T2S-Tensor : Productively Generating High-Performance Spatial Hardware for Dense Tensor Computations

Nitish Srivastava*, Hongbo Rong†, Prithayan Barua‡, Guanyu Feng§, Huanqi Cao§, Zhiru Zhang*, David Albonesi*,
Vivek Sarkar‡, Wenguang Chen§, Paul Petersen†, Geoff Lowney†, Adam Herr†, Christopher Hughes†,
Timothy Mattson† and Pradeep Dubey†
*Cornell University. Email:{nks45, zhiruz, dha7}@cornell.edu
†Intel. Email:{hongbo.rong, paul.petersen, geoff.lowney, adam.w.herr, christopher.j.hughes, timothy.g.mattson, pradeep.dubey}@intel.com
‡Georgia Institute of Technology. Email:{prithayan, vsarkar}@gatech.edu
§Tsinghua University. Email:{fgy18, caohq18}@mails.tsinghua.edu.cn, cwg@tsinghua.edu.cn

***Abstract*— We present a language and compilation framework for productively generating high-performance systolic arrays for dense tensor kernels on spatial architectures, including FPGAs and CGRAs. It decouples a functional specification from a spatial mapping, allowing programmers to quickly explore various spatial optimizations for the same function. The actual implementation of these optimizations is left to a compiler. Thus, productivity and performance are achieved at the same time.**

We used this framework to implement several important dense tensor kernels. We implemented dense matrix multiply for an Arria-10 FPGA and a research CGRA, achieving 88% and 92% of the performance of manually written, and highly optimized expert ("ninja") implementations in just 3% of their engineering time. Three other tensor kernels, including MTTKRP, TTM and TTMc, were also implemented with high performance and low design effort, and for the first time on spatial architectures.

I. Introduction

High-performance computing (HPC) on spatial architectures tends to be limited by very low design productivity — it is not unusual for industry experts to spend several months or even more than a year to deliver one seemingly simple kernel with good quality of results (QoRs) [1]. While HPC programming is presumably challenging on any architecture including CPUs/GPUs, it is especially painful on spatial architectures like field-programmable gate arrays (FPGAs) due to their much longer compile time and primitive debugging support. In addition, FPGA designs often require construction of specialized user-managed on-chip memory hierarchy, which significantly increases the programming complexity. Coarse-grain reconfigurable architectures (CGRAs) have been proposed to address the slow compilation problem, but how to efficiently program CGRAs remains an open question.

To address these long-standing challenges, we propose a novel programming system consisting of a language and compiler, named T2S (Temporal To Spatial), for productively generating high-performance spatial hardware. The work in this paper is named T2S-Tensor, since its main focus is tensor computations using T2S. We observe that *a computation suitable for a spatial architecture is usually a dataflow function, and a high-performance spatial design partitions the computation into many sub-computations. These sub-computations are distributed over the spatial architecture, and are connected with channels, i.e. FIFOs. They run in parallel, and are individually optimized by a series of loop and data transformations.* Based on this observation, T2S enables programmers to describe a computation in a functional notation, followed by a spatial mapping that describes compute partition and loop and data transformations. A compiler then composes these optimizations and synthesizes them to run on a spatial architecture. Thus, T2S allows programmers to succinctly *specify* different optimizations and leave the actual implementation of the optimizations to the compiler.

We focus on accelerating dense tensor computations to demonstrate the effectiveness of our proposal. Tensors are a generalization of two-dimensional matrices to higher dimensions. Dense tensor algebra is a powerful tool for computing multi-dimensional data and has many applications in machine learning, data analytics, engineering, and scientific computing [2]–[6]. Well-known dense tensor kernels include general matrix multiply (GEMM), tensor times matrix (TTM), matricized tensor times khatri-rao product (MTTKRP), and tensor times matrix-chain (TTMc). These dense tensor kernels have regular memory access patterns and high parallelism, making them a good match for spatial architectures. Yet how to productively accelerate the tensor kernels on spatial architectures for high performance remains a challenge. Firstly, tensors often have many dimensions and a tensor kernel can involve many tensors. Secondly, every tensor kernel has many possible designs. Implementing any design efficiently on spatial hardware like FPGAs usually takes significant engineering effort.

In this work, we show that T2S enables dense tensor kernels to be succinctly expressed and effectively optimized for spatial architectures. Our major technical contributions are as follows:

- We propose a concise yet expressive programming abstraction that decouples a spatial mapping from a functional specification. The spatial mapping can direct the compiler to realize many sophisticated optimizations, achieving productivity and performance at the same time.
- We identify a set of key compiler optimizations that are essential for creating high-performance spatial hardware for dense tensor computations, and implement them in a comprehensive compilation framework. Our compiler further provides composability of these optimizations, where various combinations of transformations can be applied and the compiler automatically generates the correct low-level code.
- We demonstrate the efficacy of our approach on an Arria-10 FPGA as well as a research CGRA by generating OpenCL

and assembly code for the two architectures, respectively. Our GEMM implementations achieve 88% and 92% of the performances achieved by codes that were manually written, highly optimized, and extensively tuned by experts, with only around 3% of engineering time (two weeks vs. 18 months on the FPGA, and three days vs. three months on the CGRA). The T2S GEMM implementation on the Arria-10 FPGA is also 76% faster than a (tuned) NDRange-style OpenCL implementation [7].

- Using our system, three other important tensor kernels, including MTTKRP, TTM, and TTMc, were also implemented in a productive and high-performance manner, and for the first time on spatial architectures, to the best of our knowledge.

The rest of the paper is organized as follows: Section II gives a brief overview of our programming model, Section III illustrates the spatial optimizations in our system with GEMM as a working example, Section IV describes our compiler flow and optimizations, followed by experimental results in Section V, and related work in Section VI. Finally, we conclude the paper in Section VII.

II. Overview of The Programming Model

A T2S program is a specification consisting of two parts, a temporal definition, and a spatial mapping. The former defines a computation functionally, while the latter specifies how to map the computation to a spatial architecture. In this paper, we focus on the programming model and the associated compilation framework, not the language syntax[1]. Hence we will describe the programming abstraction with small intuitive examples to facilitate understanding. The same principle can be applied to construct very sophisticated spatial designs.

Fig. 1(a) shows a trivial computation that computes a function `B` with an input vector `A`. To map it to a spatial architecture, we isolate out two functions: `A_loader` for loading the input values and `B_unloader` for saving the computed values from function `B`. This is called *compute partition*. It leads to a spatial layout shown in Fig. 1(b).

A corresponding T2S specification is shown in Fig. 1(c). Line 1-4 are the temporal definition expressing the original computation, where `A` is a one-dimensional single-precision floating-point vector, `i` is a variable, and `B` is a function that is to be run on the device. The upper loop bound of 100 is set upon execution, which is not shown.

Line 5-7 are the spatial mapping. `A_loader` and `B_unloader` are the two new functions on the device, isolated out of function `B` as a producer of the input values in `A` and a consumer of the computed values in `B`, respectively. A more detailed intermediate representation (IR) after compute partition is shown in Fig. 1(d), where `RCH` and `WCH` are primitives for reading and writing a channel. All the functions have exactly the same loop structure. However, function `A_loader` does nothing but loading values of `A` and sending them to a channel, `channel1`. Function `B` reads data from `channel1`, performs computation, and sends the results to `channel2`. Function `B_unloader` reads data from `channel2`, and stores it into memory.

[1]The syntax of T2S is similar to Halide [8].

Compute partition maintains the semantics of the original computation. Its purpose is to partition a computation into sub-computations, which in turn become accessible to optimizations and can be specialized individually (Section IV). For example, in the middle box of Fig. 1(d), the compiler finds that the loop variable `i` is no longer used, and all the inputs are from `channel1`. Therefore, the compiler automatically replaces the loop with an infinite loop, `while(true)`, which executes (or stops) if data are (not) available in `channel1`. This optimization is called *loop infinitization*.

Assuming a temporal definition, usually including simple math equations, is correctly specified by the programmer, the compiler checks the spatial directives to ensure that they do not violate the semantics of the temporal definition. This provides a correctness guarantee of the generated hardware, and composability of the directives.

Our system is built on Halide [8], a domain-specific language (DSL) for image processing on CPUs and GPUs. A key strength of Halide is to decouple a functional specification from optimizations. Many important loop-nest optimizations (e.g. loop reordering and tiling) can be easily specified in Halide. T2S extends Halide to spatial architectures with the following optimizations: (1) *Spatial layout*: compute partition. (2) *Loop transformations*: loop unrolling [2], flattening, perfectization, infinitization, and removal. (3) *Data transformations*: data forwarding, scattering, gathering, and vectorization. (4) *Data caching*: single/double buffer insertion. (5) *Control*: overlapping draining and filling of a pipeline, and drain signal generation.

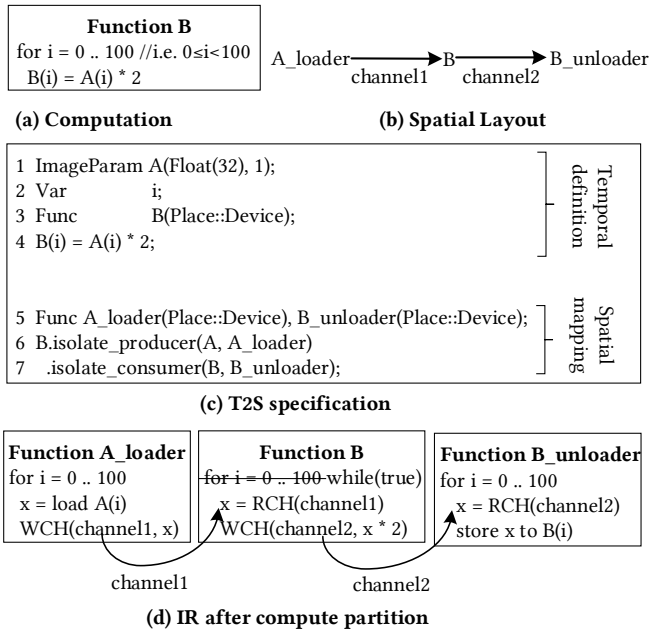


Fig. 1. An illustration of the T2S approach.

III. High-Level Illustration of The Optimizations

In this section, we introduce the spatial optimizations that can be specified with T2S, using a high-performance FPGA-targeted

[2]Different from unrolling on CPUs/GPUs, unrolling a loop of n iterations in a loop nest on a spatial architecture will hoist that loop to the outermost level, create n Processing Elements (PEs) in hardware, each for an iteration, and input/output channels will also be unrolled (Section IV). We will use the syntax `unroll(l, Hoist)` to indicate that we hoist a loop `l`, then unroll the loop and related channels into hardware.

```
1  ImageParam A(Float(32), 2), B(Float(32), 2);
2  Var        i, j;
3  RDom       k(0, K * KK * KKK);
4  Func       C;
5  C(i, j)  = 0;
6  C(i, j) += A(i, k) * B(k, j);

7  Var ii, jj, iii, jjj;
8  RDom kk, kkk;
9  C.update().tile(k, j, i, kk, jj, ii, KK * KKK, JJ * JJJ, II * III)
10            .tile(kk, jj, ii, kkk, jjj, iii, KKK, JJJ, III);
```

(a) Temporal definition

```
for i = 0 .. I * II * III
  for j = 0 .. J * JJ * JJJ
    for k = 0 .. K * KK * KKK
      C(i, j) += A(i, k) * B(k, j)
```

tiling ⟹

```
for i = 0..I
 for j = 0..J
  for k = 0..K
   for ii = 0..II
    for jj = 0..JJ
     for kk = 0..KK
      for iii = 0..III
       for jjj = 0..JJJ
        for kkk = 0..KKK
         _i = i*II*III + ii*III + iii
         _j = j*JJ*JJJ + jj*JJJ + jjj
         _k = k*KK*KKK + kk*KKK + kkk
         C(_i, _j) += A(_i, _k) * B(_k, _j)
```

(b) Function `C` before and after tiling

Fig. 2. Temporal definition of matrix multiply, including tiling.

GEMM design as the driver example. For ease of understanding, we illustrate the optimizations at a high level, and leave the compiler details to Section IV. As the optimizations can be freely composed together, one may use these optimizations to specify many different designs for the same workload. The design illustrated here is thus only one of the possibilities. In general, programmers can start from a simple design with no or few optimizations, and gradually evolve it into more complicated designs by specifying more optimizations.

A. Temporal Definition

Consider matrix multiplication `C = A * B`. A temporal definition for this is shown in Fig. 2(a). Here `A` and `B` are single-precision floating-point matrices, `i` and `j` are variables, and `k` is a reduction domain (i.e. a variable for reduction) with the range of `[0, K * KK * KKK)` (Line 1-3). In this example, we use `I, II, III, J, JJ, JJJ, K, KK`, and `KKK` to represent some compile-time constants, and assume the input matrices have a row-major storage format.

The output function `C` includes an *initial definition* (Line 5), and an *update definition* (Line 6) referred to as `C.update()`. From now on, we will focus on transforming the update definition for high performance. For convenience, by function `C`, we will refer to the update definition of function `C`, unless stated otherwise.

With the temporal definition, we get a familiar three-level loop nest as shown in the left side of Fig. 2(b). This initial implementation does not exploit data locality, which is critical for achieving high performance. To optimize for locality, Line 9-10 tile every loop twice to generate a nine-level loop nest, as shown in the right side of Fig. 2(b).

B. Evolving the Initial Specification into a High-Performance Spatial Design

Now let us see how we may transform the basic tiled loop nest (the right side of Fig. 2(b)), which has no notion of "space" at all, to a high-performance spatial design. Fig. 3 abstractly visualizes the major spatial optimizations, with the details to be described in the next section.

Compute partition (Fig. 3(a)): It isolates from function `C` the loading of matrix `A` into another function named `A_feeder`. Further, it isolates the loading of matrix `A` from `A_feeder` into another function `A_loader`. These three functions are automatically connected via channels after the isolation.

Loop unrolling (Fig. 3(b)): It unrolls loops `ii` and `jj` of function `C`. This results in an `II × JJ` array of PEs for function `C` (`II`=`JJ`=2 in Fig. 3(b)). The input channel from `A_feeder` is also replicated accordingly.

Data forwarding (Fig. 3(c)): `A(_i, _k)` can be shared when multiplied with `B(_k, _j)`, `B(_k, _j + 1)`, etc. to compute `C(_i, _j)`, `C(_i, _j + 1)`, etc. To minimize redundant memory accesses, let `A_feeder` send the data of matrix `A` to the boundary `C` PEs (`jj = 0`), which receive and forward the data to their neighbor PEs in `jj` direction. The neighbors receive and forward the data to their own neighbors in `jj` direction, and so on. This effectively allows the `jj` loop in `A_feeder` to be removed.

Buffering (Fig. 3(d)): To minimize redundant memory accesses, loop `jj` and `jjj` in `A_loader` can be removed, because they do not appear in the subscripts of the reference to matrix `A`. Such loops are called *reuse loops*. This further reduces the total accesses of matrix `A` by `JJ * JJJ` times.

However, because `A_loader` sends less data to its consumer after the loop removal, the data stream produced is different from what is expected by the consumer. To restore the correct data stream, an on-chip (double) buffer is inserted in the consumer, `A_feeder`, at an appropriate loop level so that `A_feeder` accepts the data from `A_loader`, stores the data in the buffer, and sends the buffered data to `C` in the right order. This restores the correct dataflow relationship that was broken by loop removal. Fig. 4 shows the IR after loop removal in `A_loader` and buffer insertion in `A_feeder`.

How about removing loop `j` from `A_loader`, which is also a reuse loop? That will further reduce the total accesses of matrix `A` by `J` times. This is feasible with a bigger on-chip buffer. However, in this example, we assume that loop `j` is not removed due to insufficient on-chip buffer.

Data scattering (Fig. 3(e)): A single `A_feeder` transferring data to all the boundary PEs of `C` will not scale for bigger FPGAs. Thus, unroll loop `ii` in `A_feeder` such that every `A_feeder` PE feeds a boundary `C` PE (`jj = 0`). The first `A_feeder` PE keeps the data needed by the first boundary PE of `C`, and streams the rest of the data to the next `A_feeder` PE, which works similarly. Fig. 5 shows the IR and the corresponding dataflow after loop unrolling and data scattering.

Data gathering (Fig. 3(f)): Directly writing the results to the memory from each of the `C` PEs requires a big cross-bar, which can hurt timing. To avoid this, isolate from `C` the consumer of the computed `C` values to a function, `C_drainer`. Then isolate from `C_drainer` the consumer of the computed `C` values to another function, `C_collector`. Further, isolate from

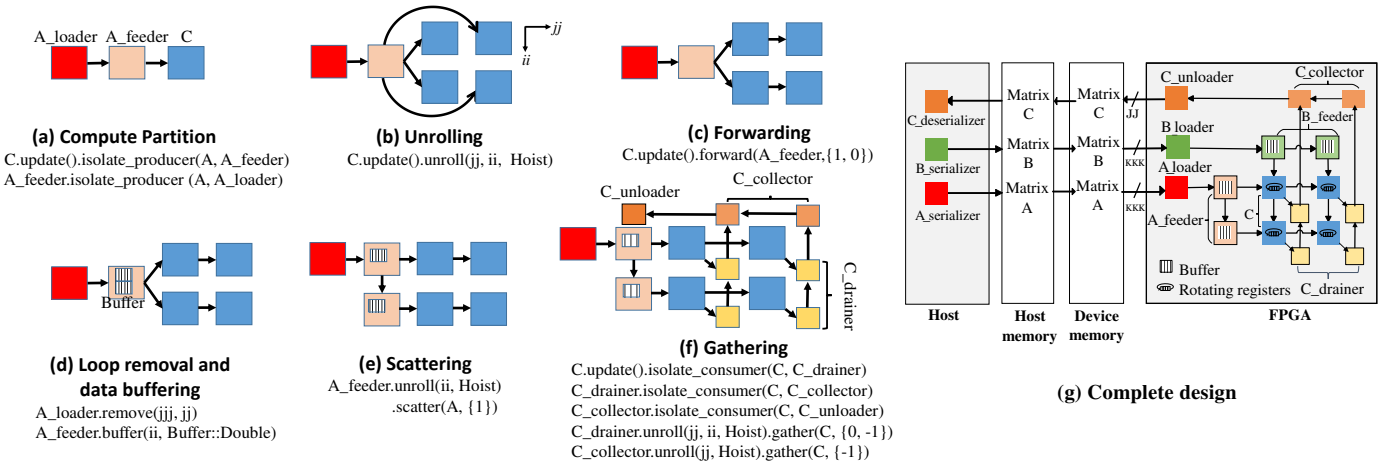


Fig. 3. Illustrating some major optimizations with GEMM as an example. For each optimization, the corresponding T2S specification is also shown.

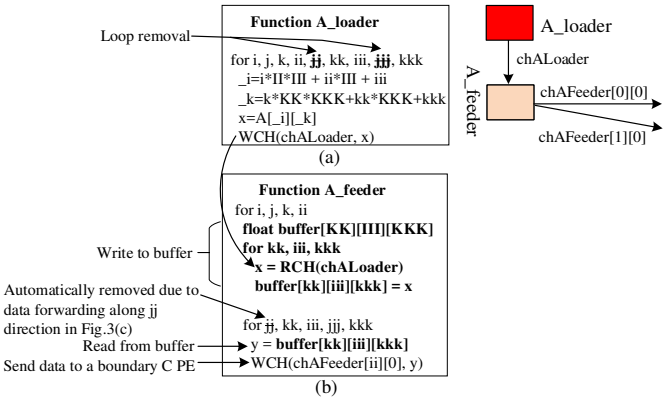


Fig. 4. IR after loop removal and data buffering in Fig. 3(d). The IR changes introduced by the optimizations are highlighted in bold fonts.

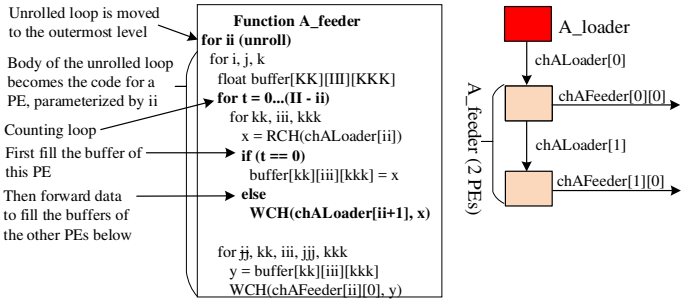


Fig. 5. IR of A_feeder after loop unrolling and data scattering in Fig. 3(e). The resulting changes to A_feeder relative to the IR in Fig. 4 are highlighted in bold fonts. The changes to A_loader (not shown) are that due to loop unrolling, channel `chALoader` is split into an array of channels `chALoader[0..II]`, and then due to data scattering, this array of channels are scattered across the A_feeder PEs as shown in the right of the figure, and A_loader writes only to `chALoader[0]`.

`C_collector` the consumer of the computed `C` values to yet another function, `C_unloader`. Unroll `C_drainer` into a 2-dimensional array of PEs, each accepting the results of a `C` PE. Each column of the drainer PEs relay the results upward. Unroll `C_collector` into a 1-dimensional array of PEs, each reading the data from a column of the drainer PEs and merging them with the data from its right, and sending the results to its left.

Data vectorization is not illustrated, but is important for performance. That is, load matrix `A` in vectors of data, vectorize related channels and operations, and save the results in vectors as well. For the GEMM example, load vectorization can be specified by `C.update().vload(A)` before the compute partition in Fig. 3(a) so that after the partition, all the functions on the data path of matrix `A`, including function `A_loader`, `A_feeder`, and `C`, are automatically vectorized. Finally, store vectorization can be specified by `C_collector.vstore(C)`.

Besides the optimizations that can be specified, there are optimizations that are automatically done by the compiler. For example, our compiler automatically flattens nested loops, because after flattening a loop nest into a single loop, the resulting loop can be pipelined efficiently.

The optimizations can also be specified to compose some other optimizations, for example, data serialization and de-serialization. In GEMM, matrix `A` is not visited sequentially (one can see this from the tiled loops in Fig. 2(b)). To maximize the usage of memory bandwidth, isolate from `A_loader` another function `A_serializer`. Unlike all the functions we have seen so far that run on the device, `A_serializer` can be declared to run on the host. `A_serializer` reads matrix `A` once, and sends the values via a *memory channel* to `A_loader` on the device. The memory channel is a virtual FIFO that our system automatically constructs with the host and device memory. This enables `A_loader` to visit the device memory completely sequentially. Similarly, `C_unloader` can send the results sequentially using a memory channel to a new function `C_deserializer` on the host, which stores the results into the host memory in the right order (i.e. data de-serialization).

So far, we mainly talked about optimizations for matrix `A`. Similar optimizations can be applied to matrix `B` to get the complete design shown in Fig. 3(g). The corresponding IR is shown in Fig. 6.

IV. Compiler Flow and Optimizations

T2S leverages the Halide compiler [8] and extends it to spatial architectures. Fig. 7 shows the overall flow of the T2S compiler. The compiler works in two modes: first in reactive, then proactive mode.

In the reactive mode, the compiler reads a programmer's specification, which includes a temporal definition and spatial mapping. The compiler then builds an IR according to the temporal

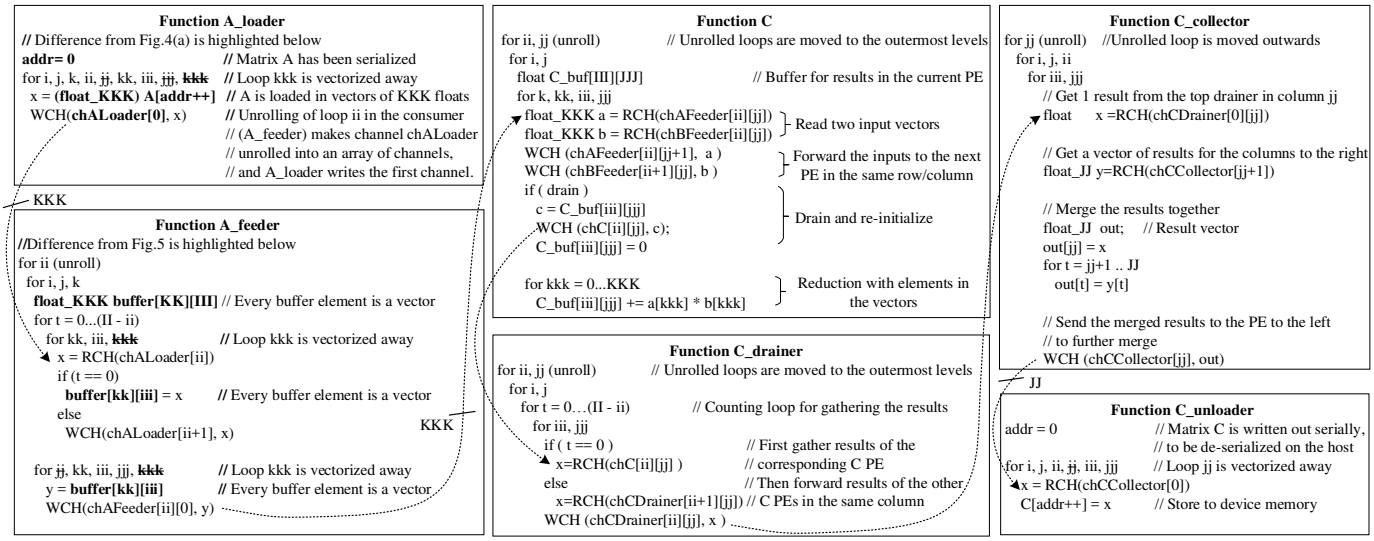


Fig. 6. IR of the complete design in Fig. 3(g). For simplicity, the data path for input `B` is skipped, loop perfectization, flattening, and infinitization and register rotation in function C are not shown. Note that function `C_drainer`, `C_collector`, and `C_unloader` have no reduction loop `k`, `kk`, or `kkk`, as they have been automatically removed due to `isolate_consumer` in Fig. 3(f).

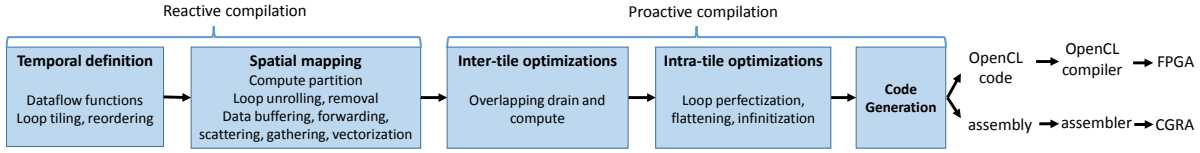


Fig. 7. T2S compiler flow.

definition, and transforms the IR according to the spatial mapping. The compiler is reactive in this phase, since its work is purely directed by the specification.

After finishing the processing of the specification, the compiler switches into the proactive mode, and performs optimizations transparent to the programmer. Finally, it generates code for the target architecture.

A. Reactive Compilation

The compiler builds up an IR according to the temporal definition. The temporal definition defines a computation with some dataflow functions, and often, also specifies how to tile or reorder the loops of the functions for better data locality or reuse.

The compiler then transforms, or equivalently annotates, the IR according to the spatial mapping. Typically, the spatial mapping specifies how to partition the computation, which loops to unroll, which loops to remove, how to manipulate data on the device including buffering, forwarding, gathering and scattering, and vectorizing of loads/stores from/to the off-chip device memory.

Compute partition – The compiler copies the IR to a new IR as a producer (or consumer), prunes the new IR so that it contains only the part that generates the input (or consumes the output) values. The compiler creates a channel, and substitutes two nodes in the IRs with the channel, such that the values flow between the two IRs through the channel. The two IRs have the same loop structure, and thus the values are communicated in the right order over the newly-created channel. If the new IR is a consumer, reduction loops are automatically removed, as the consumer is supposed to consume the final results after reduction.

Loop unrolling – The unrolled loops are hoisted to the outermost levels: all other loops become the body of the unrolled loops. Later in code generation, every iteration of the unrolled loops is turned into a hardware PE. With loop unrolling, channels in the IR have to be unrolled as well. Suppose there is a channel `ch` between a producer and a consumer. If loop *x* in the producer is unrolled `X` times, the channel is split accordingly into an array of `X` number of channels, such that instead of writing to channel `ch`, the *x*'th producer PE writes to the *x*'th channel in the new channel array, denoted by `ch[x]`. The consumer is also changed to read from `ch[x]`. We have illustrated the effect of loop unrolling in Fig. 5.

Loop removal and data buffering – Loop removal is usually accompanied by data buffering, as discussed in Section III. When directed by a specification, the compiler removes a loop in a producer IR, and inserts a buffer at a loop level in a consumer IR. The compiler calculates the buffer size with the references of the data in the body of the loop level in the consumer IR, and changes all the references in the body to refer to the buffer elements. We have illustrated the effect of loop removal and data buffering in Fig. 4.

Data buffering can also be specified without loop removal.

Data forwarding, scattering, gathering, and vectorization – Data forwarding is also called dependence localization [9]. The channels are renamed so that a producer sends data only to the boundary PEs of its consumer, and every boundary PE broadcasts the data it receives to the other PEs along a given direction. For example, in Fig. 6, `A_feeder` sends data to the first column of `C` PEs, which broadcast the data along `jj` direction.

For a PE to scatter data along a direction, the compiler

automatically inserts a counting loop `t`, counting the total data received by the PE. Depending on the value of `t`, the received data are either used by the current PE, or forwarded to the next PE along the scattering direction. Fig. 5 shows the IR after applying data scattering optimization. Gathering is the inverse operation of scattering, and works similarly.

Loads and stores can be vectorized to access the device memory in chunks of multiple contiguous data elements. This reduces the number of memory accesses and improves the achieved memory bandwidth. Currently, only the data accesses in an innermost loop are allowed for vectorization. Since the data are communicated via channels, the compiler increases the width of the channels to match the width of vectorized data, and inserts a load before/a store after the innermost loop to read/write the data as a vector. All the operations in the innermost loop are changed to operate on the vector.

B. Proactive Compilation

Usually, a problem is too big to fit on the on-chip memory, and thus has to be partitioned into tiles, and computed tile by tile, as specified by the programmer. In the proactive mode, the compiler automatically performs optimizations between tiles and within a tile, and finally generates code for the target architecture.

Overlapping drain and compute – In general, a systolic array of PEs (like the `C` PEs in GEMM) compute one tile of results, drain them, and compute the next tile of results, and so on. It is desirable to overlap the draining of one tile and the computation of the next tile. The compiler identifies all the reduction loops and inserts a local buffer right before the outermost reduction loop, as illustrated for function `C` in Fig. 6. This buffer contains the results for the current tile. The size of the buffer is calculated from the memory footprint of the results in the body of the outermost reduction loop. Then the compiler generates a drain signal and inserts code in the innermost loop to drain and re-initialize one buffer element while computing results for the next tile at the same time. If the buffer has unit-stride cyclic access pattern, the buffer can be optimized into a rotating register file. Rotating registers remove the area overhead due to address calculation. The compiler changes the memory accesses to the buffer so that a read/write access occur only at the first/last element of the buffer, and code is inserted for rotation of the buffer after the access.

Loop perfectization, flattening, and infinitization – Next, the compiler attempts to reduce the overhead of the loops by perfectizing, flattening and infinitizing them. For loop perfectization, the compiler moves an operation at an outer loop level into an inner loop level by predication. When there are more than one inner loop at the same level, the loops are merged together as a single loop whose trip count is the sum of their individual trip counts.

After the loops in a nest are perfectized, loop flattening is performed. The compiler merges all the original loops into a single loop whose trip count is the multiplication of the original loops' trip counts. Then the compiler inserts code to extract the original loop variables from the flattened loop variable. Bit masks and shift operations are used for extracting the original loop variables if all the loop bounds are powers-of-two, which are efficient for FPGAs. So far, we only support loop flattening for the loops with constant trip counts.

The compiler converts `for` loops to an infinite `while(true)` loop, when all the input/output values in a PE are read from/written to channels, and the loop variables are no longer used anywhere. The infinite loop's execution is then controlled by data availability of the input channels.

Code generation – Finally, the compiler generates code for the target architecture. Currently, the compiler generates Intel (Altera) OpenCL code for Intel FPGAs, and assembly for a research CGRA. The OpenCL code is further compiled by Intel (Altera) high-level synthesis (HLS) compiler into bitstream for FPGA. For CGRA, the assembly code is place-and-routed by an assembler and simulated by a cycle-accurate simulator. So far, the T2S compiler is fully automatic for the CGRA. For FPGAs, we are still implementing a few optimizations, including drain signal generation, data broadcasting and register rotation; thus an automatically-generated OpenCL file is modified manually to realize the missed optimizations, which are usually small changes.

V. Evaluation

We designed and wrote T2S specifications for four important tensor kernels, including GEMM, MTTKRP, TTM and TTMc. Their definitions are shown in Table I. GEMM is a core computation in many fields. MTTKRP is the computational bottleneck in Cannonical Polyadic Decomposition, and TTM and TTMc are the bottlenecks in Tucker Decomposition algorithms.

TABLE I
Definitions of the tensor kernels.

GEMM	$C(i,j) += A(i,k) * B(k,j)$
MTTKRP	$D(i,j) += A(i,k,l) * B(k,j) * C(l,j);$
TTM	$C(i,j,k) += A(i,j,l) * B(l,k)$
TTMc	$D(i,j,k) += A(i,l,m) * B(l,j) * C(m,k)$

Our designs try to match the underlying hardware architectures for the maximum efficiency, and thus for the same kernel, its designs for an FPGA and the CGRA are different. While all four tensor kernels are dominated by multiply-add operations, they vary in data reuse and compute patterns, and must be individually designed for the best performance. Therefore we have created eight different designs, which is in fact a proof of the flexibility and generality of our proposed approach.

For FPGA experiments, we use the Intel vLab Academic Cluster [10] to access one Xeon CPU and an Arria 10 GX FPGA (10AX115N2F40E2LG). For CGRA experiments, we use a research CGRA based on the triggered-instruction architecture (TIA) [11]. Our designs use the machines' native precisions, i.e. single and double-precision floats for the FPGA and CGRA, respectively.

For either architecture, a high-performance expert (the so-called "ninja") implementation of GEMM is available to us — for FPGA it is a production design implemented in OpenCL; for CGRA it is in assembly. Both implementations were manually written and highly optimized by industry experts, and have been extensively tuned for performance. These two designs took 18 and 3 months to develop, respectively. For FPGA, we also compare with an NDRange-style OpenCL implementation, a tutorial HLS design from Intel.

For the other workloads, namely MTTKRP, TTM and TTMc, we are not aware of any existing implementations on any spatial architectures. Since GEMM is a highly parallel and compute-intensive application, it provides an estimate how close one can get to the peak throughput of the target hardware. Hence, we choose to

TABLE II
COMPARISON BETWEEN AN NDRANGE OPENCL BASELINE, T2S AND NINJA IMPLEMENTATIONS FOR GEMM ON ARRIA-10 FPGA

	Baseline	T2S	Ninja
LOC	70	20	750
Systolic Array Size	—	10×8	10×8
Load Vector Length	—	16×float	16×float
Store Vector Length	—	8×float	8×float
# Logic Elements	131K (31%)	214K (50%)	230K (54%)
# DSPs	1,032 (68%)	1,282 (84%)	1,280 (84%)
# RAM Blocks	1,534 (57%)	1,384 (51%)	1,069 (39%)
Frequency (MHz)	189	215	245
Throughput (GFLOPs)	311	549	626

use the performance of the ninja GEMM as a "yardstick" to roughly estimate if the achieved throughput of a tensor kernel is high.

For FPGA experiments, we further report the absolute performance (in GFLOPS) and resource usage. For CGRA experiments, we report performance relative to the ninja implementation of GEMM. In addition, we report productivity in terms of the engineering time and lines of code (LOC).

A. Overall Performance and Productivity

For most of the workloads on either architecture, using a small fraction of lines of code, the performance achieved by T2S implementations is very close to or sometimes even higher than that of the ninja implementation of GEMM.

On the FPGA, we have achieved an absolute throughput of 549, 700, 562 and 738 GFLOPS for GEMM, MTTKRP, TTM and TTMc, respectively. On average, the T2S specifications have 29 LOC and achieve 637 GFLOPS. On the CGRA, we have achieved 92%-104% of the performance of the fine-tuned GEMM using an average 38 LOC. It took us approximately two weeks on the FPGA (due to long compile time for FPGA) and three days on the CGRA for engineering a workload, which is only about 3% of the time spent by the experts who created the ninja GEMM implementations.

B. Evaluation on the FPGA

Table II compares between the NDRange-style OpenCL baseline [7], T2S and ninja GEMM implementations. We have tuned the parameters of the baseline for the specific FPGA and show its best performance. Overall, T2S GEMM design achieves 1.76× speedup over the baseline, and 88% performance of the manually written and highly optimized ninja implementation.

As one can see from Table II, the baseline design is not able to efficiently utilize the DSP resources and achieve a lower clock frequency. The design uses NDRange kernels consisting of a 2-dimensional array of work-items (threads), each loading one data element from each of the input matrices and storing them in a buffer shared by the threads, and computing one element of a tile of the output matrix.

The T2S GEMM design for the FPGA has been shown in Fig. 3(g). Here we set its parameters such that it has ten rows and eight columns of PEs in the systolic array of function `C`, i.e., `II`=10 and `JJ`=8. We stream loads in the unit of 16 floats every memory access, i.e., `KKK`=16. Due to vectorized gathering along the systolic array columns, 8 floats are stored to the memory simultaneously. Each PE computes 1024 results with `III`=`JJJ`=32.

TABLE III
PARAMETERS FOR GEMM, MTTKRP, TTM AND TTMC DESIGNS.

	GEMM	MTTKRP	TTM	TTMc
i, j, k, l, m	32,32,32,-,-	8,4,16,16,-	8,4,4,16,-	8,4,4,4,4
ii, jj, kk, ll, mm	10,8,32,-,-	8,9,16,2,-	1,8,11,4,-	1,8,10,32,2
iii, jjj, kkk, lll, mmm	32,32,16,-,-	16,16,1,16,-	8,8,16,16,-	16,4,4,1,16
Unrolled Loops	*ii, jj*	*ii, jj*	*jj, kk*	*jj, kk*
LOC	20	28	30	37
Systolic Array Size	10×8	8×9	8×11	8×10
Load Vector Length	16×float	16×float	16×float	16×float
Store Vector Length	8×float	9×float	11×float	10×float

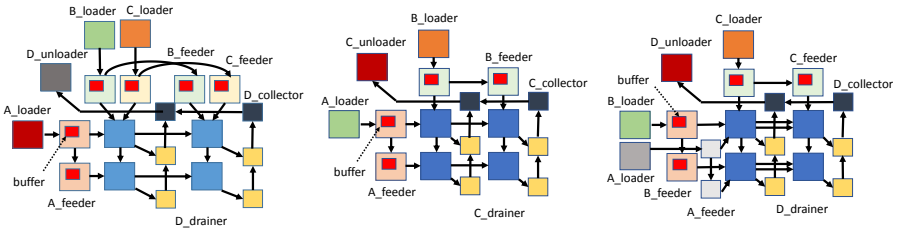
TABLE IV
EVALUATION RESULTS OF MTTKRP, TTM AND TTMC IN T2S ON THE FPGA.

	# Logic Elements	# DSPs	# RAM Blocks	Frequency (MHz)	Throughput (GFLOPs)
MTTKRP	228K (53%)	1,224 (81%)	1,526 (56%)	204	700
TTM	265K (64%)	1,416 (93%)	2,394 (88%)	201	562
TTMc	229K (54%)	1,368 (90%)	1,679 (62%)	205	738

Similarly, for MTTKRP, TTM and TTMc we tile each loop twice and unroll two of the resulting loops to get a 2-D systolic array. Table III shows the parameters for all the four designs. Data loads from the memory are vectorized in the innermost loop. For stores, vectorized gathering is applied to each design and hence the second dimension of the systolic array determines the vector length for stores. Since MTTKRP computation, $D(i,j) = \sum_{k,l} A(i,k,l) * B(k,j) * C(l,j)$, can also be expressed as $D(i,j) = \sum_k B(k,j) * (\sum_l A(i,k,l) * C(l,j))$, we manually perform this optimization in the innermost loop of the generated OpenCL code, which is a tiny code change. A similar optimization is also applied to TTMc. Such an optimization reduces the number of operations to be performed in the hardware, which results in high DSP utilization leading to throughputs even better than the ninja GEMM implementation. The designs for these workloads are shown in Fig. 8 with their resource usage and performance shown in Table II and IV.

C. Evaluation on the CGRA

Fig. 9 shows the design for GEMM on CGRA. First, the triple loop nest is tiled, similar to the FPGA design, so that a tile fits into the CGRA. Then for a tile, loader PEs read matrix `A` from external memory, and send data to a feeder. The feeder has a "virtual buffer" inside, which is functionally equivalent to a single buffer based on scratchpad memory, but is implemented using latency-insensitive channels (LICs) [11], [12]. This basic construct is important to performance: without it, GEMM loses almost 60% of performance. The same mechanism is built for matrix `B`. The reduction loop `k` is unrolled so that several PEs are accumulating partial results of the same `C(i, j)` with different parts of the matrix `A` and `B` data. The initial values of `C(i, j)` are loaded from memory, and the final values are stored back to memory. One can visually see the big differences of this design from its FPGA counterpart. Unlike the FPGA design, which is a 2-D systolic array, the CGRA design is 3-D systolic. This difference is due to the difference between the architectures — unrolling a loop results in a PE for each iteration of the unrolled loop; if the unrolled loop has inner loops not unrolled, a finite state machine (FSM) has to be constructed for each PE, which is easy to implement on an FPGA using LUTs, but is difficult to implement on a CGRA due to the limited number of control flow PEs. Hence, we fully unroll the innermost loop levels for the CGRA



(a) MTTKRP (b) TTM (c) TTMc

Fig. 8. Systolic arrays for the FPGA workloads.

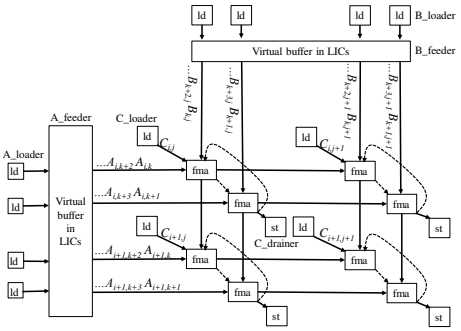


Fig. 9. Systolic array for GEMM on the CGRA

design to reduce the number of FSMs. Although this increases the number of inter-PE data communication channels, the CGRA has sufficient LICs to efficiently implement them. Following the same principle, the designs for MTTKRP, TTM and TTMc are 4-D, 4-D and 5-D systolic arrays (not shown), respectively.

Table V shows the LOCs, performance and area results for all the four kernels. All the kernels have utilized 100% fused-multiply add (FMA), except TTMc, which uses 95% FMAs, achieving a performance very close to the ninja GEMM implementation. The average LOC for these designs is 38, while the ninja GEMM implementation uses 2,280 LOC. The usage of all the other resources for these designs are well within the hardware resource constraints.

TABLE V
EVALUATION RESULTS OF T2S DESIGNS ON THE RESEARCH CGRA

	LOC	Throughput w.r.t Ninja GEMM	FMA usage
GEMM	40	92%	100%
MTKRP	32	99%	100%
TTM	47	104%	100%
TTMc	38	103%	95%

VI. Related Work

HDLs like Verilog and VHDL describe a circuit at the register-transfer level (RTL) with explicit timing [13], [14]. They can be compared to "assembly languages". HLS languages have a higher abstraction. They accept an algorithmic description of a desired behavior without clock-level timing [15]–[19]. Languages like Chisel [20], PyMTL [21], BlueSpec [22], and Hot & Spicy [23] raise level of hardware design abstraction by introducing concepts like object orientation, functional programming and guarded atomic actions in hardware design. T2S code is more succinct and at an even higher abstraction level than an HLS program. T2S code controls a compiler to generate details, instead of letting the programmer directly write the details.

DSLs also have a higher abstraction level than HLS languages [24]. Such languages express and optimize an algorithm in predefined domain-specific patterns, and lower the patterns into an HDL [8], [25]–[29]. The system we presented in this paper extends Halide [8], a DSL for image processing on CPUs/GPUs, to spatial architectures. Most of the optimizations we implemented are new to Halide as they are specific to spatial architectures. Some of them, like loop unrolling, extend the existing Halide implementation. Halide-HLS [25] is another spatial extension of Halide, where a dataflow graph of functions is specified to offload to an FPGA, with line buffers to optimize the communication. It currently focuses on image processing applications and has not implemented any of the tensor kernels we have.

Spatial [26] is a language whose abstraction level is higher than T2S. For example, for matrix multiply, Spatial specifies the multiplication, and lets the compiler determine an efficient systolic array. In comparison, T2S lets the programmer specify a systolic array in detail. This simplifies the compiler due to the diversity of the possible systolic arrays, and the difficulty for a compiler to statically determine the best choice. We have attempted to make a direct comparison with Spatial on the GEMM kernel. Unfortunately, the current Spatial provides a more robust flow for Xilinx FPGAs but fails in compilation when we target the Intel device. Nevertheless, we notice that the reported GEMM performance in the Spatial paper [26] is much lower than what we have achieved. HeteroCL [30] decouples algorithm specification from compute, data type and memory customizations. It also provides an abstraction level higher than T2S where the compiler automatically determines the systolic array for GEMM using PolySA [31] framework.

VII. Conclusion

We proposed a system that enables programmers to productively specify optimizations for constructing high-performance spatial designs. With this system, several important dense tensor kernels have achieved high performance with high productivity. In future, we plan to extend the system to more domains.

Acknowledgement

Abdullah-Al Kafi, sponsored by Charlotte Dryden, implemented the first code generator for the research CGRA, using tools engineered by Kermin Fleming, Kent Glossop and Barry Tannenbaum, et al. Sanket Tavarageri improved the code generator, supported by Bharat Kaul. Davor Capalija, Tomasz Czajkowski, and Daya Khudia provided ninja implementations of GEMM. They and Gorge Powley, Yufei Ma, Jeremy Fowers also provided the productivity data of spatial programming with various workloads. Jim Held championed the research inside the Intel Labs. We appreciate the help of John Freeman, Kunal Banerjee, Anand Venkat, Kari Pulli, Nithin Gorge, and many others at Intel. This research was funded in part under Defense Advanced Research Projects Agency (DARPA) contract FA8750-18-2-0108, DARPA MTO Software Defined Hardware program, Natural Science Foundation of China (NSFC) contract 61525202, National Science Foundation (NSF) CCF-1320545.

REFERENCES

[1] H. Rong, "Programmatic control of a compiler for generating high-performance spatial hardware," *arXiv preprint arXiv:1711.07606*, 2017.

[2] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "TensorFlow: A System for Large-Scale Machine Learning." *USENIX Symp. on Operating Systems Design and Implementation*, 2016.

[3] A. Anandkumar, R. Ge, D. Hsu, S. M. Kakade, and M. Telgarsky, "Tensor Decompositions for Learning Latent Variable Models," *The Journal of Machine Learning Research*, 2014.

[4] B. W. Bader, M. W. Berry, and M. Browne, "Discussion Tracking in Enron Email Using PARAFAC," *Survey of Text Mining II*, 2008.

[5] R. Feynman, R. Leighton, and M. Sands, "The Feynman Lectures on Physics: Volume 3, volume 3 of The Feynman Lectures on Phsyics," 1963.

[6] J. C. Kolecki, "An Introduction to Tensors for Students of Physics and Engineering," *National Aeronautics and Space Administration, Glenn Research Center*, 2002.

[7] Intel, "Intel design examples," https://www.intel.com/content/www/us/en/programmable/support/support-resources/design-examples/design-software/opencl/matrix-multiplication.html.

[8] 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 Conf. on Programming Language Design and Implementation (PLDI)*, 2013.

[9] H. Le Verge, C. Mauras, and P. Quinton, "The ALPHA Language and its use for the Design of Systolic Arrays," *Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology*, 1991.

[10] Intel, "vlab academic cluster," https://wiki.intel-research.net.

[11] M. Pellauer, A. Parashar, M. Adler, B. Ahsan, R. Allmon, N. Crago, K. Fleming, M. Gambhir, A. Jaleel, T. Krishna, D. Lustig, S. Maresh, V. Pavlov, R. Rayess, A. Zhai, and J. Emer, "Efficient Control and Communication Paradigms for Coarse-Grained Spatial Architectures," *ACM Transactions on Computer Systems (TOCS)*, 2015.

[12] K. E. Fleming, M. Adler, M. Pellauer, A. Parashar, A. Mithal, and J. Emer, "Leveraging Latency-insensitivity to Ease Multiple FPGA Design," *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2012.

[13] D. Thomas and P. Moorby, "The Verilog® Hardware Description Language," *Springer Science & Business Media*, 2008.

[14] Z. Navabi, "VHDL: Analysis and Modeling of Digital Systems," *McGraw-Hill, Inc.*, 1997.

[15] D. D. Gajski, N. D. Dutt, A. C. Wu, and S. Y. Lin, "High-Level Synthesis: Introduction to Chip and System Design," *Springer Science & Business Media*, 2012.

[16] P. Coussy and A. Morawiec, "High-Level Synthesis: from Algorithm to Digital Circuit," *Springer Science & Business Media*, 2008.

[17] 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.

[18] 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.

[19] T. Becker, O. Mencer, and G. Gaydadjiev, "Spatial Programming with OpenSPL," *FPGAs for Software Programmers*, 2016.

[20] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing Hardware in a Scala Embedded Language," *Design Automation Conf. (DAC)*, 2012.

[21] D. Lockhart, G. Zibrat, and C. Batten, "PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research," *Int'l Symp. on Microarchitecture (MICRO)*, 2014.

[22] R. S. Nikhil, "Bluespec: A General-Purpose Approach to High-Level Synthesis based on Parallel Atomic Transactions," *High-Level Synthesis*, 2008.

[23] 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.

[24] A. Van Deursen, P. Klint, and J. Visser, "Domain-Specific Languages: An Annotated Bibliography," *ACM Sigplan Notices*, 2000.

[25] 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.

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

[27] W. Luzhou, K. Sano, and S. Yamamoto, "Domain-Specific Language and Compiler for Stencil Computation on FPGA-Based Systolic Computational-memory Array," *Int'l Conf. on Reconfigurable Computing: Architectures, Tools and Applications*, 2012.

[28] 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. on Graphics*, 2014.

[29] N. George, H. Lee, D. Novo, T. Rompf, K. J. Brown, A. K. Sujeeth, M. Odersky, K. Olukotun, and P. Ienne, "Hardware System Synthesis from Domain-Specific Languages," *Int'l Conf. on Field Programmable Logic and Applications (FPL)*, 2014.

[30] 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.

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