Deep Learning Processors CPU / GPU solutions

Chixiao Chen

Overview

- CPU – Vector / SIMD processors

- GPU – Multithread, SIMT, multi-GPU communciations
  - Tensor core
  - Winograd

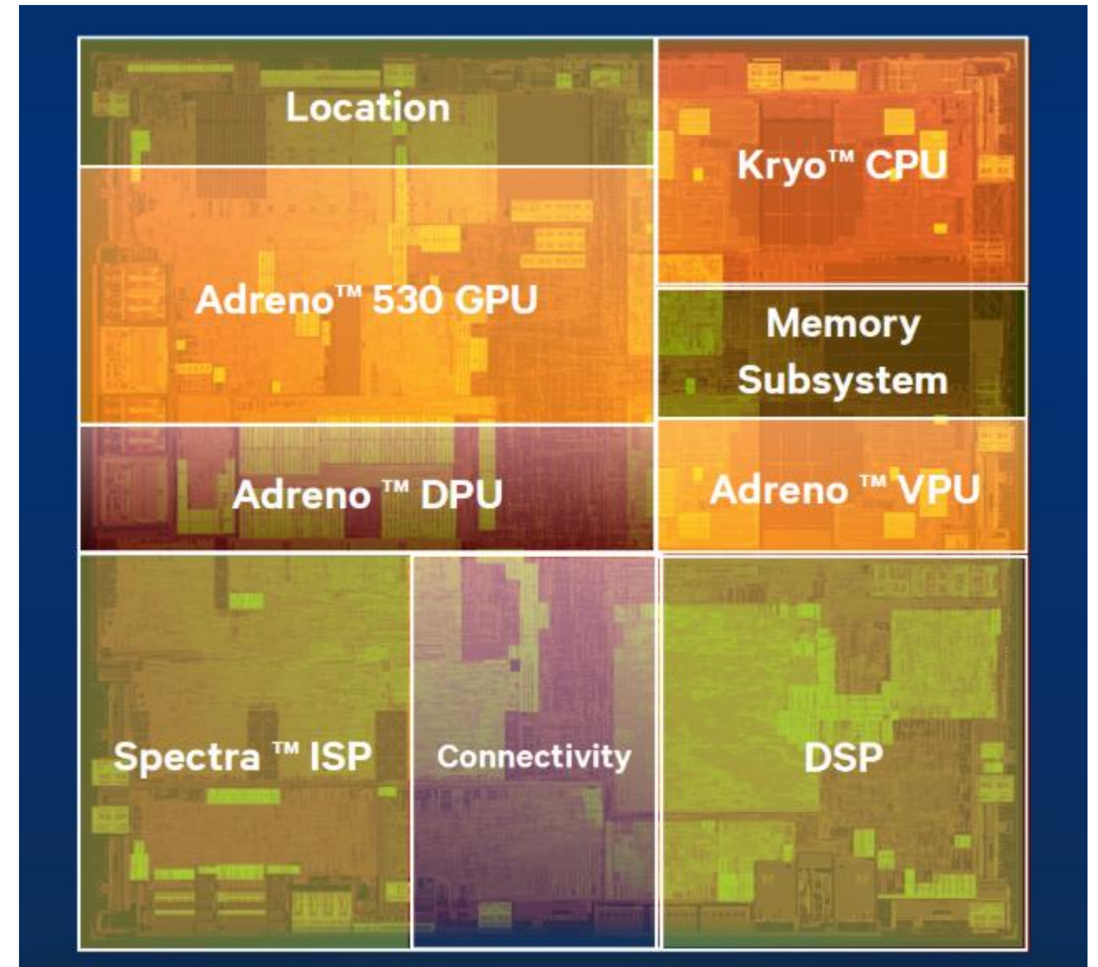
RISC CPU Evolution

for Deep Learning

RISC CPUs for Deep Learning

- Qualcomm Snapdragon 820 Mobile Processor
  - Using DSP & GPU
  - Specific Deep Learning SDK

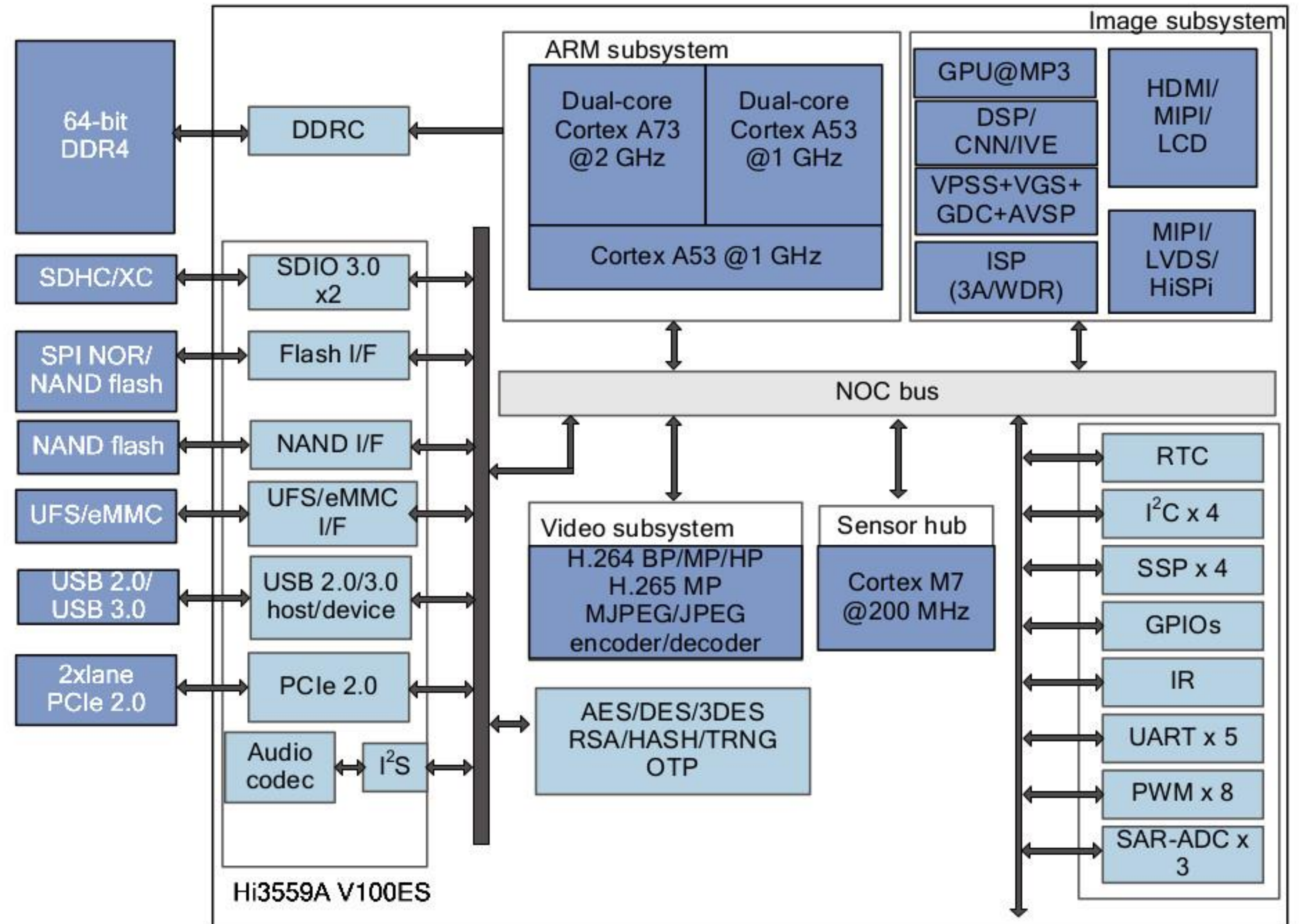
- Neuromophic IPs
  - Qualcomm Zeroth Neural Processing Unit



RISC CPUs for Deep Learning

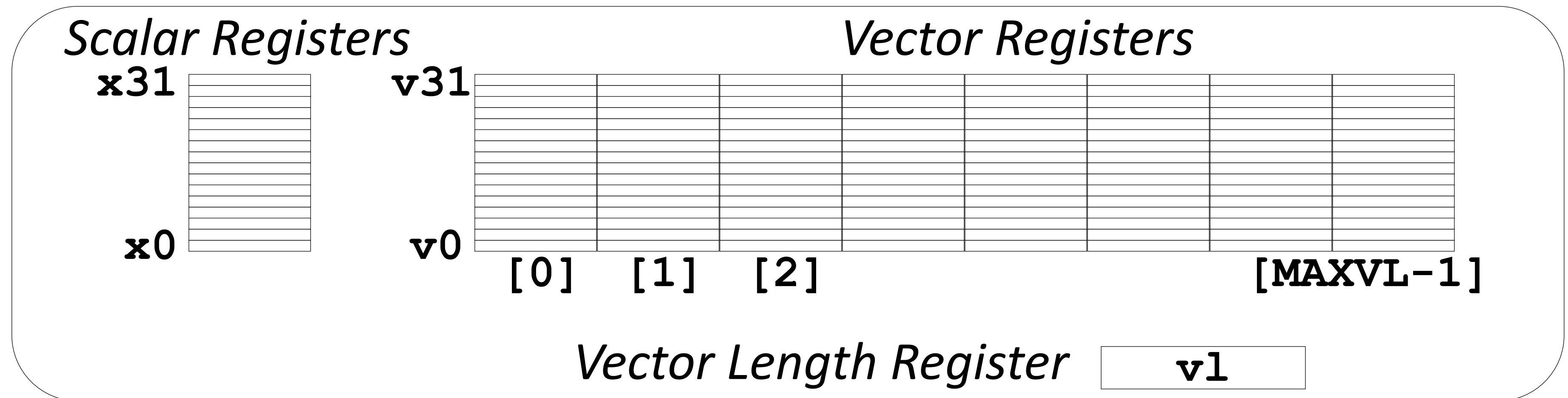
- Hisilicon H3559 for embedded video application

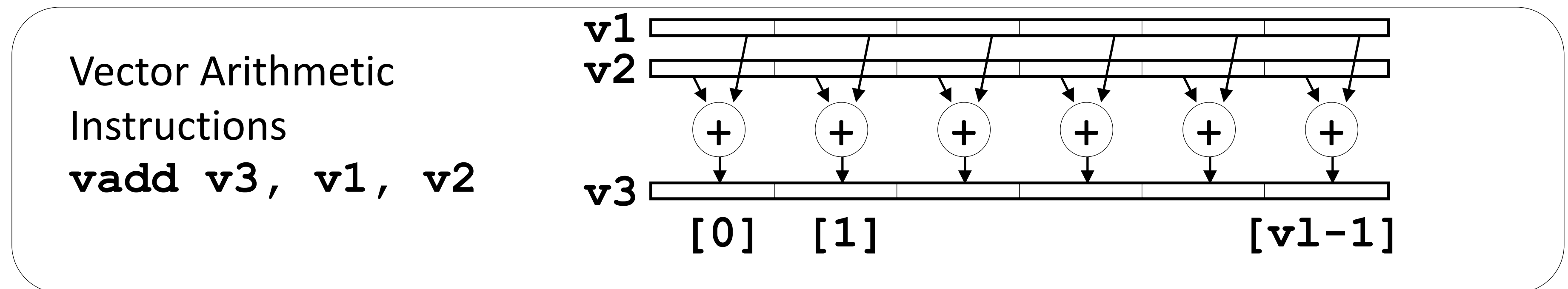
- Cortex IPs has SIMD vector instructions for Parallel Computing



Vector Processors

- Vector Architecture
- Sometimes also called SIMD





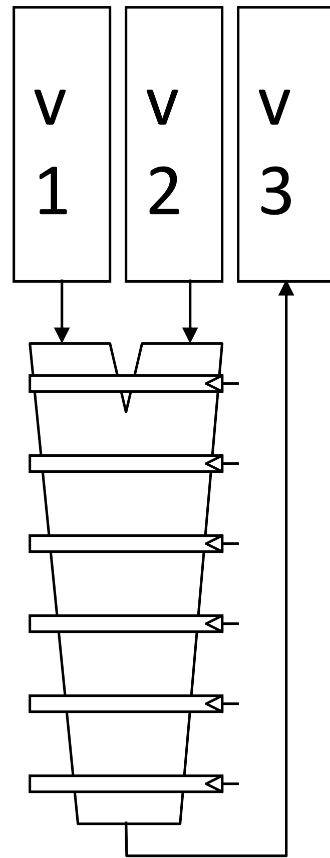
Assembly Code for Scalar and Vector Arch.

- One instruction for N operations
- But require they are **independent** but **same** type

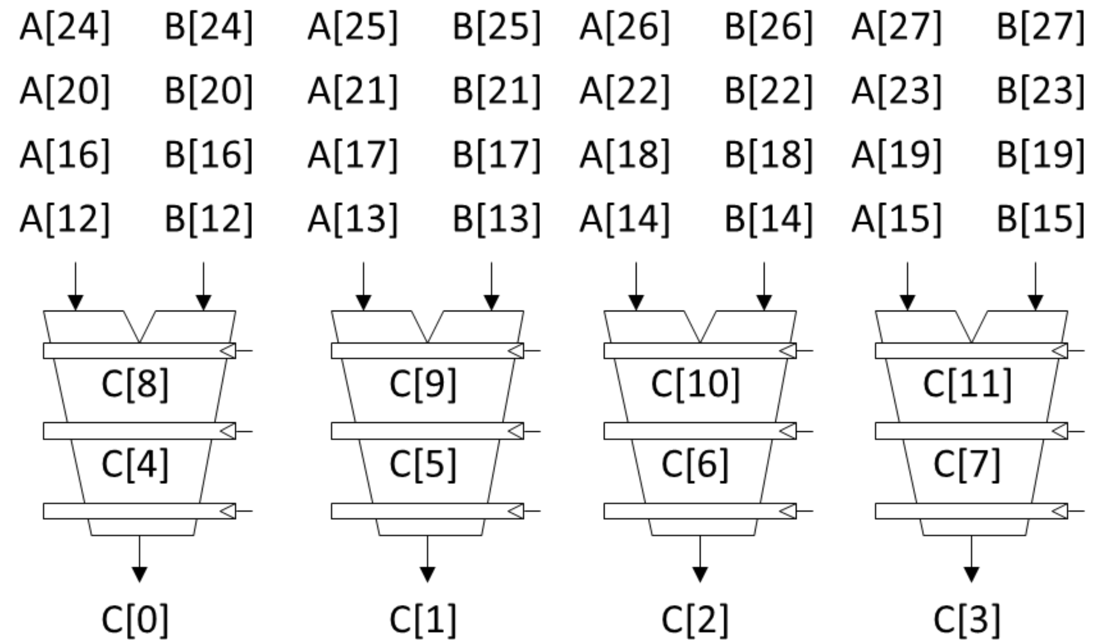
`# C code`
`for (i=0; i<64; i++)`
`  C[i] = A[i] + B[i];`
	`# Scalar Code`
	`  li x4, 64`
	`loop:`
	`  fld f1, 0(x1)`
	`  fld f2, 0(x2)`
	`  fadd.d f3,f1,f2`
	`  fsd f3, 0(x3)`
	`  addi x1, 8`
	`  addi x2, 8`
	`  addi x3, 8`
	`  subi x4, 1`
	`  bnez x4, loop`
		`# Vector Code`
		`  li x4, 64`
		`  setvl x4`
		`  vld v1, x1`
		`  vld v2, x2`
		`  vadd v3,v1,v2`
		`  vst v3, x3`

Vector Arithmetic Execution

- Method 1: Deeper Pipeline
- Advantage: No hazards !
- Example: A 6-stage multiply pipeline

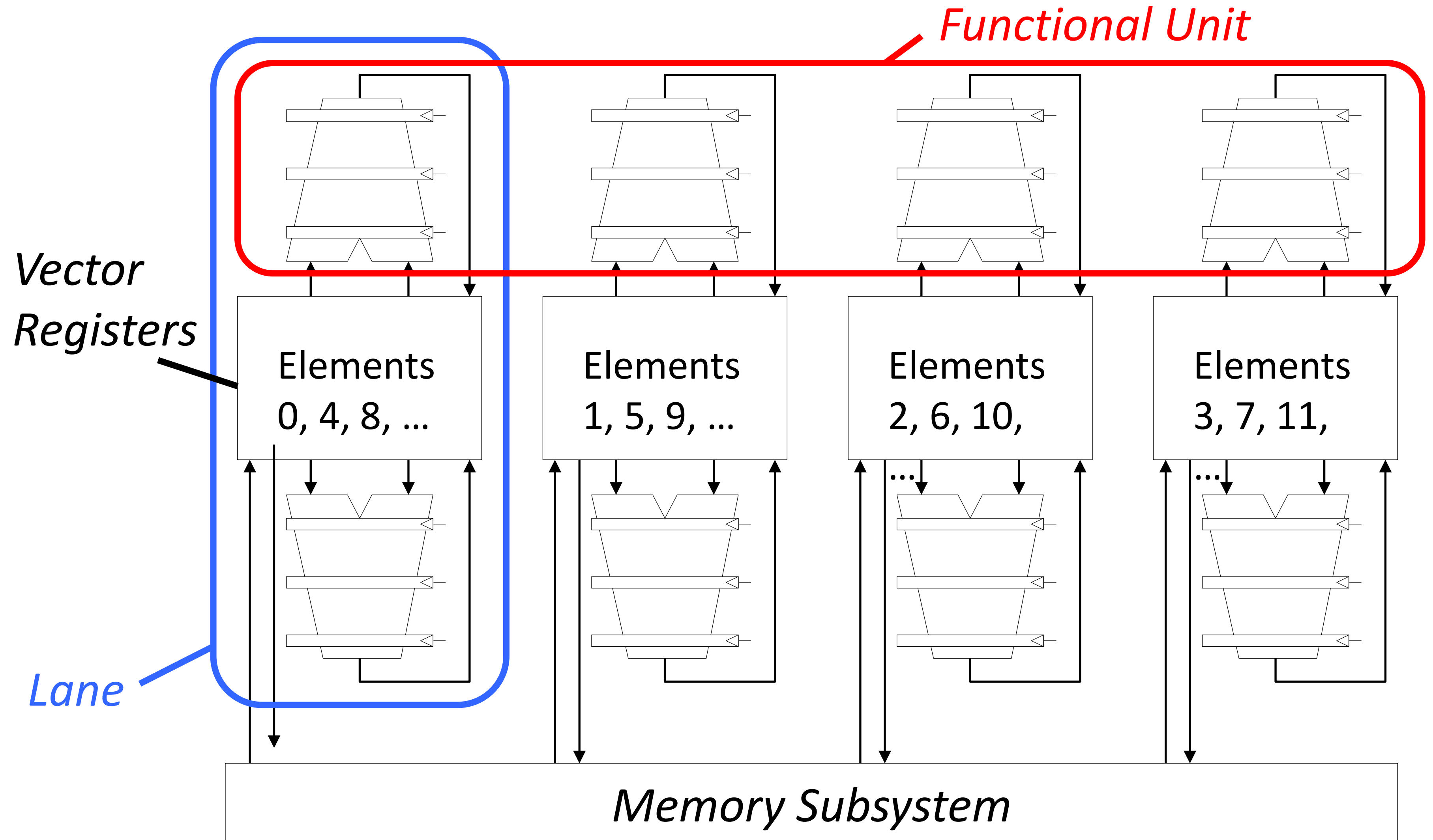


- Method 2: Interleaving
- Advantage: Power efficient



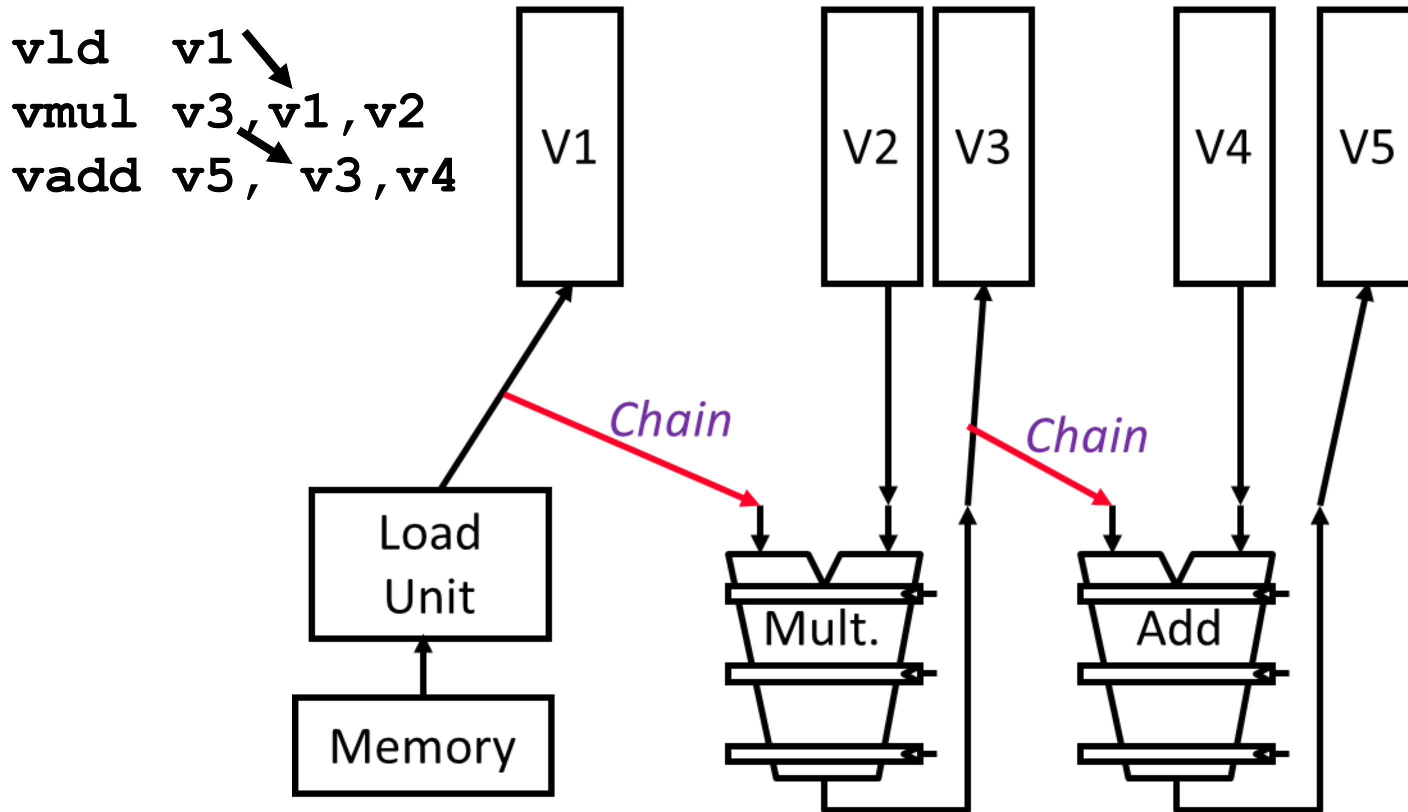
Practical Vector Unit Structure

- Interleaved memory system with busy time
- Based on pipeline depth.



Vector Chaining

- Parallel makes overlapping possible



Without Chaining: must wait until the results are generated

With Chaining: start immediately after the results appears

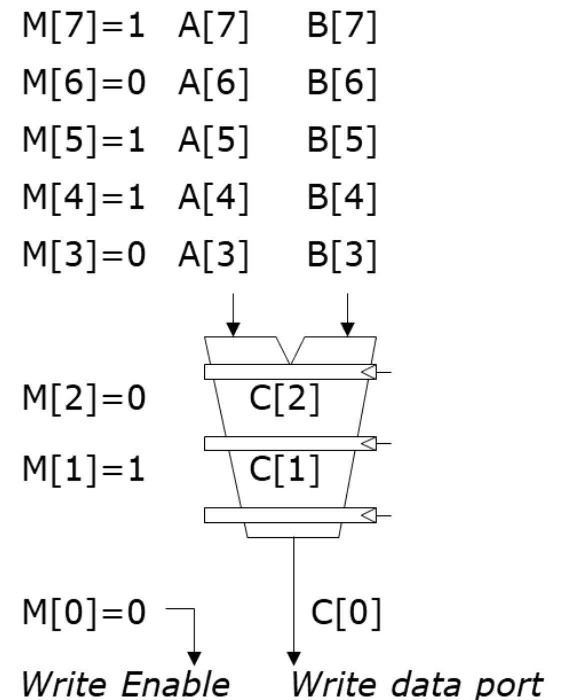
Vector Conditional Instructions

- When conditional codes appear in vectorized loops

```
for (i=0; i<N; i++)
    if (A[i]>0) then
        A[i] = B[i];
```

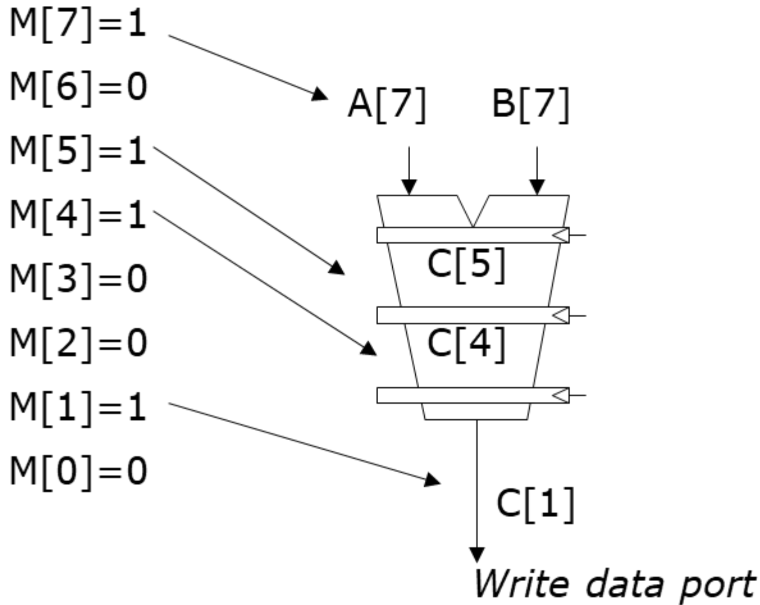
- Using vector mask (flage registers)
  - (Mask bit == 1) ? (Execute the Instruction) : NOOP

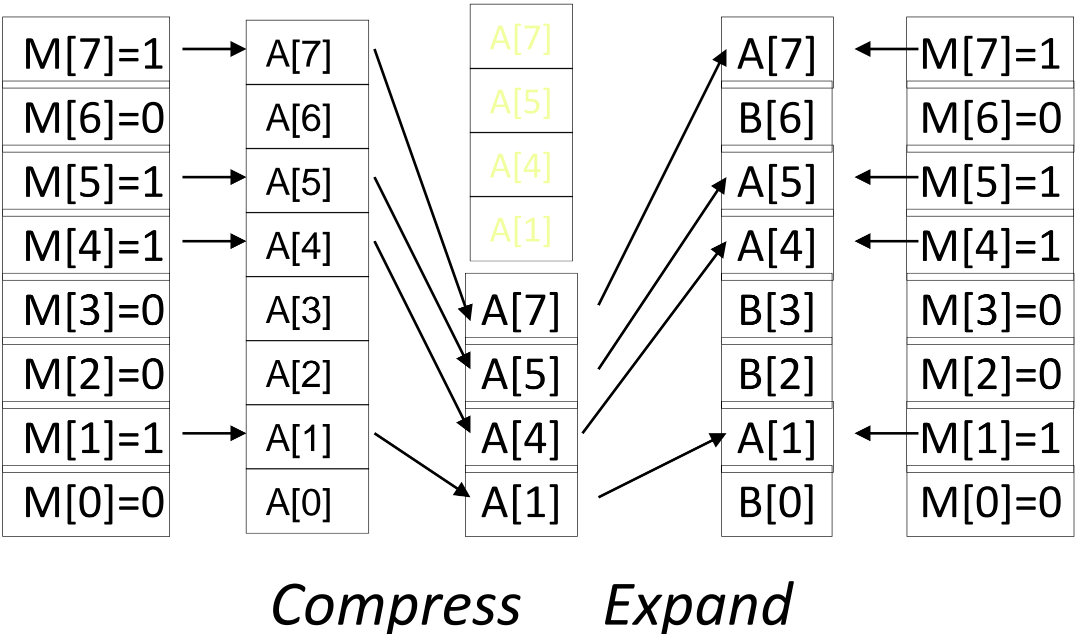
```
• cvm               # Turn on all elements
• vld vA, xA        # Load entire A vector
• vgt vA, f0        # Set bits in mask register where A>0
• vld vA, xB        # Load B vector into A under mask
• vst vA, xA        # Store A back to memory under mask
```



More on mask vector

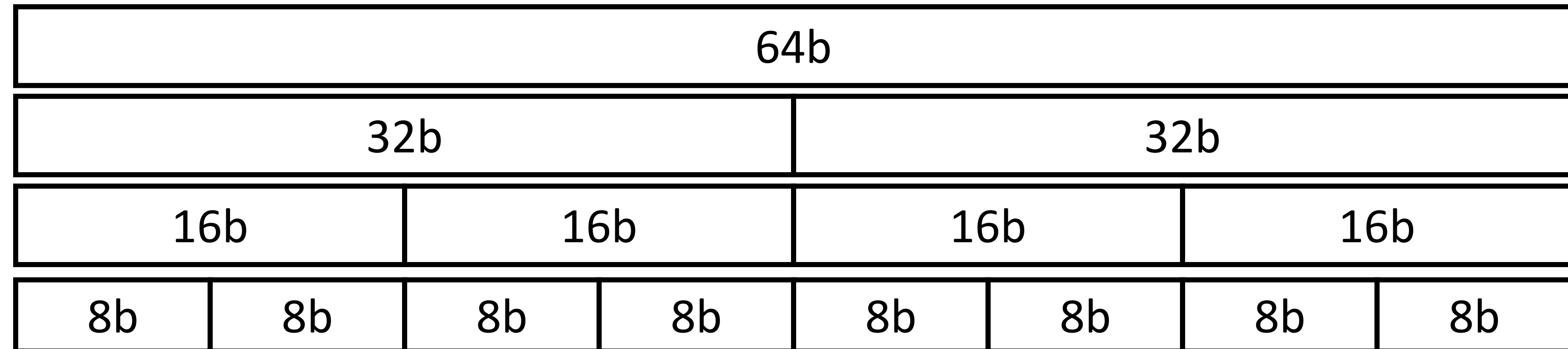
- The previous one has low utilization
- Need compress and expand operations



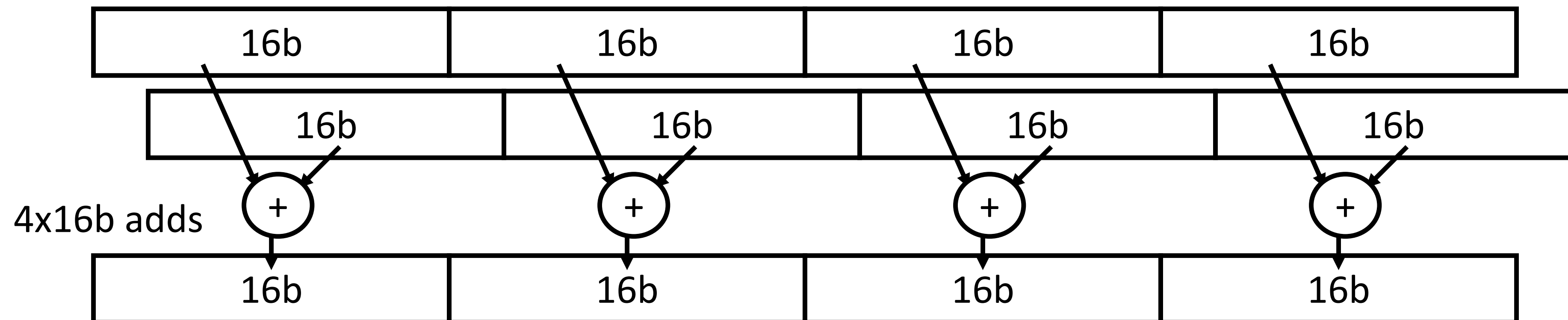


SIMD Extensions

- For those processor with fixed bit-width vectors are done by splitting words



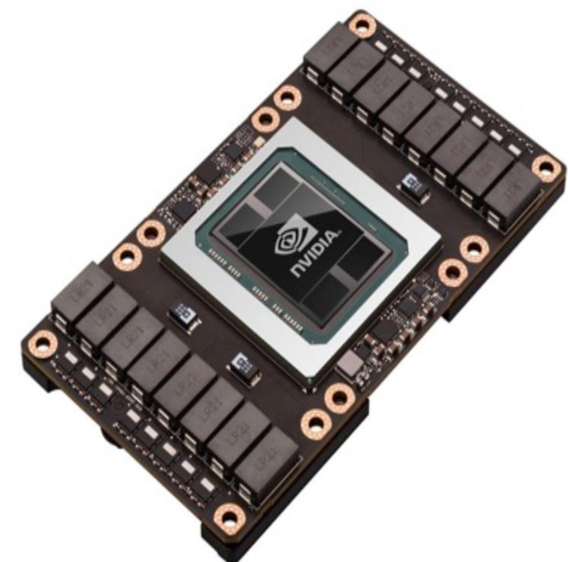
- ARM's SIMD extension called NEON



GPU Resolution

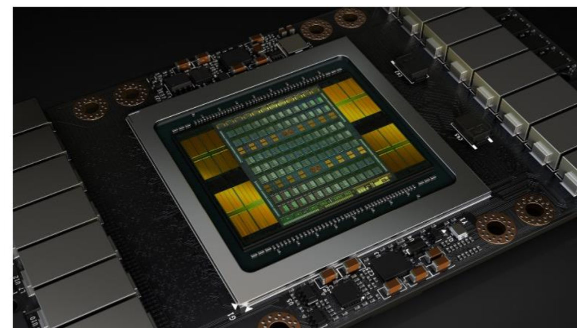
GPUs are Targeting Deep Learning

Nvidia PASCAL GP100 (2016)



- 10/20 TFLOPS FP32/FP16
- 16GB HBM – 750 GB/s
- 300W TDP
- 33/67 GFLOPS/W (FP32/FP16)
- 16nm process
- 160GB/s NV Link

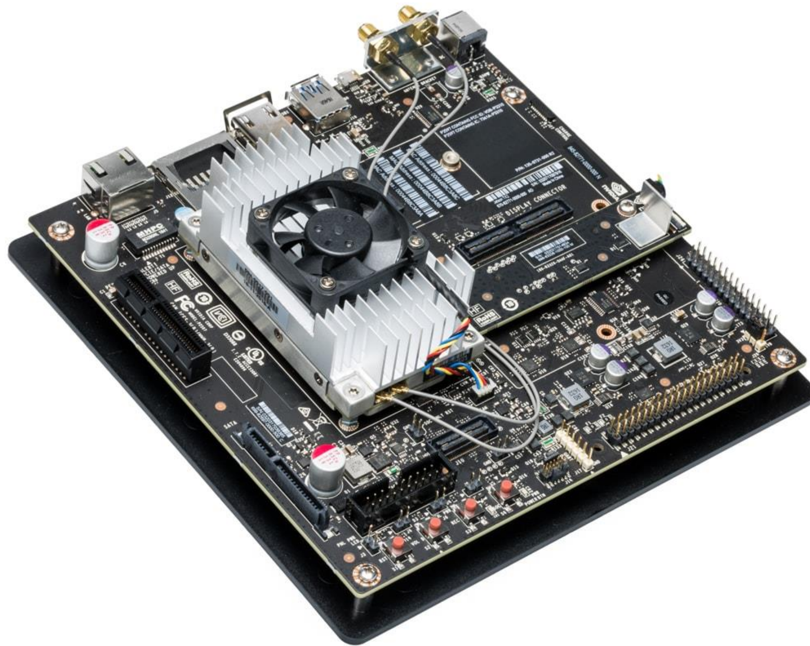
Nvidia VOLTA GV100 (2017)



- 15 TFLOPS FP32
- 16GB HBM2 – 900 GB/s
- 300W TDP
- 50 GFLOPS/W (FP32)
- 12nm process
- 300GB/s NV Link2
- Tensor Core….

NVIDIA Jetson Board

- Jetson TX2
    - ARM + GPU



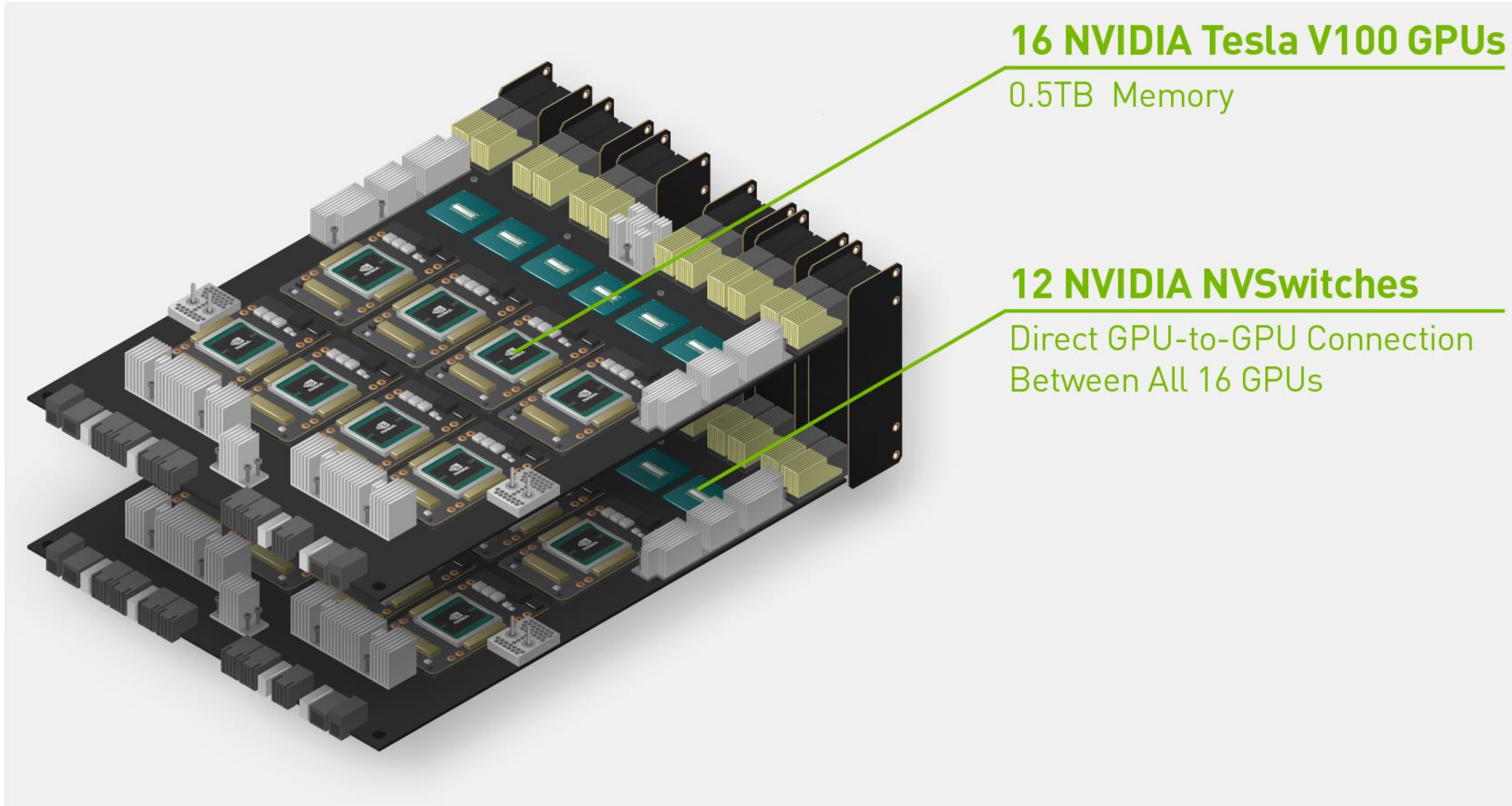
MCPU power rail
BCPU power rail
Denver 2 CPU cluster
Cortex-A57 CPU cluster
GIC
CPU switch fabric (coherent)
CoreSight
BPMP, APE, SPE, SCE
SysRAM TZRAM iRAM
Cortex-R5 BPMP
Boot, power mgmt.
DMA
APB bridges
Clock and reset
Timers, mailboxes, semaphore, config. reg., sys. config.
GPIO controller
AXI
APB bridge
CAN, PMC, I2C, SPI, DMIC, GPIO
UART, I2C, STV, SPI, fuses, thrmsnsr, PWRM
System fabric
APB bridges
XUSB, eAVB, UFS, SD/MMC, SATA
GPU power rail
General-purpose DMA
PCIE
Cortex-A9 APE audio processor
Audio IO
GPU Pascal
128 CUDA cores
128 CUDA cores
Compress Decompress
Host command buffer and synchronization
CSI camera input
Camera ISP
Video encode
Video decode
JPEG
Security TSECs and SE
2D graphics
Decompress
Display 3X heads
Decompress
Memory controller w/ECC
System MMU
32/64/128b LPDDR4
Memory fabric and arbitration

Deep Learning Systems Using GPUs

- 170 TFLOPS
- 8× Tesla P100, Dual Xeon
- NVLink Hybrid Cube Mesh
- Optimized DL Software
- 7 TB SSD Cache
- Dual 10GbE, Quad IB 100Gb
- 3RU – 3200W

Nvidia DGX-1 (2016)

Nvidia HGX -2



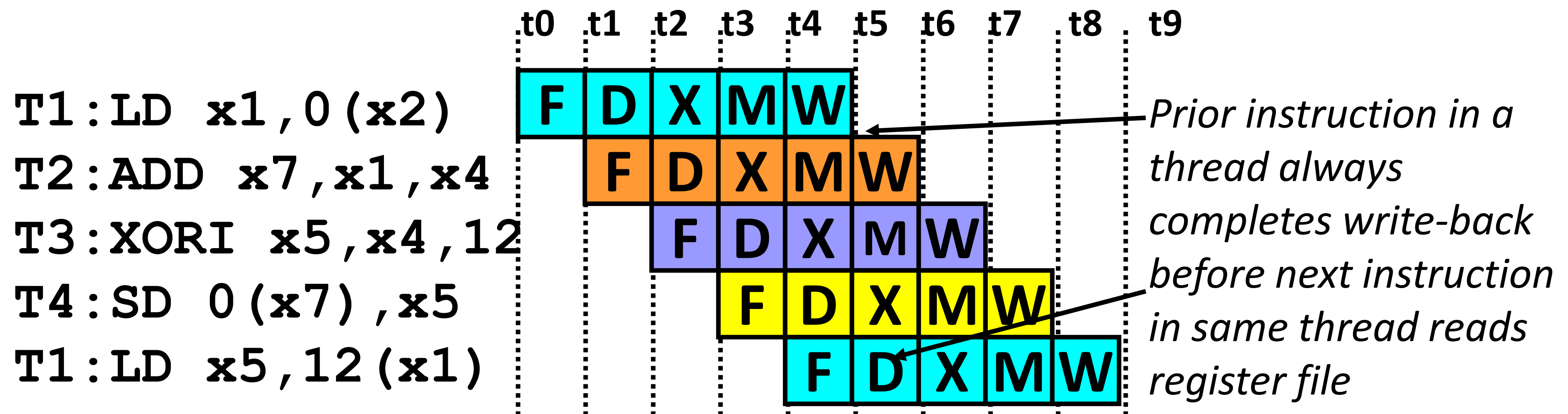
Type of Parallelism

- Instruction-Level Parallelism (ILP)
  - Execute independent instructions from one instruction stream in parallel (pipelining, superscalar, VLIW)
- Thread-Level Parallelism (TLP)
  - Execute independent instruction streams in parallel (multithreading, multiple cores)
- Data-Level Parallelism (DLP)
  - Execute multiple operations of the same type in parallel (vector/SIMD execution)

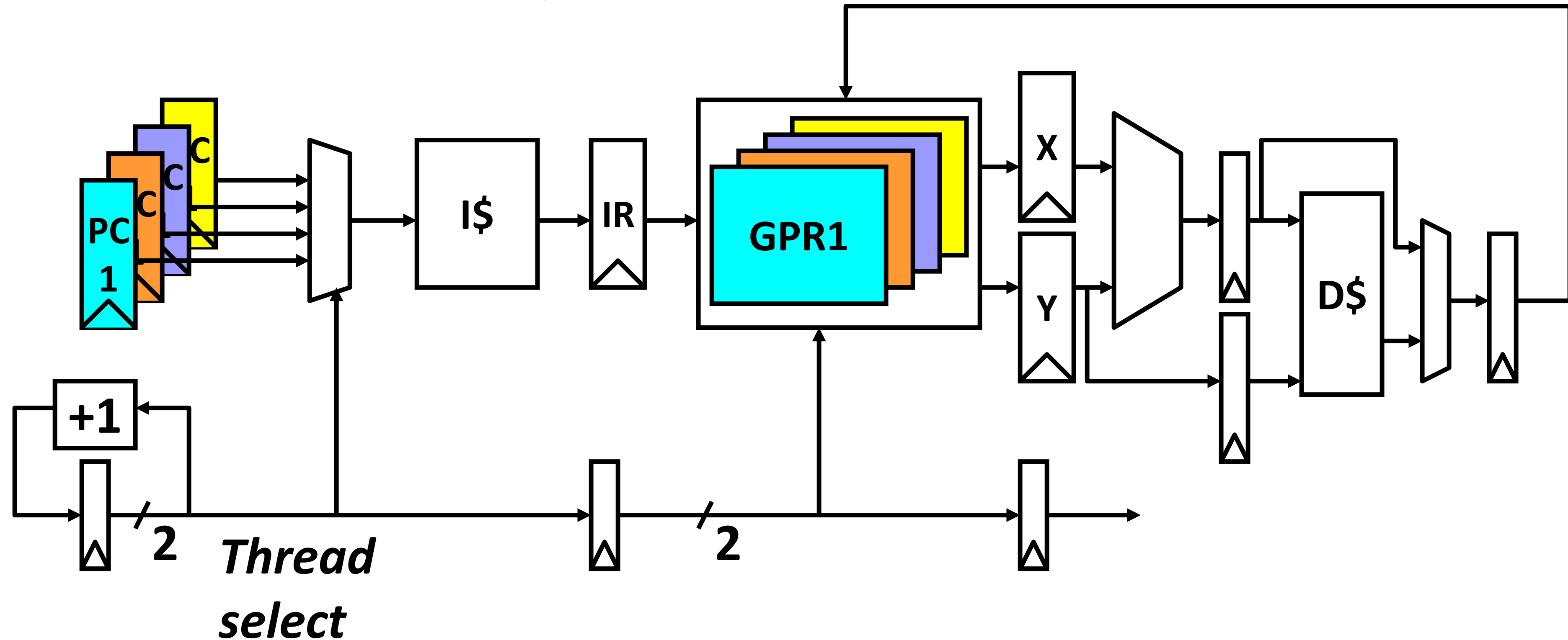
Multi-Threading

- There is no guarantee of no dependencies in a pipeline
- But we can interleaved executions of dependent instructions from different program

*Interleave 4 threads, T1-T4, on **non-bypassed** 5-stage pipe*



Multithreaded Pipeline Architecture (MTA)



- Have to carry thread select down pipeline to ensure correct state bits read/written at each pipe stage
- Appears to software (including OS) as multiple, albeit slower, CPUs

Graphics Processing Unites

- Original GPUs were dedicated fixed-function devices for generating 3D graphics (mid-late 1990s) including high-performance floating-point units
  - Provide workstation-like graphics for PCs
  - User could configure graphics pipeline, but not really program it
- Over time, more programmability added (2001-2005)
  - E.g., New language Cg for writing small programs run on each vertex or each pixel, also Windows DirectX variants
  - Massively parallel (millions of vertices or pixels per frame) but very constrained programming model

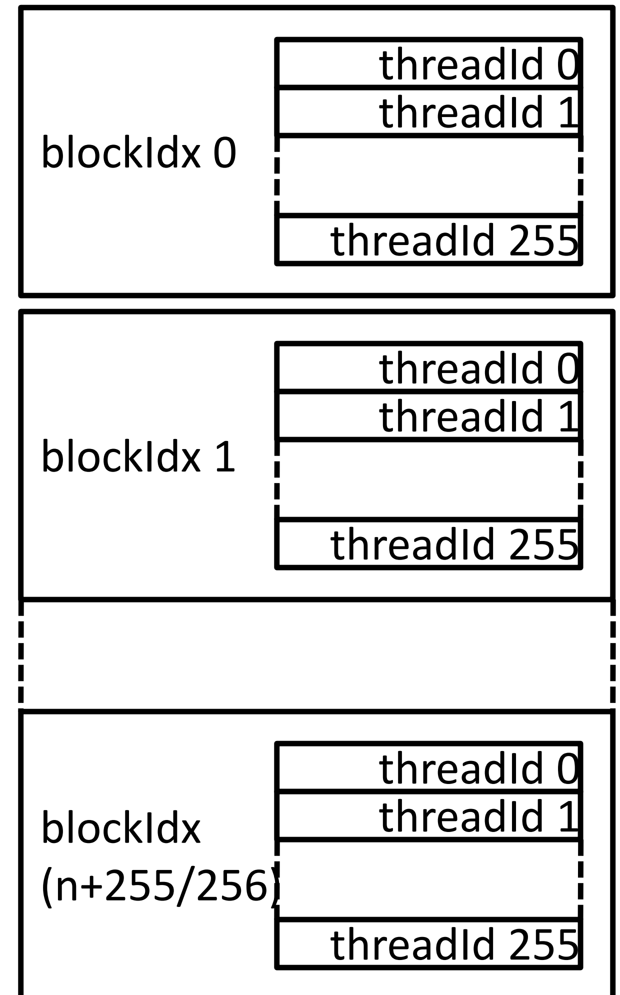
General Purpose GPUs (GP-GPUs)

- In 2006, Nvidia introduced GeForce 8800 GPU supporting a new programming language: CUDA
  - “Compute Unified Device Architecture”
  - Subsequently, broader industry pushing for OpenCL, a vendor-neutral version of same ideas.
- Idea: Take advantage of GPU computational performance and memory bandwidth to accelerate some kernels for general-purpose computing
- Attached processor model: Host CPU issues data-parallel kernels to GP-GPU for execution

CUDA Program

- Using SIMD + Multi-threading

blockDim = 256



Standard C Code

```c
void saxpy(int n, float a,
           float *x, float *y)
{
  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}

int N = 1<<20;

// Perform SAXPY on 1M elements
saxpy(N, 2.0, x, y);
```

C with CUDA extensions

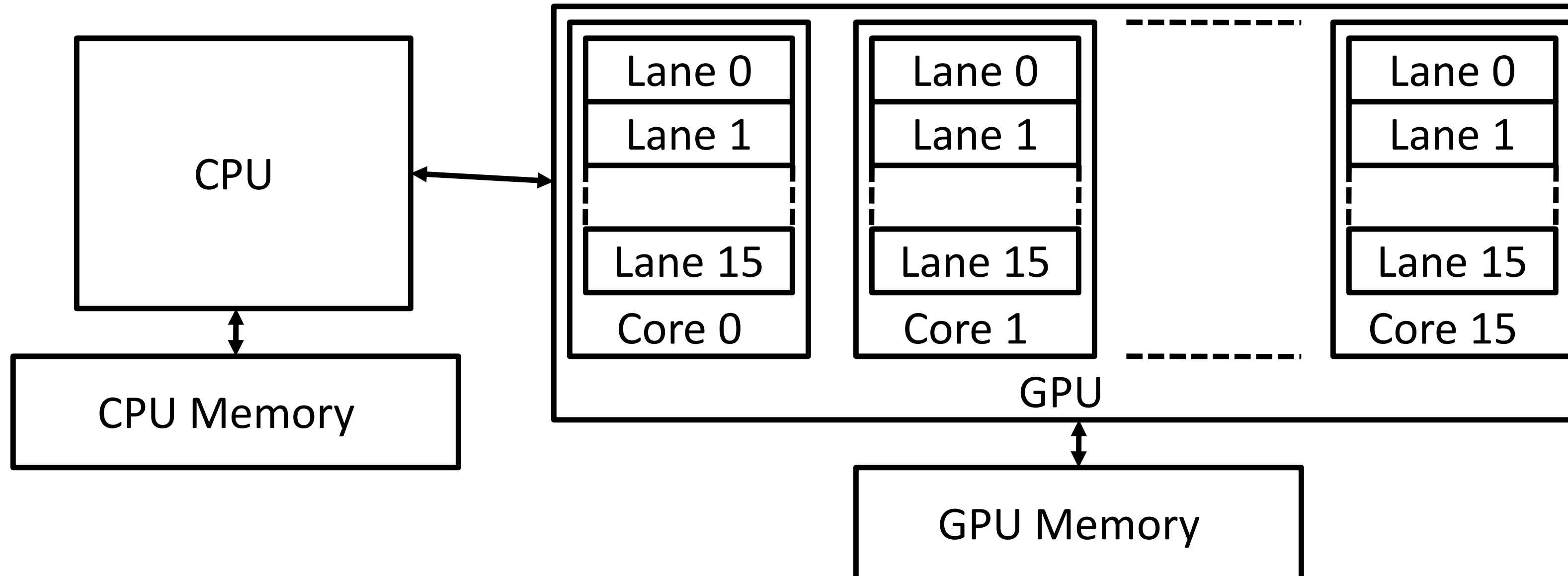
```c
__global__
void saxpy(int n, float a,
           float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

int N = 1<<20;
cudaMemcpy(x, d_x, N, cudaMemcpyHostToDevice);
cudaMemcpy(y, d_y, N, cudaMemcpyHostToDevice);

// Perform SAXPY on 1M elements
saxpy<<<4096,256>>>(N, 2.0, x, y);

cudaMemcpy(d_y, y, N, cudaMemcpyDeviceToHost);
```

GPU Hardware Execution Model

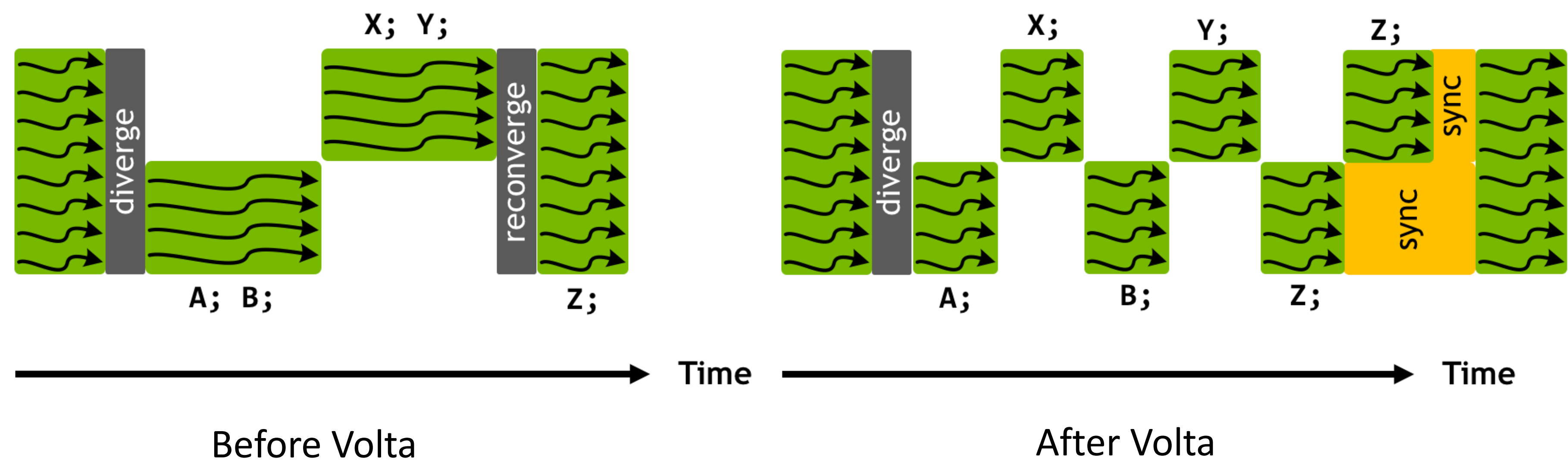


- GPU is built from multiple parallel cores, each core contains a multithreaded SIMD processor with multiple lanes but with no scalar processor
- CPU sends whole “grid” over to GPU, which distributes thread blocks among cores (each thread block executes on one core)

“Single Instruction, Multiple Thread” (SIMT)

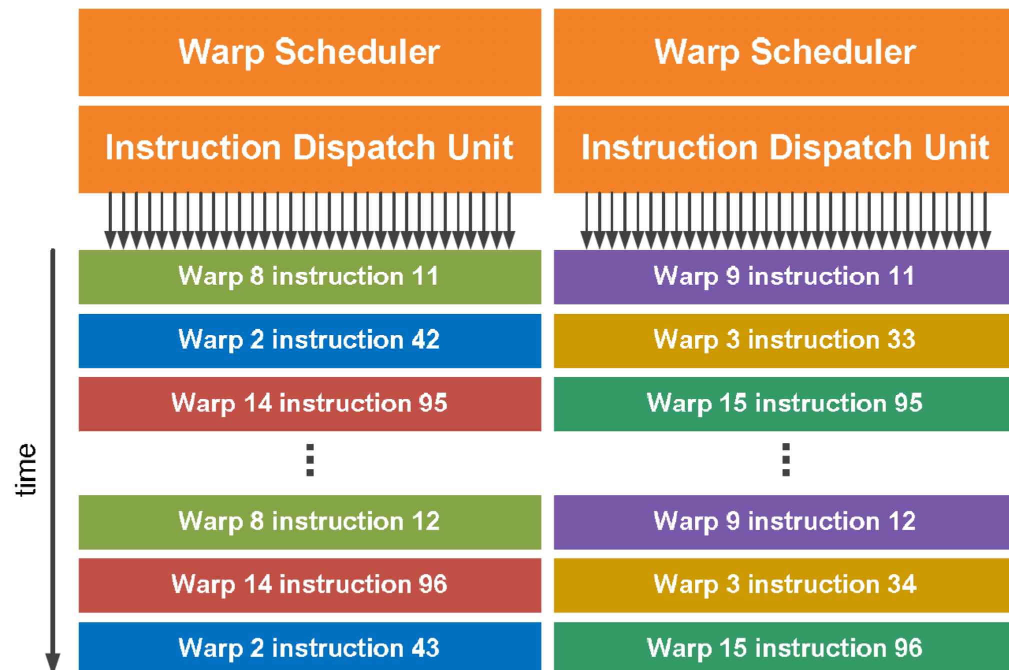
- GPUs use a SIMT model, where individual scalar instruction streams for each CUDA thread are grouped together for SIMD execution on hardware (NVIDIA groups 32 CUDA threads into a *warp*)
- What happens to branch instructions?
  - Do the Taken/NotTaken, and mask the result s (diverge and reconverge)

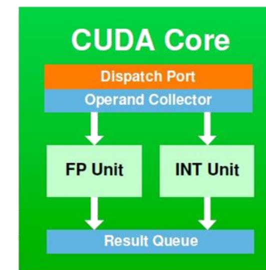
```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```

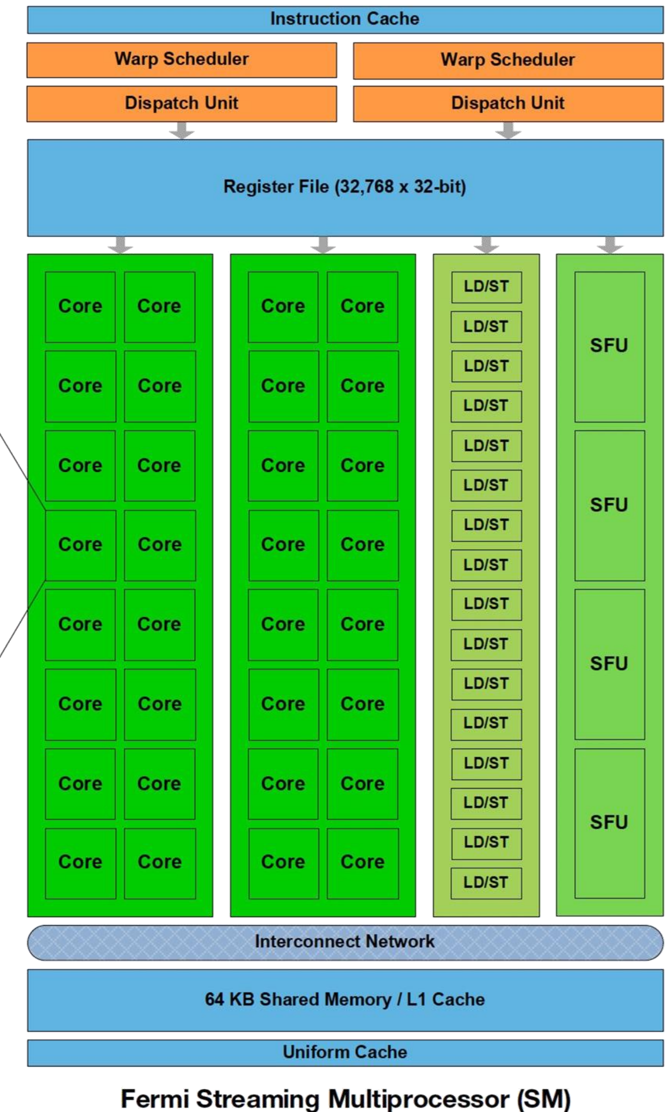


Nvidia Terminology

- Streaming multiprocessor (SM), Cuda Cores
- Warp Scheduler / Dispatch Unit (Warp = thread in nvidia GPUs)







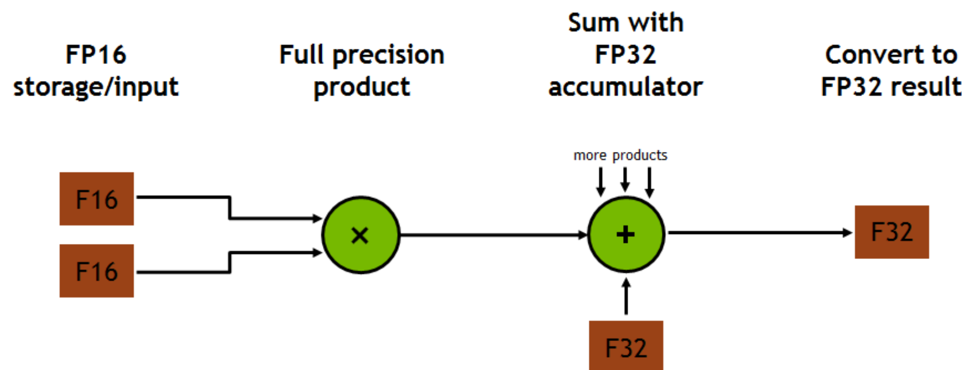
Fermi Streaming Multiprocessor (SM)

Deep Learning GPUs

- Volta Architecture GV100 → 120TFLOPS
- 1 FP32 = 2 FP 16
- 4x4 Tensor Core

$$\mathbf{D} = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

FP16 or FP32 | FP16 | FP16 | FP16 or FP32



Questions: Why 4x4, not 3x3?

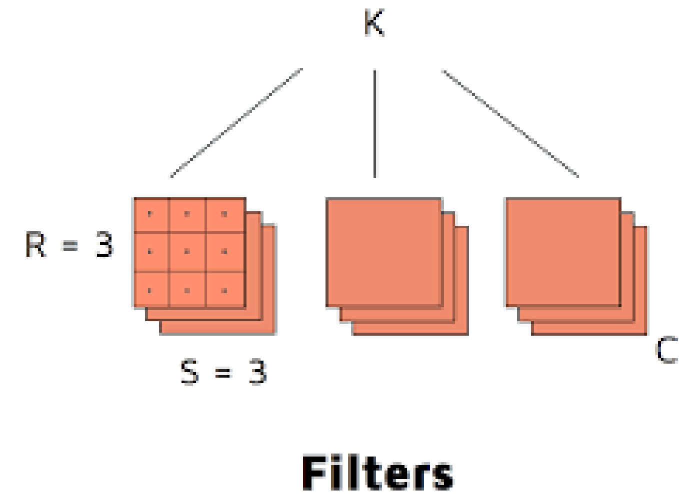
- Most CNNs use 3x3 kernels.
- Some old math: Gauss' Multiplication

$$(a+bi)(c+di) = (ac-bd) + (bc+ad)i.$$

4 multiplications + 3 additions

$$k_1 = c \cdot (a+b)$$
$$k_2 = a \cdot (d-c)$$
$$k_3 = b \cdot (c+d)$$
$$\text{Real part} = k_1 - k_3$$
$$\text{Imaginary part} = k_1 + k_2.$$

3 multiplications + 5 additions



Winograd 1D – F(2,3)

- Target: reduce multiplication by similar transform
- For convolutions, some thing more than matrix multiply
- Notation: F(size of output filter size)

input, filter

$$F(2,3) = \begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix}$$

6 multiplications + 4 additions

$$m_1 = (d_0 - d_2)g_0 \qquad m_2 = (d_1 + d_2)\frac{g_0 + g_1 + g_2}{2}$$

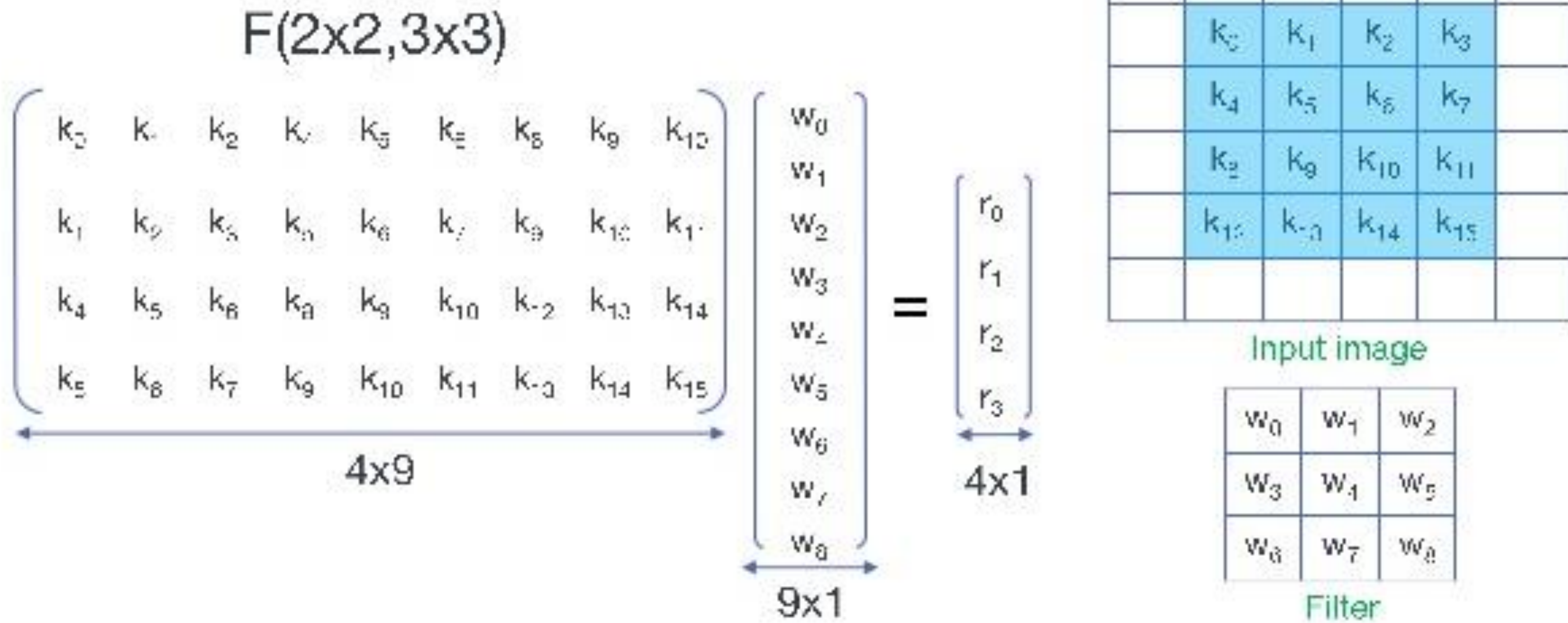
$$m_4 = (d_1 - d_3)g_2 \qquad m_3 = (d_2 - d_1)\frac{g_0 - g_1 + g_2}{2}$$

4 multiplications + 12 additions + 2 shifts

4 multiplications + 8 additions (for constant weights)

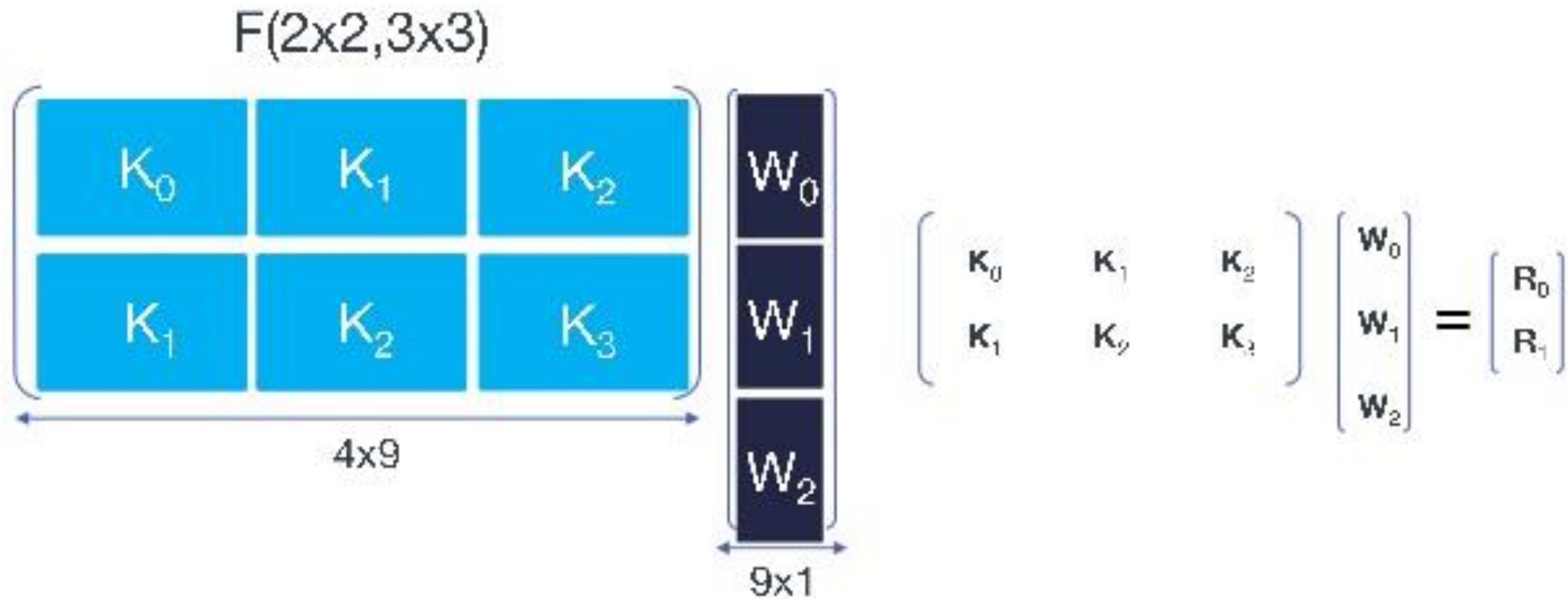
Winograd 2D

- How about F(2x2, 3x3)



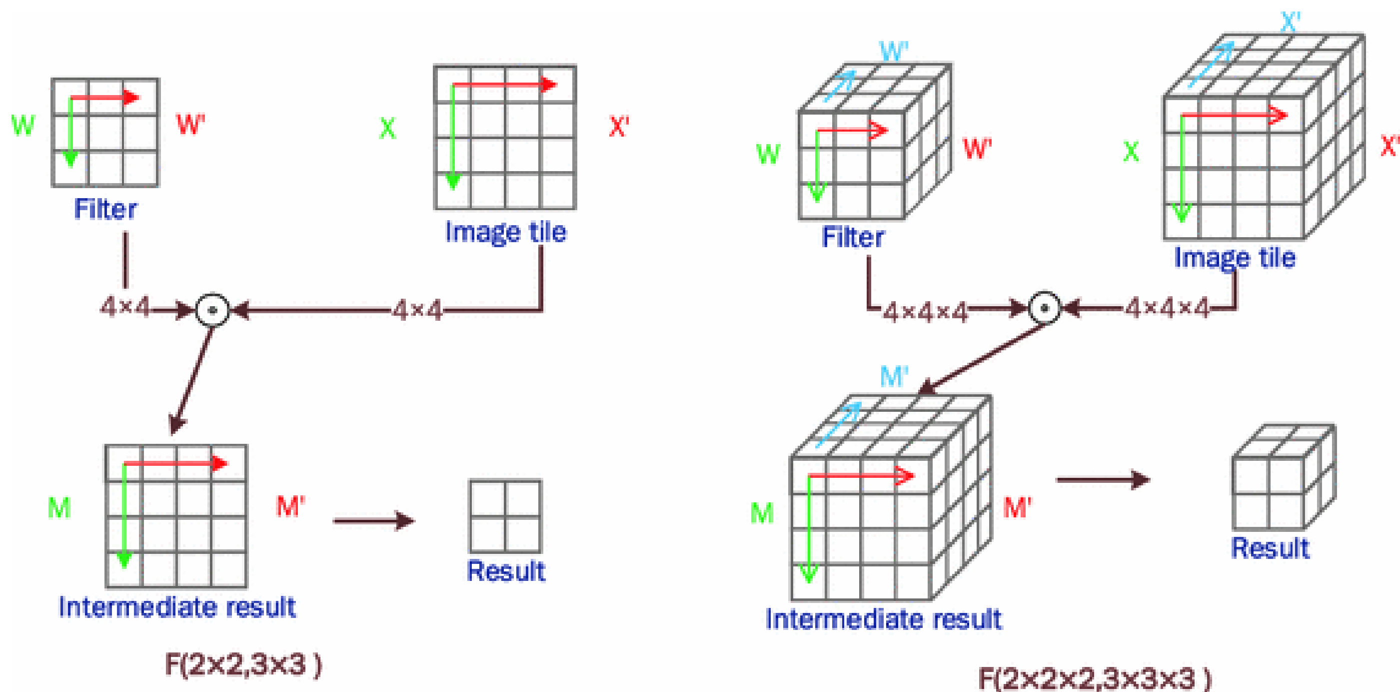
Winograd 2D: Nesting 1D

- Original 36 multiplications
- Winograd 2D: 16 multiplications → 4x4 Tensor Core



GPU Acceleration by Winograd

- CuDNN supports winograd after 2016
- More than 2D?



N	cuDNN msec	cuDNN TFLOPS	F(2x2,3x3) msec	F(2x2,3x3) TFLOPS	Speedup
1	12.52	3.12	5.55	7.03	2.26X
2	20.36	3.83	9.89	7.89	2.06X
4	104.70	1.49	17.72	8.81	5.91X
8	241.21	1.29	33.11	9.43	7.28X
16	203.09	3.07	65.79	9.49	3.09X
32	237.05	5.27	132.36	9.43	1.79X
64	394.05	6.34	266.48	9.37	1.48X

Table 5. cuDNN versus $F(2 \times 2, 3 \times 3)$ performance on VGG Network E with fp32 data. Throughput is measured in Effective TFLOPS, the ratio of direct algorithm GFLOPs to run time.

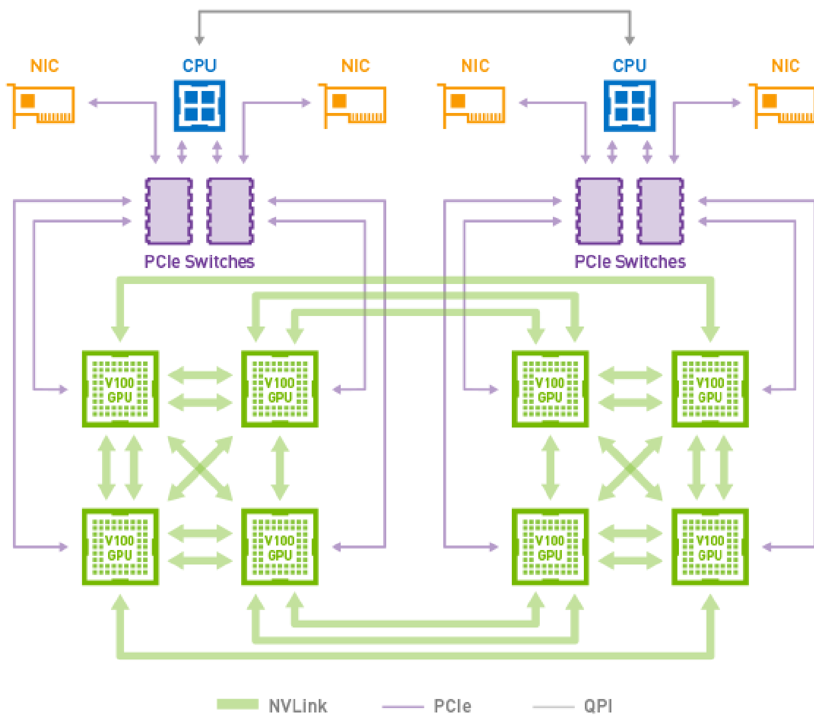
N	cuDNN msec	cuDNN TFLOPS	F(2x2,3x3) msec	F(2x2,3x3) TFLOPS	Speedup
1	14.58	2.68	5.53	7.06	2.64X
2	20.94	3.73	9.83	7.94	2.13X
4	104.19	1.50	17.50	8.92	5.95X
8	241.87	1.29	32.61	9.57	7.42X
16	204.01	3.06	62.93	9.92	3.24X
32	236.13	5.29	123.12	10.14	1.92X
64	395.93	6.31	242.98	10.28	1.63X

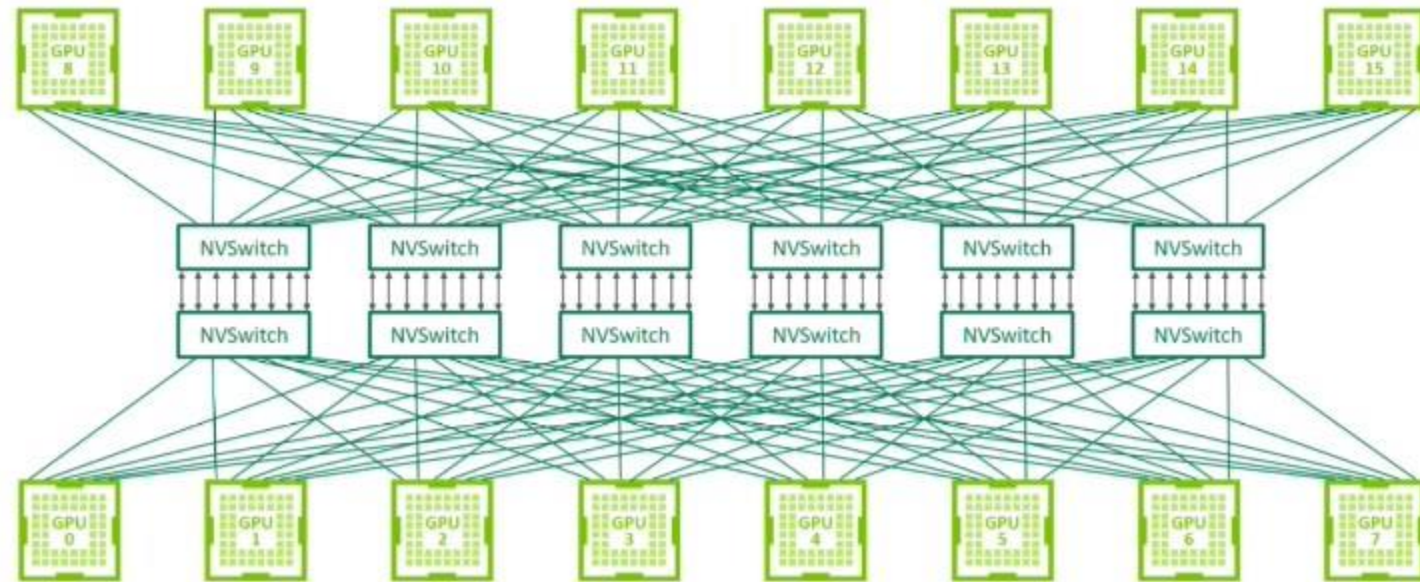
Table 6. cuDNN versus $F(2 \times 2, 3 \times 3)$ performance on VGG Network E with fp16 data.

https://blog.csdn.net/qq_32998593

Interconnection Between GPUs

- Previously NV Link → NV Swtich



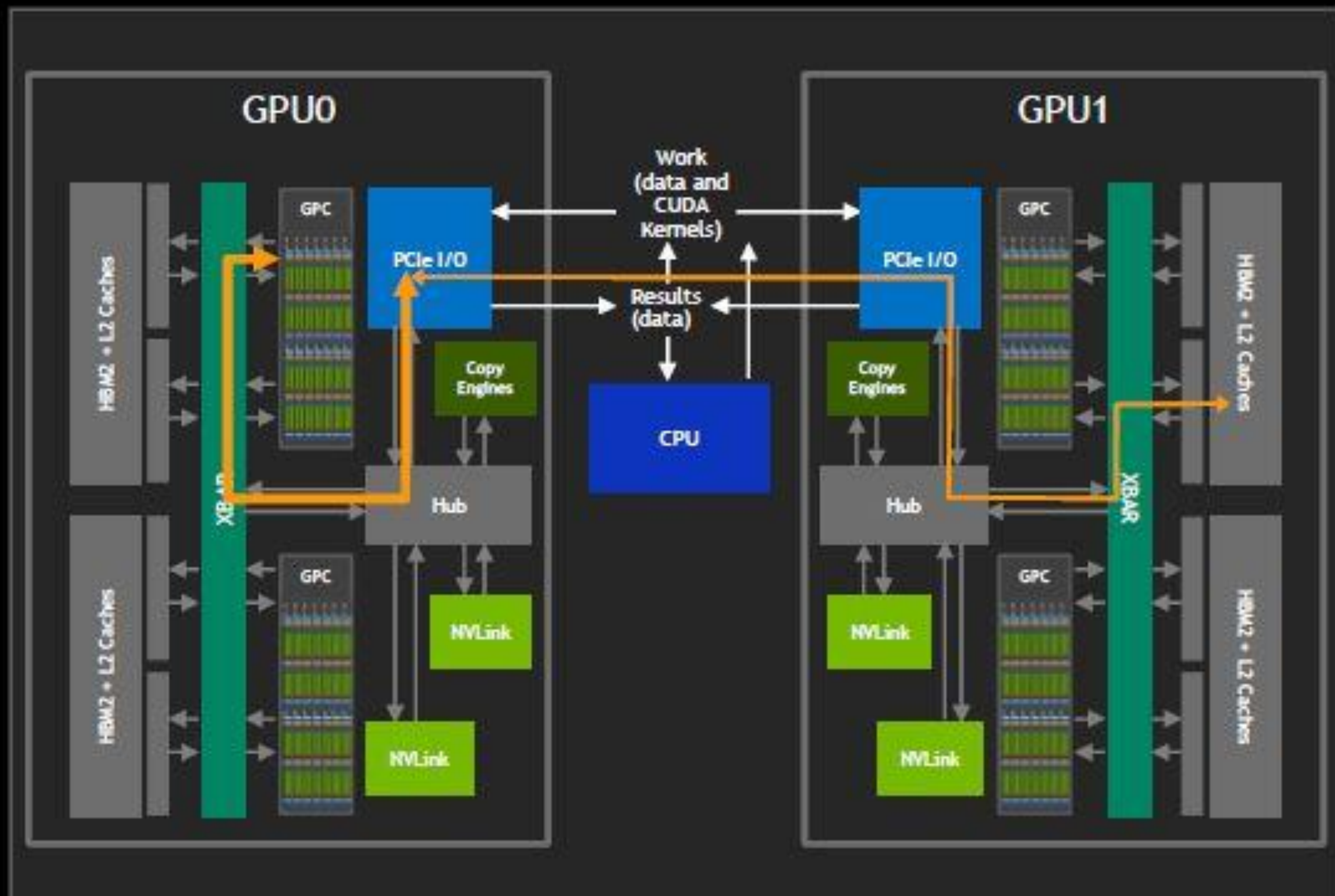


TWO GPUS WITH PCIE

Access to HBM2 of other GPU is at PCIe BW (32 GBps (bidirectional))

Interactions with CPU compete with GPU-to-GPU

PCIe is the “Wild West” (lots of performance bandits)



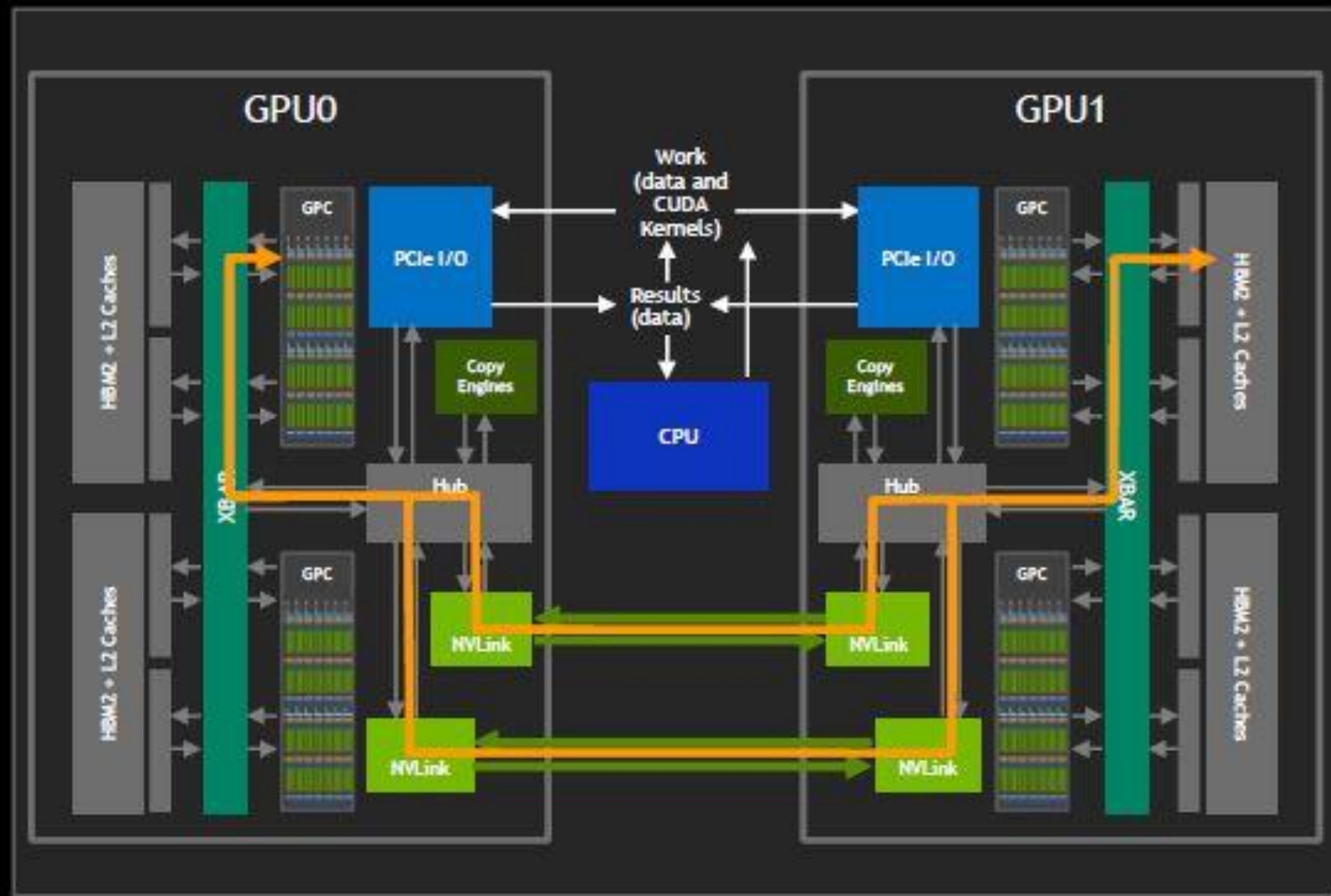
TWO GPUS WITH NVLINK

All GPCs can access all HBM2 memories

Access to HBM2 of other GPU is at multi-NVLink bandwidth (300 GBps bidirectional in V100 GPUs)

NVLinks are effectively a "bridge" between XBARs

No collisions with PCIe traffic



NVSWITCH BLOCK DIAGRAM

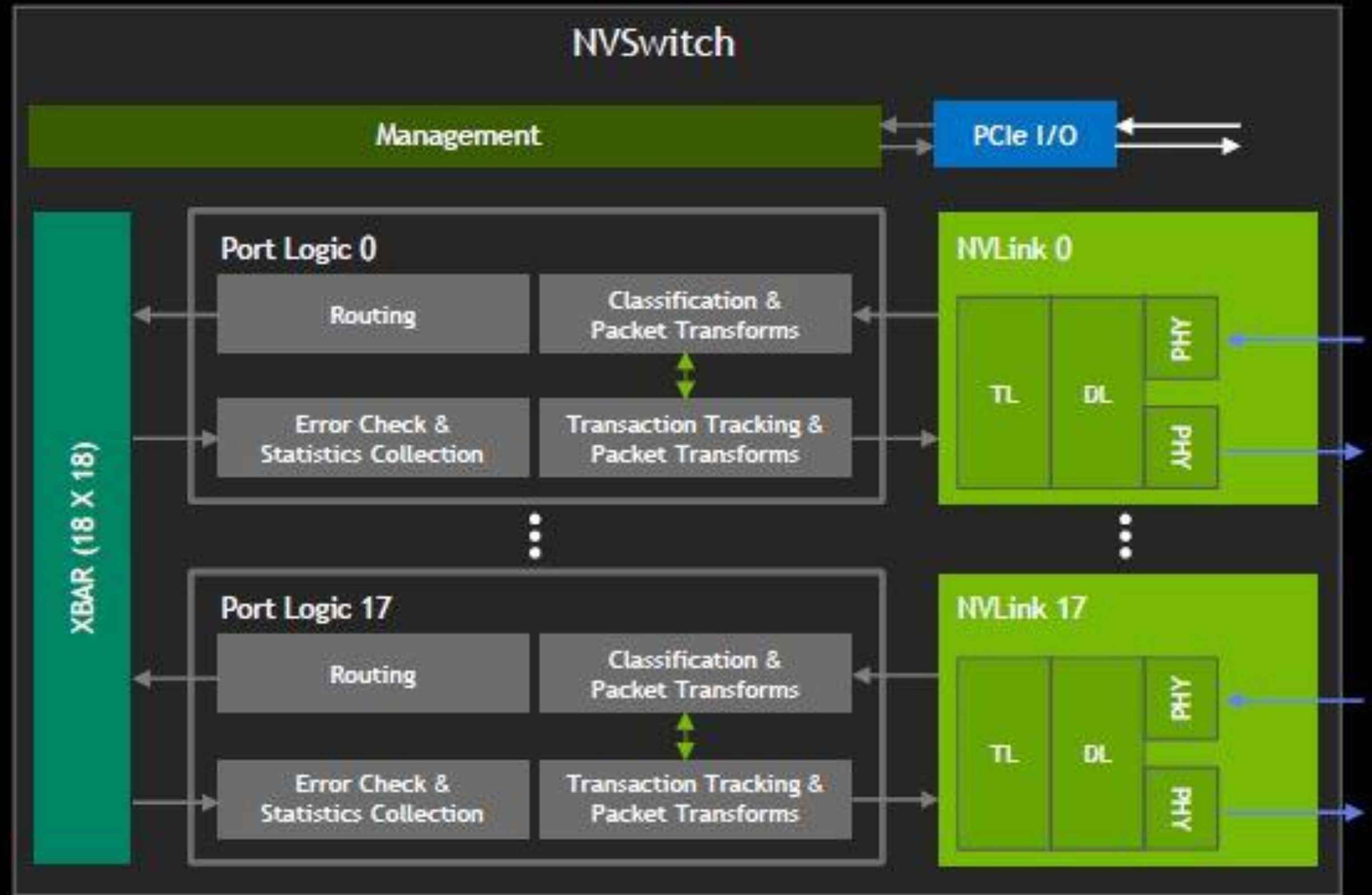
GPU-XBAR-bridging device; not a general networking device

Packet-Transforms make traffic to/from multiple GPUs look like they are to/from single GPU

XBAR is non-blocking

SRAM-based buffering

NVLink IP blocks and XBAR design/verification infrastructure reused from V100

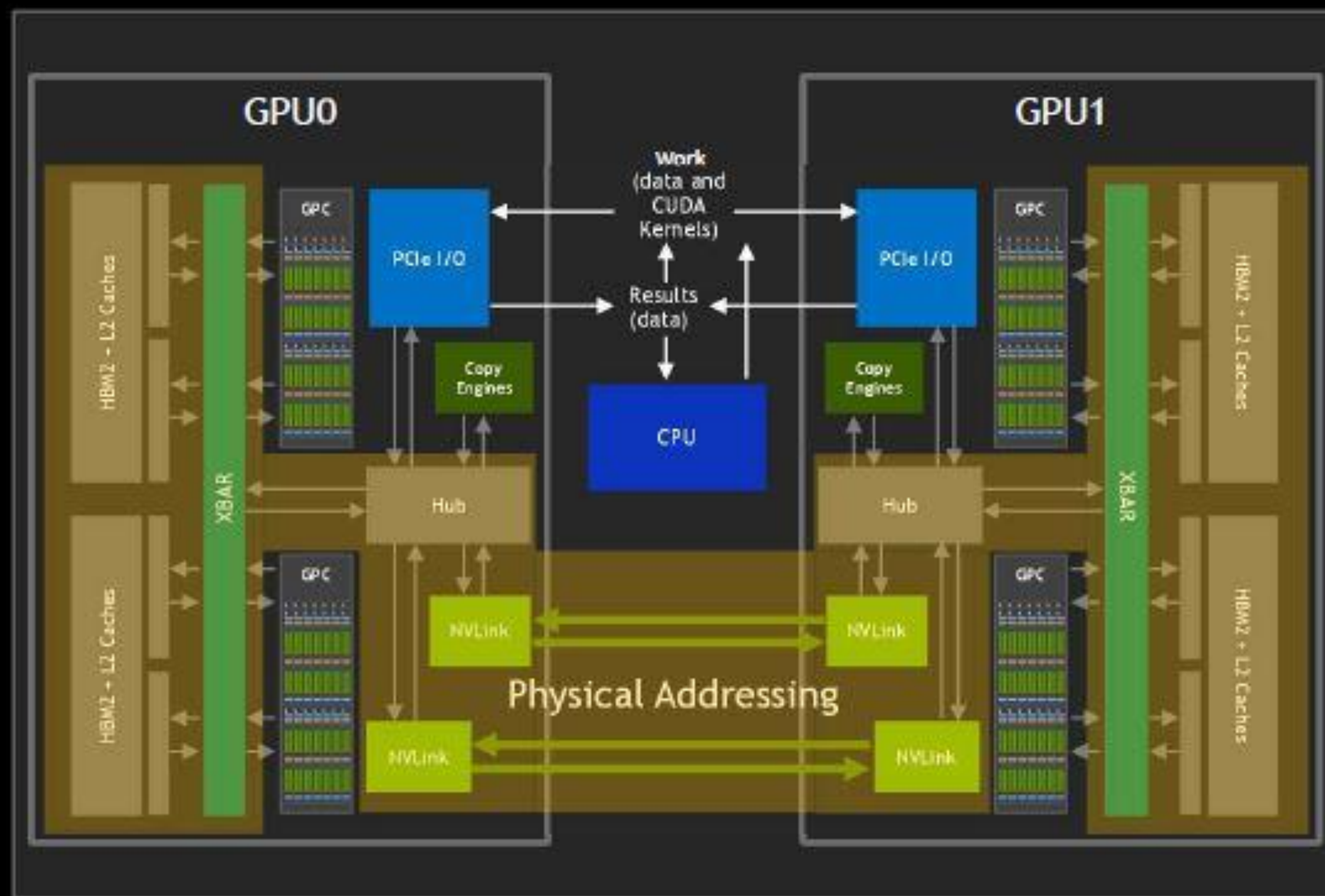


NVLINK: PHYSICAL SHARED MEMORY

Virtual-to physical address translation is done in GPCs

NVLink packets carry physical addresses

NVSwitch and DGX-2 follow same model



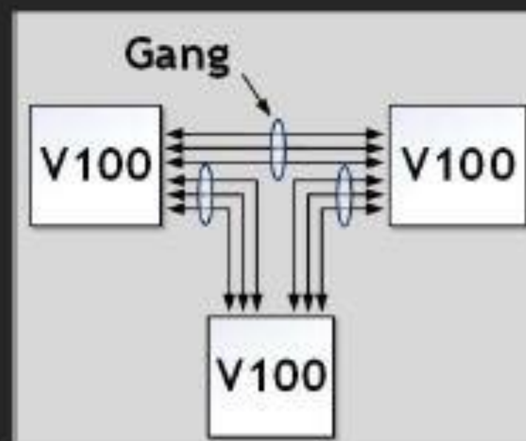
SIMPLE SWITCH SYSTEM

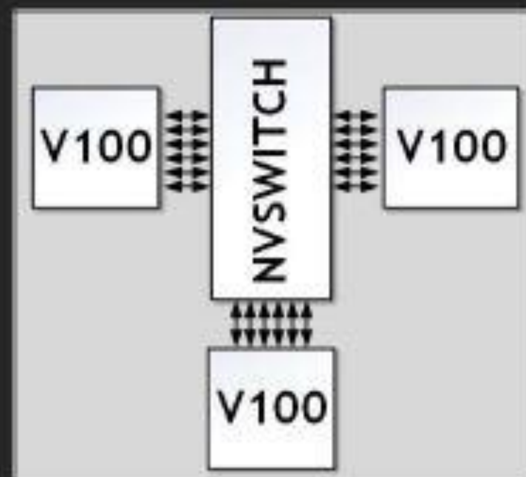
No NVSwitch

- Connect GPU ↔ directly
- Aggregate NVLinks into gangs for higher bandwidth
- Interleaved over the links to prevent camping
- Max bandwidth between two GPUs limited to the bandwidth of the gang

With NVSwitch

- Interleave traffic across all the links and to support full bandwidth between any pair of GPUs
- Traffic to a single GPU is non-blocking, so long as aggregate bandwidth of six NVLinks is not exceeded





SWITCH CONNECTED SYSTEMS

Add NVSwitches in parallel to support more GPUs

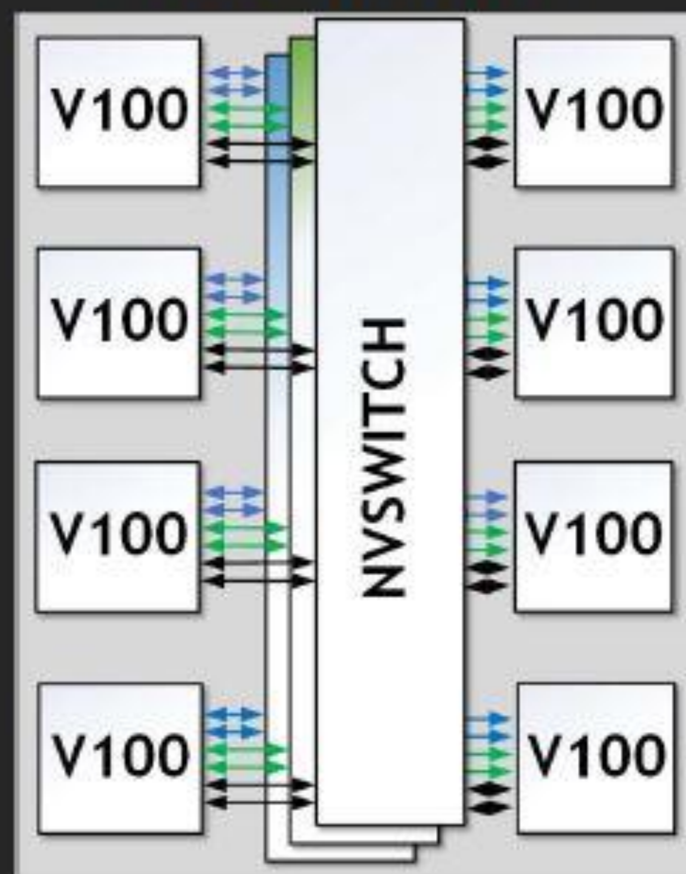
Eight GPU closed system can be built with three NVSwitches with two NVLinks from each GPU to each switch

Gang width is still six — traffic is interleaved across all the GPU links

GPUs can now communicate pairwise using the full 300 GBps bidirectional between any pair

NVSwitch XBAR provides unique paths from and source to any destination — non-blocking, non-interfering

8 GPUs, 3 NVSwitches



Conclusions