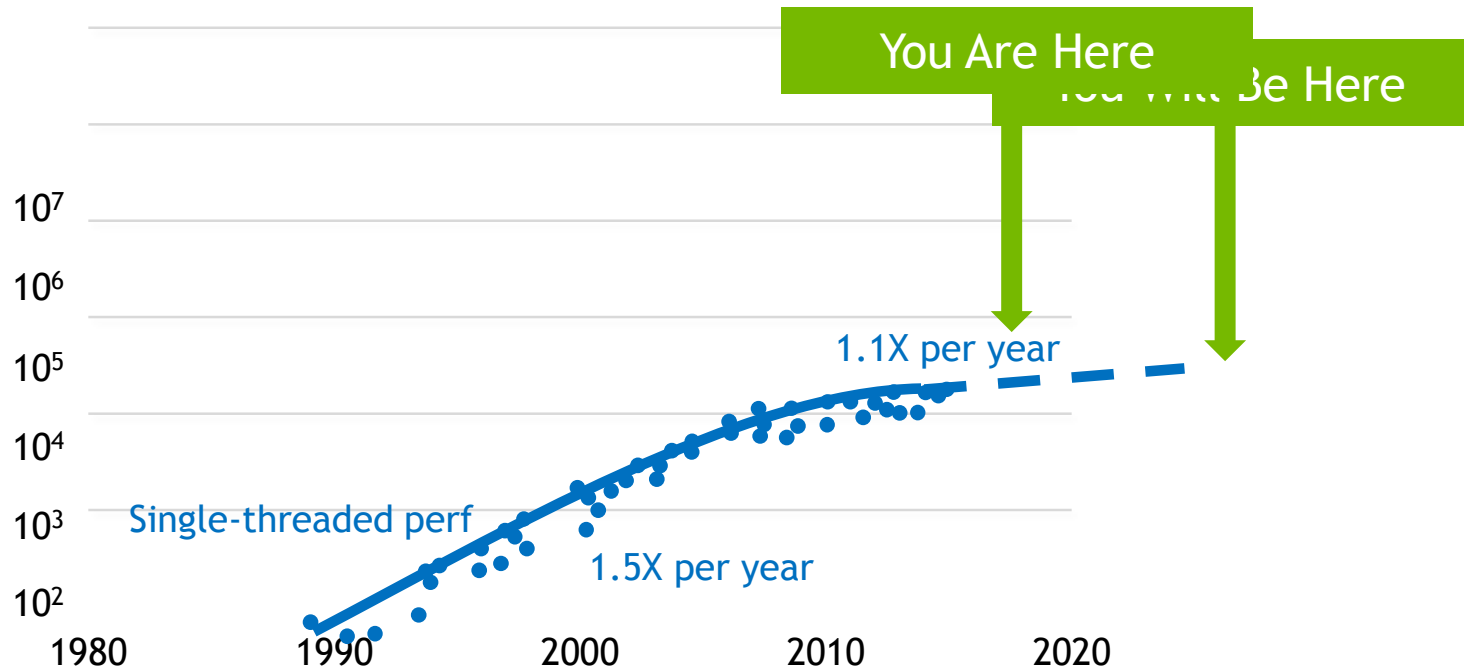

# AN ACCELERATED SYSTEM IS DIFFERENT

*Steve Oberlin @ GTCDC 2018*

# WHAT IS THE PROBLEM?

# THE CPU IS OUT OF GAS

## No Really Running on Fumes



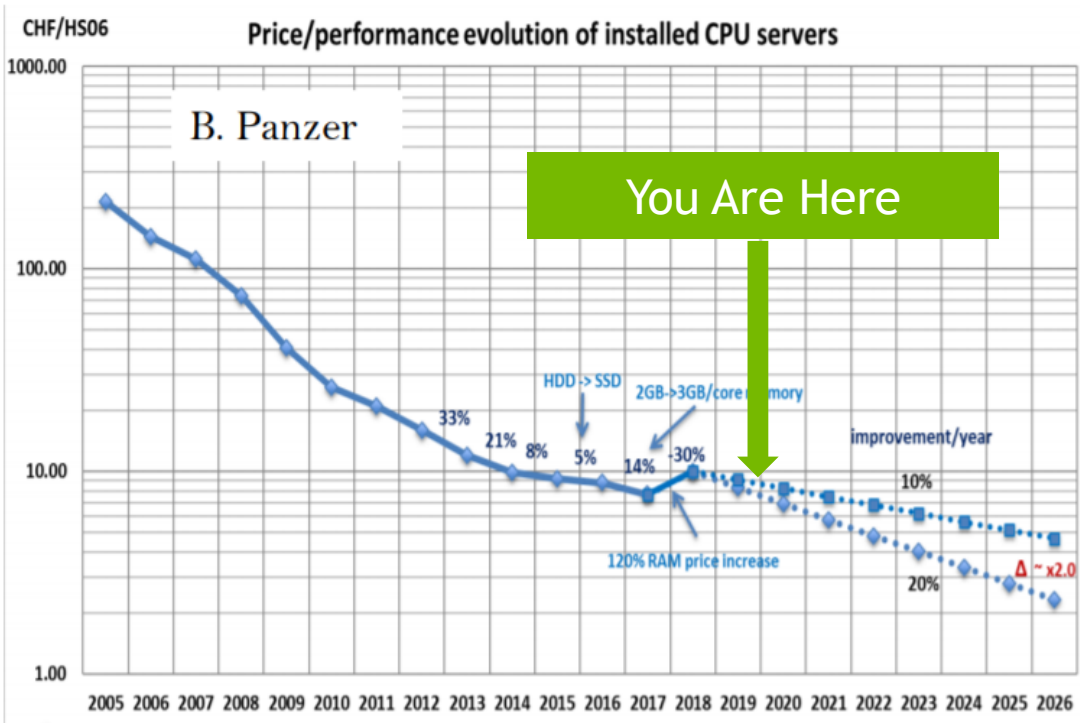

Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten New plot and data collected for 2010-2015 by K. Rupp



NVIDIA.

# PRICE/PERFORMANCE ALREADY CONSTRAINED

## CPU trends




- CPU evolution is not able to cope with the increasing demand of performance
- Depending on the application, GPUs can provide better performance and energy efficiency

# SENSE OF URGENCY?

Dennard scaling ended 13 years ago

- Single threaded performance is capped at ~ 2.5 GHz

Moore's law is slowing

- Transistor density increasing at a slower rate

The economics have changed

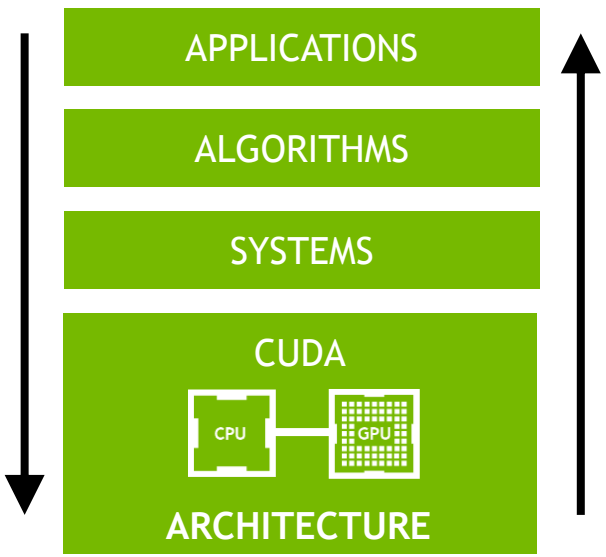
- Fewer transistors per generation cost more to produce, and don't run any faster

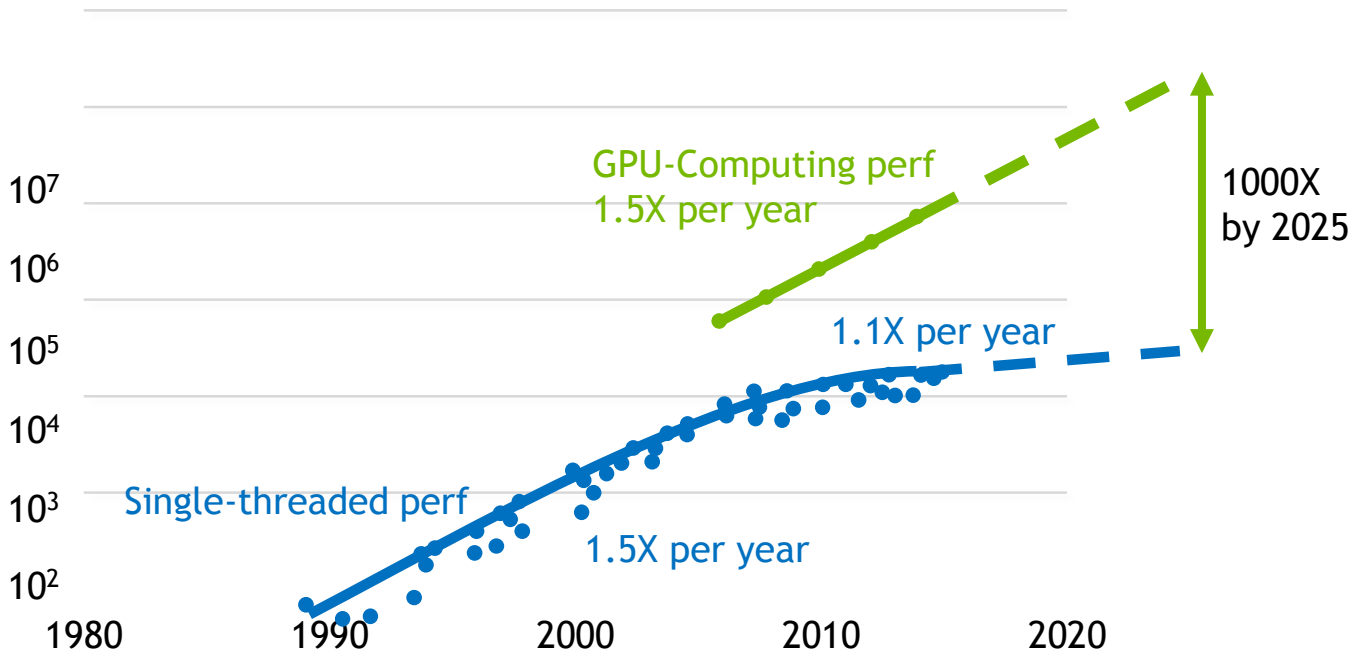
**What's Your Plan?**



NVIDIA.

# AN ACCELERATED SYSTEM CAN MAINTAIN OR EXCEED PREVIOUS PERFORMANCE TRAJECTORY



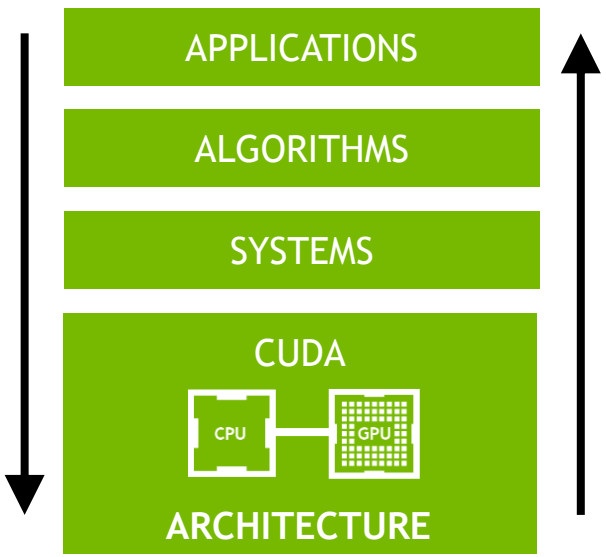



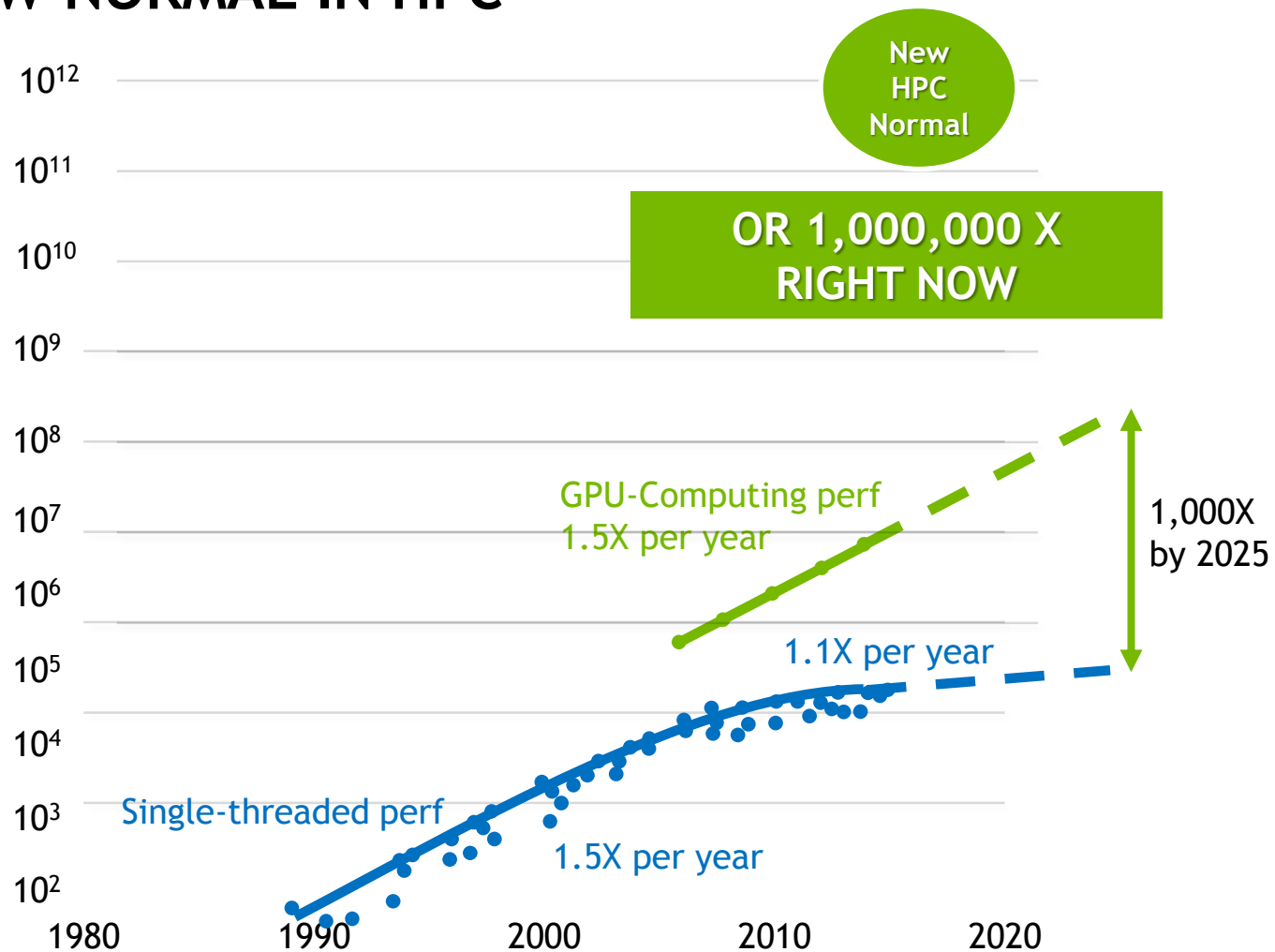

Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten New plot and data collected for 2010-2015 by K. Rupp

**BUT NOT ALL APPS can move from the BLUE line to the GREEN line**
**And their "mileage" will vary**

# THE SOLUTION

# NEW ALGORITHMS AND METHODS ARE EMERGING TO DEFINE A NEW NORMAL IN HPC




Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten New plot and data collected for 2010-2015 by K. Rupp

# THE KEY DIFFERENCES

**Unaccelerated systems**

Have homogenous nodes, where all nodes are the same

Max throughput with minimum variance regardless of workload

All the apps are modeling/simulation

Most apps are batch

Most modeling/simulation apps are double precision

**Accelerated systems**

Have heterogenous nodes, where nodes are tuned to the workload

Max throughput and maximize performance variance based on workload

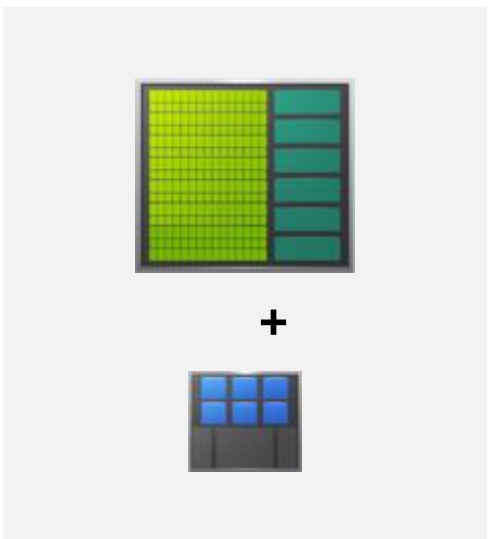
Wide range of apps: modeling/simulation to data science

A mix of batch and interactive jobs

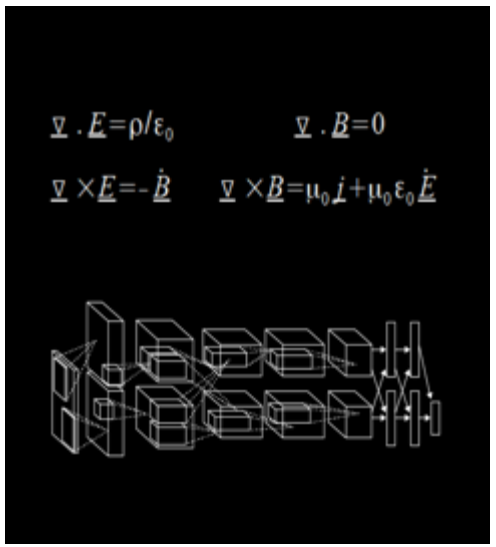
Precision selected to optimize performance


NVIDIA.

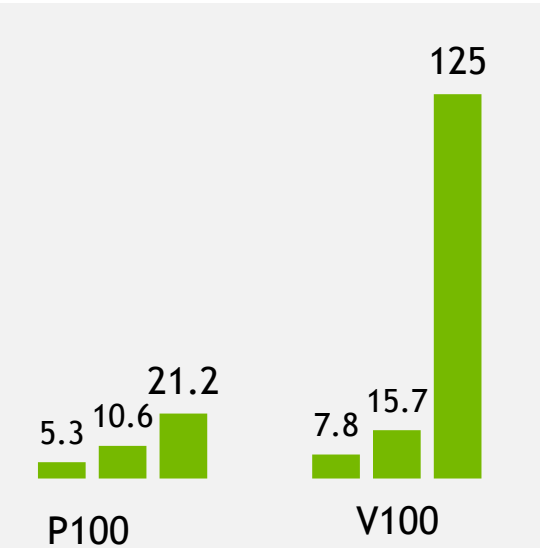
# THE PATH FORWARD



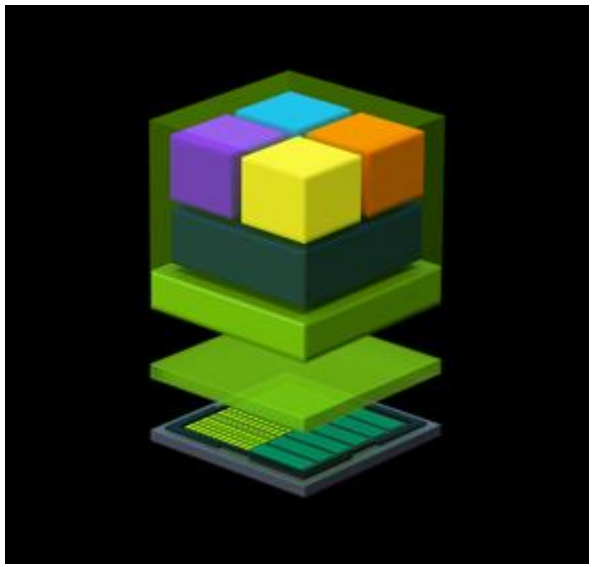
CPU + Accelerator




Simulation + AI




FP64 + Multi-Precision



Full-stack Optimization

# CUDA PLATFORM

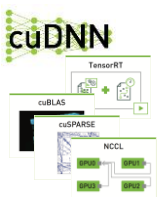
**People**

*GPU Users* | *Domain Specialists* | *Problem Specialists* | *New Developers and Optimization Experts*

**Domain Descriptors**


**Applications**

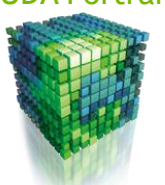
**Frameworks**




**Libraries**


**Directives and Standard Languages**

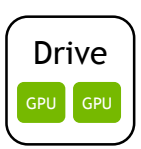
CUDA-C++
CUDA Fortran



**Extended Standard Languages**

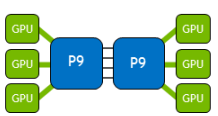
*Ease of use* ——— *Specialized Performance* →

**Systems**

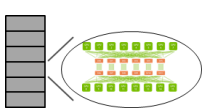




GeForce
Quadro
Tesla PCI




Supercomputing



Cloud + HGX/DGX

**Foundation**

Programming Model, GPU Architecture, System Architecture, Tools



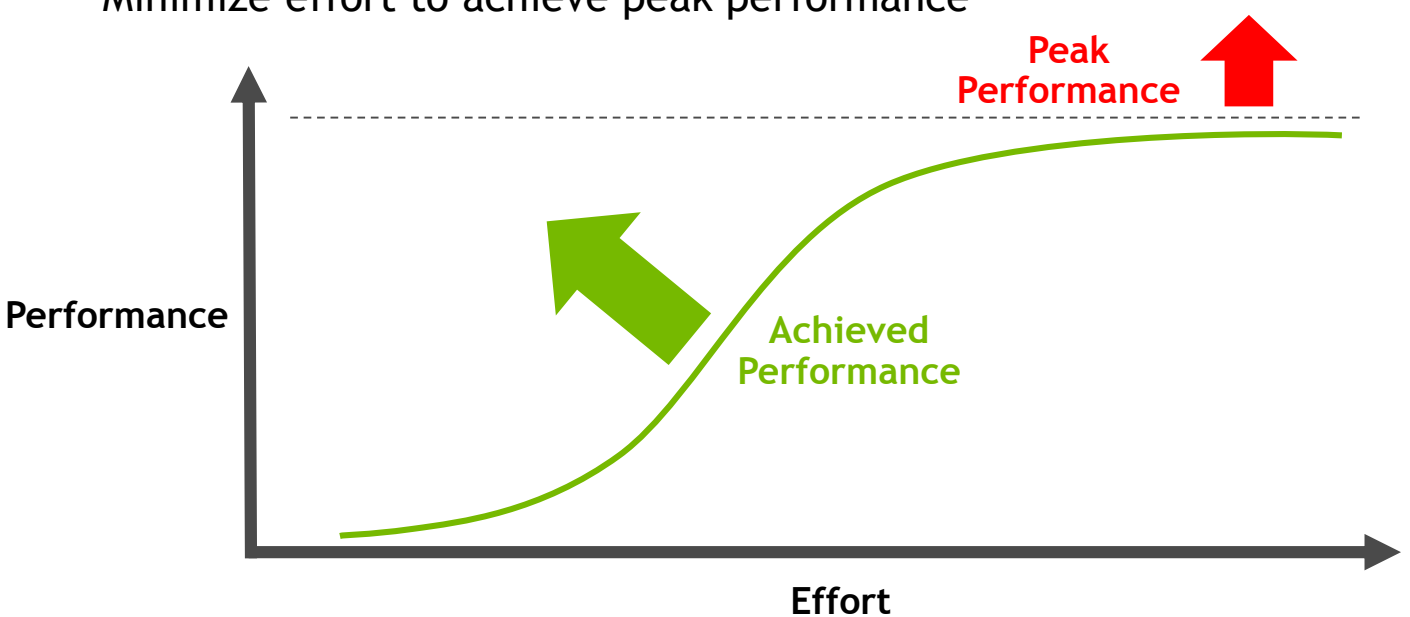
NVIDIA.

# CUDA PLATFORM GOALS

Every layer of the CUDA Platform has two fundamental goals:

Maximize achievable peak performance

Minimize effort to achieve peak performance



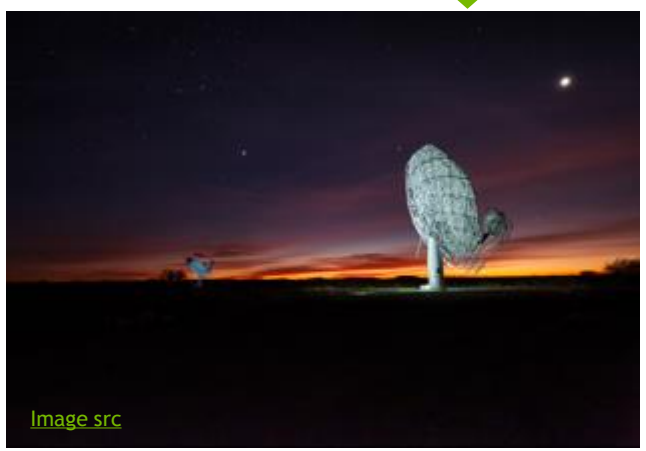

## CUDA Platform Layers

- Applications
- Frameworks & DSLs
- Libraries
- Directives and Standard Languages
- Extended Standard Languages
- System Architecture
- GPU Architecture

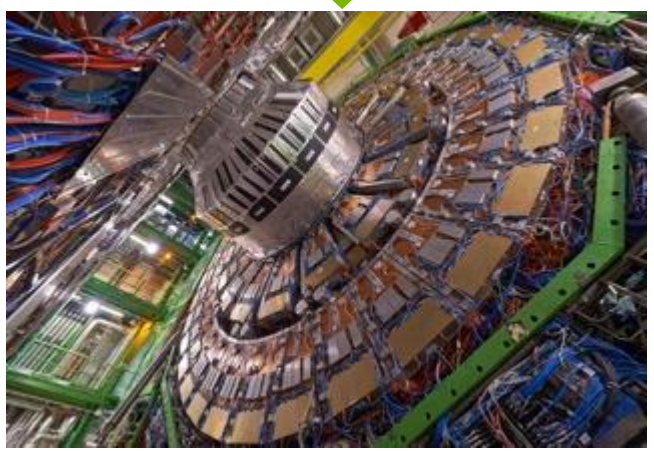


NVIDIA.

# AI: A NEW TOOL FOR SCIENCE


The SKA1 Square Kilometre Array radio telescope will generate more than an Exabyte of data every day.

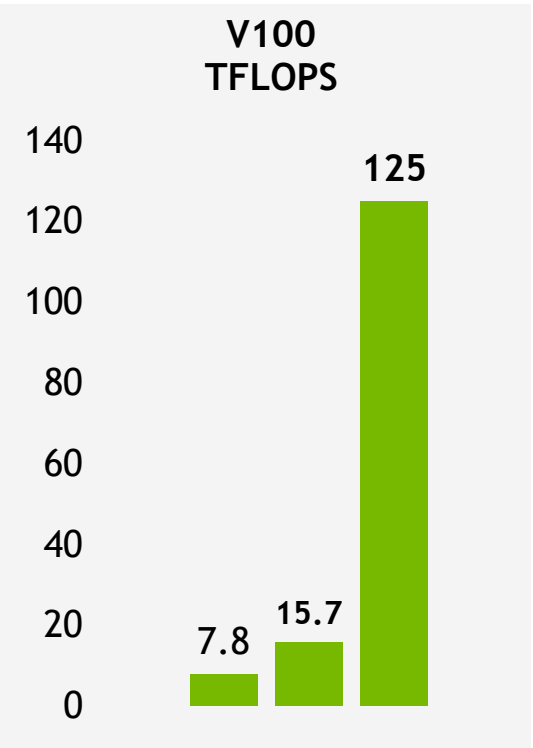


The CERN large Hadron collider's High Luminosity upgrade will result in a 10X increase in data volume.

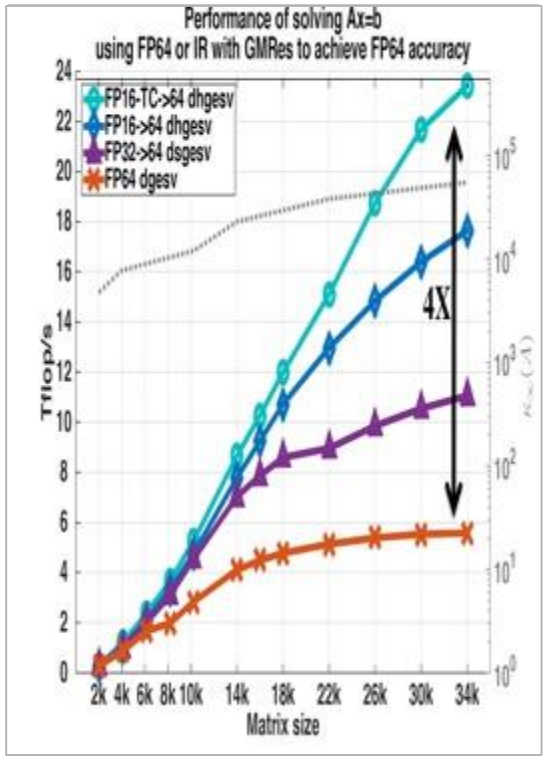
The 500 MW ITER fusion experiment will provide a 30X increase in output power over the largest previous experiment.

# TENSOR CORES FOR SCIENCE

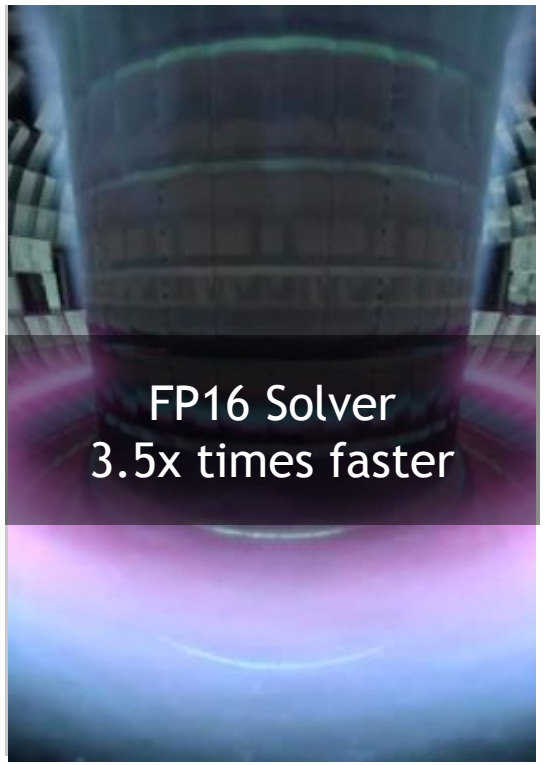
## Multi-precision computing




FP64+ MULTI-PRECISION




MAGMA Library with FP16

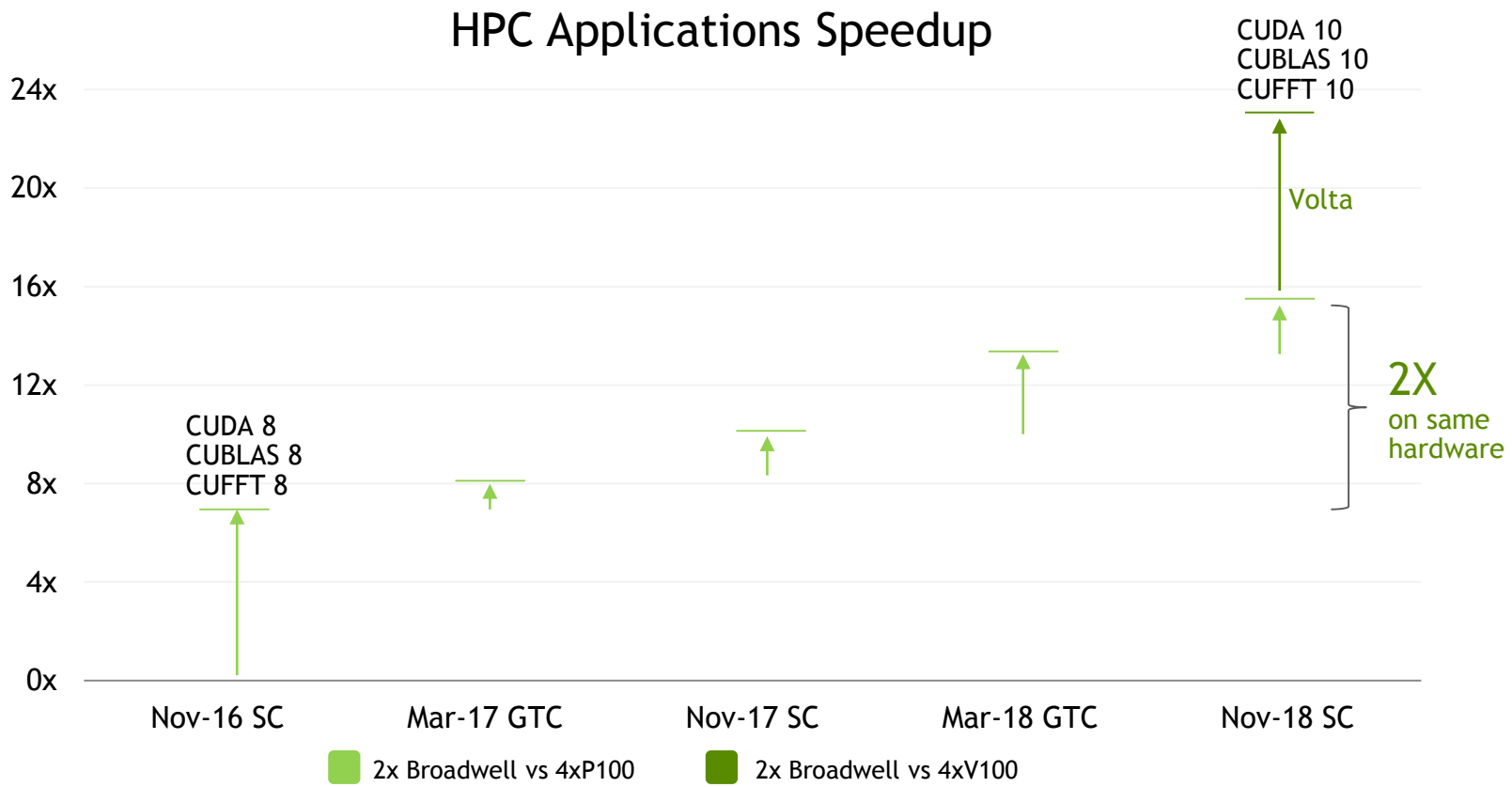



PLASMA FUSION APPLICATION


EARTHQUAKE SIMULATION

# ACCELERATED COMPUTING IS FULL-STACK OPTIMIZATION

## 2X More Performance with Software Optimizations Alone



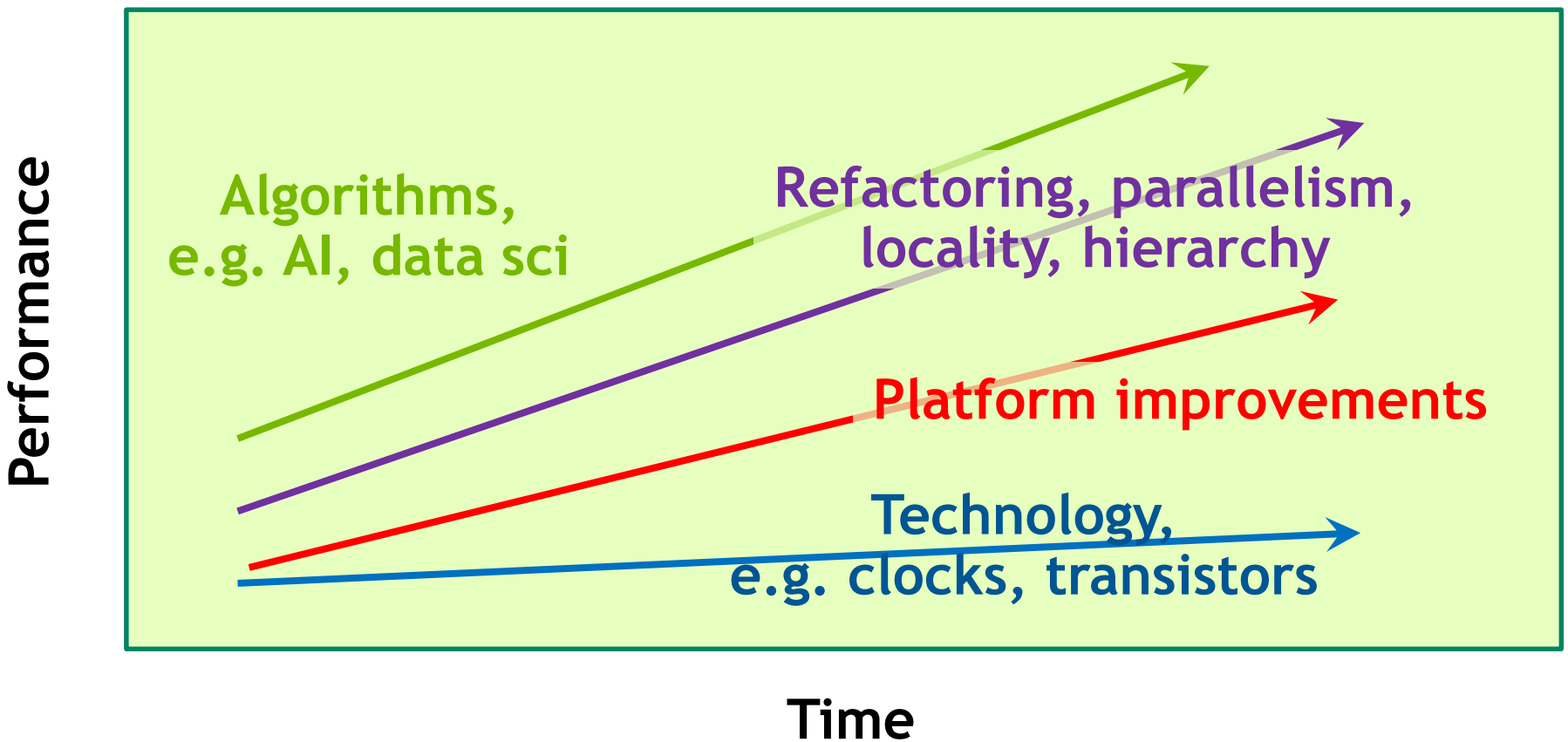

HPC Apps: AMBER, Chroma, GROMACS, GTC, LAMMPS, MILC, NAMD, QE, RTM, SPECFEM3D, VASP



NVIDIA.

# MATCH YOUR WORK TO A LINE

More investment, more return on that investment



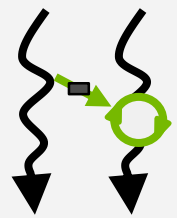



NVIDIA.

# CUDA PROGRAMMING ARCHITECTURE

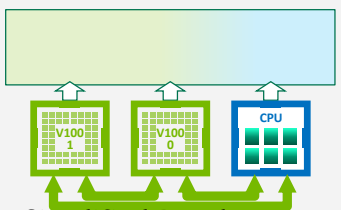
## Recent developments and directions

**Improved SIMT Model**



New Algorithms

**Unified Memory**

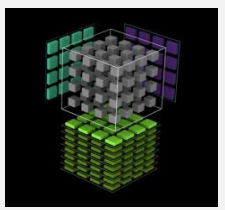



Simplified Development

**Languages, directives**

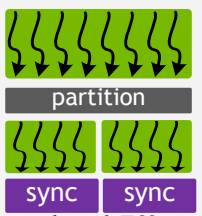

Productive parallel programming

**Tensor Core**



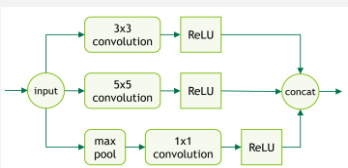
Multi-Precision

**Cooperative Groups**




Hierarchical Efficiency

**CUDA Graphs**

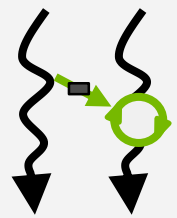



Small Task Efficiency

# CUDA PROGRAMMING ARCHITECTURE

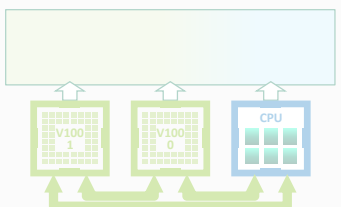
## Recent developments and directions

**Improved SIMT Model**



**New Algorithms**

Unified Memory

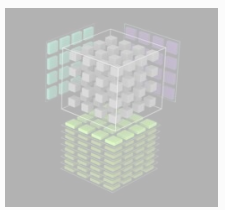



Simplified Development

Languages, directives

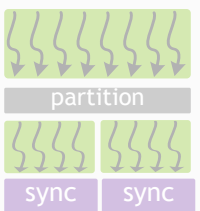

Productive parallel programming

Tensor Core



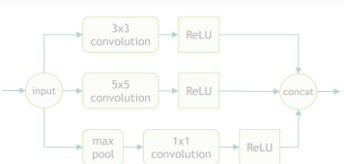
Multi-Precision

Cooperative Groups




Hierarchical Efficiency

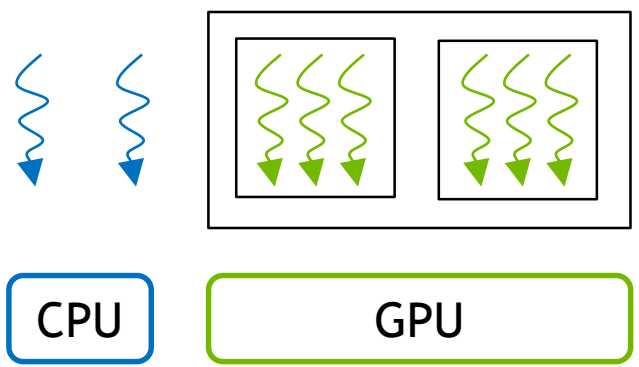
CUDA Graphs




Small Task Efficiency

# THREAD CAPABILITIES

## History




Thread Capabilities

Tesla: SIMT, Load/Store Pointers

Fermi: Double Precision, Function Pointers
64-bit addresses

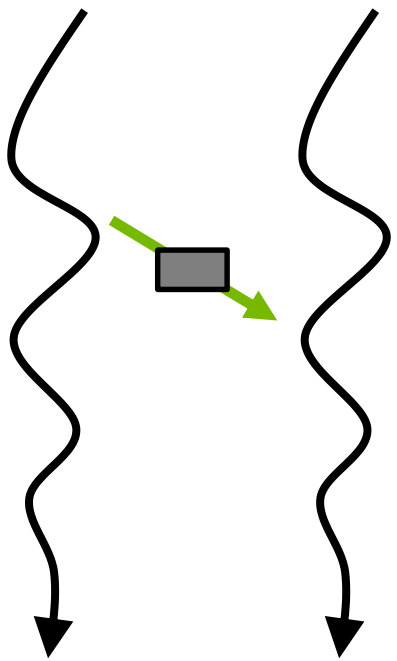
Kepler: Dynamic Parallelism

Pascal: Inter-Block Synchronization

Volta: Independent Thread Scheduling,
Robust Memory Model

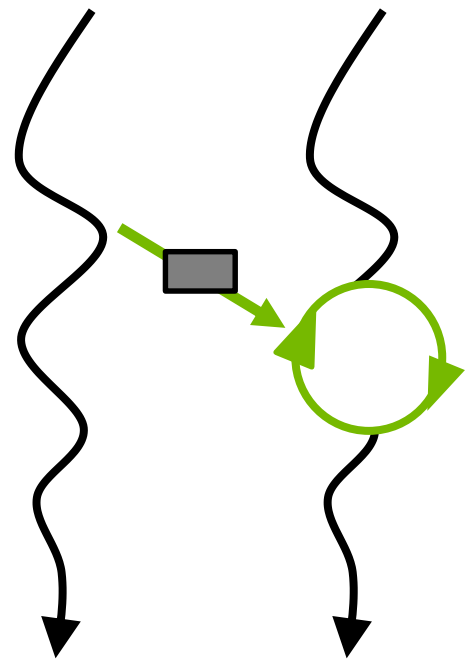
# INDEPENDENT THREAD SCHEDULING

## Communicating Algorithms



**Pascal: Lock-Free Algorithms**

Threads cannot wait for messages



**Volta/Turing: Starvation Free Algorithms**

Threads may wait for messages



NVIDIA.

## [PEOPLE THINK] GPUS ARE GOOD FOR:

1. Floats, short floats, and doubles.
2. Arrays (may be multi-dimensional).
3. Coalesced memory access.
4. Lock-free algorithms.

## [PEOPLE THINK] GPUS ARE BAD FOR:

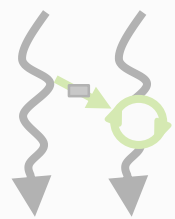
1. Strings.
2. Node-based data structures.
3. Random memory walks.
4. Starvation-free algorithms (spinlocks).

***From Olivier Giroux's (NVIDIA) CPPCON 2018 talk**

# CUDA PROGRAMMING ARCHITECTURE

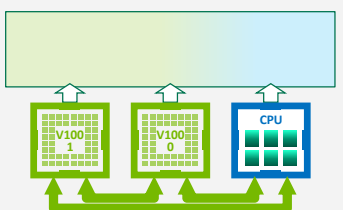
## Recent developments and directions

Improved SIMT Model



New Algorithms

**Unified Memory**

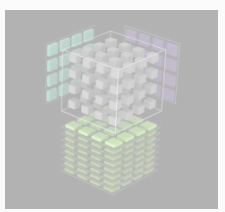



Simplified Development

Languages, directives

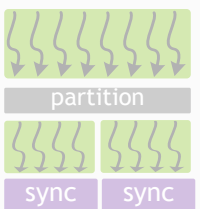

Productive parallel programming

Tensor Core



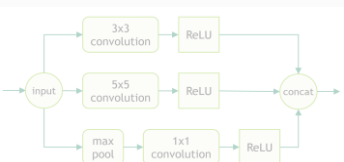
Multi-Precision

Cooperative Groups




Hierarchical Efficiency

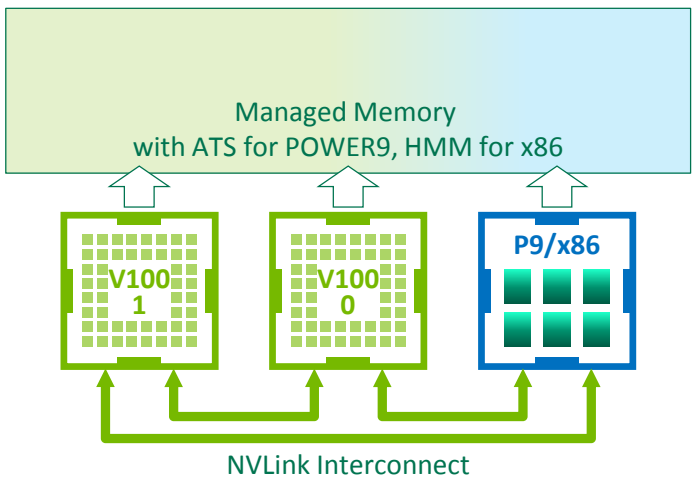
CUDA Graphs




Small Task Efficiency

# UNIFIED MEMORY

## Access all memory in the node




### ALLOCATION

Automatic access to *all* system memory: malloc, statics, globals, stack, file system

### ACCESS

All data accessible concurrently from any processor, anytime

Atomic operations resolved directly over NVLink

# WHAT YOU CAN DO WITH UNIFIED MEMORY

## Works everywhere today

```
int *data;
cudaMallocManaged(&data, sizeof(int) * n);
kernel<<< grid, block >>>(data);
```

## Works on POWER9 + CUDA 9.2 and forthcoming x86 + HMM

```
int *data = (int*)malloc(sizeof(int) * n);
kernel<<< grid, block >>>(data);
```

```
int data[1024];
kernel<<< grid, block >>>(data);
```

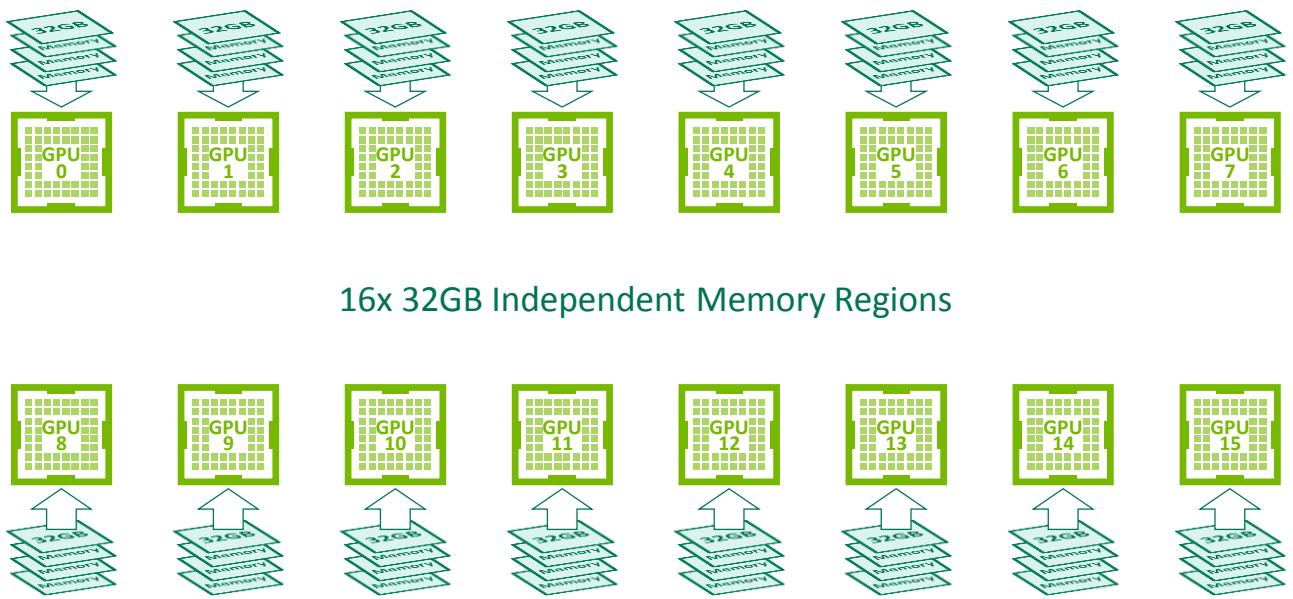
```
int *data = (int*)alloca(sizeof(int) * n);
kernel<<< grid, block >>>(data);
```

```
extern int *data;
kernel<<< grid, block >>>(data);
```



NVIDIA.

# 16 GPUs WITH 32GB MEMORY EACH



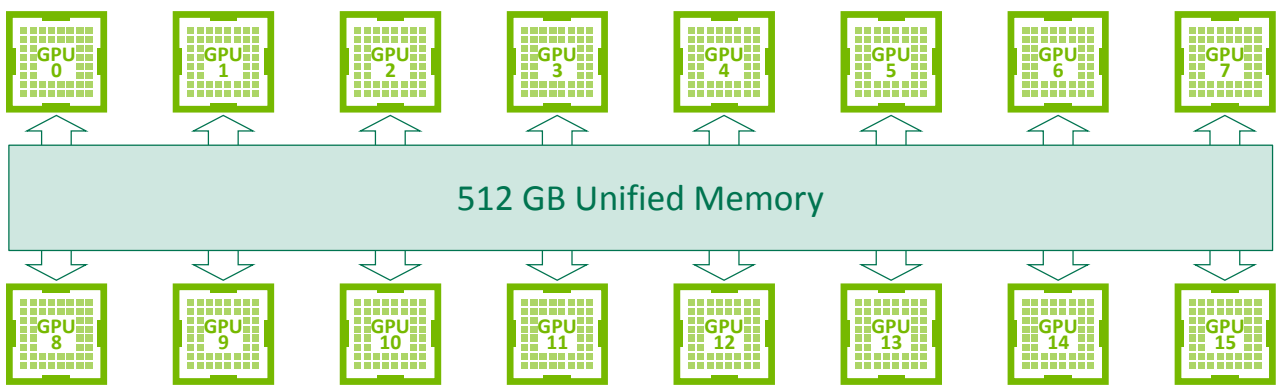

## NVSWITCH PROVIDES

All-to-all high-bandwidth peer mapping between GPUs

Full inter-GPU memory interconnect (incl. Atomics)


NVIDIA.

# UNIFIED MEMORY + DGX-2




## UNIFIED MEMORY PROVIDES

Single memory view shared by all GPUs

Automatic migration of data between GPUs

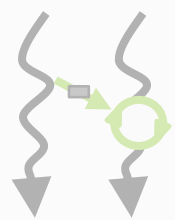
User control of data locality


NVIDIA.

# CUDA PROGRAMMING ARCHITECTURE

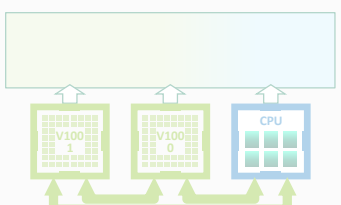
## Recent developments and directions

Improved SIMT Model



New Algorithms

Unified Memory

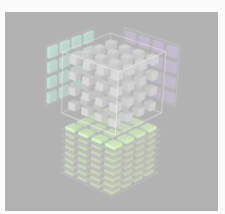



Simplified Development

**Languages, directives**

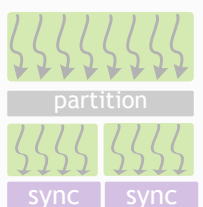

**Productive parallel programming**

Tensor Core



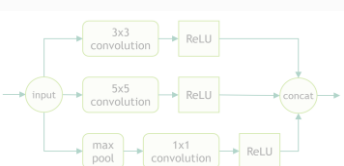
Multi-Precision

Cooperative Groups




Hierarchical Efficiency

CUDA Graphs




Small Task Efficiency

# Parallel Features in Fortran and C++

## Fortran 2018

Array syntax (F90)

FORALL (F95)

Co-arrays (F08, F18)

DO CONCURRENT (F08, F18)

## C++17

Threads (C++11)

pSTL Parallel Algorithms (C++17)

# Fortran 2018 DO CONCURRENT Construct

## Adds support for explicit shared vs private data

ISO/IEC FDIS 1539-1:2018 (E)

NOTE 5

The following code demonstrates the use of the LOCAL clause so that the X inside the DO CONCURRENT construct is a temporary variable, and will not affect the X outside the construct.

```
X = 1.0
DO CONCURRENT (I=1:10) LOCAL (X)
  IF (A (I) > 0) THEN
    X = SQRT (A (I))
    A (I) = A (I) - X**2
  END IF
  B (I) = B (I) - A (I)
END DO
PRINT *, X                          ! Always prints 1.0.
```

# C++17 Parallel Algorithms (pSTL)

## Parallel and Vector Concurrency for the Standard Algorithms

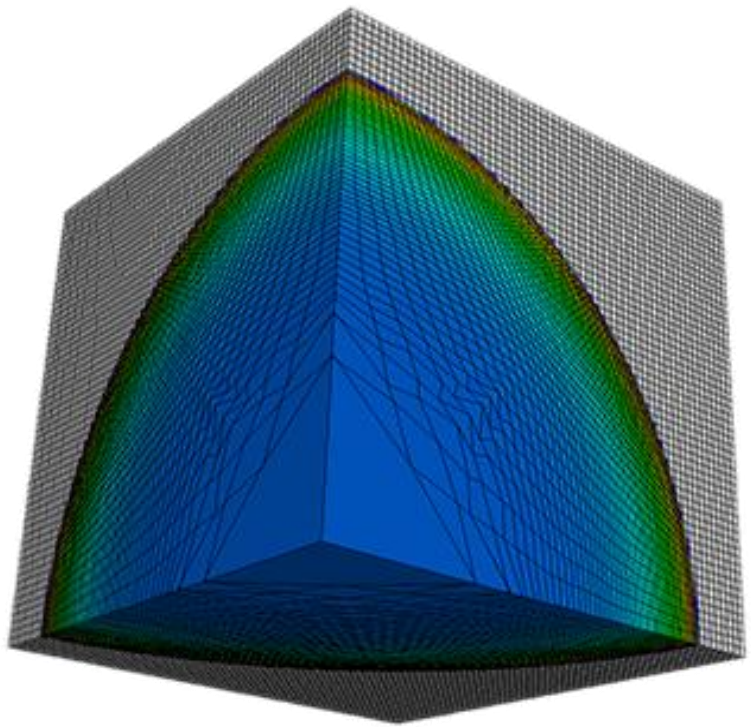
- Execution policies can now be applied to most STL algorithms
  - std::execution::seq // sequential
  - std::execution::par // parallel
  - std::execution::par_unseq // parallel+vector
- A few new parallel algorithms introduced in C++17
  - std::transform_reduce (POLICY, first1, last1, first2, trans_op, init, reduce_op)
  - std::for_each_n (POLICY, first, size, func)
  - …

*C++17 Parallel Algorithms and Beyond*, Bryce Lelbach, CppCon 2016

# The LULESH Hydrodynamics Mini-app

LLNL Hydrodynamics Proxy (Mini-App)

- ~9000 lines of C++
- Parallel versions in MPI, OpenMP, OpenACC, CUDA, RAJA
- Designed to stress compiler vectorization, parallel overheads, on-node parallelism



codesign.llnl.gov/lulesh

# C++ with OpenACC Directives

```
% pgc++ -fast -ta=tesla:managed -Minfo -c lulesh.cc
  ...
  CalcVelocityForNodes(Domain &, double, double, int):
     2103, Accelerator kernel generated
           Generating Tesla code
     2106, #pragma acc loop gang, vector(128) ! BlockIdx.x
                                              ! threadIdx.x
    ...
```

```
2100 static inline
2101 void CalcVelocityForNodes(Domain &domain, const Real_t dt,
2102                           const Real_t u_cut, Index_t numNode)
2103 {
2104
2105 #pragma acc parallel loop
2106    for ( Index_t i = 0 ; i < numNode ; ++i )
2107    {
2108      Real_t xdtmp, ydtmp, zdtmp ;
2109
2110      xdtmp = domain.xd(i) + domain.xdd(i) * dt ;
2111      if( FABS(xdtmp) < u_cut ) xdtmp = Real_t(0.0);
2112      domain.xd(i) = xdtmp ;
2113
2114      ydtmp = domain.yd(i) + domain.ydd(i) * dt ;
2115      if( FABS(ydtmp) < u_cut ) ydtmp = Real_t(0.0);
2116      domain.yd(i) = ydtmp ;
2117
2118      zdtmp = domain.zd(i) + domain.zdd(i) * dt ;
2119      if( FABS(zdtmp) < u_cut ) zdtmp = Real_t(0.0);
2120      domain.zd(i) = zdtmp ;
2121    }
2122 }
```

# C++17 Parallel Algorithm: for_each_n()

```
% pgc++ -fast -ta=tesla:managed,pstl -Minfo -c lulesh.cc
  ...
```

```
2100 static inline
2101 void CalcVelocityForNodes(Domain &domain, const Real_t dt,
2102                           const Real_t u_cut, Index_t numNode)
2103 {
2104
2105
2106    std::for_each_n (POLICY_PAR, idx(0), numNode, [&](Index_t i)
2107    {
2108      Real_t xdtmp, ydtmp, zdtmp ;
2109
2110      xdtmp = domain.xd(i) + domain.xdd(i) * dt ;
2111      if( FABS(xdtmp) < u_cut ) xdtmp = Real_t(0.0);
2112      domain.xd(i) = xdtmp ;
2113
2114      ydtmp = domain.yd(i) + domain.ydd(i) * dt ;
2115      if( FABS(ydtmp) < u_cut ) ydtmp = Real_t(0.0);
2116      domain.yd(i) = ydtmp ;
2117
2118      zdtmp = domain.zd(i) + domain.zdd(i) * dt ;
2119      if( FABS(zdtmp) < u_cut ) zdtmp = Real_t(0.0);
2120      domain.zd(i) = zdtmp ;
2121    });
2122 }
```

# PRODUCTIVE PROGRAMMING

## Conjugate Gradient implemented using NumPy

```
import numpy as np

def solve(A, b):
    x = np.zeros(A.shape[1])
    r = b - A.dot(x)
    p = r
    rsold = r.dot(r)
    for i in range(b.shape[0]):
        Ap = A.dot(p)
        alpha = rsold / (p.dot(Ap))
        x = x + alpha * p
        r = r - alpha * Ap
        rsnew = r.dot(r)
        if np.sqrt(rsnew) < 1e-10:
             break;
        beta = rsnew / rsold
        p = r + beta * p
        rsold = rsnew
     return x
```

```
import legate as np

def solve(A, b):
    x = np.zeros(A.shape[1])
    r = b - A.dot(x)
    p = r
    rsold = r.dot(r)
    for i in range(b.shape[0]):
        Ap = A.dot(p)
        alpha = rsold / (p.dot(Ap))
        x = x + alpha * p
        r = r - alpha * Ap
        rsnew = r.dot(r)
        if np.sqrt(rsnew) < 1e-10:
             break;
        beta = rsnew / rsold
        p = r + beta * p
        rsold = rsnew
     return x
```

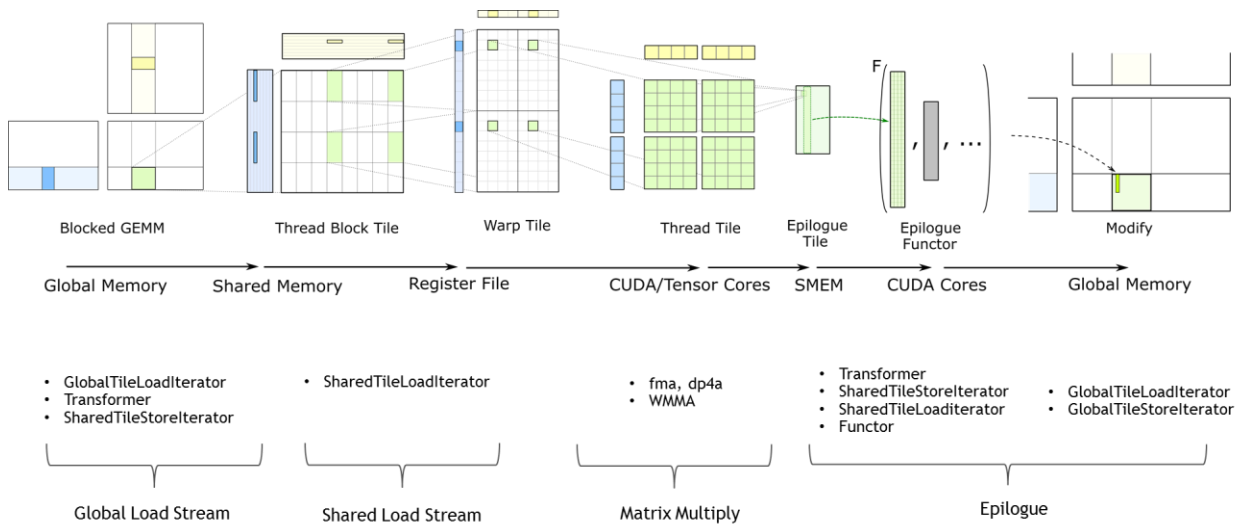
On CPU or GPU system: $ ./legate app.py

# C++ TEMPLATES: CUTLASS 1.1

## High-performance Matrix Multiplication in Open Source CUDA C++

- Turing optimized GEMMs
  - Integer (8-bit, 4-bit and 1-bit) using WMMA
- Batched strided GEMM
- Support for CUDA 10.0
- Updates to documentation and more examples

https://github.com/NVIDIA/cutlass



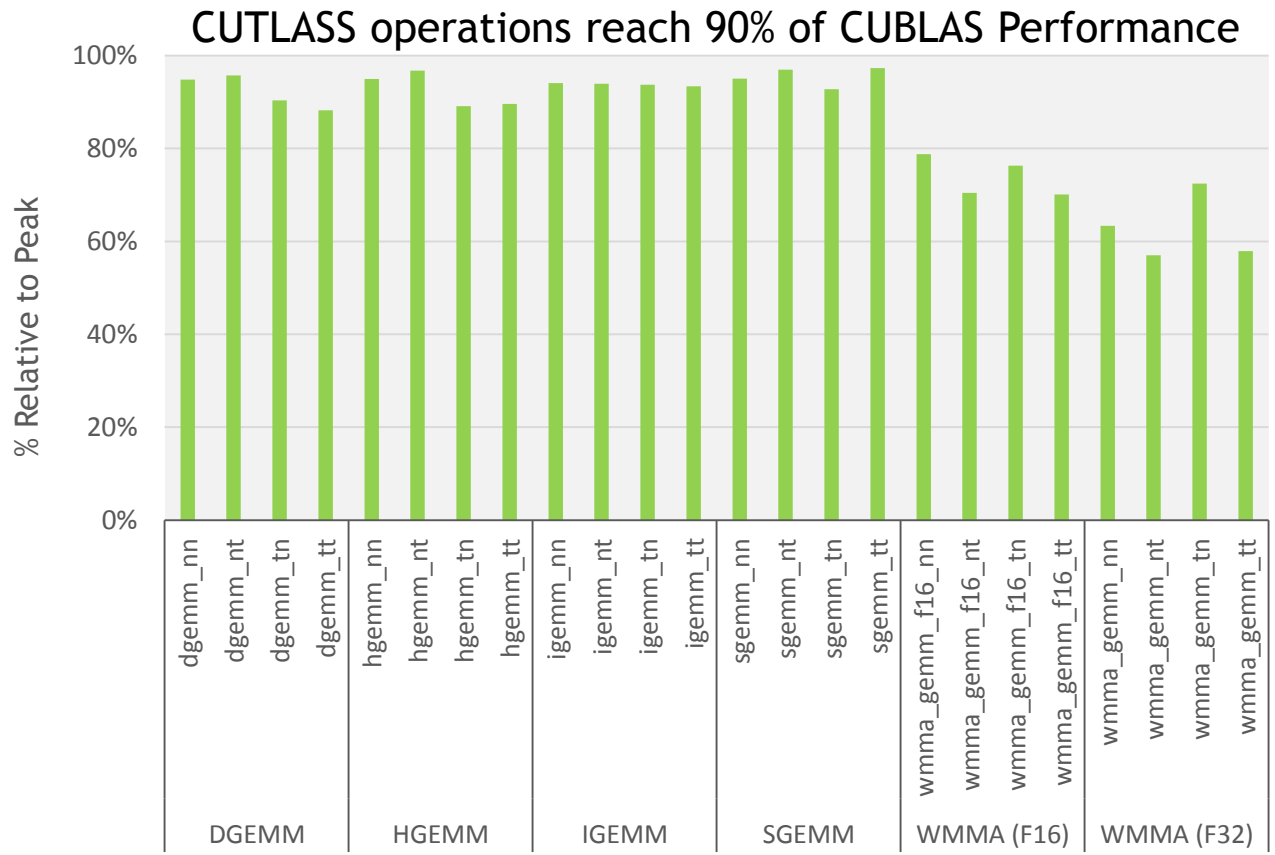

CUTLASS GEMM Structural Model

# CUTLASS 1.1

## High-performance Matrix Multiplication in Open Source CUDA C++

- Turing optimized GEMMs
  - Integer (8-bit, 4-bit and 1-bit) using WMMA
- Batched strided GEMM
- Support for CUDA 10.0
- Updates to documentation and more examples

https://github.com/NVIDIA/cutlass




*CUTLASS 1.1 on Volta (GV100)*

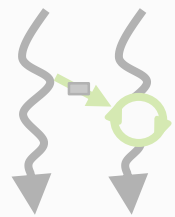


NVIDIA.

# CUDA PROGRAMMING ARCHITECTURE

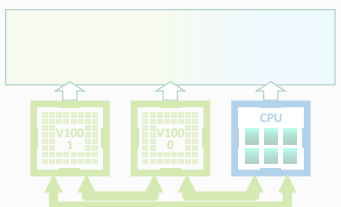
## Recent developments and directions

Improved SIMT Model



New Algorithms

Unified Memory

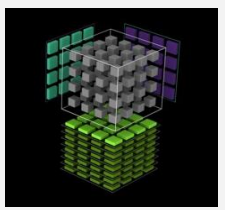



Simplified Development

Languages, directives

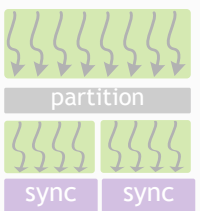

Productive parallel programming

**Tensor Core**



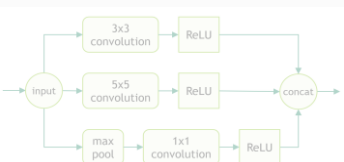
**Multi-Precision**

Cooperative Groups




Hierarchical Efficiency

CUDA Graphs




Small Task Efficiency

NVIDIA CONFIDENTIAL. DO NOT DISTRIBUTE.

# REDUCED PRECISION IN HPC

Why?

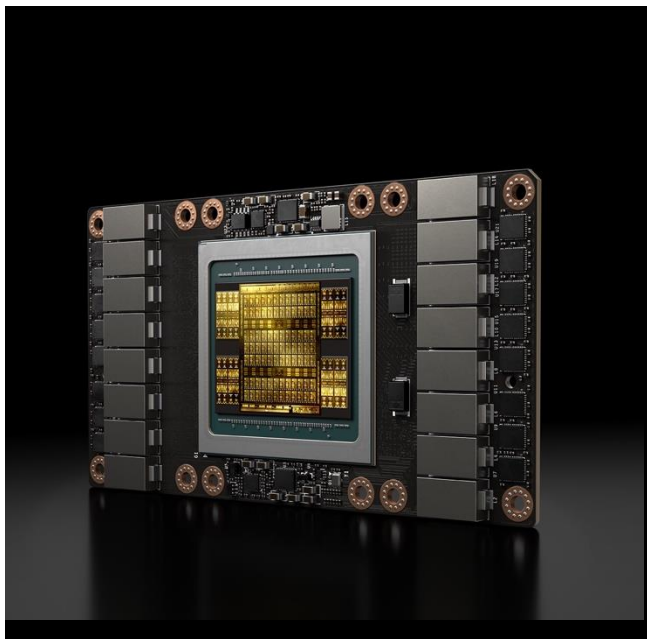
1. Reduce memory traffic

2. Reduce memory footprint

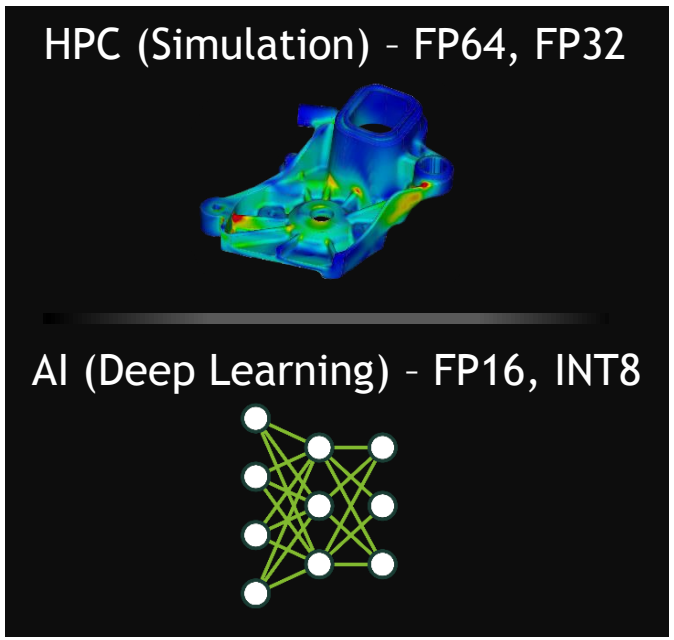
3. Utilize accelerated hardware

….without compromising the end result

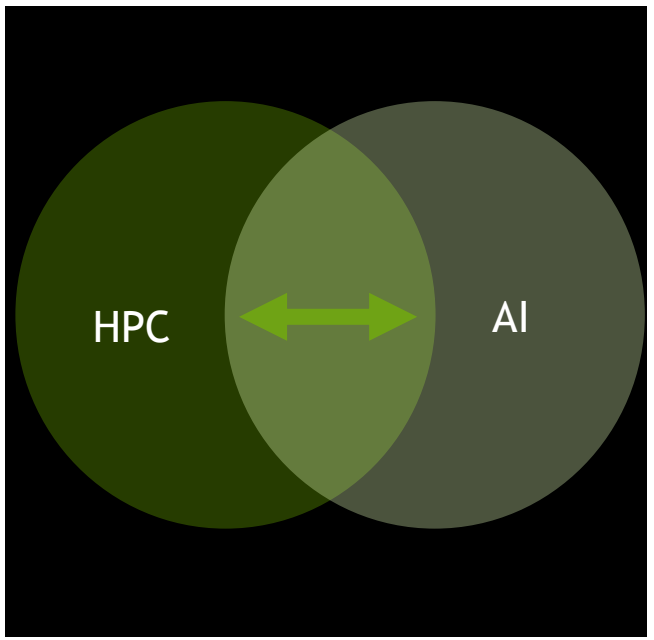
# PRECISION MATTERS FOR HPC AND AI



VOLTA TENSOR CORE GPU




MULTI-PRECISION COMPUTING




FUSION OF HPC & AI


NVIDIA.

# REDUCED PRECISION IN HPC

Mixed-precision algorithms are ever more popular

Most common use case has been combining double + single precision…
…but not the only choice

Combining floating point and integer operations for performance and determinism

- E.g., AMBER – fp32 pair interactions with int64 energy summation

Use fp16 or int16 precision to reduce memory traffic

Use accelerated fp16 or tensor-core instructions for *large* speedups

- E.g., tensor-core accelerated mixed-precision LU solver (Dongarra *et al*)

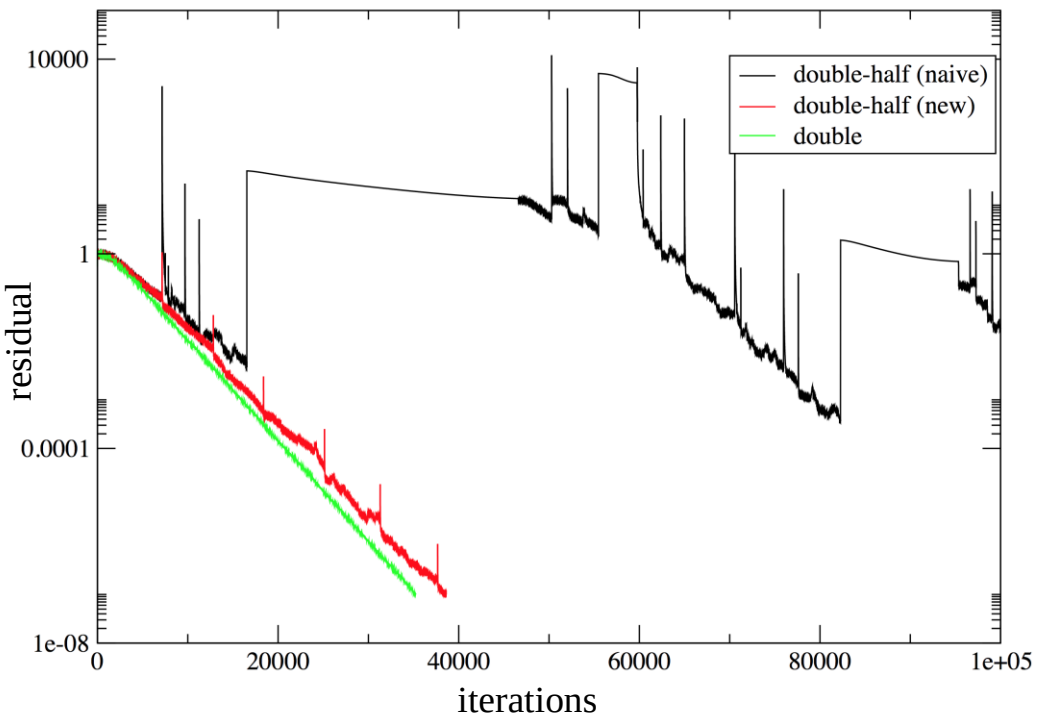
# SPARSE SOLVERS AND LOW PRECISION

Since sparse solvers are iterative, judicious use of precision can lead to large speedups

Most sparse solvers are memory bandwidth bound, so reduced precision accelerates through reduced memory traffic

Recipes need to worked out

- Wrong algorithm will not converge
- E.g., Defect correction vs reliable updates
- Keep the precision where it's needed



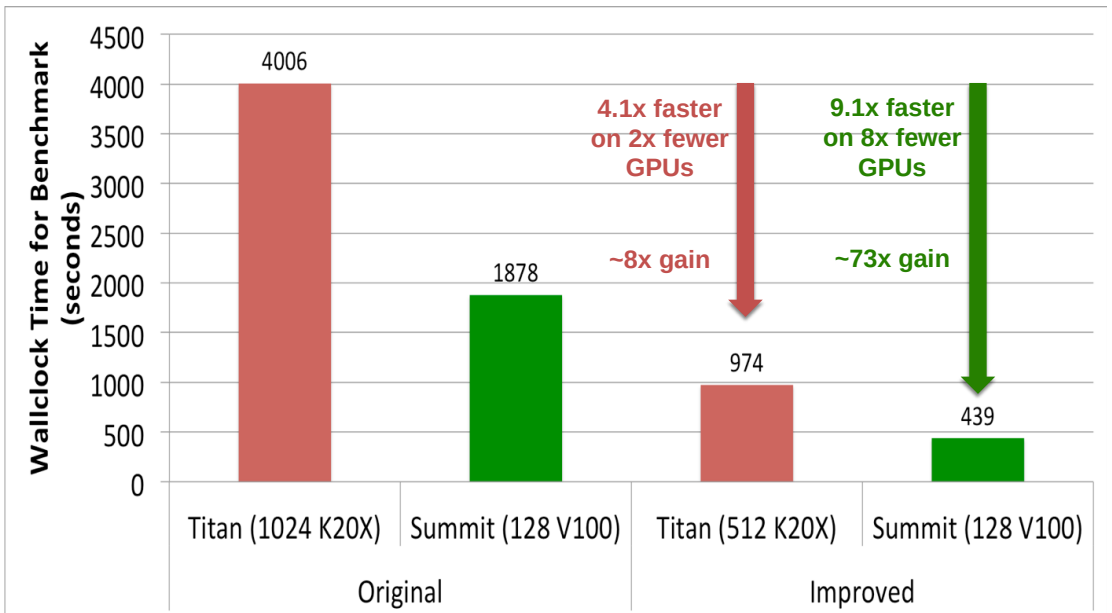

MILC/QUDA CG solver
Condition number ~$10^6$

# EXTREME EXAMPLE

Chroma/QUDA multigrid solver uses 5 different precisions

- double – outer defect correction
- single – GCR solver
- half – preconditioner
- quarter – preconditioner halo communication
- int32 – deterministic parallel coarsening

Reduced precision critical for memory traffic and footprint reduction



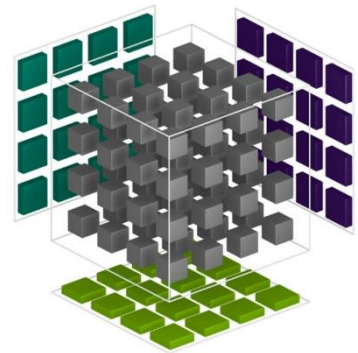

Chroma HMC benchmark (Bálint Joó, Jlab)

Machine x Algorithm = 73x speedup

ECP >50x goal achieved pre-Exascale

# CUDA TENSOR CORE PROGRAMMING

## 16x16x16 Warp Matrix Multiply and Accumulate (WMMA)



$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,\ldots} & A_{0,15} \\ A_{1,0} & A_{1,1} & A_{1,\ldots} & A_{1,15} \\ A_{\ldots,0} & A_{\ldots,1} & A_{\ldots,\ldots} & A_{\ldots,15} \\ A_{15,0} & A_{15,1} & A_{15,\ldots} & A_{15,15} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,\ldots} & B_{0,15} \\ B_{1,0} & B_{1,1} & B_{1,\ldots} & B_{1,15} \\ B_{\ldots,0} & B_{\ldots,1} & B_{\ldots,\ldots} & B_{\ldots,15} \\ B_{15,0} & B_{15,1} & B_{15,\ldots} & B_{15,15} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,\ldots} & C_{0,15} \\ C_{1,0} & C_{1,1} & C_{1,\ldots} & C_{1,15} \\ C_{\ldots,0} & C_{\ldots,1} & C_{\ldots,\ldots} & C_{\ldots,15} \\ C_{15,0} & C_{15,1} & C_{15,\ldots} & C_{15,15} \end{pmatrix}$$

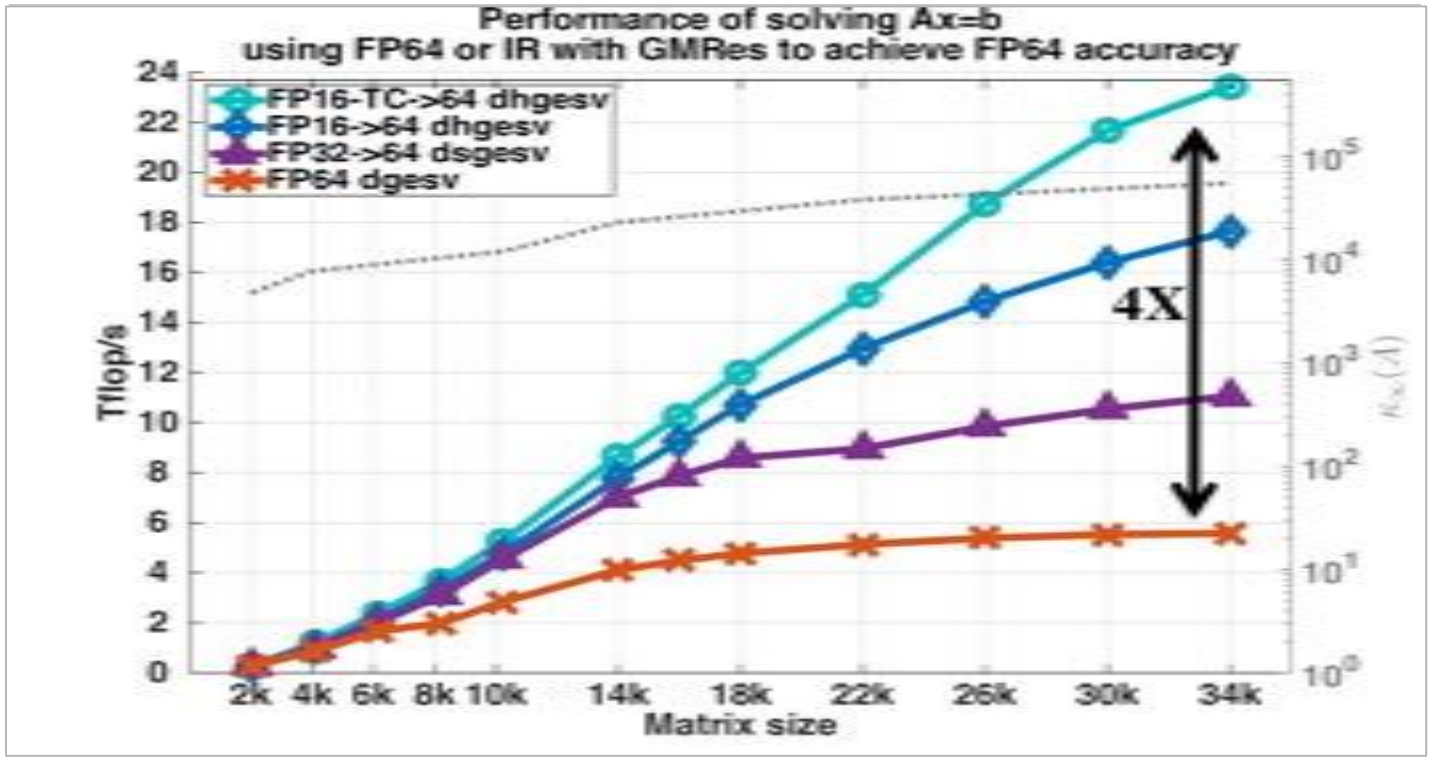
FP16 or FP32 — FP16 — FP16 — FP16 or FP32

$$D = AB + C$$



NVIDIA.

# LINEAR ALGEBRA + TENSOR CORES




MAGMA Library with FP16

## Double Precision LU Decomposition

- Compute initial solution in FP16
- Iteratively refine to obtain same accuracy as FP64

Achieved FP16->FP64 Tflops: **24**

Achieved FP64 Tflops: **5.8**

Device FP64 Tflops: **7.5**

Data courtesy of: Azzam Haidar, Stan. Tomov & Jack Dongarra, Innovative Computing Laboratory, University of Tennessee
"*Harnessing GPU Tensor Cores for Fast FP16 Arithmetic to Speed up Mixed-Precision Iterative Refinement Solvers*",
A. Haidar, S. Tomov, J. Dongarra, N. Higham SC'18
GTC 2018 Poster P8237: *Harnessing GPU's Tensor Cores Fast FP16 Arithmetic to Speedup Mixed-Precision Iterative Refinement Solves*

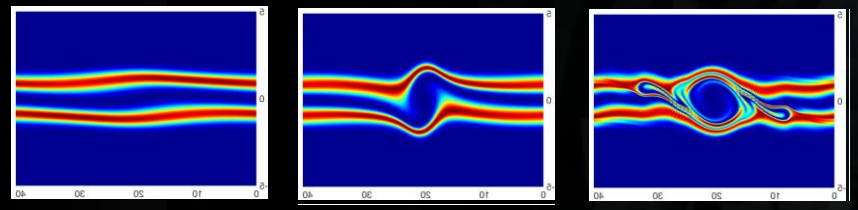
INNOVATIVE COMPUTING LABORATORY MAGMA
THE UNIVERSITY OF TENNESSEE


NVIDIA

# ADVANCING FUSION DISCOVERIES

## ASGarD: Adaptive Sparse Grid Discretization

### Two stream instability study



Scientists believe fusion is the future of energy but maintaining plasma reactions is challenging and disruptions can result in damage to the tokamak. Researchers at ORNL are simulating instabilities in the plasma to provide physicists a better understanding of what happens inside the reactor.

**With NVIDIA Tensor Cores the simulations run 3.5X faster** than previous methods so the team can simulate significantly longer physical times and help advance our understanding of how to sustain the plasma and generate energy

MAGMA

NVIDIA

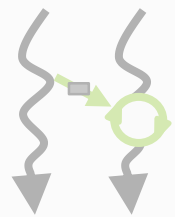
OAK RIDGE National Laboratory

Joint work with ORNL & UTK David Green, Ed Azevedo, Wael Elwasif, Graham Lopez, Tyler McDaniel, Lin Mu, Stan Tomov, Jack Dongarra

# CUDA PROGRAMMING ARCHITECTURE

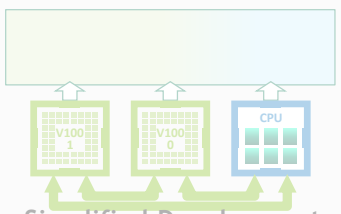
## Recent developments and directions

Improved SIMT Model



New Algorithms

Unified Memory

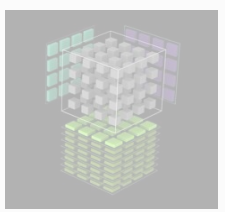



Simplified Development

Languages, directives

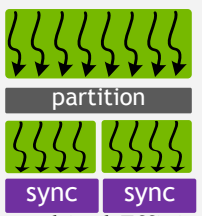

Productive parallel programming

Tensor Core



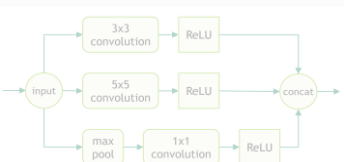
Multi-Precision

**Cooperative Groups**



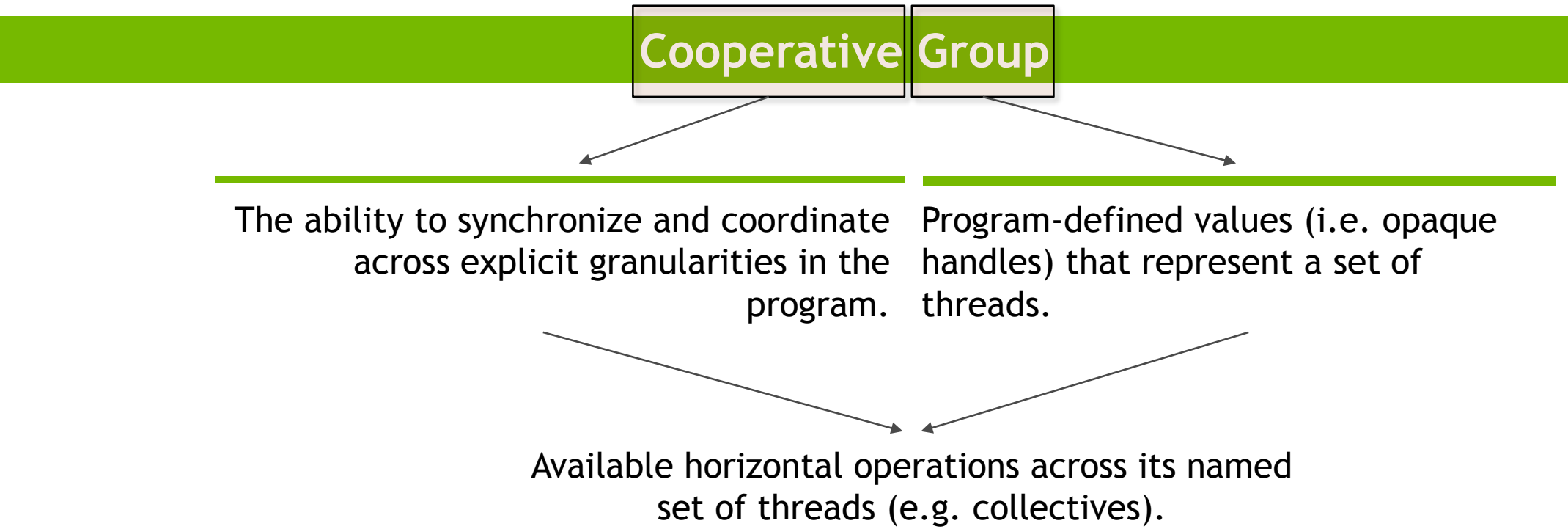

**Hierarchical Efficiency**

CUDA Graphs




Small Task Efficiency

# CUDA 9+: COOPERATIVE GROUPS



# SCOPE OF COOPERATION

Warp

SM

GPU

Node

**For currently converged threads:**
```
auto group = cg::coalesced_threads();
```

**For CUDA thread blocks:**
```
auto group = cg::this_thread_block();
```

**For cooperative grids:**
```
auto group = cg::this_grid();
```

**For cooperative grids that span devices:**
```
auto group = cg::this_multi_grid();
```

Cooperating on an SM

Cooperating across a GPU

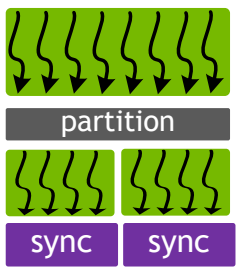
Cooperating across
many GPUs with NVLINK

# SYNCHRONIZE AT ANY SCALE

## Three Key Capabilities

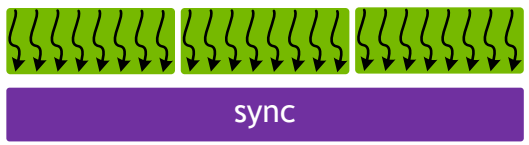
### FLEXIBLE GROUPS

Define and synchronize arbitrary groups of threads

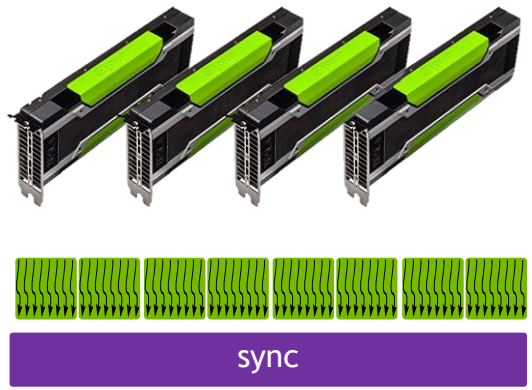



### WHOLE-GRID SYNCHRONIZATION

Synchronize multiple thread blocks




### MULTI-GPU SYNCHRONIZATION



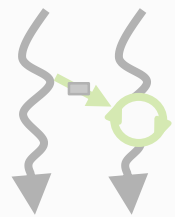



NVIDIA.

# CUDA PROGRAMMING ARCHITECTURE

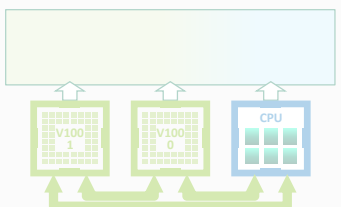
## Recent developments and directions

Improved SIMT Model



New Algorithms

Unified Memory

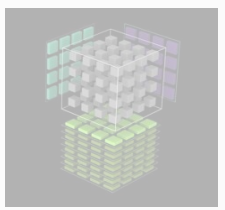



Simplified Development

Languages, directives

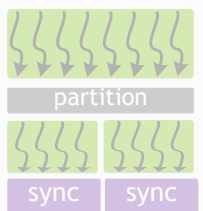

Productive parallel programming

Tensor Core



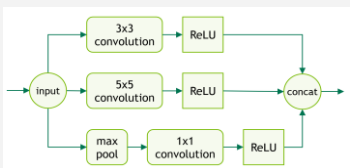
Multi-Precision

Cooperative Groups




Hierarchical Efficiency

CUDA Graphs




Small Task Efficiency

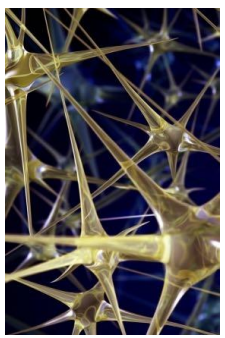
# ASYNCHRONOUS GRAPHS

## Execution optimization when workflow is known up front

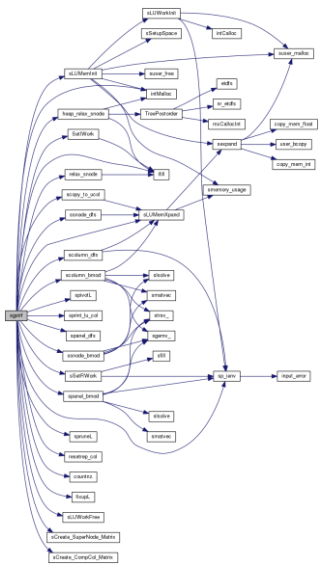
```
// Basic function to test primality.
bool IsPrime( size_t n)
{
      if (n == 2) return true;
      if ((n == 1) || ((n % 2) == 0)) return false;
      size_t iters = (unsigned int)sqrt( (double)n);
      for (size_t i = 3; i <= iters; i+=2) if (n % i == 0) return false;
      return true;
}

// Compute primes from 1 to 100,000,000.
size_t ComputePrimes()
{
      size_t Primes = 0;
      for ( size_t Start = 1; Start <= 100000000; ++Start)
      {
            if ( IsPrime( Start) )
            {
                  ++Primes;
            }
      }
      return Primes;
}
```

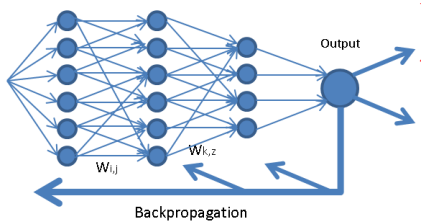
Loop & Function offload



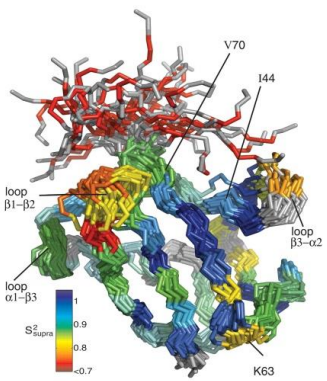
DL Inference



Linear Algebra




Deep Neural Network Training



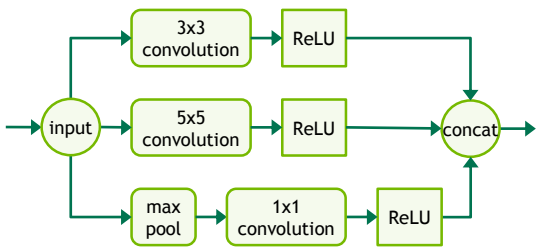

HPC Simulation



NVIDIA.

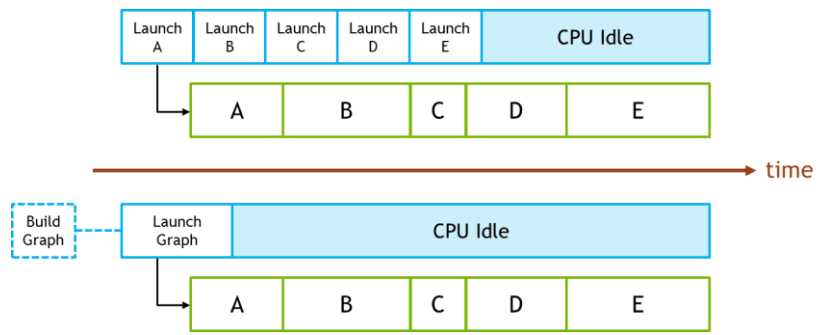
# THE GRAPH ADVANTAGE

## WHOLE WORKFLOW OPTIMIZATIONS




Seeing all work at once enables
new optimizations in hardware and software

## EFFICIENT LAUNCH OF COMPLEX WORK




Launch potentially thousands of work items
with a single call

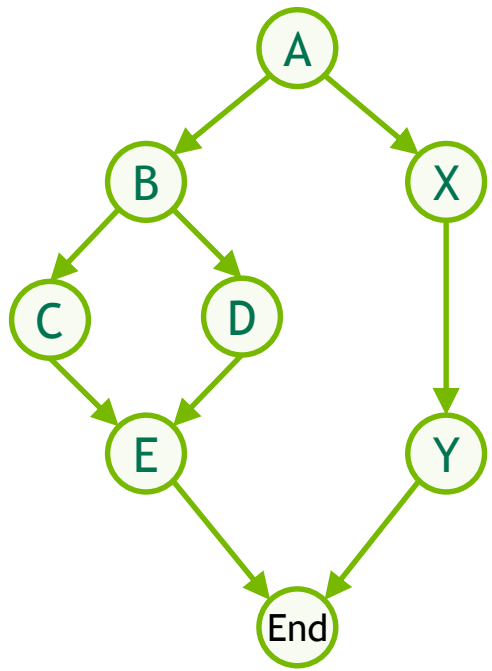
# DEFINITION OF A CUDA GRAPH

## Graph Nodes Are Not Just Kernel Launches

Sequence of operations, connected by dependencies.

Operations are one of:

| | |
|---|---|
| Kernel launch | CUDA kernel running on GPU |
| CPU function call | Callback function on CPU |
| Memcopy/Memset | GPU data management |
| Sub-Graph | Graphs are hierarchical |



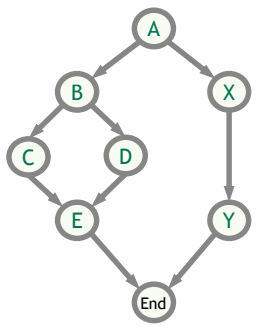


NVIDIA.

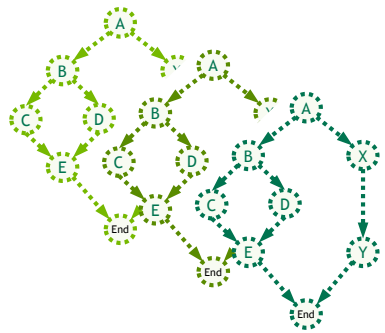
# THREE-STAGE EXECUTION MODEL

## Early structure binding amortizes overheads, late data binding enables iteration

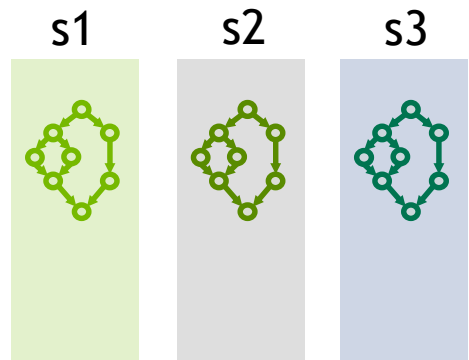
### Define




### Instantiate




### Execute




Single Graph "Template"
Mutable structure

Created in host code
or built up from libraries

Multiple "Executable Graphs"
Fixed structure, mutable data

Snapshot of template
Sets up & initializes GPU
execution structures
(create once, run many times)

Executable Graphs
Running in CUDA Streams
Fixed structure and data

Concurrency in graph
**is not** limited by stream


NVIDIA.

# GPUDIRECT FAMILY

## Technologies enabling products

### GPUDIRECT SHARED GPU-SYSMEM

GPU pinned memory shared with other RDMA-capable devices
Avoids intermediate copies

### GPUDIRECT P2P

Accelerated GPU-GPU memory copies
Inter-GPU direct load/store access

### GPUDIRECT RDMA

Direct GPU to 3rd party device transfers
E.g. direct I/O, optimized inter-node communication

### GPUDIRECT ASYNC

Direct GPU to 3rd party device synchronizations
E.g. optimized inter-node communication

https://developer.nvidia.com/gpudirect

http://docs.nvidia.com/cuda/gpudirect-rdma

# NCCL

## NVIDIA Collective Communications Library

Optimized collective communication library between CUDA devices.

Easy to integrate into any DL framework, as well as traditional HPC apps using MPI.

Runs on the GPU using asynchronous CUDA kernels, for faster access to GPU memory, parallel reductions, NVLink usage.

Similar to MPI collectives, but has a CUDA stream parameter, operates on CUDA pointers.

→ Enables use of multiple threads per MPI rank, multiple GPUs per thread.

Minimizes GPU threads to permit other computation to progress simultaneously.

NCCL 2 adds support for inter-node communication using Sockets or IB verbs.

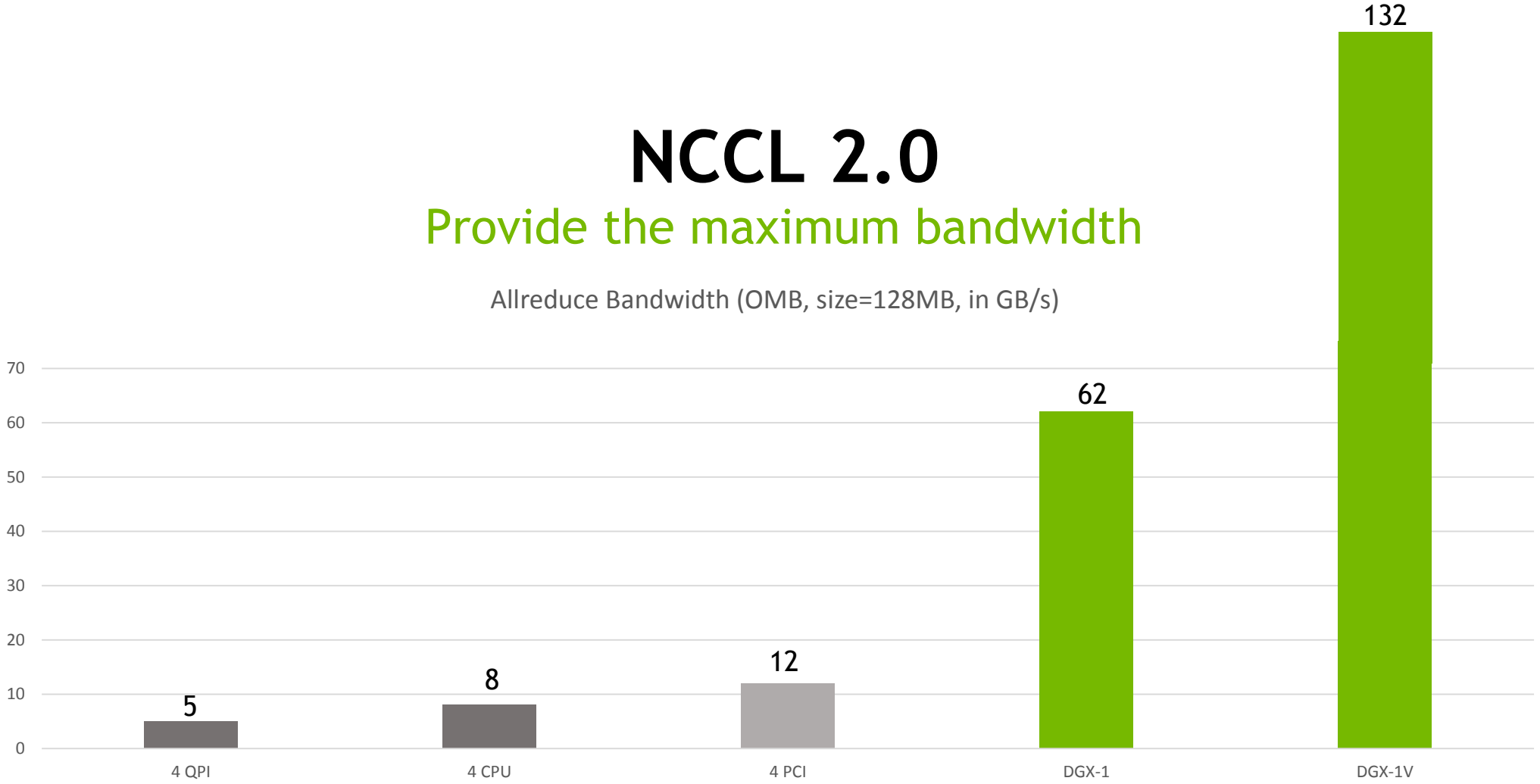
Uses staging buffers on GPU within each process, so no IPC overhead


NVIDIA.

# NCCL 2.0

## Provide the maximum bandwidth

Allreduce Bandwidth (OMB, size=128MB, in GB/s)

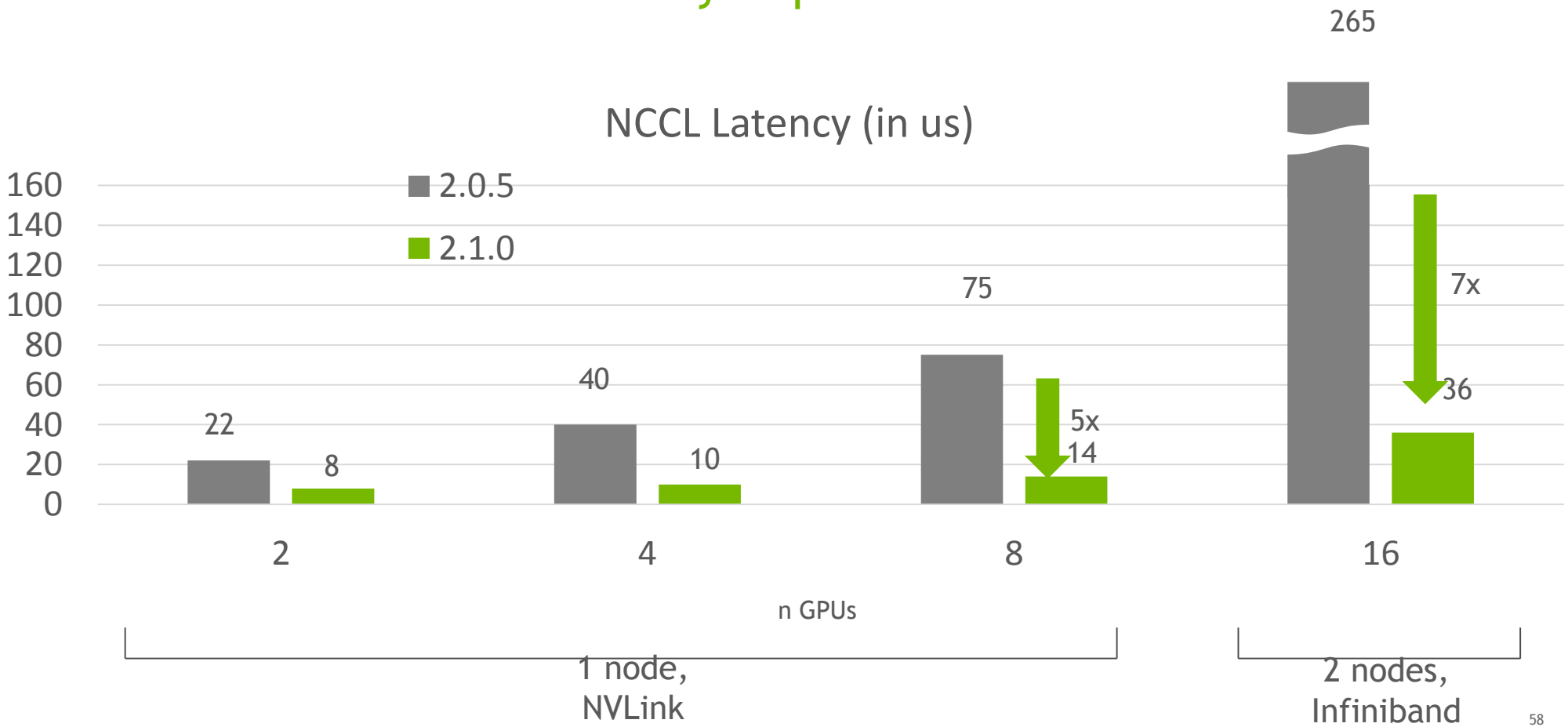





NVIDIA

# NCCL 2.1

## Latency improvement






NVIDIA.

# NVSHMEM

## Programming abstraction for communication among GPUs

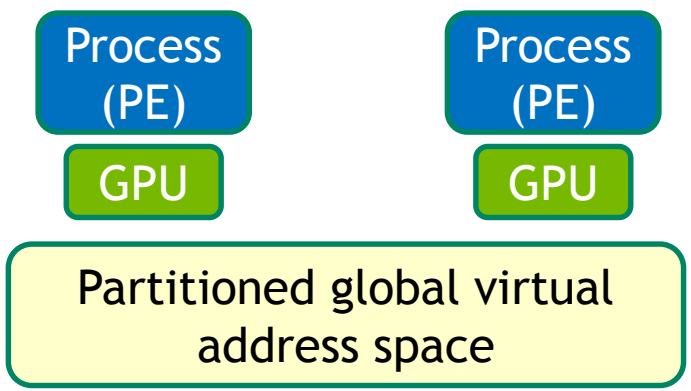
shmem_init – set up communication, including CUDA P2P, IPC

shmem_malloc – get pointer to perform allocation on local GPU

shmem_ptr – given a PE, get the base address of the symmetric heap on remote GPU

Inside or outside of GPU kernel

- ld/st using offsets relative to that base address
- put/get p/g for abstracted communications
- collectives





NVIDIA.

# CONCLUSION

# THE NEW HPC WORKLOAD

## A Dramatic Shift from the Past

**PREVIOUS NORMAL**

- Simulated data sources
- Conventional 64 bit full order ab initio simulation
- Conventional reduced order methods
- All jobs are batch

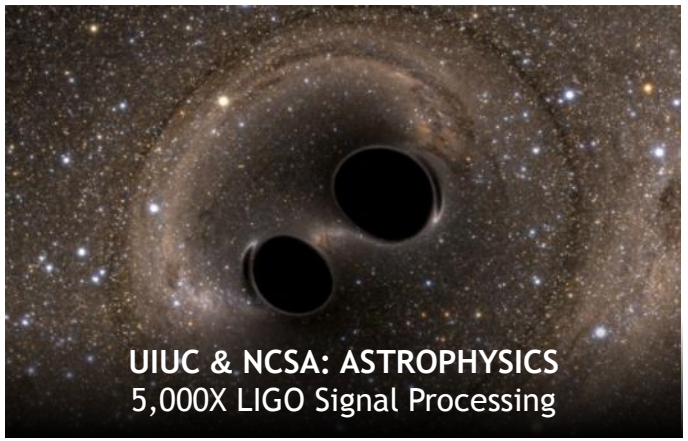
**NEW NORMAL**

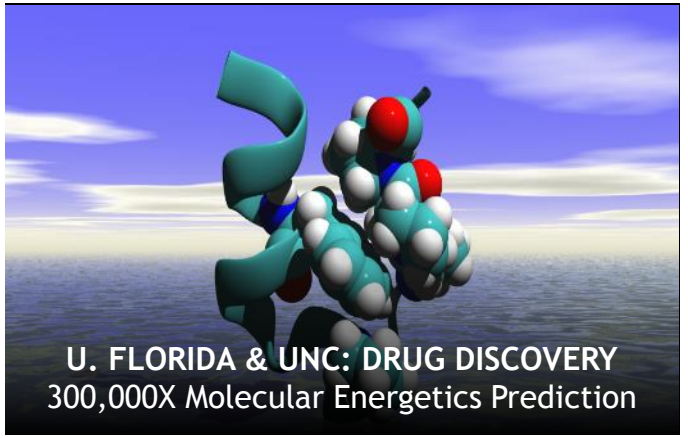
- A mix of experimental and simulated/emulated data sources
- Conventional 64 bit full order ab initio simulation
- Mixed precision algorithms with new methods for ab initio simulation
- Embedded neural nets within full order ab initio simulation
- Conventional reduced order methods
- Deep neural net and machine learning emulation
- Interactive large scale training (model parallel, data parallel and hyperparameter optimization)

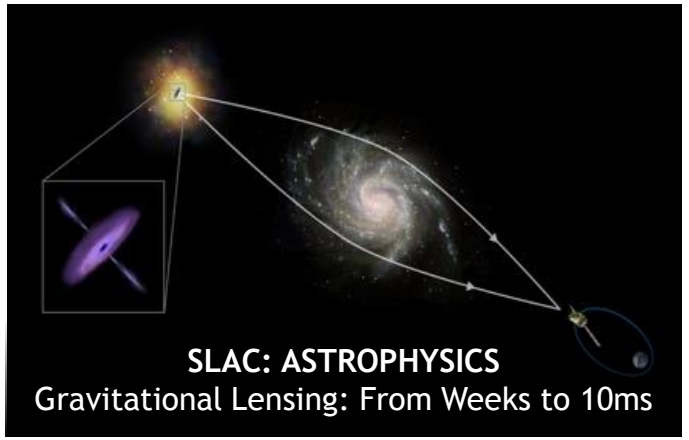


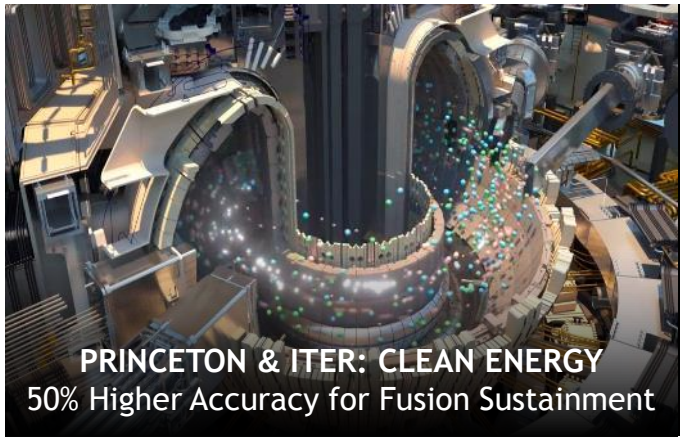
NVIDIA.

# CONVERGED HPC*AI
# EXAMPLES IN ALL DOMAINS

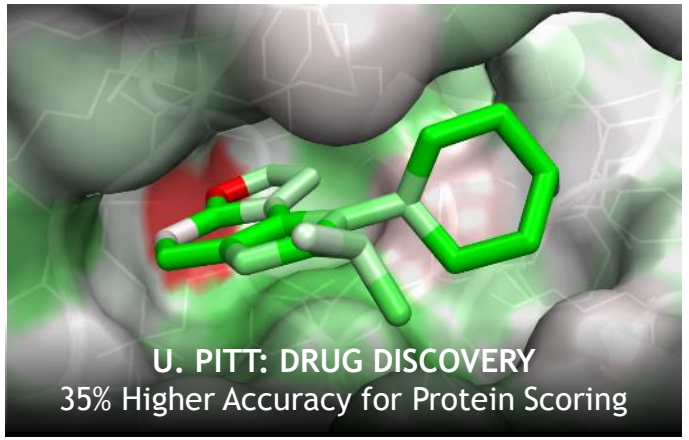


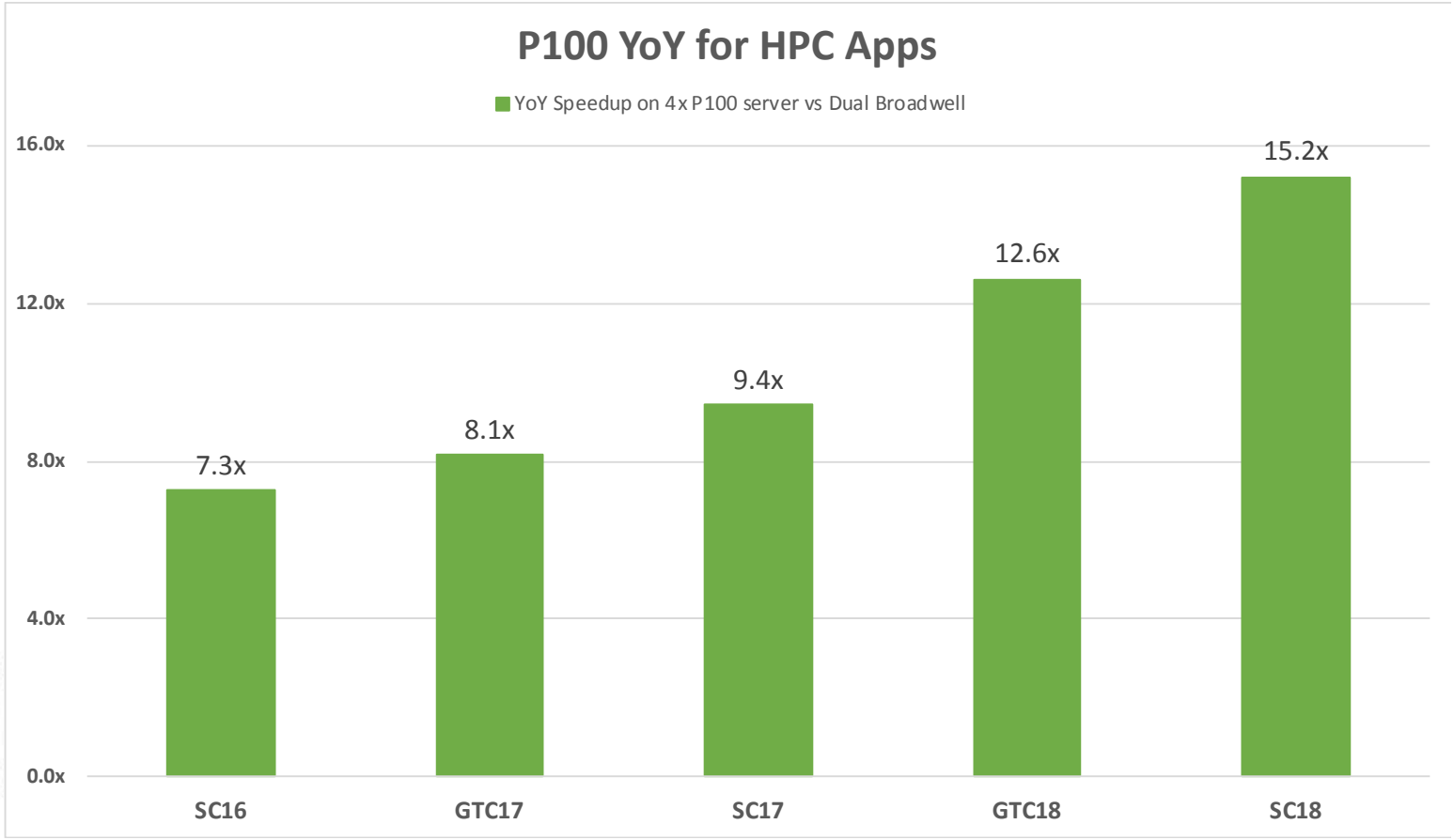




# 2X HIGHER VALUE IN 2 YEARS

## Same Hardware More Performance With Software Stack Optimizations

AMBER
Chroma
GROMACS
GTC
LAMMPS
MILC
NAMD
QE
RTM
SPECFEM3D
VASP

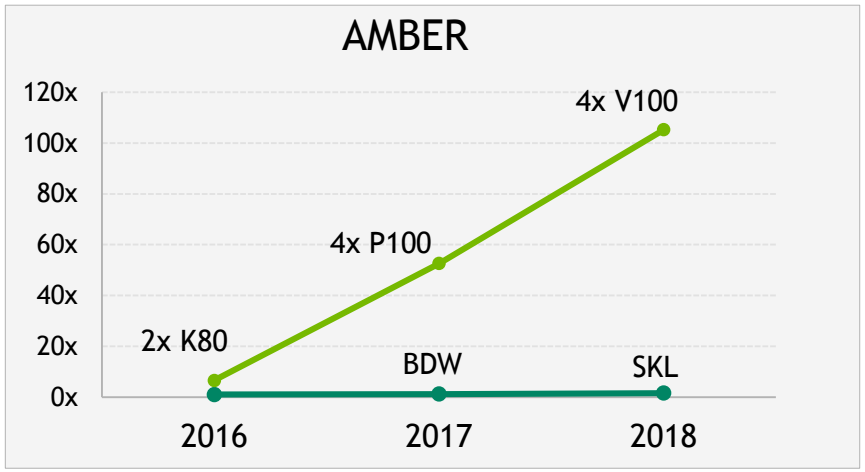


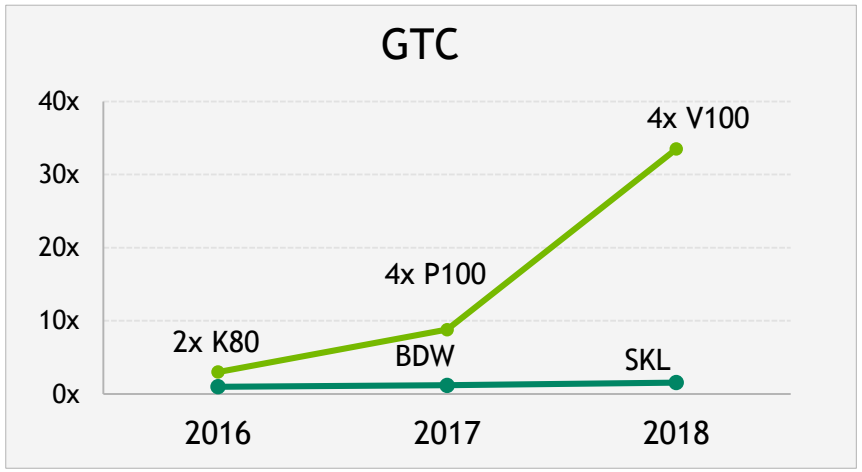

NVIDIA

# APPLICATIONS ON PATH OF ACCELERATION

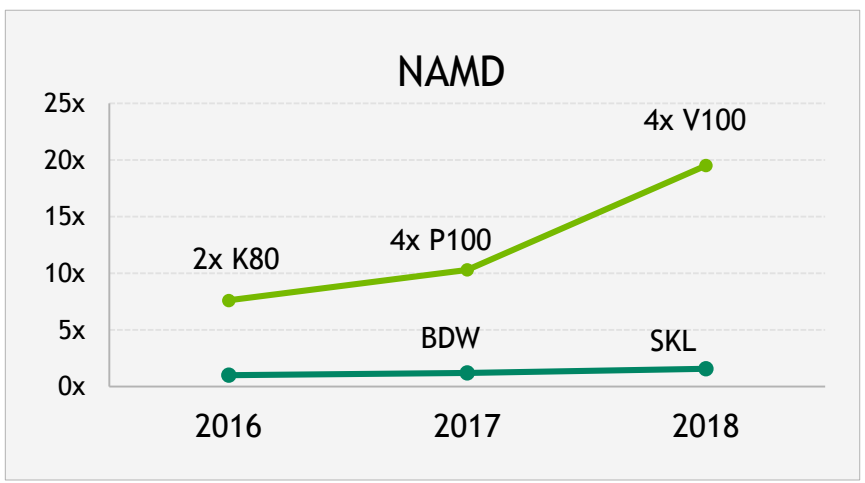
## Because status-quo yields only incremental gain

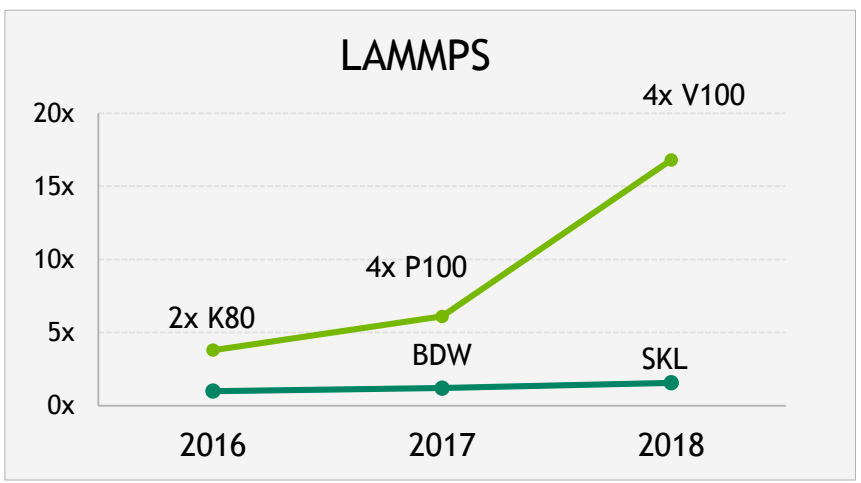
Relative to dual Haswell



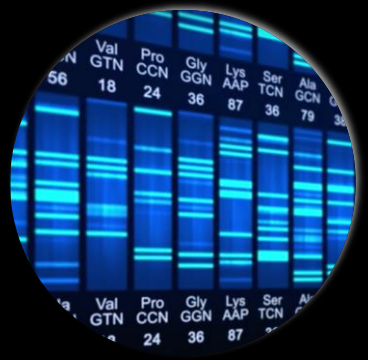

Relative to dual Haswell



# GORDON BELL NOMINEES ILLUSTRATE THE NEW NORMAL

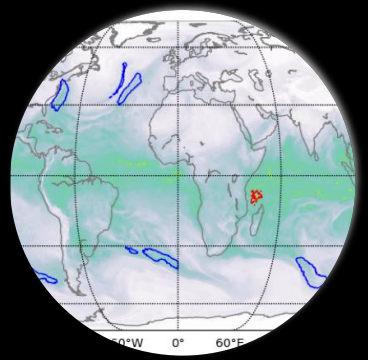
**New Algorithms**




BERKELEY LAB · OAK RIDGE National Laboratory

Genomics
2.36 ExaOps

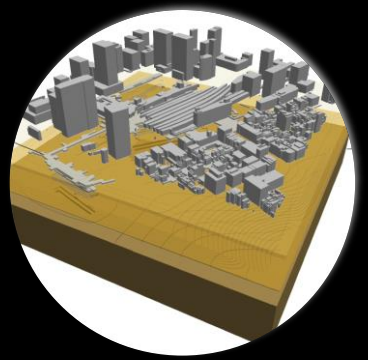
**Scalable HPC*AI**



BERKELEY LAB · NVIDIA

Weather
1.15 ExaOps

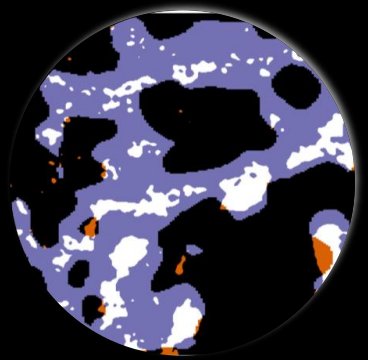
**New Algorithms HPC*AI**



東京大学 THE UNIVERSITY OF TOKYO · OAK RIDGE National Laboratory

Seismic
25X

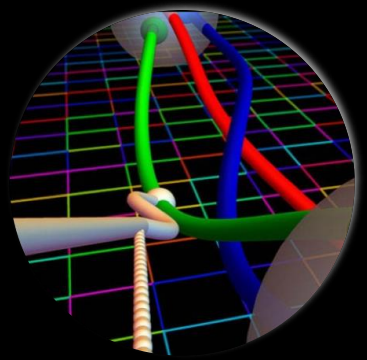
**Scalable HPC*AI**



OAK RIDGE National Laboratory

Material Science
300X

**New Algorithms**



BERKELEY LAB · Lawrence Livermore National Laboratory · NVIDIA

Quantum Chromodynamics
15x


NVIDIA