FPGA Based Acceleration for Big Data Processing
---- Challenges and Opportunities

Wei Zhang
Hong Kong University of Science and Technology

Driving Strength behind Big Data

Moore’s Law Coming to An End

The Trend in Computing Platform
- From general to customized
- From CPU to GPU, FPGA, and ASIC
- Better user experience, low cost

Technology Scaling
- 2012: 28nm
- 2018: 7nm

Error Rate on ImageNet
- 2010: SIFT+SVM ~25%
- 2012: Deep CNN ~15%
- 2013: ~11%
- 2014: ~6%
- 2015: <5%

2018: 7nm
2012: 28nm

Moore’s Law Coming to An End

The Evolution of FPGA

Generations of FPGA devices

FPGA Binaries (Bitfile)
Performance
- 405 Cores 300+ MHz 130nm
- 450 Cores 450+ MHz 90nm
- Dual Cortex-A9 Core 1.5GHz 5000 DMIPS 16nm
- Dual Cortex-R5 Core 600 MHz 13000+ DMIPS 16nm
Diversity of Accelerators

CPU
- ✔ High flexibility
- × Sequential Execution
- × Low efficiency

GPU
- ✔ High throughput
- × High power
- × Rely on batch

ASIC
- ✔ Low power
- ✔ High performance
- × Low flexibility after tape out

FPGA has overall advantages
- Low latency
- Low power
- ✔ Programmability
- ✔ Short time-to-market

FPGA based Acceleration in the Cloud and Edge

Alibaba: Database Acceleration
- X Engine
- 40% of throughput performance

Microsoft: Bing Search Engine
- Bing page ranking service
- 2x the throughput with lower latency

Amazon (AWS): DNAnexus & Edico Genome
- FPGA-based EC2 F1 instances
- Process a whole genome sequence in 1 hour

Cloud Computing
Edge Computing

Autonomous Vehicles
Smart home
VR/AR
Personal healthcare
Chat Bot
Professional assistants
Current Status of FPGA based Accelerator

Challenges: Design flow

RTL Design with HDL

- Hard to debug
- Long time development
- Hard to tune parameter
- Hardware background

Alibaba Software Framework

- Friendly to software designer
- Fast design space exploration
- Maintain high performance

- User Model
- AI Framework
- Computational Graph Optimization
- Run-time
- Heterogeneous Pooling Standard Interface: API + IR
- Device Driver/Runtime Library
- GPU
- AI ASIC
- FPGA
- Others

- Performance Monitoring
- Conversation Management
- Resource Management
- Event Interrupt

- Profiler Debugger
- TCP/RDMA
- Function Debugging
Challenge: Architecture

Which one is the best for your application?

Solution?

- Design flow challenge
  - Automatic synthesis flow from high-level language
  - Fast design space exploration to optimize the design
  - Domain-specific design flow

- Architecture challenge
  - Simulator to evaluate different CPU-FPGA architectures
  - Cache management optimization for shared cache CPU-FPGA architecture

Automation can dramatically improve the design productivity

C/C++ Application targeting SoC
High Level Synthesis

- **Programmability** of FPGA is significantly improved!
- **Lower the barrier** for software designer
- Selection of *directives/pragmas* is challenging.

Pragmas Setting

**Loop Unroll**
```
int i; for (i = 0; i < 100; i += 5)
    {a[i]=b[i]+c[i];
     a[i+1]=b[i+1]+c[i+1];
     a[i+2]=b[i+2]+c[i+2];
     a[i+3]=b[i+3]+c[i+3];
     a[i+4]=b[i+4]+c[i+4];}
```

**Loop Pipelining**
```
void func (m, n, o) {
    for (i=2, i>-0;i--)
        op_Read;
        op_Compute;
        op_Write:
    }
```

**Array Partition**
```
0  1  2  ... N-3  N-2  N-1
```

Fig. No loop pipelining

3 cycles

9 cycles

5 cycles
Large Design Space

- More than millions of combinations.
- The performance difference can be very large!

**Important:**
To choose the best configuration.

**Difficult:**
To quickly find the best one.

- For this application, COMBA spends 10 min to find the high-performance configuration, in a design space with $7.61 \times 10^{12}$ points.

Framework Overview

COMBA iterates until it finds the high-performance configuration. [ICCAD’17]
Analytical Models

- Performance Models
  - Loop unrolling
  - Loop pipelining
  - Array partitioning
  - Function pipelining
  - Dataflow

- Resource Models
  - DSP
  - BRAM

\[
C_{L_k} = C_{L_{k+1}} + \frac{B_{k+1}}{U_{k+1}} \cdot U_k + C_{L_k} \\
\text{Cycle}_{L} = D + P \cdot (T_C - 1) \\
P = P_1 + \sum_{i=2}^{n} P_i \cdot \prod_{k=1}^{i-1} f_k \\
I_2 = \max \left( I_{\min}^{\text{sub}}, I_{\max}^{\text{sub}} \right) \\
I_L = I_{\max}^{\text{sub}} = \max_i \left( I_{i}^{\text{sub}} \right)
\]

- Evaluation Metrics
  - Find the bottleneck: the longest sub-function/sub-loop is assumed to have the greatest influence.
  - Check the resource constraints: whether the resource usage exceeds the available resources on FPGA.
  - Decide how to partition: which option is better, block or cyclic.

\[
M_{\text{diff}} = C_{\text{sub}}^{\text{max}} - C_{\text{sub}}^{\text{max}} \\
M_{\text{res}} = \max \left( \frac{\text{DSP}_{\text{used}}}{\text{DSP}_{\text{total}}}, \frac{\text{BRAM}_{\text{used}}}{\text{BRAM}_{\text{total}}} \right) \\
M_{\text{aps},l} = \frac{\#\text{loads}}{\max_{i,k} (\text{index}_i^d - \text{index}_i^k + 1)} \\
M_{\text{aps},s} = \frac{\#\text{stores}}{\max_{i,k} (\text{index}_i^d - \text{index}_i^k + 1)}
\]
Metric-Guided DSE (MGDSE)

Experimental Results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Design Space</th>
<th>Performance Speed-up</th>
<th>MGDSE Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATAX</td>
<td>85</td>
<td>1.31 × 10^8</td>
<td>8.35</td>
</tr>
<tr>
<td>BICG</td>
<td>95</td>
<td>5.76 × 10^8</td>
<td>15.88</td>
</tr>
<tr>
<td>GEMM</td>
<td>85</td>
<td>1.05 × 10^10</td>
<td>8.15</td>
</tr>
<tr>
<td>GESUMMV</td>
<td>85</td>
<td>8.39 × 10^8</td>
<td>15.42</td>
</tr>
<tr>
<td>MM</td>
<td>85</td>
<td>6.34 × 10^13</td>
<td>15.22</td>
</tr>
<tr>
<td>MVT</td>
<td>95</td>
<td>1.05 × 10^10</td>
<td>15.30</td>
</tr>
<tr>
<td>SYR2K</td>
<td>85</td>
<td>1.05 × 10^10</td>
<td>7.27</td>
</tr>
<tr>
<td>SYRK</td>
<td>85</td>
<td>1.64 × 10^8</td>
<td>8.12</td>
</tr>
</tbody>
</table>

Larger design space  Better speed-up  Efficient design space exploration
Case Study: CNN

Performance Pitfall

- Different board resource constraints
- Easily memory bounded
- Hard to fully utilize on-chip resource
- Always not enough on-chip cache

for \( m = 0; m < M; m++ \)
for \( y = 0; y < Y; y++ \)
for \( x = 0; x < X; x++ \)
for \( n = 0; n < N; n++ \)
for \( i = 0; j < K; j++ \)
for \( i = 0; i < K; i++ \)

\[ O(m)[x][y] = W[m][n][j][i] \times I[n][Sx + j][Sy + i]; \]

Loop Tree

Standard Input-Output channel tiling
Performance Analysis Framework for OpenCL CNN

- LoopTrees automatically generated to form a comprehensive CNN design space
- Selecting candidate CNN Designs using Coarse-grained model
- Using Fine-grained model and performance advisor to identify the design bottleneck and fine-tune the design.

Domain-Specific Computing Flow

How about the source code not efficient?

General C/C++ code ➔ Gap ➔ Domain-specific templates

Not hardware-friendly
- Many pointer usage
- Complicated loop
- General data structure
- No dataflow processing
- No memory management
- Standard data type

High-performance
- Execution in parallel
- Efficient data structure
- Optimal pragma configuration
- Dataflow processing
- Efficient memory management
- Optimized data width

Fill in the gap: domain-specific design flow
Domain-Specific Computing Flow

Domain: DNN

[Diagram showing the Domain-Specific Computing Flow process]

FP-DNN [1]

[Diagram for FP-DNN]

DNNBuilder [2]

[Diagram for DNNBuilder]

Case Study: Computer Vision

Stereo Vision: to compute “depth” in the image.

- Finding corresponding pixels in two images from two cameras, compute the depth information (disparity) of each pixel.
- One of the most important applications in computer vision area.
- Widely used in autonomous driving, 3D reconstruction and robotics, etc.
- Requirements: real-time and high-accuracy.

Stereo Vision System: based on semi-global stereo matching (SGBM) algorithm.
Case Study: Computer Vision

FP-Stereo: an FPGA-friendly C++ library for stereo vision.
- Hardware-efficient templates for stereo matching and post-processing.
- Automatic code generation to generate a high-performance stereo vision pipeline.

Previous implementations (xfOpenCV):
1) Cost function
   - Census transform & hamming distance
2) Cost Aggregation
   - 4 paths
3) Disparity Estimation: winner-takes-all (WTA)
4) Without post-processing

Components (templates) of FP-Stereo:
1) Cost functions
   - Census transform and hamming distance
   - SAD: sum of absolute differences
   - ZSAD: zero-mean sum of absolute differences
   - Rank transform
2) Cost Aggregation
   - 4 paths
   - 5 paths
   - 8 paths
3) Disparity Estimation: winner-takes-all (WTA)
4) Post-processing
   - Consistency check: left-right computation
   - Consistency check: left computation
   - Median filter
   - Uniqueness check
   - Gap/hole filling

Case Study: Computer Vision

FP-Stereo: an FPGA-friendly C++ library for stereo vision.

Optimization techniques for each component:
- Loop pipelining
- Array partitioning
- Resource allocation
- Data packing
- Automatically optimized data width

Code Generation of Efficient Pipeline:
- Concurrent processing with Dataflow.
- FIFO initialization with optimized depth and data size between each module.
=> Each function can execute as soon as data is available.

High-performance pipeline!
Case Study: Computer Vision

FP-Stereo: an FPGA-friendly C++ library for stereo vision.

Comparison:

<table>
<thead>
<tr>
<th></th>
<th>Latency</th>
<th>Frame Per Second</th>
<th>Accuracy</th>
<th>Power</th>
<th>Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>[2]</td>
<td>0.11s</td>
<td>9.1 FPS</td>
<td>6.38%</td>
<td>NA</td>
<td>Nvidia GTX 980</td>
</tr>
<tr>
<td>[3]</td>
<td>0.0128s</td>
<td>78 FPS</td>
<td>8.24%</td>
<td>39W</td>
<td>Nvidia GTX Titan X</td>
</tr>
<tr>
<td>Ours</td>
<td>0.0178s</td>
<td>56 FPS</td>
<td>7.74%</td>
<td>&lt;5W</td>
<td>Xilinx ZCU102 MPSoC</td>
</tr>
</tbody>
</table>

- Compared to [2], FP-Stereo contains **hardware-friendly** implementation. **Low latency**
- Compared to [3], FP-Stereo utilizes more **robust** cost function. **High accuracy**
- Compared to GPU implementations, FP-Stereo costs **less power**. **Power efficiency**

Dataset: KITTI

Various CPU-FPGA System

- **Architectural Exploration**
  - Different CPU-FPGA architectures → comparison & analysis
  - Architectural choice & design

- **Emerging shared cache CPU-FPGA systems**
  - Bottleneck → memory access
  - Heavily rely on the cache hierarchy
  - Cache utilization
    - Cache hit maximization

PAAS: Simulator for CPU-Accelerator System

- **Fast architectural exploration**
  - Number of cores
  - Number of accelerated kernels on FPGA
  - Different memory/cache hierarchies between CPU and FPGA
  - Different on-chip network, such as Bus and NoC

- **A full system simulator**
  - Based on Gem5 and Verilator
  - Verified with Zynq SoC, error less than 5%.
Architectural Exploration

Enhanced communication efficiency with shared cache
- L1/L2 mode vs Mem mode
- 170X vs 100X compared to SW mode
- Reduce communication overhead 55%~70%

Shared cache
- No explicit data movement
- No explicit coherence maintenance
- Closer access with less latency
- Programming model evolved
- Flexible and quick access
- Better cache management desired

<table>
<thead>
<tr>
<th></th>
<th>FPGA part (L1)</th>
<th>FPGA part (L2)</th>
<th>FPGA part (Mem)</th>
<th>Whole (L1)</th>
<th>Whole (L2)</th>
<th>Whole (Mem)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average speedup</td>
<td>168.73</td>
<td>164.27</td>
<td>98.74</td>
<td>4.11</td>
<td>4.03</td>
<td>3.81</td>
</tr>
</tbody>
</table>

Hybrid Cache Bypassing Scheme

**Advantage**
- Simply the programming model
- Reduce the communication delay

**Challenge**
- Competition on FPGA cache
- Large overhead on cache miss

Hybrid cache bypassing scheme
- Compile-time analysis + Run-time control
- Compile-time: reuse distance analysis + provide hints for run-time bypassing
- Run-time: perform bypassing according to compile-time hints
Static Classification and Dynamic Bypassing

Each data block → 3 types for bypassing preferences
- **H-type**: caching
  Small level distances (high locality level)
- **L-type**: bypassing
  Large level distances (low locality level)
- **M-type**: both possible, decided at run-time
  Medium level distances (medium locality level)

Comparison to Other methods

Experiment Settings:
- Benchmarks based on Polybench
- Simulation: PAAS[1], a Gem5 based cycle accurate CPU-FPGA simulator
- FPGA cache: 64KB, direct-mapped

Compared with:
- PDP[2]: protecting distance-based bypassing
- Baseline: no bypassing

Average:
- Miss rate reduced by 33.5% (PDP: 9%)
- Execution time reduced by 24% (PDP: 7%)

Advantage:
- Distinguish data with different patterns

---

**Intel HARP2**

- Three links on FPGA
  - VL0 → QPI with FPGA cache
  - VH0/VH1 → PCIE
- Memory access management
  - Which link for each request?
- Consideration
  - Cache reuse → Maximize cache benefits
    - Highly reused data to VL0
    - Data which may pollute cache to VH0/VH1
  - Link balance → Fully utilize all 3 links
    - Don’t let any link idle

---

**LAMA: Link-Aware Run-time Management on HARP2**

- Monitor and balance the link usage
  - H-type → always VL0
  - M-type and L-type → assign according to the link balance
- Maintain the data coherence
  - Wrfence -> force data coherence
  - Solve RAW/WAW hazard
  - Compile-time algorithm -> where to insert Wrfence
  - Run-time management
Compared with HARP Optimization Methods

- 33.9% speedup respectively Compared to HARPdyn
  - LAMA: link selection adapts to the real-time link utilization, cache reuse benefits boosted.
  - HARPdyn
    - Fix link for every same data block within one period
    - Change such fixed link mapping periodically

- Low overhead for both latency and resource
  - Run-time latency: within 2 cycles even at 400MHz, the highest frequency in HARP v2.
  - Resource: 3220 ALMs, 1932 registers and 13312 block memory bits (<1%)

Software-Hardware Partition for Complete Flow

- Partition algorithm based on ILP, genetic algorithm, etc
- Software-hardware partition considering the modeling of the communication interface

Automation can dramatically improve the design productivity
Summary

• An exciting time for realizing the true potential of FPGA based accelerator!

• Challenges to be overcome in both design flow and architecture optimization

• An overall design automation for the system design considering both the software hardware partition and various architecture will be finally needed

• Rich ecosystem for FPGA based system is the final goal!