HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing

Yi-Hsiang Lai[1]*, Yuze Chi[2], Yuwei Hu[1], Jie Wang[2], Cody Hao Yu[2, 3], Yuan Zhou[1], Jason Cong[2], Zhiru Zhang[1]*

[1] School of Electrical and Computer Engineering, Cornell University, USA
[2] Computer Science Department, University of California, Los Angeles, USA
[3] Falcon Computing Solutions, Inc., USA
*{yl2666,zhiruz}@cornell.edu

ABSTRACT

With the pursuit of improving compute performance under strict power constraints, there is an increasing need for deploying applications to heterogeneous hardware architectures with accelerators, such as GPUs and FPGAs. However, although these heterogeneous computing platforms are becoming widely available, they are very difficult to program especially with FPGAs. As a result, the use of such platforms has been limited to a small subset of programmers with specialized hardware knowledge.

To tackle this challenge, we introduce HeteroCL, a programming infrastructure composed of a Python-based domain-specific language (DSL) and an FPGA-targeted compilation flow. The HeteroCL DSL provides a clean programming abstraction that decouples algorithm specification from three important types of hardware customization in compute, data types, and memory architectures. HeteroCL further captures the interdependence among these different customization techniques, allowing programmers to explore various performance/area/accuracy trade-offs in a systematic and productive manner. In addition, our framework produces highly efficient hardware implementations for a variety of popular workloads by targeting spatial architecture templates such as systolic arrays and stencil with dataflow architectures. Experimental results show that HeteroCL allows programmers to explore the design space efficiently in both performance and accuracy by combining different types of hardware customization and targeting spatial architectures, while keeping the algorithm code intact.

ACM Reference Format:
Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, Zhiru Zhang. 2019. HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing. In *The 2019 ACM/SIGDA International Symposiumon Field-Programmable Gate Arrays (FPGA '19), February 24–26, 2019, Seaside, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3289602.3293910

1 INTRODUCTION

Recent trends in technology scaling have led to a growing interest in non-traditional architectures that incorporate heterogeneity and specialization as a means to improve performance under strict power and energy constraints [4, 8–10, 16]. Heterogeneous architectures with extensive use of accelerators, such as GPUs and FPGAs, have shown significant potential in this role to bring in orders-of-magnitude improvement in computing efficiency for a wide range of applications. Along this line, the latest advances in the industry have produced highly integrated heterogeneous hardware platforms, such as the CPU+FPGA multi-chip packages by Intel [19] and the GPU/FPGA-enhanced AWS cloud by Amazon [29]. Although these heterogeneous computing platforms are becoming commercially available to a wide user base, they are challenging to program, especially with FPGAs. As a result, the use of such platforms has been limited to a small subset of programmers with specialized knowledge on low-level hardware details.

To address this deficiency, recent years have seen promising development on high-level synthesis (HLS) for FPGAs [12]. This is evidenced by the availability of commercial C++/OpenCL-to-FPGA solutions (e.g., Altera/Intel SDK for OpenCL [19] and Xilinx Vivado HLS [40]) and a rapidly growing number of FPGA designs synthesized by these tools [2, 18, 35, 42]. However, programming high-performance FPGA applications with HLS tools requires a deep understanding of hardware details and is entirely different from traditional software programming. In particular, current programming models for HLS entangle algorithm specifications with hardware customization techniques. This approach has several drawbacks: (1) In order to achieve good quality-of-results (QoRs), programmers are required to use various vendor-specific data types and pragmas/directives [43], rendering FPGA-targeted applications even less flexible and portable; (2) Existing HLS programming models cannot cleanly capture the interdependence among different hardware optimization techniques, thus weakening the support of user-guided or automatic design space exploration. For example, there is no easy way to inform the HLS tool that the shape of an on-chip buffer (e.g., depth and number of banks) directly depends on the degree of parallelization; (3) HLS users need to extensively restructure the source program to guide the tool to realize specialized architectures such as data reuse buffers and systolic arrays, which are nontrivial to describe with imperative code in C/C++.

There exists an active body of work attempting to further democratize accelerator programming by using domain-specific languages (DSLs) to simplify the development and optimization of applications in certain fields. For example, Halide [32] and Spark [41] are widely used in image processing and big data analytics, respectively. Another relevant example is TVM, which is a Python-based DSL for high-performance deep learning applications [6]. Similar to Halide, TVM separates the algorithm from temporal schedule optimization (e.g., loop tiling and reordering), which significantly improves code portability across different CPU and GPU architectures.

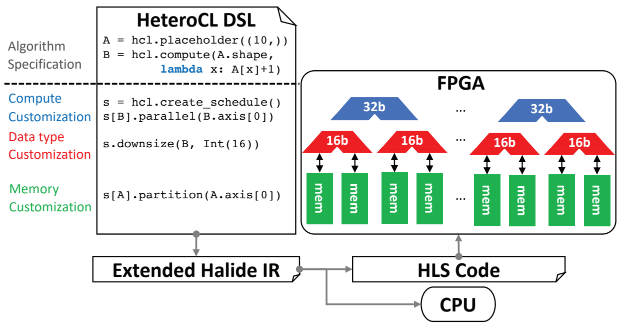
Along this direction, we propose HeteroCL — a multi-paradigm programming infrastructure for software-defined heterogeneous computing, currently targeting CPU+FPGA platforms. HeteroCL builds on the TVM framework and extends it by explicitly exposing heterogeneity in two dimensions: (1) in programming model with mixed declarative and imperative code, and (2) in optimization with decoupled algorithm and compute/data customization. HeteroCL is designed to retain the distinct strengths of each programming paradigm/customization technique, but eliminates the complexity in using them together in a single application. More concretely, our main technical contributions are as follows:

- The HeteroCL DSL provides a clean programming abstraction that decouples algorithm specification from three important types of hardware customization in compute, data types, and memory architectures. It further captures the interdependence among different types of hardware customization, enabling productive and systematic design space exploration.
- Unlike existing DSLs which primarily focus on separating algorithm specifications from temporal compute schedule, HeteroCL further supports bit-accurate types and enables decoupling between algorithms and data quantization schemes. This allows the programmer to productively express and explore the rich design trade-offs between performance/area and accuracy.
- HeteroCL nicely blends declarative symbolic expressions with imperative code. It also provides a unified interface to specify customization schemes for both declarative and imperative programs. This allows our framework to support a broad range of applications.
- The HeteroCL framework produces highly efficient spatial architectures by incorporating state-of-the-art HLS optimizations such as PolySA [13] for systolic arrays and SODA [7] for stencil with dataflow architectures. This allows productive and effective acceleration of many popular workloads from image processing and machine learning domains.
- We have developed a fully automated compilation flow from a HeteroCL program to heterogeneous compute platforms integrating CPUs and FPGAs. Our compiler generates LLVM code on CPUs and HLS code for FPGA targets (currently using the Merlin compiler [11]).

The remainder of this paper is organized as follows — In Section 2, we introduce HeteroCL with a motivating example and then describe each of its features in detail; Section 3 presents the HeteroCL compilation flow; We report the evaluation results in Section 4 and compare with related work in Section 5; Section 6 concludes this work and outlines future research directions.

2 THE PROGRAMMING MODEL

Figure 1 shows the overview of the proposed framework, where the input is a HeteroCL program composed of an algorithm specification and decoupled hardware customization schemes. We then lower it to an intermediate representation (IR) extended from Halide [32]. After that, we compile it to the back end specified by the users.



```
A = hcl.placeholder((10,))
B = hcl.compute(A.shape,
        lambda x: A[x]+1)

s = hcl.create_schedule()
s[B].parallel(B.axis[0])

s.downsize(B, Int(16))

s[A].partition(A.axis[0])
```

Figure 1: Overview of the HeteroCL framework.

HeteroCL is a Python-based DSL extended from TVM [6][1]. We choose TVM for the following reasons: (1) Python-based DSL provides programmers with a rich set of productive language features such as introspection and dynamic type system. (2) TVM is a tensor-oriented declarative DSL. Its declarative style is similar to TensorFlow [1], which compiles and executes the computation graph constructed by a programmer. This approach is beneficial for uncovering more high-level optimization opportunities in extracting parallelism and maximizing data reuse. (3) TVM inherits the idea of decoupling the algorithm specification from the temporal schedule, which is first proposed by Halide [32].

In addition to the features offered by TVM, HeteroCL further exposes heterogeneity in two dimensions: in hardware optimization techniques and programming paradigms. Figure 1 shows the key strength of HeteroCL, where HeteroCL programs can exploit various hardware optimization techniques efficiently by decoupling the algorithm specification from three classes of common hardware customization for FPGAs, which are compute, data type, and memory customization. HeteroCL further provides a clean abstraction to capture the interdependence among different optimization techniques. Moreover, HeteroCL integrates imperative programming with an embedded tensor-oriented programming model for the applications with regular parallelism. Users can choose the programming model that fits best to express a given component.

In the rest of the section, we first use a motivating example to show how HeteroCL abstracts different types of hardware customization and captures their interdependence. We then describe each customization in more detail. Finally, we present the imperative DSL in HeteroCL.

2.1 A Motivating Example

We use dot product operation as a motivating example that utilizes all three types of hardware customization. Figure 2a shows the host program, where we compute the dot product between vectors A and B. We offload function `dot_product` (L14) to FPGA for acceleration. Note that we need to batch the inputs due to FPGA on-chip resource limitation (L9). Before sending the batched inputs to the accelerator via DMA, we pack them to fully utilize the off-chip memory bandwidth (L12-13).

Figure 2b shows the optimized `dot_product` program implemented in HLS C++ code, where we apply all three types of hardware customization. First, we utilize data type customization by quantizing the data type of local buffers `local_A` and `local_B` from floating to fixed-point type `DType` (L4). By reducing the bitwidth `DW`, we increase the number of elements per memory I/O access, which shortens the

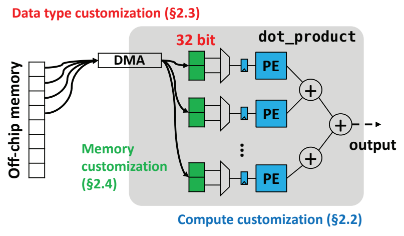
[1]Section 5 discusses the major differences between TVM and HeteroCL in more detail.

```
 1 #define N = 1024
 2 #define BATCH = 32
 3 #define MB = 64  /* off-chip memory bandwidth */
 4 #define DW = 32  /* bitwidth of the data element */
 5 #define PAR = 8  /* parallelization factor */
 6
 7 typedef MType ap_uint<MB>;
 8 void host_sw(float A[N], float B[N], float& sum){
 9   for(int i = 0; i < N; i += BATCH) {
10     MType* vec_A;
11     MType* vec_B;
12     pack(A + i, vec_A);
13     pack(B + i, vec_B);
14     sum += dot_product(vec_A, vec_B);
15   }
16 }
```

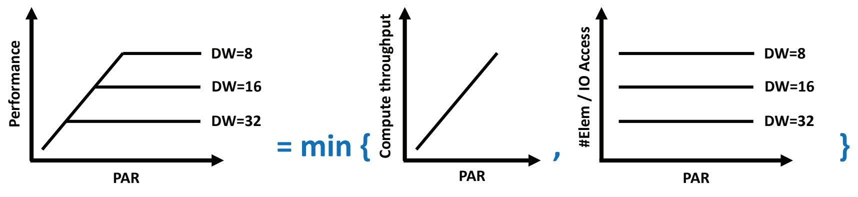
(a) Host program

```
 1 typedef DType fixed<DW, 2>;
 2
 3 DType dot_prodcut(MType* vec_A, MType* vec_B) {
 4   DType local_A[BATCH], local_B[BATCH];
 5   #pragma HLS partition variable=local_A factor=PAR
 6   #pragma HLS partition variable=local_B factor=PAR
 7   unpack(vec_A, local_A); unpack(vec_B, local_B);
 8
 9   DType psum = 0;
10   for (int i = 0; i < BATCH/PAR; i++)
11     #pragma HLS pipeline II=1
12     for (int j = 0; j < PAR; j++)
13       #pragma HLS unroll
14       psum += local_A[i*PAR+j] * local_B[i*PAR+j];
15   return psum;
16 }
```

(b) Optimized HLS code


Data type customization (§2.3)
32 bit
dot_product
Off-chip memory
DMA
PE
PE
PE
Memory customization (§2.4)
output
Compute customization (§2.2)

(c) Hardware diagram



(d) Roofline diagram

Figure 2: Motivating example: dot product — This example demonstrates the interdependence between the parallelization factor PAR and the data bitwidth DW. By tuning them with different values, the performance of the whole design can be bounded by either the compute throughput (if PAR is too small) or the number of elements per I/O access (if DW is too large).

```
 1 import heterocl as hcl
 2 # algorithm
 3 i = hcl.reduce_axis(0, BATCH)
 4 psum = hcl.compute((1,), lambda x:
 5   hcl.sum(A[i]*B[i], axis=i))
 6 # customization primitives
 7 s = hcl.create_schedule()
 8 i, j = s[psum].split(i, PAR)
 9 s[psum].pipeline(i)
10 s[psum].unroll(j)
```

(a) HeteroCL program

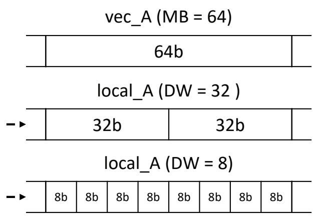
```
 1 // algorithm only
 2 for(int i = 0; i < BATCH; i++)
 3   psum += A[i] * B[i];
 4
 5 // primitives applied
 6 for(int i = 0; i < BATCH/PAR; i++)
 7   #pragma HLS pipeline II=1
 8   for(int j = 0; j < PAR; j++)
 9     #pragma HLS unroll
10     psum += A[i*PAR+j] * B[i*PAR+j];
```

(b) Equivalent HLS code

Figure 3: Example of compute customization in HeteroCL.

```
 1 # algorithm
 2 vec_A = hcl.placeholder((128,), UInt(64))
 3 local_A = hcl.unpack(vec_A)
 4
 5 # quantization scheme 1
 6 s1 = hcl.create_scheme()
 7 s1.quantize(local_A, Fixed(32, 30))
 8
 9 # quantization scheme 2
10 s2 = hcl.create_scheme()
11 s2.quantize(local_A, Fixed(8, 6))
```

(a) HeteroCL program



(b) Results after unpack

Figure 4: Example of data type customization in HeteroCL — Here we unpack the data sent from DMA vec_A to a local buffer local_A. The shape of the local buffer varies according to the quantization schemes. If we quantize local_A to a 32-bit/8-bit fixed-point buffer, each element of vec_A will be unpacked to two/eight elements in local_A.

data transfer latency but introduces the trade-off between throughput and accuracy. Second, we apply memory customization by using partition pragmas to bank the buffers (L5-6). Finally, we apply compute customization to improve the performance by tiling the loop (L10, 12) and using parallelization pragmas (L11, 13). This results in PAR processing elements (PEs) computing the multiplication and accumulation in parallel. With larger PAR, we achieve higher compute throughput with the trade-off of more on-chip resource. Moreover, there exists an interdependence between compute and memory customization, where we need to match the number of PEs with the memory banking factor. In this specific example, we set both parameters to PAR. Finally, we show the hardware diagram in Figure 2c, where we illustrate each type of hardware customization.

In addition, it is important to balance the computation time and the data communication time to maximize the hardware efficiency. Specifically, we need to carefully balance the two components by tuning the data bitwidth DW and the number of PEs PAR. We increase the compute throughput by increasing PAR. We also increase the number of elements per I/O access by lowering DW. However, the final performance is bounded by the minimum of the two. We use the Roofline [39] diagram in Figure 2d to show the relation between DW and PAR.

Figure 3 shows how we apply compute customization in HeteroCL. First, we define the algorithm in Figure 3a, where we first import the HeteroCL module (L1), define the range to be sum up (L3), and use a vector/tensor-oriented *compute operation* hcl.compute to describe the multiplication and accumulation operation that sums across i and returns a scalar (L4-5). The equivalent HLS code is shown in Figure 3b (L1-3). After that, we apply *compute customization primitives*, which are called scheduling functions in Halide/TVM, to a customization scheme created in separation of the algorithm (L7). The first primitive is a loop transformation primitive which splits loop i into a two-level nested loop i and j by a factor PAR (L8). We further apply two parallelization primitives that pipeline the outer loop i (L9) and unroll the inner loop j (L10). The equivalent code after applying customization primitives is in Figure 3b (L5-10). We can see that after applying primitives, we need to restructure the HLS code, while in HeteroCL, the algorithm specification stays unchanged.

Unlike existing DSLs, we further decouple the algorithm from data type customization. Figure 4 shows the results of applying decoupled quantization schemes in HeteroCL. In the algorithm specification, we unpack data transmitted from the 64-bit DMA vec_A to a local

```
1 # memory customization primitives
2 s = hcl.create_schedule()
3 s[local_A].partition(dim=1, factor=PAR)
```

(a) HeteroCL program

```
DType local_A[BATCH];
#pragma HLS partition variable=local_A factor=PAR
```

(b) Equivalent HLS code

Figure 5: Example of memory customization in HeteroCL.

```
 1 # algorithm specification
 2 def dot_product(vec_A, vec_B):
 3   local_A = hcl.unpack(vec_A, name="local_A")
 4   local_B = hcl.unpack(vec_B, name="local_B")
 5   i = hcl.reduce_axis(0, BATCH, "i")
 6   return hcl.compute((1,),
 7     lambda x: hcl.sum(local_A[i] * local_B[i], axis=i),
 8     name="psum")
 9
10 # exploring a range of DW and PAR
11 for DW in [4, 8, 16, 32]:
12   for PAR in [4, 8, 16, 32]:
13     # key parameters that depend on data bitwidth (DW)
14     DType = hcl.Fixed(DW, DW-2)
15     MType = hcl.UInt(MB)
16     NPACK = BATCH*DW/MB
17
18     vec_A = hcl.placeholder((NPACK,), dtype=MType)
19     vec_B = hcl.placeholder((NPACK,), dtype=MType)
20     psum = hcl.placeholder((1,), dtype=DType)
21     # data type customization
22     sm = hcl.create_scheme([vec_A, vec_B, psum], dot_product)
23     sm.quantize([dot_product.vec_A,
24                  dot_product.vec_B], DType)
25     # compute customization
26     sl = hcl.create_schedule_from_scheme(sm)
27     i, j = sl[dot_product.psum].split(dot_product.i, PAR)
28     sl[dot_product.psum].pipeline(i)
29     sl[dot_product.psum].unroll(j)
30     # memory customization
31     sl[dot_product.local_A].partition(dim=1, factor=PAR)
32     sl[dot_product.local_B].partition(dim=1, factor=PAR)
33
34     f = hcl.build(sl)
35     # evaluate f and pick the best customization scheme
36     if QoR(f) > best_QoR:
37       best_QoR = QoR(f)
38       best_scheme = sl
```

Figure 6: Complete dot product example in HeteroCL — This example demonstrates how HeteroCL explores the interdependence between the data bitwidth DW and parallelization factor PAR.

buffer `local_A` without specifying the implementation (Figure 4a L3). Then, we create a quantization scheme (L6) and quantize `local_A` to a 32-bit fixed-point buffer using a *quantization primitive* (L7). The result of unpacking is illustrated in Figure 4b. We can get a buffer with a different shape by quantizing to another bitwidth with a separate scheme (L10-11), while the algorithm stays the same.

Similar to decoupled compute and data type customization, we further decouple the algorithm from memory customization. In Figure 5a, we first create a customization scheme (L2). We then specify the *memory customization primitive* (L3). Equivalent HLS code is shown in Figure 5b.

Finally, Figure 6 shows the complete dot product kernel in HeteroCL, where we cleanly separate the algorithm specification (L1-8) from the hardware optimization specification (L14-32). We first apply data type customization to quantize the local buffers for better utilization of the off-chip memory bandwidth (L22-24). We then specify compute customization to tile and parallel the main computation for higher compute throughput (L26-29). Finally, we apply memory customization that banks the buffers to match the on-chip memory bandwidth with compute throughput (L31-32). Moreover, we use a two-level loop to explore the interdependence between DW and PAR (L11-12). We then evaluate the built kernel f generated by our back end for each pair of DW and PAR (L34) and pick the best scheme for final FPGA synthesis (L36-38).

In the following sections, we describe the syntax and semantics of HeteroCL for each type of customization in more detail.

Table 1: Compute customization primitives currently supported by HeteroCL.

Primitive	Description
	Loop transformation
`C.split(i, v)`	Split loop i of operation C into a two-level nest loop with v as the factor of the inner loop.
`C.fuse(i, j)`	Fuse two sub-loops i and j of operation C in the same nest loop into one.
`C.reorder(i, j)`	Switch the order of sub-loops i and j of operation C in the same nest loop.
`P.compute_at(C, i)`	Merge loop i of the operation P to the corresponding loop level in operation C.
	Parallelization
`C.unroll(i, v)`	Unroll loop i of operation C by factor v.
`C.parallel(i)`	Schedule loop i of operation C in parallel.
`C.pipeline(i, v)`	Schedule loop i of operation C in pipeline manner with a target initiation interval v.

```
def knn(test_img, train_img):
  diff = hcl.compute((10, 1800),
    lambda x, y: train_img[x][y] ^ test_img, "diff")
  dist = hcl.compute(diff.shape,
    lambda x, y: popcount(diff[x][y]), "dist")
  knn_mat = hcl.compute((10, 3), lambda x, y: 50, "init")
  hcl.mutate(dist.shape,
    lambda x, y: update_knn(dist, knn_mat, x, y), "update")
  return knn_mat

s = hcl.create_schedule([test_img, train_img], knn)
# loop transformation primitives
s[knn.diff].compute_at(s[knn.update], knn.update.axis[0])
s[knn.dist].compute_at(s[knn.update], knn.update.axis[0])
s[knn.update].reorder(knn.update.axis[1], knn.update.axis[0])
# parallelization primitives
s[knn.update].parallel(knn.update.axis[1])
s[knn.update].pipeline(knn.update.axis[0])
```

Figure 7: Example of combining different compute customization primitives in HeteroCL.

2.2 Compute Customization

Compute customization improves the performance of a design by performing loop transformations and executing the computation in parallel. Similar to TVM [6], we decouple the algorithm specification from compute customization schemes. Table 1 lists compute customization primitives currently supported by HeteroCL. The primitives prevent programmers from using vendor-specific pragmas, which makes HeteroCL programs portable to different back ends.

Table 2: Data types currently supported by HeteroCL.

Data Type	Description
`Int(bw)`	Bit-accurate signed integer with bw bits.
`UInt(bw)`	Bit-accurate unsigned integer with bw bits.
`Fixed(bw, fr)`	Signed fixed-point type with bw bits, where there are `fr` fractional bits.
`UFixed(bw, fr)`	Unsigned fixed-point type with bw bits, where there are `fr` fractional bits.
`Float(bw)`	Floating-point type with bw bits, where bw could be 64 or 32.

Table 3: Quantization primitives currently supported by HeteroCL.

Primitive	Description
`quantize(t, d)`	Quantize a list of tensors `t` from floating to fixed point type `d` in the format defined in Table 2.
`downsize(t, d)`	Downsize a list of tensors `t` from integers with larger bitwidth to integers `d` with smaller bitwidth in the format defined in Table 2.

Figure 7 shows an example of combining different types of compute customization primitives, where we implement KNN-based digit recognition in HeteroCL. The knn algorithm contains four operations, which are `diff`, `dist`, `init`, and `update`, respectively (L1-9). By merging different operations (L13-14), changing the loop order (L15), and applying parallelization schemes (L17-18), we can finally achieve more than 10× speedup on FPGA comparing with single-core single-thread CPU execution. We further show the step-by-step speedup results in Section 4.

2.3 Data Type Customization

Quantized computation using low-bitwidth integers and/or fixed-point types is an essential technique to achieve efficient execution on FPGAs. To represent bit-accurate data types, traditional C-based HLS tools use templates such as `ap_int<>` and `ap_fixed<>`. Although this approach allows programmers to parameterize the bitwidths, they need to run a separate script to iterate through different quantization schemes. HeteroCL addresses this challenge by utilizing Python classes to represent the data types, which allows users to try out different quantization schemes within the same program. Table 2 lists the data types currently supported by HeteroCL.

Even with the bit-accurate data type support, it remains a challenge for most application developers to determine the right data types with the right bitwidth to achieve the best trade-off between accuracy and efficiency. To solve this, HeteroCL further decouples the algorithm specification from quantization schemes. HeteroCL provides two quantization primitives in Table 3, where `quantize(t, d)` quantizes a list of floating-point variables `t` to a fixed-point type `d` whose format is defined in Table 2. In addition, `downsize(t, d)` reduces the precision of a list of integer variables `t` to an arbitrary-bit integer type `d`. With `quantize` and `downsize`, programmers can explore the trade-off between performance/area and accuracy by tuning the bitwidths of variables in the algorithm. Note that this decoupled approach is well-suited for automated bitwidth-tuning frameworks based on autotuning or rule-based heuristics. Users can further provide domain-specific knowledge such as the numerical range or the distribution of a variable to quantization primitives to guide the bitwidth searching process.

```
1  def lenet(img, w_cn1, w_cn2, w_fc1, w_fc2):
2    conv1 = conv2d_nchw(img, w_cn1, "conv1")
3    pool1 = max_pool(conv1, kernel=(2,2), stride=(2,2))
4    conv2 = conv2d_nchw(pool1, w_cn2, "conv2")
5    pool2 = max_pool(conv2, kernel=(2,2), stride=(2,2))
6    flat = flatten(pool2)
7    fc1 = dense(flat, w_fc1, "fc1")
8    fc2 = dense(fc1, w_fc2, "fc2")
9    return softmax(lenet, fc2)
10
11 for i in range(2, 33):
12   s = hcl.create_scheme([img, w_cn1, w_cn2, w_fc1, w_fc2], lenet)
13   s.quantize([lenet.conv1, lenet.conv2, lenet.fc1], Fixed(i, i-2))
14   f = hcl.build(s)
15   # run the inference and compute the accuracy
```

Figure 8: Example of exploring different quantization schemes in HeteroCL.

Table 4: Memory customization primitives currently supported by HeteroCL.

Primitive	Description
`C.partition(i, v)`	Partition dimension `i` of tensor `C` with a factor `v`.
`C.reshape(i, v)`	Pack dimension `i` of tensor `C` into words with a factor `v`.
`memmap(t, m)`	Map a list of tensors `t` with mode `m` to new tensors. The mode `m` can be either vertical or horizontal.
`P.reuse_at(C, i)`	Create a reuse buffer storing the values of tensor `P`, where the values are reused at dimension `i` of operation `C`.

Figure 8 shows an example of exploring different quantization schemes in HeteroCL, where we implement LeNet [26], a convolutional neural network (CNN) for digit recognition. The pre-trained model is in floating point. To explore different quantization schemes, we simply use a `for` loop to iterate through different bitwidths (L11). Since we know the output values of activation are between 1 and -1, we set the integer bitwidth to two (i.e., $i - (i - 2)$) (L13). We further present the evaluation results in Section 4.

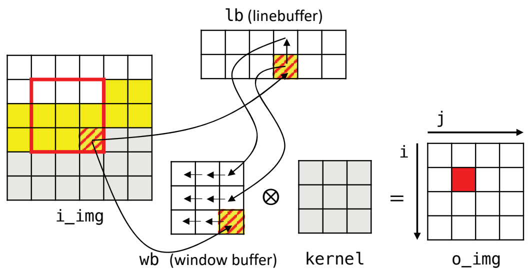
2.4 Memory Customization

Accelerating applications on FPGAs usually requires a high on-chip memory bandwidth to match the throughput of massively parallel compute units. Without customized memory architectures such as reuse buffers, the memory bandwidth could become the main hindrance preventing designs from achieving better performance. We decouple the algorithm from the memory customization and provide a set of primitives (Table 4). Moreover, programmers can apply several customization primitives in a user-defined sequence, which is not possible using pragmas supported by existing HLS tools.

Figure 9 shows an example of specifying a sequence of memory customization primitives in HeteroCL and how we define custom memory hierarchy. Figure 9a shows the implementation of 3×3 convolution in HeteroCL (L1-7). To increase the on-chip memory bandwidth, we introduce two custom reuse buffers, which are `lb` (linebuffer) and `wb` (window buffer), respectively (L10-11). We introduce data reuse between tensors/buffers via the `reuse_at` primitive. Specifically, Lines 10 and 11 specify that `lb` reads values from `i_img` and `wb` reads values from `lb`, respectively. Figure 9b illustrates how the reuse buffers operate, with the arrows indicating the data movement in each cycle. The HeteroCL compiler automatically infers the shapes of the buffers and the data to be stored based the reuse axis specified in the primitive. In this example, since `wb` reuses at

```
 1 # algorithm specification
 2 def conv(i_img, kernel):
 3     ri = hcl.reduce_axis(0, 3, 'ri')
 4     rj = hcl.reduce_axis(0, 3, 'rj')
 5     return hcl.compute((N-2, N-2), lambda i, j: hcl.sum(
 6             i_img[i+ri, j+rj] * kernel[ri, rj],
 7             axis=[ri, rj]), name='o_img')
 8 # memory customization - custom reuse buffers
 9 s = hcl.create_scheme([i_img, kernel], conv)
10 lb = s[i_img].reuse_at(conv.o_img, conv.i)
11 wb = s[lb].reuse_at(conv.o_img, conv.j)
```

(a) HeteroCL code



(b) Illustration of how reuse buffers operate in each cycle

Figure 9: Example of defining custom reuse buffers and their hierarchy in HeteroCL.

Table 5: Spatial architecture macros currently supported by HeteroCL.

Primitive	Description
`C.stencil()`	Specify operation C to be implemented with stencil with dataflow architectures using the SODA framework.
`C.systolic()`	Specify operation C to be implemented with systolic arrays using the PolySA framework.

loop `j` of operation `o_img` (L11), it contains data that are read in a single iteration of loop `j`, which corresponds to the red box in `i_img`. Similarly, the compiler infers the shape of `lb`, which stores the values of the yellow pixels in `i_img`.

2.5 Mapping to Spatial Architecture Templates

Many popular workloads from image/video processing and machine learning domains can be realized in a highly efficient manner on FPGAs using spatial architectures such as systolic arrays [33, 38]. However, with the traditional C-based HLS methodology, it typically requires extensive code restructuring and the insertion of a right combination of pragmas to guide the tool to generate a high-performance spatial architecture. This tedious and error-prone process is one of the major barriers for the mainstream adoption of HLS for FPGA designs. HeteroCL addresses this deficiency by introducing a set of optimization macros that synthesize the code into highly efficient spatial architecture templates. Each of these macros consists of a combination of compute and memory customization primitives. As indicated in Table 5, we currently support stencil with dataflow architectures and systolic arrays, each of which is described in more detail as follows.

Stencil with Dataflow Architecture – Stencil computation is commonly seen in many areas including image processing and numerical computing, where data elements are updated over a multi-dimensional grid according to some fixed, local patterns. HeteroCL incorporates the SODA framework [7], which synthesizes stencil

```
1 # algorithm specification
2 def jacobi(in_):
3   return hcl.compute(in_.shape, lambda y, x:
4       (in_[y, x-1] + in_[y-1, x] + in_[y, x] +
5        in_[y, x+1] + in_[y+1, x])/5), "out")
6 # apply compute customization and stencil macro
7 s = hcl.create_schedule([in_], jacobi)
8 s[jacobi.out].unroll(jacobi.out.axis[1], factor=3)
9 s[jacobi.out].stencil()
```

Figure 10: Example of mapping computations to stencil with dataflow architectures in HeteroCL.

patterns to a highly efficient dataflow architecture composed of reuse buffers and data streams. Figure 10 shows an example of mapping a Jacobi kernel to the stencil with dataflow architecture by using the macro `s[jacobi.out].stencil()` specified in Line 9. Given this macro, the HeteroCL back-end compiler will automatically identify the stencil pattern within the computation of `out` (L3-5) and synthesize the corresponding spatial architecture with the stencil back end (elaborated in Section 3).

```
1 # algorithm specification
2 def mmult(A, B):
3   k = hcl.reduce_axis(N)
4   return hcl.compute(A.shape,
5     lambda x, y: hcl.sum(A[x, k] * B[k, y], axis=k), "C")
6 # map to systolic arrays
7 s = hcl.create_schedule([A, B], mmult)
8 s[mmult.C].systolic()
```

Figure 11: Example of mapping a computation to systolic arrays in HeteroCL.

Systolic Array – HeteroCL further provides efficient support for mapping to systolic arrays, which are widely used spatial architectures that consist a group of processing elements locally connected to each other [25]. Featuring local interconnects and modular designs, the systolic array architecture is highly scalable and can take advantage of the enormous amount of computation resources on modern FPGAs. It is particularly suitable for applications having perfectly nested loops with uniform dependency, such as matrix-matrix multiplication. However, it is a complex task to manually create systolic array designs on FPGAs. Recent research from Intel reports that it takes several to tens of months of human effort to implement a high-performance systolic array design, even with an HLS design entry like OpenCL [33].

Similar to stencil optimization, we introduce a `systolic` macro in the HeteroCL DSL to allow convenient mapping from tensor code to systolic array architectures. Figure 11 shows an example of using the `systolic` macro for matrix-matrix multiplication. Here we specify the computation of `C` (L8) to be synthesized with a specialized back end, which incorporates the PolySA framework [13] for automatic systolic array generation (discussed in Section 3).

2.6 Mixed Declarative and Imperative Programming

HeteroCL blends imperative programming with an embedded declarative, symbolic style for expressing tensor-based code. The idea is to combine the advantages of both styles — Imperative programming is general and flexible, while symbolic tensorized code exposes higher-level optimization opportunities when the code allows. The flexibility offered by imperative programming enables designers to implement algorithms with less-regular parallelism that cannot be

```
BinOp   := + | - | * | / | % | & | ^ | >> | <<
BinEqOp := += | -= | *= | /=
CompOp  := > | >= | == | <= | = | < | !=
Expr    := Var | Tensor[Expr] | Number
           | not Expr | Expr BinOp Expr
           | Expr[Expr] # get bit
           | Expr[Expr:Expr] # get slice of bits
Cond    := Expr Comp Epxr | hcl.and_(*Cond) | hcl.or_(*Cond)
CondStmt:= hcl.if_(Cond) | hcl.elif_(Cond) | hcl.else_()
Stmt    := Tensor[Expr] = Expr | Tensor[Expr] BinEqOp Expr
           | Expr[Expr] = Expr # set bit
           | Expr[Expr:Expr] = Expr # set slice of bits
           | with CondStmt:
               Stmt
           | with hcl.for_(Expr, Expr, Expr) as Var:
               Stmt
           | # HeteroCL compute operations (e.g., compute)
           | # And more
```

Figure 12: Imperative DSL in HeteroCL — We provide equivalent semantics for commonly used expressions and statements in normal Python. We also support bit-level operations for bit-accurate data types. The imperative DSL highly resembles normal Python in that they use same indentations, same rules for variable scope, and similar keywords. This relieves new users from learning a whole new set of syntax and semantics.

```
 1 A = hcl.compute((10,), lambda x: x)
 2 def popcount(num):
 3   out = hcl.compute((1,), lambda x: 0, "out")
 4   with hcl.for_(0, A.type.bits, 1, "loop1") as i:
 5     out[0] += num[i] # bit selection operation
 6   return out[0]
 7 # extended TVM compute operation
 8 B = hcl.compute(A.shape, lambda x: popcount(A[x]))
 9 # decoupled data type customization
10 sm = hcl.create_scheme()
11 sm.downsize(B.out, h.UInt(4))
12 # decoupled compute customization
13 sl = hcl.create_schedule_from_scheme(sm)
14 sl[B].unroll(B.loop1)
```

Figure 13: Example of applying hardware customization to imperative DSL — We extend existing compute operations (e.g., `compute`) and customization primitives (e.g., `unroll`) in declarative code to further support imperative programming.

efficiently represented with declarative programming (e.g., sorting algorithms). It also gives full control to programmers for specifying the algorithmic details.

Some of the existing Python-based DSLs use normal Python to support imperative programming, such as TVM [6] and Hot&Spicy [34]. This approach, however, has some drawbacks: (1) The normal Python semantic is too flexible to be FPGA synthesizable. (2) A designated parser/compiler must be built, which could be error-prone. Instead of using normal Python to support imperative programming, HeteroCL provides an imperative DSL listed in Figure 12. HeteroCL further extends existing compute operations (e.g., `compute`) and develops new operations for mixed-paradigm programming. We show examples of supported compute operations in Table 6.

Programmers can also apply hardware customization to the imperative DSL. For instance, Figure 13 shows the popcount algorithm implemented using the imperative DSL, where we apply both compute and data type customization. Programmers access the vectors and loops declared within a HeteroCL compute operation by their names. For example, `B.out` in Line 11 refers to the vector `out` declared in Line 3; `B.loop1` in Line 14 refers to the `for` loop `loop1` in Line 4. Note that the algorithm behaves differently with different quantization schemes, where the bound of `loop1` is determined by the bitwidth of `A` (Line 4).

Table 6: Compute operations currently supported by HeteroCL.

Operation	Description
`compute(s, f)`	Compute a new tensor of shape `s`. The value of each element in the new tensor is calculated according to lambda function `f`.
`update(t, f)`	Update each element of tensor `t` according to lambda function `f`.
`mutate(s, f)`	Write a for loop of shape `s` in vector code, where `f` is a lambda function describing the for loop body.

3 BACK-END CODE GENERATION AND OPTIMIZATION

The HeteroCL framework has multiple back end supports including CPU and HLS flows targeting FPGA. Specifically, we extend the Halide IR used by TVM [6, 32] for our multi-paradigm programming model and customization primitives. The extended Halide IR serves as a unified representation for all back-end flows. In this section, we briefly summarize our FPGA back-end code generation flow.

General Back End – The HeteroCL compiler can generate a corresponding accelerator kernel in many languages, including HLS C/C++, OpenCL, and Merlin C. Merlin C is an OpenMP-like programming model used by the Merlin compiler [11] from Falcon Computing Solutions. We choose the Merlin compiler as one of our back-end tools for two reasons. First, it leverages a small set of OpenMP-like pragmas to apply certain architecture structures by source-to-source C code transformation. Since Merlin pragmas share lots of similarity with HeteroCL customization primitives, it is relatively straightforward to integrate with HeteroCL. Second, the Merlin compiler generates both HLS C kernels and OpenCL kernels for FPGAs from the unified Merlin C source code.

Table 8 shows the correspondence between HeteroCL primitives and Merlin C pragmas. The primitive `unroll` implies fine-grained parallelism, which indicates all loop body logic to be scheduled in the same hardware module. As a result, all sub-loops in the target loop is flattened if a user applies `unroll` to a non-innermost loop. On the other hand, the primitive `parallel` indicates coarse-grained parallelism (e.g., a PE array). In addition, if a `pipeline` primitive is assigned to a non-innermost loop, we map it to a coarse-grained pipeline architecture.

Since HeteroCL primitives and Merlin C pragmas mainly specify loop scheduling or memory organization, the implied architecture can be represented as a composable, parallel, and pipeline (CPP) architecture [14]. The authors in [14] have demonstrated that the CPP architecture can be applied to broad classes of applications with a good performance.

Stencil Back End – We incorporate the SODA framework proposed in [7] to implement stencil patterns with optimized dataflow architecture that minimizes the on-chip reuse buffer size. SODA takes in a lightweight DSL that describes the stencil compute patterns and design parameters. After the HeteroCL compiler identifies stencil patterns according to user-specified macros, it generates the proper DSL code to the SODA framework for hardware generation. In addition, hardware customization primitives such as loop unrolling and data quantization are also reflected in the SODA DSL as design parameters, which in turn guide the SODA framework for further optimization.

Table 7: Evaluation results of benchmarks in HeteroCL — The speedup is over a single-core single-thread CPU execution.

Benchmark	Data Sizes & Type	# LUTs	# FFs	# BRAMs	# DSPs	Freqency (MHz)	Speedup	Back End
KNN Digit Recognition [43] Image classification	K=3 #images=1800 `uint49`	4009	5835	88	0	250	12.5	General
K-Means Clustering	K=16 #elem=320 × 32 `int32`	212708	235011	32	1536	190.6	16.0	General
Smith-Waterman [36] Genomic sequencing	string len=128 `uint2`	110841	88369	1409	0	152.2	20.9	General
Seidel [30] Image processing / linear algebra	2160 pixel × 3840 pixel `fixed16`	21719	31663	46	96	250	5.9	Stencil
Gaussian [30] Image processing	2160 pixel × 3840 pixel `fixed16`	70833	131160	46	688	250	13.2	Stencil
Jacobi [30] Linear algebra	2160 pixel × 3840 pixel `fixed16`	14883	22485	46	48	250	5.0	Stencil
GEMM Matrix-matrix multiplication	1024 × 1024 × 1024 `fixed16`	454492	800283	932	2507	236.8	8.9	Systolic Array
LeNet Inference [26] Convolutional neural network	MNIST [15] `fixed16`	362291	660186	739.5	1368	250	10.6	Systolic Array

Table 8: Correspondence between HeteroCL primitives and Merlin C pragmas.

`unroll(i, v)` ⟶ `#pragma ACCEL parallel flatten factor=v` Partial unroll the target loop by factor `i` and fully unroll all its sub-loops.
`parallel(i)` ⟶ `#pragma ACCEL parallel` Wrap the body of loop `i` to a function and form a PE array.
`pipeline(i, v)` ⟶ `#pragma ACCEL pipeline` Wrap the body of loop `i` to a function and form a load-compute-store coarse-grained pipeline.

Systolic Array Back End – Similar to the stencil back end, our compiler analyzes the user-specified `systolic` macros and generates annotated HLS C++ code as an input to the PolySA framework [13], which further performs automated design space exploration that optimizes the systolic array architecture including the shape of it and the interconnection between PEs.

4 EVALUATION

In this section, we evaluate the accelerators generated by HeteroCL. The platform we target is the AWS EC2 `f1.2xlarge` instance, which has 8 vCPU cores, 122GiB main memory, and a Xilinx Virtex UltraScale+™ VU9P FPGA. The default target frequency for this platform is 250 MHz.

We select several common FPGA benchmarks from a broad range of applications that are applied with either general, stencil, or systolic array back ends. For the general back end, we have (1) KNN-based digit recognition, which is simplified from that of Rosetta [43], (2) K-means algorithm, and (3) Smith-Waterman [36]. For the stencil back end, we have (1) Gaussian, (2) Jacobi, and (3) Seidel. All of them are from Polybench [30]. For the systolic back end, we use (1) general matrix multiplication (GEMM) and (2) deep learning inference with LeNet model [26]. Among these benchmarks, KNN-based digit recognition, K-means, and Smith-Waterman need to be implemented with the HeteroCL imperative DSL.

Table 7 shows the benchmarks and the overall evaluation results. We run the baseline designs on one CPU core with a single thread. For the two systolic benchmarks, we are comparing our FPGA implementations with the GEMM function provided in Intel MKL [20] and a LeNet model optimized with TVM [6], respectively. We include memory transfer time (i.e., between DDR4 and FPGA) as part of the total run time. After applying proper customization primitives, we achieve up to 20.9× speedup for the benchmarks with the general back end. Moreover, we can achieve up to 13.2× and 10.6× speedup for benchmarks applied with stencil and systolic array back end, respectively. In the rest of this section, we show the detailed evaluation of each back end.

Table 9: Speedup over CPU with customization primitives.

Benchmark	No Primitive	+Parallel	+Loop Transform
KNN Digit Recognition	1.2	1.6	12.5
K-Means	1.9	2.8	16.0
Smith-Waterman	0.7	20.9	20.9

Table 10: Speedup over CPU for the stencil back end – To achieve a higher speedup, we apply both compute and data type customizations on top of `stencil` macro. The ideal speedup is determined by the maximum off-chip memory bandwidth.

Benchmark	`+stencil`	`+unroll`	`+quantize`	Ideal
Seidel	0.5	2.9	5.9	6.8
Gaussian	1.1	6.7	13.2	15.6
Jacobi	0.4	2.3	5.0	5.4

General Back End – We first evaluate the impact of HeteroCL customization primitives on performance with the general back end. Table 9 shows the speedup of generated accelerator kernels after step-by-step applications of customization primitives. We first show the speedup without applying any customization primitive and that after applying parallelization primitives such as `unroll` and `parallel`. However, without applying appropriate loop transformation primitives such as `split` and `reorder`, the performance improvement could be limited. Thus, we also show the results after applying those primitives to the benchmarks. Table 9 demonstrates the permutability of HeteroCL, where programmers can easily explore the design space just by adding or removing primitives without changing the algorithm code.

Stencil Back End – Table 10 shows the speedup of the benchmarks after applying the `stencil` macro, customization primitives, and the ideal speedup. If we only apply the macro, we are only up to 1.1× faster because of the limited parallelism. To improve the performance, we apply parallelization primitives (i.e., `unroll`), which

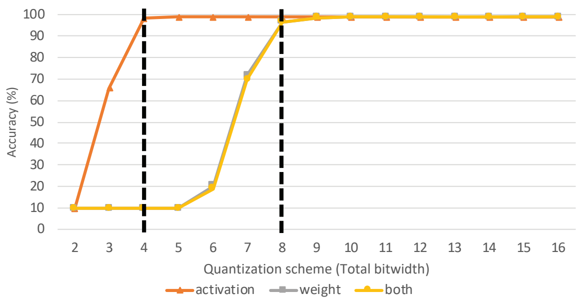


Figure 14: Accuracy of LeNet with different quantization schemes.

Table 11: Performance results of systolic array benchmarks – The performance is in terms of Giga operations per second (GOPs).

Benchmark	Backend	Data Type	Performance (GOPs)	Speedup
LeNet Inference	CPU [6]	`float32`	15.4	1.0
	FPGA	`float32`	79.8	5.2
		`fixed16`	137.8	8.9
GEMM	CPU [20]	`float32`	76.0	1.0
	FPGA	`float32`	245.9	3.2
		`fixed16`	807.6	10.6

results in up to 6.7× speedup. At this stage, since the performance bottleneck of all benchmarks is the off-chip memory bandwidth (about 13.8GB/s), we cannot get higher throughput by further increasing the unrolling factor. To address this, we quantize the single-precision floating-point numbers to 16-bit fixed-point numbers. As a result, the required external memory bandwidth could be reduced and we can achieve an additional 2× speedup. As a reference, the last column shows the ideal speedup assuming the memory bandwidth is perfectly utilized. In summary, Table 10 shows that by combining spatial architecture macros with other types of hardware customization, we can further improve the performance.

Systolic Array Back End – We finally evaluate the benchmarks applied with `systolic` macros. For both applications, we evaluate the impact of data type customization on performance, which we present in Table 11. By using both spatial architecture macros and data type customization primitives, we can improve the performance of both designs.

We further show the accuracy results after applying different quantization schemes to LeNet benchmark in Figure 14, where the X-axis shows the number of total bitwidth and the Y-axis shows the accuracy. We demonstrate three different scenarios, which are quantizing the activation, weights, and both, respectively. We observe that with 8-bit fixed-point type, the accuracy degradation is marginal. Moreover, if we choose to quantize the activation only, 4-bit fixed-point type is the best choice.

5 RELATED WORK

There exists a large body of work on HLS and domain-specific programming. In this section, we survey a small subset of representative efforts on C-based HLS, DSLs for hardware accelerator designs, and those that support decoupled algorithm and optimizations.

C-based HLS – HLS tools such as LegUp [5], Intel FPGA SDK [21], and Xilinx Vivado HLS [40] allow developers to write FPGA designs in C/C++ and OpenCL, delivering higher productivity than traditional register-transfer-level (RTL) designs. The recently introduced Merlin compiler greatly simplifies the HLS design by applying source-to-source transformation to automatically generate optimized HLS-C or OpenCL programs [11]. However, to achieve good QoRs, developers are required to use various vendor-specific data types and pragmas/directives, rendering FPGA design with HLS less flexible and portable.

HeteroCL lifts the abstraction level of FPGA programming and provides developers with a systematic way to efficiently explore various trade-offs, making FPGA design more portable and productive.

DSLs for Hardware Accelerator Design – There is a growing interest in compiling programs written in high-level languages (e.g., Python, Scala) into reconfigurable hardware accelerators. Hot & Spicy compiles annotated Python code into HLS C/C++, where the annotations are translated into pragmas [34]. DHDL introduces a representation of hardware using parameterized templates in Scala that captures locality and parallelism information and compiles the representation into FPGAs and CGRAs [24]. Spatial extends DHDL by adding a set of low-level abstractions for control and memory [23]. However, in these DSLs the algorithm specification is tightly entangled with hardware optimizations, making design space exploration less productive.

HeteroCL decouples algorithm specification from hardware customization, and abstracts three important types of hardware customization into a set of customization primitives, enabling productive and systematic design space exploration. HeteroCL further offers additional macros in `stencil` and `systolic` for efficient mapping to highly optimized spatial architecture templates.

DSLs with Decoupled Algorithm and Optimization – Most computing patterns in image processing and deep learning can be concisely described as nested loops in a declarative programming paradigm, as illustrated in a lot of DSLs [17, 22, 27, 37]. Halide first proposes to decouple the algorithm specification from the temporal schedule [32]. Tiramisu extends Halide by adding explicit communication, synchronization, and mapping buffers to different memory hierarchies [3, 33]. Jing Pu, et al. also extend Halide to support custom reuse buffers and support FPGAs and CGRAs as back end [31]. T2S extends Halide by decoupling the spatial schedule from the algorithm specification, which allows programmers to define systolic-array-like architectures [33]. TVM builds a deep learning compiler stack on top of Halide IR, supporting both CPUs and GPUs [6]. While the declarative programming paradigm in these DSLs is powerful, it cannot express applications beyond image processing and deep learning.

HeteroCL, as a multi-paradigm programming infrastructure, nicely blends declarative symbolic expressions with imperative code, and provides a unified interface to specify customization schemes for both declarative and imperative programs. This allows HeteroCL to support a broader range of applications.

More specifically, we list the major differences between TVM and HeteroCL as follows: (1) TVM extensively uses declarative programming to target deep learning applications, while HeteroCL supports mixed imperative and declarative programming to target general applications. (2) TVM tries to solve the optimization challenges mainly for CPUs and GPUs, while HeteroCL focuses on hardware customization for FPGA and incorporates advanced spatial architecture templates. (3) TVM programs can target FPGAs as back end by using VTA, a programmable accelerator that uses a RISC-like programming abstraction to describe tensor operations [28]. On the other hand, HeteroCL programs are not limited to tensor operations.

In addition, programmers can apply various hardware customization techniques with provided primitives while the hardware generated by VTA is pre-defined. (4) HeteroCL supports bit-accurate data types, which are not available in TVM. Furthermore, HeteroCL proposes to decouple the quantization scheme from algorithm specification.

6 CONCLUSIONS AND FUTURE WORK

We have presented HeteroCL, a multi-paradigm programming infrastructure for heterogeneous platforms integrating CPUs and FPGAs. HeteroCL not only provides a clean abstraction that decouples the algorithm from compute/data customization, but it also captures the interdependence among them. Moreover, HeteroCL incorporates spatial architecture templates including systolic arrays and stencil with dataflow architectures. We believe HeteroCL can help developers to focus more on designing efficient algorithms rather than being distracted by low-level implementation details.

We are releasing the proposed framework in an open-source format. The programming infrastructure as well as the associated documents and example designs are publicly available on the authors' website. Additionally, we plan to introduce primitives for data and device placement, and also data streaming interfaces. We will also integrate HeteroCL with more spatial architecture templates, distributed autotuning capabilities, and accurate QoR estimation boosted by machine learning techniques.

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/Intel CAPA Award #1723773, DARPA Young Faculty Award D15AP00096, NSF Awards #1453378, #1436827, and #1707408, and research gifts from Intel and Xilinx. We thank Amazon for providing AWS EC2 credits. We thank Prof. Adrian Sampson (Cornell), Dr. Hongbo Rong (Intel), and Dr. Justin Gottschlich (Intel) for providing helpful feedback on the HeteroCL framework. We also thank Ritchie Zhao, Ziyan Feng, Shaojie Xiang, Yichi Zhang, Patrick Clobridge, and Qing Yu for their contributions to the HeteroCL benchmarks.

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. *arXiv preprint arXiv:1603.04467*, 2016.

[2] A. Althoff and R. Kastner. A Scalable FPGA Architecture for Nonnegative Least Squares Problems. *Int'l Conf. on Field Programmable Logic and Applications (FPL)*, 2015.

[3] R. Baghdadi, J. Ray, M. B. Romdhane, E. Del Sozzo, P. Suriana, S. Kamil, and S. Amarasinghe. Tiramisu: A Code Optimization Framework for High Performance Systems. *arXiv preprint arXiv:1804.10694*, 2018.

[4] S. Borkar and A. A. Chien. The Future of Microprocessors. *Communications of the ACM*, 2011.

[5] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski. LegUp: High-level Synthesis for FPGA-Based Processor/Accelerator Systems. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2011.

[6] T. Chen, T. Moreau, Z. Jiang, H. Shen, E. Yan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. TVM: End-to-End Optimization Stack for Deep Learning. *arXiv preprint arXiv:1802.04799*, 2018.

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

[8] A. A. Chien, A. Snavely, and M. Gahagan. 10x10: A General-Purpose Architectural Approach to Heterogeneity and Energy Efficiency. *Procedia Computer Science*, 2011.

[9] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai. Single-Chip Heterogeneous Computing: Does the Future Include Custom Logic, FPGAs, and GPGPUs? *Int'l Symp. on Microarchitecture (MICRO)*, 2010.

[10] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, K. Gururaj, and G. Reinman. Accelerator-Rich Architectures: Opportunities and Progresses. *Design Automation Conf. (DAC)*, 2014.

[11] J. Cong, M. Huang, P. Pan, D. Wu, and P. Zhang. Software Infrastructure for Enabling FPGA-Based Accelerations in Data Centers. *Int'l Symp. on Low Power Electronics and Design (ISLPED)*, 2016.

[12] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2011.

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

[14] J. Cong, P. Wei, C. H. Yu, and P. Zhang. Automated Accelerator Generation and Optimization with Composable, Parallel and Pipeline Architecture. *Design Automation Conf. (DAC)*, 2018.

[15] L. Deng. The MNIST Database of Handwritten Digit Images for Machine Learning Research. *IEEE Signal Processing Magazine*, 2012.

[16] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. *Int'l Symp. on Computer Architecture (ISCA)*, 2011.

[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] G. Inggs, S. Fleming, D. Thomas, and W. Luk. Is High Level Synthesis Ready for Business? A Computational Finance Case Study. *Int'l Conf. on Field Programmable Technology (FPT)*, 2014.

[19] Intel. Xeon+FPGA Platform for the Data Center. https://www.ece.cmu.edu/ calcm/carl/lib/exe/fetch.php? media=carl15-gupta.pdf.

[20] Intel. Intel Math Kernel Library. 2007.

[21] Intel. Intel High Level Synthesis Compiler User Guide. 2017.

[22] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe. The Tensor Algebra Compiler. *Intl'l Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, 2017.

[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] D. Koeplinger, R. Prabhakar, Y. Zhang, C. Delimitrou, C. Kozyrakis, and K. Olukotun. Automatic Generation of Efficient Accelerators for Reconfigurable Hardware. *Int'l Symp. on Computer Architecture (ISCA)*, 2016.

[25] H. Kung and C. E. Leiserson. Systolic Arrays (for VLSI). *Sparse Matrix Proceedings*, 1979.

[26] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 1998.

[27] R. Membarth, O. Reiche, F. Hannig, J. Teich, M. Körner, and W. Eckert. Hipa^{cc}: A Domain-Specific Language and Compiler for Image Processing. *IEEE Transactions on Parallel and Distributed Systems*, 2016.

[28] T. Moreau, T. Chen, Z. Jiang, L. Ceze, C. Guestrin, and A. Krishnamurthy. VTA: An Open Hardware-Software Stack for Deep Learning. *arXiv preprint arXiv:1807.04188*, 2018.

[29] D. Pellerin. Fpga accelerated computing using aws f1 instances. *AWS Public Sector Summit*, 2017.

[30] L.-N. Pouchet. Polybench: The Polyhedral Benchmark Suite. *URL: http://www. cs. ucla. edu/pouchet/software/polybench*, 2012.

[31] 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.

[32] 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 SIGPLAN Notices*, 2013.

[33] H. Rong. Programmatic Control of a Compiler for Generating High-Performance Spatial Hardware. *arXiv preprint arXiv:1711.07606*, 2017.

[34] S. Skalicky, J. Monson, A. Schmidt, and M. French. Hot & Spicy: Improving Productivity with Python and HLS for FPGAs. *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2018.

[35] Z. Wang, B. He, and W. Zhang. A Study of Data Partitioning on OpenCL-Based FPGAs. *Int'l Conf. on Field Programmable Logic and Applications (FPL)*, 2015.

[36] M. Waterman. Identification of Common Molecular Subsequence. *Mol. Biol*, 1981.

[37] R. Wei, V. Adve, and L. Schwartz. DLVM: A Modern Compiler Infrastructure for Deep Learning. *arXiv preprint arXiv:1711.03016*, 2017.

[38] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong. Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs. *Design Automation Conf. (DAC)*, 2017.

[39] S. Williams, A. Waterman, and D. Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Communications of the ACM*, 2009.

[40] Xilinx Inc. Vivado Design Suite User Guide: High-Level Synthesis. 2012.

[41] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. *HotCloud*, 2010.

[42] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin, M. Srivastava, R. Gupta, and Z. Zhang. Accelerating Binarized Convolutional Neural Networks with Software-Programmable FPGAs. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2017.

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