102 / 106 (1x)	90	88 (1.20x)
256	64	75 / 79 / 82 (1x)	72	72 (1.14x)
1,024	2	2,324 / 2,552 / 2,581 (1x)	2,133	1,729 (1.49x)
1,024	4	1,206 / 1,321 / 1,336 (1x)	1,111	914 (1.46x)
1,024	8	640 / 700 / 708 (1x)	593	499 (1.42x)
1,024	16	355 / 388 / 393 (1x)	332	289 (1.36x)
1,024	32	219 / 239 / 243 (1x)	208	190 (1.28x)
1,024	64	147 / 161 / 165 (1x)	140	133 (1.24x)
16,384	2	35,960 / 39,608 / 40,057 (1x)	32,889	26,633 (1.50x)
16,384	4	18,047 / 19,872 / 20,097 (1x)	16,512	13,391 (1.50x)
16,384	8	9,082 / 9,997 / 10,110 (1x)	8,315	6,761 (1.50x)
16,384	16	4,593 / 5,053 / 5,111 (1x)	4,210	3,439 (1.49x)
16,384	32	2,361 / 2,595 / 2,625 (1x)	2,170	1,790 (1.47x)
16,384	64	1,240 / 1,362 / 1,379 (1x)	1,145	960 (1.44x)

We also evaluated sensitivity to the choice of model. We ran Haiku 4.5 [5] at the setting labeled low, Opus 4.7 [8] at xHigh, and Opus 5 [6] and Sonnet 5 [7] at low/medium/high/xHigh on FP32 $N = 16{,}384/SW = 32$ and $N = 1{,}024/SW = 4$, three trials per setting (60 in total). These are the experiment's setting labels, without an assumed common mapping to model-specific API parameters. Fifty-seven trials produced RTL identical to the validated reference; the only non-identical outputs were the three Haiku 4.5 low trials on $N = 16{,}384/SW = 32$. This supports repeatability across the tested models and settings, with a reproducible exception for Haiku 4.5 on the larger design.

5.5 Comparison to State-of-the-Art

We evaluate the radix-2 streaming FFTs produced by our compiler flow against four baselines spanning the FFT design space: two SOTA academic generators (SPIRAL [22], SGen [32]) and two Xilinx vendor IPs (LogiCore [3], the RTL FFT IP, and Vitis SSR FFT [4], the HLS FFT IP). We sweep three axes: transform size $N \in \{256, 1{,}024, 16{,}384\}$, streaming width $SW \in \{2, 4, 8, 16, 32, 64\}$ (samples streamed in per cycle), and datatype (32-bit floating point and scaled 16-bit fixed point). For each design point we report the initiation interval in cycles, the latency in cycles from first input to last output, and resource usage after RTL synthesis (FF, LUT, BRAM18, DSP). "ERR" indicates a synthesis failure: SGen at small unrolling factors generates so many shift registers that synthesis fails. For the baselines, SPIRAL is configured with the default BRAM option (−1), and LogiCore is configured for the real-time mode. In Tables 5–10, parentheses report signed latency changes, $100(L - L_{\text{ref}})/L_{\text{ref}}\%$, rounded to one decimal place. The reference L_{ref} is SGen's latency at the same configuration, or SPIRAL's when SGen fails synthesis; negative values indicate lower latency and positive values indicate higher latency.

Across the six tables (Tables 5–10) we draw three comparisons, one per baseline category. (i) Against the academic generators (SPIRAL, SGen), our flow matches the ideal $II = N/SW$ at every entry; latency is lower at every design point and the gap widens with SW (e.g., 1.48× to 3.28× over SGen as SW goes from 2 to 64 at $N = 256$ in floating point); we use roughly 43–57% fewer FFs than SPIRAL in floating point at the cost of 3–22% more LUTs and 8–32% more DSPs (still fewer DSPs than SGen), and roughly 40% fewer DSPs in fixed point at small SW (e.g., 17 vs. SPIRAL's 28 at $N = 256, SW = 2$). (ii) Against the Xilinx LogiCore RTL FFT IP, our HLS-generated IP is surprisingly competitive: lower latency in all fixed-point configurations and in most floating-point configurations, with fewer DSPs everywhere. LogiCore leads on latency only when SW is large relative to N (e.g., N=256, $SW \geq 8$ and N=1,024, $SW \geq 32$ in FP32); at the other extreme, N=16,384, SW=2 FP32 (Table 10), LogiCore uses roughly 43× as many FFs and 10× as many LUTs, the extreme end of the BRAM/FF–LUT trade-off discussed below. For fixed-point configurations, we use fewer FFs and LUTs everywhere but more BRAMs in three of the four comparisons at $N \geq 1{,}024$. For floating-point configurations, the two designs make a different storage-mapping trade-off: in every case where we use more FFs, we use no BRAMs whereas LogiCore does, and in most cases where we use fewer FFs, we use more BRAMs. Exploring this trade-off is a natural direction for future work. (iii) Against the other HLS baseline (Vitis SSR FFT), we improve II by 7.2–14.4× and latency by 2.4–4.3× at $SW = 2$ while using fewer DSPs and FFs (e.g., floating-point $N = 1{,}024$: II 512 vs. 7,372, latency 1,729 vs. 7,375, 40 vs. 193 DSPs). LUT usage is lower in four of the six comparisons, with increases of 13% and 6% for FP32 $N = 256$ and $N = 1{,}024$, respectively. Vitis SSR does not reach the ideal II in the configurations reported here. On coverage, our flow and SPIRAL complete all 36 design points, while SGen, LogiCore, and Vitis SSR each have fewer reported results. A missing (N, SW) row denotes a configuration without a validated result in this evaluation.

Table 5: Scaled fixed-point, $N = 256$. II and first-to-last latency are in cycles. Parentheses: latency change vs. SGen.

Design	SW	II	Latency (f-to-l)	BRAM18	DSP	FF	LUT
SPIRAL	2	128	531 (+3.9%)	0	28	3,748	3,956
SPIRAL	4	64	306 (+5.5%)	0	48	6,049	5,634
SPIRAL	8	32	192 (+8.5%)	0	84	10,693	8,943
SPIRAL	16	16	132 (+10.0%)	0	156	20,094	15,713
SPIRAL	32	8	94 (+10.6%)	0	300	33,731	27,023
SPIRAL	64	4	70 (+9.4%)	0	588	55,179	46,607
SGen	2	128	511 (0%)	16	22	3,241	2,189
SGen	4	64	290 (0%)	28	40	6,099	3,727
SGen	8	32	177 (0%)	48	76	11,988	6,750
SGen	16	16	120 (0%)	80	148	22,851	13,466
SGen	32	8	85 (0%)	128	292	39,634	23,662
SGen	64	4	64 (0%)	192	580	65,692	39,939
LogiCore	2	128	545 (+6.7%)	10	28	6,887	6,297
LogiCore	4	64	289 (−0.3%)	16	36	8,521	9,102
Vitis SSR	2	1,013	1,098 (+114.9%)	0	28	7,106	13,062
Ours	2	128	437 (−14.5%)	2	17	4,222	4,605
Ours	4	64	227 (−21.7%)	0	28	6,267	7,249
Ours	8	32	124 (−29.9%)	0	56	10,685	12,329
Ours	16	16	73 (−39.2%)	0	120	17,337	21,362
Ours	32	8	46 (−45.9%)	0	259	27,334	34,901
Ours	64	4	32 (−50.0%)	0	584	35,354	47,215

We attribute these gains to two properties of our HLS-centered backend. First, expressing arithmetic in HLS C++ lets the backend pick the DSP mapping most appropriate for the target device. This can explain the lower fixed-point DSP counts noted in (i). Second, our compiler fuses adjacent stages whose interface does not require a stream buffer. We also trade the last FFT stage's buffer for direct FIFOs for reduced latency. As SW grows, more stages fall into one fused region. The HLS scheduler then co-schedules a longer chain into a single FSM, whereas SPIRAL and SGen lower their formula representations to RTL [22, 32]. Stage fusion is also studied in streaming permutation generators [31]; our distinction here is joint HLS scheduling of the fused arithmetic region. This can explain the latency advantage that widens with SW in (i).

5.6 Frequency Scaling

The comparisons above use a 250 MHz synthesis target. Here, we evaluate FP32 $N = 16{,}384$, $SW = 16$ after place-and-route. Table 11 sweeps targets from 250 to 550 MHz, with matching HLS scheduling and Vivado implementation targets for our design. Table 12 includes all additional clock pairs, extending our HLS target to 700 MHz and SPIRAL's implementation target to 750 MHz. We report post-route $F_{\text{max}} = 1000/(T_{\text{target}} - \text{WNS})$ in MHz, with the target period and routed WNS in ns, following AMD's methodology [1]. Performance comparisons use this F_{max} for each routed design, including implementations with negative WNS at their requested target.

Table 6: Scaled fixed-point, $N = 1{,}024$. II and first-to-last latency are in cycles. Parentheses: latency change vs. SGen.

Design	SW	II	Latency (f-to-l)	BRAM18	DSP	FF	LUT
SPIRAL	2	512	1,885 (+1.4%)	14	36	3,685	4,205
SPIRAL	4	256	996 (+2.3%)	12	64	6,986	7,189
SPIRAL	8	128	550 (+4.0%)	0	116	15,590	16,347
SPIRAL	16	64	324 (+5.9%)	0	220	27,014	24,881
SPIRAL	32	32	208 (+7.8%)	0	428	49,430	40,988
SPIRAL	64	16	142 (+9.2%)	0	844	85,423	69,703
SGen	2	512	1,859 (0%)	24	30	4,013	3,448
SGen	4	256	974 (0%)	40	56	7,490	5,031
SGen	8	128	529 (0%)	64	108	14,674	9,088
SGen	16	64	306 (0%)	112	212	28,104	17,472
SGen	32	32	193 (0%)	192	420	54,638	34,670
SGen	64	16	130 (0%)	320	836	98,076	62,496
LogiCore	2	512	1,915 (+3.0%)	10	36	9,034	9,955
LogiCore	4	256	1,023 (+5.0%)	20	48	11,523	13,817
Vitis SSR	2	3,746	4,089 (+120.0%)	12	36	9,101	18,765
Ours	2	512	1,683 (−9.5%)	20	21	5,306	6,839
Ours	4	256	855 (−12.2%)	10	36	8,637	12,789
Ours	8	128	441 (−16.6%)	8	72	14,427	19,214
Ours	16	64	235 (−23.2%)	0	152	25,532	31,079
Ours	32	32	133 (−31.1%)	0	328	41,449	53,392
Ours	64	16	79 (−39.2%)	0	705	71,004	92,583

Table 7: Scaled fixed-point, $N = 16{,}384$. II and first-to-last latency are in cycles. Parentheses: latency change vs. SGen (SPIRAL at $SW = 2$).

Design	SW	II	Latency (f-to-l)	BRAM18	DSP	FF	LUT
SPIRAL	2	8,192	28,721 (0%)	284	52	4,921	5,460
SPIRAL	4	4,096	14,440 (+0.2%)	280	96	9,124	9,930
SPIRAL	8	2,048	7,298 (+0.5%)	272	180	17,719	19,007
SPIRAL	16	1,024	3,724 (+0.8%)	256	348	34,927	36,946
SPIRAL	32	512	1,934 (+1.4%)	224	684	64,949	72,461
SPIRAL	64	256	1,036 (+2.4%)	192	1,356	128,164	134,325
SGen	2	ERR	ERR	ERR	ERR	ERR	ERR
SGen	4	4,096	14,406 (0%)	188	88	10,303	13,982
SGen	8	2,048	7,265 (0%)	216	172	20,007	15,187
SGen	16	1,024	3,694 (0%)	272	340	38,645	26,452
SGen	32	512	1,907 (0%)	384	676	75,601	51,329
SGen	64	256	1,012 (0%)	640	1,348	149,051	101,762
LogiCore	2	8,192	28,847 (+0.4%)	30	52	50,041	53,310
LogiCore	4	4,096	15,451 (+7.3%)	32	72	49,007	65,244
Vitis SSR	2	59,050	64,869 (+125.9%)	48	52	13,073	126,762
Ours	2	8,192	26,578 (−7.5%)	216	29	8,033	11,861
Ours	4	4,096	13,320 (−7.5%)	222	52	13,102	19,748
Ours	8	2,048	6,690 (−7.9%)	228	104	23,556	34,397
Ours	16	1,024	3,371 (−8.7%)	256	216	40,751	58,847
Ours	32	512	1,707 (−10.5%)	384	453	64,824	89,539
Ours	64	256	876 (−13.4%)	224	968	130,421	199,194

Our $F_{\max}$ rises from 255.8 MHz at a 250/250 MHz (HLS/Vivado) target pair to a maximum of 516.5 MHz at 550/550. Raising the HLS target further, to 600–700 MHz, does not improve this maximum: cycle latency grows from 3,576 to 3,584–3,642, and FF usage rises from 325.4K to 332.4–374.4K. SPIRAL reaches its highest $F_{\max}$ of 541.1 MHz at a 600 MHz implementation target, retaining its 4,056-cycle latency. LogiCore peaks at 236.5 MHz with a 450 MHz target.

At each design's best $F_{\max}$, all three have $II = 1{,}024$, so throughput is $16F_{\max}$ Msamples/s. Ours delivers 8,264.0 Msamples/s versus SPIRAL's 8,657.6 Msamples/s, 4.5% lower throughput, but its shorter cycle latency yields 6.92 versus 7.50 μs first-to-last latency, a 7.6% reduction. At these operating points, ours uses 15.5% fewer FFs and 11.6% fewer LUTs than SPIRAL, with 12.6% more DSPs. Relative to LogiCore, ours achieves 2.18× the frequency and throughput, with 6.92 versus 17.32 μs latency, a 60.0% reduction.

Table 8: Floating-point (FP32), $N = 256$. II and first-to-last latency are in cycles. Parentheses: latency change vs. SGen.

Design	SW	II	Latency (f-to-l)	BRAM18	DSP	FF	LUT
SPIRAL	2	128	723 (+2.6%)	12	28	28,093	22,719
SPIRAL	4	64	482 (+1.9%)	0	48	54,958	43,881
SPIRAL	8	32	368 (+5.4%)	0	84	104,363	81,153
SPIRAL	16	16	308 (+5.5%)	0	156	203,533	155,876
SPIRAL	32	8	270 (+5.1%)	0	300	392,541	300,939
SPIRAL	64	4	246 (+4.2%)	0	588	760,405	583,840
SGen	2	128	705 (0%)	34	44	32,335	23,810
SGen	4	64	473 (0%)	56	80	61,579	44,512
SGen	8	32	349 (0%)	96	152	117,619	84,748
SGen	16	16	292 (0%)	160	296	228,816	164,021
SGen	32	8	257 (0%)	256	584	441,188	313,877
SGen	64	4	236 (0%)	384	1,160	851,046	619,258
LogiCore	2	128	497 (−29.5%)	4	46	25,498	16,153
LogiCore	4	64	277 (−41.4%)	8	82	27,850	15,545
LogiCore	8	32	167 (−52.1%)	44	164	28,676	13,361
LogiCore	16	16	95 (−67.5%)	30	318	16,413	22,113
LogiCore	32	8	76 (−70.4%)	126	642	32,968	26,319
LogiCore	64	4	52 (−78.0%)	254	1,294	60,908	24,039
Vitis SSR	2	1,733	1,735 (+146.1%)	0	153	20,146	22,335
Ours	2	128	477 (−32.3%)	4	32	15,787	25,155
Ours	4	64	276 (−41.6%)	0	56	30,912	49,016
Ours	8	32	169 (−51.6%)	0	104	56,274	91,345
Ours	16	16	115 (−60.6%)	0	200	100,592	178,929
Ours	32	8	88 (−65.8%)	0	392	185,883	327,339
Ours	64	4	72 (−69.5%)	0	776	327,976	603,049

Table 9: Floating-point (FP32), $N = 1{,}024$. II and first-to-last latency are in cycles. Parentheses: latency change vs. SGen.

Design	SW	II	Latency (f-to-l)	BRAM18	DSP	FF	LUT
SPIRAL	2	512	2,129 (+1.2%)	34	36	35,260	28,714
SPIRAL	4	256	1,224 (+1.4%)	32	64	67,429	54,332
SPIRAL	8	128	778 (+3.6%)	48	116	131,839	104,825
SPIRAL	16	64	552 (+4.5%)	0	220	263,667	210,788
SPIRAL	32	32	436 (+5.1%)	0	428	520,445	402,617
SPIRAL	64	16	370 (+5.1%)	0	844	1,015,924	753,751
SGen	2	512	2,103 (0%)	46	60	40,702	30,318
SGen	4	256	1,207 (0%)	80	112	78,222	57,055
SGen	8	128	751 (0%)	136	216	150,879	109,038
SGen	16	64	528 (0%)	224	424	295,436	212,674
SGen	32	32	415 (0%)	384	840	583,310	428,336
SGen	64	16	352 (0%)	640	1,672	1,139,409	834,050
LogiCore	2	512	1,839 (−12.6%)	4	58	82,814	56,952
LogiCore	4	256	998 (−17.3%)	14	104	92,310	61,679
LogiCore	8	128	552 (−26.5%)	58	214	96,634	66,551
LogiCore	16	64	299 (−43.4%)	92	412	105,977	78,728
LogiCore	32	32	171 (−58.8%)	62	821	33,003	79,658
LogiCore	64	16	120 (−65.9%)	254	1,660	70,354	89,580
Vitis SSR	2	7,372	7,375 (+250.7%)	20	193	25,408	28,137
Ours	2	512	1,729 (−17.8%)	36	40	18,138	29,841
Ours	4	256	914 (−24.3%)	20	72	38,412	64,763
Ours	8	128	499 (−33.6%)	16	136	72,588	119,033
Ours	16	64	289 (−45.3%)	0	264	134,580	229,866
Ours	32	32	190 (−54.2%)	0	520	262,608	450,615
Ours	64	16	133 (−62.2%)	0	1,032	475,222	810,964

5.7 Generalization to Non-Affine Swizzles

We evaluate three isolated CUTLASS-style layout permutations with FP32 payloads: Temporal Unit (8 × 8) [27], Crosswise (TensorOp operand layout, 128 × 64) [28], and Conv3D (192 × 32) [25]. Temporal Unit and Crosswise use `Swizzle<3,0,-3>` and `Swizzle<3,3,3>`, respectively. Conv3D uses the bank-conflict-free `Swizzle<3,2,3>` activation layout documented as an alternative in CUTLASS's Ampere gather-scatter convolution example [25], applied to three 2,048-element tiles per frame.

CODO [42] and Stream-HLS [9] rely on integer-affine access analysis, which does not capture the non-affine XOR reordering in

Table 10: Floating-point (FP32), $N = 16{,}384$. II and first-to-last latency are in cycles. Parentheses: latency change vs. SGen (SPIRAL at $SW = 2$).

Design	SW	II	Latency (f-to-l)	BRAM18	DSP	FF	LUT
SPIRAL	2	8,192	29,069 (0%)	574	52	49,792	40,262
SPIRAL	4	4,096	14,772 (+0.2%)	572	96	96,144	77,064
SPIRAL	8	2,048	7,630 (+0.6%)	568	180	188,699	150,795
SPIRAL	16	1,024	4,056 (+1.0%)	560	348	372,201	297,327
SPIRAL	32	512	2,266 (+1.7%)	544	684	740,747	589,772
SPIRAL	64	256	1,368 (+2.5%)	512	1,356	1,478,874	1,137,008
SGen	2	ERR	ERR	ERR	ERR	ERR	ERR
SGen	4	4,096	14,739 (0%)	372	176	111,563	90,208
SGen	8	2,048	7,587 (0%)	424	344	217,404	160,056
SGen	16	1,024	4,016 (0%)	528	680	428,312	317,692
SGen	32	512	2,229 (0%)	736	1,352	848,829	630,900
SGen	64	256	1,334 (0%)	1,280	2,696	1,688,816	1,254,235
LogiCore	2	8,192	28,651 (−1.4%)	48	82	1,061,088	409,054
LogiCore	4	4,096	15,368 (+4.3%)	52	148	1,082,773	413,713
LogiCore	8	2,048	8,009 (+5.6%)	114	296	1,084,591	420,551
LogiCore	16	1,024	4,096 (+2.0%)	212	586	1,098,175	436,500
Vitis SSR	2	82,667	90,671 (+211.9%)	96	273	36,028	219,639
Ours	2	8,192	26,633 (−8.4%)	428	56	24,686	41,964
Ours	4	4,096	13,391 (−9.1%)	440	104	51,305	81,947
Ours	8	2,048	6,761 (−10.9%)	448	200	98,933	159,968
Ours	16	1,024	3,439 (−14.4%)	480	392	183,704	319,311
Ours	32	512	1,790 (−19.7%)	704	776	377,426	632,935
Ours	64	256	960 (−28.0%)	448	1,544	769,096	1,384,197

Table 11: Frequency scaling of the FP32 $N = 16{,}384$, $SW = 16$ design: post-route $F_{\max}$ (MHz), first-to-last latency (cycles), and post-route FF/LUT counts (thousands) at each target clock. Bold marks each design's best $F_{\max}$ across this table and Table 12. DSP and BRAM18 counts are constant (ours 392/480, SPIRAL 348/560, LogiCore 586/212); LogiCore was not run at 550 MHz.

Design	Target (MHz)	250	300	350	400	450	500	550
Ours	$F_{\max}$ (MHz)	255.8	312.1	354.4	400.0	439.6	478.0	**516.5**
	Latency	3,439	3,465	3,466	3,505	3,521	3,573	3,576
	FF (K)	183	198	200	231	264	323	325
	LUT (K)	266	256	258	267	264	257	255
SPIRAL	$F_{\max}$ (MHz)	294.4	325.6	377.1	417.9	470.1	491.9	501.8
	Latency	4,056	4,056	4,056	4,056	4,056	4,056	4,056
	FF (K)	382	382	383	383	384	384	385
	LUT (K)	288	288	288	288	288	289	289
LogiCore	$F_{\max}$ (MHz)	211.4	226.9	228.2	235.1	**236.5**	227.6	–
	Latency	4,096	4,096	4,096	4,096	4,096	4,096	–
	FF (K)	1,098	1,098	1,098	1,098	1,098	1,098	–
	LUT (K)	437	437	437	437	437	437	–

these swizzles. Their generated kernels retain computed array indexing without an explicit stream-permutation interface. We use total unroll factor 16 and wrap their unchanged array kernels in load/compute/store DATAFLOW adapters connected through frame buffers, allowing successive frames to overlap. All designs expose identical 16-lane, 512-bit streams with row-major packet order and one invocation per frame. HSPL uses RTL refinement, including early start for the two larger units.

Using the same FPGA, clock, and HLS version as the FFT experiments, with start-FIFO depth eight, all nine designs pass 16-frame RTL co-simulation with continuously supplied and drained streams and four input fixtures repeated four times. Table 13 reports median latency and the mean of the last eight frame intervals. Resources are whole-top HLS estimates, including adapters and taken before HSPL's RTL refinement.

Table 12: Complete additional clock-target results beyond Table 11, with our best 550/550 point repeated for reference. Targets and $F_{\max}$ are in MHz, WNS in ns, latency in cycles, and post-route FF/LUT counts in thousands. Bold marks the best $F_{\max}$ across both tables. All rows have II 1,024; DSP/BRAM18 counts are unchanged from Table 11.

Design	HLS/Vivado	WNS	$F_{\max}$	Latency	FF (K)	LUT (K)
Ours	500/450	+0.000	450.0	3,573	322.4	255.9
Ours	550/450	+0.006	451.3	3,576	325.1	253.3
Ours	550/500	−0.105	475.1	3,576	325.3	254.3
Ours	550/550	−0.118	**516.5**	3,576	325.4	255.2
Ours	600/450	+0.000	450.0	3,584	332.4	253.8
Ours	600/500	−0.036	491.2	3,584	332.5	254.2
Ours	600/550	−0.239	486.1	3,584	332.7	254.6
Ours	600/600	−0.401	483.6	3,584	332.6	254.7
Ours	650/500	+0.000	500.0	3,642	371.5	268.8
Ours	650/550	−0.142	510.2	3,642	371.5	269.4
Ours	700/500	+0.000	500.0	3,642	374.2	268.9
Ours	700/550	−0.244	485.0	3,642	374.3	269.2
Ours	700/600	−0.280	513.6	3,642	374.4	269.4
SPIRAL	–/600	−0.181	**541.1**	4,056	384.9	288.8
SPIRAL	–/650	−0.356	528.0	4,056	382.8	291.2
SPIRAL	–/700	−0.517	513.9	4,056	384.1	291.7
SPIRAL	–/750	−0.672	498.8	4,056	383.0	291.6

Table 13: Swizzles with matched 16-lane stream interfaces. II and median transaction latency are RTL co-simulation cycles. Resources are HLS estimates before RTL refinement.

Swizzle	Compiler	II	Latency	BRAM18	FF	LUT
Temporal Unit [27] ($N = 64$)	HSPL	**4**	**8**	0	523	342
	CODO	7	20	64	2,080	2,212
	Stream-HLS	12	34	0	11,118	9,732
Crosswise [28] ($N = 8{,}192$)	HSPL	**512**	**516**	0	1,051	1,142
	CODO	515	1,544	128	4,165	15,745
	Stream-HLS	517	1,550	272	9,019	29,704
Conv3D [25] ($N = 6{,}144$)	HSPL	**384**	**386**	0	1,051	3,234
	CODO	387	1,160	128	4,168	15,765
	Stream-HLS	389	1,166	272	761	4,455

HSPL maintains constant latency and ideal $II = N/16$ throughout all three runs, with lower latency than both baselines. Temporal Unit achieves 1.75× and 3× their respective throughput; Crosswise and Conv3D have similar throughput across the three designs.

HSPL analyzes these XOR maps over $\mathbb{F}_2$ to derive bank layouts and access schedules. It uses no BRAM18 and 27–96% fewer LUTs across the six comparisons. Crosswise and Conv3D use shallow LUTRAM banks, while the baseline array interfaces retain full-frame adapter buffers. FF use is lower in five comparisons; Conv3D uses 1,051 FFs versus 761 for Stream-HLS, whose frame buffers map to BRAM.

6 Related Work

6.1 HLS Accelerator Compilation

Our implementation uses xDSL/MLIR [15, 19] to build a sequence of domain-specific IRs and lowering passes. PyLog [16] provides an algorithm-centric Python programming and synthesis flow for FPGAs. MLIR-based HLS optimizers such as ScaleHLS [39], HIDA [40], Allo [11], and Stream-HLS [9] operate as scheduling and dataflow optimizers inside an HLS flow, and higher-level accelerator DSLs such as Spatial [18] target tile-, unroll-, and memory-hierarchy scheduling on FPGAs; none expose stream-permutation interfaces as first-class compiler objects. CODO [42] and Stream-HLS use integer-affine access analyses that do not capture XOR-based swizzle indices. Task-parallel HLS systems make inter-task communication explicit: TAPA [12] provides FIFO channels, and PASTA [17] adds partitioned, ping-pong buffer channels with physical-design support. HSPL complements these interface abstractions by deriving permutation-specific bank/slot maps and early-start offsets from $\mathbb{F}_2$ descriptions.

6.2 FFT Generation and Layout Representations

SPL represents signal transforms as structured formulas that enable systematic generation of high-performance implementations [38]. SPIRAL and SGen extend this line of work to streaming hardware and direct RTL generation [22, 32], and Püschel et al. [29] use linear maps over $\mathbb{F}_2$ to derive bank-conflict-free RAM layouts for linear streaming permutations. Serre et al. [30] derive switching-optimal spatial/temporal factorizations, and subsequent work fuses permutations to reduce FFT memory requirements [31]. Binary index algebra also underlies CuTe's XOR swizzles in CUTLASS [26] and Triton's Linear Layouts [43]. The FFT generators optimize their streaming representations before emitting RTL directly. Our backend instead lowers stream-permutation interfaces to HLS, retaining its local scheduling, intra-module fusion, arithmetic mapping, and platform-specific lowering. HP-FFT was previously presented as work in progress [34]. The present work introduces HSPL interfaces, $\mathbb{F}_2$-derived conflict-free layouts, and agent-guided refinement, and extends the comparison to $N = 16{,}384$ and $SW = 64$.

6.3 Streaming Languages and Timing Types

Streaming hardware languages and timing-aware type systems reason about value- or register-level timing: when a signal is valid, how pipeline stages align, or how spatial operators compose [14]. Filament [24] introduces timeline types for this purpose, which Xia et al. [37] extend to unify HLS and RTL timing contracts at the HLS–RTL interface. Reconfigurable Stream Network [35] exposes computation and communication latencies to coordinate streaming across heterogeneous functional units. These systems are complementary to HSPL, which focuses on stream-permutation interfaces: conflict-free access order, bank and slot organization, producer-consumer offset, and whether an interface is realized as FIFOs, crossbars, or RAM-based PIPOs.

6.4 Agent-Assisted Hardware Design

Agent and LLM-based systems have been explored for code generation, debugging, and RTL design assistance [10, 20, 33]. LLM-DSE [36] uses cooperating agents to search HLS directive parameters, while *From Pragmas to Partners* [41] argues for HLS as an abstraction layer for agentic hardware optimization. Our agent operates after HLS: it composes compiler-vetted RTL patch primitives to realize interface semantics, with each refinement tied to compiler facts rather than a search over source-level directives.

7 Conclusion

This paper argues that the main obstacle to using HLS for cycle-critical streaming designs built around non-affine stream permutations is not weak intra-module scheduling, but the lack of HDL's cycle-level expressiveness for inter-module interface behavior.

HSPL addresses this gap by extending HLS with explicit stream-permutation interfaces. From these interfaces, the compiler derives conflict-free addressing and safe early-start offsets, and synthesizes the corresponding wires, muxes, FIFOs, and RAM-based PIPOs, while HLS retains responsibility for local scheduling, intra-module fusion, and arithmetic mapping. For interface semantics that current HLS backends cannot express directly, an agent-assisted post-HLS RTL refinement layer treats HLS-generated RTL as an editable compiler-backend artifact under compiler-emitted RTL refinement contracts and vetted patch primitives.

Our streaming FFT results show a practical division of responsibility: HSPL captures cycle-accurate stream interfaces, HLS optimizes local computation, and refinement realizes backend behavior the HLS source cannot convey. The resulting HP-FFT generator produces designs with lower latency than well-engineered vendor RTL IP in most cases; to our knowledge, it is the first HLS-based flow to reach this quality on cycle-critical streaming FFTs. The same interfaces carry over unchanged to CUTLASS-style swizzles, where refined HSPL achieves lower frame latency and constant frame II in all three cases, while throughput gains vary by case. HSPL offers limited throughput improvement when an interface uses a plain FIFO or PIPO that native HLS already sustains at rate; its value lies in the stream-permutation interfaces HLS cannot express. The agent-guided RTL refinement we rely on is itself a consequence of today's commercial HLS backends being black boxes that users cannot extend; HLS tools with open or extensible backends could absorb HSPL's interface semantics directly. Yet unsupported corners are inherent to any commercial HLS tool, since no fixed backend can anticipate every application's needs; the agent-guided refinement we present offers a reference approach for extending HLS reach when an application hits the tool's limits, without waiting for vendor support.

Acknowledgments

We thank James Hoe and Naifeng Zhang for helpful discussions on FFTs and SPIRAL. This work was supported in part by PRISM Center under JUMP 2.0 program sponsored by SRC and DARPA, the CDSC industrial partners (https://cdsc.ucla.edu/partners), and the AMD HACC Program. J. Cong has financial interests in AMD, IBM, and Qualcomm.

References

[1] AMD. 2025. *Assessing the Maximum Frequency of the Design.* https://docs.amd.com/r/2025.1-English/ug949-vivado-design-methodology/Assessing-the-Maximum-Frequency-of-the-Design UltraFast Design Methodology Guide (UG949), version 2025.1.

[2] AMD. 2025. Vitis High-Level Synthesis User Guide (UG1399). https://docs.amd.com/r/2025.1-English/ug1399-vitis-hls/Control-driven-Task-level-Parallelism?contentId=qTRdKWHT~7gWz2QDeOzccQ. Version 2025.1, Control-driven Task-level Parallelism. Accessed: 2026-09-16.

[3] AMD. 2025. Xilinx: Fast Fourier Transform (FFT) LogiCORE IP Product Guide (PG109). https://docs.amd.com/r/en-US/pg109-xfft. Accessed: 2026-04-22.

[4] AMD. 2025. Xilinx Vitis Libraries DSP: Fast Fourier Transform (FFT). https://docs.amd.com/r/2025.1-English/Vitis_Libraries/dsp/index.html?contentId=XgXUp6RclG2cL~r4Cny7nw. Accessed: 2026-04-22.

[5] Anthropic. 2025. Claude Haiku 4.5. https://platform.claude.com/docs/en/models/haiku-4-5/overview. Accessed: 2026-09-16.

[6] Anthropic. 2026. Claude Opus 5. https://platform.claude.com/docs/en/models/opus-5/overview. Accessed: 2026-09-16.

[7] Anthropic. 2026. Claude Sonnet 5. https://platform.claude.com/docs/en/models/sonnet-5/overview. Accessed: 2026-09-16.

[8] Anthropic. 2026. Introducing Claude Opus 4.7. https://www.anthropic.com/news/claude-opus-4-7. Accessed: 2026-09-16.

[9] Suhail Basalama and Jason Cong. 2025. Stream-HLS: Towards Automatic Dataflow Acceleration. In *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA '25)*. Association for Computing Machinery, New York, NY, USA, 103–114. doi:10.1145/3706628.3708878

[10] Jason Blocklove, Siddharth Garg, Ramesh Karri, and Hammond Pearce. 2023. Chip-Chat: Challenges and Opportunities in Conversational Hardware Design. In *2023 ACM/IEEE 5th Workshop on Machine Learning for CAD (MLCAD) (MLCAD '23)*. IEEE, 1–6. doi:10.1109/MLCAD58807.2023.10299874

[11] Hongzheng Chen, Niansong Zhang, Shaojie Xiang, Zhichen Zeng, Mengjia Dai, and Zhiru Zhang. 2024. Allo: A Programming Model for Composable Accelerator Design. *Proc. ACM Program. Lang.* 8, PLDI, Article 171 (June 2024), 28 pages. doi:10.1145/3656401

[12] Yuze Chi, Licheng Guo, Jason Lau, Young-kyu Choi, Jie Wang, and Jason Cong. 2021. Extending High-Level Synthesis for Task-Parallel Programs. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 204–213. doi:10.1109/FCCM51124.2021.00032

[13] James W Cooley and John W Tukey. 1965. An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation* 19, 90 (1965), 297–301.

[14] David Durst, Matthew Feldman, Dillon Huff, David Akeley, Ross Daly, Gilbert Louis Bernstein, Marco Patrignani, Kayvon Fatahalian, and Pat Hanrahan. 2020. Type-Directed Scheduling of Streaming Accelerators. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20)*. Association for Computing Machinery, 408–422. doi:10.1145/3385412.3385983

[15] Mathieu Fehr, Michel Weber, Christian Ulmann, Alexandre Lopoukhine, Martin Paul Lücke, Théo Degioanni, Christos Vasiladiotis, Michel Steuwer, and Tobias Grosser. 2025. xDSL: Sidekick Compilation for SSA-Based Compilers. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization (CGO '25)*. Association for Computing Machinery, 179–192. doi:10.1145/3696443.3708945

[16] Sitao Huang, Kun Wu, Hyunmin Jeong, Chengyue Wang, Deming Chen, and Wen-Mei W. Hwu. 2021. PyLog: An Algorithm-Centric Python-Based FPGA Programming and Synthesis Flow. *IEEE Trans. Comput.* 70, 12 (Dec. 2021), 2015–2028. doi:10.1109/TC.2021.3123465

[17] Moazin Khatti, Xingyu Tian, Yuze Chi, Licheng Guo, Jason Cong, and Zhenman Fang. 2023. PASTA: Programming and Automation Support for Scalable Task-Parallel HLS Programs on Modern Multi-Die FPGAs. In *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 12–22. doi:10.1109/FCCM57271.2023.00011

[18] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Spatial: A Language and Compiler for Application Accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '18)*. Association for Computing Machinery, 296–311. doi:10.1145/3192366.3192379

[19] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2–14. doi:10.1109/CGO51591.2021.9370308

[20] Yao Lu, Shang Liu, Qijun Zhang, and Zhiyao Xie. 2024. RTLLM: An Open-Source Benchmark for Design RTL Generation with Large Language Model. In *Proceedings of the 29th Asia and South Pacific Design Automation Conference (ASP-DAC '24)*. IEEE, 722–727. doi:10.1109/ASP-DAC58780.2024.10473904

[21] Peter A. Milder. 2010. *A Mathematical Approach for Compiling and Optimizing Hardware Implementations of DSP Transforms.* Ph. D. Dissertation. Carnegie Mellon University, Pittsburgh, PA, USA.

[22] Peter A. Milder, Franz Franchetti, James C. Hoe, and Markus Püschel. 2012. Computer Generation of Hardware for Linear Digital Signal Processing Transforms. *ACM Transactions on Design Automation of Electronic Systems* 17, 2, Article 15 (2012), 33 pages. doi:10.1145/2159542.2159547

[23] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, Jason Anderson, and Koen Bertels. 2016. A Survey and Evaluation of FPGA High-Level Synthesis Tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 10 (2016), 1591–1604. doi:10.1109/TCAD.2015.2513673

[24] Rachit Nigam, Pedro Henrique Azevedo de Amorim, and Adrian Sampson. 2023. Modular Hardware Design with Timeline Types. *Proc. ACM Program. Lang.* 7, PLDI, Article 120 (2023), 26 pages. doi:10.1145/3591234

[25] NVIDIA Corporation. 2025. CUTLASS: Ampere Gather-Scatter Convolution Shared-Memory Layouts. https://github.com/NVIDIA/cutlass/blob/e64a9136dd929639e5f7c969fe5af3bf7415cd4f/examples/59_ampere_gather_scatter_conv/ampere_conv_kernel.h#L121-L159. Source file: ampere_conv_kernel.h. Pinned commit `e64a913`. Accessed: 2026-09-16.

[26] NVIDIA Corporation. 2025. CUTLASS: CUDA Templates for Linear Algebra Subroutines. https://github.com/NVIDIA/cutlass. Swizzle benchmarks at commit `e64a913`.

[27] NVIDIA Corporation. 2025. CUTLASS: CuTe Swizzle Layout Unit Tests. https://github.com/NVIDIA/cutlass/blob/e64a9136dd929639e5f7c969fe5af3bf7415cd4f/test/unit/cute/core/swizzle_layout.cpp#L103-L108. Source file: swizzle_layout.cpp. Pinned commit `e64a913`. Accessed: 2026-09-16.

[28] NVIDIA Corporation. 2025. CUTLASS: TensorOp Crosswise Layout Unit Test. https://github.com/NVIDIA/cutlass/blob/e64a9136dd929639e5f7c969fe5af3bf7415cd4f/test/unit/cute/layout/layout_operator.cu#L107-L113. Source file: layout_operator.cu. Pinned commit `e64a913`. Accessed: 2026-09-16.

[29] Markus Püschel, Peter A. Milder, and James C. Hoe. 2009. Permuting Streaming Data Using RAMs. *Journal of the ACM* 56, 2, Article 10 (2009), 34 pages. doi:10.1145/1502793.1502799

[30] François Serre, Thomas Holenstein, and Markus Püschel. 2016. Optimal Circuits for Streamed Linear Permutations Using RAM. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '16)*. Association for Computing Machinery, 215–223. doi:10.1145/2847263.2847277

[31] François Serre and Markus Püschel. 2018. Memory-Efficient Fast Fourier Transform on Streaming Data by Fusing Permutations. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '18)*. Association for Computing Machinery, 219–228. doi:10.1145/3174243.3174263

[32] François Serre and Markus Püschel. 2019. DSL-Based Hardware Generation with Scala: Example Fast Fourier Transforms and Sorting Networks. *ACM Transactions on Reconfigurable Technology and Systems* 13, 1, Article 1 (2019), 23 pages. doi:10.1145/3359754

[33] Shailja Thakur, Baleegh Ahmad, Zhenxing Fan, Hammond Pearce, Benjamin Tan, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. 2023. Benchmarking Large Language Models for Automated Verilog RTL Code Generation. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1–6. doi:10.23919/DATE56975.2023.10137086

[34] Chengyue Wang, Jiahao Zhang, Yingquan Wu, and Jason Cong. 2025. HP-FFT: A General High-Performance FFT Generator Using High-Level Synthesis. In *2025 IEEE 33rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 19–23. doi:10.1109/FCCM62733.2025.00026

[35] Chengyue Wang, Xiaofan Zhang, Jason Cong, and James C. Hoe. 2025. Reconfigurable Stream Network Architecture. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25)*. Association for Computing Machinery, 1848–1866. doi:10.1145/3695053.3731088

[36] Hanyu Wang, Xinrui Wu, Zijian Ding, Su Zheng, Chengyue Wang, Neha Prakriya, Tony Nowatzki, Yizhou Sun, and Jason Cong. 2025. LLM-DSE: Searching Accelerator Parameters with LLM Agents. arXiv:2505.12188 [cs.AR] doi:10.48550/arXiv.2505.12188

[37] Jingtao Xia, Ayana Alemayehu, Sijie Kong, Ben Hardekopf, Rachit Nigam, and Jonathan Balkind. 2026. Unifying HLS and RTL with Timeline Types. In *Workshop on Languages, Tools, and Techniques for Accelerator Design (LATTE)*. https://capra.cs.cornell.edu/latte26/paper/latte26-final17.pdf

[38] Jianxin Xiong, Jeremy Johnson, Robert Johnson, and David Padua. 2001. SPL: A language and compiler for DSP algorithms. *ACM SIGPLAN Notices* 36, 5 (2001), 298–308.

[39] Hanchen Ye, Cong Hao, Jianyi Cheng, Hyunmin Jeong, Jack Huang, Stephen Neuendorffer, and Deming Chen. 2022. ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA).*

IEEE, 741–755. doi:10.1109/HPCA53966.2022.00060

[40] Hanchen Ye, Hyegang Jun, and Deming Chen. 2024. HIDA: A Hierarchical Dataflow Compiler for High-Level Synthesis. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '24)*. Association for Computing Machinery, 215–230. doi:10.1145/3617232.3624850

[41] Niansong Zhang, Sunwoo Kim, Shreesha Srinath, and Zhiru Zhang. 2026. From Pragmas to Partners: A Symbiotic Evolution of Agentic High-Level Synthesis. arXiv:2602.01401 [cs.CL] doi:10.48550/arXiv.2602.01401

[42] Weichuang Zhang, Yiquan Wang, Xinzhou Zhang, Chi Zhang, Yu Feng, Xiaofeng Hou, Chao Li, Jieru Zhao, and Minyi Guo. 2026. CODO: An Automated Compiler for Comprehensive Dataflow Optimization. In *2026 ACM/IEEE 53rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1–17. doi:10.1109/ISCA66397.2026.00018

[43] Keren Zhou, Mario Lezcano-Casado, Adam P. Goucher, Akhmed Rakhmati, Jeff Niu, Justin Lebar, Pawel Szczerbuk, Peter Bell, Phil Tillet, Thomas Raoux, and Zahi Moudallal. 2026. Linear Layouts: Robust Code Generation of Efficient Tensor Computation Using $\mathbb{F}_2$. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '26)*. Association for Computing Machinery, 132–146. doi:10.1145/3760250.3762221