SuSy: A Programming Model for Productive Construction of High-Performance Systolic Arrays on FPGAs

Yi-Hsiang Lai[1*], Hongbo Rong[2**], Size Zheng[3], Weihao Zhang[4], Xiuping Cui[3], Yunshan Jia[3], Jie Wang[5], Brendan Sullivan[1], Zhiru Zhang[1*], Yun Liang[3], Youhui Zhang[4], Jason Cong[5], Nithin George[2], Jose Alvarez[2], Christopher Hughes[2], Pradeep Dubey[2]

[1] School of Electrical and Computer Engineering, Cornell University, USA
[2] Intel, USA [3] Peking University, China [4] Tsinghua University, China
[5] Computer Science Department, University of California, Los Angeles, USA
*{yl2666,zhiruz}@cornell.edu**hongbo.rong@intel.com

ABSTRACT

Systolic algorithms are one of the killer applications on spatial architectures such as FPGAs and CGRAs. However, it requires a tremendous amount of human effort to design and implement a high-performance systolic array for a given algorithm using the traditional RTL-based methodology. On the other hand, existing high-level synthesis (HLS) tools either (1) force the programmers to do "micro-coding" where too many optimizations must be carried out through tedious code restructuring and insertion of vendor-specific pragmas, or (2) give them too little control to influence a push-button compilation flow to achieve high quality of results.

To tackle these challenges, we introduce SuSy, a programming framework composed of a domain-specific language (DSL) and a compilation flow that enables programmers to productively build high-performance systolic arrays on FPGAs. With SuSy, programmers express the design functionality in the form of uniform recurrence equations (UREs), which can describe algorithms from a wide spectrum of applications as long as the underlying computation has a uniform dependence structure. The URE description in SuSy is followed by a set of decoupled spatial mapping primitives that specify how to map the equations to a spatial architecture. More concretely, programmers can apply space-time transformations and several other memory and I/O optimizations to build a highly efficient systolic architecture productively. Experimental results show that SuSy can describe various algorithms with UREs and generate high-performance systolic arrays by spatial optimizations. For instance, the SGEMM benchmark written in SuSy can approach the performance of the manual design optimized by experts, while using 30× fewer lines of code.

CCS CONCEPTS

• **Hardware → Hardware-software codesign**.

KEYWORDS

DSL, FPGA, Systolic Array, Space-Time Transformation, URE

ACM Reference Format:
Yi-Hsiang Lai, Hongbo Rong, Size Zheng, Weihao Zhang, Xiuping Cui, Yunshan Jia, Jie Wan, Brendan Sullivan, Zhiru Zhang, Yun Liang, Youhui Zhang, Jason Cong, Nithin George, Jose Alvarez, Christopher Hughes, Pradeep Dubey. 2020. SuSy: A Programming Model for Productive Construction of High-Performance Systolic Arrays on FPGAs. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '20), November 2–5, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3400302.3415644

1 INTRODUCTION

Systolic algorithms have been extensively studied and employed in many important application domains such as bioinformatics, image processing, linear algebra, machine learning, and relational database [7, 9, 12, 14, 18, 20, 26, 33]. In a systolic algorithm, the dependence structure is uniform, where every data dependence has a constant distance. Mapping such dependence structures to spatial architectures lead to near-neighbor connections. The connected processing elements (PEs) jointly compose a *systolic array* that works rhythmically — at every time step, each PE reads inputs from some neighbors, performs computation, and forwards the inputs and results to other neighbors [17].

The characteristics of near-neighbor connections make systolic arrays a great match for FPGAs, where it is particularly important to minimize long interconnects to meet the target clock frequency. Indeed recent years have seen a growing number of application-specific systolic arrays implemented on modern FPGAs for efficient compute acceleration [5, 8, 12, 26, 33]. While systolic arrays typically have a very regular structure, it is far from trivial to achieve high performance unless the following optimizations are carried out: 1) finding an efficient mapping between a systolic algorithm and the physical array, 2) building an input/output (I/O) network to transfer data within the bandwidth limit, 3) constructing customized on-chip storage for data reuse, 4) vectorizing data accesses to better utilize the off-chip memory bandwidth, and 5) pipelining control signals to further increase throughput.

Obviously, any of the above optimizations would require substantial effort using the traditional RTL-based design methodology. The introduction of high-level synthesis (HLS) helps raise the level of design abstraction and hence increase productivity [4]. However, it

ICCAD '20, November 2–5, 2020, Virtual Event, USA

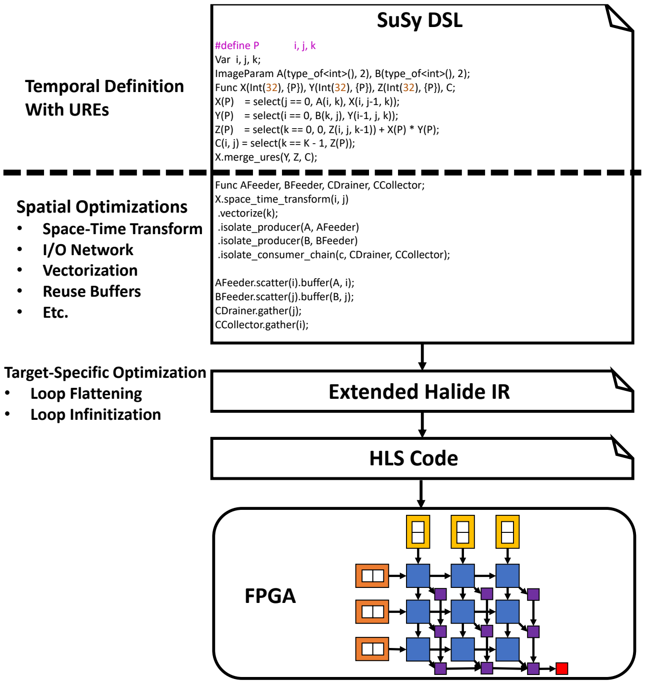
ACM ISBN 978-1-4503-8026-3/20/11…$15.00
https://doi.org/10.1145/3400302.3415644

remains challenging to strike the right balance between design quality and productivity using the existing HLS tools. To achieve high quality of results (QoRs), HLS users often have to perform "microcoding", where some of the low-level micro-architectural details must be explicitly described and mixed into the behavioral specification that is supposed to be algorithmic and target-independent. In fact, it is not uncommon for HLS experts in the industry to spend several months on building a high-performance systolic array architecture, even for a seemingly simple computation [30]. Some of the recent HLS research has proposed end-to-end compilation flow to generate application-specific systolic arrays from C/C++ programs in a push-button manner [2, 5, 11, 31]. This approach allows programmers to focus on the algorithms, while the compiler automatically explores the design space and generates systolic arrays. Unfortunately, the existing methods either lack support for key optimizations (e.g., vectorization and I/O isolation) or fail to support a general class of systolic algorithms.

There exists another line of work that further raises the abstraction level of FPGA programming by using domain-specific languages (DSLs) [1, 16, 21, 25, 27, 32]. One recent example is HeteroCL [21], a Python-based embedded DSL that provides a back end for mapping designs to systolic arrays. It is worth noting that systolic array support in HeteroCL also employs a push-button compilation flow and shares the same problems as other systolic compilers mentioned earlier. Another example is T2S-Tensor [32], which is a DSL built on Halide [29] that generates high-performance systolic arrays. With the T2S-Tensor DSL, programmers can productively explore different optimizations with decoupled temporal definition and spatial mappings. However, this DSL is restricted to dense tensor computations.

Along this line, we propose SuSy, a programming framework built upon Halide [29] for productively building high-performance systolic arrays on FPGAs. SuSy decouples the algorithm specification from spatial optimizations, where the former can concisely express any systolic algorithm while the latter can describe essential optimizations for systolic arrays. Figure 1 provides a high-level overview of the proposed framework. The input program is specified in the SuSy DSL, which is composed of (1) an algorithm (or temporal definition) expressed in uniform recurrence equations (UREs) and (2) decoupled spatial optimization. The SuSy compiler lowers the input to an intermediate representation (IR) extended from Halide, where we perform user-specified optimizations and several target-specific transformations. The compiler then produces the HLS code (in OpenCL) as output, which is eventually compiled to bitstream for FPGA execution. Our main technical contributions are summarized as follows:

- This work is the first to demonstrate that high-performance customized systolic arrays can be built with many optimizations succinctly expressed in a DSL that is not tied to a specific application domain. The proposed SuSy DSL provides a clean programming model that decouples temporal algorithmic definitions from spatial mappings. Notably, the URE-based temporal specification can model a rich set of systolic algorithms used in many different applications. Examples include the Smith-Waterman algorithm in bioinformatics, convolution in deep learning, matrix multiplication in linear algebra, and sorting.



```
#define P        i, j, k
Var  i, j, k;
ImageParam A(type_of<int>(), 2), B(type_of<int>(), 2);
Func X(Int(32), {P}), Y(Int(32), {P}), Z(Int(32), {P}), C;
X(P)    = select(j == 0, A(i, k), X(i, j-1, k));
Y(P)    = select(i == 0, B(k, j), Y(i-1, j, k));
Z(P)    = select(k == 0, 0, Z(i, j, k-1)) + X(P) * Y(P);
C(i, j) = select(k == K - 1, Z(P));
X.merge_ures(Y, Z, C);
```

```
Func AFeeder, BFeeder, CDrainer, CCollector;
X.space_time_transform(i, j)
 .vectorize(k);
 .isolate_producer(A, AFeeder)
 .isolate_producer(B, BFeeder)
 .isolate_consumer_chain(c, CDrainer, CCollector);

AFeeder.scatter(i).buffer(A, i);
BFeeder.scatter(j).buffer(B, j);
CDrainer.gather(j);
CCollector.gather(i);
```

Figure 1: Overview of the SuSy programming framework.

- We introduce in SuSy an explicit and concise representation of space-time transformation, which allows the programmers to explore the trade-offs between performance and area with various temporal scheduling on different shapes of systolic arrays. In addition, SuSy further supports several essential spatial optimizations for building highly efficient systolic arrays, including vectorization, customized reuse buffer, data gathering/scattering for the I/O network.
- We have developed a comprehensive compilation flow targeting Intel FPGAs for SuSy. Experimental results show that SuSy can close the expert-designer performance gap on widely used compute kernels such as SGEMM, convolution, and Smith-Waterman. For dense tensor computations, we achieve more than 96% DSP efficiency. While for Smith-Waterman, we achieve 6.3× higher performance over a state-of-the-art framework.

The remainder of this paper is organized as follows: Section 2 provides the background knowledge for SuSy through examples; Sections 3 and 4 explain the programming model and the compilation flow in detail respectively; we report the evaluation results in Section 5 and compare with previous work in Section 6; Section 7 concludes this work and outlines future research directions.

2 BACKGROUND

This section introduces the concepts of UREs and space-time transformations, and provides two illustrating examples.

2.1 Uniform Recurrence Equations (UREs)

Given an n-dimensional iteration space D, a system of UREs consists of a set of recurrence equations expressed in the following form [15]:

$$V_i(z) = f(V_1(z - d_1), V_2(z - d_2), ..., V_p(z - d_p)), \text{ for } z \in D$$

where $V_1, V_2, ..., V_p$ are variables, f is an arbitrary function, z is an n-dimensional vector representing a computation point (i.e., an iteration) in D, and d_i is an n-dimensional **constant** vector representing the distance from z. Basically, the UREs collectively represent an n-dimensional perfectly nested loop with constant dependence distances.

UREs have been extensively used in many programming frameworks for generating systolic arrays. The main reasons are twofold: 1) they are general and expressive enough to describe probably most systolic algorithms [23, 28, 35], and 2) they can specify both computation and data flow, which exposes more optimization opportunities to the compiler and programmers [22].

```
1  // select(cond, then_case, else_case) = then_case if cond is true
2  // otherwise it returns else_case (if provided)
3  for (k = 0; k < K; k++)
4   for (j = 0; j < J; j++)
5    for (i = 0; i < I; i++)
6     // URE for computing the multiplication and accumulation
7     Z(i, j, k) = select(k == 0, 0, Z(i, j, k-1)) + A(i, k)*B(k, j);
8     // Assign results to the output C
9     C(i, j)    = select(k == K - 1, Z(i, j, k));
```

(a) GEMM

```
1  // MAX = maximum possible value
2  for (i = 0; i < N; i++)
3   for (j = 0; j < N; j++)
4    X(i, j) = select(j == 0,
5                  max(A(i), select(i == 0, MAX, Y(i-1, j))),
6                  select(i >= j, max(X(i, j-1), Y(i-1, j))));
7    Y(i, j) = select(i == j,
8                  select(j == 0, A(i), X(i, j-1)),
9                  select(i > j, min(select(j == 0, A(i), X(i, j-1)),
10                                   select(i == 0, MAX, Y(i-1, j)))));
11   B(j) = select(i == N-1, Y(i, j));
```

(b) Insertion sort

Figure 2: Examples of using UREs.

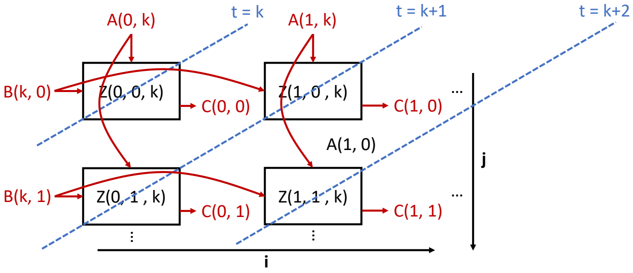
Here we show two examples of UREs in Figure 2 along with the loop nests representing the iteration space. In Figure 2a, a general matrix multiplication (GEMM) kernel is described with UREs. In this example, we calculate C = A×B, where A is an I×K matrix, B is a K×J matrix, and C is an I×J matrix. We use a single URE (L7) to describe the multiplication and accumulation, where we have one variable Z in a 3-dimensional domain `(i, j, k)` for storing the partial sum. After the calculation completes, we assign the results to output C in L9. Note that if the `select` expression does not have a false case, nothing is performed should the condition fail. Another example is shown in Figure 2b, where we perform insertion sort on an input vector A and store the final output in B. Here we have two UREs (L4-L10) with variable Y storing the sorted results after step j and X being an auxiliary variable. From these two examples, we can see that as long as an algorithm has constant dependence distances, we can describe it using UREs.

2.2 Space-Time Transformation

UREs alone only describe the function of the systolic algorithm without providing any spatial information. To build a systolic array from UREs, we need to determine the mapping between the domain of the UREs and the physical array dimensions. Space-time transformation [19, 22] is in essence a loop transformation that specifies the mapping. To be more specific, the transformation maps an n-deep loop nest to a time loop and $n-1$ space loops. The space loops are mapped to different PEs, and the time loop is used to schedule the original iterations to run on the PEs. The transformation can be described by a transformation matrix T:

```
// t = k + j + i
time for (t = 0; t < I+J+K-2; t++)
 space for (j = 0; j < J; j++)
  space for (i = 0; i < I; i++)
   // recover the original loop variable before transformation
   k = t - j - i;
   // only compute if it is in the original domain
   if (0 <= k < K)
    // we use the last dimension to represent the time distance
    Z(i, j, 0) = select(k == 0, 0, Z(i, j, 1)) + A(i, k) * B(k, j);
    C(i, j)    = select(k == K - 1, Z(i, j, 0));
```

(a) Loop structure after space-time transformation.



(b) Mapped systolic arrays.

Figure 3: Example of applying space-time transformation to GEMM UREs.

$$T = \begin{pmatrix}\Pi \\ \tau\end{pmatrix},$$

where τ is a *scheduling vector* that generates the time loop and Π is an $(n-1) \times n$ *projection matrix* that generates space loops. A transformation matrix is valid only if it preserves the data dependence, and if no two iterations are scheduled to run on the same PE at the same time. In this work, we always set the projection matrix Π to be an identity matrix. This is a common practice when experts manually build systolic arrays [12, 26]. The support for non-identity projection matrices is left as future work.

Figure 3 shows an example of applying space-time transformation to the UREs in Figure 2a, where

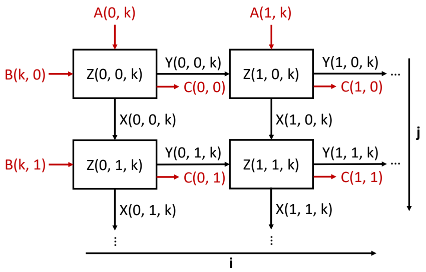
$$T = \begin{pmatrix}100\\010\\111\end{pmatrix}, \tau = \begin{pmatrix}111\end{pmatrix}, \Pi = \begin{pmatrix}100\\010\end{pmatrix}.$$

If we take a look at the loop structure after the transformation (Figure 3a), loops i and j become space loops and loop k is replaced with a time loop t. In other words, after transformation, we have a total of $I \times J$ PEs. Then, we need to check if the data dependence is still preserved by calculating new distances, which can be done by multiplying T with the distance vector. For example, the new distance of `Z(i, j, k-1)` can be calculated by $\begin{pmatrix}100\\010\\111\end{pmatrix}\begin{pmatrix}0\\0\\1\end{pmatrix} = \begin{pmatrix}0\\0\\1\end{pmatrix}$. This is a positive dependence vector, and thus the original data dependence is preserved [22]. [1]

[1] Traditionally, all data dependence should be strongly satisfied for time loops (i.e., the dependence distance should be greater than zero). However, such a transformation usually introduces loop skewing that leads to complicated hardware. In SuSy, we allow the dependence to be weakly satisfied (i.e., the dependence distance could be zero) and let the hardware compiler take over the scheduling of PEs.

```
1  for (k = 0; k < K; k++)
2   for (j = 0; j < J; j++)
3    for (i = 0; i < I; i++)
4     // UREs for reusing the inputs
5     X(i, j, k) = select(j == 0, A(i, k), X(i, j-1, k));
6     Y(i, j, k) = select(i == 0, B(k, j), Y(i-1, j, k));
7     // URE for computing the product and accumulation
8     Z(i, j, k) = select(k == 0, 0,
9                  Z(i, j, k-1)) + X(i, j, k)*Y(i, j, k);
10    // Assign results to the output C
11    C(i, j)    = select(k == K - 1, Z(i, j, k));
```

(a) New UREs with input data reuse.



(b) Mapped systolic arrays.

Figure 4: Example of modifying UREs with data reuse.

We also need to make sure the computation after transformation lies in the original domain by recovering the original loop indices and adding a check (L5-8). We can now easily map the computation into a 2-D systolic array, as shown in Figure 3b, where the red lines represent the communication with I/O and the dotted blue lines represent the time schedule.

2.3 Design Space Exploration

In this section, we demonstrate how we can explore different design choices by combining UREs and space-time transformation. As can be seen in Figure 3b, inputs A and B are broadcast to all PEs, which is not scalable. To solve this, we can reuse the inputs by sending them through neighbor PEs. We can describe such data flow between PEs by modifying the UREs (Figure 4).

Figure 4a shows the new set of UREs with data reuse by introducing two new equations in L5-6. Specifically, variables X and Y store the values of inputs A and B, respectively. After applying the same space-time transformation, the mapped systolic array is shown in Figure 4b, where the black lines represent the communication between PEs. This simple example demonstrates how UREs provide programmers more flexibility when exploring the design space. Similarly, by choosing different transformation matrices, programmers can explore the trade-offs between area and performance.

3 THE PROGRAMMING MODEL

The SuSy programming model is built upon Halide [29], and the main reasons are as follows: 1) The Halide DSL cleanly decouples the algorithm specification and temporal schedule. In SuSy, we inherit the same concept by decoupling the temporal definition from spatial optimizations, allowing programmers to efficiently explore different design choices without modifying the algorithm definition. 2) Halide abstracts algorithms composed of multi-level loop nests with declarative programming, which is a good fit for UREs because of the underlying multi-dimensional iteration space. The bounds can either be inferred from the input shapes or explicitly specified by the users. 3) Halide provides a concise yet expressive IR, which can be easily extended for describing optimizations required to generate high-performance systolic arrays.

In this section, we explain the SuSy programming model in detail. We first explain how we use UREs to describe temporal definitions in Section 3.1. Then we demonstrate how we apply a set of spatial optimizations in Section 3.2. For better illustration, we continue to use the GEMM example.

3.1 Temporal Definition

In SuSy, we extend Halide to express UREs, since the original Halide syntax does not support recurrent functions.

```
1  // Define the inputs with integer type and two dimensions
2  ImageParam A(type_of<int>(), 2);
3  ImageParam B(type_of<int>(), 2);
4  // Extend Halide's syntax for describing data type and placement
5  // We use C macros to simplify the code
6  #define ftype Int(32), {i, j, k}, Place::Device
7  Var i, j, k;
8  Func X(ftype), Y(ftype), Z(ftype), C;
9  X(i, j, k) = select(j == 0, A(i, k), X(i, j-1, k));
10 Y(i, j, k) = select(i == 0, B(k, j), Y(i-1, j, k));
11 Z(i, j, k) = select(k == 0, 0,
12                     Z(i, j, k-1)) + X(i, j, k)*Y(i, j, k);
13 C(i, j)    = select(k == K - 1, Z(i, j, k));
```

Figure 5: Describing UREs for GEMM in SuSy.

We show an example in Figure 5, where we describe the UREs for GEMM. We first declare the input matrices A and B with ImageParam, where we specify the data type and the number of dimensions (L2-3). Currently, SuSy supports the same set of data types as Halide (i.e., 64/32-bit float and 64/32/16/8-bit integer types). Then we define the iteration space and variables with Var and Func, respectively (L6-8). Unlike Halide, programmers can specify the data placement with either Place::Device (i.e., FPGA) or Place::Host (i.e., CPU). Since we offload the entire application to the FPGA, we choose Place::Device for all variables. We write down the UREs in L9-13 by referencing Figure 4a. With the declarative programming style, programmers do not need to explicitly write down the loop nests.

3.2 Spatial Optimization

After describing the temporal definition with UREs, we need to specify how we map them to systolic arrays as well as other spatial optimizations. With the decoupled programming style, users can efficiently apply different spatial mappings by using the SuSy *primitives* (or *scheduling functions* in terms of Halide). In this section, we describe the syntax and semantics of selected primitives in detail. Table 1 shows the set of primitives we currently support.

Space-Time Transformation – As we have described in Section 2, to map UREs to a physical systolic array, we need to perform space-time transformation. An example is shown in Figure 6.

```
1  X.merge_ures(Y, Z, C);
2  X.space_time_transform({i, j},     // space loops
3                         {0, 0, 1}); // scheduling vector
```

Figure 6: Primitive for specifying the transformation.

To specify the transformation in SuSy, we first need to define the target set of UREs, which can be achieved by using the primitive merge_ures (L1). Then we establish the transformation by employing the primitive space_time_transform (L2-3), where the

Table 1: Primitives for spatial optimizations in SuSy.

Primitive	Description
F.merge_ures(U_1, U_2, ..., U_n)	Define the set of UREs F, U_1, U_2, ..., U_n to optimize.
F.space_time_transform(space, tau)	Specify the space-time transformation that will be applied to F, where space is the set of space loops, and tau is the scheduling vector.
F.vectorize(var)	Vectorize the specified loop variable var of F.
F.reorder(var_1, var_2, ..., var_n)	Reorder the loop nest for F according to the specified order, starting from the innermost level.
F.tile(var, var_o, var_i, factor)	Tile the loop variable var of F into two new variables var_o and var_i with a factor of factor.
F.isolate_producer({E_1, E_2, ...}, P)	Isolate a list of expressions {E_1, E_2, ...} (usually inputs) in F to a separate producer kernel P.
F.isolate_consumer(E, C)	Isolate an expression E (usually an output) in F to a separate consumer kernel C.
F.remove(var)	Remove loop var of F.
F.buffer(E, v, mode)	Insert a reuse buffer at loop v for expression E with mode (either Buffer::Single or Buffer::Double).
F.scatter(E, var)	Reduce data communication overhead (i.e., data broadcast) by scattering the expression E to the consumer along loop var.
F.gather(E, var)	Reduce data communication overhead (i.e., data broadcast) by gathering the expression E from the producer along loop var.

first argument specifies the space loops, and the second argument defines the scheduling vector. To better illustrate the optimizations without losing the generality, here we use a simpler time schedule than the one in Figure 3b. We omit the space matrix here since it is an identity matrix as mentioned in Section 2.2. There are several constraints to the arguments. First, only the inner-most loops can be space loops. Otherwise, programmers need to perform loop reordering with reorder before applying space-time transformation. Second, the transformation matrix must be valid in terms of preserving the dependence.

Tiling and Vectorization – In most cases, the problem size may be too large to fit the given hardware resources. To solve that, we can tile the design and compute only the partial results of each tile on-chip. In addition, with tiling, programmers can explore another dimension of parallelism by applying vectorization, where we compute a fixed-length of data at a single time. With vectorization, we can perform vector loads/stores from/to the off-chip memory to better utilize the off-chip memory bandwidth. An example is shown in Figure 7a.

```
1 // Tile loop k with factor KI
2 X.tile(k, ko, ki, KI)
3 // Vectorize the inner loop
4 X.vectorize(ki);
```

(a) Vectorization

```
1 // Define the loaders and unloaders
2 Func A_Loader, B_Loader, C_Unloader;
3 // Isolate the inputs to loaders
4 X.isolate_producer(A, A_Loader)
5  .isolate_producer(B, B_Loader)
6 // Isolate the output to unloaders
7  .isolate_consumer(C, C_Unloader);
```

(b) Isolation

Figure 7: Applying vectorization and isolation in SuSy.

In this example, we first tile the k loop into ko and ki with a factor of KI via the primitive tile (L2). Then, we vectorize the ki loop in L4. After vectorization, we are computing a total of I×J×KI computations in parallel.

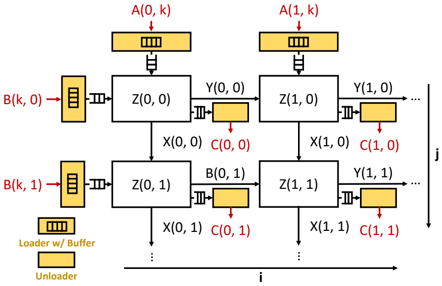
Input/Output Isolation – To further improve the performance, we can overlap the execution of the off-chip memory accesses with on-chip computations so that the communication latency does not throttle the overall throughput of the systolic array. We name such an optimization as *isolation*, which is conceptualized in Figure 7b.

In the GEMM example, we have three off-chip memory accesses, which are *loading* input values from A and B and *unloading* output values to C. To isolate the access, we introduce new computation stages – two *loaders* for reading the input values and one *unloader* for writing the output values (L2). To describe the behavior, we use the primitives isolate_producer to isolate inputs (L4-5) and isolate_consumer to isolate outputs (L7). After isolation, the main computation kernel reads/writes data from/to loaders/unloaders instead of the off-chip memory.

Reuse Buffer Insertion – In many cases, we are loading repeated data from inputs due to the underlying iteration space. For instance, in GEMM, input A only depends on loop i and k. However, under the three-dimensional iteration space, we need to load the same data for J times. To reduce the memory accesses, we can load the data once from the off-chip memory and store it into an on-chip reuse buffer. In other words, all succeeding data accesses will load from the reuse buffer instead of the host memory. Figure 8a L2-3 show how we remove the loop with repeated access via remove and insert a reuse buffer via buffer. Users can further specify the loop level for inserting the buffer, which allows users to explore the trade-off between area (buffer size) and throughput.

```
1 // Remove the reuse loop and insert double buffers
2 A_Loader.remove(j).buffer(A, j, Buffer::Double);
3 B_Loader.remove(i).buffer(B, i, Buffer::Double);
```

(a) Primitives for inserting reuse buffer.



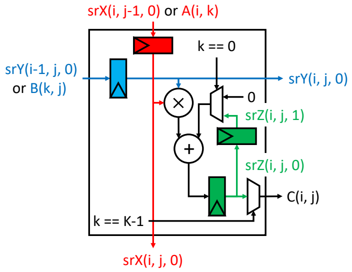
(b) Optimized systolic arrays.

Figure 8: Inserting reuse buffer to SuSy.

Finally, Figure 8b shows the systolic array after applying all spatial optimizations mentioned above. After isolation and buffer insertion, the main computation kernel reads input data from the double buffers inside the loaders. Meanwhile, loaders read input data from the off-chip memory.

```
1  int srX[I][J][1], srY[I][J][1], srZ[I][J][2];
2  for (t = 0; t < K; t++)
3   // shift register logics
4   unrolled for (j = 0; j < J; j++)
5    unrolled for (i = 0; i < I; i++)
6     unrolled for (s = 0; s < 1; s++)
7      srZ(i, j, 1-s) = srZ(i, j, 0-s);
8      // no need to shift srA and srB
9   // computations
10  unrolled for (j = 0; j < J; j++)
11   unrolled for (i = 0; i < I; i++)
12    k = t;
13    srX(i, j, 0) = select(j==0, A(i, k), srX(i, j-1, 0));
14    srY(i, j, 0) = select(i==0, B(k, j), srY(i-1, j, 0));
15    srZ(i, j, 0) = select(k==0, 0, srZ(i, j, 1)) +
16                         srX(i, j, 0) * srY(i, j, 0);
17    C(i, j)      = select(k==K-1, srZ(i, j, 0));
```

(a) Equivalent HLS code.



(b) Hardware architecture of PE(i, j).

Figure 9: Equivalent HLS code and corresponding PE architecture after performing space-time transformation in Figure 6 — In the hardware architecture, we can see that there are three shift registers, which are srX, srY, and srZ respectively. For srX and srY, they take values from either inputs or neighbor PEs and send the values to the neighbor PEs. On the other hand, srZ is updated with its previous value within the same PE and sends out the results only when the accumulation is complete.

Other Optimizations – SuSy provides several additional spatial optimizations, including gathering/scattering and data serialization/de-serialization. With gathering and scattering, we reduce the number of connections between the systolic array and off-chip memory, which makes our design more scalable. Meanwhile, data serialization improves the utilization of the off-chip memory bandwidth by serializing data on the host before sending them to the systolic array. Similarly, we can perform de-serialization after we collect the results from the systolic array.

4 COMPILATION

In this section, we first explain the PE architecture generated by SuSy. We then describe a few representative back-end specific optimizations that are automatically applied. We also briefly discuss how we generate HLS code and deploy it to FPGAs.

PE Architecture – There are two ways for each PE to communicate with each other. First, they can communicate *asynchronously* through channels. However, channels may introduce unnecessary control overhead in hardware (e.g., handshaking). Therefore, SuSy generates *synchronous* architecture using shift registers. Specifically, each PE is associated with several shift registers that store the values of each variable (Figure 9b). For instance, variable X is associated with a shift register srX. The equivalent loop structure with shift registers is shown in Figure 9a, where we have three shift register for the variables X, Y, and Z (L1). The shift register size equals to the maximal time distance plus one. For example, the maximal time distance for variable Z is one (L15) as described in Section 2.2. Thus, the size of shift register srZ[i][j] for PE (i,j) is two (L1). The registers are shifted at the beginning of each time step (L3-8), right before we perform the computations (L10-16). In addition, after space-time transformation, we mark the space loops as unrolled while the time loops are pipelined automatically by the HLS compiler.

Target-Specific Optimizations – The SuSy compiler also applies a set of optimizations automatically to further improve the performance. These are designed for the back end we currently target, namely, the Intel HLS tool. The specific optimizations include 1) loop flattening, which flattens a loop nest by combining neighbor loops into a single loop to reduce the control overhead, and 2) loop inifinitization, which replaces a flattened loop with a `while(1)` loop to further reduce pipeline stalls.

Code Generation – We extend the Halide OpenCL code generation to generate Intel HLS code. Since data serialization, de-serialization, and some low-level optimizations are still under development at this stage, we manually implemented them by slightly changing the generated HLS code. Then we push the code through the Intel HLS compiler and downstream CAD tool flow to produce the final bitstream that runs on the hardware.

5 EVALUATION

In this section, we evaluate the systolic arrays generated by SuSy. All experiments are conducted on Intel vLab Academic Cluster [13], equipped with Intel Xeon Platinum 8280 CPU (2.70 GHz) and Intel Arria 10 GX FPGA. We first demonstrate the flexibility and productivity of SuSy by showing results on four benchmarks from different application domains, including single-precision general matrix multiplication (SGEMM), tensor-tensor multiplication (TTM), convolution (Conv), and Smith-Waterman (SW). We further provide in-depth analysis on SGEMM, Conv, and SW, where we perform quantitative comparison against existing frameworks such as Spatial [16], HeteroCL [21], T2S-Tensor [32], and PolySA [5].

Table 2: Specifications of two FPGAs used in evaluation.

	Intel Arria 10 GX	Xilinx VU9P
Targeted By	[32][33] [SuSy]	[5][16][21]
Technology Node	Intel 20nm	TSMC 14nm/16nm
Soft Logic	427K ALMs	1,182K LUTs
DSPs	1,518 FP DSP	6,840 DSP48E2
BRAMs	2,713	2,160
Max Device Frequency	500 MHz	800 MHz

Table 2 lists the key characteristics of Intel Arria 10 GX and Xilinx UltraScale+ VU9P; these two FPGA devices are used by the related work that we are comparing against in the remaining section. Note that each single-precision floating-point (FP) multiplication and accumulation (MAC) operation maps to one hardened FP DSP on Intel Arria 10, whereas the same MAC operation consumes five 27x18 DSP48E2 units on Xilinx Ultrascale+ VU9P. There are 6840 DSP48E2 units in total on VU9P, which roughly translates to

6840/5=1368 Intel FP DSPs (vs. 1518 on Arria 10). Hence we argue that these two FPGAs have a similar computation power in terms of the peak throughput on MAC, although the Xilinx VU9P is listed with a higher maximum device frequency.

Table 3: Evaluation results for benchmarks in SuSy.

Benchmark	Problem Size	LOC	#ALMs	#DSPs	#BRAMs	Freq. (MHz)
SGEMM [5][16][21][32]	(1024, 1024, 1408)	25	40%	93%	32%	202
TTM [32]	(256, 64, 256, 352)	25	33%	93%	31%	209
Conv [5][33]	3rd Layer of VGG-16 (64, 128, 112, 112, 3, 3)	28	35%	84%	30%	220
SW [16][21]	(1M, 128)	44	33%	0%	20%	225

General Evaluation – First, we evaluate the flexibility and productivity of SuSy using four benchmarks, including SGEMM and TTM in linear/tensor algebra, convolution in deep learning, and SW from bioinformatics. Table 3 shows that we can describe a rich set of systolic algorithms in SuSy, each with just tens of lines of code. If we compare with related work in terms of the expressiveness, only Spatial [16] and HeteroCL [21] can describe benchmarks that are not dense tensor computations. However, these two frameworks cannot achieve the same level of performance as SuSy. Another existing framework PolySA [5], which is based on a polyhedral compiler, can only handle algorithms without dynamic control flows such as SGEMM and Conv. The work proposed by Wei et al. [33] can generate highly efficient systolic arrays, but only for convolutional neural networks.

In the following, we provide more detailed case studies on SGEMM, Conv, and SW to compare SuSy and other frameworks.

Table 4: Performance impact of different spatial optimizations on a reduced SGEMM – We select a smaller input size ($512 \times 512 \times 512$) and also a smaller systolic array (8×8 with a vector length of 8 if applicable).

	+ Space-Time Transform	+ Vectorize	+ Isolate	+ Buffer & Others
#LUTs/ALMs	28%	21%	33%	24%
#DSPs	4.2%	34%	34%	34%
#BRAMs	16%	20%	16%	19%
Frequency (MHz)	250	203	225	259
Throughput (GFLOPs)	2.29	18.8	52.8	255
DSP Efficiency	7.2%	9.0%	23%	96%

Case Study: SGEMM – We first demonstrate how each spatial optimization affects the performance by using a smaller problem size ($512 \times 512 \times 512$) since, for large inputs, some of the design variants can be time-consuming for bitstream generation or do not even fit the device. In Table 4, we show not only the performance numbers, but also the resource usage, frequency, and DSP efficiency. To calculate the DSP efficiency, we divide the throughput by theoretical throughput, which is defined as $\#DSP \times 2 \times Frequency/K$, where K is a target-dependent constant. We set $K = 5$ for VU9P and $K = 1$ for Arria 10.

From the table, we observe a trend of increasing throughput and DSP efficiency after each optimization step. We select the design with space-time transformation as our baseline, where we unroll the space loops and map them to PEs. After applying vectorization, a degradation of frequency occurs because the number of PEs increases. However, the increased computation power covers frequency degradation, and the throughput is consequently better. After I/O isolation, both frequency and DSP efficiency are improved, and the bottleneck now becomes the off-chip memory bandwidth. Introducing reuse buffers and other optimizations such as data scattering and gathering solve the issue (i.e., the DSP efficiency is now close to 100%). With all optimizations combined, the throughput is over 100× better than that of the baseline.

Table 5: Performance comparison for SGEMM.

	Spatial [16]	HeteroCL [21]	PolySA [5]	T2S-Tensor [32]	Ninja [32]	SuSy
LOCs	44	16	7	20	750	25
Systolic Aarray	No	Yes	Yes	Yes	Yes	Yes
Target FPGA	VU9P	VU9P	VU9P	Arria10	Arria10	Arria10
#LUTs/ALMs	36%	52%	49%	50%	54%	40%
#DSPs	12%	58%	89%	84%	84%	93%
#BRAMs	23%	45%	89%	51%	39%	32%
Frequency (MHz)	200	198	229	215	245	202
Throughput (GFLOPs)	2.4	246	555	549	626	547
DSP Efficiency	3.5%	79%	98%	99%	99%	96%

To further analyze the quality of results, we compare with other programming frameworks, including Spatial, HeteroCL, PolySA, and T2S-Tensor. We also compare with the Ninja implementation [32], which is written in HLS OpenCL by experts. We show the results in Table 5.

To begin with, there exists a stark difference in performance between the designs implemented without and with systolic arrays, namely, Spatial versus other frameworks. Naturally, there also exists a gap between general-purpose frameworks (i.e., HeteroCL) and those designed for generating systolic arrays (i.e., PolySA, T2S-Tensor, and SuSy). Finally, SuSy achieves similar throughput and DSP efficiency compared with other systolic array compilers specialized for certain application domains. Notably, SuSy achieves 87% of the throughput of the hand-written Ninja implementation, while only using 30× fewer lines of code (LOC). Moreover, if we compare on the same FPGA device (i.e., Arria 10), SuSy requires much less resource usage in ALMs and BRAMs mainly because we generate synchronous architectures with shift registers while T2S-Tensor and the Ninja manual design adopt asynchronous architectures with channels. As for PolySA, although it uses fewer LOCs and achieves similar performance, it is not as general as SuSy, as mentioned earlier.

Case Study: Convolutional Layer – We further compare the quality of results among frameworks that generate high-performance systolic arrays (Table 6). The design generated by Wei et al. [33]

Table 6: Performance comparison for convolutional layer – The array shape is interpreted as width × height × vector length.

	PolySA [5]	Wei et al. [33]	SuSy
Target FPGA	VU9P	Arria 10	Arria 10
Systolic Array Shape	8 × 19 × 8	8 × 19 × 8	8 × 10 × 16
#LUTs/ALMs	- (49%)	- (57%)	150K (35%)
#DSPs	- (89%)	- (81%)	1,280 (84%)
#BRAMs	- (71%)	- (45%)	827 (30%)
Frequency (MHz)	229	253	220
Throughput (GFLOPs)	548	600	551
DSP Efficiency	98%	97%	98%

can achieve higher throughput at a higher frequency since their framework is designed explicitly for convolutional neural networks by mapping to manually optimized systolic array templates. However, this means they are not as general as SuSy. Moreover, under the same problem size and similar systolic array size, there also exists a reduction in resource usage similar to SGEMM. For PolySA, we can reach the same conclusion as the previous case study.

Table 7: Performance comparison for Smith-Waterman.

	Spatial [16]	HeteroCL [21]	SuSy
Target FPGA	VU9P	VU9P	Arria 10
#LUTs/ALMs	330K (28%)	111K (9.4%)	139K (33%)
#DSPs	0 (0%)	0 (%)	0 (0%)
#BRAMs	1,409 (65%)	470 (22%)	539 (20%)
Frequency (MHz)	200	152	250
Throughput (GCUPs)	0.11	1.25	7.89

Case Study: Smith-Waterman Algorithm – In this final case study, we compare the results with the two general-purpose frameworks (i.e., Spatial and HeteroCL). For Smith-Waterman, the typical performance metric is cell updates per second (CUPs), which can be derived by dividing the number of cells (i.e., the product of the lengths of the two input sequences) by the run time. Table 7 shows that SuSy achieves more than 5× performance improvement compared with HeteroCL and more than 70× improvement compared with Spatial. In addition, we are running at a much higher frequency because, with SuSy, we can explicitly skew the iteration space by using space-time transformation to better pipeline the design.

6 RELATED WORK

There exists a large body of literature on systolic array synthesis that enables programmers to generate systolic arrays at a high abstraction level [2, 5, 10, 11, 21, 24, 31–33].

Systolic array compilers with a push-button compilation flow – Compilers such as [2, 5, 11, 31] provide an end-to-end flow to generate systolic arrays without much user intervention. These compilers select the space-time transformation and other necessary optimizations based on built-in heuristics or automatic design space exploration. While delivering high productivity, these compilers usually fail to achieve high performance due to two major reasons: incomplete optimization directives and design space. Many optimizations are missing in the previous work. For example, vectorization and I/O isolation are missing in [2, 31]. Loop infinitization is missing in PolySA [5]. The missing of such optimizations could lead to sub-optimal designs. Apart from the compilers that target general systolic algorithms, there are also efforts attempting to generate domain-specific systolic arrays [8, 33]. For instance, Gemmini [8] and the framework proposed by Wei et al. [33] propose to generate efficient systolic arrays for deep neural networks (DNNs). Although both of them adopt configurable templates that generate high-performance systolic arrays, they are limited to DNNs.

Systolic array compilers with user-guided optimizations – In comparison, works such as MMAlpha [10] and T2S-Tensor [32] take in user-specified optimizations. MMAlpha [10] is built upon UREs and lets programmers specify the space-time transformation. It supports both manual and automatic scheduling selection similar to the works in the previous category. However, it lacks the support for optimizations such as vectorization and reuse buffer insertion. A more recent work T2S-Tensor [32] incorporates richer optimizations compared with MMAlpha. It is the first work that inherits the principle to decouple the computation from the scheduling in designing systolic arrays. Nonetheless, T2S-Tensor can only generate systolic arrays for dense tensor kernels. In addition to those kernels, SuSy can generate systolic arrays for a much wider range of applications with UREs. Moreover, users can explore a larger design space with space-time transformation. Finally, by generating synchronous hardware, we can largely reduce the resource usage.

General HLS compilers – Beyond generating systolic arrays, there is also a plethora of work targeting implementing general applications on FPGAs [3, 6, 16, 21, 34]. However, experimental results show that there still exists a performance gap between such frameworks and dedicated systolic array compilers like SuSy.

7 CONCLUSIONS AND FUTURE WORK

We have presented SuSy, a programming model for productively building high-performance systolic arrays. With SuSy, programmers can describe any systolic algorithm with UREs and also efficiently explore different spatial optimizations, such as space-time transformation and reuse buffer insertion. Moreover, we provide an end-to-end compilation flow targeting Intel FPGAs. Experiment results show that we can indeed achieve high performance on not only dense tensor kernels but also bioinformatics benchmarks. We believe SuSy can bridge the gap between productivity and quality of the development of systolic arrays on FPGAs.

We plan to release the proposed programming model in an open-source format. Moreover, we will introduce more features to SuSy, such as autotuning, reusing systolic arrays, and customized throughput analysis. We also plan to extend SuSy to support more complex benchmarks such as an entire deep learning model.

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 Awards #1723715 and #1723773, and NSF Award #1909661. We thank Geoff Lowney, John C Kreatsoulas, and Bernhard Friebe for sponsoring the project within Intel. We also appreciate the useful help from Justin Gottschlich, John Freeman, Mohamed Issa, and Nitish Srivastava.

REFERENCES

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

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

[3] 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.

[4] 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.

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

[6] 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.

[7] J. de Fine Licht, G. Kwasniewski, and T. Hoefler. Flexible Communication Avoiding Matrix Multiplication on FPGA with High-Level Synthesis. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2020.

[8] H. Genc, A. Haj-Ali, V. Iyer, A. Amid, H. Mao, J. Wright, C. Schmidt, J. Zhao, A. Ou, M. Banister, et al. Gemmini: An Agile Systolic Array Generator Enabling Systematic Evaluations of Deep-Learning Architectures. *arXiv preprint arXiv:1911.09925*, 2019.

[9] W. M. Gentleman and H. Kung. Matrix Triangularization by Systolic Arrays. *Real-Time Signal Processing IV*, 1982.

[10] A.-C. Guillou, F. Quilleré, P. Quinton, S. Rajopadhye, and T. Risset. Hardware Design Methodology with the Alpha Language. *FDL'01*, 2001.

[11] F. Hannig, H. Ruckdeschel, H. Dutta, and J. Teich. PARO: Synthesis of Hardware Accelerators for Multi-dimensional Dataflow-intensive Applications. *International Workshop on Applied Reconfigurable Computing*, 2008.

[12] Intel. Accelerating Genomics Research with OpenCL, and FPGAs. 2017.

[13] Intel. vLab Academic Cluster. *URL: https://wiki.intel-research.net*, 2019.

[14] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al. In-Datacenter Performance Analysis of a Tensor Processing Unit. *Int'l Symp. on Computer Architecture (ISCA)*, 2017.

[15] R. M. Karp, R. E. Miller, and S. Winograd. The Organization of Computations for Uniform Recurrence Equations. *Journal of the ACM (JACM)*, 1967.

[16] 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.

[17] H. Kung and C. Leiserson. Systolic Arrays for (VLSI). 1978.

[18] H. Kung, B. McDanel, and S. Q. Zhang. Packing Sparse Convolutional Neural Networks for Efficient Systolic Array Implementations: Column Combining under Joint Optimization. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[19] S. Kung. VLSI Array Processors. *IEEE ASSP Magazine*, 1985.

[20] J. Kurzak, P. Luszczek, M. Gates, I. Yamazaki, and J. Dongarra. Virtual Systolic Array for QR Decomposition. *Int'l Parallel and Distributed Processing Symp. (IPDPS)*, 2013.

[21] 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.

[22] D. Lavenier, P. Quinton, and S. Rajopadhye. Advanced Systolic Design. *Digital Signal Processing for Multimedia Systems*, 1999.

[23] 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.

[24] A. W. Lim and M. S. Lam. Maximizing Parallelism and Minimizing Synchronization with Affine Transforms. *ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, 1997.

[25] 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.

[26] D. Moss, S. Krishnan, E. Nurvitadhi, and et al. A Customizable Matrix Multiplication Framework for the Intel HARPv2 Platform - A Deep Learning Case Study. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2018.

[27] 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.

[28] P. Quinton. Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations. *ACM SIGARCH Computer Architecture News*, 1984.

[29] 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.

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

[31] R. Schreiber, S. Aditya, S. Mahlke, V. Kathail, B. R. Rau, D. Cronquist, and M. Sivaraman. PICO-NPA: High-level Synthesis of Nonprogrammable Hardware Accelerators. *Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology*, 31(2):127–142, 2002.

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

[33] 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.

[34] Xilinx. SDAccel: Enabling Hardware-Accelerated Software. 2020.

[35] J. Xue. Formal Synthesis of Control Signals for Systolic Arrays. 1992.