6.5930/1
Hardware Architectures for Deep Learning

GPU Computation (continued)

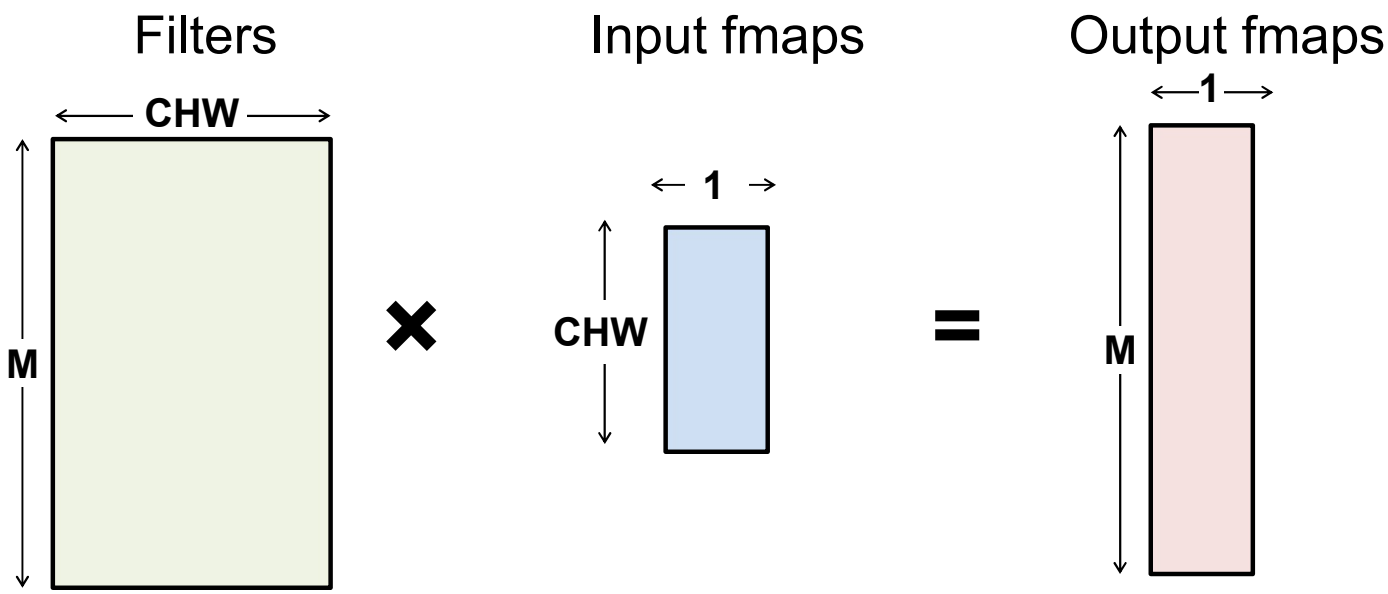
March 8, 2023

Joel Emer and Vivienne Sze
Acknowledgement: Srini Devadas (MIT)

Massachusetts Institute of Technology
Electrical Engineering & Computer Science

Fully-Connected (FC) Layer

- Matrix–Vector Multiply:
  - Multiply all inputs in all channels by a weight and sum



Fully Connected Computation

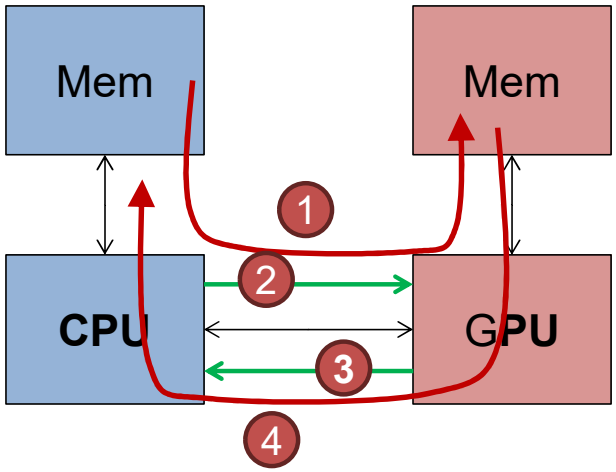
```
int i[C*H*W];     # Input activations
int f[M*C*H*W];   # Filter Weights
int o[M];         # Output activations


for (m = 0; m < M; m++) {
  o[m] = 0;
  CHWm = C*H*W*m;
  for (chw = 0; chw < C*H*W; chw++) {
     o[m] += i[chw] * f[CHWm + chw];
  }
}
```

Parallelize here

$I[C_0\ H_0\ W_0]$ $I[C_0\ H_0\ W_1]$ …
$I[C_0\ H_1\ W_0]$ $I[C_0\ H_1\ W_1]$ …
$I[C_0\ H_2\ W_0]$ $I[C_0\ H_2\ W_1]$ …
.
.
$I[C_1\ H_0\ W_0]$ $I[C_1\ H_0\ W_1]$ …
$I[C_1\ H_1\ W_0]$ $I[C_1\ H_1\ W_1]$ …
$I[C_1\ H_2\ W_0]$ $I[C_1\ H_2\ W_1]$ …
.
.
.

$F[M_0\ C_0\ H_0\ W_0]$ $F[M_0\ C_0\ H_0\ W_1]$ …
$F[M_0\ C_0\ H_1\ W_0]$ $F[M_0\ C_0\ H_1\ W_1]$ …
$F[M_0\ C_0\ H_2\ W_0]$ $F[M_0\ C_0\ H_2\ W_1]$ …
.
.
$F[M_0\ C_1\ H_0\ W_0]$ $F[M_0\ C_1\ H_0\ W_1]$ …
$F[M_0\ C_1\ H_1\ W_0]$ $F[M_0\ C_1\ H_1\ W_1]$ …
$F[M_0\ C_1\ H_2\ W_0]$ $F[M_0\ C_1\ H_2\ W_1]$ …
.
.
.
$F[M_1\ C_0\ H_0\ W_0]$ $F[M_1\ C_0\ H_0\ W_1]$ …
$F[M_1\ C_0\ H_1\ W_0]$ $F[M_1\ C_0\ H_1\ W_1]$ …
$F[M_1\ C_0\ H_2\ W_0]$ $F[M_1\ C_0\ H_2\ W_1]$ …
.
.
.

GPU Kernel Execution



1. Transfer input data from CPU to GPU memory
2. Launch kernel (grid)
3. Wait for kernel to finish (if synchronous)
4. Transfer results to CPU memory

- Data transfers can dominate execution time
- Integrated GPUs with unified address space → no copies, but…

FC - CUDA Main Program

```
int i[C*H*W],    *gpu_i;     # Input activations
int f[M*C*H*W], *gpu_f;     # Filter Weights
int o[M],        *gpu_o;     # Output activations

# Allocate space on GPU
cudaMalloc((void**) &gpu_i, sizeof(int)*C*H*W);
cudaMalloc((void**) &gpu_f, sizeof(int)*M*C*H*W);

# Copy data to GPU
cudaMemcpy(gpu_i, i, sizeof(int)*C*H*W);
cudaMemcpy(gpu_f, f, sizeof(int)*M*C*H*W);

# Run kernel
fc<<<M/256 + 1, 256>>>(gpu_i, gpu_f, gpu_o, C*H*W, M);

# Copy result back and free device memory
cudaMemcpy(o, gpu_o, sizeof(int)*M, cudaMemCpyDevicetoHost);
cudaFree(gpu_i);
...
```

References to data on GPU

Num thread blocks

Thread block size

FC – CUDA Terminology

- CUDA code launches 256 threads per block

- CUDA vs vector terminology:
  - Thread = 1 iteration of scalar loop [1 element in vector loop]
  - Block = Body of vectorized loop [VL=256 in this example]
    - Warp size = 32 [Number of vector lanes]
  - Grid = Vectorizable loop

[vector terminology]

FC - CUDA Kernel

```
__global__
void fc(int *i, int *f, int *o, const int CHW, const int M){

    int tid=threadIdx.x + blockIdx.x * blockDim.x;
    int m = tid

    int sum = 0;

    if (m < M){
        for (int chw=0; chw <CHW; chw ++) {
            sum += i[chw]*f[(m*CHW)+chw];
        }

        o[m] = sum;
    }
}
```

Code for one thread

Calculate "global" thread id (tid)

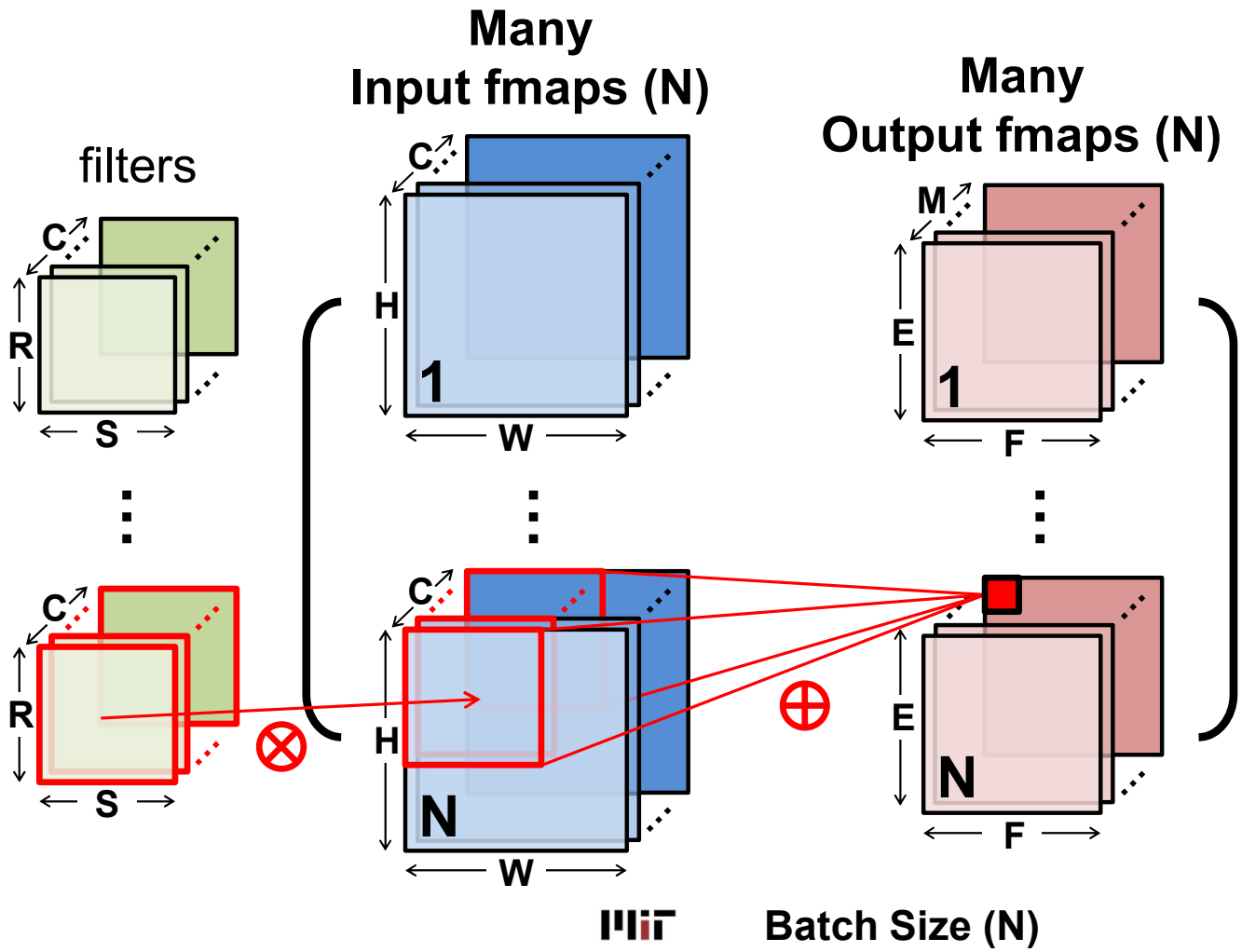
tid is "output channel" number

Can be unrolled

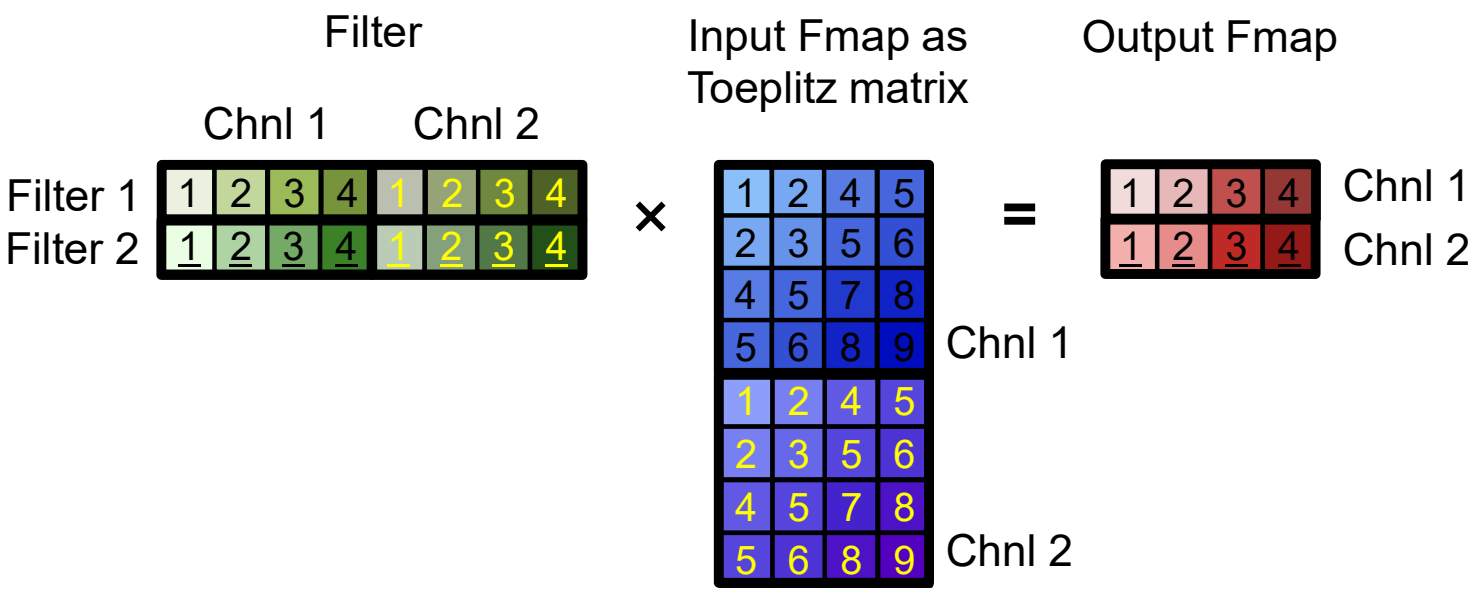
Any consequences of f[(m*CHW)+chw]? Yes, strided references

Any consequences of f[(chw*M)+m]? Yes, different data layout

Convolution (CONV) Layer



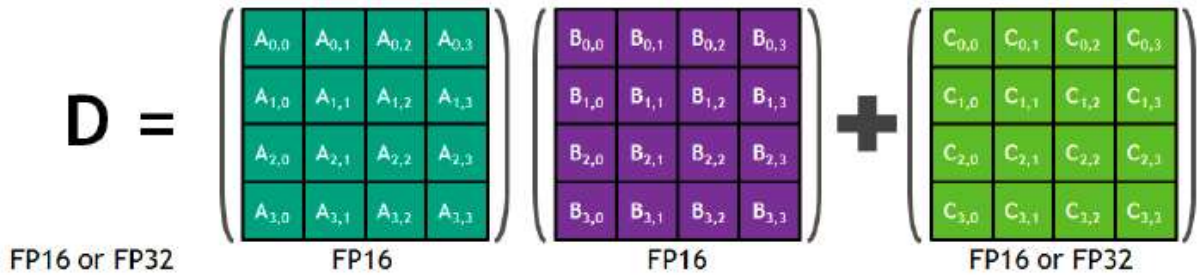
Convolution (CONV) Layer



GPU Implementation:

- Keep original input activation matrix in main memory
- Conceptually do tiled matrix-matrix multiplication
- Copy input activations into scratchpad to small Toeplitz matrix tile
- On Volta tile again to use ‘tensor core’

GV100 – "Tensor Core"



New opcodes – Matrix Multiply Accumulate (HMMA)

How many FP16 operands? Inputs 48 / Outputs 16

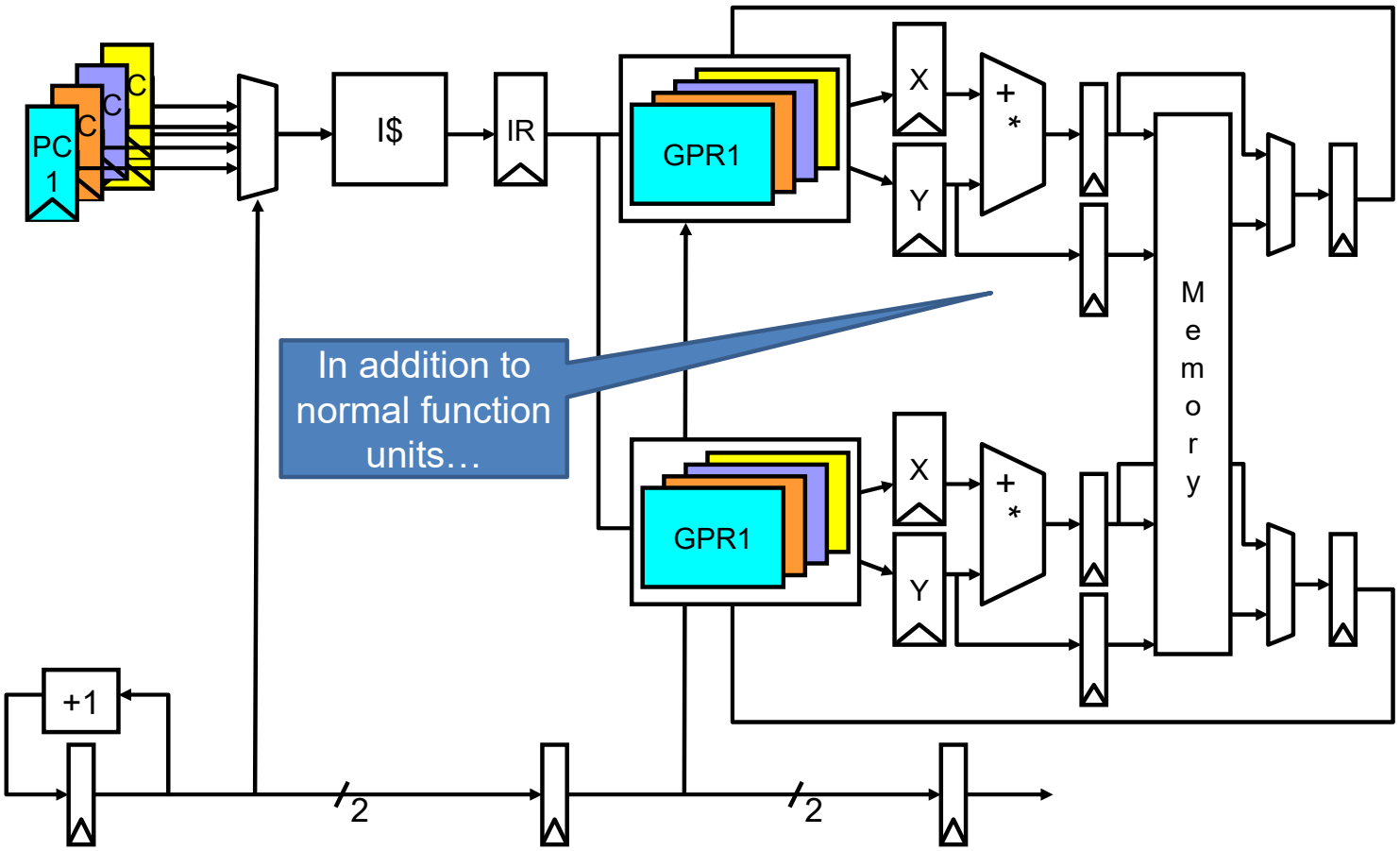
How many multiplies? 64

How many adds? 64

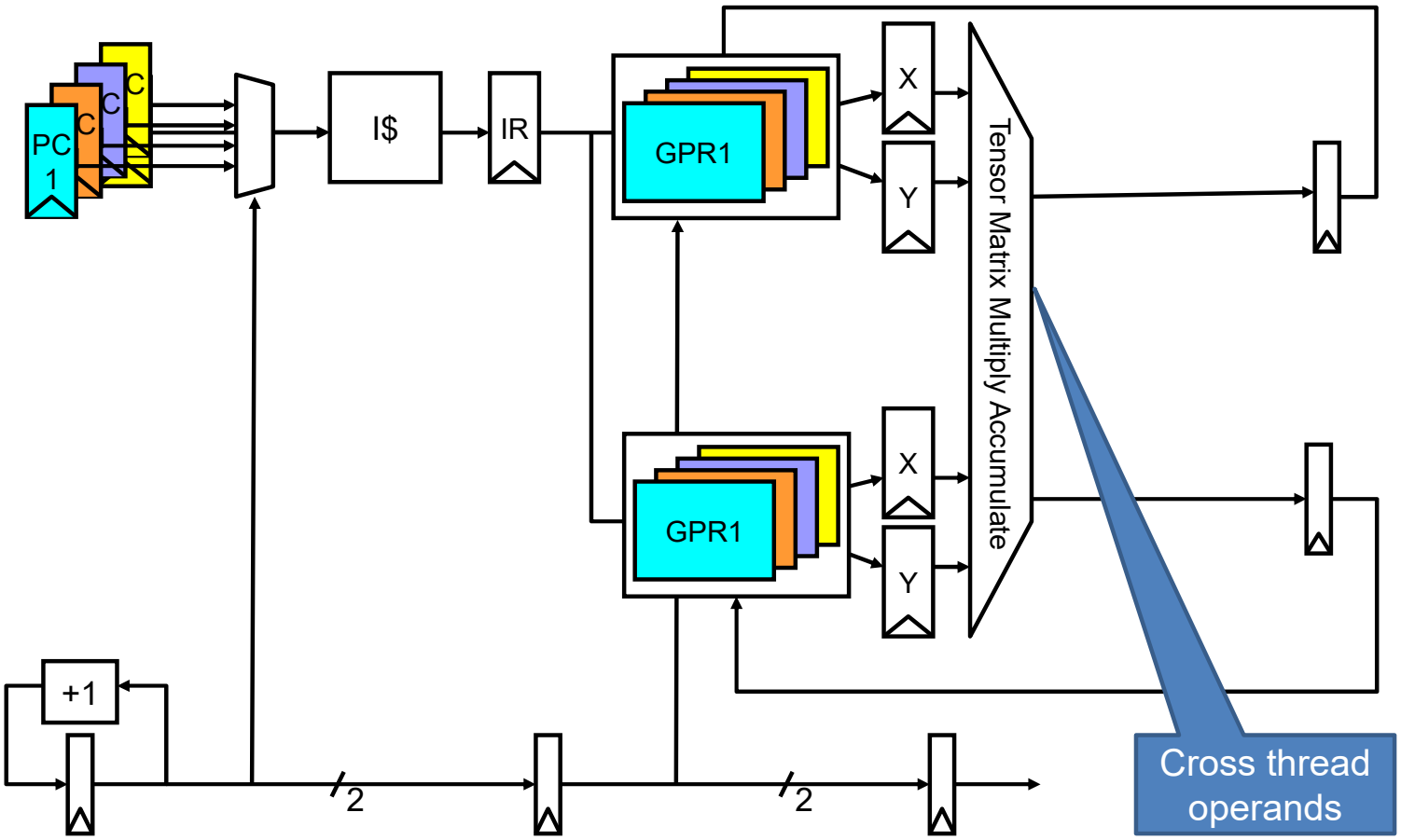
Volta tensor Core…. 120 TFLOPS (FP16), 400 GFLOPS/W (FP16)

Toeplitz expansion is essential to exploit hardware

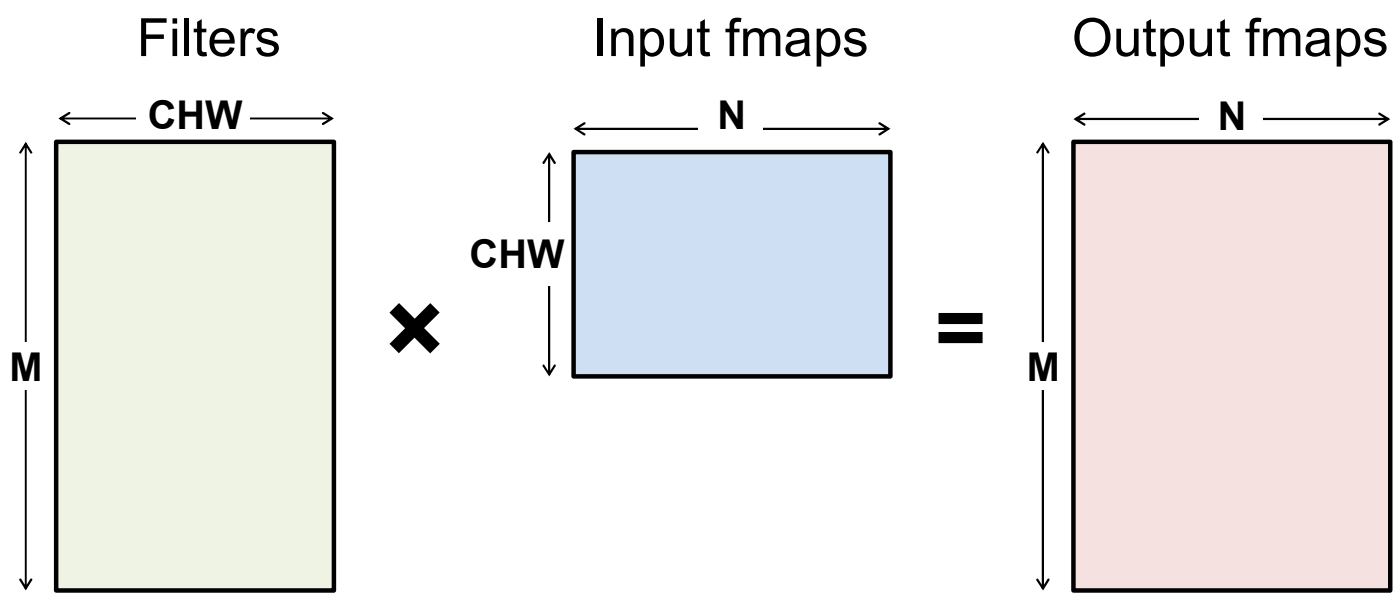
Tensor Core Integration



Tensor Core Integration

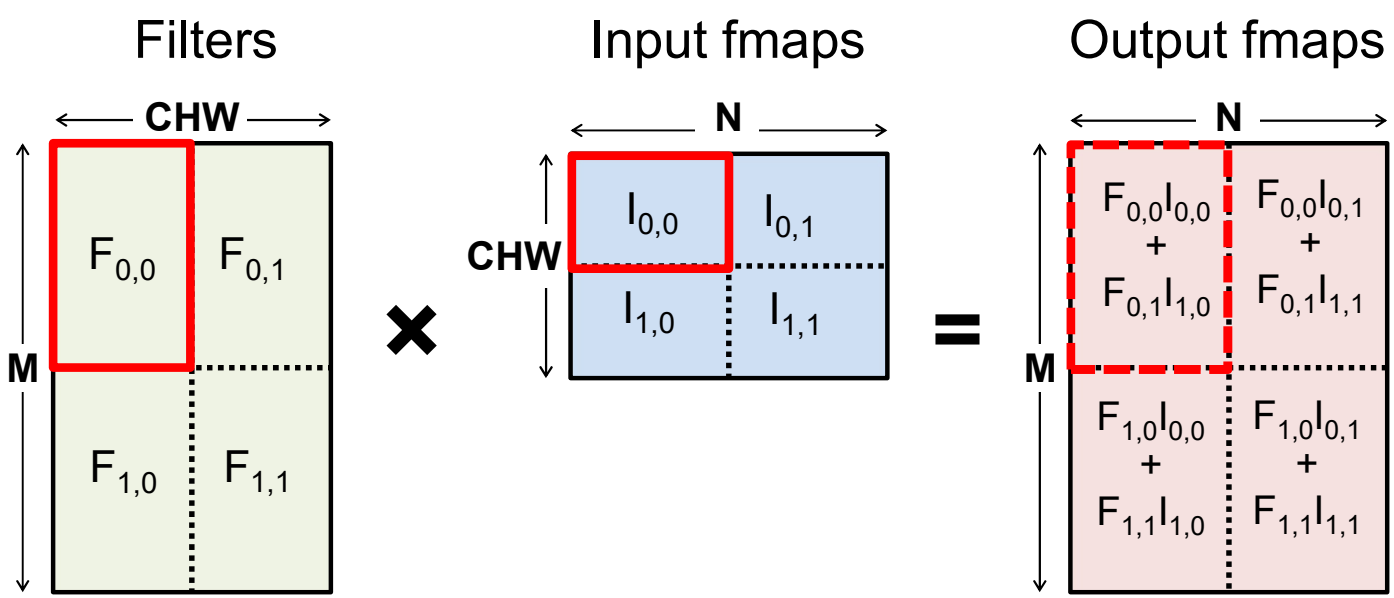


Fully-Connected (FC) Layer



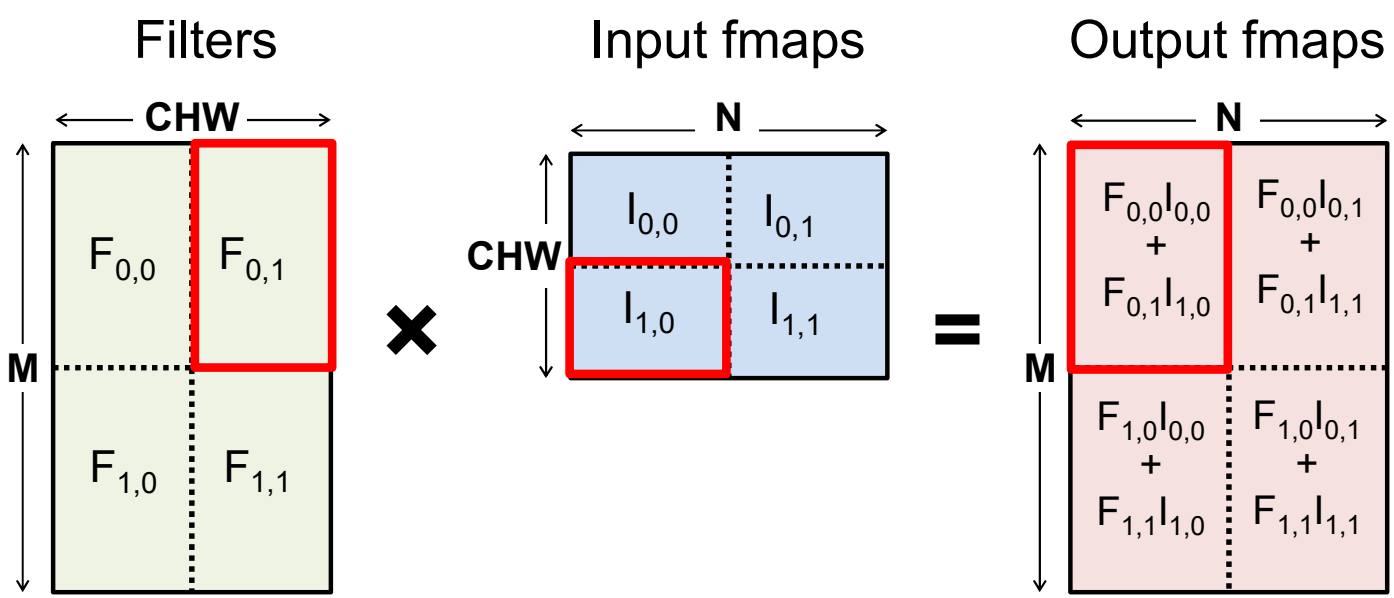
- After flattening, having a batch size of N turns the matrix-vector operation into a matrix-matrix multiply

Tiled Fully-Connected (FC) Layer



Matrix multiply tiled to fit in tensor core operation
and computation ordered to maximize reuse of data in scratchpad
and then in cache.

Tiled Fully-Connected (FC) Layer



Matrix multiply tiled to fit in tensor core operation
and computation ordered to maximize reuse of data in scratchpad
and then in cache.

Vector vs GPU Terminology

Type	More descriptive name	Closest old term outside of GPUs	Official CUDA/ NVIDIA GPU term	Book definition
Program abstractions	Vectorizable Loop	Vectorizable Loop	Grid	A vectorizable loop, executed on the GPU, made up of one or more Thread Blocks (bodies of vectorized loop) that can execute in parallel.
	Body of Vectorized Loop	Body of a (Strip-Mined) Vectorized Loop	Thread Block	A vectorized loop executed on a multithreaded SIMD Processor, made up of one or more threads of SIMD instructions. They can communicate via Local Memory.
	Sequence of SIMD Lane Operations	One iteration of a Scalar Loop	CUDA Thread	A vertical cut of a thread of SIMD instructions corresponding to one element executed by one SIMD Lane. Result is stored depending on mask and predicate register.
Machine object	A Thread of SIMD Instructions	Thread of Vector Instructions	Warp	A traditional thread, but it contains just SIMD instructions that are executed on a multithreaded SIMD Processor. Results stored depending on a per-element mask.
	SIMD Instruction	Vector Instruction	PTX Instruction	A single SIMD instruction executed across SIMD Lanes.
Processing hardware	Multithreaded SIMD Processor	(Multithreaded) Vector Processor	Streaming Multiprocessor	A multithreaded SIMD Processor executes threads of SIMD instructions, independent of other SIMD Processors.
	Thread Block Scheduler	Scalar Processor	Giga Thread Engine	Assigns multiple Thread Blocks (bodies of vectorized loop) to multithreaded SIMD Processors.
	SIMD Thread Scheduler	Thread scheduler in a Multithreaded CPU	Warp Scheduler	Hardware unit that schedules and issues threads of SIMD instructions when they are ready to execute; includes a scoreboard to track SIMD Thread execution.
	SIMD Lane	Vector Lane	Thread Processor	A SIMD Lane executes the operations in a thread of SIMD instructions on a single element. Results stored depending on mask.
Memory hardware	GPU Memory	Main Memory	Global Memory	DRAM memory accessible by all multithreaded SIMD Processors in a GPU.
	Private Memory	Stack or Thread Local Storage (OS)	Local Memory	Portion of DRAM memory private to each SIMD Lane.
	Local Memory	Local Memory	Shared Memory	Fast local SRAM for one multithreaded SIMD Processor, unavailable to other SIMD Processors.
	SIMD Lane Registers	Vector Lane Registers	Thread Processor Registers	Registers in a single SIMD Lane allocated across a full thread block (body of vectorized loop).

[H&P5, Fig 4.25]

Summary

- GPUs are an intermediate point in the continuum of flexibility and efficiency

- GPU architecture focuses on throughput (over latency)
  - Massive thread-level parallelism, with
    - Single instruction for many threads
  - Memory hierarchy specialized for throughput
    - Shared scratchpads with private address space
    - Caches used primarily to reduce bandwidth not latency
  - Specialized compute units, with
    - Many computes per input operand

- Little's Law useful for system analysis
  - Shows relationship between throughput, latency and tasks in-flight

6.5930/1
Hardware Architectures for Deep Learning

Accelerator Architecture

March 8, 2023

Joel Emer and Vivienne Sze

Massachusetts Institute of Technology
Electrical Engineering & Computer Science

What is Moore's Law

- CPU performance will double every two years*
- Chip performance will double every two years*
- The speed of transistors will double every two years*
- Transistors will shrink to half size every two years*
- Gate width will shrink by $\sqrt{2}$ every two years*
- Transistors per die will double every two years*
- The economic sweet spot for the number of devices on a chip will double every two years*

* Or 12 or 18 months…

Moore's (Original) Law

The complexity for minimum component costs has increased at a rate of roughly a factor of two per year.

- Gordon Moore

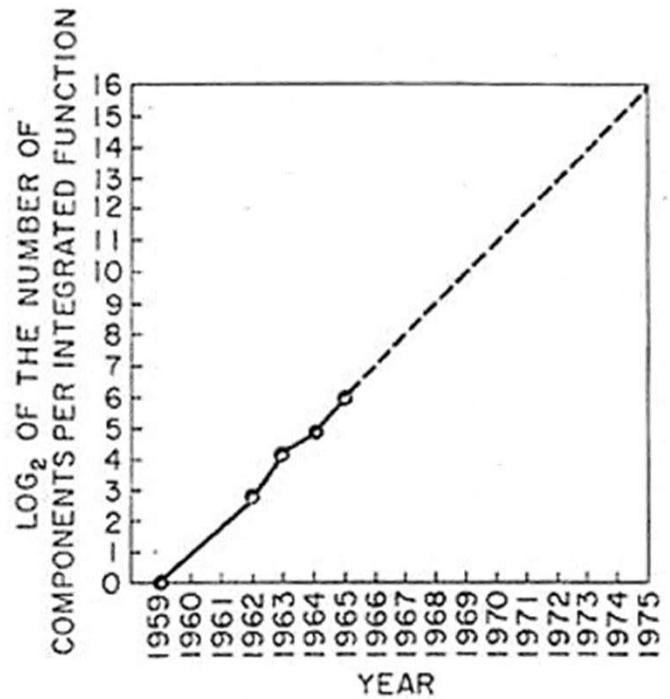
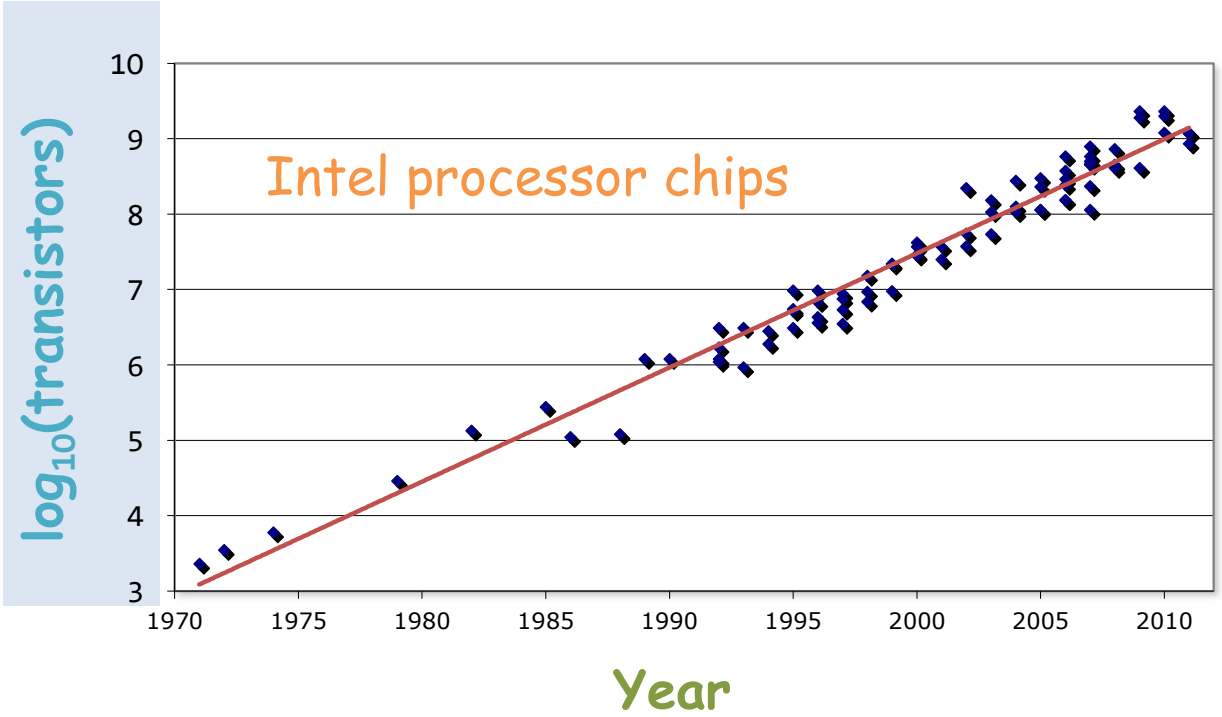


Fig. 2 Number of components per integrated function for minimum cost per component extrapolated vs time.

Moore, "Cramming more components onto integrated circuits", Electronics 1965.]

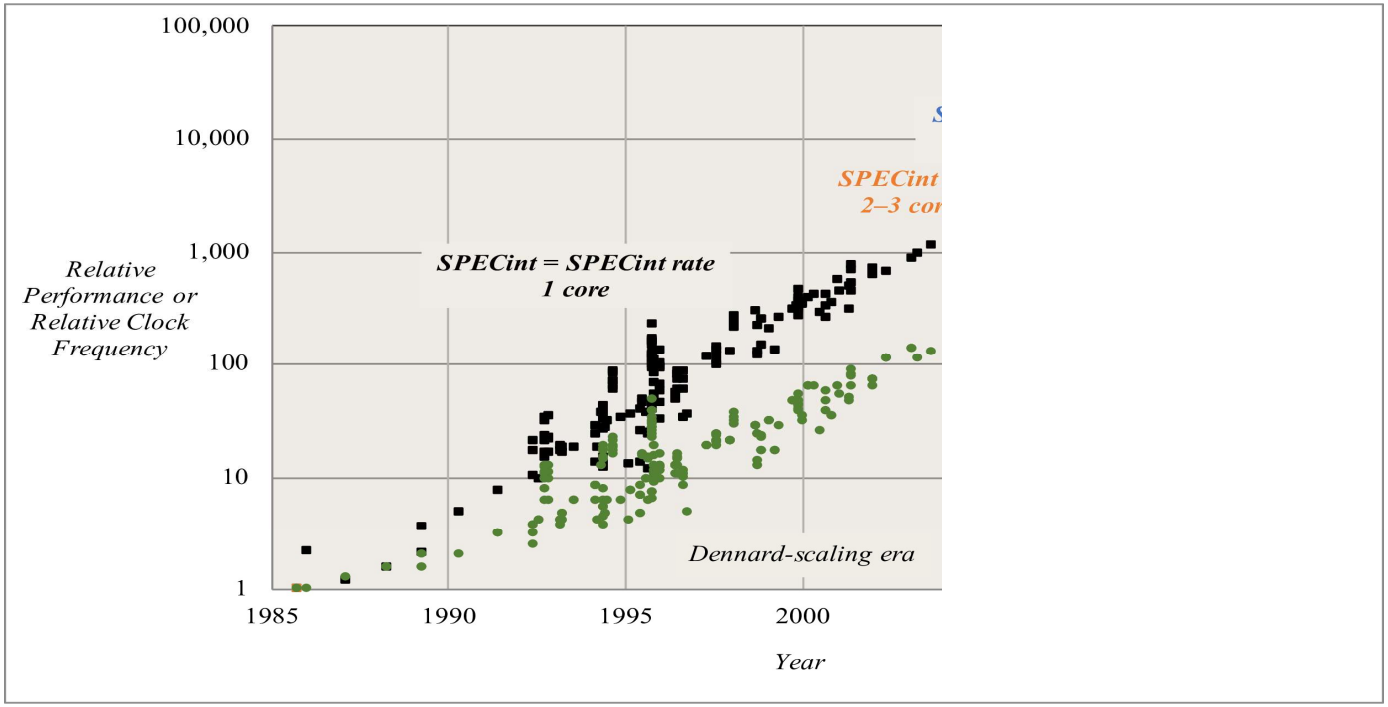
Moore's (Transistor) Law



[Moore, Progress in digital integrated electronics, IEDM 1975]

Number of transistors has been doubling

Moore's (Performance) Law



[Leiserson et al., There's Plenty of Room at the Top, Science]

Energy and Power Consumption

- **Energy Consumption (Joules)** = $\alpha \times C \times V^2$

Switching activity factor (between 0 to 1)

Capacitance

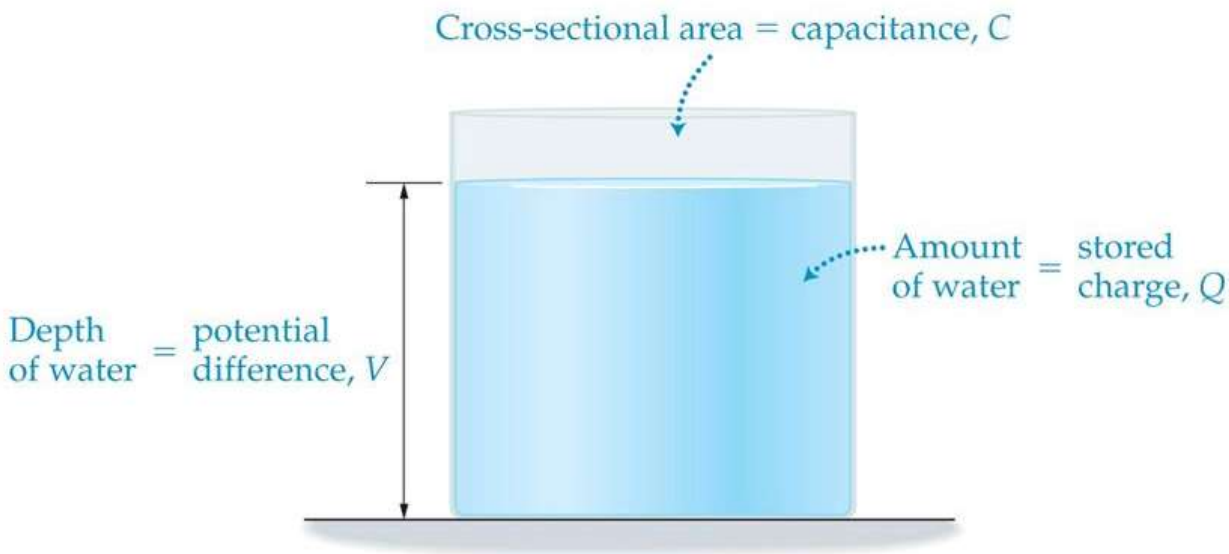
Voltage

- **Power Consumption (Joules/sec or Watts)** = $\alpha \times C \times V^2 \times f$

Frequency

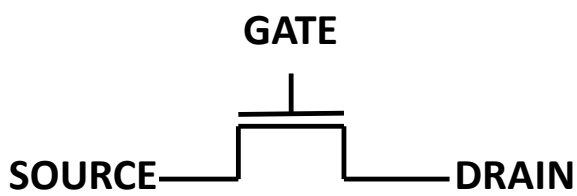
Capacitance and Energy Storage

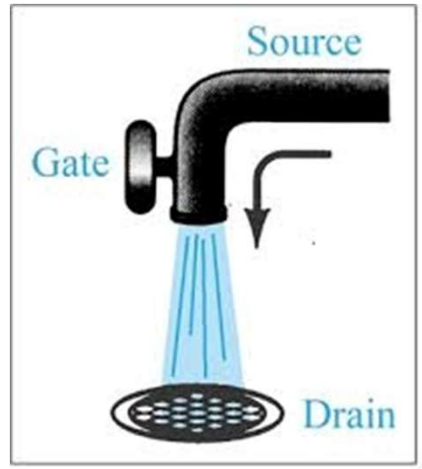
- A bucket of water provides a useful analogy when thinking about capacitors, as shown in the figure below.



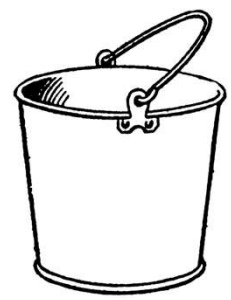
Energy Consumption = α x C x V^2

Analogy: Water = Charge





Full Bucket = '1'

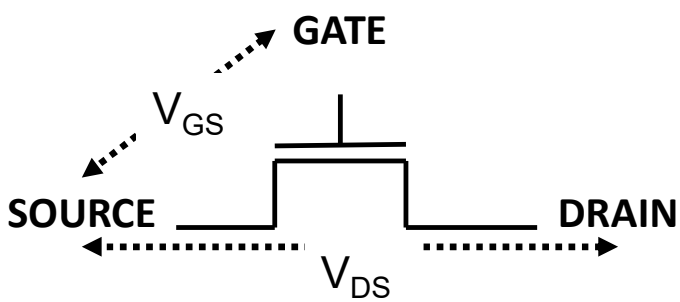


Size of bucket proportional to load capacitance

What affects the rate of flow of the water (i.e. current)?

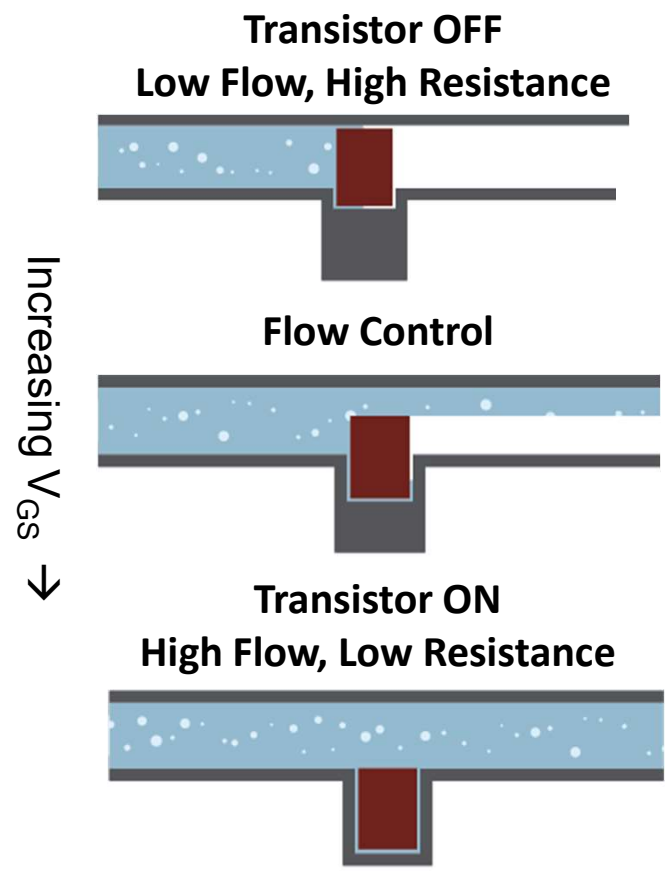
Analogy: Water = Charge

Transistor has multiple states



Saturation Mode: $V_{DS} \geq V_{GS} - V_T$

$$I_D = \frac{k'_n}{2}\frac{W}{L}(V_{GS} - V_T)^2(1 + \lambda V_{DS})$$

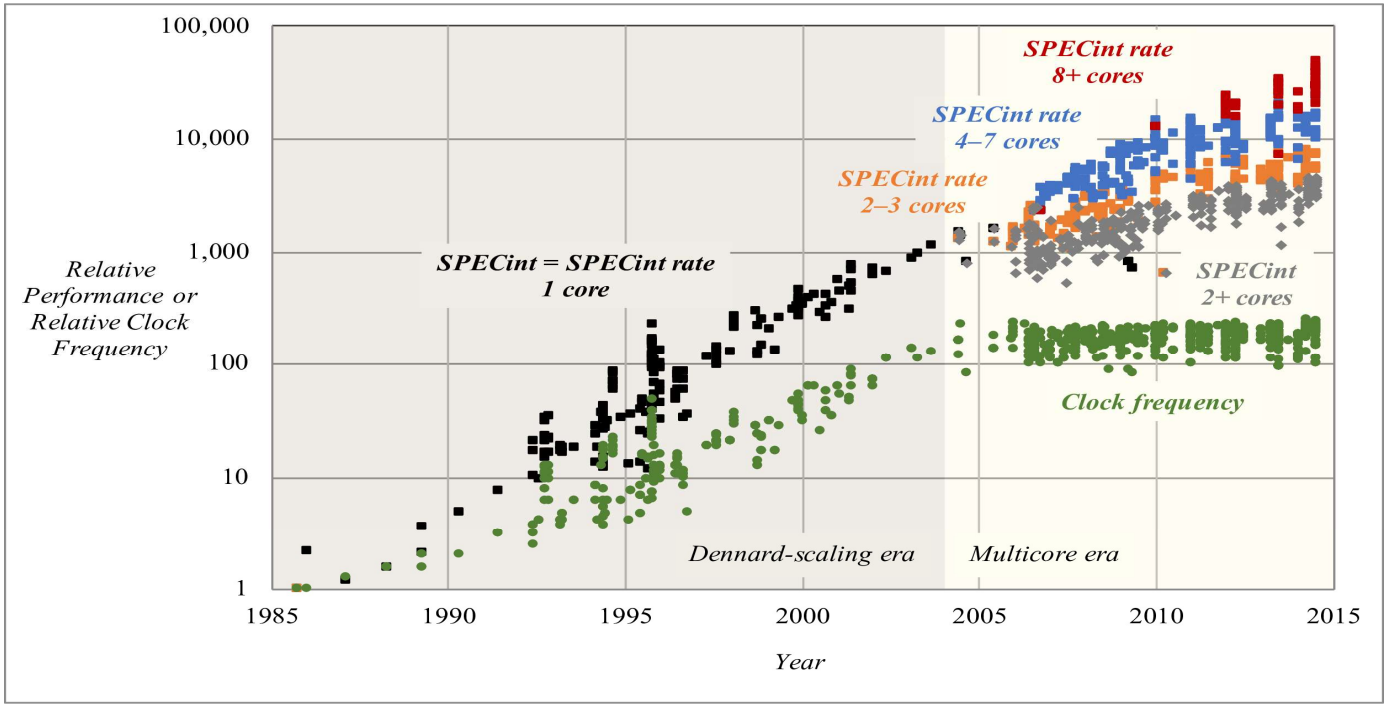


Dennard Scaling (idealized)

	Gen X	Gen X+1
Gate Width	1.0	0.7
Device Area/ Capacitance	1.0 (1.0)	0.5 (0.7), 0.5 (0.7)
Voltage	1.0	0.7
Energy	~ $1.0 \times 1.0^2 = 1.0$	~ $2 \times 0.7 \times 0.7^2 = 0.65$
Delay	1.0	0.7
Frequency	$1/1.0 = 1.0$	$1/0.7 = 1.4$
Power	~ $1.0 \times 1.0^2 \times 1.0 = 1.0$	~ $2 \times 0.7 \times 0.7^2 \times 1.4 = 1.0$

[Dennard et al., "Design of ion-implanted MOSFET's with very small physical dimensions", JSSC 1974]

The End of Historic Scaling



[Leiserson et al., There's Plenty of Room at the Top, Science]

Voltage scaling slowed down → Power density increasing!

During the Moore + Dennard's Law Era

- Instruction-level parallelism (ILP) was largely mined out by early 2000s
- Voltage (Dennard) scaling ended in 2005
- Hit the power limit wall in 2005
- Performance is coming from parallelism using more transistors since ~2007
- But….

Moore's Law in DRAMs

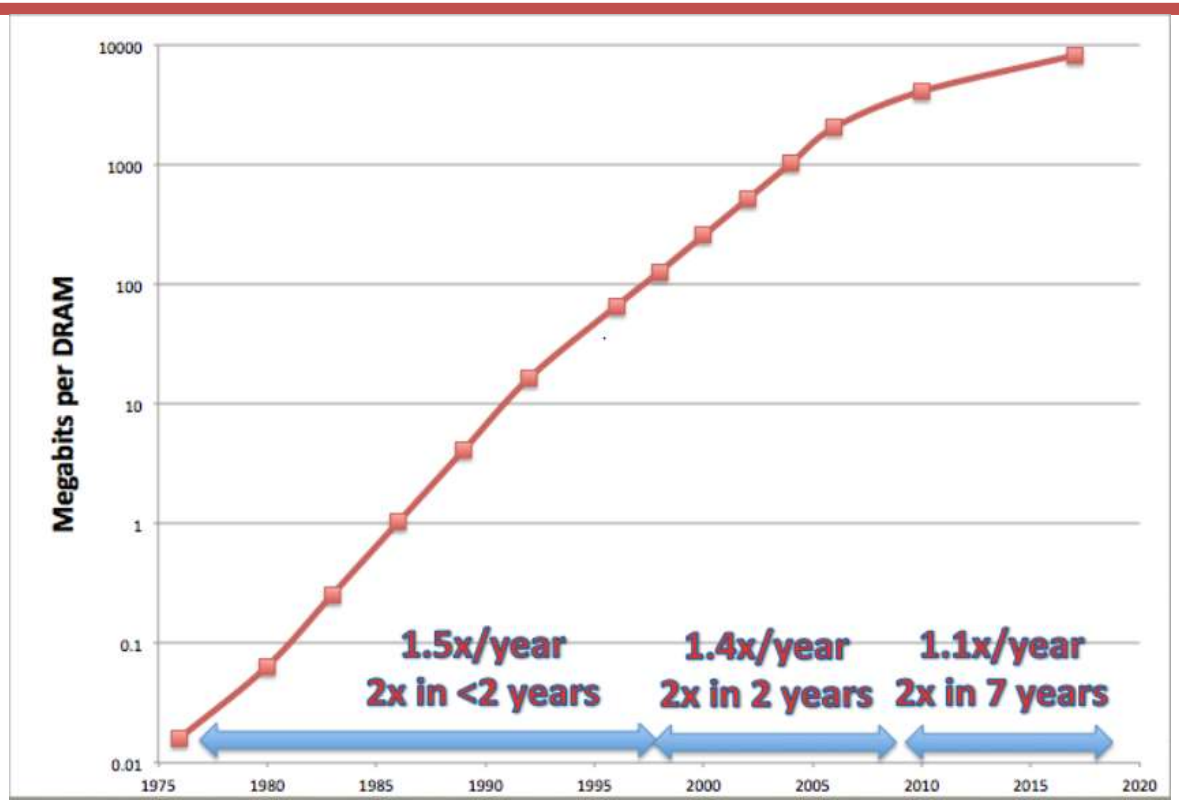


Image source: John Hennessy

After multi-core, specialization (i.e., accelerators) seems to be the most attractive architectural option to cope with the end of Moore's Law

The High Cost of Data Movement

Fetching operands more expensive than computing on them

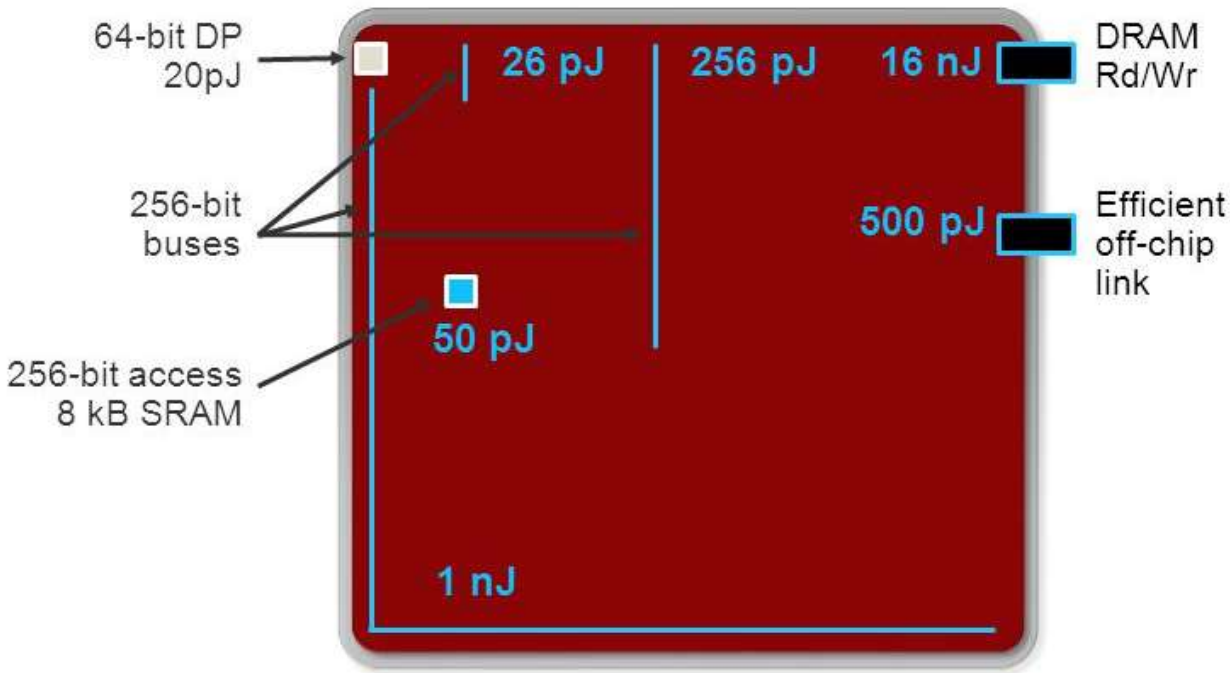


Image source: Bill Dally

Now the key is how we use our transistors most effectively.

Accelerator Design Attributes

- Integration into system
  - How is the accelerator integrated into the system?
- Operation specification/sequencing
  - How is activity managed within the accelerator?
- Data management
  - How is data movement orchestrated in the accelerator?

Remember, however, the world is '*fractal*'!

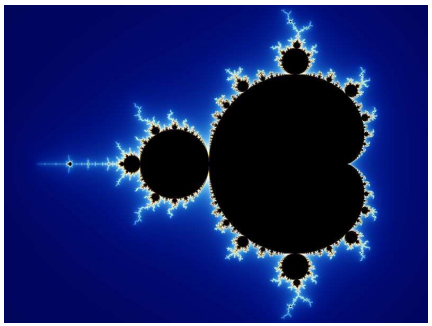
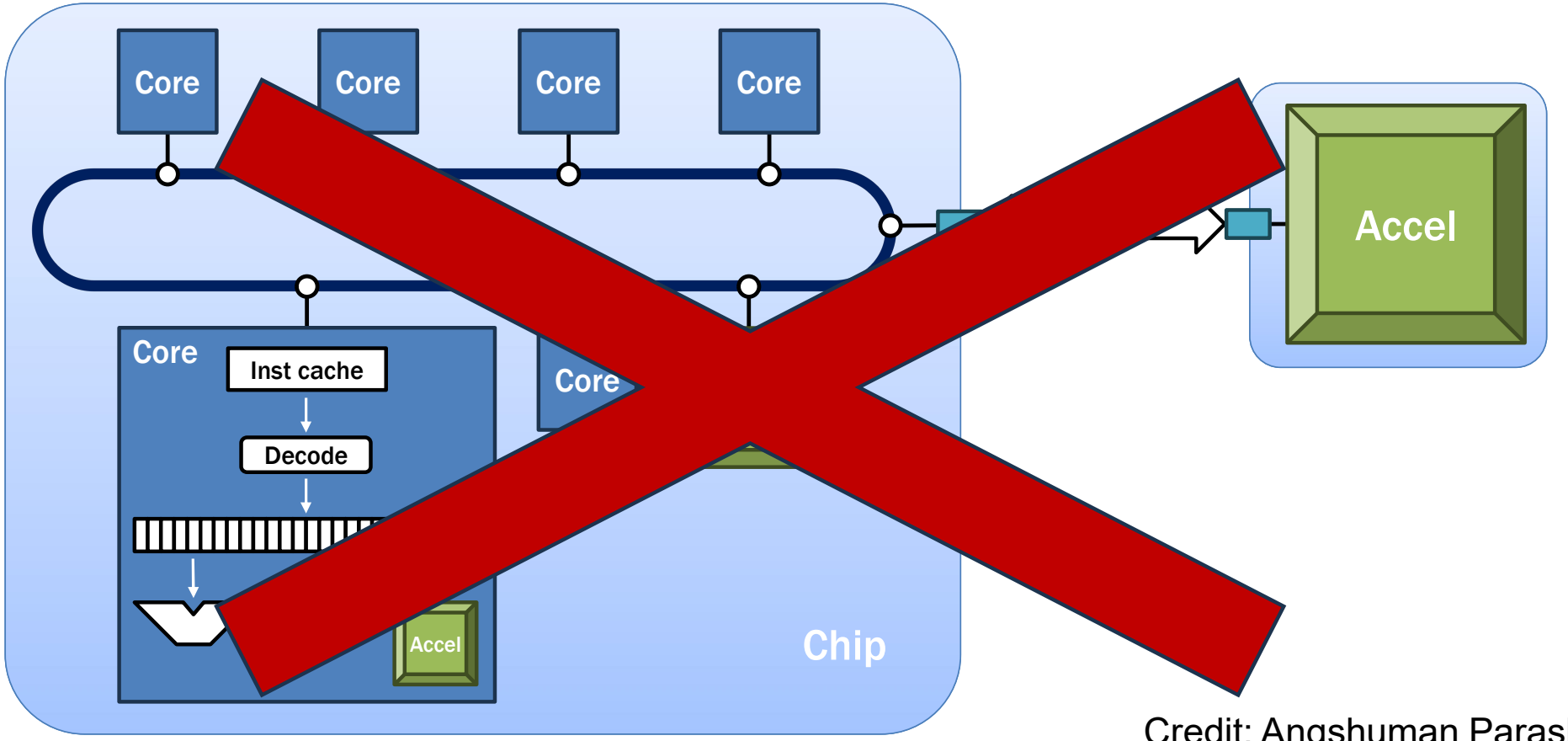


Image: Wikipedia

System Integration

Accelerator Integration Taxonomy



Credit: Angshuman Parashar

Accelerator Architectural Choices

- State
  - Local state – Is there local state? (i.e. context)
  - Global state – e.g., main memory shared with CPU
- Data Operations
  - Custom data operations in the accelerator
- Memory Access Operations
  - Operations to access shared global memory – Do they exist?
- Control Flow Operations
  - How is accelerator sequenced relative to CPU?

How to Sequence Accelerator Logic?

- Synchronous
  - Accelerated operations inside processor pipeline
    - E.g., as a separate function unit
  - Control handled by standard control instructions

- Asynchronous
  - A standalone logical machine
    - Accelerator started by processor that continues running

What factors mitigate for one form or the other?

Latency of operation

Size of logic

Existence of concurrent activity

Size of operands

Eight Alternatives

Architectural semantics		
Asynchronous	**Accesses memory**	**Has context**
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Characteristics of "no memory access" choice?

Good for smaller operands

Simpler, e.g., no virtual memory

No 'Little's Law' storage requirement

Eight Alternatives

Architectural semantics		
Asynchronous	**Accesses memory**	**Has context**
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Characteristics of “no context” choice?

Simpler, no context switch mechanism

Long operations run to completion

More limited reuse opportunities

Accelerator Architectural Alternatives

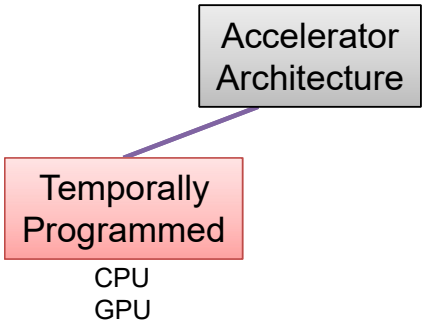
Architectural semantics			Example
Asynchronous	Accesses memory	Has context	
0	0	0	New function unit, like tensor core in GPU
0	0	1	Accumulating data reduction instruction
0	1	0	Memory-to-memory vector unit
0	1	1	Register-based vector unit including load store ops
1	0	0	Complex function calculator?
1	0	1	Security co-processor (TPM)
1	1	0	Network adapter
1	1	1	GPU with virtual memory

Accelerator Architectural Alternatives

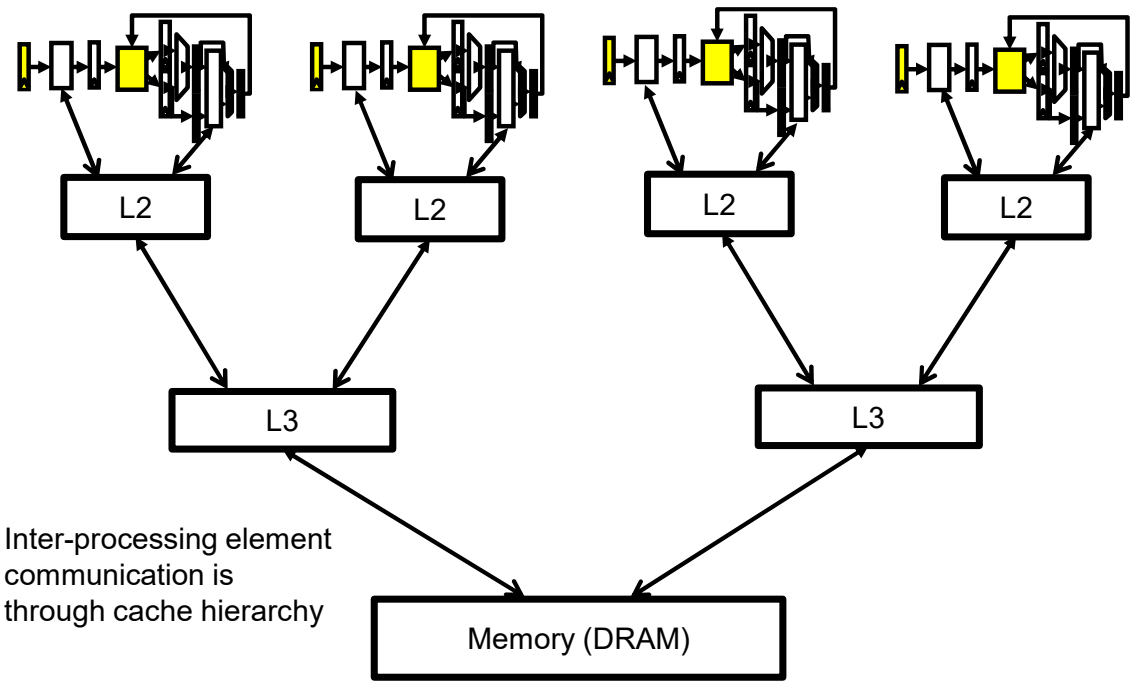
Architectural semantics			Example
Asynchronous	Accesses memory	Has context	
0	0	0	New function unit, like tensor core in GPU

Operation Sequencing

Accelerator Taxonomy



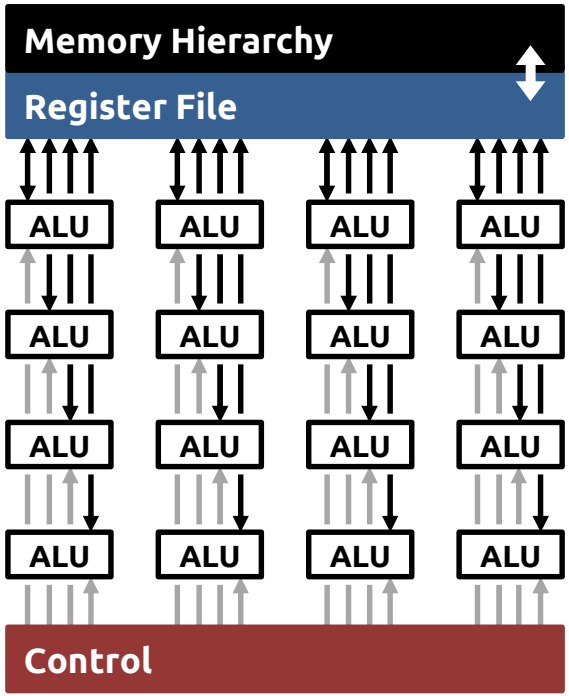
Multiprocessor



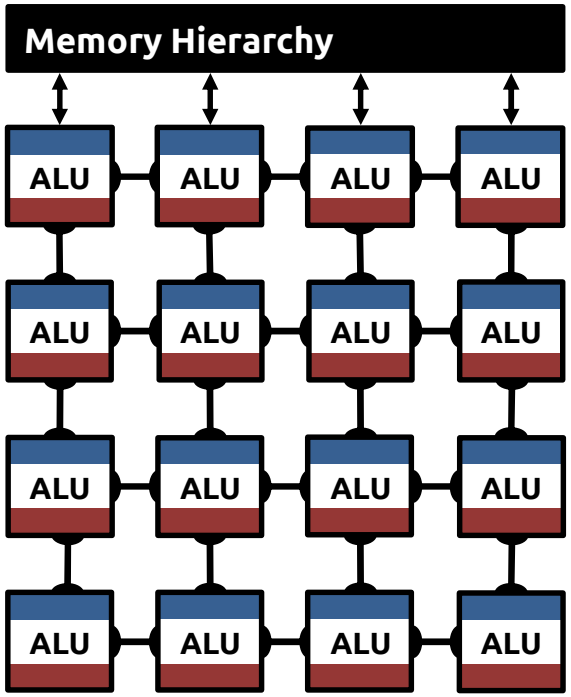
Inter-processing element communication is through cache hierarchy

Highly-Parallel Compute Paradigms

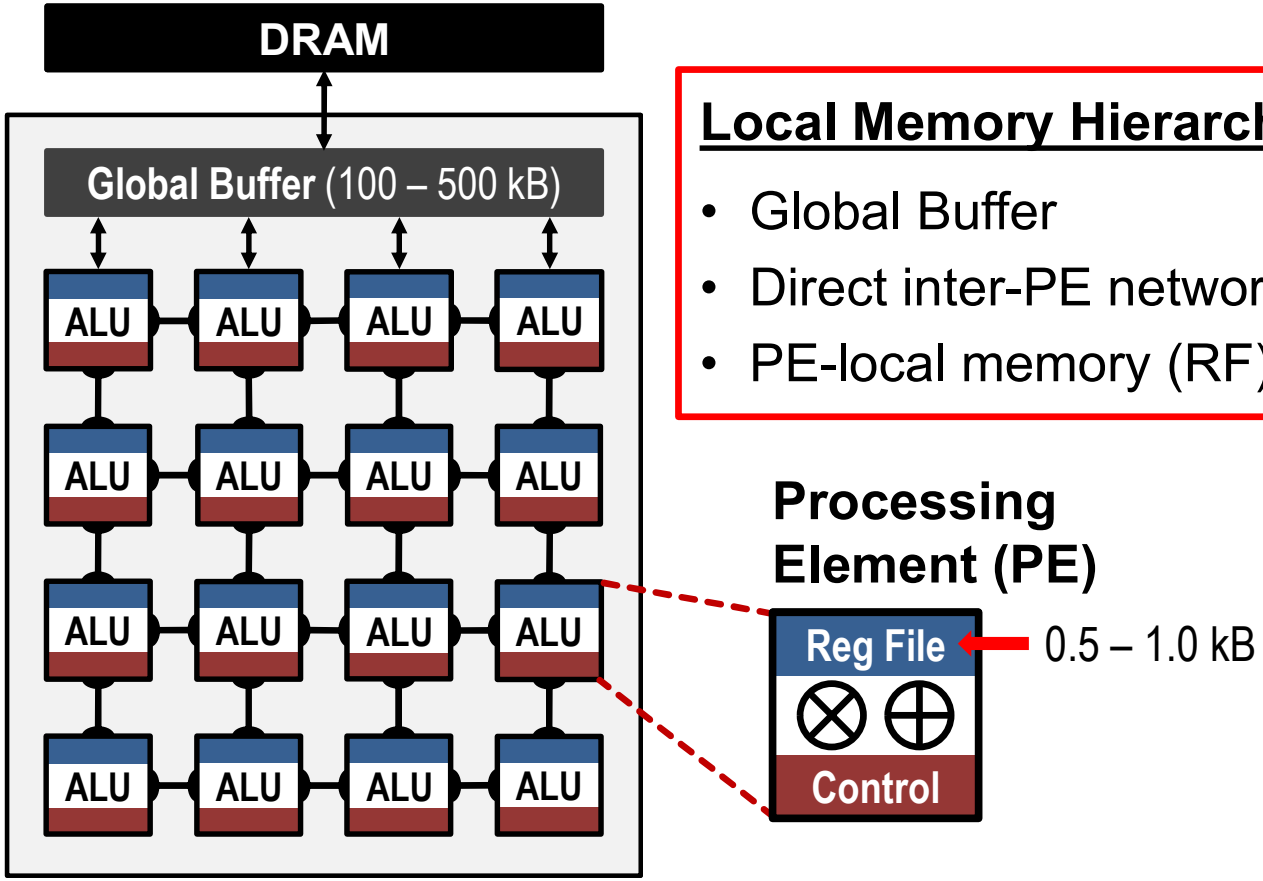
Temporal Architecture (SIMD/SIMT)



Spatial Architecture (Dataflow Processing)



Spatial Architecture for DNN

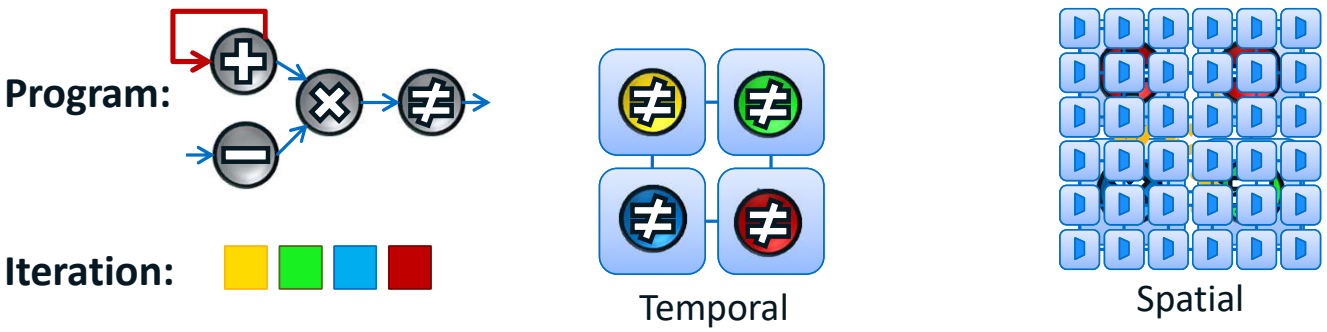


Local Memory Hierarchy

- Global Buffer
- Direct inter-PE network
- PE-local memory (RF)

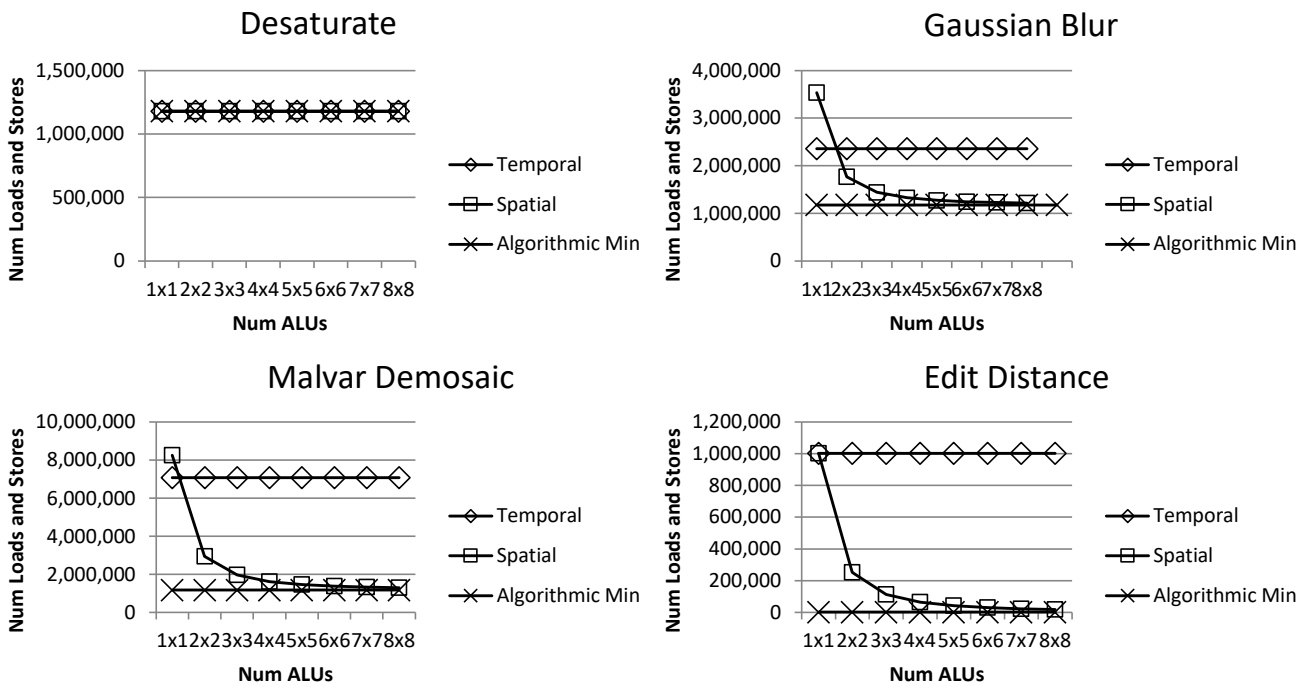
Processing Element (PE)

Parallel Control: Temporal vs Spatial



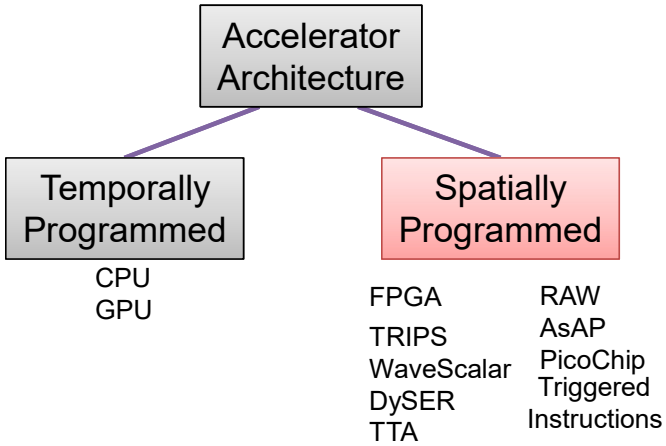
- Style dictates many later hardware choices
  - Temporal: Dynamic instruction stream, shared memory
  - Spatial: Tiny processing elements (PEs), direct communication

Memory Access: Spatial vs Temporal

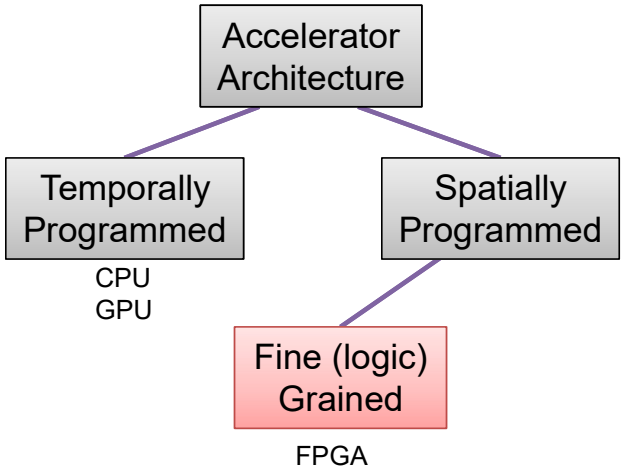


Large benefits with even small array sizes

Accelerator Taxonomy

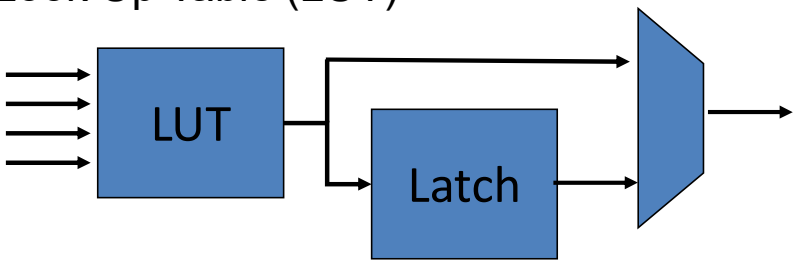


Accelerator Taxonomy



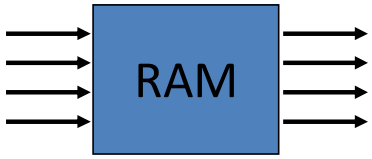
Field Programmable Gate Arrays

Look Up Table (LUT)

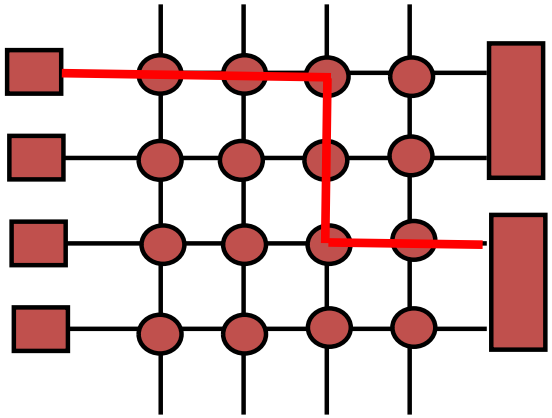


And	
00	0
01	0
10	0
11	1

Or	
00	0
01	0
10	1
11	1

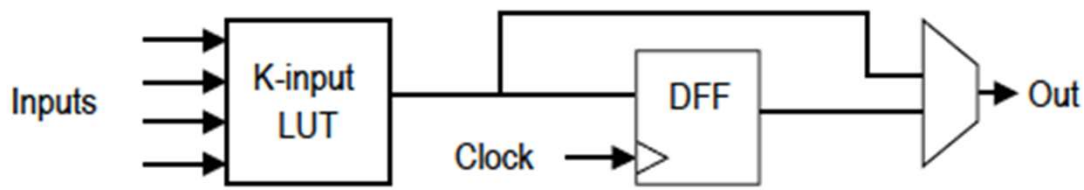


.....



Configurable Logic Blocks (CLB)

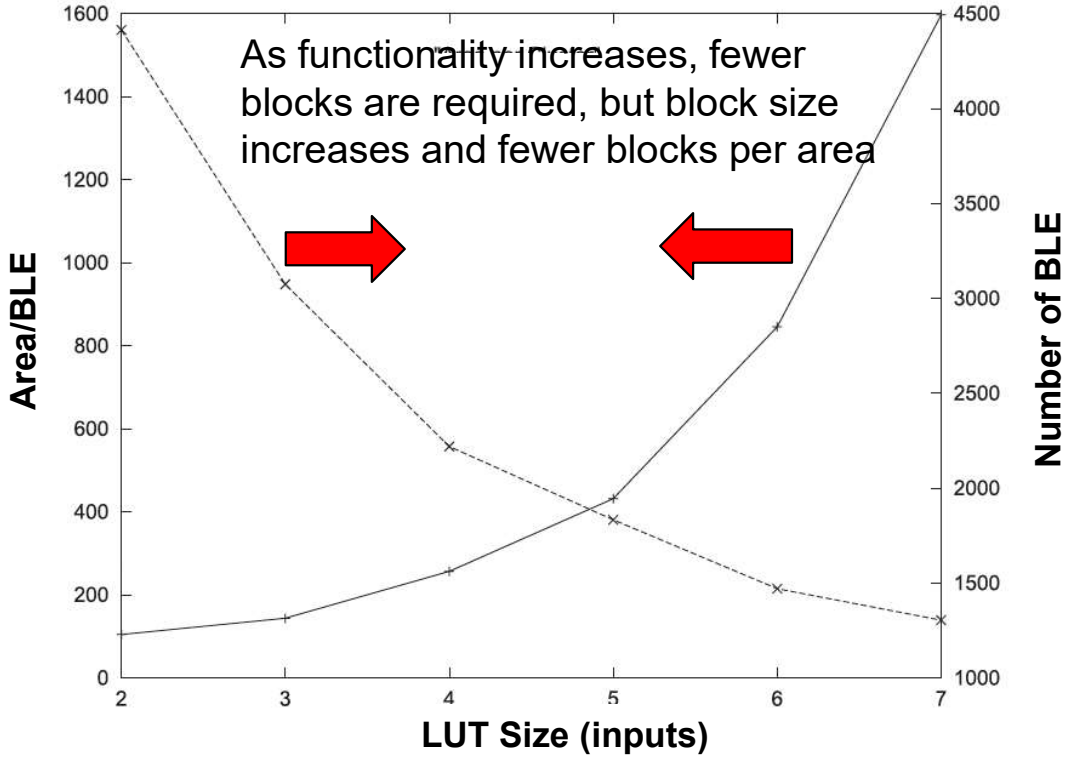
- CLB used to implement sequential and combinational logic
- CLB are comprised of several Basic Logic Elements (BLE)
- Each BLE contains:
  - Look up tables (LUT) are used to implement logic function
  - Registers to store data
  - Multiplexer to select desired output



As number of inputs grow (k), increase size of LUT by 2^k and routing

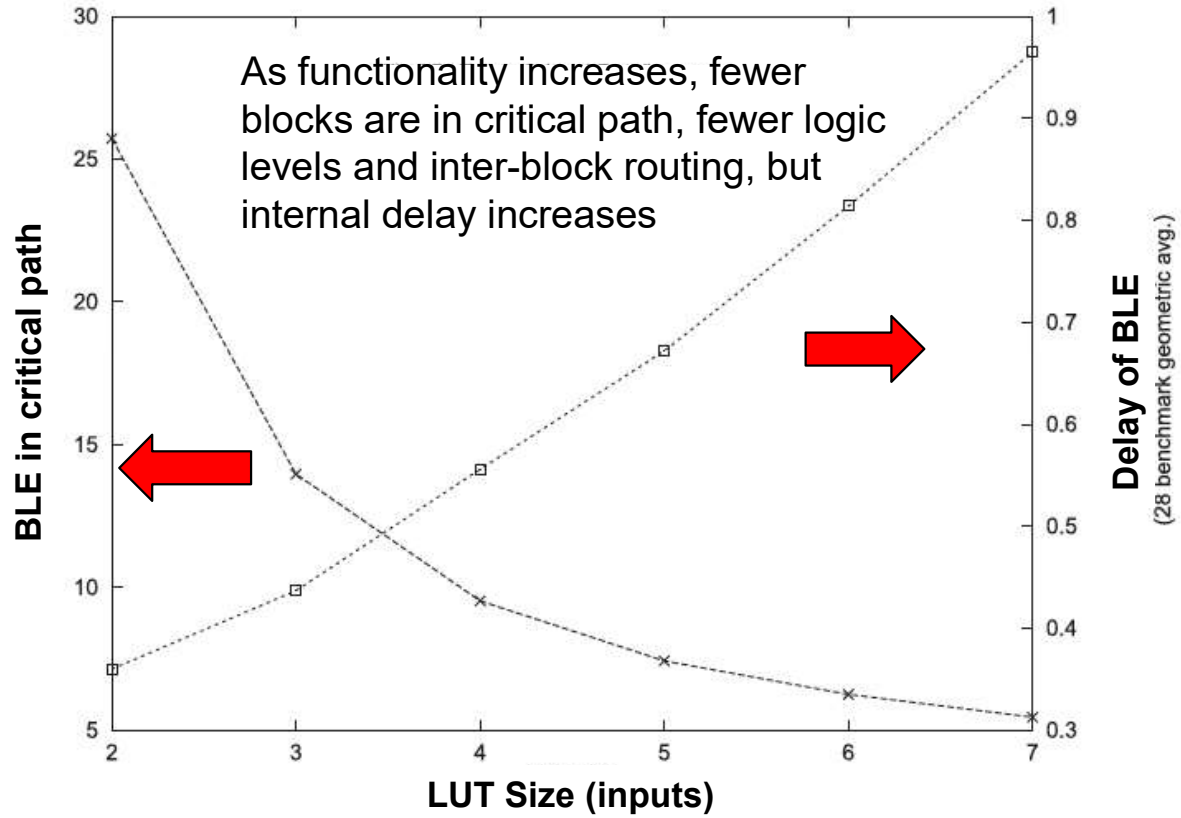
Kuon, Ian, Russell Tessier, and Jonathan Rose. "FPGA architecture: Survey and challenges." Foundations and Trends in Electronic Design Automation 2.2 (2008): 135-253.

Area Trade-off (Size of LUT)



Kuon, Ian, Russell Tessier, and Jonathan Rose. ”FPGA architecture: Survey and challenges." Foundations and Trends in Electronic Design Automation 2.2 (2008): 135-253.

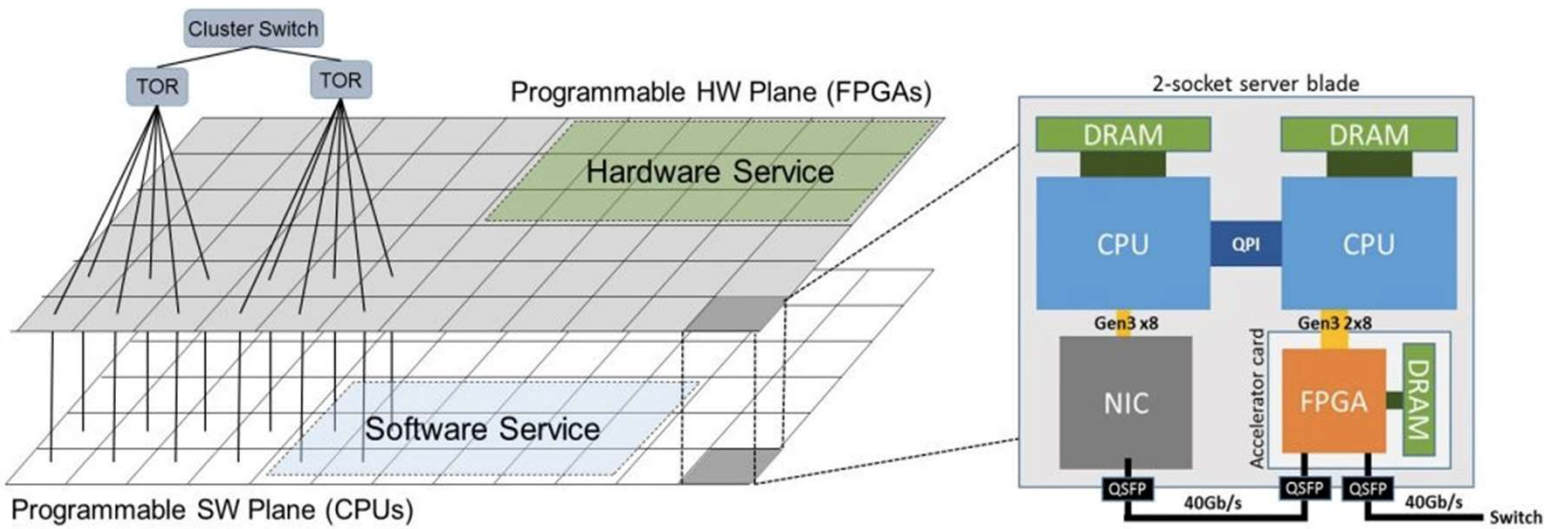
Size of LUT (Speed Trade-off)



Kuon, Ian, Russell Tessier, and Jonathan Rose. "FPGA architecture: Survey and challenges." Foundations and Trends in Electronic Design Automation 2.2 (2008): 135-253.

Microsoft Project Catapult

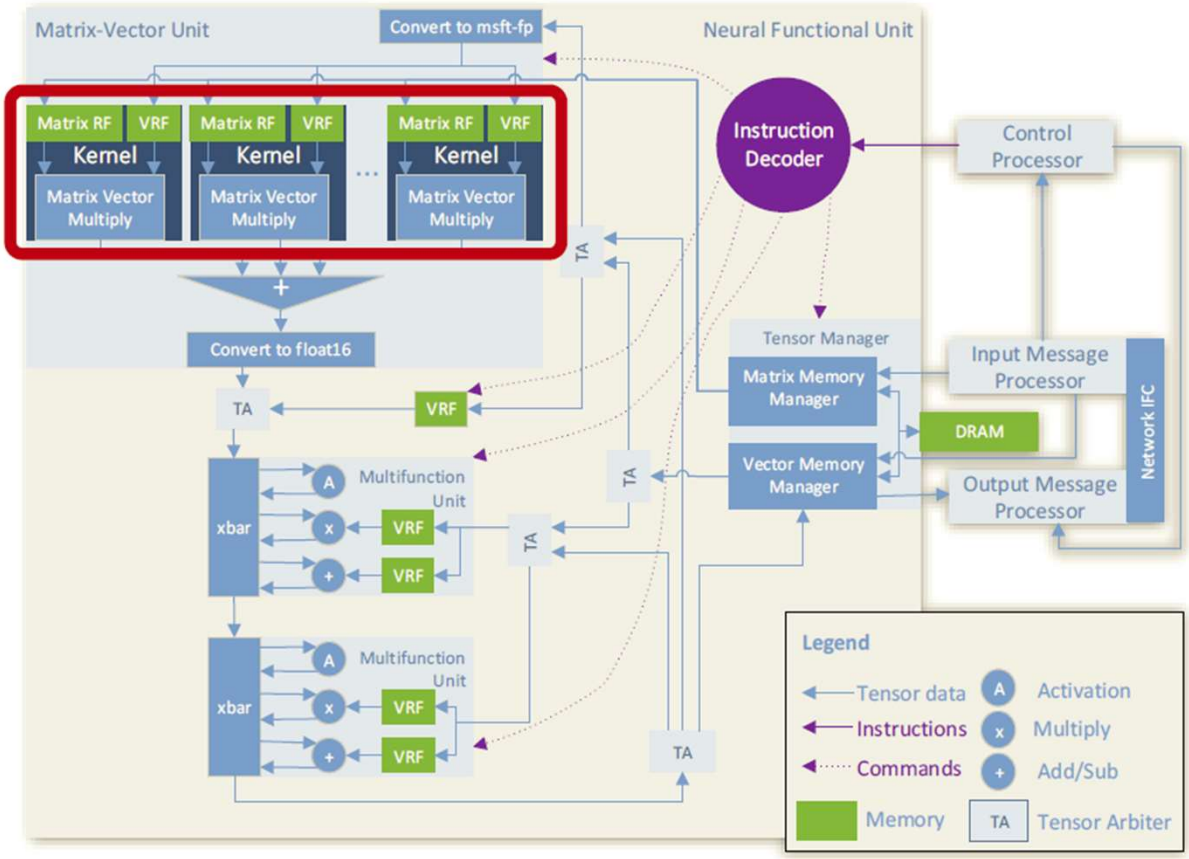
Configurable Cloud (MICRO 2016) for Azure



Accelerate and reduce latency for

- Bing search
- Software defined network
- Encryption and Decryption

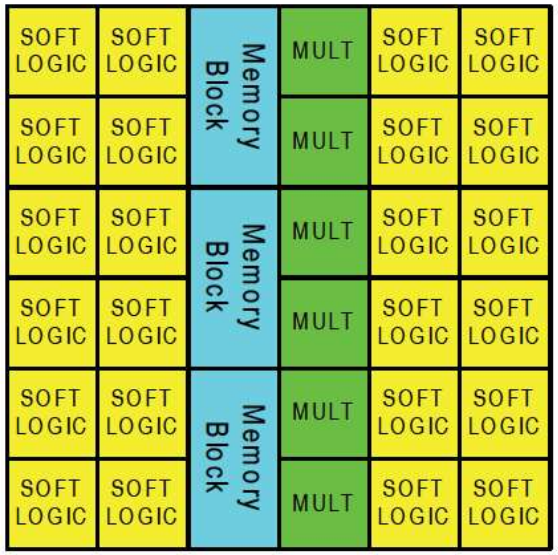
Microsoft Brainwave Neural Processor



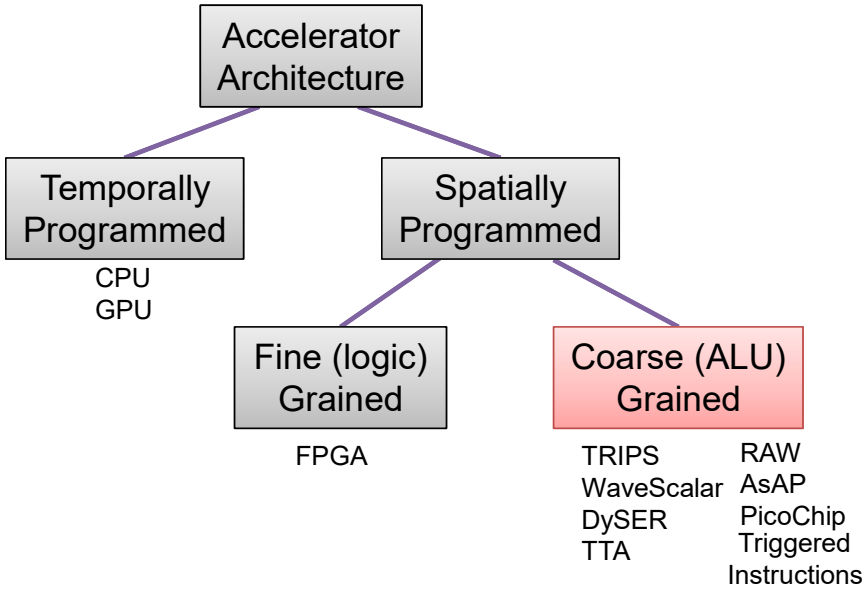
Source: Microsoft

Heterogeneous Blocks

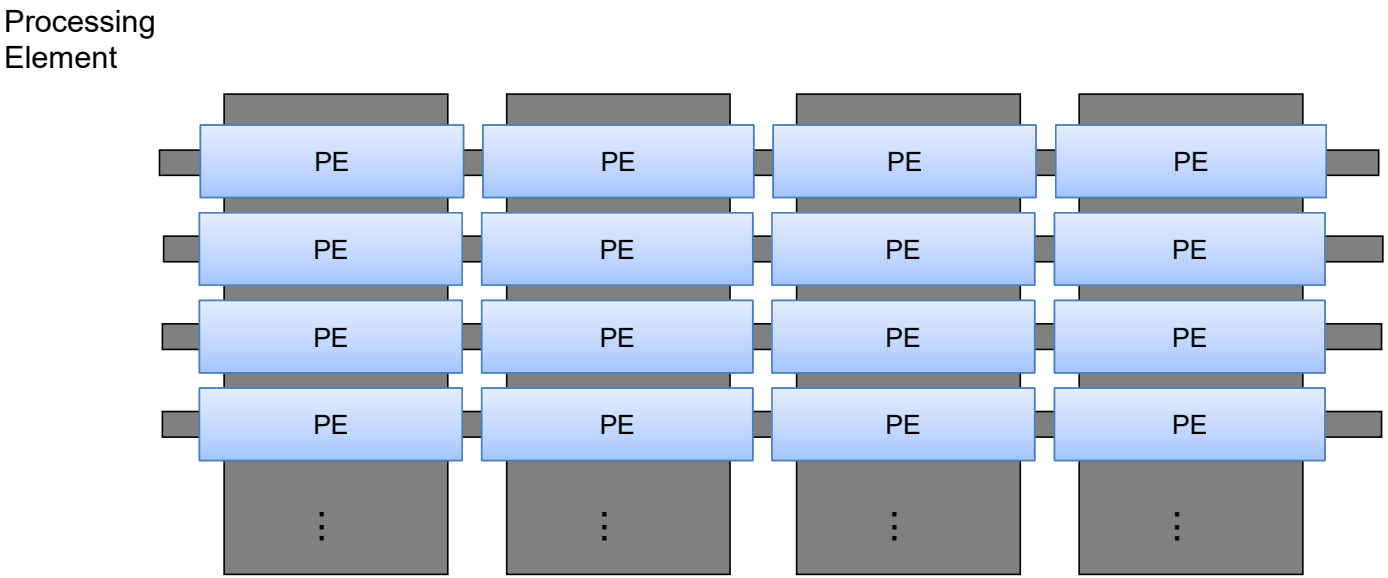
- Add *specific purpose logic* on FPGA
  - Efficient if used (better area, speed, power), wasted if not
- Soft fabric
  - LUT, flops, addition, subtraction, carry logic
  - Convert LUT to memories or shift registers
- Memory block (BRAM)
  - Configure word and address size (aspect ratio)
  - Combine memory blocks to large blocks
  - Significant part for FPGA area
  - Dual port memories (FIFO)
- Multipliers /MACs → DSP
- CPUs and processing elements



Accelerator Taxonomy

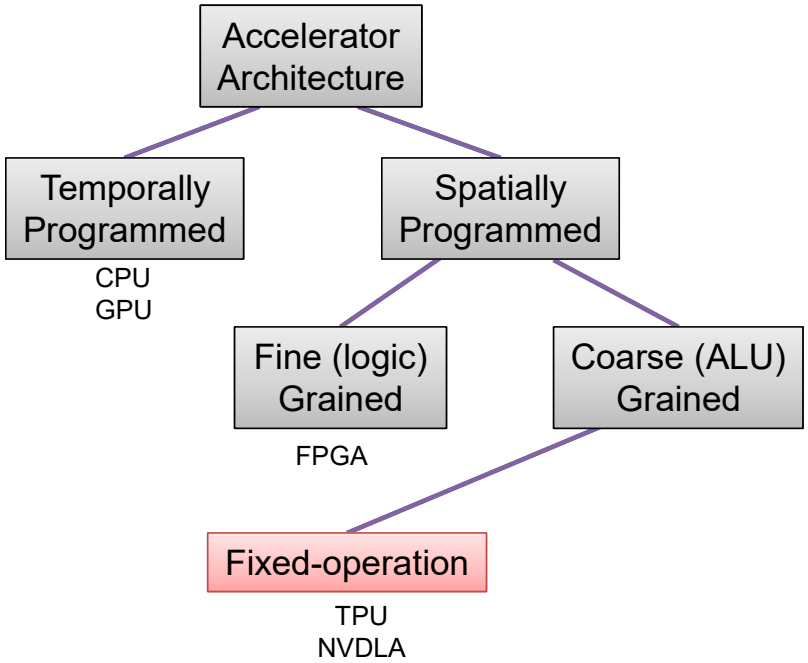


Programmable Accelerators



Many Programmable Accelerators look like an array of PEs, but have dramatically different architectures, programming models and capabilities

Accelerator Taxonomy

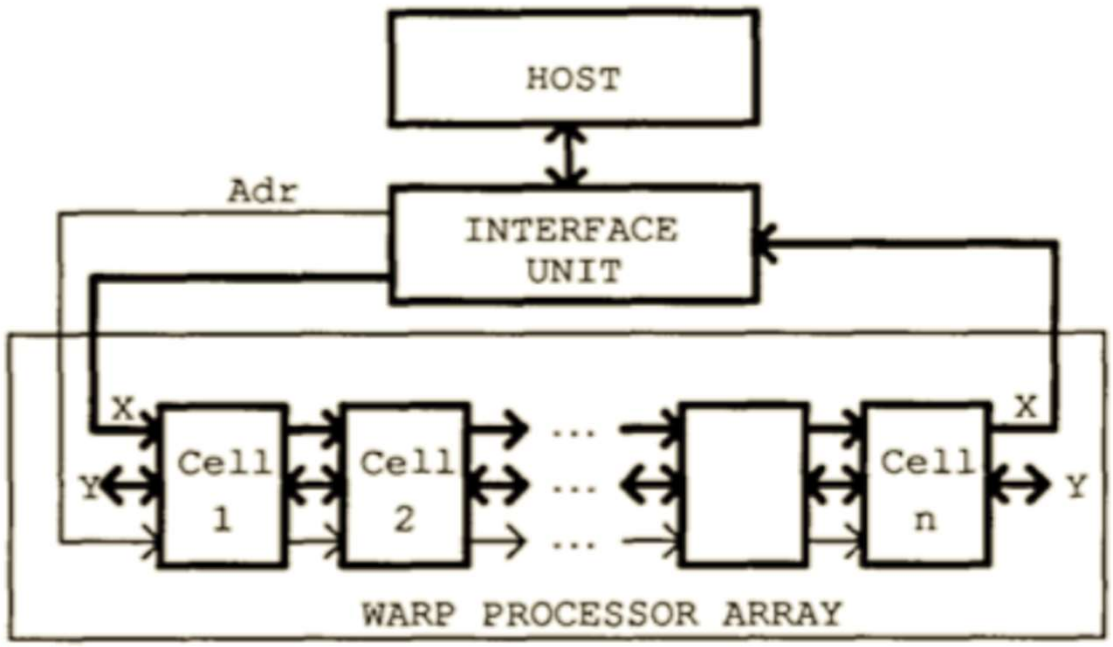


Fixed Operation - Systolic Array

- Each PE hard-wired to one operation
- Purely pipelined operation
  - no backpressure in pipeline

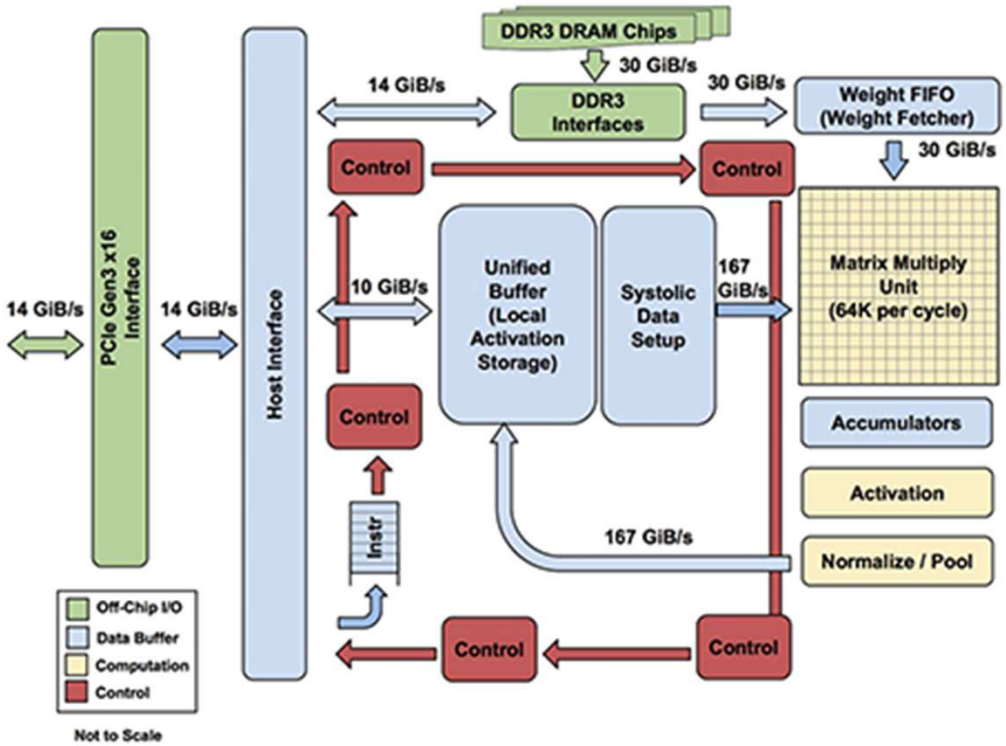
- Attributes
  - High-concurrency
  - Regular design, but
  - Regular parallelism only!

Configurable Systolic Array - WARP



Source: WARP Architecture and Implementation, ISCA 1986

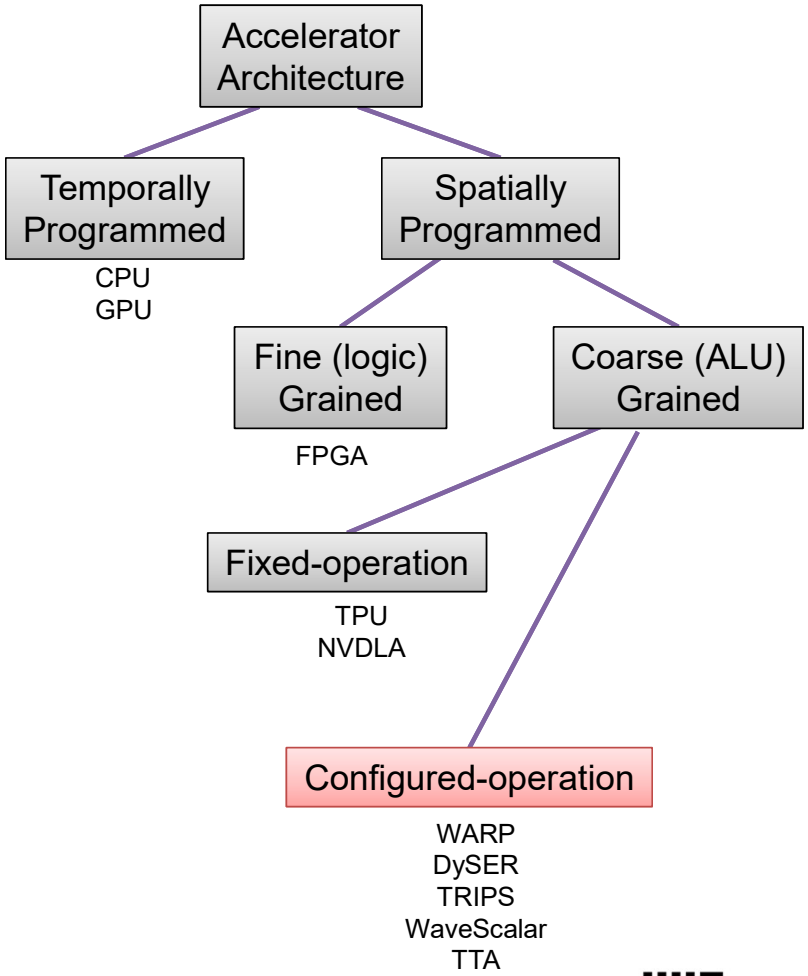
Fixed Operation - Google TPU



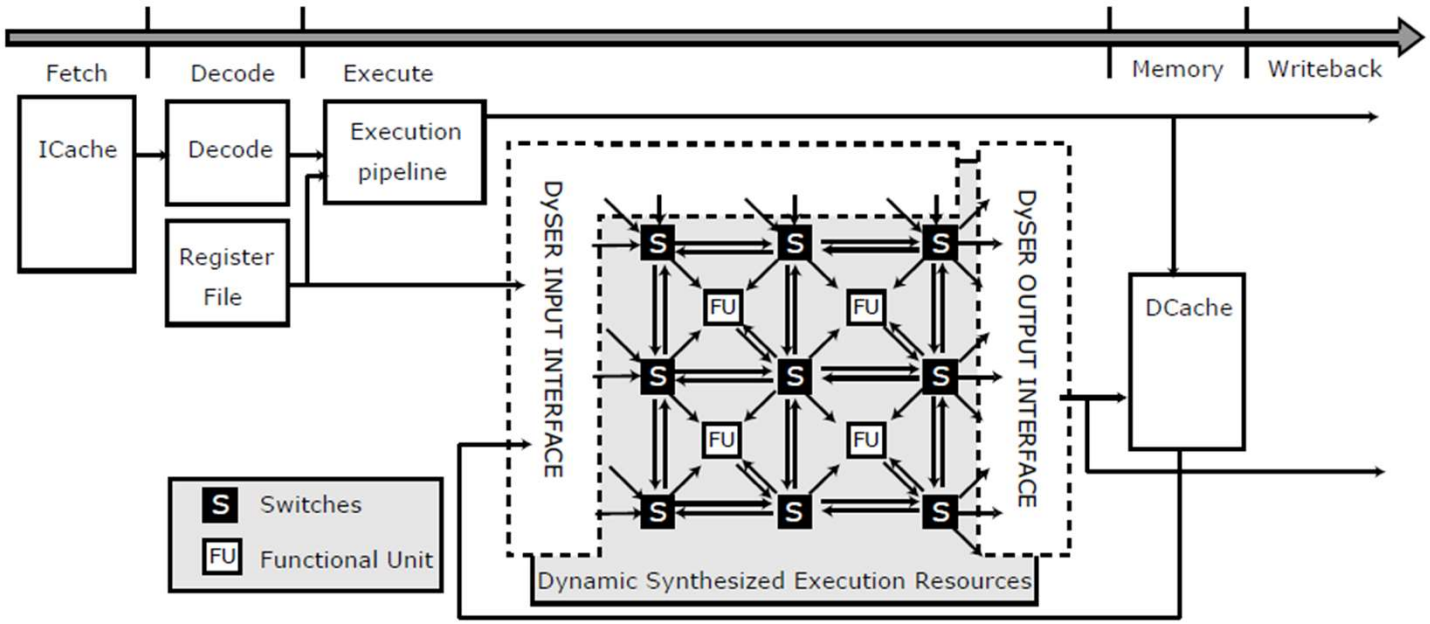
Systolic array does 8-bit 256x256 matrix-multiply accumulate

Source: Google

Accelerator Taxonomy

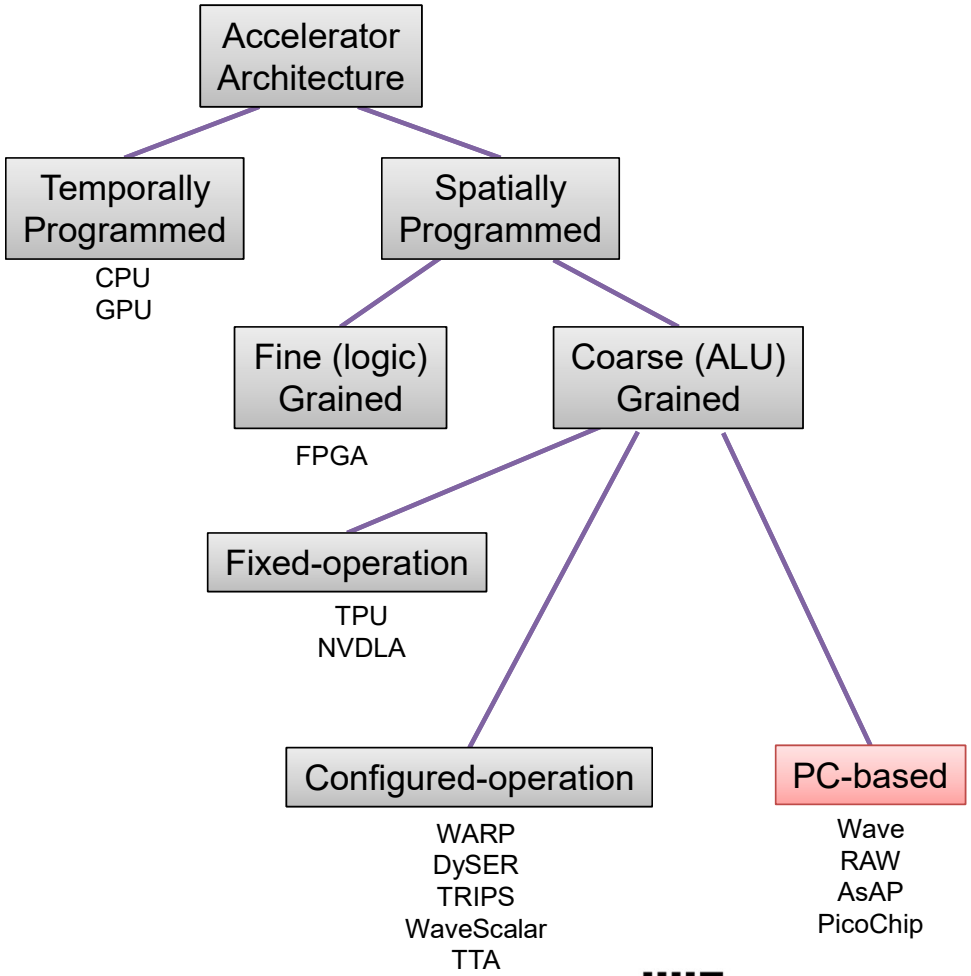


Single Configured Operation - Dyser

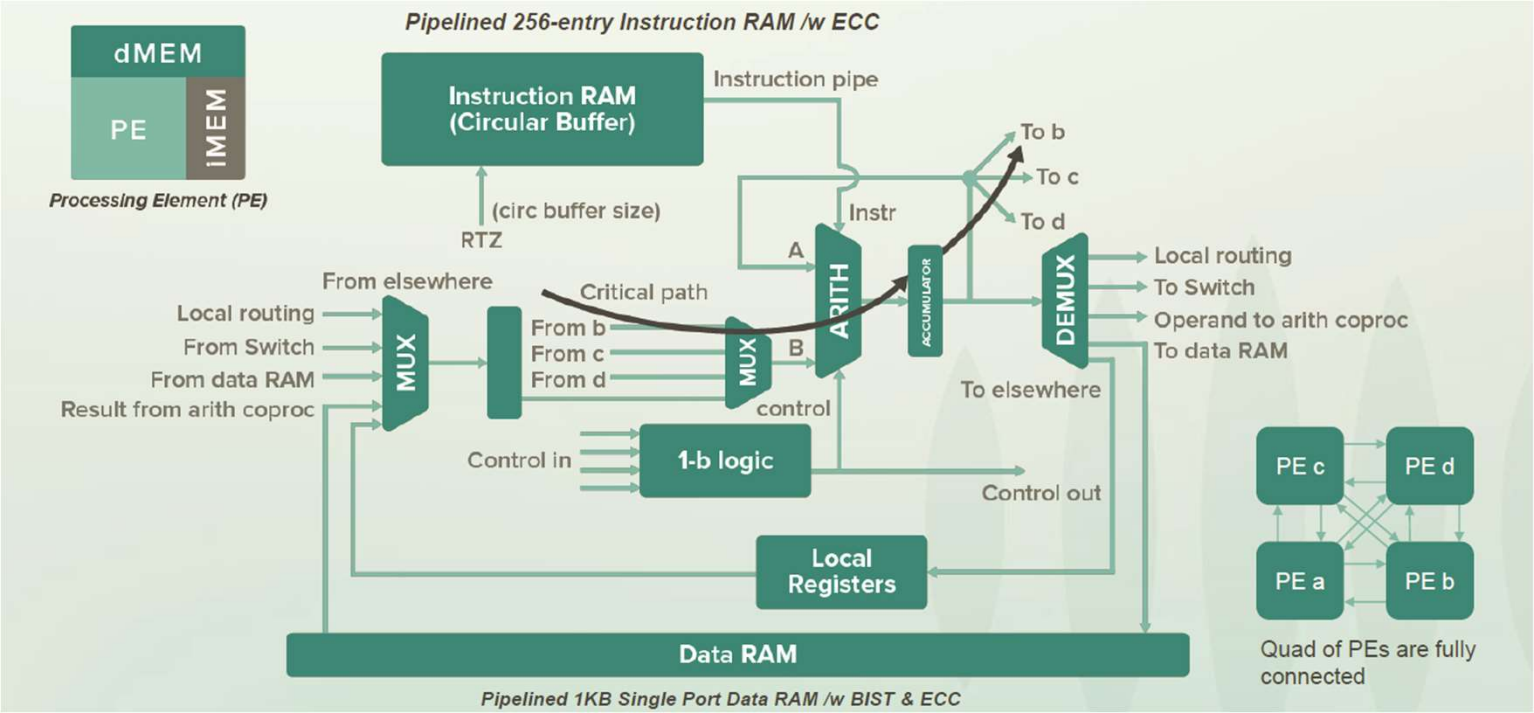


Source: Dynamically Specialized Datapaths for Energy Efficient Computing. HPCA11

Accelerator Taxonomy

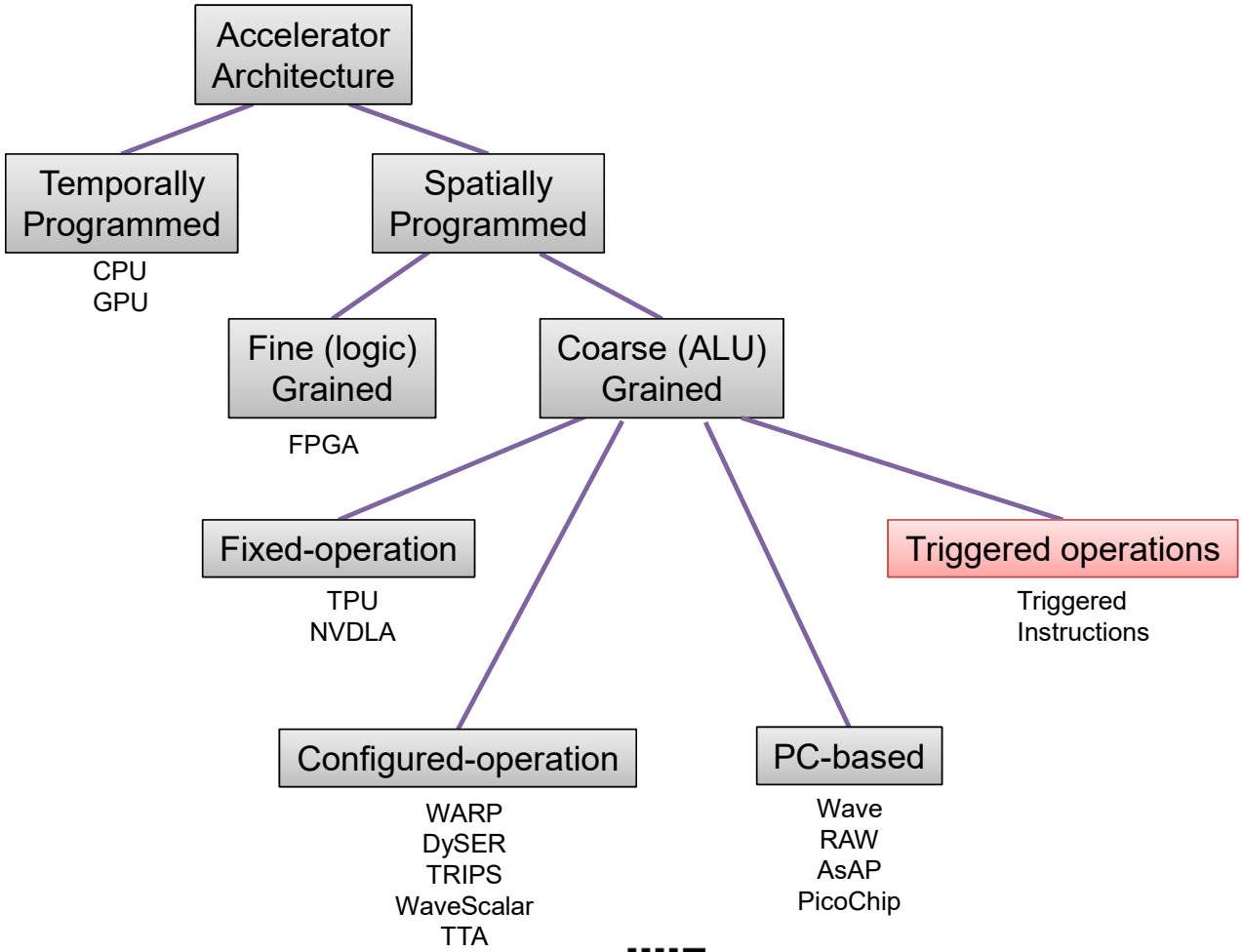


PC-based Control – Wave Computing



Source: Wave Computing, Hot Chips '17

Accelerator Taxonomy



Guarded Actions

```
reg A;    reg B;    reg C;

rule X (A > 0 && B != C)
{
    A <= B + 1;
    B <= B - 1;
    C <= B * A;
}

rule Y (…) {…}

rule Z (…) {…}

Scheduler
```

- Program consists of **rules** that may perform computations and read/write state
- Each rule specifies conditions (**guard**) under which it is allowed to fire
- Separates description and execution of data (rule body) from control (guards)
- A **scheduler** is generated (or provided by hardware) that evaluates the guards and schedules rule execution
- Sources of Parallelism
  - Intra-Rule parallelism
  - Inter-Rule parallelism
  - Scheduler overlap with Rule execution
  - Parallel access to state

Triggered Instructions (TI)

- Restrict guarded actions down to efficient ISA core:

```
doPass when (p_did_cmp && !p_cur_is_larger)
%out0.data = %in0.first;
%in0.deq;
p_did_cmp = false;
```

Trigger
read any number of 1-bit predicates

Operation
read/write data regs
channel control

Predicate Op
write 1-bit preds
(data-dependent)

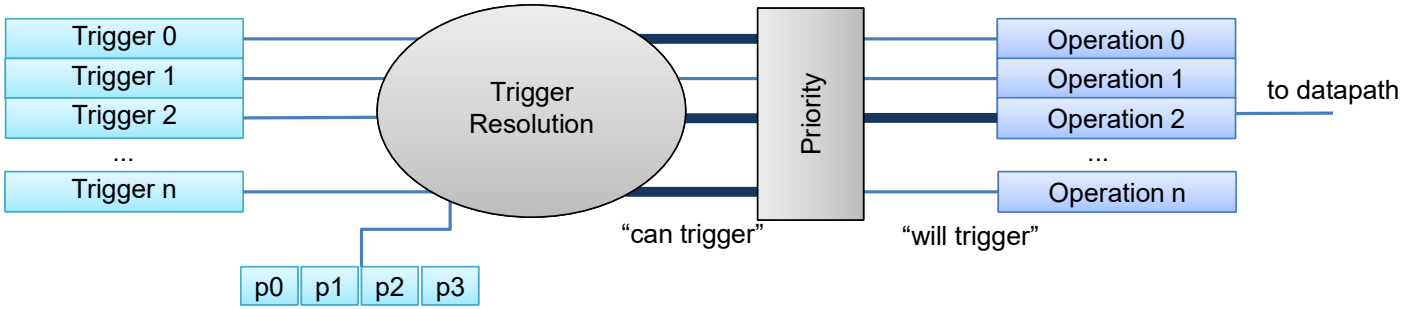
When can this happen?

What does it do?

What can happen next?

No program counter or branch instructions

Triggered Instruction Scheduler



- Use combinational logic to evaluate triggers in parallel
- Decide winners if more than one instruction is ready
  - Based on architectural fairness policy
  - Could pick multiple non-conflicting instructions to issue (superscalar)
- Note: no wires toggle unless status changes

Next Lecture: Dataflows

Thank you!