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

Jie Wang, Licheng Guo, Jason Cong
University of California, Los Angeles
Outline

• A Brief Background
• Details about AutoSA Compilation Flow
• Demos
  • Matrix Multiplication
  • Convolutional Neural Network
• Auto-Tuning in AutoSA
• Using AutoBridge to Boost the Design Frequency
AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGAs


Input: C code

Output: Systolic array design in HLS C
How Did We Get Started?

• Manual systolic array design for matrix multiplication in Summer’2014
• Time: ~5 months
• A manual-designed RTL-based 1D systolic array with over 1,700 line of codes
• 198.1 GFLOPs on Xilinx VC709
## Related Work on Automated SA Compilation

<table>
<thead>
<tr>
<th></th>
<th>MMAlpha '01 Bondhugula et al. '07</th>
<th>PolySA '18 SuSy '20</th>
<th>AutoSA '21</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Generality</strong></td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Imperfectly Nested Loops</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Multi-Statement</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>Performance</strong></td>
<td>Generate designs with performance as high as possible.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Latency Hiding</td>
<td>No</td>
<td>No</td>
<td>Auto</td>
</tr>
<tr>
<td>Double Buffering</td>
<td>No</td>
<td>No</td>
<td>Auto</td>
</tr>
<tr>
<td><strong>Productivity</strong></td>
<td>Shorten the development time as much as possible.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Auto-Tuning</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Outline

• A Brief Background
• **Details about AutoSA Compilation Flow**
• Demos
  • Matrix Multiplication
  • Convolutional Neural Network
• Auto-Tuning in AutoSA
• Using AutoBridge to Boost the Design Frequency
AutoSA is Open-Sourced

- Github: [https://github.com/UCLA-VAST/AutoSA](https://github.com/UCLA-VAST/AutoSA)
Current Capabilities

• Back-end
  • Xilinx HLS C (Mature, recommended)
  • Intel OpenCL (Experimental)
  • Catapult HLS C (Experimental)

• High-Performance designs on Xilinx platforms

<table>
<thead>
<tr>
<th>Matrix Multiplication</th>
<th>FP32</th>
<th>int16</th>
<th>Int8</th>
</tr>
</thead>
<tbody>
<tr>
<td>MHz</td>
<td>300</td>
<td>250</td>
<td>300</td>
</tr>
<tr>
<td>TFLOPs</td>
<td>0.93</td>
<td>3.41</td>
<td>6.95</td>
</tr>
</tbody>
</table>

Board: Xilinx Alveo U250
Current Capabilities

• **Architectural exploration** of different dataflows
• Example: 10 dataflows of CNN
Current Capabilities

• Auto-tuning support

• Exhaustive search with pruning
  • mins to hours
• Genetic search
  • < 1 minute
Compilation Flow

Construct and optimize PE arrays.
- Space-time mapping, array partitioning, latency hiding, vectorization

Construct and optimize I/O network.
- I/O network analysis, double buffering, data-packing

Manual

User-Specified Optimization

Auto

Auto Tuner

Search for optimal design configurations.

Generate target hardware code.

Model Extraction
- Extract polyhedral model from the source code.

Polyhedral IR
- Examine if the target program can be mapped to systolic array.

Legality Check

Computation Management

Communication Management

Code Generation
- HLS C
- ...

Optimization

Manual vs. Auto
Compilation Flow

The input program of AutoSA

```c
#pragma scop
for (int i = 0; i < M; ++i)
    for (int j = 0; j < N; ++j) {
        S0:  C[i][j] = 0;
            for (int k = 0; k < K; ++k)
                S1:  C[i][j] += A[i][k] * B[k][j];
    }
#pragma endscop
```
Polyhedral Model

- A **mathematical** compilation framework for **loop nest optimization**.
- Applicable to a **subset** of general loop programs (static control of parts (SCoP)).
- **Complete** and **robust** toolchains.
  - Integer Set Library
Compilation Flow

Legality Check

- All dependences are **uniform** dependences (with constant dependence distance)
  
  Statically-scheduled design

- Dependence distances on space loops (loops to be mapped to PE dimensions) to be **no greater than one**.
  
  Local communication
Compilation Flow

PE Array

I/O Network

DRAM

L2

L3

C

B
Compilation Flow

Model Extraction
Polyhedral IR
Legality Check
PE
PE
PE
PE
A
B
C
L1
L2
L3
L2
L3
L2
L3
PE
PE
PE
PE

Communication Management
Computation Management
Compilation Flow

Construct and optimize the PE array
Compilation Flow

Construct and optimize the I/O network
How to generate the I/O network?

- **Key idea:** Data communication is determined by the data dependences.

Read dependences (Read-After-Read, RAR) → Read-only data
Flow dependences (Read-After-Write, RAW) → Intermediate data
Output dependences (Write-After-Write, WAW) → Computation results
Communication Management

```java
for (int i = 0; i < M; ++i)
    for (int j = 0; j < N; ++j) {
        for (int k = 0; k < K; ++k)
            S1: C[i][j] = C[i][j] + A[i][k] * B[k][j];
    }
```

<table>
<thead>
<tr>
<th>Dependence</th>
<th>Dependence Type</th>
<th>Array Access</th>
<th>Dependence Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>WAW D4</td>
<td>Read (RAR)</td>
<td>A[i][k]</td>
<td>D1</td>
</tr>
<tr>
<td>RAW D3</td>
<td>Flow (RAW)</td>
<td>C[i][j]</td>
<td>D3</td>
</tr>
<tr>
<td>RAR D1</td>
<td>Read (RAR)</td>
<td>B[k][j]</td>
<td>D2</td>
</tr>
<tr>
<td>RAR D2</td>
<td>Read (RAR)</td>
<td>C[i][j]</td>
<td>D4</td>
</tr>
</tbody>
</table>

*We omit the statement of array initialization for the sake of brevity.*
### Dependence Table

<table>
<thead>
<tr>
<th>Dependence</th>
<th>Dependence Type</th>
<th>Array Access</th>
<th>Dependence Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1</td>
<td>Read (RAR)</td>
<td>A[i][k]</td>
<td>(0, 1, 0)</td>
</tr>
<tr>
<td>D2</td>
<td>Read (RAR)</td>
<td>B[k][j]</td>
<td>(1, 0, 0)</td>
</tr>
<tr>
<td>D3</td>
<td>Flow (RAW)</td>
<td>C[i][j]</td>
<td>(0, 0, 1)</td>
</tr>
<tr>
<td>D4</td>
<td>Output (WAW)</td>
<td>C[i][j]</td>
<td>(0, 0, 1)</td>
</tr>
</tbody>
</table>

_We omit the statement of array initialization for the sake of brevity._

---

5/12/2021
Communication Management

How to optimize the I/O network?

• I/O network localization
• Double buffering, data packing, etc.
Compilation Flow

C

Model Extraction

Polyhedral IR

Legality Check

Computation Management

Communication Management

Code Generation

HLS C

...
Outline

• A Brief Background
• Details about AutoSA Compilation Flow
• Demos
  • Matrix Multiplication
  • Convolutional Neural Network
• Auto-Tuning in AutoSA
• Using AutoBridge to Boost the Design Frequency
Demo 1: Matrix Multiplication

• Document: https://autosa.readthedocs.io/en/latest/examples/mm.html

• Example directory: ${AUTOSA_ROOT}/autosa_tests/mm
A Closer Look at Computation Management

- **Space-time mapping**: transforming the program to a systolic array with space-time mapping
- **Array partitioning**: partitioning the array into smaller sub-arrays to fit limited on-chip resource
- **Latency hiding**: permuting parallel loops inside to hide computation latency
- **SIMD vectorization**: vectorizing computation to amortize the PE control overheads

```c
#pragma scop
for (int i = 0; i < M; ++i)
    for (int j = 0; j < N; ++j) {
        for (int k = 0; k < K; ++k)
            C[i][j] += A[i][k] * B[k][j];
    }
#pragma endscop
```

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

---sa-sizes="{Kernel[]->space_time[3]}"
A Closer Look at Computation Management

- **Space-time mapping**: transforming the program to a systolic array with space-time mapping
- **Array partitioning**: partitioning the array into smaller sub-arrays to fit limited on-chip resource
- **Latency hiding**: permuting parallel loops inside to hide computation latency
- **SIMD vectorization**: vectorizing computation to amortize the PE control overheads

```
for (int i = 0; i < I; i++)
    for (int j = 0; j < J; j++)
        for (int k = 0; k < K; k++)
            C[i][j] += A[i][k] * B[k][j];
```

---sa-sizes="{Kernel[]->array_part[4,4,4]}"

![Diagram showing space-time mapping, array partitioning, and latency hiding.](image)
A Closer Look at Computation Management

- **Space-time mapping**: transforming the program to a systolic array with space-time mapping
- **Array partitioning**: partitioning the array into smaller sub-arrays to fit limited on-chip resource
- **Latency hiding**: permuting parallel loops inside to hide computation latency
- **SIMD vectorization**: vectorizing computation to amortize the PE control overheads

```
for (int i = 0; i < M/4; i++)
    for (int j = 0; j < N/4; j++)
        for (int k = 0; k < K/4; k++)
            for (int ii = 0; ii < 2; ii++)
                for (int jj = 0; jj < 2; jj++)
                    for (int kk = 0; kk < 4; kk++)
```

```
#pragma HLS PIPELINE II=1
for (int i = 0; i < M/4; i++)
    for (int j = 0; j < N/4; j++)
        for (int k = 0; k < K/4; k++)
            for (int ii = 0; ii < 2; ii++)
                for (int jj = 0; jj < 2; jj++)
                    for (int kk = 0; kk < 4; kk++)
```

---

For the 4x4 array:
```
--sa-sizes="{Kernel[]->latency[2,2]}
```

For the 2x2 array:
```
```

5/12/2021
A Closer Look at Computation Management

- **Space-time mapping**: transforming the program to a systolic array with space-time mapping
- **Array partitioning**: partitioning the array into smaller sub-arrays to fit limited on-chip resource
- **Latency hiding**: permuting parallel loops inside to hide computation latency
- **SIMD vectorization**: vectorizing computation to amortize the PE control overheads

```c
for (int i = 0; i < M/4; i++)
for (int j = 0; j < N/4; j++)
for (int k = 0; k < K/4; k++)
  for (int ii = 0; ii < 2; ii++)
    for (int jj = 0; jj < 2; jj++)
      for (int kk = 0; kk < 4; kk++)
        for (int iii = 0; iii < 2; iii++)
          #pragma HLS PIPELINE II=1
          for (int jjj = 0; jjj < 2; jjj++)
            S1;
```

```c
for (int i = 0; i < M/4; i++)
for (int j = 0; j < N/4; j++)
for (int k = 0; k < K/4; k++)
  for (int ii = 0; ii < 2; ii++)
    for (int jj = 0; jj < 2; jj++)
      for (int kk = 0; kk < 2; kk++)
        for (int iii = 0; iii < 2; iii++)
          #pragma HLS PIPELINE II=1
          for (int jjj = 0; jjj < 2; jjj++)
            S1;
```

--sa-sizes="{Kernel[]-> simd[2]}"
Demo 2: CNN

• Example directory: ${AUTOSA_ROOT}/autosa_tests/cnn
• A Brief Background
• Details about AutoSA Compilation Flow
• Demos
  • Matrix Multiplication
  • Convolutional Neural Network
• Auto-Tuning in AutoSA
• Using AutoBridge to Boost the Design Frequency
Auto-Tuning

A Large Design Space to Explore

Example: Matrix Multiplication

\[
\begin{align*}
1024 & \times 1024 = 1024 \\
A & \times B = C \\
\end{align*}
\]

Dataflow types (6) x Dataflow configurations \((O(2^{40}))\)
Auto-Tuning

Generality

- Program with restrictions
  - Rectangular iteration domain
- DSP, BRAM

- Arbitrary program
  - DSP, BRAM, LUT, FF

Convergence Time

- Genetic Search: < 1 minute
- Exhaustive Search with Pruning: Minutes to hours
Auto-Tuning

- Exhaustive search

- Genetic search

- Demo:
  - Use genetic search for matrix multiplication on Alveo U250
Auto-Tuning

• Genetic search

• Demo:
  • Use genetic search for matrix multiplication on Alveo U250

---

**Step 1: Generate Design Descriptor**
Contains necessary information for resource and latency estimation.

**Step 2: Search**
Use genetic search to find near-optimal designs.
Outline

• A Brief Background
• Details about AutoSA Compilation Flow
• Demos
  • Matrix Multiplication
  • Convolutional Neural Network
• Auto-Tuning in AutoSA
• Using AutoBridge to Boost the Design Frequency
Getting High-Frequency Systolic Array Designs with **Floorplanning**

Default place/router make suboptimal choices.
Design Freq.: **216MHz**

**Floorplanning**-guided approach spreads the logic across chip to reduce local congestion.
Design Freq.: **329MHz**

Example: Improve the CNN systolic array design frequency using AutoBridge
Automated Interconnect Pipelining for Multi-Die FPGA Designs


- AutoBridge is an HLS-based automatic floorplanning flow.
- Improved the average frequency of 43 designs from 147MHz to 297MHz with a negligible area overhead.
- Github: https://github.com/Licheng-Guo/AutoBridge

```c
#pragma scop
for (int i = 0; i < M; ++i)
    for (int j = 0; j < N; ++j) {
        S0:  C[i][j] = 0;
            for (int k = 0; k < K; ++k)
        S1:  C[i][j] += A[i][k] * B[k][j];
    }
#pragma endscop
```

Input: C code

HLS C code

HLS synthesized RTL + floorplanning constraints

Vivado

AutoBridge

AutoSA
More Details

• An example of using AutoBridge with AutoSA
• https://autosa.readthedocs.io/en/latest/tutorials/auto_bridge.html
Other Features

• Support for structural sparsity
Other Features

• Support for HBM
  • https://autosa.readthedocs.io/en/latest/examples/mm_hbm.html
How Can This Tool Benefit the Community?

• Understand the mechanisms of systolic array architecture, HLS design methodology, and polyhedral compilation
  • UCLA CS259

• Integrate AutoSA-generated designs into the accelerator framework
  • FlexCNN, HeteroCL

• Architecture studies based on an open-sourced and performant baseline
  • Architecture trade-offs for applications (dataflows, array configs, ...)
  • Programmable systolic array
  • Comparison against CGRA
  • ...

5/12/2021
Conclusion

• AutoSA, an end-to-end systolic array compiler for FPGA

• Goal: maximize generality, performance, and productivity when generating systolic arrays.

• Open-source and under active development. Welcome for any feedbacks and contribution!

• Any other questions?

• jiewang@cs.ucla.edu
# User-Specified Optimizations

<table>
<thead>
<tr>
<th>Option</th>
<th>Explanation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>array-contraction</td>
<td>Apply array contraction to reduce the memory usage.</td>
<td>True/False</td>
</tr>
<tr>
<td>axi-stream</td>
<td>Generate AXI stream interface.</td>
<td>True/False</td>
</tr>
<tr>
<td>block-sparse</td>
<td>Use block sparsity.</td>
<td>True/False</td>
</tr>
<tr>
<td>block-sparse-ratio</td>
<td>Block sparsity ratio.</td>
<td>[kernel[]-&gt;A[2,4]]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Matrix A is sparse, every 4 elements are grouped into a vector, with 2 non-zero elements.</td>
</tr>
<tr>
<td>data-pack</td>
<td>Enable data-packing.</td>
<td>True/False</td>
</tr>
<tr>
<td>data-pack-sizes</td>
<td>Specifies the maximal data-packing factors at each I/O level.</td>
<td>[kernel[]-&gt;data_pack[8,32,64]]</td>
</tr>
<tr>
<td>double-buffer</td>
<td>Enable double buffering.</td>
<td>True/False</td>
</tr>
<tr>
<td>double-buffer-assign</td>
<td>Assign specific arrays to be double-buffered.</td>
<td>[kernel[]-&gt;A[]]</td>
</tr>
</tbody>
</table>
## User-Specified Optimizations

<table>
<thead>
<tr>
<th>Option</th>
<th>Explanation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>explore-loop-permute</td>
<td>Explore loop permutation in the step of array partitioning.</td>
<td>True/False</td>
</tr>
<tr>
<td>fifo-depth</td>
<td>FIFO depth.</td>
<td>2</td>
</tr>
<tr>
<td>hbm</td>
<td>Use multi-port DRAM/HBM</td>
<td>True/False</td>
</tr>
<tr>
<td>hbm-port-num</td>
<td>Specify HBM port number per array.</td>
<td>2</td>
</tr>
<tr>
<td>hls</td>
<td>Generate Xilinx HLS host.</td>
<td>True/False</td>
</tr>
<tr>
<td>host-serialize</td>
<td>Serialize/deserialize the host data.</td>
<td>True/False</td>
</tr>
<tr>
<td>io-module-embedding</td>
<td>Embed I/O modules inside PEs if possible.</td>
<td>True/False</td>
</tr>
<tr>
<td>loop-infinitize</td>
<td>Apply loop infinitization optimization (Intel OpenCL only)</td>
<td>True/False</td>
</tr>
</tbody>
</table>
## User-Specified Optimizations

<table>
<thead>
<tr>
<th>Option</th>
<th>Explanation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>local-reduce</td>
<td>Generate non-output-stationary array with local reduction.</td>
<td>True/False</td>
</tr>
<tr>
<td>reduce-op</td>
<td>Reduction operator.</td>
<td>“+”</td>
</tr>
<tr>
<td>max-sa-dim</td>
<td>Maximal systolic array dimension.</td>
<td>2</td>
</tr>
<tr>
<td>sa-sizes</td>
<td>Computation management parameters.</td>
<td>[kenrel[]-&gt;array_part[4,4]]</td>
</tr>
</tbody>
</table>