Automatic Customizable Computing
— From DSLs to FPGAs for Deep Learning and Beyond

Jason Cong
Distinguished Chancellor’s Professor, UCLA
Director, Center for Domain-Specific Computing (CDSC)
http://cadlab.cs.ucla.edu/~cong

Supported by the IEEE CEDA Distinguished Lecture Program

Our Research Focus since 2008:
Customization for Energy Efficient Computing

Parallelization
Customization
Adapt the architecture to application domain

Power doubles every 4 years

Parallelization
Customization
Adapt the architecture to application domain

NSF Expeditions in Computing Award
IEE Design & Test, 2011
Potential of Customized Computing?

- Case study 1: AES Encryption [Schaumont et al., IEEE Computer 2003]
  - **AES 128bit key, 128bit data**
  - Throughput: 3.84 Gbits/sec, Power: 350 mW, Figure of Merit (Gbits/MW): 11 (1/11)
  - FPGA [1]: 1.32 Gbits/sec, 460 mW, 2.7 (1/2.7)
  - ARM StrongARM [2]: 31 Mbit/sec, 240 mW, 0.13 (1/0.13)
  - ARM Pentium III [3]: 648 Mbits/sec, 41.4 W, 0.015 (1/800)
  - C Emb, Sparc [4]: 133 Kbits/sec, 120 mW, 0.0011 (1/100,000)
  - Java (64-bit) Emb, Sparc [5]: 450 bits/sec, 120 mW, 0.0000037 (1/3,000,000)

- Case study 2: H.264 [Hameed et al., ISCA’2010]
  - Optimization using SIMD + VLIW → 10x energy efficiency
  - Customized instruction fusion → 1.6x energy efficiency
  - Still 50x away from ASICS – Not enough!

Energy Breakdown of Pipeline Components [DAC’14]

- Removing ‘Non-Computing’ Portions of the Pipeline
  - Remaining: ~ 10% + 26% = 36%

- Typical Superscalar OoO Pipeline
  - Parameters: 152
  - Instruction width: 5
  - Integer ALUs: 3
  - FPU ALUs: 3
  - ROB entries: 56
  - Integer ALUs: 64
  - L1 Cache: 128 KB, 8-way set associative
  - L2 Cache: 256 KB, 8-way set associative
  - L3 Cache: 1 MB, 8-way set associative
Overview of Our Approach -- Customized Computing with Accelerator-Rich Architectures

- Extensive use of accelerators at all levels of computing hierarchy
  - Most of them should be programmable or composable
  - Most computations are carried on accelerators – not on processors!
- A fundamental departure from von Neumann architecture
  - Datapath + control: general-purpose
- Why now?
  - Previous architectures are device/transistor limited
  - Von Neumann architecture allows maximum device reuse
    - One pipeline serves all functions, fully utilized
- Future architectures
  - Plenty of transistors, but power/energy limited (dark silicon)
  - Customization and specialization for maximum energy efficiency
- A story of specialization

Lessons from Nature: Human Brain and Advance of Civilization

- High power efficiency (20W) of human brain comes from specialization
  - Different region responsible for different functions
- Remarkable advancement of civilization also from specialization
  - More advanced societies have higher degree of specialization
Ways for Customization: ASIC Accelerators

◆ 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

◆ Problems:
  - too costly
  - take too much time to build

Google TPU: In-Datacenter Performance Analysis of a Tensor Processing Unit, ISCA 2017

---

Ways for Customization: FPGAs Accelerators

- FPGA: Field-Programmable Gate Arrays
- Island-style configurable mesh routing
- Configurable block
- Dedicated components
  - Specialization allows optimization
  - Memory/Multiplier
  - I/O, Processor
  - …
- Can be customized in seconds

- Focus of this talk

Examples of CPU-FPGA Platforms

FPGA-Enabled Computing Clusters at UCLA [DAC’16]

Developed in 2013

Boost Small-Core Performance

- 8 Xilinx ZC706 boards
- 24-port Ethernet switch
- ~100% power

Enhance High-Performance Clusters

1 master / driver
1 10GbE switch
22 workers
1 file server

Alpha Data board:
1. Virtex-7 FPGA
2. 16GB on-board RAM

Each node:
1. Two Xeon processors
2. One FPGA PCIe card (Alpha Data)
3. 64 GB RAM
4. 10GBE NIC
GPU and FPGA for Accelerated Computing on Amazon AWS

P2: GPU-accelerated computing
- Enabling a high degree of parallelism – each GPU has thousands of cores
- Consistent, well-documented set of APIs (CUDA, OpenACC, OpenCL)
- Supported by a wide variety of ISVs and open source frameworks

F1: FPGA-accelerated computing
- Massively parallel – each FPGA includes millions of parallel system logic cells
- Flexible – no fixed instruction set, can implement wide or narrow datapaths
- Programmable using available, cloud-based FPGA development tools

Microsoft Brain [MICRO ‘16]

- FPGAs integrated in datacenter network -- “bump-in-the-wire”
But ... How to Program Such Beasts?

From High-Level Programming Languages to Bits?

Logic Synthesis for FPGAs

- If you have a Boolean network ...
- Core logic synthesis:
  - Technology mapping
- FlowMap (ICCAD’92, TCAD’94):
  - Map any k-bounded fanin Boolean network to k-input LUTs with the minimum depth
  - Runs in polynomial time
C/C++ Based High-Level System (HLS)
xPilot (UCLA 2006) -> AutoPilot (AutoESL) -> Vivado HLS (Xilinx 2011)

How do you get a Boolean network?

- Platform-based C to RTL synthesis
- Synthesize pure ANSI-C and C++, GCC-compatible compilation flow
- 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
[T-CAD'2011, keynote paper]

AutoPilot Results: Sphere Decoder (from Xilinx)

- Wireless MIMO Sphere Decoder
  - ~4000 lines of C code
  - Xilinx Virtex-5 at 225MHz
- Compared to optimized IP
  - 11-31% better resource usage

<table>
<thead>
<tr>
<th>Metric</th>
<th>RTL Expert</th>
<th>AutoPilot Expert</th>
<th>Diff (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUTs</td>
<td>32,708</td>
<td>29,060</td>
<td>-11%</td>
</tr>
<tr>
<td>Registers</td>
<td>44,885</td>
<td>31,000</td>
<td>-31%</td>
</tr>
<tr>
<td>DSP48s</td>
<td>225</td>
<td>201</td>
<td>-11%</td>
</tr>
<tr>
<td>BRAMs</td>
<td>128</td>
<td>99</td>
<td>-26%</td>
</tr>
</tbody>
</table>

TCAD April 2011 (keynote paper)
"High-Level Synthesis for FPGAs: From Prototyping to Deployment"
## Vivado HLS is Widely Used

High-level synthesis of dynamic data structures: A case study using Vivado HLS

<table>
<thead>
<tr>
<th>Design</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>High-level synthesis of dynamic data structures: A case study using Vivado HLS</td>
<td>Develops a high-level description of dynamic data structures using Vivado HLS.</td>
</tr>
<tr>
<td>High-level synthesis of dynamic data structures: A case study using Vivado HLS</td>
<td>Demonstrates how Vivado HLS can be used to implement complex data structures efficiently.</td>
</tr>
</tbody>
</table>

This application showcases how Vivado HLS can simplify the development of complex data structures, making systems more efficient and scalable.

## However, Not All C Programs Lead to Good Performance

### Example: The Needleman-Wunsch algorithm for sequence alignment

```c
void engine(...) {
    int M[129][129];
    ...
    loop1: for(\text{i=0; \text{i<129; \text{i++}} \text{\{}} \text{M[0][i]=...}}
    loop2: for(\text{j=0; \text{j<129; \text{j++}} \text{\{}} \text{M[j][0]=...}}
    loop3: for(\text{i=1; \text{i<129; \text{i++}} \text{\{}} \text{for(\text{j=1; \text{j<129; \text{j++}} \text{\{}} \text{M[i][j]=...}}
    ...
}

void kernel(char seqAs[], char seqBs[], char alignedAs[], char alignedBs[]) {
    for (int i=0; i<NUM_PAIRS; i++) {
        engine(seqAs+i*128, seqBs+i*128,
               alignedAs+i*256, alignedBs+i*256);
    }
```
Inefficiency in Out-of-Box SDAs

void engine(...) {
    int M[129][129];
    ...
    loop1: for(i=0; i<129; i++) {M[0][i]=...}
    loop2: for(j=0; j<129; j++) {M[j][0]=...}
    loop3: for(i=1; i<129; i++) {
        for(j=1; j<129; j++) {...
            M[i][j]=...
        }
    }
    ...
}

void kernel(char seqAs[], char seqBs[], char alignedAs[], char alignedBs[]) {
    for (int i=0; i<NUMPAIRS; i++) {
        engine(seqAs+i*128, seqBs+i*128,
               alignedAs+i*256, alignedBs+i*256);
    }
}
Inefficiency in Out-of-Box SDAs

void engine(...) {
    int M[129][129];
    ...
    loop1: for(i=0; i<129; i++) {M[0][i]=...}
    loop2: for(j=0; j<129; j++) {M[j][0]=...}
    loop3: for(i=1; i<129; i++) {
            for(j=1; j<129; j++) {...
            M[i][j]=...}
    }
    ...
    }
void kernel(char seqAs[], char seqBs[],
            char alignedAs[], char alignedBs[])
    {
        for (int i=0; i<NUM_PAIRS; i++) {
            engine(seqAs+i*128, seqBs+i*128,
                   alignedAs+i*256, alignedBs+i*256);
        }
    }

Inefficiency in Out-of-Box SDAs

void engine(...) {
    int M[129][129];
    ...
    loop1: for(i=0; i<129; i++) {M[0][i]=...}
    loop2: for(j=0; j<129; j++) {M[j][0]=...}
    loop3: for(i=1; i<129; i++) {
            for(j=1; j<129; j++) {...
            M[i][j]=...}
    }
    ...
    }
void kernel(char seqAs[], char seqBs[],
            char alignedAs[], char alignedBs[])
    {
        for (int i=0; i<NUM_PAIRS; i++) {
            engine(seqAs+i*128, seqBs+i*128,
                   alignedAs+i*256, alignedBs+i*256);
        }
    }

Common Practice: Manual Code Reconstruction (Considering Micro-architecture Optimization)

Example of Manual Optimization

```c
void engine(char seqAs[], char seqBs[], char alignedAs[], char alignedBs[]) {
    for (int i=0; i<4; i++) {
        #pragma HLS unroll
        engine(seqAs+i*128, seqBs+i*128, alignedAs+i*256, alignedBs+i*256);
    }
}

void load(...) {...} // off-chip data load
void store(...) {...} // off-chip data store

void kernel(char seqAs[], char seqBs[], char alignedAs[], char alignedBs[]) {
    char seqAs_buf_x[128*TILE_PAIRS];
    char seqAs_buf_y[128*TILE_PAIRS];
    int num_tiles = NUM_PAIRS/4;
    for (int i=0; i<num_tiles+2; i++) {
        if (1 + 2 == 0) {
            load(...);
            engine(seqAs_buf_y, seqBs_buf_y, alignedAs_buf_y, alignedBs_buf_y)
            store(...);
        } else {
            load(...);
            engine(seqAs_buf_x, seqBs_buf_x, alignedAs_buf_x, alignedBs_buf_x)
            store(...);
        }
    }
}
```

10x speedup!
But ...

Not Everyone is Trained to Perform such Optimization

Overview of Our Current Research

- Support domain specific languages
  - Spark [DAC ’18]
  - Caffe [DAC ’17]
  - Halide (ongoing)

- Matched computation patterns: Apply the built-in architecture/IP
  - Systolic Array [DAC ’17, ICCAD ’18*]
  - Stencil [CCAD ’18*]

- Other patterns: Apply learning-based design space exploration
  - Use multi-armed bandit approach to organize several algorithms (in submission)

Goal: “Democratize” FPGA Accelerator Designs
Specialized Architecture #1: Systolic Array Compilation for Deep Learning and Beyond

Systolic Array Architecture (Kung-Leiserson, 1979)

- A network of locally interconnected processing elements (PE) that rhythmically process data at each time step.
  - High scalability
    - Modular design
  - High energy-efficiency
    - Near-neighbor communication
Systolic Arrays for Deep Learning

Example: Google TPUs

TPU(1st gen) block diagram

TPU(2nd gen) block diagram

What about FPGAs?

Strong Interest in FPGA Acceleration for Deep Learning

Total Citation: 469 (as of Aug 2018)

The first paper on FPGA acceleration for deep learning
Computation Model for Deep Learning

Challenge 1: Abstraction Gap for Deep Learning
**Challenge 2: Need of Automation for General Systolic Array Compilation**

- Programming efforts
  - Highly non-trivial
  - Goal: Several months -> several hours
- Optimization objectives
  - Latency/throughput
  - Area
  - Energy efficiency
- Portability
  - To be future-proof
  - Connect to other compilation systems, s.t.
    - TVM, Tensor Comprehension, HeteroCL

---

**PolySA: Polyhedral-based Systolic Array Architecture Compilation Framework [ICCAD’18]**

End-to-end compilation flow
- Systematic and efficient DSE
- Comparable performance to manual designs
**Polyhedral Model**

- A framework for performing loop transformation
- **Loop representation**: using polyhedrons to achieve fine-grained representation of program
- **Loop transformation**: transforming loops by doing affine transformation on polyhedrons
- **Dependency test**: several mathematical methods for validating transformation on loop polyhedrons
- **Code generation**: generating transformed code from loop polyhedrons

```
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    C[i][j] = 0;
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    C[i][j] += A[i][k] * B[k][j];
```

Louis-Noël Pouchet. Polyhedral Compilation Foundations, lecture 1.1: Iteration domains and polyhedra

---

**Space-time Mapping for MM Example**

```
#pragma SA
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    C[i][j] = 0;
    for (int k = 0; k < N; k++)
      C[i][j] += A[i][k] * B[k][j];
#pragma endSA
```

C/C++  
Example code of MM

Iteration domain in Polyhedral IR
**Space-time Mapping for MM Example**

**Transformation example**

Processor assignment:

\[
\begin{bmatrix}
1 & 1 & 0 \\
1 & 0 & 1
\end{bmatrix} = \begin{bmatrix} T \end{bmatrix}
\]

Time assignment:

\[
\begin{cases}
\vec{d} = (0,0,1) \\
\vec{s} = (1,1,1)
\end{cases}
\]

Causality: \( \vec{d} \cdot \vec{d} > 0 \)

Availability: \( \vec{d} \cdot \vec{s} > 0 \)

**Different transformation generates different systolic arrays.**

---

**Architecture Optimization**

**Front-End**

- Additional Design Optimization
- Multi-projection
- Interior I/O Elimination
- Task Interleaving
- Array Partition
**Architecture Optimization**

- Additional Design Optimization
- Multi-projection

---

**Architecture Optimization**

- Additional Design Optimization
- Multi-projection
- Interior I/O Elimination

---
**Architecture Optimization**

- Additional Design Optimization
- Multi-projection
- Interior I/O Elimination
- Task Interleaving

![Diagram showing task queue and PE efficiency]

**Architecture Optimization**

- Additional Design Optimization
- Multi-projection
- Interior I/O Elimination
- Task Interleaving
- Array Partition

![Diagram showing array partition]
**Evaluation on MM**

- The compiler can generate designs cover state-of-the-art systolic arrays with various shape.

![Diagram showing different systolic array designs of MM example.](image)

**Evaluation on CNN**

Different systolic array designs of CNN example.

![Diagram showing different systolic array designs of CNN example.](image)
## Evaluation on MM and CNN

- Design Space Exploration
  - The latency and resource models are highly accurate with a relative error within 10% for all the designs.

  ![Design space of MM and CNN.](image)

- Combining all techniques together, the compiler can generate designs with comparable performance to manual designs.

### MM

<table>
<thead>
<tr>
<th>MM</th>
<th>Board</th>
<th>BRAM</th>
<th>DSP</th>
<th>FF</th>
<th>LUT</th>
<th>MHz</th>
<th>GFLOPs</th>
<th>Projected GFLOPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>PolySA</td>
<td>Xilinx VU9P</td>
<td>89%</td>
<td>89%</td>
<td>39%</td>
<td>49%</td>
<td>228.8</td>
<td>555.4</td>
<td>758.5</td>
</tr>
<tr>
<td>Baseline [Intel FPGA ’18]</td>
<td>Intel Arria 10</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>312.5</td>
<td>800.0</td>
<td></td>
</tr>
</tbody>
</table>

### CNN

<table>
<thead>
<tr>
<th>MM</th>
<th>Board</th>
<th>BRAM</th>
<th>DSP</th>
<th>FF</th>
<th>LUT</th>
<th>MHz</th>
<th>GFLOPs</th>
<th>Projected GFLOPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>PolySA</td>
<td>Xilinx VU9P</td>
<td>71%</td>
<td>89%</td>
<td>39%</td>
<td>49%</td>
<td>229.5</td>
<td>548.4</td>
<td>603.6</td>
</tr>
<tr>
<td>Baseline [UCLA DAC ’17]</td>
<td>Intel Arria 10</td>
<td>47%</td>
<td>81%</td>
<td>40%</td>
<td>59%</td>
<td>252.6</td>
<td>600.3</td>
<td></td>
</tr>
</tbody>
</table>
**Ongoing Progress**

- Extend the framework to support more applications
  - E.g., Cholesky, LU, QR, Smith-Waterman
- Support Intel platform
- Explore efficient algorithms for polyhedral transformation

**Specialized Architecture #2: Stencil**
Stencil Computation Has Demanding Memory Access Pattern

**Need of memory partitioning**

Cyclic partitioning [TODAES'11]

Unroll factor | Memory Banks | Initiation Interval |
---|---|---|
1 | 7 | 1 |
3 | 6 | 3 |

A

A

A

A

A

A

Bank 1
Bank 2
Bank N

Bank 2
Bank 1

Bank 3
Bank 4
Bank 5
Bank 6

Bank 3
Bank 4

Bank 5
Bank 6

Bank 7

N: Partition Factor

Size ≈ K, Bandwidth = N*p

Cyclic Partition

\[
\begin{align*}
A_1 & \rightarrow A_2 & \rightarrow A_3 & \rightarrow \cdots & \rightarrow A_{N+1} \\
A_{N+2} & \rightarrow A_{N+3} & \rightarrow \cdots & \rightarrow A_{2N+1} & \rightarrow \cdots \\
& & & \cdots & \\
& & & & & \cdots
\end{align*}
\]

Optimization for Stencil Computation [DAC’14 & SIGGRAPH’14]

**Non-uniform partitioning to avoid BRAM contention**

- Holds input data from the first access to the last access
- Achieves optimal size of reuse buffer
- Achieves optimal number of memory banks

void blur(float input [N][M], float output [N][M])
{
    int i, j;
    for (j = 1; j < N - 1; ++j)
        for (i = 1; i < M - 1; ++i)
            // Unroll factor=3
            // Memory Banks=6
            // Initiation Interval=3
            output[j][i] = (input[j-1][i] + input[j][i-1] + input[j][i] + input[j+1][i] + input[j][i+1]) / 5.f;
}

†: An Optimal Microarchitecture for Stencil Computation Acceleration Based on Non-Uniform Partitioning of Data Reuse Buffers, Cong et al., DAC’14
Optimization for Stencil Computation [DAC'14 & SIGGRAPH'14]

- Non-uniform partitioning to avoid BRAM contention

- Holds input data from the first access to the last access
- Achieves optimal size of reuse buffer†
- Achieves optimal number of memory banks†

Optimization for Stencil Computation [DAC'14 & SIGGRAPH'14]

- Non-uniform partitioning to avoid BRAM contention

- Holds input data from the first access to the last access
- Maximum data reuse
- Optimal size of reuse buffer†
- Optimal # memory banks†
Stencil with Optimized Dataflow Architecture [ICCAD’18]

- Microarchitecture: loop unrolling & non-uniform partitioning
  - Duplicate PEs at fine-grained level
  - High throughput kernel without BRAM contention
  - Fully streamlined input / output access
  - Achieves optimal number of data transfer
  - Achieves optimal size of reuse buffer

```c
void blur(float input[N][M],
  float output[N][M])
{
    for(int j = 1; j < N-1; ++j)
      for(int i = 1; i < M-1; ++i)
        #pragma HLS unroll factor=3
        output[j][i] =
            input[j-1][i] +
            input[j  ][i-1] +
            input[j  ][i  ] +
            input[j+1][i] +
            input[j+1][i  ] / 5.f;
}
```

Stencil with Optimized Dataflow Architecture [ICCAD’18]

- Microarchitecture: loop unrolling & non-uniform partitioning

```c
void blur(float input[N][M],
  float output[N][M])
{
    for(int j = 1; j < N-1; ++j)
      for(int i = 1; i < M-1; ++i)
        #pragma HLS unroll factor=3
        output[j][i] =
            input[j-1][i] +
            input[j  ][i-1] +
            input[j  ][i  ] +
            input[j+1][i] +
            input[j+1][i  ] / 5.f;
}
```
Stencil with Optimized Dataflow Architecture [under submission]

- Microarchitecture: loop unrolling & non-uniform partitioning

```c
void blur(float input[N][M],
    float output[N][M]) {
    for(int j = 1; j < M-1; ++j)
        for(int i = 1; i < N-1; ++i)
            output[i][j] = (input[i+1][j] + input[i-1][j] + input[i][j+1] + input[i][j-1] + input[i+1][j+1] + input[i-1][j-1] + input[i+1][j-1] + input[i-1][j+1]) / 8.f;
}
```

Stencil with Optimized Dataflow Architecture [ICCAD'18]

- Microarchitecture: loop unrolling & non-uniform partitioning

```c
void blur(float input[N][M],
    float output[N][M]) {
    for(int j = 1; j < M-1; ++j)
        for(int i = 1; i < N-1; ++i)
            output[i][j] = (input[i+1][j] + input[i-1][j] + input[i][j+1] + input[i][j-1] + input[i+1][j+1] + input[i-1][j-1] + input[i+1][j-1] + input[i-1][j+1]) / 8.f;
}
```
Stencil with Optimized Dataflow Architecture [ICCAD'18]

Microarchitecture: loop unrolling & non-uniform partitioning

```c
void blur(float input[N][M],
  float output[N][M])
{
  for(int j = 0; j < N-1; ++j)
  for(int h = 1; h < M-1; ++h)
    output[j+1][h+1] = (input[j][h+1] + input[j][h] + input[j][h-1]
                       + input[j+1][h] + input[j-1][h]) / 5.f;
}
```

Stencil with Optimized Dataflow Architecture [ICCAD'18]

Microarchitecture: loop unrolling & non-uniform partitioning

```c
void blur(float input[N][M],
  float output[N][M])
{
  for(int j = 0; j < N-1; ++j)
  for(int h = 1; h < M-1; ++h)
    output[j+1][h+1] = (input[j][h+1] + input[j][h] + input[j][h-1]
                       + input[j+1][h] + input[j-1][h]) / 5.f;
}
```

<table>
<thead>
<tr>
<th>Design</th>
<th># Banks</th>
<th># PEs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uniform partition</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>Uniform partition with unrolling</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>Non-uniform partition</td>
<td>2(2FFs)</td>
<td>1</td>
</tr>
<tr>
<td>Non-uniform partition with unrolling</td>
<td>6(2FFs)</td>
<td>3</td>
</tr>
</tbody>
</table>
Stencil with Optimized Dataflow Architecture [ICCAD'18]

- Automation framework
  - SODA DSL to bitstream
  - Provide an IR for higher-level language
- Dataflow architecture
  - High frequency
  - Accurate resource modeling

Model-driven design-space exploration
- One-pass HLS to get resource usage for each module
- Analytical model-based resource modeling
- Roofline model for throughput
- Fast exploration via branch-and-bound
- Finish in ~minutes

Stencil with Optimized Dataflow Architecture

- Automation framework
  - SODA DSL to bitstream
  - Provide an IR for higher-level language
  - Dataflow architecture
  - High frequency
  - Accurate resource modeling

- Model-driven design-space exploration
  - One-pass HLS to get resource usage for each module
  - Analytical model-based resource modeling
  - Roofline model for throughput
  - Fast exploration via branch-and-bound
  - Finish in ~minutes
**Stencil with Optimized Dataflow Architecture** [ICCAD’18]

- **Model prediction**

<table>
<thead>
<tr>
<th>Prediction Item</th>
<th>BRAM</th>
<th>DSP</th>
<th>LUT</th>
<th>FF</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ave. Error</td>
<td>1.84%</td>
<td>0%</td>
<td>6.23%</td>
<td>7.58%</td>
<td>4.22%</td>
</tr>
</tbody>
</table>

- **Performance** (Jacobi2D, float)

<table>
<thead>
<tr>
<th>Platform</th>
<th>Alpha Data</th>
<th>AWS F1</th>
<th>Xeon (12c24t)</th>
<th>Titan (PACT’16)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max. Bw. (GB/s)</td>
<td>10.7</td>
<td>~48</td>
<td>136.6</td>
<td>288</td>
</tr>
<tr>
<td>Achieved Bw. (GB/s)</td>
<td>9.1</td>
<td>42.3</td>
<td>43.9</td>
<td>63</td>
</tr>
<tr>
<td>Bw. Util.</td>
<td>85%</td>
<td>88%</td>
<td>32.1%</td>
<td>22%</td>
</tr>
<tr>
<td>GFlops</td>
<td>70</td>
<td>328</td>
<td>341</td>
<td>488</td>
</tr>
<tr>
<td>TDP (W)</td>
<td>25</td>
<td>—</td>
<td>170</td>
<td>250</td>
</tr>
<tr>
<td>Energy (Norm.)</td>
<td>1</td>
<td>—</td>
<td>0.71</td>
<td>0.69</td>
</tr>
</tbody>
</table>

PACT’16: Resource-Conscious Reuse-Driven Tiling for GPUs

**Specialized Architecture #3:**
Composable Parallel and Pipeline Architecture (CPP)
Customizable, Parallel, Pipeline (CPP) Arch.

- FPGA Fabric
- NW Accelerator
  - Load
    - seqAs_buf
    - seqBs_buf
  - Compute
    - Engine
    - Loop1_Parallel
    - Loop2_Parallel
    - Loop3_Pipeline
  - Store
    - alignedAs_buf
    - alignedBs_buf
- Device DRAM
  - seqAs
  - seqBs
  - alignedAs
  - alignedBs

Overlap Communication and Computation
High off-chip bandwidth
Customizable, Parallel, Pipeline Arch.

FPGA Fabric

NW Accelerator

Load

seqAs_buf

seqBs_buf

Compute

Engine

alignedAs_buf

alignedBs_buf

On-chip buffer for data caching
Partitioned for parallel data supply

Partition #1

Partition #2

Partition #N

Engine

Engine

Partition #1

Partition #2

Partition #N

All loops are mapped to parallel or pipeline circuits

Device DRAM

seqAs

seqBs

alignedAs

alignedBs
• Benefits:

A well-defined design space,
Enable analytical models for fast exploration.

AutoAccel Framework

✦ A C-to-FPGA compilation framework
  • Input: A C program (accelerator kernel)
  • Output: An accelerator design w. the CPP architecture
**Analytical Design Space Exploration**

- A fast accurate analytical model for the CPP arch
  - Initialized by the HLS reports and kernel analysis
    - Reflect any optimization done by commercial HLS tool
  - >99% (perf.) and >92.5% (resource) compared to Vivado HLS
  - ~0.2 sec per design point

---

**Model Establishment**

```c
void foo(...) {
    // singleton statements
    ...
    // a loop
    loop_1:
    for (int i=0; i<128; i++) {
        #pragma HLS pipeline
        ...
    }
    // a subfunction
    bar(...)
}
```

<table>
<thead>
<tr>
<th>Func Name</th>
<th>Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>500</td>
</tr>
<tr>
<td>bar</td>
<td>350</td>
</tr>
</tbody>
</table>

First, we run Vivado HLS once to establish the model

<table>
<thead>
<tr>
<th>Loop Name</th>
<th>Depth</th>
<th>II</th>
<th>Trip Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop_1</td>
<td>16</td>
<td>1</td>
<td>128</td>
</tr>
</tbody>
</table>

By parsing the HLS report, we obtain a number of values of the parameters

Next, we explore the possible unroll factor of Loop_1

- Cycle_foo = Cycle_bar + Cycle_loop + Cycle_singleton
- Cycle_loop = Depth + II * Trip_count – 1

II′ = 1
Trip_count = 128
Cycle_singleton = 7
Model Establishment

```c
void foo(...) {
    // singleton statements
    ...
    // a loop
    loop_1:
    for (int i=0; i<128; i++) {
        #pragma HLS unroll factor=X
        #pragma HLS pipeline
        ...
    }
    // a subfunction
    bar(...)
}
```

With the already obtained model parameters, we do not need to run Vivado HLS any more for different unroll factor X

\[
\text{Cycle}_{\text{bar}} = \text{Depth} + \text{II} \times \text{Trip\_count} - 1 = 15 + 128 / X
\]

Cycle bar can be recursively modeled in the same way

Finally, we obtain the relationship between the cycle count and the design parameters (unroll factor in this simple case)

Evaluation Results

- MachSuite benchmarks
- Baseline: An Intel Xeon CPU core
- FPGA: AlphaData board (Virtex-7) w. 16G RAM

Design space exploration
  - Converged within 3 mins
  - Bandwidth bound:
    - AES, SPMV, KMP, STENCIL
  - BRAM bound:
    - FFT, GEMM
  - LUT bound:
    - VITERBI, NW

Performance improvement
  - 27,000x over OoB-SDA
  - 27x over an Xeon CPU core

Energy efficiency gain
  - 1678x over OoB-SDA
  - 260x over an Xeon CPU core
Solution #2: A Learning-based Design Space Exploration Framework

Machine Learning is Making Great Progress!

**Learning-based Design Space Exploration Framework**

```plaintext
#pragma ACCEL parallel
factor=auto1(1..128)
#pragma ACCEL pipeline auto2(on,off)
for (let i = 0; i < N; i++) {
  ...
}
auto1={1,2,3,...,128}
auto2={on, off}
```

**Build Design Space using The Merlin Compiler**

- **Overview**
  - Provide a small set of pragmas to guide source-level code transformation
  - A much smaller design space compared to HLS pragmas

<table>
<thead>
<tr>
<th>Merlin Supported Opt.</th>
<th>VivadoHLS</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRAM buffer bit-width</td>
<td>+</td>
<td>#pragma ACCEL var=data bus_bitwidth=512</td>
</tr>
<tr>
<td>Coarse-grained parallel</td>
<td>-</td>
<td>#pragma ACCEL parallel factor=4</td>
</tr>
<tr>
<td>Fine-grained parallel</td>
<td>++</td>
<td>#pragma ACCEL parallel factor=4</td>
</tr>
<tr>
<td>Coarse-grained pipeline</td>
<td>-</td>
<td>#pragma ACCEL pipeline</td>
</tr>
<tr>
<td>Fine-grained pipeline</td>
<td>++</td>
<td>#pragma ACCEL pipeline flatten</td>
</tr>
</tbody>
</table>
ML Toolkit: OpenTuner: An Open-Source Auto-Tuner

- An extensible open source auto-tuner for software compilers by MIT
  - J. Ansel et al, OpenTuner: An Extensible Framework for Program Autotuning, PACT, 2014
- Leverage multi-arms bandit to make use of a set of reinforcement learning algorithms
  - Prioritize a learning algorithm which predicts better design points
- Has been used for other design flows
  - C. Xu et al, A Parallel Bandit-Based Approach for Autotuning FPGA Compilation, FPGA ’17
- Overview

![Diagram of OpenTuner framework]

- Example: Tuning gcc flags

Overview

my.cpp

OpenTuner

The best gcc config

g++ <selected config> my.cpp
time ./a.out

However, HW Design is A Lot More Complex ...

- Simply leverage OpenTuner does not give us good performance
- Impediment 1: Expensive evaluation approach
  - We set the total exploration time to 4 hours
  - One HLS run takes about 10 minutes
  - 240/10 = 24 evaluated design points → Too few to be explored!
- Impediment 2: Design space is complex
  - Design space parameters have dependency to each other
    - e.g. Pipeline flatten disables all design factors inside
  - Design space is non-smooth
    - e.g. Parallel factor=1 and 2 have non-linear behavior → Hard to learn!
Our Approach

- Intelligent search
  - Smart heuristics (e.g. gradient guided search)
  - Combined with reinforcement based machine learning
- Efficient and accurate estimation models
  - Performance
  - Resource usage (BRAM, DSP, LUT, FF, etc.)
  - Both are needed for gradient computation

Evaluations on Amazon EC2 F1 Instance

- Machsuite

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Out-of-box Xilinx SDx (ms)</th>
<th>Manual Speedup</th>
<th>Our Speedup</th>
<th>Kernel</th>
<th>Out-of-box Xilinx SDx (ms)</th>
<th>Manual Speedup</th>
<th>Our Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>13820</td>
<td>812x</td>
<td>812x</td>
<td>VIBERBl</td>
<td>11</td>
<td>1.43x</td>
<td>1.43x</td>
</tr>
<tr>
<td>GEMM</td>
<td>17</td>
<td>117x</td>
<td>117x</td>
<td>BACKPROP</td>
<td>268</td>
<td>3.70x</td>
<td>3.45x</td>
</tr>
<tr>
<td>KMP</td>
<td>4870</td>
<td>39x</td>
<td>39x</td>
<td>BFS</td>
<td>135</td>
<td>20x</td>
<td>20x</td>
</tr>
<tr>
<td>NW</td>
<td>9130</td>
<td>1702x</td>
<td>1702x</td>
<td>FFT</td>
<td>21</td>
<td>52x</td>
<td>52x</td>
</tr>
<tr>
<td>SPMV</td>
<td>1100</td>
<td>13x</td>
<td>13x</td>
<td>MD</td>
<td>2716</td>
<td>14x</td>
<td>14x</td>
</tr>
<tr>
<td>STENCIL</td>
<td>59</td>
<td>122x</td>
<td>122x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Financial modeling

<table>
<thead>
<tr>
<th>Example Designs</th>
<th>CPU time (ms)</th>
<th>Out-of-box Xilinx SDx (ms)</th>
<th>Manual Speedup (ms)</th>
<th>Our Speedup (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Black Scholes Asian</td>
<td>32399</td>
<td>34430</td>
<td>830 (39X)</td>
<td>910 (10X)</td>
</tr>
<tr>
<td>Black Scholes European</td>
<td>8542</td>
<td>8590</td>
<td>230 (37X)</td>
<td>270 (31X)</td>
</tr>
<tr>
<td>Heston European</td>
<td>22907</td>
<td>34430</td>
<td>1530 (15X)</td>
<td>1450 (15X)</td>
</tr>
<tr>
<td>Heston European Barrier</td>
<td>2754</td>
<td>17220</td>
<td>750 (4X)</td>
<td>720 (4X)</td>
</tr>
</tbody>
</table>

Results achieved by Falcon Computing: https://www.falcon-computing.com
**One Application: Apache Spark Acceleration**

- Popular open-source big data frameworks are implemented in Java/Scala
- It is impractical to accept user pragmas in the Java/Scala source code
  - Domain knowledge and semantic mapping
- Apply the learning-based DSE to optimize the generated C kernel
- Accelerate ML kernels by 181.5x speedup over the baseline, 85% of the manual designs

Spark application in Scala

```java
rdd.map((seqA, seqB) => nw(seqA, seqB))
```

Functional equivalent C kernel

```c
void kernel(int N, char seqA[], char seqB[], char out1[], char out2[])
{
    for (int i=0; i<N; i++)
        nw(seqA+i*128, seqB+i*128, out1+i*256, out2+i*256);
}
```

500x slow down to a single core CPU!

---

**S2FA: A Spark-to-FPGA Accelerator Framework [DAC’18]**

![Diagram of S2FA framework]

Blaze application (Scala) -> Java Compiler -> Bytcode-to-C compiler -> Kernel (C code)

Data Methods (Bytecode) -> Method Generator

Blaze application (Bytecode) -> Data layout (Config)

Accelerator (Bit-stream) -> Finalization

AutoDSE

- Decision Tree Table
- Design Space Identification
- Design Space Partitioning
- Scheduling

Design Point Allocation (Multi-Armed Bandit)

Learning Algorithm N

- Design Point Selection
- Local Design Point Set
- HLS Tool

OpenTuner
FPGA Acceleration Results from UCLA/CDSC

Simulation
Big-Data
Neural Network
Medical Applications

Accelerate Neural Networks
• CNN
  • 4.8x speed-up
  • 24.6x energy-efficient

Accelerate Neural Networks
• LSTM-RNN
  • 215x energy-efficient

Accelerate Compression
• Deflate
  • 39x speed-up

Accelerate Image Reconstruction
• 3D CT Image
  • 26.9x speed-up

Accelerate DNA Read Aligner
• Smith-Waterman
  • 26.4x speed-up

Accelerate Sorting
• Priority Queue
  • 4.3x speed-up
  • 21.5x energy-efficient

Accelerate Simulation
Big-Data
Neural Network
Medical Applications

Summary – Report Card

Good progress, a lot more to be done!

Support domain specific languages
- Spark [DAC ’18]
- Caffe [DAC ’17]
- Halide (ongoing)

Matched computation patterns: Apply the built-in architecture/IP
- Systolic Array [DAC ’17, ICCAD ’18*]
- Composable, Parallel and Pipeline (CPP) [DAC ’18]
- Stencil [ICCAD ’18*]

Other patterns: Apply learning-based design space exploration
Use multi-armed bandit approach to organize several algorithms (in submission)

Goal: You innovate (in algorithm, application ...),
we automate (compiling to customized hardware)
Acknowledgements:
NSF, CFAR, CRISP, and CDSC Industrial Partners

Multi-year Efforts by Students, Postdocs, and Collaborators