Lecture 15:

MultiGPU + New GPUs + Future

Modern Parallel Computing

John Owens

EEC 289Q, UC Davis, Winter 2018

Credits

- Jim Rogers and Buddy Bland, Oak Ridge National Laboratory
- Unnamed Mellanox presentation
- Cliff Woolley, Mark Harris, Luke Durant, Jiri Kraus, Davide Rossetti, Sylvain Jeaugey, NVIDIA

Logistics for Your Talks

- Suggested 3 slides per group:
  - Motivation: why is your problem important?
  - Approach: how did you go about solving the problem?
  - Results: what did you learn?
  - Let me know if you don't have a laptop to present
- 4 minutes per group
  - Historically people go long—practice!
  - All students must speak
- If you have a demo, put your slides on the same machine as your demo

Tips on Presentations

- Most important: Impact on GPU computing
- Consider your audience
  - Likely that your colleagues don't know your subject in much detail
  - I don't either
- Thus, concentrate on high level and results
- Pick a few interesting points and explain them well

Your Writeups

- Talk: Graded on effectiveness of talk
- Writeup: Graded on content
- Most important: Impact on GPU computing
  - What did you learn about GPU computing and your application of choice?
  - How will what you learned impact *future parallel computing systems*?

Your Writeups

- Think about presenting to our guest speakers …
  - High-level motivation: Why is the problem you chose important?
  - Technical detail: Why is your solution a good one? (Make sure to detail your contribution!)
    - Describe previous work
    - Concentrate on impact on GPU computing hardware and software
    - Throughout, think about: Analysis!
- http://www.ece.ucdavis.edu/~jowens/commonerrors.html
- http://www.ece.ucdavis.edu/~jowens/biberrors.html

What to turn in

- Writeup: PDF (brevity is appreciated)
- Presentation: PDF lecture slides
- Code: Zip up your code please

Multi-GPU Computing

Processor Architecture:
Power vs. Single Thread Performance

- Multi-core architectures are a good first response to power issues
  - Performance through parallelism, not frequency
  - Exploit on-chip locality
- However, conventional processor architectures are optimized for single thread performance rather than energy efficiency
  - Fast clock rate with latency(performance)-optimized memory structures
  - Wide superscalar instruction issue with dynamic conflict detection
  - Heavy use of speculative execution and replay traps
  - Large structures supporting various types of predictions
  - Relatively little energy spent on actual ALU operations
- Could be much more energy efficient with multiple simple processors, exploiting vector/SIMD parallelism and a slower clock rate
- **But serial thread performance is really important (Amdahl's Law):**
  - If you get great parallel speedup, but hurt serial performance, then you end up with a niche processor (less generally applicable, harder to program)

Courtesy of Steve Scott
presented at SciDAC 2010

Exascale Conclusion: Heterogeneous Computing

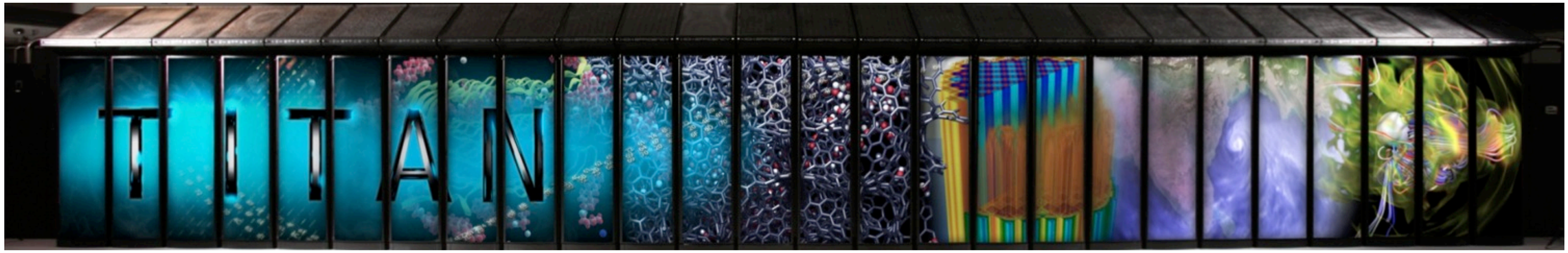
- To achieve scale and sustained performance per {$,watt}, must adopt:
  - …a *heterogeneous* node architecture
    - fast serial threads coupled to many efficient parallel threads
  - …a deep, explicitly managed memory hierarchy
    - to better exploit locality, improve predictability, and reduce overhead
  - …a microarchitecture to exploit parallelism at all levels of a code
    - distributed memory, shared memory, vector/SIMD, multithreaded
    - (related to the "concurrency" challenge—leave no parallelism untapped)

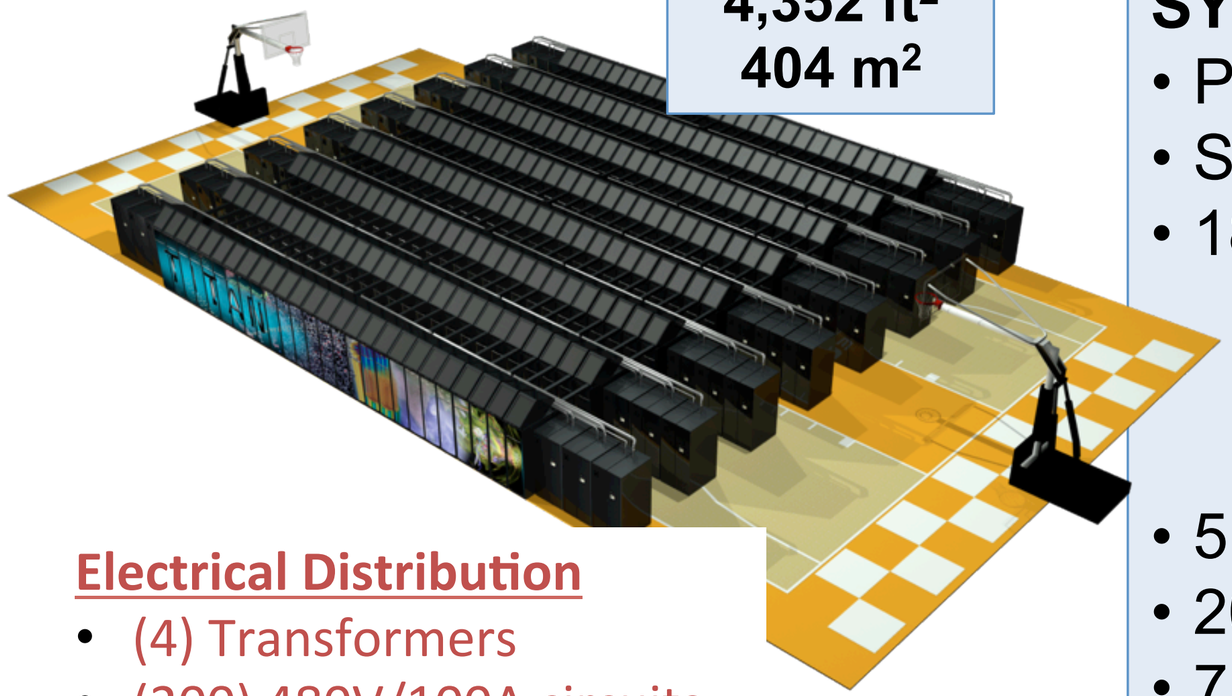
- This sounds a lot like GPU accelerators…
  - NVIDIA Fermi™ has made GPUs feasible for HPC
    - Robust error protection and strong DP FP, plus programming enhancements
  - Expect GPUs to make continued and significant inroads into HPC
    - Compelling technical reasons + high volume market
  - Programmability remains primary barrier to adoption
    - Cray is focusing on compilers, tools and libraries to make GPUs easier to use
    - There are also some structural issues that limit applicability of current designs…
  - Technical direction for Exascale:
    - Unified node with "CPU" and "accelerator" on chip sharing common memory
    - Very interesting processor roadmaps coming from Intel, AMD and NVIDIA….

Courtesy of Steve Scott
presented at SciDAC 2010

ORNL's Cray XK7 Titan

- A Hybrid System with 1:1 AMD Opteron CPU and NVIDIA Kepler GPU





4,352 ft²
404 m²

Electrical Distribution

- (4) Transformers
- (200) 480V/100A circuits
- (48) 480V/20A circuits

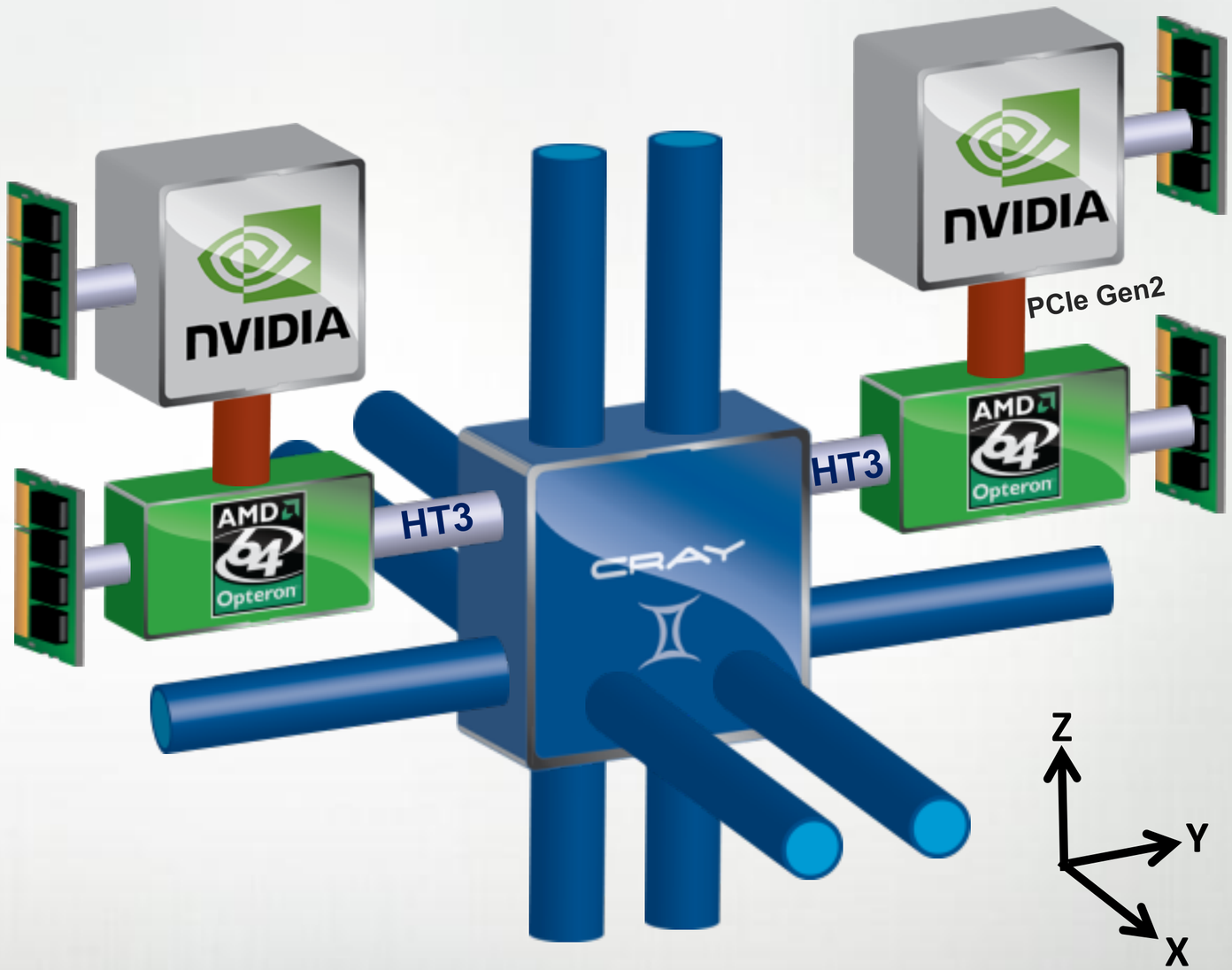
SYSTEM SPECIFICATIONS:

- Peak performance of 27 PF
- Sustained performance of 17.59 PF
- 18,688 Compute Nodes each with:
  - 16-Core AMD Opteron CPU
  - NVIDIA K20x (Kepler) GPU
  - 32 + 6 GB memory
- 512 Service and I/O nodes
- 200 Cabinets
- 710 TB total system memory
- Cray Gemini 3D Torus Interconnect
- 8.9 MW peak energy measurement

Note: ~$1M/MW/year

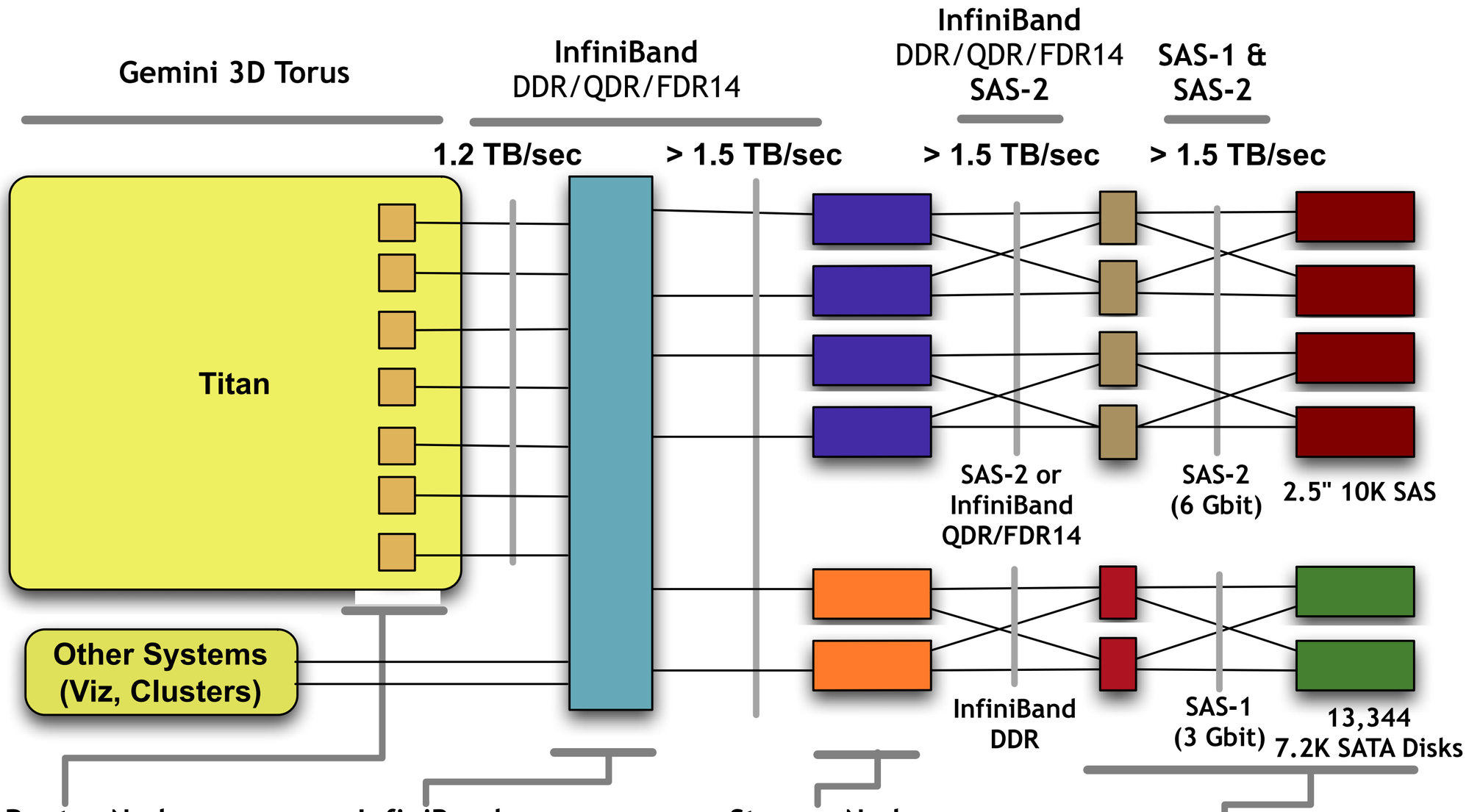
Cray XK7 Compute Node

XK7 Compute Node Characteristics
AMD Opteron 6274 16 core processor - 141 GF
Tesla K20x - 1311 GF
Host Memory 32GB 1600 MHz DDR3
Tesla K20x Memory 6GB GDDR5
Gemini High Speed Interconnect



Slide courtesy of Cray, Inc.

Titan Parallel I/O Architecture



Lustre Router Nodes
Forward I/O operations from Compute Clients to Storage Nodes
384 "XIO nodes"

InfiniBand
switch complex
1200+ port 32 or 54 Gbit/sec
3000+ port 20 Gbit/sec InfiniBand switch complex

Storage Nodes
run parallel file system software and manage incoming FS traffic.

Enterprise Storage
controllers and large racks of disks are connected via SAS

Bandwidths (Peak)

- Fastest CPU memory system: 102 GB/s (Intel [Xeon E7 v3] E7-8890)
- Fastest GPU memory system: 900 GB/s (NVIDIA Tesla V100)
- PCI Express v3.0: 16 GB/s (bidirectional = each way)
- PCI Express v4.0: 32 GB/s (bidirectional = each way)
- NVLink: 25 GB/s bidirectional; V100 has 6 links = 300 GB/s total
- Infiniband 4xEDR: 3.125 GB/s
- QPI (between processors): 16 GB/s (bidirectional)
- SATA v3: 0.6 GB/s
- Gigabit Ethernet: 0.125 GB/s
- 10 Gb Ethernet: 1.25 GB/s

http://www.goland.org/channel_bandwidt/

CUDA-AWARE DEFINITION

Regular MPI

```
//MPI rank 0
cudaMemcpy(s_buf_h,s_buf_d,size,…);
MPI_Send(s_buf_h,size,…);

//MPI rank n-1
MPI_Recv(r_buf_h,size,…);
cudaMemcpy(r_buf_d,r_buf_h,size,…);
```

CUDA-aware MPI

```
//MPI rank 0
MPI_Send(s_buf_d,size,…);

//MPI rank n-1
MPI_Recv(r_buf_d,size,…);
```

CUDA-aware MPI makes MPI+CUDA easier.

MPI API SUPPORT

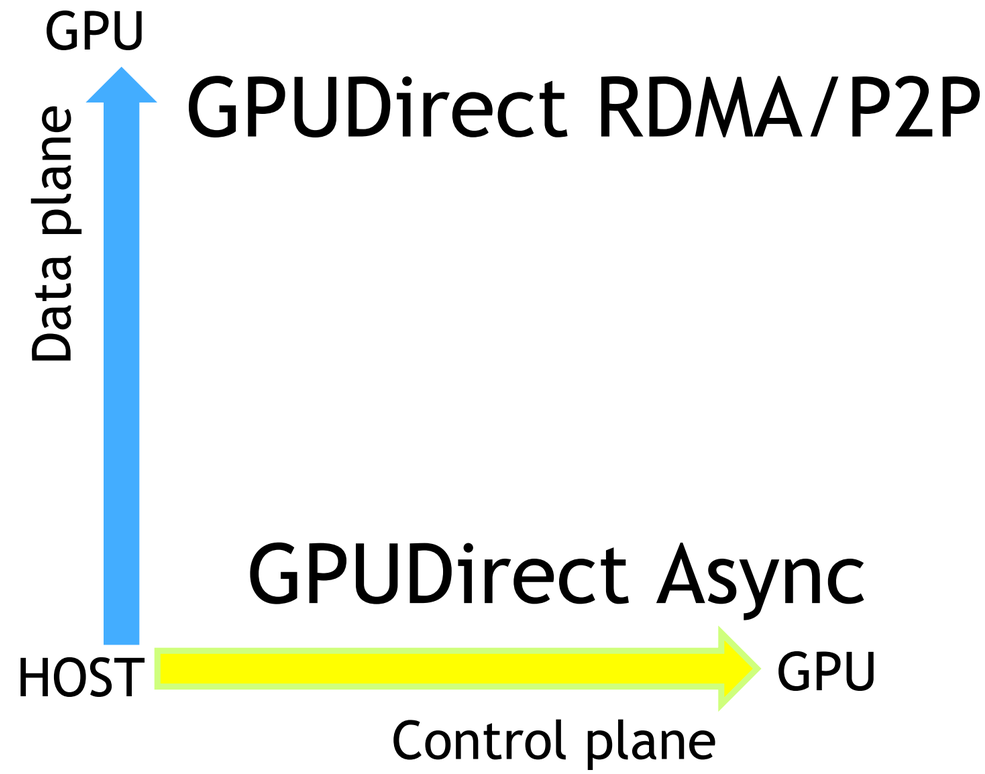
- Yes
  - All send and receive types
  - All non-arithmetic collectives
- No
  - Reduction type operations – MPI_Reduce, MPI_Allreduce, MPI_Scan
  - Non-blocking collectives
  - MPI-2 and MPI-3 (one sided) RMA

- FAQ will be updated as support changes

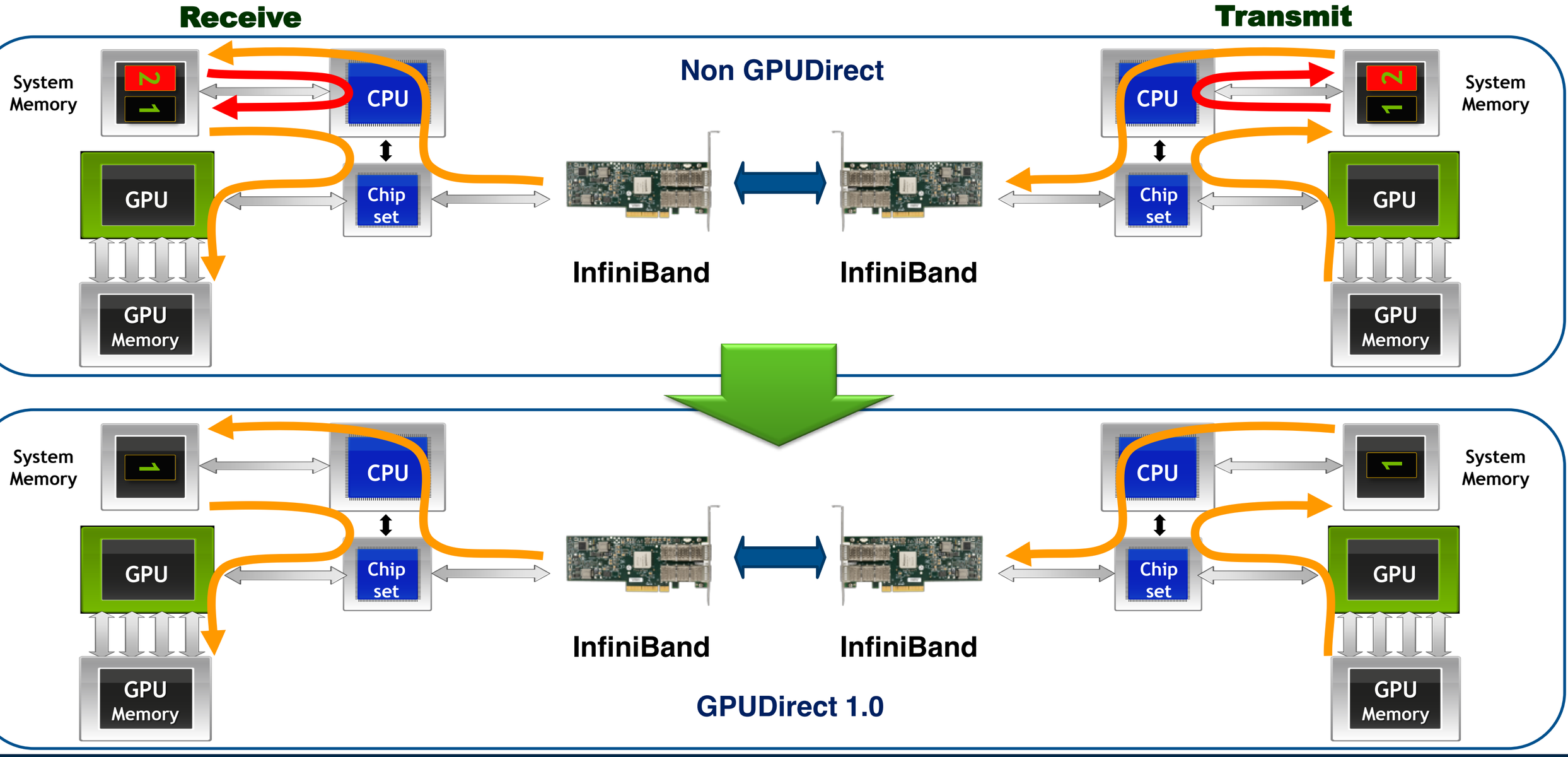
GPUDIRECT

scopes

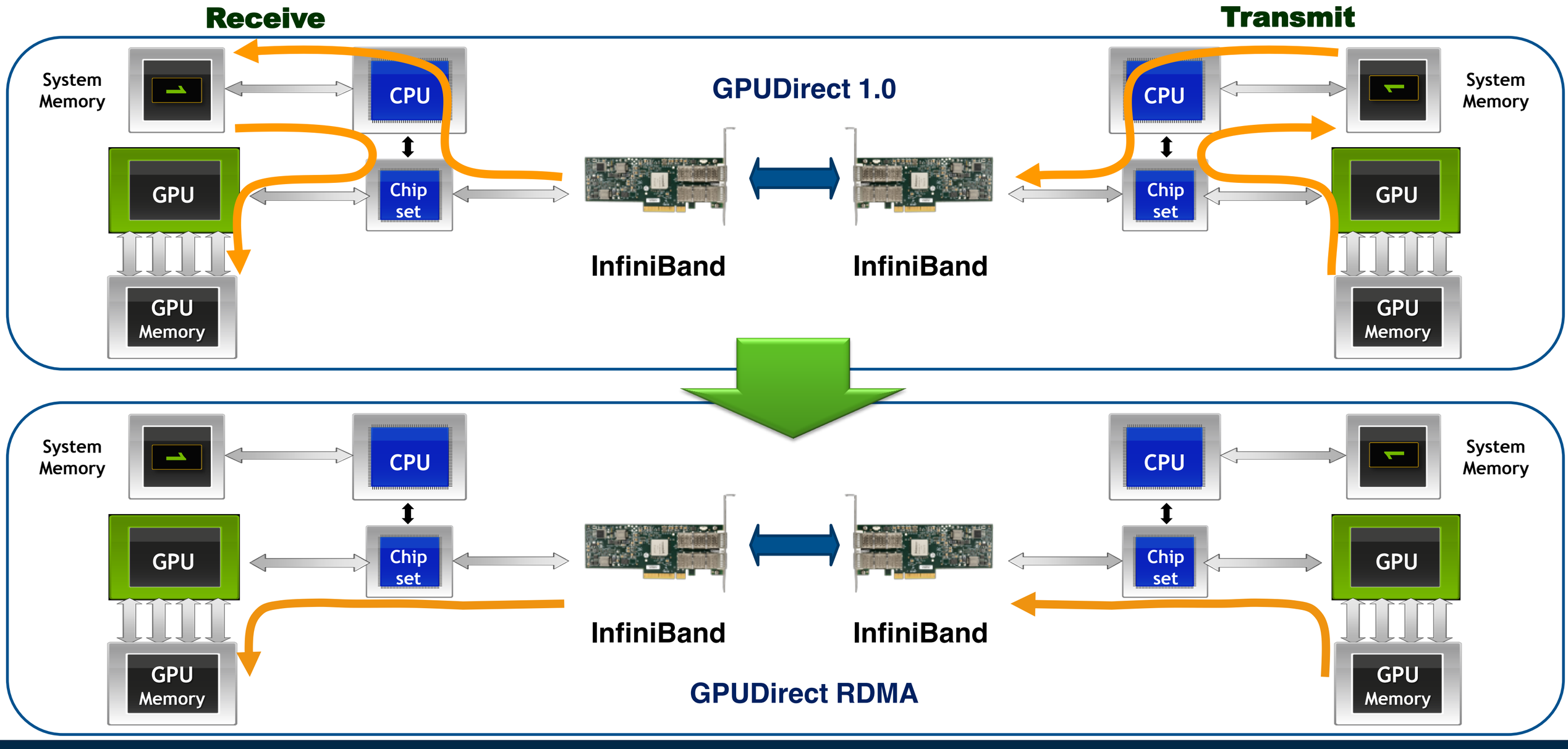
- GPUDirect P2P → data
  - Intra-node
  - GPUs both master and slave
  - Over PCIe or NVLink
- GPUDirect RDMA → data
  - Inter-node
  - GPU slave, 3rd party device master
  - Over PCIe
- GPUDirect Async → control
  - GPU & 3rd party device, master & slave
  - Over PCIe



GPUDirect 1.0

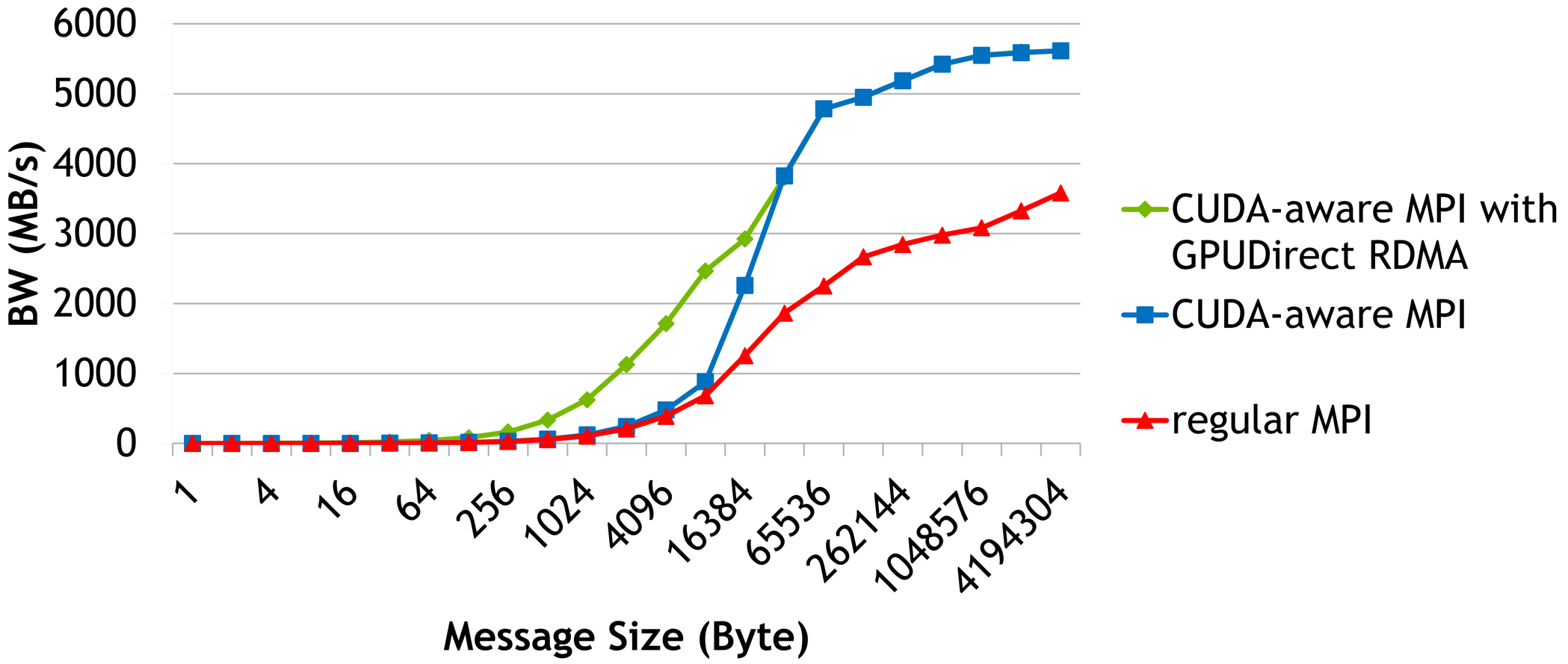


GPUDirect RDMA



PERFORMANCE RESULTS GPUDIRECT RDMA

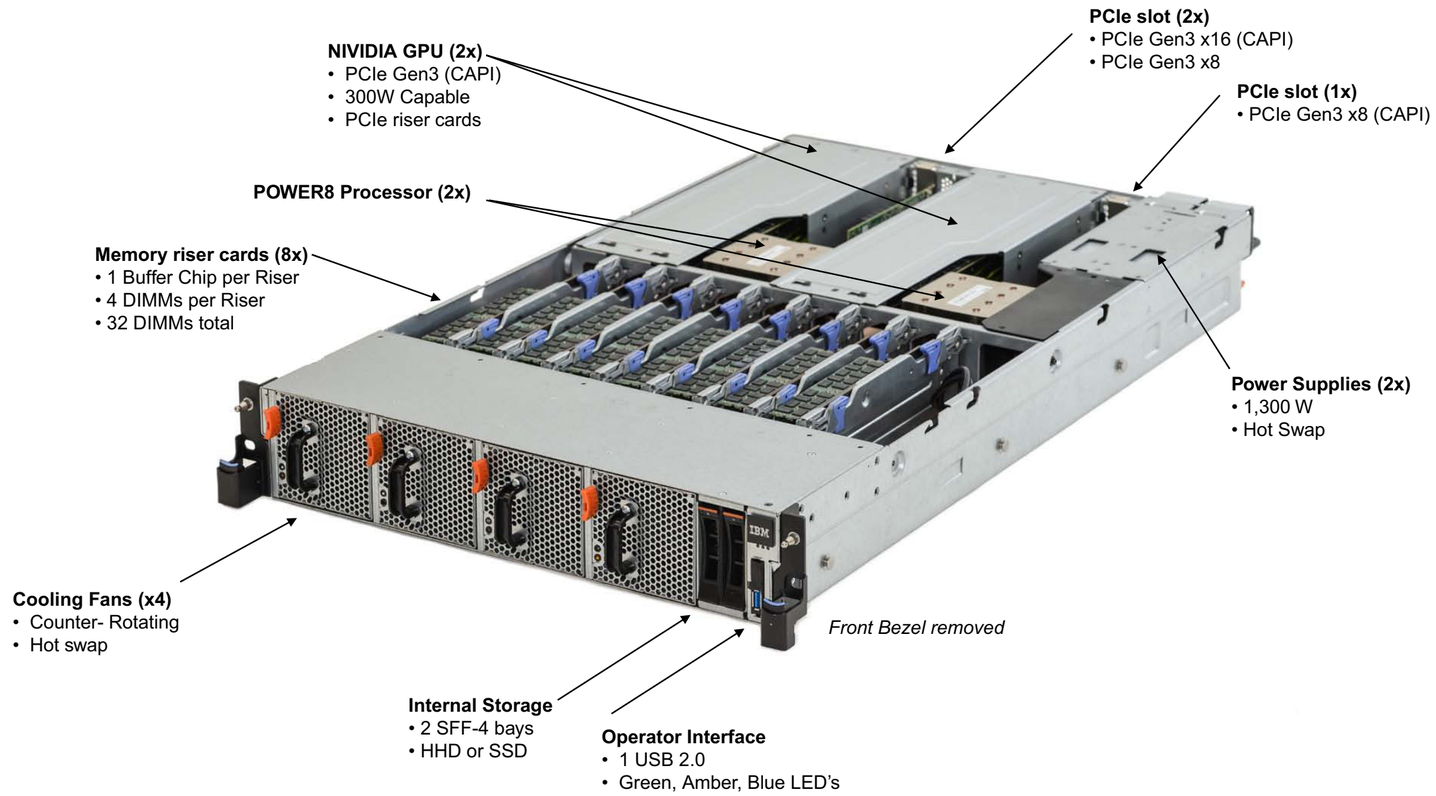
OpenMPI 1.10.4 MLNX FDR IB (4X) Tesla P100 PCI-E 16GB

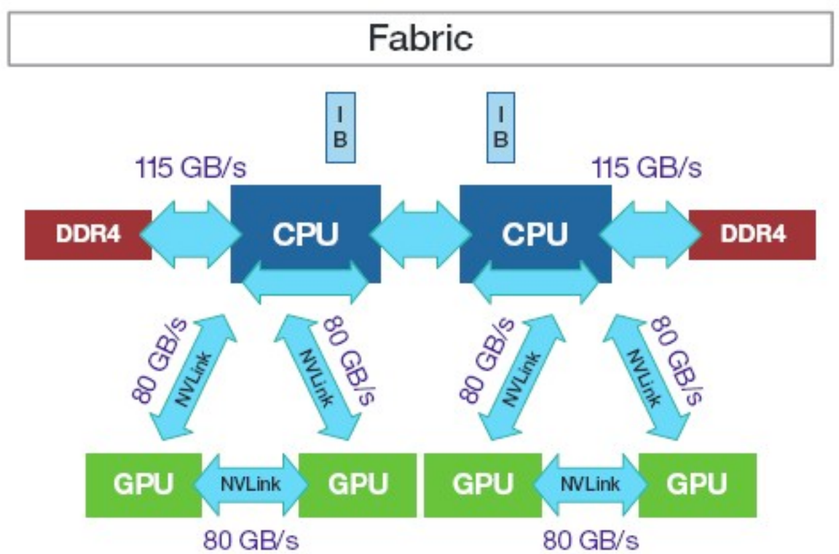


Latency (1 Byte)	15.65 us	16.50 us	4.87 us

IBM Power Systems S822LC

- 2U chassis
- 2 POWER8 processors (8 and 10 core)
- 1 TB DRAM
- 2 × NVIDIA K80 GPUS (4 sockets)
- 2 × PCIe connections (latest machine: NVLink)





http://www.redbooks.ibm.com/abstracts/redp5283.html?Open

https://www.nextplatform.com/2016/09/08/refreshed-ibm-power-linux-systems-add-nvlink/

DGX-1 SYSTEM TOPOLOGY

GPU – CPU link:

PCIe

12.5+12.5 GB/s eff BW

GPUDirect P2P:

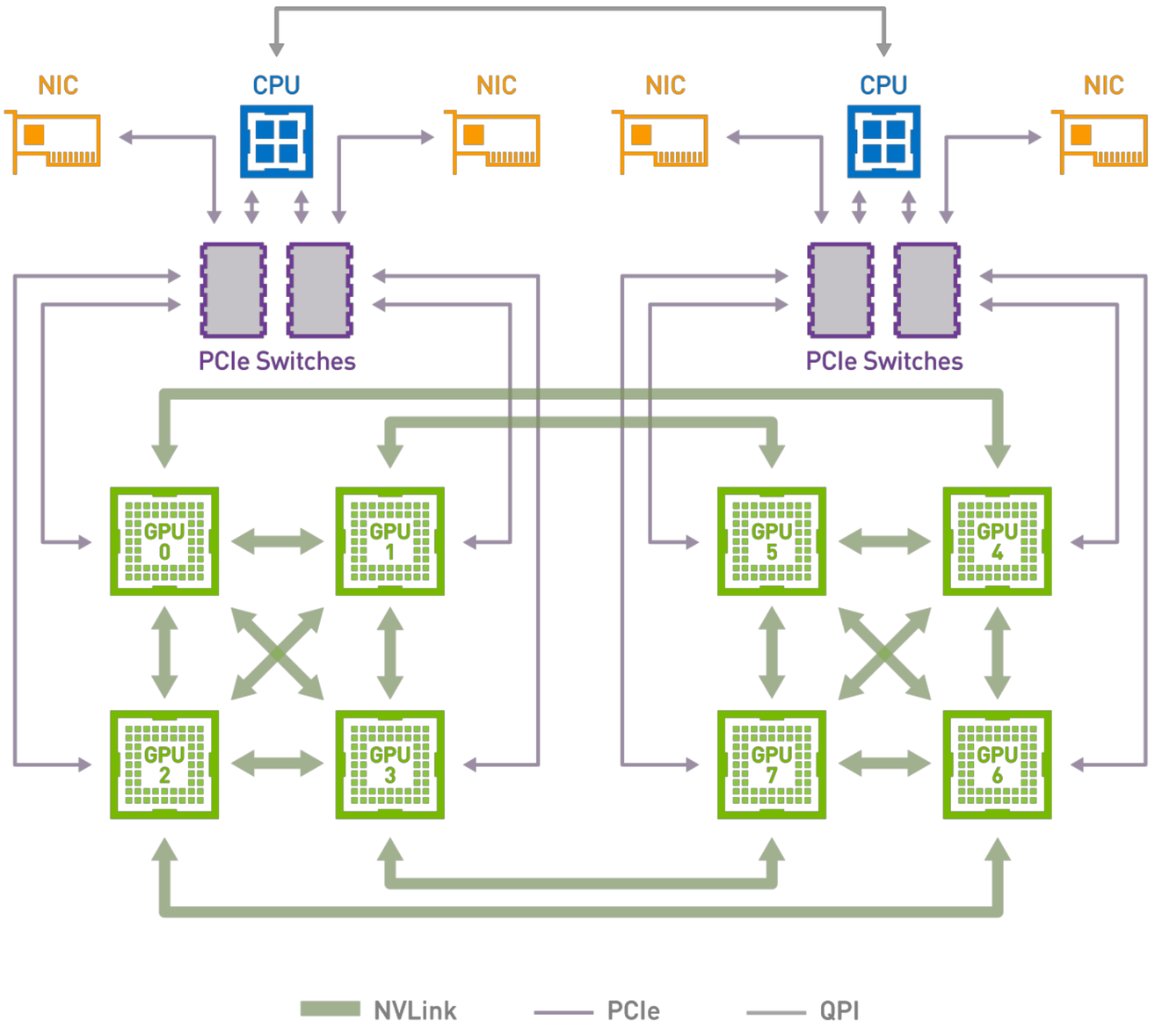
GPU – GPU link is NVLink

Cube mesh topology

not all-to-all

GPUDirect RDMA:

GPU – NIC link is PCIe



NVIDIA NCCL

BACKGROUND

What limits the scalability of parallel applications?

Efficiency of parallel computation tasks

- Amount of exposed parallelism
- Amount of work assigned to each processor

Expense of communications among tasks

- Amount of communication
- Degree of overlap of communication with computation

COMMUNICATION AMONG TASKS

What are common communication patterns?

Point-to-point communication

- Single sender, single receiver
- Relatively easy to implement efficiently

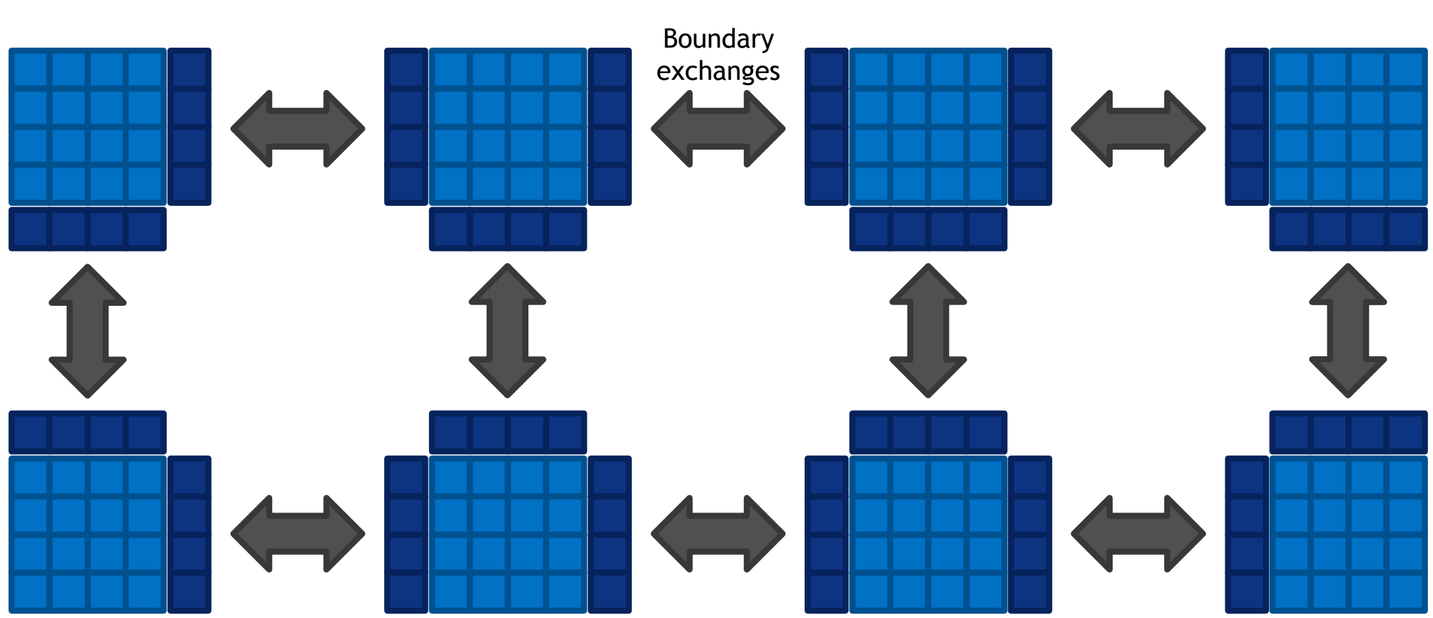
Collective communication

- Multiple senders and/or receivers
- Patterns include broadcast, scatter, gather, reduce, all-to-all, …
- Difficult to implement efficiently

POINT-TO-POINT COMMUNICATION

Single-sender, single-receiver per instance

Most common pattern in HPC, where communication is usually to nearest neighbors

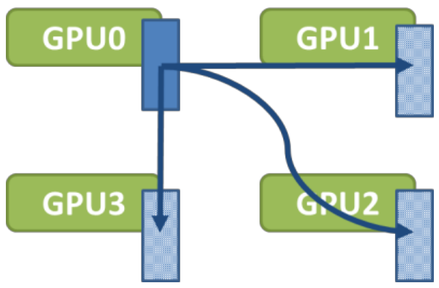


2D Decomposition

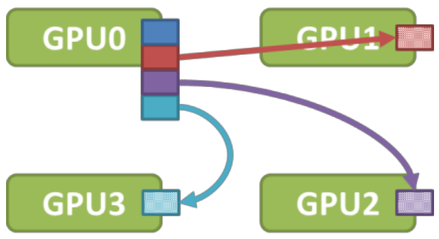
COLLECTIVE COMMUNICATION

Multiple senders and/or receivers

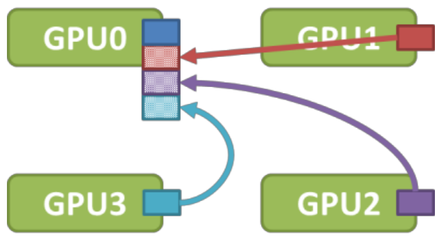
Broadcast



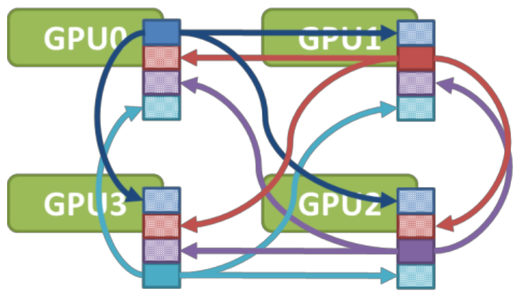
Scatter



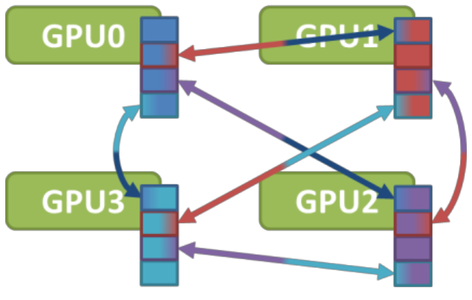
Gather



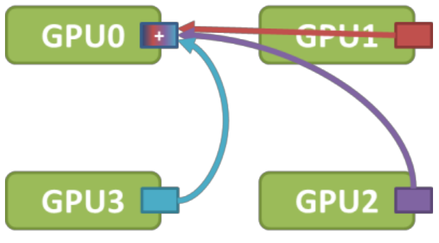
All-Gather



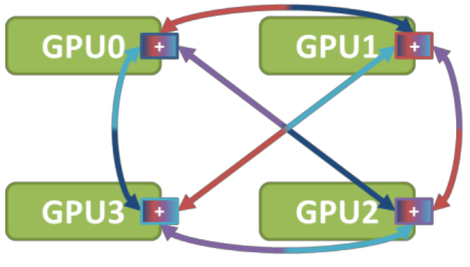
All-to-All



Reduce



All-Reduce



THE CHALLENGE OF COLLECTIVES

Collectives are often avoided because they are expensive. Why?

Having multiple senders and/or receivers compounds communication inefficiencies

- For small transfers, latencies dominate; more participants increase latency
- For large transfers, bandwidth is key; bottlenecks are easily exposed
- May require topology-aware implementation for high performance
- Collectives are often blocking/non-overlapped

THE CHALLENGE OF COLLECTIVES

If collectives are so expensive, do they actually get used? YES!

Collectives are central to scalability in a variety of key applications:

- Deep Learning (All-reduce, broadcast, gather)
- Parallel FFT (Transposition is all-to-all)
- Molecular Dynamics (All-reduce)
- Graph Analytics (All-to-all)
- …

NCCL FEATURES AND FUTURES

(Green = Currently available)

Collectives

- Broadcast
- All-Gather
- Reduce
- All-Reduce
- Reduce-Scatter
- Scatter
- Gather
- All-To-All
- Neighborhood

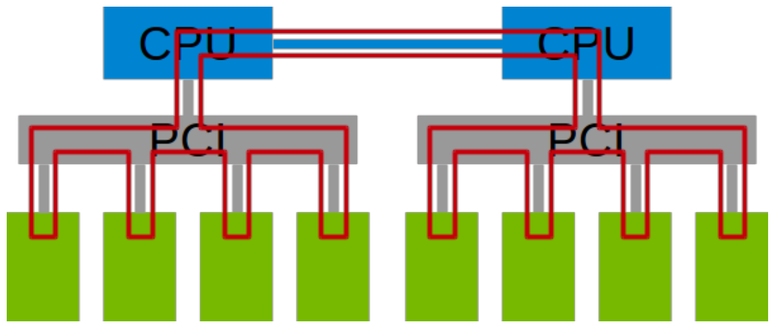
Key Features

- Single-node, up to 8 GPUs
- Host-side API
- Asynchronous/non-blocking interface
- Multi-thread, multi-process support
- In-place and out-of-place operation
- Integration with MPI
- Topology Detection
- NVLink & PCIe/QPI* support

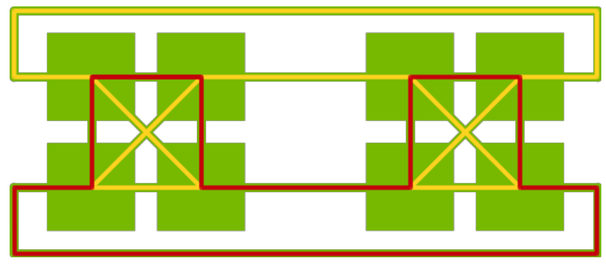
DESIGN

Rings

NCCL uses rings to move data across all GPUs and perform reductions.



PCIe / QPI : 1 unidirectional ring



DGX-1 : 4 unidirectional rings

NCCL EXAMPLE

All-reduce

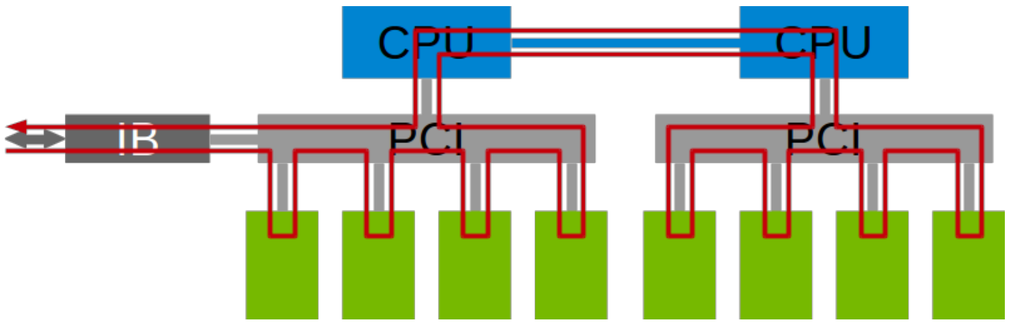
```
#include <nccl.h>
ncclComm_t comm[4];
ncclCommInitAll(comm, 4, {0, 1, 2, 3});

foreach g in (GPUs) { // or foreach thread
  cudaSetDevice(g);
  double *d_send, *d_recv;
  // allocate d_send, d_recv; fill d_send with data
  ncclAllReduce(d_send, d_recv, N, ncclDouble, ncclSum, comm[g], stream[g]);
  // consume d_recv
}
```

NCCL 2.0

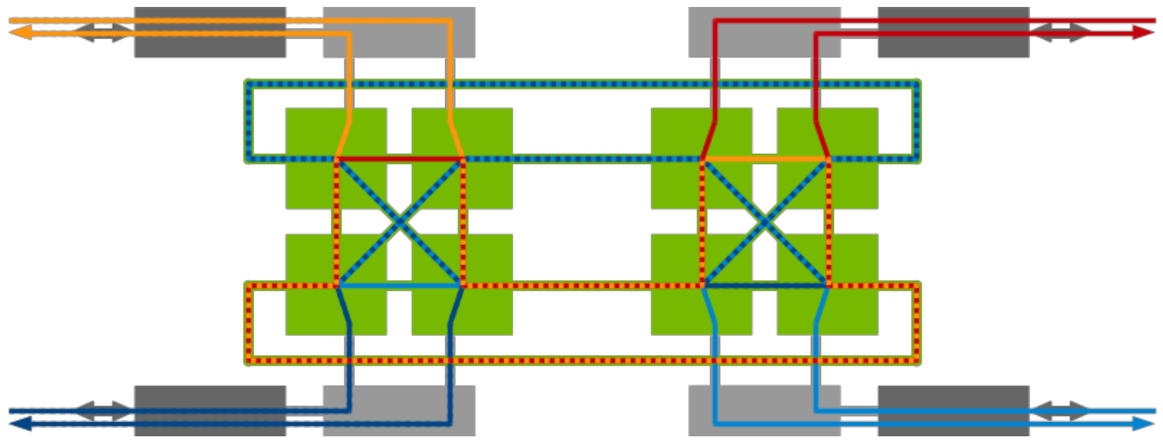
Inter-node communication

Inter-node communication using Sockets or Infiniband verbs, with multi-rail support, topology detection and automatic use of GPU Direct RDMA.

Optimal combination of NVLink, PCI and network interfaces to maximize bandwidth and create rings across nodes.



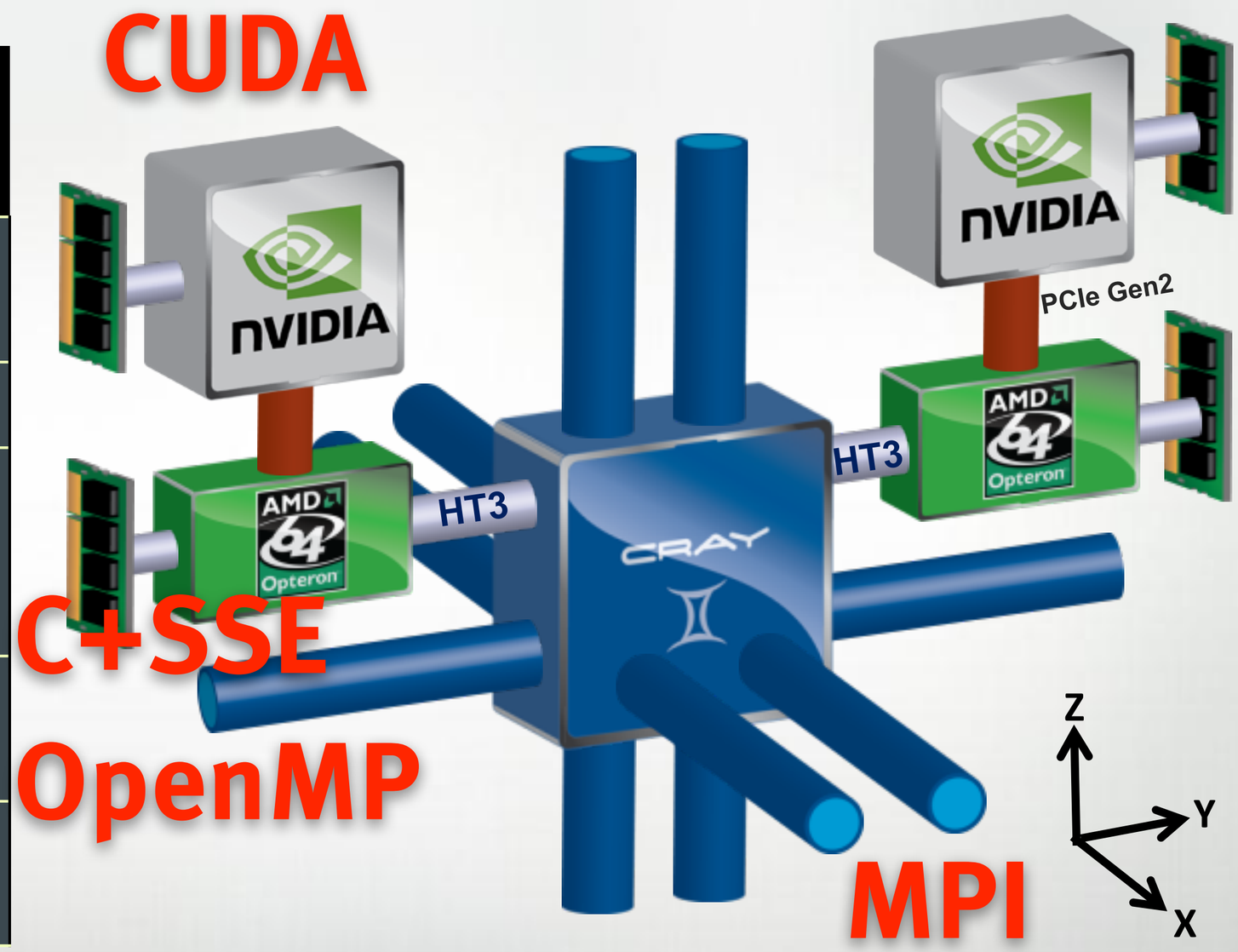
PCIe, Infiniband



DGX-1 : NVLink, 4x Infiniband

Cray XK7 Compute Node

XK7 Compute Node Characteristics
AMD Opteron 6274 16 core processor - 141 GF
Tesla K20x - 1311 GF
Host Memory 32GB 1600 MHz DDR3
Tesla K20x Memory 6GB GDDR5
Gemini High Speed Interconnect



Slide courtesy of Cray, Inc.

NVIDIA Volta

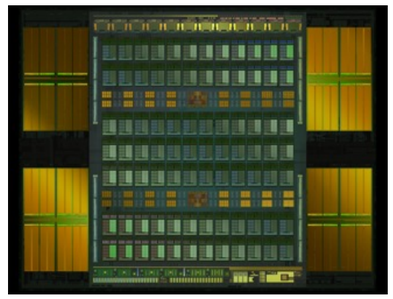
General trend

- "Fatter" nodes
  - Titan: 2 CPUs/node, 2 GPUs/node
  - CORAL (currently):
    - 2 10-core Power8+ CPUs
    - 4 Tesla P100 GPUs (now)
    - 4 Tesla V100 GPUs (in Sierra)

INTRODUCING CUDA 9

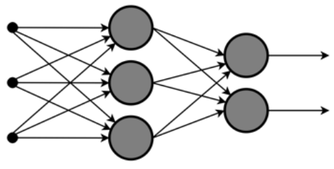
BUILT FOR VOLTA

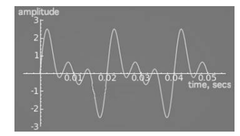
Tesla V100
New GPU Architecture
Tensor Cores
NVLink
Independent Thread Scheduling



FASTER LIBRARIES

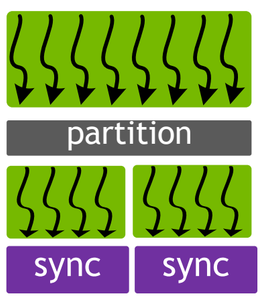
cuBLAS for Deep Learning
NPP for Image Processing
cuFFT for Signal Processing





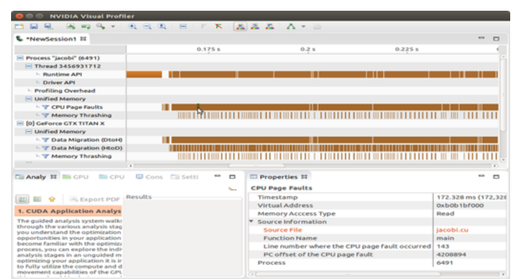
COOPERATIVE THREAD GROUPS

Flexible Thread Groups
Efficient Parallel Algorithms
Synchronize Across Thread Blocks in a Single GPU or Multi-GPUs



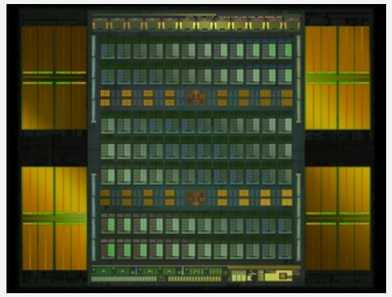
DEVELOPER TOOLS & PLATFORM UPDATES

Faster Compile Times
Unified Memory Profiling
NVLink Visualization
New OS and Compiler Support



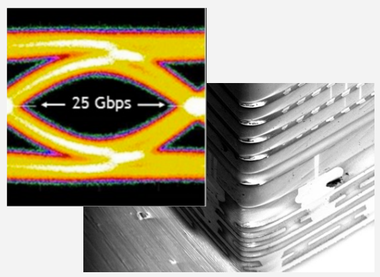
INTRODUCING TESLA V100

Volta Architecture



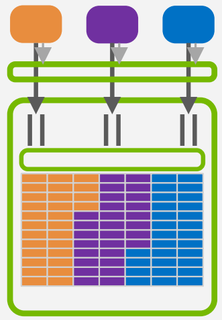
Most Productive GPU

Improved NVLink & HBM2



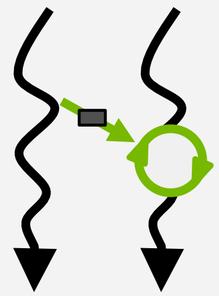
Efficient Bandwidth

Volta MPS



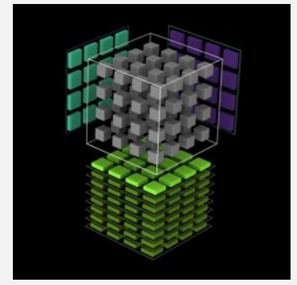
Inference Utilization

Improved SIMT Model



New Algorithms

Tensor Core

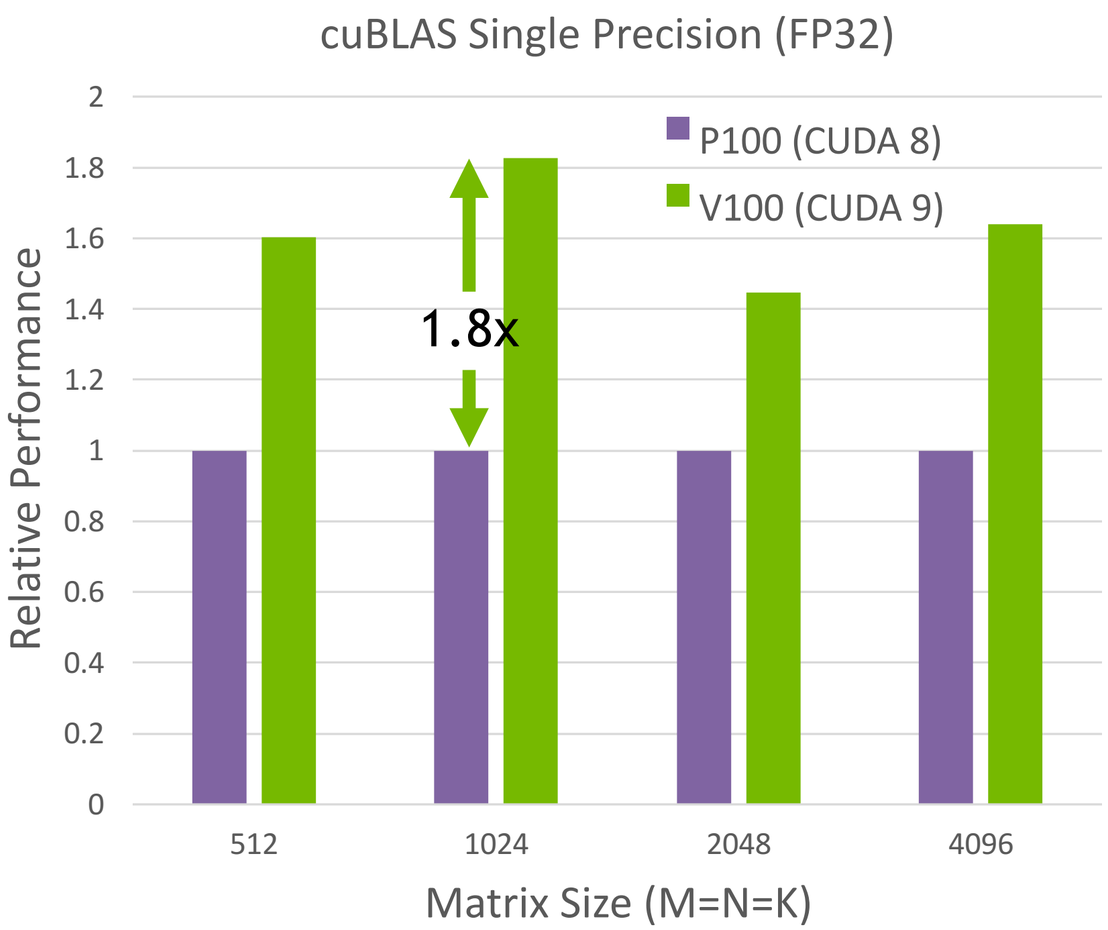


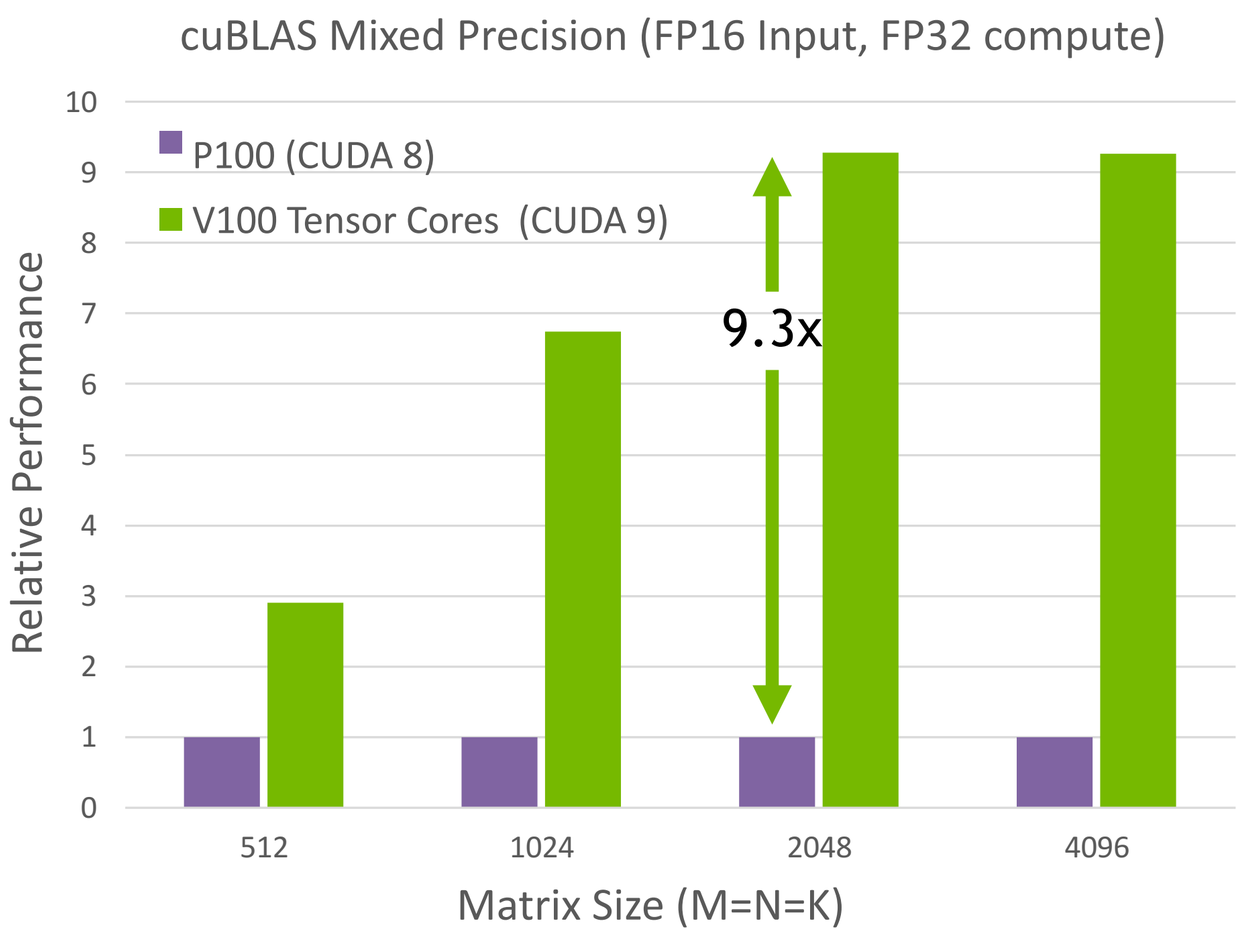
125 Programmable TFLOPS Deep Learning

The Fastest and Most Productive GPU for Deep Learning and HPC

cuBLAS GEMMS FOR DEEP LEARNING

V100 Tensor Cores + CUDA 9: over 9x Faster Matrix-Matrix Multiply





Note: pre-production Tesla V100 and pre-release CUDA 9. CUDA 8 GA release.

TESLA V100

21B transistors
815 mm²

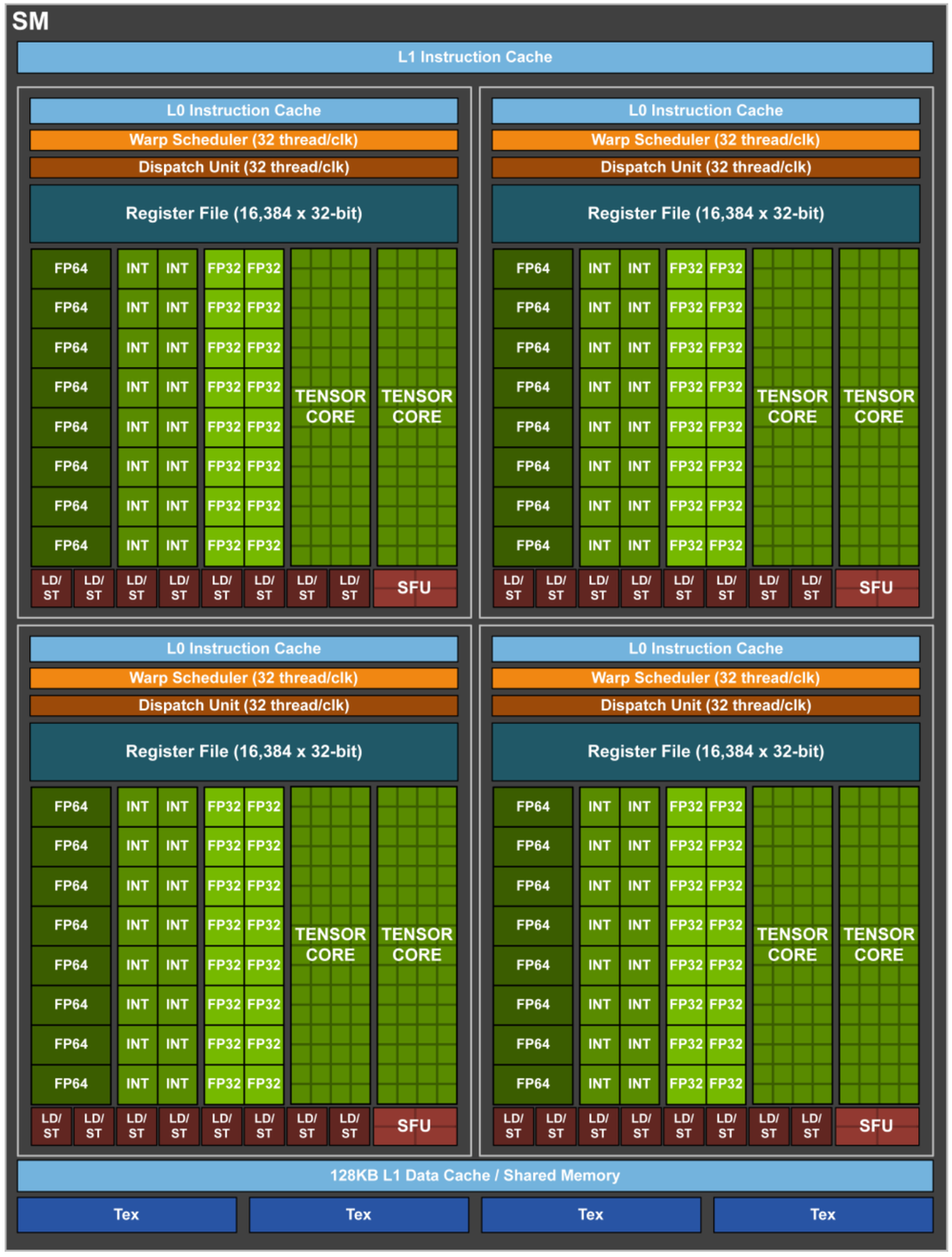
80 SM
5120 CUDA Cores
640 Tensor Cores

16 GB HBM2
900 GB/s HBM2
300 GB/s NVLink

*full GV100 chip contains 84 SMs

VOLTA GV100 SM

	GV100
FP32 units	64
FP64 units	32
INT32 units	64
Tensor Cores	8
Register File	256 KB
Unified L1/Shared memory	128 KB
Active Threads	2048



VOLTA GV100 SM

Redesigned for Productivity

Completely new ISA

Twice the schedulers

Simplified Issue Logic

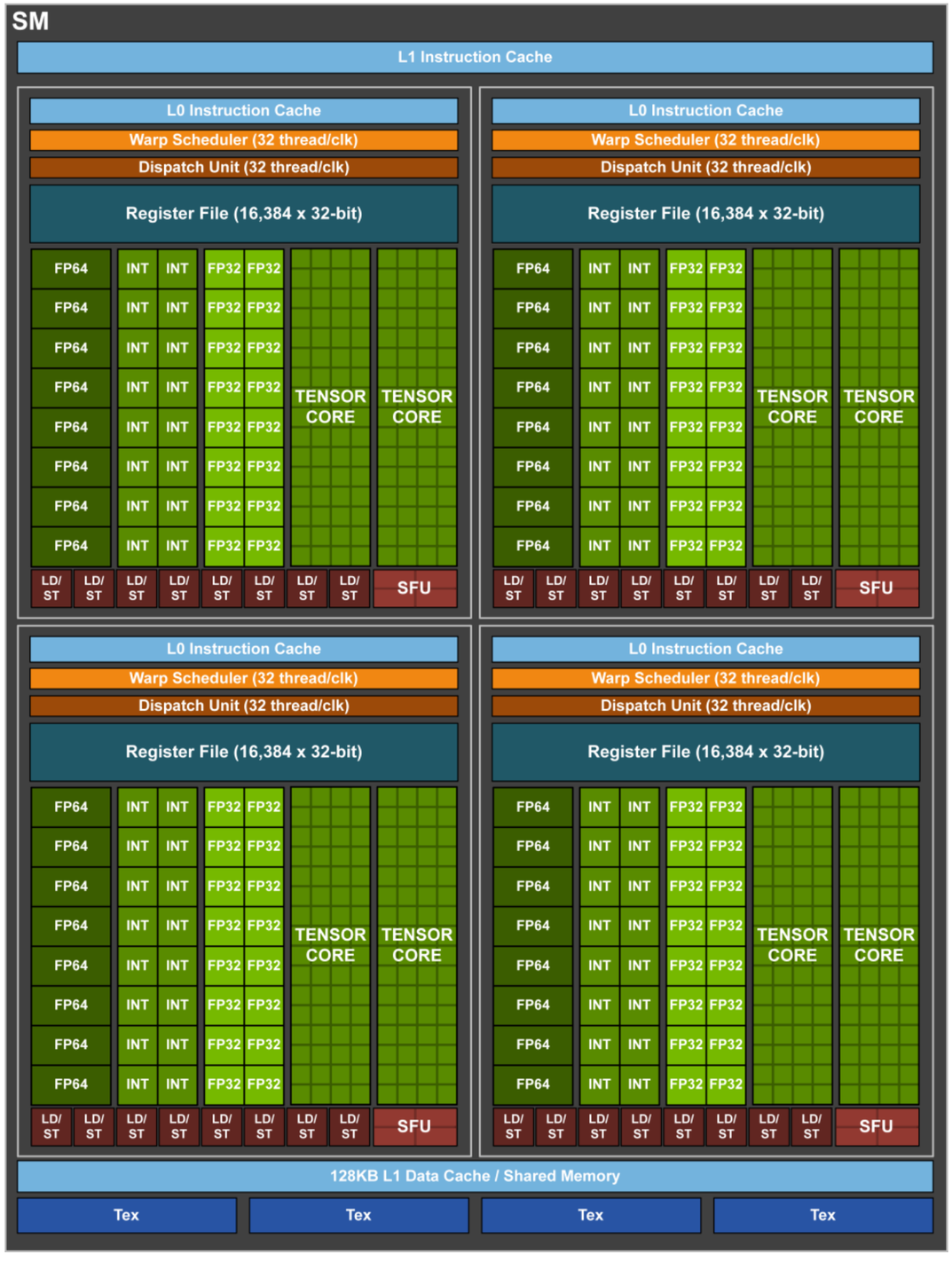
Large, fast L1 cache

Improved SIMT model

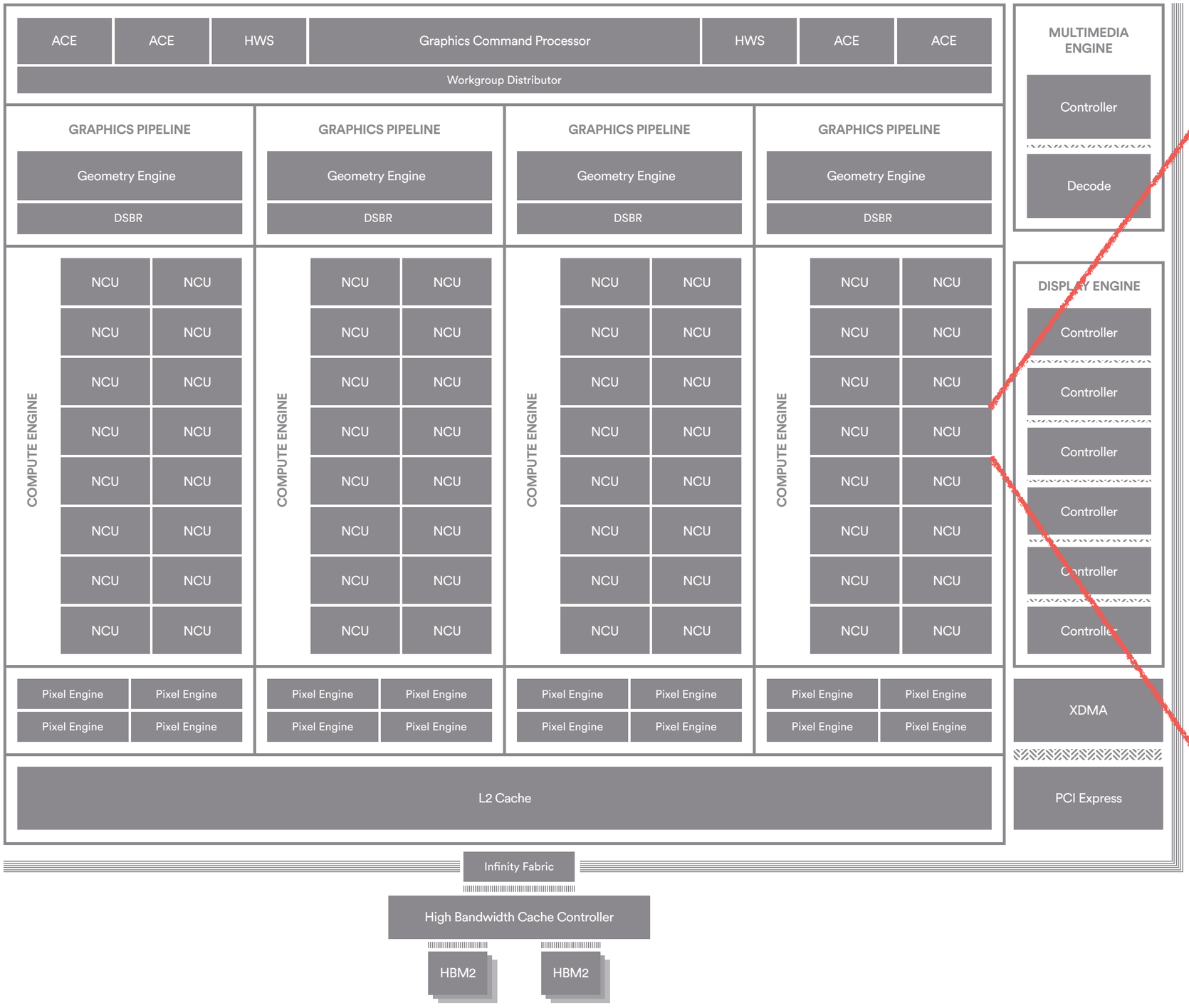
Tensor acceleration

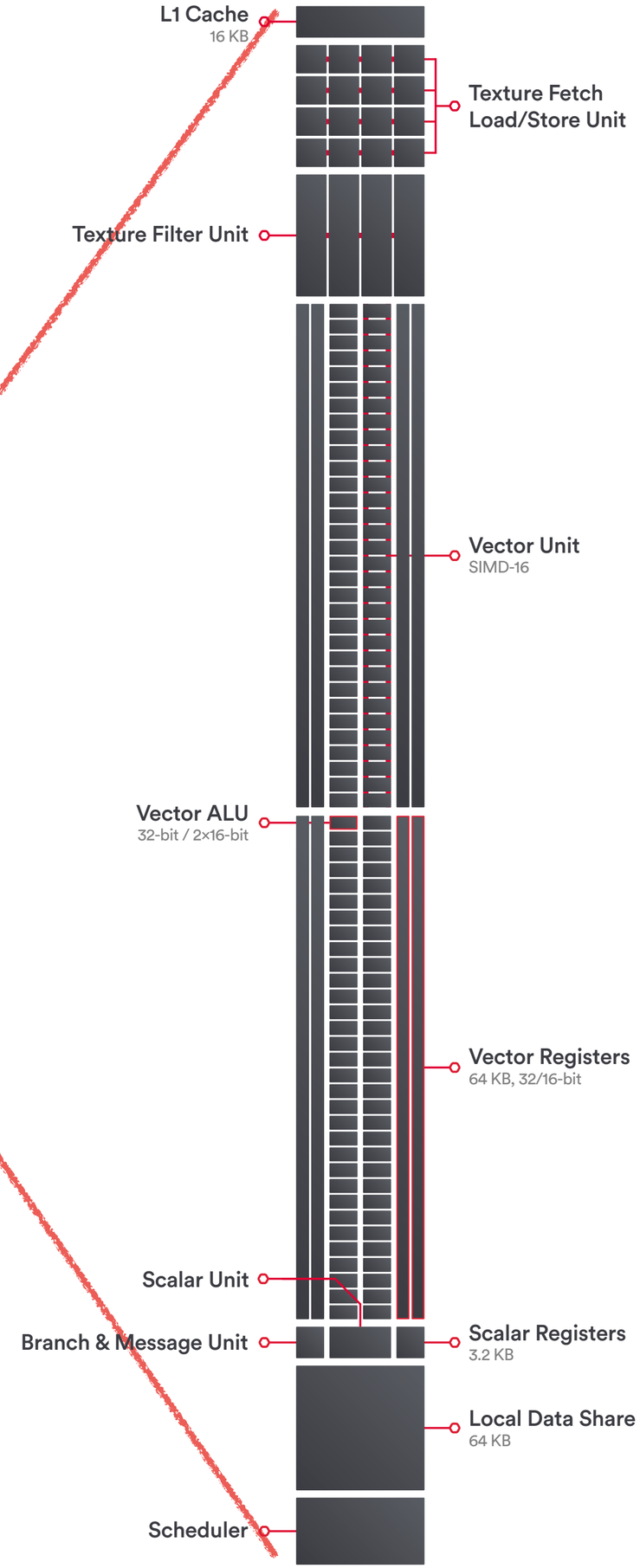
=

The easiest SM to program yet



Vega block diagram





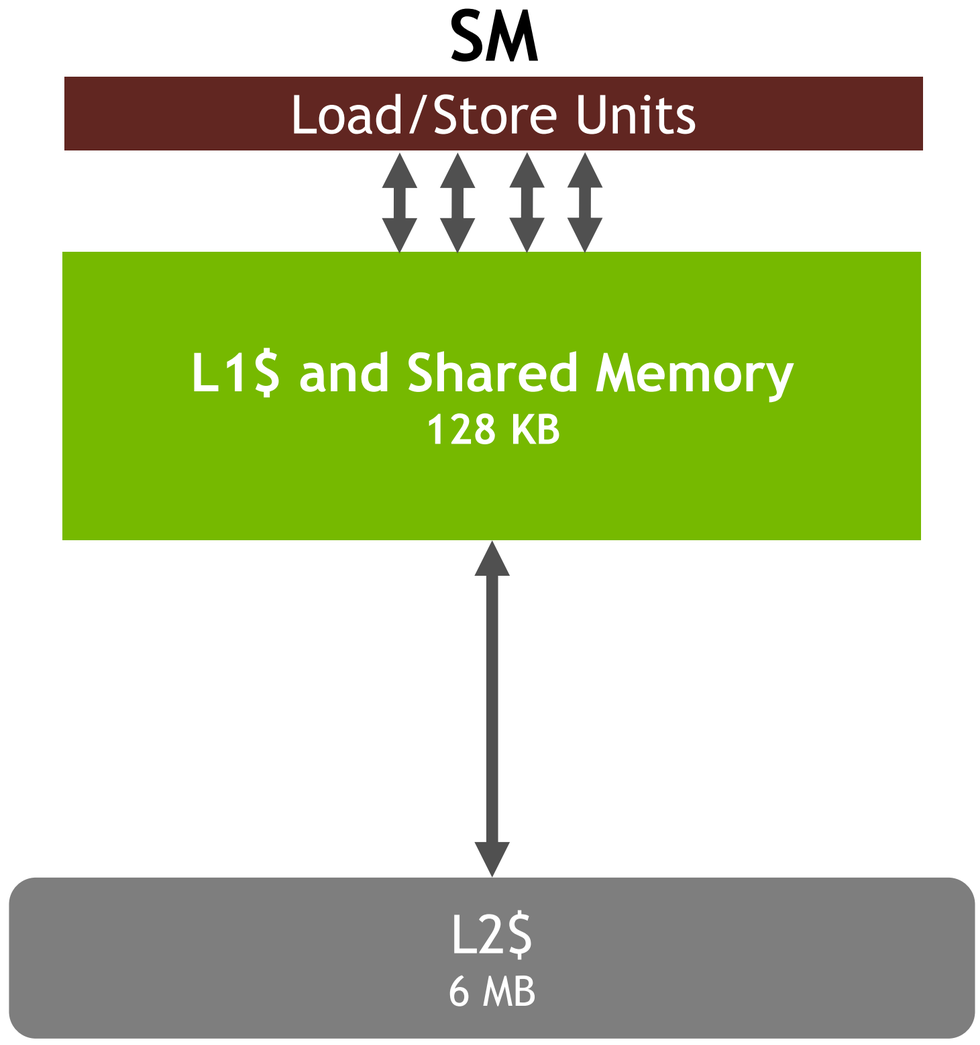
VOLTA L1 AND SHARED MEMORY

Volta Streaming L1$:

Unlimited cache misses in flight
Low cache hit latency
4x more bandwidth
5x more capacity

Volta Shared Memory :

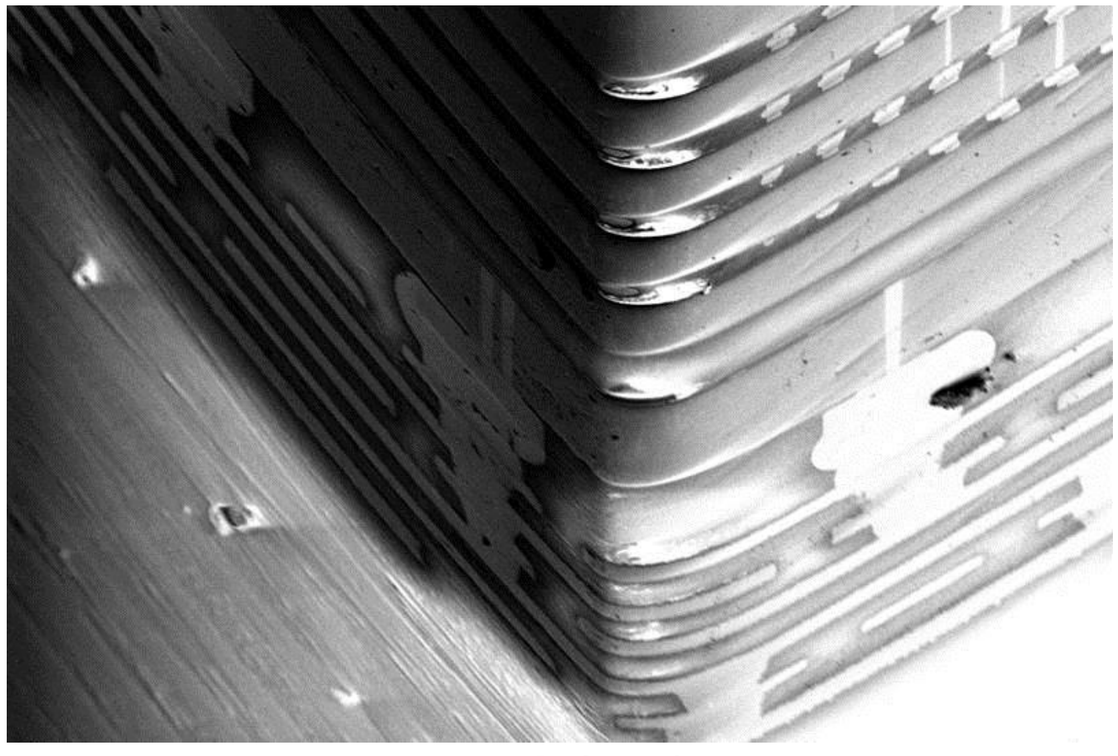
Unified storage with L1
Configurable up to 96KB



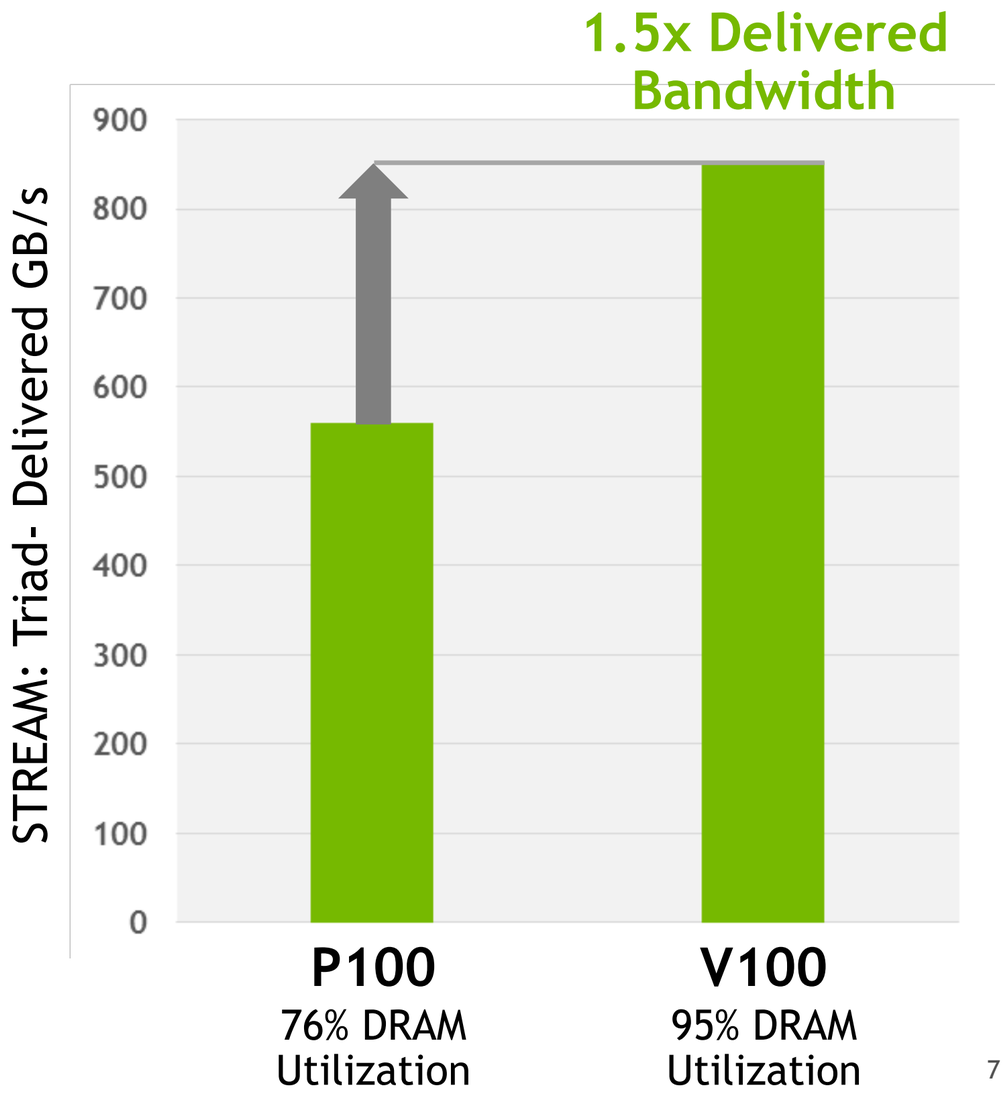
GPU PERFORMANCE COMPARISON

	P100	V100	Ratio
Training acceleration	10 TOPS	120 TOPS	12x
Inference acceleration	21 TFLOPS	120 TOPS	6x
FP64/FP32	5/10 TFLOPS	7.5/15 TFLOPS	1.5x
HBM2 Bandwidth	720 GB/s	900 GB/s	1.2x
NVLink Bandwidth	160 GB/s	300 GB/s	1.9x
L2 Cache	4 MB	6 MB	1.5x
L1 Caches	1.3 MB	10 MB	7.7x

NEW HBM2 MEMORY ARCHITECTURE



HBM2 stack



V100 measured on pre-production hardware.

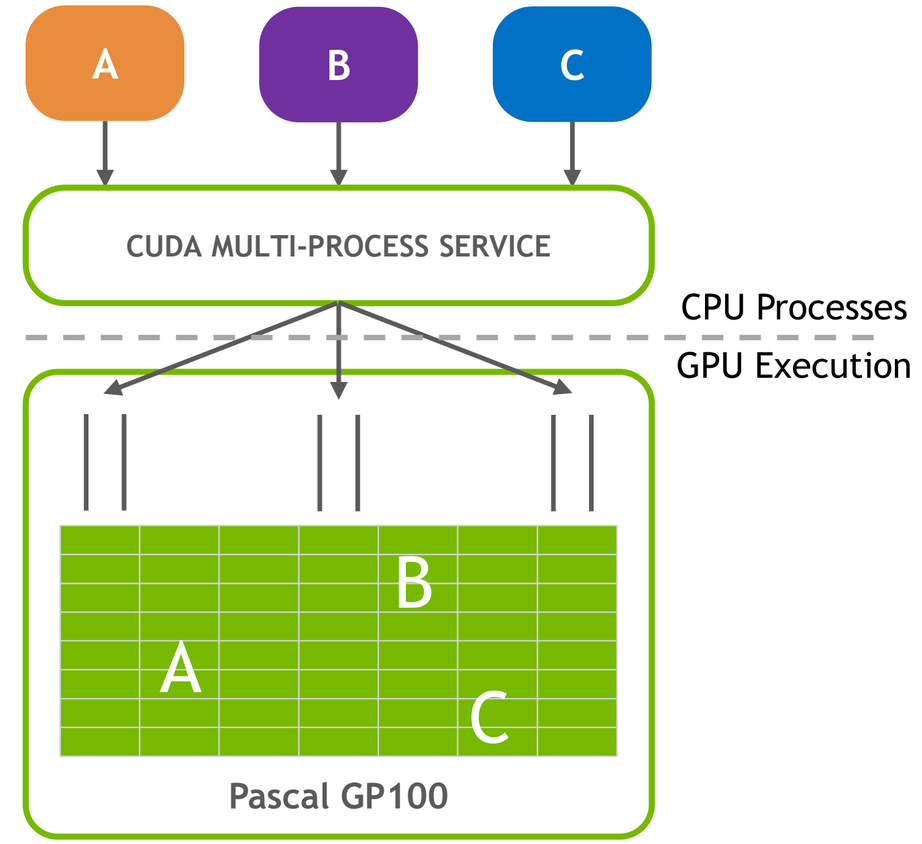
PASCAL MULTI-PROCESS SERVICE

CUDA Multi-Process Service:

Improve GPU utilization by sharing resources amongst small jobs

Software Work Submission

Limited Isolation



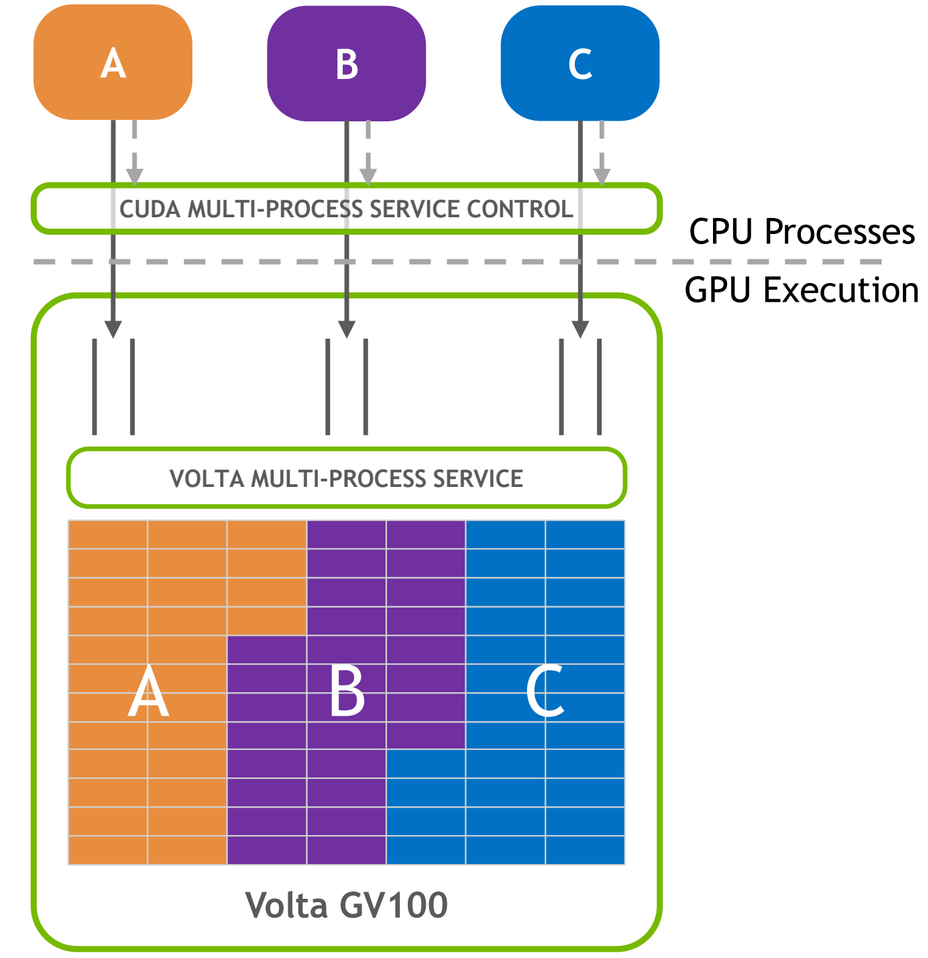
Opt-in: Limited isolation, peak throughput optimized **across** processes

VOLTA MULTI-PROCESS SERVICE

Volta MPS Enhancements:

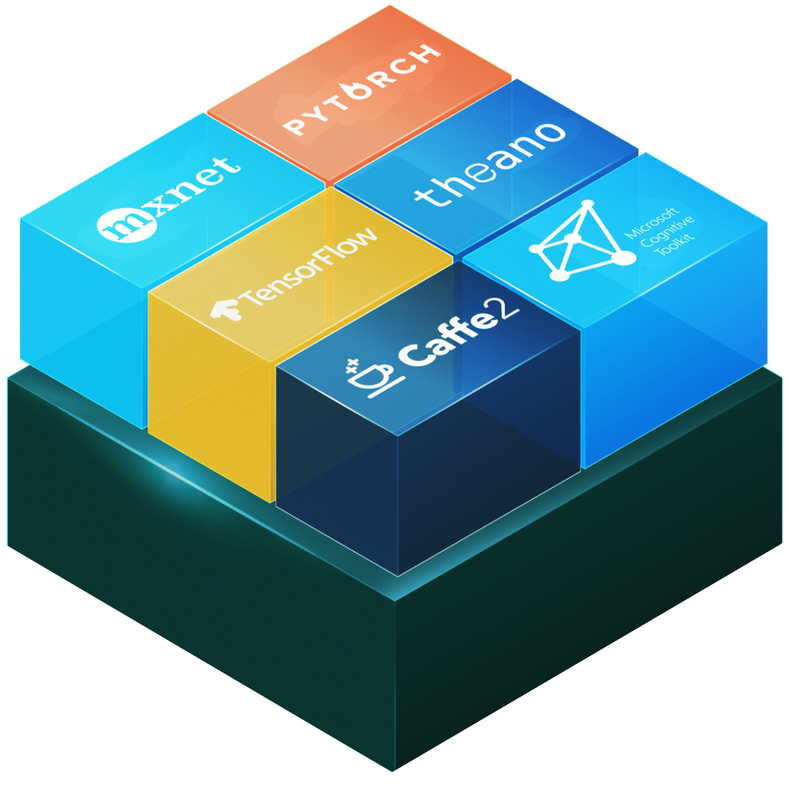
- Reduced launch latency
- Improved launch throughput
- Improved quality of service with scheduler partitioning
  - More reliable performance
- 3x more clients than Pascal

Hardware Accelerated Work Submission



Hardware Isolation

USING TENSOR CORES



NVIDIA cuDNN, cuBLAS, TensorRT

Volta Optimized Frameworks and Libraries

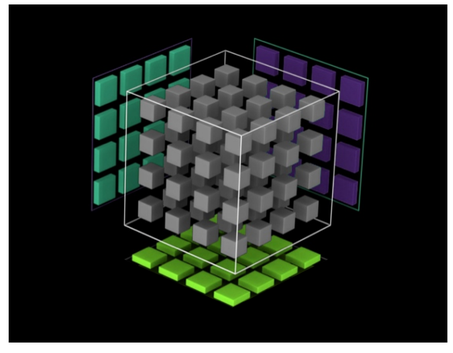
```
__device__ void tensor_op_16_16_16(
    float *d, half *a, half *b, float *c)
{
  wmma::fragment<matrix_a, …> Amat;
  wmma::fragment<matrix_b, …> Bmat;
  wmma::fragment<matrix_c, …> Cmat;

  wmma::load_matrix_sync(Amat, a, 16);
  wmma::load_matrix_sync(Bmat, b, 16);
  wmma::fill_fragment(Cmat, 0.0f);

  wmma::mma_sync(Cmat, Amat, Bmat, Cmat);

  wmma::store_matrix_sync(d, Cmat, 16,
                          wmma::row_major);
}
```

CUDA C++

Warp-Level Matrix Operations

TENSOR CORE

Mixed Precision Matrix Math
4x4 matrices



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

FP16 or FP32 — FP16 — FP16 — FP16 or FP32

$$D = AB + C$$

COOPERATIVE GROUPS

COOPERATIVE GROUPS

Flexible and Scalable Thread Synchronization and Communication

Define, synchronize, and partition groups of cooperating threads

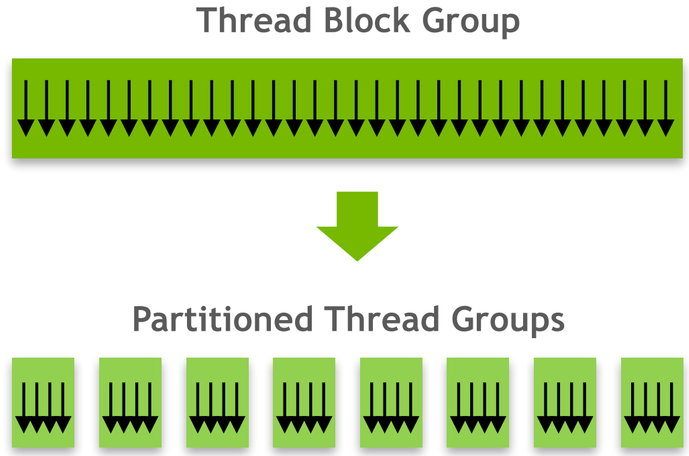
Clean composition across software boundaries

Optimize for hardware fast path

Scalable from a few threads to all running threads

Deploy Everywhere: Kepler and Newer GPUs

Supported by CUDA developer tools

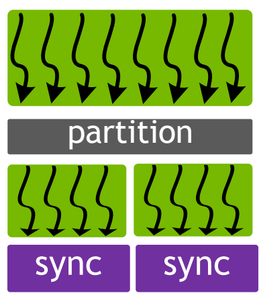


SYNCHRONIZE AT ANY SCALE

Three Key Capabilities

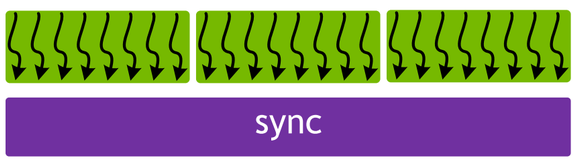
FLEXIBLE GROUPS

Define and Synchronize Arbitrary Groups of Threads

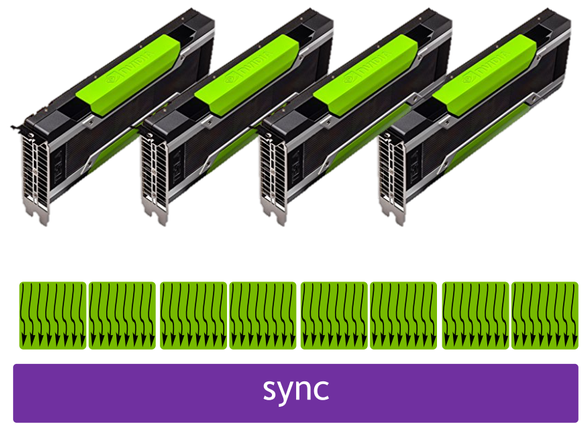


WHOLE-GRID SYNCHRONIZATION

Synchronize Multiple Thread Blocks



MULTI-GPU SYNCHRONIZATION



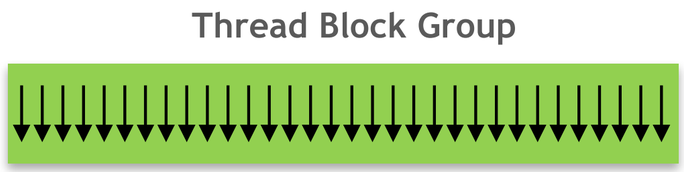
* Note: Multi-Block and Mult-Device Cooperative Groups are only supported on Pascal and above GPUs

COOPERATIVE GROUPS BASICS

Flexible, Explicit Synchronization

Thread groups are explicit objects in your program

```
thread_group block = this_thread_block();
```

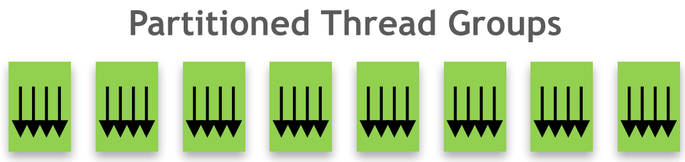


You can synchronize threads in a group

```
block.sync();
```

Create new groups by partitioning existing groups

```
thread_group tile32 = tiled_partition(block, 32);
thread_group tile4 = tiled_partition(tile32, 4);
```



Partitioned groups can also synchronize

```
tile4.sync();
```

Note: calls in green are part of the cooperative_groups:: namespace

EXAMPLE: PARALLEL REDUCTION

Composable, Robust and Efficient

Per-Block

```
g = this_thread_block();
reduce(g, ptr, myVal);
```

Per-Warp

```
g = tiled_partition<32>(this_thread_block());
reduce(g, ptr, myVal);
```

```
__device__ int reduce(thread_group g, int *x, int val) {
  int lane = g.thread_rank();
  for (int i = g.size()/2; i > 0; i /= 2) {
    x[lane] = val;        g.sync();
    val += x[lane + i];   g.sync();
  }
  return val;
}
```

LAUNCHING COOPERATIVE KERNELS

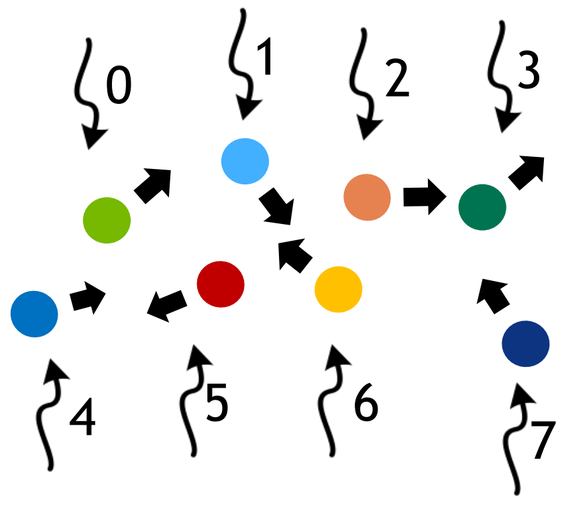
Three Synchronization Scales

Block or Sub-Block Sync	Launch with <<<>>> or `cudaLaunchKernel()`
Multi-Block Sync	Launch with `cudaLaunchCooperativeKernel()`
Multi-Device Sync	Launch with `cudaLaunchCooperativeKernelMultiDevice()`

EXAMPLE: PARTICLE SIMULATION

Without Cooperative Groups

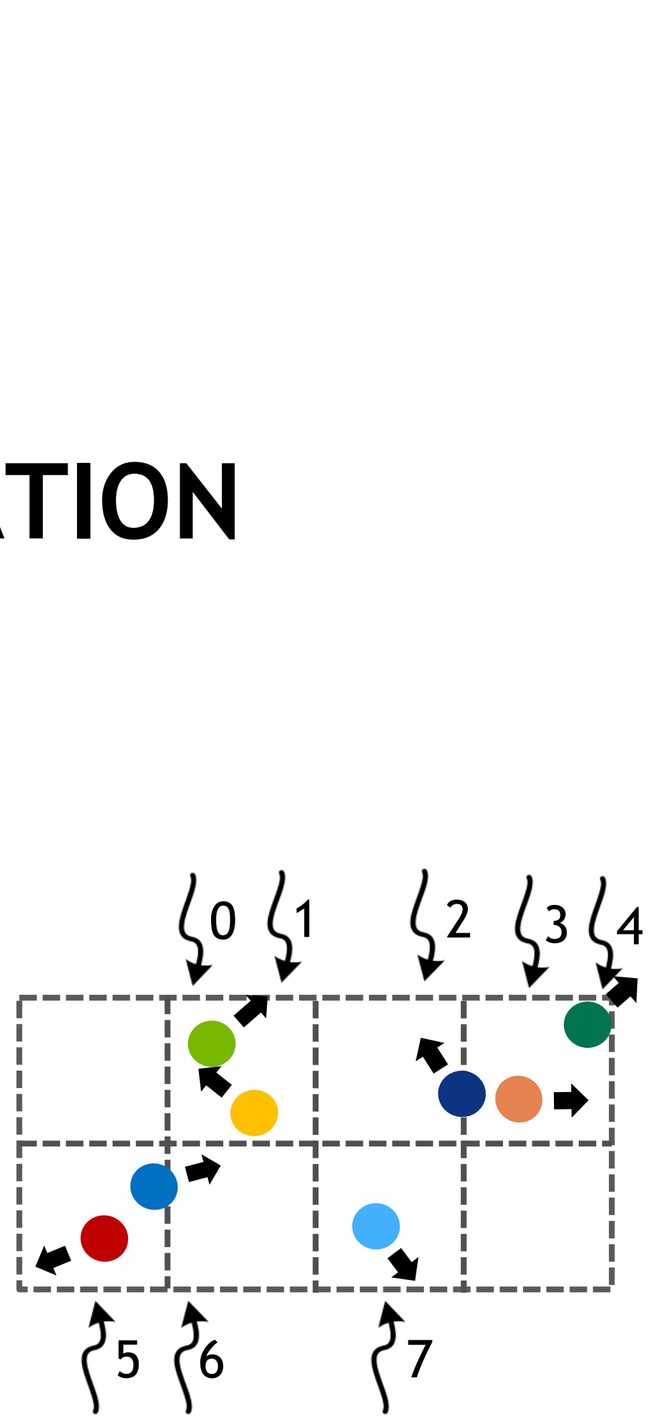
```
// threads update particles in parallel
integrate<<<blocks, threads, 0, stream>>>(particles);
```



EXAMPLE: PARTICLE SIMULATION

Without Cooperative Groups

```
// threads update particles in parallel
integrate<<<blocks, threads, 0, s>>>(particles);



// Collide each particle with others in neighborhood
collide<<<blocks, threads, 0, s>>>(particles);
```

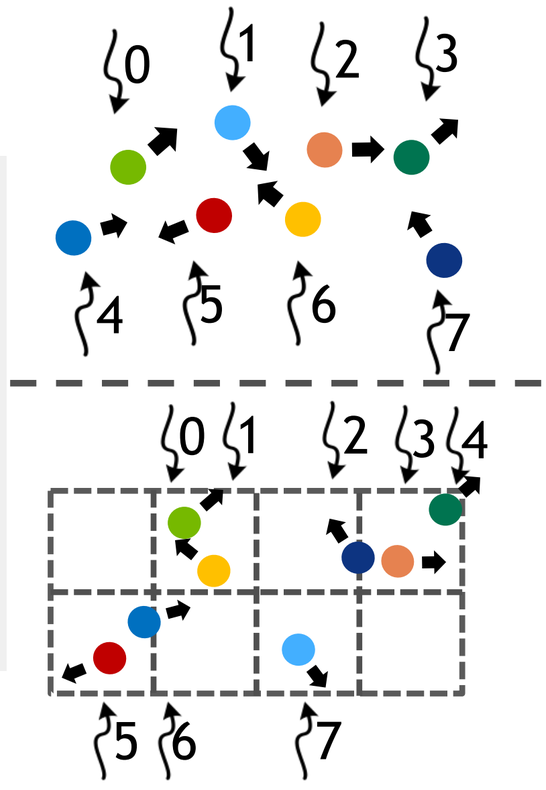


Note change in how threads map to particles in acceleration data structure

EXAMPLE: PARTICLE SIMULATION

Without Cooperative Groups

```
// threads update particles in parallel
integrate<<<blocks, threads, 0, s>>>(particles);

// Note: implicit sync between kernel launches

// Collide each particle with others in neighborhood
collide<<<blocks, threads, 0, s>>>(particles);
```

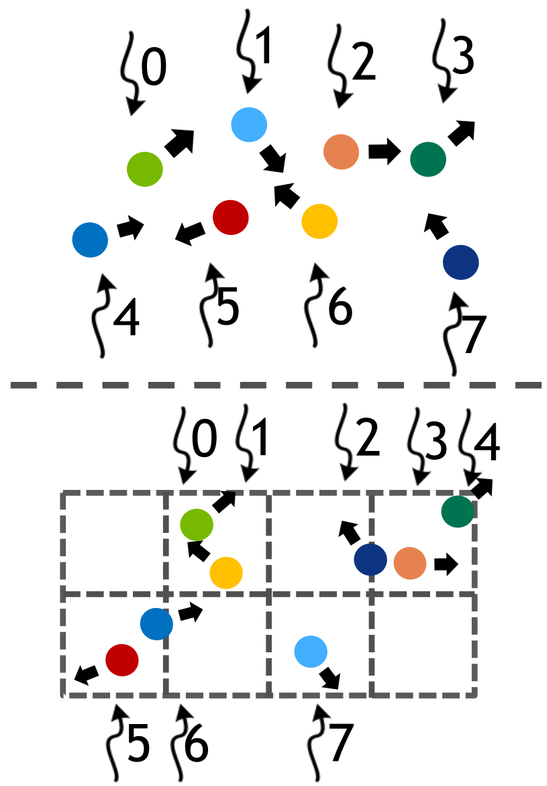


Note change in how threads map to particles in acceleration data structure

WHOLE-GRID COOPERATION

Particle Simulation Update in a Single Kernel

```
__global__ void particleSim(Particle *p, int N) {

  grid_group g = this_grid();

  for (i = g.thread_rank(); i < N; i += g.size())
    integrate(p[i]);

  g.sync() // Sync whole grid!

  for (i = g.thread_rank(); i < N; i += g.size())
    collide(p[i], p, N);
}
```

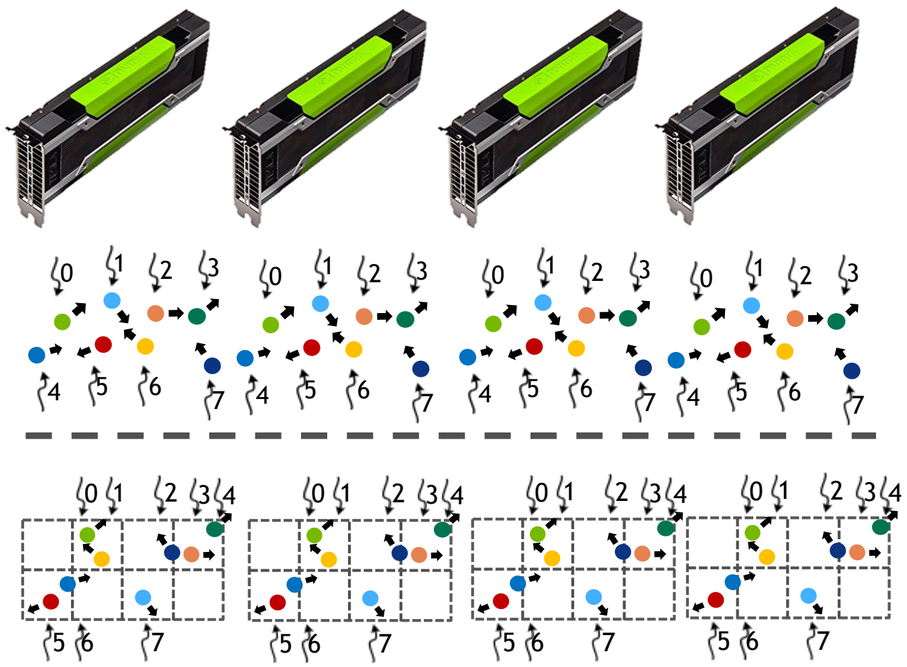


Launch using `cudaLaunchCooperativeKernel(…)`

MULTI-GPU COOPERATION

Large-scale Multi-GPU Simulation in a Single Kernel

```
__global__ void particleSim(Particle *p, int N) {

  multi_grid_group g = this_multi_grid();

  for (i = g.thread_rank(); i < N; i += g.size())
    integrate(p[i]);

  g.sync() // Sync all GPUs!

  for (i = g.thread_rank(); i < N; i += g.size())
    collide(p[i], p, N);
}
```



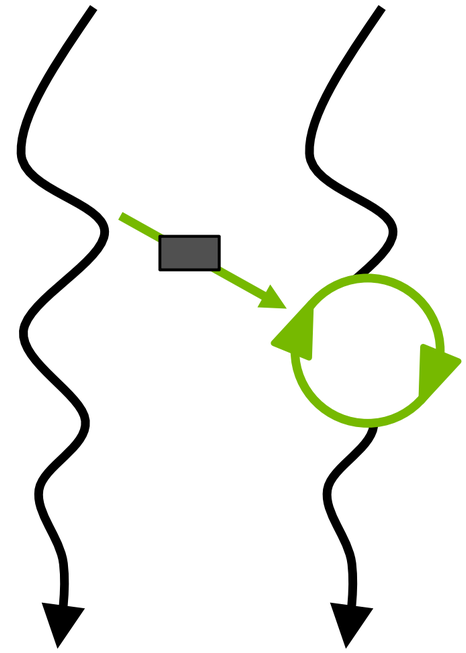
Launch using `cudaLaunchCooperativeKernelMultiDevice(...)`

ROBUST AND EXPLICIT WARP PROGRAMMING

Adapt Legacy Code for New Execution Model

Volta Independent Thread Scheduling:

- Program familiar algorithms and data structures in a natural way
- Flexible thread grouping and synchronization



Use explicit synchronization, don’t rely on implicit convergence

- CUDA 9 provides a fully explicit synchronization model

ROBUST AND EXPLICIT WARP PROGRAMMING

Adapt Legacy Code for New Execution Model

Eliminate *implicit* warp synchronous programming on all architectures

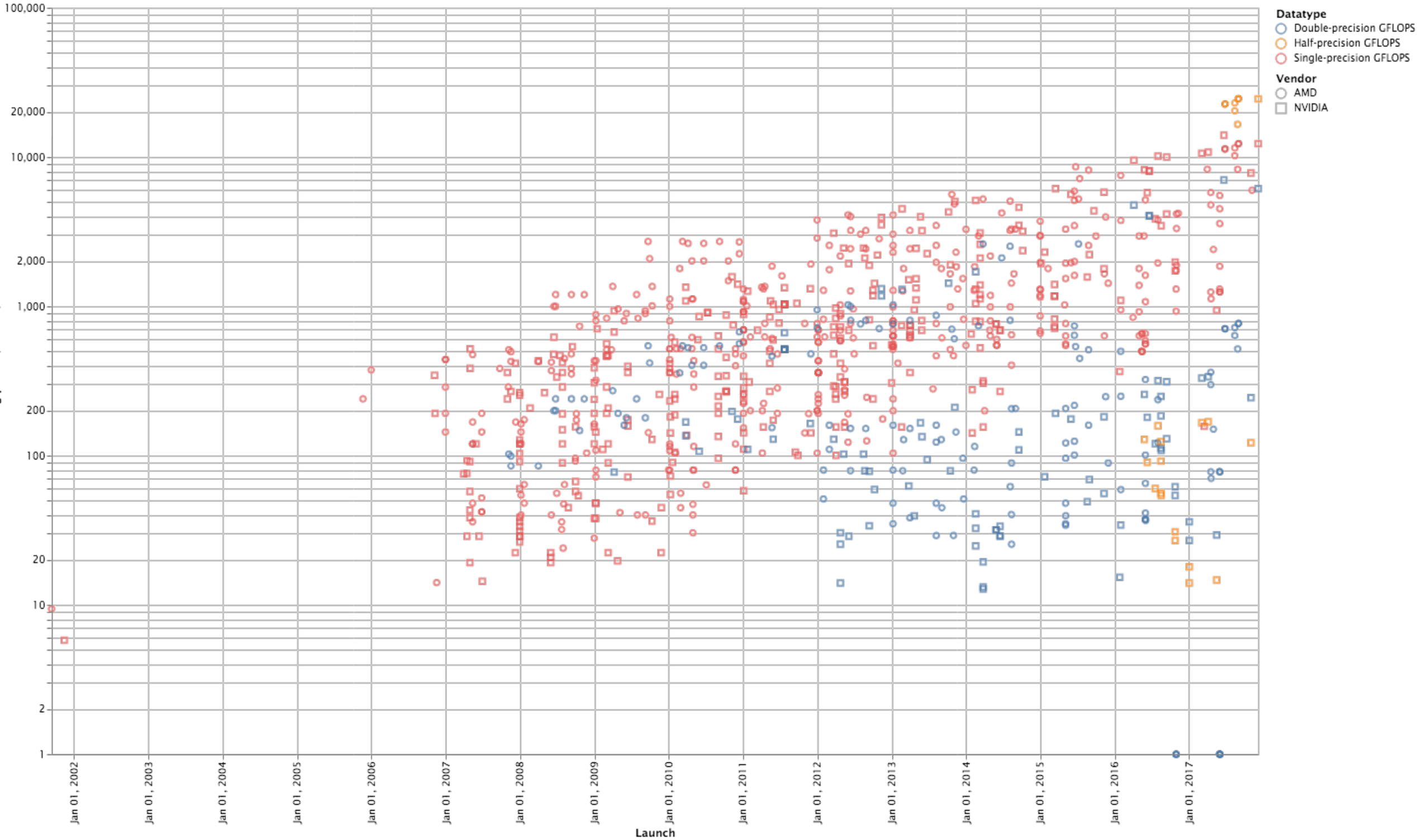
- Use explicit synchronization
- Focus synchronization granularity with Cooperative Groups

Transition to new *_sync() primitives

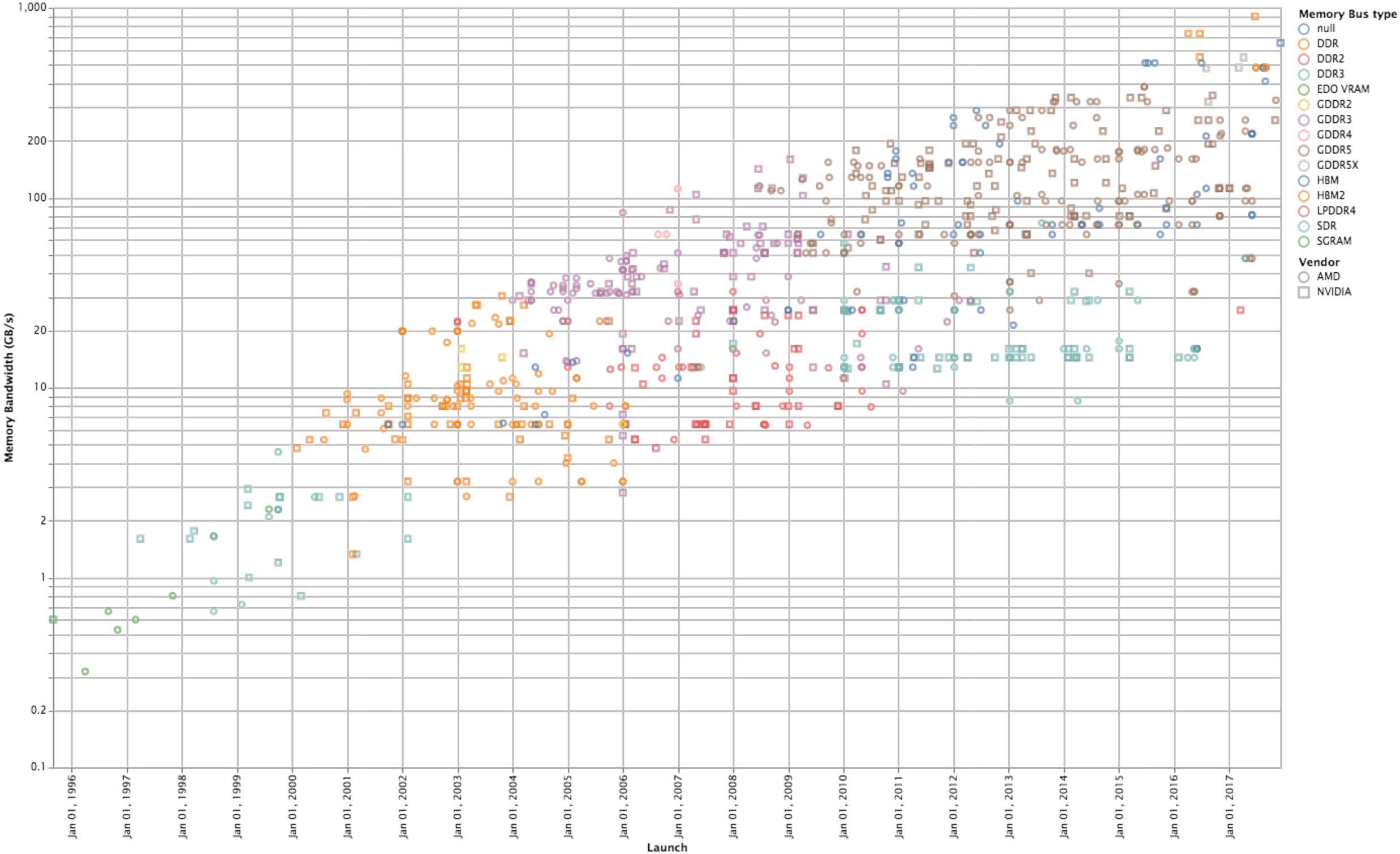
- __shfl_sync(), __ballot_sync(), __any_sync(), __all_sync(), __activemask()
- CUDA 9 deprecates non-synchronizing __shfl(), __ballot(), __any(), __all()

Trends

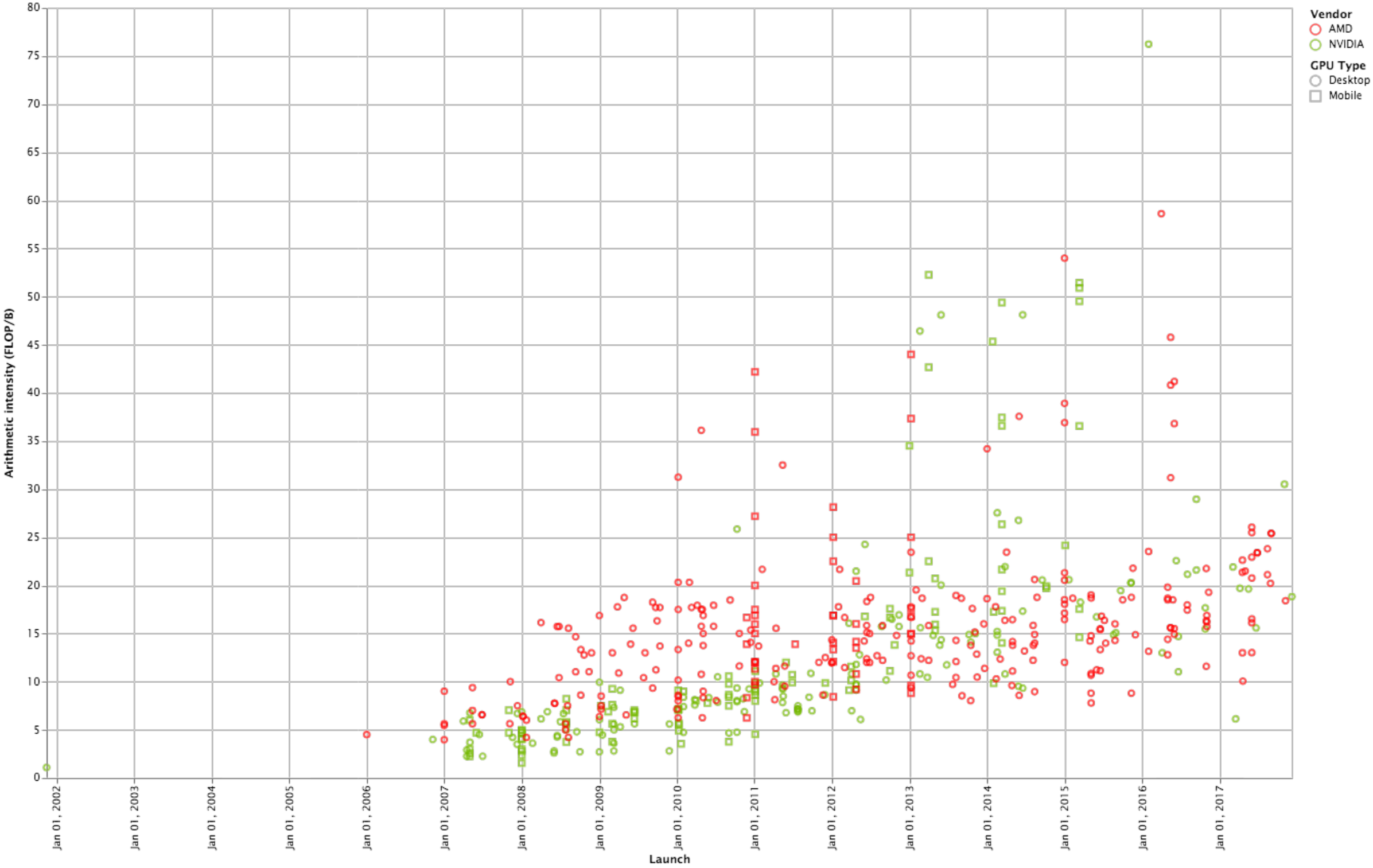
GFLOPS vs. time



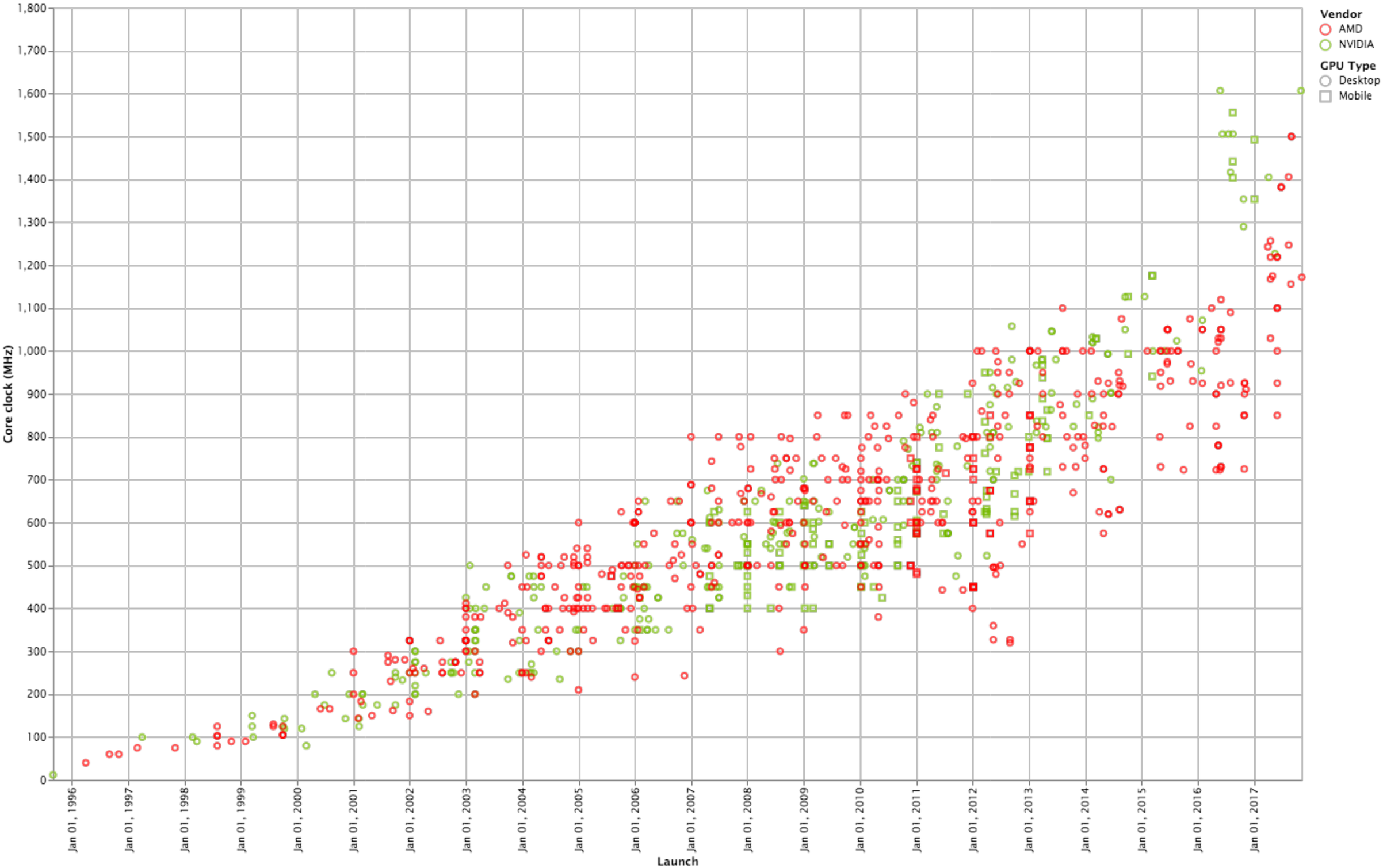
Memory Bandwidth vs. Time



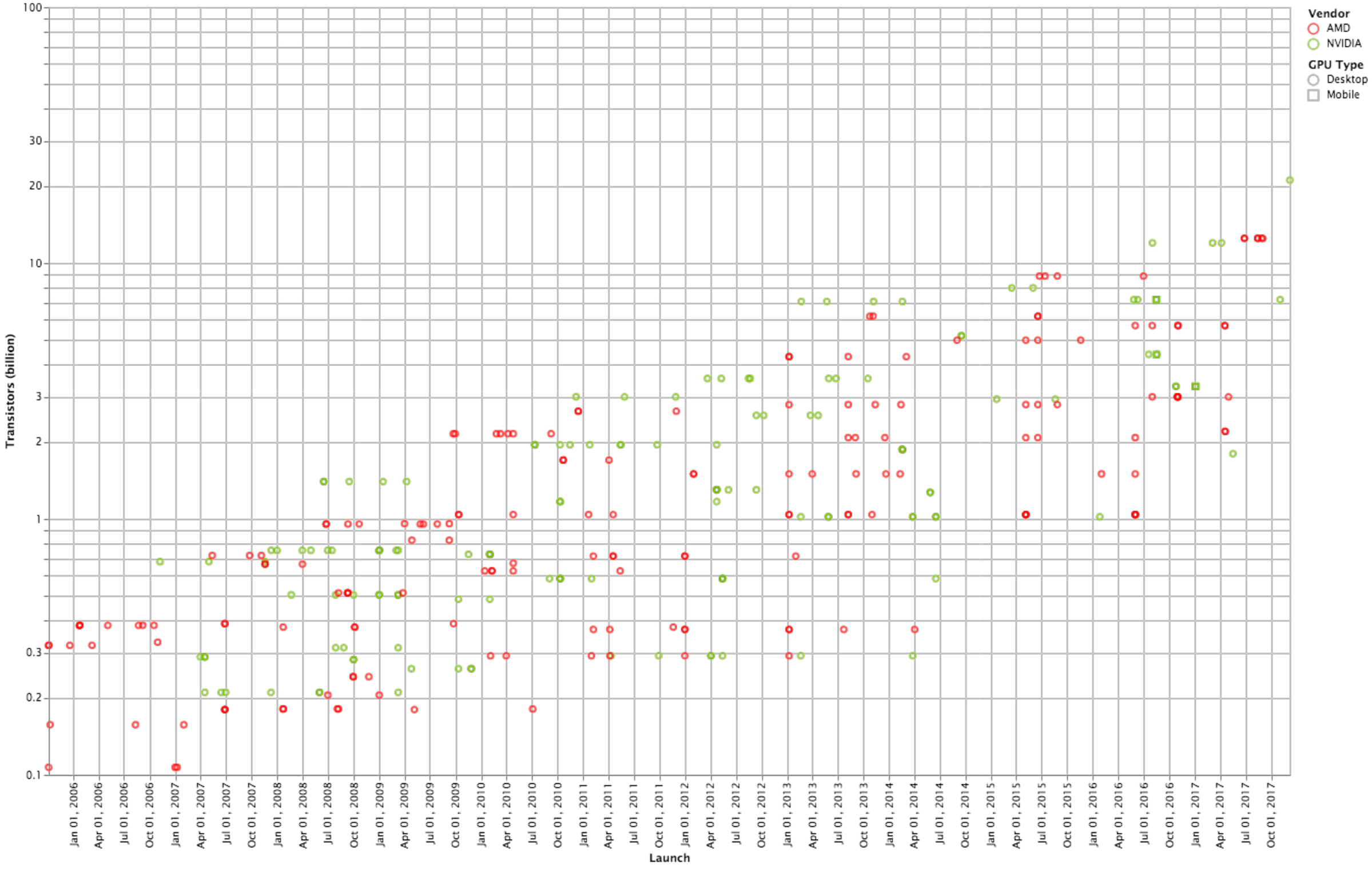
Arithmetic Intensity over Time



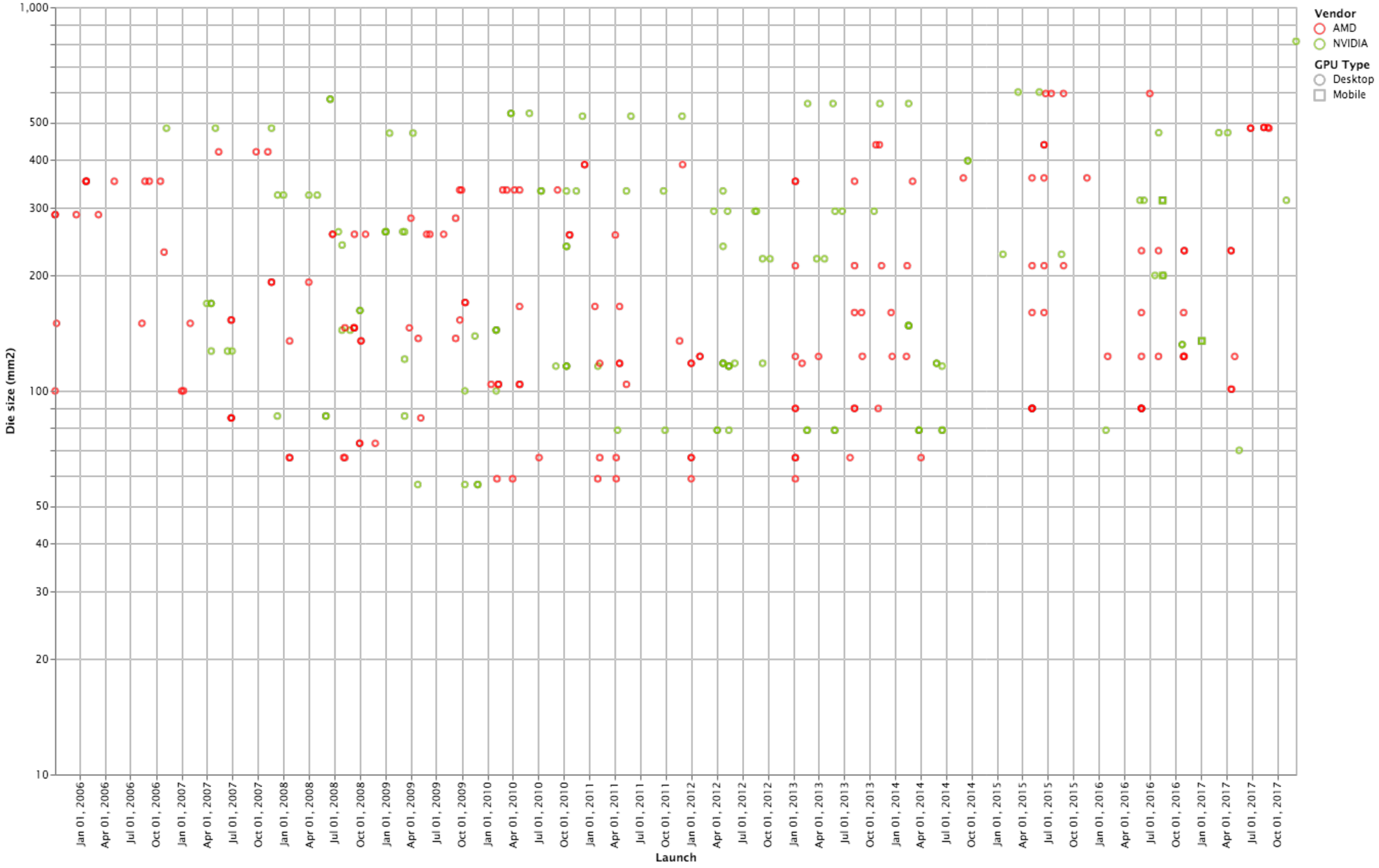
Clock Rate over Time



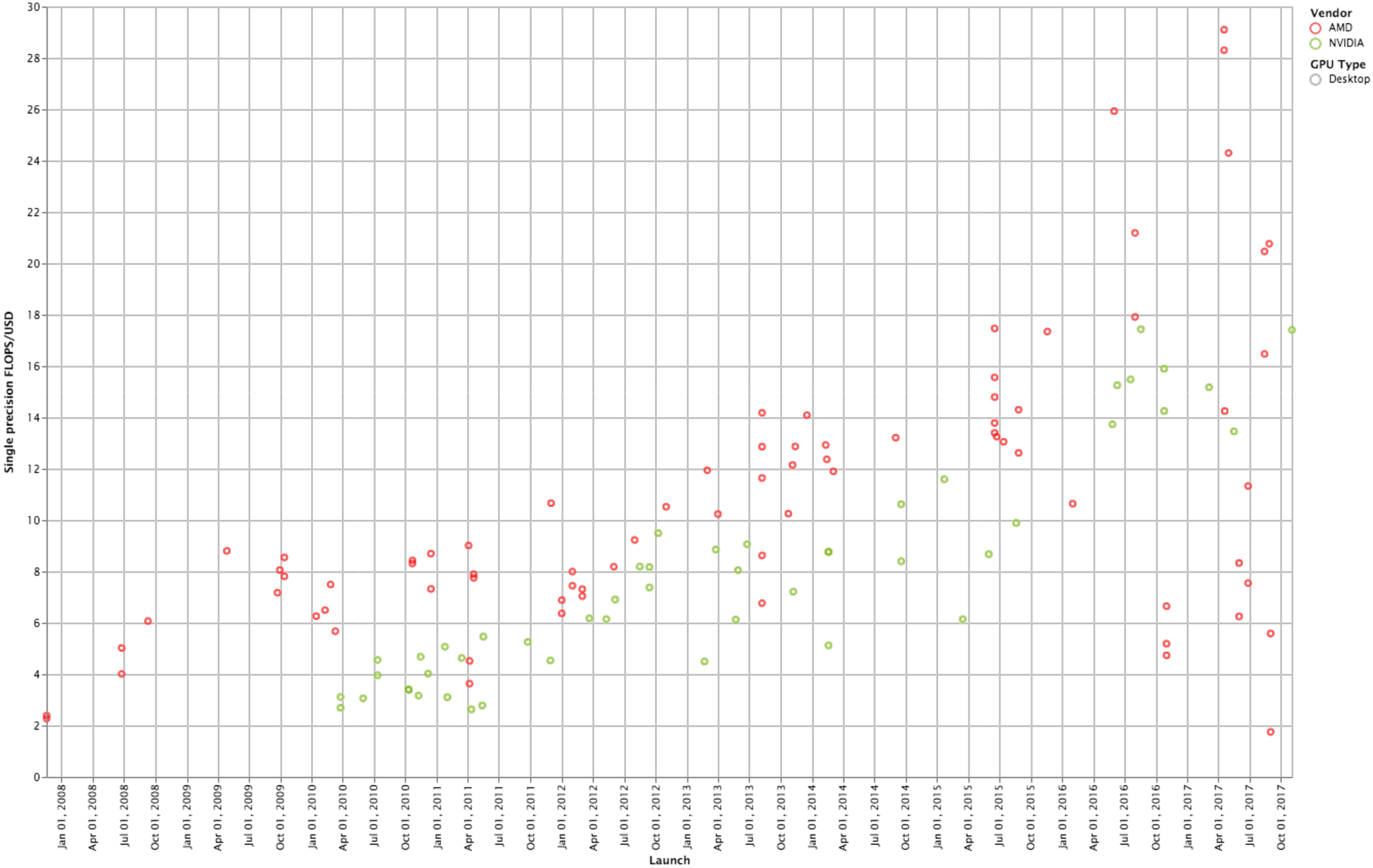
Transistor Count over Time



Die Size over Time



FLOPS per dollar vs. time



FLOPS per Watt vs. time