HIDA: A Hierarchical Dataflow Compiler for High-Level Synthesis

Hanchen Ye
University of Illinois
Urbana-Champaign
United States
hanchen8@illinois.edu

Hyegang Jun
University of Illinois
Urbana-Champaign
United States
hgjun2@illinois.edu

Deming Chen
University of Illinois
Urbana-Champaign
United States
dchen@illinois.edu

Abstract

Dataflow architectures are growing in popularity due to their potential to mitigate the challenges posed by the memory wall inherent to the Von Neumann architecture. At the same time, high-level synthesis (HLS) has demonstrated its efficacy as a design methodology for generating efficient dataflow architectures within a short development cycle. However, existing HLS tools rely on developers to explore the vast dataflow design space, ultimately leading to suboptimal designs. This phenomenon is especially concerning as the size of the HLS design grows. To tackle these challenges, we introduce *HIDA*[1], a new scalable and hierarchical HLS framework that can systematically convert an algorithmic description into a dataflow implementation on hardware. We first propose a collection of efficient and versatile dataflow representations for modeling the hierarchical dataflow structure. Capitalizing on these representations, we develop an automated optimizer that decomposes the dataflow optimization problem into multiple levels based on the inherent dataflow hierarchy. Using FPGAs as an evaluation platform, working with a set of neural networks modeled in PyTorch, HIDA achieves up to 8.54× higher throughput compared to the state-of-the-art (SOTA) HLS optimization tool. Furthermore, despite being fully automated and able to handle various applications, HIDA achieves 1.29× higher throughput over the SOTA RTL-based neural network accelerators on an FPGA.

1 Introduction

With the decline of Moore's law, it is no longer possible to expect the price of computation to decrease year over year. As a result, *customized* and *domain-specific* accelerators are becoming well accepted in combating the physical limitations of silicon, including those implemented on ASICs [24, 28, 40] and reconfigurable platforms, such as FPGAs [72, 75, 84]. Historically, the cost of developing hardware accelerators has always remained astronomically high. In this context, high-level synthesis (HLS) is a promising solution that can *synthesize* high-level algorithmic description to a hardware description language (HDL) implementation [12].

Dataflow Architecture. An important computation architecture for customized hardware accelerators is *dataflow*, which enables the parallel temporal execution of multiple coarse-grained tasks [50, 59, 78]. Unlike the Von Neumann architecture that constantly grapples with the memory wall, dataflow architecture can exploit the on-chip communication between tasks to avoid frequent external memory access. As long as an application is dataflow feasible, a well-designed dataflow architecture can efficiently execute the application with reduced power and bandwidth utilization [27, 65, 67].

Existing Dataflow Approaches. Commercial HLS tools typically provide programming interfaces for users to implement dataflow structures, such as the AMD Vitis HLS `dataflow` directive [34], Intel HLS *system of tasks* [35], and LegUp `thread` APIs [36]. However, it is still difficult to implement a dataflow-oriented HLS design with sequential languages, such as C/C++. Therefore, academic HLS tools have pushed for approaches that decouple algorithm specification from hardware customizations in compute, data types, and memory [4, 33, 43, 68], or introduce specialized HLS primitives [10, 29, 42, 49]. These approaches have effectively improved productivity and quality compared to industrial HLS tools. Note that there also exist recent frameworks [26, 70] that can automatically generate dataflow design without manual code rewriting. However, these automated tools cannot systematically model dataflow architectures, limiting them to the generation of suboptimal simple designs.

Unexplored Opportunities. Although existing HLS tools can enable dataflow designs, they still heavily rely on the user to make the hard design decisions, including but not limited to parallelization strategy, tiling strategy, memory hierarchy, data layout, etc. More importantly, the design spaces of different tasks in the dataflow are tightly coupled with each other due to two reasons: (1) an efficient dataflow architecture demands the latency to be balanced across different tasks as the critical task determines the overall achievable performance; (2) the inter-task communications are often established through streaming channels or on-chip buffers instead of hierarchical shared memory. Meanwhile, large-scale dataflow often gravitates towards a *hierarchical* structure, as dataflow tasks are naturally represented by nested graphs, further complicating the design space.

As a result, the vast design space can prohibit programmers from reasoning about various design choices and finding the optimized design point. This can eventually lead

[1] https://github.com/UIUC-ChenLab/ScaleHLS-HIDA

to non-ideal performance and efficiency, thereby thwarting the promise of existing dataflow approaches. We observed that many HLS-augmentation tools have proposed DSE engines using different algorithms, including polyhedral techniques [1, 80, 86, 87], graph analysis [32, 71, 79, 83], and machine learning [26, 41, 64, 73]. These tools can effectively explore the local design space of a single task or kernel. However, they cannot handle the dataflow-oriented exploration of multiple tasks due to the inter-task coupling and the complicated dataflow hierarchy.

HIDA Approach. With the discussion above, we concluded that the challenges presented in the design and optimization of dataflow architecture *cannot* be fully addressed by existing HLS approaches, which rely on programmers to explore the vast design space manually. We argue that compilers will and should play an important role in the design process - the hierarchical characteristics of dataflow architecture should be systematically represented and modeled, on which an optimization pipeline should be built to handle the inter- and intra-task optimizations comprehensively.

Under this mantra, we propose *HIDA*, an HLS framework with hierarchical dataflow intermediate representations (IR) and optimizations, enabling the automated transformation of algorithmic hardware descriptions to efficient dataflow architectures. The main contributions of HIDA are as follows:

- We propose a new dataflow IR called HIDA-IR that models dataflow at two different levels of abstraction, *Functional* and *Structural*, to capture the dataflow characteristics and multi-level hierarchy, enabling effective optimizations.
- We propose a new dataflow optimizer called HIDA-OPT, featuring a pattern-driven task fusion algorithm and an intensity- and connection-aware dataflow parallelization algorithm geared toward maximum efficiency.
- We enable an end-to-end and extensible compilation stack supporting PyTorch and C++ inputs, empowering the user to rapidly experiment with various design parameters and prototype new dataflow architectures.
- We perform comprehensive FPGA evaluations of HIDA. On a set of neural networks, HIDA achieves 8.54× and 1.29× higher throughputs over the SOTA HLS optimization framework and RTL-based neural network accelerator.

2 Motivation

Due to the inherent disjoint between the Von Neumann-centric programming model and the dataflow programming model, it is easy for HLS designs to leave a large portion of the achievable performance on the table. To better understand the challenges presented in existing HLS tools, we implemented an HLS-based LeNet [47] accelerator on an AMD PYNQ-Z2 FPGA as a case study.

Design Process. Table 1 shows the structure of the LeNet model, which consists of 6 layers in total. We followed the steps below to design the HLS-based accelerator:

1. We rewrote the LeNet model in C++ as the baseline design, which is also used for the testbench and simulation in all subsequent steps. (2 hours)
2. We applied *layer fusion* and *parallelization* to the baseline design following the strategy summarized in Table 1. The listed parallel factors are selected with heuristics [76] and implemented with manual loop tiling and loop `unroll` directive insertion. (10 hours)
3. We rewrote the design to enable coarse-grained *dataflow* and loop *pipeline*. Specifically, we outlined all tasks, implemented off-chip memory interfaces, and partitioned inter-task on-chip buffers with heuristics [66]. (8 hours)
4. We iterated on different settings of parallel factors and directive configurations by rewriting and evaluating the design in AMD Vitis HLS until we were satisfied with the quality of the results. (20 hours)

Overall, we spent around 40 hours designing and fine-tuning the HLS-based LeNet accelerator. Then, we parameterized all six parallel factors listed in Table 1 and developed a TCL script exhaustively traversing each configuration under both dataflow and non-dataflow settings. This took another 170 hours. Finally, as a comparison, we automatically generated an HIDA-based design, which took 0.4 minutes to compile and 9.5 minutes for AMD Vitis HLS to generate RTL.

Results and Analysis. Figure 1 shows the exhaustive search results of the LeNet accelerator in the throughput-resource space. Table 2 summarizes the evaluation results and development cycles of the expert design, the best design from the exhaustive search, and the HIDA design. The key observations are summarized as follows:

- *Dataflow designs are Pareto-dominating.* We can clearly observe a large throughput/resource gap between the Pareto frontiers with and without dataflow. The best dataflow design achieves 3.13× higher throughput than the non-dataflow counterpart under the same resource constraints.
- *Dataflow cannot guarantee a good trade-off.* We can observe tons of dataflow designs dominated by non-dataflow designs. Under the same resource constraints, the best non-dataflow design achieves 3.83× higher throughput than the dataflow design with the worst quality.
- *Dataflow design space is vast.* In the layer fusion, spatial parallelization, and array partition steps, we have pruned a large amount of design points based on heuristics. However, the resulting design space still contains more than 2.4×10^4 points and costs hundreds of CPU hours to search exhaustively, owing to the fact that each design point takes 2-10 minutes for Vitis HLS to evaluate.
- *Dataflow design space is difficult to comprehend.* In Table 2, we can observe that the *exhaustive* design achieves 1.20× higher throughput than the hand-tuned *expert* design. We attribute this to the inter-task design space coupling. The complicated dataflow design space makes it substantially

Table 1. LeNet accelerator design. *CPF* and *KPF* denote the channel and kernel parallel factor.

Layer	Task	Factor	Range
(All Layers)	-	*BATCH*	{1, 5, 10, 15, 20}
Conv+ReLU / Pool	Task1	KPF_{task1}	{1, 2, 3, 6}
Conv+ReLU / Pool	Task2	KPF_{task2} CPF_{task2}	{1, 2, 4, 8, 16} {1, 2, 3, 6}
Conv+ReLU	Task3	KPF_{task3} CPF_{task3}	{1, 2, 3, 4, 6, 8} {1, 2, 4, 8, 16}
Linear	Task4	-	-

Table 2. Evaluation results of LeNet.

	Expert	Exhaustive	HIDA
Resource Util.	95.5%	99.2%	95.0%
Throu. (Imgs/s)	41.6k	49.9k	53.2k
Develop Cycle	40 hours	210 hours	9.9 mins

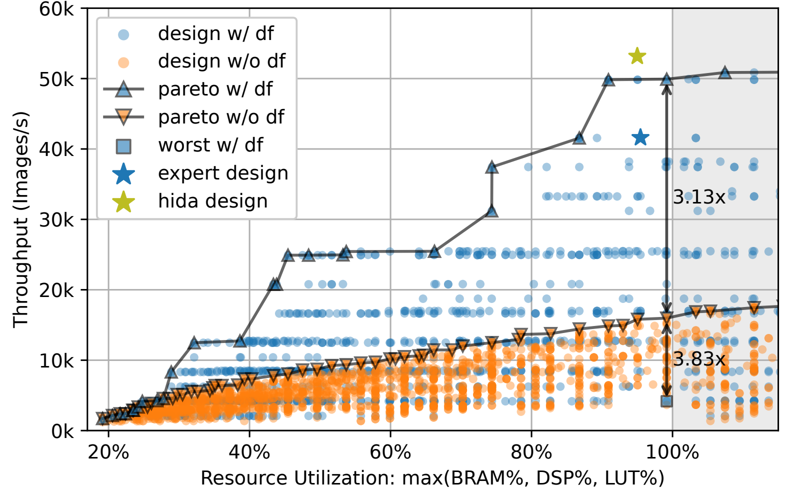

Figure 1. Exhaustive design space search of the LeNet accelerator. *w/ df* and *w/o df* indicate whether dataflow is enabled. *Worst w/ df* is the dataflow design with the worst quality. *Expert design* is hand-tuned by HLS experts. *HIDA design* is automatically generated from HIDA.

difficult to find the optimized design point by reasoning about the trade-off empirically.

- *Automated tool outperforms exhaustive search.* HIDA further improves the throughput of the *exhaustive* design by 1.06×. After comparing the two designs, we found HIDA to have automatically explored additional parallelizable dimensions apart from the six in the *Factor* column of Table 1, such as the feature map width and height dimensions, presenting a path that can increase the design efficiency further. This indicates that using heuristics suitable for single-task or non-dataflow designs may accidentally prune away valuable design points for dataflow designs.

Need for Scalable Automation Tools. For the LeNet case study, while the expert design took tens of hours to develop and ended up with a sub-optimal design, HIDA only took minutes to generate the design and achieved the best quality of results. One should expect that as the complexity and size of the target design increase, the development time will grow dramatically, while the manual design quality will decrease due to the vast and complicated dataflow design space. In summary, *productivity*, *performance*, and *scalability* problems of dataflow architecture are the three strong motivators for a scalable HLS optimization tool.

3 Background

3.1 MLIR Framework

MLIR [20, 46] is a compilation framework supporting multiple levels of functional and representational hierarchy. In the remainder of this paper, we use *MLIR* to refer to the MLIR framework and *IR* for the intermediate representation of

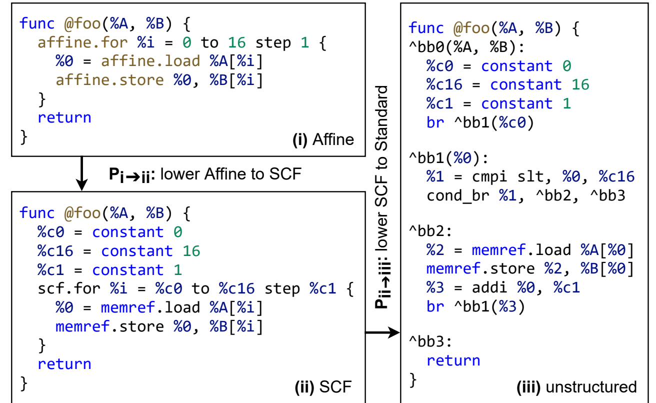

Figure 2. An IR example. `affine` and `scf` dialect are structured control flow IRs that can be lowered to unstructured IR. All types are omitted for simplicity.

programs in MLIR. MLIR includes a single static assignment (SSA) style IR [23] where an *Operation* is the minimal unit of code. Each operation accepts a set of typed *Operands* and produces a set of typed *Results*. Connections between the results of one operation and the operands of another operation describe the SSA-style flow of data. For instance, `%3 = addi %0, %c1` in Figure 2(iii) is an operation with operands `%0` and `%c1` and result `%3`. Each operation can also be parameterized by a set of *Attributes* indicating important characteristics of the operation. Unlike operands, which typically model values produced by other operations when a program is executed, attributes have values that are known and fixed at compile time. A sequential list of operations without control

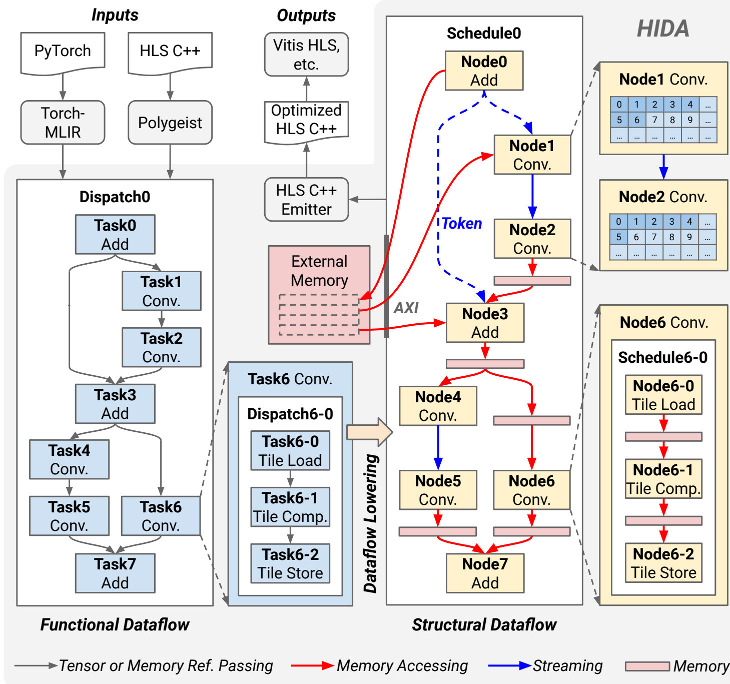

Figure 3. HIDA framework overview.

Table 3. HIDA-IR key operations. *Region* is a sequential list of operations to be executed.

Operation	Description
	Functional Dataflow
`task`	Own a transparent region, can contain nested dispatch operations with sub-tasks.
`dispatch`	Launch multiple tasks in its region.
	Structural Dataflow
`node`	Own an isolated region, can contain nested schedule operations with sub-nodes. Carry explicit I/O memory effect information.
`schedule`	An isolated region with multiple nodes. Carry explicit scheduling information.
`buffer`	A buffer with variadic stages and ports and automatic ping-pong buffering semantics. Carry explicit partition and layout information.
`stream`	A stream channel with variadic entries.
	Module Interface
`port`	A memory or stream port with explicit type.
`bundle`	A named bundle of ports.
`pack`	Pack an external memory block into a port.

flow is defined as a *Block* and a control flow graph (CFG) of blocks is organized into a *Region* in MLIR. Regions are, in turn, contained by operations, enabling the description of arbitrary design hierarchy. In MLIR, *Function* is defined as a built-in callable operation always owning one region. For instance, function `@foo` in Figure 2(iii) owns one region containing four blocks, `bb0` to `bb3`.

A *Dialect* in MLIR defines a namespace for a group of related operations, attributes, and types. MLIR not only provides multiple built-in dialects to represent common functionalities, but also features an open infrastructure allowing to define new dialects at different abstraction levels. *Pass* is a key component of compiler which traverses the IR for the purpose of optimization or analysis. Similar to LLVM [44], users can design *Transform* and *Analysis* passes in MLIR to perform the IR transformation and analysis. However, in the context of MLIR, *Transform* typically refers to the transformation within a dialect. The transformation between different dialects is typically referred as *Conversion*, while the transformation between MLIR and external representation is referred as *Translation*. *Lowering* is a terminology referring to the process of lowering the abstraction level of IR.

3.2 Relevant MLIR Dialects

Many dialects in MLIR are immediately applicable for representing nested loop programs commonly used in HLS. The `linalg` dialect provides a structured representation of linear algebra operations. The `affine` dialect provides a powerful abstraction for affine operations in order to make dependence analysis and loop transformations efficient and reliable. The `affine` dialect defines *Affine Map* as a mathematical function that transforms a list of affine values into a list of results. Affine operations (e.g., `affine.for` and `if`) must take affine values as input operands, therefore the loop bounds of `affine.for` operation and conditions of `affine.if` operation must be the expression of affine values. The `scf` (structured control flow) dialect defines control flow operations (e.g., `scf.for` and `if`) whose loop bounds or conditions can be any SSA values. Therefore, `scf` operations are not constrained by the affine requirements and can represent a wider range of programs. MLIR also provides several fundamental built-in dialects to represent basic arithmetic operations (e.g., `addf`), unstructured control flow operations (e.g., `br` and `cond_br`), and memory-related operations (e.g., `load` and `store`). Taking Figure 2 as an example, the structured control flows in Figure 2(i) and (ii) represented with `affine` and `scf` operations are flattened to the unstructured `br` and `cond_br` operations in Figure 2(iii).

4 HIDA Overview

Figure 3 shows the overall architecture of HIDA. HIDA is built on top of the MLIR infrastructure [20, 45] and can take deep learning models written in PyTorch [57] or generic HLS C++ code as design entries and produce optimized HLS C++ code. For the PyTorch and C++ inputs, we use Torch-MLIR [22] and Polygeist [51] as front-ends to parse source codes. After the optimizations are completed in HIDA, we use an HLS C++ emitter [70] to generate synthesizable HLS

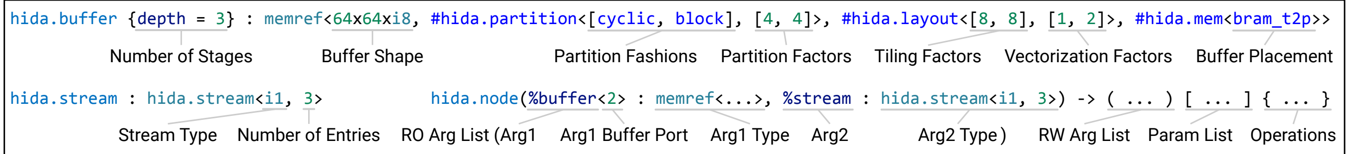

Figure 4. `buffer`, `stream`, and node operation syntax in *Structural* dataflow. *RO* and *RW* denote read-only and read-write.

C++ code, which can then be mapped to RTL designs with downstream HLS tools [34–36]. HIDA proposes two new techniques to handle the *representation* and *optimization* of dataflow compilation, which are the key enablers to tackle the challenges discussed in Section 2:

- *Hierarchical Dataflow IR (HIDA-IR).* As shown in Figure 3, HIDA consists of *Functional* and *Structural* dataflow IR carved for different purposes. Table 3 summarizes the key operations of HIDA-IR. Details can be found in Section 5.
- *Hierarchical Dataflow Optimizer (HIDA-OPT).* HIDA decouples the HLS optimization problems of *Functional* and *Structural* dataflow to handle HLS designs at scale. Details can be found in Section 6.

5 HIDA-IR

Currently, HLS tools [70, 79, 80] employ call graphs to represent HLS structures using sequential IRs. However, due to the lack of expressiveness for the parallel characteristics and micro-architecture of dataflow, these IRs have limited capability when used in scalable dataflow optimization flows. To address this problem, we propose a holistic HIDA-IR with two levels of representation, which we refer to as *Functional* and *Structural* dataflow. The *Functional* dataflow is designed to capture the high-level characteristics and hierarchy of HLS designs, driving the algorithmic optimizations and task fusion. In contrast, the *Structural* dataflow is a low-level abstraction that captures the micro-architectural details and is optimized to handle the scheduling and parallelization.

5.1 Functional Dataflow

Hierarchical Structure. In the *Functional* dataflow, we introduce a `dispatch` operation that contains the computation graph to be dispatched. Within the `dispatch` operation, all graph nodes are partitioned into multiple `task` operations to represent the dispatch strategy. As accelerators often have multiple levels of dataflow to achieve higher parallelism, HIDA-IR supports a hierarchical structure by allowing the recursive nesting of `task` and `dispatch` operations. Figure 3 visualizes a hierarchical *Functional* dataflow, where *Task6* contains three sub-tasks that can be executed in a dataflow manner to hide the latency of loads and stores, allowing us to utilize the computational capability to a greater extent.

Transparent from Above. At the *Functional* level, tasks often need to be manipulated as different dispatch strategies can lead to different trade-offs. Based on this observation, the `dispatch` and `task` operations are designed to be transparent and share the global context, simplifying the process of the fusing and splitting of tasks. As a result, we can efficiently explore various dispatch strategies at the *Functional* level. Meanwhile, thanks to the transparency, buffers and tensors defined in the global context can be accessed by tasks at all hierarchies without indirection. Therefore, the *Functional* dataflow can enable effective algorithmic optimizations for both PyTorch and C++ programs, which model the computations at tensor and memory levels, respectively.

5.2 Structural Dataflow

Memory-Mapped and Stream Buffer. In order to precisely capture the on-chip and off-chip memory accessing behaviors, we introduce two types of buffers at the *Structural* dataflow level, the memory-mapped buffer and the stream buffer, which are represented with the `buffer` and `stream` operations, respectively. Figure 4 shows their syntax, where `%` denotes a single static assignment (SSA) value [23]. The embedded partition and data layout attributes of the `buffer` operation are designed to be converted to semi-affine maps [18], enabling polyhedral-based dependency analysis and transformation [5] in HIDA-OPT. Notably, to facilitate dataflow optimizations, `buffer` operations inherently carry ping-pong buffering semantics, allowing buffer instances to interleave the accesses from producers and consumers to improve the communication efficiency. Figure 3 visualizes the combined usage of the two types of buffers, where the red boxes and blue arrows represent the memory-mapped buffer and stream buffers. Dashed blue arrows denote single-bit stream buffers. In addition to the buffers, we introduce the `port` operation to represent memory-mapped or stream interfaces, capturing the interface characteristics, such as latency, that can have a considerable impact on the dataflow efficiency. For instance, in Figure 3, *Node0*, *Node1*, and *Node2* are scheduled to communicate through external memory, where the AXI interfaces are modeled with `port` operations.

Isolated from Above. In the *Structural* dataflow, we introduce `schedule` and `node` operation as the counterparts of `dispatch` and `task` operation. While *Structural* dataflow has a hierarchical structure similar to *Functional* dataflow, an important distinction between the two is that the `schedule` and `node` operations are isolated from the external context. Therefore, external values must be passed into `schedule` and `node` as arguments. In addition, the `node` operation carries explicit memory effect information for each argument to

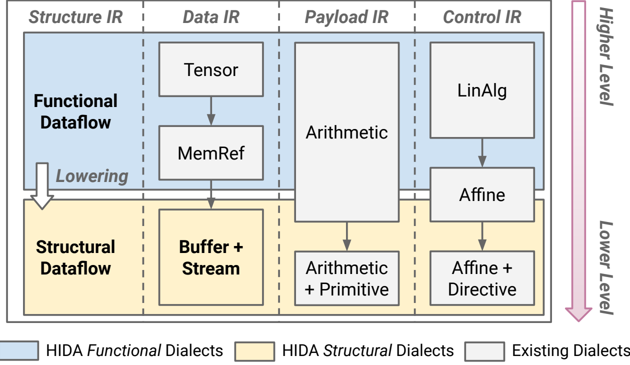

Figure 5. Integration with MLIR dialects.

avoid unnecessary inter-node effect analysis. Figure 4 shows the syntax of `node` operation, where `%buffer` and `%stream` are passed in as read-only arguments in this specific example. For simplicity, we omit the read-write arguments and constant parameters in Figure 4. The rationale behind this design decision is driven by how the *Structural* dataflow carries the architecture optimization. By isolating the context of `schedule` and `node`, the dataflow optimization problem can be cleanly partitioned into multiple local intra-node optimizations and global inter-node optimization, leading to a decoupled scalable solution. Details of the optimization process is elaborated in Section 6.

5.3 Integration with MLIR Dialects

HIDA reuses a set of MLIR built-in dialects to represent the common program components in PyTorch or C/C++. Figure 5 illustrates the integration of HIDA with the built-in MLIR dialects. The vertical axis of Figure 5 represents the various abstraction levels of the dialects. Horizontally, all dialects are categorized into four different types: *structure*, *data*, *payload*, and *control-flow*. At every abstraction level, the corresponding four types of dialects are combined with each other to represent the complete functionality of a program. The *Functional* dataflow is first combined with `tensor`, `arith`, and `linalg` dialects to represent the tensor-level programs compiled from PyTorch. Then, the `tensor` and `linalg` dialects are lowered to `memref` and `affine` dialects, respectively, while the lowered IRs are still combined with the *Functional* dataflow due to its adaptability with both tensor and memory semantics. Once the optimizations at the *Functional* dataflow level are completed, the *Functional* dataflow is lowered to the *Structural* dataflow, while the `memref` operations are lowered to the *Structural* buffer and stream operations. Note that the `affine` dialect is used in both the *Functional* and *Structural* dataflows for loop analyses and transformations. HIDA reuses the *Primitive* and *Directive* IRs from ScaleHLS [70] to represent the HLS-specific structures, such as the loop pipelining directive. By integrating with the existing dialects, HIDA can carry out loop and directive-level HLS optimizations in a hierarchical manner.

Algorithm 1 Functional dataflow construction

Require: m, top module of the initial computation graph
Ensure: Updated m, top module of the *Functional* dataflow
1: **for** n **in** postorder_walk(m, has_region()) **do**
2:     **if** is_dispatchable(get_region(n)) **then**
3:         $d \leftarrow$ wrap_ops(get_ops(n), new(`dispatch`))
4:         **for** op **in** get_ops(d) **do**
5:             wrap_ops({op}, new(`task`))

6 HIDA-OPT

Although previous works [26, 70] have enabled inter-task parallelization through dataflow, they could not conduct dataflow-oriented optimizations. Specifically, ScaleHLS [70] could automatically legalize a computation graph into a dataflow model and enable code generation but ignored the inter-task design space coupling [41], resulting in suboptimal dataflow designs. HPVM2FPGA [26] used an ML-driven algorithm [53] for DSE, but it only introduced a boolean parameter to enable/disable global dataflow, again, leading to the restricted search of the design space. In this section, we propose a HIDA-OPT solution consisting of five steps to tackle the dataflow optimization problem.

6.1 Functional Dataflow Construction

We leverage the transformations available in MLIR [20, 45] to generate a *hierarchical* computation graph at the level of linear algebra [19] or loop [18]. Algorithm 1 shows the pseudo-code of converting the initial computation graph to the *Functional* dataflow. From line 1 to 3, we wrap each *dispatchable* region with a `dispatch` operation in a bottom-up manner, where a region is defined as *dispatchable* if it is owned by an iterative operation, such as `loop` and `func`, while containing at least two iterative operations. For instance, a loop region containing two child loops is considered to be *dispatchable* as the two child loops can be dataflowed. Then, from line 4 to 5, each operation is wrapped with a separate `task` operation to construct a legal dataflow model.

6.2 Functional Dataflow Optimization

Once the initial *Functional* dataflow is constructed, we can optionally fuse dataflow tasks to balance the task workloads while reducing the communication cost. Algorithm 2 shows the task fusion process, where the inputs are the initial dataflow and a set of pre-defined profitable fusion patterns, such as element-wise operations fusion. From line 1 to 10, we recursively partition each `dispatch` operation in a top-down manner. Specifically, we first fuse adjacent tasks into new tasks through a pattern-driven worklist algorithm (lines 2 to 6). The pre-defined task fusion patterns are recursively applied to the dataflow until no pattern can be matched. Then,

Algorithm 2 Functional dataflow task fusion

Require: m, top module of the *Functional* dataflow
Require: *patterns*, set of profitable task fusion patterns
Ensure: Updated m, top module of the partitioned dataflow

1: **for** d **in** preorder_walk(m, is_instance(`dispatch`)) **do**
2:     *worklist* ← queue(get_tasks(d))
3:     **while not** is_empty(*worklist*) **do**
4:         t ← pop(*worklist*)
5:         **if** t' ← get_matched_task(t, *patterns*) **then**
6:             push(*worklist*, wrap_ops($\{t, t'\}$, new(`task`)))
7:     **repeat** t_0, t_1 ← get_least_critical_tasks(d, 2)
8:         wrap_ops($\{t_0, t_1\}$, new(`task`))
9:     **until not** is_fusion_profitable(t_0, t_1)
10:     simplify_dispatch_hierarchy(d)

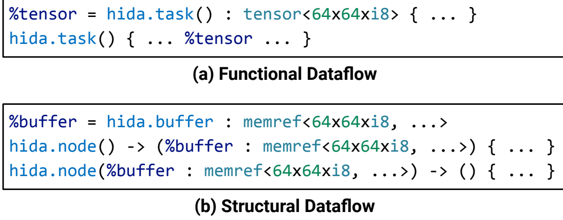

Figure 6. *Functional* to *Structural* dataflow lowering.

we continuously fuse the least critical two adjacent tasks to re-balance the dataflow until the fusion begins to generate a new critical task (lines 7 to 9). Finally, in line 10, we simplify the hierarchy by canonicalizing the `dispatch` and `task` operations. For instance, a task containing only one sub-task should be canonicalized to a single task. Notably, HIDA-IR's systematic dataflow representation allows the task fusion process to be expanded with different heuristics or algorithms under varying scenarios.

6.3 Structural Dataflow Construction

The *Functional* dataflow can be lowered to *Structural* dataflow for low-level optimizations and code generation. The dataflow lowering is composed of three procedures: (1) `buffer` operation generation; (2) `dispatch` to `schedule` operation mapping; (3) `task` to `node` operation mapping. Figure 6 shows a simplified example of the *Functional* to *Structural* dataflow lowering, where the two `task` operations in Figure 6(a) are mapped to `node` operations in Figure 6(b) correspondingly. The distinction between tensor and memory semantics draws a line between the *Functional* and *Structural* dataflow: tensors are immutable objects passed between producers and consumers directly; buffers are mutable objects that can be instantiated in hardware and modified multiple times. Therefore, in Figure 6(a), the `%tensor` object is produced by the first `task` operation and directly consumed inside of the second `task` operation. In contrast, we can observe that the original `%tensor` object is lowered to a `buffer` operation

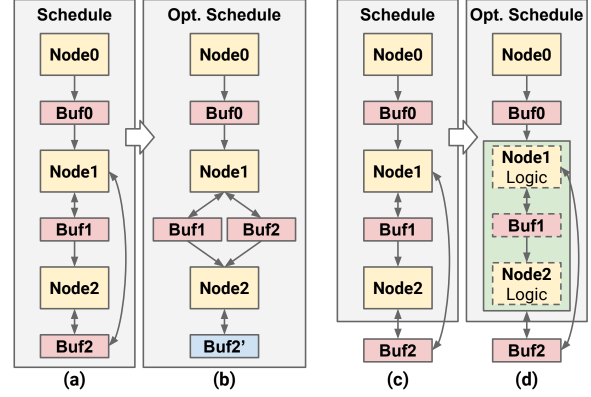

Figure 7. Eliminate multiple producers. Double arrows indicate both read and write on a buffer. For instance, in (a), *Node1* may first read from *Buf2* and then write results to it.

in Figure 6(b). Correspondingly, the first `node` operation becomes a user of the `%buffer` object with read-write effect, while the second `node` operation uses the `%buffer` object with read-only effect, given the syntax defined in Figure 4.

Automation. For procedure (1) above, each tensor result of each `task` operation is converted to a `buffer` operation with default partition fashions, tiling, vectorization factors, and placement annotations. For procedures (2) and (3) above, because the *Functional* operations are transparent while the *Structural* operations are isolated from above context, the live-ins and memory effects are analyzed during the mapping. As shown in Figure 4, the arguments of the `node` operations are explicitly grouped based on their memory effects. Notably, procedure (1) to (3) can generate a legal *Structural* dataflow, and the `node` and `buffer` operations will be optimized later in the dataflow balancing and parallelization.

6.4 Structural Dataflow Optimization

At the *Structural* dataflow level, we propose two optimizations that are crucial for dataflow efficiency but have not been thoroughly studied in existing tools: *(1) Multi-producer elimination*, which can eliminate multiple nodes writing to the same buffer and improve the dataflow parallelism; *(2) Data path balancing*, which can balance different data paths by inserting on-chip or external buffers, balancing the pipeline execution rate to achieve the best throughput.

6.4.1 Eliminate Multiple Producers. Dataflow architectures often contain buffers written to by multiple producers, leading to inefficient dataflow execution. For example, in Figure 7(a), correctly managing the memory access of *Buf2* is challenging as *Node1* and *Node2* simultaneously write to it. As a result, to preserve correctness, the dataflow structure must be executed sequentially.

Solution. HIDA-OPT resolves this problem considering two cases: *(1) Buffer duplication.* In the case of Figure 7(a), we eliminate the multiple producers by duplicating *Buf2* into the

Algorithm 3 Multiple producers elimination

Require: s, dataflow schedule
Ensure: Updated s, transformed dataflow schedule

```
 1: for b in get_internal_buffers(s) do
 2:     p_list ← topo_sort(get_producers(b))
 3:     for p in drop_front(p_list) do   ▷ exclude the 1st producer
 4:         b′ ← clone(b)
 5:         if read_effect(p, b) then
 6:             copy ← new(copy, b, b′)
 7:             insert_to_front(copy, get_region(p))
 8:         for u in get_users(b) do
 9:             if dominate(p, u) then       ▷ include p itself
10:                 replace_use_with(b, b′, u)
11: for b in get_external_buffers(s) do
12:     p_list ← get_producers(b)
13:     wrap_ops(p_list, new(node))          ▷ merge producers
```

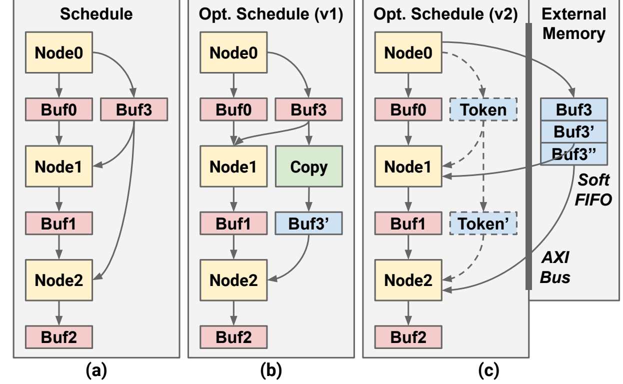

Figure 8. Balance data paths. Dash line block represents a 1-bit token buffer. Soft FIFO is allocated in external memory and interfaced with dataflow through AXI interconnect.

Buf2' as shown in Figure 7(b). As a result, *Node1* and *Node2* no longer write to *Buf2* simultaneously, allowing the structure to be scheduled in a pipelined manner. This duplication is possible due to the semantics of the *Structural* IR, where *Buf2* is allocated inside the context of its parent schedule, ensuring that no external side-effect operation can access *Buf2*. *(2) Node fusion.* For Figure 7(c), *Node1* and *Node2* write to *Buf2* allocated outside of its parent schedule. In this case, we cannot apply the same transformation as case (1) because there may exist an external node having write-effect on *Buf2*. Specifically, if we duplicate *Buf2* into *Buf2'*, only the original *Buf2* can be updated by the external write-effect nodes, leaving outdated data in *Buf2'*. Therefore, in the next iteration, *Buf2'* may hold incorrect data and eventually lead to incorrect functionality. Thus, to eliminate the multi-producer violation, we fuse *Node1* and *Node2* into a new node and sequentially execute them as shown in Figure 7(d).

Automation. Lines 1 to 10 of Algorithm 3 show the pseudo-code of case (1) above. For each internal buffer, we first collect all its *producer*s and sort them based on SSA dominance to maintain the predetermined memory access order in the subsequent transformations. Then, for each producer except the first one, we duplicate a new buffer b' for it (line 4) and create an explicit memory copy from the original buffer b to b' (lines 5 to 7) if the current producer p reads from b. Finally, we replace uses of b with b' if the user is dominated by the p (lines 8 to 10), leaving exactly one producer for the original buffer b. Lines 11 to 13 of Algorithm 3 shows the pseudo-code of case (2) that merges all producers of each external buffer into a single node to avoid data racing.

Complexity. One can observe that case (2) employs a more conservative transformation and enables the dataflow execution of the whole design by only enforcing the sequential execution of *Node1* and *Node2*. One may argue for a more comprehensive inter-node analysis of *Buf2* to determine whether the external write-effect nodes will intervene if we were to duplicate *Buf2* - this could work on small dataflow architectures, but it does not scale well. Many dataflow nodes at different hierarchies can access shared buffers; as a result, an inter-node analysis has a complexity of $O(mn^2)$, where m denotes the number of shared buffers and n is the number of nodes accessing the same buffer.

6.4.2 Balance Data Path.

A complicated dataflow structure often has multiple data paths; some paths may have more dataflow nodes to execute. If left unoptimized, the unbalanced paths can significantly degrade the overall performance of the final design. For instance, in Figure 8(a), the *Node0-Node1-Node2* path is longer than the *Node0-Node2* path. As a result, *Node0* must wait until the longer path completes before it can process the next data frame. This situation is very common in real-world applications, such as ResNet [30], which has shortcut paths in the residual blocks. Note that there are two levels of balancing in HIDA: one is the *data path* balancing we discuss in this section; the other is *node delay* balancing that will be handled separately during the dataflow parallelization.

Solution. HIDA-OPT resolves this issue using two methods: *(1) On-chip buffer duplication.* We can duplicate buffers on the shorter data paths to balance the execution speed. For instance, in Figure 8(b), *Buf3* is duplicated to *Buf3'*, followed by an automatic insertion of a copy node between *Buf3* and *Buf3'*. Through this approach, the *Node0-Copy-Node2* data path can execute in a pipelined manner at the same rate as another data path, such that *Node0* no longer needs to wait. *(2) Soft FIFO in external memory.* As shown in Figure 8(c), a soft FIFO is allocated in the external memory to substitute *Buf3*. The FIFO is *soft* because data is not really shifted in the FIFO. Instead, the memory access addresses of dataflow nodes are rotated to access the correct data. For instance, in Figure 8(c), *Node0* is writing to *Buf3*, while *Node1* and *Node2* are reading from *Buf3'* and *Buf3"*, respectively. Then in the next dataflow iteration, *Node0* will write to *Buf3"*, and *Node1* and *Node2* will read from *Buf3* and *Buf3'*, respectively.

Listing 1. A dataflow example in C++. We assume each nested loop is a dataflow node.

```
1  float A[32][16];
2  NODE0_I: for (int i=0; i<32; i++)
3    NODE0_K: for (int k=0; k<16; k++)
4      A[i][k] = ...; // Load array A.
5
6  float B[16][16];
7  NODE1_K: for (int k=0; k<16; k++)
8    NODE1_J: for (int j=0; j<16; j++)
9      B[k][j] = ...; // Load array B.
10
11 float C[16][16];
12 NODE2_I: for (int i=0; i<16; i++)
13   NODE2_J: for (int j=0; j<16; j++)
14     NODE2_K: for (int k=0; k<16; k++)
15       C[i][j] = A[i*2][k] * B[k][j];
```

Table 4. Node connections appearing in Listing 1. S and T denote the source and target nodes of the connection. $\emptyset$ denotes empty.

Source	Target	Buffer	Permutation Map		Scaling Map	
			S-to-T	T-to-S	S-to-T	T-to-S
Node0	Node2	A	[0, $\emptyset$, 1]	[0, 2]	[0.5, 1]	[2, $\emptyset$, 1]
Node1	Node2	B	[$\emptyset$, 1, 0]	[2, 1]	[1, 1]	[$\emptyset$, 1, 1]

Table 5. Node parallelization results of Listing 1 assuming a maximum parallel factor of 32.

Node	Intensity	Parallel Factor		Loop Unroll Factors			
		w/o IA	w/ IA	IA+CA	IA	CA	Naive
Node0	512	32	4	[4, 1]	[2, 2]	[8, 4]	[4, 8]
Node1	256	32	2	[1, 2]	[1, 2]	[4, 8]	[4, 8]
Node2	4,096	32	32	[4, 8, 1]	[4, 8, 1]	[4, 8, 1]	[4, 8, 1]

Elastic Node Execution. For method (2) above, after the soft FIFO is generated, the original *Buf3* is substituted with external memory interfaces, such as memory-mapped AXI interfaces. Therefore, the dependencies between dataflow nodes associated with *Buf3* are no longer explicit - they access external memories through assigned addresses instead of sharing *Buf3*. To maintain the correct execution order, HIDA can automatically construct a token flow between these dataflow nodes. For instance, in Figure 8(c), once *Node0* completes its computation, it will send a *Token*, and *Node1* and *Node2* will not start until they receive the *Token* and *Token'* respectively. This way, the token flows elastically maintain the execution order, and no static logic in the form of an FSM is needed to control the execution.

6.5 Structural Dataflow Parallelization

After dataflow optimization, HIDA will parallelize each node to improve the overall throughput and latency. However, automatic parallelization is often very challenging for several reasons: *(1) Memory data layout.* Degradation of dataflow performance may occur without proper alignment between the computation pattern and memory layouts. *(2) Connectedness of nodes.* We define two nodes as having a *connection* if they communicate through shared buffers. Due to reason (1), when two nodes are connected, the parallelism of each node should be aware of the shared memory layout. *(3) Computation intensity of nodes.* We define the number of operations contained by a node as its *intensity*. To maximize the overall throughput while minimizing resource utilization, the optimal parallel factors should be proportional to the intensity of dataflow nodes. Due to the coupling of local design spaces, the local optimality of the scheduling of each node can no longer automatically lead to the global optimal solution for dataflow architectures.

6.5.1 Intensity and Connection-Aware Approach.

The challenges discussed above are tightly coupled and need to be handled holistically. In HIDA, we propose an intensity-aware (IA) and connection-aware (CA) approach to determine the best parallelization strategies:

Step (1) Intensity and Connection Analysis. We first construct two maps to record the intensity and connections of each dataflow node. For each connection, we record the source and target node, associated buffer, permutation maps holding the loop level alignment, and scaling maps holding the stride alignment. For instance, in Listing 1, *Node0* and *Node2* are connected through array *A*. Because *Node0* writes to array *A* with the first and second loops, while *Node2* reads with the first and third loops, the *Node0-to-Node2* permutation map is [0, $\emptyset$, 1], where $\emptyset$ denotes empty. Meanwhile, the *Node0-to-Node2* scaling map is [0.5, 1] as *Node2* reads array *A* with a stride of 2. Table 4 shows the two connections in Listing 1. The permutation maps and scaling maps will be used to align the loop unroll factors of connected nodes in step (4) below.

Step (2) Node Sorting. We then sort all dataflow nodes into a worklist in descending order of the number of connections with the computation intensity as tie-breaker. This determines the order in which we parallelize each node. Intuitively, the parallelization strategy of a node with more connections will affect more nodes, while higher-intensity nodes, being more computationally complex, are more sensitive to optimization. Therefore, following the results of step (1), the order of nodes in Listing 1 in terms of criticality is *Node2*, *Node0*, and then *Node1*.

Step (3) Parallel Factor Generation. Guided by resource constraints, HIDA-OPT determines the maximum parallel factor that can be applied to a dataflow node. Then, the parallel factor of each node will be set proportionally to its

Algorithm 4 Node parallelization

Require: n, dataflow node
Ensure: Updated n, transformed dataflow node

```
 1: constraints_list ← new([][])                          ▷ list of constraints
 2: c_list ← get_connected_nodes(n)
 3: for c in c_list do
 4:     unroll_factors ← get_unroll_factors(c)
 5:     s_map ← get_scaling_map(c)                          ▷ from step (1)
 6:     p_map ← get_permutation_map(c)                      ▷ from step (1)
 7:     constraints ← permute(unroll_factors ⊙ s_map, p_map)
 8:     append(constraints_list, constraints)
 9: parallel_factor ← get_parallel_factor(n)                ▷ from step (3)
10: dse ← init_dse(n)                                       ▷ initialize DSE engine
11: repeat unroll_factors ← propose_unroll_factors(dse)
12:     is_valid ← true
13:     for constraints in constraints_list do
14:         for constr, uf in zip(constraints, unroll_factors) do
15:             if constr % uf != 0 and uf % constr != 0 then
16:                 is_valid ← false
17:     if product(unroll_factors) >= parallel_factor then
18:         is_valid ← false
19:     if is_valid then
20:         evaluate_and_evolve_dse(unroll_factors, dse)
21:     else
22:         evolve_dse(unroll_factors, dse)
23: until is_converged(dse) or is_early_terminated(dse)
24: apply_unrolling(n, get_final_unroll_factors(dse))
```

intensity. In Listing 1, assuming the maximum parallel factor is 32, the parallel factor of each node is listed in Table 5.

Step (4) Node Parallelization. Dataflow nodes are parallelized in the order determined by step (2). Algorithm 4 shows the pseudo-code of node parallelization. For each node n, we first query whether any nodes connected with it have been parallelized (line 2). If so, each connected node's unroll factors are multiplied with the scaling map and then permutated with the permutation map generated in step (1) (lines 3 to 8). The processed unroll factors are recorded in a *constraints_list* to constrain the intra-node DSE. Meanwhile, in line 9, we also get the parallel factor of node n generated in step (3) to constrain the overall parallelism of n.

With all the constraints generated, lines 10 to 23 of Algorithm 4 illustrate a simplified intra-node DSE for node n. In each iteration of exploration, the DSE engine will propose new unroll factors for evaluation. However, the proposed factors could fail to meet the constraints in two cases: (1) If any of the unroll factors are mutually indivisible with the corresponding constraint (lines 13 to 16); or (2) the overall parallelism exceeds the pre-calculated parallel factor (lines 17 to 18). Case (1) can cause unaligned inter-node memory access behavior, while case (2) can cause imbalanced dataflow execution, both leading to sub-optimal dataflow efficiency. If all the constraints are fulfilled, HIDA employs a quality of results (QoR) estimator from [70] to evaluate the

Table 6. Array partition results of Listing 1.

Array	Array Partition Factors				Bank Number			
	IA+CA	IA	CA	Naive	IA+CA	IA	CA	Naive
A	[8, 1]	[8, 2]	[8, 4]	[8, 8]	8	16	32	64
B	[1, 8]	[2, 8]	[4, 8]	[8, 8]	8	16	32	64
C	[4, 8]	[4, 8]	[4, 8]	[4, 8]	32	32	32	32

performance and resource utilization of the proposed factors (line 20). Then, the evaluation results or failure message are passed to the DSE and evolve the exploration. Finally, in line 23, we terminate the DSE if the results have converged or met the early termination criteria. The proposed algorithm can identify the Pareto frontier in the local design space and selects the best design point under the imposed constraints.

6.5.2 Discussion. We summarize the results of IA+CA, IA-only, CA-only, and naive parallelization in Table 5, where the naive solution applies the maximum parallel factor 32 to all dataflow nodes. One can observe our IA+CA approach achieves the best unroll factors, eventually leading to the least computation resource utilization. Meanwhile, IA and CA can also reduce memory resource utilization. Table 6 shows the array partition results. Array partitioning is an HLS technique that divides a large array into smaller subarrays to enable parallel access. One can observe that our IA+CA approach achieves the lowest number of banks for arrays appearing in Listing 1, resulting in the lowest memory utilization. For this small example, the margin can already reach 8× on arrays A and B compared to the naive solution. For large-scale dataflow applications, the parallelization solution can determine whether the overall solution is scalable; an ablation study is conducted in Section 7.3.

7 Evaluation

To evaluate HIDA, we use FPGAs as the target platform and perform two sets of experiments using C++ and PyTorch inputs and an ablation study on a ResNet-18 model. As depicted in Figure 3, AMD Vitis HLS 2022.1 [34] is used for generating RTL code. All reported performances and resource utilization are collected from the synthesis results of Vitis HLS.

7.1 C++ Kernels Evaluation

Experiment Settings. We evaluate HIDA with a set of C++ benchmarks from PolyBench [58]. The benchmarks cover multiple categories, including blas routines (gesummv, symm, and syr2k), linear algebra kernels (2mm, 3mm, atax, bicg, and mvt), data mining (correlation), and stencils (jacobi-2d and seidel-2d). The target platform is AMD-Xilinx ZU3EG FPGA. Table 7 shows the evaluation results. Although Vitis HLS can automatically apply optimizations such as loop `pipeline`, it cannot conduct complex dataflow analysis and optimizations.

Table 7. Evaluation results for C++ kernels. *ScaleHLS* designs are automatically generated by [70]. *SOFF* results are ported from their paper [37], which compared with SDAccel (previous name of Vitis). *Vitis* designs are solely optimized by Vitis HLS.

Kernel	HIDA Compile Time (s)	LUT Number	FF Number	DSP Number	Throughput (Samples/s)*			
					HIDA	ScaleHLS [70]	SOFF [37]	Vitis [34]
2mm	0.65	38.8k	27.4k	269	239.22	122.39 (1.95×)	30.67 (7.80×)	1.23 (194.88×)
3mm	0.79	38.7k	27.8k	243	175.43	92.33 (1.90×)	-	1.04 (167.99×)
atax	2.06	44.6k	34.6k	260	1,021.39	932.26 (1.10×)	2,173.17 (0.47×)	103.18 (9.90×)
bicg	0.72	16.0k	15.1k	61	2,869.69	2,869.61 (1.00×)	2,295.75 (1.25×)	104.19 (27.54×)
correlation	0.91	14.5k	12.3k	66	67.33	59.77 (1.13×)	3.96 (16.99×)	1.32 (50.97×)
gesummv	0.60	34.2k	22.8k	232	31,685.68	31,685.68 (1.00×)	3,466.70 (9.14×)	266.65 (118.83×)
jacobi-2d	1.98	91.4k	56.6k	352	257.27	128.63 (2.00×)	-	2.71 (94.95×)
mvt	0.42	23.8k	16.5k	162	9,979.04	4,989.02 (2.00×)	870.01 (11.47×)	62.13 (160.62×)
seidel-2d	3.59	5.5k	2.5k	4	0.14	0.14 (1.00×)	-	0.11 (1.28×)
symm	1.05	14.9k	9.5k	74	2.62	2.62 (1.00×)	-	2.02 (1.29×)
syr2k	0.69	14.3k	12.8k	78	27.68	27.67 (1.00×)	-	1.44 (19.23×)
Geo. Mean	**0.99**					**1.29×**	**4.49×**	**31.08×**

* Numbers in () show throughput improvements of HIDA over others.

As a result, HIDA achieves 31.08× higher throughput on average over Vitis HLS.

Comparison with Previous Works. Compared with the state-of-the-art (SOTA) HLS optimization framework ScaleHLS [70], and another HLS framework SOFF [37], HIDA achieved 1.29× and 4.49× higher throughput, respectively. We observed that for single-loop kernels (bicg, gesummv, seidel-2d, symm, and syr2k), the performance of HIDA was on par with ScaleHLS due to these kernels not presenting any dataflow optimization opportunities. For the multi-loop kernels, HIDA outperforms ScaleHLS due to dataflow optimizations. When only considering multi-loop kernels, HIDA achieves 1.57× higher throughput than ScaleHLS. We concluded that the dataflow scheduling and parallelization problems are pervasive based on the evaluation results. Thus, HIDA-OPT can better optimize these kernels, ultimately leading to an increased performance.

7.2 PyTorch Models Evaluation

Experiment Settings. We evaluate HIDA with a set of deep neural network (DNN) benchmarks written in PyTorch to understand its performance on large-scale dataflow applications. The benchmarks cover multiple categories of DNN models, including image classification (ResNet-18 [30], MobileNet [31], ZFNet [74], and VGG-16 [63]), object detection (YOLO [60]), and fully-connected networks (MLP). The optimization for these models exhibit significant variations under dataflow setting, owing to the distinct layer types and interconnections. The targeted platform is one super logic region (SLR) of an AMD-Xilinx VU9P FPGA. Table 8 shows the evaluation results. Even for these complicated DNN models, HIDA only takes 108.7 seconds on average to compile them into dataflow implementations.

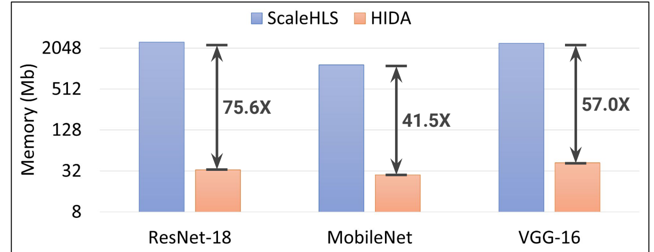

Figure 9. Memory utilization compared with ScaleHLS [70].

Comparison with Previous Works. Again, we compare HIDA with ScaleHLS [70], where we observe an 8.54× higher throughput. The throughput gains are much more significant than the C++ kernels due to large DNN models exposing more opportunities for HIDA to optimize the dataflow architecture. For ZFNet and YOLO, ScaleHLS cannot produce results due to the DNNs having irregular convolution sizes and high-resolution inputs, respectively, demonstrating the superior flexibility and scalability of HIDA. For the four benchmarks supported by ScaleHLS, we use DSP efficiency to compare the two frameworks, calculated as:

$$Efficiency_{DSP} = \frac{Throughput \times OPs}{Number_{DSP} \times Frequency}, \quad (1)$$

where *OPs* denotes the number of multiply-accumulate (MAC) operations per sample of the DNN, *Throughput* is samples per second, and *Frequency* denotes the clock frequency constant at 200MHz for both ScaleHLS and HIDA. DSP efficiency is a common metric for comparing the efficiency of DNN accelerators across different platforms or frameworks. A 100% of DSP efficiency indicates all instantiated DSPs in the accelerator continuously operating without stalling. HIDA

Table 8. Evaluation results for PyTorch models. *DNNBuilder* results are directly from their paper [77]. To make fair comparison, we constrained the FPGA resources to the same with DNNBuilder. *ScaleHLS* designs are automatically generated by [70].

Model	HIDA Compile Time (s)	LUT Number	DSP Number	Throughput (Samples/s)*			DSP Efficiency*		
				HIDA	DNNBuilder [77]	ScaleHLS [70]	HIDA	DNNBuilder [77]	ScaleHLS [70]
ResNet-18	83.1	142.1k	667	45.4	-	3.3 (13.88×)	73.8%	-	5.2% (14.24×)
MobileNet	110.8	132.9k	518	137.4	-	15.4 (8.90×)	75.5%	-	9.6% (7.88×)
ZFNet	116.2	103.8k	639	90.4	112.2 (0.81×)	-	82.8%	79.7% (1.04×)	-
VGG-16	199.9	266.2k	1118	48.3	27.7 (1.74×)	6.9 (6.99×)	102.1%	96.2% (1.06×)	18.6% (5.49×)
YOLO	188.2	202.8k	904	33.7	22.1 (1.52×)	-	94.3%	86.0% (1.10×)	-
MLP	40.9	21.0k	164	938.9	-	152.6 (6.15×)	90.0%	-	17.6% (5.10×)
Geo. Mean	**108.7**				**1.29×**	**8.54×**		**1.07×**	**7.49×**

* Numbers in () show throughput/DSP efficiency improvements of HIDA over others.

achieves 7.49× higher DSP efficiency than ScaleHLS on average and 14.24× for ResNet-18. We attribute the much higher efficiency for ResNet-18 to HIDA's ability to optimize shortcut data paths. For VGG-16, we observe an over 100% DSP efficiency, attributing the excess percentage to the back-end RTL generator where MAC operations can be instantiated with LUTs when resources are abundant.

Apart from the throughput and efficiency improvements, we also observe substantial on-chip memory reduction by HIDA compared to ScaleHLS. As Figure 9 shows, HIDA can reduce memory utilization by 41.5× to 75.6× due to several factors: (1) HIDA can leverage loop tiling and local buffer creation to only cache small tiles of intermediate results while enabling the dataflow execution. In comparison, ScaleHLS must keep all intermediate results on-chip due to the lack of external memory access support. (2) The IA+CA parallelization can drastically reduce the buffer sizes. In summary, HIDA can utilize computation and memory resources more efficiently and achieve substantial throughput improvements on DNN models compared to SOTA frameworks.

Comparison with Dedicated DNN Accelerator. In addition to the comparison with HLS optimization frameworks, we further compare HIDA with a dedicated DNN acceleration framework, DNNBuilder [77]. DNNBuilder has RTL-based and human-designed DNN IPs and can enable the dataflow execution of all the instantiated IPs to achieve SOTA throughput and efficiency on FPGAs. As shown in Table 8, HIDA achieves 1.29× and 1.07× higher throughput and DSP efficiency compared to DNNBuilder, which already has an extremely high DSP efficiency. Note that DNNBuilder doesn't support ResNet-18 and MobileNet due to its lack of support for shortcut paths and depthwise convolutions. Through this comparison, we demonstrate the productivity and performance of HIDA outperforming a dedicated DNN acceleration framework. Additionally, we demonstrate the flexibility of HIDA, which can automatically adapt to a wide range of computational patterns.

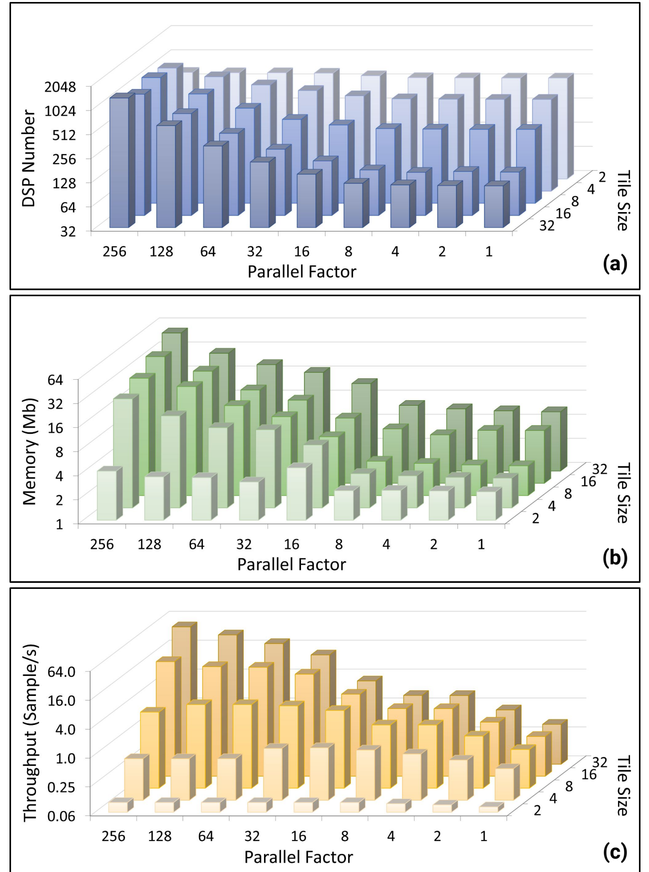

Figure 10. Parallel factor and tile size ablation on ResNet-18.

7.3 Ablation Study on ResNet-18

Parallel Factor and Tile Size Ablation. To understand the scalability of HIDA, we perform an ablation study on ResNet-18 by sweeping the maximum parallel factor from 1 to 256 and tile size from 2 to 32. Three metrics (DSP utilization, memory utilization, and throughput) are measured for each combination of parallel factor and tile size. Figure 10

shows the ablation study results. Overall, as the parallel factor increases, all three metrics increase as expected, showing promising *scalability* of HIDA. Meanwhile, the overall trend predictably showed more memory instances being utilized as we increased the tile size. However, we also observed some interesting findings summarized as follows:

- *Small tile can drastically increase DSP utilization.* Counterintuitively, the number of DSPs increases when the tile size decreases, highlighted by the data point with a parallel factor of 1 and a tile size of 2, instantiating 518 DSPs. Analysis shows that due to small tiles needing fine-grained control of memory accesses, a large number of DSPs are used for address calculations instead of actual computations.
- *Smaller tile may not reduce memory utilization.* For small parallel factors (1 to 8) and small tile sizes (2 to 8), the memory utilization does not change noticeably within the range. Analysis shows that as the minimum memory instance is BRAM (block RAM) for FPGAs, increasing parallel factor or tile size may not demand more BRAM instances as long as the current BRAMs are large enough to hold the data tiles and can provide enough bandwidth.
- *Throughput and tile size may positively correlate.* Ideally, different tile sizes should not affect the throughput. However, experimental results disagree: throughput increases with tile size being larger, especially for large parallel factors (32 to 256). Analysis shows two main reasons for the observed behavior: (1) Small tile sizes cannot provide sufficient bandwidth, resulting in a degradation of the level of parallelism.(2) Small tile sizes cannot provide sufficient burst length for consecutive external memory access, leading to poor external memory efficiency.

Node Parallelization Ablation. Section 6 introduces IA and CA approaches to drive the dataflow parallelization. To better understand the performance of the two methods on large-scale applications, we conduct an ablation study on ResNet-18. In this study, we set four groups of experiments (IA+CA, IA-only, CA-only, and Naive) for comparison, and for each group, we sweep the maximum parallel factor from 1 to 256. As expected, the four groups show a similar trend in DSP numbers - they increase proportionally with the parallel factor. However, the IA+CA group shows a drastically different trend in memory and throughput compared with the other groups - only IA+CA scales well when we increase the parallel factor. For instance, for the data samples with a parallel factor of 64, compared with the other groups, IA+CA utilizes 3.7× less DSP and 1.2× less memory, yet achieves 44.3× better throughput. After studying the generated accelerators, we found for all the other groups except IC+CA, the compiler generates overly-complicated control logics due to the mismatch between node unroll factors and memory layouts, ultimately falling back to flawed designs. Meanwhile, even when all groups can be scaled well,

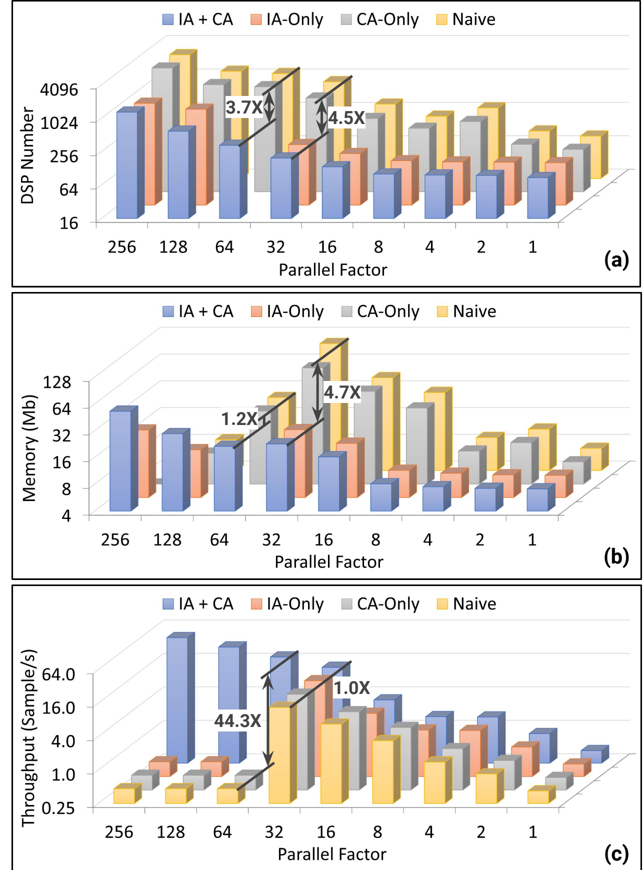

Figure 11. Intensity-aware (IA) and connection-aware (CA) dataflow parallelization ablation on ResNet-18.

IA+CA shows substantially better resource utilization efficiency than other groups. For instance, for the data samples with a parallel factor of 32, compared with the other groups, IA+CA performs on par in throughput but utilizes 4.5× less DSP and 4.7× less memory. Through these comparisons, we demonstrate the superior scalability and efficiency of our proposed IA+CA dataflow parallelization approach.

8 Related Works

8.1 MLIR Infrastructure

MLIR [20, 45] is a compiler infrastructure with multiple levels of representation, allowing users to tailor domain-specific compilers. MLIR also provides commonly used IRs and optimizations for different domains, including tensor [21], linear algebra [19], and loop [18]. As MLIR is an infrastructure for compilers, it does not natively support hardware-oriented compilation; however, many related compilers have been developed on top of MLIR. IREE [17] provides compilation and runtime support for hardware accelerators like GPU. CIRCT [16] is a toolchain for circuit design and optimization, supporting HLS, HDL-generation languages [3], and HDL.

8.2 HLS Languages

Different domain-specific languages (DSL) have been proposed to improve the productivity and performance of commercial HLS tools. TAPA [10, 29] introduced specialized interfaces to enable dataflow and efficient external memory access, but didn't provide solution for dataflow DSE. As a result, TAPA requires users to make many design decisions, such as task parallelization and buffer implementation. Spatial [42] abstracted interfaces for describing the control, memory, and I/O structures, enabling auto-tuning-based DSE. Dahlia [54] proposed an affine-based approach to improve the predictability of HLS designs. Aetherling [25] proposed a strong type system to tackle the fine-grained scheduling and DSE of hardware design. Another cluster of works extends existing language syntax, such as Python, to further raise the abstraction level of HLS designs, including DaCe [4] and PyLog [32]. These languages are orthogonal to HIDA, which attempts to address the dataflow optimization problem with compiler techniques. As future works, they can be integrated as front-ends to further improve the productivity of HIDA.

HeteroCL. HeteroCL [43] and HeteroFlow [68] decoupled the algorithmic description and optimization of HLS designs by providing a versatile set of computation, data type, and data movement customization primitives. HeteroCL incorporated third-party frameworks for automated HLS optimization, including PolySA [13] for systolic arrays and SODA [9] for stencil applications. However, these automated optimizations are intra-task and domain-specific. For inter-task optimizations and computation patterns that are not covered by third-party frameworks, designers must make every design decision empirically. Meanwhile, HeteroCL cannot directly take PyTorch as an input - designers must rewrite the PyTorch model in HeteroCL DSL.

8.3 HLS Compilers

Existing works have explored different approaches for the fine-grained scheduling problem in HLS, including static scheduling (Spatial [42], SOFF [37], and Calyx [55]), dynamic scheduling (Dynamatic [39] and TAPAS [49]), and static-dynamic hybrid scheduling (DASS [8] and Hector [69]). Fine-grained scheduling deals with operator-level parallelism, such as multipliers, instead of task-level and thus is very different from the problems we addressed in HIDA. For instance, fine-grained scheduling does not consider the intra-task parallelization. We have seen a large amount of HLS optimization tools, including LLVM-based (Merlin [11], COMBA [79], and HPVM2FPGA [26]) and MLIR-based (ScaleHLS [70], POLSCA [80], and SODA-Opt [1]). However, as discussed in Section 1, these tools either did not consider dataflow during the compilation or are limited in dataflow-oriented optimizations. Another alternative solution is leveraging existing multi-core CPU or CGRA compilers, such as PolyMage [52], Tapir [62], Unified Buffer [48], and Revet [61], for the purpose of HLS. However, the dataflow optimizations are not compatible with these compilers due to the fundamentally different programming model and memory hierarchy of HLS-based dataflow accelerators.

CIRCT. Handshake dialect [15] is a CIRCT dialect implementing the elastic circuit components and dynamic scheduling algorithms proposed in Dynamatic [39]. Coarse-grained tasks are abstracted as *function*s in the handshake dialect, where scalar intermediate results are passed between different functions through handshaking FIFOs. However, for tensor intermediate results, the handshake dialect adopts a load-store queue-based [38] shared memory model instead of the dataflow model for inter-function communication, drawing a line between the two. Elastic silicon interconnect (ESI) dialect [14] is another CIRCT dialect aiming to provide type-safe and latency-insensitive interface abstraction for FPGA/ASIC design. In contrast, HIDA aims to tackle the DSE problem of fine-grained and coarse-grained scheduling, which is orthogonal to the mission of the ESI dialect.

8.4 DNN Compilers

DNN layer fusion algorithms have been studied in recent years [2, 82] to reduce the layer-wise communication cost of DNN training or inference. In contrast, HIDA's task fusion is a general-purpose algorithm for dataflow applications in different domains. The patterns proposed by [2, 82] can be implemented and plugged into HIDA. The scheduling problem of cache-based or scratchpad-based DNN accelerators has also been thoroughly studied, such as in Eyeriss [7], Diannao [6], and Timeloop [56], either from an architecture or compiler perspective. However, the scheduling of dataflow-based DNN accelerators is still under intensive study [61, 81, 85]. The intention to balance the dataflow pipeline while enabling buffer sharing or streaming between layer instances (either a single DNN layer, a fused layer, or a decomposed layer) significantly complicates the scheduling problem. The reason is, as we mentioned in Section 6.5, the local optimality of each layer cannot lead to the global optimal dataflow architecture. Although existing works, such as FINN [65] and DNNBuilder [77], explored the problem to some extent, their framework can only target a subset of DNNs; for instance, DNNBuilder only supports CNNs.

9 Conclusion

In this paper, we propose HIDA, an HLS framework that can systematically transform an algorithmic description into an efficient dataflow implementation. We propose a two-level dataflow representation, HIDA-IR, and a hierarchical dataflow optimizer, HIDA-OPT, significantly improving the productivity, performance, and scalability of HLS dataflow accelerators. To demonstrate the performance of HIDA, we evaluate a set of DNN models and C++ kernels, where HIDA outperforms the existing SOTA hand-tuned RTL-based DNN

accelerator and compilation-based HLS frameworks. We hope that the HIDA framework will serve as a new open infrastructure for future dataflow architectural research, allowing researchers to explore the vast design space effectively.

Acknowledgments

We thank all anonymous reviewers and Adrian Sampson of Cornell University for their valuable feedback and suggestions. This work is supported in part by AMD Center of Excellence at UIUC, AMD Heterogeneous Adaptive Compute Cluster (HACC) initiative, NSF 2117997 grant through the A3D3 institute, and Semiconductor Research Corporation (SRC) 2023-CT-3175 grant.

References

[1] Nicolas Bohm Agostini, Serena Curzel, Vinay Amatya, Cheng Tan, Marco Minutoli, Vito Giovanni Castellana, Joseph Manzano, David Kaeli, and Antonino Tumeo. 2022. An MLIR-based Compiler Flow for System-Level Design and Hardware Acceleration. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*. 1–9.

[2] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. 2016. Fused-layer CNN accelerators. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–12.

[3] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*. 1216–1225.

[4] Tal Ben-Nun, Johannes de Fine Licht, Alexandros N Ziogas, Timo Schneider, and Torsten Hoefler. 2019. Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[5] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. 2010. The polyhedral model is more widely applicable than you think. In *Compiler Construction: 19th International Conference, CC 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings 19*. Springer, 283–303.

[6] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM SIGARCH Computer Architecture News* 42, 1 (2014), 269–284.

[7] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. 2016. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE journal of solid-state circuits* 52, 1 (2016), 127–138.

[8] Jianyi Cheng, Lana Josipovic, George A Constantinides, Paolo Ienne, and John Wickerson. 2020. Combining dynamic & static scheduling in high-level synthesis. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 288–298.

[9] Yuze Chi, Jason Cong, Peng Wei, and Peipei Zhou. 2018. SODA: Stencil with optimized dataflow architecture. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

[10] 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.

[11] Jason Cong, Muhuan Huang, Peichen Pan, Yuxin Wang, and Peng Zhang. 2016. Source-to-source optimization for HLS. *FPGAs for Software Programmers* (2016), 137–163.

[12] 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.

[13] Jason Cong and Jie Wang. 2018. PolySA: Polyhedral-based systolic array auto-compilation. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

[14] CIRCT Contributors. 2023. *CIRCT ESI Dialect*. https://circt.llvm.org/docs/Dialects/ESI/

[15] CIRCT Contributors. 2023. *CIRCT Handshake Dialect*. https://circt.llvm.org/docs/Dialects/Handshake/

[16] CIRCT Contributors. 2023. *The CIRCT Project*. https://github.com/llvm/circt

[17] IREE Contributors. 2023. *The IREE Project*. https://github.com/openxla/iree

[18] MLIR Contributors. 2023. *MLIR Affine Dialect*. https://mlir.llvm.org/docs/Dialects/Affine/

[19] MLIR Contributors. 2023. *MLIR LinAlg Dialect*. https://mlir.llvm.org/docs/Dialects/Linalg/

[20] MLIR Contributors. 2023. *MLIR Project*. https://github.com/llvm/llvm-project/tree/main/mlir

[21] MLIR Contributors. 2023. *MLIR Tensor Dialect*. https://mlir.llvm.org/docs/Dialects/TensorOps/

[22] Torch-MLIR Contributors. 2023. *Torch-MLIR Project*. https://github.com/llvm/torch-mlir/

[23] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 4 (1991), 451–490.

[24] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. 2015. ShiDianNao: Shifting vision processing closer to the sensor. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 92–104.

[25] 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*. 408–422.

[26] Adel Ejjeh, Leon Medvinsky, Aaron Councilman, Hemang Nehra, Suraj Sharma, Vikram Adve, Luigi Nardi, Eriko Nurvitadhi, and Rob A Rutenbar. 2022. HPVM2FPGA: Enabling true hardware-agnostic FPGA programming. In *2022 IEEE 33rd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 1–10.

[27] Farah Fahim, Benjamin Hawks, Christian Herwig, James Hirschauer, Sergo Jindariani, Nhan Tran, Luca P Carloni, Giuseppe Di Guglielmo, Philip Harris, Jeffrey Krupa, et al. 2021. hls4ml: An open-source codesign workflow to empower scientific low-power machine learning devices. *arXiv preprint arXiv:2103.05579* (2021).

[28] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, et al. 2021. Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 769–774.

[29] Licheng Guo, Yuze Chi, Jason Lau, Linghao Song, Xingyu Tian, Moazin Khatti, Weikang Qiao, Jie Wang, Ecenur Ustun, Zhenman Fang, et al. 2022. TAPA: A Scalable Task-Parallel Dataflow Programming Framework for Modern FPGAs with Co-Optimization of HLS and Physical Design. *arXiv preprint arXiv:2209.02663* (2022).

[30] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE*

conference on computer vision and pattern recognition. 770–778.

[31] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[32] 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.

[33] Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. 2022. Exocompilation for productive programming of hardware accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 703–718.

[34] Advanced Micro Devices Inc. 2022. *Vitis High-Level Synthesis User Guide UG1399 (v2022.2)*.

[35] Intel Inc. 2022. *Intel High Level Synthesis Compiler Pro Edition Reference Manual (22.4)*.

[36] Microchip Technology Inc. 2021. *LegUp 2021.1 Documentation*.

[37] Gangwon Jo, Heehoon Kim, Jeesoo Lee, and Jaejin Lee. 2020. SOFF: An OpenCL high-level synthesis framework for FPGAs. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 295–308.

[38] Lana Josipovic, Philip Brisk, and Paolo Ienne. 2017. An out-of-order load-store queue for spatial computing. *ACM Transactions on Embedded Computing Systems (TECS)* 16, 5s (2017), 1–19.

[39] Lana Josipović, Radhika Ghosal, and Paolo Ienne. 2018. Dynamically scheduled high-level synthesis. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 127–136.

[40] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*. 1–12.

[41] HyeGang Jun, Hanchen Ye, Hyunmin Jeong, and Deming Chen. 2023. AutoScaleDSE: A Scalable Design Space Exploration Engine for High-Level Synthesis. *ACM Trans. Reconfigurable Technol. Syst.* (2023).

[42] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, et al. 2018. Spatial: A language and compiler for application accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 296–311.

[43] 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*. 242–251.

[44] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 75–86.

[45] 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.

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

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

[48] Qiaoyi Liu, Jeff Setter, Dillon Huff, Maxwell Strange, Kathleen Feng, Mark Horowitz, Priyanka Raina, and Fredrik Kjolstad. 2023. Unified Buffer: Compiling Image Processing and Machine Learning Applications to Push-Memory Accelerators. *ACM Transactions on Architecture and Code Optimization* 20, 2 (2023), 1–26.

[49] Steven Margerm, Amirali Sharifian, Apala Guha, Arrvindh Shriraman, and Gilles Pokam. 2018. TAPAS: Generating parallel accelerators from parallel programs. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 245–257.

[50] Thierry Moreau, Tianqi Chen, Ziheng Jiang, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. VTA: an open hardware-software stack for deep learning. *arXiv preprint arXiv:1807.04188* (2018).

[51] William S Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. 2021. Polygeist: Raising C to polyhedral MLIR. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 45–59.

[52] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. Polymage: Automatic optimization for image processing pipelines. *ACM SIGARCH Computer Architecture News* 43, 1 (2015), 429–443.

[53] Luigi Nardi, David Koeplinger, and Kunle Olukotun. 2019. Practical design space exploration. In *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 347–358.

[54] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. 2020. Predictable accelerator design with time-sensitive affine types. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 393–407.

[55] Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. 2021. A compiler infrastructure for accelerator generators. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 804–817.

[56] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W Keckler, and Joel Emer. 2019. Timeloop: A systematic approach to dnn accelerator evaluation. In *2019 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE, 304–315.

[57] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[58] Louis-Noël Pouchet et al. 2012. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/pouchet/software/polybench* 437 (2012), 1–1.

[59] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2017. Plasticine: A reconfigurable architecture for parallel paterns. *ACM SIGARCH Computer Architecture News* 45, 2 (2017), 389–402.

[60] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 779–788.

[61] Alexander Rucker, Shiv Sundram, Coleman Smith, Matthew Vilim, Raghu Prabhakar, Fredrik Kjolstad, and Kunle Olukotun. 2023. Revet: A Language and Compiler for Dataflow Threads. *arXiv preprint arXiv:2302.06124* (2023).

[62] Tao B Schardl, William S Moses, and Charles E Leiserson. 2017. Tapir: Embedding fork-join parallelism into LLVM's intermediate representation. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 249–265.

[63] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[64] 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.

[65] 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.

[66] Yuxin Wang, Peng Li, Peng Zhang, Chen Zhang, and Jason Cong. 2013. Memory partitioning for multidimensional arrays in high-level synthesis. In *Proceedings of the 50th Annual Design Automation Conference*. 1–8.

[67] Xuechao Wei, Yun Liang, Xiuhong Li, Cody Hao Yu, Peng Zhang, and Jason Cong. 2018. TGPA: Tile-grained pipeline architecture for low latency CNN inference. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. ACM, 1–8.

[68] Shaojie Xiang, Yi-Hsiang Lai, Yuan Zhou, Hongzheng Chen, Niansong Zhang, Debjit Pal, and Zhiru Zhang. 2022. Heteroflow: An accelerator programming model with decoupled data placement for software-defined fpgas. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 78–88.

[69] Ruifan Xu, Youwei Xiao, Jin Luo, and Yun Liang. 2022. HECTOR: A Multi-Level Intermediate Representation for Hardware Synthesis Methodologies. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*. 1–9.

[70] 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.

[71] Hanchen Ye, HyeGang Jun, Hyunmin Jeong, Stephen Neuendorffer, and Deming Chen. 2022. ScaleHLS: a scalable high-level synthesis framework with multi-level transformations and optimizations. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 1355–1358.

[72] Hanchen Ye, Xiaofan Zhang, Zhize Huang, Gengsheng Chen, and Deming Chen. 2020. HybridDNN: A framework for high-performance hybrid DNN accelerator design and implementation. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[73] Mang Yu, Sitao Huang, and Deming Chen. 2021. Chimera: A hybrid machine learning-driven multi-objective design space exploration tool for fpga high-level synthesis. In *Intelligent Data Engineering and Automated Learning–IDEAL 2021: 22nd International Conference, IDEAL 2021, Manchester, UK, November 25–27, 2021, Proceedings 22*. Springer, 524–536.

[74] Matthew D Zeiler and Rob Fergus. 2014. Visualizing and understanding convolutional networks. In *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part I 13*. Springer, 818–833.

[75] Bingyi Zhang, Rajgopal Kannan, and Viktor Prasanna. 2021. Boostgcn: A framework for optimizing gcn inference on fpga. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 29–39.

[76] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays*. 161–170.

[77] Xiaofan Zhang, Junsong Wang, Chao Zhu, Yonghua Lin, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. 2018. DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. ACM, 1–8.

[78] 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.

[79] Jieru Zhao, Liang Feng, Sharad Sinha, Wei Zhang, Yun Liang, and Bingsheng He. 2017. COMBA: A comprehensive model-based analysis framework for high level synthesis of real applications. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 430–437.

[80] Ruizhe Zhao, Jianyi Cheng, Wayne Luk, and George A Constantinides. 2022. POLSCA: Polyhedral High-Level Synthesis with Compiler Transformations. In *32nd International Conference on Field Programmable Logic and Applications (FPL'22)*.

[81] Tian Zhao, Alexander Rucker, and Kunle Olukotun. 2023. Sigma: Compiling Einstein Summations to Locality-Aware Dataflow. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 718–732.

[82] Zhuoran Zhao, Kamyar Mirzazad Barijough, and Andreas Gerstlauer. 2018. Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 11 (2018), 2348–2359.

[83] Guanwen Zhong, Alok Prakash, Yun Liang, Tulika Mitra, and Smail Niar. 2016. Lin-analyzer: A high-level performance analysis tool for FPGA-based accelerators. In *Proceedings of the 53rd Annual Design Automation Conference*. 1–6.

[84] Peipei Zhou, Jiayi Sheng, Cody Hao Yu, Peng Wei, Jie Wang, Di Wu, and Jason Cong. 2021. Mocha: Multinode cost optimization in heterogeneous clouds with accelerators. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 273–279.

[85] Jinming Zhuang, Jason Lau, Hanchen Ye, Zhuoping Yang, Yubo Du, Jack Lo, Kristof Denolf, Stephen Neuendorffer, Alex Jones, Jingtong Hu, et al. 2023. CHARM: Composing Heterogeneous AcceleRators for Matrix Multiply on Versal ACAP Architecture. *arXiv preprint arXiv:2301.02359* (2023).

[86] Wei Zuo, Warren Kemmerer, Jong Bin Lim, Louis-Noël Pouchet, Andrey Ayupov, Taemin Kim, Kyungtae Han, and Deming Chen. 2015. A polyhedral-based systemc modeling and generation framework for effective low-power design space exploration. In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 357–364.

[87] Wei Zuo, Louis-Noel Pouchet, Andrey Ayupov, Taemin Kim, Chung-Wei Lin, Shinichi Shiraishi, and Deming Chen. 2017. Accurate high-level modeling and automated hardware/software co-design for effective SoC design space exploration. In *Proceedings of the 54th Annual Design Automation Conference 2017*. 1–6.