ST-Flow: A Hardware Compiler for Automating Spatial-Temporal Dataflow Acceleration

Suhail Basalama*
basalama@ucla.edu
University of California, Los Angeles
Los Angeles, California, USA

Stéphane Pouget*
pouget@cs.ucla.edu
University of California, Los Angeles
Los Angeles, California, USA

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

Abstract

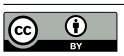
The rapid growth of AI workloads has driven a shift toward specialized hardware accelerators that provide significantly higher efficiency than general-purpose CPUs and GPUs. However, their broader adoption is hindered by the complexity and expertise required for hardware design, as well as long development cycles that lag behind the pace of AI innovation. Existing approaches to DNN acceleration generally follow three paradigms: (1) large, massively parallel engines executing layers sequentially; (2) fully spatial designs mapping each layer to dedicated hardware for concurrent execution; and (3) hybrid approaches combining temporal and spatial execution.

Prior efforts to automate accelerator generation from high-level specifications face several limitations. Many lack true end-to-end compilation and only report EDA tools' estimates. Others generate inefficient non-streaming architectures or depend on fully spatial mappings that do not scale well. Additionally, the design space is massive, and automated scheduling is either absent or limited, often without accurate performance modeling, leading to suboptimal designs.

This work introduces ST-Flow, an *open-source* end-to-end hardware compiler that translates PyTorch models into efficient FPGA-based streaming dataflow accelerators. ST-Flow provides a unified methodology that supports the complete spectrum of spatial, temporal, and hybrid dataflow mappings, supplemented by a hybrid mapping heuristic and a flexible non-divisor tiling strategy that decouples hardware unrolling from strict loop-bound divisibility. An accurate performance model guides optimization, tuning parallelism levels and scheduling execution order to maximize inter-module streaming and concurrency, and the compiler generates FPGA accelerators ready for deployment.

We evaluate ST-Flow through 107 physical FPGA deployments, including a GPT-2 case study. Our hybrid mapping heuristic achieves up to 1.59× improvement in area-delay product over prior pure spatial mapping approaches, and our non-divisor tiling strategy achieves 2.12× average and 1.43× geo. mean speedups (up to 22.9×) over divisor-only baselines. ST-Flow is open-sourced at https://github.com/UCLA-VAST/ST-Flow (artifact: [6]).

*Both authors contributed equally to this research.



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/10.1145/3838684.3846865

CCS Concepts

• **Hardware → Hardware-software codesign**; **Operations scheduling**; **Hardware accelerators**; **Emerging languages and compilers**; • **Computer systems organization → Data flow architectures**; **Reconfigurable computing**.

Keywords

Streaming Dataflow Architecture, Spatial-Temporal Dataflow, Hardware Accelerator, Hardware Reuse, Time-Multiplexing, HLS, FPGA, MLIR, Design Space Exploration, Hardware Compiler, Deep Neural Networks, Transformers, GPT-2

ACM Reference Format:
Suhail Basalama, Stéphane Pouget, and Jason Cong. 2026. ST-Flow: A Hardware Compiler for Automating Spatial-Temporal Dataflow Acceleration. 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/10.1145/3838684.3846865

1 Introduction

The growing computational demands of Artificial Intelligence (AI) workloads, particularly Deep Neural Networks (DNNs) and large language models (LLMs), are pushing the limits of traditional computing paradigms. General-purpose processors such as CPUs and GPUs have long served as the backbone of AI execution, but scaling these architectures to datacenter-level LLM deployment has become increasingly inefficient. This challenge is compounded by the slowdown of Moore's law [47] and the breakdown of Dennard scaling [9, 19], which historically sustained performance improvements. A fundamental source of inefficiency lies in the von Neumann architecture itself, where the need for complex control logic and instruction-set architectures (ISAs) to preserve generality introduces significant performance and energy overheads.

To overcome these limitations, both academia and industry have shifted toward domain-specific hardware acceleration. Prior work has shown that tailoring architectures to specific workloads can yield orders-of-magnitude improvements in performance and energy efficiency [18, 26]. This trend has accelerated with the emergence of custom Neural Processing Units (NPUs) and AI accelerators across the computing stack. Hyperscalers have developed in-house ASICs such as Google's TPUs [29, 30] and Amazon's Inferentia [2], while field-programmable gate arrays (FPGAs) continue to play a critical role in enabling flexible specialization at scale, most notably in Microsoft's cloud infrastructure [11, 15]. In parallel, a growing ecosystem of startups is exploring novel spatial and dataflow architectures, including the wafer-scale processors from Cerebras [36] and the streaming processors from Groq [1]. These domain-specific designs exploit parallelism, data locality, and customized dataflows

Table 1: Comparison of ST-Flow with prior manual and automation frameworks.

Framework (Venue)	Streaming	HW Mapping*			Non-Divisor	DSE	On-Board
		S	T	ST	Tiling		Verification
Allo [12] (PLDI'24)	✓	M	✗	✗	✗	N/A	✓
InTAR [25] (FCCM'25)	✓	M	M	M	✗	N/A	✓
ScaleHLS [58] (HPCA'22)	✗	✓	✗	✗	✗	Limited	✗
HIDA [59] (ASPLOS'24)	✗	✓	✗	✗	✗	Limited	✗
Stream-HLS [5] (FPGA'25)	✓	✓	✗	✗	✗	Global	✗
StreamTensor [57] (MICRO'25)	✓	✓	✗	✗	✗	Limited	✓
CODO [61] (ISCA'26)	✓	✓	✗	✗	✗	Limited	✓
ST-Flow (this work)	✓	✓	✓	✓	✓	**Global**	✓

*S: Spatial, T: Temporal, ST: Spatial-Temporal Dataflow Accelerator. M: supported via manual implementation.

while eliminating instruction overhead. In the context of DNNs, accelerator designs typically follow three paradigms: (1) temporal mapping using a single, highly parallel compute engine, (2) spatial mapping with dedicated hardware per layer, and (3) hybrid spatial-temporal mapping strategies that balance resource utilization and performance.

Despite these advantages, the adoption of custom hardware accelerators remains constrained by the complexity of hardware design. Traditional Register-Transfer Level (RTL) development using hardware description languages (HDLs) such as Verilog and VHDL can take months to years and requires substantial expertise. High-Level Synthesis (HLS) [16] has improved accessibility by raising the abstraction level to C/C++, enabling broader use of FPGA-based acceleration across domains including machine learning [7, 54, 60, 63], image processing [14, 28], genome sequencing [10, 37], and scientific computing [40, 56]. However, achieving high performance still requires navigating large design spaces and hardware-aware optimization. Recent advances in compiler infrastructures, particularly those based on Multi-Level Intermediate Representation (MLIR) [33, 34], aim to further automate this process by translating high-level machine learning frameworks (e.g., PyTorch) directly into efficient hardware implementations, reducing both development effort and time-to-deployment.

While multiple prior works have attempted to automate the generation of hardware accelerators from high-level model specifications, several critical challenges remain unresolved. First, many existing frameworks [5, 58, 59] lack true end-to-end compilation capabilities, often functioning as prototyping tools rather than generating fully functional, deployable hardware designs. Second, they frequently generate inefficient, non-streaming architectures [58, 59] or are constrained to strictly ***spatial mappings*** where every layer requires dedicated hardware [5, 57, 61]. Relying on spatial mapping is inherently unscalable for modern, deep networks due to the prohibitive consumption of on-chip resources. Finally, these frameworks often lack comprehensive, automated design space exploration (DSE) [57–59, 61]. When DSE is present, it is usually constrained by rudimentary local greedy optimizations and rigid, divisor-only tiling limitations, which fail to capture the nuances of non-uniform loop bounds and ultimately lead to sub-optimal, resource-inefficient designs. Table 1 shows a high-level comparison of ST-Flow against state-of-the-art works.

To overcome these limitations, we present ST-Flow, an *open-source* end-to-end hardware compiler that translates PyTorch models into highly optimized, FPGA-based streaming dataflow accelerators. ST-Flow expands the scope of dataflow architectural styles of prior spatial-only approaches by introducing a unified methodology for spatial mapping, temporal mapping, and hybrid spatial-temporal mapping. In this work, the terms *kernel* and *layer*, as well as *tiling* and *unrolling*, are used interchangeably. ST-Flow presents a novel mapping strategy, which automatically identifies opportunities to merge structurally and temporally compatible kernels into shared, time-multiplexed hardware modules. To effectively navigate the exponentially-large design space, we formulate a highly accurate analytical performance and resource model that supports flexible, ***non-divisor*** tiling factors and complex loop permutations. Guided by this model, ST-Flow employs advanced heuristic and exact search methods including Mixed Integer Non-Linear Programming (MINLP), Simulated Annealing (SA), and Genetic Algorithms (GA) to discover best hardware configurations that balance latency and resource constraints. We conduct a comprehensive evaluation using both synthetic workloads and real-world models, such as GPT-2 [46], demonstrating that ST-Flow significantly improves resource efficiency and enables the end-to-end acceleration of complex AI models on FPGAs.

The main contributions of this work include:

- **End-to-End MLIR-Based Hardware Compiler**: We present ST-Flow, an automated *open-source* framework that translates PyTorch models into deployable, streaming dataflow accelerators for FPGAs through a multi-level IR stack.
- **A Methodology for Spatial-Temporal Dataflow Acceleration**: We introduce a unified methodology that encompasses spatial, temporal, and hybrid spatial-temporal dataflow mappings. Further, we introduce a novel hybrid mapping heuristic that navigates the super-exponential design space to automatically identify a good balance between dedicated hardware specialization and resource-efficient temporal reuse.
- **Unified Analytical Modeling with Non-Divisor Tiling**: We formulate a comprehensive performance and resource model that generalizes prior dataflow models to support spatial-temporal mappings and decouples hardware unrolling from strict loop-bound divisibility through flexible tiling strategies, achieving speedups of a 2.12× average and 1.43× geo. mean (up to 22.9×) over traditional divisor-only baselines.
- **Automated DSE and On-Board Validation**: We implement and compare multiple heuristic search methods (MINLP, SA, and GA) for design space exploration and validate our performance model and compiler end-to-end through physical deployment of 107 unique designs on a Xilinx Alveo U55C FPGA.

Section 2 motivates our approach with an example. Section 3 describes the ST-Flow compiler architecture, while Section 4 details our unified performance modeling and DSE framework. Section 5 evaluates our approach on synthetic and real-world models.

2 Background and Motivation

We use a moderately complex multi-layer perceptron (MLP) with a branching topology (MLP Diamond Rand., Section 5) to illustrate the design tradeoffs addressed in this work. The model is shown on the left of Figure 1. Each configuration in the figure maps this same model to a different hardware dataflow architecture.

Figure 1 compares three classes of accelerators: spatial mapping (blue), temporal mapping (red), and hybrid spatial-temporal

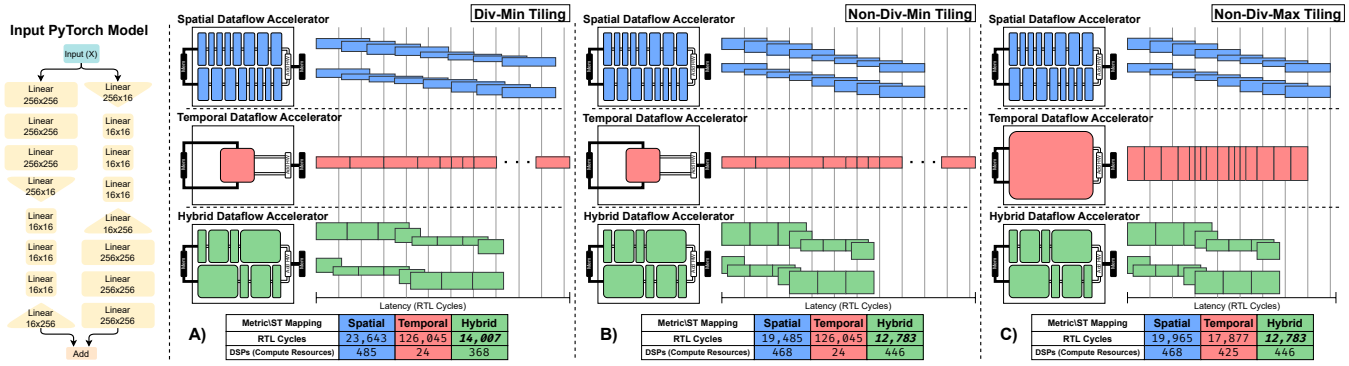


Figure 1: Motivational example comparing spatial mapping (blue), temporal mapping (red), and hybrid spatial-temporal mapping (green) dataflow accelerators under three tiling strategies: Div-Min (A), Non-Div-Min (B), and Non-Div-Max (C).

mapping (green). Each row shows execution timelines, where overlapping rectangles indicate concurrent execution across modules: a producer forwards output elements through FIFO channels as they finalize, so a dependent consumer starts before the producer completes. This holds for reductions as well (e.g., with the reduction loop innermost, GEMM outputs finalize and stream out progressively), and the achievable overlap depends on matching the producer's and consumer's access orders, which our DSE resolves through loop permutation (Section 4.2). Rectangle sizes in the architectures and rectangle heights in time traces reflect the degree of parallelism (allocated compute resources), while width reflects execution latency.

2.1 Expanding Hardware Mapping Design Space

Prior ML compilers, such as ScaleHLS [58] and HIDA [59] demonstrated end-to-end flows from high-level frameworks to HLS designs. However, these designs relied on shared-memory communication, resulting in largely sequential execution. Stream-HLS [5] introduced streaming channels to enable concurrent spatial dataflow execution. Recent efforts, including StreamTensor [57] and CODO [61], follow this paradigm but all these works remain restricted to fully ***spatial mappings*** (first row in blue in Figure 1).

In spatial dataflow accelerators, performance is primarily determined by (1) loop permutation and (2) loop unrolling. Loop permutation affects both initiation interval (II) and inter-module concurrency. For example, placing reduction loops innermost limits its II due to loop-carried dependencies, while placing them outermost delays output production, reducing overlap with downstream modules. Thus, maximizing local throughput may conflict with maximizing global concurrency.

Even with optimal scheduling, purely spatial designs cannot fully exploit concurrency when the model contains inherent temporal dependencies (e.g., sequential layers or imbalanced branches). This is evident in Figure 1A, where the spatial design achieves significant overlap but still leaves idle periods due to these dependencies. As prior works [25, 62] have shown, spatial-temporal dataflow accelerators can mitigate this issue and outperform purely spatial or temporal mappings, but these designs are typically restricted to *manual* implementation efforts.

In contrast, hybrid spatial-temporal mappings (green) selectively time-multiplex dependent layers while preserving parallelism across independent branches. This enables better alignment between computation structure and model dependencies. As shown in Figure 1A, the hybrid design outperforms the best spatial design by 1.7× while using fewer resources.

2.2 Expanding Compute Allocation Design Space

Compute allocation (loop unrolling/tiling) is equally critical. All configurations in Figure 1 assume a budget of 500 DSPs and a minimum temporal tile size of 8.

Div-Min (A). A common approach is to restrict tiling factors to divisors of all loop bounds. For this MLP with layer dimensions $(256, 256), (256, 16), (16, 256), (16, 16)$, the common divisor space collapses to divisors of 16. This leads to small tiling factors (e.g., $(2, 2)$), resulting in very low resource utilization. Thus, the temporal design in Figure 1A uses only 24 DSPs and suffers from high latency.

Non-Div-Min (B). Relaxing the divisor constraint while still bounding tiling by the minimum loop size (16) increases flexibility. This improves spatial and hybrid designs by enabling better load balancing, yielding ∼20% and ∼10% gains, respectively.

Non-Div-Max (C). Further relaxing tiling bounds to the maximum loop size (256) significantly expands the design space. This allows the temporal design to utilize substantially more resources, improving performance by ∼7× over **Div-Min** and surpassing the spatial design. Still, the hybrid design is the best overall, as it combines temporal reuse with spatial parallelism across independent branches.

The flexibility introduced by hardware mapping and non-divisor tiling leads to an exponential configuration space (up to 5.19×10^{33} for this MLP example). Given this expanded scope and enormous design space, the two main challenges are: ***(1) how to automate generating spatial, temporal, or hybrid dataflow architectures, and (2) how to efficiently explore this massive design space under resource and correctness constraints.***

ST-Flow addresses both challenges through a unified compilation and global design space exploration framework.

3 The ST-Flow Compiler Overview

In this section, we provide an overview of how ST-Flow processes a model into a hardware accelerator as highlighted in Figure 2.

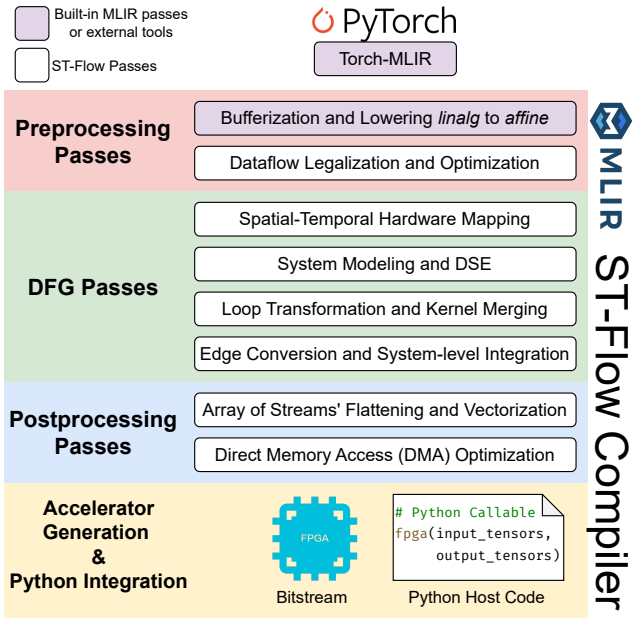


Figure 2: Overview of the ST-Flow compilation and optimization framework.

3.1 Scope and Overview

ST-Flow targets multi-kernel affine applications with regular, rectangular access patterns. It accepts affine programs composed of perfectly nested loops with constant bounds operating on scalars or multi-dimensional arrays with affine accesses. This model captures a broad set of hardware-accelerated workloads, including common DNNs (e.g., MLPs and transformers) and scientific kernels such as matrix multiplication, reductions, elementwise operations, normalization, and activations. While our methodology (Section 4) is applicable to convolutional neural networks (CNNs), handling sliding-window, non-contiguous accesses in hardware requires additional engineering effort and is not a primary focus of this work.

The ST-Flow compiler is composed of two main components: 1) a C++ library built on top of MLIR [33, 34] for core transformations, and 2) a Python wrapper that orchestrates the end-to-end flow. It leverages **Torch-MLIR** [52] as a front-end to ingest PyTorch models and can optionally utilize the AMPL solver [4] for optimization. Consequently, ST-Flow supports any PyTorch model that Torch-MLIR can lower to the *linalg* dialect and that satisfies the affine restrictions above (e.g., MLPs, transformers, and scientific kernels).

3.2 ST-Flow MLIR Library

Our compiler's core logic is implemented as a sequence of MLIR passes operating on standard dialects (*affine*, *memref*, *arith*) and a specialized HLS/Dataflow dialect [5, 58] that enables streaming. The preprocessing passes reuse standard MLIR lowering, and the system modeling and edge conversion passes extend those of StreamHLS [5]; the spatial-temporal hardware mapping, non-divisor tiling, kernel merging, postprocessing, and Python integration passes are new contributions of this work.

3.2.1 *Preprocessing Passes.*

Bufferization and Lowering linalg to affine. ST-Flow utilizes MLIR's built-in passes to convert abstract tensors into buffers (*memref* dialect) and lower high-level *linalg* operations into perfectly nested affine loops. This step prepares the IR for hardware-oriented transformations.

Dataflow Legalization and Optimization. This stage ensures the IR satisfies hardware dataflow constraints. Key transformations include buffer duplication to eliminate multi-consumer dependencies and preparatory clean-up to establish a strict single-producer-single-consumer invariant between tasks.

3.2.2 *Dataflow Graph (DFG) Passes.*

We utilize a unified `DFG` class to represent the hardware accelerator as a directed graph where nodes represent loop nests and edges represent true/flow dependencies. All passes in this stage operate on this DFG to determine the final hardware architecture.

Spatial-Temporal Hardware Mapping. This pass determines which kernels are mapped to shared hardware modules. It supports three strategies: spatial mapping (no reuse), temporal mapping (maximum reuse), and our proposed hybrid spatial-temporal mapping. Merged nodes in the HW DFG represent physical hardware modules.

System Modeling and DSE. Using the HW DFG, this pass estimates performance and resource usage based on loop permutations and tiling factors. It employs various search engines (MINLP, SA, GA) to explore the massive design space to find best configurations.

Loop Transformation and Kernel Merging. Once a configuration is selected, this pass applies the chosen loop permutations and tiling factors. It then materializes merged nodes by creating parameterized kernels capable of executing variably-sized loop nests on the same hardware.

Edge Conversion and System-level Integration. Shared buffers between nodes are converted into streaming FIFO channels when compatible. If access patterns mismatch, layout conversion nodes are inserted. To ensure deadlock-free execution and maximize performance, we set the depth of each FIFO to match the size of its original buffer. An exception is made for edges connecting compute nodes to off-chip memory modules, where we use a fixed depth of 32 since the consumption rate significantly exceeds the production rate. Finally, a top-level module is constructed, connecting all modules via streaming interfaces.

3.2.3 *Postprocessing Passes.*

Array of Streams' Flattening and Vectorization. To reduce control logic overhead, multi-dimensional arrays of streams are flattened into 1-D arrays and vectorized. This allows multiple data elements to be transferred under a single handshake circuit.

Direct Memory Access (DMA) Optimization. This pass serializes strided off-chip memory accesses through automatic data

layout transformation in off-chip memory (and generates the corresponding host-side layout transformation helpers) to maximize burst lengths and bandwidth during runtime. It also handles AXI4 power-of-two and 512-bit-max-port-width interface constraints by padding port widths or serializing accesses to meet hardware specifications.

3.2.4 Accelerator Generation and Python Integration. The final synthesizable HLS C++ is then used to generate an FPGA bitstream to be paired with a C++ host driver. ST-Flow automatically compiles this driver and generates a Python extension using **Pybind11**. This allows the resulting FPGA accelerator to be imported and invoked as a standard Python function, taking NumPy or PyTorch tensors as input and returning results directly to the Python workspace. This end-to-end integration ensures that ST-Flow is not just a hardware generator, but a complete system for hardware-accelerated AI.

4 Methodology

This section explains in detail the methodology and rationale behind the ST-Flow compiler. It first details the three hardware mapping strategies, then it discusses our system modeling and design space exploration framework.

4.1 Spatial-Temporal Hardware Mapping

The design space for mapping N DFG nodes onto hardware modules grows super-exponentially with N (defined by the N-th Bell number), rendering exhaustive search intractable. As illustrated in Section 2, these mappings span two architectural extremes: temporal mapping, which executes all kernels sequentially on a single, high-parallelism module, and spatial mapping, which allocates dedicated hardware modules for concurrent execution. The effectiveness of these mappings is further influenced by the compute allocation strategy (i.e., tiling and unrolling), which we discuss in Section 4.2.3.

The optimal mapping depends on both the DFG topology and individual kernel characteristics. While spatial dataflow accelerators allow concurrent execution, they often suffer from inter-module stalls due to temporal dependencies, such as reduction loops that delay output production (see Figure 1). Temporal dataflow accelerators maximize compute density by allocating all resources to one kernel at a time, but they can suffer from padding overheads and low utilization when kernel dimensions mismatch the hardware parameters.

To navigate this trade-off, ST-Flow proposes a novel heuristic that selects a hybrid spatial-temporal mapping. Our approach balances these architectural styles by selectively time-multiplexing dependent kernels while preserving spatial parallelism across independent branches. This strategy aligns hardware execution with the model's dependency structure to maximize overall throughput and resource efficiency.

Before presenting our hybrid mapping strategy, we define the criteria for hardware merging. ST-Flow restricts merging to **compute nodes** (i.e., kernels performing arithmetic operations), while non-compute nodes (off-chip communication nodes and layout converters) are always given a dedicated unit, as their control logic is not amenable to time-multiplexing. Furthermore, nodes can only be merged if they share a common **type**, which we define as a structural signature encoding the loop nesting depth, the sequence of mathematical operations, and the ranks of the accessed tensors. While these constraints ensure architectural efficiency by avoiding the complexity of general-purpose modules, they remain flexible enough to capture a wide range of common workloads.

Algorithm 1: Hybrid Hardware Mapping Heuristic

Input: DFG $G = (V, E)$; similarity thresholds $\tau_{\text{trip}}, \tau_{\text{ub}}$
Output: Hardware mapping $\mathcal{M}$: hw-unit → set of node IDs

```
clusters ← {}     // type → list of node groups sharing one HW unit
foreach node v ∈ TopologicalSort(G) do
    if v is a Compute node then
        Find the first cluster C ∈ clusters[v.type] such that
            (i) every member of C is a predecessor of v in G, and
            (ii) v has similar arithmetic intensity to the last member of C
        if such C exists then
            add v to C
        else
            start a new singleton cluster {v} in clusters[v.type]
    else
        assign v its own dedicated hardware unit in M
foreach type t and its clusters [C_0, C_1, ...] in clusters do
    M[t_unit_k] ← C_k     for each k
return M
```

Algorithm 1 assigns compute nodes to hardware clusters while respecting structural compatibility and arithmetic-intensity similarity. For each compute node v, the algorithm attempts to place it in an existing cluster C of the same type if (i) every node in C is a predecessor of v in the DFG (i.e. on the same dependence chain), and (ii) v is structurally similar to the last member of C (u) based on two thresholds. Specifically, the total trip count ratio must satisfy:

$$r_{\text{trip}} = \frac{\max(TC_u, TC_v)}{\min(TC_u, TC_v)} \leq \tau_{\text{trip}} \tag{1}$$

where TC is the product of all loop bounds in a kernel. Furthermore, for each loop level i in a kernel of depth d, the individual upper bound ratio must satisfy:

$$r_i = \frac{\max(UB_{u,i}, UB_{v,i})}{\min(UB_{u,i}, UB_{v,i})} \leq \tau_{\text{ub}}, \quad \forall i \in \{0, \ldots, d-1\} \tag{2}$$

These constraints ensure that merged kernels have similar execution profiles, preventing excessive padding or under-utilization in the shared hardware. The thresholds expose a trade-off: tighter values (close to 1) merge only near-identical kernels and forfeit reuse opportunities, whereas looser values admit dissimilar kernels whose bound mismatch is absorbed as padding on the shared unit. We selected $\tau_{\text{trip}} = \tau_{\text{ub}} = 1.1$ empirically from a sweep of candidate values, as it yielded the best overall area-delay product across our benchmarks (Section 5); we use this setting in all experiments. If no eligible cluster exists, a new singleton cluster is started. The resulting clusters form the final mapping $\mathcal{M}$, which is used to materialize the merged hardware modules.

4.2 System Modeling

While EDA tools' reports have been used to estimate system performance [51], running EDA tools for a single design point can take minutes to hours making exploring huge design space infeasible. Consequently, we chose to rely on analytical models. Our mathematical system modeling is based on the modeling proposed in [5], which showed good performance and accurate cycle estimates. Due to page limit, we only give a high-level overview of the model here and the reader can refer to [5] for all the details. Given a directed acyclic graph (DAG) representing a DFG, where a node (n) represents a perfect loop nest and edges are the true/flow (read-after-write) dependencies, the model uses three main equations:

- $st(n)$: the start time of node n
- $fw(n)$: the time node n produces its first output
- $lw(n)$: the time node n produces its last output, which is also the end time of the node

These three equations capture the degree of overlap or lack thereof between two dependent nodes depending on the access order (loop permutation) between the producer (n') and consumer (n) nodes. When the dependent nodes have compatible access orders, producer node can forward ready data via stream/FIFO channels while it is executing, and $fw(n) < lw(n)$, otherwise, streaming is not feasible, and the consumer node must wait until the end of producer's execution, thus $fw(n) = lw(n)$. The full closed-form formula is constructed by chaining these three equations using the topological sort of the DAG. The overall latency of a DFG is then calculated as $lw(sink)$, where $sink$ is an artificial node connecting all the output nodes. The concrete evaluation of this formula depends on two sets of decision variables for each node n with a loop nest of depth d:

- B_n: a set of $d!$ binary indicator variables representing loop permutation, which determines the degree of concurrency
- X_n: a set of d integer variables representing the tiling factor of each dimension, which determines the degree of parallelism

Because the original formulation was limited to spatial dataflow accelerators, we extend and generalize that model with the following three contributions.

4.2.1 Unified Modeling. Two challenges arise when extending the model for temporal mapping and hybrid spatial-temporal mapping strategies. First, in the pure spatial mapping case, each kernel is mapped to a dedicated hardware module, thus, each node in the DFG has its own B_n and X_n variables. However, in the case of temporal mapping or hybrid spatial-temporal mapping, multiple kernels are time-multiplexed on a single hardware module. Therefore, our model enforces nodes to be merged to share the same permutation and tiling variable sets. Secondly, the original model relied on write and read access functions (WAF and RAF) to determine whether dependent nodes can execute concurrently. Now, when dependent nodes share the same hardware, they must execute sequentially regardless of their write/read access functions, and $fw(n) = lw(n)$ for these nodes.

4.2.2 Hardware Resource Estimation. The original formulation estimates total DSP (digital signal processing) compute resources of a design by summing, across all nodes, the product of each node's tiling and unroll factors with a scaling coefficient based on the node's arithmetic operation type. This is the most important factor to constrain the parallelism of a dataflow accelerator during design space exploration. However, another important aspect that was not considered in the original formulation is the on-chip memory utilization. As discussed in Section 3.2, when two nodes do not have compatible streaming access patterns after the DSE stage, an on-chip shared-buffer is needed for layout transformation. Therefore, to minimize the insertion of these costly converter nodes, we compute the potential footprint of all these converter nodes using the B_n and write/read access functions to determine whether the two dependent nodes require a converter or not. Note that, unlike the DSP estimation that actually reflects the number of DSPs used in the final design, our on-chip memory estimate is simply a proxy for on-chip memory utilization. We now have two metrics for hardware resource utilization ($Total_{DSP}$ and $Total_{Mem}$) that can be used to guide design space exploration.

4.2.3 Tiling and Unrolling Constraints. We use tiling and unrolling interchangeably because, in HLS, pipelining a tiled loop nest fully unrolls the inner (tile) loops into replicated hardware, while the outer loops define the temporal iteration bounds of that hardware; the tiling factor X_i thus directly sets the degree of hardware parallelism. The original formulation included equations of the form $Z_i \times X_i = UB_i$, where X_i is the tiling factor, UB_i is the loop upper bound, and Z_i is the trip count of the remaining time loop. These equations enforce that all tiling factors are strictly divisors of the original loop bounds. However, as we will show in Section 5, this greatly restricts the design space for parallelism allocation to the DFG's nodes, especially for temporal mapping and hybrid spatial-temporal mapping strategies. We, therefore, extended this formulation to allow non-divisor tiling factors. Furthermore, when kernels share the same hardware, we have different strategies for assigning the shared tiling loop bounds (UB_{shared}). This expands our tiling strategy to:

- **Div-Min**: Enforces divisor-only tiling where $X_i \times Z_i = UB_{shared}$. When multiple loops share hardware, $UB_{shared} = \min(UB_i)$. This maximizes shared hardware's runtime utilization, but potentially restricts maximum parallelism.
- **Non-Div-Min**: Allows non-divisor tiling with the constraints $X_i \times Z_i \geq UB_i$ and $X_i \times Z_i \leq X_i + UB_{shared} - 1$, where $UB_{shared} = \min(UB_i)$. This bounds $Z_i = \lceil UB_i / X_i \rceil$ ensuring dynamic latency padding.
- **Non-Div-Max**: Allows non-divisor tiling similar to **Non-Div-Min**, but sets $UB_{shared} = \max(UB_i)$. This maximizes potential parallelism at the cost of some runtime under-utilization for smaller loops sharing the same hardware.

4.3 Design Space Exploration

Now that we have a comprehensive unified closed-form formula to estimate the latency and hardware cost of a design point under loop permutation and tiling factor parameters, we can efficiently and reliably explore the massive design space of dataflow accelerators. Since our system modeling is independent of the specific design space exploration method, we are able to easily explore three different approaches.

4.3.1 Objective and Constraints. The objective of the DSE process is to find ***legal*** values for B_n and X_n that minimize the latency of a

dataflow design under a given hardware mapping strategy. Initially, we used the following equation as the DSE objective:

$$\min_{b \in B_n, x \in X_n} lw(\text{Sink}) \quad \forall n \in G \tag{3}$$

However, we noticed that in many cases the solutions found included redundant converter nodes that increased on-chip memory utilization without improving the performance. Therefore, we extended our objective to include the $Total_{Mem}$ term as follows:

$$\min_{b \in B_n, x \in X_n} \left(w_L \frac{lw(\text{Sink})}{Max_L} + w_M \frac{Total_{Mem}}{Max_{Mem}} \right) \quad \forall n \in G \tag{4}$$

where w_L and w_M are user-defined weights prioritizing latency and memory, respectively. Because latency (measured in clock cycles) and memory footprint (measured in bytes) operate on entirely different scales, we normalize each metric by its theoretical maximum to ensure balanced optimization. Specifically, Max_L is approximated as the sum of the trip counts across all nodes (representing the worst-case sequential execution), and Max_{Mem} is the sum of the buffer sizes for all edges in the DFG (representing the worst-case on-chip memory footprint where every dependency requires a converter).

Constraints. To ensure the formulated problem yields valid hardware configurations, the model strictly enforces the following four categories of constraints during the design space exploration:

(1) **Permutation (B_n) Constraints:** Each node n with loop depth d has $d!$ binary permutation candidate variables. The solver must select exactly one valid permutation per node. This is enforced as a one-hot constraint: $\sum_{i=1}^{d!} b_{n,i} = 1$.
(2) **Tiling Consistency Constraints:** To minimize the number of independent tiling variables (X_n) and avoid incompatible hardware generation, loop levels that share the same memory reference and dimension across the DFG are grouped. We use a Union-Find algorithm to enforce that such interdependent loop levels share a single unified X variable.
(3) **Tiling (X_n) Constraints:** As described in Section 4.2, the assigned tiling variables must adhere to the bounds and relations dictated by our selected tiling strategy (**Div-Min**, **Non-Div-Min**, or **Non-Div-Max**).
(4) **DSP Resource Constraints:** To respect the available hardware area, the accumulated compute requirement must not exceed a user-defined DSP_{limit}. For each compute node n, its DSP utilization is modeled as its intrinsic DSP factor multiplied by all its assigned tiling variables $\prod X_n$. The sum of DSP utilization across all nodes is constrained to be $\leq DSP_{limit}$.

Since our unified modeling and resource estimates are completely decoupled from the exploration algorithm itself, ST-Flow supports multiple search methods to navigate the massive space of loop permutations and tiling factors. For each method, we support two search strategies: (i) a *two-stage* approach that sequentially explores loop permutations then tiling factors to improve scalability, and (ii) a *combined* approach that explores both sets of parameters simultaneously to potentially find better global optima. We implemented the following three search methods.

4.3.2 Mixed Integer Non-Linear Programming (MINLP). In this method, we automatically translate the closed-form performance model and constraints into an AMPL mathematical formulation. This formulation is then solved by an external exact or heuristic MINLP solver. Because MINLP operates purely on the mathematical model, all hardware resource constraints, one-hot permutation rules, and tiling consistency equations are natively enforced by the solver engine. The solver generates candidates internally using mathematical optimization techniques to find optimal or high-quality solutions. While providing the highest quality results, this method may experience long execution times or infeasibility for large DFGs.

4.3.3 Simulated Annealing (SA). We use SA to iteratively explore the design space by mutating valid hardware configurations. To efficiently generate neighboring candidates, the mutation operator intrinsically preserves structural constraints. For example, when mutating a permutation configuration, it strictly maintains the one-hot constraint; when mutating a tiling factor, it randomly selects valid divisors and updates the associated loop trip counts accordingly. The generated candidate is then evaluated against systemic constraints (e.g., the DSP_{limit}) using a validation function. To avoid getting trapped in local minima, SA occasionally accepts worse valid solutions with a probability that decreases over time following a simulated cooling schedule.

4.3.4 Genetic Algorithm (GA). Our GA evolves a diverse population of valid candidates over multiple generations. To generate new candidates, GA applies a crossover operator that combines traits from two high-performing solutions selected via tournament selection. To minimize generating invalid offspring, the crossover operator carefully swaps entire groups of dependent variables (e.g., exchanging a complete loop permutation configuration from one parent). Subsequently, a mutation operator, identical to the one in SA, is applied. The population evolves by advancing the top-performing solutions to guarantee the global best configuration is never lost.

5 Evaluation

5.1 Experimental Setup

Table 2: Summary of Benchmarks and Evaluation Workloads

Benchmark	Config.	Key Features & Tested Challenges
DNN Workloads (from SOTA architectures)		
MLP-Mixer [53]	$N = 64, d = 128$	Alternating mixing logic; matrix transpose
RWKV-CM [41]	$d = 128$	ReLU, gating; non-linear activations
TabM [23]	8-head, $d = 256$	Fan-out logic; parallel stream management
SwiGLU [49]	$d = 128, d_{ffn} = 256$	SiLU, element-wise prod
DIN [64]	$64 \leftrightarrow 1024$	Asymmetric GEMM; var. expansion ratios
MNIST [35]	3-layer MLP	Classic digit recognition MLP
Synthetic Benchmarks* (topological stress tests)		
MM-Seq	7-GEMM chain	Inter-layer communication
MM-Tree	7-GEMM reduction	Non-linear DFG; parallel merging
MLP-Seq	Sequential	9-layer Deep MLP
MLP-Diamond	Branch-merge	Multi-path MLP (Figure 1)

*Synthetic benchmarks use both **Uniform** and **Random** (8 total) loop bounds to evaluate hybrid spatial-temporal mapping under diverse tiling constraints.

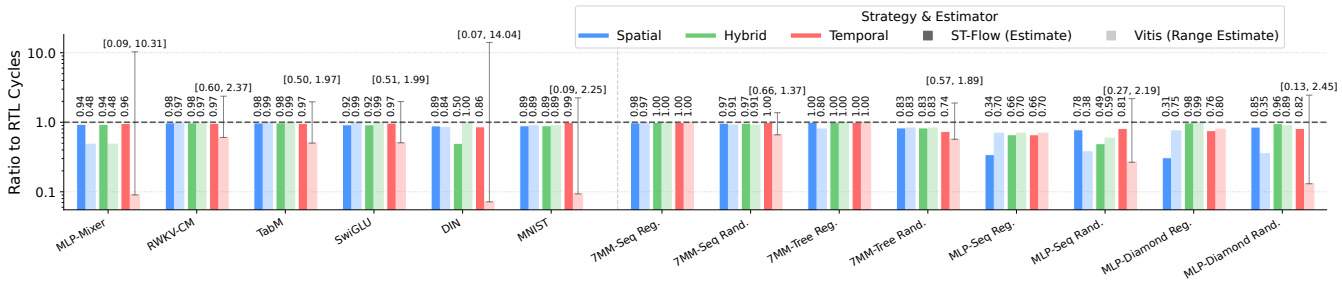


Figure 3: ST-Flow predicted cycles vs. Vitis HLS cycle estimates compared to RTL cycles (normalized).

Table 3: Experimental Configuration Grid

Category	Configurations / Parameters
HW Mapping	*Spatial*, *Hybrid* ($\tau_{\text{trip}} = \tau_{\text{ub}} = 1.1$, Sec. 4.1), *Temporal*
Tiling	**Div-Min**, **Non-Div-Min**, **Non-Div-Max**
DSE Solvers	**MINLP** (AMPL [4]/Gecode [48]), SA, GA
Objective	$w_L = 0.8$, $w_M = 0.2$ (Sec. 4.2)
SA Config	$T_{init} = 1000$, $\alpha = 0.999$, $10k$ iterations
GA Config	$Population = 50$, $Gen = 1000$, $\mu = 0.1$, $cr = 0.8$
Platform	Alveo U55C @ 300MHz (Vitis/Vivado 2024.2)
Total Exp.	**126 (Standard) + 80 (GPT-2)**

We evaluate ST-Flow on 14 workloads (Table 2) across a multi-dimensional design space summarized in Table 3. This exhaustive evaluation covers 126 standard configurations and an additional 80 experiments for a GPT-2 Tiny [46] case study (a full transformer block, $T = 64$, $d_{\text{model}} = 128$, 4 heads, $d_{\text{ff}} = 256$). We validate all configurations through C-simulation. The **MINLP** solver demonstrates high efficiency, typically converging in 20–30s. However, for GPT-2, the MINLP becomes prohibitive; we instead perform 20 independent runs for each heuristic method (SA/GA) using **Div-Min** tiling to accelerate these stochastic methods. We use a 500 DSP_{limit} for the 126 DNN and Synthetic designs and a 1000 DSP_{limit} for GPT-2 Tiny to reduce the runtime of the hardware synthesis and bitstream generation tasks, which take hours to days per run.

5.2 System Modeling Validation

First, we measure the accuracy and correlation of our mathematical closed-form formula in estimating the latency of our dataflow accelerators against RTL cycle-accurate simulation as the ground truth. Figure 3 shows the results for the **Div-Min** experiments, comparing the estimated cycles of our ST-Flow versus the Vitis HLS estimates, which provide a range of minimum and maximum cycle estimates. For the DNN models, our ST-Flow's estimates achieve a mean absolute percentage error (MAPE) of 8.24% with a near-perfect Pearson correlation of 0.9999. In contrast, Vitis HLS estimates show a MAPE between 31.41% and 158.05%, with correlation ranging from 0.7022 to 0.9738. For the Synthetic benchmarks, ST-Flow maintains a MAPE of 17.82% and a correlation of 0.9967, while Vitis HLS estimates range between 26.24% and 32.62% MAPE with correlations of 0.9868 to 0.9924. Over all the 126 experiments, ST-Flow achieves an aggregate MAPE of 17.38% and correlation of 0.9879, significantly outperforming the Vitis HLS min-max range of $[28.58\%, 80.38\%]$ MAPE and $[0.6931, 0.9639]$ correlation.

The largest errors occur when Vitis HLS fails to achieve the initiation interval (II) of 1 assumed by our model [5]. Inspecting the Vitis HLS logs of all 126 designs, 17 (13%) contain a compute kernel that misses its target II (achieving 2–16); the remaining II warnings affect only DMA repacking loops and do not reduce throughput. All 17 are *Spatial* (13) or *Hybrid* (4) designs of small-batch workloads (batch-1 MLPs, **DIN**, **MNIST**), where large unroll factors along the reduction dimension create a loop-carried dependence on the accumulator that Vitis HLS cannot break; no *Temporal* design is affected. Their MAPE is 38.9% (worst case 0.24× of the RTL cycles, **MLP-Seq Unif.**) versus 14.0% for the other 109. Since the model is optimistic for exactly these configurations, the DSE cannot penalize them; an accumulator reassociation pass or an II-aware DSE constraint is a direct fix left for future work.

Furthermore, we analyze the reliability of our resource modeling under strict constraints. Across 126 experiments targeting a budget of 500 DSPs, the ST-Flow DSE engine strictly respected the limit in all cases (100% worst-case), while the final bitstream implementation exhibited an average utilization of 77.5% and a worst-case of 108.8% (544 DSPs).

5.3 Divisor vs. Non-divisor Unroll Factors

In Figure 4, we evaluate the effectiveness of our non-divisor tiling strategies by comparing them against a traditional divisor-only baseline (**Div-Min**). We find that the performance benefits are highly dependent on the underlying hardware mapping strategy. The *Temporal* strategy is the primary beneficiary, showing an average speedup of 3.8× when flexible tiling (**Non-Div-Max**) is enabled. The *Hybrid* strategy also sees significant improvements (1.43×), while *Spatial* mappings show the least sensitivity (1.12×). These results demonstrate that as the hardware mapping strategy merges kernels of various sizes/shapes, the flexibility of the tiling strategy becomes a critical enabler for maximizing parallelism, preventing the tiling factors from being bottlenecked by divisors only or the smallest kernel in a model.

The most flexible tiling strategy (**Non-Div-Max**), which permits unroll factors larger than the smallest kernel dimensions, consistently outperforms the less flexible non-divisor strategy (**Non-Div-Min**) that is bounded by the smallest kernel. While non-divisor

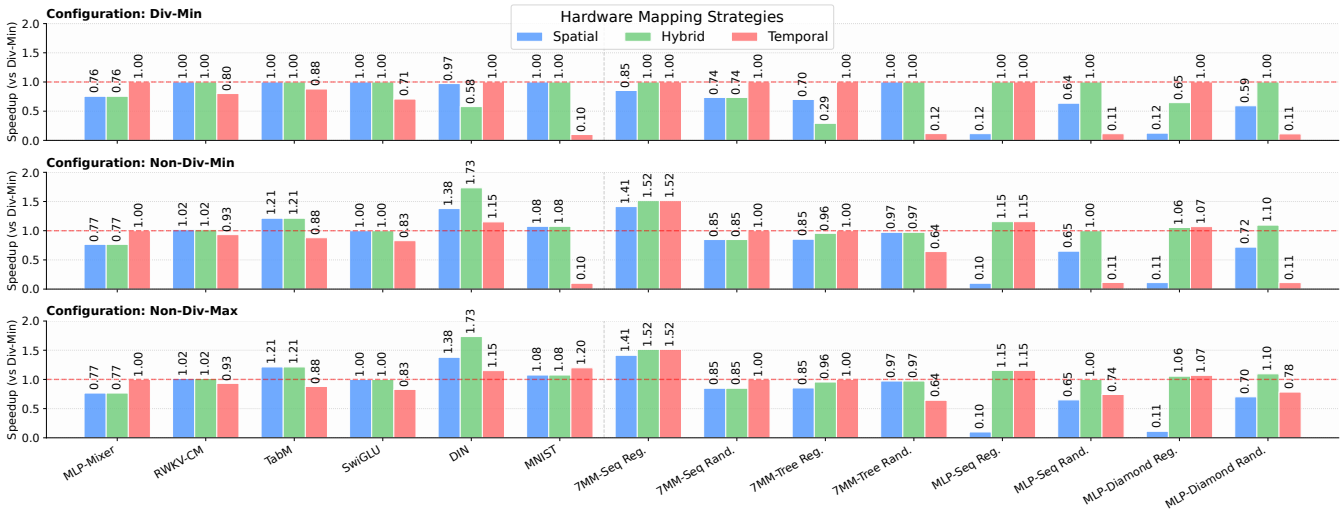


Figure 4: Normalized speedup for DNN and Synthetic workloads relative to the Div-Min baseline using RTL cycles.

tiling factors can recover performance over divisor only (**Div-Min**) by enabling using better tiling factors (e.g., 3.0× speedup in the **DIN** *Hybrid* benchmark), it is still unable to accommodate extremely unbalanced models. In extreme cases, such as the **MLP-Diamond Rand.** and **MNIST** workloads, **Non-Div-Max** achieves between 7.1× and 22.9× speedup relative to **Non-Div-Min**. This suggests that for modern dataflow accelerators, the ability to handle non-divisible dimensions without significant hardware or cycle-counting penalties is essential for supporting the diverse kernel shapes found in contemporary DNN architectures.

5.4 Hybrid Mapping Strategy

To evaluate the impact of hardware strategies and the efficacy of our Hybrid mapping strategy, we define an Area-Delay Product (ADP) metric. The delay component is derived from the total execution cycles obtained through RTL simulation (or on-board execution when applicable). The area component is represented by a Normalized Resource Score, which accounts for both DSP and Memory utilization relative to the target FPGA capacity (9,024 DSPs, 4,032 BRAMs, and 960 URAMs). Given the high density of URAMs on modern devices, we utilize a memory-equivalent formula ($BRAM + 16 \times URAM$) to unify heterogeneous storage resources into a single area contribution. The final Area Score is calculated as the sum of the percentage utilization of these two primary resource pools.

The aggregate results presented in Table 4 are calculated using the geometric mean of the ADP across all benchmarks within a given tiling configuration (e.g., **Div-Min**, **Non-Div-Min**). This approach ensures that performance improvements are not skewed by the varying execution scales of different workloads (e.g., small GEMM kernels vs. large Transformer layers). The Improvement (Imp.) factor quantifies the efficiency gain of each strategy relative to the worst strategy within the same tiling configuration, where a higher factor indicates a more optimal trade-off between hardware parallelism and resource consumption. This analysis allows us to verify that the *Hybrid* strategy consistently identifies the architectural "sweet spot," avoiding both the under-utilization of *Spatial* and the extreme serialization overheads often introduced by the *Temporal* approach.

Table 4: On-Board Aggregate ADP Breakdown (Geo. Means)

Config	Strategy	Delay	Area	ADP	Imp.
Div-Min	*Spatial*	188,273	0.107	20076	1.00×
	Hybrid	**154,720**	0.082	12652	1.59×
	Temporal	241,704	**0.067**	16085	1.25×
Non-Div-Min	*Spatial*	166,821	0.123	20441	1.00×
	Hybrid	**124,363**	0.120	14888	1.38×
	Temporal	239,193	**0.086**	20478	1.00×
Non-Div-Max	*Spatial*	167,156	0.123	20482	1.12×
	Hybrid	**138,959**	0.120	16650	1.38×
	Temporal	199,535	**0.115**	23000	1.00×

5.5 DSP Runtime Efficiency

The ADP metric captures the trade-off between area and delay, but it does not reveal how well the allocated compute resources are actually used at runtime. We therefore measure the *DSP runtime efficiency* of each design as the ratio of useful DSP work to available DSP capacity: the total DSP-weighted arithmetic operations of the model (using the same per-operation DSP costs as the resource model in Section 4.2, e.g., a multiply-accumulate costs 5 DSPs), divided by the product of the Vitis synthesis DSP count and the RTL execution cycles. An efficiency of 100% means every allocated DSP is busy every cycle. Table 5 reports the geometric mean across the 14 DNN and Synthetic workloads for each mapping and tiling strategy, together with the best mapping per workload (bottom row).

The *Temporal* strategy achieves the highest efficiency (69.2% on average, up to 99.8%), since a shared engine remains busy across the phases of all the kernels it serves. The *Spatial* strategy is the least efficient (40.1%): dedicated engines are sized per kernel and idle whenever a producer or consumer stalls, dropping below 10% for

Table 5: DSP Runtime Efficiency (Geo. Means, %)

Strategy	Div-Min	Non-Div-Min	Non-Div-Max	All
Spatial	38.1	41.2	41.1	40.1
Hybrid	64.3	64.7	64.7	64.5
Temporal	**77.6**	**71.0**	60.0	**69.2**
Best of three	81.2	78.1	69.6	76.2

the batch-1 MLP chains (**MLP-Seq Unif.**, **MLP-Diamond Unif.**) where consecutive layers offer little inter-kernel overlap. The *Hybrid* strategy sits in between (64.5%) while delivering the lowest latency (Table 4), confirming that it trades a fraction of engine utilization for parallelism only where it pays off. With the best mapping selected per workload, ST-Flow reaches a geometric mean efficiency of 76.2% across all 42 configurations. Finally, we observe that efficiency decreases as the tiling strategy becomes more flexible for *Temporal* (77.6% → 60.0%): unroll factors larger than the smallest kernel dimension leave lanes partially idle on the small kernels of a shared engine, which is the price paid for the latency gains reported in Section 5.3.

5.6 Comparison with Prior Work

In this section, we evaluate the performance and efficiency of ST-Flow relative to CODO [61], a recent state-of-the-art framework for streaming dataflow architectures. We omit direct comparisons with ScaleHLS [58] and HIDA [59] as they target shared-memory architectures and have been previously outperformed by Stream-HLS [5]. Since ST-Flow naturally generalizes the spatial-only mapping and performance model of Stream-HLS, our *Spatial* strategy with divisor-only tiling (discussed in Sections 5.3 and 5.4) serves as a representative baseline for that approach. We verified this empirically by running the open-source Stream-HLS and ST-Flow (*Spatial*, **Div-Min**) under identical settings (500-DSP budget and Vitis HLS 2024.2 targeting the Alveo U55C) on the seven Polybench kernels, **RWKV-CM**, and **MLP-Seq Rand.** Both frameworks select the same design point (identical predicted latency and DSP count) on all nine benchmarks except **Atax** and **RWKV-CM**, where ST-Flow's post-processing replaces an on-chip reuse buffer with a stream converter, and the RTL simulation cycle counts differ by at most 0.85% (**GEMM**: 133,189 vs. 132,057 cycles; **2MM**: 192,857 vs. 192,855), with ST-Flow being 2% faster on **RWKV-CM**. Direct comparison with StreamTensor [57] is precluded by its closed-source nature and its use of quantization in its experiments.

We built the open-source artifact of CODO [50] and generated designs using our DNN and Synthetic benchmarks and our GPT-2 Tiny model following the GPT-2 sample provided. For a fair comparison, we also added 7 samples from the Polybench [42] test suite used in their paper. To match ST-Flow's end-to-end execution model, we augmented the CODO designs with standardized AXI-DMA interfaces for off-chip communication. All CODO's tests passed C-simulation correctness except for Gemm and 2MM. For RTL simulation, GPT-2 Tiny and all DNN tests failed RTL simulation due to correctness issues, except for GPT-2 and MLP-Mixer, which resulted in a deadlock. Only the TabM example passed RTL simulation correctly, but it utilized over 17,000 DSPs, far exceeding the physical capacity of the Alveo U55C FPGA. We noticed

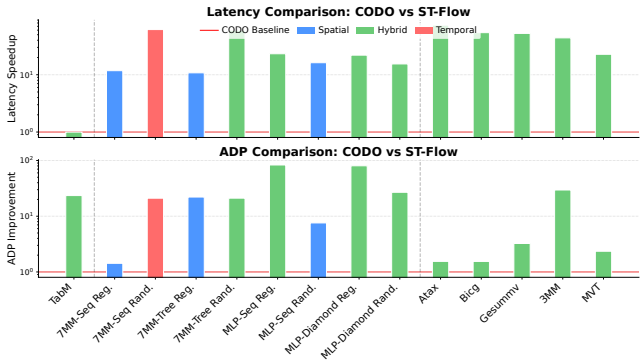


Figure 5: Normalized comparison of ST-Flow vs. CODO [61] for Latency Speedup and ADP Improvement using RTL cycles.

that CODO's design space exploration often bypassed user-defined resource limits.

Figure 5 summarizes the results normalized against the augmented CODO successful runs as a baseline. Our framework consistently outperforms CODO across all benchmark sets. Even when restricted to our most conservative divisor-only tiling strategy (**Div-Min**), ST-Flow achieves an aggregate latency speedup of 23.11× and an ADP improvement of 10.65× over the CODO baseline. These gains are particularly pronounced in Polybench and synthetic kernels, where ST-Flow achieves latency speedups of 46.15× and 22.21×, respectively. These results demonstrate the critical importance of ST-Flow's unified analytical modeling, resource-aware DSE, and DMA passes in identifying superior, hardware-feasible architectural configurations compared to the state-of-the-art.

5.7 Case Study: GPT-2

We analyze the efficiency of our heuristic design space exploration (DSE) and validate the final designs through physical on-board execution on a Xilinx Alveo U55C FPGA.

Table 6: GPT-2 DSE Comparison (Costs in 10^6)

Strat.	Method	Best	Mean	CV	Imp.
Spatial	**SA-Comb**	1.37	2.02	44.9%	16.0×
	SA-SA	1.21	2.30	52.1%	14.1×
	GA-Comb	1.78	2.56	35.1%	12.6×
	GA-GA	1.91	2.59	13.6%	12.5×
Hybrid	**SA-Comb**	1.25	1.96	38.0%	16.6×
	SA-SA	1.24	2.02	49.1%	16.0×
	GA-Comb	1.73	2.57	43.1%	12.6×
	GA-GA	1.56	2.32	17.4%	13.9×
Temporal	**SA-Comb**	2.41	6.00	23.9%	22.8×
	SA-SA	2.54	4.98	32.7%	27.5×
	GA-Comb	5.81	6.58	9.3%	20.8×
	GA-GA	2.50	5.37	28.4%	25.5×

5.7.1 Heuristic Search Efficiency. Table 6 reports, for each strategy and search method, the best and mean final cost over the 20 runs,

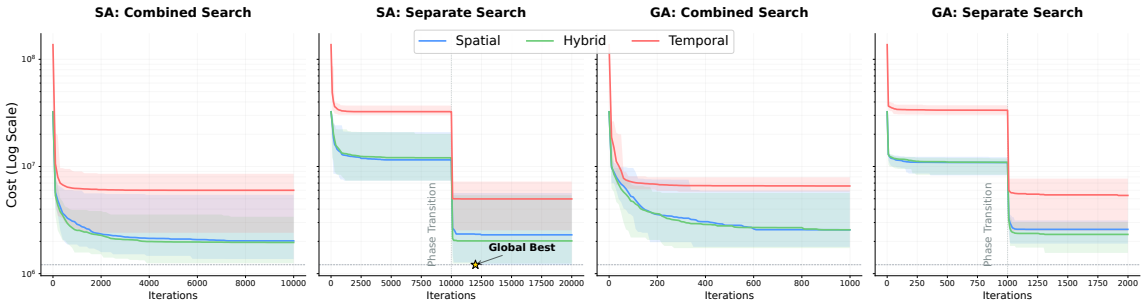


Figure 6: Convergence trace of our DSE heuristic for GPT-2 Tiny, illustrating the search efficiency in navigating the complex hardware design space across multiple search methods.

the coefficient of variation (CV), and the improvement (Imp.) of the mean final cost over the initial (iteration-0) cost of that strategy. SA-based methods generally identify superior hardware configurations, with **SA-SA** achieving a best-case cost of 1.21×10^6 for the *Spatial* strategy—a 36% improvement over **GA-GA**. However, this optimization comes at varying computational costs: on average, **SA-Comb** requires ~3 minutes per run, and **SA-SA** takes ~5.5 minutes. In contrast, the GA-based methods are significantly more expensive, with **GA-Comb** and **GA-GA** requiring ~18 minutes and ~32 minutes per run, respectively. The GA methods demonstrate higher stability, with **GA-Comb** maintaining a low Coefficient of Variation (CV) of 9.3%. The convergence traces in Figure 6 further illustrate that while SA explores deeper global minima, GA provides a more consistent optimization path, making it a robust alternative for highly constrained design spaces.

5.7.2 On-board Validation and Resource Utilization. We successfully deployed three GPT-2 Tiny designs to the Xilinx Alveo U55C FPGA, representing the *Spatial*, *Hybrid*, and *Temporal* strategies. As summarized in Table 7, the physical execution confirms the accuracy of our mathematical performance model, with an average prediction error of 26.8% relative to the physical cycle count. All designs achieved a robust clock frequency of approximately 292 MHz.

Table 7: On-board FPGA Evaluation and Resource Usage

Strategy	Method	DSP	BRAM	FPGA Cyc.	Freq.
Spatial	**SA-SA**	961	1,754	41×10^4	291 MHz
Hybrid	**SA-SA**	1,091	2,510	34×10^4	292 MHz
Temporal	**GA-GA**	1,037	2,224	68×10^4	292 MHz

In terms of resource utilization, the designs consume approximately 10.6% to 12.1% of the available DSP slices on the U55C, and between 1754 and 2510 BRAM units. The *Hybrid* strategy achieved the lowest hardware latency (0.34×10^6 cycles), benefiting from a more efficient trade-off between parallel processing units and local memory bandwidth compared to the more extreme reuse strategies. These results demonstrate that ST-Flow can generate high-frequency, resource-efficient hardware for complex transformer models while maintaining high performance predictability.

5.8 Compilation Reliability and Robustness

To evaluate the practical usability of ST-Flow, Table 8 breaks down the reliability of our ST-Flow compiler across the 126 DNN and Synthetic designs by mapping strategy. ST-Flow generated functional bitstreams for 105 designs, 104 of which were confirmed through on-board execution on the Alveo U55C. All 126 designs completed placement; the 21 failures occurred at routing, and a single *Hybrid* design failed on-board despite a successful bitstream. The failures correlate strongly with the mapping strategy: *Temporal* accounts for 15 of the 21 (36% of its designs), *Hybrid* for 5 (12%), and *Spatial* for only 1 (2%). This follows from how each strategy distributes the DSP budget. *Temporal* concentrates the entire budget into a few shared engines, producing large, densely connected modules whose fan-out and multiplexing logic congest routing, a problem well known even for hand-written designs [24]. *Hybrid* and *Spatial* spread the same budget across more, smaller modules that the placer can distribute over the device.

GPT-2 Tiny exhibits the same trend at a larger scale: for both *Spatial* and *Temporal*, the lowest-cost design found by the DSE failed routing and the deployed designs in Table 7 are the next-best candidates, whereas the *Hybrid* design routed on the first attempt. *Spatial* fails here for a different reason than *Temporal*: its dedicated kernels consume more total area (Table 4; the failed *Spatial* design used 2,438 BRAMs versus 1,754 for the deployed one), and at this scale area pressure rather than module density exhausts routing resources. These results indicate that *Hybrid* is not only the best in ADP but also the most robust option for large models, and that congestion-aware floorplanning [24] is a natural complement to ST-Flow for the *Temporal* extreme.

Table 8: Bitstream Generation and On-board Reliability

Strategy	Total	Success	Failure		
			Placement	Routing	On-board
Spatial	42	41	0	1 (2%)	0
Hybrid	42	36	0	5 (12%)	1
Temporal	42	27	0	15 (36%)	0
Total	**126**	**104**	**0**	**21 (17%)**	**1**

5.9 Limitations and Future Work

Our evaluation targets a single FPGA, on which fully spatial or temporal designs of billion-parameter LLMs are not feasible; nevertheless, the hybrid mapping heuristic and tiling strategies depend only on graph topology, kernel types, and loop-bound ratios, not on parameter count, and we plan to extend ST-Flow to multi-FPGA partitioning. **MINLP** becomes prohibitive beyond a few dozen nodes (Section 5.1); SA and GA remain tractable at any scale (up to ~32 minutes per run for GPT-2 Tiny) but without an optimality guarantee. Lastly, downstream EDA synthesis, placement, and routing dominate the total cost (hours to days per design), with routing failures growing with module density for *Temporal* (Section 5.8), for which congestion-aware floorplanning [24] would be a natural complement.

6 Related Work

High-Level HLS Frameworks. A plethora of research has elevated hardware design from RTL to higher-level software abstractions. Frameworks like PyLog [27], HeteroCL [32], Allo [12], and ScaleHLS [58] provide Python- or MLIR-based flows for accelerating affine programs. However, these tools primarily focus on optimizing individual kernels (e.g., via polyhedral loop transformations and pragma insertion) or rely on shared-memory architectures that execute kernels sequentially. In contrast, ST-Flow targets graph-level optimization for deeply pipelined dataflow architectures.

Streaming and Dataflow Accelerators. To overcome von Neumann architectural bottlenecks, several stream-centric frameworks have been proposed. Tools such as TAPA [13], Allo [12], Dato [20], and Stream-HLS [5] introduced robust streaming abstractions to enable true concurrent execution across hardware modules. Recent works, including Prometheus [43], StreamTensor [57], and CODO [61], build upon these paradigms to optimize end-to-end dataflow for complex workloads like LLMs. However, these systems predominantly map each graph node to a dedicated hardware unit (a purely spatial mapping). While maximizing concurrency, this "one-to-one" mapping severely limits scalability due to rapid compute and on-chip memory depletion. ST-Flow directly addresses this limitation by introducing a hybrid spatial-temporal mapping strategy that selectively time-multiplexes structurally equivalent compute nodes without sacrificing spatial parallelism where it matters most.

Systolic Arrays and Space-Time Mapping. Time-multiplexing computation onto a fixed set of processing elements is a classical idea. Early systolic-array work [31, 38] and its extension to fixed-size arrays [39] derive linear space-time transformations that assign each iteration of a single, uniform loop nest to a processor and a time step, and Feautrier's affine scheduling [21, 22] generalizes such schedules to arbitrary affine programs. Polyhedral compilers such as PolySA [17] and AutoSA [55] automate this space-time mapping for FPGAs, but each targets a single kernel mapped onto one systolic array. ST-Flow does not claim temporal mapping itself as novel; rather, it lifts the space-time decision from iterations of one loop nest to kernels of a multi-kernel dataflow graph, automatically deciding which structurally compatible kernels should share hardware and which should execute spatially, and integrating that decision with tiling, loop permutation, and stream-based inter-kernel concurrency in a single analytical DSE.

Hardware Design Space Exploration (DSE). Automating the search for optimal hardware configurations has transitioned from synthesis-in-the-loop search (e.g., AutoDSE [51]) to analytical and mathematical modeling (e.g., NLP-DSE [45], Sisyphus [44], and Odyssey [8]). While these tools efficiently explore unrolling, tiling, and pipelining directives, they generally operate at the single-kernel level or assume fully spatial mapping across the graph. ST-Flow expands the scope of analytical DSE by introducing a unified mathematical model that natively captures the complex latency and resource trade-offs of spatial-temporal execution, allowing efficient exploration of combined loop transformations and inter-task hardware sharing.

7 Conclusion

We presented ST-Flow, an end-to-end hardware compiler that automates the translation of PyTorch models into optimized spatial-temporal dataflow accelerators. Validated through 107 physical deployments on an Alveo U55C FPGA, including a complex GPT-2 model, ST-Flow demonstrates that model-driven hardware compilation is feasible for automating high-performance AI acceleration, effectively democratizing access to specialized hardware efficiency for next-generation workloads.

Acknowledgments

This work was partially supported by National Science Foundation (NSF) under Award No. 2533041 for NQVL, the CDSC industrial partners (https://cdsc.ucla.edu/partners/), the AMD HACC Programs, and the PRISM Center under the JUMP2.0 Program.

References

[1] Dennis Abts, Jonathan Ross, Jonathan Sparling, Mark Wong-VanHaren, Max Baker, Tom Hawkins, Andrew Bell, John Thompson, Temesghen Kahsai, Garrin Kimmell, Jennifer Hwang, Rebekah Leslie-Hurd, Michael Bye, E.R. Creswick, Matthew Boyd, Mahitha Venigalla, Evan Laforge, Jon Purdy, Purushotham Kamath, Dinesh Maheshwari, Michael Beidler, Geert Rosseel, Omar Ahmad, Gleb Gagarin, Richard Czekalski, Ashay Rane, Sahil Parmar, Jeff Werner, Jim Sproch, Adrian Macias, and Brian Kurtz. 2020. Think Fast: A Tensor Streaming Processor (TSP) for Accelerating Deep Learning Workloads. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 145–158. doi:10.1109/ISCA45697.2020.00023

[2] Amazon Web Services. 2024. AWS Inferentia. https://aws.amazon.com/machine-learning/inferentia/.

[3] AMD Xilinx. 2026. AMD Alveo U55C High Performance Compute Card. https://www.amd.com/en/products/accelerators/alveo/u55c/a-u55c-p00g-pq-g.html. Accessed September 2026.

[4] AMPL Optimization Inc. 2026. AMPL: A Modeling Language for Mathematical Programming. https://ampl.com/. Accessed September 2026.

[5] 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

[6] Suhail Basalama, Stéphane Pouget, and Jason Cong. 2026. *ST-Flow: A Hardware Compiler for Spatial-Temporal Dataflow Acceleration (PACT '26 Artifact)*. doi:10.5281/zenodo.21881993

[7] Suhail Basalama, Atefeh Sohrabizadeh, Jie Wang, Licheng Guo, and Jason Cong. 2023. FlexCNN: An End-to-End Framework for Composing CNN Accelerators on FPGA. *ACM Transactions on Reconfigurable Technology and Systems* 16, 2 (2023), 1–32. doi:10.1145/3570928

[8] Suhail Basalama, Jie Wang, and Jason Cong. 2025. A Comprehensive Automated Exploration Framework for Systolic Array Designs. In *Proceedings of the 60th Annual ACM/IEEE Design Automation Conference* (San Francisco, California, United States) *(DAC '23)*. IEEE Press, 1–6. doi:10.1109/DAC56929.2023.10248016

[9] Mark Bohr. 2007. A 30 year retrospective on Dennard's MOSFET scaling paper. *IEEE Solid-State Circuits Society Newsletter* 12, 1 (2007), 11–13.

[10] David Castells-Rufas, Santiago Marco-Sola, Juan Carlos Moure, Quim Aguado, and Antonio Espinosa. 2022. FPGA acceleration of pre-alignment filters for short read mapping with HLS. *IEEE Access* 10 (2022), 22079–22100.

[11] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. 2016. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13. doi:10.1109/MICRO.2016.7783710

[12] 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 (jun 2024), 28 pages. doi:10.1145/3656401

[13] 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.

[14] Nitin Chugh, Vinay Vasista, Suresh Purini, and Uday Bondhugula. 2016. A DSL compiler for accelerating image processing pipelines on FPGAs. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*. 327–338.

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

[16] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. 2011. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30, 4 (2011), 473–491.

[17] Jason Cong and Jie Wang. 2018. PolySA: polyhedral-based systolic array auto-compilation. In *Proceedings of the International Conference on Computer-Aided Design* (San Diego, California) *(ICCAD '18)*. Association for Computing Machinery, New York, NY, USA, Article 117, 8 pages. doi:10.1145/3240765.3240838

[18] William J Dally, Yatish Turakhia, and Song Han. 2020. Domain-specific hardware accelerators. *Commun. ACM* 63, 7 (2020), 48–57.

[19] Robert H Dennard, Jin Cai, and Arvind Kumar. 2018. A perspective on today's scaling challenges and possible future directions. In *Handbook of Thin Film Deposition*. Elsevier, 3–18.

[20] Shihan Fang, Hongzheng Chen, Niansong Zhang, Jiajie Li, Han Meng, Adrian Liu, and Zhiru Zhang. 2025. Dato: A task-based programming model for dataflow accelerators. *arXiv preprint arXiv:2509.06794* (2025).

[21] Paul Feautrier. 1992. Some efficient solutions to the affine scheduling problem. Part I. One-dimensional time. *International Journal of Parallel Programming* 21, 5 (1992), 313–347.

[22] Paul Feautrier. 1992. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *International Journal of Parallel Programming* 21, 6 (1992), 389–420.

[23] Yury Gorishniy, Ivan Rubachev, Artem Babkin, Artem Vorotnikov, and Artem Babenko. 2024. TabM: Advancing Tabular MLPs with Multi-Head Structures. *arXiv preprint arXiv:2407.00603* (2024).

[24] Licheng Guo, Yuze Chi, Jie Wang, Jason Lau, Weikang Qiao, Ecenur Ustun, Zhiru Zhang, and Jason Cong. 2021. AutoBridge: Coupling coarse-grained floorplanning and pipelining for high-frequency HLS design on multi-die FPGAs. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 81–92.

[25] Zifan He, Anderson Truong, Yingqi Cao, and Jason Cong. 2025. InTAR: Inter-Task Auto-Reconfigurable Accelerator Design for High Data Volume Variation in DNNs. In *2025 IEEE 33rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 123–132.

[26] Mark Horowitz. 2014. 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE international solid-state circuits conference digest of technical papers (ISSCC)*. IEEE, 10–14.

[27] Sitao Huang, Kun Wu, Hyunmin Jeong, Chengyue Wang, Deming Chen, and Wen-Mei Hwu. 2021. Pylog: An algorithm-centric python-based FPGA programming and synthesis flow. *IEEE Trans. Comput.* 70, 12 (2021), 2015–2028.

[28] Dillon Huff, Steve Dai, and Pat Hanrahan. 2021. Clockwork: Resource-efficient static scheduling for multi-rate image processing applications on FPGAs. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 186–194.

[29] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Clifford Young, Xiang Zhou, Zongwei Zhou, and David A Patterson. 2023. TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 82, 14 pages. doi:10.1145/3579371.3589350

[30] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. *SIGARCH Comput. Archit. News* 45, 2 (jun 2017), 1–12. doi:10.1145/3140659.3080246

[31] H. T. Kung. 1982. Why systolic architectures? *Computer* 15, 1 (1982), 37–46. doi:10.1109/MC.1982.1653825

[32] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. 2019. HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) *(FPGA '19)*. Association for Computing Machinery, New York, NY, USA, 242–251. doi:10.1145/3289602.3293910

[33] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A compiler infrastructure for the end of Moore's law. *arXiv preprint arXiv:2002.11054* (2020).

[34] 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.

[35] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[36] Sean Lie. 2022. Cerebras architecture deep dive: First look inside the hw/sw co-design for deep learning: Cerebras systems. In *2022 IEEE Hot Chips 34 Symposium (HCS)*. IEEE, 1–34.

[37] Michael Lo, Zhenman Fang, Jie Wang, Peipei Zhou, Mau-Chung Frank Chang, and Jason Cong. 2020. Algorithm-hardware co-design for BQSR acceleration in genome analysis toolkit. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 157–166.

[38] Dan I. Moldovan. 1983. On the design of algorithms for VLSI systolic arrays. *Proc. IEEE* 71, 1 (1983), 113–120. doi:10.1109/PROC.1983.12532

[39] Dan I. Moldovan and Jose A. B. Fortes. 1986. Partitioning and mapping algorithms into fixed size systolic arrays. *IEEE Trans. Comput.* C-35, 1 (1986), 1–12. doi:10.1109/TC.1986.1676652

[40] Fahad Bin Muslim, Liang Ma, Mehdi Roozmeh, and Luciano Lavagno. 2017. Efficient FPGA implementation of OpenCL high-performance computing applications via high-level synthesis. *IEEE Access* 5 (2017), 2747–2762.

[41] Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Stella Biderman, Huanqi Cao, Xin Cheng, Michael Chung, Leon Derczynski, Xingjian Du, Matteo Grella, et al. 2023. RWKV: Reinventing RNNs for the Transformer Era. In *Findings of the Association for Computational Linguistics: EMNLP 2023*. 14048–14077.

[42] Louis-Noël Pouchet and Tomofumi Yuki. 2012. Polybench: The polyhedral benchmark suite. *URL: http://www.cs.ucla.edu/pouchet/software/polybench* (2012).

[43] Stéphane Pouget, Michael Lo, Louis-Noël Pouchet, and Jason Cong. 2025. Holistic Optimization Framework for FPGA Accelerators. *ACM Trans. Des. Autom. Electron. Syst.* 31, 1, Article 7 (Nov. 2025), 37 pages. doi:10.1145/3769307

[44] Stéphane Pouget, Louis-Noël Pouchet, and Jason Cong. 2025. A Unified Framework for Automated Code Transformation and Pragma Insertion. 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, 187–198. doi:10.1145/3706628.3708873

[45] Stéphane Pouget, Louis-Noël Pouchet, and Jason Cong. 2025. Automatic Hardware Pragma Insertion in High-Level Synthesis: A Non-Linear Programming Approach. *ACM Trans. Des. Autom. Electron. Syst.* 30, 2, Article 26 (Feb. 2025), 44 pages. doi:10.1145/3711847
[46] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.
[47] Robert R Schaller. 1997. Moore's law: past, present and future. *IEEE Spectrum* 34, 6 (1997), 52–59.
[48] Christian Schulte, Guido Tack, and Mikael Z. Lagerkvist. 2026. Gecode: Generic Constraint Development Environment. https://www.gecode.dev/. Release 6.4.0.
[49] Noam Shazeer. 2020. GLU Variants Improve Transformer. *arXiv preprint arXiv:2002.05202* (2020).
[50] SJTU Zhao Lab. 2026. CODO: An Automated Compiler for Comprehensive Dataflow Optimization. https://github.com/sjtu-zhao-lab/codo-artifact.
[51] Atefeh Sohrabizadeh, Cody Hao Yu, Min Gao, and Jason Cong. 2022. AutoDSE: Enabling software programmers to design efficient FPGA accelerators. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 27, 4 (2022), 1–27.
[52] The LLVM Project. 2024. Torch-MLIR Project. https://github.com/llvm/torch-mlir.
[53] Ilya O Tolstikhin, Neil Houlsby, Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Thomas Unterthiner, Jessica Yung, Andreas Steiner, Daniel Keysers, Jakob Uszkoreit, et al. 2021. MLP-Mixer: An all-MLP architecture for vision. *Advances in Neural Information Processing Systems* 34 (2021), 24261–24272.
[54] Yaman Umuroglu, Nicholas J Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. 2017. FINN: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*. 65–74.
[55] Jie Wang, Licheng Guo, and Jason Cong. 2021. AutoSA: A polyhedral compiler for high-performance systolic arrays on FPGA. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 93–104.
[56] Dennis Weller, Fabian Oboril, Dimitar Lukarski, Juergen Becker, and Mehdi Tahoori. 2017. Energy efficient scientific computing on FPGAs using OpenCL. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 247–256.
[57] Hanchen Ye and Deming Chen. 2025. StreamTensor: Make Tensors Stream in Dataflow Accelerators for LLMs. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*. 201–216. doi:10.1145/3725843.3762817
[58] 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.
[59] 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*. 215–230.
[60] Chen Zhang, Guangyu Sun, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. 2019. Caffeine: Toward Uniformed Representation and Acceleration for Deep Convolutional Neural Networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 11 (2019), 2072–2085. doi:10.1109/TCAD.2017.2785257
[61] 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)*. 1–17. doi:10.1109/ISCA66397.2026.00018
[62] Xiaofan Zhang, Hanchen Ye, Junsong Wang, Yonghua Lin, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. 2020. DNNExplorer: a Framework for Modeling and Exploring a Novel Paradigm of FPGA-based DNN Accelerator. In *Proceedings of the 39th International Conference on Computer-Aided Design*. 1–9.
[63] Yichi Zhang, Junhao Pan, Xinheng Liu, Hongzheng Chen, Deming Chen, and Zhiru Zhang. 2021. FracBNN: Accurate and FPGA-efficient binary neural networks with fractional activations. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 171–182.
[64] Guorui Zhou, Xiaoqiang Zhu, Chenru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. 2018. Deep Interest Network for Click-Through Rate Prediction. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 1059–1068.

A Artifact Appendix

Overview. ST-Flow is a full-stack compiler from PyTorch models to FPGA bitstreams, so the results in Section 5 originate from four layers: the analytical DSE solver, Vitis HLS C-simulation, Vitis/Vivado synthesis and place-and-route, and on-board execution on an Alveo U55C [3]. Reproducing all 206 experiments from source would take days to weeks of licensed Vivado time and a physical U55C. The artifact [6] therefore archives every experiment with its generated sources, solver traces, HLS/Vivado reports and on-board logs (5.6 GB, 41,873 files), together with the compiler source, a prebuilt LLVM/MLIR toolchain and pinned Python wheels. It is self-contained and requires no network access during evaluation.

Evaluation tiers. The artifact is organized in four independent tiers of increasing tooling requirements:

- **Tier 1 – Compiler correctness** (Linux, Python 3.12, Vitis HLS 2024.2; ∼40 min). Builds the compiler and runs a 66-test suite (Polybench, DNN, Synthetic, and transformer front-ends), each compiling a PyTorch model and validating the generated design against a PyTorch reference via C-simulation.
- **Tier 2 – Paper reproduction** (Python only; ∼11 s). A single script, `scripts/ae_verify.sh`, replays the analysis pipeline from the archived reports and regenerates every figure and table of Section 5: the model validation (Fig. 3), non-divisor speedups (Fig. 4), the ADP summary (Table 4), the CODO comparison (Fig. 5), the GPT-2 DSE study (Fig. 6, Table 6), the on-board results (Table 7) and the robustness statistics (Table 8). All CSVs and LaTeX tables are byte-identical to the shipped references, and the figure PDFs are content-identical (differing only in embedded timestamps). The MAPE/correlation and speedup statistics quoted in Section 5.2 and Section 5.3 are printed during regeneration.
- **Tier 3 – Per-design regeneration** (Vitis HLS 2024.2; minutes to hours per design). Re-runs DSE, HLS synthesis and RTL cosimulation for individual designs and compares the resulting latency, RTL cycles and resource counts against the archived `main.csv` rows. **MINLP** designs reproduce these metrics exactly on the same tool version; SA/GA designs are stochastic by construction and land on comparably good but different points.
- **Tier 4 – Bitstream and on-board execution** (Vivado 2024.2, XRT, Alveo U55C; hours to days per design). Each design directory is self-contained with a plain `Makefile` for bitstream generation and on-board validation; SLURM drivers are provided as a convenience for the full 126-design sweep.

Baseline. The CODO [61] designs of Section 5.6 were generated from its open-source artifact [50] with only top-level pragmas changed to use AXI-DMA interfaces. The generated designs and full Vitis reports are archived, and the comparison data is re-derived from the raw reports as part of Tier 2.

Scope. The artifact does not claim bit-exactness of synthesized netlists or bitstreams across Vitis/Vivado patch levels, nor of re-derived SA/GA design points; absolute solver runtimes (e.g., 20–30 s for **MINLP**) are machine-dependent. Tiers 1 and 2 were re-validated from a clean clone on a 20-core x86_64 workstation with Python 3.12, LLVM/MLIR 20, Vitis HLS 2024.2 and AMPL [4]/Gecode [48].