Onyx: A 12nm SoC with a Coarse-Grained Reconfigurable Array Accelerating Dense and Sparse Applications

Kalhan Koul*, Maxwell Strange, Jack Melchert, Alex Carsello, Po-Han Chen, Jake Ke, Yuchen Mei, Olivia Hsu, Taeyoung Kong, Keyi Zhang, Gedeon Nyengele

Stanford University
What’s Next?

- Amber demonstrated using DSLs to generate an accelerator
  - This is not unique to our system!
- So what’s special about AHA?
  - Our DSLs generate the collateral for the compiler each time the hardware changes
  - Allows exploring a large design space and evaluate on full application kernels
Leverage the AHA Design Flow

- Reminder: **PEak** to design PEs, **Lake** to design memories with controllers, and **Canal** to design the interconnect
Onyx Contributions

Compute Density

Application-driven automatic PE design
Onyx Contributions

**Compute Density**
Application-driven automatic PE design

**Improved Pipelining**
Hardware and software techniques to speed up dense applications
Onyx Contributions

**Compute Density**
Application-driven automatic PE design

**Improved Pipelining**
Hardware and software techniques to speed up dense applications

**Sparse Applications**
Accelerate sparse applications on a CGRA
Onyx Contributions

**Compute Density**
Application-driven automatic PE design

**Improved Pipelining**
Hardware and software techniques to speed up dense applications

**Sparse Applications**
Accelerate sparse applications on a CGRA
Challenge 1: Compute Density

- Amber PEs have a high interconnect overhead and the applications were compute-limited.
- How should reconfigurable accelerators determine compute complexity given a set of applications?
Contribution 1: Specialized PEs

- We developed the Automated PE Exploration (APEX) tool to produce an optimal PE given a set of applications.
Subgraph Mining and Maximal Independent Set Analysis

Subgraph Mining

- **Input:** Application Graph(s)
- **Output:** List of subgraphs and their frequencies

(a) CoreIR Application Graph
(b) Subgraph 1 Frequency: 4
(c) Subgraph 2 Frequency: 4
(d) Subgraph 3 Frequency: 4
Subgraph Mining and Maximal Independent Set Analysis

Subgraph Mining

- Input: Application Graph(s)
- Output: List of subgraphs and their frequencies

Maximal Independent Set (MIS) Analysis

- Finds the maximum number of non-overlapping occurrences of a subgraph in an application graph
Frequent Subgraphs for Onyx

- Analyzed our application set using subgraph mining and maximal independent set analysis
- Produced 4 frequent subgraphs with high MIS values
  - Multiply-Add
  - Add-Add
  - Multiply-Shift
  - Min-Max
Subgraph Merging

- Given a set of operations, generate an efficient PE
  - Take two subgraphs and create a bipartite graph of potential merging opportunities
Subgraph Merging

- Given a set of operations, generate an efficient PE
  - Take two subgraphs and create a bipartite graph of potential merging opportunities
  - Convert into a compatibility graph - each potential merging is represented as a node, and each compatible merging is represented as an edge, which is weighted with the area reduction
Subgraph Merging

- Given a set of operations, generate an efficient PE
  - Take two subgraphs and create a bipartite graph of potential merging opportunities
  - Convert into a compatibility graph - each potential merging is represented as a node, and each compatible merging is represented as an edge, which is weighted with the area reduction
  - Using the maximum weight clique of the compatibility graph, the lowest cost merging of the two subgraphs can be constructed
Amber PE to Onyx PE

- The Onyx PE adds four key operations used in image processing, computer vision, and machine learning:
  - Multiply-add (mac)
  - Add-add (tadd)
  - Multiply-shift right (mul-shr)
  - Min-max (crop)
Onyx PE Results

- Reducing PE count allows us to further unroll applications
- No longer compute-limited!
Onyx Contributions

Compute Density
Application-driven automatic PE design

Improved Pipelining
Hardware and software techniques to speed up dense applications

Sparse Applications
Accelerate sparse applications on a CGRA
Challenge 2: Low Application Frequencies

How do we design a reconfigurable accelerator with the benefits of a static schedule, but achieve high frequencies?

Amber Application Frequencies

<table>
<thead>
<tr>
<th>Application</th>
<th>App Freq (MHz)</th>
<th>Max Freq (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Harris</td>
<td>400</td>
<td>750</td>
</tr>
<tr>
<td>Camera</td>
<td>300</td>
<td>750</td>
</tr>
<tr>
<td>Unsharp</td>
<td>250</td>
<td>750</td>
</tr>
<tr>
<td>Blur</td>
<td>300</td>
<td>750</td>
</tr>
</tbody>
</table>
Contribution 2: Improved Application Frequencies

- Hardware Improvement: Used Canal to change the interconnect hardware
- Software Improvement: Added pipelining improvements to the Mapper and Place & Router
Hardware Improvements

- Added an IO register to break long paths between the global buffer and CGRA
Hardware Improvements

- Added an IO register to break long paths between the global buffer and CGRA
- Application broadcast signals (flush) need to go to every memory tile in the application: harden those signals
Hardware Improvements

- Most critical paths are dominated by hops on the interconnect
- Different hop directions have a wide range of delays (150ps-300ps)
- Ex: N->S 150ps, but S->N is 220ps (for PEs)
- By further constraining these paths in physical design we can reduce the decrease the variety and reduce the critical path
Software Pipelining Techniques

Compute Pipelining
Software Pipelining Techniques

Compute Pipelining

Register File Pipelining

Reg File
Software Pipelining Techniques

Compute Pipelining

Placement Optimizations

Cost function = \((x+y)^n\)
Software Pipelining Techniques

Compute Pipelining

Placement Optimizations

Cost function = $(x+y)^n$
Application Frequency Results

Next steps:
- Increase our max frequency by pushing the timing in PD
- Explore timing-driven P&R for CGRAs
Dense Application Results: Runtime

Runtime (ms/frame)

- Harris: Intel Xeon CPU (1 Core) 36.5ms, Intel Xeon CPU (12 Cores) 41.3ms, NVIDIA Tesla K40 GPU 5.9ms, Amber 1.1ms, Onyx 1.09ms
- Camera: Intel Xeon CPU (1 Core) 29.2ms, Intel Xeon CPU (12 Cores) 41.3ms, NVIDIA Tesla K40 GPU 6.8ms, Amber 4.09ms, Onyx 2ms
- Unsharp: Intel Xeon CPU (1 Core) 26.4ms, Intel Xeon CPU (12 Cores) 29.2ms, NVIDIA Tesla K40 GPU 6.8ms, Amber 6.53ms, Onyx 2ms
- Blur: Intel Xeon CPU (1 Core) 7.5ms, Intel Xeon CPU (12 Cores) 26.4ms, NVIDIA Tesla K40 GPU 6.27ms, Amber 1.78ms, Onyx 2ms
Dense Application Results: EDP

Energy Delay Product EDP (uJ.s/frame)

- Harris: Intel Xeon CPU (1 Core) - 19000, Intel Xeon CPU (12 Cores) - 24000, NVIDIA Tesla K40 GPU - 280, Amber - 20.3, Onyx - 1.17
- Camera: Intel Xeon CPU (1 Core) - 6100, Intel Xeon CPU (12 Cores) - 940, NVIDIA Tesla K40 GPU - 6.19, Amber - 2.79, Onyx - 1.59
- Unsharp: Intel Xeon CPU (1 Core) - 12000, Intel Xeon CPU (12 Cores) - 7900, NVIDIA Tesla K40 GPU - 190, Amber - 15.8, Onyx - 1.59
- Blur: Intel Xeon CPU (1 Core) - 9909600, Intel Xeon CPU (12 Cores) - 940, NVIDIA Tesla K40 GPU - 940, Amber - 17.7, Onyx - 4.75
Onyx Contributions

**Compute Density**
Application-driven automatic PE design

**Improved Pipelining**
Hardware and software techniques to speed up dense applications

**Sparse Applications**
Accelerate sparse applications on a CGRA
Sparse Applications

- Sparse datasets are seen in scientific computing, data analytics and ML
- Sparse applications are growing in popularity and offer a great opportunity to extend our agile flow to another application domain
Challenge 3: System Limited to Dense Applications

- Our hardware and compiler are currently limited to dense linear algebra
Contribution 3: Sparse Applications

- Use **Canal** to design a ready-valid interconnect to support data-dependent dataflow
- Use **Lake** to design sparse primitives
- Use **TACO** and the **Sparse Abstract Machine (SAM)** to generate the dataflow graph
Enabling Sparse Applications on our CGRA

- Vector-Vector Element Add Example
  - $x[] = b[] + c[]$
  - coordinates (crd): location in tensor
  - references (ref): location in memory
  - value (val): actual value
  - fiber: coordinate, reference, value tuple
Enabling Sparse Applications on our CGRA

- Vector-Vector Element Add Example
  - $x[] = b[] + c[]$
  - coordinates (coord): location in tensor
  - references (ref): location in memory
  - value (val): actual value
  - fiber: coordinate, reference, value tuple
Enabling Sparse Applications on our CGRA

- Vector-Vector Element Add Example
  - $x[] = b[] + c[]$
  - coordinates (coord): location in tensor
  - references (ref): location in memory
  - value (val): actual value
  - fiber: coordinate, reference, value tuple

Ex SAM Graph: Vector-Vector Element Add

- FiberLookup $b$
- FiberLookup $c$
- FiberWrite $x$
- Array $b$
- Array $c$
- Coord $x$
- Coord $ref\_in\_b$
- Coord $ref\_in\_c$
- Coord $coord\_in\_b$
- Coord $coord\_in\_c$
- Coord $ref\_out\_b$
- Coord $ref\_out\_c$
Enabling Sparse Applications on our CGRA

- Vector-Vector Element Add Example
  - $x[] = b[] + c[]$
  - coordinates (coord): location in tensor
  - references (ref): location in memory
  - value (val): actual value
  - fiber: coordinate, reference, value tuple

Ex SAM Graph: Vector-Vector Element Add

FiberLookup $b$
FiberLookup $c$

$\text{union}$

FiberWrite $x$

Array $b$
Array $c$

Add

FiberWrite $x$
Enabling Sparse Applications on our CGRA

- Blue boxes are broken down into sparse primitive(s) that we will add to our CGRA

Ex SAM Graph: Vector-Vector Element Add

- FiberLookup b
- FiberLookup c
- Union
- FiberWrite x
- Array b
- Array c
- Add
- FiberWrite x

ref_in_b, ref_in_c, coord_in_b, coord_in_c, ref_out_b, ref_out_c, b val, c val, x val, x coord
Sparse Primitives

- Used Lake to design sparse primitives
  - Block Buffer - Coarse-grain, tile-level N-buffered (N=8 in Onyx) memory controller
  - Write/Read Scanner - manage communication between the global buffer and memory tiles
Sparse Primitives

- Used Lake to design sparse primitives
  - Block Buffer - Coarse-grain, tile-level N-buffered (N=8 in Onyx) memory controller
  - Write/Read Scanner - manage communication between the global buffer and local SRAM
Sparse Primitives

- Used Lake to design sparse primitives
  - Block Buffer - Coarse-grain, tile-level N-buffered (N=8 in Onyx) memory controller
  - Write/Read Scanner - manage communication between the global buffer and local SRAM
Sparse Primitives cont.

- Used Lake to design sparse primitives
  - Intersect/Union - calculates intersection/union of two streams of coordinates
  - Reduce - accumulates a stream of data (sum reduction across values)
  - Repeat - broadcasts a stream over another

Coordinates

\[\begin{align*}
10 & 8 & 7 & 4 & 2 \\
9 & 8 & 4 & 3 & 1 \\
8 & 4 & \end{align*}\]
Sparse Primitive Grouping using APEX

- Again we leveraged subgraph mining and maximal independent set analysis
  - At the primitive level (not mathematical operation)
- The write scanner and read scanner interacting with the block buffer is by far the most common subgraph
Sparse Primitive Grouping

- Placed write scanner, read scanner, and buffet primitives in the memory tile since they are memory facing
- All other primitives placed in the PE tile
Area Overhead of Adding Sparsity Support to PEs

- Area reduction from tsmc16 to gf12 across the board
- Total area increase is 60%
- 11% from new PE complexity
- 28% from adding ready-valid capabilities
- 12% from sparse hardware
Area Overhead of Adding Sparsity Support to MEMs

- Area reduction from tsmc16 to gf12 across the board
- Total area increase is 13%
- -15% from unified buffer optimizations
- 7% from adding ready-valid capabilities
- 22% from sparse hardware
Sparse Application Status

- Verified applications show in green
- Next steps: verify the rest of the application kernels and confirm our applications are performant

<table>
<thead>
<tr>
<th>Name</th>
<th>Expression</th>
<th>Dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>SpMV</td>
<td>$x_i = \sum_j B_{ij} c_j$</td>
<td>SuiteSparse</td>
</tr>
<tr>
<td>SpM*SpM</td>
<td>$X_i = \sum_j B_{ij} C_{jk}$</td>
<td>SuiteSparse</td>
</tr>
<tr>
<td>MMAdd</td>
<td>$X_{ij} = B_{ij} + C_{ij}$</td>
<td>SuiteSparse</td>
</tr>
<tr>
<td>Plus3</td>
<td>$X_{ij} = B_{ij} + C_{ij} + D_{ij}$</td>
<td>SuiteSparse</td>
</tr>
<tr>
<td>SDDMM</td>
<td>$X_{ij} = \sum_k B_{ij} C_{ik} D_{jk}$</td>
<td>SuiteSparse</td>
</tr>
<tr>
<td>MatTransMul</td>
<td>$x_i = \sum_j aB^T_{ij} c_j + \beta d_i$</td>
<td>SuiteSparse</td>
</tr>
<tr>
<td>Residual</td>
<td>$x_i = b_i - \sum_j C_{ij} d_j$</td>
<td>SuiteSparse</td>
</tr>
<tr>
<td>TTV</td>
<td>$X_{ij} = \sum_k B_{ijk} c_k$</td>
<td>Facebook, Frostt</td>
</tr>
<tr>
<td>TTM</td>
<td>$X_{ijk} = \sum_l B_{ijl} C_{kl}$</td>
<td>Facebook, Frostt</td>
</tr>
<tr>
<td>MTTKRP</td>
<td>$X_{ij} = \sum_{kl} B_{ijkl} C_{jk} D_{jl}$</td>
<td>Facebook, Frostt</td>
</tr>
<tr>
<td>InnerProd</td>
<td>$x = \sum_{ijk} B_{ijk} C_{ijk}$</td>
<td>Facebook, Frostt</td>
</tr>
<tr>
<td>Plus2</td>
<td>$X_{ijk} = B_{ijk} + C_{ijk}$</td>
<td>Facebook, Frostt</td>
</tr>
</tbody>
</table>
Summary of Key Contributions

● We designed Onyx leveraging the AHA methodology
● We improve upon our CGRA in three key ways
  ○ Improved compute density - targeting 768 Peak GOPS (Amber 367 Peak GOPs)
  ○ Improved runtime through HW/SW pipelining techniques - 2.7-6.2x runtime improvement
  ○ Added the ability to do sparse applications with less than 50% added area to our tile array
● Our methodology enables quick design space exploration and can be adapted to different application domains