Cascade: An Application Pipelining Toolkit for Coarse-Grained Reconfigurable Arrays

Jackson Melchert, Yuchen Mei, Kalhan Koul, Qiaoyi Liu, Mark Horowitz, Priyanka Raina
Motivation

- Our EDP and runtime will only be competitive with FPGAs, CPUs, and GPUs if we have high application frequency.

- FPGA and ASIC commercial design tools allow for timing analysis and pipelining:
  - Designers can insert registers into the RTL and tools will retime them.
  - HLS tools can automatically insert registers to meet timing targets.
Introduction

- Cascade includes:
  - A methodology for generating timing models of CGRAs
  - Static timing analysis tool that uses the timing model to determine the critical path of a CGRA application
  - Automated software pipelining techniques for both dense and sparse applications, and a hardware optimization for further frequency improvements
Background - CGRA Architecture
Background - CGRA Architecture

- Several classes of CGRA interconnects exist:
  1. Exhaustively pipelined interconnects
     - Expensive NoC
  2. Non-pipelined interconnects
     - Cannot run at high frequencies
  3. Interconnects with configurable registers
     - Need pipelining techniques to run fast
Modeling Application Frequency
Modeling Application Frequency
Modeling Application Frequency
Modeling Application Frequency
Methodology for Generating Timing Model

Interconnect Specification

Post P&R Netlist

Timing Queries (report_timing)

Application STA Tool

All Timing Paths

Worst-Case Timing Model

IN to PE: 240ps

PE to OUT: 200ps

IN to PE: 240ps

PE to OUT: 200ps
Application Compiler Toolchain

Application in Halide → Scheduling with Halide Compiler + Clockwork

Application Compiler Toolchain

Dataflow Graph → Mapper → Dataflow Graph of PE/MEM

Placer & Router → Placement on CGRA → Bitstream Generator → CGRA Bitstream

Example: 3x3 Convolution

```
conv(x, y) += kernel(r.x, r.y) *
  input(x+r.x, y+r.y);
conv.in().compute_root();
conv.in()
  .tile(x,y,xo,yo,xi,yi,64,64)
  .hw_accelerate(xi, xo);
conv.update()
  .unroll(r.y, 3)
  .unroll(r.x, 3);
conv.compute_at(conv.in(), xo);
input.stream_to_accelerator();
```
We construct the delay model using the output of the place and route tool.
Static Timing Analysis Model

Critical Path
Software Pipelining Techniques

1. Compute Pipelining
2. Broadcast Signal Pipelining
3. Placement Algorithm Cost Function Optimization
4. Post-Place-and-Route Pipelining
5. Low Unrolling Duplication
Compute Pipelining

- At compute mapping, we know how many PEs we will use and how they are connected.
- The combinational paths through compute kernels need to be pipelined.
Compute Pipelining

- At compute mapping, we know how many PEs we will use and how they are connected.
- The combinational paths through compute kernels need to be pipelined.
- Registers are added to the compute graph:
  - Need to be packed into the PEs or placed onto the routing fabric.
Broadcast Signal Pipelining

- Broadcast signals are frequently the most expensive paths in our applications
- They are routed with minimum wirelength, resulting in long critical paths
- We specifically pipeline broadcast signals to form register trees
Placement Algorithm Cost Function Optimization

- The cost metric for placement is total wirelength
  - Generally leads to good routability

- For pipelining we care about max wirelength, not total wirelength
  - Using max wirelength alone as a cost metric for placement leads to routability issues
  - Instead we can create a new cost metric that incorporates both maximum and total wirelength
  - HPWL$^n$ for $n = [1...20]$ works well

half-perimeter wirelength = $x+y$
Post Place and Route Pipelining

- After Place and Route, we use the STA tool to determine the critical path
- Iteratively break the critical path, re-analyze and determine new critical path
- Continue until there are no more registers available to use on the interconnect
  - Adding pipelining registers to a placement result is not always possible because branch delay matching is required
Post Place and Route Pipelining Example
Post Place and Route Pipelining Example
Post Place and Route Pipelining Example

Critical Path Break
Low Unrolling Duplication

- In practice, the place-and-route tool struggles with large applications
  - We want to utilize as much of the CGRA as possible
- We can break each application into small pieces, and then duplicate the bitstream across the array

Low frequency
High utilization

High frequency
Low utilization

High frequency
High utilization
Rescheduling CGRA Applications

Statically scheduled applications need to be updated after inserting registers

1. Statically schedule the application without registers
2. Map and place and route the application using software pipelining techniques
3. Analyze the application and compute new kernel latencies
4. Reschedule the application before bitstream generation
Software Pipelining Flow
Hardware Pipelining Techniques

- Some broadcast signals are too big to pipeline.
- In our CGRA, we harden the flush signal to remove it from the interconnect.
Pipelining Sparse Applications

- Sparse applications use ready-valid interfaces for all communication
  - These can be pipelined using FIFOs rather than registers
- We can use the same pipelining techniques with FIFO insertion to pipeline all sparse applications
Results - STA Model Evaluation

- STA model predicts the actual clock period accurately
  - Above 500 MHz, the average error is 13%
- All clock period errors are positive
  - Our model is pessimistic, it will provide a lower bound for the clock frequency
Results - Software Pipelining Dense Applications

The diagram illustrates the runtime performance of dense applications with different optimizations. The x-axis represents the applications: Gaussian, Unsharp, Camera, Harris, and ResNet. The y-axis represents the runtime (normalized). The chart shows the percentage reduction in runtime when using various optimizations:

- Unpipelined
- + Compute pipelining
- + Broadcast pipelining
- + Placement optimization
- + Low unrolling duplication
- + Post-PnR pipelining

The applications are shown with their respective runtime improvements: -40%, -63%, -78%, -81%, -81%, -81%, -48%, -85%, and -83%.
Results - Software Pipelining Dense Applications

-86%  
-94%  
-98%  
-99%  
-96%

EDP (μJ/s/frame)

Gaussian  Unsharp  Camera  Harris  ResNet

Unpipelined  All software pipelining
Results - Hardware Pipelining Dense Applications

-31%  
-32%  
-53%  
-41%  
-56%

- All software pipelining  
- + Hardened broadcasts

Runtime (normalized)

Gaussian  Unsharp  Camera  Harris  ResNet
## Unpipelined vs Pipelined Dense Applications

<table>
<thead>
<tr>
<th>Dense Application</th>
<th>Frequency (MHz)</th>
<th>Runtime (ms/frame)</th>
<th>Power (mW)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unpipelined</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gaussian</td>
<td>103</td>
<td>22.6</td>
<td>156</td>
</tr>
<tr>
<td>Unsharp</td>
<td>66</td>
<td>21.4</td>
<td>139</td>
</tr>
<tr>
<td>Camera</td>
<td>47</td>
<td>28.3</td>
<td>318</td>
</tr>
<tr>
<td>Harris</td>
<td>30</td>
<td>70.6</td>
<td>85</td>
</tr>
<tr>
<td>ResNet</td>
<td>57</td>
<td>31.7</td>
<td>119</td>
</tr>
<tr>
<td><strong>Pipelined</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gaussian</td>
<td>610</td>
<td>3.66</td>
<td>841</td>
</tr>
<tr>
<td>Unsharp</td>
<td>532</td>
<td>1.99</td>
<td>903</td>
</tr>
<tr>
<td>Camera</td>
<td>457</td>
<td>2.96</td>
<td>678</td>
</tr>
<tr>
<td>Harris</td>
<td>571</td>
<td>1.90</td>
<td>614</td>
</tr>
<tr>
<td>ResNet</td>
<td>457</td>
<td>3.96</td>
<td>304</td>
</tr>
</tbody>
</table>
Results - Software Pipelining Sparse Applications

- Vector Element Add: -51%
- Matrix Element Mul: -23%
- Tensor MTTKRP: -60%
- Tensor TTV: -55%

- Placement optimization:
  - Vector Element Add: -28%
  - Matrix Element Mul: -34%
  - Tensor MTTKRP: -44%
  - Tensor TTV: -46%

- Post-PnR pipelining:
  - Vector Element Add: -51%
  - Matrix Element Mul: -34%
  - Tensor MTTKRP: -60%
  - Tensor TTV: -55%
Results - Software Pipelining Sparse Applications

-58%  
-35%  
-71%  
-76%

-58%  
-35%  
-71%  
-76%

Vector Element Add  
Matrix Element Mul  
Tensor MTTKRP  
Tensor TTV

EDP (fJ·s)
# Unpipelined vs Pipelined Sparse Applications

<table>
<thead>
<tr>
<th>Sparse Application</th>
<th>Frequency (MHz)</th>
<th>Runtime (μs)</th>
<th>Power (mW)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Compute Pipelining</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector Elementwise Add</td>
<td>305</td>
<td>0.83</td>
<td>187</td>
</tr>
<tr>
<td>Matrix Elementwise Mul</td>
<td>435</td>
<td>1.38</td>
<td>246</td>
</tr>
<tr>
<td>Tensor MTTKRP</td>
<td>300</td>
<td>33.9</td>
<td>194</td>
</tr>
<tr>
<td>Tensor TTV</td>
<td>260</td>
<td>10.0</td>
<td>170</td>
</tr>
<tr>
<td><strong>All Software Pipelining</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector Elementwise Add</td>
<td>599</td>
<td>0.42</td>
<td>303</td>
</tr>
<tr>
<td>Matrix Elementwise Mul</td>
<td>599</td>
<td>0.99</td>
<td>316</td>
</tr>
<tr>
<td>Tensor MTTKRP</td>
<td>617</td>
<td>14.2</td>
<td>320</td>
</tr>
<tr>
<td>Tensor TTV</td>
<td>617</td>
<td>3.52</td>
<td>325</td>
</tr>
</tbody>
</table>
Conclusion

- Cascade is CGRA application pipelining toolkit that enables high performance and energy efficiency:
  - Methodology for generating timing models of CGRAs
  - Static timing analysis tool for CGRA applications
  - Software/hardware pipelining techniques

- 7-34x shorter critical paths and 7-190x lower EDP for dense apps
- 2-4.4x shorter critical paths and 1.5-4.2x lower EDP for sparse apps