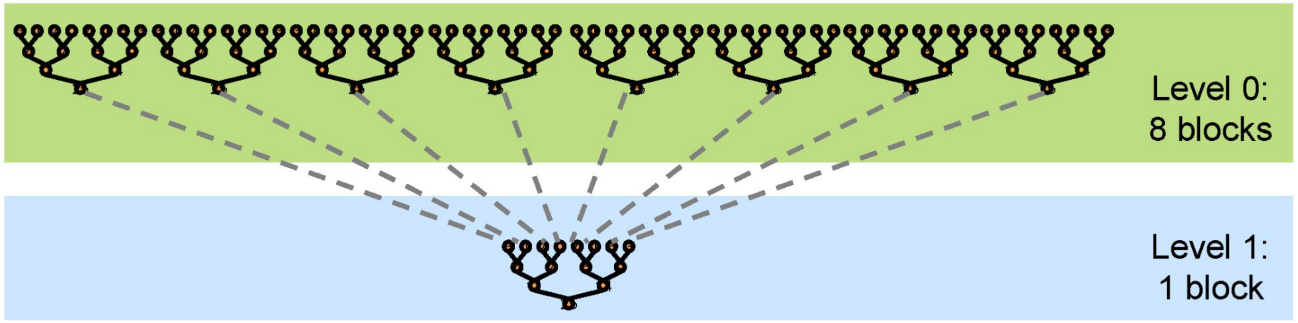
CUDA

Scott Cheng

Recall: Reduction

Given a vector [v_0, v_1, ... v_n] it returns a scalar v_0 + v_1 + ... + v_n



Performance for 4M element reduction

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x
Kernel 5: unroll last warp	0.536 ms	31.289 GB/s	1.8x	15.01x
Kernel 6: completely unrolled	0.381 ms	43.996 GB/s	1.41x	21.16x
Kernel 7: multiple elements per thread	0.268 ms	62.671 GB/s	1.42x	30.04x

Kernel 7 on 32M elements: 73 GB/s!

Warp Shuffle Functions

```
T __shfl_sync(unsigned mask, T var, int srcLane, int width=warpSize);
```

Direct copy from indexed lane

```
T __shfl_up_sync(unsigned mask, T var, unsigned int delta, int width=warpSize);
```

Copy from a lane with lower ID relative to caller

```
T __shfl_down_sync(unsigned mask, T var, unsigned int delta, int width=warpSize);
```

Copy from a lane with higher ID relative to caller

```
T __shfl_xor_sync(unsigned mask, T var, int laneMask, int width=warpSize);
```

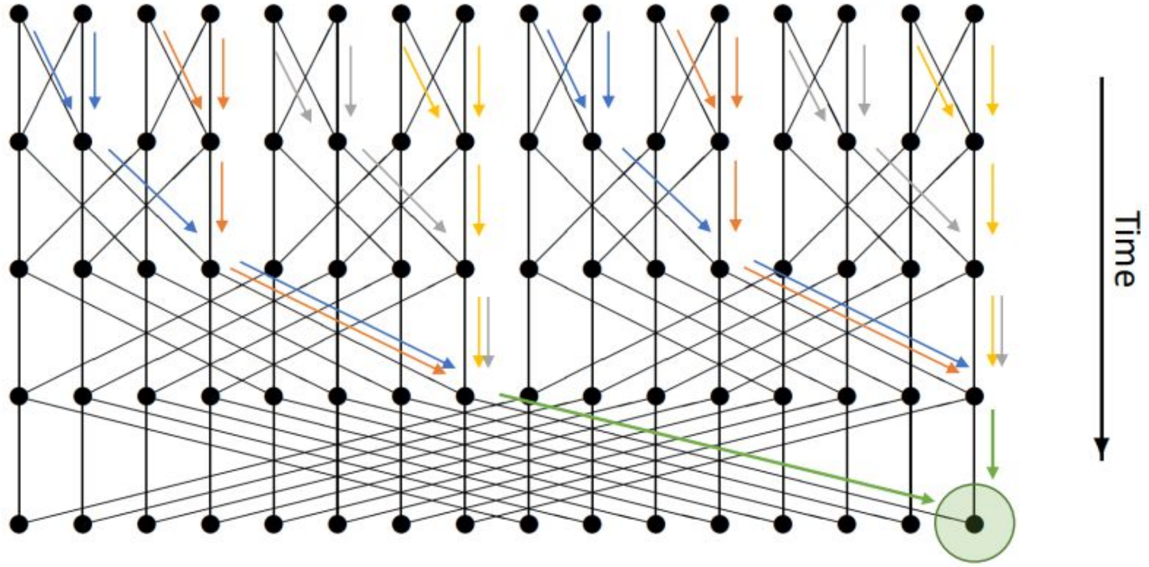
Copy from a lane based on bitwise XOR of own lane ID

Example: Broadcast of a single value across a warp

```
#include <stdio.h>

__global__ void bcast(int arg) {
    int laneId = threadIdx.x & 0x1f;
    int value; // Note unused variable for all threads except lane 0
    if (laneId == 0)
        value = func(arg);
    // Synchronize all threads in warp, and get "value" from lane 0
    value = __shfl_sync(0xffffffff, value, 0);
    // every thread in the warp get the same "value" now
}
```

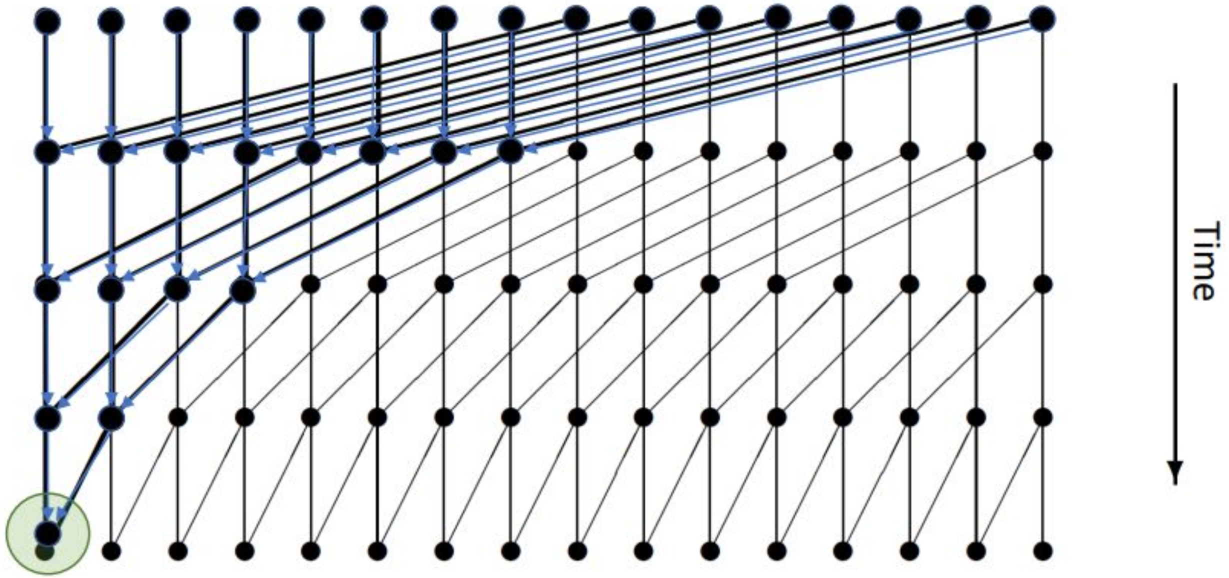
Reduction: __shfl_xor_sync

```
for (int i=1; i<32; i*=2)
    value += __shfl_xor_sync(-1, value, i);
```



Reduction: __shfl_down_sync

```
for (int i=16; i>0; i=i/2)
    value += __shfl_down_sync(-1, value, i);
```



Warp Reduce Functions

Supported by devices of compute capability 8.x or higher.

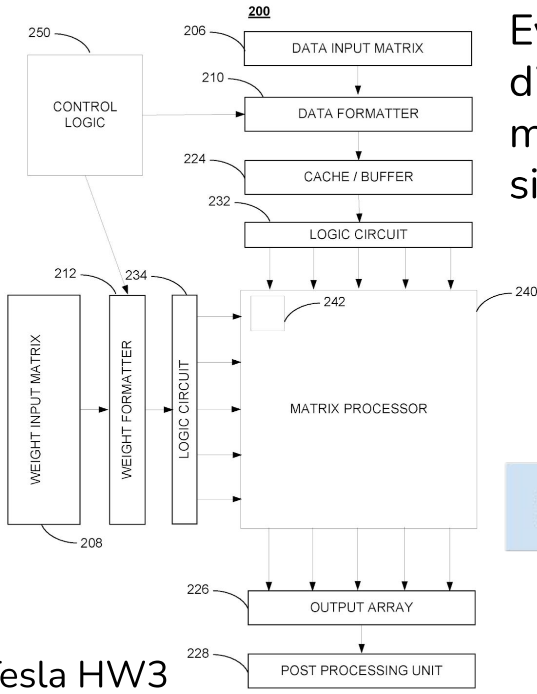
```
// add/min/max
unsigned __reduce_add_sync(unsigned mask, unsigned value);
unsigned __reduce_min_sync(unsigned mask, unsigned value);
unsigned __reduce_max_sync(unsigned mask, unsigned value);
int __reduce_add_sync(unsigned mask, int value);
int __reduce_min_sync(unsigned mask, int value);
int __reduce_max_sync(unsigned mask, int value);


// and/or/xor
unsigned __reduce_and_sync(unsigned mask, unsigned value);
unsigned __reduce_or_sync(unsigned mask, unsigned value);
unsigned __reduce_xor_sync(unsigned mask, unsigned value);
```

Tensor Core

Proliferation of TCUs

Many companies have now developed chips with TCUs to accelerate DNN computation:

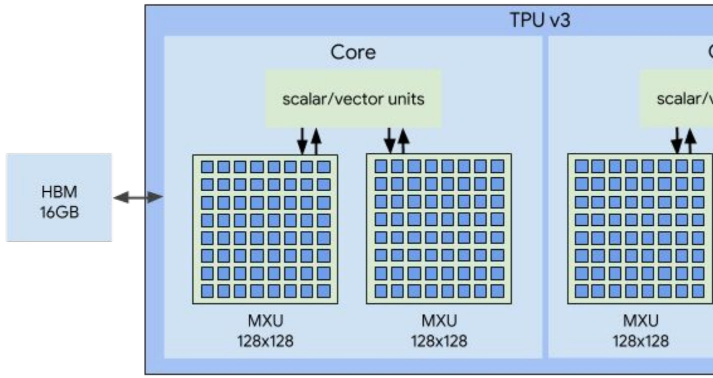
- NVIDIA TensorCores
- Google's TPU
- Tesla HW3
- Apple's A11
- Intel AMX/DLBoost
- Many many startups



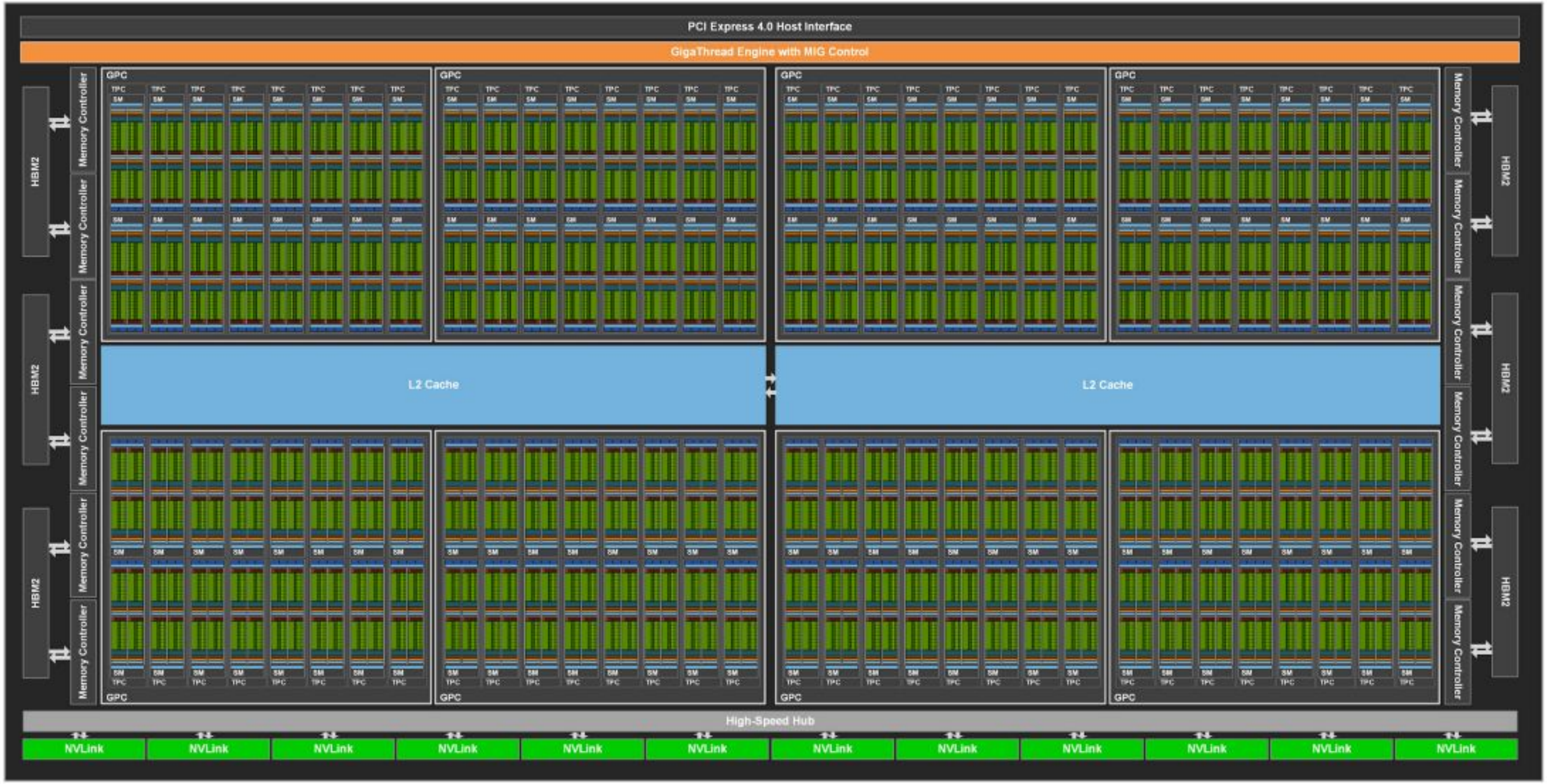
Tesla HW3

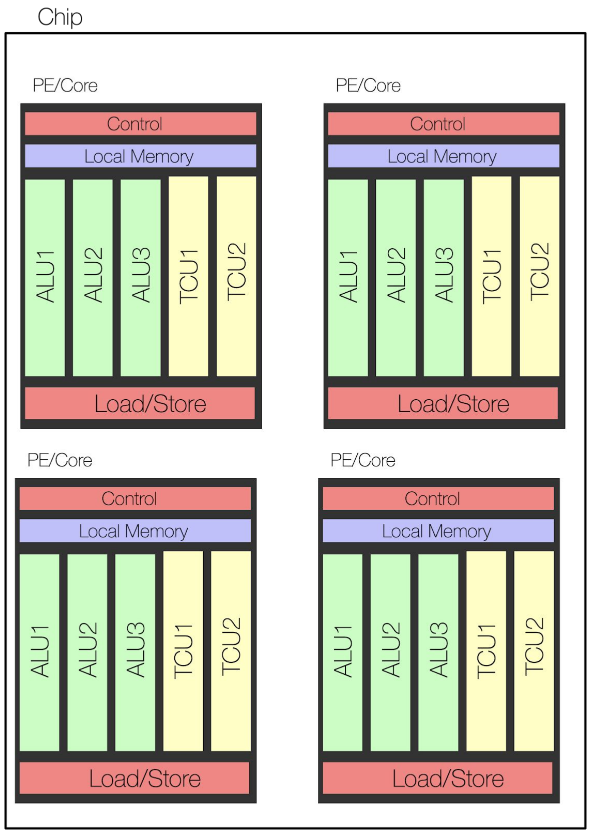
Even though they are named differently, they are all designed as matrix multiplication units for fixed size matrices.

Google TPU

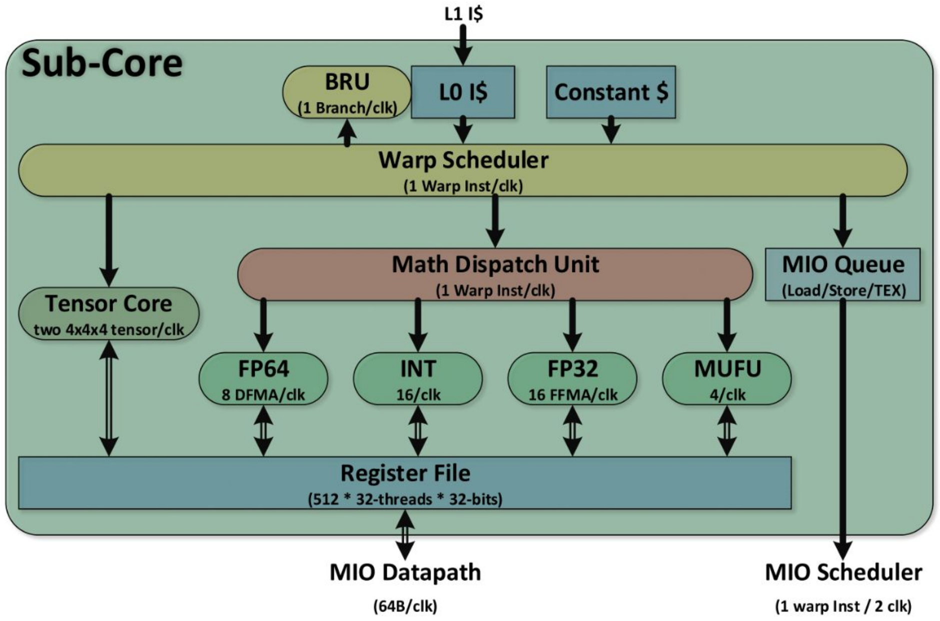


A100 block diagram: 108 SMs



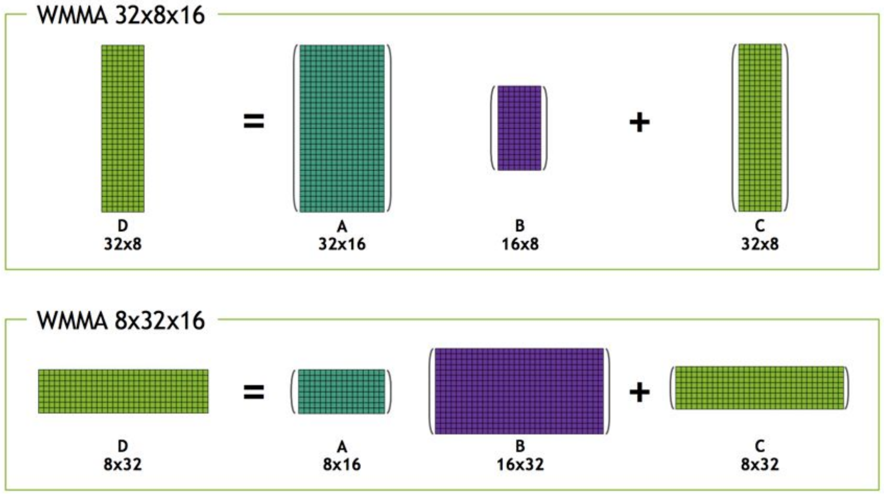


NVIDIA V100 SubCore



TensoreCore Operations

- Fixed size matrix multiplication: usually 4x4, 16x16, or 64x64
- NVIDIA supports non-square matrix dimensions



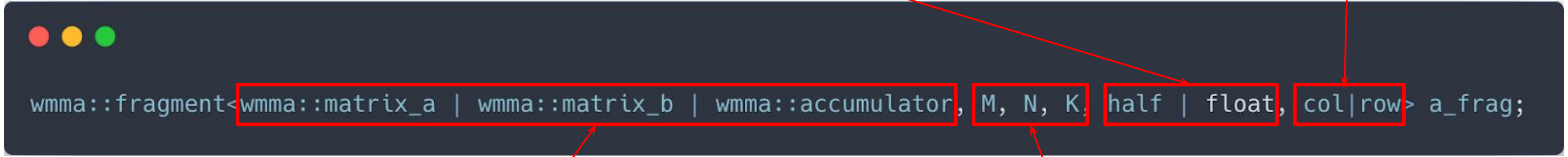
Warp Matrix Functions

```
template<typename Use, int m, int n, int k, typename T, typename Layout=void> class fragment;


void load_matrix_sync(fragment<...> &a, const T* mptr, unsigned ldm);

void load_matrix_sync(fragment<...> &a, const T* mptr, unsigned ldm, layout_t layout);

void store_matrix_sync(T* mptr, const fragment<...> &a, unsigned ldm, layout_t layout);

void fill_fragment(fragment<...> &a, const T& v);

void mma_sync(fragment<...> &d, const fragment<...> &a, const fragment<...> &b, const
fragment<...> &c, bool satf=false);
```

wmma::fragment

Element type of fragment. Float is only supported for accumulator fragments (mixed precision)

Layout of fragment: column or row major order.



```
wmma::fragment<wmma::matrix_a | wmma::matrix_b | wmma::accumulator, M, N, K, half | float, col|row> a_frag;
```

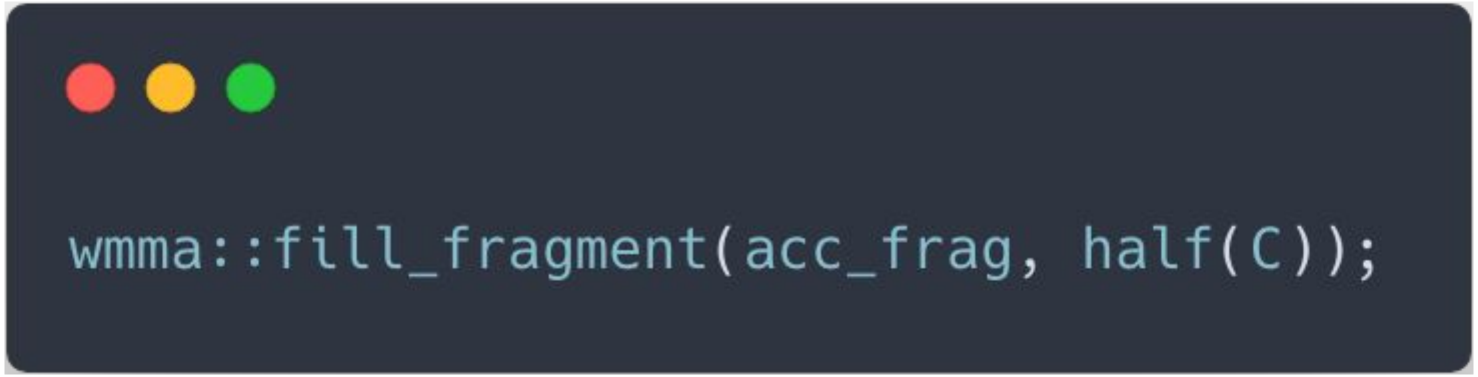
Kind of fragment:
An A matrix, B matrix, or a C (accumulator) matrix

Dimensions of fragment. For our purposes these are 16, 16, 16, but some other values are possible

A memory fragments (internally a set of registers).

Result =A * B + C

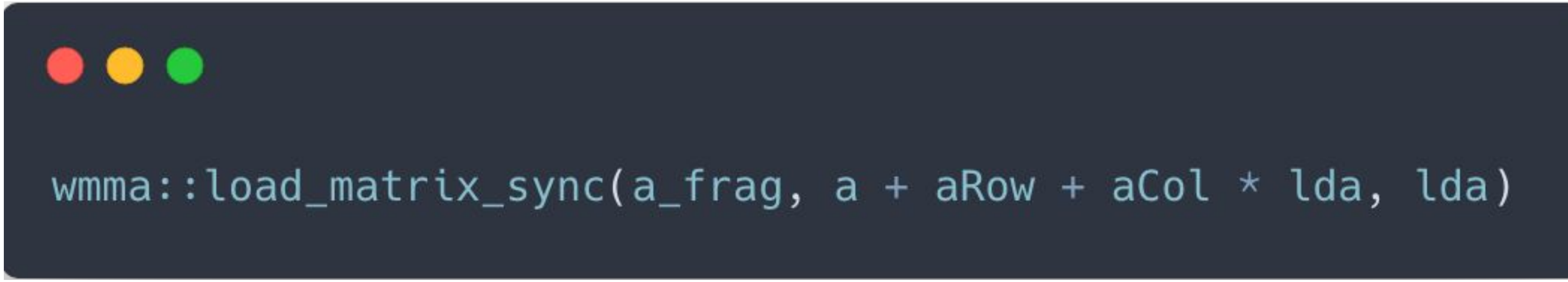
wmma:fill_fragment



```
wmma::fill_fragment(acc_frag, half(C));
```

Fills the fragment with some constant value **C**

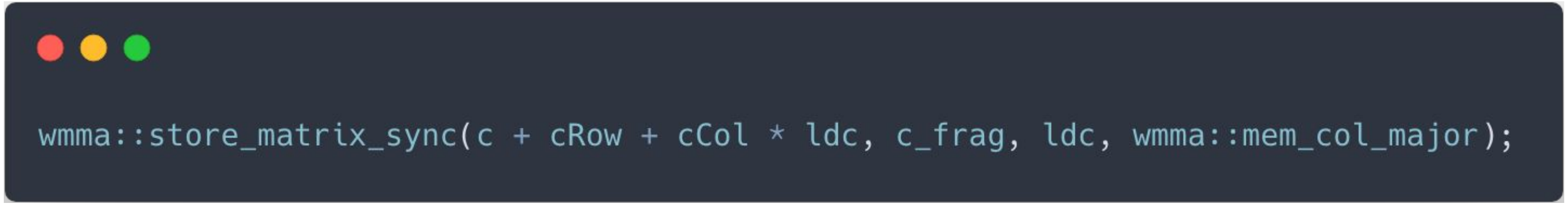
wmma:load_matrix_sync



```
wmma::load_matrix_sync(a_frag, a + aRow + aCol * lda, lda)
```

Loads data from global or shared memory with specified stride into the fragment. Here we load a tile of a matrix (at **`&a[aRow+aCol*lda]`**) into **`a_frag`** with **`lda`** as the stride

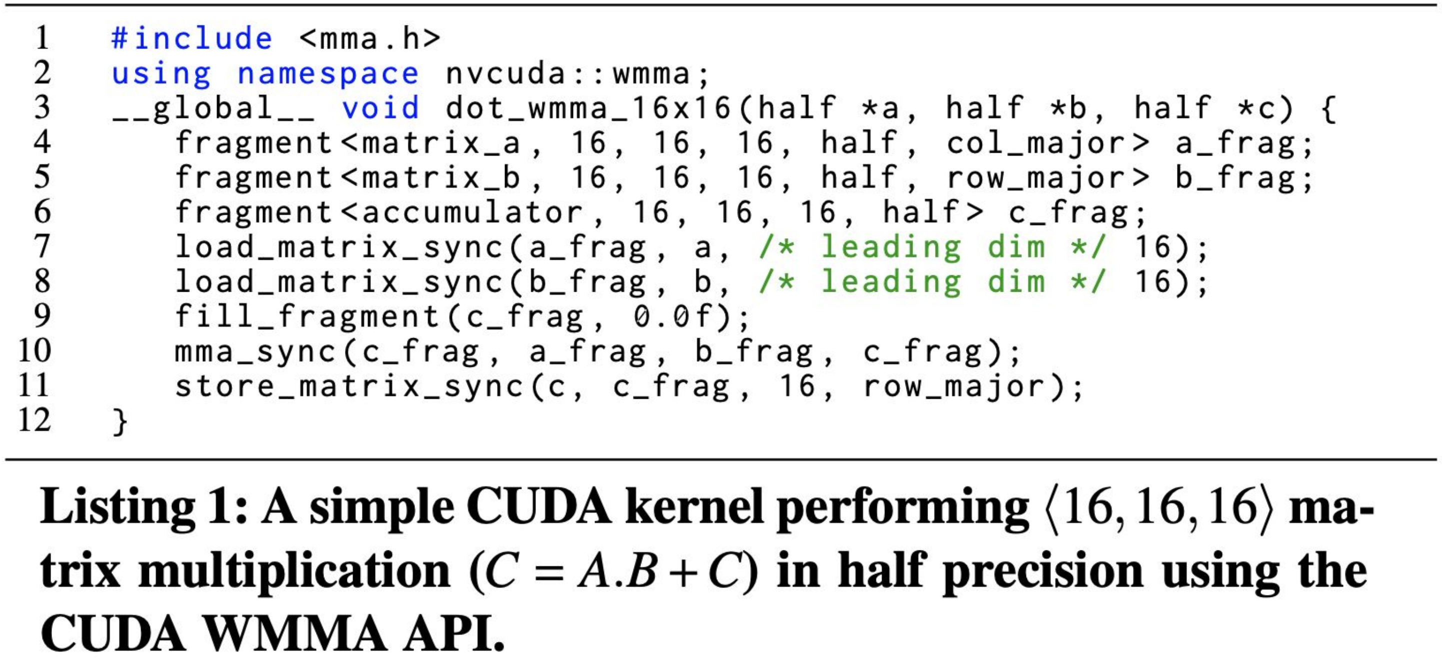
wmma:store_matrix_sync



```
wmma::store_matrix_sync(c + cRow + cCol * ldc, c_frag, ldc, wmma::mem_col_major);
```

Stores the fragment into global or shared memory with specified stride and layout.

Example

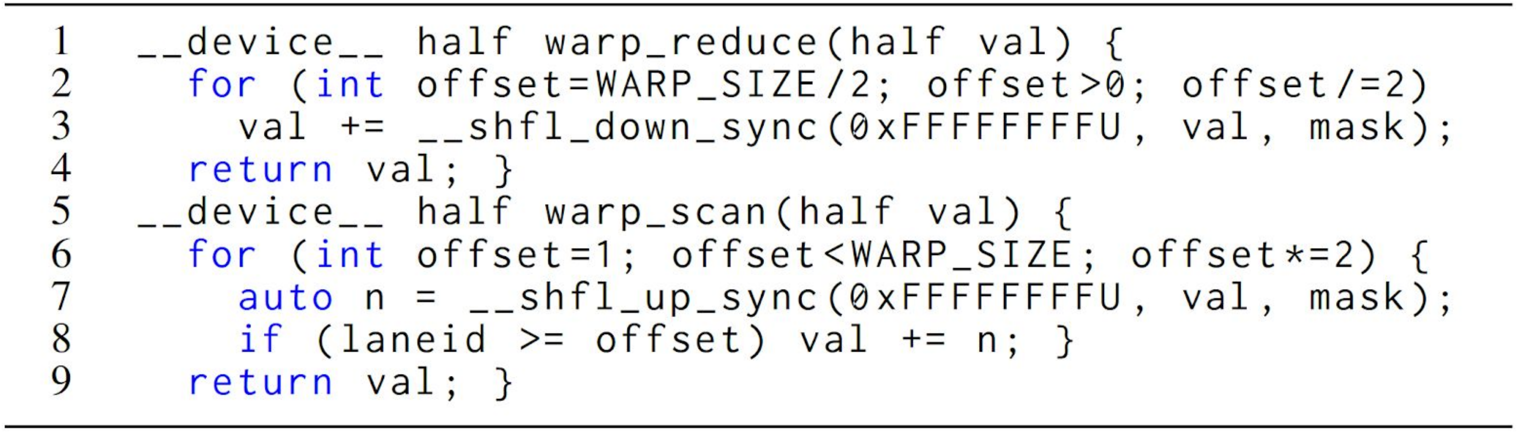


```
#include <mma.h>
using namespace nvcuda::wmma;
__global__ void dot_wmma_16x16(half *a, half *b, half *c) {
   fragment<matrix_a, 16, 16, 16, half, col_major> a_frag;
   fragment<matrix_b, 16, 16, 16, half, row_major> b_frag;
   fragment<accumulator, 16, 16, 16, half> c_frag;
   load_matrix_sync(a_frag, a, /* leading dim */ 16);
   load_matrix_sync(b_frag, b, /* leading dim */ 16);
   fill_fragment(c_frag, 0.0f);
   mma_sync(c_frag, a_frag, b_frag, c_frag);
   store_matrix_sync(c, c_frag, 16, row_major);
}
```

Listing 1: A simple CUDA kernel performing $\langle 16, 16, 16 \rangle$ matrix multiplication ($C = A.B + C$) in half precision using the CUDA WMMA API.

Recall

Warp reduction is performed using the shuffle instructions



```
__device__ half warp_reduce(half val) {
  for (int offset=WARP_SIZE/2; offset>0; offset/=2)
    val += __shfl_down_sync(0xFFFFFFFFU, val, mask);
  return val; }
__device__ half warp_scan(half val) {
  for (int offset=1; offset<WARP_SIZE; offset*=2) {
    auto n = __shfl_up_sync(0xFFFFFFFFU, val, mask);
    if (laneid >= offset) val += n; }
  return val; }
```

Listing 2: NVIDIA's recommended warp-level reduction and scan implementations utilizing shuffle instructions.

We want to do the same thing, but now using TensorCores

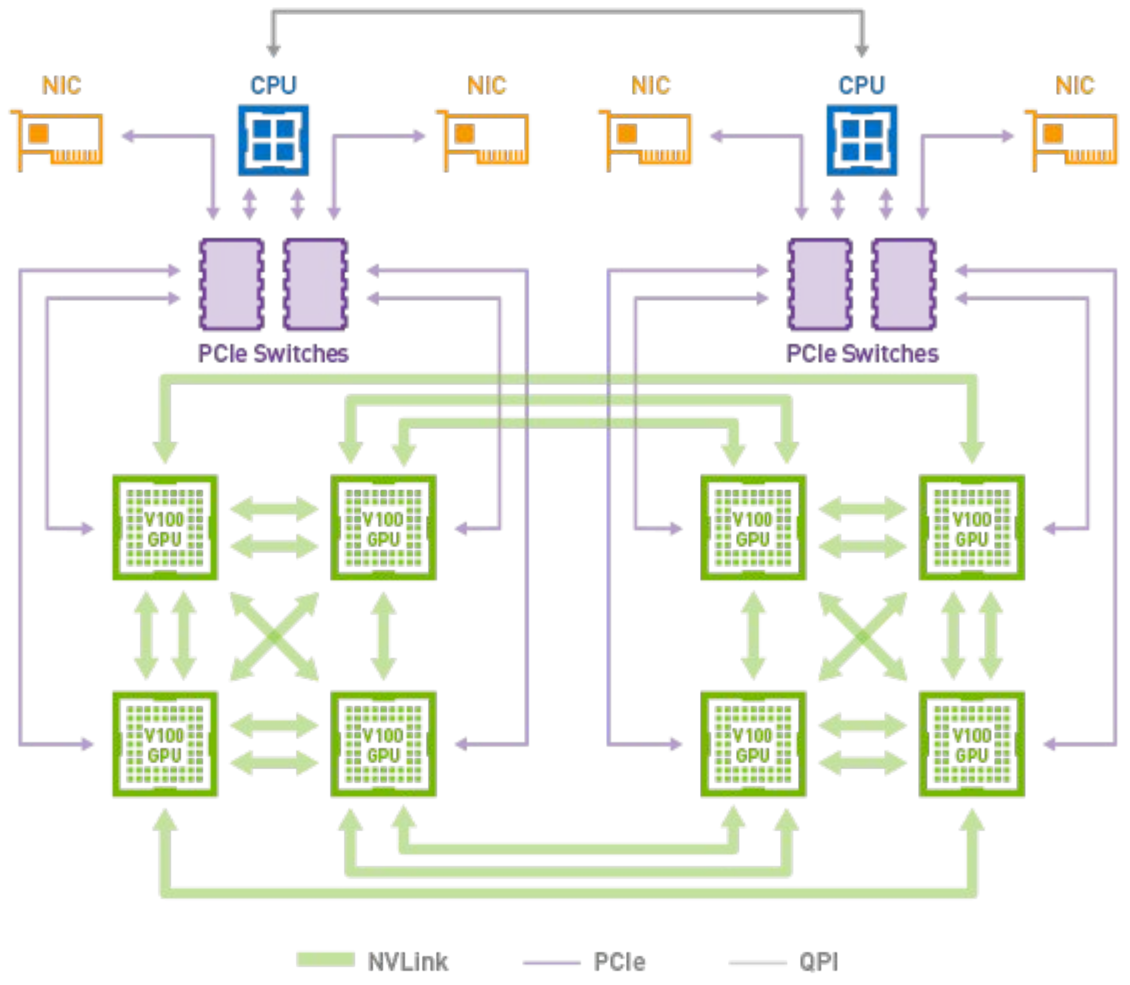
Reduction

Easy to represent it as a matrix multiplication

$$Reduction\left[a_1, a_2, \ldots, a_n\right] = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{pmatrix} \cdot \begin{pmatrix} a_1 & a_2 & \ldots & a_n \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{pmatrix}^T = \begin{pmatrix} {}_{i=1}^{n} a_i & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{pmatrix}$$

There is a lot of compute waste here

NVLink



PCIe Gen3: 32GB/sec

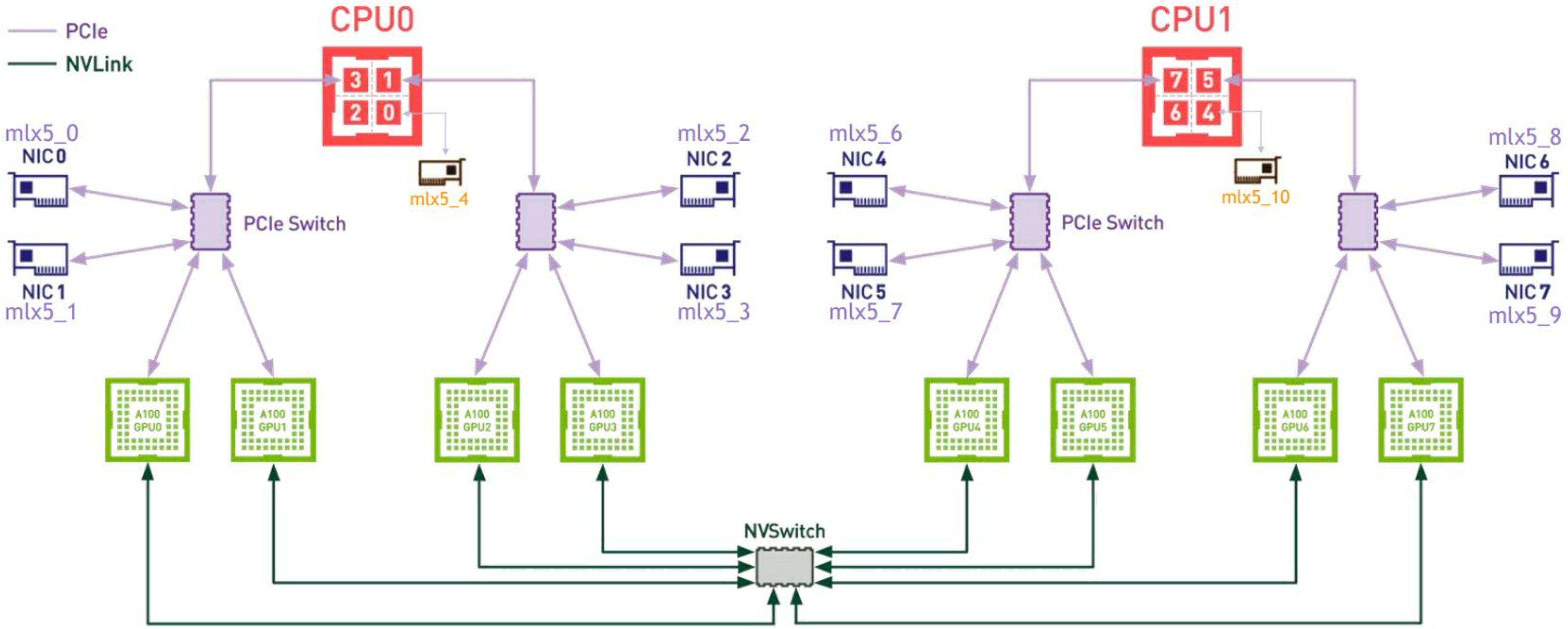
HBM2 BW: 900GB/sec

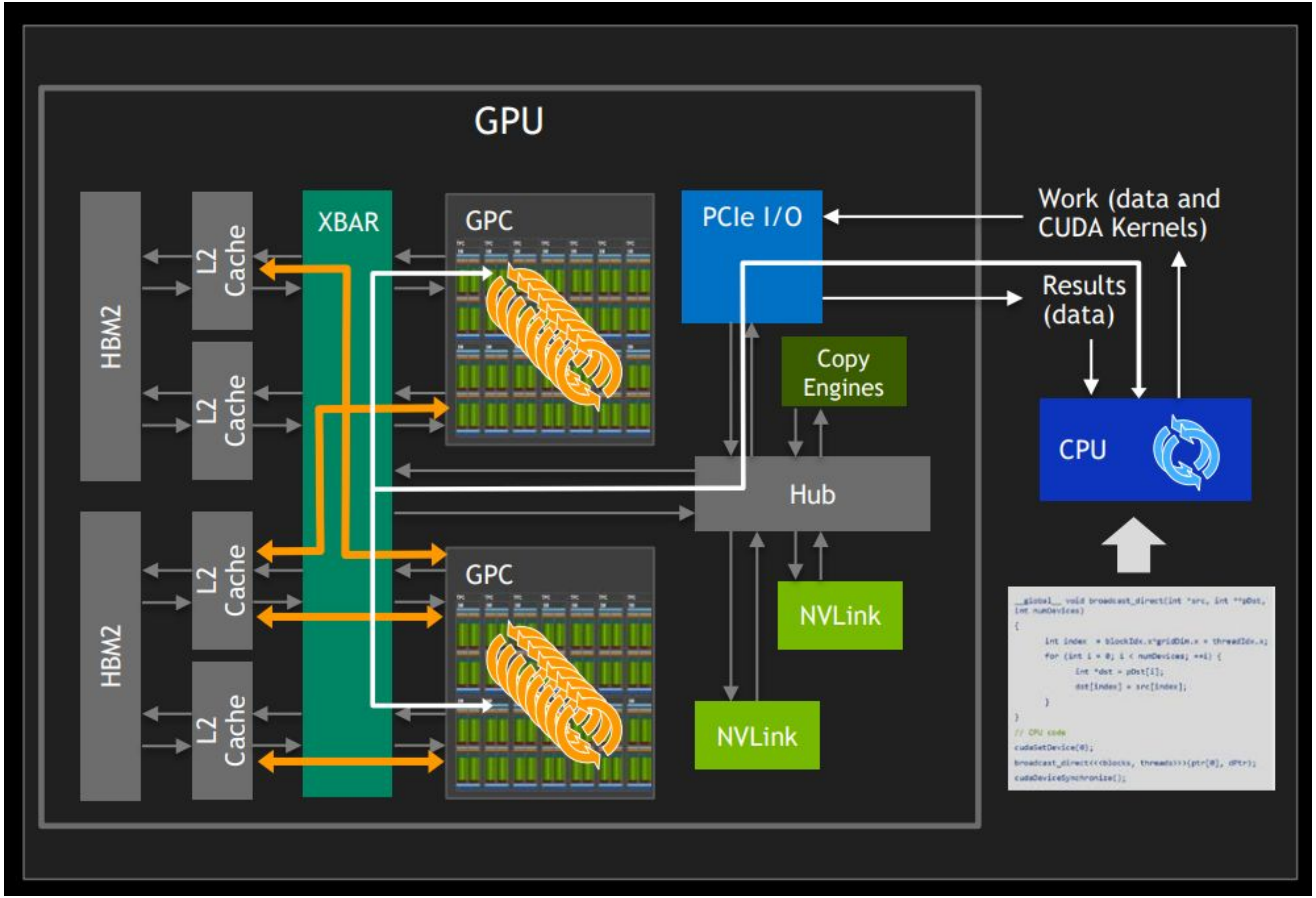
NVLINK 2.0: 300GB/sec

DGX A100

High-level Topology Overview (with options)

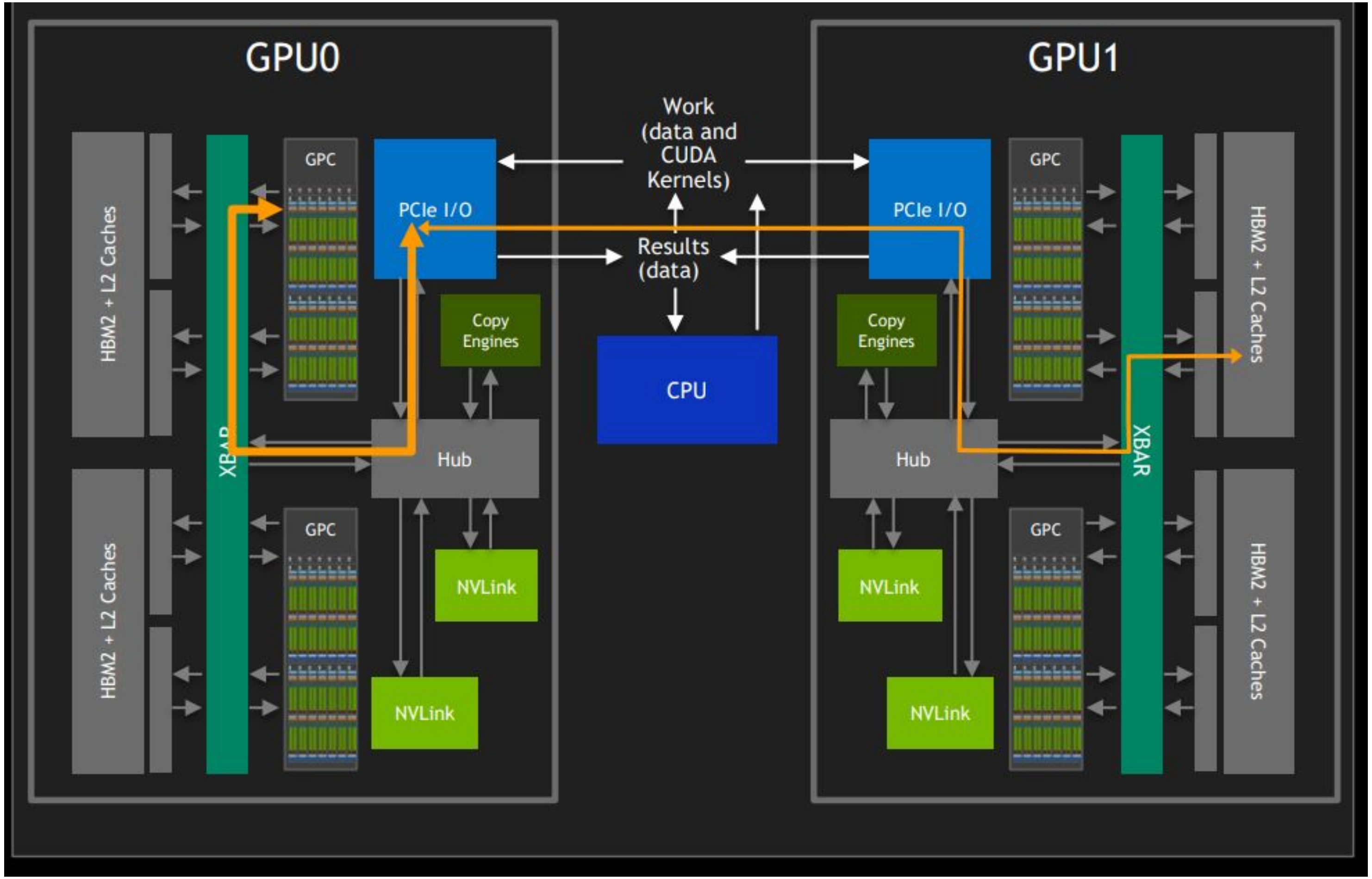
Data plane (can be used as eth or IB)
Compute plane (IB)



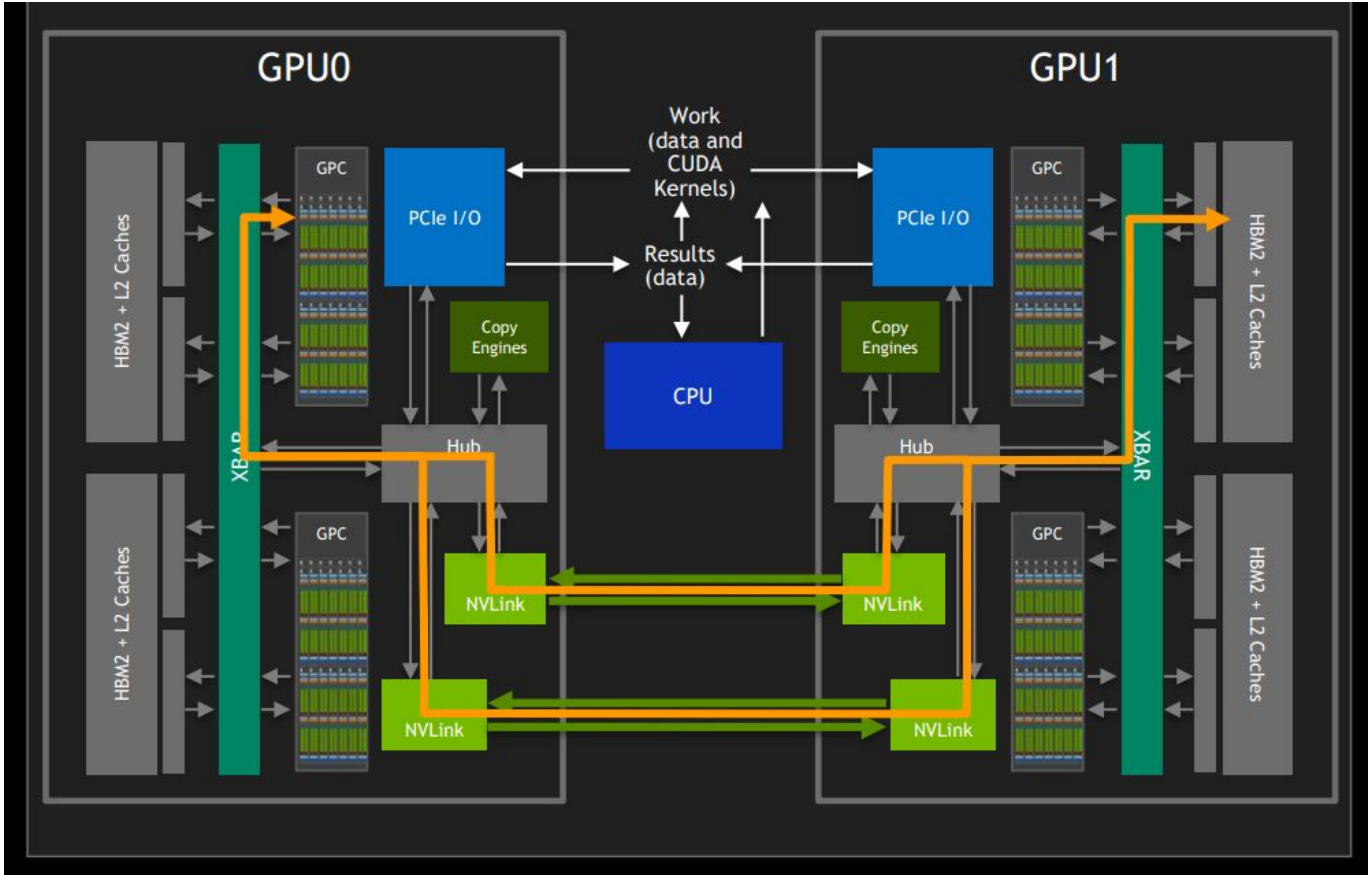


```
__global__ void broadcast_direct(int *src, int **pDst,
int numDevices)
{
        int index  = blockIdx.x*gridDim.x + threadIdx.x;
        for (int i = 0; i < numDevices; ++i) {
                int *dst = pDst[i];
                dst[index] = src[index];
        }
}
// CPU code
cudaSetDevice(0);
broadcast_direct<<<blocks, threads>>>(ptr[0], dPtr);
cudaDeviceSynchronize();
```

Interactions with CPU compete with GPU-to-GPU

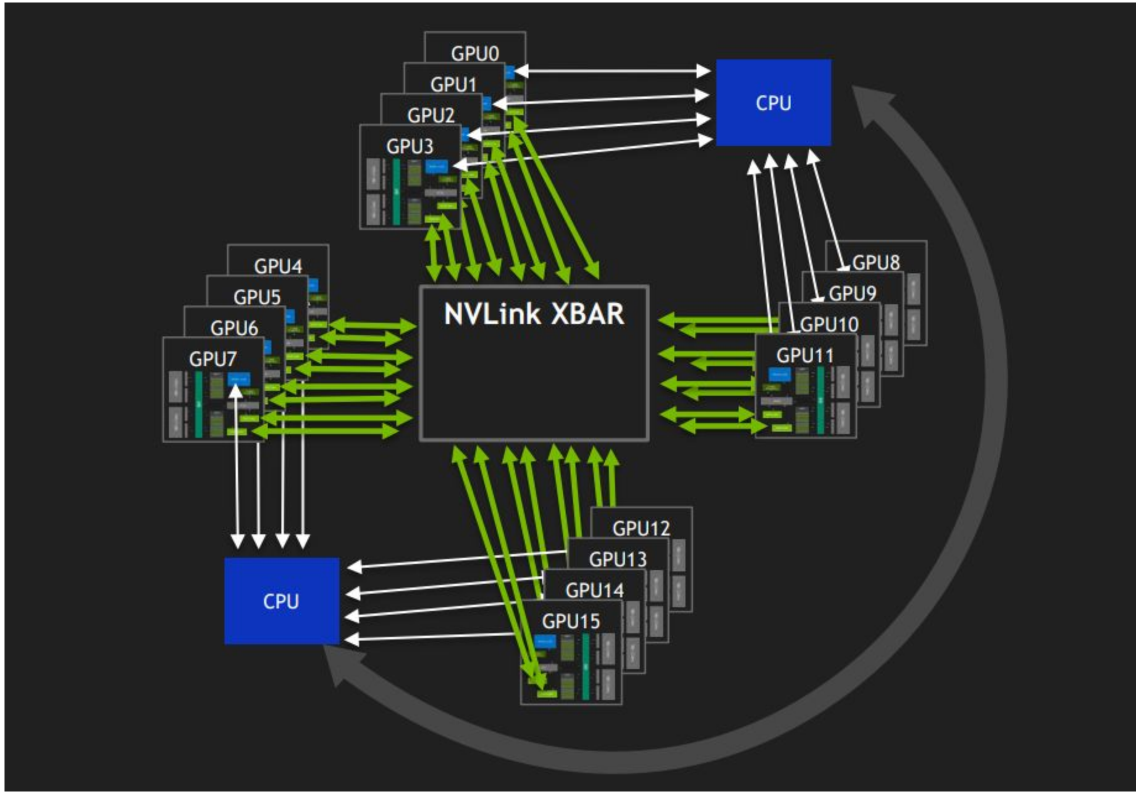


- NVLinks are effectively a “bridge” between XBARs
- No collisions with PCIe traffic



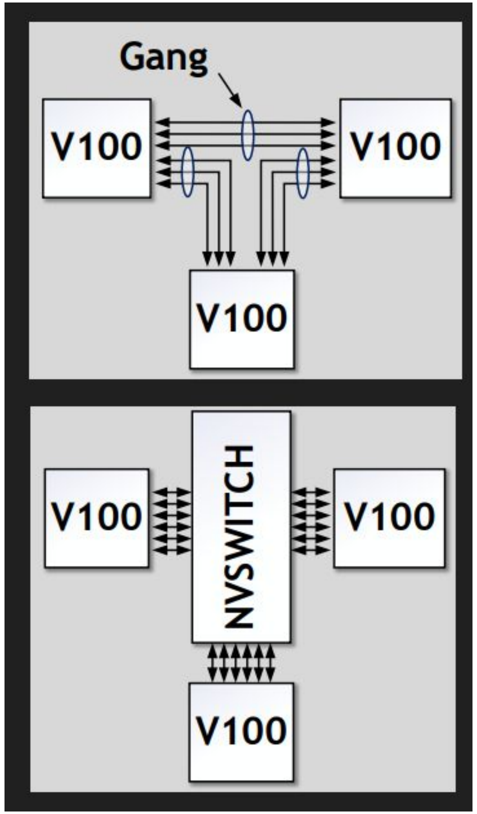
THE “ONE GIGANTIC GPU” IDEAL

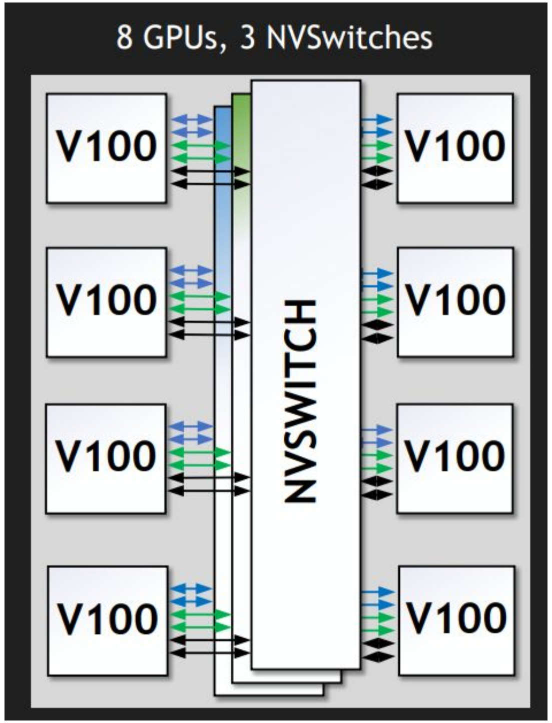
- Highest number of GPUs possible
- Single GPU Driver process controls all work across all GPUs
- From perspective of GPCs, all HBM2s can be accessed without intervention by other processes (LD/ST instructions, Copy Engine RDMA, everything “just works”)
- Access to all HBM2s is independent of PCIe
- Bandwidth across bridged XBARs is as high as possible (some NUMA is unavoidable)



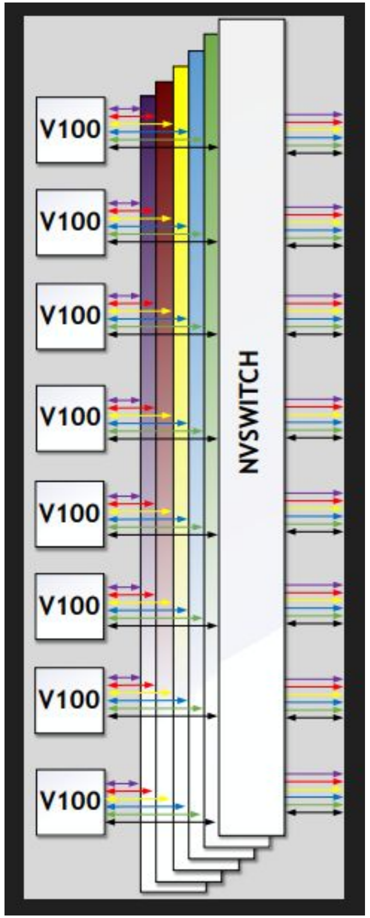
NVSwitch

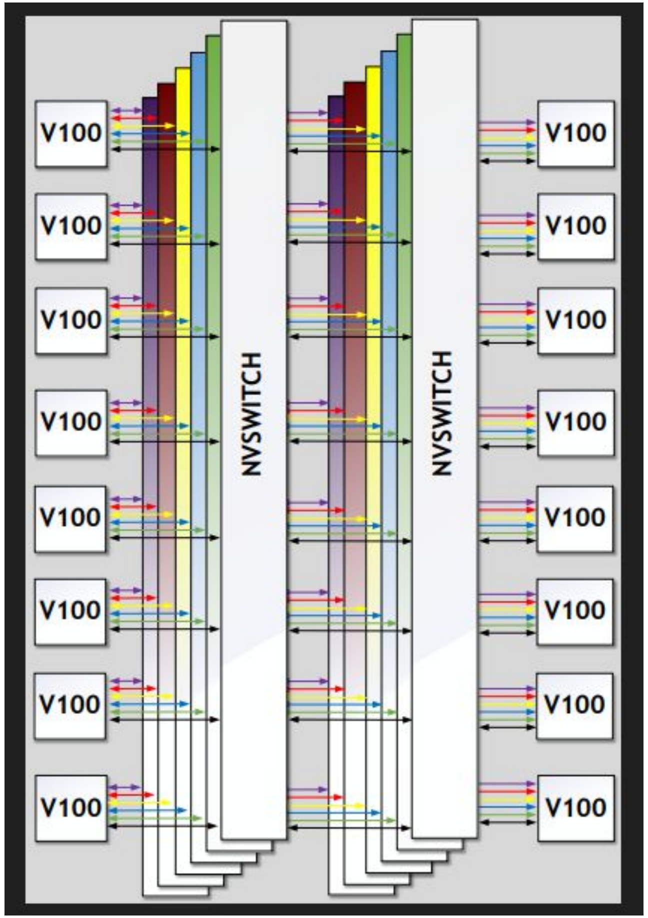
- No NVSwitch
    - Connect GPU directly
    - Aggregate NVLinks into gangs for higher bandwidth
    - Interleaved over the links to prevent camping
    - Max bandwidth between two GPUs limited to the bandwidth of the gang
- With NVSwitch
    - Interleave traffic across all the links and to support full bandwidth between any pair of GPUs
    - Traffic to a single GPU is nonblocking, so long as aggregate bandwidth of six NVLinks is not



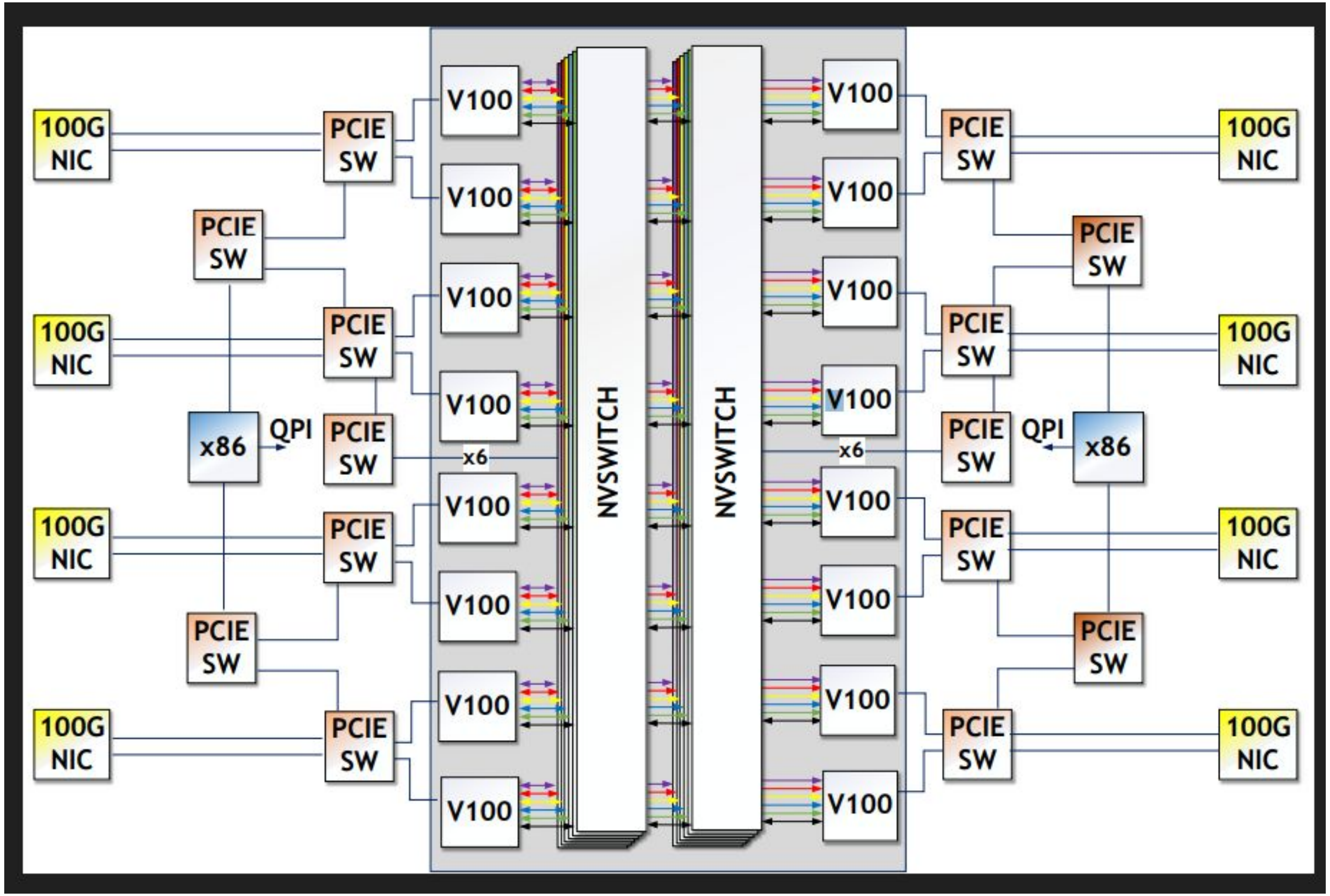

8 GPUs, 3 NVSwitches
V100
V100
V100
V100
V100
V100
V100
V100
NVSWITCH

DGX-2


V100
V100
V100
V100
V100
V100
V100
V100
NVSWITCH

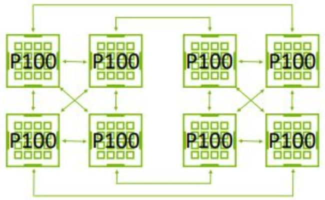

V100
V100
V100
V100
V100
V100
V100
V100
NVSWITCH
NVSWITCH
V100
V100
V100
V100
V100
V100
V100
V100

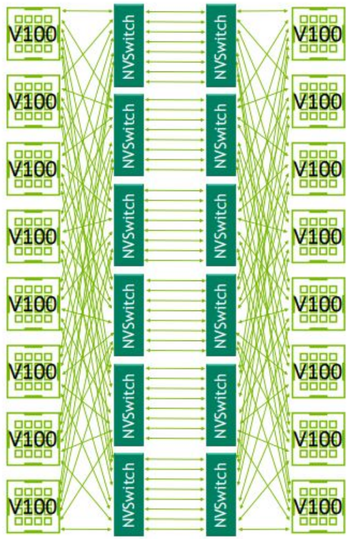
DGX-2 PCIe Network

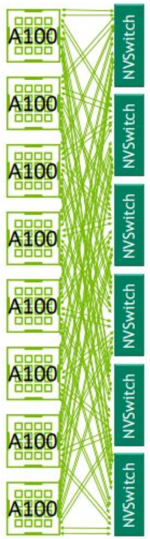


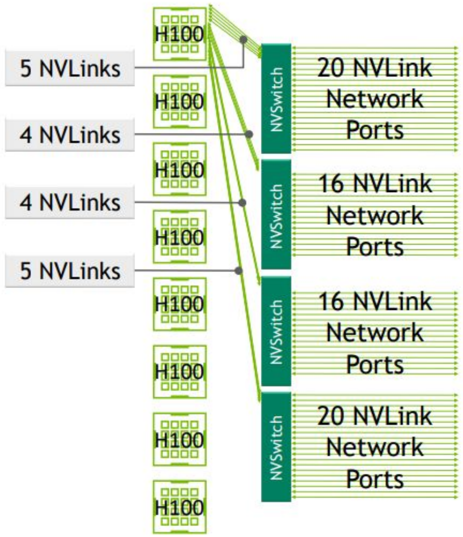
NVLINK-ENABLED SERVER GENERATIONS

Any-to-Any Connectivity with NVSwitch









2016

DGX-1 (P100)

140GB/s Bisection BW
40GB/s AllReduce BW

2018

DGX-2 (V100)

2.4TB/s Bisection BW
75GB/s AllReduce BW

2020

DGX A100

2.4TB/s Bisection BW
150GB/s AllReduce BW

2022

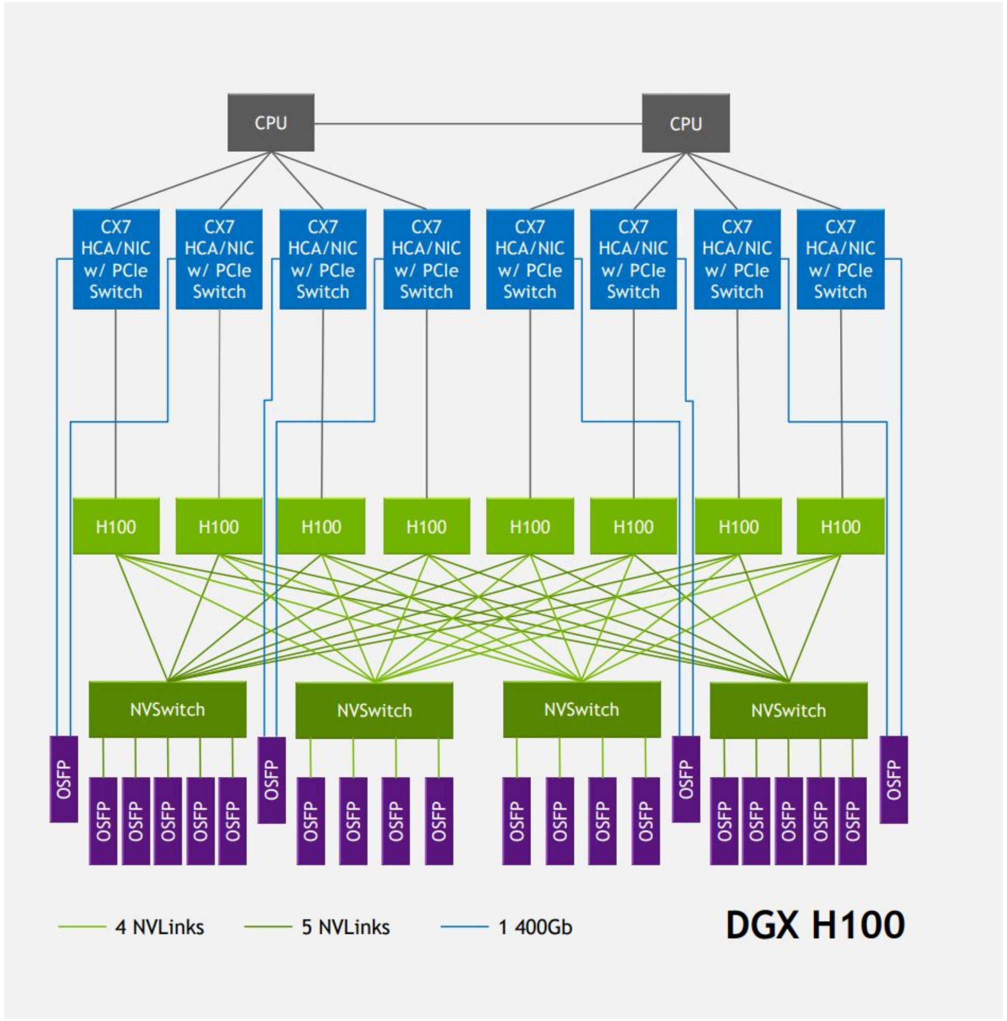
DGX H100

3.6TB/s Bisection BW
450GB/s AllReduce BW

MAPPING TO TRADITIONAL NETWORKING

NVLink Network is Tightly Integrated with GPU

Concept	Traditional Example	NVLink Network
Physical Layer	400G electrical/optical media	Custom-FW OSFP
Data Link Layer	Ethernet	NVLink custom on-chip HW and FW
Network Layer	IP	New NVLink Network Addressing and Management Protocols
Transport Layer	TCP	NVLink custom on-chip HW and FW
Session Layer	Sockets	SHARP groups CUDA export of Network addresses of data-structures
Presentation Layer	TSL/SSL	Library abstractions (e.g., NCCL, NVSHMEM)
Application Layer	HTTP/FTP	AI Frameworks or User Apps
NIC	PCIe NIC (card or chip)	Functions embedded in GPU and NVSwitch
RDMA Off-Load	NIC Off-Load Engine	GPU-internal Copy Engine
Collectives Off-Load	NIC/Switch Off-Load Engine	NVSwitch-internal SHARP Engines
Security Off-Load	NIC Security Features	GPU-internal Encryption and “TLB” Firewalls
Media Control	NIC Cable Adaptation	NVSwitch-internal OSFP-cable controllers



DGX H100: DATA-NETWORK CONFIGURATION

Full-BW Intra-Server NVLink

- All 8 GPUs can simultaneously saturate 18 NVLinks to other GPUs within server
- Limited only by over-subscription from multiple other GPUs

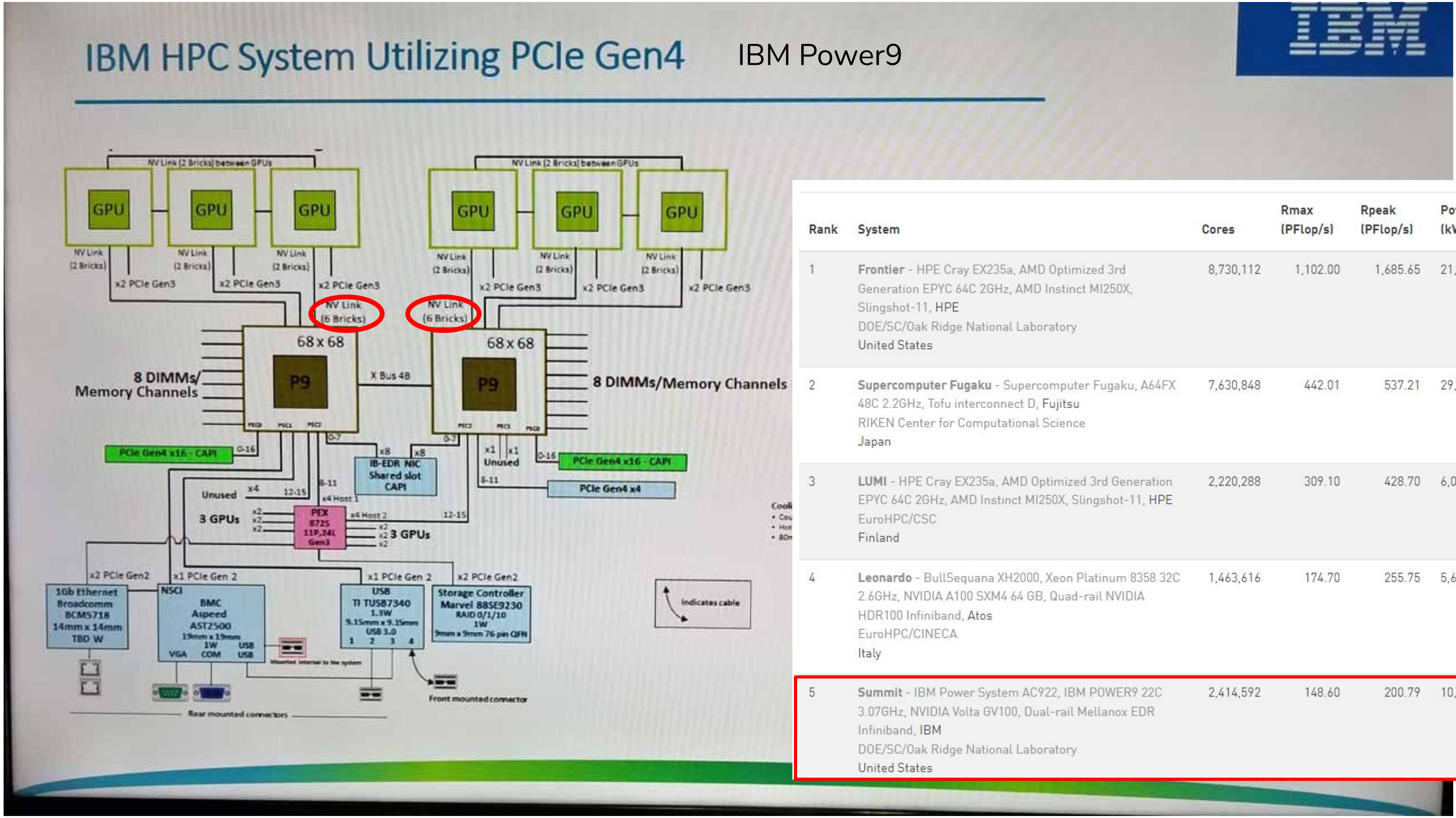
Half-BW NVLink Network

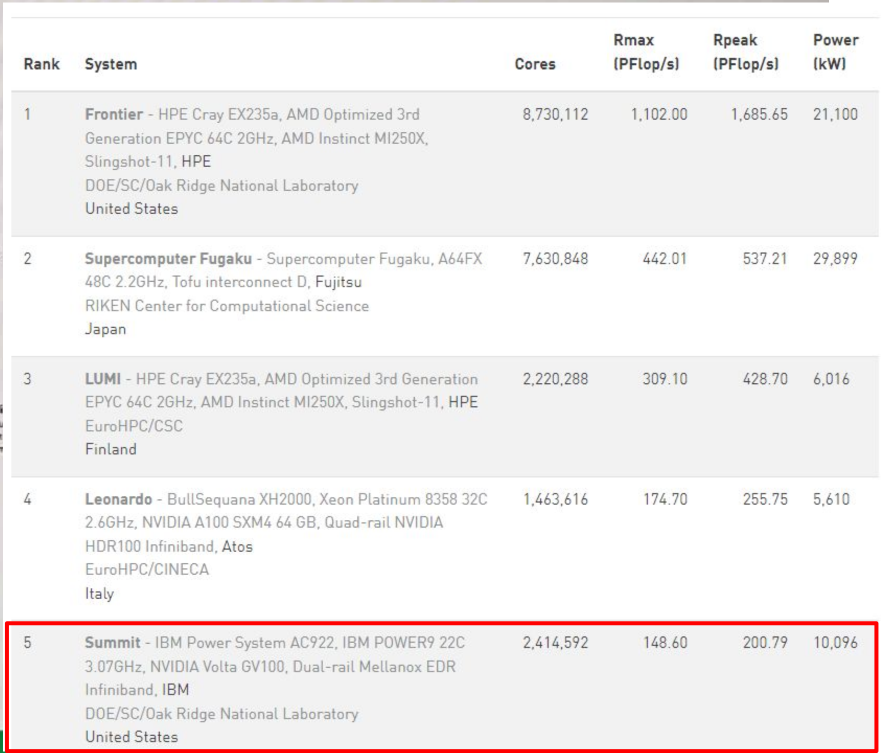
- All 8 GPUs can half-subscribe 18 NVLinks to GPUs in other servers
- 4 GPUs can saturate 18 NVLinks to GPUs in other servers
- Equivalent of full-BW on AllReduce with SHARP
- Reduction in All2All BW is a balance with server complexity and costs

Multi-Rail InfiniBand/Ethernet

- All 8 GPUs can independently RDMA data over its own dedicated 400 Gb/s HCA/NIC
- 800 GBps of aggregate full-duplex to non-NVLink Network devices

IBM Power9

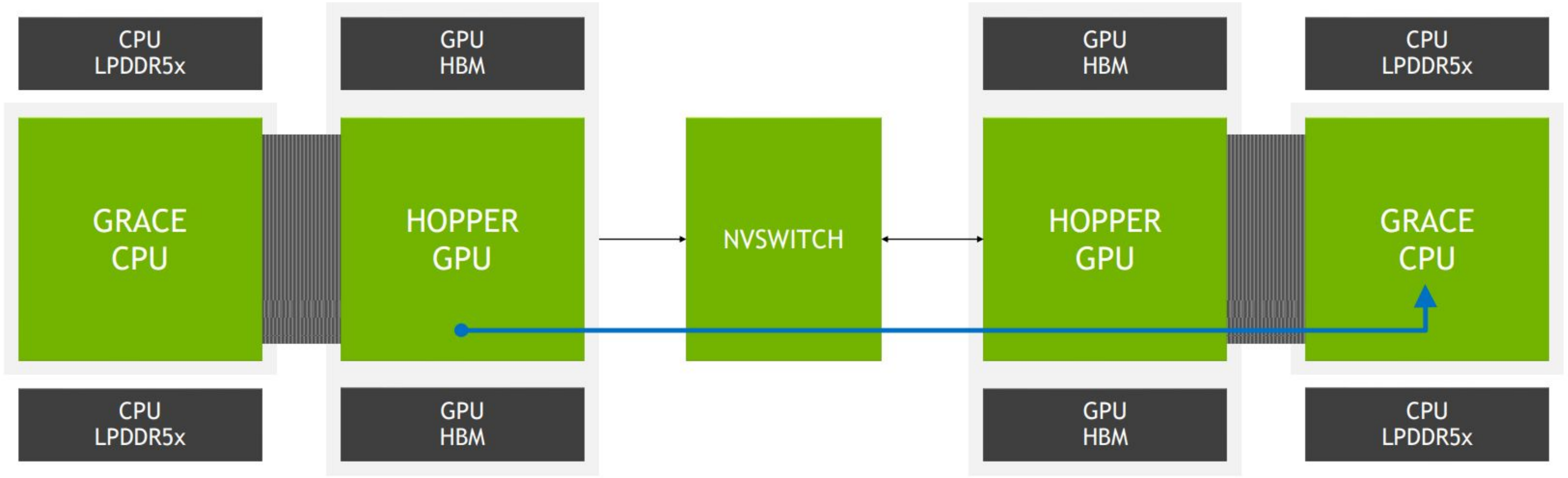




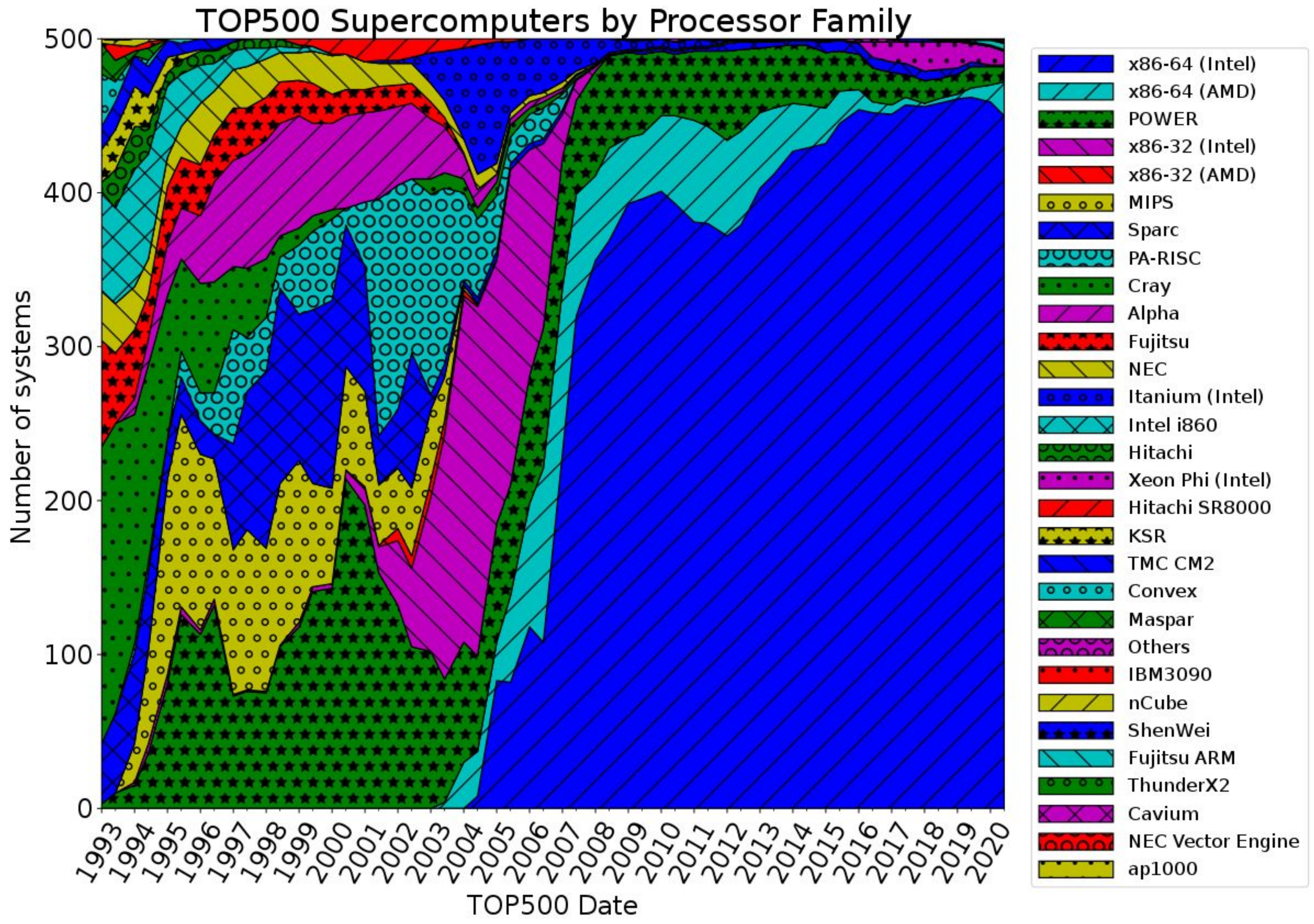
Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,730,112	1,102.00	1,685.65	21,100
2	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
3	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,220,288	309.10	428.70	6,016
4	**Leonardo** - BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 64 GB, Quad-rail NVIDIA HDR100 Infiniband, Atos EuroHPC/CINECA Italy	1,463,616	174.70	255.75	5,610
5	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148.60	200.79	10,096

NVLINK-C2C

Superchip Scaling | CPU/GPU | Extended GPU Memory



Enables remote NVLINK connected GPUs, to access Grace's memory at native NVLINK speeds


TOP500 Supercomputers by Processor Family
Number of systems
0
100
200
300
400
500
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
TOP500 Date
x86-64 (Intel)
x86-64 (AMD)
POWER
x86-32 (Intel)
x86-32 (AMD)
MIPS
Sparc
PA-RISC
Cray
Alpha
Fujitsu
NEC
Itanium (Intel)
Intel i860
Hitachi
Xeon Phi (Intel)
Hitachi SR8000
KSR
TMC CM2
Convex
Maspar
Others
IBM3090
nCube
ShenWei
Fujitsu ARM
ThunderX2
Cavium
NEC Vector Engine
ap1000

Reference

- Dakkak, Abdul et al. “Accelerating reduction and scan using tensor core units.” *Proceedings of the ACM International Conference on Supercomputing* (2018): n. Pag.
- (HC2018) NVSWITCH AND DGX-2 NVLINK-SWITCHING CHIP AND SCALE-UP COMPUTE SERVER
- (HC2022) THE NVLINK-NETWORK SWITCH: NVIDIA’S SWITCH CHIP FOR HIGH COMMUNICATION-BANDWIDTH SUPERPODS