HeteroFlow: An Accelerator Programming Model with Decoupled Data Placement for Software-Defined FPGAs

Shaojie Xiang*, Yi-Hsiang Lai, Yuan Zhou, Hongzheng Chen, Niansong Zhang, Debjit Pal, Zhiru Zhang*
[1] School of Electrical and Computer Engineering, Cornell University, USA
*{sx233,zhiruz}@cornell.edu

ABSTRACT

To achieve high performance with FPGA-equipped heterogeneous compute systems, it is crucial to co-optimize data placement and compute scheduling to maximize data reuse and bandwidth utilization for both on- and off-chip memory accesses. However, optimizing the data placement for FPGA accelerators is a complex task. One must acquire in-depth knowledge of the target FPGA device and its associated memory system in order to apply a set of advanced optimizations. Even with the latest high-level synthesis (HLS) tools, programmers often have to insert many low-level vendor-specific pragmas and substantially restructure the algorithmic code so that the right data are accessed at the right loop level using the right communication schemes. These code changes can significantly compromise the composability and portability of the original program.

To address these challenges, we propose HeteroFlow, an FPGA accelerator programming model that decouples the algorithm specification from optimizations related to orchestrating the placement of data across a customized memory hierarchy. Specifically, we introduce a new primitive named `.to()`, which provides a unified programming interface for specifying data placement optimizations at different levels of granularity: (1) coarse-grained data placement between host and accelerator, (2) medium-grained kernel-level data placement within an accelerator, and (3) fine-grained data placement within a kernel. We build HeteroFlow on top of the open-source HeteroCL DSL and compilation framework. Experimental results on a set of realistic benchmarks show that, programs written in HeteroFlow can match the performance of extensively optimized manual HLS design with much fewer lines of code.

CCS CONCEPTS

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

KEYWORDS

Programming Model; Decoupled Data Placement; DSL

ACM Reference Format:

Shaojie Xiang, Yi-Hsiang Lai, Yuan Zhou, Hongzheng Chen, Niansong Zhang, Debjit Pal, 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 (FPGA '22), February 27-March 1, 2022, Virtual Event, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3490422.3502369

1 INTRODUCTION

FPGA accelerators leverage distributed, specialized datapaths to enable massively parallel and/or deeply pipelined computation on-chip. In comparison, off-chip memory accesses remain a bottleneck in terms of both latency and bandwidth. Hence contemporary accelerators typically allocate significant resources to form customized memory hierarchies that ensure the many parallel compute engines are fed with data at a sufficient rate.

A key distinction from general-purpose computing is that FPGA programmers must leverage the application/domain knowledge to explicitly manage the orchestration of the important data, namely, choosing the "right" type of memory storage to place the "right" granularity of data and moving them in and out at the "right" moment. For example, a recent study shows that carefully managing the data layout and communication schemes of the FPGA accelerators can result in a 3–8× performance improvement on a collection of benchmarks [30]. In this paper, we use the term *data placement* to broadly refer to the programmer-managed control of data orchestration and marshaling across the accelerator's memory hierarchy. A well-designed data placement scheme should act in concert with compute scheduling to maximize data reuse and bandwidth utilization for both on- and off-chip memory accesses.

Optimizing the data placement for FPGA accelerators is by no means an easy task, especially using the conventional RTL-based design methodology. Modern high-level synthesis (HLS) tools improve the programmability of FPGAs by raising the level of design abstraction using software programming languages [11, 26]. However, there are still several major downsides of the current HLS programming models, which often result in less satisfactory performance and repeated engineering effort:

- To achieve a high-performance, programmers must acquire in-depth knowledge of the target FPGA device and its associated memory system before applying a set of optimizations that account for the host-accelerator communications as well as the interactions between different sub-modules within the accelerator.
- Current HLS programming models entangle algorithm descriptions with hardware customization techniques including data placement. HLS users often have to substantially perturb the structure of the (originally algorithmic) code and insert many low-level vendor-specific constructs such as pragmas and library calls. This significantly lowers the readability and portability of the design.
- The contemporary C-based HLS methodology lacks a concise and consistent abstraction for expressing data placement schemes across different levels of the custom memory hierarchy, compromising the design modularity and composability. The lack of explicit programming abstraction of data placement may also hinder effective compiler analyses and optimizations.

Some recent HLS research has proposed end-to-end compilation flow using polyhedral analysis to generate high-performance FPGA accelerators from C/C++ programs in a push-button manner [6, 9, 12, 43]. The high-level goal is to allow programmers to focus on the algorithms, while the compiler automatically explores architecture design space. However, these methods mainly focus on kernel-level compilation,[1] where the compute kernels are restricted to highly regular loop nests such as those commonly seen in systolic algorithms. For example, AutoSA [43] automatically builds systolic arrays from a plain C/C++ program without sophisticated manual annotations or code changes. While it provides autotuning capabilities that explore different data placement schemes (e.g., weight- versus output-stationary), the AutoSA compiler is limited to systolic kernels and it does not offer a programming abstraction to facilitate the integration with other non-systolic portions of the accelerator.

Another active line of work attempts to further raise the abstraction level of FPGA programming by leveraging domain-specific languages (DSLs) and promoting separation of concerns [24, 25, 27, 35, 40]. One recent example is HeteroCL [24], which provides a Python-based embedded DSL and compiler for FPGA accelerator programming. Inspired by Halide [36] and TVM [8], HeteroCL separates an algorithm specification from a temporal compute schedule such as loop reordering and tiling. HeteroCL further decouples the algorithm from on-chip memory customization and data quantization schemes. However, it does not provide programming support for the explicit management of data placement.

In this paper we propose *HeteroFlow*, an FPGA accelerator programming model that supports a data placement specification decoupled from the algorithm description and other hardware customizations. HeteroFlow provides a unified programming interface for customizing: (1) host-accelerator data placement, where a programmer can specify in a concise and portable manner the data schemes between the CPU host memory and the FPGA accelerator (or the device memory associated with the accelerator); (2) inter-kernel data placement, where efficient on-chip data streaming (via FIFO and multi-buffers) can be easily enabled between different compute kernels within an accelerator; (3) intra-kernel data placement, which allows productive yet expressive specifications of various fine-grained dataflow patterns commonly used in spatial architectures such as systolic arrays.

Our main technical contributions are as follows:

- To our knowledge, this is the first work to introduce an FPGA-focused high-level programming model with decoupled data placement specifications. The proposed HeteroFlow approach separates the concerns of algorithmic optimizations from orchestrating the placement of data across a customized memory hierarchy, improving both design productivity and portability.
- Unlike conventional HLS, HeteroFlow provides a unified programming interface named `.to()`, for expressing data placement optimizations at different levels of the design/memory hierarchy (i.e., host-accelerator, kernel-to-kernel, and intra-kernel), resulting in a more modular and composable design specification.
- We extend the open-source HeteroCL framework [24] to implement the `.to()` primitive, enabling programmers to co-optimize data placement schemes with other hardware customization techniques such as tiling and data quantization. With HeteroFlow, a programmer can further leverage `.to()` to seamlessly integrate non-systolic kernels with optimized systolic arrays (either directly specified in HeteroFlow or generated by an existing optimizing compiler [43]).
- We evaluate our proposed framework on a set of realistic benchmarks and show that programs written in HeteroFlow can match the performance of extensively optimized manual HLS design with much fewer lines of code.

[1]Here we use a generic term "kernel" to loosely define a sub-module in the accelerator design that contains a loop nest (or function) with tens to hundreds of operations.

2 BACKGROUND

Data Placement for FPGA Accelerators. When programming an FPGA using HLS, the designer is responsible for orchestrating the placement and movement of data between memory buffers inside the FPGA chip, and between the FPGA and the CPU host. Common methods for on-chip memory management include single buffer, double buffer, and streaming FIFO. CPU-FPGA communication can be realized using DMA engines. On certain platforms, the FPGA is allowed to directly read data from the CPU's host memory or cache.

Inspired by [33], we categorize these data placement methods along two dimensions. The first dimension is whether the requester of data loading is also the consumer of the data. This is common for traditional computational platforms (e.g., CPUs with caches and GPUs with shared scratchpad memories) and we refer to this data access scheme as *coupled access-execute*. In this scenario, data access and computation cannot be performed at the same time. For heterogeneous CPU-FPGA platforms, directly reading from the host memory or performing communication using a single buffer inside the FPGA fall into this category. On the contrary, it is common for one stage of an FPGA accelerator to consume the data in a buffer while a separate stage is storing data into the buffer. This approach is referred to as *decoupled access-execute* (DAE) because data access and execution can be performed in parallel. Loading data from the host using DMA, and performing on-chip communication through FIFO or double buffers fall into this category.

The second dimension is whether the requester of data loading has complete knowledge and control about the exact location of the data in the memory hierarchy. When caches are present in the memory hierarchy, the load initiator only interacts with the first-level cache, and the memory system decides how to transfer the data and where to keep the data. Such a scheme is *implicit* and alleviates the designer's burden of managing the memory hierarchy. However, the area and performance penalty of implicit data orchestration is often too high for hardware accelerators. As a result, FPGA accelerators usually adopt *explicit* data orchestration. For on-chip communication, the accelerator knows the exact location of the data when passing data using FIFOs or buffers. Specifically, loading data from host memory using DMA belongs to explicit data orchestration, because the DMA engine in the accelerator knows the address to the host memory buffer.

Programming with Decoupled Hardware Customizations. The core tenet of a decoupled programming model is to separate the algorithmic description from the specification of target-dependent optimizations (e.g., vectorization). The algorithm only describes what is computed, while the customizations specify how the computation should be performed on hardware. The decoupled programming

Table 1: Example customization primitives in HeteroCL [24].

(a) Compute Customization	
`s[stage].pipeline(axis, ii)`	pipeline loop with II=`ii`
`s[stage].unroll(axis, factor)`	unroll loop with target `factor`
`s[stage].tile(i, factors)`	tile loop with `factors`
(b) Memory Customization	
`s.reuse_at(tensor, stage)`	create reuse buffer for tensor in `stage`
(c) Data Type Customization	
`s.quantize(tensor, dtype)`	quantize `tensor` to fixed-point type

model was original proposed in Halide [36], and it is later adopted by several other frameworks such as TVM [8]and HeteroCL [24].

Among these decoupled programming models, HeteroCL is the one primarily focusing on FPGA-based computing. Similar to Halide, HeteroCL separates an algorithm specification from compute customization techniques such as loop reordering, tiling, unrolling, and pipelining. HeteroCL further decouples the algorithm from memory architectures and data quantization schemes, which are both essential for efficient hardware acceleration. With respect to memory customization, HeteroCL provides primitives to create custom on-chip memory hierarchy through banking and reuse buffers.

Table 1 shows a subset of customization primitives provided by HeteroCL. In HeteroCL, a kernel, which contains a loop nest or function to perform computations, is defined as a *compute stage*. The decoupled customization primitives are applied to either a stage (i.e., compute customization) or the memory and data used by a stage (e.g., memory and data type customization).

It is worth noting that HeteroCL does not provide an explicit abstraction to model data placement, which essentially captures the interdependence between custom memories and compute units. The programmers need to either embed their placement schemes into the algorithm code or rely on the compiler to generate a default scheme.

3 MOTIVATIONAL EXAMPLE

We use image blurring as a motivational example to demonstrate the limitations of programming data placement and related optimizations in HLS. The algorithm takes in a 2D image as an input and computes the output by pushing it through two back-to-back 1D convolution kernels. To achieve better performance, we apply several hardware customization techniques such as loop tiling, data reuse, and data quantization. In the following, we focus on constructs and optimizations related to data placement.

To begin with, we describe the boundary between host and accelerator. In HLS, we need to maintain two sets of codes: one for describing the offloaded logic (i.e., the accelerator code) and one for handling data transfer (i.e., the host code). We show the optimized accelerator code in Figure 1. Throughout this example, we use Vivado HLS syntax . With HLS, in addition to defining a top-level function (L3-4), we need to specify the data communication interface via vendor-specific directives such as `pragma HLS interface` (L5-6). If a user decides to shift the boundary between host and accelerator, they need to extensively restructure the code by modifying the top-level function signature, the directives, and the function body, which is less productive and more error-prone.

More work needs to be done when the number of data transferred exceeds the number of physical ports. In this case, programmers need to manually schedule the I/O. Here, after loop tiling and unrolling, we end up with 8 compute units executing in parallel. As shown in L8,

```
1  typedef ap_int<W> DTYPE;
2  // max number of ports per DRAM bank = 14
3  void blur(DTYPE* input0, ..., DTYPE* input6,
4            DTYPE* output0, ..., DTYPE* output6) {
5    #pragma HLS interface port=input0 bundle=g0 burst=32
6    #pragma HLS interface port=input1 bundle=g1 burst=32
7    ...
8    stream<DTYPE> fifo_in[8], fifo_out[8];
9    input_io_schedule(fifo_in, input0, ..., input6);
10   compute_units(fifo_in, fifo_out);
11   output_io_schedule(fifo_out, output0, ..., output6);}
```

Figure 1: Accelerator code for blur in HLS.

we need 8 input and 8 output ports (i.e., 16 ports in total). However, assuming the target accelerator only has 14 ports per DRAM bank, we need to schedule the I/O due to insufficient ports (L9 and L11).

```
1  void compute_units(stream<DTYPE> fifo_in[8], fifo_out[8]) {
2    stream<DTYPE> fifo_inter[8]; #pragma HLS dataflow
3    #pragma HLS stream var=fifo_inter[0] depth=32
4    #pragma HLS stream var=fifo_inter[1] depth=32
5    ...
6    conv1(fifo_in, fifo_inter);
7    conv2(fifo_inter, fifo_out);}
```

Figure 2: Describing task-level dataflow with FIFOs and vendor-specific directives in HLS.

To exploit task-level parallelism, one way is to execute the two convolution kernels in a dataflow fashion. In HLS, as shown in Figure 2, we need to first define the FIFOs that connect the two compute kernels in L3. We also need to configure the FIFO depth in L4-5. Finally, we may need to include vendor-specific directives such as `pragma dataflow` for the HLS compiler to generate the right hardware architecture (L2). Such an approach does not scale well when the number of compute kernels increases.

```
1  void PE(DTYPE weight, stream<DTYPE> Xin, Yin, Yout)
2    Yout.write(weight * Xin.read() + Yin.read());
3  void conv2(stream<DTYPE> fifo_inter[8], fifo_out[8]) {
4    for (yo=0; yo<128; yo++)
5      for (xoo=0; xoo<16; xoo++) {
6        for (xoi=0; xoi<8; xoi++) { #pragma HLS unroll
7          stream<DTYPE> Xin[3], Yin[3], Yout[3];
8          broadcast(fifo_inter, Xin[0], Xin[1], Xin[2]);
9          PE(w2[0],Xin[0],Yin[0],Yout[0]); Yin[1]=Yout[0];
10         PE(w2[1],Xin[1],Yin[1],Yout[1]); Yin[2]=Yout[1];
11         PE(w2[2],Xin[2],Yin[2],Yout[2]);
12         data_drainer(Yout[2], fifo_out);}}}
```

Figure 3: Realizing loop-level dataflow using a systolic array.

Finally, for loop-level dataflow, a common approach is to generate high-performance spatial architectures such as systolic arrays. Figure 3 shows the HLS code that implements the second convolution kernel as a weight-stationary (semi-)systolic array. With HLS, describing a systolic array is usually widely different from describing general computation. For instance, we need to define the behavior of each processing element (PE) in the array (L1-2). We also need to define the connections between the PEs (L7-12), which includes nontrivial data orchestration such as broadcasting and draining. Any misconnections may break functionality or result in deadlocks.

To summarize, to apply data placement at different levels, we need not only to use a wide variety of vendor-specific directives but to take care of low-level target-specific details as well. The tightly entangled algorithm specification and data placement schemes make the codes

Table 2: Semantics of the `.to()` primitive for data placement — At each level of placement, multiple data orchestration methods with different performance-area trade-offs are supported. We mark the default mode(s) as **bold**. Inter- and intra-kernel data placement schemes share the same set of modes.

.to(data, destination, mode=**default**)

Level of Placement	Data Granularity	Destination	Mode	Access Order	Hardware Implementation	Area Overhead	DAE	Data Movement
Host-accelerator	Tensor	Storage Media	Cache	Arbitrary	Cache/Cache Interface	High	N	Implicit
			DMA	Sequential	FIFO+BRAM	Medium	Y	Explicit
				Arbitrary	FIFO+BRAM	Medium-High	N	Explicit
Inter-kernel	Tensor	Compute Stage(s)	**Stream**	Sequential	SRL, BRAM, etc.	Low	Y	Explicit
			Single Buffer	Arbitrary	BRAM, Register, etc.	Low-Medium	N	Explicit
Intra-kernel	Scalar		Double Buffer	Arbitrary	BRAM, Register, etc.	Medium	Y	Explicit

```
void conv1(stream<DTYPE> fifo_in[8], fifo_inter[8]) {
  DTYPE buffer[2][8][64];
  for (yo=0; yo<128; yo++) {
    for (xoo=0; xoo<16; xoo++) {
      for (xoi=0; xoi<8; xoi++) { #pragma HLS unroll
        write_buffer(buffer[xoo%2][xoi], fifo_in[xoi]);
        compute_conv(buffer[1-xoo%2][xoi], fifo_inter[xoi]);
}}}}
```

Figure 4: Co-optimizing data placement with other hardware customization in HLS.

```
import heteroflow as hf
...
conv1 = conv1d(img, w1)
conv2 = conv1d(conv1, w2)
s = hf.create_schedule()
p = platform.xcel_system

# host-accelerator
s.to([img,w1,w2], p.xcel)
s.to(conv2, p.host)
# inter-kernel
s.to(conv1, conv2)
# intra-kernel
PEs=s[conv1].unroll(axis=1)
pe0, pe1, pe2 = PEs
s.to(img.v,[pe0,pe1,pe2])
s.to(pe0.Y, pe1).to(pe2)
```

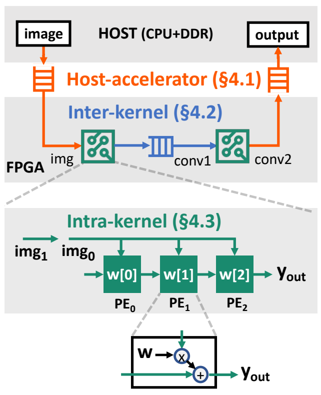


Figure 5: Overview of HeteroFlow programming model.

even more tedious. Not to mention the increased coding complexity brought by co-optimizations with other hardware customization. Thus, we need a unified yet concise programming abstraction to describe data placement from all design levels for better productivity and composability.

4 THE PROGRAMMING MODEL

To overcome the deficiencies of HLS programming mentioned in Section 3, we propose HeteroFlow to enable programmers to specify data placement at different levels of design hierarchy and data granularity using a unified interface via the `.to()` primitive. Figure 5 provides an overview of the HeteroFlow programming model using the image blur example. With the `.to()` primitive, programmers can concisely specify data placement at multiple design hierarchies without changing the algorithm description as shown in the code snippet on the left. The resulting accelerator implementation on a CPU+FPGA platform is sketched on the right.

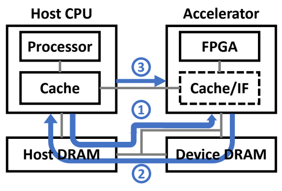
Table 2 describes the semantics of the `.to()` primitive, which takes in three arguments: the data to be placed, the destination where the data is placed, and the mode of placement. The types and values of these arguments may vary at different levels of design hierarchy:

- At the host-accelerator level, a tensor object (single or multi-dimensional array) is typically transferred between the host memory and the FPGA accelerator (either to/from the device memory or directly to/from the on-chip memories). The data can be accessed either through DMA or a cache-coherent interface.
- At the inter-kernel level, `.to()` can be used to place the data produced by one kernel (or a compute stage) to another kernel. Here each kernel is loosely defined as a loop nest or a function and typically each kernel produces one or more tensor objects. If the data elements in the tensor object are produced and consumed in the same sequential order, the compiler infers a FIFO to decouple the kernel computation from the communication. If the access order is arbitrary, a multi-buffer (typically a double-buffer) is generated. Unlike existing dataflow-oriented programming languages [5, 9, 14, 28, 34, 41], HeteroFlow does not require the explicit insertion of FIFO reads/writes. It is worth noting that `.to()` does not define the data dependence between kernels. Instead, the dependence is already defined by the algorithm specification. Here, `.to()` only describes the mechanism of the data placement.
- Finally, at the intra-kernel level, a sequence of `.to()` primitives can be combined to describe various fine-grained dataflow patterns within a kernel (i.e., a loop nest). The compiler can then leverage these explicitly specified patterns to infer highly efficient spatial architectures such as systolic arrays.

Table 2 also shows some additional attributes associated with the different modes of data placement. In the following, we describe how `.to()` and its associated modes operate at each level in more detail.

4.1 Host-Accelerator Data Placement

For a realistic application, it is usually not practical or beneficial to offload the entire program to the FPGA. Thus, programmers need to determine which portion(s) of the program should be accelerated. As shown in Section 3, the HLS users need to maintain both the accelerator code and host code. Using such an approach, if a user decides to change the placement of certain data (e.g., from host to accelerator), they have to extensively modify both portions. Moreover, programmers need to carefully manage the I/O scheduling with vendor-specific directives and other low-level library calls.



(a) Different data placement modes on an FPGA platform

```
1 import heteroflow as hf
2 p = hf.Platform.xcel_system
3 # Case 1. DMA (direct streaming)
4 s.to(data, p.xcel)
5 # Case 2. DMA (via device DRAM)
6 s.to(data,p.xcel.DRAM).to(p.host)
7 # Case 3. Data access over
      cache-coherent interface
8 s.to(tensor, p.xcel, hf.IO.Cache)
```

(b) HeteroFlow code

Figure 6: Example use cases of host-accelerator `.to()`.

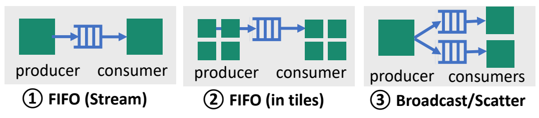
In contrast, HeteroFlow uses the `.to()` primitive to decouple the host-accelerator data placement from the algorithm specification, which has several advantages. First, the decoupled primitive allows the programmer to flexibly move the boundary between the host and accelerator. In other words, users only need to add or remove primitives, or change the data or destination without tainting the algorithmic code. Second, since the primitive is largely device independent, it is much easier to port the same program to other FPGA-equipped platforms. Meanwhile, the HeteroFlow compiler performs legality checks on the compatibility of the specified data placement modes. Finally, the I/O management and its optimization are now taken care of by the HeteroFlow compiler (more details available in Section 5.1).

The common use cases of `.to()` for host-accelerator data placement is shown in Figure 6. First, we import the predefined platform from HeteroFlow in L1-2. We then specify the destination and the mode for data placement. For the destination, every platform has two attributes: `host` and `xcel`. In Case 1, without specifying the mode, it is set to `DMA` by default. The HeteroFlow compiler automatically infers the low-level target-specific communication mechanism. By default, the data is transferred via direct streaming. However, if the data cannot be accessed in sequence, the data is first placed on the device memory (e.g., DRAM). Then, the data is loaded to the accelerator via communication protocols such as AXI. Instead of letting the compiler infer the communication mechanism, users can also explicitly specify the exact storage to place data. For example, in Case 2, we specify that we transfer data back to the host via device DRAM. Users can also set mode to `Cache` as in Case 3 if the target device provides on-chip caches or cache-coherent interfaces (i.e., Cache/IF in Figure 6).

4.2 Inter-Kernel Data Placement

To achieve efficient data streaming between compute kernels, the common practice is to use FIFOs to connect kernels or use double buffers to store the intermediate results. However, as we have shown in Section 3, both approaches are non-trivial in HLS. For FIFO-based connections, programmers need to explicitly replace the original array accesses with FIFO reads/writes (e.g., `hls::stream`) and provide additional vendor-specific directives (e.g., `pragma HLS dataflow`). For double buffers, programmers need to manage the buffer indices and the read/write behaviors. Both methods require tremendous effort to restructure the original program.

With the `.to()` primitive, programmers can easily apply various data placement schemes by setting the mode without touching the algorithm specification. Figure 7 shows examples of use cases of `.to()`. If no `.to()` is specified, the HeteroFlow compiler generates a single buffer by default. However, once it is specified as in Case 1, we now transfer the tensor to the consumer stage via either FIFO



```
# Case 1. User-specified FIFO with depth
s.to(tensor, consumer, hf.IO.Stream, depth=10)
# Case 2. Generate buffer under specific loop level
s.to(tensor, consumer, hf.IO.Buffer, axis=consumer.axis[1])
# Case 3. Broadcast/scatter tensor to consumers via FIFOs
s.to(tensor, [consumer0, consumer1, ...], hf.IO.Stream)
# Case 4. Auto-infer double buffer or FIFO
s.to(tensor, p.xcel).to(consumer)
```

Figure 7: Example use cases of inter-kernel `.to()`.

or double buffer depending on the access order. The programmer can also enforce the use of FIFO with a specific depth. Case 2 shows a more advanced use case, which further specifies the loop axis of the consumer stage. This is useful for both single and double buffer modes when we only want to transfer a subset of the data (e.g., a tile) at a time. In addition to one-to-one connection between two kernels, we can use `.to()` to concisely express the broadcast to a list of kernels as shown in Case 3. Finally, we can also combine inter-kernel data placement with host-accelerator data placement by *cascading* `.to()`. In Case 4, by cascading the two primitives, depending on the access order, if it is sequential, we directly transfer the tensor to the consumer stage via FIFO. Otherwise, an on-chip double buffer is generated to store the tensor. With the double buffer, the compiler generates a stream to transfer the tensor to the accelerator via FIFO.

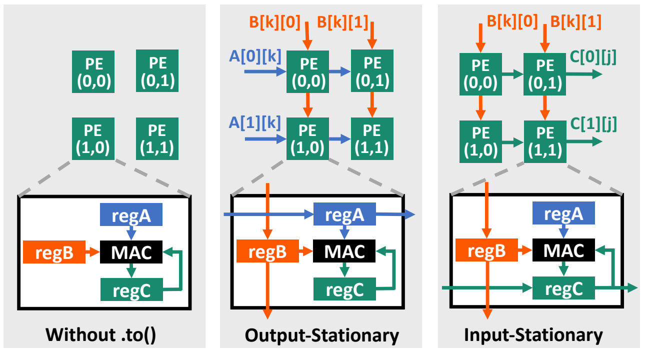
4.3 Intra-Kernel Data Placement

With HeteroFlow, a programmer can further leverage `.to()` to specify fine-grained data placement schemes. More concretely, a sequence of `.to()` primitives can be cascaded to describe near-neighbor connections that are commonly seen in efficient spatial architectures such as systolic arrays. In Figure 8, Case 1 shows an example of the cascaded `.to()`'s, where a scalar is propagated through multiple PEs. Here PEs are also compute stages, which can result from loop unrolling (i.e., if a loop is unrolled N times, we end up with N new stages). We can also use `.to()` to broadcast a scalar to a set of PEs as shown in Case 2. Here we use `tensor.v` to differentiate from `tensor`; the former is a scalar while the latter is a tensor. By concisely expressing these commonly-used design patterns such as cascade and broadcast, we can productively describe (and explore) various systolic and semi-systolic structures in a decoupled way without changing the algorithm code. Note that If we opt to use systolic array compilers such as AutoSA [43] as a back end, we can still leverage the user-specified dataflow patterns to constrain the search space of backend optimization.

```
# Case 1. Propagate a scalar through PEs by cascading .to()
s.to(PE0.X, PE1).to(PE2).to(PE3)
# Case 2. Broadcast a scalar to a list of PEs
s.to(tensor.v, [PE0, ...]);
```

Figure 8: Example use cases of intra-kernel `.to()`.

Figure 9 shows a more complete example with general matrix multiplication (GEMM). The algorithm is defined in Figure 9b L1-5. In this example, we generate a 2×2 systolic array for simplicity. It can be easily extended to a larger size. If we unroll the loop in L6 without



(a) Hardware architecture for PE arrays

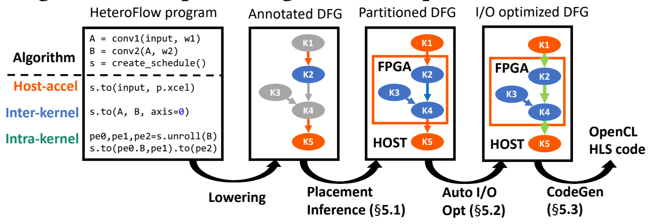
```
1   def gemm(A, B):
2       k = hf.reduce_axis(0,2)
3       return hf.compute((2,2), lambda i, j:
4           sum(A[i,k] * B[k,j], axis=k), "C")
5   s = hf.create_schedule(gemm)
6   PEs = s[C].unroll(axis=[i,j])
7   # output-stationary systolic array
8   [s.to(A[i].v, PEs[i,0]).to(PEs[i,1]) for i in range(0,2)]
9   [s.to(B[:][j].v, PEs[0,j]).to(PEs[1,j]) for j in range(0,2)]
10  # input-stationary systolic array
11  [s.to(B[:][j].v, PEs[0,j]).to(PEs[1,j]) for j in range(0,2)]
12  [s.to(PEs[i,0].C, PEs[i,1]).to(C[i].v) for i in range(0,2)]
```

(b) Describing different systolic arrays using `.to()`.

Figure 9: Systolic arrays with GEMM in HeteroFlow.

```
1   import heteroflow as hf
2   image = hf.placeholder(1024,1024)
3   def conv_1d(W, X):
4       k = hf.reduce_axis(0, 3)
5       return hf.compute(X.shape, lambda i,j:
6           sum(X[i,j+k]*W[k]), "Y")
7   conv1 = conv_1d(image, weight1)
8   conv2 = conv_1d(conv1, weight2)
9   s = hf.create_schedule()
10  p = hf.Platform.xcel_system
11
12  # host-accelerator data placement
13  # co-optimization with tiling and data reuse
14  yo, yi, xo, xi=s[conv1].tile(axis=[0,1], factor=[8,8])
15  xoo, xoi = s[conv1].split(axis=xo, factor=8)
16  s[conv1].reorder(yo, xoo, xoi, yi, xi)
17  s[conv1].unroll(axis=xoi)
18  buf = s.to(image, p.xcel)
19        .to(conv1, hf.IO.DoubleBuffer, axis=xoi)
20  s.reuse_at(buf, conv1)
21  s.to(conv2, p.host)
22
23  # inter-kernel data placement
24  # co-optimization with tiling
25  yo, yi, xo, xi=s[conv2].tile(axis=[0,1], factor=[8,8])
26  xoo, xoi = s[conv2].split(axis=xo, factor=8)
27  s[conv2].reorder(yo, xoo, xoi, yi, xi)
28  s[conv2].unroll(axis=xoi)
29  s.to(conv1, conv2)
30
31  # intra-kernel data placement
32  PEs = s[conv2].unroll(axis=5)
33  s.to(conv1, [pe0, pe1, pe2]); s.to(pe0.Y, pe1).to(pe2)
34  # co-optimization with data quantization
35  s.quantize([pe0.Y, pe1.Y, pe2.Y], hcl.Fixed(32,12))
```

Figure 10: Complete image blur example in HeteroFlow.



```
A = conv1(input, w1)
B = conv2(A, w2)
s = create_schedule()
s.to(input, p.xcel)
s.to(A, B, axis=0)
pe0,pe1,pe2=s.unroll(B)
s.to(pe0.B,pe1).to(pe2)
```

Figure 11: Compilation Flow in HeteroFlow – In the DFG, grey denotes the edge (vertex) missing placement information. Orange denotes that the edge (vertex) is placed off-chip, while blue denotes on-chip placement. Green denotes optimized I/O access.

using `.to()`, we end up with four PEs with no data movement in between. Each PE performs a multiplication and accumulation (MAC) operation (i.e., Figure 9a left). To describe an output-stationary systolic array (Figure 9a middle), we describe the data movement of both inputs `A` and `B` (L9-10). For input `A`, we propagate it horizontally through the PEs by using cascading `.to()`. Similarly, `B` is propagated vertically through the PEs. To describe a different dataflow pattern, users only need to use a different set of `.to()` without modifying the algorithm. For instance, to generate an input-stationary systolic array (Figure 9a right), we just need to specify the data placement for input `B` and output `C` (L13-14).

4.4 A Complete Example

Figure 10 shows a complete image blur example that uses the `.to()` primitives. We first describe the image blurring algorithm using HeteroCL APIs (L1-10). We then apply host-accelerator data placement in L18 and 21. For kernel-level data placement, we generate a local buffer by cascading two `.to()` in L18-19 and connect the two convolution kernels with another `.to()` in L29. In addition, we specify a weight-stationary systolic structure for the convolution in L32-33.

To achieve a high performance, we can further combine `.to()` with other hardware customization primitives to co-optimize compute units and memory. As we have seen in Section 3, with HLS, many different hardware customization techniques are embedded into the algorithmic code and tightly entangled with each other. On the other hand, with decoupled hardware customization, HeteroFlow can easily combine different customization techniques without the need of modifying the algorithm. One example is to combine loop tiling with both host-accelerator and inter-kernel data placement. With the image blur example in Figure 10, we first tile, split, and reorder the loop of the first convolution kernel (L14-16). Then, we use `.to()` along with a specified axis to load tiles to an on-chip double buffer (L18). Similarly, we apply the same set of compute customization to the second convolution kernel (L24-26) for proper streaming. Another important optimization to ensure the FIFO connection is inserting reuse buffers via `.reuse_at()` (L20). We also unroll the tiled loops to have multiple compute units calculate the outputs in parallel (L17 and 28). Finally, we combine data type customization using `.quantize()` (L36). To sum up, HeteroFlow provides a unified and compact programming interface via `.to()`, which applies to different levels of the design hierarchy and inter-operates with other hardware customization primitives. It is worth noting that aside from the example shown in Figure 10, programmers can also compose `.to()` with other primitives to explore various trade-offs in the comprehensive design space for data placement customization.

5 COMPILATION FLOW

This section describes an end-to-end compilation flow that generates HLS code from an input HeteroFlow program, as shown in Figure 11. The HeteroFlow compiler is built on the open-source HeteroCL framework [24]. It first lowers the input program to an intermediate representation (IR) and constructs a dataflow graph (DFG) annotated with user-specified data placement. Since users may only specify placement for a subset of the data objects, the compiler infers the placement for other objects and also the compute. The DFG is then partitioned into subgraphs based on the inference results (i.e., either host or accelerator). Notably, the HeteroFlow compiler opportunistically applies a set of optimizations for off-chip memory accesses to improve the bandwidth utilization. Finally, HeteroFlow generates optimized HLS C/C++ or OpenCL code. In the following, we provide more details on the placement inference, off-chip memory access optimization, and code generation.

5.1 Inference of Compute/Data Placement

To free programmers from tediously marking the placements of all data in the program, HeteroFlow automatically infers placement scheme for the portion of data and compute that is not explicitly annotated with `.to()`. We formulate placement inference as an integer linear programming (ILP) problem. Given a HeteroFlow program modeled as a DFG, $G = (V, E)$, where each vertex $v \in V$ represents a compute stage, and each edge $e \in E$ represents the data dependency between a pair of vertices with respect to a particular data object, we define a set of binary variables N_v to represent the placement of computation at each node. For $\forall v \in V$, $N_v = 1$ if and only if node v is mapped to the accelerator. To represent the data placement, we define another set of binary variables M_e, where for $\forall e = (u, v) \in E$, $M_e = XOR(N_u, N_v)$, i.e., $M_e = 1$ if and only if the edge e corresponds to an off-chip memory read or write. Each node is associated with a list of resource estimates $res_i(v)$, if this node is implemented on the accelerator. Each edge is associated with an estimated latency $lat(e) = data\_size(e)/bw$, if the edge involves off-chip communication. Here the off-chip memory bandwidth bw is measured through profiling. For each data array with user-specified data placement, we constrain its direct consumers in the DFG to be placed onto the same device as the data itself. As an example, if the user specifies `.to(image, p.xcel)`, then any DFG node that directly consumes the array *image* will be placed onto the accelerator. The set of DFG nodes affected by these user-specified constraints is denoted as $U \subset V$. We use a set of binary constants c_v to represent the user-specified constraints: for $\forall v \in U$, $c_v = 1$ if and only if the node v must be placed onto the accelerator. With these definitions, we can formulate the ILP as follows:

$$
\begin{aligned}
&\textit{Minimize} && \sum_{e \in E} lat(e) \times M_e \\
&\text{subject to} && \forall v \in U,\ N_v = c_v \\
& && \forall e = (u, v) \in E,\ M_e \le N_u + N_v,\ M_e \ge N_u - N_v, \\
& && M_e \ge N_v - N_u,\ M_e \le 2 - N_u - N_v \\
&\text{(optional)} && \forall i \in \{BRAM, LUT, FF, DSP\},\ \sum_{v \in V} res_i(v) \times N_v \le b_i
\end{aligned}
$$

where the second set of constraints linearize the XOR relationship between M_e, N_u, and N_v. The last set of constraints imply that the total resource utilization of all DFG nodes mapped to the accelerator must be below the amount of available resources. Figure 11 shows an example of placement inference, , where the inference algorithm takes in the partially annotated DFG and decides placement schemes for each data and compute. With complete placement information for all nodes and edges in the DFG, the accelerator-specific subgraph can be extracted, and hardware customizations can be applied to improve the performance of the accelerator.

5.2 Automatic I/O Optimizations

After graph partitioning, the HeteroFlow compiler automatically optimizes the off-chip memory accesses at the boundaries of the FPGA subgraph(s). These optimizations aim to saturate the off-chip memory bandwidth and maximize the throughput of memory accesses. The hardware information (e.g., device DRAM capacity and physical I/O port limit) needed by the compiler to make optimization decisions is included in the `Platform` object in HeteroFlow code.

Memory Coalescing: The maximum number of data bits that can be read out from the off-chip memory per access is usually larger than the data bitwidths used in the program. Memory coalescing tries to saturate the off-chip memory bandwidth by grouping multiple narrow memory accesses into one wider access. For each loop nest with a contiguous off-chip memory access pattern, HeteroFlow replaces the narrow memory operation with a bit-slice from a coalesced memory operation. The loop trip count and the bitwidth of the affected memory ports are also updated accordingly.

AXI Controller Configuration: The off-chip memory requests are initiated by on-chip AXI controllers and sent to off-chip memory controllers through the AXI bus. For modern FPGAs, the AXI controller is software-configurable and the configuration (e.g., burst length, I/O bundle) can affect the bandwidth efficiency in a subtle way. We employ a similar approach as proposed in [30] to profile microbenchmarks on a target platform and empirically decide the default threshold for each parameter based on the data size/bitwidth.

Memory Banking: Modern FPGA platforms are often equipped with multi-bank off-chip memories, e.g., DRAM or high bandwidth memory (HBM). To maximize off-chip memory throughput, the HeteroFlow compiler explores different off-chip data layouts and tries to minimize memory access conflicts by assigning competing off-chip memory accesses to different off-chip memory banks. Currently, HeteroFlow uses a simple greedy algorithm to decide the memory banking assignment – the compiler determines the priority of different memory requests based on their data transfer size, and then assigns the off-chip memory requests to any available off-chip memory that gives best performance for that memory request.

I/O Scheduling: Each off-chip memory bank on an FPGA can only serve a limited number of off-chip memory requests from different AXI controllers at the same time. An accelerator cannot be synthesized if it has too many parallel off-chip memory access requests. In such cases, the HeteroFlow compiler will insert a static scheduler inside the accelerator to arbitrate the off-chip memory requests to a limited number of AXI controllers. Similar to the memory banking optimization, the HeteroFlow compiler uses a greedy algorithm to assign the memory requests to the AXI controllers based on their priority (i.e., data transfer size). For memory requests that are assigned to the same AXI controller, the HeteroFlow compiler creates a for-loop in the generated HLS code to access data through the shared AXI controller. The AXI controller is shared between different requesters in a time-multiplexed fashion.

FIFO Inference: In addition to off-chip I/O optimization, HeteroFlow also automatically optimizes on-chip communication. Specifically,

```
1   // intra-kernel data placement with FIFOs
2   void conv_systolic_array(stream<DTYPE>& fifo_inter0,
         stream<DTYPE>& fifo_inter1) {
3     #pragma HLS dataflow
4     stream<DTYPE> fifo_in[M], fifo_out[N];
5     #pragma HLS stream var=fifo_in[0]
6     data_loader(fifo_inter0, fifo_in);
7     PE<0,0>(fifo_in[0], fifo_out[0]);
8     ...
9     PE<M,N>(fifo_in[M-1], fifo_out[N-1]);
10    data_drainer(fifo_in[M-1], fifo_inter1);}
11  // top-level function on accelerator
12  void fpga(DTYPE* dma_mm, stream<DTYPE>& dma_fifo, int iter) {
13      #pragma HLS interface m_axi port=dma_mm burst=factor
14      #pragma HLS interface axis port=dma_fifo burst=factor
15      for (i=0; i<K;i++) {
16        DTYPE in1 = dma_fifo.read();
17        DTYPE.in2 = dma_mm[INDEX[i]];
18        compute1(in1.range(31,0), in2.range(63,32), ...);}
19      // inter-kernel FIFOs and double buffer
20      stream<DTYPE> fifo_inter[N];
21      #pragma HLS stream var=fifo_inter[0]
22      DTYPE double_buf[2][SIZE];
23      conv_systolic_array(fifo_inter[0], fifo_inter[1]);
24      compute2(fifo_inter[1], double_buf[iter%2]);
25      compute3(double_buf[1-iter%2]);}
```

Figure 12: Example of HLS code generated by HeteroFlow.

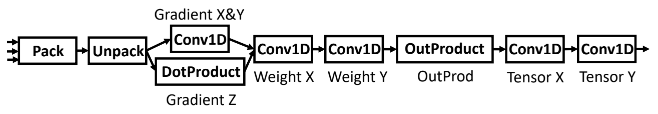
our compiler can infer FIFO channels for sequential in-order inter-kernel communication. For intra-kernel data placement, the compiler automates several aspects of the systolic array generation such as insertions of data loader and drainer modules and the inter-PE communication media (FIFOs or shift registers), according to user-specified dataflow patterns using the `.to()` primitives.

5.3 Code Generation

The HeteroFlow compiler backend emits OpenCL or HLS C/C++ code that can be compiled and deployed on mainstream FPGA platforms. This backend generates high-performance code for the communication channels with vendor-specific libraries and pragmas, and further leverages the existing HeteroCL compiler to realize an optimized accelerator according to other user-specified hardware customizations. Figure 12 shows the HLS code snippets generated by HeteroFlow, which includes data placement specification at different levels. Inside the compute kernel function mapped to a systolic array, HeteroFlow generates parallel PEs connected by FIFOs for intra-kernel data movement (L3-9). In the top-level function on the FPGA accelerator, HeteroFlow assigns different memory interfaces according to the memory access pattern to achieve the best memory bandwidth with minimal hardware overhead (L13-14). Additionally, HeteroFlow automatically applies memory coalescing in the data loading loop to saturate the off-chip memory bandwidth (L15-18). On-chip FIFOs and double buffers are automatically generated to fulfill the requirements for inter-kernel data placement (L20-25).

6 EVALUATION

In this section, we select a set of realistic benchmarks and evaluate the accelerators generated by HeteroFlow. We target two mainstream commercial FPGA boards: Xilinx Alveo U280 Accelerator Card and Intel Stratix 10 (S10) GX2800. We use Xilinx Vitis 2019.2 [44] and Intel FPGA SDK for OpenCL 18.0 [19] to synthesize bitstream.



```
1   # (a) data placement between xcel and off-chip memory
2   s.to(packed_frames, p.xcel); s.to(output, p.host)
3   # (b) data placement between compute kernels
4   s.to(tensor_y, tensor_weight_y)
5   # (c) reuse input in kernel and quantize intermediate data
6   s.reuse_at(tensor_weight_y.in_, tensor_weight_y.axis[0])
7   s.quantize([grad_x, ...], Fixed(23,12))
```

Figure 13: DFG and HeteroFlow code for optical flow.

```
1   void optical_flow(frame_t* frames, output_t* outputs) {
2     #pragma HLS INTERFACE m_axi port=frames bundle=gmem0
3     ...
4     #pragma HLS dataflow
5     hls::stream<DTYPE> inter_fifo_0;
6     #pragma HLS stream var=inter_fifo_0 depth=1024
7     ...
8     tensor_weight_y(inter_fifo_5, inter_fifo_6);
9     tensor_weight_x(inter_fifo_6, inter_fifo_7);
10    flow_calc(inter_fifo_7, outputs);}
```

Figure 14: Manually optimized HLS code for optical flow.

6.1 Case Study: Optical Flow

Optical flow is a widely used video processing algorithm for motion detection. We show the dataflow graph in Figure 13, where each block represents a loop nest that processes the input frame(s) and generates output in raster scan order. We choose the algorithm implemented in the Rosetta benchmark suite [46]. The algorithm reads in a sequence of HD video frames (436×1024) and outputs a 2D vector field that reveals the object's motion.

Figure 13 also shows the HeteroFlow code to optimize optical flow. In L2, we move the packed input frames from `Pack` stage to the accelerator, and the final output back to the host. Consequently, the `Pack` stage is computed on the host, and all other stages are accelerated on the FPGA device. The `Unpack` stage reads the packed frame from off-chip device memory, unpacks it, and sends it to the following stages. We also connect stages with inter-kernel FIFOs (L4) and co-optimize data placements with reuse buffer and quantization (L6-7). For comparison, Figure 14 shows the HLS counterpart for defining the I/O interface, which is much more verbose.

We evaluate the design on different FPGAs and compare the performance with manually optimized HLS design from [46]. The results are shown in Table 3. Our design matches the performance of the manually optimized design in HLS C++, while requiring 3.6× fewer lines of code. To evaluate portability, we also map the design to an Intel Stratix 10 FPGA. Since the Intel OpenCL SDK for FPGA does not provide direct support for fixed-point data types, we use floating-point data types for evaluation. The full-precision design results in more resource consumption on Intel Stratix 10 FPGA, and its run time is slightly longer due to a lower frequency.

6.2 Case Study: GEMM

We use 64×64×64 GEMM as an example and use `.to()` to implement it with systolic arrays of different dataflow patterns. Due to limited on-chip resource on an FPGA, we can fully unroll the loops to build a 64x64 systolic array. Instead, we tile the loop nest with a factor of (4,4) and implement the inner loops with a 4×4 systolic array that

Table 3: Evaluation on optical flow in HeteroFlow — In addition to the resource usage, we show the maximum frequency (Fmax), run time (RT), and the number of lines of code (LoC).

	FPGA	# LUTs/FF	# BRAM/DSP	Fmax(MHz)	RT(ms)	LoC
Rosetta [46]	U280	21.7K/30.5K	66/196	300	3.49	742
HeteroFlow	U280	23.8K/32.6K	64/182	300	3.43	206
HeteroFlow	S10	29.5K/58.1K	484/106	286	3.82	206

computes GEMM in tiles. Due to space limitation, we do not show HeteroFlow code here, which is similar to Figure 9b. The systolic array can be implemented in different dataflow patterns as discussed in Section 4.3. Here we evaluate two representative dataflow patterns: output-stationary (OS) and input-stationary (IS).

HeteroFlow can generate systolic arrays using AutoSA [43] or the HLS C/OpenCL backend. The AutoSA backend provides a push-button solution to generate high-performance systolic array code, but it has limited support for quantization. We run the experiments on Xilinx U280 FPGA, where the input and output data are transferred between host and accelerator through HBM banks. The results are shown in Table 4. The IS/OS systolic array generated by the AutoSA backend has close performance. AutoSA generates a different I/O network to load (drain) data to (from) systolic arrays of different dataflow patterns. Such an I/O network could be complex and consume more on-chip resource. In HeteroFlow, we used `.to()` to optimize the off-chip memory (i.e., double buffer and memory coalescing), and we are able to achieve very close performance with AutoSA using less resource. We can further quantize the design to fixed-point to achieve an even better throughput. Notably, HeteroFlow can also integrate optimized systolic arrays generated by AutoSA with other kernels using the `.to()` interface, although currently AutoSA only generates single systolic array kernel.

Table 4: Evaluation on GEMM systolic array in HeteroFlow — We measure the throughput in Giga operations per second (GOPS).

	Data type	# LUT/FF	# BRAM/DSP	GOPS
IS (HF-AutoSA)	FP32	30.9K/44.1K	47/48	2.07
OS (HF-AutoSA)	FP32	42.3K/57.9K	103/48	2.06
OS (HF-HLSC)	FP32	25.4K/32.9K	23/48	2.03
OS (HF-HLSC)	Fixed<16,4>	10.2K/15.2K	15/16	4.26

6.3 Case Study: K-Nearest Neighbors

K-nearest neighbors (KNN) is a classification algorithm used in a wide range of domains such as machine learning and data mining [3, 15]. In this case study, we port an HLS-based KNN implementation from uBench [30] to HeteroFlow and show that the HeteroFlow compiler can automatically optimize I/O to improve the performance.

Figure 15 shows the KNN code snippet in HeteroFlow. Since paired distances between the query point and data points in the KNN search space can be calculated independently, we duplicate multiple PEs to compute (L5). After each PE calculates and sorts the local distance, it sends the top-K results to a global merger to generate the final top-K distances (L6). These PEs access input data from the off-chip DRAM bank, and output is written back to the same location (L9). In this case study, we map the KNN design to a Xilinx U280 FPGA, and use only one DRAM bank on U280 to evaluate HeteroFlow's automatic I/O optimizations in a resource constrained situation. A single DRAM bank can only serve memory requests from up to 15 AXI controllers at the same time. To make PEs and the global merger execute in parallel, programmers need to reserve one AXI controller for the global merger to write outputs to, and 14 AXI controllers for 14 PEs to read inputs from the same off-chip DRAM bank.

```
1  def KNN(query, inputs):
2      def PE(input_):
3          local_dis = compute_distance(input_, query)
4          return sort(local_dis)
5      PEs = [PE(inputs[n]) for n in range(N)]
6      output = merger(PEs); return output
7  ...
8  # (a) data movement between host and accelerator
9  s.to(inputs,p.xcel.DRAM); s.to(output,p.xcel.DRAM).to(p.host)
10 # (b) data movement between compute kernels
11 [s.to(KNN.PEs[n], KNN.merger) for n in range(N)]
```

Figure 15: KNN algorithm in HeteroFlow.

Table 5: Ablation analysis on automatic I/O optimization in HeteroFlow — N/A means the design is not synthesizable because physical I/O ports on FPGA are not enough to serve 28 PEs.

(a) KNN with 14 PEs.

Optimization	RT(s)	Speedup
baseline	49.37	1.00x
+mem-coalescing	24.29	2.03x
+axi-controller [30]	10.14	4.82x
+io-scheduling	10.14	4.82x

(b) KNN with 28 PEs.

Optimization	RT(s)	Speedup
baseline	N/A	–
+mem-coalescing	N/A	–
+axi-controller [30]	N/A	–
+io-scheduling	9.31	5.30x

From the results shown in Table 5(a), we can obtain 4.82× speedup with optimizations in memory coalescing and AXI controller configuration. Since the total number of off-chip memory accesses in 14-PE KNN does not exceed the physical port limit, the I/O scheduling optimization does not improve the performance. HeteroFlow automatically optimizes off-chip memory accesses in 14-PE KNN and achieves same performance as the manually optimized design in [30] with much fewer lines of code. In Table 5(b), we increase the PE number to 28. This doubles the number of total parallel I/O requests, which exceeds the physical port limit of one DRAM bank. As a result, the design becomes non-synthesizable. With I/O scheduling optimization, HeteroFlow assigns two PEs to one AXI controller, which requests data from off-chip memory and sends data to the two PEs. Consequently, the 28-PE KNN design becomes synthesizable with limited I/O ports and achieves an even higher speedup of 5.30×.

6.4 Case Study: UltraNet

UltraNet [45] is an object detection neural network implemented on FPGAs, and the winner of the 2020 DAC System Design Contest. UltraNet has 9 convolution layers implemented as matrix multiplication units. Figure 16 shows the HeteroFlow code for UltraNet where we map the third Conv2D layer to a systolic array. The algorithm is defined in L1-4. We connect the second and third layers with a FIFO (L7). To map the third Conv2D layer to a systolic array, we first tile and reorder the outermost loops (L9-10), unroll the middle loops to spatial PEs (L11), and customize inter-PE data placement to build an output-stationary systolic array (L12-15). Then, we vectorize PE's inner loop (L17) to compute multiple MAC operations in SIMD. We further quantize inputs and weights to 4-bit integers (L18).

We evaluate the optimized UltraNet design with a systolic array and compare the results with the original design as baseline. The baseline implementation has eight vectorized PEs with 16 SIMD lanes. Each SIMD lane computes input pixels in parallel, and each PE computes output channels in parallel. Our 4×4 systolic array

```
def ultranet(image):
  out1 = layer1_conv2d_im2col(image, weight1)
  out2 = layer2_conv2d_im2col(out1, weight2)
  out3 = layer3_conv2d_im2col(out2, weight3)
  ...
# inter-kernel data placement
s.to(out2, layer3_conv2d_im2col)
# build output-stationary systolic array
yo, yi, xo, xi = s[out3].tile(axis=[0,1], factor=[4,4])
s[out3].reorder(yo, xo, yi, xi)
PEs = s[out3].unroll(axis=[yi, xi])
for r in range(4):
  s.to(out2[r][:].X, PEs[r,0]).to(PEs[r,1]).to(PEs[r,2])...
for c in range(4):
  s.to(out2[:][c].W, PEs[0,c]).to(PEs[1,c]).to(PEs[2,c])...
for PE in PEs:
  s[PE].vectorize(axis=PE.j, factor=32)
  s.quantize(PE.X, PE,W], hf.Int(4))
```

	# LUTs	# FFs	# BRAM	# DSPs	Fmax(MHz)	RT(ms)
Baseline	60.2K	39.6K	377	508	231	2.97
+Systolic Array	69.8K	39.4K	375	594	233.8	2.27

Figure 16: Evaluation on UltraNet in HeteroFlow.

with 32 SIMD lanes theoretically offers 4× acceleration for the third layer. Hardware emulation shows that the third layer in baseline design takes 1.843M cycles to complete, while the systolic array implementation only takes 0.461M cycles. We show that with less than 10 lines of code, we achieve 3.99× speed up for the third layer, and an overall latency improvement from 2.97ms to 2.27ms.

7 RELATED WORK

Dataflow HLS: FPGA is an excellent fit for dataflow execution due to the availability of massive distributed hardware resources. Many HLS tools [20–22, 42] can automatically convert dataflow HLS programs into dataflow graphs followed by generation of dataflow circuits. Dynamatic [20, 22] generates fine-grained elastic circuits to enable dynamically scheduled HLS. TAPA [10] defines a programming interface to describe dataflow parallelism within an application to construct heterogeneous pipelines. Optimus [18], Maxeler [31], and ST-Accel [37] propose a programming model to describe streaming applications as dataflow graphs. In comparison, HeteroFlow introduces a programming model with decoupled data placement and leverages the capability of downstream HLS tools to generate efficient dataflow accelerators for FPGAs.

Dataflow DSL: Several works propose DSLs and compilers [2, 5, 9, 13, 29, 38, 39] to automatically synthesize dataflow circuits. Darkroom [17] compiles image-processing programs directly into line-buffered pipelines. Spatial [23] defines special constructs to describe data movements between kernels in the program. SODA [9] is a DSL for stencil applications, and it compiles declarative operations into high performance dataflow architectures. In comparison, HeteroFlow provides a decoupled and unified programming interface for expressing data placement at different levels of memory hierarchy resulting in a modular and composable design specification.

DSLs with Decoupled Optimization: Halide [36] and TVM [8] decouple the algorithm definition from its schedule for building high performance kernels for image processing and deep learning applications. T2S [40] and SuSy [25] provide decoupled scheduling primitives to generate high-performance systolic architectures on FPGAs. HeteroCL [24] decouples the algorithm from a temporal compute schedule, on-chip memory customization, and data-quantanization scheme. TIRAMISU [4] is based on the polyhedral model with a rich scheduling language allowing fine-grained control of optimizations. Fireiron [16] is a data-movement-aware scheduling language for GPUs that customizes compute of kernel and data movements between memory hierarchies. HeteroFlow represents the first FPGA-focused DSL that enables fully decoupled data placement and co-optimization with other hardware customizations such as tiling and data quantization.

	Design Entry	Decoupled Compute	Decoupled DP*	Unified DP* Interface	Design Complexity
HLS	C++	No	No	No	Complete design
Spatial [23]	DSL	No	No	No	Complete design
SODA [9]	DSL	No	No	No	Single kernel (stencil)
AutoSA [43]	C++	No	No	No	Single kernel (systolic)
HeteroHalide [27]	DSL	Yes	No	No	Complete design
T2S [40], SuSy [25]	DSL	Yes	Partially	No	Single kernel (systolic)
HeteroCL [24]	DSL	Yes	No	No	Single kernel
HeteroFlow	DSL	Yes	Yes	Yes	Complete design

Table 6: Comparison between HeteroFlow and other programming frameworks. *DP stands for data placement.

Data Placement in Deep Learning Frameworks PyTorch [32] provides a `.to()` interface for users to explicitly move tensors and computation to accelerator devices. In contrast, TensorFlow [1] and MXnet [7] can automatically infer the location of the computation based on the manually-specified placement of input tensors. While the `.to()` interface in HeteroFlow shares some similar features with PyTorch, HeteroFlow can also infer the placement of computations like TensorFlow and MXNet. HeteroFlow also supports fine-grained control over on-chip data communication, which is important for achieving high performance and area efficiency on FPGAs.

Table 6 shows a comprehensive comparison between HeteroFlow and a set of representative prior arts. To summarize, HeteroFlow is the first to provide a decoupled and unified programming interface for expressing data placement optimizations for complete accelerator design (instead of a single kernel).

8 CONCLUSION

We have presented HeteroFlow, an FPGA accelerator programming model that provides a unified interface `.to()` for describing data placement optimizations from different design levels in host-accelerator, inter-kernel, and intra-kernel. Furthermore, we decouple the data placement specification from the algorithm specification and other hardware customizations, which enables better productivity and portability. Our evaluation results on a set of realistic benchmarks show that programs written in HeteroFlow can match the performance of highly optimized manual HLS counterparts with much fewer lines of code. Our future work will focus on automating the co-optimization of data placement and temporal loop-level scheduling to further reduce the FPGA design complexity.

ACKNOWLEDGEMENTS

This research was supported in part by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, NSF Awards #1453378, #1909661, #2118709, NSF/Intel CAPA Awards #1723715, and research gifts from Intel and Xilinx. We thank Xingyu Tian and Prof. Zhenman Fang of Simon Fraser for their helpful discussions and the anonymous reviewers for their feedback on earlier versions of this manuscript.

REFERENCES

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. Tensorflow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, 2016.

[2] M. Abid, K. Jerbi, M. Raulet, O. Déforges, and M. Abid. System Level Synthesis of Dataflow Programs: HEVC Decoder Case Study. *Electronic System Level Synthesis Conf. (ESLsyn)*, 2013.

[3] N. S. Altman. An Introduction to Kernel and Nearest-Neighbor Nonparametric Regression. *The American Statistician*, 46(3):175–185, 1992.

[4] R. Baghdadi, J. Ray, M. B. Romdhane, E. D. Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. Amarasinghe. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. *Int'l Symp. on Code Generation and Optimization (CGO)*, 2019.

[5] S. S. Bhattacharyya, G. Brebner, J. W. Janneck, J. Eker, C. von Platen, M. Mattavelli, and M. Raulet. OpenDF: A Dataflow Toolset for Reconfigurable Hardware and Multicore Systems. *SIGARCH Comput. Archit. News*, 2009.

[6] U. Bondhugula, J. Ramanujam, and P. Sadayappan. Automatic Mapping of Nested Loops to FPGAs. *ACM SIGPLAN Conf. on Principles and Practice of Parallel Programming (PPoPP)*, 2007.

[7] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *arXiv preprint arXiv:1512.01274*, 2015.

[8] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, et al. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. *USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, 2018.

[9] Y. Chi, J. Cong, P. Wei, and P. Zhou. SODA: Stencil with Optimized Dataflow Architecture. *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2018.

[10] Y. Chi, L. Guo, Y. Choi, J. Wang, and J. Cong. Extending High-Level Synthesis for Task-Parallel Programs. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2021.

[11] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, 2011.

[12] J. Cong and J. Wang. PolySA: Polyhedral-Based Systolic Array Auto-Compilation. *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2018.

[13] D. Diamantopoulos and C. Kachris. High-level Synthesizable Dataflow MapReduce Accelerator for FPGA-Coupled Data Centers. *Int'l Conf. on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, 2015.

[14] J. Eker and J. Janneck. CAL Language Report: Specification of the CAL Actor Language. *ERL Technical Memo UCB/ERL*, 2003.

[15] E. Fix and J. L. Hodges. Discriminatory Analysis. Nonparametric Discrimination: Consistency Properties. *Int'l Statistical Review / Revue Internationale de Statistique*, 57(3):238–247, 1989.

[16] B. Hagedorn, A. S. Elliott, H. Barthels, R. Bodík, and V. Grover. Fireiron: A data-movement-aware scheduling language for GPUs. *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2020.

[17] J. Hegarty, J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and P. Hanrahan. Darkroom: Compiling High-Level Image Processing Code into Hardware Pipelines. *ACM Trans. Graph.*, 2014.

[18] A. Hormati, M. Kudlur, S. Mahlke, D. Bacon, and R. Rabbah. Optimus: Efficient Realization of Streaming Applications on FPGAs. *Intl'l Conf. on Compilers, Architectures and Synthesis of Embedded Systems (CASES)*, 2008.

[19] Intel. Intel FPGA SDK for OpenCL. https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html. Accessed: 2021-12-30.

[20] L. Josipović, R. Ghosal, and P. Ienne. Dynamically Scheduled High-Level Synthesis. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2018.

[21] L. Josipovic, A. Guerrieri, and P. Ienne. Synthesizing General-Purpose Code Into Dynamically Scheduled Circuits. *IEEE Circuits and Systems Magazine*, 2021.

[22] L. Josipović, S. Sheikhha, A. Guerrieri, P. Ienne, and J. Cortadella. Buffer Placement and Sizing for High-Performance Dataflow Circuits. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2020.

[23] D. Koeplinger, M. Feldman, R. Prabhakar, Y. Zhang, S. Hadjis, R. Fiszel, T. Zhao, L. Nardi, A. Pedram, C. Kozyrakis, et al. Spatial: A Language and Compiler for Application Accelerators. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2018.

[24] Y.-H. Lai, Y. Chi, Y. Hu, J. Wang, C. H. Yu, Y. Zhou, J. Cong, and Z. Zhang. HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2019.

[25] Y.-H. Lai, H. Rong, S. Zheng, W. Zhang, X. Cui, Y. Jia, J. Wang, B. Sullivan, Z. Zhang, Y. Liang, et al. SuSy: A Programming Model for Productive Construction of High-Performance Systolic Arrays on FPGAs. *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2020.

[26] Y.-H. Lai, E. Ustun, S. Xiang, Z. Fang, H. Rong, and Z. Zhang. Programming and Synthesis for Software-defined FPGA Acceleration: Status and Future Prospects. *ACM Trans. on Reconfigurable Technology and Systems (TRETS)*, 14(4):1–39, 2021.

[27] J. Li, Y. Chi, and J. Cong. HeteroHalide: From image processing DSL to efficient FPGA acceleration. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2020.

[28] R. Li, Y. Yang, L. Berkley, and R. Manohar. Fluid: An Asynchronous High-level Synthesis Tool for Complex Program Structures. *IEEE Int'l Symp. on Asynchronous Circuits and Systems (ASYNC)*, 2021.

[29] T. Liang, J. Zhao, L. Feng, S. Sinha, and W. Zhang. Hi-ClockFlow: Multi-Clock Dataflow Automation and Throughput Optimization in High-Level Synthesis. *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2019.

[30] A. Lu, Z. Fang, W. Liu, and L. Shannon. Demystifying the Memory System of Modern Datacenter FPGAs for Software Programmers through Microbenchmarking. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2021.

[31] Maxeler. Maxeler high-performance dataflow computing systems. https://www.maxeler.com/products/software/maxcompiler/. Accessed: 2021-12-30.

[32] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in Neural Information Processing Systems (NIPS)*, 2019.

[33] M. Pellauer, Y. S. Shao, J. Clemons, N. Crago, K. Hegde, R. Venkatesan, S. W. Keckler, C. W. Fletcher, and J. Emer. Buffets: An Efficient and Composable Storage Idiom for Explicit Decoupled Data Orchestration. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[34] F. Peverelli, M. Rabozzi, E. Del Sozzo, and M. D. Santambrogio. OXiGen: A Tool for Automatic Acceleration of C Functions into Dataflow FPGA-Based Kernels. *Int'l Parallel and Distributed Processing Symp. Workshops (IPDPSW)*, 2018.

[35] J. Pu, S. Bell, X. Yang, J. Setter, S. Richardson, J. Ragan-Kelley, and M. Horowitz. Programming Heterogeneous Systems from an Image Processing DSL. *ACM Trans. on Architecture and Code Optimization (TACO)*, 2017.

[36] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. *ACM SIGPALN Notices*, 2013.

[37] Z. Ruan, T. He, B. Li, P. Zhou, and J. Cong. ST-Accel: A High-Level Programming Platform for Streaming Applications on FPGA. *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2018.

[38] C. Rubattu, F. Palumbo, C. Sau, R. Salvador, J. Sérot, K. Desnos, L. Raffo, and M. Pelcat. Dataflow-Functional High-Level Synthesis for Coarse-Grained Reconfigurable Accelerators. *IEEE Embedded Systems Letters*, 2019.

[39] J. Sérot, F. Berry, and S. Ahmed. CAPH: A Language for Implementing Stream-Processing Applications on FPGAs. *Embedded Systems Design with FPGAs*, 2013.

[40] N. Srivastava, H. Rong, P. Barua, G. Feng, H. Cao, Z. Zhang, D. Albonesi, V. Sarkar, W. Chen, P. Petersen, et al. T2S-Tensor: Productively Generating High-Performance Spatial Hardware for Dense Tensor Computations. *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2019.

[41] M. Technologies. Maxcompiler white paper. https://www.maxeler.com/media/documents/MaxelerWhitePaperMaxCompiler.pdf. Accessed: 2021-12-30.

[42] R. Townsend, M. A. Kim, and S. A. Edwards. From Functional Programs to Pipelined Dataflow Circuits. *Int'l Conf. on Compiler Construction (CC)*, 2017.

[43] J. Wang, L. Guo, and J. Cong. AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2021.

[44] Xilinx. Vitis Unified Software Platform 2019.2. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug902-vivado-high-level-synthesis.pdf. Accessed: 2021-12-30.

[45] K. Zhan, J. Guo, B. Song, W. Zhang, and Z. Bao. UltraNet: An FPGA-based Object Detection for the DAC-SDC 2020. https://github.com/heheda365/ultra_net, 2020.

[46] Y. Zhou, U. Gupta, S. Dai, R. Zhao, N. Srivastava, H. Jin, J. Featherston, Y.-H. Lai, G. Liu, G. A. Velasquez, et al. Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software Programmable FPGAs. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2018.