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:
Jie Wang, Licheng Guo, and Jason Cong. 2021. AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA. In *Proceedings of the 2021 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '21), February 28-March 2, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3431920.3439292

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, 23, 26, 52, 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 MMAlpha [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.

FPGA '21, February 28-March 2, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8218-2/21/02…$15.00
https://doi.org/10.1145/3431920.3439292

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., MMAlpha [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. For example, AutoSA achieves up to 934 GFLOPs, 3.41 TOPs, and 6.95 TOPs for the floating point/16-bit integer/8-bit integer matrix multiplication on Xilinx Alveo U250, respectively.

[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

Table 1: Comparison between different frameworks.

Feature	AutoSA	MMAlpha [17]	[7]	PolySA [10]	SuSy [30]
Generality					
Imperfectly Nested Loops	Yes	No	No	No	No
Multi-Statement	Yes	Yes	No	No	Yes
Performance					
Array Partitioning	Auto	No	Auto	Auto	Semi-Auto
Latency Hiding	Auto	No	No	Auto	Semi-Auto
SIMD Vectorization	Auto	No	No	Limited	Semi-Auto
Double Buffering	Auto	No	No	Auto	Semi-Auto
Data Packing	Auto	No	No	Limited	Semi-Auto
Productivity					
Input	C	DSL	C	C	DSL
Auto-Tuning	Yes	No	No	Yes	No
Space-Time Transformation	Auto	Semi-Auto	Auto	Auto	Semi-Auto

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]. Gemmini [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 CGRAs [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 conditions 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[3].

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 $\{S0[i,j,k] : 0 \le i < M \wedge 0 \le j < N \wedge 0 \le k < K\}$. Throughout the paper, to represent the components of the polyhedral model, we use the same format as *integer set library* (isl) [49], which is a library for polyhedral compilation.

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 $\{S0[i,j,k] \rightarrow A[i,k]; S0[i,j,k] \rightarrow B[k,j]; S0[i,j,k] \rightarrow C[i,j]\}$.

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 $\{S1[i,j,k] \rightarrow [i,j,k]\}$. The schedule 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

[3]The matrix initialization is omitted for brevity.

```
for (int i = 0; i < M; ++i)
    for (int j = 0; j < N; ++j)
        for (int k = 0; k < K; ++k)
S0:         C[i][j] += A[i][k] * B[k][j];
```

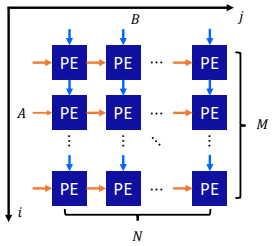
Figure 1: Example code of matrix multiplication.

DOMAIN : $\{S0[i,j,k]: 0 \le i < M \wedge 0 \le j < N \wedge 0 \le k < K\}$
BAND : $\{S0[i,j,k] \rightarrow [i,j,k]\}$

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

```
// space loops
for (int i = 0; i < M; ++i)
  for (int j = 0; j < N; ++j)
    // time loop
    for (int k = 0; k < K; ++k)
S0:   C[i][j] += A[i][k] * B[k][j];
```

(a) Optimized code after space-time transformation.



(b) 2D systolic array for MM.

Figure 3: 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 $D2 := \{S0[i,j,k] \rightarrow S0[i,j+1,k]\}$ and $D3 := \{S0[i,j,k] \rightarrow S0[i+1,j,k]\}$, 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

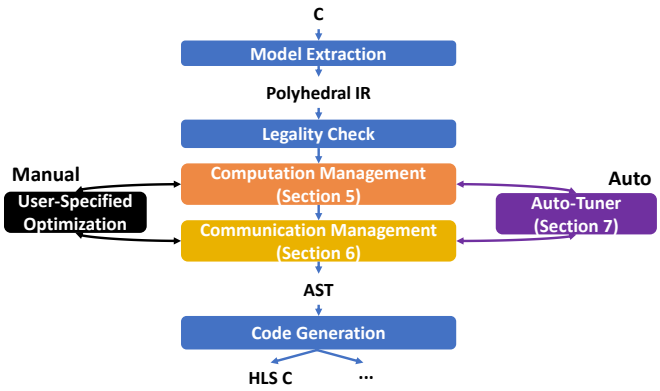


Figure 4: Overview of AutoSA compilation flow.

discussion about output and anti dependences for brevity). Besides, dependence distances on all three loops are no greater than one. Therefore, all three loops are eligible to be selected as the space loops. As an example, we select the first two loops i and j as space loops and leave the loop k as the time loop. The transformed code after space-time transformation is shown in Figure 3a. This transformation leads to a 2D systolic array with the dimensions of $M \times N$. Each PE in this array computes one array element `C[i][j]`. In addition, according to the two read dependences $D2$ and $D3$, data of matrices A and B are reused along the loops j and i, respectively. Therefore, horizontal and vertical interconnects between PEs are inserted to transfer the reused data. The complete array is depicted in Figure 3b.

4 FRAMEWORK OVERVIEW

In this section, we first clarify the application scope that AutoSA covers. Then, we give a bird's-eye view of the overall compilation flow of AutoSA.

4.1 The Scope of AutoSA

AutoSA is built on the polyhedral framework, which takes SCoP programs as the input. In addition, AutoSA assumes all the dependences of the input programs have been rendered uniform before the compilation. Prior studies on uniformization such as [35] which converts affine dependences to uniform dependences can be integrated as a pre-processing step to relax this contraint. This work is orthogonal to our framework and will be left for future work.

4.2 Compilation Flow

Figure 4 presents the overall compilation flow of AutoSA.

Model extraction: This step extracts the polyhedral model consisted of the iteration domains, access relations, schedule, and data dependences from the input C program.

Legality check: This step checks to see if the input program can legally be mapped to a systolic array. Specifically, we check the three constraints as mentioned in Section 3.2. If the new schedule fails to meet any of those constraints, AutoSA will skip the following steps and dump out a CPU code from the current schedule.

Computation and communication management: A complete systolic array architecture consists of both the PE array and the on-chip I/O network. AutoSA separates the process of building 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.

Algorithm 1: Space-time transformation.

Inputs : A schedule tree s
Outputs: A list of schedule trees S
Initialize the space loop candidate pool $P \leftarrow \emptyset$;
Extract the outermost permutable loop band d from s;
for *each loop l in the band d* **do**
  if *all flow/read dependence distances on loop $l \leq 1$* **then**
    $P \leftarrow P \cup l$;
/* Generate 2D systolic array. */
for *each pair of loops (l_1, l_2) in the pool P* **do**
  Duplicate the schedule tree $s' \leftarrow s$;
  Modify s' by permuting the loops l_1, l_2 to outermost;
  $S \leftarrow S \cup s'$;
/* Generate 1D systolic array (omitted), similar to 2D case with only one space loop selected. */

5 COMPUTATION MANAGEMENT

The stage of computation management consists of four steps: space-time transformation, array partitioning, latency hiding, and SIMD vectorization. We will go though each step in the following subsections.

5.1 Space-Time Transformation

This step performs the space-time transformation to map the input program to a systolic array. Algorithm 1 describes how AutoSA applies the space-time transformation. AutoSA searches for the loops in the outermost loop band with flow/read dependence distances no greater then one. Those loops are put into a pool as the candidate space loops. Next, AutoSA enumerates all space loop combinations from the candidate pool. The selected space loops are permuted outermost. All the loops below the space loops are assigned as time loops. 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

the outermost permutable 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 $(4, 4, 4)$. 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 4×4.

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 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[4]. 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

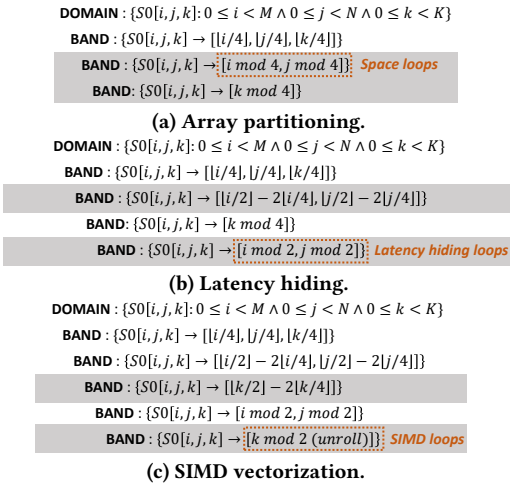


Figure 5: Optimized schedules with different PE optimization techniques.

for the MM example, we could transfer data elements from matrix A and B via horizontal and vertical interconnects.

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

[4]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.

Table 2: Dependence relations of the MM example.

Type	Dependence Relation	Array Access
Read	$D1 := \{S0[i,j,k] \rightarrow S0[i,j+1,k]\}$	`A[i][k]`
Read	$D2 := \{S0[i,j,k] \rightarrow S0[i+1,j,k]\}$	`B[k][j]`
Flow	$D3 := \{S0[i,j,k] \rightarrow S0[i,j,k+1]\}$	`C[i][j]`
Output	$D4 := \{S0[i,j,k] \rightarrow S0[i,j,k+1]\}$	`C[i][j]`

data structure named *I/O group* that contains the necessary information required to construct the I/O network. Algorithm 2 describes the details of this step.

An I/O group g is defined as a tuple of $g = (A, D)$ 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 array 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.

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* **then**
      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. */`
for *each pair of I/O groups* (g_1, g_2) *in G* **do**
  if *is_shared*(g_1, g_2) **then**
    Merge the two I/O groups $g \leftarrow merge(g_1, g_2)$;
    Update the I/O group set $G \leftarrow G \setminus (g_1 \cup g_2) \cup g$;
`/* Compute the I/O group copy-in/copy-out sets. */`
for *each I/O group g in G* **do**
  if *g is associated with read dependences* **then**
    $g.W_{in} \leftarrow \cup_{d_i}(src(d_i) \cup dst(d_i))$;
  if *g is associated with flow dependences* **then**
    $g.W_{in} \leftarrow \cup_{d_i} dst(d_i)$;
    $g.W_{out} \leftarrow \cup_{d_i} src(d_i)$;
  if *g is associated with output dependences* **then**
    $g.W_{out} \leftarrow \cup_{d_i}(dst(d_i) - src(d_i))$;

Table 3: I/O groups of the MM example.

No.	A	D	W_{in}/W_{out}
g_1	`A[i][k]`	$D1$	$W_{in} := S0[i,j,k] : 0 \le i < M \wedge 0 \le j < N \wedge 0 \le k < K\}$
g_2	`B[k][j]`	$D2$	$W_{in} := S0[i,j,k] : 0 \le i < M \wedge 0 \le j < N \wedge 0 \le k < K\}$
g_3	`C[i][j]`	$D2$	$W_{in} := S0[i,j,k] : 0 \le i < M \wedge 0 \le j < N \wedge 0 < k < K\}$
			$W_{out} := S0[i,j,k] : 0 \le i < M \wedge 0 \le j < N \wedge 0 \le k < K-1\}$
g_4	`C[i][j]`	$D3$	$W_{out} := S0[i,j,k = K-1] : 0 \le i < M \wedge 0 \le j < N$

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 */`
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 $io\_level \le dim$ **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) */`

We start with the optimized schedule from the computation management. In the first step, we isolate the statement instances

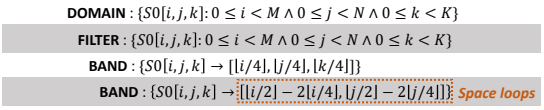


Figure 6: Insert the filter to isolate the statement instances of group g_2.

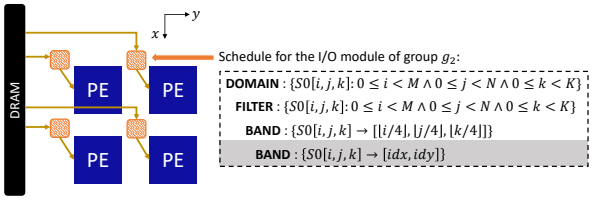
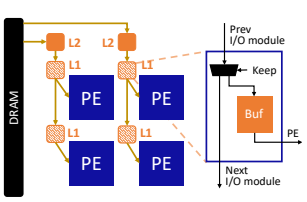
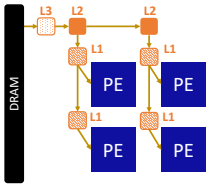


Figure 7: PE array with the I/O module for g_2 and its loop schedule.



(a) Clustering L1 I/O modules.



(b) Clustering L2 I/O modules.

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

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,

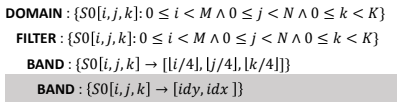
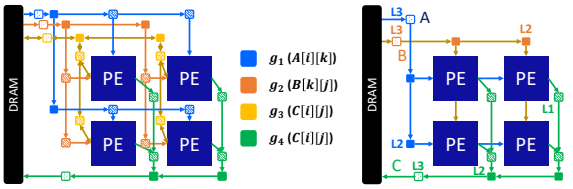


Figure 9: Updated schedule for the clustered L1 I/O module by applying the transformation $T1 := [c0, c1] \rightarrow [c1, c0]$ on the space loops in the original schedule as shown in Figure 7.



(a) Initial array. **(b) Optimized array.**

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.

I/O clustering is realized by applying loop transformation on the space loops. For example, when clustering the L1 I/O modules for the group g_2, since the group g_2 has the I/O type as exterior I/O and I/O direction as $(1, 0)$, indicating that data are reused along the x-axis, we will cluster the I/O modules along the $(1, 0)$ direction. This is achieved by applying a partial schedule transformation $T1 := \{[c_0, c_1] \rightarrow [c_1, c_0]\}$ on the space loops. Figure 9 shows the updated schedule for the L1 I/O module. Next, when clustering the L2 I/O modules, only the first space dimension is involved. For simplicity, we apply an identity transformation $T2 := \{[c0] \rightarrow [c0]\}$ that clusters the L2 I/O modules along the y-axis. Note that, when choosing the I/O clustering directions for the L1 I/O modules of the I/O groups with exterior I/O, AutoSA uses the default I/O direction of the group. As for the rest of the cases, at present, AutoSA simply clusters the I/O modules vertically or horizontally. However, users can also provide their own inputs to direct the I/O clustering manually. Figure 10a depicts the final array architecture after the I/O clustering for all the I/O groups in Table 3.

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 are pruned away to save the off-chip communication and on-chip resource. As an example, for the MM example, the I/O modules for the group g_3 will be pruned away since the data of matrix C are accumulated locally inside each PE. Figure 10b shows the optimized array by applying two techniques as mentioned above.

Data packing: To reduce the data transfer latency between the I/O modules, AutoSA performs data packing between I/O modules. Packing more data helps reduce the data transfer latency, however,

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:

$$\begin{aligned} \underset{x \in DF(P)}{\text{minimize}} \quad & latency(x) \\ \text{subject to} \quad & resource_i(x) \leq b_i(D), \\ & i = \text{FF, LUT, DSP, BRAM} \end{aligned}$$

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. $latency(x)$ and $resource_i(x)$ 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 SIMD 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 pipelined loop can be calculated as $loop\_counts \times II + loop\_depth$. AutoSA extracts the $loop\_counts$ from the generated AST for each module. We set II as one by default. For the $loop\_depth$, 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 $loop\_depth$ 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.

Application	Problem Size	#Stmts	LOC
Matrix Multiplication	$[i, j, k] : [1024, 1024, 1024]$	2	7
CNN	$[i, o, h, w, p, q] : [512, 512, 56, 56, 3, 3]$	2	10
MTTKRP [48]	$[i, k, l, j] : [512, 512, 512, 512]$	2	9
TTMc [48]	$[i, j, k, l, m] : [128, 128, 128, 128, 128]$	2	9
LU Decomposition	$[n] : [12/16/20/24]$	9	27

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-tuning 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.

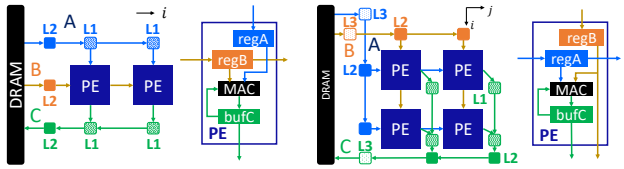
In the following sections, we first perform two case studies on matrix multiplication and LU decomposition. We use the case study of matrix multiplication to assess the performance of AutoSA and the case study of LU decomposition to assess the generality of the AutoSA. Lastly, we present the rest of 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 ❶i, ❷j, and ❸k as the space loop for 1D arrays, and loops ❹(i, j), ❺(i, k), and ❻(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].



(a) Design 1. (b) Design 4.

(c) Design 5.

Figure 11: The array architecture of MM designs 1, 4, 5.

Table 5: Performance comparison of different MM designs.

Designs	SA Sizes	MHz	GFLOPs	LUT	FF	BRAM	DSP
1	128×8	346	555	38%	31%	10%	42%
4	13×16×8	300	934	52%	42%	41%	68%
5	13×12×8	300	660	46%	37%	10%	51%

Design 5, 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 A need to be sent directly to each PE. The data of matrix B are reused along the i-axis. This architecture can be seen in previous works [15, 23].

We use designs 1 and 4 to study the impacts of different array dimensions, and use designs 4 and 5 to study the impacts of different space mappings. All the experiments are in floating point. Table 5 shows the detailed results of these three designs.

We observe several interesting trade-offs from this table:

Design 1 vs. Design 4: 1D systolic array limits the design space with one less space dimension to be explored. The best 1D systolic array we found has 128 PEs with a SIMD factor as 8. Placing more PEs will either lead to routing failure or wasted cycles for computing the padded elements that lowers the effective GFLOPs. However, 1D systolic array achieves a higher frequency than the 2D counterpart, which is contributed by both a more regular architecture and less resource.

Design 4 vs. Design 5: We are able to place more PEs for design 4 compared to design 5, albeit this requires more resource. This is due 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.

Table 6: MM performance comparison with previous works.

	PolySA [10]	SuSy [30]	[37]			[22]		AutoSA		
Data Type	FP32	FP32	FP32	int16	int8	FP32	int16	FP32	int16	int8
Platform	Xilinx VU9P	Intel Arria 10	Intel Arria 10			Xilinx VU9P		Xilinx Alveo U250		
MHz	229	202	315	N/A	N/A	265	323	300	250	300
DSP	89%	93%	100%	N/A	N/A	75%	60%	68%	67%	52%
LUTs/ALMs	49%	40%	N/A	N/A	N/A	78%	5%	52%	38%	45%
TOPs	0.56	0.55	0.8	1.53	3.83	0.68	2.64	**0.93**	**3.41**	**6.95**
DSP Efficiency	98%	96%	99%			100%		**94%**		

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

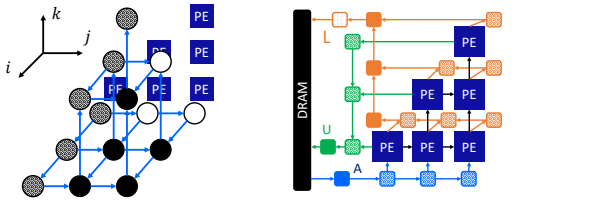
LU decomposition is an important kernel that has been used in solving the systems of linear equations. It factorizes a matrix A as the product of a lower triangular matrix L and an upper triangular matrix U ($A = L \times U$). We choose the algorithm implemented in PolyBench [56]. Figure 12a shows the 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 L are drained out from all the PEs, while the results of matrix U 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].

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

Benchmark		Platform	SA Sizes (Row×Col×SIMD)	Data Type	LUT	FF	BRAM	DSP	GFLOPs	MHz	DSP Efficiency	Input Code LOC
CNN	[52]	Intel Arria 10	8×19×8	FP32	59%	40%	47%	81%	602.8	253	97%	10
	Ours	Xilinx Alveo U250	16×14×8	FP32	58%	46%	30%	73%	**950.2**	272	97%	10
MTTKRP	[48]	Intel Arria 10	8×9×16	FP32	N/A	N/A	56%	81%	700	204	99%	32
	Ours	Xilinx Alveo U250	16×8×8	FP32	42%	32%	26%	67%	**896.7**	296	99%	9
TTMc	[48]	Intel Arria 10	8×10×16	FP32	N/A	N/A	62%	90%	738	205	94%	38
	Ours	Xilinx Alveo U250	16×8×8	FP32	42%	32%	21%	67%	**886.2**	290	99%	9



(a) Dependence graph. (b) Generated array.

Figure 12: The dependence graph of LU decomposition and the mapped array.

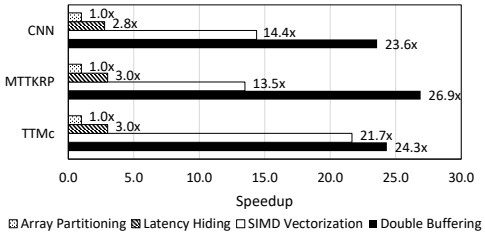


Figure 13: Ablation analysis of several hardware optimization techniques.

Table 8: LU performance comparison with LAPACK.

Matrix Size	12	16	20	24	Geo. Mean
LAPACK	11.69us	41.06us	78.09us	83.38us	
Ours	3.42us	5.23us	7.68us	10.44us	
Speedup	3.4×	7.8×	10.2×	8.0×	**6.8×**

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 6.8× 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 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 detailed results of these benchmarks and compares them with the results from the other systolic array compilation works.

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 applied by AutoSA. All the other techniques are enabled by default. We collect the design latency by incrementally applying each technique. All the performance numbers are normalized against the baseline applied with array partitioning. As can be seen from the figure, all the techniques in the computation management combined together bring an average speedup of 16.1×. The double buffering in the communication management further increases the throughput, on average, by 1.5×. The benefits of double buffering on TTMc is minimal as the application is more compute-intensive. This indicates the needs for an auto-tuner to make the right trade-offs between different design factors.

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 backend support to Intel platforms, adding the power metric to the auto-tuner, and improving the auto-tuning efficiency.

ACKNOWLEDGMENTS

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.

REFERENCES

[1] Amazon. 2020. *AWS Inferentia.* https://aws.amazon.com/machine-learning/inferentia

[2] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*. 303–316.

[3] Marcus Bednara and Jürgen Teich. 2003. Automatic synthesis of FPGA processor arrays from loop algorithms. *The Journal of Supercomputing* 26, 2 (2003), 149–165.

[4] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. 2010. The polyhedral model is more widely applicable than you think. In *International Conference on Compiler Construction*. Springer, 283–303.

[5] Uday Bondhugula. 2013. Compiling affine loop nests for distributed-memory parallel architectures. In *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.

[6] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 101–113.

[7] Uday Bondhugula, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2007. Automatic mapping of nested loops to FPGAs. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 101–111.

[8] Xiaoming Chen, Yinhe Han, and Yu Wang. 2020. Communication Lower Bound in Convolution Accelerators. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 529–541.

[9] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 367–379.

[10] Jason Cong and Jie Wang. 2018. PolySA: Polyhedral-based systolic array auto-compilation. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

[11] Jason Cong, Peng Wei, Cody Hao Yu, and Peipei Zhou. 2018. Latte: Locality aware transformation for high-level synthesis. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 125–128.

[12] Roshan Dathathri, Chandan Reddy, Thejas Ramashekar, and Uday Bondhugula. 2013. Generating efficient data movement code for heterogeneous architectures with distributed-memory. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE, 375–386.

[13] Johannes de Fine Licht, Grzegorz Kwasniewski, and Torsten Hoefler. 2020. Flexible Communication Avoiding Matrix Multiplication on FPGA with High-Level Synthesis. In *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 244–254.

[14] Tiziano De Matteis, Johannes de Fine Licht, and Torsten Hoefler. 2019. FBLAS: streaming linear algebra on FPGA. *arXiv preprint arXiv:1907.07929* (2019).

[15] Hasan Genc, Ameer Haj-Ali, Vighnesh Iyer, Alon Amid, Howard Mao, John Wright, Colin Schmidt, Jerry Zhao, Albert Ou, Max Banister, et al. 2019. Gemmini: An Agile Systolic Array Generator Enabling Systematic Evaluations of Deep-Learning Architectures. *arXiv preprint arXiv:1911.09925* (2019).

[16] Paolo Gorlani, Tobias Kenter, and Christian Plessl. 2019. OpenCL implementation of Cannon's matrix multiplication algorithm on Intel Stratix 10 FPGAs. In *2019 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 99–107.

[17] Anne-Claire Guillou, Fabien Quilleré, Patrice Quinton, S Rajopadhye, and Tanguy Risset. 2001. Hardware design methodology with the Alpha language. *FDL'01* (2001).

[18] Licheng Guo, Yuze Chi, Jie Wang, Jason Lau, Weikang Qiao, Ecenur Ustun, Zhiru Zhang, and Jason Cong. 2021. AutoBridge: Coupling Coarse-Grained Floorplanning and Pipelining for High-Frequency HLS Design on Multi-Die FPGAs. In *Proceedings of the 2021 ACM/SIGDA international symposium on Field-programmable gate arrays*.

[19] Licheng Guo, Jason Lau, Yuze Chi, Jie Wang, Cody Hao Yu, Zhe Chen, Zhiru Zhang, and Jason Cong. 2020. Analysis and Optimization of the Implicit Broadcasts in FPGA HLS to Improve Maximum Frequency. In *57th ACM/IEEE Design Automation Conference*.

[20] Intel. 2017. *Accelerating Genomics Research with OpenCL and FPGAs.* https://www.intel.com/content/www/us/en/healthcare-it/solutions/documents/genomics-research-with-opencl-and-fpgas-paper.html

[21] Ju-Wook Jang, Seonil B Choi, and Viktor K Prasanna. 2005. Energy-and time-efficient matrix multiplication on FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 13, 11 (2005), 1305–1319.

[22] Liancheng Jia, Liqiang Lu, Xuechao Wei, and Yun Liang. 2020. Generating Systolic Array Accelerators With Reusable Blocks. *IEEE Micro* 40, 4 (2020), 85–92.

[23] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 1–12.

[24] Sheng-Chun Kao, Geonhwa Jeong, and Tushar Krishna. 2020. ConfuciuX: Autonomous Hardware Resource Assignment for DNN Accelerators using Reinforcement Learning. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 622–636.

[25] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, et al. 2018. Spatial: A language and compiler for application accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 296–311.

[26] HT Kung, Bradley McDanel, and Sai Qian Zhang. 2019. Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 821–834.

[27] Sun Yuan Kung. 1988. VLSI array processors. *ph* (1988).

[28] Hyoukjun Kwon, Prasanth Chatarasi, Vivek Sarkar, Tushar Krishna, Michael Pellauer, and Angshuman Parashar. 2020. MAESTRO: A Data-Centric Approach to Understand Reuse, Performance, and Hardware Cost of DNN Mappings. *IEEE Micro* 40, 3 (2020), 20–29.

[29] Young kyu Choi, Yuze Chi, Jie Wang, Licheng Guo, and Jason Cong. 2020. When HLS Meets FPGA HBM: Benchmarking and Bandwidth Optimization. arXiv:2010.06075 [cs.AR]

[30] Yi-Hsiang Lai, Hongbo Rong, Size Zheng, Weihao Zhang, Xiuping Cui, Yunshan Jia, Jie Wang, Brendan Sullivan, Zhiru Zhang, Yun Liang, Jason Cong, Nithin George, Jose Alvarez, Christopher Hughes, and Pradeep Dubey. 2020. SuSy: A Programming Model for Productive Construction of High-Performance Systolic Arrays on FPGAs. In *2020 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.

[31] Dominique Lavenier, Patrice Quinton, and Sanjay Rajopadhye. 1999. Advanced systolic design. *Digital Signal Processing for Multimedia Systems* (1999), 657–692.

[32] Amy W Lim and Monica S Lam. 1997. Maximizing parallelism and minimizing synchronization with affine transforms. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 201–214.

[33] Junyi Liu, John Wickerson, Samuel Bayliss, and George A Constantinides. 2017. Polyhedral-based dynamic loop pipelining for high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 9 (2017), 1802–1815.

[34] Divya Mahajan, Jongse Park, Emmanuel Amaro, Hardik Sharma, Amir Yazdanbakhsh, Joon Kyung Kim, and Hadi Esmaeilzadeh. 2016. Tabla: A unified template-based framework for accelerating statistical machine learning. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 14–26.

[35] Manju Manjunathaiah, Graham M Megson, S Rajopadhye, and Tanguy Risset. 2001. Uniformization of affine dependence programs for parallel embedded system design. In *International Conference on Parallel Processing, 2001.* IEEE, 205–213.

[36] Dan I Moldovan. 1983. On the design of algorithms for VLSI systolic arrays. *Proc. IEEE* 71, 1 (1983), 113–120.

[37] Duncan JM Moss, Srivatsan Krishnan, Eriko Nurvitadhi, Piotr Ratuszniak, Chris Johnson, Jaewoong Sim, Asit Mishra, Debbie Marr, Suchit Subhaschandra, and Philip HW Leong. 2018. A customizable matrix multiplication framework for the Intel HARPv2 Xeon+ FPGA platform: A deep learning case study. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 107–116.

[38] Luigi Nardi, Artur Souza, David Koeplinger, and Kunle Olukotun. 2019. HyperMapper: a Practical Design Space Exploration Framework. In *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 425–426.

[39] Netlib. 2020. *LAPACK — Linear Algebra PACKage.* http://www.netlib.org/lapack/

[40] Tony Nowatzki, Newsha Ardalani, Karthikeyan Sankaralingam, and Jian Weng. 2018. Hybrid optimization/heuristic instruction scheduling for programmable accelerator codesign. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. 1–15.

[41] Louis-Noel Pouchet, Peng Zhang, Ponnuswamy Sadayappan, and Jason Cong. 2013. Polyhedral-based data reuse optimization for configurable computing. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*. 29–38.

[42] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2017. Plasticine: A reconfigurable architecture for parallel patterns. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 389–402.

[43] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices* 48, 6 (2013), 519–530.

[44] SV Rajopadhye. 1988. Systolic arrays for LU decomposition. In *1988., IEEE International Symposium on Circuits and Systems*. IEEE, 2513–2516.

[45] Sailesh K Rao and Thomas Kailath. 1988. Regular iterative algorithms and their implementation on processor arrays. *Proc. IEEE* 76, 3 (1988), 259–269.

[46] Robert Schreiber, Shail Aditya, Scott Mahlke, Vinod Kathail, B Ramakrishna Rau, Darren Cronquist, and Mukund Sivaraman. 2002. PICO-NPA: High-level synthesis of nonprogrammable hardware accelerators. *Journal of VLSI signal processing systems for signal, image and video technology* 31, 2 (2002), 127–142.

[47] Atefeh Sohrabizadeh, Cody Hao Yu, Min Gao, and Jason Cong. 2020. AutoDSE: Enabling Software Programmers Design Efficient FPGA Accelerators. arXiv:2009.14381 [cs.AR]

[48] Nitish Srivastava, Hongbo Rong, Prithayan Barua, Guanyu Feng, Huanqi Cao, Zhiru Zhang, David Albonesi, Vivek Sarkar, Wenguang Chen, Paul Petersen, et al. 2019. T2S-Tensor: Productively generating high-performance spatial hardware for dense tensor computations. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 181–189.

[49] Sven Verdoolaege. 2010. isl: An integer set library for the polyhedral model. In *International Congress on Mathematical Software*. Springer, 299–302.

[50] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, Jose Ignacio Gomez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 4 (2013), 1–23.

[51] Sven Verdoolaege, Serge Guelton, Tobias Grosser, and Albert Cohen. 2014. Schedule trees. In *International Workshop on Polyhedral Compilation Techniques, Date: 2014/01/20-2014/01/20, Location: Vienna, Austria.*

[52] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. 2017. Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs. In *Proceedings of the 54th Annual Design Automation Conference 2017*. 1–6.

[53] Jian Weng, Sihao Liu, Vidushi Dadu, Zhengrong Wang, Preyas Shah, and Tony Nowatzki. 2020. Dsagen: Synthesizing programmable spatial accelerators. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 268–281.

[54] WikiChip. 2020. *FSD Chip - Tesla.* https://en.wikichip.org/wiki/tesla_(car_company)/fsd_chip

[55] Xuan Yang, Mingyu Gao, Qiaoyi Liu, Jeff Setter, Jing Pu, Ankita Nayak, Steven Bell, Kaidi Cao, Heonjae Ha, Priyanka Raina, et al. 2020. Interstellar: Using Halide's Scheduling Language to Analyze DNN Accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 369–383.

[56] Tomofumi Yuki and Louis-Noël Pouchet. 2020. PolyBench/C 4.2. https://sourceforge.net/projects/polybench/