CREATING HIGH PERFORMANCE APPLICATIONS WITH INTEL'S FPGA SDK FOR OPENCL™

Presenter: Andrew Ling, Ph.D., Machine Learning Engineering Manager

Authors: Andrew Ling Ph.D., Davor Capalija Ph.D., Utku Aydonat Ph.D., Shane O'Connell, Gordon Chiu

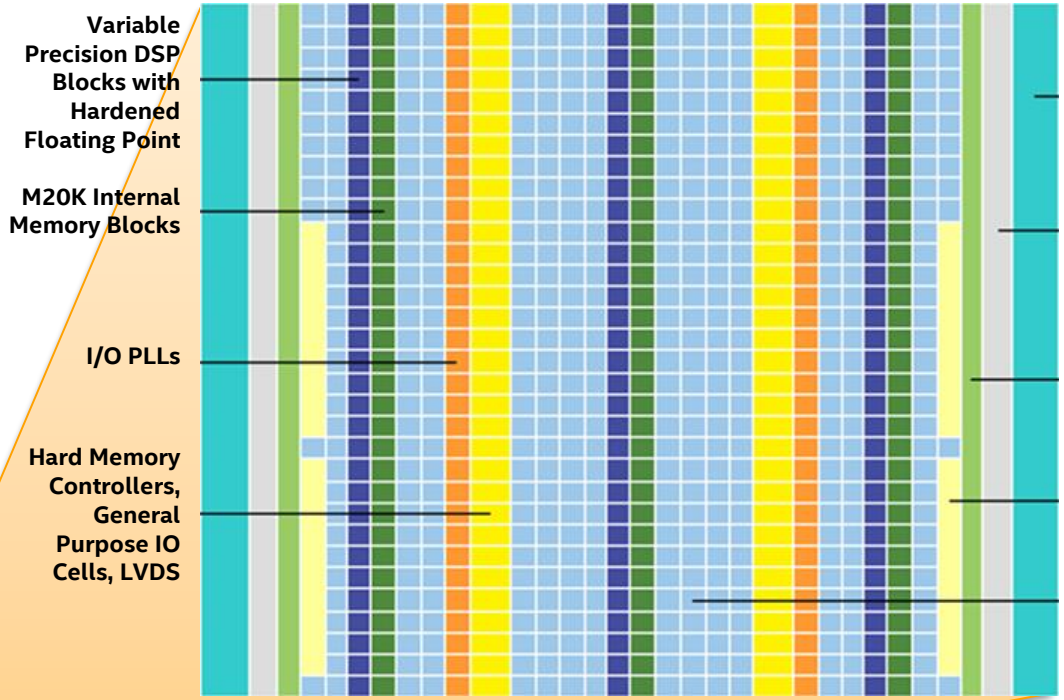
Disclaimer

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, visit **Intel Performance Benchmark Limitations**.

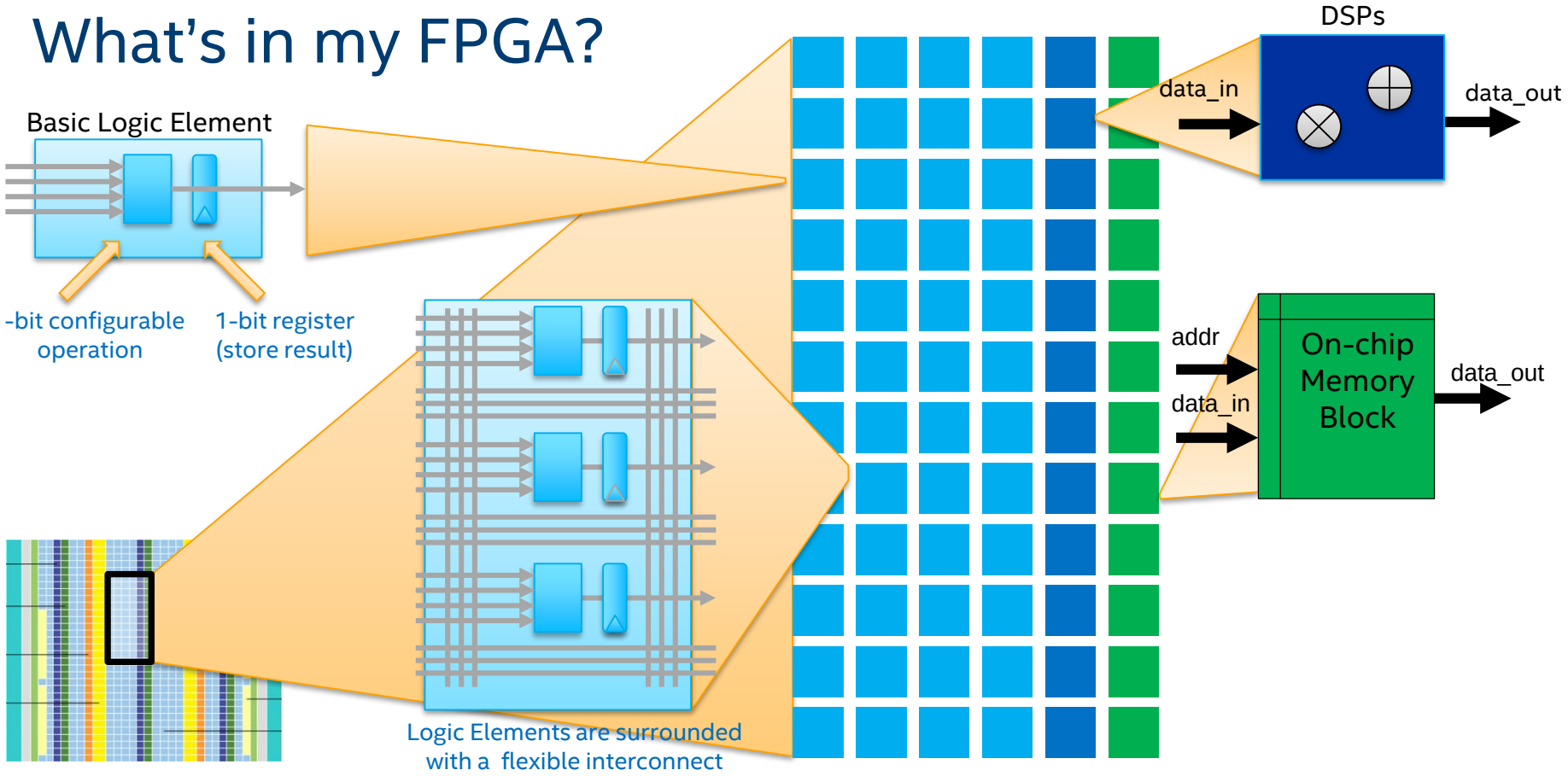
Some Results have been simulated and are provided for informational purposes only. Results were derived using simulations run on an architecture simulator. Any difference in system hardware or software design or configuration may affect actual performance.

Intel does not control or audit the design or implementation of third party benchmarks or websites referenced in this document. Intel encourages all of its customers to visit the referenced websites or others where similar performance benchmarks are reported and confirm whether the referenced benchmarks are accurate and reflect performance of systems available for purchase.

What's in my FPGA?



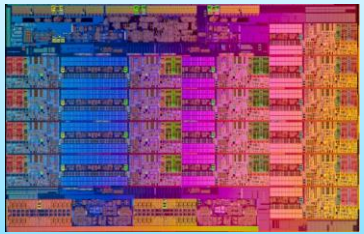
What's in my FPGA?

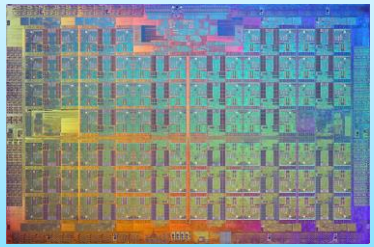


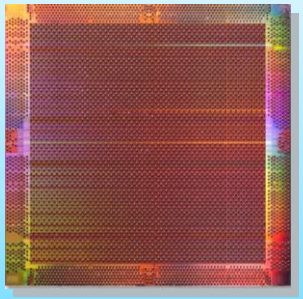
Performance in the Data Centre

- Towards more parallelism through spatial computing

Coarse Grain Parallelism → *Fine Grain Parallelism*







Intel® Xeon® processor E7 v4 product family: up to 24 Cores

Intel® Xeon® Phi Processor Family: up to 72 Cores

Intel® Stratix 10: up to 5510 equivalent logic elements

ACHIEVING 1TFLOP PERFORMANCE ON ARRIA 10 WITH OPENCL

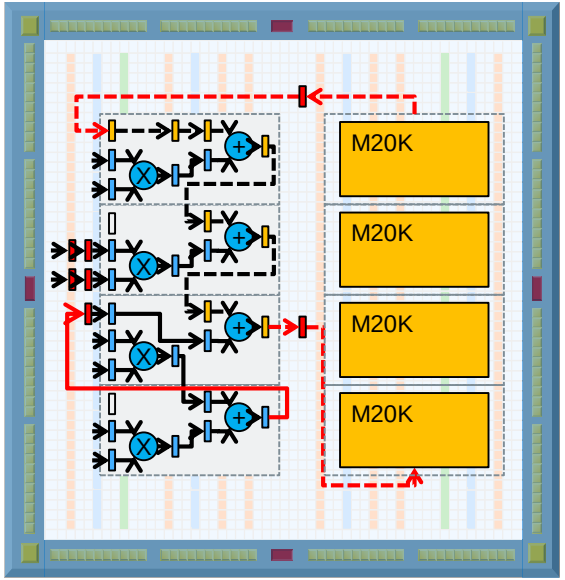
Why are FPGAs great for SGEMM? (Intel® Arria® 10)

1 TFLOP floating point performance in mid-range part

- 35W total device power
- FP32 DSP Support - **Use every DSP, every clock cycle compute spatially**

8 TB/s internal memory bandwidth to keep the state on chip!

- Exceeds available external bandwidth by **factor of 50**
- Random access, low latency (2 clks)
- **Place all data in on-chip memory compute temporally**



Fine-grained & low latency between compute and memory

Insights for best possible performance for SGEMM

Architecture can map to the algorithm

Regular 2D-based spatial architectures map well to FPGA

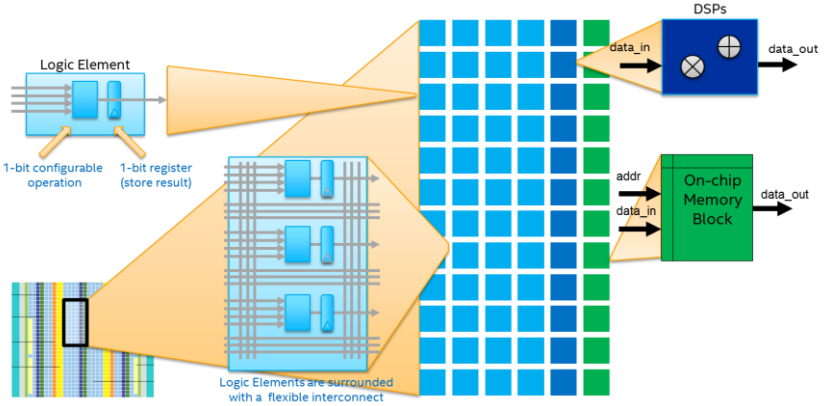
On-die memory leads to efficient data processing

MAP ARCHITECTURE TO ALGORITHM

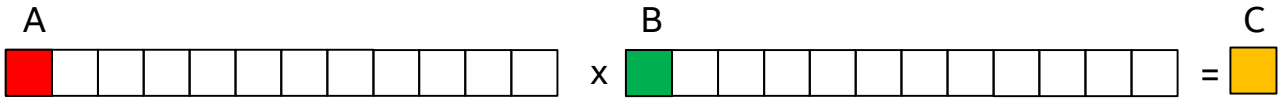
Insight #1:
Map Architecture to Algorithm

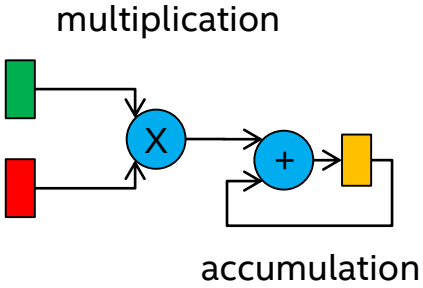
Local memory architecture and register connectivity reconfigurable

- Can create custom data reordering optimizations
- Can create custom caches!



Vector multiplication

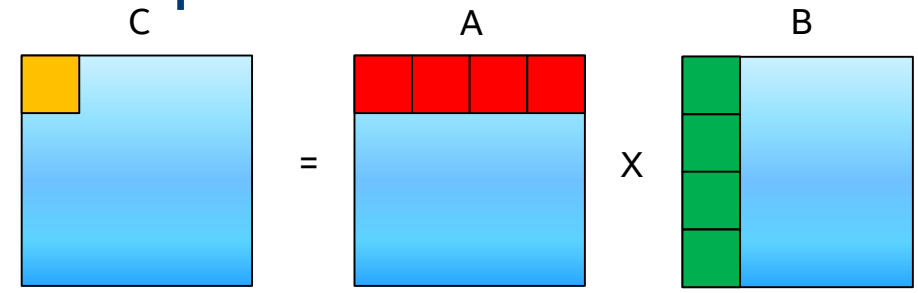


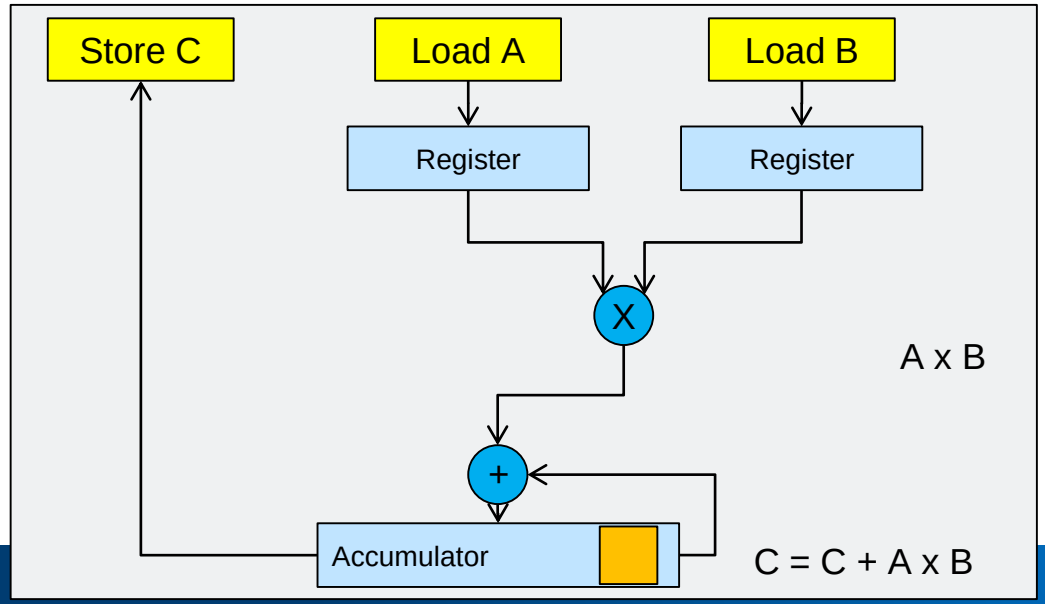


Common technique:
multiply-and-accumulate

basic DSP mode on A10

Matrix Multiplication: Basic



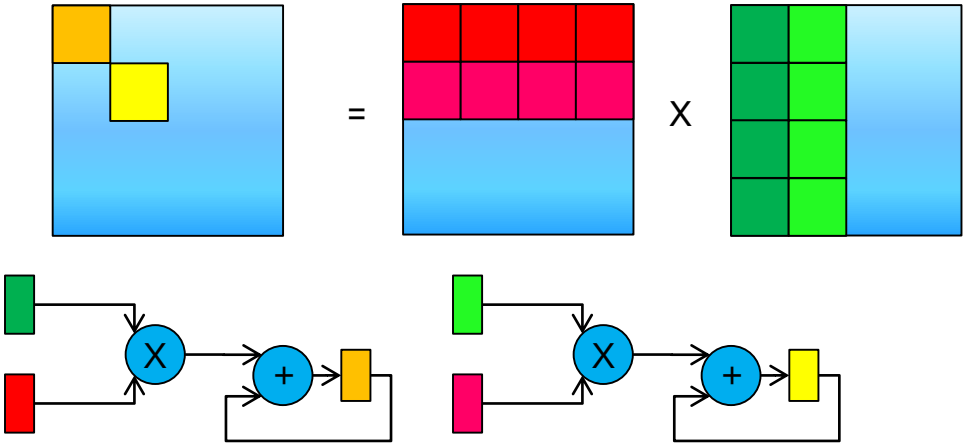


single matrix element

Improving performance

Parallel computation: multiply more vectors in parallel

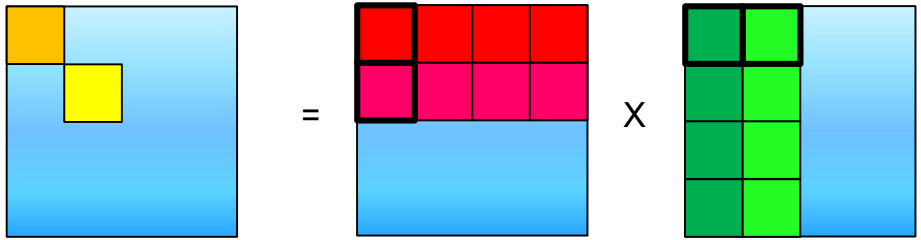
- More multipliers and adders

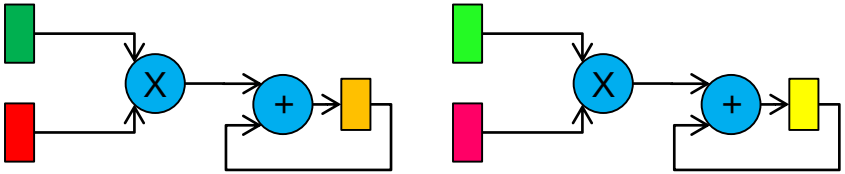


Improving performance

Parallel computation: multiply more vectors in parallel

- More multipliers and adders





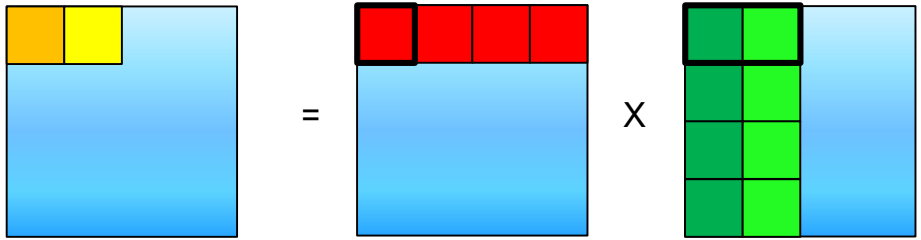
Need 4 loads every cycle
(from 4 vectors)
to feed 2 Multiply-accumulate blocks (MACs)

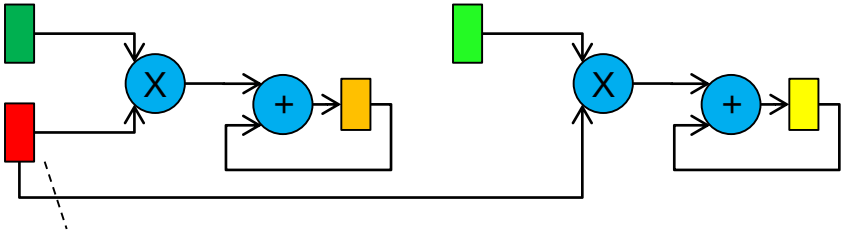
Load : MAC = 2

Improving performance

Data reuse: **multiply blocks (not individual vectors)**

- Better use of off-chip bandwidth





Reuse and feed to two MACs

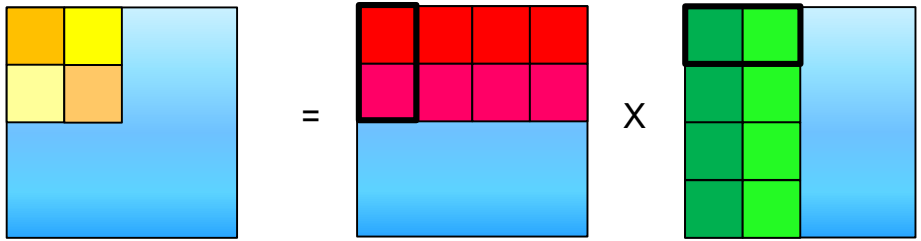
Need 3 loads every cycle
(from 3 vectors)
to feed 2 MACs

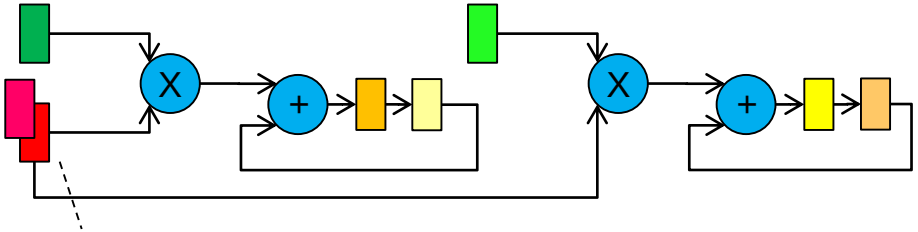
Load : MAC = 1.5

Improving performance

Data reuse: multiply blocks **and interleave**

- Better use of off-chip bandwidth





Reuse and feed to two MACs, and interleave

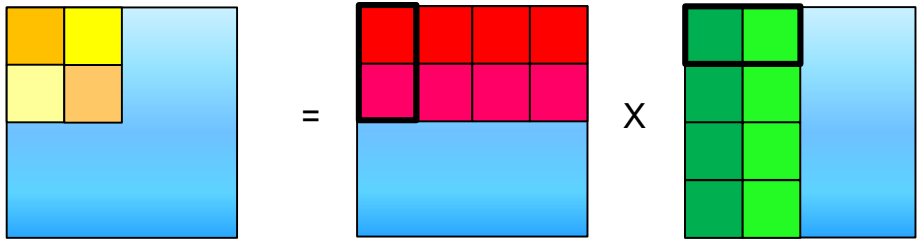
Need 4 loads every 2 cycles
(from 4 vectors)
to feed 2 MACs

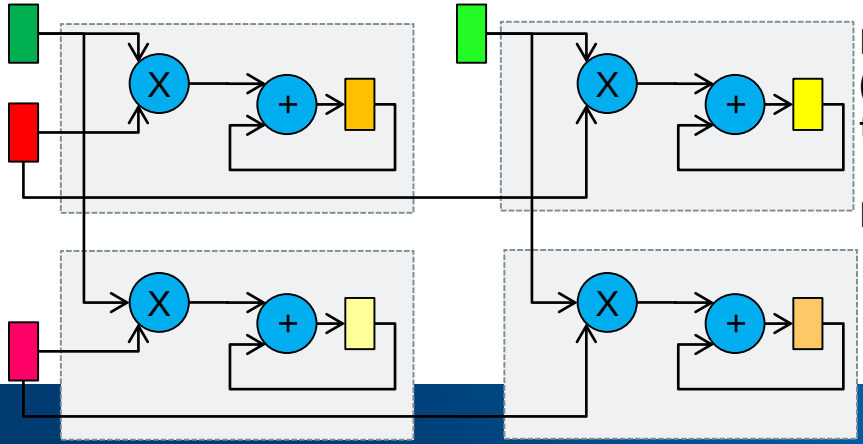
Load : MAC = 1

Systolic Array

2 dimensional array of processing elements (PEs)

- Regular structure, localized processing and storage





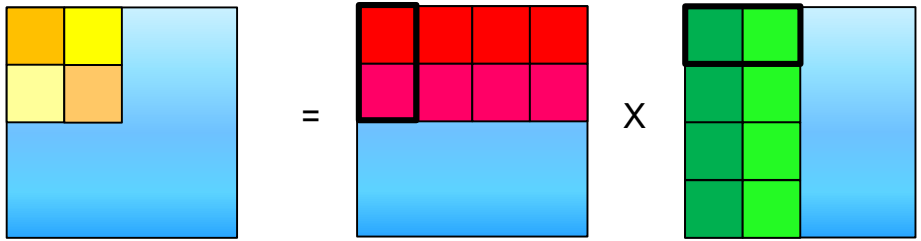
Need 4 loads every cycle
(from 4 vectors)
to feed 4 MACs

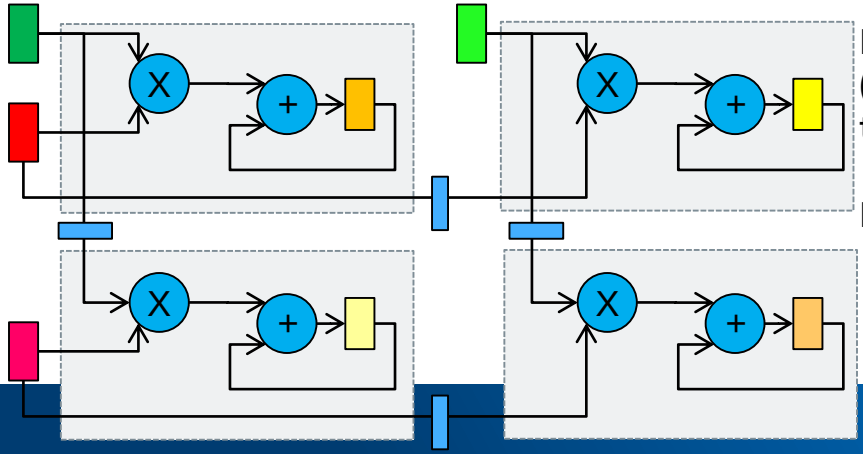
Load : MAC = 1

Systolic Array

Forwarding pipelines instead of fan-outs

- Also called daisy chains



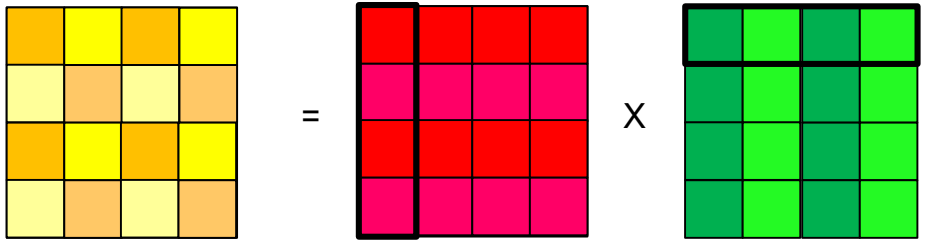


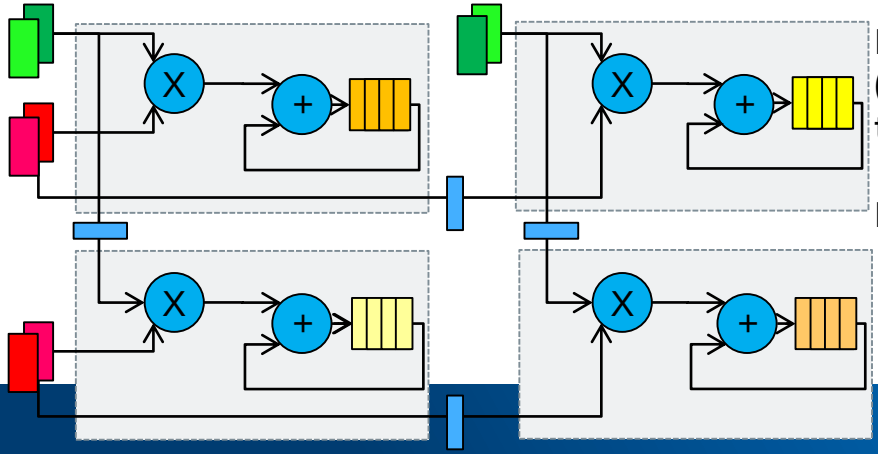
Need 4 loads every cycle
(from 4 vectors)
to feed 4 MACs

Load : MAC = 1

Systolic Array

And then interleave in both dimensions





Need 8 loads every 4 cycles
(from 8 vectors)
to feed 4 MACs

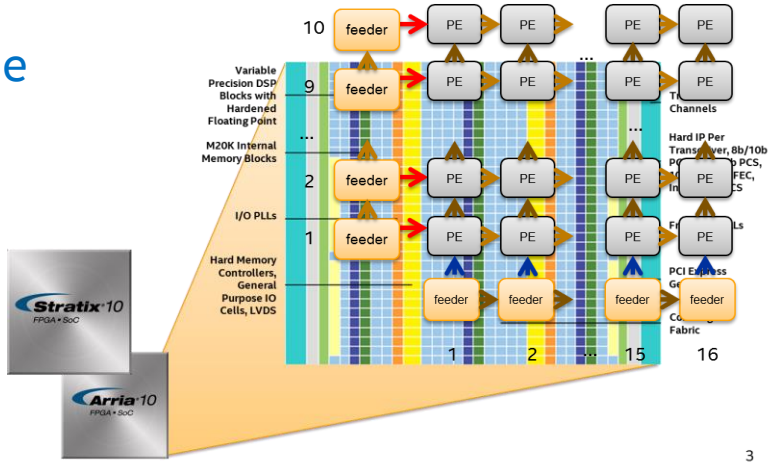
Load : MAC = 0.5

CREATE SPATIALLY REGULAR ARCHITECTURE

Insight #2:
Create spatially regular architecture

Systolic array maps well to 2D Reconfigurable Logic plane

- Makes job “easy” for hardware compiler tools
- Be mindful of interface placement and architecture interaction with interfaces



3

PE, Feed and Drain Arrays in OpenCL

Every PE is a kernel

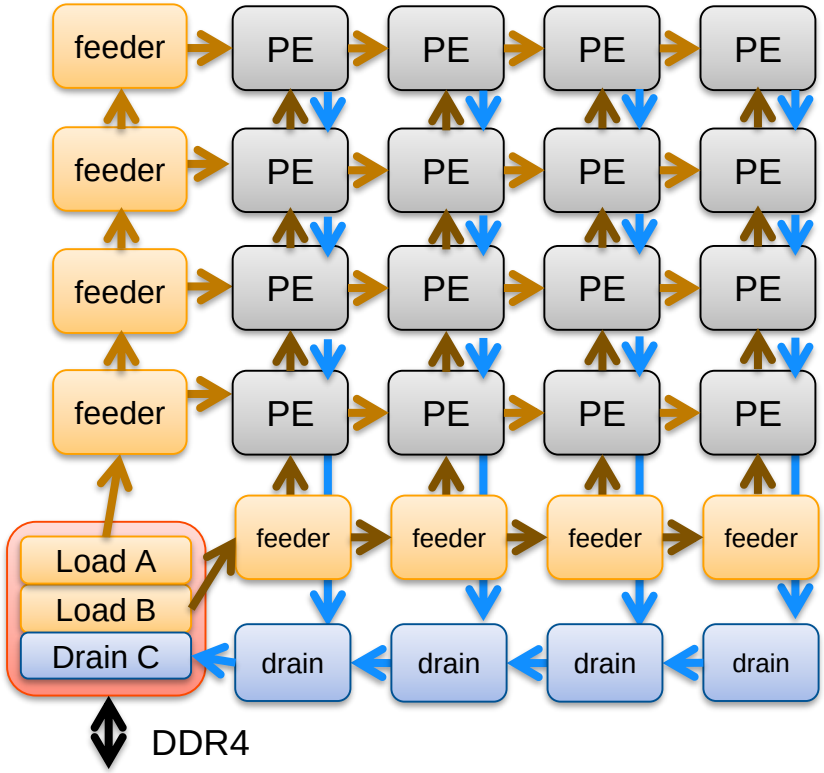
Every feed/drain is a kernel

Communicate via OpenCL channels

- Vendor specific extensions
- Elastic, latency-insensitive, ***allows for concurrent execution and data sharing***

Spatial data-flow model

- Similar to Khan networks



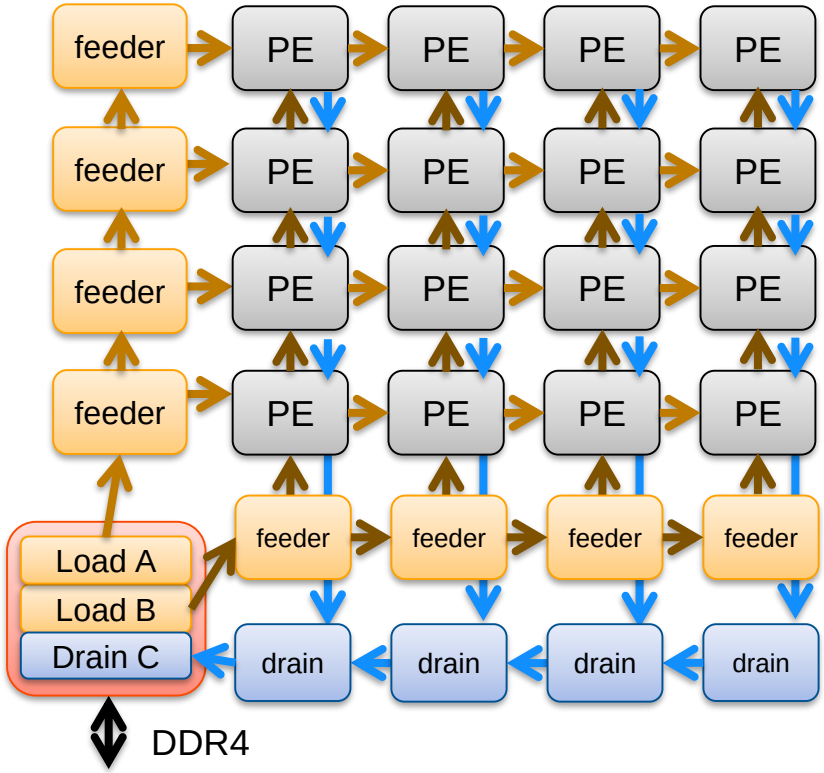
PE, Feed and Drain Arrays in OpenCL

10x16 array, dot8 PEs

10 + 16 feeders

~200 kernels

- Initially generated using a script

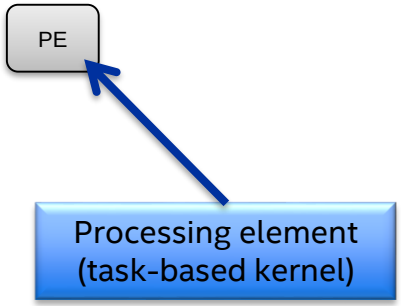


Kernel Replication with num_compute_units

Step #1: Design an efficient kernel

Step #2: How do we replicate it with OpenCL?

```
kernel void PE() {


   …
}
```



- `get_compute_id(1);`

Kernel Replication with num_compute_units

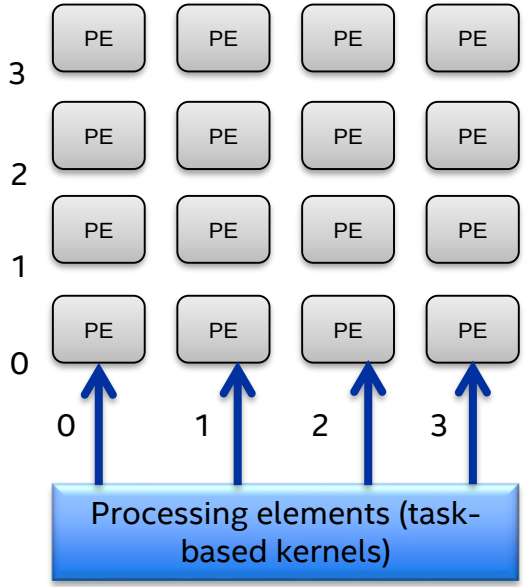
Attribute to specify 1D / 2D arrays of kernels

Add API to identify kernel in the array

```
__attribute__((num_compute_units(4,4)))
kernel void PE() {

   row = get_compute_id(0);
   col = get_compute_id(1);

   …
}
```

- `get_compute_id(1);`

Compile-time constants allows compiler to specialize each PE

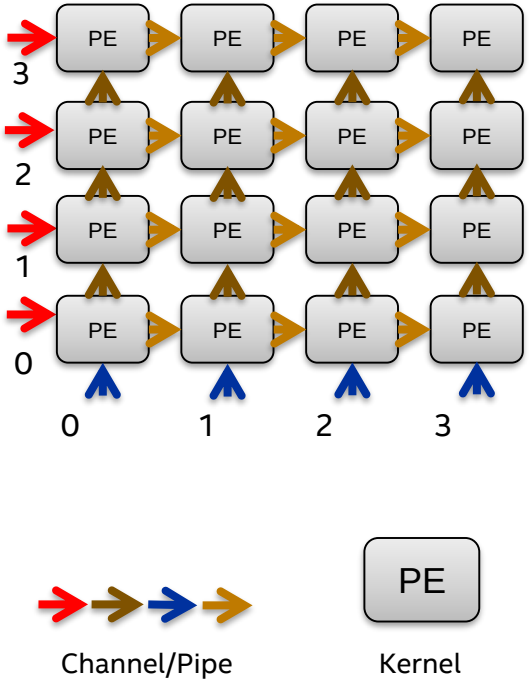


Kernel Replication with num_compute_units

Topology can be expressed with software constructs

- Channel connections specified by compute IDs

```
channel float4 ch_PE_row[4][4];
channel float4 ch_PE_col[4][4];
channel float4 ch_PE_row_side[4];
channel float4 ch_PE_col_side[4];

__attribute__((num_compute_units(4,4)))
kernel void PE() {
   row = get_compute_id(0);
   col = get_compute_id(1);

   float4 a,b;

   if (row==0)
     a = read_channel(ch_PE_col_side[col]);
   else
     a = read_channel(ch_PE_col[row-1][col]);

   if (col==0)

      …
}
```

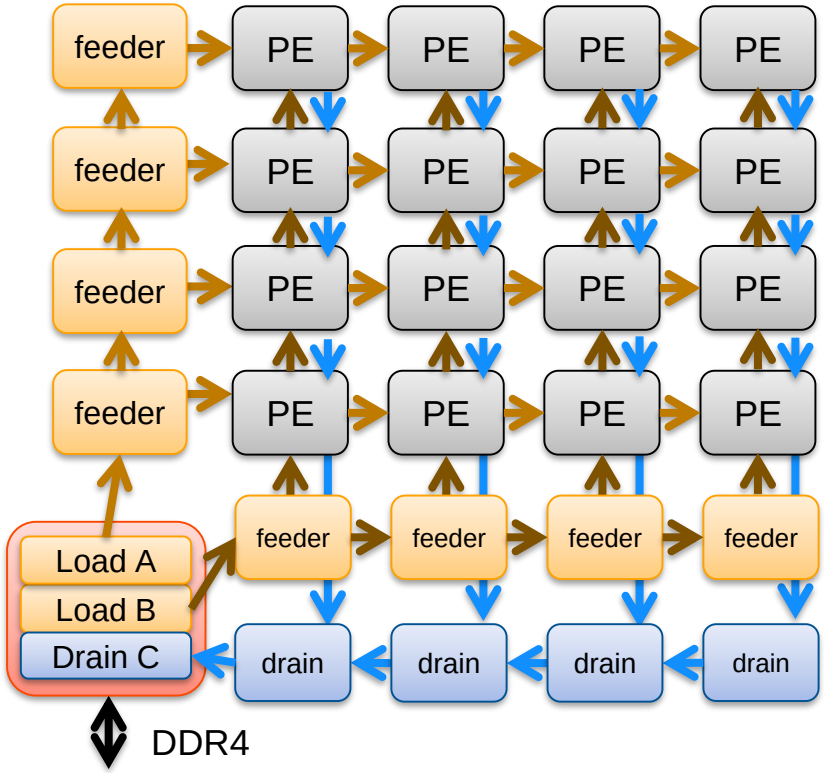


PE, Feed and Drain Arrays in OpenCL

2D PE array

1D feeder array (2x)

1D drain array

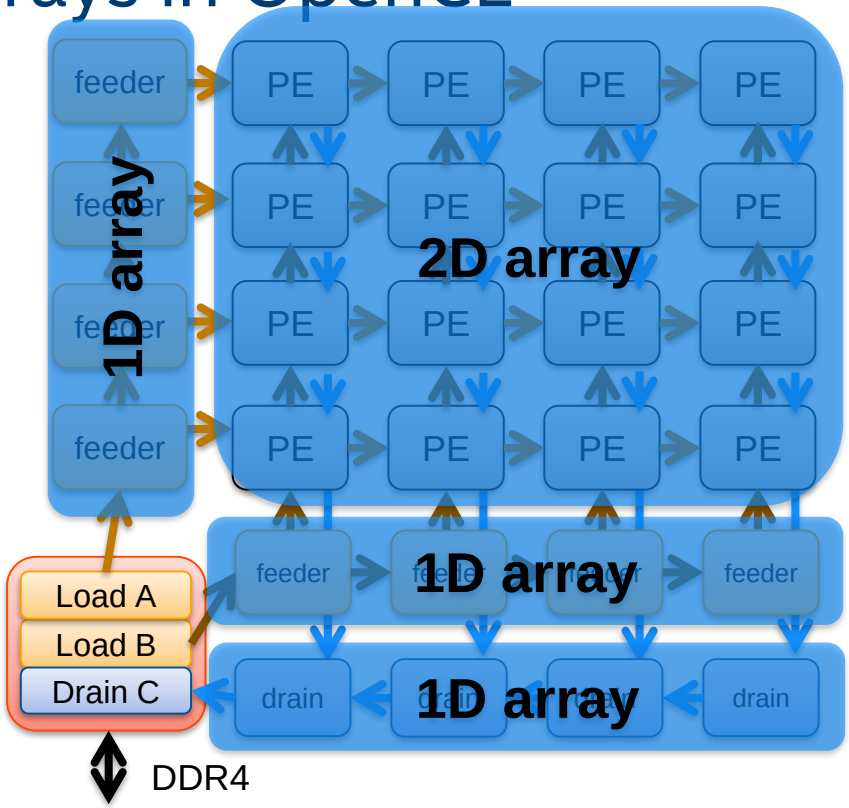


PE, Feed and Drain Arrays in OpenCL

2D PE array

1D feeder array (2x)

1D drain array



PE, Feed and Drain Arrays in OpenCL

2D PE array

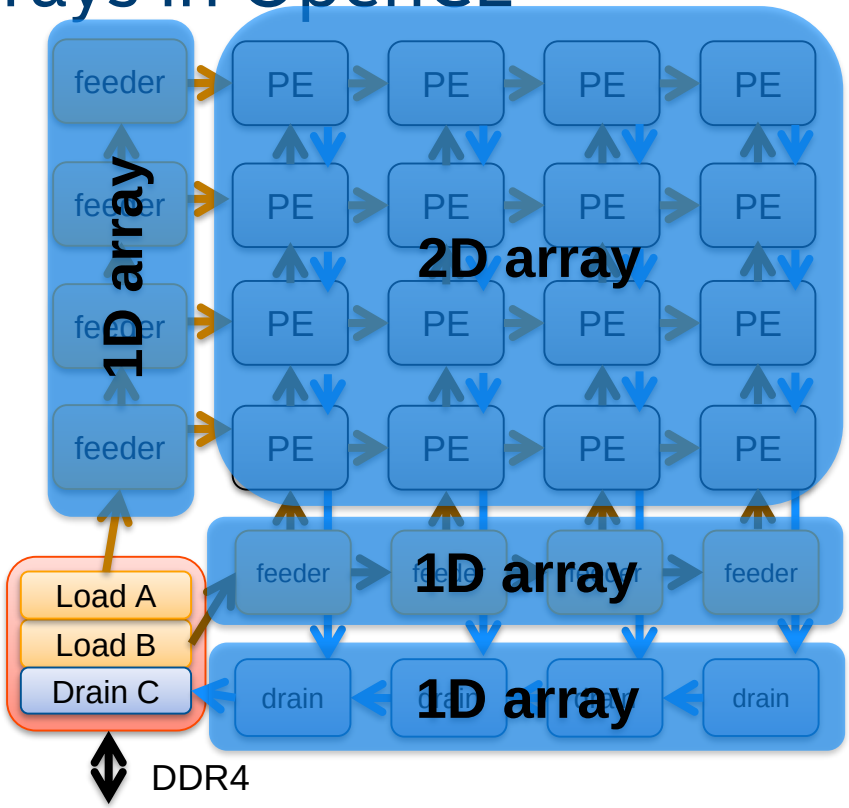
1D feeder array (2x)

1D drain array

Programmer writes 7 kernels

Currently ~700 lines of code

With further compiler improvements can be cut down to ~500 lines of code

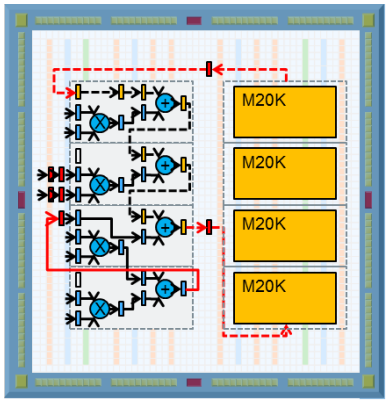


TAKE ADVANTAGE OF ON-DIE MEMORY

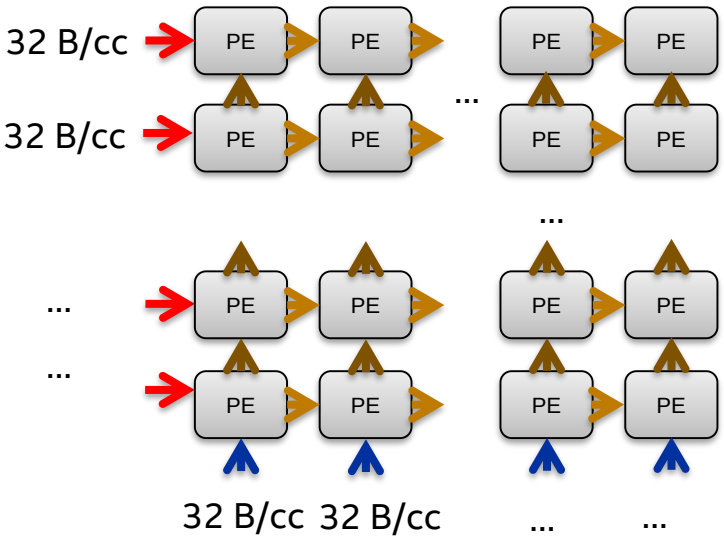
Insight #3:
Take advantage of on-die memory

Leverage plentiful internal memory BW (***8 TB/s***)

- Can create 100% DSP efficiency
- Can reduce external memory BW requirements by several orders of magnitude



Off-chip Mem BW Limitations



Dot8 based PE

- float8 data-paths

10 rows

16 columns

BW for each PE row/column

- 32 B/cc → 32B * 10 * 16 / cc = 5120B / cc
- At 400 MHz → **2048 GB/s**

Data Reuse via Matrix Blocking

10 rows

- 10 row feeders

16 columns

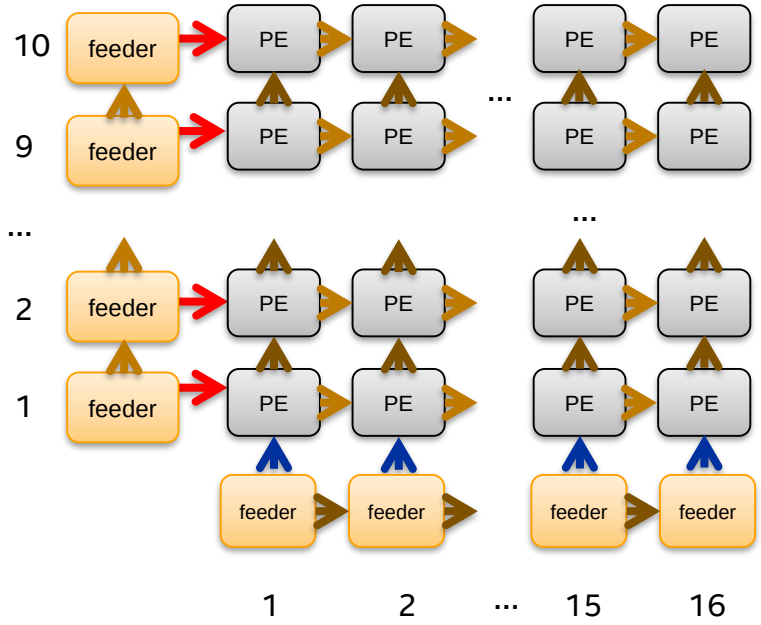
- 16 column feeders

Matrix A block size:

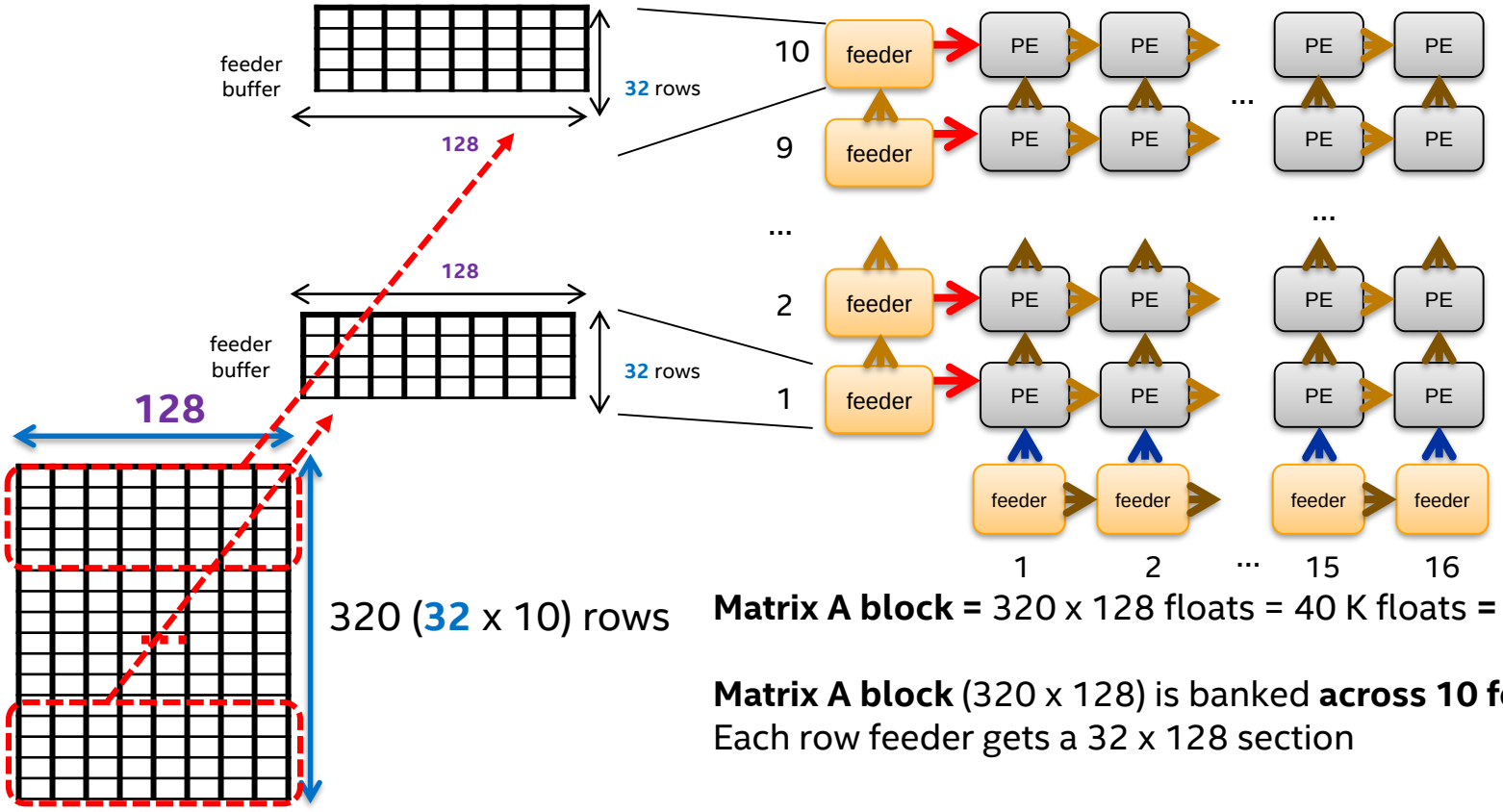
- 320 x 128 floats

Matrix B block size:

- 128 x 512 floats



Matrix A Block: Banked Across Row Feeders

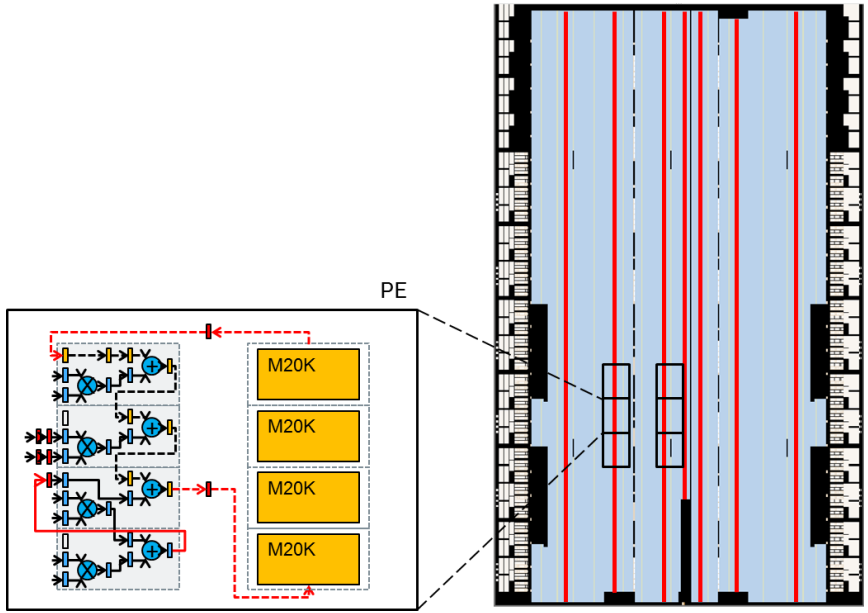


Matrix A block = 320 x 128 floats = 40 K floats = **160 KB**

Matrix A block (320 x 128) is banked **across 10 feeders**:
Each row feeder gets a 32 x 128 section

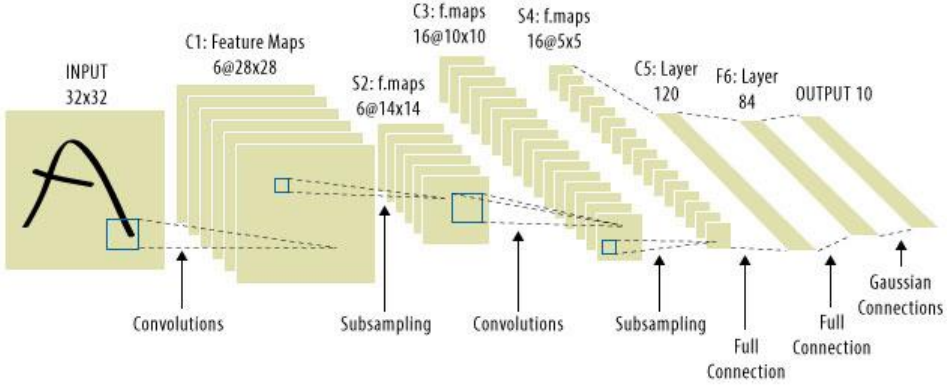
1 TFLOPS on Arria 10 – 1150

	Arria 10
Board throughput, FP32	**1010 GFLOPS**
Power	**21.5 W**
Clock Frequencey	**365 MHz**
DSPs	**1408 (93%)**
ALMs	**234K (55%)**
Registers	**710K**
Block Rams (M20Ks)	**2176 (80%)**



CONVOLUTIONAL NEURAL NETS ON FPGAS

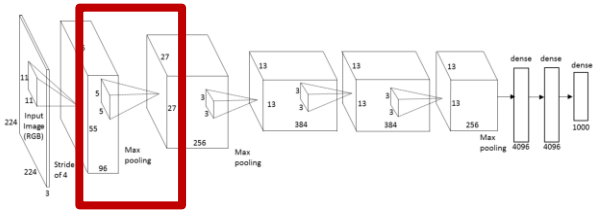
Rise of Convolutional Neural Nets



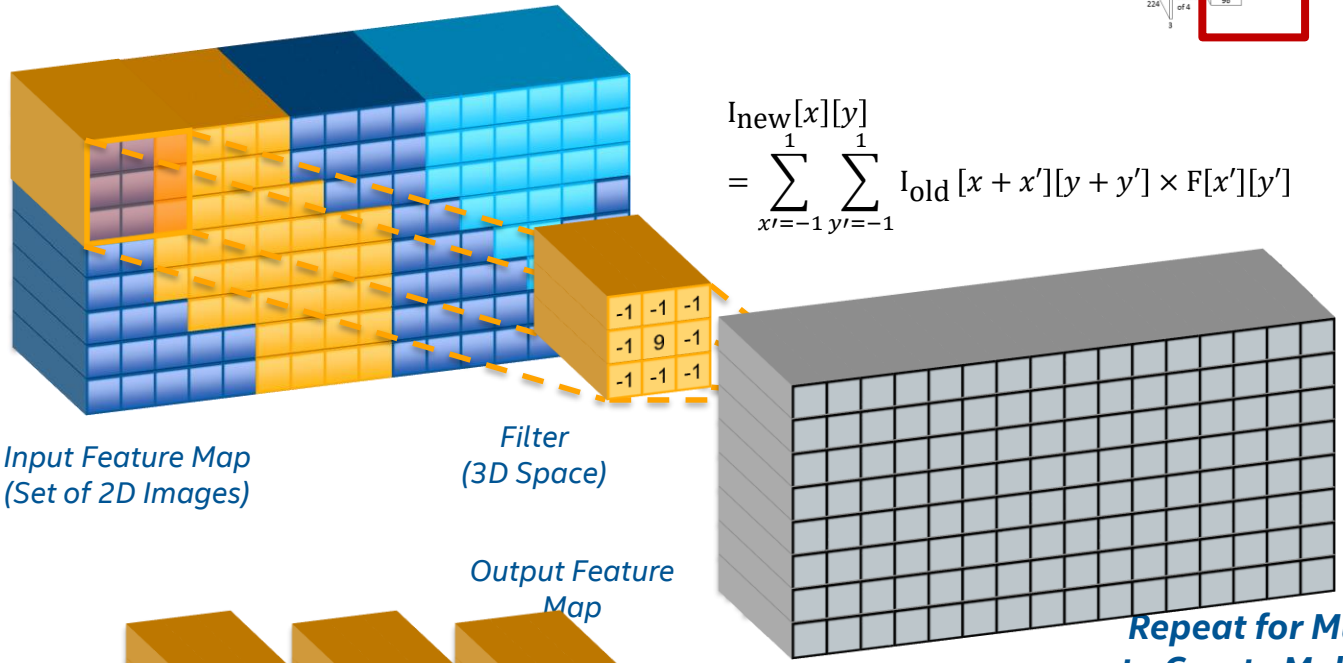
Convolutions validated as a practical means of tackling deep learning classification problems.

FPGAs are known to be efficient when performing convolutions.

CNN Computation in One Slide



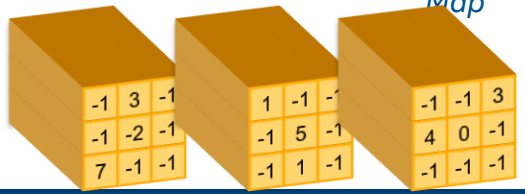
$$I_{new}[x][y] = \sum_{x'=-1}^{1} \sum_{y'=-1}^{1} I_{old}[x+x'][y+y'] \times F[x'][y']$$



Input Feature Map (Set of 2D Images)

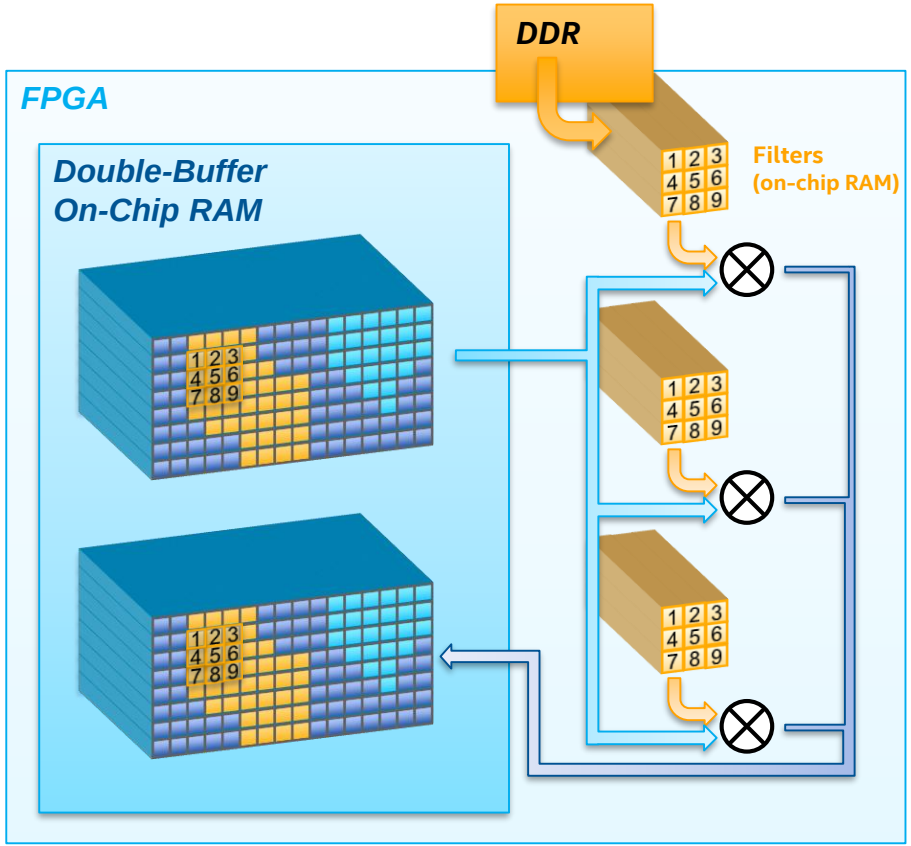
Filter (3D Space)

Output Feature Map



Repeat for Multiple Filters to Create Multiple "Layers" of Output Feature Map

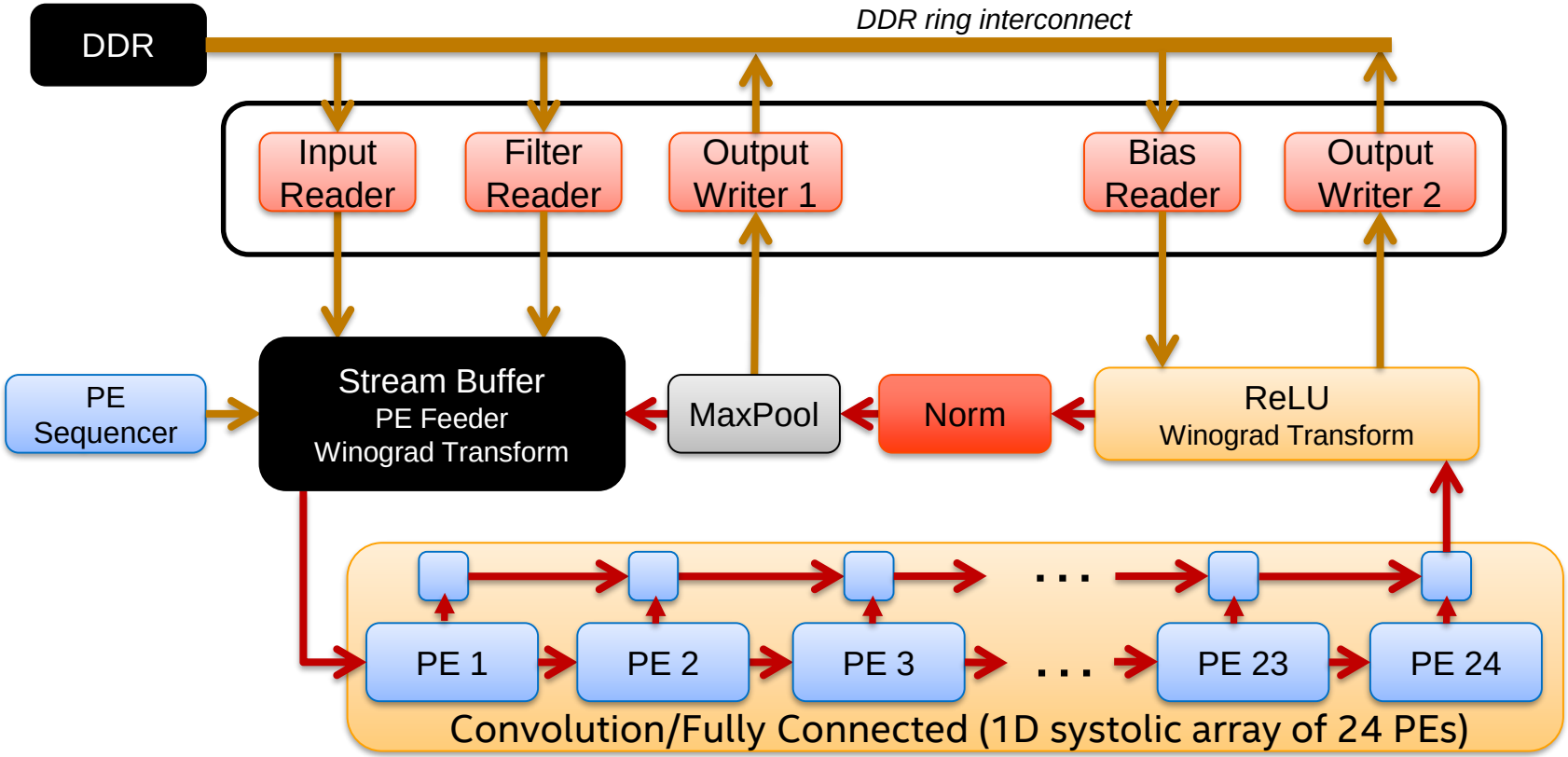
One Architecture *Any* Device *Any* Market



Leverages 3 insights from SGEMM:

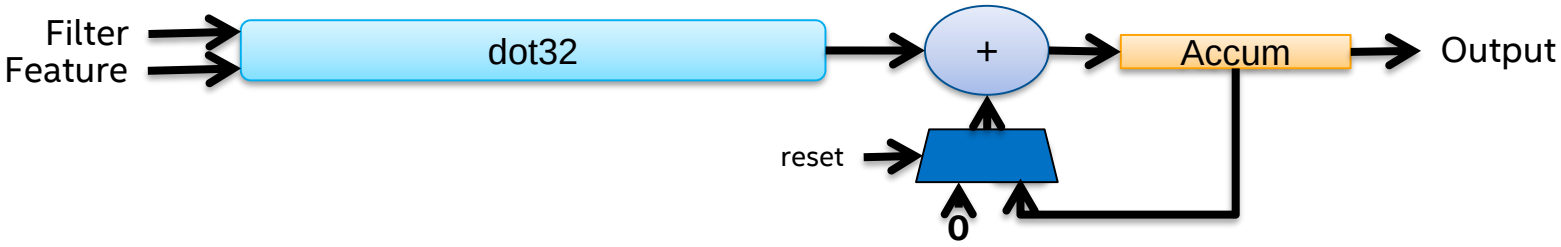
- Customized Spatial architecture that optimizes data accesses and creates efficient dataflow
- Creates large double-buffer to store all feature maps on-die
- Creates a systolic PE array to map nicely on 2D Spatial Logic Array on FPGA

Intel® Deep Learning Accelerator on FPGA



PE – Dot-product Engine

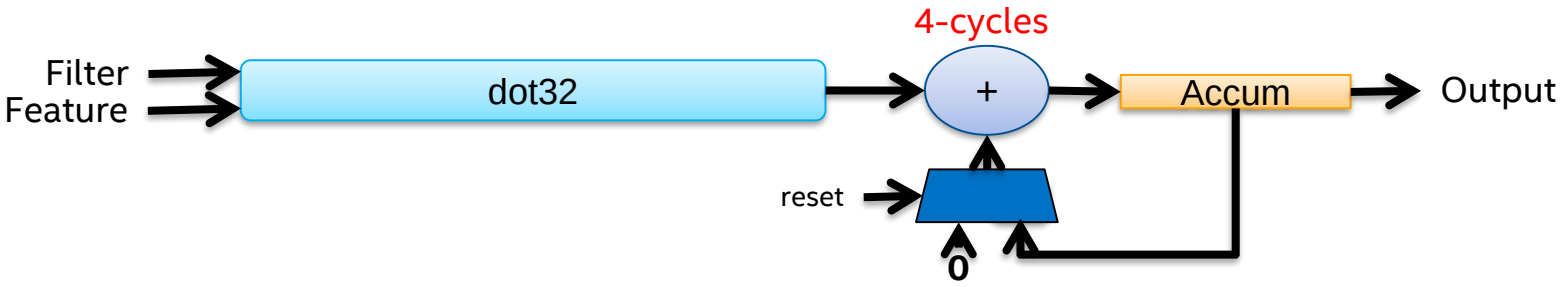
They perform the dot-product of feature data with filter data and accumulate the result.



PE – Dot-product Engine

They perform the dot-product of input data with filter data and accumulate the result.

Problem: Can only do one dot32 every 4-cycles.

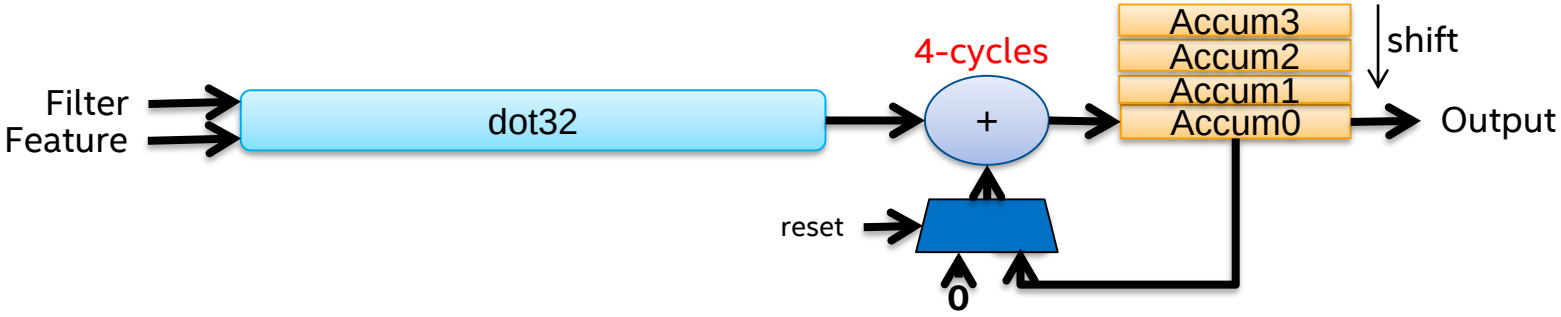


PE – Dot-product Engine

They perform the dot-product of input data with filter data and accumulate the result.

Problem: Can only do one dot32 every 4-cycles.

Solution: Do four dot32's every 4-cycles.



- In a 32x32 configuration, each convolution kernel receives 32 floats of feature data and 32 floats of filter data.

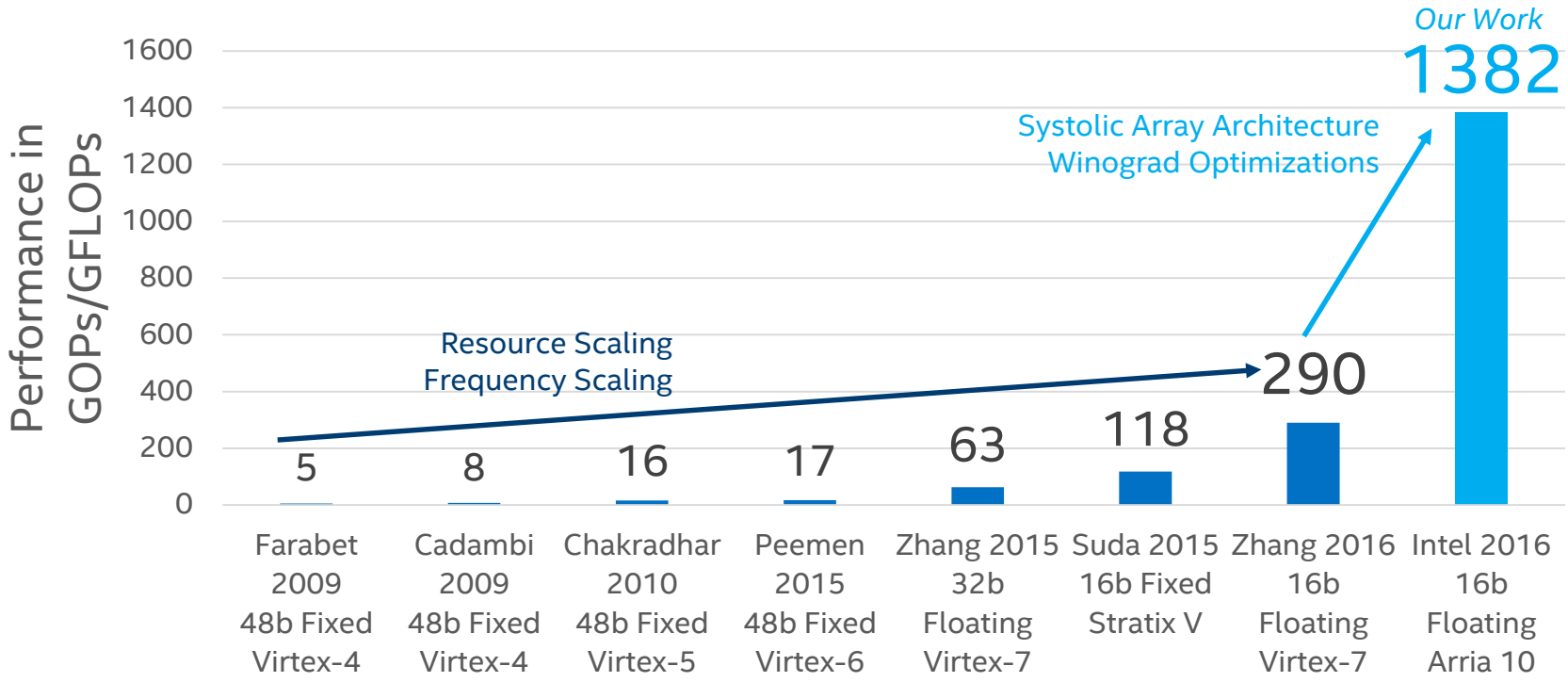
PE – Dot-product Engine

```
kernel void PE() {
  for(int cycle = 0; cycle < total_cycles; cycle++) {
    float accum[INTERLEAVE];
    for(int interleave = 0; interleave < INTERLEAVE; interleave++) {
      convolution_control cont = read_channel_intel( control_channel[k] );
      float_vec_t filter = filter_cache[cont.filter_addr];
      float_vec_t feature = read_channel_intel(feature_channel[k][vec]);
      float dot = 0;
      #pragma unroll
      for (int c = 0; c < C_VECTOR; c++) {
        dot += feature.v[c] * filter.v[c];
      }
      float conv = cont.reset ? dot[vec] : accum[0][vec] + dot[vec];
      #pragma unroll
      for(int i = 0; i < INTERLEAVE-1; i++)
          accum[i][vec] = accum[i+1][vec];
        accum[INTERLEAVE-1][vec] = conv[vec];
      }
      if( cont.send ) write_channel_intel( output_channel[k][vec], conv[vec] );
    }
}
```

outer loop
accumulator
interleave loop
read control
read filter
read feature
dot-product
accumulate
shift accumulators
send output

DEEP LEARNING ACCELERATOR PERFORMANCE

CNN Acceleration on FPGA through the Ages



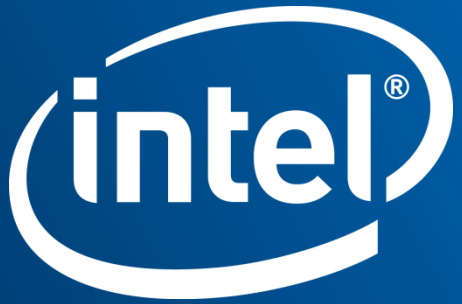
Acknowledgements

Special thanks to the HLD team on making SGEMM happen:

Dr. Andrei Hagiescu, Alan Baker, Dr. Peter Yiannacouras, Nitika Shanker,

Dr. Michael Kinsner, Dr. Tomasz Czajkowski, John Freeman, Byron Sinclair,

Bernhard Oelkrug,


intel®