Energy-Efficient Acceleration of RNNs using CGRA

Aviral Shrivastava, Arizona State University
CGRA: Coarse-Grain Reconfigurable Arrays

- An array of Processing Elements (PEs); each PE has ALU-like functional unit that works on an operation at every cycle.

- Array configurations vary in terms of –
  - Array Size
  - Functional Units
  - Reg. File Architectures
  - Interconnect Network

- CGRAs can achieve power-efficiency of several 10s of GOps/Sec per Watt.
  - ADRES CGRA chip, up to 60 GOps/sec per Watt [IMEC, HiPEAC 2008]
  - HyCUBE chip, about 63 MIPS/mW [M. Karunaratne et al., DAC 2017]

- Popular in Embedded Systems and Multimedia [Samsung SRP processor]
Iterative Modulo Scheduling – Every operation is executed at II cycles.

Initiation Interval aka II is performance metric.

Software Pipelining – Operations from different iterations can be executed simultaneously. This empowers to accelerate even non-parallel loops through the CGRAs.

Mapping Applications on CGRAs

for(i=1; i<1000; i++){
  a: A[i] = B[i-1] - 4;
  c: C = A[i] * 3
  d: D[i] = C + 7;
}

Sample Loop

Performance (loop execution time) critically depends on the mapping obtained by compiler.

Data Dependency Graph

Compiling Maps DDG onto CGRA

Time-Extended CGRA

Performance (loop execution time) critically depends on the mapping obtained by compiler.
CGRAs Becoming a Hotbed of Research

Recently, several techniques and evaluations for CGRAs or CGRA-like spatial architectures have been presented including:

- Control Flow Coalescing on a Hybrid Dataflow/von Neumann GPGPU. Dani Voitsechov Yoav Etsion. In MICRO 2015. [Technion, Israel]
- A space-and energy-efficient code compression/decompression technique for coarse-grained reconfigurable architectures. Bernhard Egger et al. In CGO 2017. [SNU and Samsung]
Key Features of CGRA Accelerators

- With software-pipelined execution, CGRA PEs can efficiently accelerate loops with lower parallelism
  - E.g. loops with loop-carried dependence, inter-twined loops, loops with high branch divergence etc.

- Avoids one of the fundamental bottlenecks of Von-Neumann architecture i.e., CGRAs are not subjected to dynamic fetching and decoding of instructions.
  - CGRA instructions are pre-decoded in memory, and PEs transfer data directly among each other, without necessarily going through centralized register file/memory.

- Efficient mapping of loop operations is done by compiler, no programmer intervention is needed.
  - Performance-critical kernels of several irregular applications can benefit from acceleration.

Fig 21 from recent Intel patent on Configurable Spatial Architecture (CSA)

Article:
CCF: CGRA Compiler+Simulation Framework

Mark the loops

Extract the loop

Map and schedule on CGRA

Use CGRACC as the compiler

Generate the DFG

Instruction selection

Run on gem5

- llvm 5.0 and gem5 as foundation
- Public Release: https://github.com/cmlasu/ccf
- Video: https://www.youtube.com/watch?v=iGV1Hsjiy4w

ARM Summit 2018

Web page: aviral.lab.asu.edu
Our Recent Work on Compiling for CGRAs

- Efficient software pipeline

- Efficiently using distributed register files

- Register file organization

- Efficient mapping of if-then-else’s
RAMP [DAC 2018]: Selecting a Routing Alternative

- Various routing strategies

  (a) Routing via PEs
  (b) Spill to distributed RFs
  (c) Spill to Memory
  (d) Load Read-Only from Memory
  (e) Re-Computation

- Failure Analysis

  - Dependent operations are scheduled at distant time; managing the data with large lifetime in registers is not possible
    - Route by PEs, Spill to memory/distributed RFs
  - Source operand is a live-in value, and cannot be managed in the registers
    - Load the live-in value from the memory
  - Dependent operations are scheduled at the consecutive cycles; routing is not possible due to limited interconnect/unavailability of free PEs
    - Re-compute, Route by a PE, Re-schedule
RAMP vs. RegiMap and MemMap

For the top performance-critical loops from 8 MiBench benchmarks, previous techniques failed to obtain mappings for almost all loops, when highly constrained by the resources.

RAMP accelerated the top performance-critical loops of 8 embedded applications from MiBench by 23× as compared to sequential execution, and by 2.13× over REGIMap, and by 3.39× over MEMMap.

Table 1: Specifications of CGRA architecture configurations

<table>
<thead>
<tr>
<th>Config. #</th>
<th>Size</th>
<th>RF</th>
<th>Reg. in RF</th>
<th>Memory Units (PEs)</th>
<th>Sharing of Memory Bus</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2x2</td>
<td>Centralized</td>
<td>16</td>
<td>3, 4</td>
<td>dedicated</td>
</tr>
<tr>
<td>2</td>
<td>2x2</td>
<td>Centralized</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2x2</td>
<td>Local</td>
<td>2</td>
<td>Homogeneous PEs (All)</td>
<td>shared among PEs of a row</td>
</tr>
<tr>
<td>4</td>
<td>2x2</td>
<td>Local</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4x4</td>
<td>Centralized</td>
<td>64</td>
<td></td>
<td>dedicated</td>
</tr>
<tr>
<td>6</td>
<td>4x4</td>
<td>Local</td>
<td>2</td>
<td>Homogeneous PEs (All)</td>
<td>shared among PEs of a row</td>
</tr>
<tr>
<td>7</td>
<td>4x4</td>
<td>Local</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>4x4</td>
<td>Local</td>
<td>4</td>
<td>2, 4, 6, 8</td>
<td>dedicated</td>
</tr>
<tr>
<td>9</td>
<td>8x8</td>
<td>Centralized</td>
<td>128</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>8x8</td>
<td>Local</td>
<td>4</td>
<td>Homogeneous PEs (All)</td>
<td>shared among PEs of a row</td>
</tr>
<tr>
<td>11</td>
<td>8x8</td>
<td>Local</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>8x8</td>
<td>Local</td>
<td>8</td>
<td>1, 3, 5, 7, 9, 11, 13, 15, 19, 21</td>
<td>dedicated</td>
</tr>
</tbody>
</table>
Residual Blocks from ResNet-18

Input Feature Map
- 3x3 conv 64 filters, s=1
  - 56x56x64
  - Batch Normalization
  - Scale
  - ReLU
  - 3x3 conv 64 filters, s=1
    - 56x56x64
    - Batch Normalization
    - Scale

Data Reuse Weights
- Ifmap
- 36k (N)
- 196k (2)

Input Feature Map
- 3x3 conv 512 filters, s=1
  - 7x7x512
  - Batch Normalization
  - Scale
  - ReLU
  - 3x3 conv 512 filters, s=1
    - 7x7x512
    - Batch Normalization
    - Scale

Data Reuse Weights
- Ifmap
- 2304k (N)
- 24.5k (2)

*N: Batch size
**Execution Mechanism for Inference**

RF Size: 0.64 kB

⇒ Simultaneously operate on 32 (input) channels of 1 (output) filter

**Controller**

**Scratch-Pad Memory**

- ~2k
- ~41k
- ~25k

**Filter Weights**

**Input Feature Map** (padded, quad-buffered)

**Output Feature Map** (quad-buffered)

Dark shaded areas indicate buffered data (in use by PEs)

Light shade indicates prefetching/write-back

3x3 conv
512 filters, s=1

7x7x512

3x3 conv
512 filters, s=1

Batch Normalization

Scale

ReLU

Batch Normalization

Scale

 additive
Executing Residual Block on CGRA

- Pre-load filter weights to RF

![Diagram of Residual Block on CGRA](image)
Output Stationary Dataflow for Convolutions

**weight**

<table>
<thead>
<tr>
<th>(3,3)</th>
<th>(3,2)</th>
<th>(3,1)</th>
<th>(2,3)</th>
<th>(2,2)</th>
<th>(2,1)</th>
<th>(1,3)</th>
<th>(1,2)</th>
<th>(1,1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ch1</td>
<td>ch1</td>
<td>ch1</td>
<td>ch1</td>
<td>ch1</td>
<td>ch1</td>
<td>ch1</td>
<td>ch1</td>
<td>ch1</td>
</tr>
<tr>
<td>ch1</td>
<td>ch1</td>
<td>ch1</td>
<td>ch1</td>
<td>ch1</td>
<td>ch1</td>
<td>ch1</td>
<td>ch1</td>
<td>ch1</td>
</tr>
<tr>
<td>ch1</td>
<td>ch1</td>
<td>ch1</td>
<td>ch1</td>
<td>ch1</td>
<td>ch1</td>
<td>ch1</td>
<td>ch1</td>
<td>ch1</td>
</tr>
<tr>
<td>512-channels</td>
<td>512</td>
<td>512</td>
<td>512</td>
<td>512</td>
<td>512</td>
<td>512</td>
<td>512</td>
<td>512</td>
</tr>
<tr>
<td>9x9 (padded)</td>
<td>8x3</td>
<td>7x7 (partial)</td>
<td>512 filters</td>
<td>512 filters</td>
<td>512 filters</td>
<td>512 filters</td>
<td>512 filters</td>
<td>512 filters</td>
</tr>
</tbody>
</table>

**PE**

<table>
<thead>
<tr>
<th>(7,7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,1) ch1</td>
</tr>
<tr>
<td>(1,2) ch1</td>
</tr>
<tr>
<td>(1,3) ch1</td>
</tr>
<tr>
<td>(2,1) ch1</td>
</tr>
<tr>
<td>(2,2) ch1</td>
</tr>
<tr>
<td>(2,3) ch1</td>
</tr>
<tr>
<td>(3,1) ch1</td>
</tr>
<tr>
<td>(3,2) ch1</td>
</tr>
<tr>
<td>(3,3) ch1</td>
</tr>
<tr>
<td>512-entry RF</td>
</tr>
<tr>
<td>psum ofmap1</td>
</tr>
<tr>
<td>512-entry RF</td>
</tr>
<tr>
<td>512-entry RF</td>
</tr>
<tr>
<td>512-entry RF</td>
</tr>
<tr>
<td>512-entry RF</td>
</tr>
<tr>
<td>512-entry RF</td>
</tr>
<tr>
<td>512-entry RF</td>
</tr>
</tbody>
</table>

**partial sum**

**Web page:** aviral.lab.asu.edu
Streaming Feature Maps for MACs

- Input feature map streamed through PEs
Output Stationary Dataflow for Convolutions

Ifmap (channel 512) x x x x + + + +
weight (3,3) (3,2) (3,1) (2,3) (2,2) (2,1) (1,3) (1,2) (1,1) = partial sum

PE (7,7)
512-entry RF

(1,1) ch481 (1,2) ch481 (1,3) ch481
(1,2) ch512 (1,3) ch512
(2,1) ch512 (2,2) ch512
(2,3) ch512 (2,4) ch512
(3,1) ch512 (3,2) ch512
(3,3) ch512

512 channels
9x9 (padded)
7x7
512
512 filters
psum ofmap1

512
512 filters

(1,1) (1,2) (1,3) (1,4) (1,5) (1,6) (1,7)
(2,1) (2,2) (2,3) (2,4) (2,5) (2,6) (2,7)
(3,1) (3,2) (3,3) (3,4) (3,5) (3,6) (3,7)
(4,1) (4,2) (4,3) (4,4) (4,5) (4,6) (4,7)
(5,1) (5,2) (5,3) (5,4) (5,5) (5,6) (5,7)
(6,1) (6,2) (6,3) (6,4) (6,5) (6,6) (6,7)
(7,1) (7,2) (7,3) (7,4) (7,5) (7,6) (7,7)

7x7 (padded)
9x9 *
= 512 filters

512-entry RF

channels
512
512 filters

arm.summit.2018
Output Stationary Dataflow for Convolutions

- Input feature map streamed through PEs
  - PEs perform MACs on input data and/or passes to neighboring PES
  - Partial sums stored in the RF
- Batch Normalization, Scaling and ReLU performed before storing output feature map
Experimental Setup

- Neural networks evaluated: ResNet-18 and ResNet-34 models
- Computations on N-dimension feature maps defined in C/C++
- Dataset: ImageNet
  - Input feature map: 224x224x3.
- Cycle-count performance comparison of the 2 approaches
  - Baseline: Intel Core i7-870 CPU (2.93GHz, Quad-Core)
    - 256 kB L1, 1 MB L2, 8 MB L3
    - 8 GB system memory (4 GB DIMMs, DDR3 1.33GHz)
    - Performance measured in terms of execution cycles on a core (linear scaling to 4 cores)
    - Profiling: GNU Perf (stat collection from hardware counters)
    - Compilation with g++ -O3 (aggressive loop optimizations, and auto-vectorization enabled)
    - Algorithmic representation for 4D convolutions
      - Conventional representation shown in Minimizing Computation in Convolutional Neural Networks by J. Cong et al., in ICANN 2014.
  - Performance model for CGRA (efforts ongoing):
    - In-house C++ simulator (integration with Gem5 ongoing)
    - 4 clusters of PEs; each cluster is mounted with 68kB scratch-pad memory (total 196 PEs)
    - Dataflow execution: Output stationary (streaming data/MAC/compare takes 1 cycle, pipelined)
    - DMA model for scratch-pad memory management: latency (cycles) = 291 + 0.24*bytes
**Early Results**

**Throughput Improvement**

<table>
<thead>
<tr>
<th>N</th>
<th>ResNet-18</th>
<th>ResNet-34</th>
</tr>
</thead>
<tbody>
<tr>
<td>N=1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N=4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N=16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N=64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N=128</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

N: Batch size
Baseline: Intel i7-870 Quad-core CPU
OS: Output Stationary dataflow for Convolutions
OPT1: Batch normalization, ReLu, pooling on CGRA
OPT2: Software prefetching on SPM enabled through quad-buffering.

Further Optimizations Possible:

- Interleave partial sum computations across filters instead of channels, i.e., operate on various filters for an input channel, to better reuse input feature maps. ~ 4X speedup over OS+OPT[1-2]
- Design dataflow execution to consider variation in data reuse opportunities (filter weights vs. ifmap for early residual blocks in model)
- Design-space exploration of the architecture for using all-PE communication through input/output FIFOs instead of streaming the data through a single PE.
Computing System Stack [Ongoing]

Standing On The Shoulder Of open-source frameworks developed by Giants

Pragma based annotations ➔
Support for accelerator execution ➔
Front-end to work on Tensors
Schedule tensors on accelerator clusters

Application
Algorithm
Programming Language
Libraries/Utilities
Compiler
Operating (Runtime) System
Instruction Set Architecture
Microarchitecture
Logic (Register-Transfer Level)
Circuits
Devices/Technology

TensorFlow
(Tensor-Graph)
VTA (UW)
VTA deep learning accelerator
ELVM
Pragma based annotations
Support for accelerator execution
Front-end to work on Tensors
Schedule tensors on accelerator clusters

MAERI (GaTech)
LegUp (UToronto)

SiFive
10/10/18
Web page: aviral.lab.asu.edu
Summary and Next Steps

Highlights
- DATE 2018 and DAC 2018 papers on CGRA
  - LASER: A Hardware/Software Approach to Accelerate Complicated Loops on CGRAs [DATE 2018]
  - RAMP: Resource-Aware Application Mapping on CGRAs [DAC 2018]
- Released the first version of the first open source compiler-simulator toolchain for CGRAs
  - CCF: https://github.com/cmlasu/ccf

Next steps
- Complete simulation model and performance model.
  - Embed energy model (e.g. through McPAT)
- Design space exploration to settle upon the spatially programmable solution and the corresponding dataflow execution.
  - Set the interconnections, RF size, CGRA-SPM to DRAM bandwidth
- Development of a light-weight compiler from tensor flow to the CGRA accelerator.
  - Integrate routines for accelerator execution and software prefetching to TensorFlow library routines.