AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA

Jie Wang  
University of California, Los Angeles  
jiewang@cs.ucla.edu

Licheng Guo  
University of California, Los Angeles  
lcguo@cs.ucla.edu

Jason Cong  
University of California, Los Angeles  
cong@cs.ucla.edu

ABSTRACT
While systolic array architectures have the potential to deliver tremendous performance, it is notoriously challenging to customize an efficient systolic array processor for a target application. Designing systolic arrays requires knowledge for both high-level characteristics of the application and low-level hardware details, thus making it a demanding and inefficient process. To relieve users from the manual iterative trial-and-error process, we present AutoSA, an end-to-end compilation framework for generating systolic arrays on FPGA. AutoSA is based on the polyhedral framework, and further incorporates a set of optimizations on different dimensions to boost performance. An efficient and comprehensive design space exploration is performed to search for high-performance designs. We have demonstrated AutoSA on a wide range of applications, on which AutoSA achieves high performance within a short amount of time. As an example, for matrix multiplication, AutoSA achieves 934 GFLOPs, 3.41 TOPs, and 6.95 TOPs in floating point, 16-bit and 8-bit integer data types on Xilinx Alveo U250.

KEYWORDS
polyhedral model; systolic array; compilation; FPGA

ACM Reference Format:

1 INTRODUCTION
The systolic array architecture is capable of delivering high performance for a wide range of applications, such as linear algebra [37], machine learning [52], and genomics [20]. In recent years, we have also seen a wide adoption of systolic array architectures in the field of deep learning [1, 8, 9, 15, 17, 26, 54]. In recent years, we have also seen a wide adoption of systolic array architectures in the field of deep learning [1, 8, 9, 15, 17, 26, 54].

However, designing high-performance systolic arrays is never an easy task. It requires the expert knowledge for both the target application and the hardware. Specifically, designers need to identify the systolic array execution pattern from the application, transform the algorithm to describe a systolic array, write the hardware code for the target platform, and tune the design to achieve the optimal performance. Each step will take significant efforts, raising the bar to reap the benefits of such an architecture.

To lower the programming efforts of systolic arrays, there is an active research domain to automate the systolic array generation [3, 7, 10, 15, 17, 30, 46, 48, 52]. Previous works [3, 4, 7, 17, 46] have proposed various compilation flows that use the polyhedral model [4, 49] to generate systolic array designs. These works perform the dependence analysis on the program and transform the program using the space-time transformation [27, 31] to generate systolic arrays. Although the polyhedral model based compilers can analyze the program and perform transformations automatically, most of them suffer from low performance that does not match the hand-written designs. The key problem is that many important hardware optimization techniques are missing in these tools. For example, the framework MMA Alpha [17] does not support array partitioning, which is essential in handling large-scale programs given the limited hardware. The recent work PolySA [10] is the first work that covers the most optimization techniques and generates designs with comparable performance to the manual designs. However, PolySA suffers from low generality as the framework only supports programs with a single statement in perfectly nested loops.

Apart from the polyhedral compilers, there have been several recent works [30, 48] that develop domain-specific language (DSL) compilers for systolic arrays based on the Halide [43] infrastructure and achieve comparable performance to the manual designs. However, these tools require programmers to analyze the program and write the DSL to set the legal program transformations manually prior to the compilation. In addition, there is no auto-tuning support in these works and programmers need to examine different transformations manually to find the best design. This task can be as challenging as finding out the legal transformations given the vast design space. All of these hurdles have raised barriers for programmers to access such tools and stretched out the development cycles, decreasing the productivity.

In summary, we found that previous works are faced with the following limitations that prohibit them from being used in practice:

- Limited generality. Works such as [7, 10] place rigid restrictions on the input programs that limit the application scope of the compiler.
- Limited performance. Works such as [3, 7, 17, 46] support limited program optimizations that limit the performance of the generated designs.
- Limited productivity. Works such as [30, 48, 55] require programmers to analyze the program and to describe and explore the program transformations manually. This, in turn, leads to long development cycles.
In this paper, we propose a new compilation framework, AutoSA, to overcome the previous limitations. AutoSA is a polyhedral model based compilation framework. The top priority for us is to improve the generality by compiling any programs to systolic arrays, as long as they can be supported by the polyhedral framework and can legally be mapped to a systolic array. AutoSA supports SCoP programs\(^1\) with imperfectly nested loops and multiple statements. To compile such programs to systolic arrays, we propose techniques and optimizations for automatic translation of regular sequential programs to parallel ones that describe a complete system of systolic arrays, including both the processing elements (PEs) and the on-chip I/O network. We would like to emphasize that the compilation of even this restricted class of programs is very challenging and that no automatic and general solution exists despite decades of research. To support these transformations, previous works either rely on a semi-automatic workflow that requires users to determine the transformation manually prior to the compilation (e.g., MMAAlpha [17], SuSy [30]), or restrain themselves to a narrow set of programs and techniques (e.g., [7], PolySA [10]). In this paper, we demonstrate that AutoSA is not only able to handle applications with regular dependence structure such as matrix multiplication and convolution, but also supports applications with complicated and irregular dependence structure such as LU decomposition.

In addition, AutoSA further improves performance and productivity. AutoSA covers a superset of all the previous optimization techniques that have been applied on systolic arrays and extend with new techniques to further improve the performance. Several representative techniques that AutoSA supports can be found in Table 1. For example, SIMD vectorization is an important technique to increase parallelism and resource efficiency. AutoSA supports auto-detection of vectorizable loops with rigid dependence and access analysis and performs automatic program transformation and code generation to produce a vectorized design. In comparison, such a feature is either not supported in the previous work, or requires human intervention to analyze and transform the program. This could lead to the missing of such an optimization opportunity with sub-optimal performance.

AutoSA takes C code as the input that requires the minimal lines of code to describe an algorithm. The entire compilation flow is automated with minimal human intervention. Besides, an auto-tuning module is implemented to ease the efforts of design space exploration. In the evaluation section, we show that programmers are able to generate high-performance systolic arrays with AutoSA within a short amount of time.

The contributions of this paper are as follows:

- **Automation**: We introduce a new open-source compilation framework\(^2\), AutoSA, that generates systolic arrays on FPGAs automatically.
- **Algorithms**: We propose a set of efficient and effective algorithms based on the polyhedral framework to construct and optimize systolic arrays.
- **Experiments**: We evaluate AutoSA on a suite of benchmarks. We show that AutoSA is able to generate high-performance systolic arrays within a short amount of time.

As mentioned in its paper, it takes up to 23 minutes to perform the transformation, these compilers are semi-automatic and require programmers to transform the program into a form with perfected nested loops. For instance, SuSy [30] requires programmers to transform the program into a form with perfected nested loops to satisfy the language requirement. Data reuse should be explicitly described in SuSy. As we will show later, such a job is

### Table 1: Comparison between different frameworks.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Generality</td>
<td>Auto</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Imperfectly Nested Loops</td>
<td>Auto</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Multi-Statement</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Performance</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Array Partitioning</td>
<td>Auto</td>
<td>No</td>
<td>Auto</td>
<td>Semi-Auto</td>
</tr>
<tr>
<td>Latency Hiding</td>
<td>Auto</td>
<td>No</td>
<td>No</td>
<td>Semi-Auto</td>
</tr>
<tr>
<td>SIMD Vectorization</td>
<td>Auto</td>
<td>No</td>
<td>No</td>
<td>Semi-Auto</td>
</tr>
<tr>
<td>Double Buffering</td>
<td>Auto</td>
<td>No</td>
<td>No</td>
<td>Semi-Auto</td>
</tr>
<tr>
<td>Data Packing</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Semi-Auto</td>
</tr>
<tr>
<td>Productivity</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Input</td>
<td>C</td>
<td>DSL</td>
<td>C</td>
<td>DSL</td>
</tr>
<tr>
<td>Auto-Tuning</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Space-Time Transformation</td>
<td>Auto</td>
<td>Semi-Auto</td>
<td>Auto</td>
<td>Semi-Auto</td>
</tr>
</tbody>
</table>

---

\(^1\)SCoP, as an abbreviation for Static Control Part, is a category of programs that can be supported by the polyhedral model. Details will be introduced in Section 3.

\(^2\)https://github.com/UCLA-VAST/AutoSA

### 2 RELATED WORK

**Polyhedral compilers**: The polyhedral model is a compilation framework for loop transformation [4, 32, 33, 41, 49]. Most prior systolic array compilers are built upon the polyhedral framework [3, 7, 10, 17, 46]. The commonality of these frameworks is the use of space-time transformation [27, 31] to convert an algorithm into a new program that describes the architecture and execution of the systolic array. These compilers are all fully automatic and improve the productivity, but most of them fail to deliver the performance on par with manual designs because they miss several hardware optimization techniques that help increase the compute and communication efficiency. The first four columns of Table 1 compare AutoSA with three other representative polyhedral frameworks. Among the previous works, PolySA [10] covers the largest set of optimization techniques. However, PolySA is limited in its generality as it only supports the program with a single statement in perfectly nested loops. Besides, PolySA is limited in its implementation which is built upon Matlab and is unscalable in handling complex designs. As mentioned in its paper, it takes up to 23 minutes to perform the polyhedral transformation on a single layer of CNN, which finishes within seconds in AutoSA.

**DSLs**: There are several recent works that implement a domain-specific language (DSL) based compiler for generating systolic arrays [30, 48, 55]. T2S-Tensor [48] is a DSL compiler extended from Halide [43] for generating systolic arrays for tensor programs. Following the similar design principle of Halide, T2S-Tensor decouples the compute and communication optimization of systolic arrays. The recent work SuSy [30] has further improved T2S-Tensor by supporting general applications that can be mapped to systolic arrays. Both T2S-Tensor and SuSy are able to achieve high performance. However, since Halide does not have dependence analysis and relies on conservative rules to determine the legality of program transformation, these compilers are semi-automatic and require programmers to analyze the systolic array execution pattern from the algorithms and specify the necessary transformation required to generate the desired array. For instance, SuSy [30] requires programmers to transform the program into a form with perfected nested loops to satisfy the language requirement. Data reuse should be explicitly described in SuSy. As we will show later, such a job is
challenging when handling complicated applications with irregular dependence structure. Furthermore, neither tools implement the auto-tuning. All of these limitations have reduced the productivity of using such tools.

**Other frameworks:** There are some other frameworks that target a specific domain of applications [13–16, 22, 37, 52]. Gemini [15] is a framework for generating systolic arrays for matrix multiplication. It uses a code template that can be reconfigured to generate different arrays. Wei et al. [52] implement a template-based generator for convolution kernels in deep neural networks. These frameworks achieve high performance with many application-specific optimization techniques. However, the mapping mythology is only limited to specific applications and cannot extend to general ones. Lastly, apart from systolic arrays, there is also an active research domain on mapping applications to CGRs [25, 28, 34, 40, 42, 53]. CGRAs differ from systolic arrays with more flexible PE architecture and I/O interconnects. The techniques proposed in this work will be helpful in addressing some similar issues such as computation scheduling and I/O mapping.

### 3 BACKGROUND

In this section, we describe the polyhedral model, which is the foundation of the algorithms that we proposed in AutoSA. We also introduce the space-time transformation, which is the basis of the automatic systolic array compilation.

#### 3.1 Polyhedral Model

The polyhedral model is a mathematical framework for loop nest optimization. Loop nests that satisfy the requirements of the polyhedral model are called Static Control Part (SCoP) [4, 6]. A SCoP is defined as a set of statements with loop bounds and as affine functions of the enclosing loop iterators and variables that are constant during the SCoP execution.

A program in the polyhedral model is typically represented by three components: *iteration domains, access relations, and a schedule*. We use a running example of matrix multiplication (MM) to illustrate these concepts. Figure 1 shows the example code.

The iteration domain contains the loop instances of the statements in the program. The iteration domain of the statement S0 in the example program has the form \( \{ S0[i,j,k]_0 \leq i < M \land 0 \leq j < N \land 0 \leq k < K \} \). Throughout the paper, to represent the communication between PEs \( \{ S0[i,j,k] \} \rightarrow \{ [i,j,k] \} \). The access relation maps a statement instance to an array index. For example, the access relations for the read accesses in the statement S0 have the form \( \{ S0[i,j,k] \rightarrow A[i,k] \land S0[i,j,k] \rightarrow B[i,j] \} \).

Finally, a schedule maps instance sets to multi-dimensional time. The statement instances are executed following the lexicographic order of the multi-dimensional time. As an example, the schedule of the statement S0 has the form \( \{ [i,j,k]_0 \leq i < M \land 0 \leq j < N \land 0 \leq k < K \} \). The schedule tree of a SCoP program can be represented by schedule trees [51]. Figure 2 shows the schedule tree of the example program. The schedule tree starts with a domain node that defines the iteration domain of the program, followed with band nodes that encode the partial

\[ \text{for (int } i = 0; i < M; ++i) \]
\[ \text{for (int } j = 0; j < N; ++j) \]
\[ \text{for (int } k = 0; k < K; ++k) \]
\[ \text{S0: } C[i,j] += A[i,k] \times B[k,j]; \]

**Figure 1:** Example code of matrix multiplication.

**Figure 2:** Initial schedule of MM in schedule tree form.

\[ \text{for (int } i = 0; i < M; ++i) \]
\[ \text{for (int } j = 0; j < N; ++j) \]
\[ \text{for (int } k = 0; k < K; ++k) \]
\[ \text{S0: } C[i,j] += A[i,k] \times B[k,j]; \]

**Figure 3:** Example of space-time transformation.

**Figure 4:** Example of space-time transformation.

schedules at each loop dimension. The isl library manipulates the schedule tree of the program to perform the loop transformation. To generate the final code, an AST is obtained from the schedule tree which is then lowered to the target code (e.g., C).

#### 3.2 Space-Time Transformation

Space-time transformation [27, 31, 36, 45] is the foundation of the automatic systolic array synthesis. It applies loop transformations on the target program and assigns new semantics (space and time) to the generated loops. Space loops map loop instances to different PEs that execute concurrently, while time loops describe the computation inside each PE.

To generate a legal systolic array, the following constraints should be satisfied by the loop transformation: First, the transformation should be semantics-preserving. Second, all dependences should be uniform (with constant dependence distance). Third, the dependence distances on space loops should be no greater than one so that the data communication only happens between neighbor PEs. Note that for the first and second constraints, we consider all types of dependences (flow, anti, output and input/read dependences). We take into account the read dependences since the data transfer needs to be managed explicitly in systolic arrays including the read-only data. As for the third constraint, we only examine the flow and read dependences which are associated with the inter-PE communication. Since each PE has its own address space, anti and output dependences do not contribute to the data communication between PEs [5].

For the MM example in Figure 1, we obtain one flow dependence (domain constraints omitted for brevity) as \( D1 := \{ S0[i,j,k] \rightarrow S0[i,j,k+1] \} \), and two read dependences for array references \( A[i][k] \) and \( B[k,j] \) as \( D2 := \{ S0[i,j,k] \rightarrow A[i,j,k+1] \} \) and \( D3 := \{ S0[i,j,k] \rightarrow B[i+1,j,k] \} \), respectively. One possible space-time transformation is \( S := \{ S0[i,j,k] \rightarrow [i,j,k] \} \), which is an identity mapping that keeps the original loop. We could calculate the dependence distances for the above-mentioned three dependences \( D1, D2, \) and \( D3 \) under the schedule \( S \), which are \( (0, 0, 1), (0, 1, 0), \) and \( (1, 0, 0) \). All dependences are uniform (we omit the
Therefore, horizontal and vertical interconnects between PEs are necessary. There will be multiple systolic arrays generated from this pool. At present, AutoSA generates 1D and 2D systolic arrays. This constraint can be relaxed to generate higher-dimensional arrays if necessary. There will be multiple systolic arrays generated from this step, each with a unique schedule. Users can choose which array to process manually, or leave it to be explored by the auto-tuner.

5.2 Array Partitioning

Given the limited on-chip resource, array partitioning is mandatory when mapping a large array to FPGA. To achieve this, AutoSA tiles these two components into two stages: computation and communication management. The stage of computation management constructs the PE and optimizes its micro-architecture. After that, the stage of communication management builds the I/O network for transferring data between PEs and the external memory. Details of these two stages will be covered in Sections 5 and 6, respectively.

**Code generation:** After the previous stages, AutoSA generates the AST from the optimized program. The AST is then traversed to generate the final design for the target hardware.

**Auto-tuner:** The stages of computation and communication management involve multiple optimization techniques, each introducing several tuning options. AutoSA implements tunable knobs for these techniques which can be set by users manually or tuned by an auto-tuner. Details of the auto-tuner are covered in Section 7.
the outermost permissible loop band in the schedule tree which contains the space loops. The tiling factors can be chosen by the users or set by the auto-tuner during the design space exploration. Figure 5a shows one example in which we tile the outermost loop band in the MM example (shown in Figure 2) with the tiling factors of $(4, 4)$. The point loops from the original loops $i$ and $j$ are kept as the space loops. This will lead to a 2D systolic array with the dimensions of $4 \times 4$.

### 5.3 Latency Hiding

Latency hiding helps hide the pipeline stalls caused by the loop-carried dependence of the compute statements. In the MM example, the multiply-and-add (MAC) operation in the statement $S0$ introduces loop-carried dependence on the loop $k$, resulting in an initial interval (II) greater than one. To resolve this issue, AutoSA looks for parallel loops in the schedule tree, strip-mines them and permutes the point loops innermost. As an example, loops $i$ and $j$ are parallel loops in the MM example. We will strip-mine them with the tiling factors of $(2, 2)$ and permute the point loops innermost. Since there is no loop-carried dependence on the innermost loop, the PE could now achieve $\text{II}=1$. The newly generated schedule is shown in Figure 5b. Similar as the previous stage, AutoSA allows users to specify the loops to be tiled and the tiling factors. Alternatively, such choices will be explored by the auto-tuner to maximize the performance.

### 5.4 SIMD Vectorization

SIMD vectorization duplicates the compute units inside each PE, which still share the same control logic. This helps amortize the control overheads and improve the resource efficiency of the design. At present, AutoSA detects the vectorizable loop by examining the following two criteria: 1) The loop should be a parallel loop or a reduction loop\(^1\). 2) All array references within the loop are stride-one or stride-zero in regard to this loop. In the MM example, the loop $k$ is a reduction loop. Array references $C[i, j]$ and $A[i] [k]$ are stride-zero and stride-one with regard to loop $k$. The array reference $B[k][j]$ requires a layout transformation to $B[j][k]$ so that it becomes a stride-one access that enables the vectorization. Figure 5c shows the vectorized code in which we strip-mine the loop $k$ with a factor of 2. The point loop is permuted innermost and marked *unroll* which will be handled by HLS tools at last. During the compilation, AutoSA examines each loop and enumerates all the possible layout transformations to expose the SIMD opportunities. Users may choose one loop to proceed or let the auto-tuner take over and make the choice.

### 6 COMMUNICATION MANAGEMENT

So far we have finished the PE construction and optimization. However, the current array is still not functional as we are missing the other key component, the I/O network. The I/O network is a network on chip that supports two types of data communication:

- **Inner-array communication**: This refers to the data communication between PEs. An an example, in Figure 3b, we show that

\[\text{DOMAIN} : \{ S0[i,j,k] : 0 \leq i < M \land 0 \leq j < N \land 0 \leq k < K \}\]
\[\text{BAND} : \{ S0[i,j,k] \rightarrow [i/4, j/4, k/4] \}\]  
- **Outer-array communication**: This refers to the data communication between PEs and the external memory (e.g., DRAM). In the MM example, the I/O network needs to fetch the data elements of matrices $A$ and $B$ to feed the array, drain the final results of matrix $C$ from the array and write out to the external memory.

The stage of communication management in AutoSA analyzes the program and constructs the I/O network as mentioned above. We show that I/O network can be built automatically via data dependence analysis in the polyhedral model. Furthermore, as the topology of the I/O network plays an important role in the frequency of the design, we extend the algorithm to build an I/O network that only involves local interconnects, hence, guaranteeing the sustained high frequency.

The following subsections explain our approaches in detail. Section 6.1 describes how we analyze the dependences in the program to extract the necessary information for constructing the I/O network. Section 6.2 builds the I/O network using the information extracted from the previous step. Section 6.3 discusses several I/O optimization techniques to further improve the I/O performance.

#### 6.1 I/O Analysis

The data communication is associated with the data dependences. Previous works such as [5, 12] have demonstrated how to implement the data transfer scheme for MPI programs using the polyhedral model. Our algorithms for building the I/O network of systolic arrays are inspired by that thread of work but with further extension to take into account the uniqueness of systolic arrays. To build the I/O network, AutoSA analyzes the following three types of data dependences:

- **Read dependence**: For transferring the read-only data.
- **Flow dependence**: For transferring the intermediate results.
- **Output dependence**: For transferring the final results.

Table 2 lists the dependences extracted from the MM example. The step of I/O analysis interprets such dependences and extracts a

---

\(^1\)The current polyhedral framework that AutoSA builds on lacks the capability to detect the reduction loop, which requires the user annotation prior to the compilation.
we compute the copy-out set by subtracting the source domains $g$ we construct an I/O group. The detailed procedure of this step.

An I/O group $g$ is defined as a tuple of $(A, D)$ where $A$ is a set of array accesses that are associated with the current group and $D$ is the set of data dependences associated with the array accesses in $A$. In Algorithm 2, we first populate the initial I/O group set $G$. A single I/O group is constructed for each array access and its associated data dependence. For each I/O group, the following two properties are computed:

- **I/O direction**: This is the component of the dependence distance vector on the space loops.
- **I/O type**: The I/O group is classified as exterior I/O if the dependence is carried by the space loops. Otherwise, it is classified as interior I/O.

As an example, in the MM example, for the array access $B[k][j]$, we construct an I/O group $g$ from the array access $B[k][j]$ and its associated dependence $D2$ as shown in Table 2. The dependence distance of $D2$ on the space loops is $(1, 0)$. Therefore, we assign the I/O direction as $g.dir = (1, 0)$ and the I/O type as $g.type = exterior$.

The next step of our algorithm merges the I/O groups that share the same properties. I/O groups are merged together if satisfying the following constraints: 1) They have the same I/O direction and I/O type. 2) They are associated with the same current group and the same type of dependence. Later, AutoSA will allocate a set of I/O modules for each I/O group, as will be discussed in detail in Section 6.2.

The last step is to compute the statement instances that require such data. We divide them into two sets: copy-in set $W_{in}$ and copy-out set $W_{out}$. These sets contain the statement instances that require the data to be copied in or copied out, respectively. For I/O groups with read dependences, we compute the copy-in set as the union of all source and destination domains of the dependences as all the data are required. The copy-out set is left empty. For I/O groups with flow dependences, the copy-in set consists of the destination domains of the dependences and the copy-out domain consists of the source domains. Lastly, for I/O groups with output dependences, we compute the copy-out set by subtracting the source domains from the destination domains as we are only interested in the last updated elements. The copy-in set is left empty. Table 3 includes the final I/O groups extracted from the MM example and their copy-in/copy-out sets. They will be used for I/O network construction in the next section.

<table>
<thead>
<tr>
<th>Type</th>
<th>Dependence Relation</th>
<th>Array Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>$D1 := {S0[i, j, k] \rightarrow S0[i, j, k + 1]}$</td>
<td>$A[1][k]$</td>
</tr>
<tr>
<td>Read</td>
<td>$D2 := {S0[i, j, k] \rightarrow S0[i + 1, j, k]}$</td>
<td>$B[k][j]$</td>
</tr>
<tr>
<td>Flow</td>
<td>$D3 := {S0[i, j, k] \rightarrow S0[i, j, k + 1]}$</td>
<td>$C[i][j]$</td>
</tr>
<tr>
<td>Output</td>
<td>$D4 := {S0[i, j, k] \rightarrow S0[i, j, k + 1]}$</td>
<td>$C[i][j]$</td>
</tr>
</tbody>
</table>

### 6.2 I/O Construction

This step constructs the I/O modules based on the I/O grouping information extracted from the previous step. For each I/O group, AutoSA allocates a set of I/O modules for transferring the data between PEs and the external memory. Algorithm 3 describes the detailed procedure of this step.

- **Algorithm 2: I/O group construction.**
  - **Inputs**: Access relations $A$, dependence relations $D_{read}, D_{flow}, D_{output}$, schedule $s$
  - **Outputs**: A set of I/O groups $G$
  - Initialize the I/O group set $G \leftarrow \emptyset$;
  - #Populate the I/O groups. */
  - for each array access $acc$ in $A$
do
  - for each dependence $d$ in $D_{read}, D_{flow}, D_{output}$
do
  - if $acc$ is associated with $d$
do
  - Construct a new I/O group $g(acc, d)$;
  - Compute the properties of the group $g$: I/O direction $g.dir$ and I/O type $g.type$;
  - $G \leftarrow G \cup g$;
  - /* Merge the I/O groups. */
do
  - for each pair of I/O groups $(g1, g2)$ in $G$
do
  - if is_shared($g1, g2$) then
  - Merge the two I/O groups $g \leftarrow merge(g1, g2)$;
  - Update the I/O group set $G \leftarrow G \setminus \{g1 \cup g2\}$;
  - /* Compute the I/O group copy-in/copy-out sets. */
do
  - for each I/O group $g$ in $G$
do
  - if $g$ is associated with read dependences then
  - $g.W_{in} \leftarrow \bigcup \{src(d) \cup dst(d)\}$;
  - if $g$ is associated with flow dependences then
  - $g.W_{in} \leftarrow \bigcup \{dst(d)\}$;
  - if $g$ is associated with output dependences then
  - $g.W_{out} \leftarrow \bigcup \{src(d)\}$;

- **Algorithm 3: I/O construction.**
  - **Inputs**: Schedule $s$, I/O groups $G$, number of space loops $dim$
  - **Outputs**: A list of schedules for I/O modules $L$
  - $L \leftarrow \emptyset$;
  - /* Copy-in modules */
do
  - for each I/O group $g$ in $G$
do
  - Duplicate the schedule $s' \leftarrow s$;
  - Insert the domain filter $g.W_{in}$ into the schedule $s'$;
  - $io.level \leftarrow 1$;
  - while $io.level \leq dim$
do
  - Perform I/O module clustering on the first $dim - io.level + 1$ space loops $s' = io.clustering(s', dim - io.level + 1, g)$;
  - Add $s'$ to $L$;
  - $io.level \leftarrow io.level + 1$;
  - /* Copy-out modules (omitted for brevity) */
do

We start with the optimized schedule from the computation management. In the first step, we isolate the statement instances
that are involved with the data communication from the current group by inserting a filter node into the schedule tree with the copy-in/copy-out set. The filter node restrains the iteration domains of its children nodes by intersecting the current iteration domain with the filter set \[49, 51\]. As an example, Figure 6 shows the updated schedule with the filtered domain for the I/O group \(g_2\) in Table 3 (loops inside the space loops are omitted for brevity). At this stage, we could already generate a set of I/O modules that load the data from the external memory and send the data directly to each PE. This can be realized by equating the space loops to the PE indices \(idx\) and \(idy\) in the updated schedule and using it to generate the code inside each I/O module. Figure 7 shows the generated array and the corresponding schedule for each I/O module.

However, this architecture may not be scalable as the data are scattered directly from the external memory which causes high fan-outs and could lead to routing failure. To resolve this issue, we choose to localize the I/O network by using a daisy-chain architecture that have been seen in many previous works \[11, 37, 48, 52\]. In this architecture, each I/O module fetches data from the upper-stream I/O modules. The I/O module works as a filter that keeps the data belonging to the PEs that it is associated with and passes the rest of the data to the down-stream I/O modules. As for the architecture in Figure 7, we name the I/O modules that are directly connected to PEs as level-one (L1) I/O modules. We could first cluster the L1 I/O modules along the x-axis, as shown in Figure 8a. Every two L1 modules along the x-axis are connected to an upper-level (L2) I/O modules, which helps to reduce the memory fan-outs from four to two. We name such a process as I/O clustering. I/O clustering can be applied multiple times in a hierarchical way. For example, we could apply the I/O clustering again on the L2 I/O modules, generating one L3 I/O module that connects to the DRAM.

6.3 I/O Optimization

In this step, AutoSA applies multiple passes to further optimize the I/O network.

I/O module embedding: L1 I/O modules with exterior I/O are embedded into the PEs to save the resource.

I/O module pruning: When transferring the data between different sub-array tiles, AutoSA checks if the copy-out set of the previous tile equals the copy-in set of the current tile at the PE level. If two sets are equal at the PE level, it indicates the data are located on-chip and hence the data transfer from the external memory is unnecessary. For such a case, the I/O modules for this I/O group will be pruned away since the data of matrix \(C\) are accumulated locally inside each PE. Figure 10a depicts the final array architecture after the I/O clustering for all the I/O groups in Table 3.

Figure 6: Insert the filter to isolate the statement instances of group \(g_2\).

Figure 7: PE array with the I/O module for \(g_2\) and its loop schedule.

Figure 8: I/O clustering example for group \(g_2\).

Figure 9: Updated schedule for the clustered L1 I/O module by applying the transformation \(T_1 := [c0, c1] \rightarrow [c1, c0]\) on the space loops in the original schedule as shown in Figure 7.

Figure 10: A complete 2D systolic array for the MM example, as shown in Figure 8b. Eventually, we reduce the memory fan-outs from four to one.
it leads to FIFOs with a larger width and higher resource usage. Therefore, AutoSA offers options to set the data packing factor at each I/O level, which can also be set by the auto-tuner during the design space exploration.

**Double buffering:** By default, AutoSA allocates a local buffer inside the L1 I/O modules for I/O groups with interior I/O or inside the L2 I/O modules for I/O groups with the exterior I/O. For such I/O modules with local buffers inside, AutoSA offers options to enable the double buffering that helps overlap the memory transfer with the PE computation.

7 AUTO-TUNING

7.1 Problem Statement

The optimizations described in the previous sections introduce many design factors that compose a large design space which is impractical to explore manually. AutoSA provides an auto-tuner to find a good design with high performance.

Given an input program $P$ and a target FPGA device $D$, AutoSA searches for the design with the least latency without over-utilizing the on-chip resource. The optimization problem is summarized as:

\[
\begin{align*}
\text{minimize} & \quad \text{latency}(x) \\
\text{subject to} & \quad \text{resource}_i(x) \leq b_i(D), \quad i = \text{FF, LUT, DSP, BRAM}
\end{align*}
\]

where $DF(P)$ includes all legal design factor choices of the program $P$, and $b_i(D)$ is the resource limit for different types of resource $i$ on-chip. \text{latency}(x) and \text{resource}_i(x)$ are the latency and resource usage of the design optimized with the design factor $x$.

7.2 Resource and Latency Modeling

AutoSA builds analytical models to estimate the resource and latency of the target design.

**Resource modeling:** AutoSA randomly samples the design space to build a suite of training samples (16 samples for the current implementation). Then, we run HLS synthesis to collect the resource usage of each design. Based on the collected synthesis results, AutoSA builds a linear regression model to predict the resource usage of FF, LUT, and DSP. As for input features, based on our experiments, three features including the SMD factor, data packing factor, and local buffer sizes suffice to provide an acceptable prediction accuracy. BRAM usage is directly calculated based on the local buffer sizes. The resource models achieve an error rate within 10% on all the evaluated benchmarks.

**Latency modeling:** The latency of a pipeline loop can be calculated as \text{loop_counts} $\times$ II $+$ \text{loop_depth}. AutoSA extracts the \text{loop_counts} from the generated AST for each module. We set II as one by default. For the \text{loop_depth}, we set it as one as an estimation for loops without any statements accessing the DRAM. For loops with statements accessing the DRAM, AutoSA further examines the memory coalescing and sets the \text{loop_depth} as 182 ns as an approximation for the non-coalesced access obtained from a recent work [29]. As the entire systolic array is implemented using a dataflow architecture, the final design latency is calculated as the maximum of all the modules. The latency model achieves an error rate within 5% on all the evaluated benchmarks.

### Table 4: Benchmark description.

<table>
<thead>
<tr>
<th>Application</th>
<th>Problem Size</th>
<th>#Stmts</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix Multiplication</td>
<td>$[i, j, k]$ : [1024, 1024, 1024]</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>CNN</td>
<td>$[i, n, h, w, e, p, q]$ : [512, 512, 512, 512]</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>MTTKPP [48]</td>
<td>$[i, k, l, j]$ : [512, 512, 512, 512]</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>TTMc [48]</td>
<td>$[i, j, k, l, m]$ : [128, 128, 128, 128, 128]</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>LU Decomposition</td>
<td>$[n]$ : [12/16/20/24]</td>
<td>9</td>
<td>27</td>
</tr>
</tbody>
</table>

The auto-tuner examines all designs using exhaustive search. We have also applied several pruning strategies to shrink the design space. Besides, the auto-tuning is multi-processed to further speedup the procedure. With these optimizations, the auto-tuner can finish within hours on a normal work station for all the experiments evaluated in this paper. In the future, we will improve the tuning efficiency by considering recent design space exploration works that leverage the machine learning techniques [2, 24, 38, 47].

8 EVALUATION

AutoSA is built upon PPCG [50]. The core of AutoSA is implemented in C/C++ with about 30K lines of code. The auto-tuner is written in Python. We use Xilinx Vitis 2019.2 for synthesizing and implementing the FPGA designs and target Xilinx Alveo U250 board. We also adopt AutoBridge [18, 19] to improve the design frequency. Table 4 describes the details of the benchmarks that we used to evaluate AutoSA.

In the following sections, we first perform two case studies on matrix multiplication and LU decomposition. We use the case study of matrix multiplication to assess the performance of AutoSA and the case study of LU decomposition to assess the generality of the AutoSA. Lastly, we present the rest of the results on the other benchmarks and evaluate the productivity of our tool.

8.1 Case Study 1: Matrix Multiplication

AutoSA is able to generate six different systolic arrays for MM. This is realized by selecting loops $\Theta_i, \Theta_j$, and $\Theta_k$ as the space loop for 1D arrays, and loops $\Theta(i, j), \Theta(i, k)$, and $\Theta(j, k)$ as space loops for 2D arrays. We denote these six arrays as designs 1-6 in sequence. Among these six designs, designs 1 and 2, designs 5 and 6 are symmetric. Finally, we choose to conduct experiments on designs 1, 4, and 5 for simplicity. Figures 11a, 11b, and 11c depict the architecture of these three designs.

In design 1, as shown in Figure 11a, loop $i$ is assigned as the space loop. As a result, matrix $A$ is associated with interior I/O and is fed to each PE directly. The elements of matrix $B$ are reused along the $i$-axis. Each PE accumulates the elements of matrix $C$ locally. Therefore, we allocate a local buffer for matrix $C$ (as denoted by $bufC$) in the PE to store the intermediate results. After the computation is finished, the final results of matrix $C$ are drained out and sent to the DRAM. Such an architecture can be found in previous works like [13, 21].

Design 4, as shown in Figure 11b, is generated by selecting loops $i$ and $j$ as the space loops. The elements of matrix $A$ and $B$ are reused along the $j$-axis and $i$-axis, respectively. The data of matrix $C$ are accumulated inside PEs and will be drained out after the computation is completed. Such an architecture can be found in previous works [10, 30, 37, 48].
Design 4, as shown in Figure 11c, is generated by choosing loops \(i\) and \(k\) as the space loops. The key difference between design 5 and design 4 is that the elements of matrix \(C\) are now accumulated along the \(k\)-axis. Therefore, the local buffer (bufC) is saved. However, the data from matrix \(B\) need to be sent separately to each PE via L1 I/O modules. As shown in Figure 11b, design 4 exploits the reuse of matrix \(A\) and therefore only generates \(B\) loops. The iteration domain is in a pyramid shape which cannot be reduced to a simpler I/O network. As shown in Figure 11b, design 4 exploits both the data reuse of matrix \(A\) and \(B\) and therefore only generates \(L2\) I/O modules at the PE boundary. However, design 5 (shown in Figure 11c) only exploits the reuse of matrix \(B\). Data elements of matrix \(A\) need to be sent separately to each PE via \(L1\) I/O modules. This increases the routing complexity and limits the design scale that we could explore.

We found that design 4 achieves a balance in terms of the resource and routing complexity and therefore achieves the highest performance among these designs. The comprehensive design space that AutoSA provides and the generality of both the front-end and back-end of AutoSA enable us to explore such as a design space for various studies. This case study shows one example for architectural exploration. We are also working on adding power as one of the new metrics in the auto-tuner to provide more architectural insights into the systolic array architecture.

Furthermore, we compare the best designs generated by AutoSA with other systolic array compilation frameworks. AutoSA supports different data types. Table 6 shows the best results that AutoSA achieved in the floating point, 16-bit and 8-bit integer types, as well as numbers from the previous works.

AutoSA achieves 934 GFLOPs for the floating point. As for int16, since the number of DSPs for each MAC operation is reduced from 5 (in FP32) to 1, the performance is improved to 3.41 TOPs. For int8, we combine the logic and DSPs to implement the MACs and achieve 6.95 TOPs. AutoSA achieves higher throughput compared to the previous works with more DSPs utilized or higher frequency. To better understand the performance, we also compare the DSP efficiency for FP32 designs. AutoSA achieves similar DSP efficiency compared to the previous works.

### 8.2 Case Study 2: LU Decomposition

LU decomposition is an important kernel that has been used in solving the systems of linear equations. It factorizes a matrix \(A\) as the product of a lower triangular matrix \(L\) and an upper triangular matrix \(U\) \((A = L \times U)\). We choose the algorithm implemented in PolyBench [56]. Figure 12a shows the dependence graph of the LU algorithm. In the dependence graph, each node represents a loop instance in the program. Nodes are connected if there is any dependence associated with the loop instance they represent. The dependence structure of LU decomposition has introduced several new challenges to the systolic array compilation frameworks compared to other regular kernels such as matrix multiplication:

- The iteration domain is in a pyramid shape which cannot be handled by the current Halide-based frameworks that only support rectangular domains.
- The statement instances are non-uniform and conduct different computations. As shown in Figure 12a, there are three types of nodes marked by different shades. This has introduced a more complex dependence structure which is challenging to analyze manually.

We use the LU decomposition as a stress test to assess the robustness of our framework. With the general algorithms we have proposed in the previous sections, AutoSA is able to compile such an application and generate the systolic array. Figure 12b shows an example array generated by AutoSA by choosing the loops \(j\) and \(k\) as the space loops. This mapping leads to a 2D systolic array in a triangular shape. In this array, data of matrix \(A\) are fed only to the first row of the array. The final results of matrix \(U\) are drained out from all the PEs, while the results of matrix \(L\) are drained only from the PEs on the diagonal of the array. Such an architecture can be found in several previous manual designs [27, 44].
We have further evaluated AutoSA on three other benchmarks, including the convolutional kernel from the convolutional neural network (CNN) and two tensor contraction kernels, matricized tensor times Khatri-Rao product (MTTKRP), and tensor times matrix-chain (TTMc) as studied by previous works [30, 48]. Table 7 presents the results from the other systolic array compilation works. The FPGA results here are collected from RTL simulation with the assumed frequency of 250MHz. The LAPACK routine is evaluated on a server with an Intel Xeon E5-2699 v3 CPU and 189 GB of main memory. We call the functions 10000 times and calculate the average as the final results.

The systolic array achieves an average speedup of 6.8x compared to the LAPACK baseline. This result is not surprising as all the PEs are fully-pipelined and the systolic array extracts the maximal pipeline parallelism from the application with the dataflow-like architecture.

### 8.3 Other Results

We would like to thank Marci Baun for helping edit the paper and the anonymous reviewers for their valuable feedbacks. This work is partially supported by the Intel and NSF joint research center for Computer Assisted Programming for Heterogeneous Architectures (CAPA), NSF NeuroNex Award DBI-1707408, and the members from the CDSC Industrial Partnership Program. We acknowledge the valuable support of the Xilinx Adaptive Compute Clusters (XACC) Program. We also appreciate the authors of PPCG for open-sourcing the tool.

### ACKNOWLEDGMENTS

As there is no reported numbers for LU decomposition in the previous automation frameworks due to its high irregularity, we compare the performance of the generated arrays with LAPACK benchmark [39]. The comparison results are shown in Table 8. The FPGA results here are collected from RTL simulation with the assumed frequency of 250MHz. The LAPACK routine is evaluated on a server with an Intel Xeon E5-2699 v3 CPU and 189 GB of main memory. We call the functions 10000 times and calculate the average as the final results.

The systolic array achieves an average speedup of 6.8x compared to the LAPACK baseline. This result is not surprising as all the PEs are fully-pipelined and the systolic array extracts the maximal pipeline parallelism from the application with the dataflow-like architecture.

### Table 7: Comparison results for CNN, MTTKRP, and TTMc.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Platform</th>
<th>SA Sizes (RowxColxSIMD)</th>
<th>Data Type</th>
<th>LUT</th>
<th>FF</th>
<th>BRAM</th>
<th>DSP</th>
<th>GPFLOPs</th>
<th>MHz</th>
<th>DSP Efficiency</th>
<th>Input Code LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNN</td>
<td>Intel Arria 10</td>
<td>8x10x8</td>
<td>FP32</td>
<td>59%</td>
<td>40%</td>
<td>47%</td>
<td>47%</td>
<td>35%</td>
<td>81%</td>
<td>662.8</td>
<td>253</td>
</tr>
<tr>
<td>MTTKRP</td>
<td>Ours Xilinx Alveo U250</td>
<td>16x14x8</td>
<td>FP32</td>
<td>58%</td>
<td>46%</td>
<td>18%</td>
<td>18%</td>
<td>73%</td>
<td>956.2</td>
<td>272</td>
<td>97%</td>
</tr>
<tr>
<td>TTMc</td>
<td>Intel Arria 10</td>
<td>8x9x16</td>
<td>FP32</td>
<td>N/A</td>
<td>N/A</td>
<td>56%</td>
<td>81%</td>
<td>700</td>
<td>264</td>
<td>99%</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>Ours Xilinx Alveo U250</td>
<td>16x9x8</td>
<td>FP32</td>
<td>42%</td>
<td>32%</td>
<td>26%</td>
<td>67%</td>
<td>886.2</td>
<td>296</td>
<td>N/A</td>
<td>9</td>
</tr>
</tbody>
</table>

As seen in the table, AutoSA achieves higher throughput than the previous work on all the benchmarks with more DSPs utilized and higher frequency. With the wide coverage of different hardware optimization techniques and the help of an auto-tuner, the designs generated by AutoSA achieve an average DSP efficiency of 99%.

Figure 13 presents an ablation study of a few optimization techniques. AutoSA requires minimal lines of code as the input (i.e., C). The polyhedral compilation takes a few seconds, and the training and searching of the auto-tuner takes one to two hours. Taking into account the time for FPGA tools to synthesize and implement the designs, which usually finishes within one day, AutoSA is able to generate high-performance designs within one or two days, which significantly boosts the productivity of developing systolic arrays.

### 9 CONCLUSION

This paper presents AutoSA, an open-source compiler framework for generating high-performance systolic arrays on FPGA. We present general techniques and optimizations implemented in the polyhedral framework that help improve the compute and communication efficiency of systolic array designs. We evaluate AutoSA on a suite of benchmarks and achieve high performance. AutoSA strikes a balance between generality, performance, and productivity. We hope such a tool can facilitate more architectural studies and applications on systolic arrays. Future work includes the backward support to Intel platforms, adding the power metric to the auto-tuner, and improving the auto-tuning efficiency.