Allo: A Programming Model for Composable Accelerator Design

HONGZHENG CHEN*, Cornell University, USA
NIANSONG ZHANG*, Cornell University, USA
SHAOJIE XIANG, Cornell University, USA
ZHICHEN ZENG†, University of Science and Technology of China, China
MENGJIA DAI†, University of Science and Technology of China, China
ZHIRU ZHANG, Cornell University, USA

Special-purpose hardware accelerators are increasingly pivotal for sustaining performance improvements in emerging applications, especially as the benefits of technology scaling continue to diminish. However, designers currently lack effective tools and methodologies to construct complex, high-performance accelerator architectures in a productive manner. Existing high-level synthesis (HLS) tools often require intrusive source-level changes to attain satisfactory quality of results. Despite the introduction of several new accelerator design languages (ADLs) aiming to enhance or replace HLS, their advantages are more evident in relatively simple applications with a single kernel. Existing ADLs prove less effective for realistic hierarchical designs with multiple kernels, even if the design hierarchy is flattened.

In this paper, we introduce Allo, a composable programming model for efficient spatial accelerator design. Allo decouples hardware customizations, including compute, memory, communication, and data type from algorithm specification, and encapsulates them as a set of customization primitives. Allo preserves the hierarchical structure of an input program by combining customizations from different functions in a bottom-up, type-safe manner. This approach facilitates holistic optimizations that span across function boundaries. We conduct comprehensive experiments on commonly-used HLS benchmarks and several realistic deep learning models. Our evaluation shows that Allo can outperform state-of-the-art HLS tools and ADLs on all test cases in the PolyBench. For the GPT2 model, the inference latency of the Allo generated accelerator is 1.7× faster than the NVIDIA A100 GPU with 5.4× higher energy efficiency, demonstrating the capability of Allo to handle large-scale designs.

CCS Concepts: • **Hardware → High-level and register-transfer level synthesis**; • **Software and its engineering → Compilers**.

Additional Key Words and Phrases: Hardware accelerators, schedule language, accelerator design language, compiler optimization

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

*Equal contribution.
†Work was done when Zhichen and Mengjia interned at Cornell.

Authors' addresses: Hongzheng Chen, Cornell University, USA, hzchen@cs.cornell.edu; Niansong Zhang, Cornell University, USA, nz264@cornell.edu; Shaojie Xiang, Cornell University, USA, sx233@cornell.edu; Zhichen Zeng, University of Science and Technology of China, China, zhichenzeng@mail.ustc.edu.cn; Mengjia Dai, University of Science and Technology of China, China, mjd20021014@mail.ustc.edu.cn; Zhiru Zhang, Cornell University, USA, zhiruz@cornell.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2024 Copyright held by the owner/author(s).
ACM 2475-1421/2024/6-ART171
https://doi.org/10.1145/3656401

1 INTRODUCTION

With the recent trends in technology scaling, computer engineers are increasingly turning to special-purpose hardware accelerators to meet the escalating computational demands of emerging applications, such as large language models (LLMs) [62, 74, 87]. One architectural paradigm that has gained popularity is spatial architecture [27, 31, 42, 43, 56, 97], which instantiates specialized processing engines interconnected through direct wires or streaming buffers to increase throughput and reduce off-chip memory accesses. While hardware specialization can significantly improve performance and energy efficiency, it does entail a substantially higher development effort. Specifically, manually constructing spatial architectures has been notably challenging, particularly with the traditional register-transfer-level (RTL) design abstraction. Consequently, modern accelerator designs are increasingly embracing high-level synthesis (HLS) to expedite RTL code generation and enable rapid exploration of diverse design alternatives [15, 16, 51]. However, to achieve high performance, HLS users must extensively restructure the source program to guide the tool toward realizing specialized architectures like systolic arrays. Additionally, they are required to employ various vendor-specific data types and pragmas, diminishing design reusability and portability.

In this context, we identify two major challenges to the productive development of high-performance accelerators.

Challenge 1: Balancing manual control with automated compiler optimizations. Kernels manually created by experts deliver high-performance implementations but require substantial manual effort for design and validation. Also, these kernels usually adhere to specific data types and function signatures, which hampers their ability to keep up with rapidly evolving applications and hardware advancements. There is an increasing use of automated compiler techniques such as polyhedral compilation to generate on-chip buffers [71], streaming dataflow architectures [13], or systolic arrays [17, 92] from a plain C/C++ code without sophisticated loop annotations. However, these tools typically do not provide adequate control to the designers to explore various performance/cost trade-offs and customize the memory hierarchies and communication schemes for new applications. Embracing domain-specific languages (DSLs) simplifies tasks for both programmers and compilers [25, 28, 35, 57, 84], but most DSLs are inherently tailored for specific application domains, such as image processing, machine learning, and network processing, and they lack support for general-purpose language constructs essential for accelerator hardware design [51].

Challenge 2: Bridging the gap from single-kernel optimization to complex multi-kernel accelerator design. In recent trends, DSLs for hardware design are evolving to become more generalized, incorporating flexible imperative language constructs or being embedded in general host languages such as C++, Python, or Scala [40, 47, 49, 59, 83]. We refer to this category of programming models as accelerator design languages (ADLs). Inspired by Halide [75] and TVM [11], several recent ADLs further separate algorithm definition from hardware optimizations [49, 83], which improves both productivity and portability. However, existing ADLs primarily focus on optimizing single application kernels like convolution and matrix multiplication. In the case of realistic multi-kernel applications, these ADLs tend to generate monolithic flattened designs, sidestepping the intricacies of composing distinct kernels, which may present incompatible interfaces or conflicting optimizations. The inadequate support for composability compromises modularity, debuggability, and often leads to suboptimal performance, as pre-optimized kernels cannot easily be integrated into a hierarchical program structure.

To tackle these challenges, we propose *Allo*, a new programming model for composable design of high-performance spatial accelerator architectures[1]. The key design principles of Allo are to

[1]Allo means "atypical" and reflects our focus on developing non-traditional hardware architectures. The framework is open-source and available at https://github.com/cornell-zhang/allo. The dinosaur in the project logo is an Allosaurus.

provide decoupled hardware customization primitives, modularize the accelerator design process, and facilitate type-safe composition of individual components.

Progressive Hardware Customizations. We inherit the idea from popular schedule languages like TVM [11] and Halide [75] to decouple hardware customizations (e.g., caching and pipelining) from algorithm specifications. Each hardware customization is a primitive that performs a rewrite on the program. We not only decouple the loop-based transformations, but also extend the decoupling to memory, communication, and data types. Each customization primitive can be verified individually and progressively applied to a vanilla program to conduct optimizations.

Reusable Parameterized Kernel Templates. Allo supports declaring type variables during kernel creation and instantiating the kernel when building the hardware executable, which is a feature absent in most hardware ADLs [40, 47, 49] but is important for building reusable hardware kernel libraries. Allo introduces a concise grammar for creating kernel templates, eliminating the need for users to possess complicated metaprogramming expertise.

Composable Schedules. Allo empowers users to construct kernels incrementally from the bottom up, adding customizations one at a time while validating the correctness of each submodule. Ultimately, multiple schedules are progressively integrated into a complete design using the `.compose()` primitive. This approach, unachievable by prior top-down methods, significantly enhances productivity and debuggability.

Holistic Dataflow Optimizations. We introduce a hierarchical dataflow graph to support the composition of multiple kernels within a complex design while maintaining the function boundaries. To ensure the correctness of the interfaces when integrating distinct kernels, we model the interface unification problem as a type inference problem and solve it efficiently through dataflow analysis. Leveraging the hierarchical dataflow graph, we can effectively size the streaming buffers (FIFOs) between stages.

To improve the usability of Allo, we have implemented the frontend language in Python, allowing for a flexible programming style with minimal type annotations. We also present an end-to-end optimizing compiler for Allo, allowing users to write Python programs and generate the hardware bitstream. Moreover, we provide an MLIR dialect that supports decoupled hardware customizations at the IR level and potentially supports multiple different input languages. In summary, our contributions are as follows:

- We introduce Allo, a composable programming model that enables progressive hardware customizations, transforming a vanilla program into a high-performance design, with each step being verifiable.
- We propose composable schedules, enabling users to construct modular hardware accelerators from the ground up by combining customized kernels and external IPs. A type system for the memory layout is also proposed to ensure type safety during schedule composition. Additionally, we introduce holistic dataflow optimizations to ensure functional correctness and enhance performance further.
- We conduct comprehensive experiments on both realistic benchmarks and large neural networks. For PolyBench [69], we outperform several state-of-the-art HLS tools and ADLs [40, 49, 59, 102], across all design cases. Furthermore, we demonstrate the applicability of our programming model in the context of large neural network designs. To the best of our knowledge, we are the first to employ such an ADL for a complete evaluation of LLMs on an FPGA. Our experimental results reveal a 1.7× speedup and 5.4× higher energy efficiency on the GPT2 model compared to the A100 GPU.

2 AN ALLO EXAMPLE

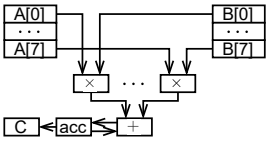
```
1  import allo
2  from allo.ir.types import float32
3  # Algorithm specification
4  M, N, K = 32, 32, 32
5  def gemm(A: float32[M, K],
6           B: float32[K, N],
7           C: float32[M, N]):
8   for i in range(M):
9    for j in range(N):
10    for k in range(K):
11     C[i, j] += A[i, k] * B[k, j]
12
13 # Schedule construction
14 s = allo.customize(gemm)
15 s.unroll("k", 8)
16 s.partition(A, dim=1, factor=8)
17 s.partition(B, dim=0, factor=8)
18
19 # Codegen
20 s.build(target="hls")
```

(a) An example Allo program

```
module {
 func.func @gemm(
    %arg0: memref<32x32xf32>,
    %arg1: memref<32x32xf32>,
    %arg2: memref<32x32xf32>) {
 // Algorithm specification
 affine.for %arg2 = 0 to 32 {
  affine.for %arg3 = 0 to 32 {
   affine.for %arg4 = 0 to 32 {
    // ... Computation
   } {loop_name = "k"}
  } {loop_name = "j"}
 } {loop_name = "i", op_name = "Sijk"}
 %lk = allo.loop_handle "Sijk", "k"
 // Decoupled customizations
 allo.unroll(%lk)
 allo.partition(%arg0, 1, 8)
 allo.partition(%arg1, 0, 8)
 return
}}
```

(b) The corresponding MLIR code

```
void gemm(
 float A[32][32], float B[32][32],
 float C[32][32]) {
 #pragma partition var=A cyclic \
   factor=8 dim=1
 #pragma partition var=B cyclic \
   factor=8 dim=0
 for (int i = 0; i < 32; ++i) {
   for (int j = 0; j < 32; ++j) {
    for (int k = 0; k < 32; ++k) {
    #pragma unroll factor=8
    // ...
}}}}}
```



(c) The generated HLS code

Fig. 1. An example Allo program and the corresponding MLIR and C++ code — The code snippets are simplified for demonstration purposes.

Existing HLS tools often demand users to restructure their application code and insert vendor-specific data types and pragmas to achieve high performance, which are not portable and maintainable. Moreover, with the prevalent use of Python-based frameworks [65, 93] for deep learning models, manually translating those models into HLS C++ is impractical. Therefore, we emphasize the following features as key principles when designing Allo: (1) **Pythonic**: embracing the Python ecosystem makes the Allo coding experience similar to using native Python and effectively reduces the learning burden; (2) **Separation of concerns**: decoupled hardware customizations make the high-performance programs easier to write and maintain; and (3) **Composability**: all the kernels, primitives, and schedules should be composable to form complex designs.

In the following, we begin by implementing a general matrix multiplication (GEMM) kernel in Allo to illustrate the basic syntax and provide clues on why Allo can offer greater productivity compared to HLS C++. As shown in Fig. 1a, we first define the algorithm specification of the GEMM kernel (Lines 5-11), which specifies *what* the kernel computes. Since Allo is a Python-embedded programming language, it supports all the imperative grammars in Python (e.g., `if-else`, `for`, and `while`), with the distinction that users must provide explicit type annotations for function arguments and variable declaration. This requirement arises from the dynamic typing nature of Python, which may not be inherently suitable for hardware generation where static data types are necessary to determine the accurate data bitwidth. The type annotation in Allo consists of the basic element types and shapes. Formal definitions of the types can be found in Supplementary Material A. Apart from the native integer and floating-point data types in Python, Allo accommodates arbitrary-bitwidth integer and fixed-point types. This generality is important for designing high-performance accelerators that declare bitwidth only as needed, ensuring adaptability to diverse hardware requirements.

Once the algorithm is specified, we create a *schedule* by calling `allo.customize` (Line 14). The function passed into `.customize()` is treated as an Allo kernel and will be parsed by the Allo compiler. The schedule is a sequence of optimizations, which specifies *how* the kernel is executed on real hardware. These optimizations can be applied to different algorithms and are independent of any specific algorithm, which allows us to decouple them from the algorithm and encapsulate each customization as a primitive. We unroll the innermost loop by a factor of 8 and provide

```
// Vanilla
void gemm(
  float A[1024][1024], float B[1024][1024],
  float C[1024][1024]
) {
  for (int i = 0; i < 1024; ++i) {
    for (int j = 0; j < 1024; ++j) {
      for (int k = 0; k < 1024; ++k) {
        C[i][j] += A[i][k] * B[k][j];
}}}}

// Inner-product
void ip_gemm(
  float A[1024][1024], float B[1024][1024],
  float C[1024][1024]
) {
  #pragma partition var=A cyclic factor=32 dim=1
  #pragma partition var=B cyclic factor=32 dim=0
  for (int i = 0; i < 1024; ++i) {
    for (int j = 0; j < 1024; ++j) {
      for (int k = 0; k < 1024; ++k) {
      #pragma HLS pipeline II=1 Is it achievable?
      #pragma HLS unroll factor=32
        C[i][j] += A[i][k] * B[k][j];
}}}}
```

	Latency (ms)	II	Freq. (MHz)	Speedup
Vanilla	25074	7	427	1×
Inner-product	17950	128	240	1.4×
Row-wise product	**112**	1	427	223×

```
1  // Row-wise product
2  void rp_gemm(
3    float A[1024][1024], float B[1024][1024],
4    float C[1024][1024]
5  ) {
6    #pragma partition var=B cyclic 32 dim=1
7    #pragma partition var=C cyclic 32 dim=1
8    float buf_C[1024];
9    #pragma partition var=buf_C cyclic 32 dim=0
10   l_i: for (int i = 0; i < 1024; i++) {
11     // 1) initialization
12     l_j_init: for (int j = 0; j < 1024; j++) {
13     #pragma pipeline II=1
14     #pragma unroll factor=32
15       buf_C[j] = C[i][j];
16     }
17     // 2) computation
18     l_k: for (int k = 0; k < 1024; k++) {
19     // reordered reduction loop
20       float a = A[i][k];
21       l_j: for (int j = 0; j < 1024; j++) {
22       #pragma pipeline II=1
23       #pragma unroll factor=32
24         buf_C[j] += a * B[k][j];
25     }}
26     // 3) write-back
27     l_j_back: for (int j = 0; j < 1024; j++) {
28     #pragma pipeline II=1
29     #pragma unroll factor=32
30       C[i][j] = buf_C[j];
31 }}}
```

Fig. 2. HLS code for three different implementations of GEMM kernels — The loop unrolling factors are set as 32. The latency, II, and frequency results are obtained from the HLS report.

multiple banks for array A and B for parallel access using the provided primitives (Lines 15-17). Allo utilizes MLIR as the intermediate representation (IR) and provides an MLIR dialect to decouple these hardware customizations at the IR level, as shown in Fig. 1b.

Lastly, we call `s.build` (Line 20) to lower the MLIR module to the target backend, generating the HLS code as depicted in Fig. 1c. The inserted pragmas align with the schedule in the frontend program, and the generated accelerator executes the GEMM kernel with a parallelism factor of 8.

3 PITFALLS IN HLS-BASED HARDWARE ACCELERATOR DESIGN

In this section, we delve deeper into the limitations of existing HLS tools, which motivates the design of Allo. We identify two common pitfalls in HLS and conduct several experiments to demonstrate these issues. For the experiments, we use a widely used commercial HLS tool and target the AMD Alveo U280 FPGA [96] with a frequency set to 300 MHz.

3.1 Single-Kernel Design

We still leverage the GEMM kernel as an example. Even in this simple case, achieving high performance is not straightforward.

Pitfall I: Simply inserting pragmas cannot lead to high performance. As depicted on the left in Fig. 2, an HLS programmer initially defines a vanilla floating-point GEMM kernel of size 1024×1024, consisting of a loop nest of three levels. If this code is directly fed to HLS, the resulting latency is 25 074 ms even though the HLS tool attempts to automatically pipeline the inner loop.

To further exploit the parallelism of the kernel, an intuitive idea is to unroll and pipeline the innermost loop. Programmers can specify the target initiation interval (II) of the design using `#pragma pipeline`. Given that the innermost loop is unrolled with a factor of 32, arrays `A` and `B`

need to be partitioned into multiple banks to facilitate parallel access. Surprisingly, the latency does not reduce to 1/32 but only 70% of the latency of the original design, with an unfavorable increase in II. This is primarily due to a loop-carried dependency in the floating-point accumulation of `C[i][j]`, which requires more than one cycle to finish, preventing effective pipelining with an II equal to one [20]. Furthermore, the increased II leads to a reduced frequency, potentially causing routing issues during backend synthesis.

To resolve this issue, we can change the loop order to avoid updating the same matrix element in consecutive iterations. As shown on the right of Fig. 2, by swapping the loops of `j` and `k` (Lines 18-25), we transform the accumulation pattern into row-wise product, ensuring that adjacent iterations update different elements of the output matrix. Additionally, a buffer of size 1024 is introduced to store intermediate results (Line 8), which are written back to memory only after iterating through one row. As a result, we can achieve a 112 ms latency with II=1, which achieves a 223× speedup compared to the vanilla implementation.

This example underscores the importance of source-level transformation in HLS-based hardware accelerator design. Adding pragmas alone does not result in high performance; instead, it requires careful program restructuring to enable desired optimizations. Unfortunately, even with the latest design-space exploration (DSE) techniques in HLS compilers [81, 82, 102, 104], identifying such optimizations may prove challenging. These DSE methods commonly search for parameters associated with loop splitting, pipelining, or unrolling, yet they often lack support for crucial memory customizations, as discussed in §8.2. Allo resolves this issue by providing memory customization primitives, allowing users to insert buffers at a given axis (§5.1). A full Allo example can be found in Supplementary Material C.

Further optimizing a GEMM kernel may adopt a systolic array architecture, which requires streaming connections between multiple processing elements and constructing complex I/O networks to achieve high performance. These optimizations require substantial code rewriting and also cannot be accomplished by simple pragma insertion. For example, a high-performance 2×2 systolic array for GEMM already requires more than 1,100 lines of C++ code [92], which demonstrates the complexity of single-kernel HLS design.

3.2 Multi-Kernel Design

Once we have optimized a single-kernel GEMM design, the next challenge is to employ it as a fundamental building block for large designs. In this context, we aim to construct a two-layer feed-forward network (FFN) module, a component commonly used in Transformer models [23, 74, 90]. However, it remains a non-trivial task even though we already have an optimized GEMM kernel.

Pitfall II: Simply calling optimized kernels does not guarantee a high-quality design. As depicted on the left of Fig. 3, within the top-level function, we input an initial tensor `X` and two weight parameters, `W_A` and `W_B`, followed by the output being written to `Y` (Lines 7-10). In the main body, we create an intermediate tensor `Z` (Line 11), reuse the `rp_gemm` kernel defined in Fig. 2, and invoke it twice to perform a linear layer computation (Lines 12-13). This approach intuitively chains two function calls together.

Based on the results in Fig. 2, cascading two GEMM kernels should yield a latency of 224 ms, since a single-kernel GEMM has a latency of 112 ms. However, the HLS report in Fig. 3 indicates a latency of 280 ms, which is 1.25× slower than expected. Furthermore, reusing the GEMM kernel for these two function calls should maintain resource usage at the same level as a single kernel, yet the HLS report indicates a doubling of resource utilization. Closer examination reveals that HLS generates two distinct copies of the GEMM kernel, named `rp_gemm` and `rp_gemm_1`, with `rp_gemm_1` exhibiting a worse latency than `rp_gemm`. The root cause is the function interface,

```
// Simple cascade
void rp_gemm(
  float A[1024][1024], float B[1024][1024],
  float C[1024][1024]
) { /*  See Fig. 2  */ }

void top(float X[1024][1024],
         float W_A[1024][1024],
         float W_B[1024][1024],
         float Y[1024][1024]) {
  float Z[1024][1024];
  rp_gemm(X, W_A, Z);
  rp_gemm(Z, W_B, Y);
}
```

	Latency (ms)	BRAM	DSP	FF	LUT
Simple cascade	280	1984	320	42761	24896
+ rp_gemm	112	64	160	21391	11765
+ rp_gemm_1	168	64	160	21364	11856
Interface uni.	**224**	1920	160	21377	16068
+ rp_gemm	112	64	160	21372	11913

```
// Interface unification
void rp_gemm(
  float A[1024][1024], float B[1024][1024],
  float C[1024][1024]
) {
  // explicitly partition A
  #pragma partition var=A cyclic factor=32 dim=1
  #pragma partition var=B cyclic factor=32 dim=1
  #pragma partition var=C cyclic factor=32 dim=1
  // ...
}

void top(float X[1024][1024],
         float W_A[1024][1024],
         float W_B[1024][1024],
         float Y[1024][1024]) {
  #pragma allocation instances=rp_gemm limit=1
  float Z[1024][1024];
  rp_gemm(X, W_A, Z);
  rp_gemm(Z, W_B, Y);
}
```

Fig. 3. HLS code for cascading two GEMM kernels — Changes are highlighted in yellow.

where, in Fig. 3, we partition the second and third arguments (A and B) for the rp_gemm function. These two arguments correspond to the arrays W_A and Z in the top-level function. However, Z, already a partitioned array, is once again passed into rp_gemm as the first argument, triggering partitioning of the first argument of the rp_gemm function. This divergence in partitioning leads HLS to view the two rp_gemm kernels as distinct, with the first kernel partitioning the latter two arguments, while the second kernel partitions all three arguments. Thus, two different copies of the rp_gemm kernel are generated. Consequently, two distinct copies of the rp_gemm kernel are generated, and an unintended partition scheme causes HLS to make incorrect assumptions about loop variable dependencies, resulting in increased latency.

To rectify this issue and ensure proper sharing of function units while generating a design with the anticipated latency, we work towards unifying the function interface. As shown on the right of Fig. 3, we explicitly partition the first argument of the rp_gemm kernel, thereby ensuring that all inputs and outputs are partitioned consistently. Additionally, we enforce an allocation pragma to ensure the generation of only one function instance. As a result, HLS produces a single copy of the rp_gemm kernel, as indicated by the resource usage in the bottom-left of Fig. 3. Moreover, the latency is twice that of a single-kernel latency, totaling 224 ms, aligning with our expectations.

This example highlights the inherent complexity of composing multiple kernels, requiring careful consideration of appropriate interfaces for each kernel and effective connection through intermediate buffers. Allo introduces composable schedules and holistic optimizations to resolve this issue. Further insights will be discussed in §6.

4 ALLO OVERVIEW

Recently, various accelerator design languages (ADLs) have been proposed to mitigate the limitations of HLS. Some of these approaches expose hardware customizations in a higher-level language, requiring users to follow specific coding styles and relying on compilers to generate high-performance implementations [26, 47, 86]. While this approach can partially resolve Pitfall I if the compiler is able to generate a proper memory hierarchy for the design, it requires users to write code in a functional language or in their custom formats, subsequently generating HDL code in Verilog or Chisel [4]. This imposes a significant burden on programmers to translate their applications and debug in these languages. Conversely, other ADLs are built on top of the original HLS C++ toolchain [40, 49, 59]. HeteroCL [49] introduces the concept of separation of concerns in

Table 1. Comparison between Allo and existing high-level hardware languages — APL denotes accelerator programming language, ADL is accelerator design language, and IL denotes intermediate language.

				Single-Kernel Design (§5)							Multi-Kernel Design (§6)	
					Customizations							
Framework	Type	Input Language	Output Format	Decoupled Y/N	Compute	Memory	Types	Comm.	Verifiable Rewrites	Template Kernels	Composable Optimizations	Dataflow Optimizations
TVM [11]	APL	Python	N/A	✓	✓	✓	✓	✗	✗	✓	✗	✗
Exo [41]	APL	Python	N/A	✓	✓	✓	✓	✗	✓	✗	✗	✗
Spatial [47]	ADL	Custom	Chisel	✗	✓	✓	✓	✓	✗	✓	✗	✓
Aetherling [26]	ADL	Haskell	Chisel	✗	✗	✓	✗	✗	✗	✗	✗	✓
Fleet [86]	ADL	Scala	Chisel	✗	✗	✓	✗	✗	✗	✗	✗	✗
ScaleHLS [102]	IL	C++/MLIR	HLS C++	✗	✓	✗	✗	✗	✗	✗	✗	✗
PyLog [40]	ADL	Python	HLS C++	✗	✓	✓	✓	✗	✗	✗	✗	✗
HeteroCL [49]	ADL	Python	HLS C++	✓	✓	✓	✓	✗	✗	✗	✗	✗
Dahlia [59]	ADL	Custom	HLS C++	✗	✓	✓	✗	✗	✗	✗	✗	✗
Allo	ADL/IL	Python/MLIR	HLS C++	✓	✓	✓	✓	✓	✓	✓	✓	✓

hardware design and provides primitives for users to optimize the program. Dahlia [59] proposes a type system to ensure the consistency of memory partitioning and loop unrolling but lacks crucial customizations for pipelining and dataflow. Both ScaleHLS [102] and PyLog [40] automate hardware design and produce HLS C++ code as output. Nevertheless, most of these ADLs focus on optimizing a single kernel and cannot efficiently address Pitfall II.

In this section, we present an overview of the Allo programming model and compilation flow. A comparison between Allo and other high-level hardware languages and compilers is listed in Table 1. Allo fully decouples hardware customizations from algorithm specifications, with a particular focus on enhancing memory and communication customizations. This approach effectively addresses Pitfall I (see §5). Furthermore, Allo provides the ability to declare parameterized kernels, thereby improving the usability of single-kernel designs. What differentiates Allo from other ADLs is its ability to compose individual kernels and construct large-scale, high-performance designs. Allo proposes composable schedules and holistic dataflow optimization to efficiently tackle Pitfall II (see §6). Moreover, we leverage Allo to design a spatial architecture for large language models (LLMs) and execute the design on an FPGA. The FPGA on-board evaluation shows its functionality and high performance, which is unachievable by prior ADLs (§8.3).

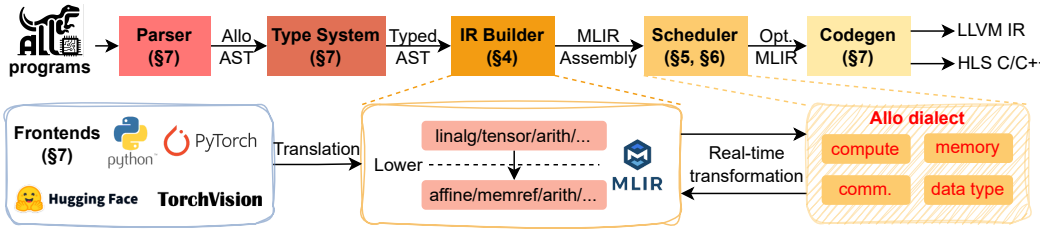


Fig. 4. Overview of the Allo compilation flow.

As illustrated in Fig. 4, Allo offers a Python-embedded ADL for improved productivity, and the Allo compiler follows a conventional compilation workflow. Users can either use the Python frontend to write a Python kernel or leverage the PyTorch frontend to directly import deep learning models from TorchVision [55] and HuggingFace [93], which will be further discussed in §7. For the Python kernel, Allo first parses it into an abstract syntax tree (AST). The AST then proceeds through a type system that performs essential tasks such as type checking, type inference, and type conversions based on user-provided annotations (§7). The typed AST is subsequently passed into an intermediate representation (IR) builder. We develop an Allo dialect within the MLIR ecosystem, which facilitates the separation of hardware customizations at the IR level (§5). The IR builder generates MLIR programs for customization and code generation. Once the IR is obtained, program transformations are applied using the provided customization primitives (§5). This approach seamlessly integrates with different in-tree MLIR dialects, enabling the generation of high-performance

Table 2. A partial list of the customization primitives supported by Allo.

Compute Customizations		Memory Customizations	
`s.split(i,v)`	Split loop `i` into a two-level nested loop with `v` as the bound of the inner loop.	`s.buffer_at(A,i)`	Create an intermediate buffer at loop `i` to store the results of array `A`.
`s.fuse(*l)`	Fuse multiple sub-loops `l` in the same nest loop into one.	`s.reuse_at(A,i)`	Create a buffer storing the values of array `A`, where the values are reused at loop `i`.
`s.reorder(*l)`	Switch the order of sub-loops `l` in the same nest loop.	`s.partition(A,d,v)`	Cyclic/Block partition dimension `d` of array `A` with a factor `v`.
`s.compute_at (Op1,Op2,i)`	Merge loop `i` of the operation `Op1` to the corresponding loop level in operation `Op2`.	`s.pack(A,i,v)`	Pack dimension `i` of array `A` into words with a factor `v`.
`s.unroll(i,v)`	Unroll loop `i` by factor `v`.		
`s.unfold(i)`	Unfold loop `i` as hardware instances.	**Communication Customizations**	
`s.pipeline(i,v)`	Schedule loop `i` in a pipeline manner with a target initiation interval `v`.	`s.relay(A,Dst,v)`	Connect array `A` to destination `Dst` with a FIFO of depth `v`.

designs for diverse backend targets. Lastly, we generate LLVM IR [52] for CPU simulation and HLS C/C++ [100] for hardware synthesis (§7). Notice most of the hardware customizations are target-independent, allowing our compilation flow to target ASIC designs as well. We plan to integrate the CIRCT [14] project as our backend to support custom circuit generation.

5 CUSTOMIZABLE HARDWARE TRANSFORMATIONS

In this section, we use a systolic array [48] as an example to illustrate the language features of Allo, showcasing its capabilities in handling complex transformations for a single kernel design. The systolic array is a prevalent spatial architecture extensively employed in deep learning accelerators such as Google TPUs [43] and AWS Inferentia [3]. It comprises a set of processing elements (PEs) that iteratively execute repetitive operations. By reusing data across these PEs, it minimizes off-chip memory access, resulting in high performance with minimal energy consumption.

5.1 Schedule Construction

Given the initial algorithm definition in Lines 2-6 of Fig. 5, we transform the algorithm specification into a tangible hardware implementation. Here, we formally define a *schedule* $\mathcal{S}$ of a program P_0 as a sequence of transformations $(p_i)_{i=1}^{N}$ such that

$$P_0 \overset{p_1}{\rightsquigarrow} P_1 \overset{p_2}{\rightsquigarrow} \cdots \overset{p_N}{\rightsquigarrow} P_N\,, \tag{1}$$

where $\overset{p_i}{\rightsquigarrow}$ denotes a program rewrite with a primitive p_i, and N is the number of customization primitives in this schedule.

Table 2 lists the primitives supported by Allo. The compute customizations transform the loops and attach necessary attributes, which inherit the idea from existing schedule languages [11, 41, 49, 75]. Notice instead of implementing monolithic compiler passes for the primitives [11, 49, 75], Allo adopts an approach akin to Exo [41], which treats primitives as program rewrites, ensuring correctness for each transformation. Users can print the intermediate module after each customization to inspect real-time program transformations, providing deeper insights into the customization primitives. Moreover, we develop an Allo-MLIR dialect to implement those customizations primitives at the IR level, with each primitive corresponding to an operation in the Allo dialect. It enables Allo to serve as an intermediate language and support different frontends. In the subsequent discussion, we will primarily focus on memory and communication customizations, which distinguish Allo from other ADLs.

As illustrated in Fig. 5, users can create a schedule by invoking the `allo.customize` function and progressively apply primitives to the newly-formed schedule (Line 9). Each primitive exactly does one transformation as shown on the right of Fig. 5. We start by creating intermediate buffers for `A` and `B` arrays (Lines 10-11), which creates a line buffer for peripheral PEs to efficiently load

```
1   # Algorithm specification
2   def gemm(A: int8[M, K], B: int8[K, N],
3            C: int16[M, N]):
4     for i, j in allo.grid(M, N, "PE"):
5       for k in range(K):
6         C[i, j] += A[i, k] * B[k, j]
7
8   # Schedule construction
9   s = allo.customize(gemm)
10  buf_A = s.buffer_at(s.A, "j")            # p0
11  buf_B = s.buffer_at(s.B, "j")            # p1
12  pe = s.unfold("PE", axis=[0, 1])         # p2
13  s.partition(s.C, dim=[0, 1])             # p3
14  s.partition(s.A, dim=0)                  # p4
15  s.partition(s.B, dim=1)                  # p5
16  s.relay(buf_A, pe, axis=1, depth=M + 1)  # p6
17  s.relay(buf_B, pe, axis=0, depth=N + 1)  # p7
```

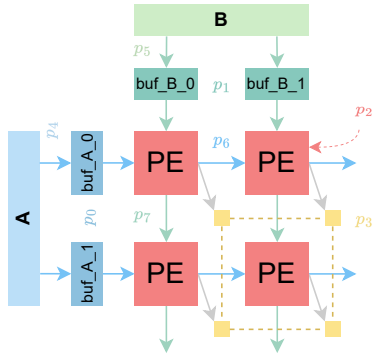


Fig. 5. Allo program for an integer output-stationary systolic array — M, N, and K are predefined integer constants. `allo.grid` is a syntactic sugar for multiple Python `range`-for loops. A and B are located in DRAM.

data from off-chip memory. After that, we can easily generate $M \times N$ PEs along the 0th and 1st axes by invoking `.unfold()` (Line 12). Further configurations on the number of PEs can be controlled by loop tiling using the `.split()` primitive. As these PEs are actual hardware instances, and each PE requires parallel memory access, C is partitioned in both dimensions to accommodate the nature of output-stationary accumulation (Line 13). Furthermore, A and B arrays are also partitioned to create multiple banks, facilitating parallel data input into the line buffers (Lines 14-15). More importantly, users need to specify the intra-kernel communication. Allo offers a seamless way to connect neighboring PEs with FIFOs through the `.relay()` primitive. As shown in Lines 16-17, `buf_A` is connected along the 1st axis, while `buf_B` is connected along the 0th axis, and these connections will subsequently be synthesized as FIFOs within the hardware. A generated HLS C++ code for this systolic array is attached in Supplementary Material D for reference.

Notice this approach is general enough to allow users to express programs in different forms. Users can start from an arbitrary program status P_i using our programming model and apply customization primitives p_i to obtain a transformed program P_{i+1}. For example, in Fig. 5, the program P_1 after applying p_0 is still functional, as it essentially moves data from DRAM to an intermediate buffer and lets the computation logic load the same data from this buffer. This is not achievable by ad-hoc systolic array compilers like AutoSA [92]. The optimization process in AutoSA is not transparent, and programmers cannot easily configure the architecture of the generated systolic arrays. In contrast, our approach allows customization of compute, memory, data types, and communication. We can further create an additional memory hierarchy using the `.buffer_at()` primitive for `buf_A` and `buf_B`, which reduces the memory fan-out to one. Additionally, Allo provides more flexibility than semi-manual systolic array generators like SuSy [50] and T2S [83], which require users to write verbose uniform recurrence equations and conduct complex space-time transformations manually. On the contrary, Allo can start from a vanilla GEMM kernel and progressively transform it into a functional systolic array using eight lines of schedule code. Leveraging the provided primitives, Allo can strike the right balance between compiler-based optimizations and manual optimizations. As demonstrated in §8.2, our programming model can not only be used to generate systolic arrays but is also general enough to support different applications.

5.2 Verification

Ensuring the correctness of the generated accelerator is of great importance. Allo employs two key verification procedures to enhance the reliability of the generated code. First, Allo leverages the CPU backend to conduct functional simulation testing (§7). Second, Allo integrates an equivalence

```
1  # Algorithm specification
2  def gemm1(A: float[M, K], B: float[K, N],
3            C: float[M, N]):
4      ...
5  def gemm2(C: float[M, N], D: float[N, P],
6            E: float[M, P]):
7      ...
8  def two_mm(A: float[M, K], B: float[K, N],
9             D: float[N, P], E: float[M, P]):
10     C: float[M, N]
11     gemm1(A, B, C)
12     gemm2(C, D, E)
13 # Schedule construction
14 s_orig = allo.customize(two_mm)
15 # Duplicate schedule for verification
16 s = allo.customize(two_mm)
17 s.reorder("gemm2:k", "gemm2:j")
18 s.buffer_at(s.C, axis="gemm2:i")
19 s.relay(s.A, "gemm2")
20 s.reorder("gemm1:j", "gemm1:i")
21 # 1. Functional simulation testing
22 f = s.build()
23 # ... Initialize NumPy arrays (omitted)
24 f(np_A, np_B, np_D, np_E)
25 # 2. Formal equivalence checking
26 allo.verify(s, s_orig)
```

(a) Allo code snippet

```
1  void gemm1(float A[M][K], float B[K][N], stream C) {
2    for (int j = 0; j < N; j++) {
3      for (int i = 0; i < M; i++) {
4        float sum = 0;
5        for (int k = 0; k < K; k++) {
6          sum += A[i][k] * B[k][j];
7        C.write(sum);
8  }}}}
9  void gemm2(stream C, float D[N][P], float E[M][P]) {
10   float buf_E[N];
11   for (int i = 0; j < M; j++) {
12     // ... Initialize buf_E (omitted)
13     for (int k = 0; k < N; k++) {
14       // A mismatch between read and write
15       float c = C.read();
16       for (int j = 0; j < P; j++) {
17         buf_E[j] += c * D[k][j];
18     }}
19     // ... Write-back to E (omitted)
20 }}
21 void two_mm(float A[M][K], float B[K][N],
22             float D[N][P], float E[M][P]) {
23   stream C_fifo;
24   gemm1(A, B, C_fifo);
25   gemm2(C_fifo, D, E);
26 }
```

(b) Corresponding HLS C++ code snippet

Fig. 6. A buggy Allo example with data streaming and loop reordering — The two matrix multiplications are computed back-to-back. The code marked in red indicates bugs in the program.

checker [70] to formally verify the equivalence of the programs before and after customizations, provided that the programs have statically interpretable control-flow (SICF). SICF requires the problem size to be known at compile-time and does not support parametric loop nest analysis.

Fig. 6 shows a data access order bug caused by incorrect customizations. In this example, Lines 17-18 of Fig. 6a transform the second matrix multiplication from an inner-product to a row-wise product, which reads the input `C` in a row-major order. However, Line 20 reorders the first matrix multiplication loops to send the output `C` in a column-major order. This discrepancy in data receiving and sending orders violates the requirement of in-order access on a stream FIFO. The `.verify()` on Line 26 invokes the equivalence checker which takes the schedule before (Line 14) and after customizations (Line 20) to formally verify the program semantic equivalence. In this example, the customizations break the accelerator design and cause a semantic difference in the customized code. The difference in the symbolic representation is detected and reported by the equivalence checker to facilitate debugging. Notice the verification can be conducted after each primitive is applied, ensuring the correctness of the transformations at each step.

5.3 Parameterized Kernel Templates

We initially constructed a systolic array with fixed dimensions and data types, which lacks flexibility when handling variable-sized input matrices. In the following, we leverage the previously defined systolic array to introduce a tiled design that accommodates inputs of arbitrary sizes.

Allo provides a user-friendly parameterization template to facilitate polymorphism. As illustrated in Fig. 7, we parameterize the `systolic` function with type parameters. Users can define the function signature using the syntax `def <func>[<type params>](<args>)`. Again, given Allo's decoupling of data types from the algorithm specification, both data types and shapes can serve as type parameters. Allo permits additional constraints for parameterized data types. For instance, `Ty: (int32, float32)` specifies that the data type `Ty` must be either `int32` or `float32`. If any other data types are used, an error is raised. Within the `tiled_systolic` function, we partially

```
def systolic[TyA, TyB, TyC, Mt: index, Nt: index, K: index]
    (A: TyA[Mt, K], B: TyB[K, Nt], C: TyC[Mt, Nt])

def tiled_systolic[TyA, TyB, TyC, M: index, N: index, K: index]
    (A: TyA[M, K], B: TyB[K, N], C: TyC[M, N]):
    local_A: TyA[8, K]; local_B: TyB[K, 8]; local_C: TyC[8, 8]
    for mi, ni in allo.grid(M // 8, N // 8, name="outer_tile"):
        # ... load_A_tile, load_B_tile
        systolic[TyA, TyB, TyC, 8, 8, K](local_A, local_B, local_C)
        # ... store_C_tile
```

Fig. 7. Tiled systolic array implementation in Allo.

specialize the inner systolic array with a fixed size of 8×8, allowing us to derive a tiled version of the systolic array capable of accommodating inputs of varying dimensions.

In Allo, we have encapsulated several template kernels as libraries, each accompanied by predefined schedules. These templates include commonly used deep learning operators and high-performance systolic arrays for matrix-matrix multiplications and matrix-vector multiplication. This approach allows users to conveniently reuse these kernels for their own workloads, reducing the burden of writing efficient schedules. Additionally, this parameterized interface facilitates auto-tuning and auto-scheduling [12, 78, 107], which we plan to explore in future work.

6 COMPOSABLE SCHEDULES

In this section, we explore the process of composing multiple schedules to construct a complete design. We begin by introducing the `.compose()` primitive and delve into the implementation details of the hierarchical dataflow graph. We then present the algorithm for schedule replay and memory layout composition. Lastly, we extend Allo to support the composition of external kernels and present an algorithm for holistic optimization.

6.1 `.compose()` Primitive

We leverage the systolic array implementation in Fig. 7 to illustrate how to cascade two systolic arrays to create a larger design. As depicted in Fig. 8, the two systolic arrays are arranged in sequence, with an intermediate tensor Z facilitating data transfer. This aligns with common practices in nowadays neural network implementations [65].

```
def top(X: int8[32, 64]) -> int8[64, 32]:
    Z: int8[32, 64] = 0
    Y: int8[64, 32] = 0
    W_A: int8[64, 64] = W_A_cst
    W_B: int8[64, 64] = W_B_cst
    tiled_systolic[int8, int8, int16, \
        32, 64, 64, "FFN1"](X, W_A, Z)
    tiled_systolic[int8, int8, int16, \
        64, 32, 64, "FFN2"](Z, W_B, Y)
    return Y

s_top = allo.customize(top)
s_top.compose(s) # `s` is an optimized schedule
s_top.relay(s_top.Z, "tiled_systolic_FFN2")
```

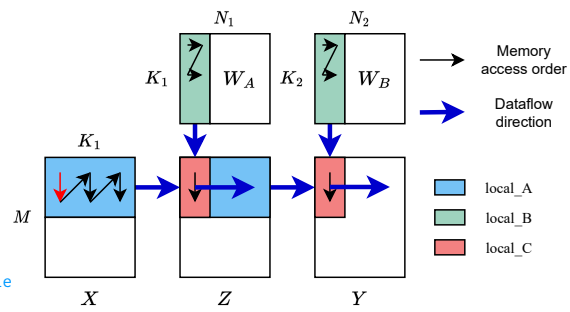


Fig. 8. Cascading two systolic arrays — `W_A_cst` and `W_B_cst` are predefined weight parameters. Automatic type conversion on Z is performed between two kernels.

Based on the primitives proposed in §4, we can optimize individual kernels independently. However, we need a mechanism to connect these smaller kernels. Therefore, we propose a new primitive, `s.compose(<new_schedule>,<id>)`, for incorporating a `<new_schedule>` into the original schedule `s`. In Fig. 8, we call `s_top.compose(s)` to integrate an optimized schedule `s` of `tiled_systolic`

to `s_top` (Line 13). The additional `<id>` argument is used to distinguish between different function calls. For instance, there are two callers to `tiled_systolic` in Lines 6 and 8. By including an additional identifier in the type parameter list (e.g., `"FFN1"`, `"FFN2"`), users can customize them differently. The Allo compiler will generate two distinct instances for those functions. This fine-grained control empowers users to customize specific functions to meet their requirements, offering flexibility not available in traditional compilers with fixed patterns in the compiler pass. Finally, the two systolic arrays are linked with another FIFO to establish a dataflow at the top level (Line 14).

Moreover, `.compose()` primitive can act as an IP integrator to import C++ kernels from existing HLS-based kernel libraries [98, 99]. Allo parses the function interface of the HLS IPs, and the function arguments will be translated into internal representations labeled with Allo-supported data types. As long as users specify the correct number of arguments, Allo automatically generates wrapper functions with PyBind11 [72], making it more general and extensible to different customizations.

6.2 Hierarchical Dataflow Graph

```
def add(A: T[M, N]) -> T[M, N]:
    B = A + 1
    return B
def mul(A: T[M, K], B: T[K, N]) -> T[M, N]:
    # ... Calculate matrix multiply of A and B
def top(A: T[M, K], B: T[K, N]) -> T[M, N]:
    C = mul(A, B)
    D = add(C)
    return D
```

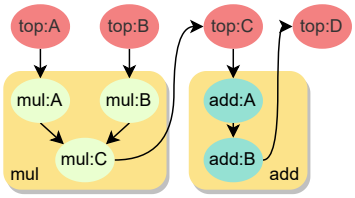


Fig. 9. Example of a hierarchical dataflow graph.

Traditional dataflow graphs used in compilers for analysis are typically flattened, which removes the boundaries between functions and may miss potential optimization opportunities at the graph level [11, 49, 76]. To preserve the hierarchy of the modules during scheduling, we require a new data structure capable of representing the dataflow graph in a way that facilitates the analysis of interfaces between functions. As a solution, we propose a hierarchical dataflow graph that connects nested functions in a hierarchical way.

Given that the constructed MLIR is in SSA form, each node in the dataflow graph represents an operation in the IR. As shown in Fig. 9, the `top` function calls two subfunctions, `mul` and `add`, within its body. To represent nodes in the dataflow graph, we use variable names with the function name as a prefix. We maintain information about the caller and callee in the dataflow graph and explicitly connect function arguments with the caller's operands using edges. For example, array `C` is passed into the function `add`, so there is an edge from `top:C` to `add:A`. Since our primary concern is how data flows and not how many iterations are needed, control flow is eliminated from the graph. This simplifies the composition process discussed in Section 6.4.

6.3 Schedule Replay

We formalize the process of schedule composition as follows. Consider a schedule of program P, represented as a sequence of primitives $\mathcal{S}_P = (p_0^P; \ldots; p_{N_P}^P)$. Composing this schedule $\mathcal{S}_P$ with another schedule $\mathcal{S}_Q$ involves appending the customization primitives of $\mathcal{S}_Q$ after the primitives of $\mathcal{S}_P$, i.e., $\mathcal{S}_Q \circ \mathcal{S}_P$, where $\circ$ denotes sequence concatenation.

We outline a general algorithm for composing two schedules in Algorithm 1, and it is easy to extend it to multiple schedules. This algorithm traverses the primitives in the input schedules and replays them in the context of the new program space. Since some functions may be renamed or duplicated due to the `<id>` interface, we must update the arguments of the primitives to ensure they

Algorithm 1: Composing multiple schedules with schedule replay

Data: Two schedules $\mathcal{S}_P$ and $\mathcal{S}_Q$ for programs P and Q
Result: Composition of the schedules $\mathcal{S}_{out} = \mathcal{S}_Q \circ \mathcal{S}_P$ and the output program P' after applying $\mathcal{S}_{out}$

1 Initialize $\mathcal{S}_{out} = \mathcal{S}_P$;
2 **foreach** *primitive* $p_i \in \mathcal{S}_Q$ **do**
3 Update the arguments of p_i to refer to the functions and arguments in program P;
4 **if** p_i *conflicts with primitives in* $\mathcal{S}_{out}$ **then**
5 Composition fails, raise an error;
6 Append p_i to $\mathcal{S}_{out}$;
7 Apply each primitive in $\mathcal{S}_{out}$ to the program P to obtain P'

apply to the correct function and operations (Line 3). Before applying a new primitive, we verify that it will not conflict with previously applied primitives (Line 4). For compute customizations, conflicts occur only when the same operation is targeted. In such cases, an error is raised because the operation has already been transformed (Line 5). Conflicts related to `.partition()` and `.relay()` will be discussed further in §6.4. The primitive is then appended to the new schedule (Line 6). Primitives are applied to the program P only when all the sub-schedules are integrated into the top-level function and are ready for backend executable construction (Line 7), which saves redundant transformation time. The resulting $\mathcal{S}_{out}$ can be used for subsequent composition. This progressive composition process allows us to combine small designs step by step, culminating in the construction of a large design, with each submodule thoroughly tested, as discussed in §5.2.

6.4 Memory Layout Composition

When the customization primitives only affect the inner computation, it is straightforward to replay them with Algorithm 1. However, complexity arises when schedules overlap through the function interface. If a sub-schedule changes the function interface, the parent program must also change to avoid conflicts, as discussed in § 3.2. It is important to maintain the consistency between function call arguments and actual function definitions. Array partitioning is an example of this challenge. In Fig. 5, when the schedule is integrated into Fig. 7, the `local_A`, `local_B`, and `local_C` arrays should also be partitioned. This is because these arrays are partitioned within the `systolic` function and are passed into the function as arguments.

Since hardware memory partitioning essentially alters data layout, we can explicitly represent data layouts as types [53, 66] and conduct analysis within this type system. As shown in the left side of Fig. 10, we consider the partition type of an N-dimensional array. The partition type τ is a composite type consisting of the base type $\hat{\tau}$ for each dimension. Each base type can assume one of four choices. $\bot$ means fully partitioning this dimension, allowing parallel access to all elements. $\top$ represents no partition in this dimension, resulting in only one memory bank on the hardware. $\mathcal{C}_\alpha$ represents cyclic partitioning with a factor of α, where the elements in the original array are interleaved. $\mathcal{B}_\alpha$ denotes block partitioning with a factor of α, where the original array is divided into consecutive blocks. Denote s_i as the size of dimension i, and α should be an integer factor of s_i (not including 1 and itself). We can construct subtype relations for these base types. If $X <: Y$, it means the code expecting a memory with partition type Y is also compatible with a memory with the partition type X. For example, $\bot$ is a subtype of $\mathcal{C}_2$ because complete partitioning already partitions the array into cyclic with a factor of 2. If a kernel requires an array to be cyclic partitioned into two banks to access the elements in parallel, it is also fine to pass in a fully partitioned array since it offers more memory banks. Notably, this subtyping relation is covariant, which means that the subtyping relation of base types $\hat{\tau}$ applies to composite types τ as well. This subtyping relation

actually forms a lattice, where each pair of elements in the type definition has a unique supremum $\top$ and a unique infimum $\bot$. The right side of Fig. 10 shows an example Hasse diagram.

$$\tau := (\hat{\tau}_1, \ldots, \hat{\tau}_N)$$
$$\alpha := \mathbb{N}$$
$$\hat{\tau} := \bot \mid \mathcal{C}_\alpha \mid \mathcal{B}_\alpha \mid \top$$

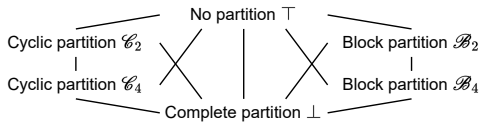


Fig. 10. Left: Definition of the partition types. Right: Example lattice of partition types for a 1D array of shape $(8,)$ — Since 8 can be divided into 2×4, there are six elements in this lattice.

After constructing this simple language for memory layout, we can define the typing rules for these partition types. As shown in Fig. 11, the first row demonstrates the subtyping relations of base types, and the second row shows the composite rule and function application. For function application, it is important to ensure the partition types of function signatures and caller operands are compatible, which results in a subtyping relation between τ_3 and τ_1. For example, in Fig. 3, the original function `rp_gemm` has already partitioned array `Z` in $\mathcal{C}_{32}$, while the program attempts to pass in `Z`, which has no partitions (i.e., $\top$). Therefore, our type system directly rejects this program, avoiding possible performance issues in Pitfall II.

S-Bottom-C
$$\frac{}{\bot <: \mathcal{C}_\alpha}$$

S-Bottom-B
$$\frac{}{\bot <: \mathcal{B}_\alpha}$$

S-Cyclic
$$\frac{\alpha_2 \equiv 0 (\bmod \alpha_1)}{\mathcal{C}_{\alpha_2} <: \mathcal{C}_{\alpha_1}}$$

S-Block
$$\frac{\alpha_2 \equiv 0 (\bmod \alpha_1)}{\mathcal{B}_{\alpha_2} <: \mathcal{B}_{\alpha_1}}$$

S-Top-C
$$\frac{}{\mathcal{C}_\alpha <: \top}$$

S-Top-B
$$\frac{}{\mathcal{B}_\alpha <: \top}$$

S-Array
$$\frac{\exists i \in \{1, \ldots, N\} : \ \hat{\tau}_i <: \hat{\tau}'_i}{(\hat{\tau}_1, \ldots, \hat{\tau}_N) <: (\hat{\tau}'_1, \ldots, \hat{\tau}'_N)}$$

FuncApp
$$\frac{\Gamma \vdash f : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e : \tau_3 \quad \tau_3 <: \tau_1}{\Gamma \vdash f e : \tau_2}$$

Fig. 11. A partial list of typing rules for the partition types in Fig. 10 — Γ is the typing context.

Based on the typing rules, new partition types need to be assigned to each variable in the program after schedule composition. The unification algorithm commonly employed in functional languages for type inference does not fit in this case, due to the presence of subtyping relations [18, 19, 37]. In general, conducting type inference with subtypes can be a challenging task, which requires complex algebraic operations or leveraging an SMT solver to solve the constraints [24, 64, 68]. However, given the lattice property of the subtyping relations, as well as the hierarchical dataflow graph constructed in §6.2, we can apply dataflow analysis on this dataflow graph to efficiently assign types for the variables in the transformed program.

Consider a dataflow graph with M nodes, we can use Algorithm 2 to calculate the proper memory layout of each node. This iterative algorithm resembles the Worklist algorithm used in static dataflow analysis [44]. We use a concrete example in Fig. 9 to illustrate the process. Suppose we apply fully partition on array `C`, i.e., $t'_{in} = \bot$. We first add the target node `top:C` and the target partition type t'_{in} to the worklist (Line 1). In the first iteration, we calculate `top:C`'s type as $t_{\texttt{top:C}} \leftarrow \bot \sqcap \top = \bot$ (Line 4), where $\sqcap$ is the greatest lower bound (GLB) operator. Since its type changes (Line 5), we traverse its predecessors (i.e., `mul:C`) and successors (i.e., `add:A`) (Line 6) and append them to the worklist since they are not in the same function (`top`) with `C` (Line 7-8). Similarly, in the latter iterations, $t_{\texttt{mul:C}}$ and $t_{\texttt{add:A}}$ are updated to have type $\bot$, and no more dataflow nodes in the worklist update the types, which finalizes the algorithm. Notice the $\sqcap$ operator in Line 4 is used to handle

Algorithm 2: Partition type inference (Memory layout propagation)

Data: The partition type $(t_1^{(0)}, \ldots t_M^{(0)})$ of the nodes $(n_1, \ldots, n_M)$ in the hierarchical dataflow graph, and a `.partition()` primitive on node n_{in} that transforms type t_{in} to t'_{in}

Result: Result partition type $(t_1^{(out)}, \ldots t_M^{(out)})$

Initialize Worklist $\leftarrow \{(n_{in}, t'_{in})\}$;
while *Worklist is not empty* **do**
  Pick an item of dataflow node and target type (n, t') from Worklist;
  Update type $t_n^{(next)} \leftarrow t' \sqcap t_n^{(curr)}$;
  if $t_n^{(next)} \neq t_n^{(curr)}$ **then**
    foreach *predecessors and successors* $\tilde{n}$ *of* n **do**
      if $\tilde{n}$ *and* n *are in different functions* **then**
        Add $(\tilde{n}, t_n^{(next)})$ to Worklist;

more general cases of merging two partition types, which makes sure, for example, even for two different types $\mathcal{C}_4$ and $\mathcal{B}_2$ in Fig. 10, they can be cast to a common type (e.g., $\perp$).

Algorithm 2 is guaranteed to terminate in linear time, as formally stated in the following:

Theorem 6.1. *Algorithm 2 can terminate in $O(M)$ steps.*

This termination condition can be established through the Knaster-Tarski Fixed-Point Theorem [85] given the fact that the depths of the lattices are fixed constants not related to M. A formal proof can be found in Supplementary Material B. Since in each iteration, t_i only changes from one partition type to another on the lattice in one direction, this algorithm is efficient in inferring the partition types. Experimental results in §8.2 demonstrate its overhead is negligible.

It is worth noting that Algorithm 2 can also be applied for streaming type propagation. Due to space constraints, we do not provide the full details here.

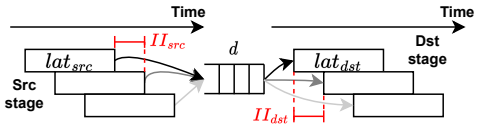
6.5 Holistic Optimization

To achieve high-performance spatial architecture, functions are interconnected using FIFOs, forming distinct dataflow stages. While this creates a functional architecture, dataflow may suffer from performance issues, particularly when there are data stalls. HLS alone cannot determine the ideal FIFO size between stages. Dataflow stalls can arise from two primary situations: (1) when the production rate exceeds the consumption rate, potentially filling the FIFO and causing stalling, (2) or when the production rate is slower, leading to starvation in subsequent stages. Therefore, determining appropriate FIFO sizes is crucial for high performance.

We formulate the problem as follows. Suppose the source stage can generate C_{src} outputs per II_{src} cycles, where II_{src} is the initiation interval of the previous stage. The destination stage demands C_{dst} inputs per II_{dst} cycles for computation. The communication volume between the two stages is denoted as V. We have functions, f_{prod} (production rate) and f_{con} (consumption rate), that track the number of samples generated and consumed at any given time t. If there is no data in the FIFO, the consequential stage cannot perform computation since it does not receive any data. Therefore, the consumption rate is always smaller or equal to the production rate.

$$f_{prod}(t) = \begin{cases} C_{src}\lfloor t/II_{src} \rfloor & t \leq V/C_{src}II_{src} \\ V & t > V/C_{src}II_{src} \end{cases} \quad (2)$$

$$f_{con}(t) = \max\left(C_{dst}\lfloor t/II_{dst} \rfloor, f_{prod}(t)\right) \quad (3)$$



Therefore, the FIFO depth between the source and destination stages can be calculated as:

$$d = \max_{t} \left(f_{prod}(t) - f_{con}(t) + 1 \right), \quad t \in \left[0, \arg\min_{t'} \operatorname{abs} \left(V - \sum_{t'} f_{con}(t) \right) \right], \tag{4}$$

where the maximum t represents the time required to receive all inputs. II_{src} and II_{dst} can be obtained by running a high-level synthesis process.

This method is effective for optimizing a single connection between two stages. However, in cases involving multiple stages, the production rate of the previous stages may influence all subsequent stages. Based on Equation 3, let $f_{con}(t) = f_{prod}(t)$, we can obtain the new $II'_{dst} = II_{src}C_{dst}/C_{src}$. This information is then propagated through the dataflow graph from the top down. In cases where multiple producers feed into a single consumer, the resulting II is the maximum of them, as the slowest stage dictates the overall pace. Thus, we can determine the proper FIFO sizes for the entire dataflow graph. Notably, this optimization only eliminates the first type of dataflow stalling but does not address potential design issues inside a kernel that might lead to a second type of stalling.

7 IMPLEMENTATION

Allo is implemented with 9K lines of Python code for the frontend ADL and 10K lines of C++ code for the MLIR dialect and backend code generation. In this section, we provide the implementation details of the frontend, type system, and codegen.

Frontend. Allo supports both imperative and declarative programming. Unlike several schedule languages [11, 49] that rely on tracing-based techniques to generate AST and the corresponding IR, Allo utilizes Python's AST that provides the ability to handle control flow effectively. Therefore, Allo can seamlessly accommodate the latest Python language features and remain compatible with the vast Python library ecosystem. Consequently, Allo kernels can be executed with the native Python runtime to verify functional correctness, with minimal effort required to migrate a Python program into the Allo representation.

To accommodate larger designs, such as neural networks, Allo offers direct support for importing vision models from TorchVision [55] and language models from HuggingFace [93] to achieve maximum flexibility. Allo here serves as an intermediate language, showing its generality for hardware accelerator design. We provide a backend for TorchDynamo [73] in PyTorch 2.0 [2], so users can call `torch.compile(model,"allo")` to invoke the Allo compiler. We employ `torch.fx` [76] as the high-level IR and translate each PyTorch operator into a library function call within Allo. PyTorch-level optimizations (e.g., operator fusion) are orthogonal to Allo's optimization. As long as the model can be represented in `torch.fx`, Allo can take in and perform hardware-specific customizations. Compared to writing Allo kernels in Python, this approach eliminates the need for users to rewrite the model and construct the schedule themselves. The PyTorch frontend further simplifies programming by allowing users to directly import a model and utilize the high-performance Allo schedule out-of-the-box. As our IR is constructed on top of MLIR, we also plan to support other frontends within the MLIR ecosystem [22, 58, 88] in the future.

Type System. Allo is equipped with a type inference engine designed to manage both built-in and custom data types. The Allo type system differs from the Python native one, as it includes arbitrary bitwidth integers, fixed-point types, and additional shape information in the type hints. Allo's type system consistently prevents overflow for any-bitwidth integers and fixed-point numbers, promoting data types with larger bitwidths when necessary. Based on the predefined typing rules, the type inference engine starts from the annotations at the top-level function and tries to infer the data types of each inner variable. In cases where the inferred data type deviates from the user's

annotations, the engine attempts automatic type conversion if it is deemed feasible. Furthermore, Allo incorporates shape information within the type declaration, facilitating shape inference, and thus supporting array slicing and automatic broadcasting.

Code Generation. After customizing the program, users can call `s.build(<target>)` to generate a valid program for CPU simulation or FPGA bitstream. For the CPU backend, Allo lowers custom operations and data types to LLVM IR and uses the Just-in-Time execution engine [53] to run the program. For the FPGA backend, Allo generates HLS C++ code for AMD Vivado/Vitis HLS [100]. Since these tools accept programs written in C/C++, Allo directly generates code from the `affine` and `memref` dialects, bypassing the need for further lowering to lower-level dialects. Annotated attributes such as pipelining and unrolling are converted into HLS pragmas during code generation.

8 EXPERIMENTS

In this section, we first present our experiment settings and evaluate Allo against several baselines on a comprehensive benchmark and large neural network models.

8.1 Experiment Settings

For single-kernel evaluation, we compare Allo with ScaleHLS [102], PyLog [40], HeteroCL [49], Merlin [101], and Dahlia [59], all of which generate HLS C++ code as output. They represent the state-of-the-art ADLs and compilers that are publicly available. We evaluate them on PolyBench [69], a C-based benchmark suite consisting of commonly used kernels in scientific computing. All experiments use the standard medium problem size and `float32` data types.

For multi-kernel evaluation, we evaluate three different convolutional neural networks (CNNs): ResNet-18 [34], VGG [80], and MobileNet [39]. These models are implemented in PyTorch [65] and imported from the TorchVision [55] library. We run model inference and compare the results with ScaleHLS, which is the only frontend providing direct model import from PyTorch. Other ADLs listed in Table 1 do not provide Python bindings [26, 47, 86] and do not generate HLS C++ code for backend synthesis. Thus, it is challenging to reimplement these designs in their input languages, especially for large deep neural networks, making a fair comparison difficult.

To demonstrate the practical feasibility of Allo in generating large-scale designs running on real hardware, we implement an accelerator for the GPT2 [74] model, the only open-sourced model in the GPT family. GPT2 is a Transformer-based, decoder-only architecture widely used in text generation tasks, with 355M parameters, 24 hidden layers, 16 heads in the attention module, and a hidden size of 1024. We quantize the model into 4-bit weight and 8-bit activation (W4A8) for efficient deployment [30, 95, 106], and verify the results against the quantized model in PyTorch to maintain accuracy. We run backend synthesis for the design generated by Allo and deploy the bitstream on an FPGA. For accelerators of such scale, all of the baseline ADLs fail to generate valid designs that satisfy the resource constraint. Even when attempts are made to reduce the size of these designs, they still run into errors in the routing stage due to excessive memory access, leading to lengthy on-board wires. Consequently, we directly compare Allo with DFX [38], a state-of-the-art Transformer accelerator written in SystemVerilog. We further compare the accelerator with two GPU devices, the NVIDIA GeForce GTX 1080Ti GPU, a widely-used commercial GPU, and the NVIDIA Tesla A100 GPU, a high-end GPU commonly employed for large-scale model training and inference. Note that GPU requires "fake" quantization, so the actual low-bit performance is lower than the optimized fp16 performance, especially for models with less than one billion parameters [21]. Therefore, we report the best fp16 performance for GPUs in our experiments.

All the experiments target the AMD Alveo U280 FPGA using Vitis HLS v2022.1 [100]. The U280 FPGA has 4032 BRAM 18K blocks, 9024 DSP slices, 2.6M flip-flops, 1.3M LUTs, and 960 URAM

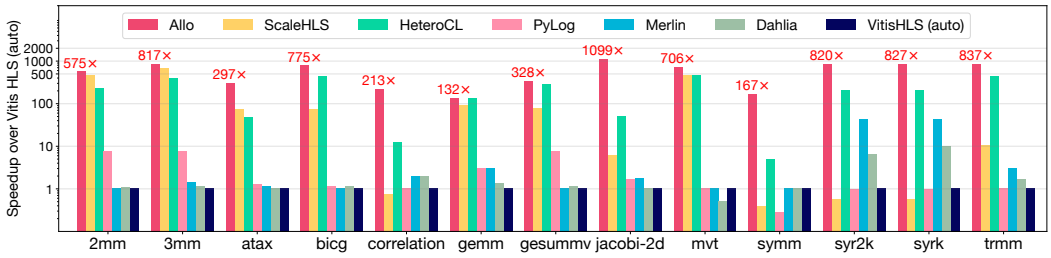


Fig. 12. PolyBench latency speedup over Vitis HLS baseline.

blocks. Allo generates the host program for Vitis in OpenCL. AMD Xilinx RunTime (XRT) and Xilinx Board Utility (xbutil) are used for hardware execution and power measurements. We set the frequency as 300 MHz for high-level synthesis and 250 MHz for on-board evaluation, since the large designs may have routing issues when we further increase the frequency.

8.2 Single-Kernel Evaluation

Fig. 12 shows the latency speedup against the Vitis HLS baseline for multiple ADLs. For Vitis HLS, we do not apply any pragma or hardware customizations other than the default automatic loop pipelining. Merlin automatically inserts pragmas. ScaleHLS searches for the best optimizations from the Pareto-optimal frontier with a design space exploration (DSE) engine. We manually design the best optimization schemes with the available customization primitives for Allo, HeteroCL, PyLog, and Dahlia. Although PyLog provides automatic pragma insertion, we find manual customizations achieve better performance. HeteroCL features kernel-level decoupled customizations with a lack of support for inter-kernel optimizations. PyLog has a limited set of customization primitives compared to HeteroCL. For example, PyLog supports compute customizations such as loop unroll, pipeline, reorder, and tiling, but cannot build a custom memory hierarchy. Dahlia focuses on the predictability of HLS results instead of performance optimizations. For instance, Dahlia guarantees consistent memory banking and loop unrolling factors but lacks important support for loop pipelining. Allo delivers up to 1099× latency speedup over Vitis HLS, 1478× over ScaleHLS, 34× over HeteroCL, 837× over PyLog, 775× over Merlin, and 1405× over Dahlia across different cases.

Table 3. Results comparison between Allo and ScaleHLS — II denotes the loop pipeline initiation interval. We report the worst II for designs with multiple pipelined loops. We compare the clock frequency after placement and routing (PnR). The compilation time only includes the time to generate the HLS code.

Benchmark	Allo						ScaleHLS				
	Latency (cycles)	II	DSP Usage	PnR Freq. (MHz)	Lines of Allo Custm.	Compile Time (s)	Latency (cycles)	II	DSP Usage	PnR Freq. (MHz)	Compile Time (s)
atax	4.9K (↓ 3.9×)	1	403 (↑ 2.9×)	411	9	1.0	19.4K	4	141	329	36.1
correlation	498.7K (↓ 290.5×)	1	4168 (↑ 38.2×)	362	19	0.8	144.9M	667	109	305	638.8
jacobi-2d	58.8K (↓ 183.1×)	1	3968 (↑ 72.1×)	411	17	0.9	10.8M	28	55	308	47.9
symm	405.7K (↓ 427.4×)	1	1208 (↑ 201.3×)	402	15	1.0	182.4M	13	6	397	3.5
trmm	492.6K (↓ 78.0×)	1	101 (↑ 14.4×)	414	12	0.8	38.4M	4	7	382	1.4

We select five designs from the PolyBench suite where Allo outperforms ScaleHLS by a significant margin. From Table 3, we see although ScaleHLS uses an automatic design-space exploration (DSE) engine to search for the best optimizations, the pipeline II of the DSE results is still high. A high pipeline II hurts the design performance in two ways: (1) the overall latency increases, and (2) HLS generates large multiplexers due to DSP reuse. These large multiplexers become critical paths and degrade the clock frequency. The frequency deterioration is especially evident for `atax`. ScaleHLS cannot fully pipeline the designs because of loop-carried dependency, excessive memory access, and

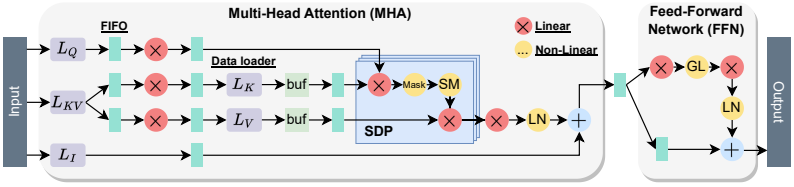


Fig. 13. A spatial architecture for the GPT2 model generated by Allo.

complex nested loop structures of a monolithic kernel. For instance, `atax` has two matrix-vector products with loop-carried dependency. We implement the row-wise product discussed in §3.1 with Allo to break the loop-carried dependency and pipeline the inner loop to II=1. `jacobi-2d` is a typical stencil kernel with a sliding window data access pattern. ScaleHLS cannot fully pipeline such cases because it does not support data reuse. Allo builds two-level reuse buffers which drastically reduces off-chip memory access and fully pipelines the design. For `correlation`, `symm`, and `trmm`, Allo composes smaller kernels each customized with the aforementioned optimizations and forms a dataflow-pipelined design. The scale factor of DSP usage and the latency speedup are marked blue in Table 3. The fully pipelined designs customized with Allo deliver higher performance per DSP while attaining higher clock frequencies. In addition, Allo improves accelerator design productivity with fewer lines of code and short compilation time. The decoupled customizations and the kernel composition are expressed succinctly in less than 20 lines of code with Allo primitives. We present full examples of customized `gemm` and `jacobi-2d` designs in Supplementary Material C.

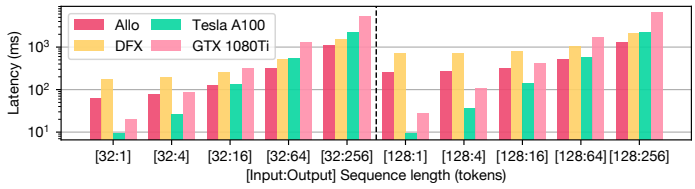
8.3 Multi-Kernel Evaluation

8.3.1 Convolutional Neural Networks. Table 4 shows the results for CNNs. Allo can achieve up to 12.7× speedup compared to ScaleHLS while remaining smaller resource usage in most cases. This is because ScaleHLS does not perform kernel fusion and lacks support for reuse buffers. Allo incorporates these optimizations to enhance data locality significantly. Additionally, ScaleHLS falls short in fully exploiting the inter-kernel dataflow of designs, missing opportunities for optimization with FIFOs. In contrast, Allo excels in efficiently leveraging dataflow characteristics, resulting in improved performance and resource utilization. Our evaluations showcase the proficiency of Allo in composing diverse kernels to form a hierarchical structure and optimizing their orchestration, presenting a great advantage over the challenges encountered by ScaleHLS.

Table 4. Allo and ScaleHLS per-sample latency and FPGA resource usage on CNN models — Vitis HLS automatically implements the Allo buffers using LUTs thus leading to zero BRAM usage.

Benchmark	Allo						ScaleHLS				
	Latency (cycles)	FF	LUT	BRAM	DSP	Speedup	Latency (cycles)	FF	LUT	BRAM	DSP
VGG16	3.85M	36K	98K	0	440	7.4×	28.31M	100K	714K	3936	882
MobileNet	0.26M	57K	128K	0	1942	8.3×	2.17M	93K	518K	6796	1778
ResNet18	8.29M	51K	124K	0	652	12.7×	104.88M	144K	992K	8416	1330

8.3.2 Large Language Models. The accelerator architecture for the GPT2 model is depicted in Fig. 13. A typical Transformer block consists of an MHA module that leverages scaled dot-product (SDP) to calculate the attention score and an FFN that cascades two linear layers. LLMs like GPT2 are inherently more complex than CNNs, both in terms of model sizes and the intricate connections within the MHA module that require splitting the heads and merging the results at the end. Existing ADLs and compilers often lack explicit memory and dataflow management [40, 49, 102]. Therefore,



	Allo	DFX
Device	U280	U280
Freq.	250MHz	200MHz
Quant.	W4A8	fp16
BRAM	384 (19.0%)	1192 (59.1%)
DSP	1780 (19.73%)	3533 (39.2%)
FF	652K (25.0%)	1107K (42.5%)
LUT	508K (39.0%)	520K (39.9%)

Fig. 14. Left: End-to-end latency on GPT2. Right: Resource utilization of Allo and DFX.

the designs they generate often demand significantly more resources – exceeding twice the on-chip resources available. In contrast, Allo enables the design and optimization of each submodule individually, providing control over memory hierarchy and data orchestration strategies. After verifying their correctness, these submodules can be composed bottom-up using the `.compose()` primitive, and later connected with `.relay()` primitive to form a complete design.

In this experiment, we consider the generative inference scenario for LLMs in a single-batch low-latency setting [9]. We adjust the input and output sequence lengths and measure the end-to-end latency from kernel launch, including the CPU-GPU/FPGA communication time. As depicted in Fig. 14, Allo consistently outperforms the state-of-the-art accelerator DFX, achieving up to 2.80× speedup in terms of latency. This is primarily attributed to the highly customized high-performance systolic array kernels and the efficient composition of multiple kernels within Allo. Allo also utilizes fewer resources than DFX from the right side of Fig. 14. This is because the spatial architecture designed in Allo maximally reduces on-chip intermediate buffers. In contrast, DFX employs an overlay design that reuses hardware units for various operators, thereby increasing resource utilization. Thus, we can achieve a higher frequency but use fewer resources than DFX. Notably, the GPT2 model is directly imported using the PyTorch frontend (§7), which does not require users to rewrite any code for the model. The accelerator fully leverages the Allo customization primitives with less than 50 lines of customization code to optimize this intricate design, which is much more productive than implementing it in a hardware description language (HDL).

Furthermore, we extend our performance evaluation to compare with two GPU devices. The FPGA-based design exhibits a notable performance gap compared to GPUs when the output sequence length is small. This discrepancy is primarily due to the extensive compute requirements during the initial stage of generative inference, known as the prefill stage, which aligns more effectively with throughput-oriented devices such as GPUs [9, 67]. However, as the output sequence length increases, the FPGA accelerator generated by Allo surpasses GPU performance. In particular, we achieve a 5.05× speedup compared to the 1080Ti GPU. Even in comparison to a high-end A100 GPU, we still attain a 1.70× speedup for longer output sequences. Additionally, the FPGA execution requires only 30 watts of measured power, whereas the A100 counterpart demands 96 watts, which means the Allo accelerator is 5.44× more energy-efficient than the A100 GPU.

9 RELATED WORK

Schedule Languages. Halide [75] first introduces the concept of algorithm and schedule decoupling in the domain of image processing. TVM [11] extends this idea to deep learning and supports end-to-end optimization workflow mapping neural network models to different hardware devices. There are also other DSLs that leverage schedule languages to generate high-performance code in their specific domains [5, 7, 8, 33, 45, 49, 89, 103]. Allo also adopts this decoupling idea but further enhances the composability of customizations and schedules. It leverages a hierarchical dataflow graph to compose smaller designs into larger ones, achieving high performance on large designs.

Recent developments explore program rewriting techniques as an alternative to the traditional schedule tree, making it possible to handle more imperative programs. For instance, TensorIR [29]

extends TVM to support a more flexible syntax for describing computations, enabling better tensorization for TensorCore on GPUs. Exo [41] formalizes program rewrite rules using an effect system to guarantee the correctness of transformations. The `xform` dialect [108] in MLIR also supports rewrites to the programs. While these efforts mainly focus on kernel-level optimizations, they lack the capability to effectively compose optimizations across multiple kernels, limiting their scalability to larger and more complex designs.

Accelerator Design Languages (ADLs) and Compilers. Numerous domain-specific languages (DSLs) have emerged to facilitate hardware designs for different applications [25, 35, 36, 54, 57, 63, 77, 79, 84, 91], providing highly optimized operators tailored for specific domains. Subsequently, various ADLs have been introduced to address more general-purpose accelerator design [6, 26, 40, 46, 47, 86]. However, their algorithm and customizations are entangled together, leading to reduced productivity and limited exploration of different customization combinations. HeteroCL [49] decouples hardware customizations from the algorithm but primarily focuses on single-kernel designs. Several ADLs also emphasize dataflow optimizations [6, 26, 47, 86, 94], yet they struggle to preserve the hierarchical structure of dataflow and cannot efficiently compose small kernels into larger designs. Consequently, they encounter challenges when scaling to accommodate large and complex models.

Recent efforts harness the MLIR toolchain to generate C/C++ HLS code [1, 102, 105]. However, the existing compiler passes and design space exploration (DSE) engines often fall short in producing high-performance accelerators, as showcased in Section 8.2. This is primarily because MLIR lacks inherent support for crucial components like quantized data types, memory, and dataflow customizations. Lastly, low-level hardware design languages (HDLs) such as Calyx [61] are designed to facilitate the process of backend synthesis. Filament [60] is also a low-level HDL that leverages timeline types to reason about timing safety. These efforts are orthogonal to ours, and we plan to support the CIRCT [14] project as a backend in the future.

10 CONCLUSION AND FUTURE WORK

In this paper, we propose Allo, a composable programming model for accelerator design. Allo proposes progressive hardware customizations allowing users to apply provable program transformations step by step, and further introduces composable schedules to combine small kernels into large designs. Nonetheless, there are several unexplored directions in our ongoing work.

For optimizations within a kernel, we plan to design an autoscheduler that can reduce the programming efforts required from developers. For composing multiple kernels, the techniques proposed in §6.5 only address FIFO sizing but do not determine where to establish these connections automatically. Some kernels may have dependency relations that prevent direct connection with FIFOs and might require additional buffers to ensure sequential memory access. We plan to develop automatic bufferization techniques to create buffers between stages and guarantee correctness.

Practical hardware design entails more than just customizing and transforming code; it also involves connecting components and guiding them through the entire backend synthesis process to generate a bitstream. Mapping dataflow regions onto multi-die FPGA boards can be challenging. Several large designs we experimented with for LLMs failed to meet timing requirements, often due to issues during the routing stage. In order to improve the frequency of the design, it is essential to explicitly bind dataflow regions to specific hardware regions and minimize cross-die communication. Although efforts such as AutoBridge [32] aim to decompose designs into smaller parts and assemble them, they cannot accommodate complex hierarchical dataflow or create double buffers. We plan to create a build system that can compile the entire design in parallel and efficiently link the components together, similar to how software linkers work. This approach will help optimize the hardware design process further and enhance performance.

ACKNOWLEDGMENTS

This work was supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA and NSF Awards #2019306 and #2118709, and an Intel ISRA award. We gratefully acknowledge the anonymous reviewers, Prof. Adrian Sampson, Prof. Louis-Noël Pouchet, Rachit Nigram, and Jialu Bao for their valuable feedback on the initial draft of this work. We thank Jiahao Zhang for providing a reference LLM accelerator implementation. We also thank Jin Yang, Jeremy Casas, and Zhenkun Yang for their insightful feedback on the initial version of the Allo framework.

ARTIFACT

The Allo code is open-source and available at the allo repository on GitHub. Detailed instructions for reproducibility and reusability are provided in an archived version on Zenodo [10] and at the allo-pldi24-artifact repository on GitHub.

REFERENCES

[1] Nicolas Bohm Agostini, Serena Curzel, Jeff Jun Zhang, Ankur Limaye, Cheng Tan, Vinay Amatya, Marco Minutoli, Vito Giovanni Castellana, Joseph Manzano, David Brooks, Gu-Yeon Wei, and Antonino Tumeo. 2022. Bridging Python to Silicon: The SODA Toolchain. *IEEE Micro* 42, 5 (2022), 78–88. https://doi.org/10.1109/MM.2022.3178580

[2] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, et al. 2024. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'24)*. Association for Computing Machinery, New York, NY, USA, 317–335.

[3] AWS. 2023. Inferentia Architecture. https://awsdocs-neuron.readthedocs-hosted.com/en/latest/general/arch/neuron-hardware/inferentia.html.

[4] 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*. Association for Computing Machinery, New York, NY, USA, 1216–1225. https://doi.org/10.1145/2228360.2228584

[5] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization* (Washington, DC, USA) *(CGO'19)*. IEEE Press, 193–205.

[6] 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* (Denver, Colorado) *(SC'19)*. Association for Computing Machinery, New York, NY, USA, Article 81, 14 pages. https://doi.org/10.1145/3295500.3356173

[7] Hongzheng Chen, Minghua Shen, Nong Xiao, and Yutong Lu. 2021. Krill: A Compiler and Runtime System for Concurrent Graph Processing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'21)*. Association for Computing Machinery, New York, NY, USA, Article 51, 16 pages. https://doi.org/10.1145/3458817.3476159

[8] Hongzheng Chen, Cody Hao Yu, Shuai Zheng, Zhen Zhang, Zhiru Zhang, and Yida Wang. 2024. Slapo: A Schedule Language for Progressive Optimization of Large Deep Learning Model Training. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS'24)*. Association for Computing Machinery, New York, NY, USA.

[9] Hongzheng Chen, Jiahao Zhang, Yixiao Du, Shaojie Xiang, Zichao Yue, Niansong Zhang, Yaohui Cai, and Zhiru Zhang. 2024. Understanding the Potential of FPGA-Based Spatial Acceleration for Large Language Model Inference. *ACM Trans. Reconfigurable Technol. Syst.* (2024).

[10] Hongzheng Chen, Niansong Zhang, Shaojie Xiang, Zhichen Zeng, Mengjia Dai, and Zhiru Zhang. 2024. Artifact for *Allo: A Programming Model for Composable Accelerator Design*. https://doi.org/10.5281/zenodo.10961342.

[11] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 579–594.

[12] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to Optimize Tensor Programs. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (Montréal, Canada) *(NIPS'18)*. Curran Associates Inc., Red Hook, NY, USA, 3393–3404.

[13] 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)*. Institute of Electrical and Electronic Engineers, New York, NY, USA, 1–8. https://doi.org/10.1145/3240765.3240850

[14] CIRCT. 2024. CIRCT: Circuit IR Compilers and Tools. https://github.com/llvm/circt.

[15] Jason Cong, Jason Lau, Gai Liu, Stephen Neuendorffer, Peichen Pan, Kees Vissers, and Zhiru Zhang. 2022. FPGA HLS Today: Successes, Challenges, and Opportunities. *ACM Trans. Reconfigurable Technol. Syst.* 15, 4, Article 51 (aug 2022), 42 pages. https://doi.org/10.1145/3530775

[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. https://doi.org/10.1109/TCAD.2011.2110592

[17] Jason Cong and Jie Wang. 2018. PolySA: Polyhedral-Based Systolic Array Auto-Compilation. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. Institute of Electrical and Electronic Engineers, New York, NY, USA, 1–8. https://doi.org/10.1145/3240765.3240838

[18] Luis Damas. 1984. *Type assignment in programming languages.* Ph. D. Dissertation. University of Edinburgh.

[19] Luis Damas and Robin Milner. 1982. Principal Type-Schemes for Functional Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Albuquerque, New Mexico) *(POPL'82)*. Association for Computing Machinery, New York, NY, USA, 207–212. https://doi.org/10.1145/582153.582176

[20] Johannes de Fine Licht, Maciej Besta, Simon Meierhans, and Torsten Hoefler. 2021. Transformations of High-Level Synthesis Codes for High-Performance Computing. *IEEE Trans. Parallel Distrib. Syst.* 32, 5 (may 2021), 1014–1029. https://doi.org/10.1109/TPDS.2020.3039409

[21] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2022. LLM.int8 (): 8-bit Matrix Multiplication for Transformers at Scale. *Advances in Neural Information Processing Systems* 35 (2022), 30318–30332.

[22] IREE Developers. 2022. IREE (Intermediate Representation Execution Environment. https://google.github.io/iree/

[23] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-Training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint arXiv:1810.04805* (2018).

[24] Stephen Dolan and Alan Mycroft. 2017. Polymorphism, Subtyping, and Type Inference in MLsub. *SIGPLAN Not.* 52, 1 (jan 2017), 60–72. https://doi.org/10.1145/3093333.3009882

[25] Javier Duarte, Song Han, Philip Harris, Sergo Jindariani, Edward Kreinar, Benjamin Kreis, Jennifer Ngadiuba, Maurizio Pierini, Ryan Rivera, Nhan Tran, et al. 2018. Fast Inference of Deep Neural Networks in FPGAs for Particle Physics. *Journal of instrumentation* 13, 07 (2018), P07027.

[26] 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* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 408–422. https://doi.org/10.1145/3385412.3385983

[27] Murali Emani, Venkatram Vishwanath, Corey Adams, Michael E. Papka, Rick Stevens, Laura Florescu, Sumti Jairath, William Liu, Tejas Nama, and Arvind Sujeeth. 2021. Accelerating Scientific Applications With SambaNova Reconfigurable Dataflow Architecture. *Computing in Science & Engineering* 23, 2 (2021), 114–119. https://doi.org/10.1109/MCSE.2021.3057203

[28] Farah Fahim, Benjamin Hawks, Christian Herwig, James Hirschauer, Sergo Jindariani, Nhan Tran, et al. 2021. hls4ml: An Open-Source Codesign Workflow to Empower Scientific Low-Power Machine Learning Devices.

[29] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, and Tianqi Chen. 2023. TensorIR: An Abstraction for Automatic Tensorized Program Optimization. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 804–817. https://doi.org/10.1145/3575693.3576933

[30] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. 2022. GPTQ: Accurate Post-training Compression for Generative Pretrained Transformers. *arXiv preprint arXiv:2210.17323* (2022).

[31] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, et al. 2021. Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration. In *2021 58th ACM/IEEE Design Automation Conference (DAC)* (San Francisco, CA, USA). IEEE Press, 769–774. https://doi.org/10.1109/DAC18074.2021.9586216

[32] 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* (Virtual Event, USA)

(FPGA'21). Association for Computing Machinery, New York, NY, USA, 81–92. https://doi.org/10.1145/3431920.3439289

[33] Bastian Hagedorn, Archibald Samuel Elliott, Henrik Barthels, Rastislav Bodik, and Vinod Grover. 2020. Fireiron: A Data-Movement-Aware Scheduling Language for GPUs. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques* (Virtual Event, GA, USA) *(PACT'20)*. Association for Computing Machinery, New York, NY, USA, 71–82. https://doi.org/10.1145/3410463.3414632

[34] 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.

[35] James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. 2014. Darkroom: Compiling High-Level Image Processing Code into Hardware Pipelines. *ACM Trans. Graph.* 33, 4, Article 144 (jul 2014), 11 pages. https://doi.org/10.1145/2601097.2601174

[36] James Hegarty, Ross Daly, Zachary DeVito, Jonathan Ragan-Kelley, Mark Horowitz, and Pat Hanrahan. 2016. Rigel: Flexible Multi-Rate Image Processing Hardware. *ACM Trans. Graph.* 35, 4, Article 85 (jul 2016), 11 pages. https://doi.org/10.1145/2897824.2925892

[37] R. Hindley. 1969. The Principal Type-Scheme of an Object in Combinatory Logic. *Trans. Amer. Math. Soc.* 146 (1969), 29–60. http://www.jstor.org/stable/1995158

[38] Seongmin Hong, Seungjae Moon, Junsoo Kim, Sungjae Lee, Minsub Kim, Dongsoo Lee, and Joo-Young Kim. 2022. DFX: A Low-latency Multi-FPGA Appliance for Accelerating Transformer-based Text Generation. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 616–630. https://doi.org/10.1109/MICRO56248.2022.00051

[39] 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).

[40] 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. https://doi.org/10.1109/TC.2021.3123465

[41] 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* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 703–718. https://doi.org/10.1145/3519939.3523446

[42] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, et al. 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. https://doi.org/10.1145/3579371.3589350

[43] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA'17)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3079856.3080246

[44] Gary A. Kildall. 1973. A Unified Approach to Global Program Optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Boston, Massachusetts) *(POPL'73)*. Association for Computing Machinery, New York, NY, USA, 194–206. https://doi.org/10.1145/512927.512945

[45] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 77 (oct 2017), 29 pages. https://doi.org/10.1145/3133901

[46] David Koeplinger, Christina Delimitrou, Raghu Prabhakar, Christos Kozyrakis, Yaqi Zhang, and Kunle Olukotun. 2016. Automatic Generation of Efficient Accelerators for Reconfigurable Hardware. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Seoul, Republic of Korea) *(ISCA'16)*. IEEE Press, 115–127. https://doi.org/10.1109/ISCA.2016.20

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

[48] H. T. Kung and Charles E. Leiserson. 1978. Systolic Arrays for (VLSI). https://api.semanticscholar.org/CorpusID:60531591

[49] 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. https://doi.org/10.1145/3289602.3293910

[50] Yi-Hsiang Lai, Hongbo Rong, Size Zheng, Weihao Zhang, Xiuping Cui, Yunshan Jia, et al. 2020. SuSy: A Programming Model for Productive Construction of High-Performance Systolic Arrays on FPGAs. In *Proceedings of the 39th International Conference on Computer-Aided Design* (Virtual Event, USA) *(ICCAD '20)*. Association for Computing Machinery, New York, NY, USA, Article 73, 9 pages. https://doi.org/10.1145/3400302.3415644

[51] Yi-Hsiang Lai, Ecenur Ustun, Shaojie Xiang, Zhenman Fang, Hongbo Rong, and Zhiru Zhang. 2021. Programming and Synthesis for Software-Defined FPGA Acceleration: Status and Future Prospects. *ACM Trans. Reconfigurable Technol. Syst.* 14, 4, Article 17 (sep 2021), 39 pages. https://doi.org/10.1145/3469660

[52] 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.

[53] 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 *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization* (Virtual Event, Republic of Korea) *(CGO '21)*. IEEE Press, 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[54] Jiajie Li, Yuze Chi, and Jason Cong. 2020. HeteroHalide: From Image Processing DSL to Efficient FPGA Acceleration. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) *(FPGA'20)*. Association for Computing Machinery, New York, NY, USA, 51–57. https://doi.org/10.1145/3373087.3375320

[55] TorchVision maintainers and contributors. 2016. TorchVision: PyTorch's Computer Vision library. https://github.com/pytorch/vision.

[56] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. 2018. NVIDIA Tensor Core Programmability, Performance & Precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 522–531. https://doi.org/10.1109/IPDPSW.2018.00091

[57] Peter Milder, Franz Franchetti, James C. Hoe, and Markus Püschel. 2012. Computer Generation of Hardware for Linear Digital Signal Processing Transforms. *ACM Trans. Des. Autom. Electron. Syst.* 17, 2, Article 15 (apr 2012), 33 pages. https://doi.org/10.1145/2159542.2159547

[58] 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)*. 45–59. https://doi.org/10.1109/PACT52795.2021.00011

[59] 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* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 393–407. https://doi.org/10.1145/3385412.3385974

[60] Rachit Nigam, Pedro Henrique Azevedo de Amorim, and Adrian Sampson. 2023. Modular Hardware Design with Timeline Types. *Proc. ACM Program. Lang.* 7, PLDI, Article 120 (jun 2023), 25 pages. https://doi.org/10.1145/3591234

[61] 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* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 804–817. https://doi.org/10.1145/3445814.3446712

[62] OpenAI. 2023. GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774* (2023).

[63] Debjit Pal, Yi-Hsiang Lai, Shaojie Xiang, Niansong Zhang, Hongzheng Chen, Jeremy Casas, Pasquale Cocchini, Zhenkun Yang, Jin Yang, Louis-Noël Pouchet, and Zhiru Zhang. 2022. Accelerator Design with Decoupled Hardware Customizations: Benefits and Challenges: Invited. In *Proceedings of the 59th ACM/IEEE Design Automation Conference* (San Francisco, California) *(DAC '22)*. Association for Computing Machinery, New York, NY, USA, 1351–1354. https://doi.org/10.1145/3489517.3530681

[64] Lionel Parreaux. 2020. The Simple Essence of Algebraic Subtyping: Principal Type Inference with Subtyping Made Easy (Functional Pearl). *Proc. ACM Program. Lang.* 4, ICFP, Article 124 (aug 2020), 28 pages. https://doi.org/10.1145/3409006

[65] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, et al. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. IEEE Press, New York, NY, USA, 172–198.

[66] Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, and Rastislav Bodik. 2014. Chlorophyll: Synthesis-Aided Compiler for Low-Power Spatial Architectures. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 396–407. https://doi.org/10.1145/2594291.2594339

[67] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2023. Efficiently Scaling Transformer Inference. In *Proceedings of Machine Learning and Systems*, Vol. 5.

[68] François Pottier. 1998. *Type Inference in the Presence of Subtyping: From Theory to Practice.* Ph. D. Dissertation. INRIA.

[69] Louis-Noël Pouchet et al. 2012. Polybench: The polyhedral benchmark suite. http://www.cs.ucla.edu/pouchet/software/polybench

[70] Louis-Noël Pouchet, Emily Tucker, Niansong Zhang, Hongzheng Chen, Debjit Pal, Gabriel Rodríguez, and Zhiru Zhang. 2024. Formal Verification of Source-to-Source Transformations for HLS. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA'24)*. Association for Computing Machinery, New York, NY, USA, 97–107. https://doi.org/10.1145/3626202.3637563

[71] Louis-Noël Pouchet, Peng Zhang, P. Sadayappan, and Jason Cong. 2013. Polyhedral-Based Data Reuse Optimization for Configurable Computing. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'13)*. Association for Computing Machinery, New York, NY, USA, 29–38.

[72] PyBind. 2023. PyBind11. https://github.com/pybind/pybind11.

[73] PyTorch. 2022. TorchDynamo Overview. https://pytorch.org/docs/master/dynamo/.

[74] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[75] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. *SIGPLAN Not.* 48, 6 (jun 2013), 519–530. https://doi.org/10.1145/2499370.2462176

[76] James Reed, Zachary DeVito, Horace He, Ansley Ussery, and Jason Ansel. 2022. torch.fx: Practical Program Capture and Transformation for Deep Learning in Python. In *Proceedings of Machine Learning and Systems*, Vol. 4.

[77] Oliver Reiche, M. Akif Özkan, Richard Membarth, Jürgen Teich, and Frank Hannig. 2017. Generating FPGA-based image processing accelerators with Hipacc: (Invited paper). In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1026–1033. https://doi.org/10.1109/ICCAD.2017.8203894

[78] Junru Shao, Xiyou Zhou, Siyuan Feng, Bohan Hou, Ruihang Lai, Hongyi Jin, Wuwei Lin, Masahiro Masuda, Cody Hao Yu, and Tianqi Chen. 2022. Tensor Program Optimization with Probabilistic Programs. In *Advances in Neural Information Processing Systems*.

[79] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. 2016. From High-Level Deep Neural Models to FPGAs. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Institute of Electrical and Electronic Engineers, Taipei, Taiwan, 1–12. https://doi.org/10.1109/MICRO.2016.7783720

[80] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv preprint arXiv:1409.1556* (2014).

[81] Atefeh Sohrabizadeh, Yunsheng Bai, Yizhou Sun, and Jason Cong. 2022. Automated Accelerator Optimization Aided by Graph Neural Networks. In *2022 59th ACM/IEEE Design Automation Conference (DAC)*. Association for Computing Machinery, New York, NY, USA, 55–60.

[82] Atefeh Sohrabizadeh, Cody Hao Yu, Min Gao, and Jason Cong. 2021. AutoDSE: Enabling Software Programmers Design Efficient FPGA Accelerators. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Virtual Event, USA) *(FPGA'21)*. Association for Computing Machinery, New York, NY, USA, 147. https://doi.org/10.1145/3431920.3439464

[83] Nitish Srivastava, Hongbo Rong, Prithayan Barua, Guanyu Feng, Huanqi Cao, Zhiru Zhang, et al. 2019. T2S-Tensor: Productively Generating High-Performance Spatial Hardware for Dense Tensor Computations. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 181–189. https://doi.org/10.1109/FCCM.2019.00033

[84] Robert Stewart, Kirsty Duncan, Greg Michaelson, Paulo Garcia, Deepayan Bhowmik, and Andrew Wallace. 2018. RIPL: A Parallel Image Processing Language for FPGAs. *ACM Trans. Reconfigurable Technol. Syst.* 11, 1, Article 7 (mar 2018), 24 pages. https://doi.org/10.1145/3180481

[85] Alfred Tarski. 1955. A Lattice-Theoretical Fixpoint Theorem and Its Applications. *Pacific J. Math.* 5 (1955), 285–309. https://api.semanticscholar.org/CorpusID:13651629

[86] James Thomas, Pat Hanrahan, and Matei Zaharia. 2020. Fleet: A Framework for Massively Parallel Streaming on FPGAs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 639–651. https://doi.org/10.1145/3373376.3378495

[87] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and Efficient Foundation Language Models. *arXiv preprint arXiv.2302.13971* (2023).

[88] Nicolas Vasilache, Oleksandr Zinenko, Aart JC Bik, Mahesh Ravishankar, Thomas Raoux, Alexander Belyaev, et al. 2022. Composable and Modular Code Generation in MLIR: A Structured and Retargetable Approach to Tensor Compiler Construction. *arXiv preprint arXiv:2202.03293* (2022).

[89] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *arXiv preprint arXiv:1802.04730* (2018).

[90] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

[91] Han Wang, Robert Soulé, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. 2017. P4FPGA: A Rapid Prototyping Framework for P4. In *Proceedings of the Symposium on SDN Research* (Santa Clara, CA, USA) *(SOSR'17)*. Association for Computing Machinery, New York, NY, USA, 122–135. https://doi.org/10.1145/3050220.3050234

[92] 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* (Virtual Event, USA) *(FPGA'21)*. Association for Computing Machinery, New York, NY, USA, 93–104. https://doi.org/10.1145/3431920.3439292

[93] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2019. Huggingface's Transformers: State-of-the-Art Natural Language Processing. *arXiv preprint arXiv:1910.03771* (2019).

[94] 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* (Virtual Event, USA) *(FPGA'22)*. Association for Computing Machinery, New York, NY, USA, 78–88. https://doi.org/10.1145/3490422.3502369

[95] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. 2023. SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models. In *International Conference on Machine Learning*. PMLR, 38087–38099.

[96] AMD Xilinx. 2021. Alveo U280 Data Center Accelerator Card. https://www.xilinx.com/products/boards-and-kits/alveo/u280.html#specifications.

[97] AMD Xilinx. 2022. AI Engines and Their Applications. https://www.xilinx.com/content/dam/xilinx/support/documents/white_papers/wp506-ai-engine.pdf

[98] AMD Xilinx. 2022. Vitis Accelerated Libraries. https://github.com/Xilinx/Vitis_Libraries.

[99] AMD Xilinx. 2022. Vitis AI: Adaptable & Real-Time AI Inference Acceleration. https://github.com/Xilinx/Vitis-AI.

[100] AMD Xilinx. 2022. Vitis HLS v2022.1. https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html.

[101] AMD Xilinx. 2023. Merlin Compiler. https://github.com/Xilinx/merlin-compiler.

[102] 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)*.

[103] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. GraphIt: A High-Performance Graph DSL. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 121 (oct 2018), 30 pages. https://doi.org/10.1145/3276491

[104] 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)*. Insititute of Electrical and Electronic Engineers, Irvine, CA, USA, 430–437. https://doi.org/10.1109/ICCAD.2017.8203809

[105] Ruizhe Zhao, Jianyi Cheng, Wayne Luk, and George A. Constantinides. 2022. POLSCA: Polyhedral High-Level Synthesis with Compiler Transformations. In *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*. Institute of Electrical and Electronic Engineers, Belfast, United Kingdom, 235–242. https://doi.org/10.1109/FPL57034.2022.00044

[106] Yilong Zhao, Chien-Yu Lin, Kan Zhu, Zihao Ye, Lequn Chen, Size Zheng, Luis Ceze, Arvind Krishnamurthy, Tianqi Chen, and Baris Kasikci. 2023. Atom: Low-Bit Quantization for Efficient and Accurate LLM Serving. *arXiv preprint arXiv:2310.19102* (2023).

[107] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 49, 17 pages.

[108] Alex Zinenko. 2022. [RFC] Interfaces and Dialects for Precise IR Transformation Control. https://discourse.llvm.org/t/rfc-interfaces-and-dialects-for-precise-ir-transformation-control/60927

Received 2023-11-16; accepted 2024-03-31