Democratizing Customized Computing

Jason Cong
Volgenau Chair for Engineering Excellence, UCLA Computer Science
Director, Center for Domain-Specific Computing (CDSC)
https://vast.cs.ucla.edu/people/faculty/jason-cong
From Parallization to Customization

... to look beyond parallelization and focus on domain-specific customization to bring significant power-performance efficiency ...

Original source: Shekhar Borkar, Intel
Customized Computing has been Our Research Focus Since 2009

Customizable Domain-Specific Computing

Yu-Ting Chen
Jason Cong
Michael Gill
Glenn Reinman
Bingjun Xiao

Synthesis Lectures on Computer Architecture

IEEE Design & Test, 2011

Synthesis Lectures on Computer Architectures, 2015

Proceedings of IEEE, 2019
Successful Examples of Customization

• Example:
  • Google TPU (Tensor Processing Unit)

• First version: 2014

• Revised TPU (2017), for training and inference
  • DRAM, 2 DDR3 -> GDDR5, 34GB/s -> 180GB/s
  • 200x perf/W of Haswell CPU, 70x perf/W of K80 GPU

Based on data in [ISCA2017]

• Limitations:
  • Too costly for individuals (or small companies) to design
  • Take too much time to build

Google TPU: In-Datacenter Performance Analysis of a Tensor Processing Unit, ISCA 2017
Customized Computing on FPGAs: Example: Scalable Sorting [ISCA 2020]

- Bonsai: Adaptive merge tree sort solution (compute and I/O optimal)
  - Optimized configuration of merge sort kernel for different memory configurations
  - Best DRAM-scale sorting performance
  - Scale to TB sorting via reconfiguration

Use of FPGAs trade-off performance for design cost, flexibility, and time-to-silicon
Power of Customization (Domain-Specific Accelerators)

- Special Data Types and Operations
  - Do in 1 cycle what normally takes 10s or 100s — 10-1000x efficiency gain

Most significant on ASIC (if one can afford cost and time)
Still very substantial speedup on FPGAs despite its overhead
Question:
Can Every Programmer Easily Design DSAs?

Or
Can Every Serious Programmer Easily Design DSAs?

Current Answer: Yes and No
It’s Natural to Think about High-Level Synthesis (HLS)?

Significant progress in the past decade

- Example: xPilot (UCLA 2006) -> AutoPilot (AutoESL) -> Vivado HLS (Xilinx 2011-)
  - Platform-based C to RTL synthesis
  - Synthesize pure ANSI-C and C++, GCC-compatible compilation flow leveraging LLVM framework
  - Full support of IEEE-754 floating point data types & operations
  - Efficiently handle bit-accurate fixed-point arithmetic
  - SDC-based scheduling
  - Automatic memory partitioning

QoR matches or exceeds manual RTL for many designs

TCAD April 2011 (keynote paper) “High-Level Synthesis for FPGAs: From Prototyping to Deployment”
**Good News: Not Difficult to Create Circuits from C/C++ Using HLS**

**Example code**
- MVT kernel from Polybench
  - Two matrix-vector multiplications

```c
void kernel_mvt(double x1[120], double x2[120],
                 double y_1[120], double y_2[120],
                 double A[120][120]) {

    for (int i = 0; i < 120; i++) {
        for (int j = 0; j < 120; j++) {
            x1[i] += A[i][j] * y_1[j];
        }
    }

    for (int i = 0; i < 120; i++) {
        for (int j = 0; j < 120; j++) {
            x2[i] += A[j][i] * y_2[j];
        }
    }
}
```
Challenge 1: Synthesized Circuit May Not Have Good Performance

- Not surprising if you have done multi-core programming – the same problem!
- Need to add *pragmas* (*microarchitecture hints*).

Example code: MVT kernel from Polybench

```c
void kernel_mvt(double x1[120], double x2[120],
                 double y_1[120], double y_2[120],
                 double A[120][120]) {
    for (int i = 0; i < 120; i++) {
        for (int j = 0; j < 120; j++) {
            x1[i] += A[i][j] * y_1[j];
        }
    }
    for (int i = 0; i < 120; i++) {
        for (int j = 0; j < 120; j++) {
            x2[i] += A[j][i] * y_2[j];
        }
    }
}
```

When targeting FPGA, 13x slower than running on a single-core CPU

After proper pragma insertions

```c
void kernel_mvt(double x1[120], double x2[120],
                 double y_1[120], double y_2[120],
                 double A[120][120]) {
    #pragma ACCEL PIPELINE flatten
    for (int i = 0; i < 120; i++) {
        for (int j = 0; j < 120; j++) {
            x1[i] += A[i][j] * y_1[j];
        }
    }
    #pragma ACCEL PARALLEL FACTOR=15
    for (int i = 0; i < 120; i++) {
        for (int j = 0; j < 120; j++) {
            x2[i] += A[j][i] * y_2[j];
        }
    }
}
```

122x speedup

Based on the Merlin Compiler, open-sourced by AMD/Xilinx
Challenge 2: # Possibilities for Pragmas Insertion Can Be Very Large!

Example code: MVT kernel from Polybench

```c
void kernel_mvt(double x1[120], double x2[120], double y_1[120], double y_2[120], double A[120][120]) {
    #pragma ACCEL PIPELINE
    for (int i = 0; i < 120; i++) {
        for (int j = 0; j < 120; j++) {
            x1[i] += A[i][j] * y_1[j];
        }
    }
}
```

When targeting FPGA, 13x slower than running on a single-core CPU

Solution space
- > 3M design choices

```c
void kernel_mvt(double x1[120], double x2[120], double y_1[120], double y_2[120], double A[120][120]) {
    #pragma ACCEL PIPELINE auto(__PIPE__L0)
    #pragma ACCEL TILE FACTOR=auto(__TILE__L0)
    #pragma ACCEL PARALLEL FACTOR=auto(__PARA__L0)
    for (int i = 0; i < 120; i++) {
        for (int j = 0; j < 120; j++) {
            x1[i] += A[i][j] * y_1[j];
        }
    }
}
```

Search space by AutoDSE
Overview of Our Approach

1. Architecture Guided Optimization: Based on common computation patterns
   - Systolic Array [DAC ’17, ICCAD ’18]
   - Stencil [ICCAD ’18]
   - Composable, Parallel and Pipeline (CPP) [DAC ’18]
   - Variable loop bounds [ICCAD ’18]

2. Apply ML or other optimization techniques for general applications (GNN-DSE) [DAC’22]

3. Compose the entire design using latency-insensitive dataflow task [FCCM’21 & FPGA’21 & 21]

Goal: “Democratize” accelerator designs for customized computing
Example of Architecture-Guided Optimization: AutoSA

Wang, Jie, Licheng Guo, and Jason Cong. "AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA." FPGA’2021

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

Input: C code

Output: Systolic array design in HLS C
Systolic Array Advantages

- Parallelism
- Locality
- Performance
- Energy Efficiency
Many Accelerators are Based on Systolic Arrays

Google TPU

Tesla Self-Driving Chip

Amazon Infrentia
Systolic Array Design Stories from Industry

Overview of AutoSA Compilation Flow

- Extract polyhedral model from the source code.
- Examine if the target program can be mapped to systolic array.
- 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
- Generate target hardware code
Challenge: Large Design Space & Many Optimization Opportunities

Example: Matrix Multiplication

\[
\begin{align*}
A \times B &= C \\
1024 \times 1024 \times 1024 &\quad \text{Dataflows types(6) X Dataflow Configurations(O(2^{40}))}
\end{align*}
\]
A Closer Look at Computation Management

• **Space-time mapping**: transforming the program to a systolic array with space-time mapping.

```
Input Code of MM:
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];

Note: Initialization of C omitted for brevity.
```

```
Space-Time Transformation: [i, j]

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];

* The generated systolic array:
```

```

10/5/22
A Closer Look at Computation Management

- **Array partitioning**: partitioning the array into smaller sub-arrays to fit limited on-chip resource.

**Space-Time Transformation: \([i, j]\)**

```java
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];
```

*The generated systolic array:

```
A
```

```
PE
...
PE
```

```
B
```

```
P
```

```
P
```

```
P
```

```
P
```

```
P
```

```
P
```

```
P
```

```
P
```

```
P
```

```
P
```

```
P
```

```
P
```

```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
```
P
```
A Closer Look at Computation Management

- **Latency hiding**: permuting parallel loops inside to hide computation latency.

```c
Latency Hiding

for (int i.0 = 0; i.0 < I/T_I1; i.0++)
    for (int j.0 = 0; j.0 < J/T_J1; j.0++)
        for (int k.0 = 0; k.0 < K/T_K1; k.0++)
            for (int i.1 = 0; i.1 < T_I1/T_I2; i.1++)
                for (int j.1 = 0; j.1 < T_J1/T_J2; j.1++)
                    for (int k.1 = 0; k.1 < T_K1; k.1++)
                        for (int i.2 = 0; i.2 < T_I2; i.2++)
                            for (int j.2 = 0; j.2 < T_J2; j.2++)
                                C[...] += A[...] * B[...];
```
A Closer Look at Computation Management

- **SIMD vectorization**: vectorizing computation to amortize the PE control overheads.

```c
for (int i.0 = 0; i.0 < I/T_I1; i.0++)
  for (int j.0 = 0; j.0 < J/T_J1; j.0++)
    for (int k.0 = 0; k.0 < K/T_K1; k.0++)
      for (int i.1 = 0; i.1 < T_I1/T_I2; i.1++)
        for (int j.1 = 0; j.1 < T_J1/T_J2; j.1++)
          for (int k.1 = 0; k.1 < T_K1/T_K2; k.1++)
            for (int i.2 = 0; i.2 < T_I2; i.2++)
              for (int j.2 = 0; j.2 < T_J2; j.2++)
                for (int k.2 = 0; k.2 < T_K2; k.2++)
                  C[...] += A[...] * B[...];
```
What about Data Communication?

- Polyhedral model supports precise data dependence analysis.

```
for (int i = 0; i < I; i++) // space
    for (int j = 0; j < J; j++) { // space
        for (int k = 0; k < K; k++) // time
            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>
<th>I/O Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1</td>
<td>Read (RAR)</td>
<td>A[i][k]</td>
<td>(0, 1, 0)</td>
<td>g1</td>
</tr>
<tr>
<td>D2</td>
<td>Read (RAR)</td>
<td>B[k][j]</td>
<td>(1, 0, 0)</td>
<td>g2</td>
</tr>
<tr>
<td>D3</td>
<td>Flow (RAW)</td>
<td>C[i][j]</td>
<td>(0, 0, 1)</td>
<td>g3</td>
</tr>
<tr>
<td>D4</td>
<td>Output (WAW)</td>
<td>C[i][j]</td>
<td>(0, 0, 1)</td>
<td>g4</td>
</tr>
</tbody>
</table>

We omit the statement of array initialization for brevity.

Example: I/O network generation based on the polyhedral model
Use Dependency to Construct Communication Network

- Polyhedral model supports precise data dependence analysis.

<table>
<thead>
<tr>
<th>Dependence</th>
<th>Dependence Type</th>
<th>Array Access</th>
<th>Dependence Distance</th>
<th>I/O Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1</td>
<td>Read (RAR)</td>
<td>A[i][k]</td>
<td>(0, 1, 0)</td>
<td>g1</td>
</tr>
<tr>
<td>D2</td>
<td>Read (RAR)</td>
<td>B[k][j]</td>
<td>(1, 0, 0)</td>
<td>g2</td>
</tr>
<tr>
<td>D3</td>
<td>Flow (RAW)</td>
<td>C[i][j]</td>
<td>(0, 0, 1)</td>
<td>g3</td>
</tr>
<tr>
<td>D4</td>
<td>Output (WAW)</td>
<td>C[i][j]</td>
<td>(0, 0, 1)</td>
<td>g4</td>
</tr>
</tbody>
</table>

We omit the statement of array initialization for brevity.

Example: I/O network generation based on the polyhedral model
Auto-Tuning in AutoSA (More in Late Slides)

Input:
A SCoP program with rectangular iteration domains.

Mathematic Programming-Based Optimizer
Evolutionary Search

Odyssey

< 1 minute

Input:
An arbitrary SCoP program.

Exhaustive Search with Pruning

minutes to hours

Generality

Search Time
## Benchmark Examples and Productivity Gain

<table>
<thead>
<tr>
<th>Application</th>
<th>Problem Size</th>
<th>#Statements</th>
<th>Input C LOC</th>
<th>Output HLS LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix Multiplication</td>
<td>$[i, j, k]: [1024, 1024, 1024]$</td>
<td>2</td>
<td>7</td>
<td>9265</td>
</tr>
<tr>
<td>CNN</td>
<td>$[i, o, h, w, p, q]: [512, 512, 56, 56, 3, 3]$</td>
<td>2</td>
<td>10</td>
<td>9861</td>
</tr>
<tr>
<td>MTTKRP</td>
<td>$[i, k, l, j]: [512, 512, 512, 512]$</td>
<td>2</td>
<td>9</td>
<td>7858</td>
</tr>
<tr>
<td>TTMc</td>
<td>$[i, j, k, l, m]: [128, 128, 128, 128, 128]$</td>
<td>2</td>
<td>9</td>
<td>7637</td>
</tr>
<tr>
<td>LU Decomposition</td>
<td>$[n]: [12/16/20/24]$</td>
<td>9</td>
<td>27</td>
<td>1316</td>
</tr>
</tbody>
</table>

Complex systolic Array from C-to-Silicon in a day! Recall that common industry practice requires 4-18 months.
## Performance

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Platform</th>
<th>Array Sizes</th>
<th>Data Type</th>
<th>GFLOPs</th>
<th>MHz</th>
<th>DSP Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CNN</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wei et al. '17</td>
<td>Intel Arria 10</td>
<td>8x19x8</td>
<td>FP32</td>
<td>602.8</td>
<td>253</td>
<td>97%</td>
</tr>
<tr>
<td>AutoSA</td>
<td>Xilinx Alveo U250</td>
<td>16x14x8</td>
<td>FP32</td>
<td>950.2</td>
<td>272</td>
<td>97%</td>
</tr>
<tr>
<td>Srivastava et al. '19</td>
<td>Intel Arria 10</td>
<td>8x9x16</td>
<td>FP32</td>
<td>700</td>
<td>204</td>
<td>99%</td>
</tr>
<tr>
<td><strong>MTTKRP</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AutoSA</td>
<td>Xilinx Alveo U250</td>
<td>16x8x8</td>
<td>FP32</td>
<td>896.7</td>
<td>296</td>
<td>99%</td>
</tr>
<tr>
<td>Srivastava et al. '19</td>
<td>Intel Arria 10</td>
<td>8x10x16</td>
<td>FP32</td>
<td>738</td>
<td>205</td>
<td>94%</td>
</tr>
<tr>
<td><strong>TTMc</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AutoSA</td>
<td>Xilinx Alveo U250</td>
<td>16x8x8</td>
<td>FP32</td>
<td>886.2</td>
<td>290</td>
<td>99%</td>
</tr>
</tbody>
</table>
AutoSA is Open-Sourced

- Github: https://github.com/UCLA-VAST/AutoSA
Some Architecture Insights from AutoSA

- Example: 1024x1024x1024 GEMM
- Common wisdom: dimensions of a systolic array be divisors of the problem size.
  - Timeloop ISPASS ’19 (MIT, Nvidia, Stanford),
  - dMazeRunner TECS ’19 (Ariazon State Univ., Yonsei Univ., Intel)
  - Interstellar ASPLOS ’20 (Stanford, Tsinghua)
- Non-divisor solution can be 50% faster

<table>
<thead>
<tr>
<th>SA Size (Cols,Rows, SIMD)</th>
<th>DSPs</th>
<th>Frequency</th>
<th>Throughput GFLOP/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>32x4x8</td>
<td>5120</td>
<td>257 MHz</td>
<td>506.71</td>
</tr>
<tr>
<td>16x13x8</td>
<td>8320</td>
<td>243 MHz</td>
<td>764.46</td>
</tr>
</tbody>
</table>

Non divisor

10/5/22
Some More Architecture Insight using AutoSA

• Common wisdom: Minimize off-chip communication. E.g
  • Marvel Arxiv ’20 (Georgia Tech, Nvidia),
  • Chen et al. HPCA ’20 (UCAS, Tsinghua Univ.)

• Again, not necessarily!

<table>
<thead>
<tr>
<th>SA Size (Cols,Rows, SIMD)</th>
<th>Minimization Goal</th>
<th>DSPs</th>
<th>Frequency</th>
<th>Throughput</th>
<th>DRAM Traffic</th>
<th>CTC (FLOP/byte)</th>
<th>Effective Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>32x4x8</td>
<td>DRAM Traffic</td>
<td>5120</td>
<td>282 MHz</td>
<td>496.16 GFLOP/s</td>
<td>16.7 MB</td>
<td>128</td>
<td>4.3 GB/s</td>
</tr>
<tr>
<td>16x13x8</td>
<td>Latency</td>
<td>8320</td>
<td>243 MHz</td>
<td>764.46 GFLOP/s</td>
<td>80.3 MB</td>
<td>26.7</td>
<td>36.5 GB/s</td>
</tr>
</tbody>
</table>
What about General C/C++ Programs?

- Adopting the Merlin Compiler, developed by Falcon Computing (acquired by Xilinx in 2020 and open-sourced in 2021)

  
  ```c
  #pragma ACCEL parallel
  - Run multiple loop iterations in parallel (instruction/task-level)
  
  #pragma ACCEL pipeline
  - Run multiple loop iterations in pipeline (instruction/task-level)
  ```

OpenMP for multi-core CPUs

```c
#pragma omp parallel for num_threads(16)
for (int i = 0; i < N; ++i) {
   c[i] += a[i] * b[i];
}
```

Merlin for FPGAs

```c
#pragma ACCEL parallel factor=16
for (int i = 0; i < N; ++i) {
   c[i] += a[i] * b[i];
}
```

**Automated code transformation and transformation**

- On-chip memory banking/partitioning/delinearization
- External memory bursting/streaming/coalescing
- Host interface and host code generation (In OpenCL)

**Advanced options for parallel:** reduction and stencil variables
AutoDSE: Bottleneck-based Optimizer [TODAES’22]

void kernel_mvt(double x1[120], double x2[120], double y_1[120], double y_2[120], double A[120][120]) {
    for (int i = 0; i < 120; i++) {
        for (int j = 0; j < 120; j++) {
            x1[i] += A[i][j] * y_1[j];
        }
    }
    for (int i = 0; i < 120; i++) {
        for (int j = 0; j < 120; j++) {
            x2[i] += A[j][i] * y_2[j];
        }
    }
}

#pragma ACCEL PIPELINE flatten
for (int i = 0; i < 120; i++) {
    for (int j = 0; j < 120; j++) {
        x1[i] += A[i][j] * y_1[j];
    }
}
#pragma ACCEL PARALLEL FACTOR=15
for (int i = 0; i < 120; i++) {
    if (i == 10) {
        #pragma ACCEL PARALLEL reduction = x2 FACTOR=12
        for (int j = 0; j < 120; j++) {
            x2[i] += A[j][i] * y_2[j];
        }
    }
}

10/5/22
Evaluation on Xilinx Vitis Library

- Tested on 33 kernels, each has 13.5 HLS optimization pragmas on average,
  - AutoDSE achieves roughly the same performance (1.04x higher)
  - Eliminated all HLS or Merlin optimization pragmas
- Both Merlin and AutoDSE keep and propagate dataflow and streaming pragmas
  - Will rely on dataflow composition using TAPA (later)
Current Goal: More Extensive DSE Using Deep Graph Learning

• Review of the problem

Manual code
• MVT kernel from Polybench
• Two matrix-vector multiplications

Solution space
• > 3M design choices

Solution:
Adopt a deep graph learning model to automatically learn the program's features
Step 1: Create a Database for Training the Model

- Database generation:
  - Adapting our previous work
  - AutoDSE [TODAES'22]
Step 2: Represent the Program as a Graph

- Build the graph using the LLVM IR to capture lower-level instructions, i.e. closer to hardware
- Need to include both the program semantic and pragma flow in the graph
  - Program semantic: control, data, and call flow
    - Adapting the latest representation proposed for including these information (ProGraML [ICML'21])
- The graph is generated once per kernel and filled with different pragma values later on
Step 3: Build a Predictive Model

- GNN-based model:
  - A single model across all applications

\[ h'_i = \sigma \left( \sum_{j \in N(i) \cup \{i\}} \alpha_{i,j} Wh_j \right) \]

Function of neighboring nodes and their edge embeddings
Design Space Exploration in GNN-DSE

- The trained model is replaced with the HLS tool for evaluating the design points.
- The top M design points are evaluated with the HLS tool and added to the training database for subsequent trainings.

Diagram:

- C/C++ Code
- Graph Generator
- Pragma Fill
- Design Config
- GNN-DSE's Predictive Model
- Design Space Explorer
- Top M Designs
- Evaluator (HLS tool)

10/5/22
Experimental Results

- Model’s performance
  - Regression loss is in RMSE

<table>
<thead>
<tr>
<th>Model</th>
<th>Method</th>
<th>Speedup</th>
<th>DSP</th>
<th>LUT</th>
<th>FF</th>
<th>BRAM</th>
<th>All</th>
<th>Accuracy</th>
<th>F1-score</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>MLP-pragma (based on Kown, et al. MLCAD’20)</td>
<td>3.28</td>
<td>0.59</td>
<td>0.31</td>
<td>0.25</td>
<td>0.34</td>
<td>4.76</td>
<td>0.52</td>
<td>0.42</td>
</tr>
<tr>
<td>M2</td>
<td>M1 + program context</td>
<td>2.94</td>
<td>0.47</td>
<td>0.24</td>
<td>0.13</td>
<td>0.16</td>
<td>3.94</td>
<td>0.78</td>
<td>0.40</td>
</tr>
<tr>
<td>M3</td>
<td>GNN-DSE</td>
<td>0.56</td>
<td>0.13</td>
<td>0.08</td>
<td>0.06</td>
<td>0.05</td>
<td>0.85</td>
<td>0.93</td>
<td>0.87</td>
</tr>
</tbody>
</table>

- Keep augmenting database until design space exploration (DSE) matches the best designs
  - Initial database:
    - 4428 total configs / 1036 valid configs
  - Final database:
    - 4752 total configs / 1278 valid configs
- More training examples lead to better accuracy
Experimental Results on Unseen Kernels

- DSE results on new kernels which were not in the database
  - All new kernels dealing with matrix vector operations
    - But with different coding styles, input sizes, and loop trip counts from our database
- Baseline: AutoDSE after 21 h
- GNN-DSE could achieve about the same performance
  - From −2% and +5% difference with a mean of +1%
  - With a maximum DSE time of 1 hour
- Adapting to domain shift in "Improving GNN-Based Accelerator Design Automation with Meta Learning [DAC'22]"

<table>
<thead>
<tr>
<th>Kernel</th>
<th># pragma</th>
<th># Design configs</th>
<th>DSE + HLS Runtime (mins)</th>
<th># Explored</th>
<th>Runtime Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>bicg</td>
<td>5</td>
<td>3,536</td>
<td>18</td>
<td>3,536</td>
<td>69x</td>
</tr>
<tr>
<td>doitgen</td>
<td>6</td>
<td>179</td>
<td>16</td>
<td>179</td>
<td>11x</td>
</tr>
<tr>
<td>gesummv</td>
<td>4</td>
<td>1,581</td>
<td>16</td>
<td>1,581</td>
<td>79x</td>
</tr>
<tr>
<td>2mm</td>
<td>14</td>
<td>492,787,501</td>
<td>74</td>
<td>78,676</td>
<td>17x</td>
</tr>
</tbody>
</table>
Current Limitation of GNN-DSE – Domain Shift

- Experimental evidence
  - Trained on a suite of 9 kernels
  - Tested on 5 different kernels with only 20 labeled designs for each of the 5 new kernels
  - Root mean square error (RMSE) on the hold-out test set of each new kernel

<table>
<thead>
<tr>
<th></th>
<th>jacobi-1d</th>
<th>fdtd-2d</th>
<th>gemm</th>
<th>3mm</th>
<th>gemver</th>
</tr>
</thead>
</table>

- DSE speedup with respect to AutoDSE after 20 hours

<table>
<thead>
<tr>
<th></th>
<th>jacobi-1d</th>
<th>fdtd-2d</th>
<th>gemm</th>
<th>3mm</th>
<th>gemver</th>
</tr>
</thead>
<tbody>
<tr>
<td>GNN-DSE</td>
<td>0.44x</td>
<td>0.06x</td>
<td>0.87x</td>
<td>0.30x</td>
<td>0.20x</td>
</tr>
</tbody>
</table>

- Accuracy drops when the testing kernels differ a lot from the training ones (domain shift), causing unsatisfactory DSE results. Meanwhile, our goal is to design a method that works well on any real-world kernel.
Proposal: Use Transfer Learning (GNN-DSE-MAML)
GNN-DSE (top) vs GNN-DSE-MAML (bottom)

Stage 1: Training
- Model (randomly initialized)
- Kernels used for training: aes, atax, gemm-blocked, gemm-ncubed, nw

Stage 2: Offline Testing
- Error on the hold-out test set:
- RMSE of util-DSP:
- RMSE of util-BRAM:

Stage 3: DSE (online testing)
- DSE speedup

Stage 4: DSE (online testing)
- DSE speedup

via MAML
Inspiration: K-shot Image Classification Using Meta-Learning

- **Meta-learning:**
  - Compute a model that can eventually generalize across many tasks
  - with good data and computation efficiency

- **Example:**
  - *K*-shot image classification task:
  - learn a classification model that can quickly adapt to a new class with only *K* images from that class
MAML for Training

**Algorithm 1** Training procedure of GNN-Dse-MAML

Require: $p(\mathcal{P}^{(\text{train})})$: distribution over kernels (programs) for training

Require: $\alpha$, $\beta$: step size hyperparameters

1: randomly initialize $\theta$
2: while not done do
3: \hspace{1em} Sample batch of kernels $\mathcal{P}_i \sim p(\mathcal{P}^{(\text{train})})$
4: \hspace{1em} for all $\mathcal{P}_i$ do
5: \hspace{2em} Sample $K$ datapoints $\mathcal{D} = \{X_j, Y_j\}$ from $\mathcal{P}_i$
6: \hspace{2em} Evaluate $\nabla_{\theta} \mathcal{L}_{\mathcal{P}_i}(f_{\theta})$ using $\mathcal{D}$ and $\mathcal{L}_{\mathcal{P}_i}$ in Equation 1
7: \hspace{2em} Compute adapted parameters with gradient descent: $\theta'_i = \theta - \alpha \nabla_{\theta} \mathcal{L}_{\mathcal{P}_i}(f_{\theta})$
8: \hspace{2em} Sample datapoints $\mathcal{D}'_i = \{x^{(j)}, y^{(j)}\}$ from $\mathcal{P}_i$ for the meta-update
9: \hspace{1em} end for
10: Update $\theta \leftarrow \theta - \beta \nabla_{\theta} \sum_{\mathcal{P}_i \sim p(\mathcal{P})} \mathcal{L}_{\mathcal{P}_i}(f_{\theta'_i})$ using each $\mathcal{D}'_i$ and $\mathcal{L}_{\mathcal{P}_i}$ in Equation 1
11: end while

A batch of kernels
MAML for Adaptation

Algorithm 1: Training procedure of GNN-Dse-MAML

Require: \( p(\mathcal{P}^{(train)}) \): distribution over kernels (programs) for training

Require: \( \alpha, \beta \): step size hyperparameters

1. randomly initialize \( \theta \)
2. while not done do
3. Sample batch of kernels \( \mathcal{P}_i \sim p(\mathcal{P}^{(train)}) \)
4. for all \( \mathcal{P}_i \) do
5. Sample \( K \) datapoints \( \mathcal{D} = \{X_j, Y_j\} \) from \( \mathcal{P}_i \)
6. Evaluate \( \nabla_{\theta} \mathcal{L}_{\mathcal{P}_i}(f_{\theta}) \) using \( \mathcal{D} \) and \( \mathcal{L}_{\mathcal{P}_i} \) in Equation 1
7. Compute adapted parameters with gradient descent: \( \theta'_i = \theta - \alpha \nabla_{\theta} \mathcal{L}_{\mathcal{P}_i}(f_{\theta}) \)
8. Sample datapoints \( \mathcal{D}'_i = \{x^{(j)}, y^{(j)}\} \) from \( \mathcal{P}_i \) for the meta-update
9. end for
10. Update \( \theta \leftarrow \theta - \beta \nabla_{\theta} \sum_{\mathcal{P}_i \sim p(\mathcal{P})} \mathcal{L}_{\mathcal{P}_i}(f_{\theta'_i}) \) using each \( \mathcal{D}'_i \) and \( \mathcal{L}_{\mathcal{P}_i} \) in Equation 1
11. end while

Only \( K \) labeled designs
Experimental Results – Offline Testing

- K=20 for adaption
- Adaptation is necessary for the unadapted model to obtain lower error
- FineTune: Naïve adaptation using the regular objective function
- Under 4 out of 5 kernels, MAML leads to a more accurate adapted model.

<table>
<thead>
<tr>
<th>Method</th>
<th>jacobi-1d</th>
<th>ftdt-2d</th>
<th>gemm</th>
<th>3mm</th>
<th>gemver</th>
</tr>
</thead>
<tbody>
<tr>
<td>GNN-Dse-Unadapted</td>
<td>4.2496</td>
<td>6.7047</td>
<td>7.5337</td>
<td>9.1584</td>
<td>4.4717</td>
</tr>
<tr>
<td>GNN-Dse-FineTune</td>
<td>3.2611</td>
<td>4.0831</td>
<td>1.7342</td>
<td>6.2930</td>
<td>3.1600</td>
</tr>
<tr>
<td>GNN-Dse-MAML</td>
<td>2.3898</td>
<td>2.4912</td>
<td>2.1116</td>
<td>5.9670</td>
<td>3.0303</td>
</tr>
</tbody>
</table>
Experimental Results – DSE

• MAML-based adaptation achieves great performance for 3 new kernels
  • 3mm: >17 trillion design candidates that AutoDSE got to explore only 149 of them after
  20 hours since it relies on the HLS tool for evaluating each candidate
  • GNN-DSE-MAML yields a significant speedup for 3mm compared to AutoDSE

<table>
<thead>
<tr>
<th>Method</th>
<th>jacobi-1d</th>
<th>fdtd-2d</th>
<th>gemm</th>
<th>3mm</th>
<th>gemver</th>
</tr>
</thead>
<tbody>
<tr>
<td>GNN-Dse-Unadapted</td>
<td>0.44×</td>
<td>0.06×</td>
<td>0.87×</td>
<td>0.30×</td>
<td>0.20×</td>
</tr>
<tr>
<td>GNN-Dse-FineTune</td>
<td>0.54×</td>
<td>0.04×</td>
<td>0.18×</td>
<td>1.00×</td>
<td>0.22×</td>
</tr>
<tr>
<td>GNN-Dse-MAML</td>
<td>1.00×</td>
<td>TO</td>
<td>1.21×</td>
<td>64.52×</td>
<td>TO</td>
</tr>
</tbody>
</table>

TO: Timed Out
Experimental Results – DSE

- For ftdt-2d and gemver, the MAML results lead to Timed Out
  - The MAML-based model uses high degree of parallelization for each section of the loop nests, overwhelming the HLS tool.
  - Such cases were not covered in the K sampled samples for adapting the model.

<table>
<thead>
<tr>
<th>Method</th>
<th>jacobi-1d</th>
<th>ftdt-2d</th>
<th>gemm</th>
<th>3mm</th>
<th>gemver</th>
</tr>
</thead>
<tbody>
<tr>
<td>GNN-Dse-Unadapted</td>
<td>0.44×</td>
<td>0.06×</td>
<td>0.87×</td>
<td>0.30×</td>
<td>0.20×</td>
</tr>
<tr>
<td>GNN-Dse-FineTune</td>
<td>0.54×</td>
<td>0.04×</td>
<td>0.18×</td>
<td>1.00×</td>
<td>0.22×</td>
</tr>
<tr>
<td>GNN-Dse-Maml</td>
<td>1.00×</td>
<td>TO</td>
<td>1.21×</td>
<td>64.52×</td>
<td>TO</td>
</tr>
</tbody>
</table>

TO: Timed Out
AutoDSE and GNN-DSE are Open-source

- [https://github.com/UCLA-VAST/GNN-DSE](https://github.com/UCLA-VAST/GNN-DSE)
- [https://github.com/UCLA-VAST/AutoDSE](https://github.com/UCLA-VAST/AutoDSE)
How to Integrate Different Approaches?

1. Architecture Guided Optimization: Based on common computation patterns
   - Systolic Array [DAC ‘17, ICCAD ‘18]
   - Stencil [ICCAD ‘18]

2. Apply ML or other optimization techniques for general applications (GNN-DSE) [DAC’22]

3. Compose the entire design using latency-insensitive dataflow task [FCCM’21 & FPGA’21 & 21]

Support domain specific languages
- Spark [DAC ‘18]
- Caffe [DAC ‘17]
- Halide [FPGA’20]
HeteroCL Programming Infrastructure [FPGA’19]

- Inspired by Halide: Separate program specification and optimization (scheduling)
  - Flexible: Mixed declarative & imperative programming
  - Portable: Clean decoupling of algorithm & hardware customizations
  - Efficient: Mapping to high-performance spatial architecture templates

Open-source: https://github.com/cornell-zhang/heterocl
**HeteroCL in a Nutshell**

**HeteroCL code**

```python
r = hcl.reduce_axis(0, 3)
c = hcl.reduce_axis(0, 3)
out = hcl.compute(N, N),
    lambda y, x:
        hcl.sum(image[x+r, y+c]*kernel[r, c],
            axis=[r, c])
```

**Corresponding C code**

```c
for (int y = 0; y < N; y++)
    for (int x = 0; x < N; x++)
        for (int r = 0; r < 3; r++)
            for (int c = 0; c < 3; c++)
                out[x, y] += image[x+r, y+c] * kernel[r, c]
```

**Algorithm**

```python
s = hcl.create_schedule()
s[out].unroll([r,c])
```

**Custom Compute**

```python
for i in range(2, 8):
    s.quantize(out, Fixed(i, i-2))
```

**Custom Data Type**

```python
linebuf = s[image].reuse_at(out, out.y)
winbuf = s[linebuf].reuse_at(out, out.x)
```

**Custom Memory**

HeteroCL: Mapping to Spatial Architecture Templates

- **Systolic Array**
  ```python
  # matrix multiply kernel
  out = hcl.compute(N, N,
    lambda y, x: sum(A[x, k] * B[k, y]), axis=k)
  s[out].systolic()
  ```

- **Stencil Architecture**
  ```python
  # jacobi kernel
  out = hcl.compute(N, N,
    lambda y, x:
      (in[y,x-1] + in[y-1,x] + in[y,x] + in[y,x+1] + in[y+1,x])/5)
  s[out].stencil()
  ```
One More Question:

Now I am good at using (enhanced) HLS, how to deal with (low) clock frequency and (long) compilation time from downstream physical synthesis?
Modern FPGAs are Large and Complex

- FPGAs are increasingly large
- Multiple dies integrated together
- High delay penalty for die-crossing
- Large IPs with pre-determined location
Modern FPGAs are Large and Complex

- FPGAs are increasingly large
- Multiple dies integrated together
- High delay penalty for die-crossing
- Large IPs with pre-determined location
Modern FPGAs are Large and Complex

- FPGAs are increasingly large
- Multiple dies integrated together
- High delay penalty for die-crossing
- Large IPs with pre-determined location

![Diagram showing die boundaries, DDR controllers, and peripheral IPs (e.g., PCIe)]

Xilinx Alveo U250
Xilinx Alveo U280
Modern FPGAs are Large and Complex

- FPGAs are increasingly large
- Multiple dies integrated together
- High delay penalty for die-crossing
- Large IPs with pre-determined location
Modern FPGAs are Large and Complex

- FPGAs are increasingly large
- Multiple dies integrated together
- High delay penalty for die-crossing
- Large IPs with pre-determined location
- HLS has limited consideration of those physical barriers
AutoBridge [FPGA'21 Best Paper Award]

- Add extra pipeline stages to long interconnects
- Couples floorplanning with HLS pipelining
- Global optimization to assure correctness
- Automate latency-insensitive design at the HLS level
- Improve average frequency from 150 MHz to 297 MHz over 43 test cases.

Successful Applications:
- [FPGA'21] AutoSA: A polyhedral compiler for high-performance systolic arrays on fpga
- [FPGA'22] Accelerating SSSP for Power-Law Graphs
- [FPGA'22] Sextans: A Streaming Accelerator for General-Purpose Sparse-Matrix Dense-Matrix Multiplication
- [DAC'22] Serpens: A High Bandwidth Memory Based Accelerator for General-Purpose Sparse Matrix-Vector Multiplication
- ......
Case Study

- Gaussian Elimination, 8 configurations

<table>
<thead>
<tr>
<th>Design</th>
<th>Default Frequency</th>
<th>Opt Frequency</th>
<th>Performance Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>24x24</td>
<td>223 MHz</td>
<td>334 MHz</td>
<td>1.4X</td>
</tr>
<tr>
<td>24x24</td>
<td>223 MHz</td>
<td>335 MHz</td>
<td>1.5X</td>
</tr>
</tbody>
</table>

- Difference in Resource Utilization
  - LUT: -0.14%
  - FF: -0.04%
  - BRAM: -0.03%
  - DSP: +0.00%
Latency-Insensitive Designs Helped Compile Time as Well!

Phase 1: Partitioning (Fully automated)

Input dataflow design in C/C++

Clock Source

Partition the design into islands
Insert anchor registers

Phase 2: Parallel Compilation (without an Overlay)

Islands and anchors placed & routed in parallel

Phase 3: Stitching

All islands stitched together & Inter-island routing

RapidStream [FPGA ‘22 Best Paper Award]
Experimental Result

- Tested on 6 large scale dataflow designs targeting Xilinx U250 FPGA with 4 SLRs (dies)
- Distribute to 4 Xeon servers, each with 56 cores
- Divide the FPGA into 32 islands (8 rows, 4 columns)
- 5-7X speedup (from C++ to fully routed checkpoint)
- Up to 1.3X frequency improvement
Use Overlay for Even Faster Compilation: OverGen [MICRO'22]

10,000x faster in re-compile

100,000x faster in reconf.
Composing Large Dataflow Designs Using TAPA

- TAPA programs explicitly decouple communication and computation
- Computation => compiled by Vitis HLS / AutoSA / AutoDSE / ...
- Communication => generated by TAPA
Example: FlexCNN Using TAPA

- FlexCNN: an end-to-end automated DNN synthesis framework
- From ONNX to bitsream on FPGAs

<table>
<thead>
<tr>
<th>Module</th>
<th>Lines of Code</th>
<th>Code Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reader 1</td>
<td>1,046</td>
<td>Template-based</td>
</tr>
<tr>
<td>Reader 2</td>
<td>446</td>
<td>Template-based</td>
</tr>
<tr>
<td>Systolic Array</td>
<td>4,801</td>
<td>Automatic</td>
</tr>
<tr>
<td>Pool</td>
<td>254</td>
<td>Template-based</td>
</tr>
<tr>
<td>Upsample</td>
<td>221</td>
<td>Template-based</td>
</tr>
<tr>
<td>Concat</td>
<td>350</td>
<td>Template-based</td>
</tr>
<tr>
<td>Add</td>
<td>314</td>
<td>Template-based</td>
</tr>
<tr>
<td>Act &amp; BN</td>
<td>320</td>
<td>Template-based</td>
</tr>
<tr>
<td>Writer</td>
<td>824</td>
<td>Template-based</td>
</tr>
<tr>
<td>Top</td>
<td>6,292</td>
<td>Automatic</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>14,868</strong></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>FlexCNN without TAPA</th>
<th>FlexCNN with TAPA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fails Placement &amp; Route</td>
<td>Achieves up to 266 MHz</td>
</tr>
</tbody>
</table>

Large dataflow design composed using TAPA
Concluding Remark 1

• I am encouraged by the progress/results on democratizing accelerator designs and customized computing
• It takes a community-wide effort
• All our tools are open-sourced, and FPGA vendors are more open as well
  • One-API from Intel
  • Merlin from AMD/Xilinx (after acquisition of Falcon Computing)
• Increasingly interested in using MLIR as an integration point
Concluding Remark 2

• Important for the architecture community to have a rapid prototyping flow
  • From Idea to Silicon in days, not months/years

• Concerned with some accelerator evaluation methodology
  • "We evaluate XXX using a C++-based cycle-level simulator.”
  • Does it consider
    • reduced memory bandwidth due to short burst length?
    • interconnect network size and latency from HBM ports to logic elements?
    • interconnect delays ...?
  • Has it been validated against any real silicon (FPGA or ASIC)?
Concluding Remark 3

• I had the pleasure working with many collaborators in other application domains.

• It’s time to enable domain experts to design their own accelerators!
• The deep learning community has done a much better job – "every" domain expert can train complex DL models
• Can we catch up? Think about broader impact!

Alex Bui and William Hsu
Low-dose CT reconstruction

Tad Blair
Real-time neural signal processing

Yizhou Sun
Graph similarity computation
Final Remark

- No doubt we are in an exciting era for computer architecture
- We want to every (serious) software programmer to participate
  - Not just architects
- Build his/her own customized accelerators on field-programmable fabrics
  - On premise or in the cloud
- I hope that many of you can join this effort
A Story ...

Q: Does everyone here do High-Level Synthesis?

A: What do you mean? We are all from Harvard Law School.
Acknowledgements:
NSF, JUMP/CRISP, and CDSC Industrial Partners

• Multi-year efforts by many students, postdocs, and collaborators

Prof. Tony Nowatzki (UCLA)
Prof. Yizhou Sun (UCLA)
Prof. Minyoung Kim (UCLA)
Prof. Zhiru Zhang (Cornell Univ.)
Prof. Peipei Zhou (Univ. of Pittsburgh)
Prof. Vivek Sarkar (Georgia Tech)

Yunsheng Bai (UCLA)
Jie Wang (UCLA)
Peng Wei (UCLA)
Hao Yu (UCLA/Falcon)
Yi-Hsiang Lai (Cornell)
Weikang Qiao (UCLA)
Yuan Zhou (Cornell)
Zhengrong Wang (UCLA)

Yuze Chi (UCLA)
Atefeh Sohrabizadeh (UCLA)
Sihao Liu (UCLA)
Zhe Chen (UCLA)
Jason Lau (UCLA)
Suhail Basalama (UCLA)
Licheng Guo (UCLA)
Jian Weng (UCLA)
Thank You!