

CS 295: Modern Systems

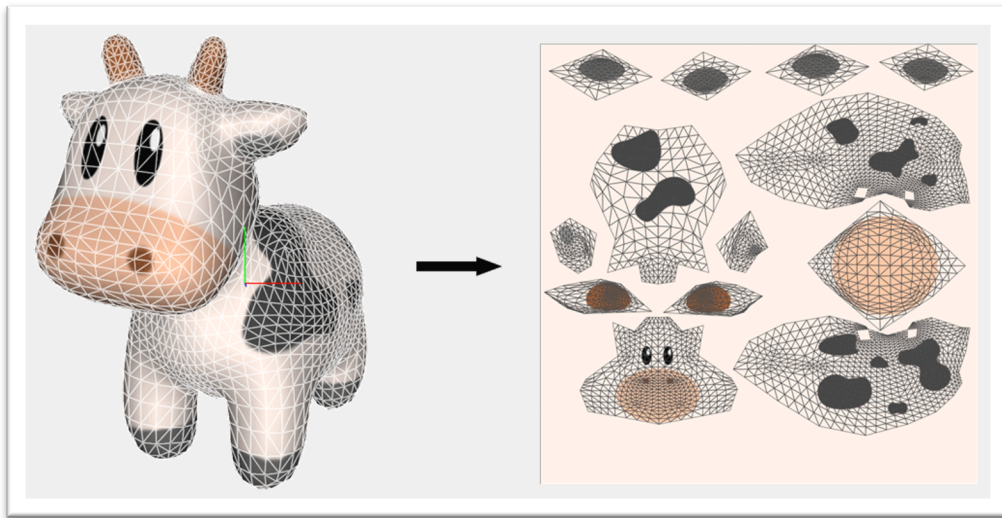
GPU Computing Introduction



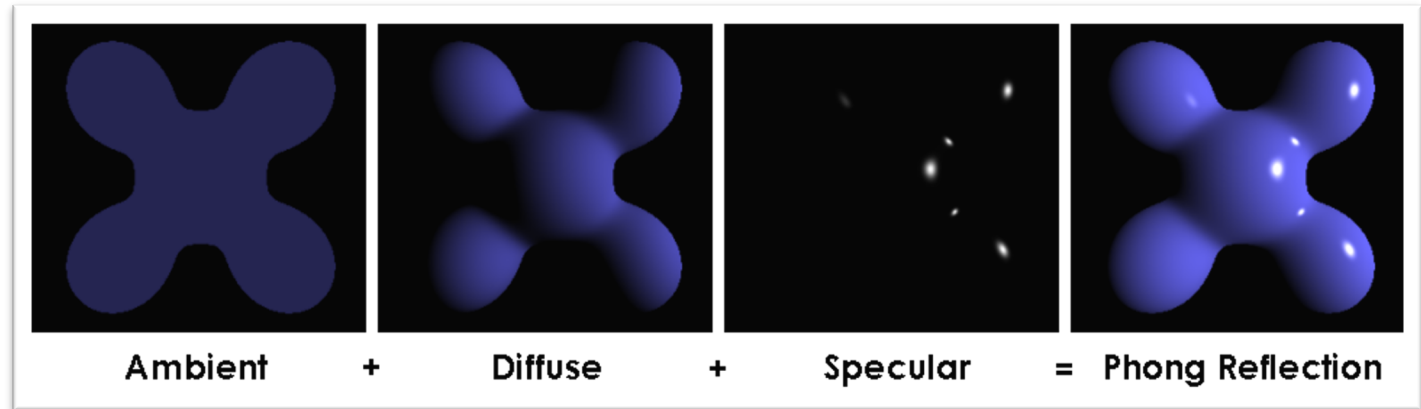
Sang-Woo Jun
Spring 2019

Graphic Processing – Some History

- ❑ 1990s: Real-time 3D rendering for video games were becoming common
 - Doom, Quake, Descent, ... (Nostalgia!)
- ❑ 3D graphics processing is immensely computation-intensive



Texture mapping



Shading

Graphic Processing – Some History

- ❑ Before 3D accelerators (GPUs) were common
- ❑ CPUs had to do all graphics computation, while maintaining framerate!
 - Many tricks were played



Doom (1993) : “Affine texture mapping”

- Linearly maps textures to screen location, disregarding depth
- Doom levels did not have slanted walls or ramps, to hide this

Graphic Processing – Some History

- ❑ Before 3D accelerators (GPUs) were common
- ❑ CPUs had to do all graphics computation, while maintaining framerate!
 - Many tricks were played



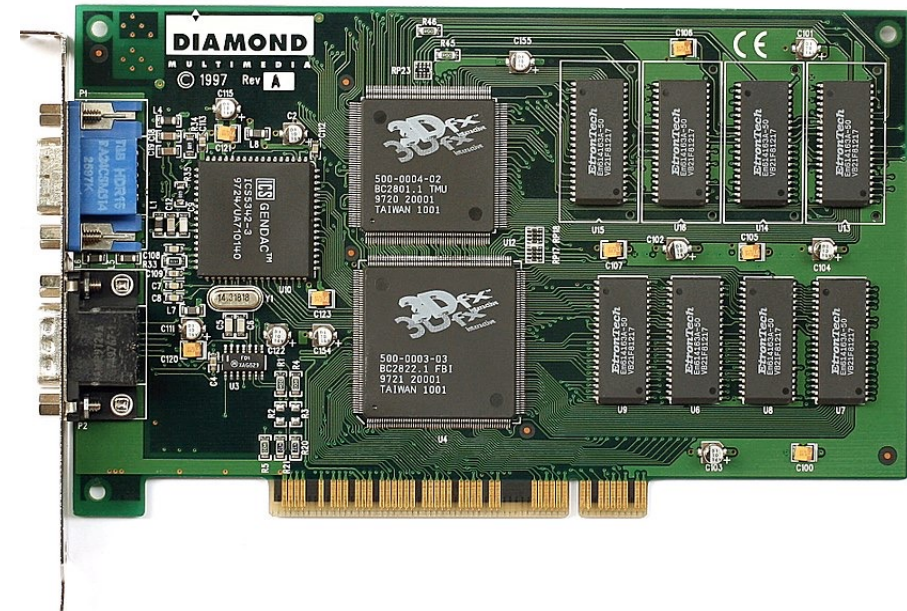
Quake III arena (1999) : “Fast inverse square root” magic!

```
float Q_rsqrt( float number )
{
    const float x2 = number * 0.5F;
    const float threehalfs = 1.5F;

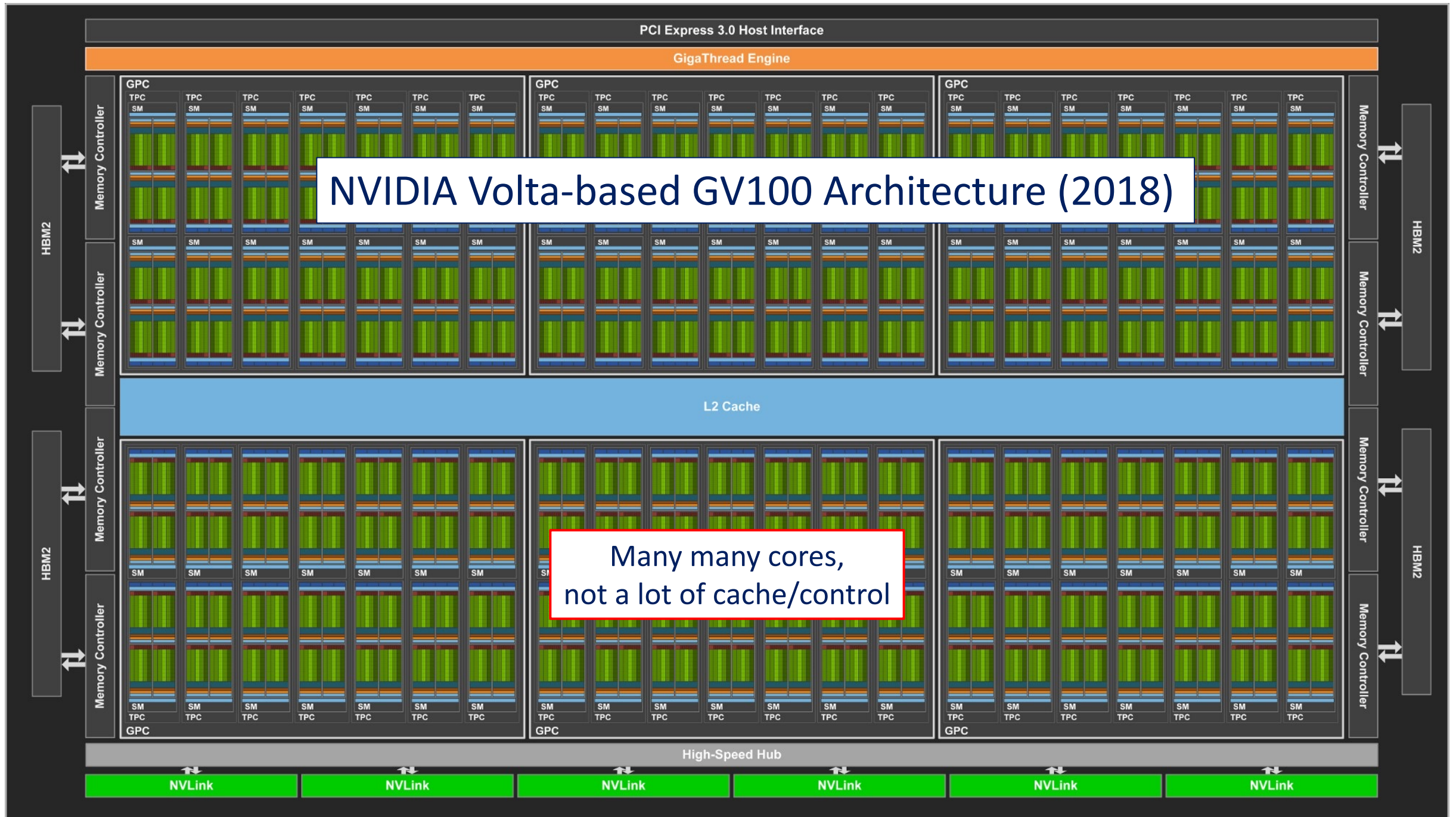
    union {
        float f;
        uint32_t i;
    } conv = {number}; // member 'f' set to value of 'number'.
    conv.i = 0x5f3759df - ( conv.i >> 1 );
    conv.f *= ( threehalfs - ( x2 * conv.f * conv.f ) );
    return conv.f;
}
```

Introduction of 3D Accelerator Cards

- ❑ Much of 3D processing is short algorithms repeated on a lot of data
 - pixels, polygons, textures, ...
- ❑ Dedicated accelerators with simple, massively parallel computation



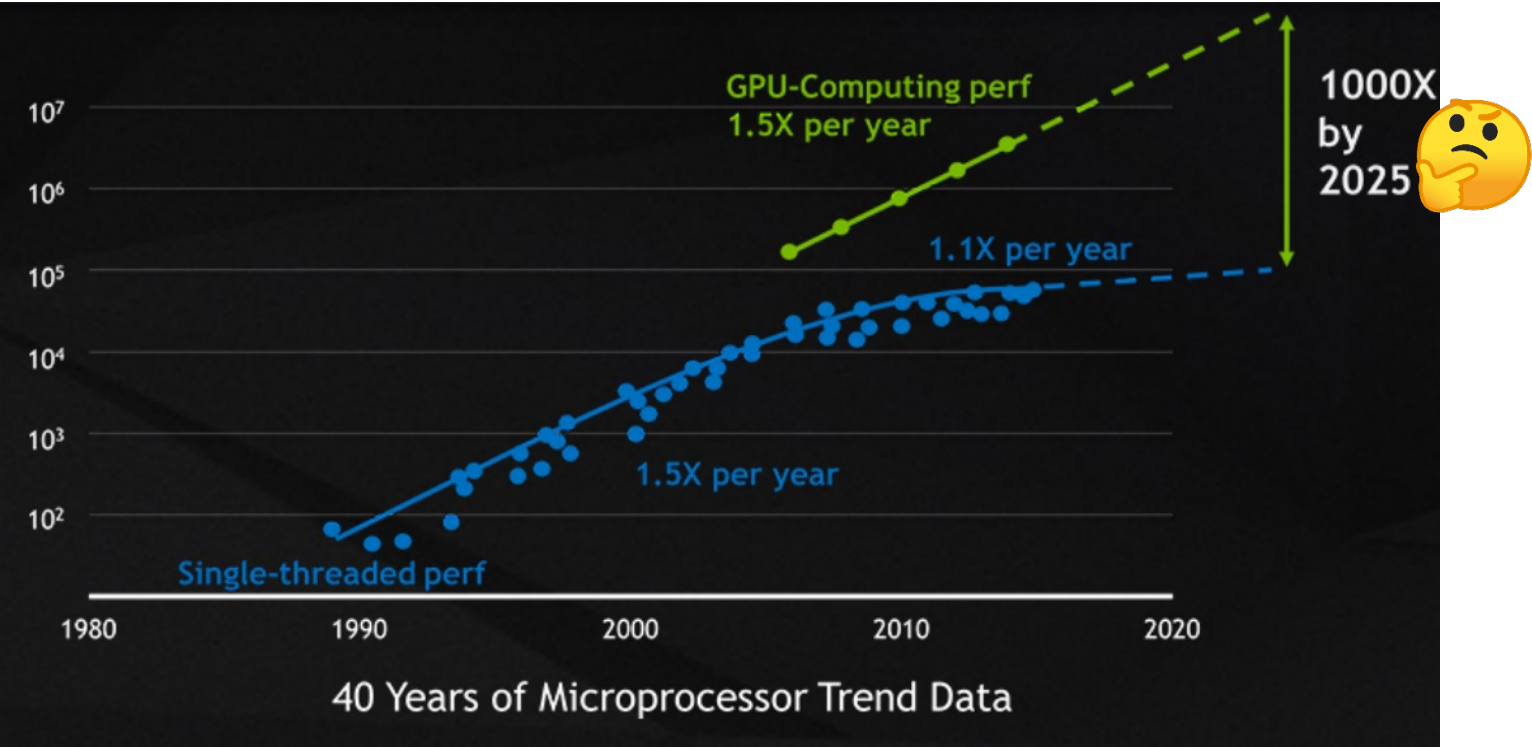
A Diamond Monster 3D, using the Voodoo chipset (1997)
(Konstantin Lanzet, Wikipedia)



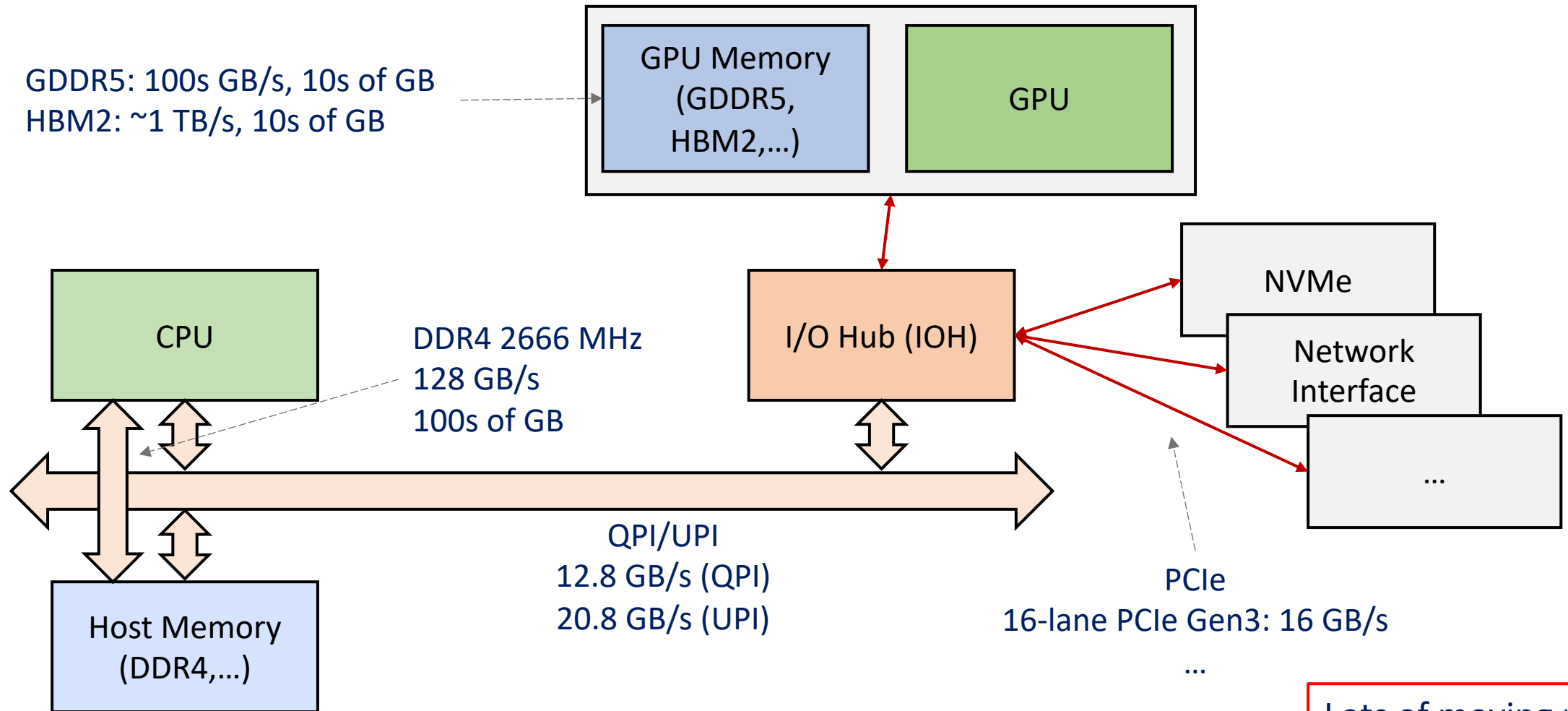
Peak Performance vs. CPU

	Throughput	Power	Throughput/Power
Intel Skylake	128 SP GFLOPS/4 Cores	100+ Watts	~1 GFLOPS/Watt
NVIDIA V100	15 TFLOPS	200+ Watts	~75 GFLOPS/Watt

Also,



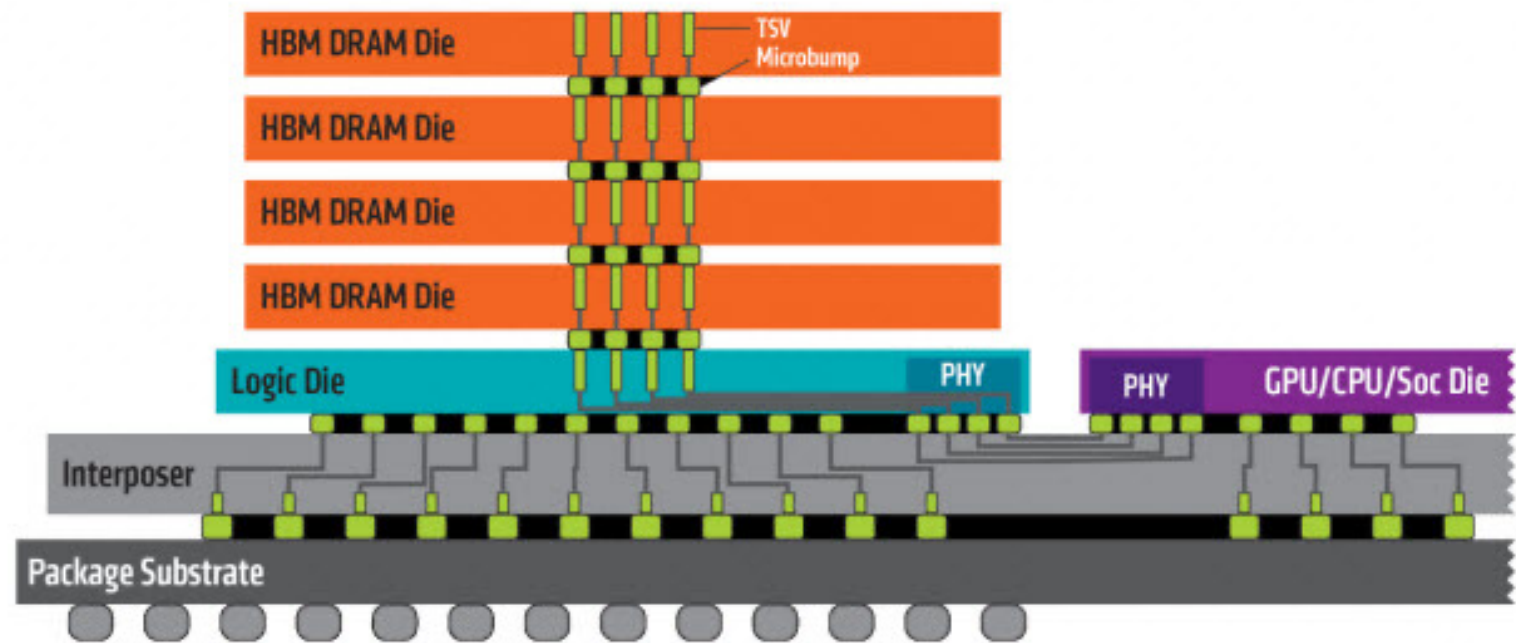
System Architecture Snapshot With a GPU (2019)



Lots of moving parts!

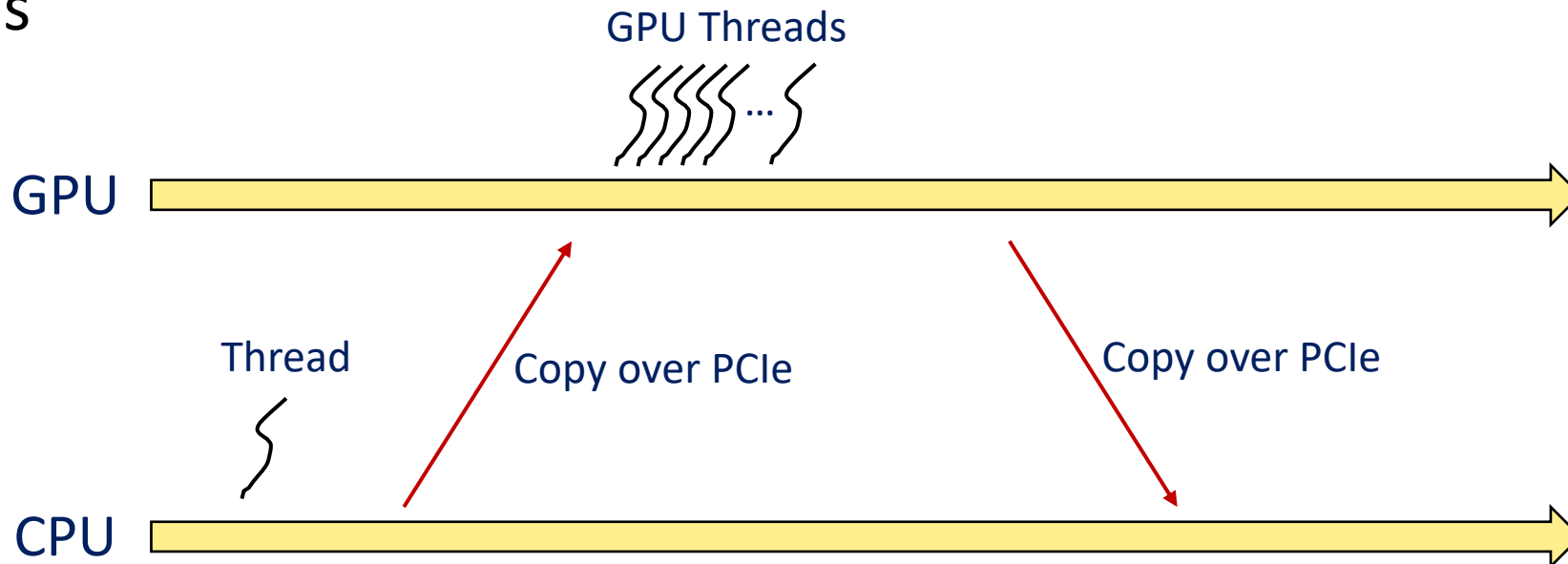
High-Performance Graphics Memory

- ❑ Modern GPUs even employing 3D-stacked memory via silicon interposer
 - Very wide bus, very high bandwidth
 - e.g., HBM2 in Volta



Massively Parallel Architecture For Massively Parallel Workloads!

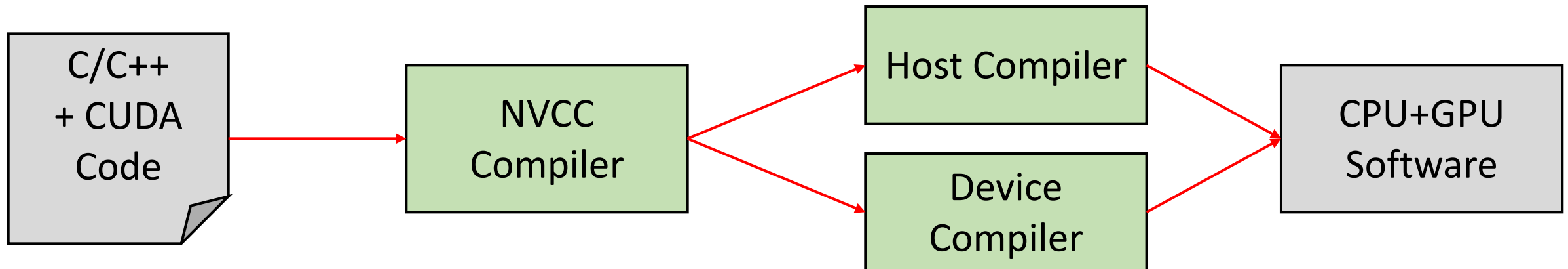
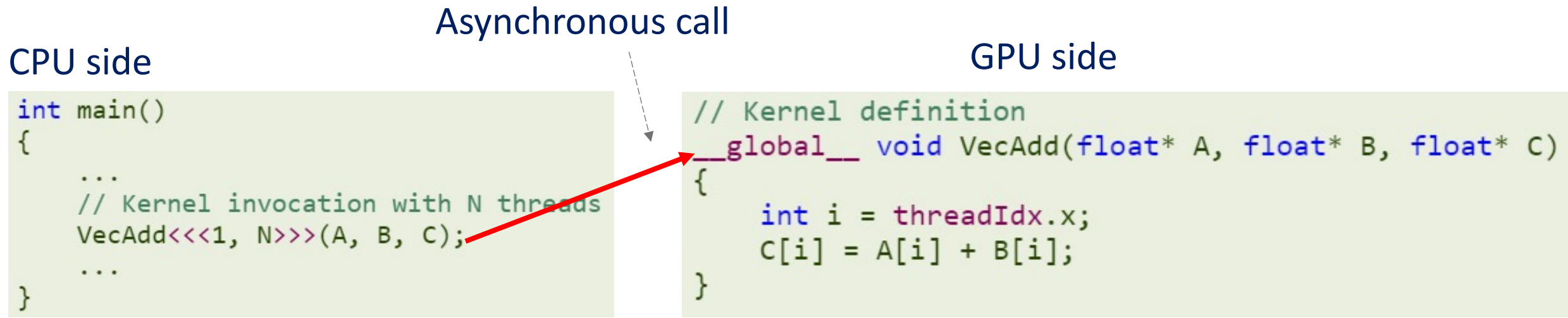
- ❑ NVIDIA CUDA (Compute Uniform Device Architecture) – 2007
 - A way to run custom programs on the massively parallel architecture!
- ❑ OpenCL specification released – 2008
- ❑ Both platforms expose synchronous execution of a massive number of threads



CUDA Execution Abstraction

- ❑ Block: Multi-dimensional array of threads
 - 1D, 2D, or 3D
 - Threads in a block can synchronize among themselves
 - Threads in a block can access shared memory
 - CUDA (Thread, Block) \sim OpenCL (Work item, Work group)
- ❑ Grid: Multi-dimensional array of blocks
 - 1D or 2D
 - Blocks in a grid can run in parallel, or sequentially
- ❑ Kernel execution issued in grid units
- ❑ Limited recursion (depth limit of 24 as of now)

Simple CUDA Example



Simple CUDA Example

```
int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
}
```

1 block

N threads per block

Should wait for kernel to finish

__global__:
In GPU, called from host/GPU

__device__:
In GPU, called from GPU

__host__:
In host, called from host

N instances of VecAdd spawned in GPU

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
```

Only void allowed

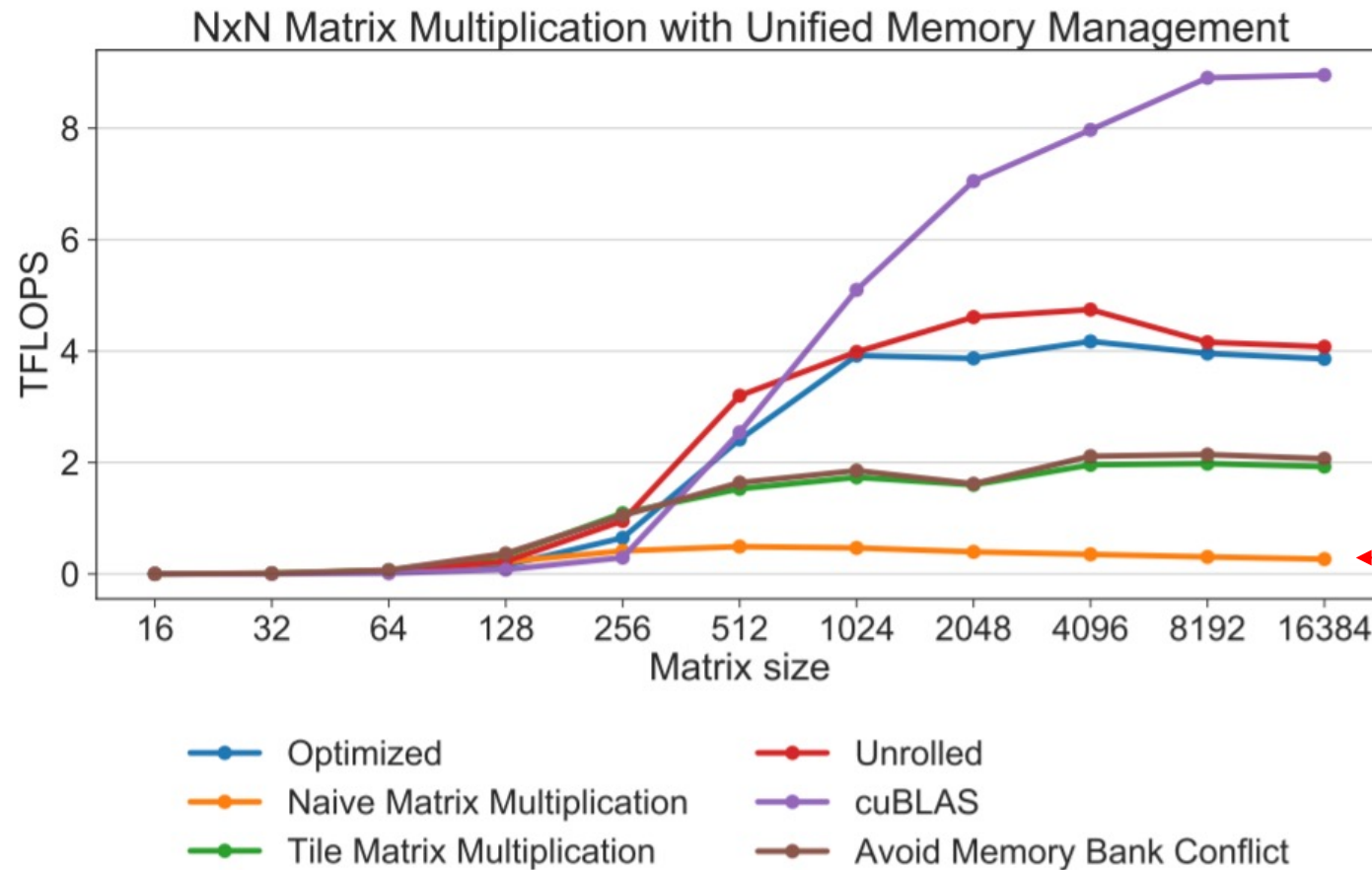
Which of N threads am I?
See also: blockIdx

One function can
be both

More Complex Example: Picture Blurring

- ❑ Slides from NVIDIA/UIUC Accelerated Computing Teaching Kit
- ❑ Another end-to-end example
<https://devblogs.nvidia.com/even-easier-introduction-cuda/>
- ❑ Great! Now we know how to use GPUs – Bye?

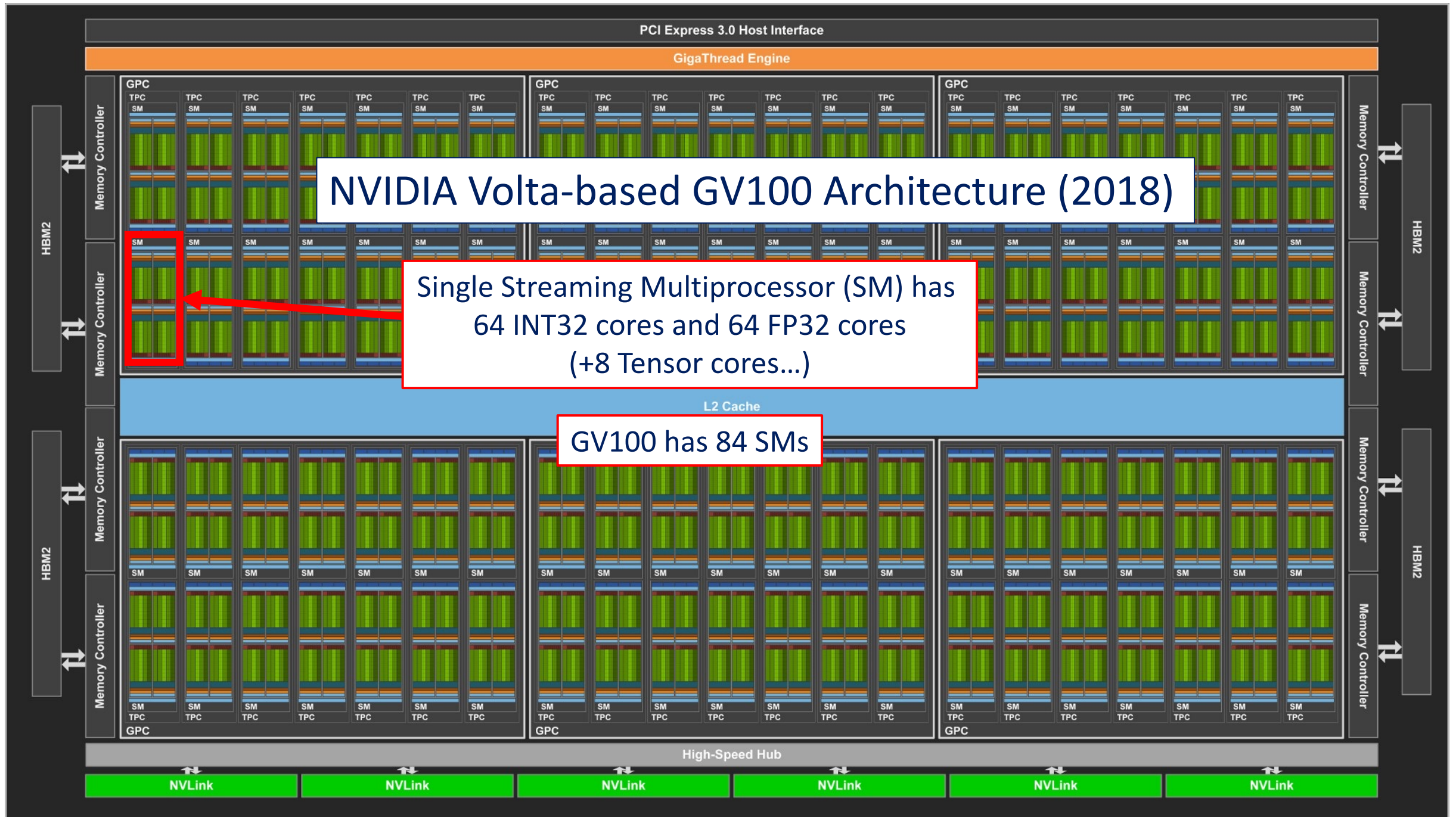
Matrix Multiplication Performance Engineering



No faster than CPU

Results from NVIDIA P100

Architecture knowledge is needed (again)



Volta Execution Architecture

- ❑ 64 INT32 Cores, 64 FP32 Cores, 4 Tensor Cores, Ray-tracing cores..
 - Specialization to make use of chip space...?
- ❑ Not much on-chip memory per thread
 - 96 KB Shared memory
 - 1024 Registers per FP32 core
- ❑ Hard limit on compute management
 - 32 blocks AND 2048 threads AND 1024 threads/block
 - e.g., 2 blocks with 1024 threads, or 4 blocks with 512 threads
 - Enough registers/shared memory for all threads must be available (all context is resident during execution)

More threads than cores – Threads interleaved to hide memory latency



Resource Balancing Details

- ❑ How many threads in a block?
- ❑ Too small: 4x4 window == 16 threads
 - 128 blocks to fill 2048 thread/SM
 - SM only supports 32 blocks -> only 512 threads used
 - SM has only 64 cores... does it matter? Sometimes!
- ❑ Too large: 32x48 window == 1536 threads
 - Threads do not fit in a block!
- ❑ Too large: 1024 threads using more than 64 registers
- ❑ Limitations vary across platforms (Fermi, Pascal, Volta, ...)

Warp Scheduling Unit

- ❑ Threads in a block are executed in 32-thread “warp” unit
 - Not part of language specs, just architecture specifics
 - A warp is SIMD – Same PC, same instructions executed on every core
- ❑ What happens when there is a conditional statement?
 - Prefix operations, or control divergence
 - More on this later!
- ❑ Warps have been 32-threads so far, but may change in the future

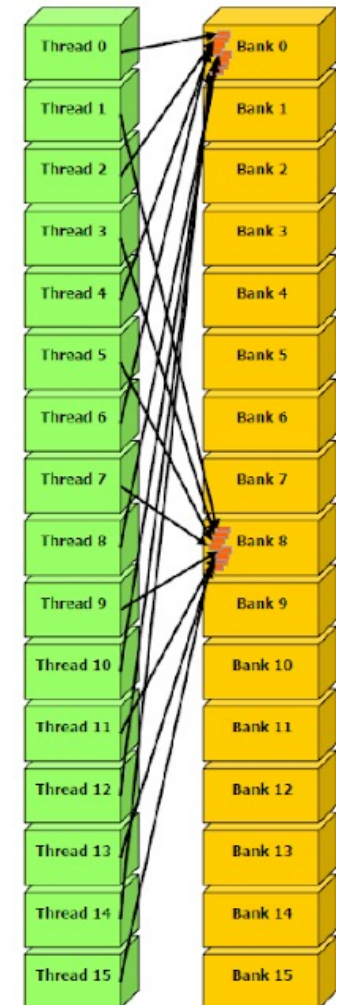
Memory Architecture Caveats

❑ Shared memory peculiarities

- Small amount (e.g., 96 KB/SM for Volta) shared across all threads
- Organized into banks to distribute access
- Bank conflicts can drastically lower performance

❑ Relatively slow global memory

- Blocking, caching becomes important (again)
- If not for performance, for power consumption...

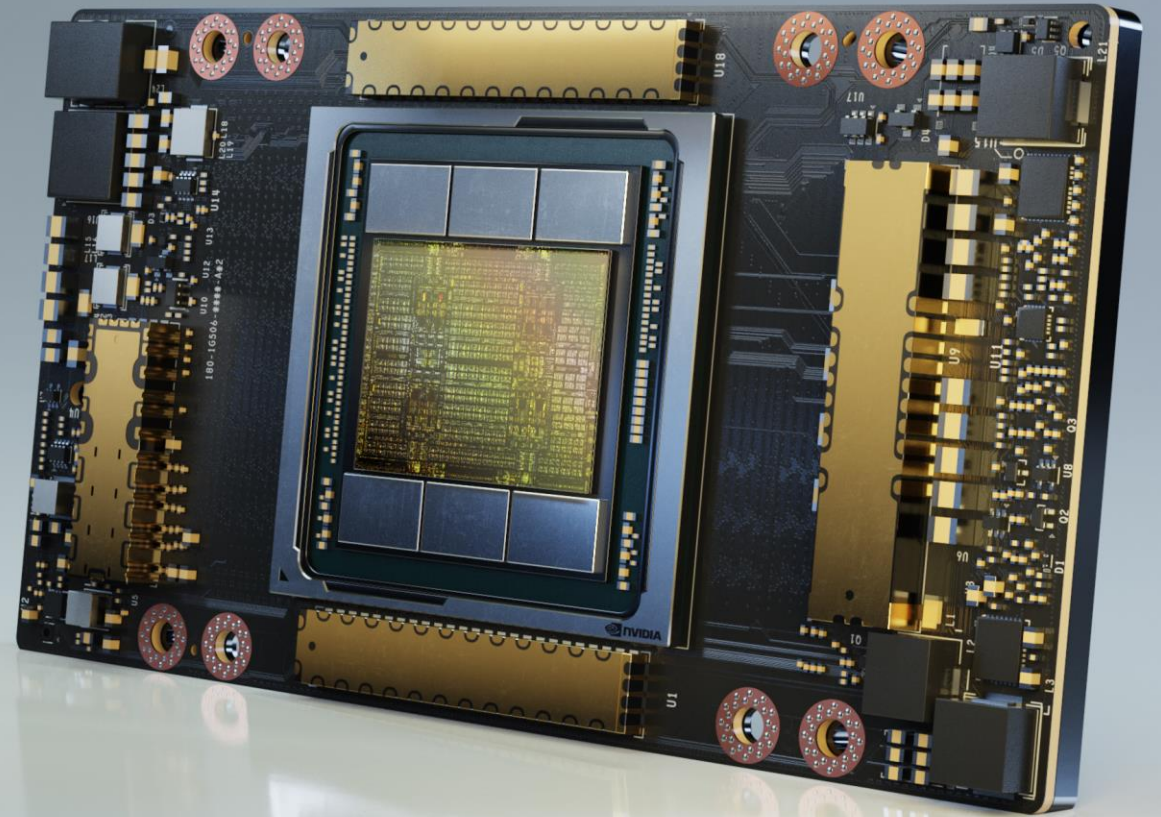


8-way bank conflict
1/8 memory bandwidth

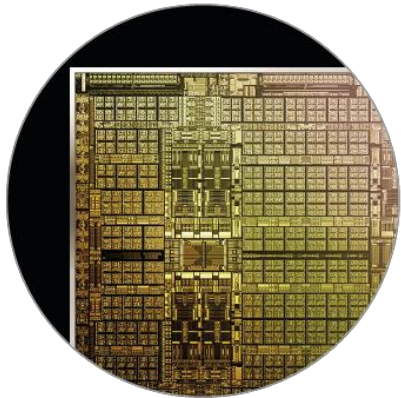
Inside the NVIDIA Ampere Architecture

Ronny Krashinsky, Olivier Giroux
GPU Architects

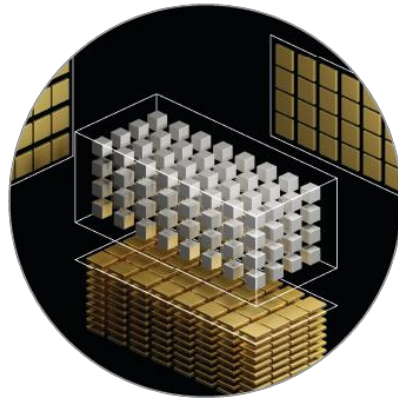
GTC 2020



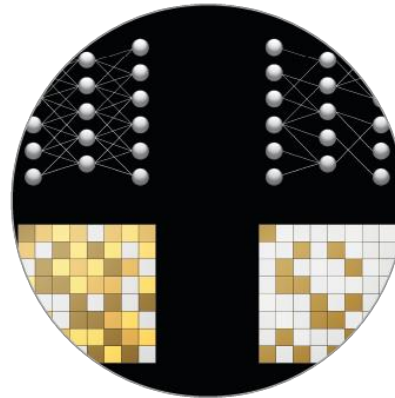
UNPRECEDENTED ACCELERATION AT EVERY SCALE



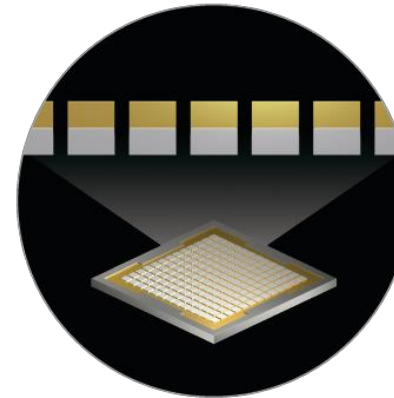
54 BILLION XTORS



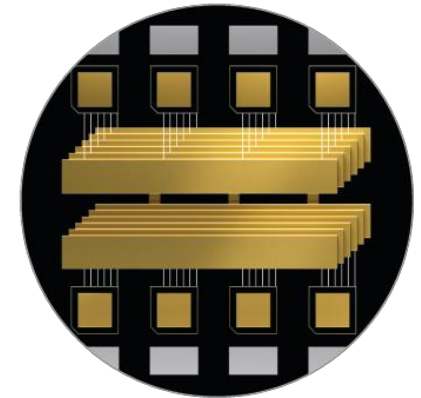
**3rd GEN
TENSOR CORES**



**SPARSITY
ACCELERATION**



MIG



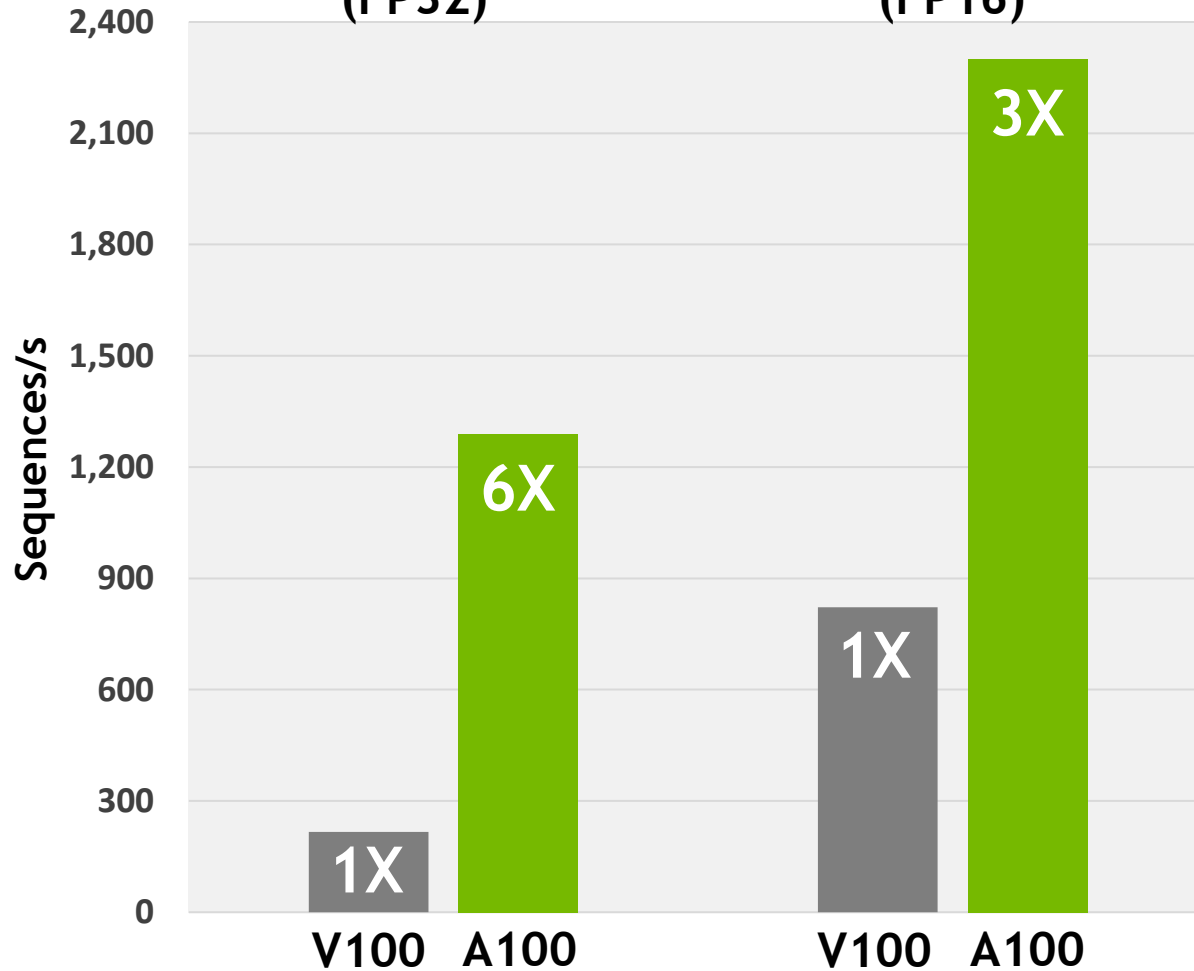
**3rd GEN
NVLINK & NVSWITCH**

UNIFIED AI ACCELERATION

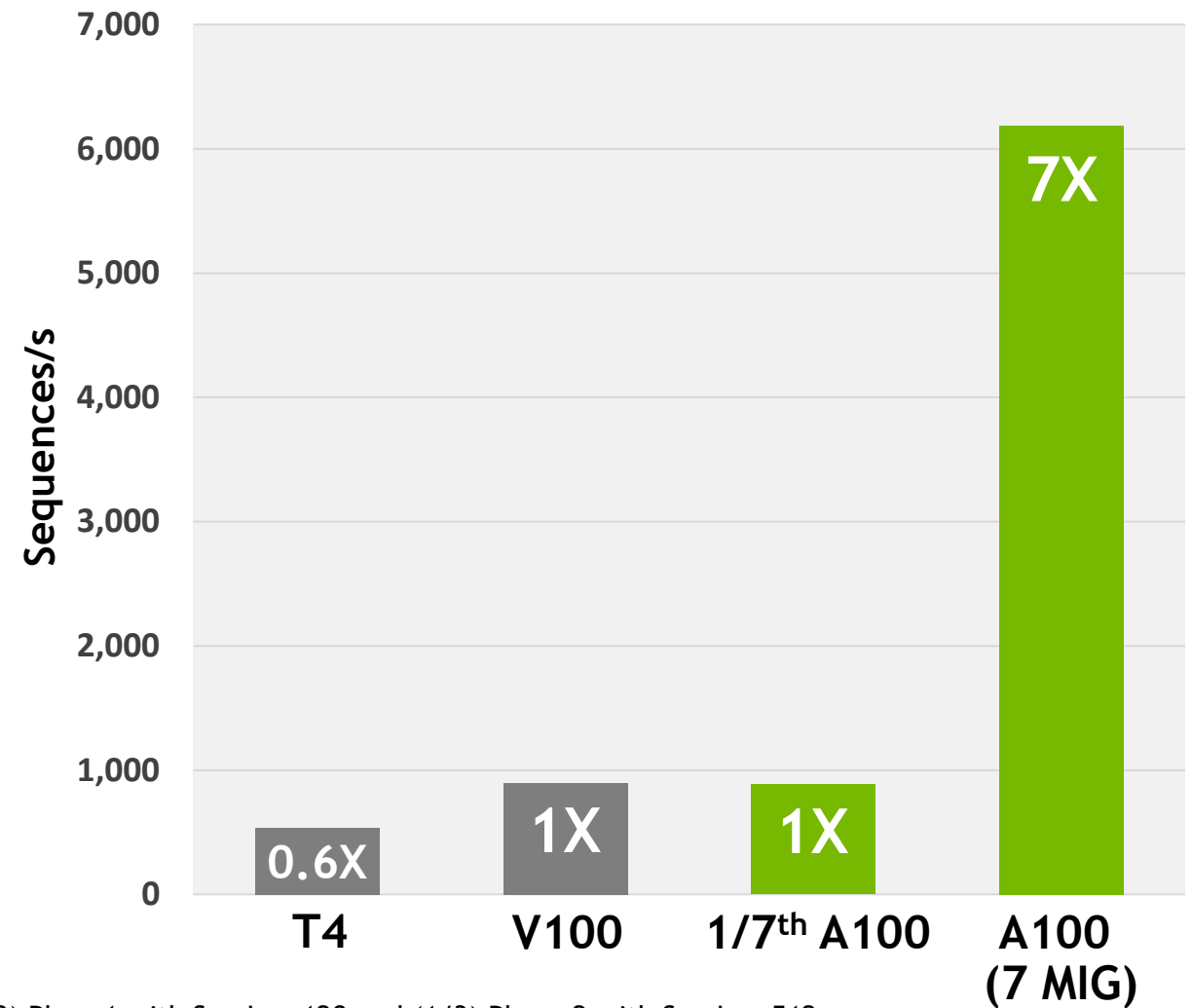
BERT-LARGE TRAINING

(FP32)

(FP16)

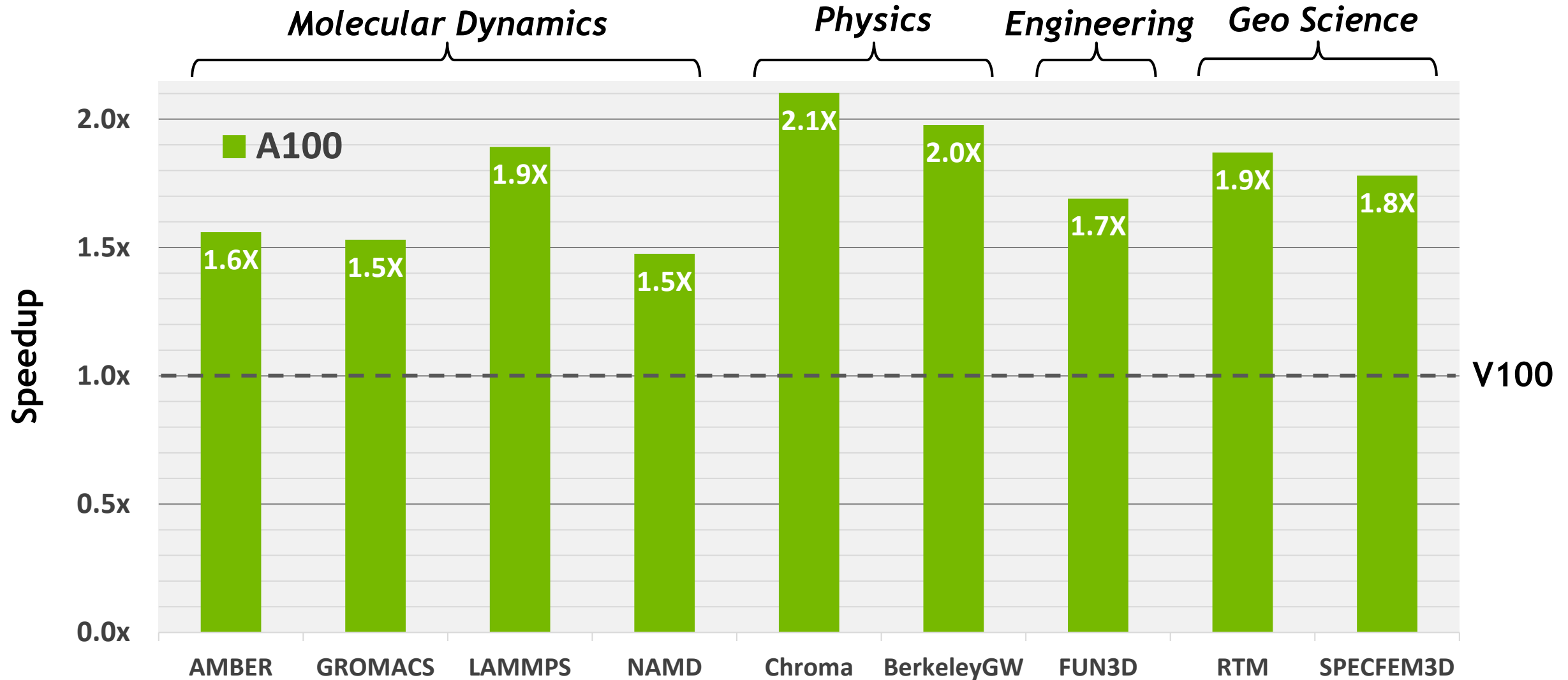


BERT-LARGE INFERENCE



All results are measured
BERT Large Training (FP32 & FP16) measures Pre-Training phase, uses PyTorch including (2/3) Phase1 with Seq Len 128 and (1/3) Phase 2 with Seq Len 512,
V100 is DGX1 Server with 8xV100, A100 is DGX A100 Server with 8xA100, A100 uses TF32 Tensor Core for FP32 training
BERT Large Inference uses TRT 7.1 for T4/V100, with INT8/FP16 at batch size 256. Pre-production TRT for A100, uses batch size 94 and INT8 with sparsity

ACCELERATING HPC



All results are measured

Except BerkeleyGW, V100 used is single V100 SXM2. A100 used is single A100 SXM4

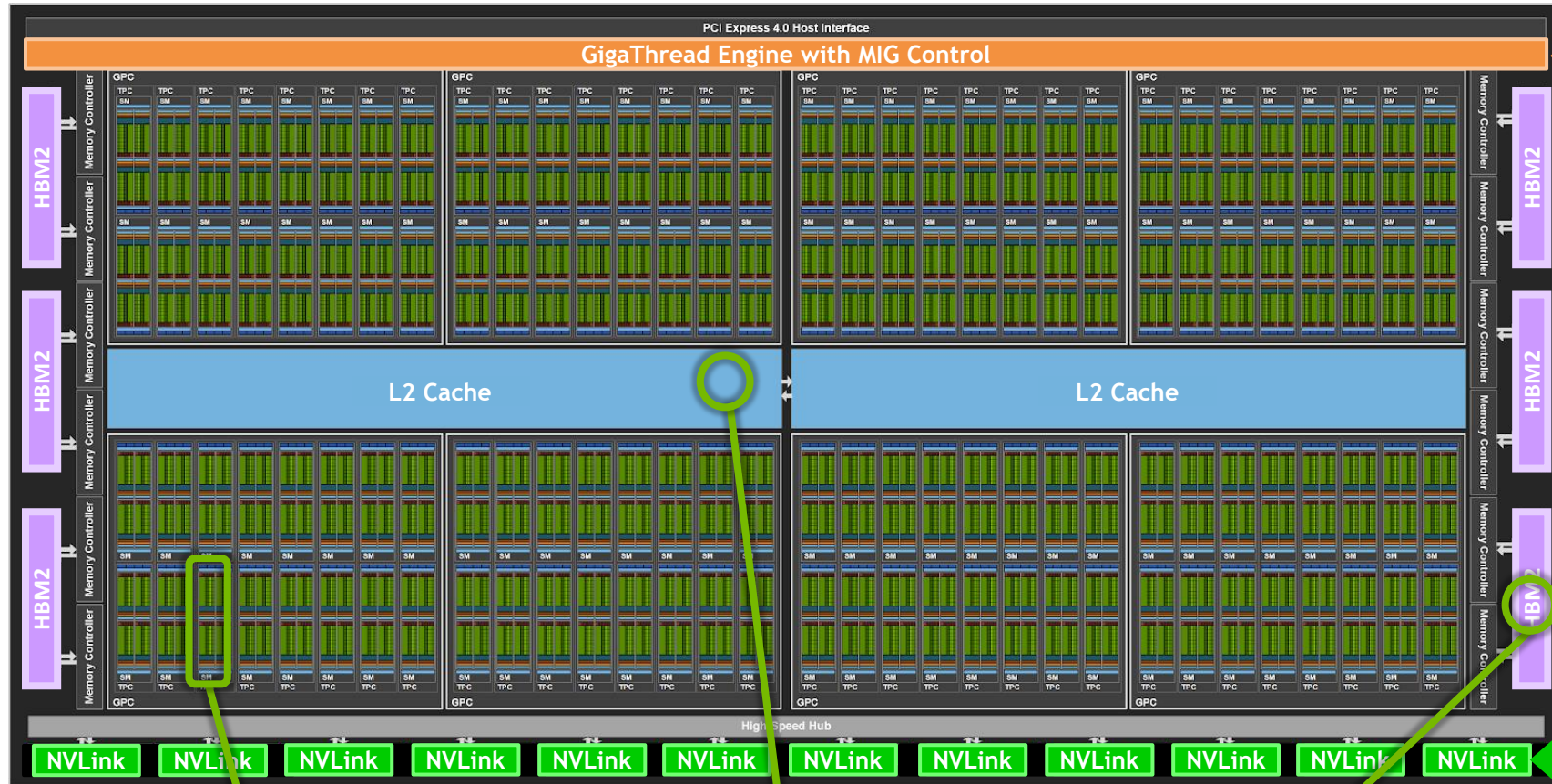
More apps detail: AMBER based on PME-Cellulose, GROMACS with STMV (h-bond), LAMMPS with Atomic Fluid LJ-2.5, NAMD with v3.0a1 STMV_NVE

Chroma with szscl21_24_128, FUN3D with dpw, RTM with Isotropic Radius 4 1024³, SPECFEM3D with Cartesian four material model

BerkeleyGW based on Chi Sum and uses 8xV100 in DGX-1, vs 8xA100 in DGX A100

A100 TENSOR-CORE GPU

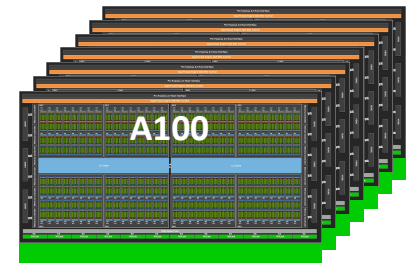
54 billion transistors in 7nm



Scale OUT

7x

Multi-Instance GPU



2x BW

Scale UP

3rd gen.
NVLINK

108 SMs
6912 CUDA Cores

40MB L2
6.7x capacity

1.56 TB/s HBM2
1.7x bandwidth

A100 SM



Third-generation Tensor Core
Faster and more efficient
Comprehensive data types
Sparsity acceleration

**Asynchronous data movement
and synchronization**

Increased L1/SMEM capacity

- 
1. New Tensor Core
 2. Strong Scaling
 3. Elastic GPU
 4. Productivity







1. New Tensor Core

2. Strong Scaling

3. Elastic GPU









4. Productivity

V100 TENSOR CORE

	INPUT OPERANDS	ACCUMULATOR	TOPS	X-factor vs. FFMA
V100	FP32 	FP32 	15.7	1x
	FP16 	FP32 	125	8x

V100
125 8x vs.
TOPS FFMA
FP16/FP32
Mixed-
precision

A100 TENSOR CORE

	INPUT OPERANDS		ACCUMULATOR		TOPS	X-factor vs. FFMA
V100	FP32		FP32		15.7	1x
	FP16		FP32		125	8x
A100	FP32		FP32		19.5	1x
	FP16		FP32		312	16x

V100 → A100

2.5x

2x













TOPS

TOPS/

FF16/FP32
Mixed-
precision


















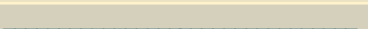

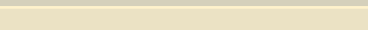
SM

A100 TENSOR CORE

	INPUT OPERANDS	ACCUMULATOR	TOPS	X-factor vs. FFMA
V100	FP32 	FP32 	15.7	1x
	FP16 	FP32 	125	8x
A100	FP32 	FP32 	19.5	1x
	TF32 	FP32 	156	8x
	FP16 	FP32 	312	16x
	BF16 	FP32 	312	16x

TF32 accelerates FP32 in/out data → 10x vs. V100 FP32
 BFloat16 (BF16) at same rate as FP16

A100 TENSOR CORE





















	INPUT OPERANDS	ACCUMULATOR	TOPS	X-factor vs. FFMA
V100	FP32 	FP32 	15.7	1x
	FP16 	FP32 	125	8x
A100	FP32 	FP32 	19.5	1x
	TF32 	FP32 	156	8x
	FP16 	FP32 	312	16x
	BF16 	FP32 	312	16x
	FP16 	FP16 	312	16x
	INT8 	INT32 	624	32x
	INT4 	INT32 	1248	64x
	BINARY 	INT32 	4992	256x

2x
2x
4x

TOPS
track
operand
width






















Inference data types

A100 TENSOR CORE

	INPUT OPERANDS	ACCUMULATOR	TOPS	X-factor vs. FFMA	SPARSE TOPS	SPARSE X-factor vs. FFMA
V100	FP32 	FP32 	15.7	1x	-	-
	FP16 	FP32 	125	8x	-	-
A100	FP32 	FP32 	19.5	1x	-	-
	TF32 	FP32 	156	8x	312	16x
	FP16 	FP32 	312	16x	624	32x
	BF16 	FP32 	312	16x	624	32x
	FP16 	FP16 	312	16x	624	32x
	INT8 	INT32 	624	32x	1248	64x
	INT4 	INT32 	1248	64x	2496	128x
	BINARY 	INT32 	4992	256x	-	-















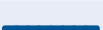
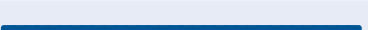





With Sparsity *another* 2x, INT8/INT4 reach *petaops*

A100 TENSOR CORE

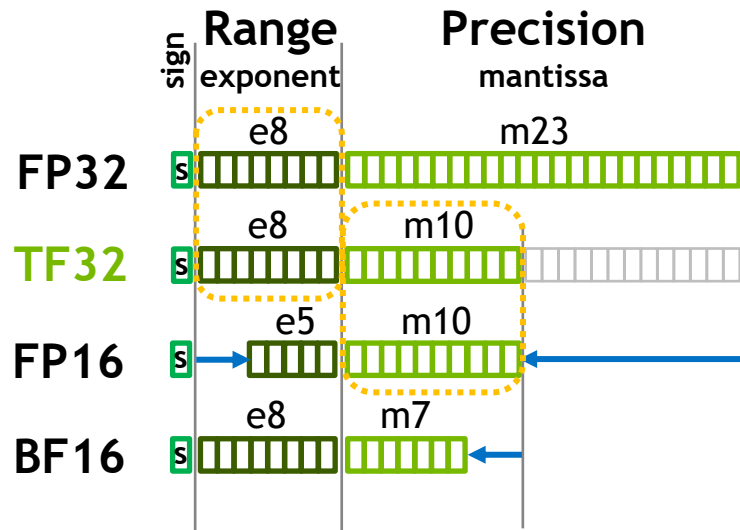
	INPUT OPERANDS	ACCUMULATOR	TOPS	X-factor vs. FFMA	SPARSE TOPS	SPARSE X-factor vs. FFMA
V100	FP32 	FP32 	15.7	1x	-	-
	FP16 	FP32 	125	8x	-	-
A100	FP32 	FP32 	19.5	1x	-	-
	TF32 	FP32 	156	8x	312	16x
	FP16 	FP32 	312	16x	624	32x
	BF16 	FP32 	312	16x	624	32x
	FP16 	FP16 	312	16x	624	32x
	INT8 	INT32 	624	32x	1248	64x
	INT4 	INT32 	1248	64x	2496	128x
	BINARY 	INT32 	4992	256x		
	IEEE FP64 		19.5	1x		

V100→A100
2.5x FLOPS
for HPC

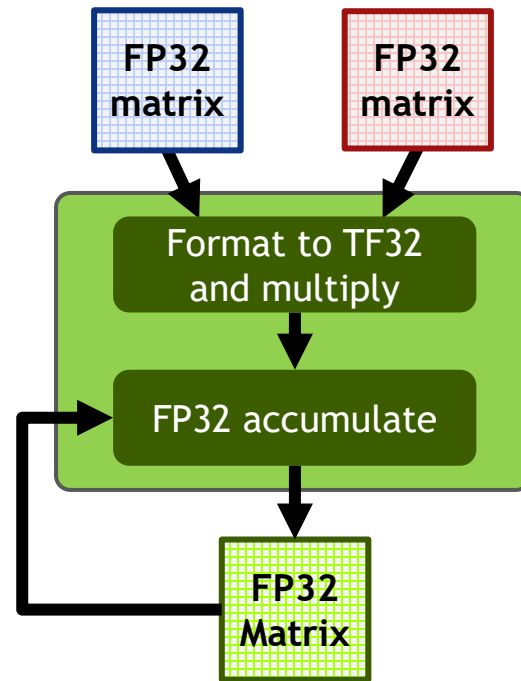
A100 TENSOR CORE

	INPUT OPERANDS	ACCUMULATOR	TOPS	X-factor vs. FFMA	SPARSE TOPS	SPARSE X-factor vs. FFMA
V100	FP32 	FP32 	15.7	1x	-	-
	FP16 	FP32 	125	8x	-	-
A100	FP32 	FP32 	19.5	1x	-	-
	TF32 	FP32 	156	8x	312	16x
	FP16 	FP32 	312	16x	624	32x
	BF16 	FP32 	312	16x	624	32x
	FP16 	FP16 	312	16x	624	32x
	INT8 	INT32 	624	32x	1248	64x
	INT4 	INT32 	1248	64x	2496	128x
	BINARY 	INT32 	4992	256x	-	-
	IEEE FP64 		19.5	1x	-	-

INSIDE A100 TensorFloat-32 (TF32)



Range of FP32 with precision of FP16



FP32 input/output
FP32 storage and math for all activations, gradients, ...
everything outside tensor cores

Out-of-the-box
tensor core
acceleration for DL

Easy step towards maximizing tensor core performance with mixed-precision (FP16, BF16)

Up to 4x speedup
on linear solvers
for HPC

→S22082: Mixed-Precision Training of Neural Networks, 5/20 2:45pm PDT

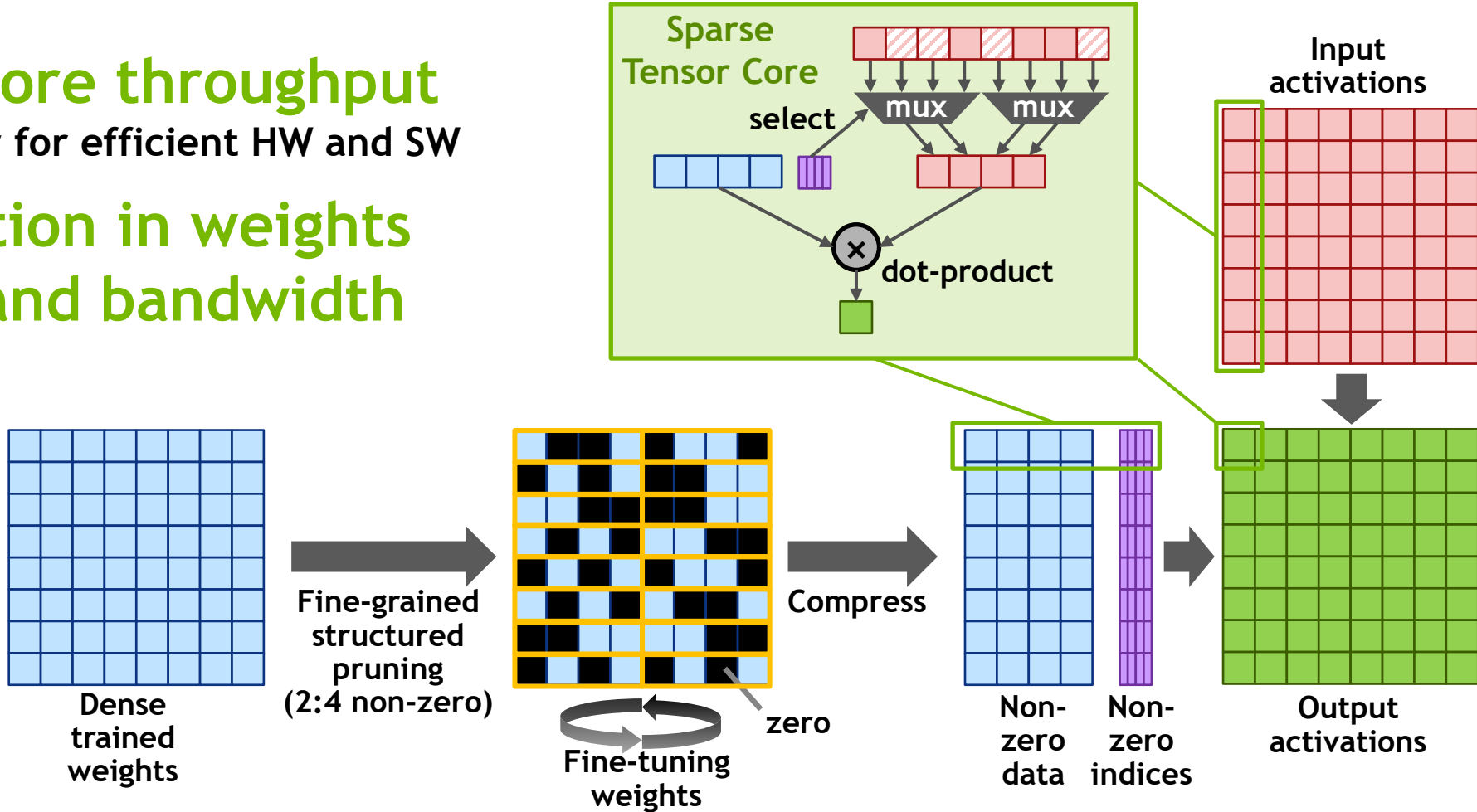
→S21681: How CUDA Math Libraries can help you unleash the power of the new NVIDIA A100 GPU (recording available)

INSIDE A100 SPARSE TENSOR CORE

2x Tensor Core throughput


Structured-sparsity for efficient HW and SW

~2x reduction in weights footprint and bandwidth



~No loss in inferencing accuracy

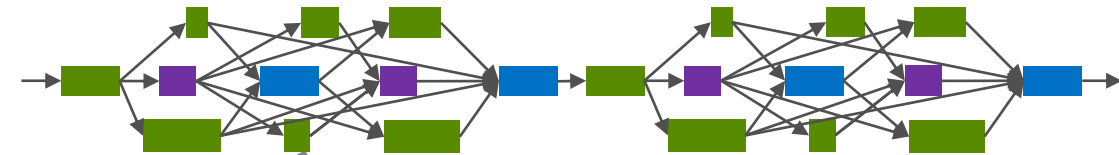
Evaluated across dozens of networks: vision, object detection, segmentation, natural language modeling, translation

- 
1. New Tensor Core
 2. Strong Scaling
 3. Elastic GPU
 4. Productivity

DL STRONG SCALING

DL networks:

Long chains of sequentially-dependent compute-intensive layers



1 layer

Weights

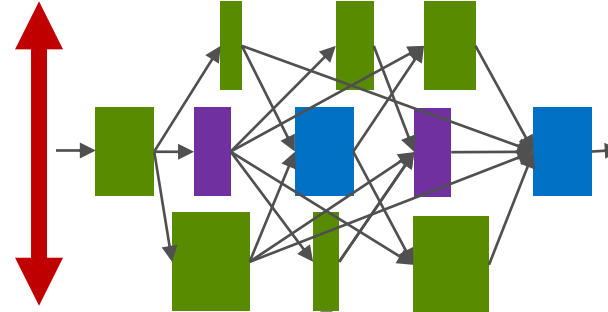
Each layer is parallelized across GPU

Tile: work for 1 SM

Input Activations

Output Activations

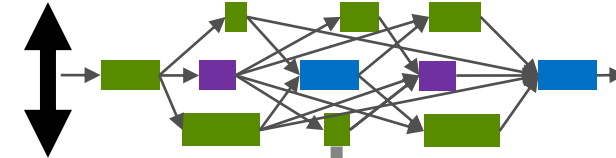
Weak scaling



Output Activations

~2.5x larger network runs in same time

Strong scaling

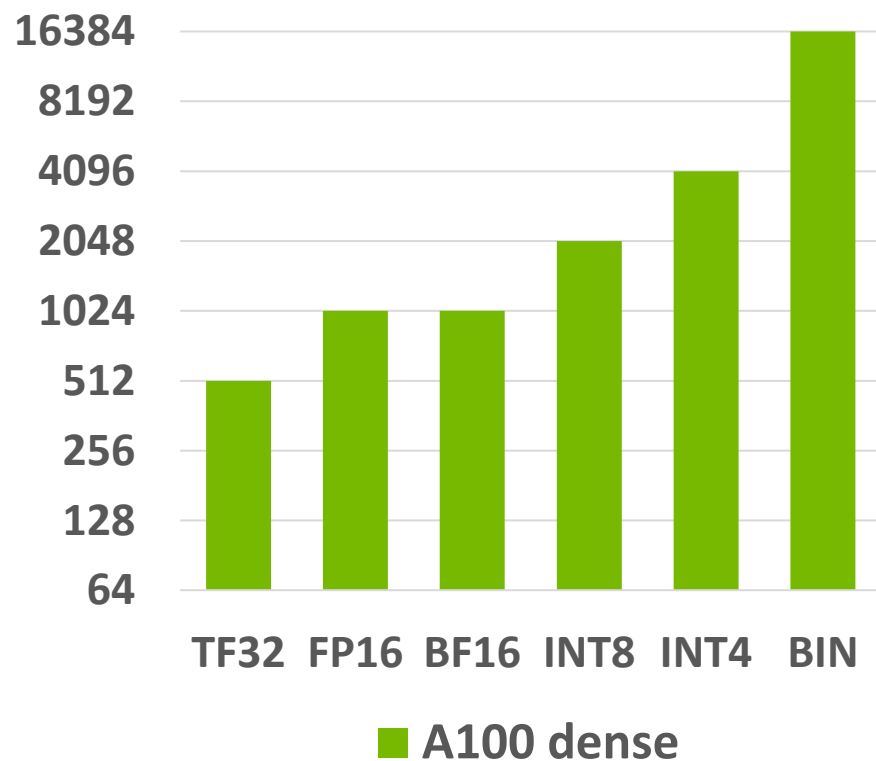


Output Activations

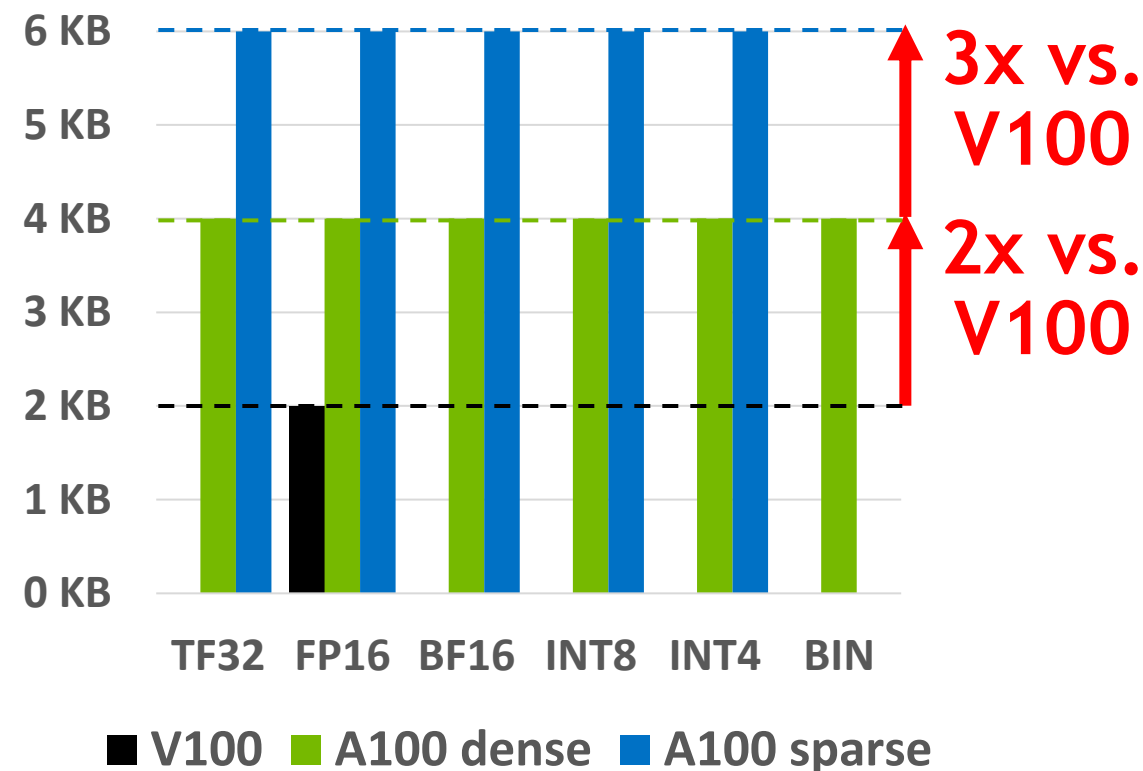
Fixed network runs ~2.5x faster

HOW TO KEEP TENSOR CORES FED?

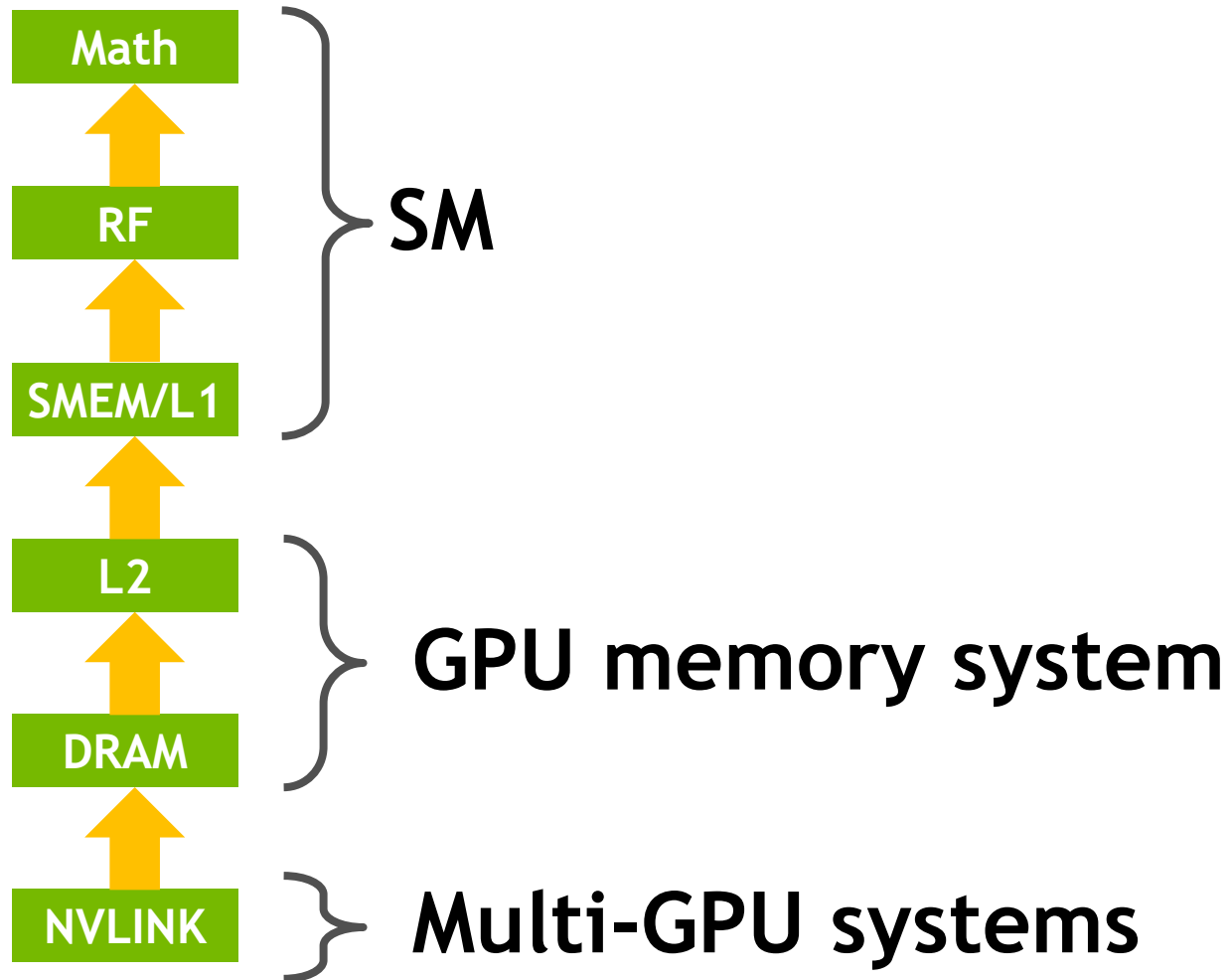
Math bandwidth
(MACs/clock/SM)



Required
data bandwidth
(A+B operands, B/clock/SM)



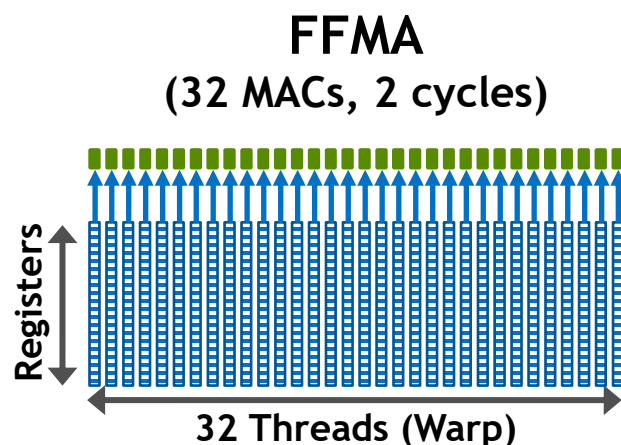
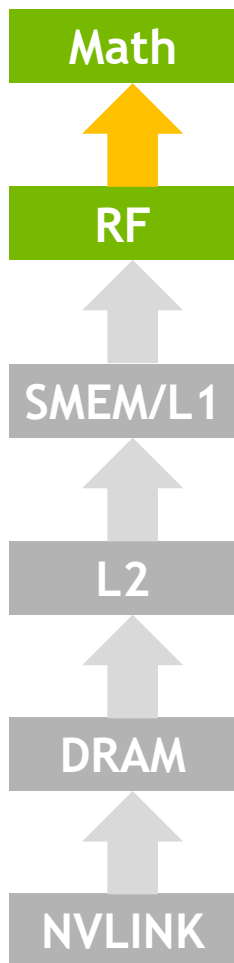
A100 STRONG SCALING INNOVATIONS



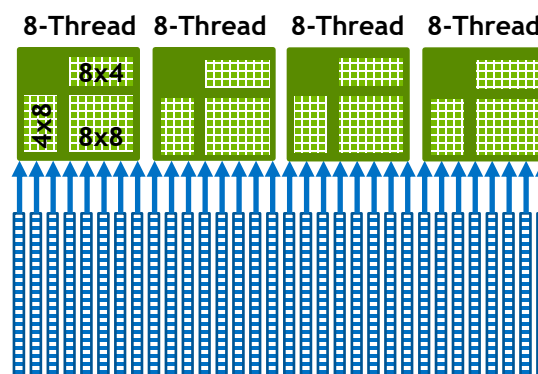
Improve speeds & feeds
and efficiency across all
levels of compute and
memory hierarchy

A100 TENSOR CORE

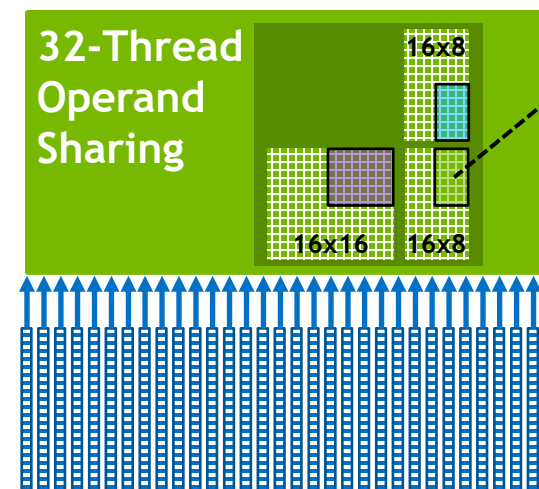
2x throughput vs. V100, >2x efficiency



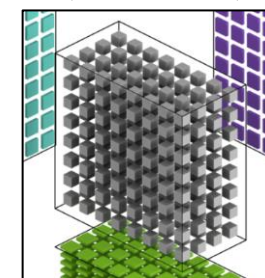
V100 TC Instruction
(1024 MACs, 8 cycles)



A100 TC Instruction
(2048 MACs, 8 cycles)



A100 TC
(1 cycle)

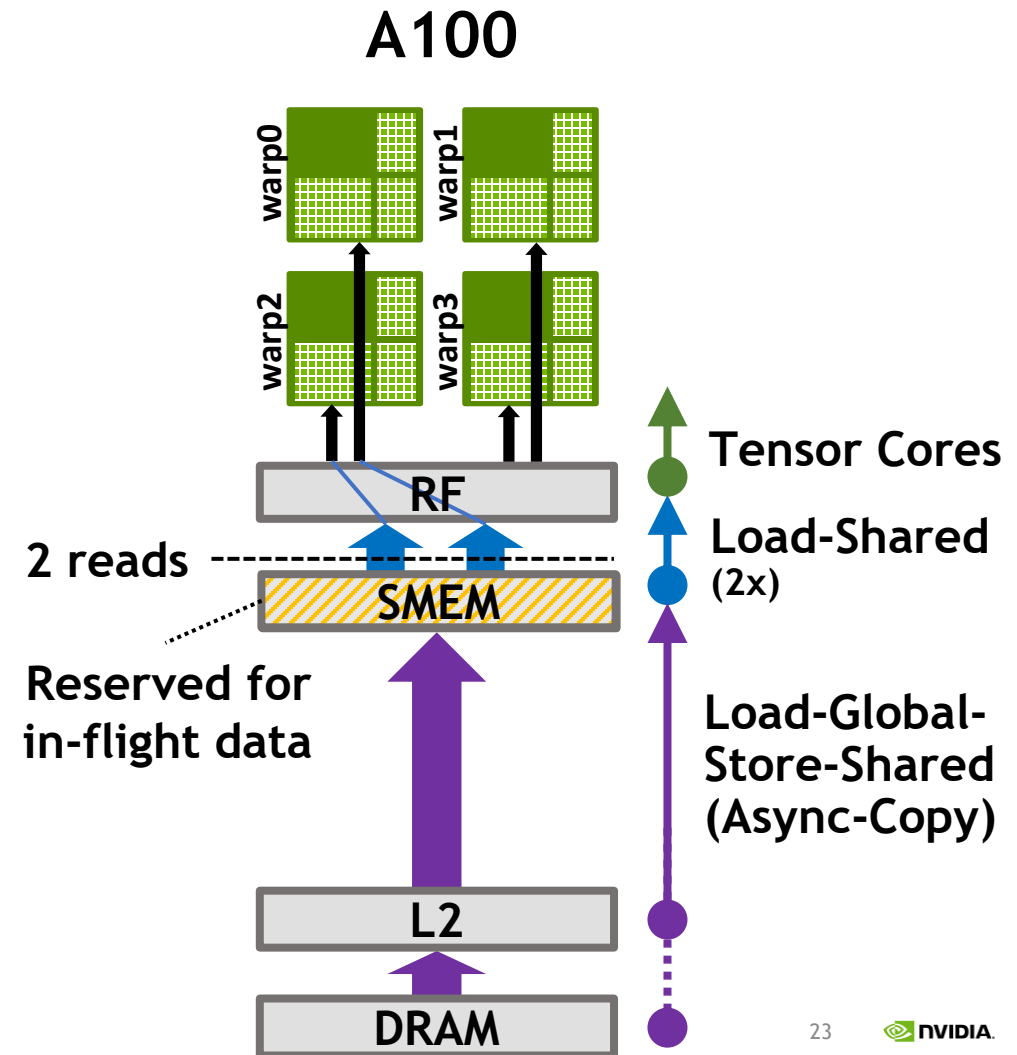
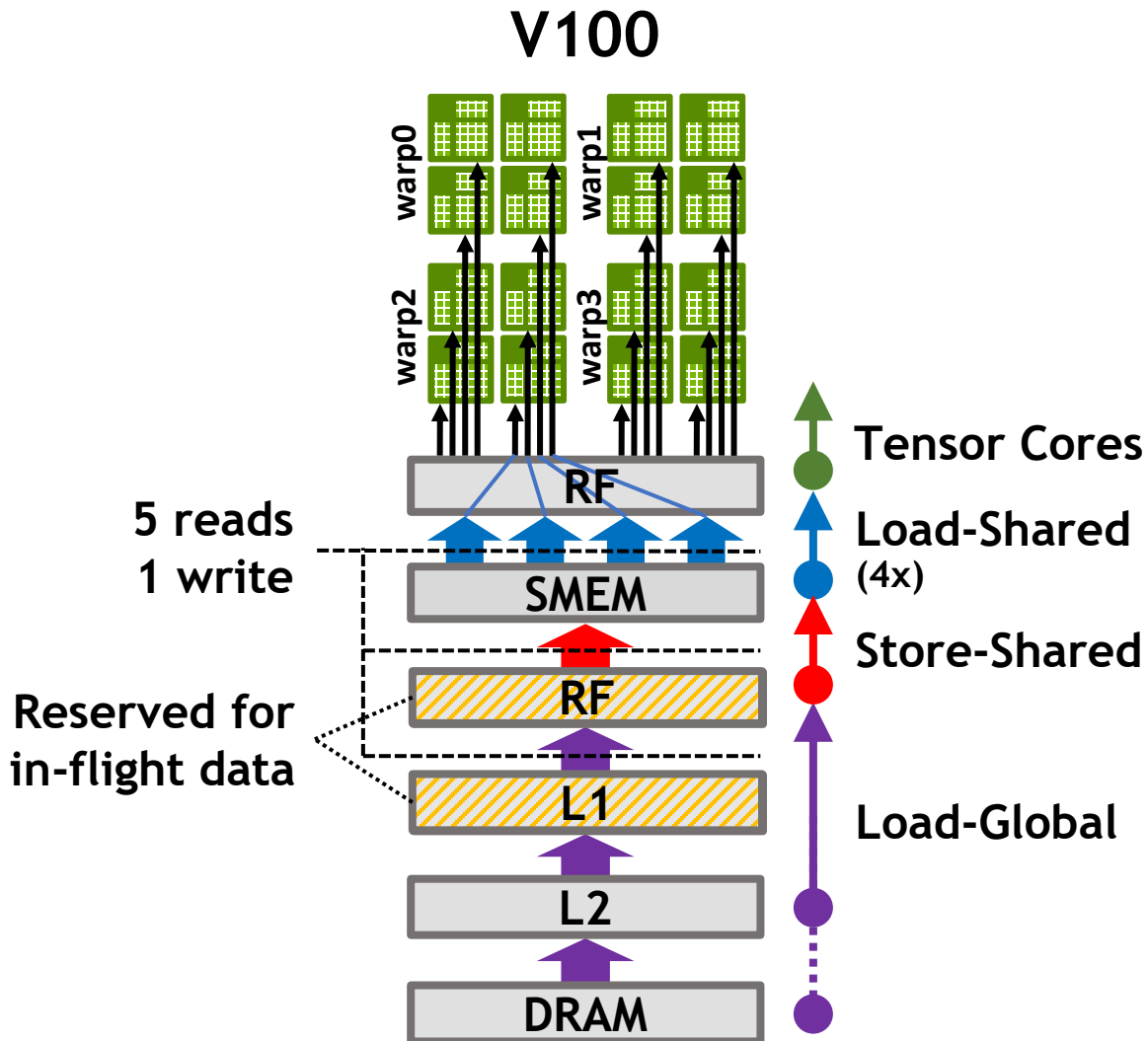
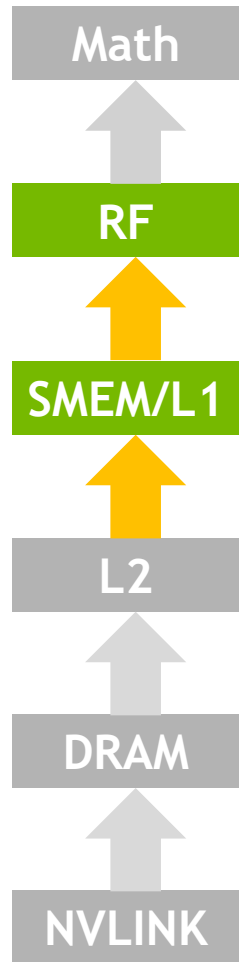


16x16x16 matrix multiply	FFMA	V100 TC	A100 TC	A100 vs. V100 (improvement)	A100 vs. FFMA (improvement)
Thread sharing	1	8	32	4x	32x
Hardware instructions	128	16	2	8x	64x
Register reads+writes (warp)	512	80	28	2.9x	18x
Cycles	256	32	16	2x	16x

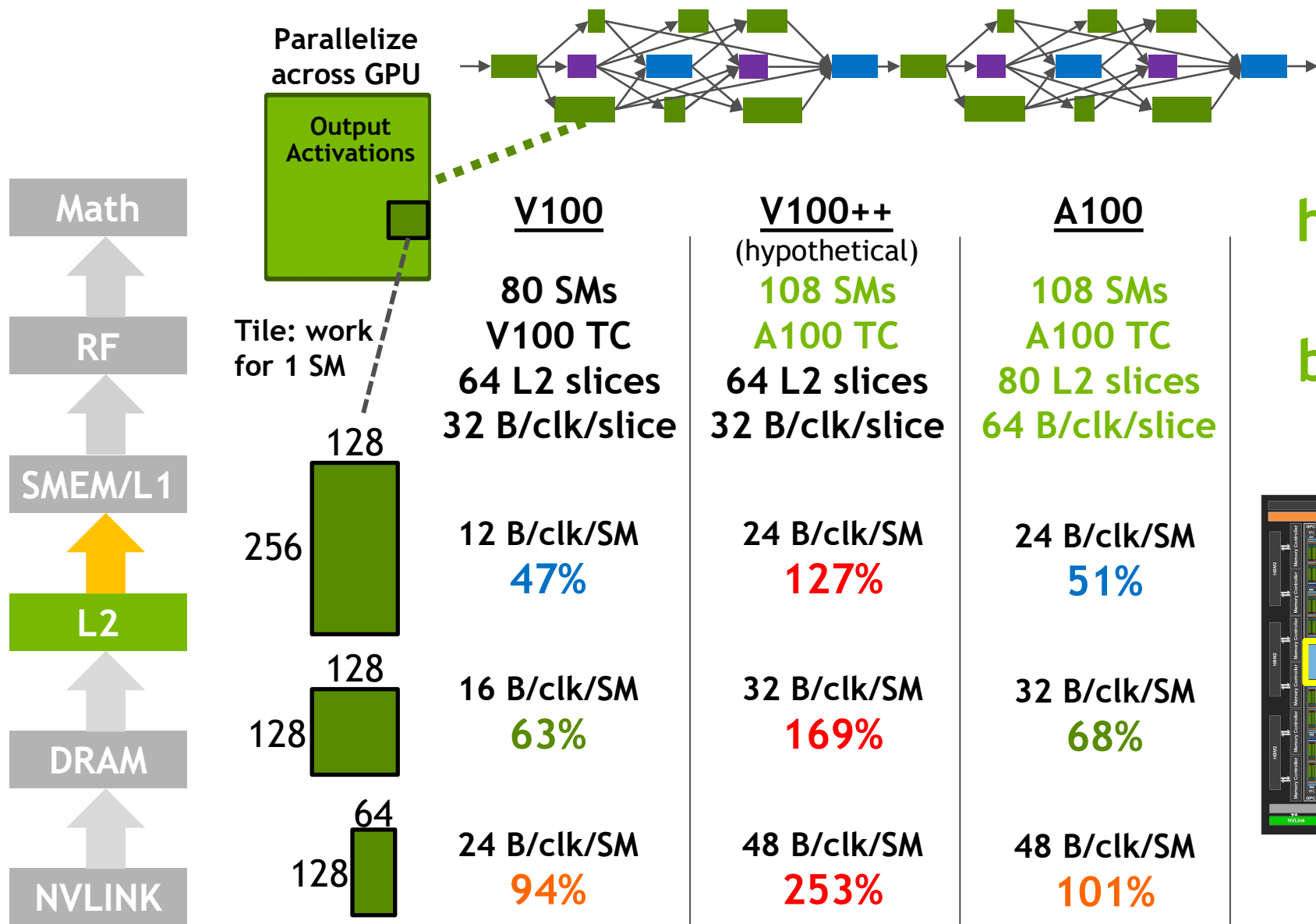
Tensor Cores assume FP16 inputs with FP32 accumulator, V100 Tensor Core instruction uses 4 hardware instructions

A100 SM DATA MOVEMENT EFFICIENCY

3x SMEM/L1 bandwidth, 2x in-flight capacity



A100 L2 BANDWIDTH



Split L2 with hierarchical crossbar - 2.3x increase in bandwidth over V100, lower latency



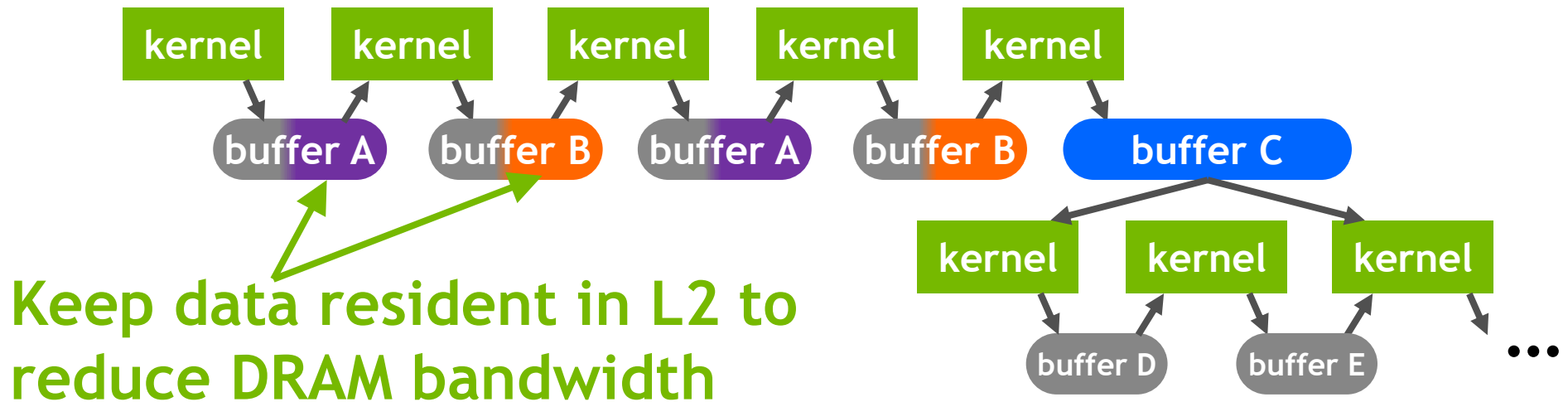
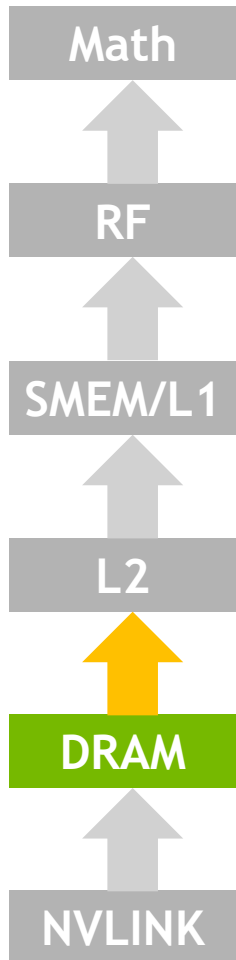
A100 DRAM BANDWIDTH

Faster HBM2

25% more pins, 38% faster clocks
→ 1.6 TB/s, **1.7x** vs. V100

Larger and smarter L2

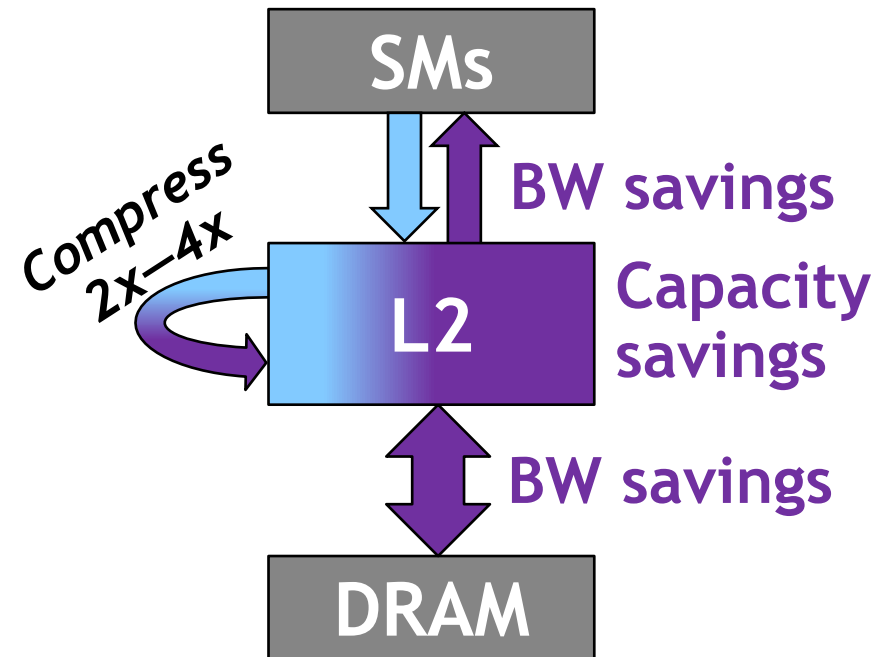
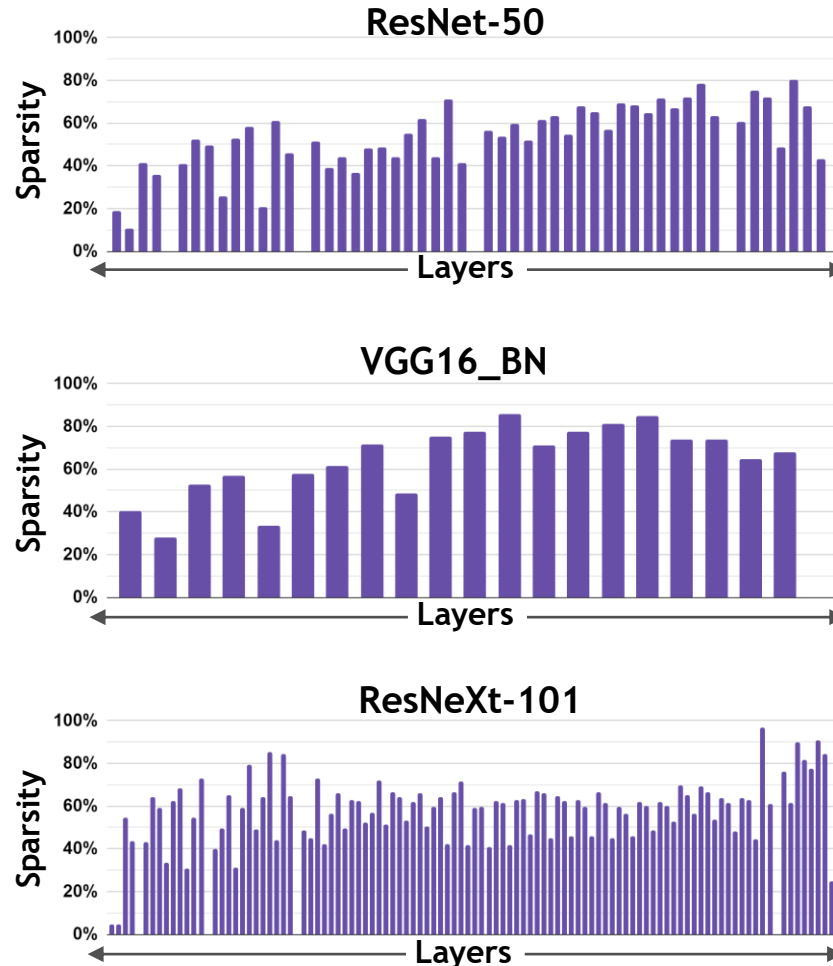
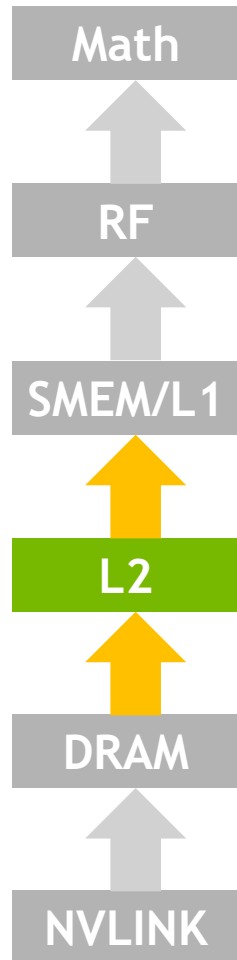
40MB L2, **6.7x** vs. V100
L2-Residency controls



A100 COMPUTE DATA COMPRESSION

Activation sparsity due to ReLU

Up to 4x DRAM+L2 bandwidth
and 2x L2 capacity
for fine-grained
unstructured sparsity



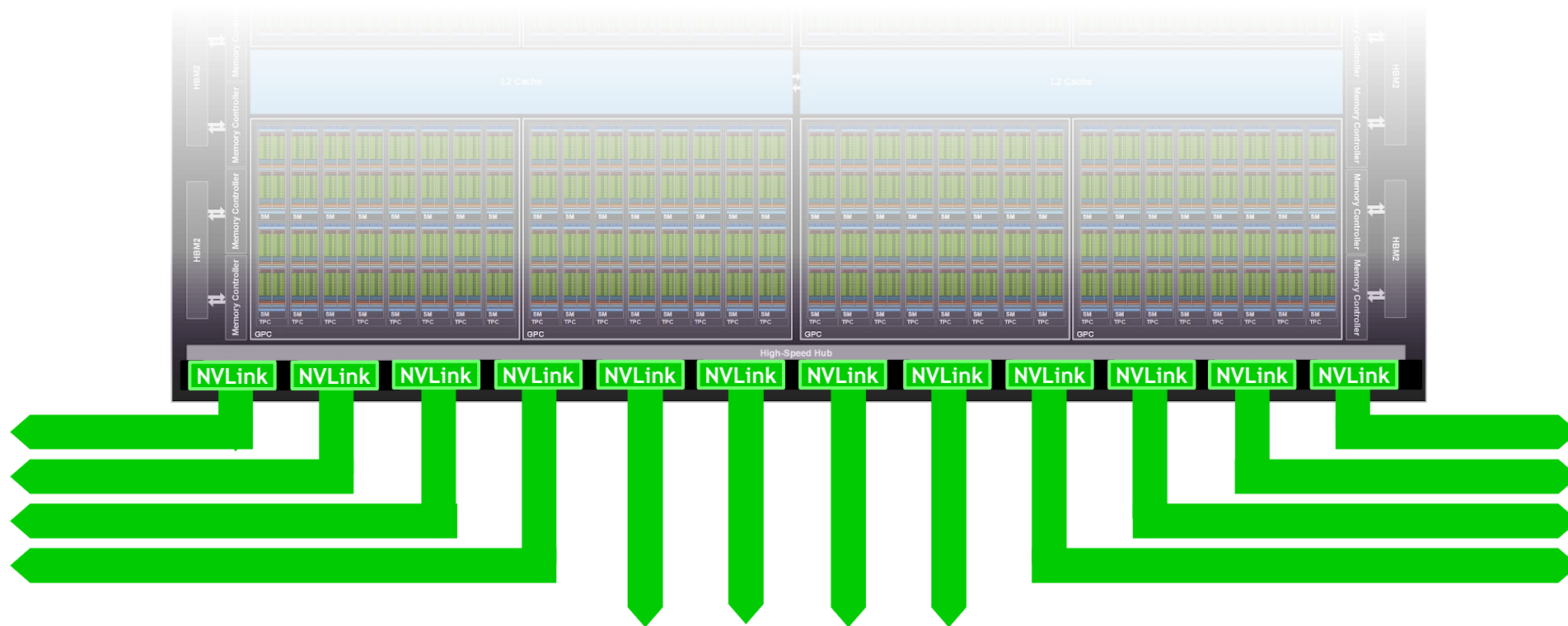
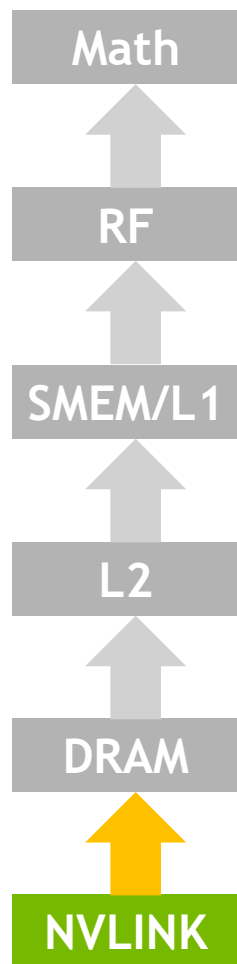
A100 NVLINK BANDWIDTH

Third Generation NVLink

50 Gbit/sec per signal pair

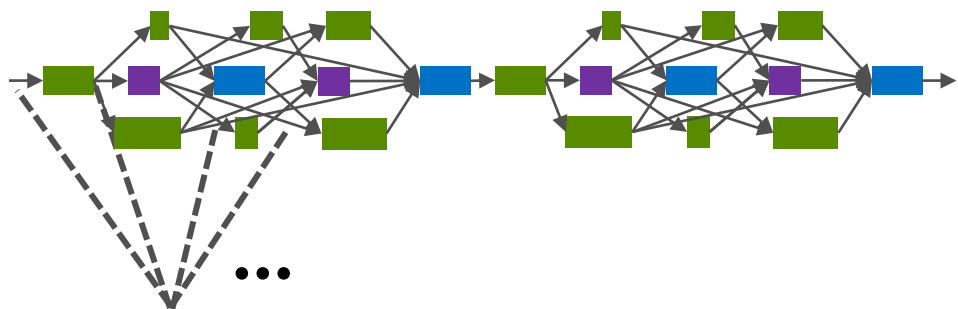
12 links, 25 GB/s in/out, 600 GB/s total

2x vs. V100



→S21884: Under the Hood of the new DGX A100 System Architecture (recording available soon)

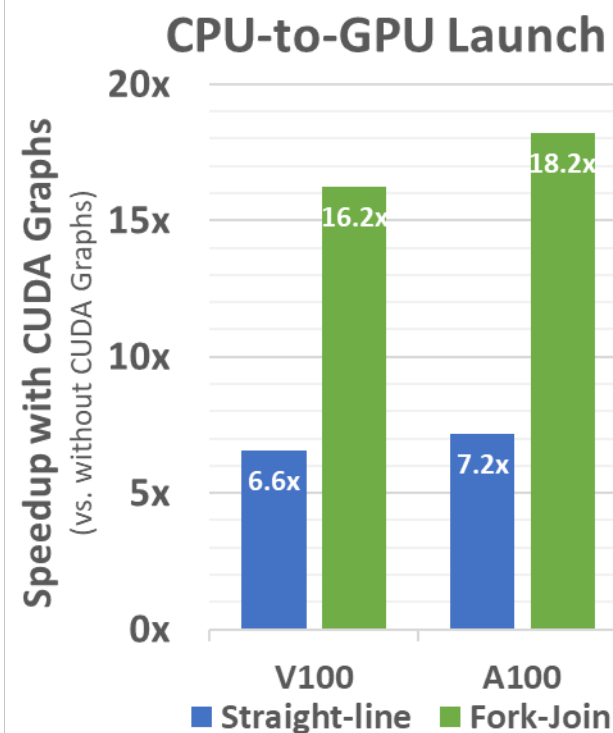
A100 ACCELERATES CUDA GRAPHS



Grid launches:

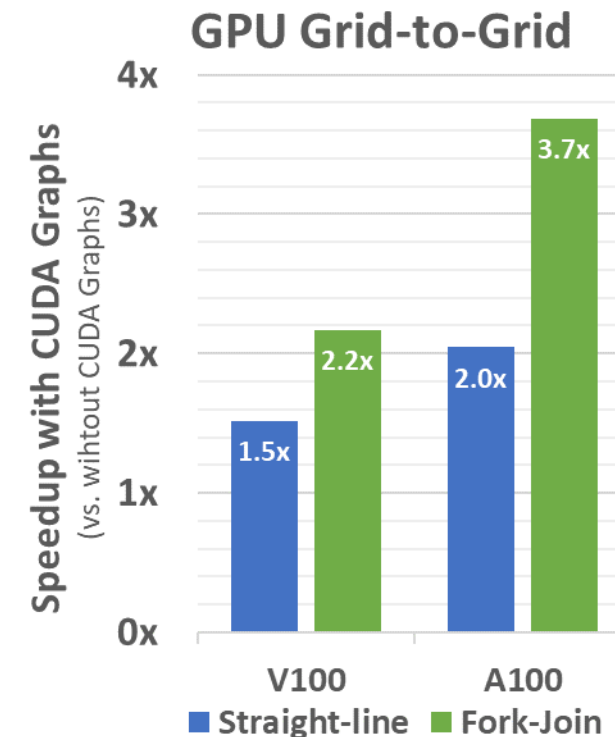
- **CPU-to-GPU**
- **GPU grid-to-grid**

With strong scaling CPU and grid launch overheads become increasingly important (Amdahl's law)



32-node graphs of empty grids, DGX1-V, DGX-A100

One-shot CPU-to-GPU graph submission and graph reuse

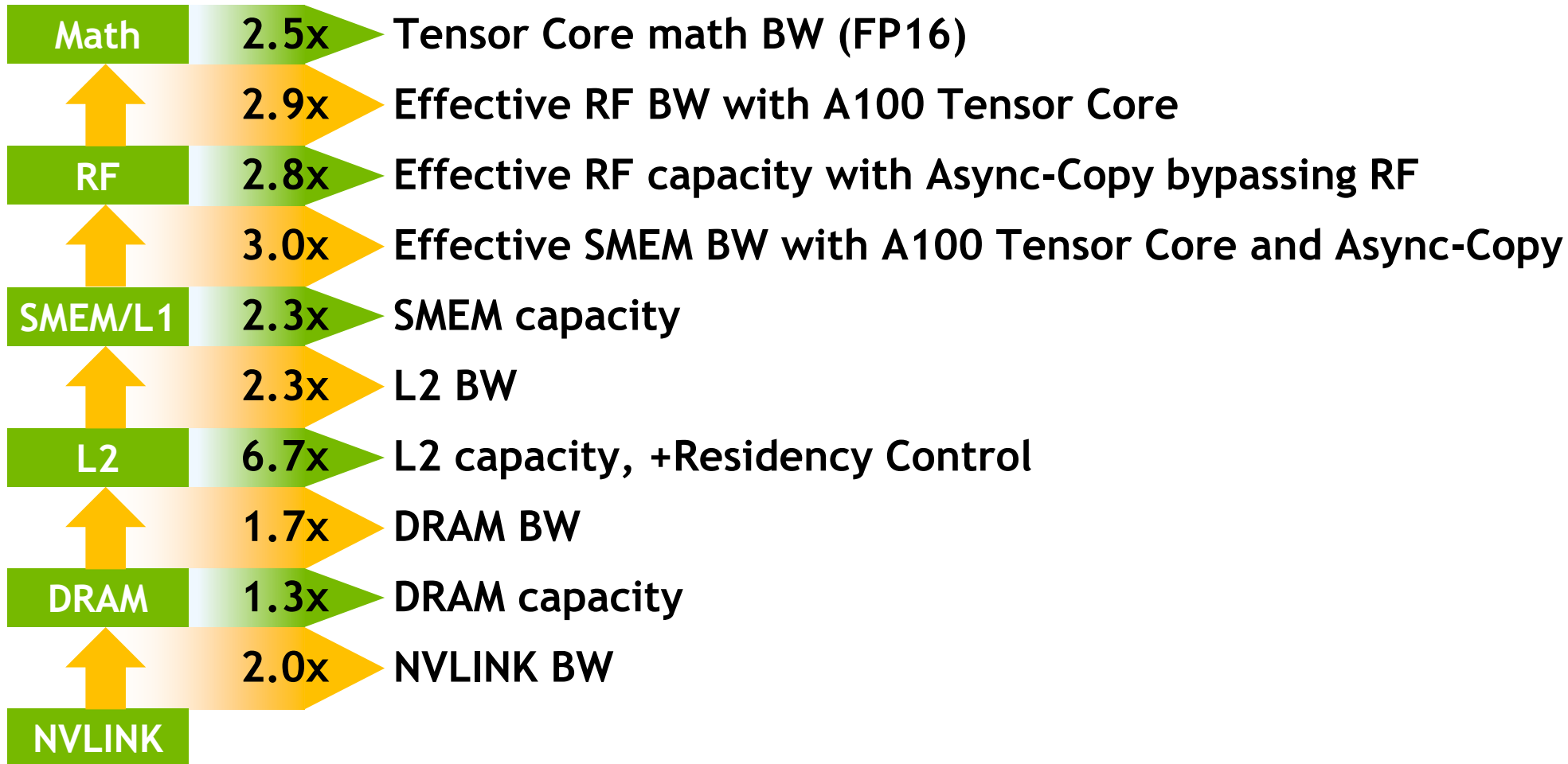


Microarchitecture improvements for grid-to-grid latencies

A100 STRONG SCALING INNOVATIONS

Delivering unprecedented levels of performance

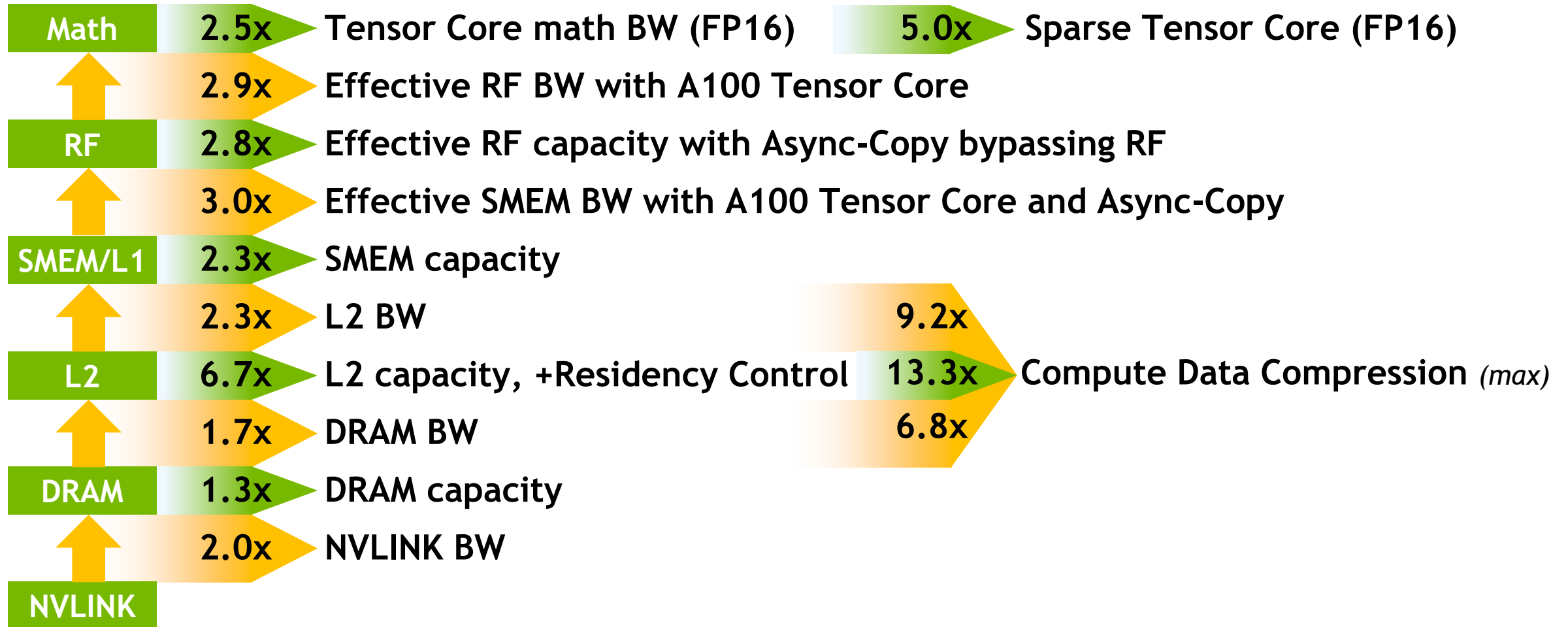
A100 improvements over V100




A100 STRONG SCALING INNOVATIONS

Delivering unprecedented levels of performance

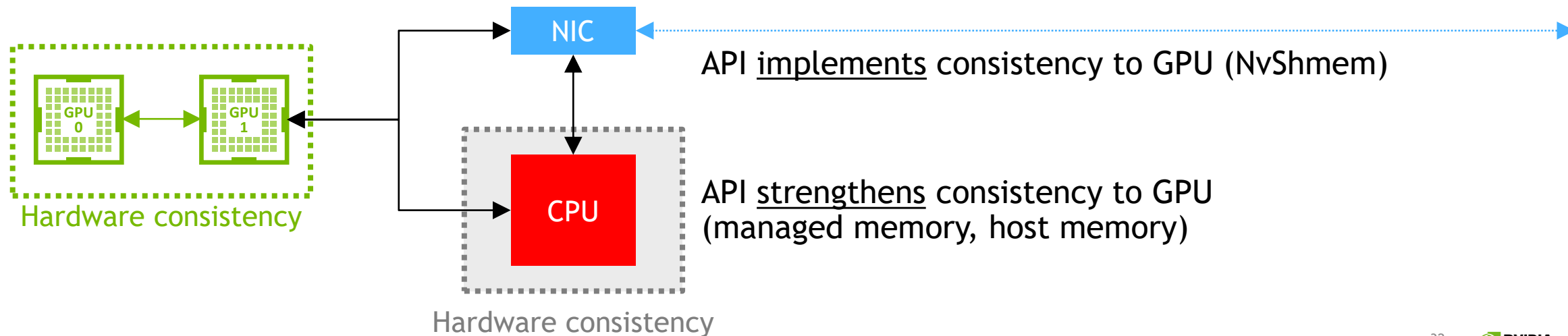
A100 improvements over V100



- 
1. New Tensor Core
 2. Strong Scaling
 3. Elastic GPU
 4. Productivity

NVLINK: ONE BIG GPU

- ▶ **InfiniBand/Ethernet**: travels a long distance, consistency is the responsibility of software
- ▶ **PCI Express**: hardware consistency for I/O, not for programming language memory models
- ▶ **NVLINK**: hardware consistency for programming language memory models, like system bus

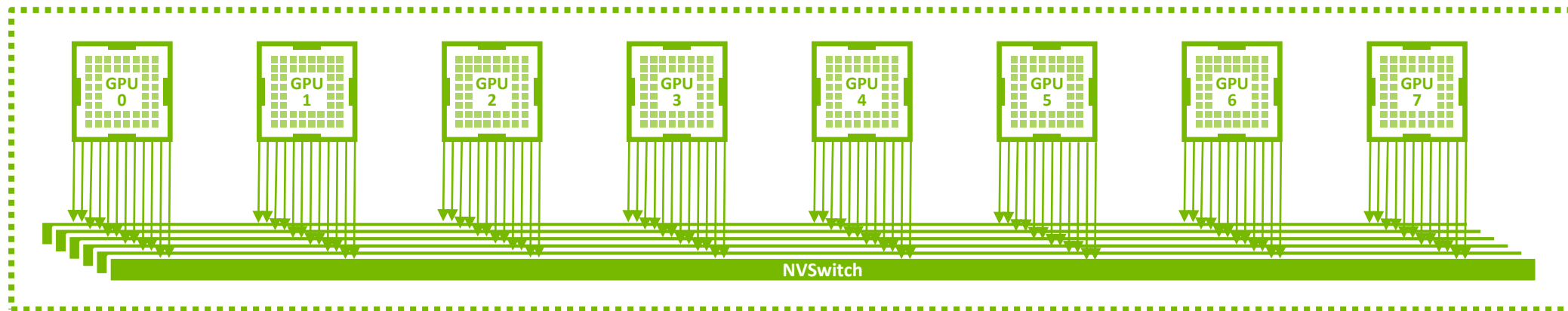


HGX A100: 3RD GEN NVLINK

- ▶ **HGX A100 4-GPU**: fully-connected system with 100GB/s all-to-all BW

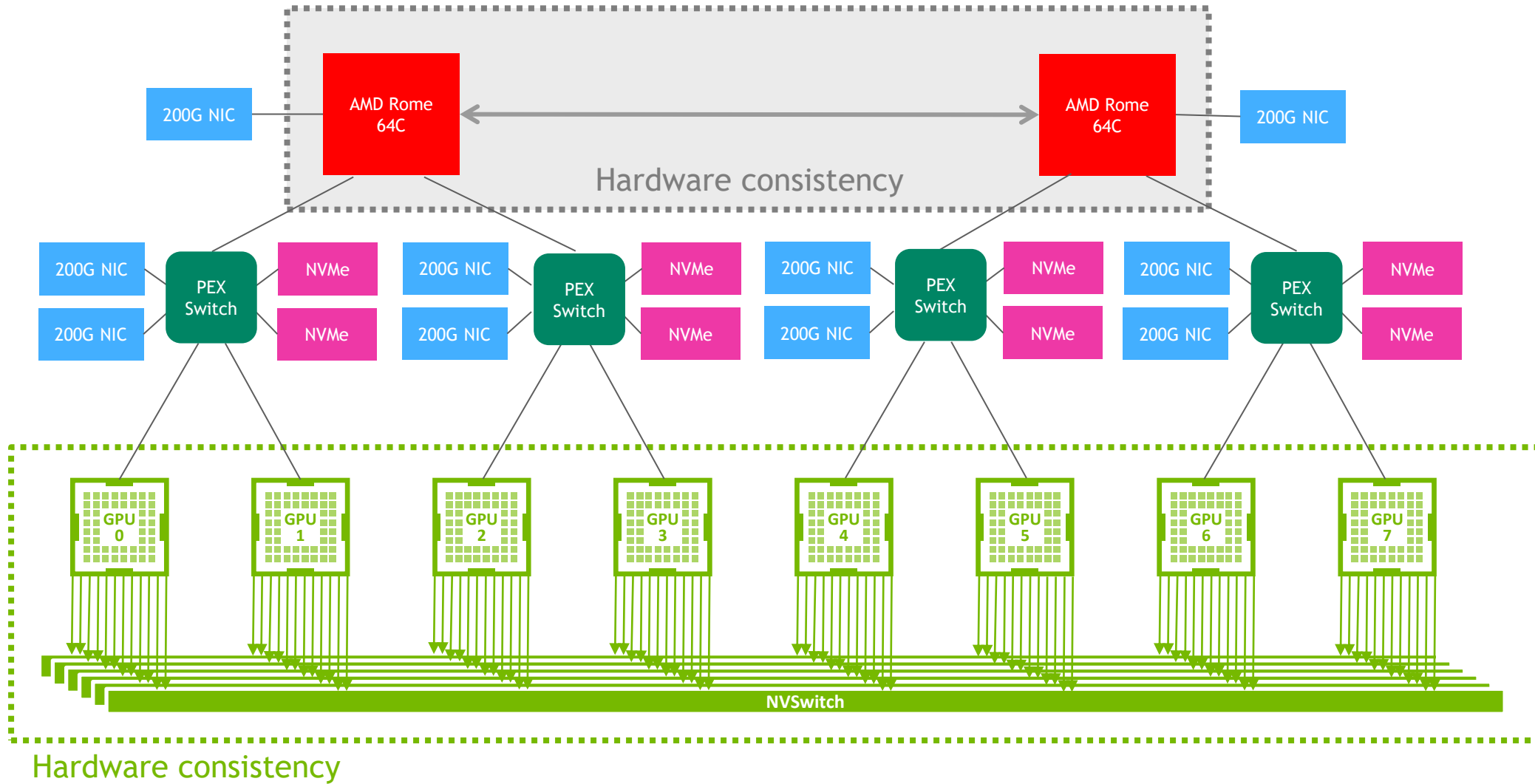
HGX A100: 3RD GEN NVLINK & SWITCH

- ▶ **HGX A100 4-GPU**: fully-connected system with 100GB/s all-to-all BW
- ▶ **New NVSwitch**: 6B transistors in TSMC 7FF, 36 ports, 25GB/s each, per direction
- ▶ **HGX A100 8-GPU**: 6x NVSwitch in a fat tree topology, 2.4TB/s full-duplex bandwidth



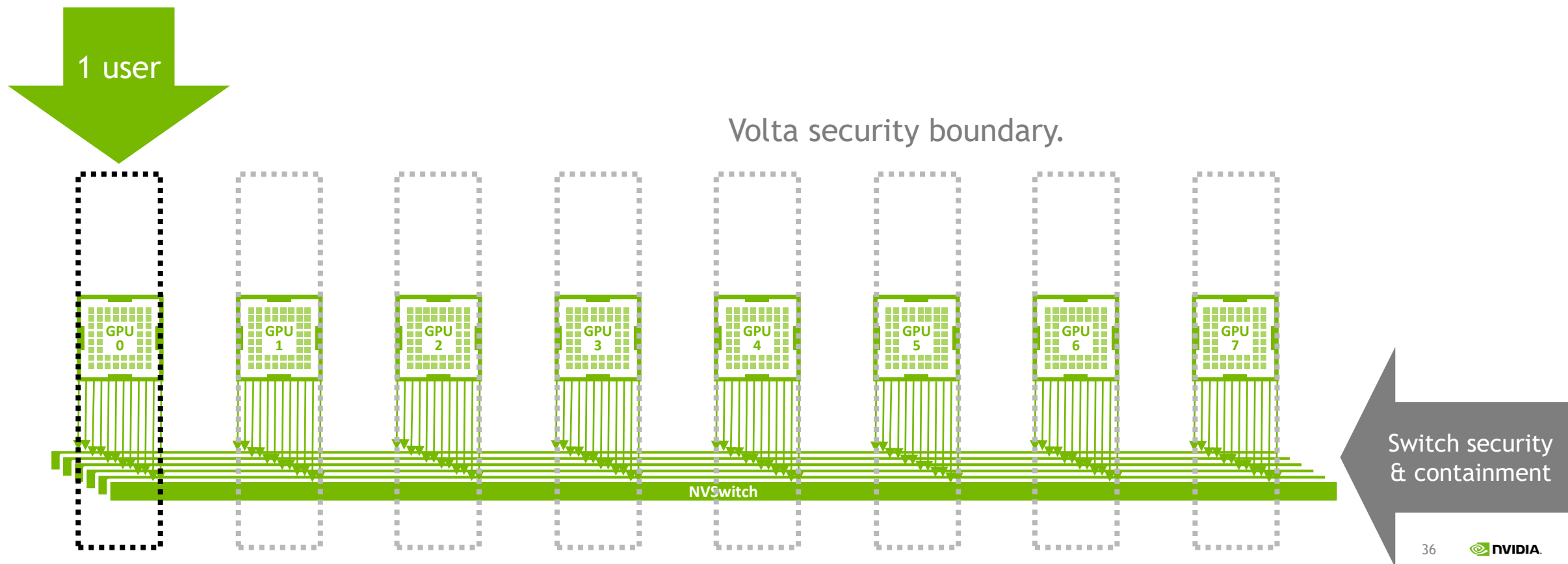
Hardware consistency

DGX A100: PCIE4 CONTROL & I/O



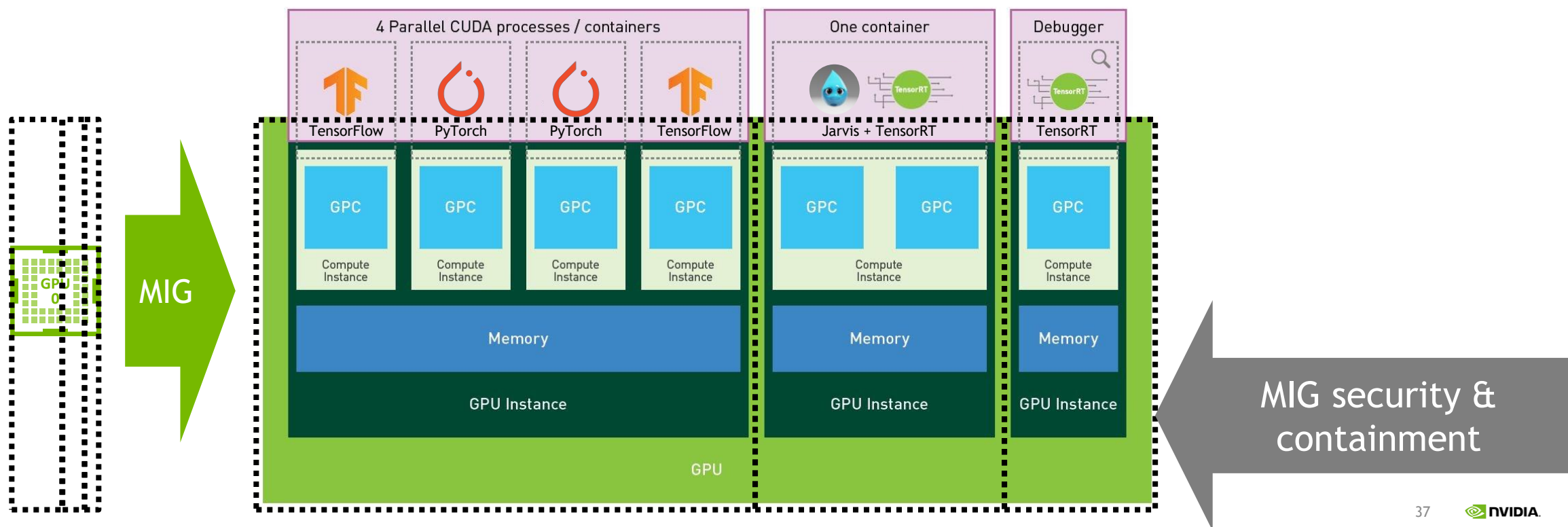
CLOUD SMALL INSTANCE USAGE

- ▶ Small workloads can under-utilize GPU cloud instances, provisioned at whole GPU level
- ▶ CSPs can't use MPS for GPU space-sharing, because it doesn't provide enough isolation



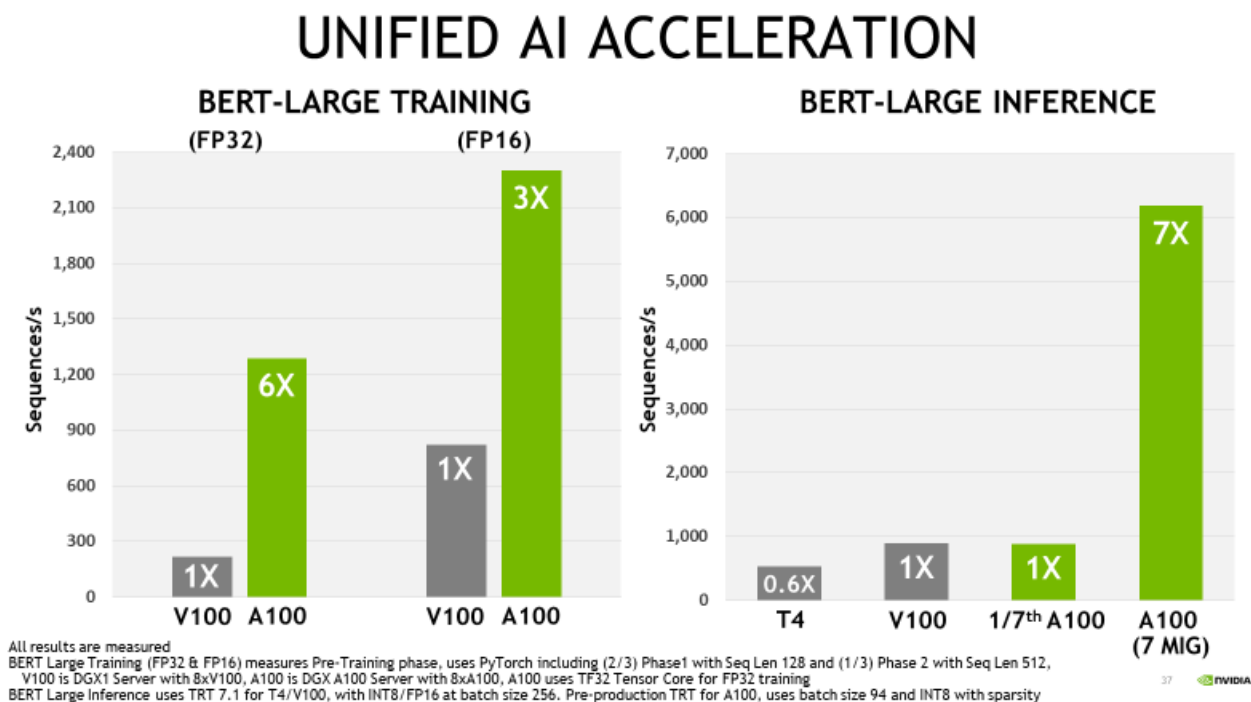
NEW: MULTI-INSTANCE GPU (MIG)

- ▶ Up to 7 instances total, dynamically reconfigurable
- ▶ **Compute instances:** compute/fault isolation, but share/compete for memory
- ▶ **GPU instances:** separate and isolated paths through the entire memory system



ELASTIC GPU COMPUTING


- ▶ Each A100 is 1 to 7 GPUs
- ▶ Each DGX A100 is 1 to 56 GPUs
- ▶ Each GPU can serve a different user, with full memory isolation and QoS



→S21975: Inside NVIDIA's Multi-Instance GPU Feature (recording available)

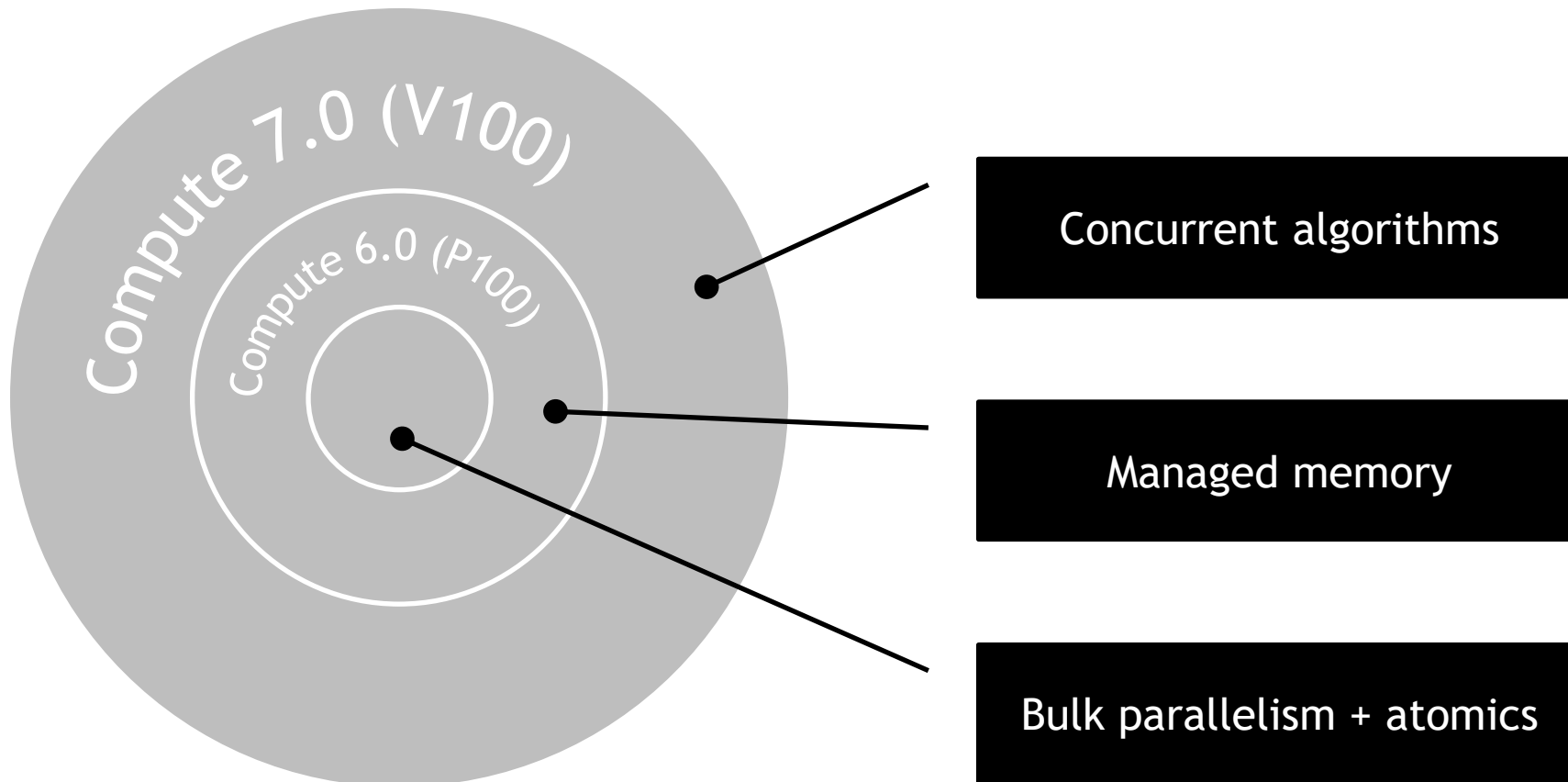
→S21884: Under the Hood of the new DGX A100 System Architecture (recording available soon)

→S21702: Introducing NVIDIA DGX A100: The Universal AI System for Enterprise, 5/20 9:00am PDT

- 
1. New Tensor Core
 2. Strong Scaling
 3. Elastic GPU
 4. Productivity

COMPUTE CAPABILITY

Programming Model Development at NVIDIA



GPU PROGRAMMING IN 2020 AND BEYOND

Math Libraries | Standard Languages | Directives | CUDA

```
std::transform(par, x, x+n, y, y,  
              [=](float x, float y) {  
                  return y + a*x;  
              });
```

```
do concurrent (i = 1:n)  
    y(i) = y(i) + a*x(i)  
enddo
```

GPU Accelerated
C++ and Fortran

```
#pragma acc data copy(x,y)  
{  
    ...  
    std::transform(par, x, x+n, y, y,  
                  [=](float x, float y) {  
                      return y + a*x;  
                  });  
    ...  
}
```

Incremental Performance
Optimization with Directives

```
__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];  
}  
  
int main(void) {  
    ...  
    cudaMemcpy(d_x, x, ...);  
    cudaMemcpy(d_y, y, ...);  
  
    saxpy<<<(N+255)/256, 256>>>(...);  
  
    cudaMemcpy(y, d_y, ...);  
}
```

Maximize GPU Performance with
CUDA C++/Fortran

GPU Accelerated Math Libraries

PROGRAMMING MODEL WANTED

Software pipelining to hide latency is hard.

```
__device__ void exhibit_A1()
{
    memcpy(/* ... */); //< blocks here
    /* more work */

    compute();          //< needed here
    /* more work */
}
```

Data

```
__device__ void exhibit_B1()
{
    compute_head();
    __syncthreads(); //< blocks here
    /* more work */


    compute_tail();    //< needed here
    /* more work */
}
```

Compute

PROGRAMMING MODEL WANTED


Software pipelining to hide latency is hard.

```
__device__ void exhibit_A2()
{
    memcpy(/* ... */); //< blocks here
    /* memcpy( ... ); */
    compute(); //< needed here
    /* compute(); */
}
```



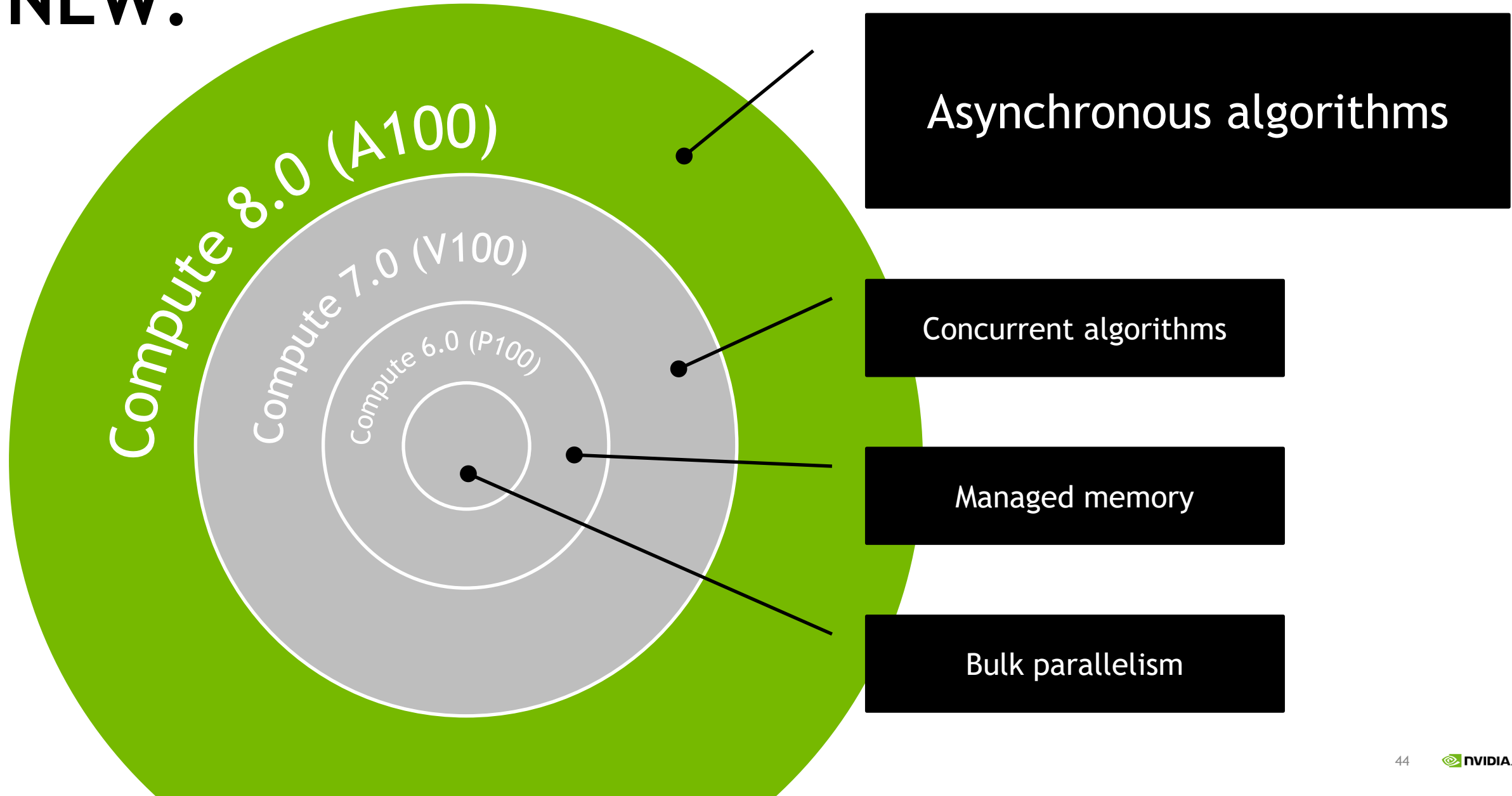
Data

```
__device__ void exhibit_B2()
{
    compute_head();
    __syncthreads(); //< blocks here
    /* compute_head();
       __syncthreads(); */
    compute_tail(); //< needed here
    /* compute_tail(); */
}
```



Compute

NEW:

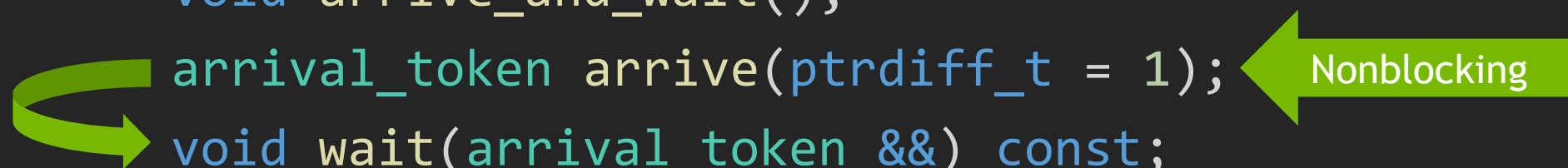


CO-DESIGNED: A100 & C++20 barrier

Key to asynchronous programming in compute_80

```
#include <cuda/barrier> // ISO C++20 conforming extension  
using barrier = cuda::barrier<cuda::thread_scope_block>;
```

```
class barrier { // synopsis  
    //...  
    void arrive_and_wait();  
    arrival_token arrive(ptrdiff_t = 1);  
    void wait(arrival_token &&) const;  
    //...  
};
```



Nonblocking

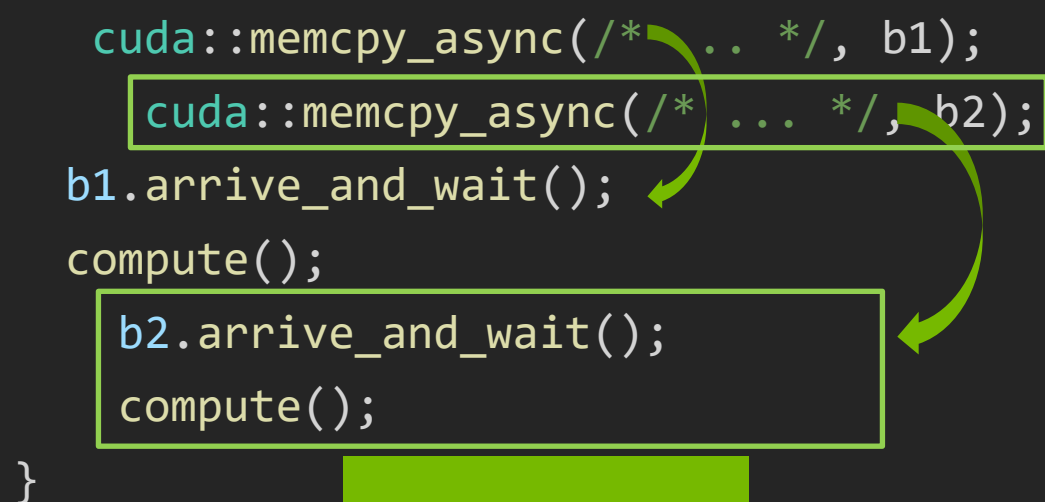
ASYNCHRONOUS COPY + BARRIER

Capability	PTX ISA	CUDA C++ API
Asynchronous barrier	<code>mbarrier.{<basis functions>}</code>	<code>cuda::barrier<...></code>
Asynchronous copy	<code>cp.async.ca +</code> <code>cp.async.mbarrier.arrive</code>	<code>cuda::memcpy_async(...)</code>
+Cache-bypass	<code>cp.async.cg</code>	CUDA 11 preview library in <code>experimental:: namespace</code>
+Zero-fill ragged edge	<code>cp.async.* ... wr-size, rd-size;</code>	
+User-level tracking	<code>cp.async.mbarrier.arrive.noinc</code>	
+Single-threaded mode	<code>cp.async.{commit_group, wait_group}</code>	

ASYNCHRONOUS PROGRAMMING MODEL

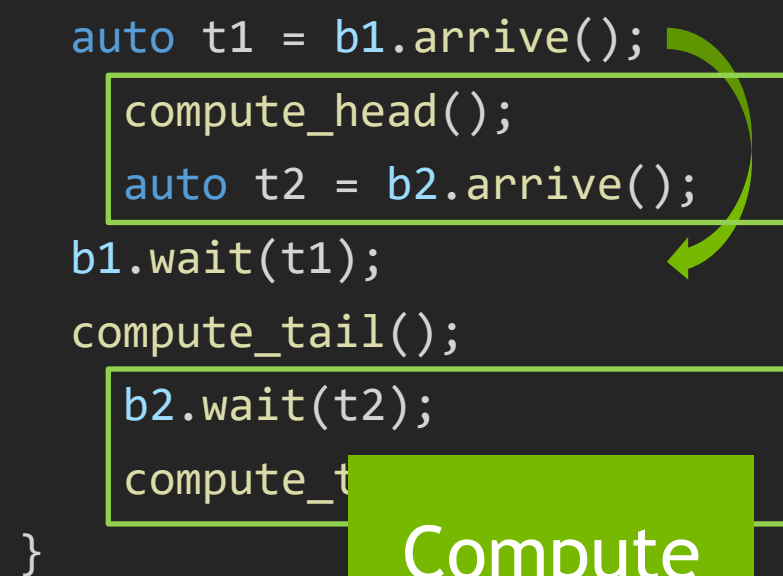
```
#include <cuda/barrier> // ISO C++20 conforming extension
using barrier = cuda::barrier<cuda::thread_scope_block>;
```

```
__device__ void exhibit_A3()
{
    __shared__ barrier b1, b2;
    // ^^initialization omitted
    cuda::memcpy_async(/*...*/, b1);
    cuda::memcpy_async(/*...*/, b2);
    b1.arrive_and_wait();
    compute();
    b2.arrive_and_wait();
    compute();
}
```



Data

```
__device__ void exhibit_B3()
{
    __shared__ barrier b1, b2;
    // ^^initialization omitted
    compute_head();
    auto t1 = b1.arrive();
    compute_head();
    auto t2 = b2.arrive();
    b1.wait(t1);
    compute_tail();
    b2.wait(t2);
    compute_t
}
```



Compute

MULTI-BUFFERING PIPELINES IN C++

```
#include <cuda/barrier> // ISO C++20 conforming extension
using barrier = cuda::barrier<cuda::thread_scope_block>;

__global__ void exhibit_C(/* ... */) {
    __shared__ barrier b[2];
    // ^^initialization omitted
    barrier::arrival_token t[2];
    cuda::memcpy_async(/* ... */, b[0]);
    t[0] = b[0].arrive();
    for(int step = 0, next = 1; step < steps; ++step, ++next) {
        if(next < steps) {
            b[next & 1].wait(t[next & 1]);
            cuda::memcpy_async(/* ... */, b[next & 1]);
            t[next & 1] = b[next & 1].arrive();
        }
        b[step & 1].wait(t[step & 1]);
        compute();
        t[step & 1] = b[step & 1].arrive();
    }
}
```

Data

Compute

MULTI-BUFFERING PIPELINES IN C++

```
#include <cuda/barrier> // ISO C++20 conforming extension
using barrier = cuda::barrier<cuda::thread_scope_block>;
```

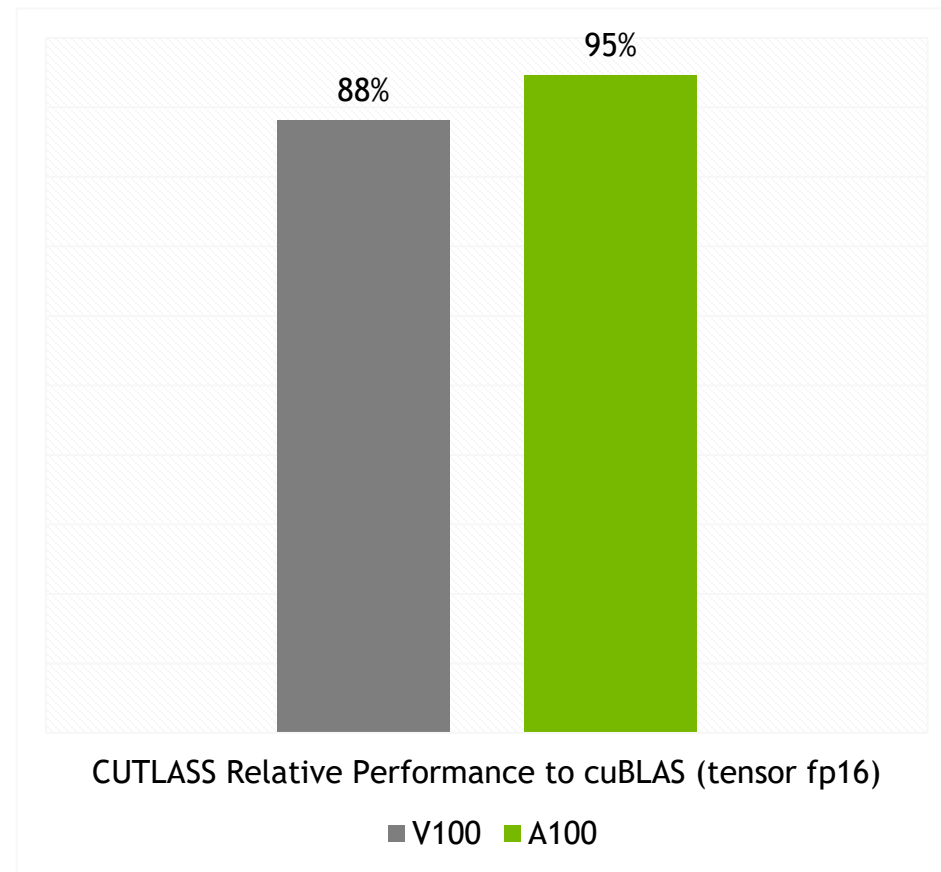
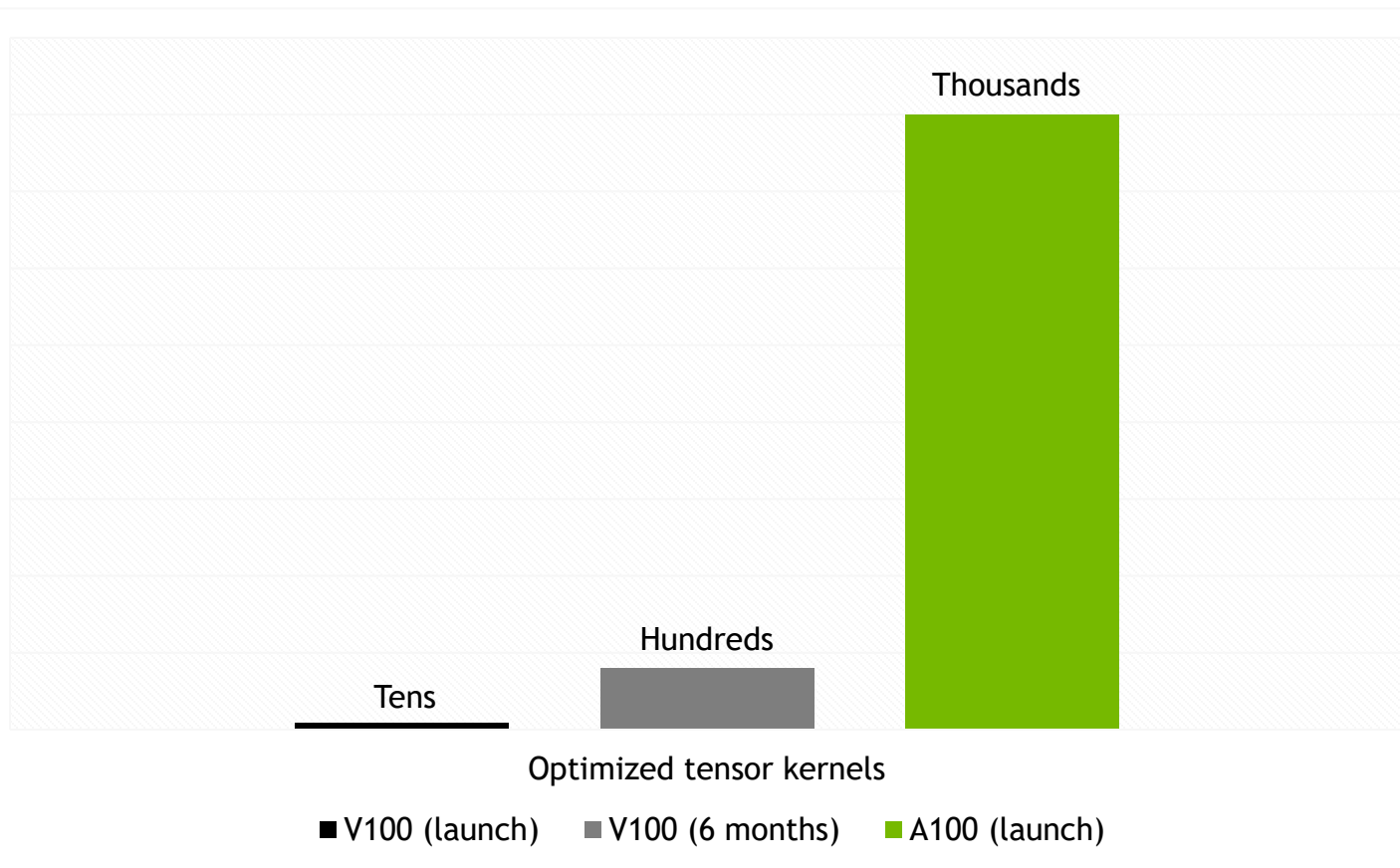
```
__global__ void exhibit_C(/* ... */) {
    __shared__ barrier b[2];
    // ^^initialization omitted
    barrier::arrival_token t[2];
    cuda::memcpy_async(/* ... */, b[0]);
    t[0] = b[0].arrive();
    for(int step = 0, next = 1; step < steps; ++step, ++next) {
        if(next < steps) {
            b[next & 1].wait(t[next & 1]);
            cuda::memcpy_async(/* ... */, b[next & 1]);
            t[next & 1] = b[next & 1].arrive();
        }
        b[step & 1].wait(t[step & 1]);
        compute();
        t[step & 1] = b[step & 1].arrive();
    }
}
```

Data



Compute

OUR PRODUCTIVITY GAINS FROM A100

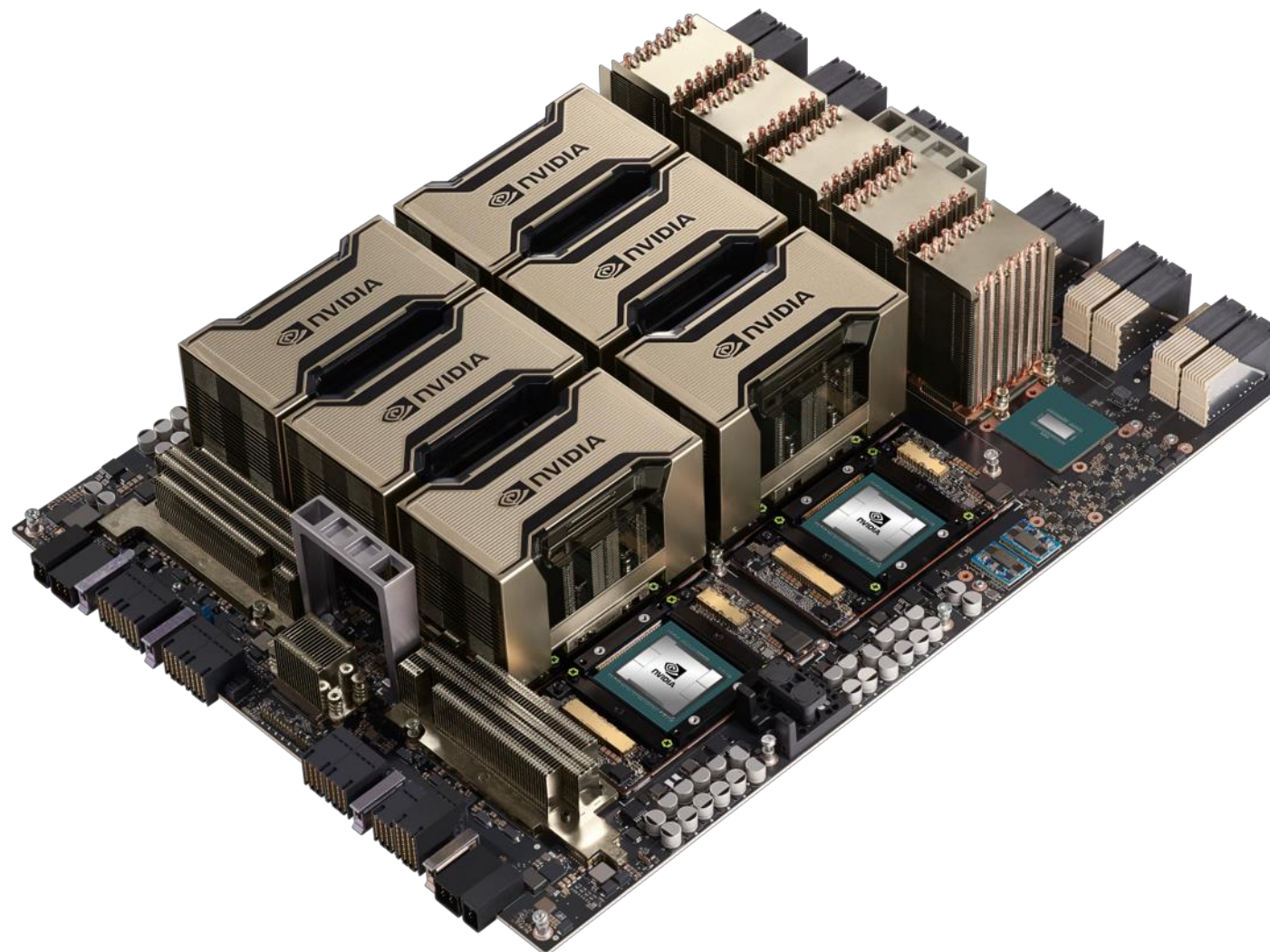


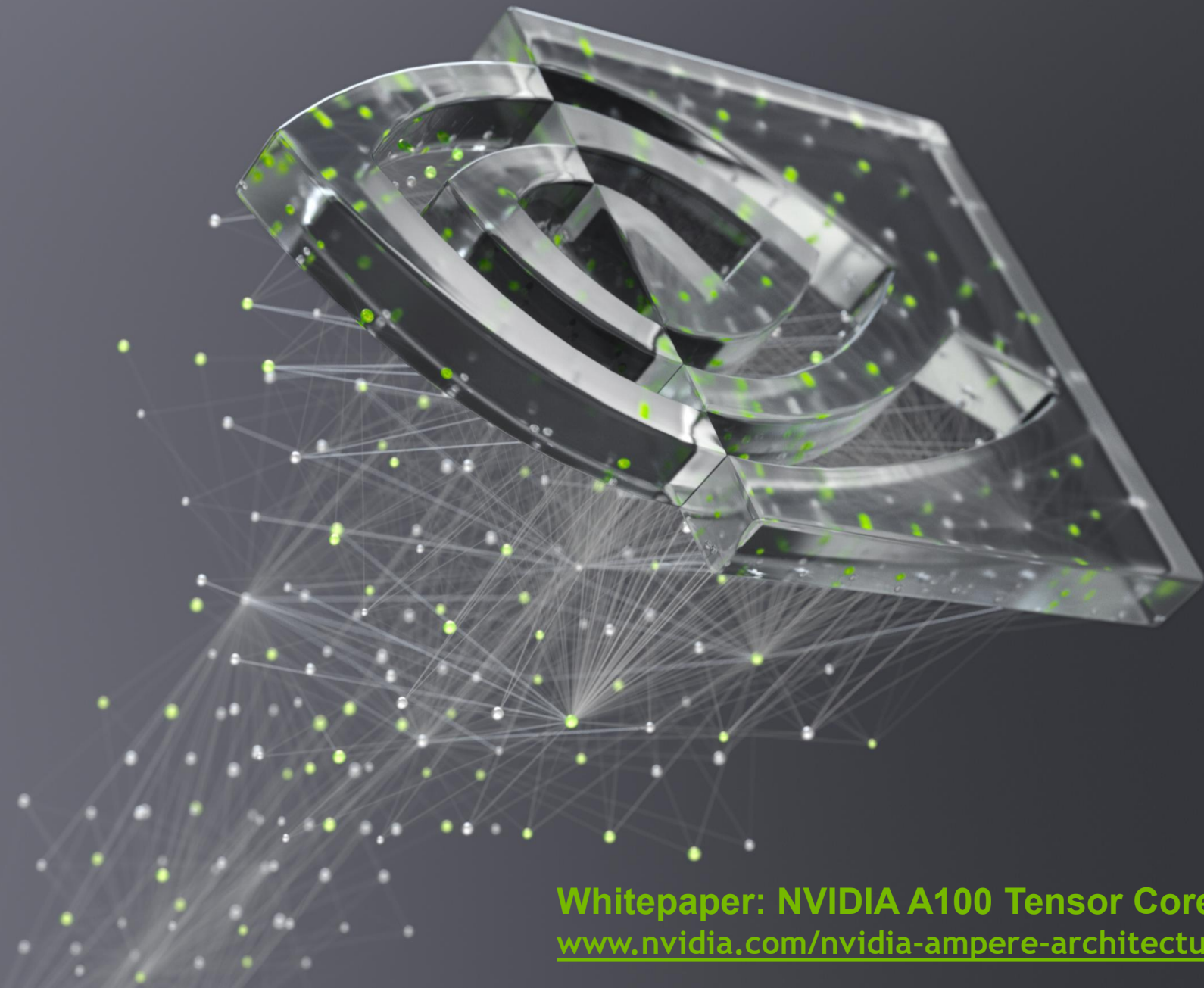
→S21745: Developing CUDA kernels to push Tensor Cores to the Absolute Limit on NVIDIA A100, 5/21 11:30AM PST



CLOSING

UNPRECEDENTED ACCELERATION AT EVERY SCALE





nVIDIA

Whitepaper: NVIDIA A100 Tensor Core GPU Architecture
www.nvidia.com/nvidia-ampere-architecture-whitepaper