Energy-Efficient Acceleration of RNNs using CGRA

Aviral Shrivastava, Arizona State University

Arizona State University

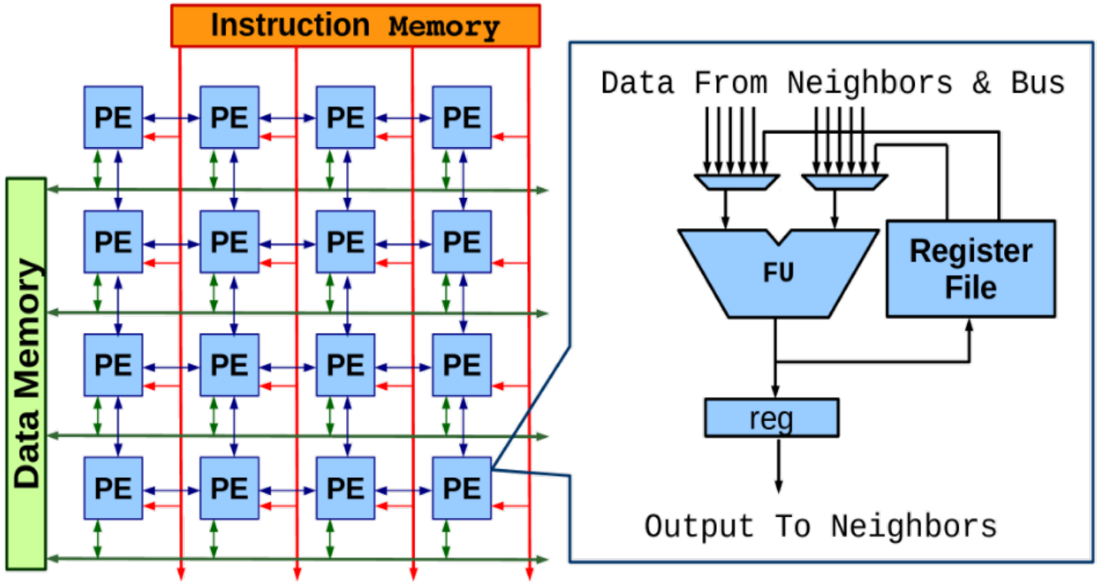
ARM Summit 2018

10/10/18

CML

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

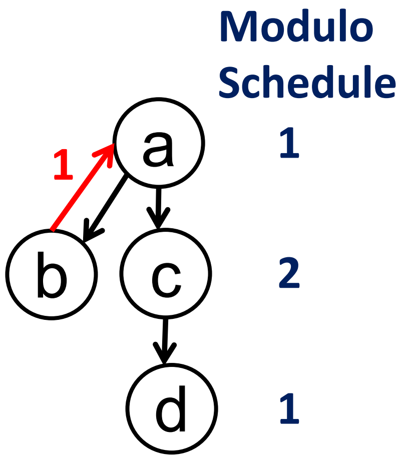


- CGRAs can achieve power-efficiency of several 10s of GOps/Sec per Watt.
  - ADRES CGRA chip, up to 60 GOps/sec per Watt [IMEC, HiPEAC 2008]
  - HyCUBE chip, about 63 MIPS/mW [M. Karunaratne et al., DAC 2017]
- Popular in Embedded Systems and Multimedia [Samsung SRP processor]

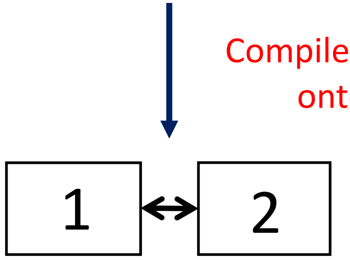
Mapping Applications on CGRAs

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

Sample Loop

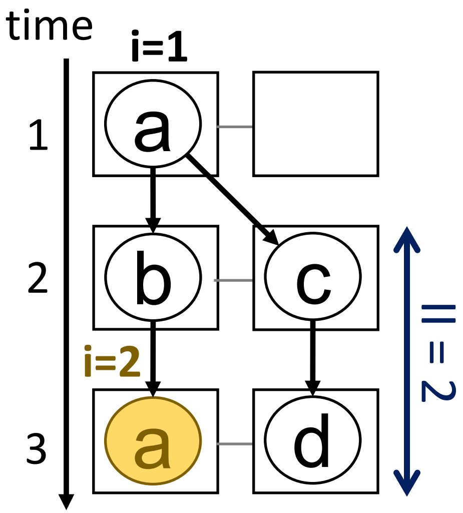


Data Dependency Graph



Compiler Maps DDG onto CGRA

1x2 CGRA



Time-Extended CGRA

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

- 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.

CGRAs Becoming a Hotbed of Research

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

- Efficient Control and Communication Paradigms for Coarse-Grained Spatial Architectures. Michael Pellauer, Angshuman Parashar, Michael Adler, Bushra Ahsan and others. In ACM TC 2015.
- DRAMA: An Architecture for Accelerated Processing Near Memory. Amin Farmahini-Farahani, Jung Ho Ahn, Katherine Morrow and Nam Sung Kim. In IEEE CAL 2015. [UWisc]
- Control Flow Coalescing on a Hybrid Dataflow/von Neumann GPGPU. Dani Voitsechov Yoav Etsion. In MICRO 2015. [Technion, Israel]
- Evaluating Programmable Architectures for Imaging and Vision Applications. Artem Vasilyev, Nikhil Bhagdikar, Ardavan Pedram, Stephen Richardson, Shahar Kvatinsky, Mark Horowitz. In MICRO 2016.
- Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. Yu-Hsin Chen, Joel Emer and Vivienne Sze. In ISCA 2016.
- A space-and energy-efficient code compression/decompression technique for coarse-grained reconfigurable architectures. Bernhard Egger et al. In CGO 2017. [SNU and Samsung]
- Hycube: A cgra with reconfigurable single-cycle multi-hop interconnect. Manupa Karunaratne, Aditi Kulkarni Mohite, Tulika Mitra and Li-Shiuan Peh. In DAC 2017. [NUS]

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.

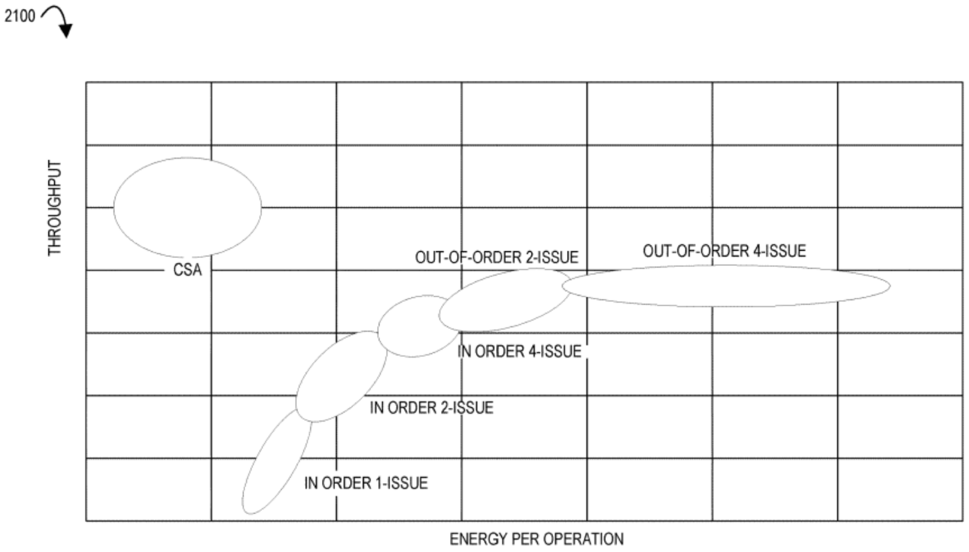
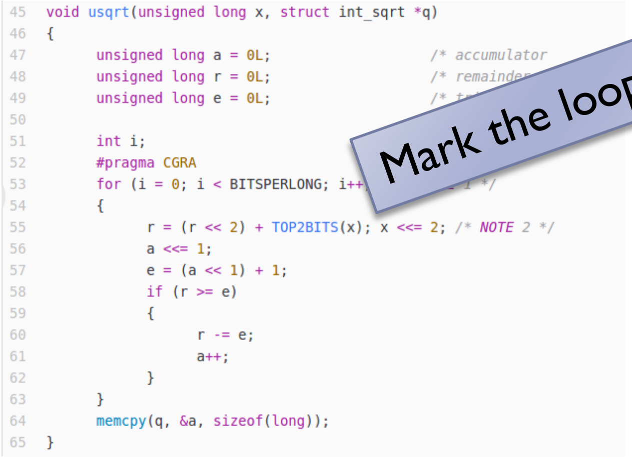


FIG. 21

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

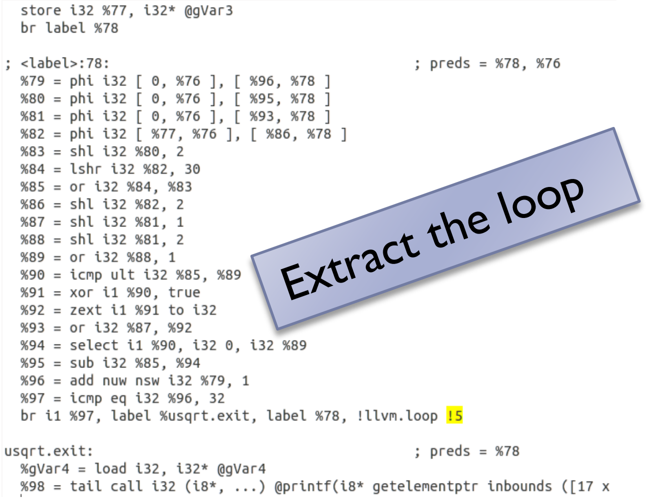
Article: https://www.nextplatform.com/2018/08/30/intels-exascale-dataflow-engine-drops-x86-and-von-neuman/

CCF: CGRA Compiler+simulation Framework



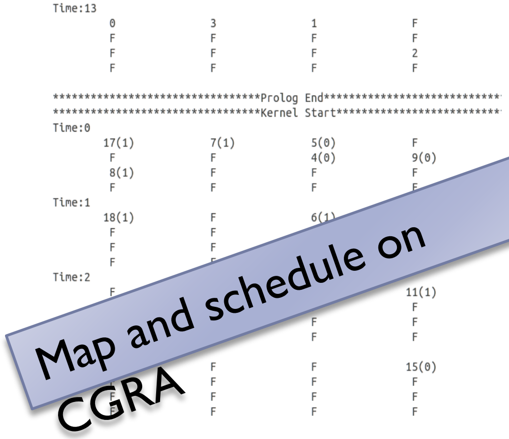
```
void usqrt(unsigned long x, struct int_sqrt *q)
{
      unsigned long a = 0L;                   /* accumulator
      unsigned long r = 0L;                   /* remainder
      unsigned long e = 0L;                   /* t

      int i;
      #pragma CGRA
      for (i = 0; i < BITSPERLONG; i++                 1 */
      {
            r = (r << 2) + TOP2BITS(x); x <<= 2; /* NOTE 2 */
            a <<= 1;
            e = (a << 1) + 1;
            if (r >= e)
            {
                  r -= e;
                  a++;
            }
      }
      memcpy(q, &a, sizeof(long));
}
```

Mark the loops



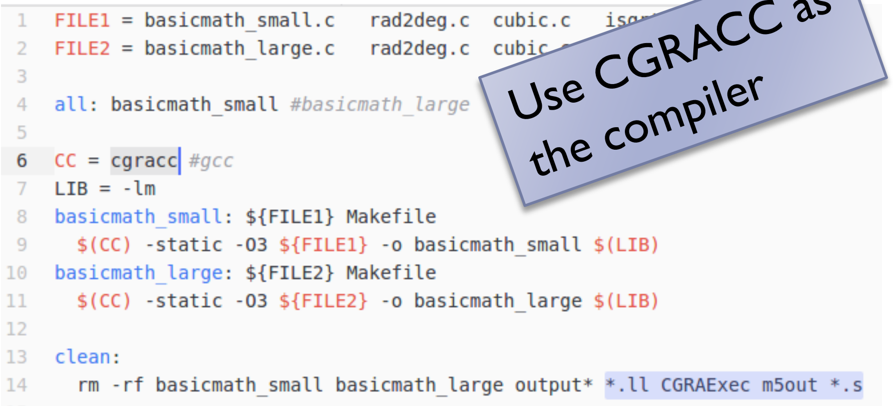
```
  store i32 %77, i32* @gVar3
  br label %78

; <label>:78:                                     ; preds = %78, %76
  %79 = phi i32 [ 0, %76 ], [ %96, %78 ]
  %80 = phi i32 [ 0, %76 ], [ %95, %78 ]
  %81 = phi i32 [ 0, %76 ], [ %93, %78 ]
  %82 = phi i32 [ %77, %76 ], [ %86, %78 ]
  %83 = shl i32 %80, 2
  %84 = lshr i32 %82, 30
  %85 = or i32 %84, %83
  %86 = shl i32 %82, 2
  %87 = shl i32 %81, 1
  %88 = shl i32 %81, 2
  %89 = or i32 %88, 1
  %90 = icmp ult i32 %85, %89
  %91 = xor i1 %90, true
  %92 = zext i1 %91 to i32
  %93 = or i32 %87, %92
  %94 = select i1 %90, i32 0, i32 %89
  %95 = sub i32 %85, %94
  %96 = add nuw nsw i32 %79, 1
  %97 = icmp eq i32 %96, 32
  br i1 %97, label %usqrt.exit, label %78, !llvm.loop !5

usqrt.exit:                                       ; preds = %78
  %gVar4 = load i32, i32* @gVar4
  %98 = tail call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([17 x
```

Extract the loop



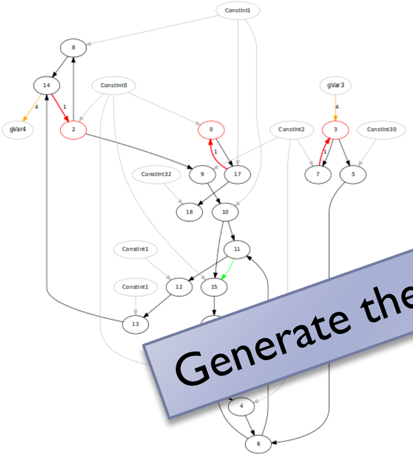
```
Time:13
        0            3            1            F
        F            F            F            F
        F            F            F            2
        F            F            F            F

*********************************Prolog End*****************************
*********************************Kernel Start***************************
Time:0
       17(1)        7(1)         5(0)          F
        F            F           4(0)         9(0)
        8(1)         F            F            F
        F            F            F            F
Time:1
       18(1)         F           6(1)
        F            F
        F            F
        F
Time:2
        F                                    11(1)
                                               F
                                  F            F
                                  F            F

                     F            F          15(0)
                     F            F            F
                     F            F            F
                     F            F            F
```

Map and schedule on CGRA

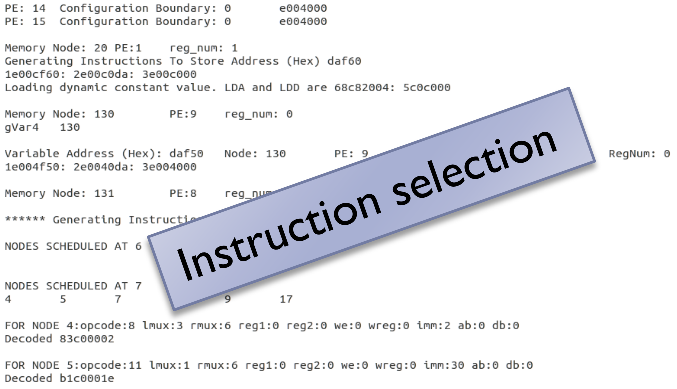


```
FILE1 = basicmath_small.c   rad2deg.c  cubic.c   isq
FILE2 = basicmath_large.c   rad2deg.c  cubic.

all: basicmath_small #basicmath_large

CC = cgracc #gcc
LIB = -lm
basicmath_small: ${FILE1} Makefile
  $(CC) -static -O3 ${FILE1} -o basicmath_small $(LIB)
basicmath_large: ${FILE2} Makefile
  $(CC) -static -O3 ${FILE2} -o basicmath_large $(LIB)

clean:
  rm -rf basicmath_small basicmath_large output* *.ll CGRAExec m5out *.s
```

Use CGRACC as the compiler

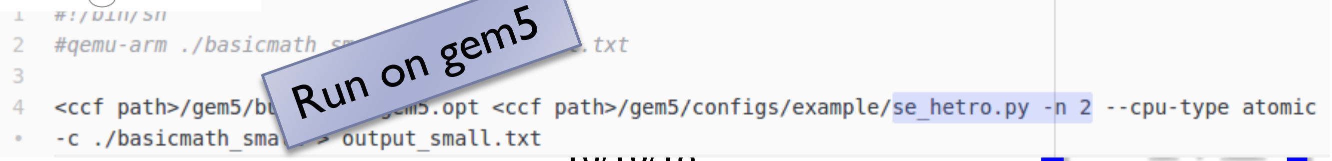


Generate the DFG



```
PE: 14  Configuration Boundary: 0       e004000
PE: 15  Configuration Boundary: 0       e004000

Memory Node: 20 PE:1    reg_num: 1
Generating Instructions To Store Address (Hex) daf60
1e00cf60: 2e00c0da: 3e00c000
Loading dynamic constant value. LDA and LDD are 68c82004: 5c0c000

Memory Node: 130        PE:9    reg_num: 0
gVar4   130

Variable Address (Hex): daf50   Node: 130       PE: 9                               RegNum: 0
1e004f50: 2e0040da: 3e004000

Memory Node: 131        PE:8    reg_num

****** Generating Instructi

NODES SCHEDULED AT 6


NODES SCHEDULED AT 7
4       5       7                9       17

FOR NODE 4:opcode:8 lmux:3 rmux:6 reg1:0 reg2:0 we:0 wreg:0 imm:2 ab:0 db:0
Decoded 83c00002

FOR NODE 5:opcode:11 lmux:1 rmux:6 reg1:0 reg2:0 we:0 wreg:0 imm:30 ab:0 db:0
Decoded b1c0001e
```

Instruction selection



```
#!/bin/sh
#qemu-arm ./basicmath_sm                    .txt

<ccf path>/gem5/bu        em5.opt <ccf path>/gem5/configs/example/se_hetro.py -n 2 --cpu-type atomic
-c ./basicmath_sma     > output_small.txt
```

Run on gem5

- llvm 5.0 and gem5 as foundation
- Public Release: https://github.com/cmlasu/ccf
- Video: https://www.youtube.com/watch?v=iGV1lHsjy4w

Web page: aviral.lab.asu.edu

Our Recent Work on Compiling for CGRAs

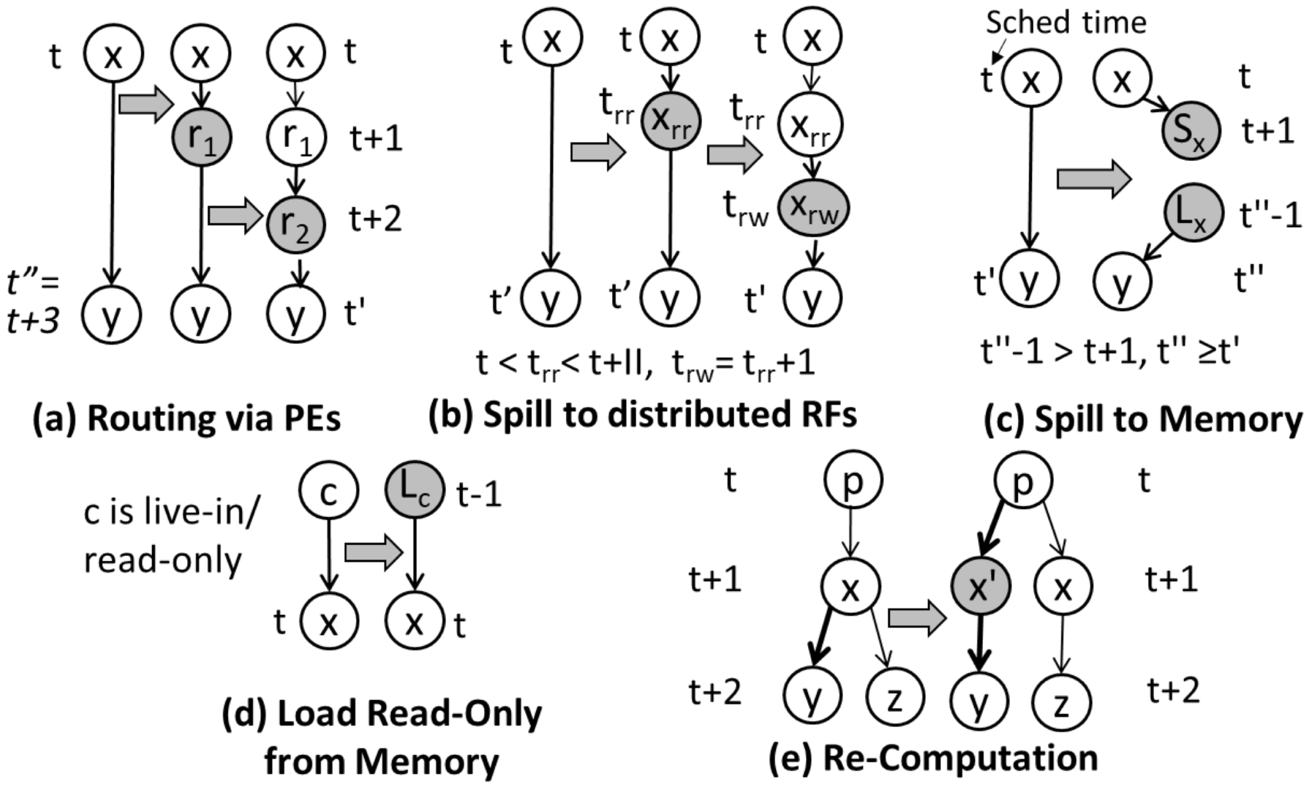
- Efficient software pipeline
  - EPIMap: Using Epimorphism to Map Applications on CGRAs. Mahdi Hamzeh, Aviral Shrivastava and Sarma Vrudhula. In DAC 2012.

- Efficiently using distributed register files
  - REGIMap: Register-aware Application Mapping on CGRAs. Mahdi Hamzeh, Aviral Shrivastava and Sarma Vrudhula. In DAC 2013.
- Register file organization
  - URECA: A Compiler Technique to Manage Unified Register File for CGRAs. Shail Dave, Mahesh Balasubramanian and Aviral Shrivastava. In DATE 2018.

- Efficient mapping of if-then-else's
  - LASER: A Hardware/Software Approach to Accelerate Complicated Loops on CGRAs. Mahesh Balasubramanian, Shail Dave, Aviral Shrivastava and Reiley Jeyapaul. In DATE 2018.

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

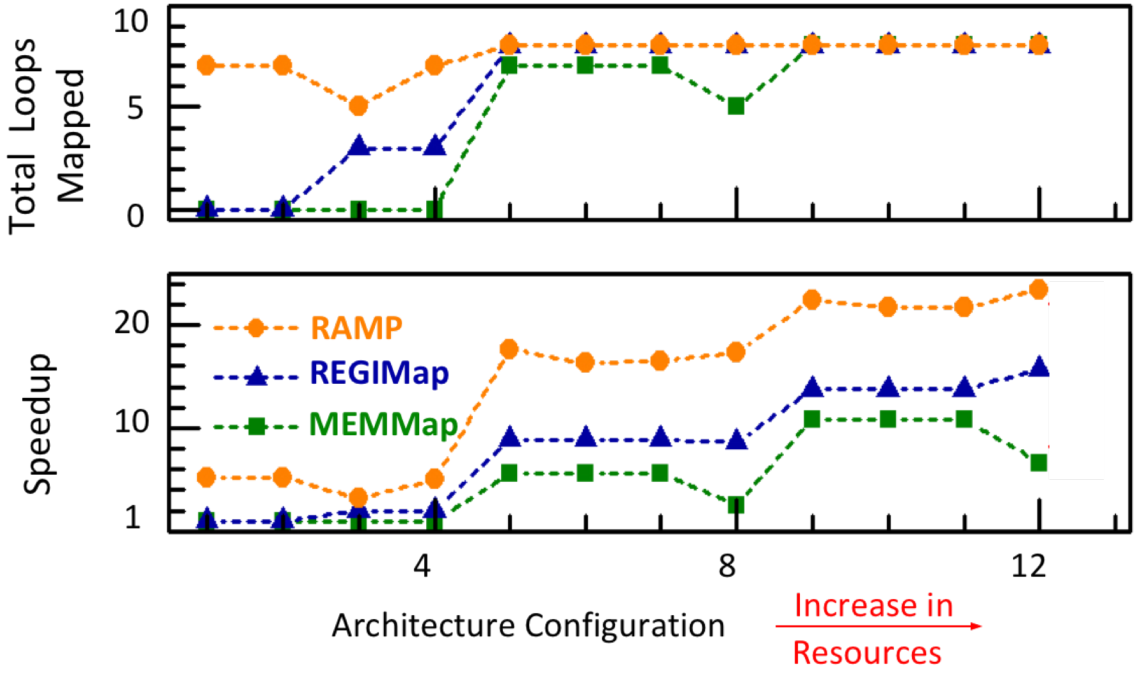
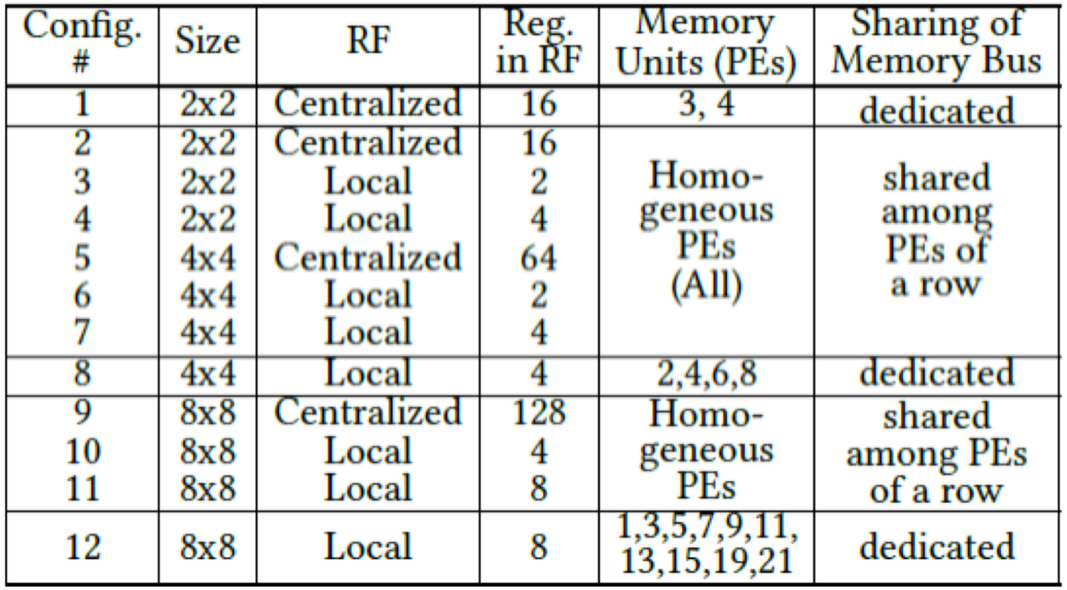


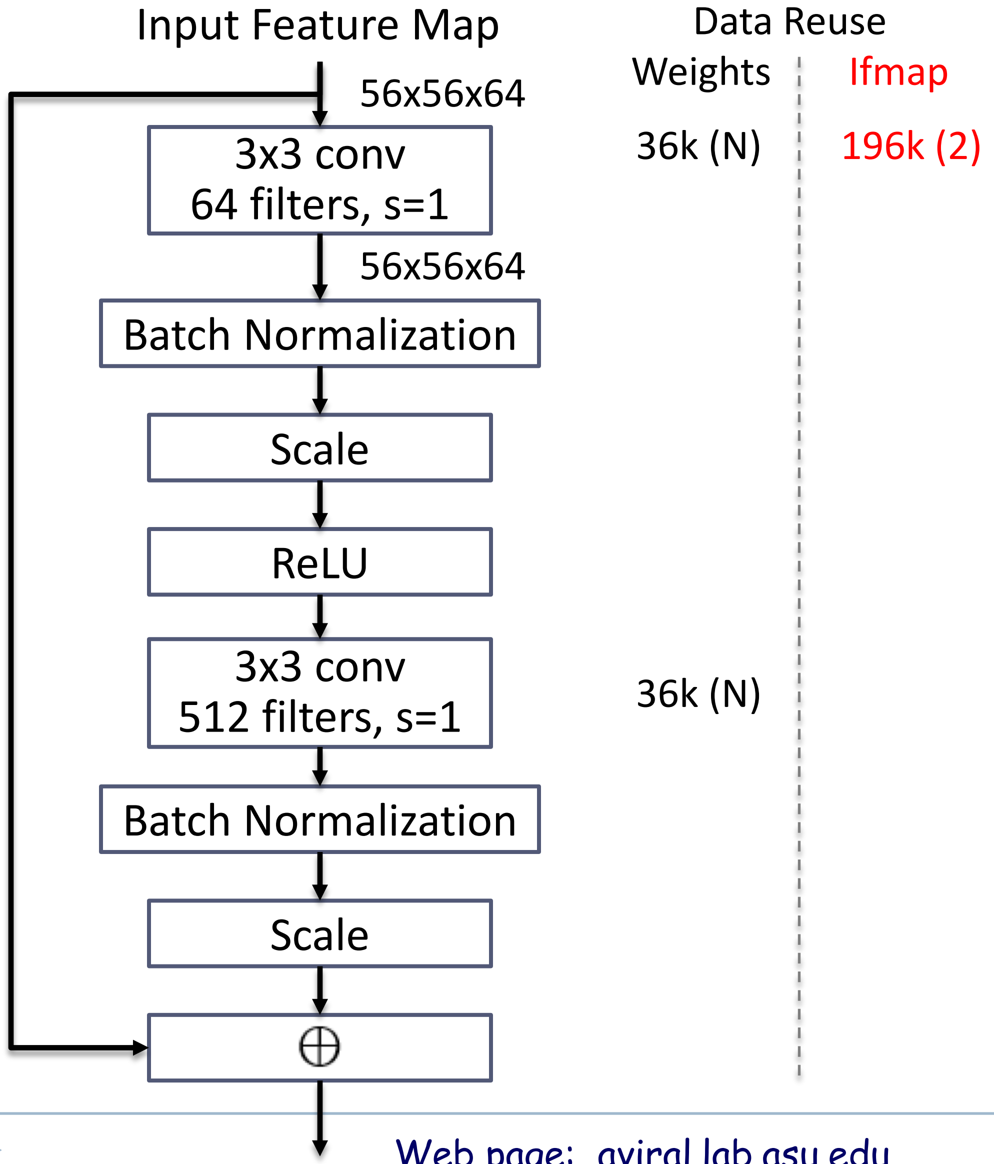
Table 1: Specifications of CGRA architecture configurations



Config. #	Size	RF	Reg. in RF	Memory Units (PEs)	Sharing of Memory Bus
1	2x2	Centralized	16	3, 4	dedicated
2	2x2	Centralized	16	Homo-geneous PEs (All)	shared among PEs of a row
3	2x2	Local	2		
4	2x2	Local	4		
5	4x4	Centralized	64		
6	4x4	Local	2		
7	4x4	Local	4		
8	4x4	Local	4	2,4,6,8	dedicated
9	8x8	Centralized	128	Homo-geneous PEs	shared among PEs of a row
10	8x8	Local	4		
11	8x8	Local	8		
12	8x8	Local	8	1,3,5,7,9,11, 13,15,19,21	dedicated

- 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.

Residual Blocks from ResNet-18



Input Feature Map

56x56x64

3x3 conv
64 filters, s=1

56x56x64

Batch Normalization

Scale

ReLU

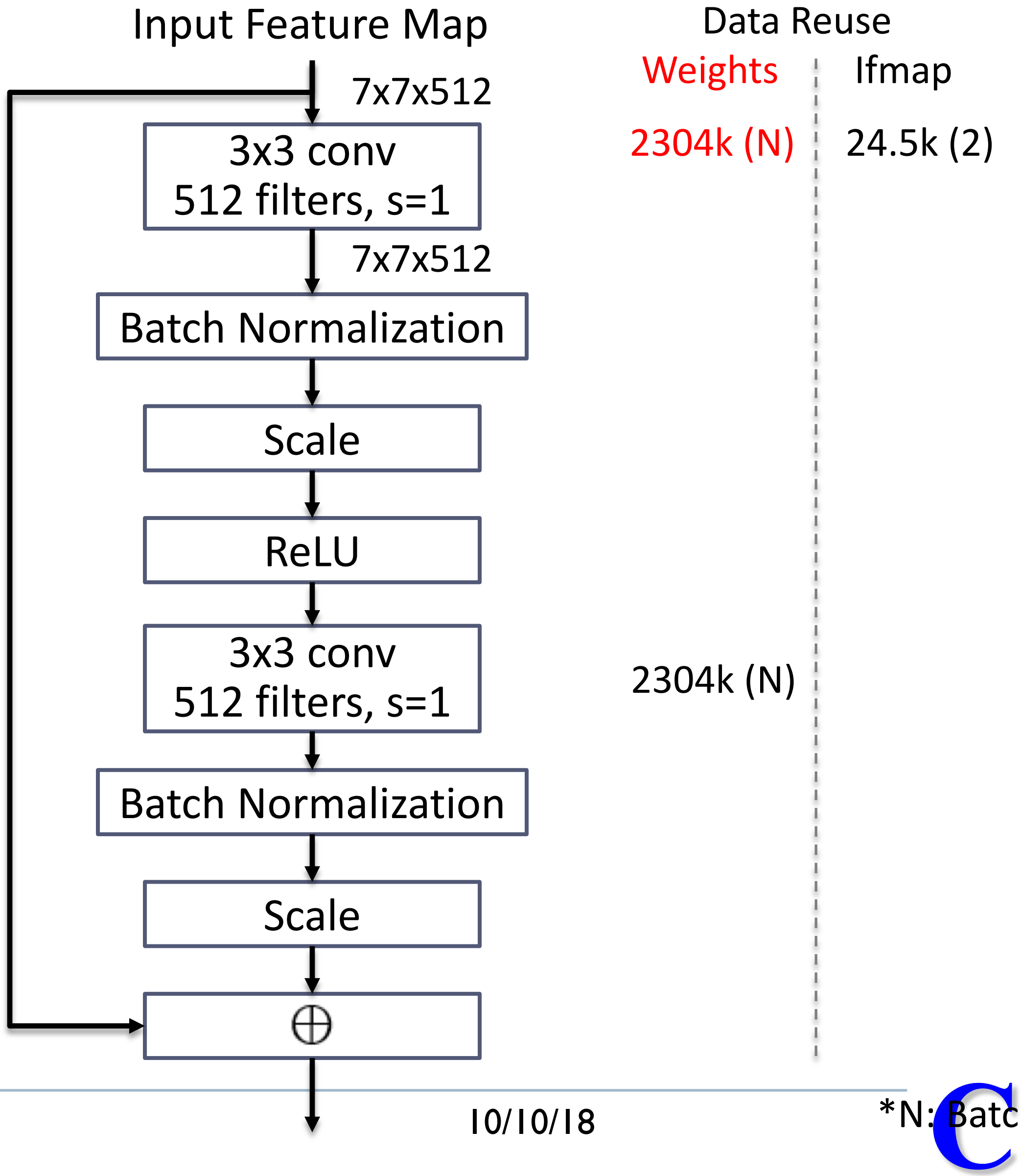
3x3 conv
512 filters, s=1

Batch Normalization

Scale

⊕

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



Input Feature Map

7x7x512

3x3 conv
512 filters, s=1

7x7x512

Batch Normalization

Scale

ReLU

3x3 conv
512 filters, s=1

Batch Normalization

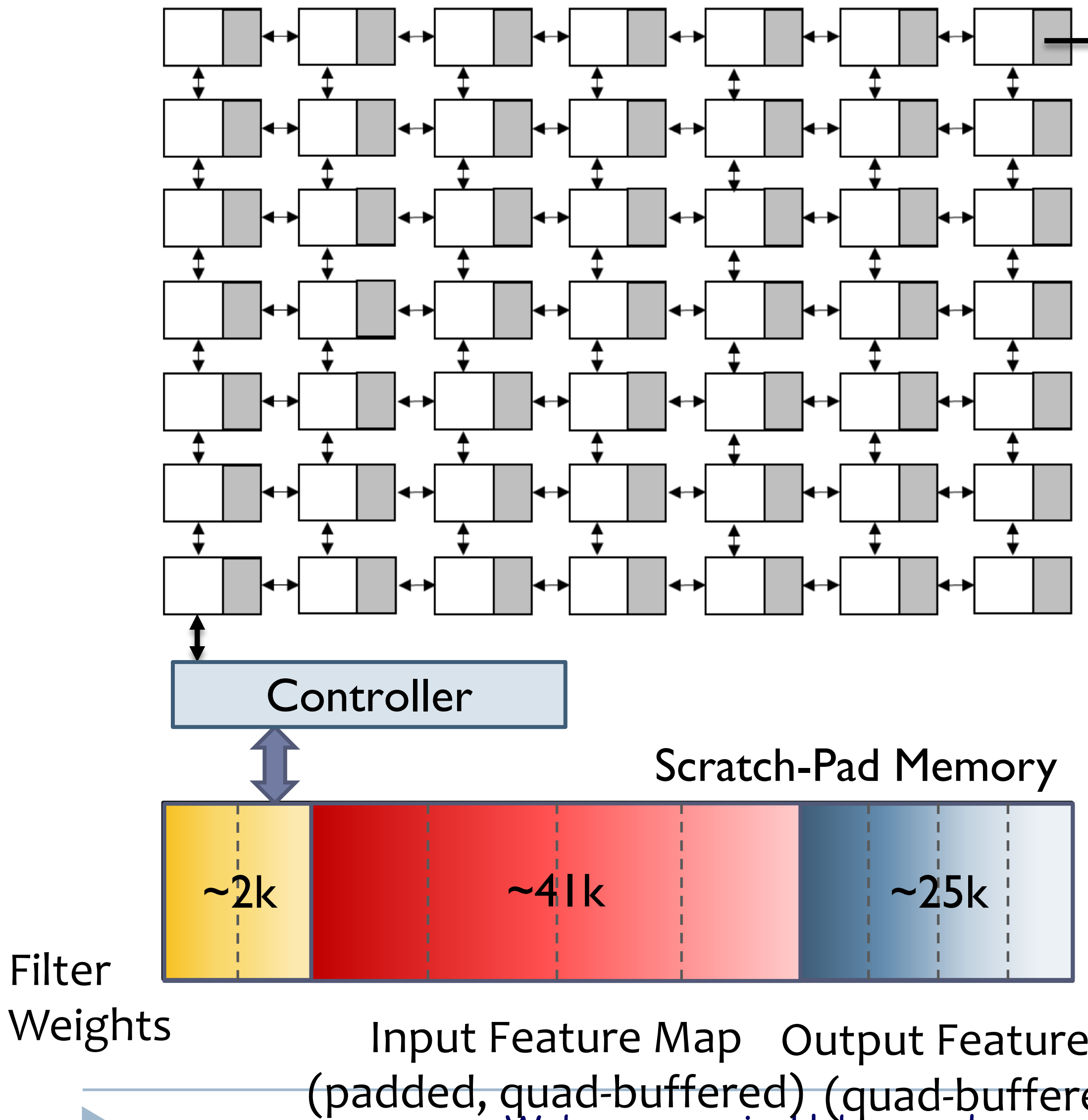
Scale

⊕

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

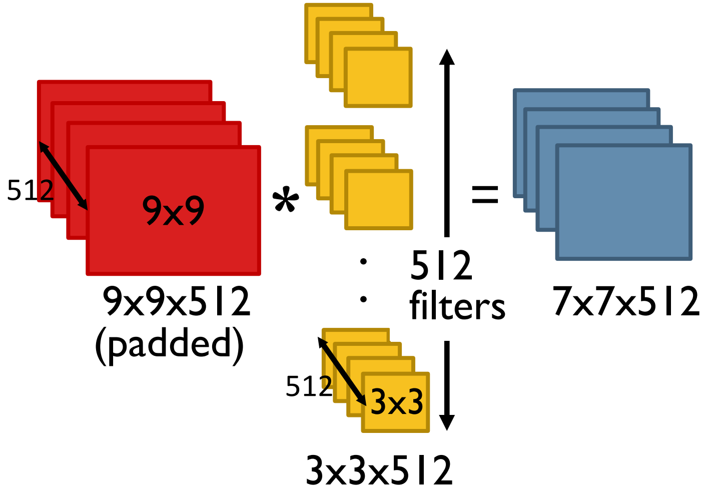
*N: Batch size

Execution Mechanism for Inference


Controller
Scratch-Pad Memory
~2k
~41k
~25k
Filter Weights
Input Feature Map (padded, quad-buffered)
Output Feature Map (quad-buffered)

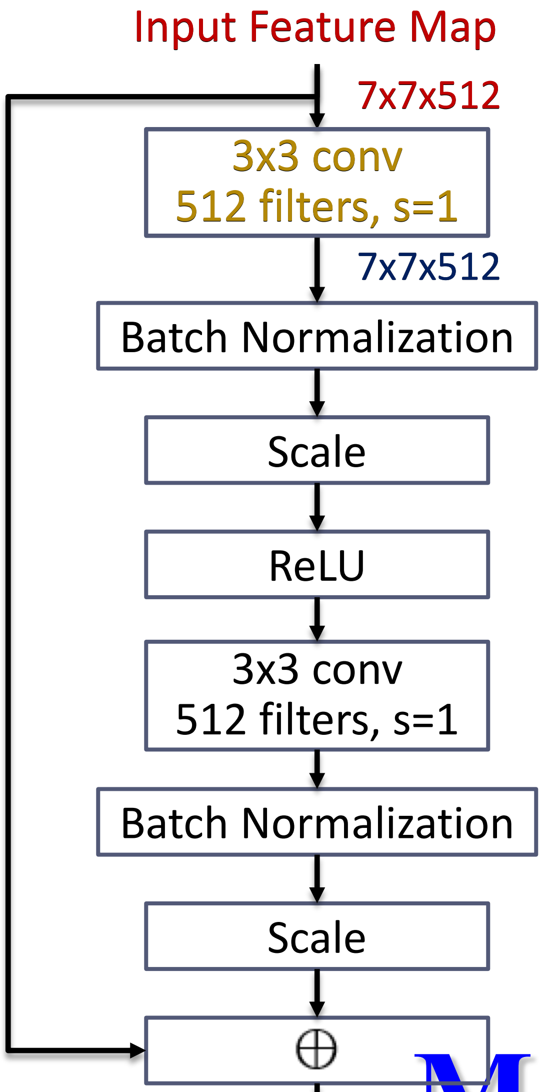
RF Size: 0.64 kB

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



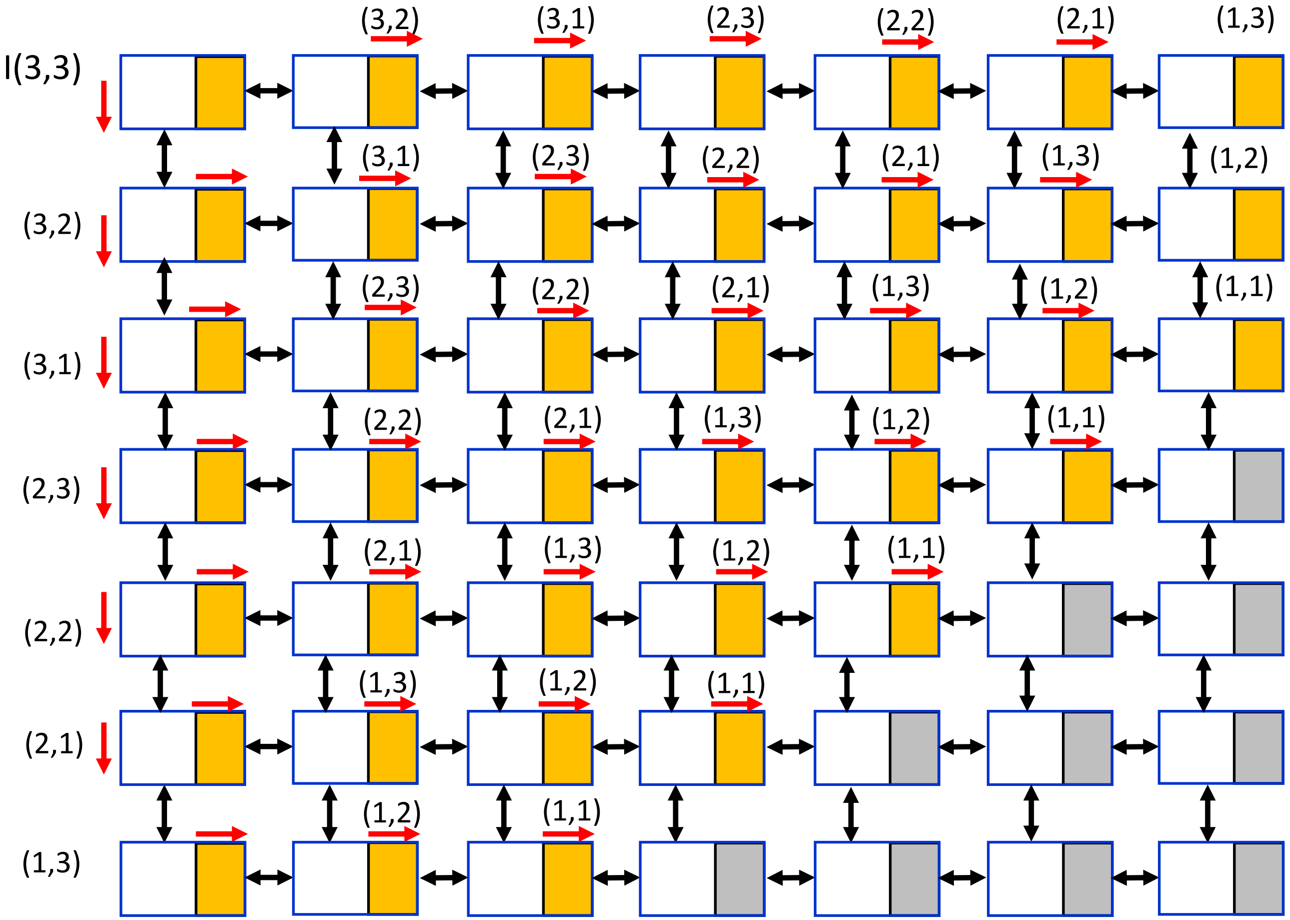
Dark shaded areas indicate buffered data (in use by PEs)

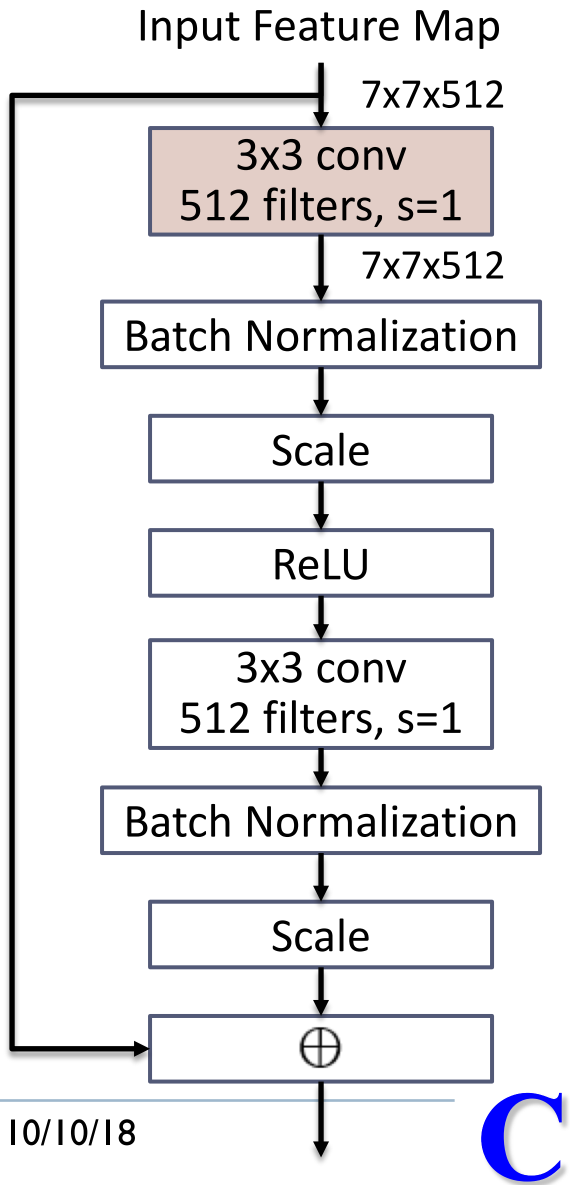
Light shade indicates prefetching/write-back



Executing Residual Block on CGRA

- Pre-load filter weights to RF





Input Feature Map

7x7x512

3x3 conv
512 filters, s=1

7x7x512

Batch Normalization

Scale

ReLU

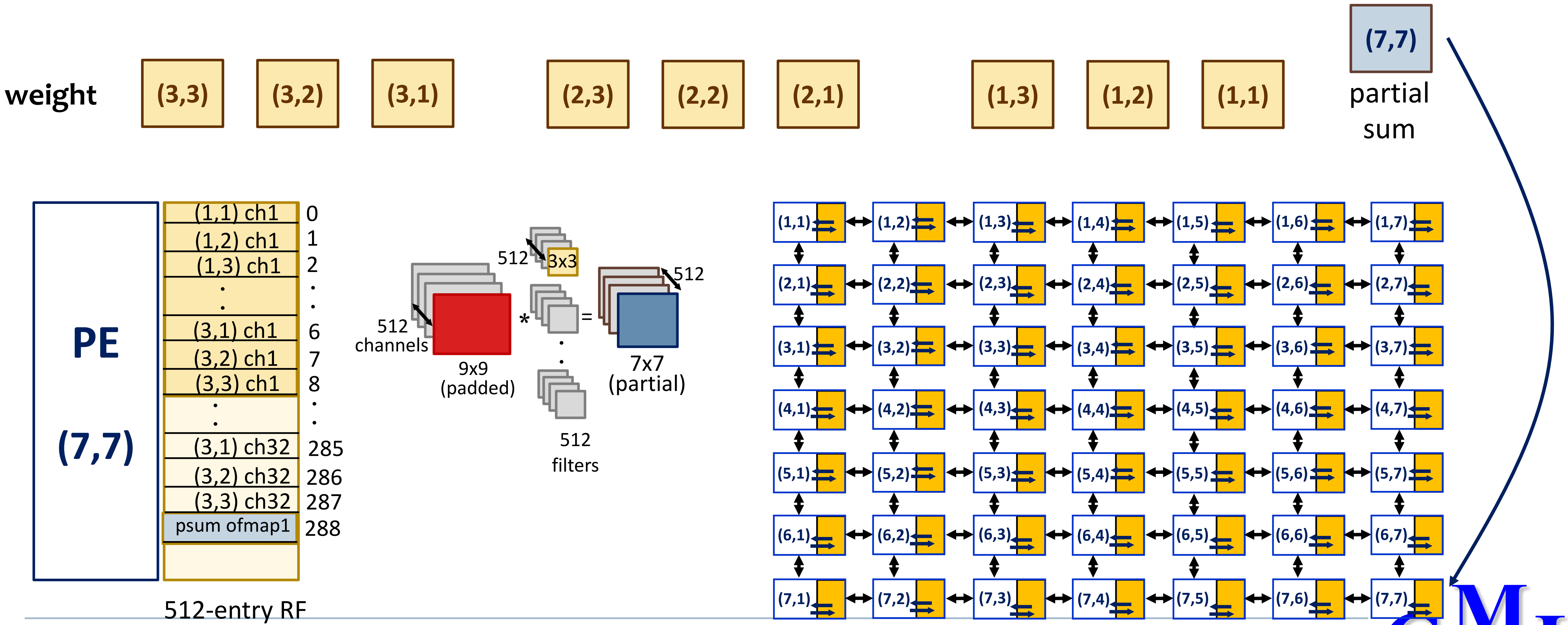
3x3 conv
512 filters, s=1

Batch Normalization

Scale

⊕

Output Stationary Dataflow for Convolutions



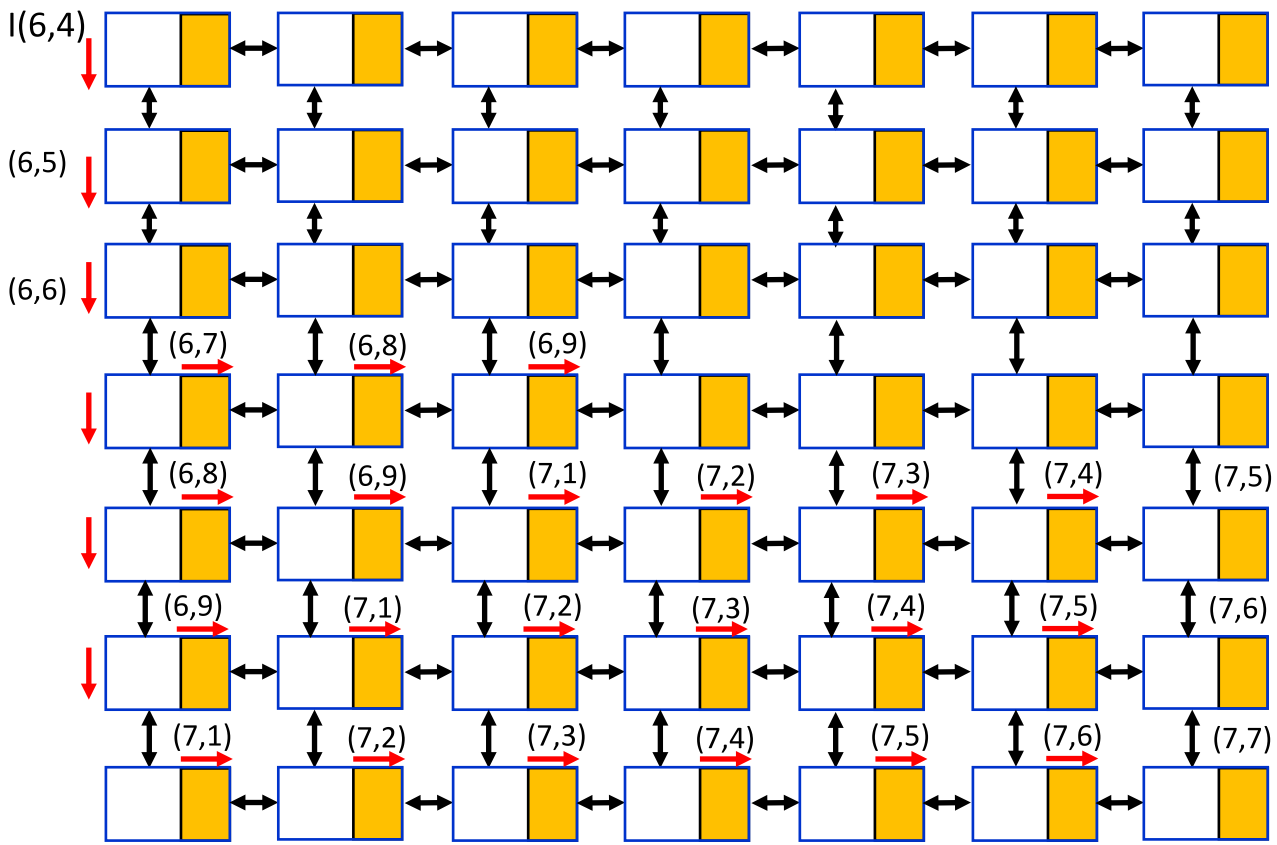
(1,1) ch1	0
(1,2) ch1	1
(1,3) ch1	2
. .	. .
(3,1) ch1	6
(3,2) ch1	7
(3,3) ch1	8
. .	. .
(3,1) ch32	285
(3,2) ch32	286
(3,3) ch32	287
psum ofmap1	288

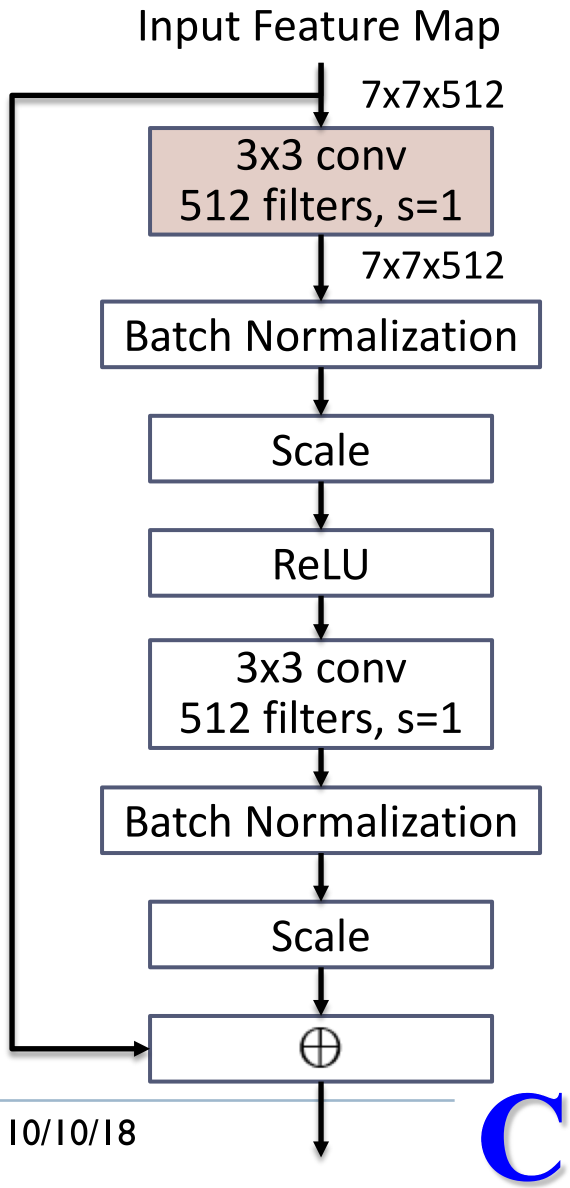
512-entry RF

(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)

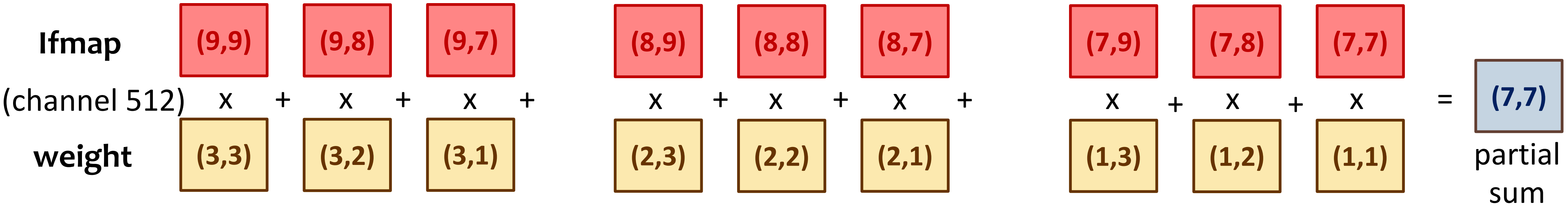
Streaming Feature Maps for MACs

- Input feature map streamed through PEs

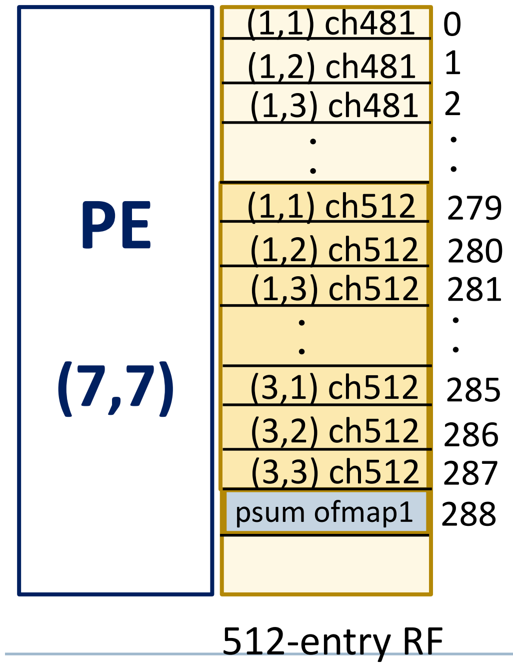




Output Stationary Dataflow for Convolutions



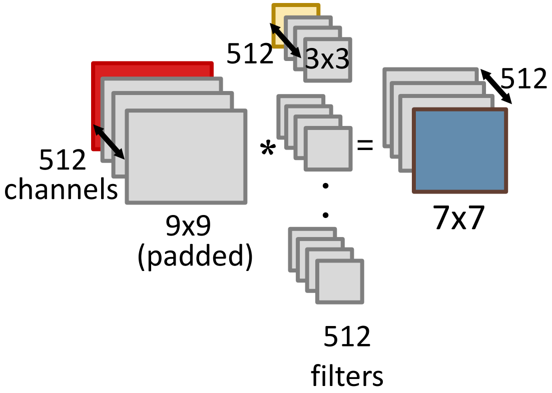
Ifmap	(9,9)	(9,8)	(9,7)	(8,9)	(8,8)	(8,7)	(7,9)	(7,8)	(7,7)	
(channel 512)	x +	x +	x +	x +	x +	x +	x +	x +	x	= (7,7)
weight	(3,3)	(3,2)	(3,1)	(2,3)	(2,2)	(2,1)	(1,3)	(1,2)	(1,1)	partial sum

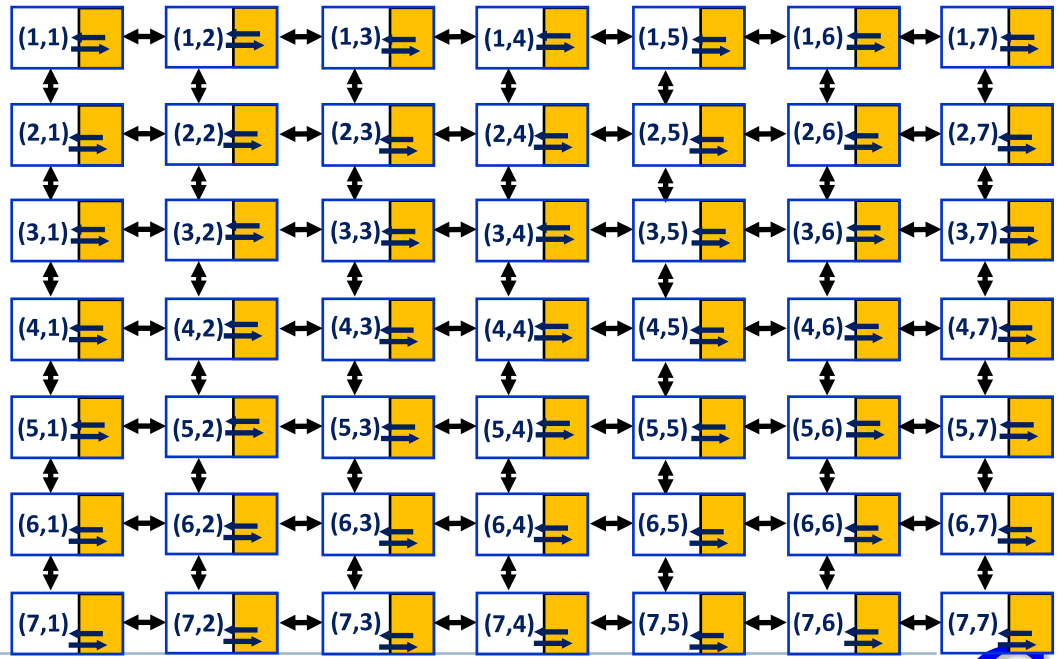


PE (7,7)

(1,1) ch481	0
(1,2) ch481	1
(1,3) ch481	2
. .	. .
(1,1) ch512	279
(1,2) ch512	280
(1,3) ch512	281
. .	. .
(3,1) ch512	285
(3,2) ch512	286
(3,3) ch512	287
psum ofmap1	288

512-entry RF

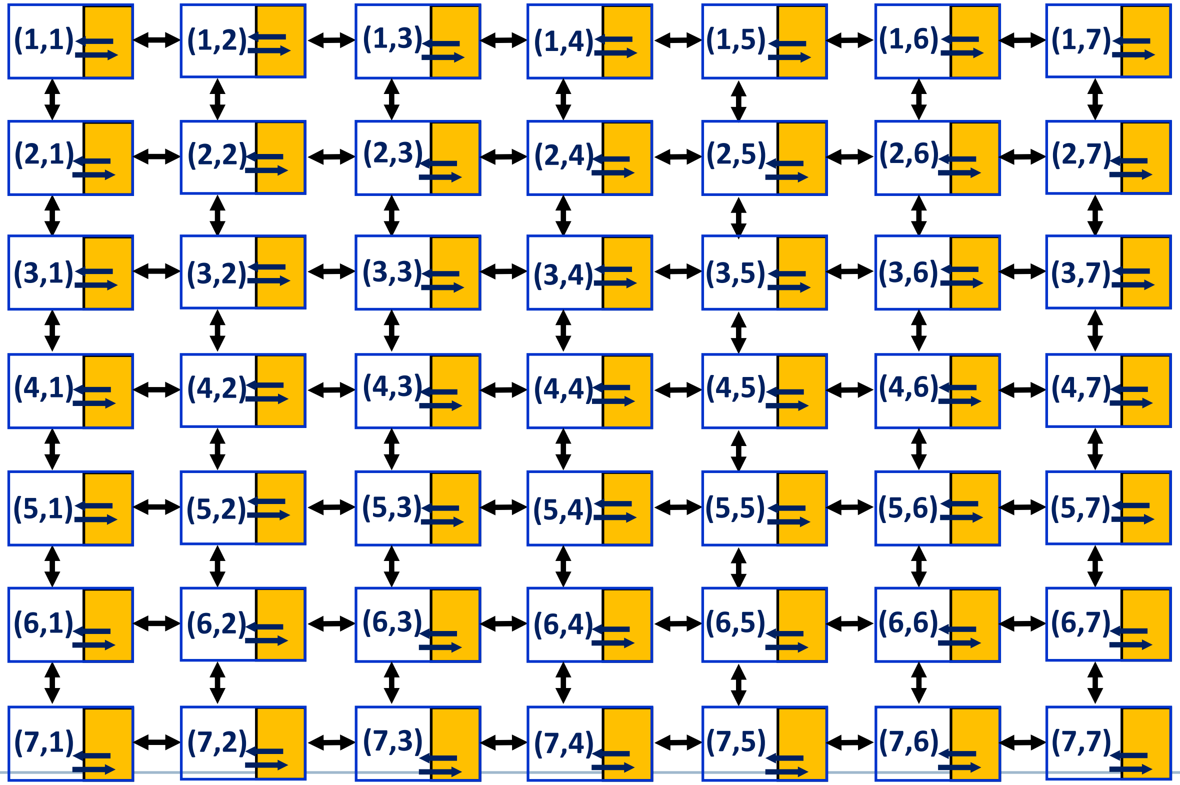


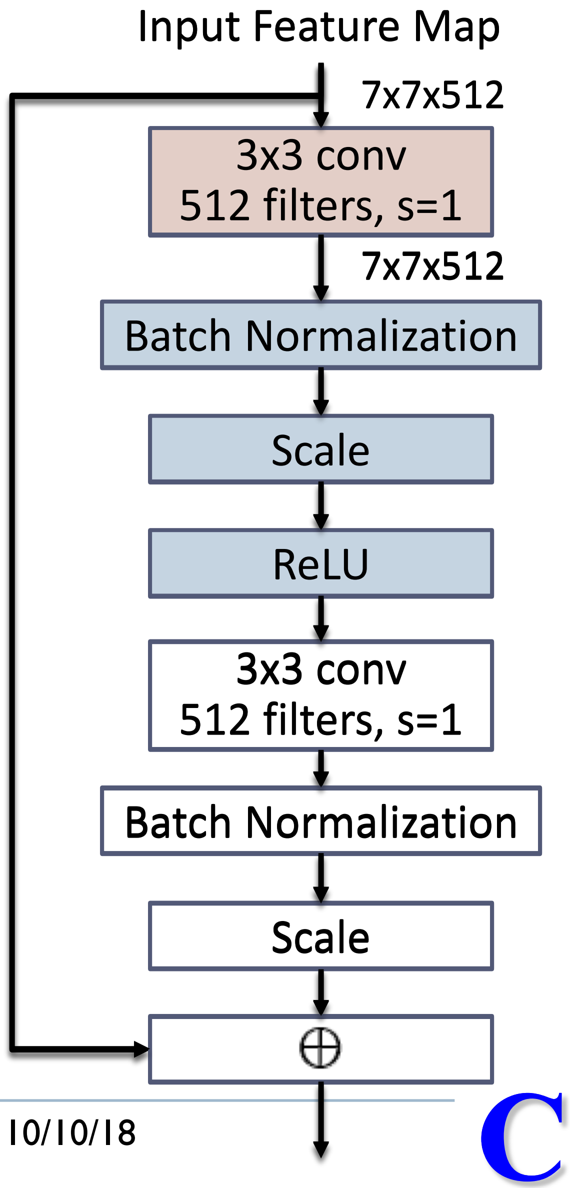


(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)
(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
(6,1)	(6,2)	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
(7,1)	(7,2)	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

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

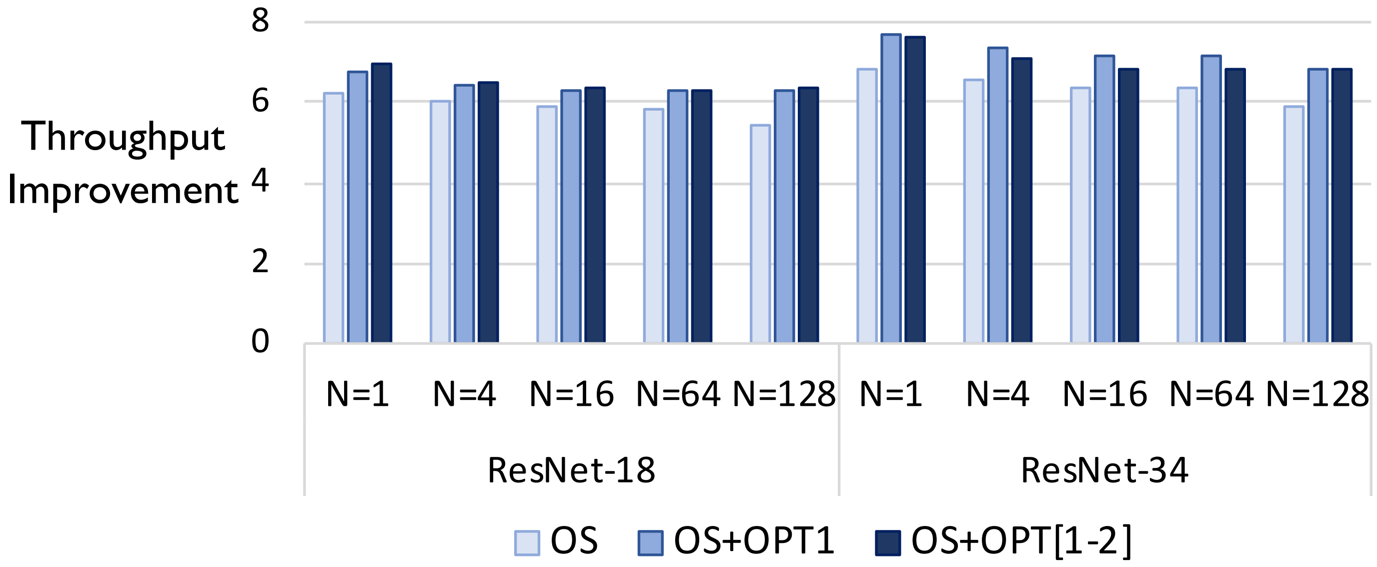




Experimental Setup

- 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



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

TensorFlow (Tensor-Graph)

Pragma based annotations →

Support for accelerator execution →

Front-end to work on Tensors Schedule tensors on accelerator clusters →

CCF CML

VTA (UW)

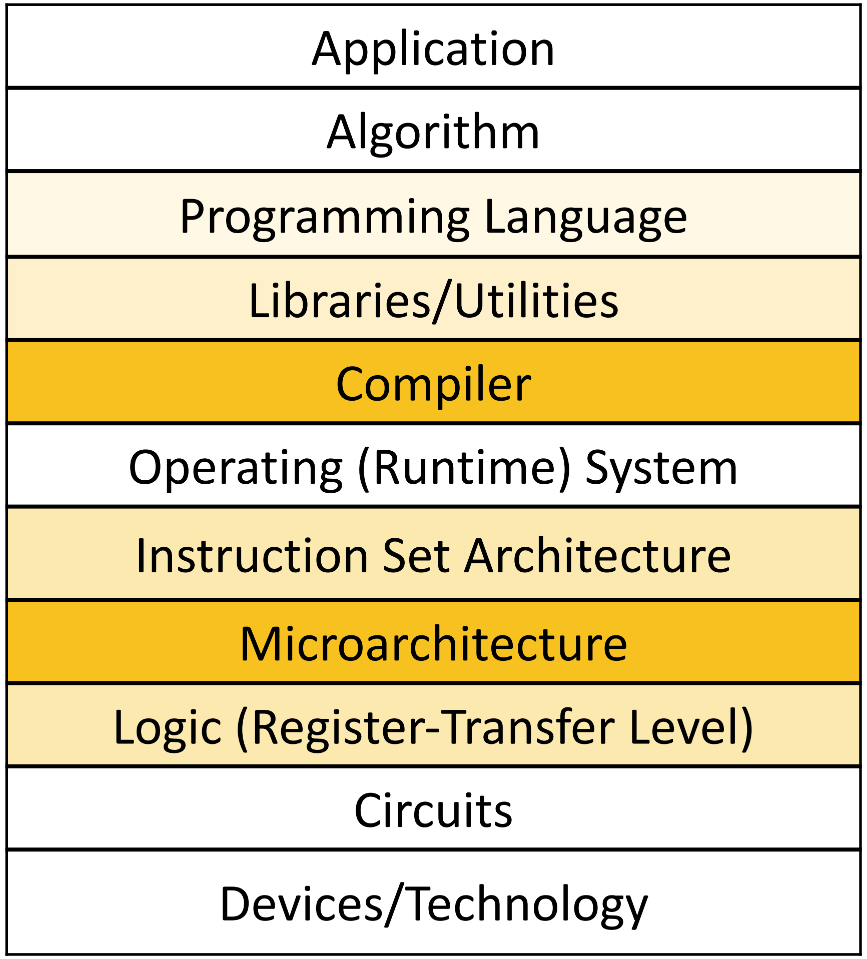
VTA deep learning accelerator

LLVM COMPILER INFRASTRUCTURE

CCF CML

gem5

MAERI (GaTech)   LegUp (UToronto)



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

SiFive

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.