PolySA: Polyhedral-Based Systolic Array Auto-Compilation

Jason Cong, Jie Wang
University of California, Los Angeles
{cong,jiewang}@cs.ucla.edu

ABSTRACT

Automatic systolic array generation has long been an interesting topic due to the need to reduce the lengthy development cycles of manual designs. Existing automatic systolic array generation approach builds dependency graphs from algorithms, and iteratively maps computation nodes in the graph into processing elements (PEs) with time stamps that specify the sequences of nodes that operate within the PE. There are a number of previous works that implemented the idea and generated designs for ASICs. However, all of these works relied on human intervention and usually generated inferior designs compared to manual designs. In this work, we present our ongoing compilation framework named PolySA which leverages the power of the polyhedral model to achieve the end-to-end compilation for systolic array architecture on FPGAs. PolySA is the first fully automated compilation framework for generating high-performance systolic array architectures on the FPGA leveraging recent advances in high-level synthesis. We demonstrate PolySA on two key applications—matrix multiplication and convolutional neural network. PolySA is able to generate optimal designs within one hour with performance comparable to state-of-the-art manual designs.

ACM Reference Format:
Jason Cong, Jie Wang. 2018. PolySA: Polyhedral-Based Systolic Array Auto-Compilation. In *IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN (ICCAD '18), November 5–8, 2018, San Diego, CA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3240765.3240838

1 INTRODUCTION

With the advancement of CMOS technology, modern FPGAs are equipped with more and more resource. For example, the latest Xilinx FPGA deployed in Amazon AWS F1 instance [2] contains approximately 2.5 million logic elements and 6,800 DSPs. With such rich resource available, how to efficiently utilize them becomes an important challenge.

Systolic array architecture, which consists of a group of identical processing elements (PEs) that are locally connected to each other, turns out to be one of the promising solutions to overcome the challenge [14]. The architecture is highly scalable. With local connections and modular PEs, the design can be easily spread out to the entire chip with high frequency. Moreover, the architecture is highly energy-efficient. In contrast to conventional designs where PEs access data from on-chip global buffers through large crossbars, PEs in systolic array can access data from neighbor PEs as well. According to [23], fetching data from neighbor PEs costs 3x less energy than fetching from on-chip global buffers.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICCAD '18, November 5–8, 2018, San Diego, CA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5950-4/18/11…$15.00
https://doi.org/10.1145/3240765.3240838

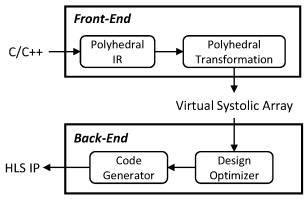


Figure 1: The PolySA compilation framework.

Systolic array is applicable to a wide range of applications, e.g., linear algebra [17], machine learning [7, 12], dynamic programming [10], etc. However, most of the above-mentioned works are designed manually from scratch, which usually take rather long development cycles. Research from Intel [20] shows that it takes several months of effort (3-19 months) to implement these designs. Such a problem has raised the needs of automating the design generation process of systolic array.

There have been some initial efforts toward automating the generation of systolic array (e.g., [6, 15, 22, 25, 26]). However, they face some limitations as follows: 1) Most works take manual inputs or outputs and require user guidance for generating the design. 2) The methodologies deployed in the compiler for mapping applications to systolic array are limited in either generability or performance. We aim to overcome these challenges in this work by presenting PolySA (as shown in Figure 1), which is the first end-to-end compilation framework for systolic array architecture on FPGAs, and it is able to generate designs with comparable performance to manual designs.

PolySA takes in applications written in high-level programming languages (C/C++), performs the mapping of systolic array based on polyhedral IR, and generates off-the-shelf FPGA IPs. PolySA addresses the above-mentioned issues with the following approaches: 1) PolySA takes in high-level programming languages and generates FPGA IPs. The whole compilation process is fully automated without any user intervention. 2) PolySA leverages the polyhedral framework to support various schedulings and efficient transformations to generate systolic array architecture. Overall, PolySA makes following contributions:

- We build an end-to-end compilation framework for generating a systolic array architecture on FPGAs. The compilation framework is fully automated to generate efficient systolic array designs for FPGAs from applications written in C/C++.
- In the front end, we implement and extend a systematic transformation methodology based on the polyhedral framework to map designs to systolic array architecture on FPGAs. With the help of such an approach, PolySA is able to identify all design alternatives that cover all previous manual designs.

- In the back end, we leverage the high-level synthesis (HLS) tools for fast design generation. We propose an analytical model to analyze the performance and resource utilization of designs that allows for a comprehensive design space exploration for the optimal designs.
- We demonstrate PolySA on two key applications—matrix multiplication and convolutional neural network. PolySA is able to reduce the development cycles from several months to within one hour, and generates high-performance designs with a performance gap within 31% compared to state-of-the-art manual designs.

The remainder of this paper is organized as follows. Section 2 introduces the background of the polyhedral framework and summarizes the related work. Section 3 presents the overview of the PolySA framework. In Section 4 we walk through the compilation flow with the example of matrix multiplication. Section 5 presents experimental results. Finally, this paper is concluded in Section 6.

2 BACKGROUND AND RELATED WORK

2.1 Polyhedral Framework

The polyhedral compilation framework performs complex loop nest restructuring for performance optimization [5]. The essence of the polyhedral model is the use of parametric polyhedra as the internal representation of programs. There are three key components in the polyhedral model: iteration domain, scattering functions, and access functions. *Iteration domain* is a set of all possible values of the iteration vectors in the loop nest which defines the shape of the polyhedron of the given problem. *Scattering functions* provide the ordering information inside the iteration domain which describes the order of loop instances to be executed relative to each other. And *access functions* describe all the memory accesses of loop instances. Each scattering function defines a unique scheduling of the given program. We use these two terms (scattering function and scheduling) interchangeably in the remaining sections of the paper. The polyhedral compilation framework performs program transformation by selecting different scattering functions (schedulings) without breaking the original data dependency of the program. More details about the polyhedral framework can be found in [3–5].

PolySA embraces the polyhedral model as its internal IR for two major reasons: 1) The great analysis power of polyhedral IR enables efficient architecture transformation and design space exploration. 2) Instead of using other user-defined IRs, as in many previous works, standard IR supports easy integration with other frameworks (e.g., Tensor Comprehension [24]) and enables PolySA to leverage the enormous legacy work from the polyhedral community.

2.2 Automatic Systolic Array Generation

Systolic array can be applied to a wide variety of applications [7, 10, 12, 17]. Recent success includes the Tensor Processing Unit by Google [12] which implements a 2D systolic array to perform matrix operations. Intel implements a 2D systolic array for sequence alignment in the genomics pipeline [10].

Automatic systolic array compilation has long been an interesting topic due to the enormous development efforts of manual designs [6, 14–16, 19, 22, 25, 26]. Wei et al. [25] proposed a compilation framework to generate 2D systolic arrays for CNN. However, the mapping approach is limited in terms of generality as the array architecture is fixed, and the simple enumeration approach adopted in the work fails to discover all design alternatives.

A more general approach named canonical mapping was proposed to map algorithms to systolic arrays by performing affine transformation [13, 14]. The approach searches for different processor allocation and time scheduling functions to find the feasible systolic array designs. Rajopadhye and Fujimoto [19] further extended the approach to generate systolic arrays with non-uniform data flow. There have been many previous works that implemented the canonical mapping [6, 15, 22, 26]. However, all of these approaches are half-automated and hard to use in practice as they rely on manual inputs or outputs.

Lim and Lam [1, 16] proposed an elegant and general approach for identifying both synchronization-free parallelism and pipelined parallelism on parallel multiprocessors. The proposed approach aimed to extract the maximal parallelism to transform the input code to the output code with fully permuted loops at the outermost level, so that it can be mapped to pipelined architectures with maximal freedom. The pipelined architecture can be further transformed to systolic array architectures with the constraint of non-zero delays between PEs. The work in [1] did not explore different processor and time assignments which will lead to on-chip designs with different area and performance.

PolySA adopts the canonical mapping and further extends it to generate high-performance designs. Details of the canonical mapping will be covered in Section 4.

3 FRAMEWORK OVERVIEW

Figure 1 presents the full compilation flow of PolySA. The compiler is composed of two parts—front end and back end. In the front end, PolySA takes in applications written in C/C++ and compiles them into polyhedral IR. Then, the compiler selects different schedulings to map algorithms to the systolic array architecture based on canonical mapping in the stage of Polyhedral Transformation. The outputs of the front end, which are named Virtual Systolic Array (VSA), are different systolic array design alternatives described by polyhedral IR. The VSA serves as a standard interface between the front-end and back-end and helps to improve the portability of the framework.

In the back end, different VSAs will be evaluated by the design optimizer for certain performance metrics. Optimal designs are picked and later fed into the code generator to generate synthesizable code on FPGA. We adopt the HLS rather than RTL considering both portability and productivity. The current version of PolySA can only support Xilinx flow by generating code written in Xilinx HLS C. Support for Intel OpenCL will be added in the future. In the following section, we will use matrix multiplication as the example to walk through the compilation process of PolySA in detail.

4 COMPILATION FLOW

4.1 Polyhedral Transformation

In the front end, PolySA takes algorithms written in C/C++ as inputs. Figure 2 shows the example code of matrix multiplication. Pragmas `#pragma sa` and `#pragma endsa` are added to mark the code region that needs to be compiled to systolic array.

```
#pragma sa
for (int i = 0; i < I; i++)
 for (int j = 0; j < J; j++){
   C[i][j] = 0;
   for (int k = 0; k < K; k++)
     C[i][j] += A[i][k] * B[k][j];
 }
#pragma endsa
```

Figure 2: Input code of matrix multiplication.

The code is compiled to polyhedral IR, which includes three key components: iteration domain $\vec{D}$, scattering functions F_S, and access functions F_A. Let us denote the original algorithm that is represented by the polyhedral model as Φ. Φ is a triple of three key components, i.e., $\Phi = (\vec{D}, F_S, F_A)$.

Figure 3a presents these three components. Both scattering and access functions are presented in the matrix format.

- The iteration domain $\vec{D}$ of matrix multiplication is a three-dimensional polyhedron (i.e., a cube). Each node in the cube corresponds to one loop instance in the original code, which can be located using an iteration vector $\vec{n}$ that is a tuple of the three iterators in MM, i.e., $\vec{n}^T = (i, j, k)$. In the example of MM, each node performs one multiply-and-accumulate (MAC) operation (`C[i][j] += A[i][k] * B[k][j]`).
- The scattering function $F_S(\vec{n})$ generates logical stamps S for each node which describe the execution order of nodes, i.e., $S = F_S(\vec{n}) \times \vec{n}$. The logical stamps generated by the default scheduling are marked in the cube.
- The access functions $F_A(\vec{n})$ define the array reference Ref to be accessed by each node, i.e., $Ref = F_A(\vec{n}) \times \vec{n}$.

Given one scattering function, we can directly map the polyhedron to an array processor using the following approach. We assign different semantics to different dimensions of the logical stamps generated by the scheduling. For example, for the polyhedron in Figure 3a, the first two dimensions of the logical stamps are devoted to space mapping that assigns the node to different PEs in the array. And the last dimension is devoted to time scheduling that assigns the execution order of nodes inside the PE. This will yield a simple 2D array which is able to finish the computation by two cycles. Details are depicted in Figure 3b.

Different scattering functions will generate different array architectures. In theory, PolySA can support any feasible scattering functions for systolic array generation. In this work, we use an iterative mapping approach to generate scattering functions that corresponds to the canonical mapping [13]. The space mapping and timing scheduling in the canonical mapping are represented by the projection vector $\vec{d}$ and scheduling vector $\vec{s}$. Nodes of the original polyhedron along the $\vec{d}$ are assigned to the same PE. All the nodes on the same hyperplane, which is orthogonal to the scheduling vector $\vec{s}$, are scheduled to execute at the same time. Such a relationship can be summarized as follows:

Space mapping: The mapping of a node $\vec{n}$ in the N-dimensional polyhedron onto a PE $\vec{c}$ in the $(N-1)$-dimensional mapped array is done by:

$$\vec{c} = P^T \vec{n} \tag{1}$$

where the *processor basis* P is an $N \times (N-1)$ matrix consisting of basis vectors in the mapped array that can be derived from the projection vector $\vec{d}$. In our MM example in Figure 3b, we choose $\vec{d}^T = (0, 0, 1)$. Therefore, we can pick up two vectors $\vec{p_1}^T = (1, 0, 0)$ and $\vec{p_2}^T = (0, 1, 0)$ which will serve as new basis vectors in the mapped array and compose the processor basis P as:

$$P^T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \tag{2}$$

For example, node $\vec{n}^T = (0, 1, 0)$ will be mapped to the PE $\vec{c}$ as:

$$\vec{c} = P^T \vec{n} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{3}$$

Time scheduling: For any node $\vec{n}$ in the polyhedron, its time stamp to compute is calculated by:

$$\vec{t} = \vec{s}^T \vec{n} \tag{4}$$

In our example, we choose $\vec{s}^T = (0, 0, 1)$. For instance, the time stamp for node $\vec{n}^T = (0, 0, 1)$ to execute is $\vec{s}^T \vec{n} = 1$. In the end, the scattering function is derived by combining space mapping and time scheduling together.

$$F_S(\vec{d}, \vec{s}) = \begin{bmatrix} P^T \\ \vec{s}^T \end{bmatrix} \tag{5}$$

And using the given scattering function, we calculate the logical stamps for the new polyhedron as:

$$S = \begin{bmatrix} \vec{c} \\ \vec{t} \end{bmatrix} = F_S(\vec{n}) \times \vec{n} = \begin{bmatrix} P^T \\ \vec{s}^T \end{bmatrix} \begin{bmatrix} \vec{n} \end{bmatrix} \tag{6}$$

where $\vec{c}$ is an $(N-1)$-dimensional vector which describes the PEs that nodes in the original polyhedron are mapped to, and t is a 1-dimensional vector (scalar) which describes the execution order of nodes that are mapped to PEs. For illustration, in Figure 3c, we present another scheduling which uses a different set of projection and scheduling vectors.

With the help of the canonical mapping, the scattering function is chosen based on the projection and scheduling vectors $\vec{d}$ and $\vec{s}$. And each scattering function will generate a different array architecture. PolySA enumerates different combinations of $(\vec{d}, \vec{s})$ to explore different design alternatives. Note that not all $(\vec{d}, \vec{s})$ are feasible choices, as the generated design must follow the original data dependency. Besides, systolic array designs require at least one unit of delay associated with each edge. Therefore, for any dependency arc $\vec{e_d}$ in the graph, the scheduling vector $\vec{s}$ needs to satisfy:

$$\vec{s}^T \vec{e_d} > 0 \tag{7}$$

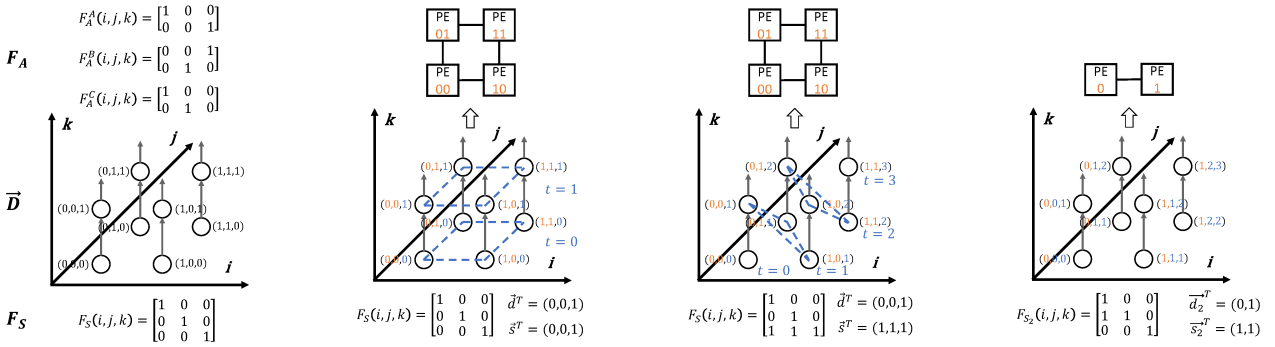
Meanwhile, all the nodes mapped to the same PE cannot operate at the same time. This is constrained by:

$$\vec{d}^T \vec{s} > 0 \tag{8}$$

PolySA applies feasible checks along with the mapping to filter out illegal array designs. More details of the feasibility check can be found in [13, 19]. The current PolySA chooses $\vec{d}$ from the candidate set $\{\vec{can} | \vec{can} = \Sigma \vec{e_i}\}$ where $\vec{e_i}$ is an N-dimensional unit vector whose i-th dimension is 1 and the rest are all zeros. And we choose $\vec{s}$ with coefficients from the set $\{0, 1\}$.

The above-mentioned scheduling maps an N-dimensional polyhedron to an $(N-1)$-dimensional array. Let us denote the polyhedron Φ which represents an N-dimensional array as Φ^N. We derive a new polyhedron Φ^{N-1} which represents an $(N-1)$-dimensional array using the scattering function F_{S_1} as shown in Figures 3b and 3c. The mapping flow is summarized as:

$$\Phi^N \xrightarrow{F_{S_1}(\vec{d_1}, \vec{s_1})} \Phi^{N-1} \tag{9}$$



(a) Polyhedral IR (b) Scheduling example 1 (c) Scheduling example 2 (d) Example of multi-projection

Figure 3: Polyhedral transformation of MM example ($I = J = K = 2$).

In principle, this method can be applied K times and thus reduce the dimension of the array to $N - K$. For example, when mapping designs from Φ^{N-1} to Φ^{N-2}, we can choose the scheduling F_{S_2} by:

$$F_{S_2} = \begin{bmatrix} P_2^T & 0 \\ \vec{s_2}^T & 0 \\ 0 & 1 \end{bmatrix} \tag{10}$$

Thus we can calculate the logical stamps for the polyhedron Φ^{N-2} as:

$$S_2 = \begin{bmatrix} \vec{c_2} \\ \vec{t_2} \end{bmatrix} = F_{S_2} \times \begin{bmatrix} \vec{c_1} \\ \vec{t_1} \end{bmatrix} \tag{11}$$

where $\vec{c_2}$ is an $(N-2)$-dimensional vector for space mapping and $\vec{t_2}$ is a 2-dimensional vector for time scheduling in the newly mapped $(N-2)$-dimensional array represented by Φ^{N-2}. The principle of this approach is straightforward. We use the formula 5 to project the $(N-1)$-dimensional array in Φ^{N-1} to an $(N-2)$-dimensional array and generate the new timestamp. Meanwhile, all the old timestamps generated in the previous mappings are inherited and left untouched. Note that $\vec{d_2}$ and $\vec{s_2}$ are $(N-1)$-dimensional vectors.

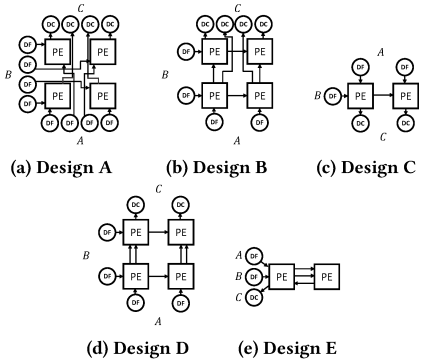
We can apply the multi-projection iteratively to reduce the dimension of the generated array as:

$$\Phi^N \xrightarrow{F_{S_1}(\vec{d}_1, \vec{s}_1)} \Phi^{N-1} \xrightarrow{F_{S_2}(\vec{d}_2, \vec{s}_2)} \Phi^{N-2} \xrightarrow{\cdots} \tag{12}$$

Starting from an N-dimensional polyhedron, the semantics of the generated logical stamps for nodes in the polyhedron change accordingly, along with the multi-projection. For Φ^{N-K}, the leading $N-K$ dimensions of the logical stamps are devoted to space mapping, and the rest of the K dimensions are devoted to time scheduling. Figure 3d presents an example where we apply another new scheduling F_{S_2} based on the scheduling F_{S_1} in Figure 3c to generate a 1D array for matrix multiplication. In the mapped array, the first dimension of the logical stamp assigns the PE. And the last two dimensions of the logical stamp assign the execution order of nodes inside PE. For instance, there are four nodes mapped to PE0. Inside PE0, we follow the lexicographic order $(0,0,0) \rightarrow (0,0,1) \rightarrow (0,1,1) \rightarrow (0,1,2)$ to execute the four nodes based on the last two dimensions of their logical stamps.

During the polyhedral transformation, every time when choosing a new scattering function, we will need to update the access functions in the newly generated polyhedron. New access functions in Φ^{N-K} are calculated by:

$$F_{A_k} = F_{A_{k-1}} \times F_{S_k}^{-1} \tag{13}$$



(a) Design A (b) Design B (c) Design C

(d) Design D (e) Design E

Figure 4: Different array designs of MM example.

$F_{S_k}^{-1}$ is the inverse matrix of F_{S_k} in the matrix format. If there is no inverse matrix existing for the given F_{S_k}, PolySA will terminate the multi-projection and start again from a different F_{S_k} instead.

With the help of access functions, PolySA analyzes the data transfer scheme for the mapped array and adds the data transfer modules and interconnects to the mapped design. Figure 4a, 4b, and 4c present the complete array architectures with data transfer modules and interconnects for designs derived from Figure 3b, 3c, and 3d, respectively. For the ease of illustration, we will name these three designs in Figure 4 as design A, B, and C, respectively. In these designs, we add data transfer modules including data feeders (DF) and data collectors (DC). DF fetches data from the on-chip global buffers and feeds PEs which it connects to. DC collects final results from PEs and writes them back to the on-chip global buffers.

In design A, as all four PEs consume different data at the same time according to its scheduling, we allocate four DFs for both matrix A and B. Moreover, all four PEs generate final results at the same time, therefore, we allocate four DCs for matrix C as well. As for design B, based on the access functions, we find that data from matrix A are reused between PEs vertically, and data from matrix B are reused between PEs horizontally. Therefore, only 2 DFs are allocated for each matrix, and data will be passed through PEs by local interconnects. Similarly, we derive the design C.

In systolic array designs, it is more favorable to feed data through boundary PEs, as the global wires introduced by the interior I/O usually lead to long delays that cause inferior performance. The three designs generated so far all contain interior I/O that introduces

Table 1: Basic design modules in Code Generator.

Module Name	Functionality
op_trans	transfers operands between neighbor PEs
int_store	stores local intermediate results
int_trans_in	fetches intermediate results from neighbor PEs
int_trans_out	writes out intermediate results to neighbor PEs
res_trans	transfer final results between neighbor PEs
compute	performs the computation
DF	fetches data from global on-chip buffers and feeds PEs
DC	collects data from PEs and writes back to global on-chip buffers

global interconnects with long delays. Manual designs [11, 17] implement double buffers inside PEs to perform the data transfer through local interconnects. Such designs are shown in Figure 4d and 4e. In PolySA, we perform the interior I/O elimination to reduce the global interconnects. We follow the similar approach as used in the manual designs by adding double buffers inside PEs to transfer data through neighbor PEs without impacting the design latency. In our examples as shown in Figure 4, the design B and C can be further optimized to the design D and E, respectively. The design A fails to be optimized without the cost of increasing latency. Such designs will be filtered out from the design space.

The front end of PolySA usually generates numerous systolic array design candidates. These designs are named as *Virtual Systolic Array*, which are described by Polyhedral IR with metadata. The metadata include the information of data transfer modules and interconnects as mentioned above. VSAs contain the complete information about the functionalities and the architecture of the generated systolic array design. The back end of the PolySA will pick up the optimal design among them by different performance metrics in the design optimizer and generate the synthesizable code for FPGA using the code generator. This will be presented in the next two sections.

4.2 Code Generator

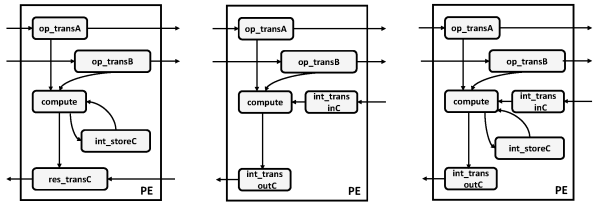
The code generator in PolySA generates designs written in high-level synthesis (HLS) C [8] based on its high productivity and portability compared to RTL. As different coding styles in HLS play an important role in the design performance, the code directly generated from the polyhedral tools is usually hard to generate high-performance designs. Therefore, in PolySA we adopt a template-based code generation approach, similar as the work [25].

To begin with, we define the basic design module templates that are used in the systolic array designs. Table 1 shows the complete list of these modules and explains their functionalities in detail.

These basic design modules serve as building blocks to compose different PE modules. Depending on the way that intermediate results are processed in the systolic array, in the code generator, we classify systolic arrays into three categories and implement three different PE templates:

- *Local designs.* Designs in which intermediate data of results are all locally accumulated inside PEs.
- *Global designs.* Designs in which intermediate data of results are all globally accumulated between PEs.
- *Mixed designs.* Designs in which intermediate data of results are both locally and globally accumulated among PEs.

Figure 5 depicts the architecture of PE templates of these three different designs. Above the level of PEs, the code generator implements an array template which consists of PEs and data transfer



(a) Local designs (b) Global designs (c) Mixed designs

Figure 5: PE templates for different designs.

modules, and connects them based on the array shape and data interconnection, which can be inferred from VSAs, as shown in Figure 4.

All the basic design modules are written in HLS. The PE and array templates are written in RTL to provide full flexibility. All the templates are parametric. PolySA parses VSAs and instantiates these templates to generate the final systolic array design. The entire design is eventually wrapped up as an HLS IP with AXI interface to enable easy integration with other designs.

4.3 Design Optimizer

The design optimizer in PolySA evaluates all systolic array candidates generated from the front end and picks up the optimal design based on certain performance metrics. In the current design optimizer, we will pick up the design with the lowest latency given the limited resource constraints. The optimization problem is defined as:

$$\begin{aligned} \underset{VSA}{\text{minimize}} \quad & L = max(L_{comp}, L_{comm}) \\ \text{subject to} \quad & resource_i(VSA) \leq resource_i^{on\_chip} \\ & \text{where } i = FF, LUT, BRAM, DSP \end{aligned} \tag{14}$$

All data transfer modules in the templates are double-buffered. Therefore, the systolic array can immediately begin to prepare the data for the next job at the beginning of the computaion of the current job. The latency of each job L is therefore dominated by the bottleneck between computation latency L_{comp} and data transfer latency L_{comm}, as calculated by $L = max(L_{comp}, L_{comm})$.

The computation latency L_{comp} can be estimated based on scattering functions. We count the number of computation nodes mapped to each PE multiplied by the computation latency of each node. The communication latency L_{comm} is estimated based on the amount of data transferred between systolic array and on-chip global buffers.

As for the resource models, the template-based code generation approach enables highly efficient and accurate resource modeling. We designed a suite of micro-benchmarks which help build up analytical models for the parametric templates. For each VSA generated by the front end of PolySA, we extract the design parameters and plug them into the parametric resource models to quickly estimate the resource consumption of each design.

Overall, our latency and resource models achieve the relative error within 10%. Details are presented in Section 5. Note that for problems with a large size, PolySA will generate a large systolic array which could be impossible to fit on-chip given the limited FPGA resource. In such case, PolySA applies array partitioning to partition the original array into smaller-size arrays and execute the tiled tasks in sequence on the newly generated array. The tiling parameters will be explored by the design optimizer as well.

Table 2: Problem configuration of CNN.

Denotation	Explanation	Configuration
IN_NUM, OUT_NUM	input/output feature map number	2, 4
IN_IMG_H, IN_IMG_W	input feature map height/width	5, 5
OUT_IMG_H, OUT_IMG_W	output feature map height/width	3, 3
P, Q	weight kernel height/width	3, 3

In PolySA, we solve the optimization problem as stated in Equation 14 by enumeration with pruning based on the resource usage. This approach suffices for our project given the complexity of the design space. In Section 5 we present the runtime breakdown of PolySA in detail.

5 EXPERIMENTAL RESULTS

5.1 Experiment Setup

In this section we evaluate the PolySA framework in detail. The current PolySA uses the Clan compiler [4] to parse C/C++ code into polyhedral IR. The front end and back end are written in Matlab. The compiler runs on our server which is equipped with Intel Xeon E7-4807 CPU and 128 GB memory. All of the designs are synthesized and implemented using Xilinx Vivado 2017.4.

5.2 Front-End Evaluation

We use two key applications—matrix multiplication (MM) and convolutional neural network (CNN)—to assess the framework. For CNN, we take the 6-level nested loop for one single convolutional layer as the input of PolySA. The sample code can be found in [25]. Note that both algorithms are fully permuted nested loops and therefore can be transformed to systolic array architecture according to [1].

In this section we use two simple examples to evaluate the front-end of PolySA. For MM, we set $I = J = K = 2$. The problem configuration of CNN is shown in Table 2. The front end of PolySA generates different design alternatives for the given algorithm. Figures 6 and 7 present all possible systolic array design alternatives that PolySA generates for these simple MM and CNN examples. The array dimension is marked alongside the array. The legends for data interconnects are shown in Figure 6f and 7p for MM and CNN, respectively. For example, for the design *MM 1* in Figure 6a, the array shape is $I \times K$. Data from matrix A and B are fed downward to PEs. Intermediate results of matrix C are accumulated across PEs rightward. And final results of matrix C are collected from the last column of PEs in the array.

The major observation here is that with the help of the polyhedral transformation, PolySA performs a systematic array mapping process and is able to discover all possible design candidates for the given algorithm, which is a superset of different systolic array architectures implemented by previous manual works. In these two examples, PolySA identifies 5 different systolic array designs for MM and 15 different systolic array designs for CNN. Designs that are implemented by manual works are noted with an asterisk mark in the figure.

Apart from designs covered by previous work, PolySA identifies 1 and 13 more new designs for MM and CNN, respectively. Previous manual designs usually only identify one or several design points as *a tip of the iceberg*. This could lead to designs with suboptimal performance. In contrast, PolySA provides a systematic and efficient approach to identify the complete design space and therefore allows for a comprehensive search for the optimal designs.

Table 3: Design comparison.

(a) MM

MM	BRAM	DSP	FF	LUT	MHz	GFLOPs	Projected GFLOPs
PolySA	89%	89%	39%	49%	228.8	555.4	758.5
Baseline [17]	-	-	-	-	312.5	800.0	

(b) CNN

CNN	BRAM	DSP	FF	LUT	MHz	GFLOPs	Projected GFLOPs
PolySA	71%	89%	39%	49%	229.5	548.39	603.55
Baseline [25]	47%	81%	40%	59%	252.6	600.27	

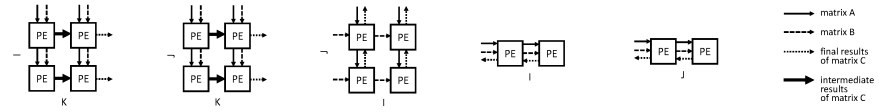
5.3 Back-End Evaluation

In the back end, PolySA uses the design optimizer to perform design space exploration and picks up the optimal design with the lowest latency. The design is generated by the code generator and wrapped as an HLS IP. We compare the generated designs from PolySA to state-of-the-art designs for both applications. Baselines of MM and CNN are chosen from [17] and [25], respectively. Both baselines are implemented on Intel Arria 10 GX1150, which is not supported by PolySA so far. We choose the Xilinx UltraScale+ VU9P as the target platform for our designs, which contains similar amounts of resource compared to Arria 10 GX1150. For MM, we set the problem size as $I = J = K = 1024$. For CNN, we choose the third layer of the VGG-16 model [21].

5.3.1 Design Space Exploration. Figure 8 depicts the design space of MM and CNN. The optimal design is marked in red color. The height of each point equals the reciprocal of the latency of the design. Therefore, the lower the latency is, the higher the node locates in the figure. And the x-y coordinates correspond to the BRAM and DSP utilization of the design. For the ease of illustration, for designs with the same usage of BRAM and DSP, we pick up the design with the lowest latency as the representative node to draw in the figure. Overall, there are approximately 113K and 17K valid design points in MM and CNN, respectively. The design optimizer searches the optimal design with the lowest latency, given the resource constraints.

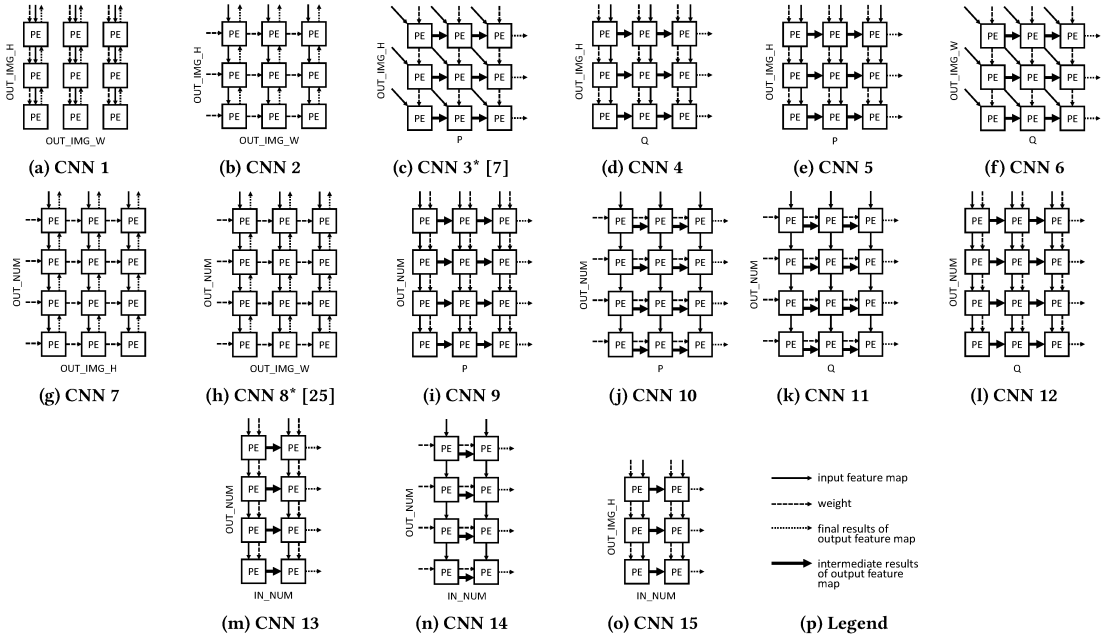
5.3.2 Resource and Latency model. For both applications, we pick up the top-three designs with the lowest latency and synthesize them on FPGAs. We compare the resource usage of these designs to estimation results from our resource models. Additionally, we collect latency results from hardware emulation and compare them to estimation results from our latency models. All the area numbers here are post-synthesis results. Detailed comparison results are shown in Figure 9. Overall, the latency and resource models are highly accurate, with a relative error within 10% for all designs.

5.3.3 Design Comparison. For MM the optimal design from PolySA implements the shape of *MM3* in Figure 6c. The generated design contains 19 rows and 8 columns for single-precision floating point. As for CNN, the optimal design picks the shape of design *CNN 8* in Figure 7h, and contains 8 rows and 19 columns for single-precision floating point. In addition, both designs implement SIMD for MAC units inside PEs as similar as baselines. The current design sets the SIMD factor as 8.



(a) MM 1* [13] (b) MM 2* [13] (c) MM 3* [17] (d) MM 4 (e) MM 5* [11] (f) Legend

Figure 6: Different systolic array designs of MM example.

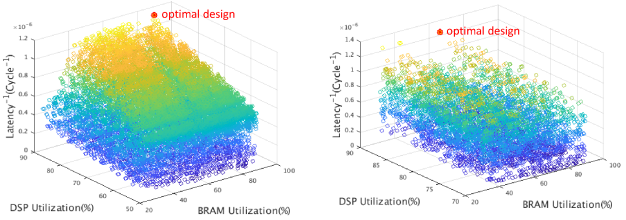


(a) CNN 1 (b) CNN 2 (c) CNN 3* [7] (d) CNN 4 (e) CNN 5 (f) CNN 6

(g) CNN 7 (h) CNN 8* [25] (i) CNN 9 (j) CNN 10 (k) CNN 11 (l) CNN 12

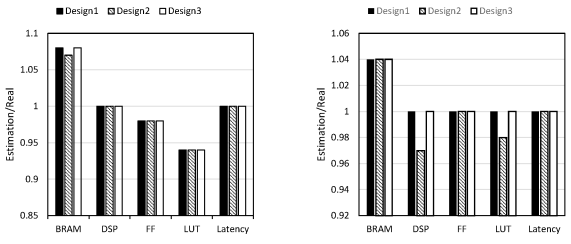
(m) CNN 13 (n) CNN 14 (o) CNN 15 (p) Legend

Figure 7: Different systolic array designs of CNN example.



(a) MM (b) CNN

Figure 8: Design space of MM and CNN.



(a) MM (b) CNN

Figure 9: Resource and latency model evaluation.

Table 3 presents the detailed comparison results. All the area and timing statistics here are post-implementation results. The throughput is calculated by dividing the computation complexity of the given algorithm by the latency measured from XSim. Compared to manual designs, designs generated by PolySA have a performance gap of 31% and 9% for MM and CNN, respectively. This gap is mainly due to the design frequency gap. This could be caused by: 1) Code implementation. Both baselines use well-tuned code for targeted applications. The MM baseline uses the well-optimized SystemVerilog code and the CNN baseline uses specialized OpenCL templates. PolySA employs a general code template with more design overheads which could lead to the frequency degradation. 2) Platform. The current designs are implemented on a Xilinx platform, whereas the baselines are implemented on an Intel platform. More evaluations on Intel platforms are needed to understand the impacts of FPGA architecture and technology on the design performance. In the future, we will adopt frequency optimization approaches like [9] to mitigate the design overheads and evaluate our designs on the Intel platform. Finally, for comparison, we calculate the projected throughput of our designs with the same frequency as baselines, as shown in Table 3. The performance difference between the baselines and our designs is reduced to around 5% for both applications.

5.4 Runtime Analysis

Table 4 presents the runtime breakdown of the PolySA compilation flow for generating the optimal designs in Section 5.3. The total time is measured starting from the C/C++ inputs to the completion of the generation of HLS IPs. As can be seen from Table 4, CNN

Table 4: Runtime breakdown for design examples.

	Polyhedral Transformation	Design Optimizer	Code Generator	Total Time
MM	2.5s	18min	8min	26min
CNN	23min	5min	8min	37min

takes a much longer time than MM during the stage of polyhedral transformation. The reason is that CNN starts from a 6-dimensional polyhedron and takes four times of multi-projection to generate a 2D systolic array—whereas MM starts from a 3-dimensional polyhedron, and can be mapped to a 2D systolic array by one single projection. The design space of scattering functions for CNN grows exponentially and takes a much longer time to finish compared to MM. In the future, we will explore approaches like [6, 18] to help improve the efficiency of the polyhedral transformation. Meanwhile, in our examples, CNN takes a shorter time to finish than MM in the design optimizer. Although PolySA generates more systolic array candidates from the front end for CNN compared to MM, as presented in Section 5.2, most of CNN designs are limited in array size which is bounded by small factors such as P and Q. These designs usually lead to suboptimal performance and will be quickly pruned during the design space exploration.

Overall, PolySA helps to significantly reduce the development cycles of systolic array designs. Compared to manual designs which normally take months of effort to finish [20], PolySA is able to finish the generation of systolic array designs within one hour—which brings several orders of magnitude speed-up in terms of development cycles.

6 CONCLUSION

In this paper we present PolySA, an end-to-end compilation framework for generating systolic array architecture. PolySA performs polyhedral-based transformation to map algorithms to systolic array architecture. We leverage the power of canonical mapping to enable efficient selection of scattering functions through the mapping process. PolySA is able to pick up the optimal design from the design space and generate off-the-shelf design IPs that can be easily integrated with other designs. This is the first fully automated framework for generating the systolic array architecture on FPGAs.

We demonstrate PolySA on two important applications—matrix multiplication and convolutional neural network. PolySA identifies all the different design alternatives which cover all state-of-the-art manual designs. It is able to generate systolic array designs with comparable performance to well-tuned manual designs. The entire compilation flow finishes within an hour, which helps save the development efforts by several orders of magnitude.

In the end, we believe PolySA is a significant advance toward improving the programmability of FPGAs by providing a highly efficient solution to generating high-performance systolic array designs that can be applied to a wide range of applications. PolySA is an ongoing effort and we are making improvements in multiple fronts, including better scattering functions, more general back end support such as for Intel FPGA platforms, and demonstrating the framework on more applications.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable comments and helpful suggestions. The authors also thank Monica S. Lam and Louis-Noël Pouchet for helpful discussions on the early systolic array synthesis and the polyhedral theory. This work is partially supported by the CAPA Program jointly funded by the NSF (CCF-1436827) and Intel, and the ICN-WEN Award jointly funded by NSF (CNS-1719403) and Intel.

REFERENCES

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
[2] Amazon. 2018. Amazon EC2 F1 Instances. (2018). https://aws.amazon.com/ec2/instance-types/f1
[3] Cédric Bastoul. 2011. *OpenScop: A Specification and a Library for Data Exchange in Polyhedral Compilation Tools.* Technical Report. Paris-Sud University, France.
[4] Cédric Bastoul, Albert Cohen, Sylvain Girbal, and et al. 2003. Putting Polyhedral Loop Transformations to Work. In *LCPC*. College Station, Texas, 209–225.
[5] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, and et al. 2010. The Polyhedral Model is More Widely Applicable Than You Think. In *ETAPS (CC'10/ETAPS'10)*. Springer-Verlag, Berlin, Heidelberg, 283–303.
[6] Uday Bondhugula, J. Ramanujam, and P. Sadayappan. 2007. Automatic Mapping of Nested Loops to FPGAS. In *PPoPP*. ACM, New York, NY, USA, 101–111.
[7] Y. H. Chen, J. Emer, and V. Sze. 2016. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *ISCA*. 367–379.
[8] J. Cong, Bin Liu, S. Neuendorffer, and et al. 2011. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *Trans. Comp.-Aided Des. Integ. Cir. Sys.* 30, 4 (April 2011), 473–491.
[9] Jason Cong, Peng Wei, Cody Hao Yu, and Peipei Zhou. 2018. Latte: Locality Aware Transformation for High-Level Synthesis. In *FCCM*.
[10] Intel. 2017. Accelerating Genomics Research with OpenCL™ and FPGAs. (2017). https://www.intel.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-accelerating-genomics-opencl-fpgas.pdf
[11] Ju-Wook Jang, S. Choi, and V. Prasanna. [n. d.]. Energy- and time-efficient matrix multiplication on FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 13, 11 ([n. d.]), 1305–1319.
[12] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, and et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *ISCA*. 12.
[13] S. Y. Kung. 1987. *VLSI Array Processors.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA.
[14] Dominique Lavenier, Patrice Quinton, and Sanjay Rajopadhye. 1999. Advanced Systolic Design. (1999).
[15] Hervé Le Verge, Christophe Mauras, and Patrice Quinton. 1991. The ALPHA language and its use for the design of systolic arrays. *Journal of VLSI signal processing systems for signal, image and video technology* 3, 3 (01 Sep 1991), 173–182.
[16] Amy W. Lim and Monica S. Lam. 1997. Maximizing Parallelism and Minimizing Synchronization with Affine Transforms. In *POPL*. ACM, New York, NY, USA, 201–214.
[17] Duncan Moss, Srivatsan Krishnan, Eriko Nurvitadhi, and et al. 2018. A Customizable Matrix Multiplication Framework for the Intel HARPv2 Platform - A Deep Learning Case Study. In *FPGA*.
[18] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. 2008. Iterative Optimization in the Polyhedral Model: Part Ii, Multidimensional Time. In *PLDI*. ACM, New York, NY, USA, 90–100.
[19] Sanjay V Rajopadhye and Richard M Fujimoto. 1990. Synthesizing systolic arrays from recurrence equations. *Parallel Comput.* 14, 2 (1990), 163 – 189.
[20] Hongbo Rong. 2017. Programmatic Control of a Compiler for Generating High-performance Spatial Hardware. *CoRR* abs/1711.07606 (2017). arXiv:1711.07606
[21] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. *CoRR* abs/1409.1556 (2014). arXiv:1409.1556
[22] Andrew Stone and Elias S. Manolakos. 2000. DG2VHDL: A Tool to Facilitate the High Level Synthesis of Parallel Processing Array Architectures. *Journal of VLSI signal processing systems for signal, image and video technology* (01 Feb 2000).
[23] V. Sze, Y. H. Chen, T. J. Yang, and J. S. Emer. 2017. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proc. IEEE* 105, 12 (Dec 2017), 2295–2329.
[24] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, and et al. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *CoRR* abs/1802.04730 (2018). arXiv:1802.04730
[25] Xuechao Wei, Cody Hao Yu, Peng Zhang, and et al. 2017. Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs. In *DAC*.
[26] Jinn-Wang Yeh, Wen-Jiunn Cheng, and Chein-Wei Jen. 1996. VASS—A VLSI array system synthesizer. *Journal of VLSI signal processing systems for signal, image and video technology* (01 May 1996).