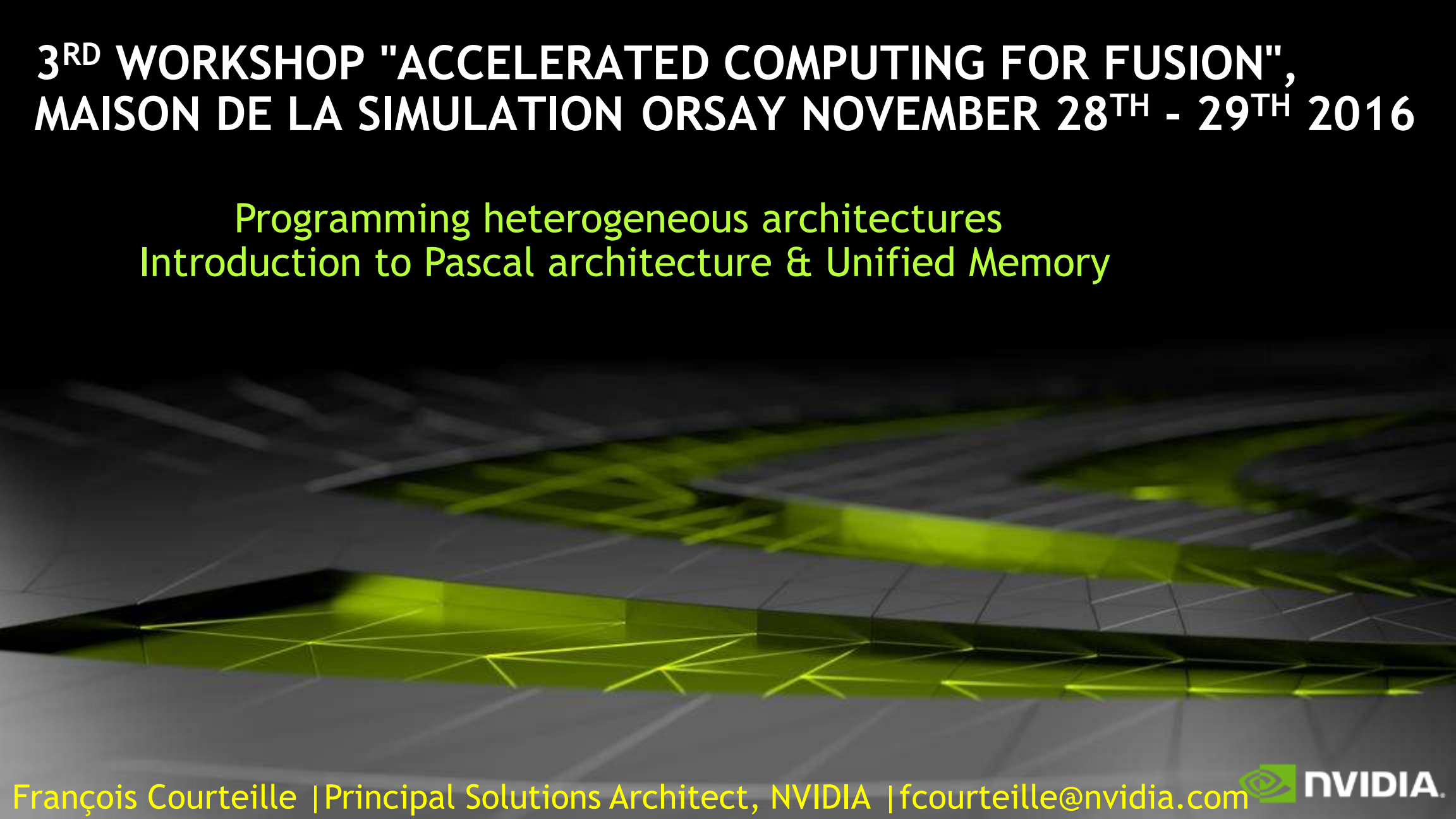
3RD WORKSHOP "ACCELERATED COMPUTING FOR FUSION", MAISON DE LA SIMULATION ORSAY NOVEMBER 28TH - 29TH 2016

Programming heterogeneous architectures
Introduction to Pascal architecture & Unified Memory



François Courteille | Principal Solutions Architect, NVIDIA | fcourteille@nvidia.com

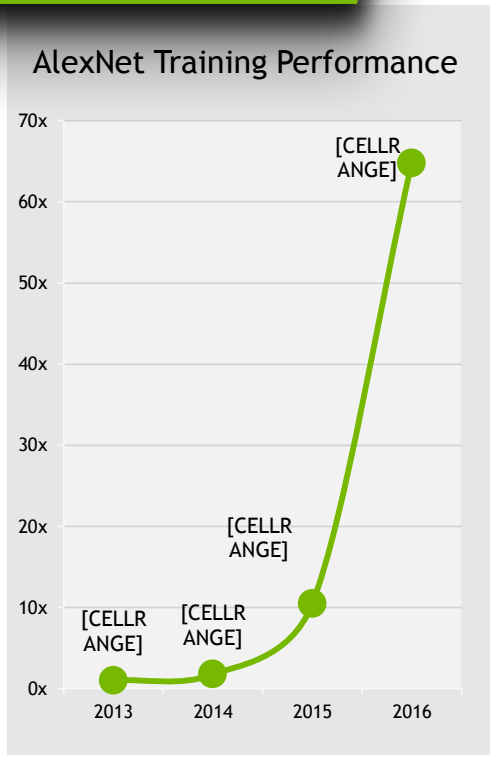
NVIDIA

ONE ARCHITECTURE BUILT FOR BOTH DATA SCIENCE & COMPUTATIONAL SCIENCE

Pascal- 5 Miracles

NVIDIA DGX-1

NVIDIA DGX SATURNV



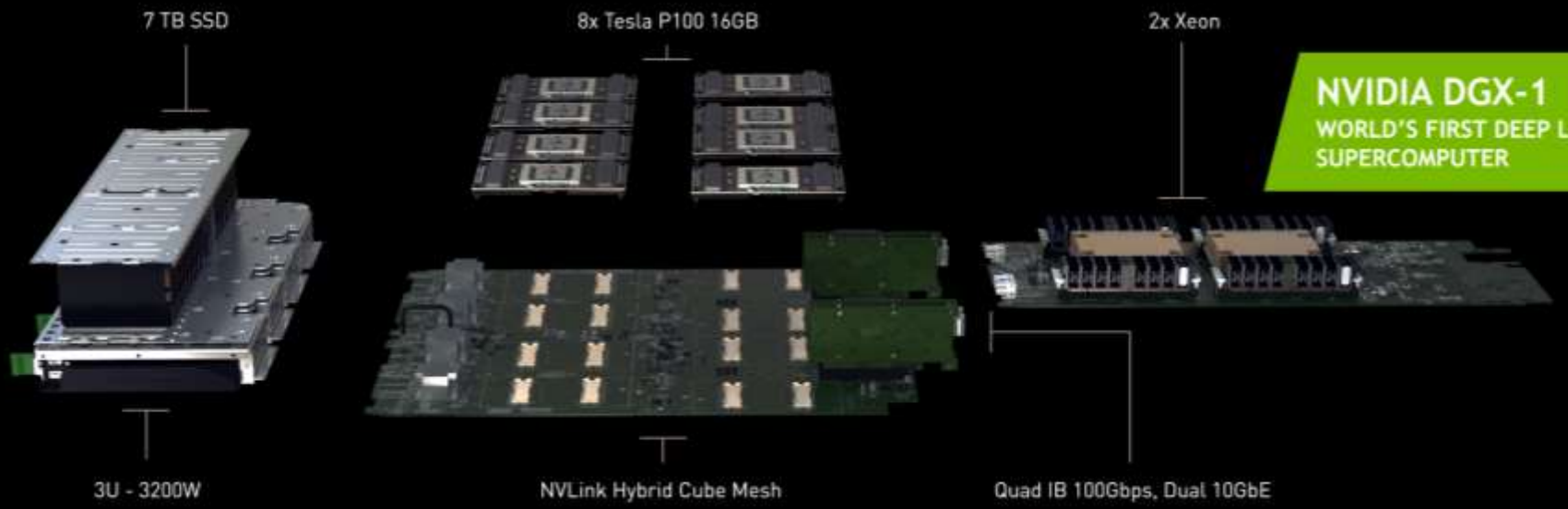
65x in 3 Years

AGENDA

The Tesla Platform : architecture and future
Rapid software development for heterogeneous architecture
ENZO : Hydrodynamics and Magnetohydrodynamics Solvers on GPU - MPS

TESLA PLATFORM

NVIDIA DGX-1 DEEP LEARNING SYSTEM

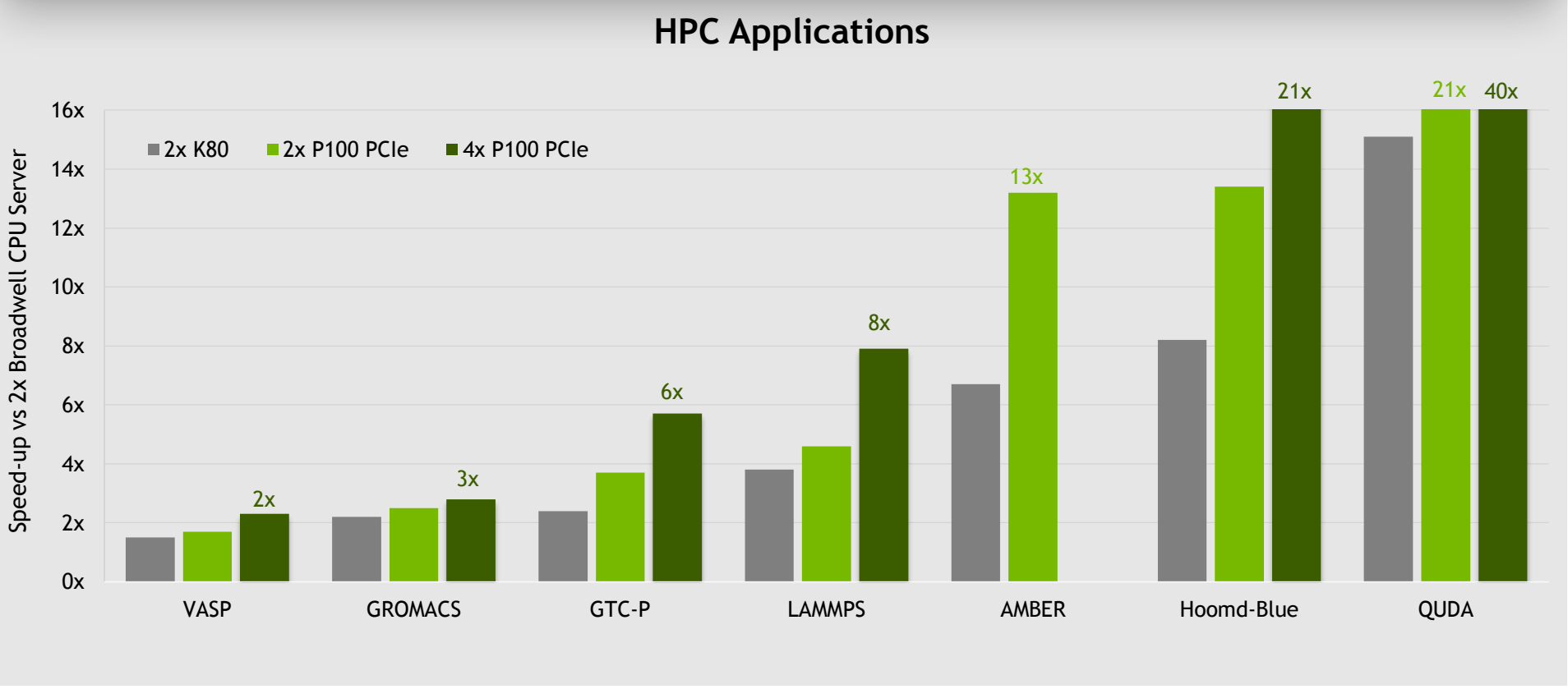


TESLA P100 ACCELERATORS

	 Tesla P100 with NVLink	 Tesla P100 for PCIe
Compute	5.3 TF DP · 10.6 TF SP · 21.2 TF HP	4.7 TF DP · 9.3 TF SP · 18.7 TF HP
Memory	HBM2: 732 GB/s · 16 GB	HBM2 16GB: 732 GB/s HBM2 12GB: 549 GB/s
Interconnect	NVLink (160 GB/s) + PCIe Gen3 (32 GB/s)	PCIe Gen3 (32 GB/s)
Programmability	Page Migration Engine Unified Memory	Page Migration Engine Unified Memory
Power	300W	250W

Interconnect Speed is measured bi-directional

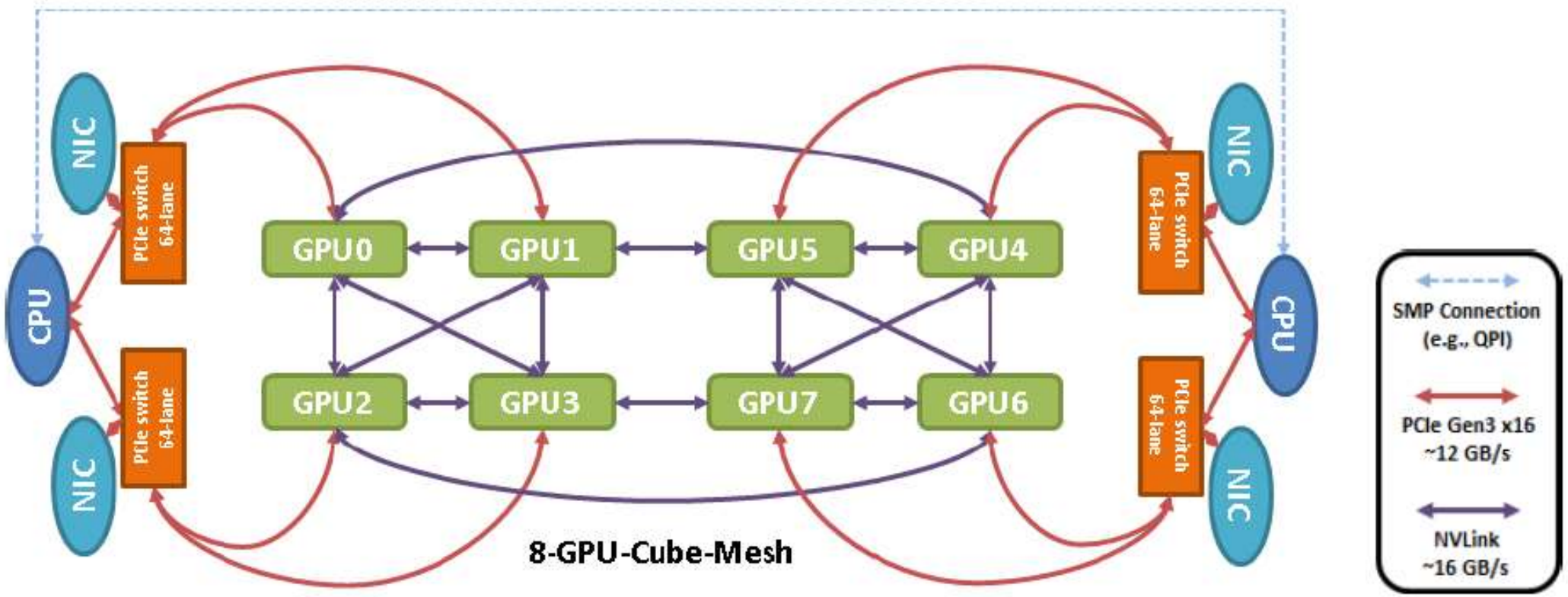
P100: FASTEST PERFORMANCE FOR HPC



CPU Server: Dual Xeon E5-2699 v4@2.2GHz (44-core CPU), GPU Servers: Dual Xeon E5-2699 v4@2.2GHz (44-core CPU) with Tesla K80s, P100s PCIe
Dataset: VASP- Silica IFPEN, GROMACS- Water 3M, GTC-P- Model A, LAMMPS- 256x256x256 local size,
QUDA 0.9 vs QPhiX- Dslash Wilson-Clover,Precisio: Double, Problem Size 32x32x32x64, AMBER- PME-Cellulose-NVE, Hoomd-Blue- lj_liquid_1m

NVLINK

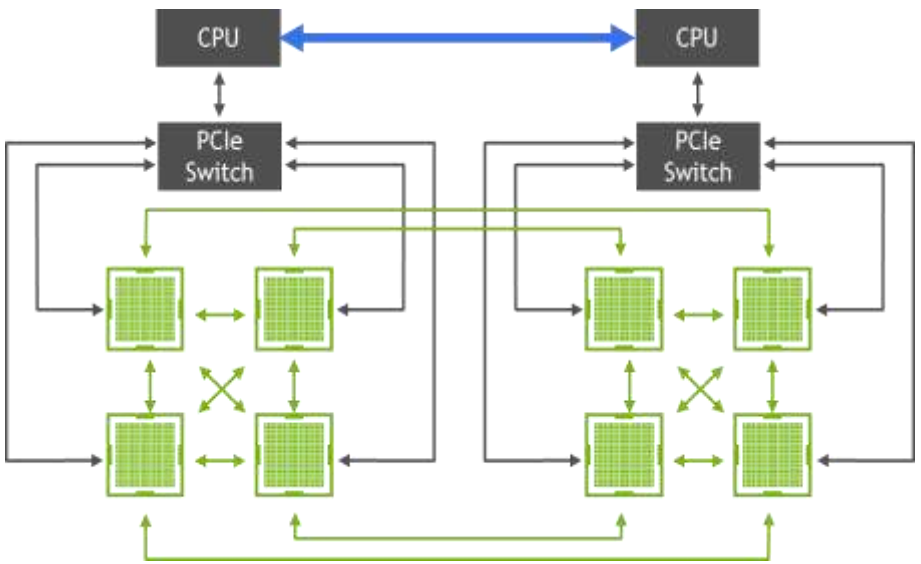
GPU-TO-GPU NVLINK TOPOLOGY



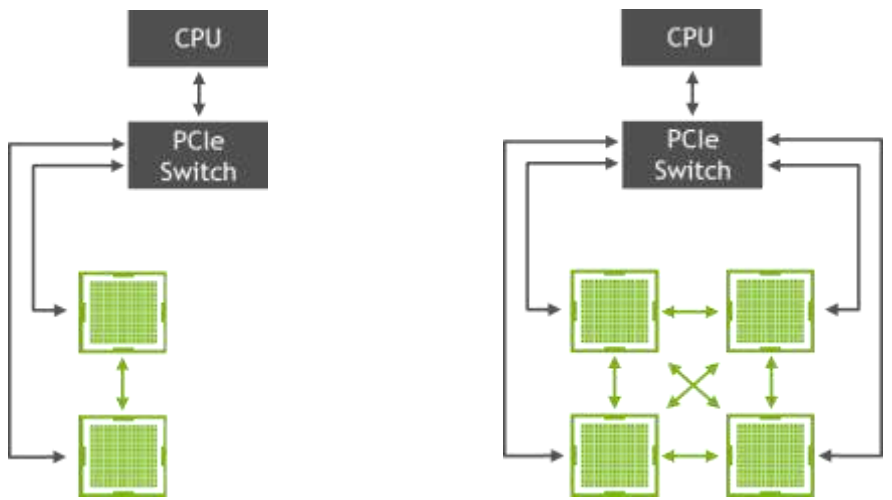
For the 8-GPU-Cube-Mesh topology, there is no need to use PCIe for any GPU-to-GPU communications (whether point-to-point or collective).

MOST SCALABLE, MOST VERSATILE

8-GPU Cube Mesh with NVLINK



Best in class scaling with 8-GPU for Wide variety of workloads



Easy to partition into logical systems of two or four GPUs

NVLINK TO CPU

IBM Power Systems Server S822LC (codename “Minsky”)

2x IBM Power8+ CPUs and 4x P100 GPUs

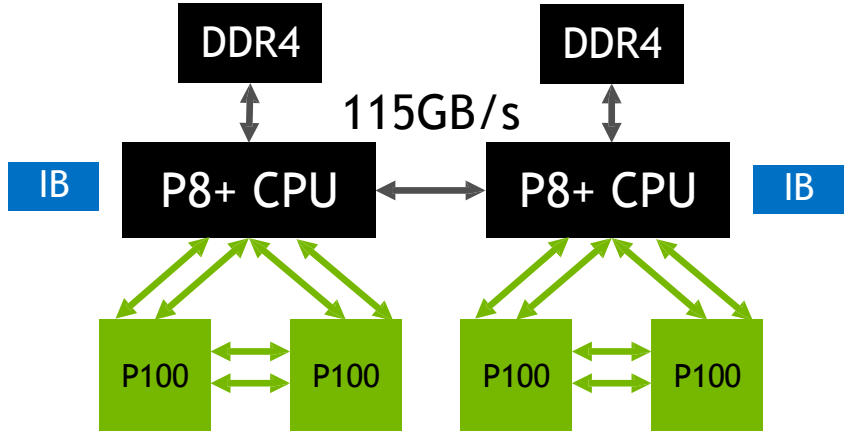
80 GB/s per GPU bidirectional for peer traffic

80 GB/s per GPU bidirectional to CPU

115 GB/s CPU Memory Bandwidth

Direct Load/store access to CPU Memory

High Speed Copy Engines for bulk data movement

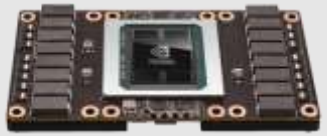


END-TO-END PASCAL PRODUCT FAMILY

Training - Tesla P100

Inference - Tesla P40 & P4

Hyperscale HPC



Tesla P100 with NVLink

Strong-Scale HPC

Tesla P100 with PCI-E

Mixed-App HPC

DGX-1

Fully Integrated DL Supercomputer

RAPID SOFTWARE DEVELOPMENT ON HETEROGENEOUS SYSTEMS

GPU COMPUTING IN STANDARD LANGUAGES

Progress on a C++ Standard Parallel Algorithms

Source Compilers Support GPUs with OpenACC/OpenMP

GCC/GFORTRAN

OpenUH

OpenARC

ACCUL

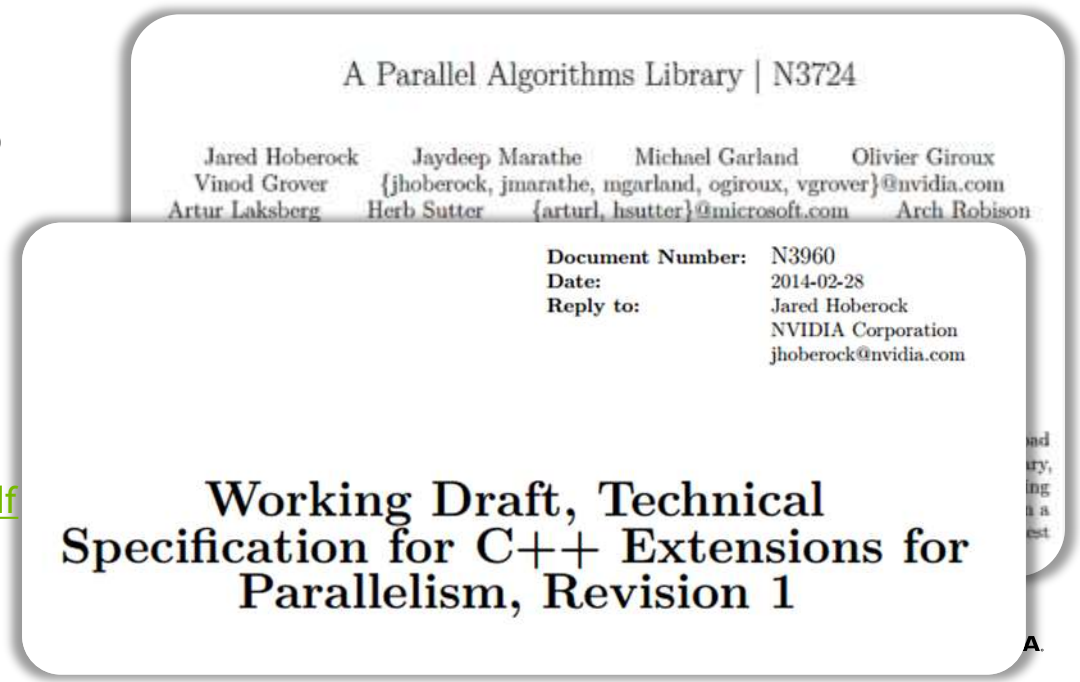
Numba: Open Source Python compiler now supports GPUs

Prototype Java Bytecode Compiler for GPUs

Portable, High-level Parallel Code TODAY

- Thrust library allows the same C++ code to target both:
  - NVIDIA GPUs
  - x86, ARM and POWER CPUs

- Thrust was the inspiration for a proposal to the ISO C++ Committee
- Committee voted unanimously to accept as official tech. specification working draft



http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3960.pdf

https://github.com/n3554/n3554

STANDARDIZING PARALLEL STL

Technical Specification for C++ Extensions for Parallelism

Published as ISO/IEC TS 19570:2015, July 2015

Draft available online

http://www.openstd.org/jtc1/sc22/wg21/docs/papers/2015/n4507.pdf

We've proposed adding this to C++17

http://www.openstd.org/jtc1/sc22/wg21/docs/papers/2015/p0024r0.html

MORE C++ PARALLEL FOR LOOPS

GPU Lambdas Enable Custom Parallel Programming Models

Kokkos

https://github.com/kokkos

```
Kokkos::parallel_for(N, KOKKOS_LAMBDA (int i) {
  y[i] = a * x[i] + y[i];
});
```

RAJA

https://e-reports-ext.llnl.gov/pdf/782261.pdf

```
RAJA::forall<cuda_exec>(0, N, [=] __device__ (int i) {
  y[i] = a * x[i] + y[i];
});
```

Hemi

CUDA Portability Library

http://github.com/harrism/hemi

```
hemi::parallel_for(0, N, [=] HEMI_LAMBDA (int i) {
  y[i] = a * x[i] + y[i];
});
```

OPENACC

World's Only Performance Portable Programming Model for HPC

Add Simple Compiler Hint

```
main()
{
  <serial code>
  #pragma acc kernels
  {
    <parallel code>
  }
}
```

Simple

ARM

PEZY

POWER

Sunway

x86 CPU

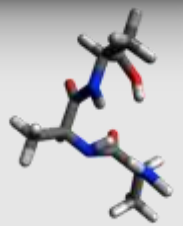
x86 Xeon Phi

NVIDIA GPU

Portable

LSDALTON

Simulation of molecular energies



Quicker Development

Lines of Code Modified

<100 Lines

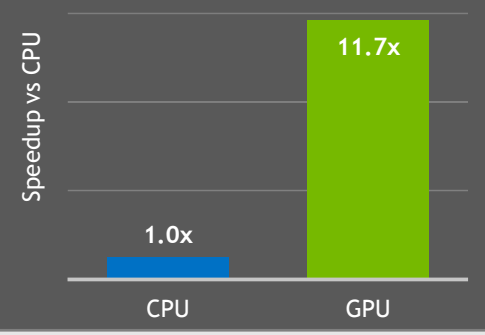
# of Weeks Required

1 Week

Big Performance

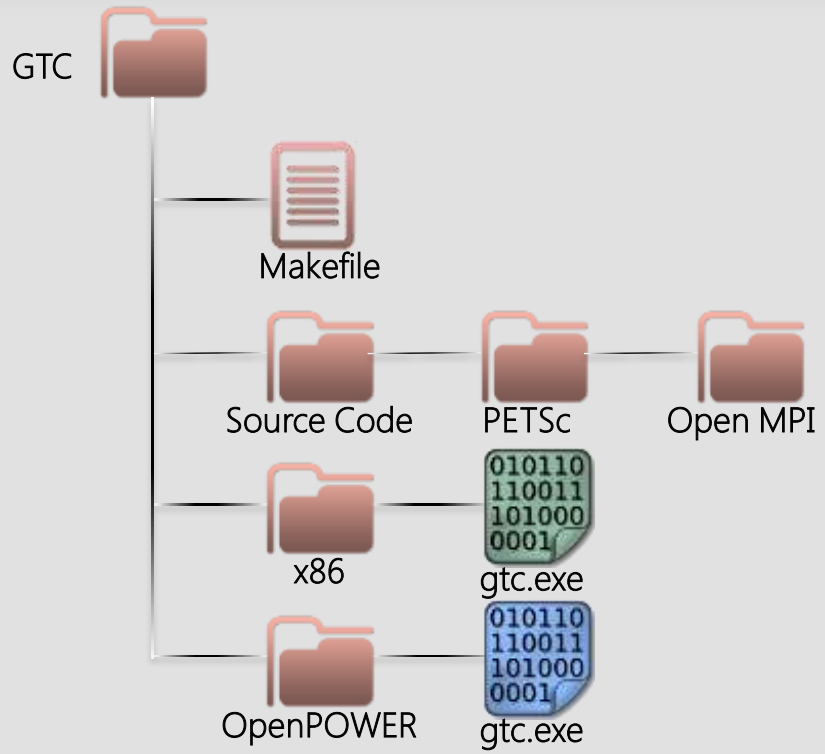
CCSD(T) Module, Alanine-3

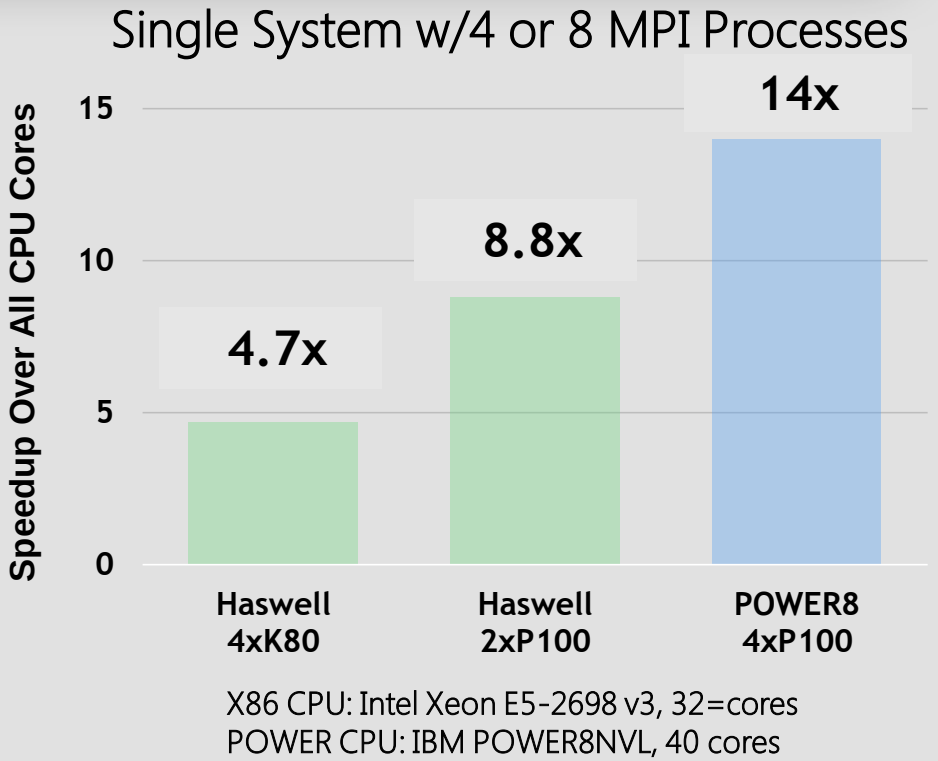
Titan System: AMD CPU vs Tesla K20X



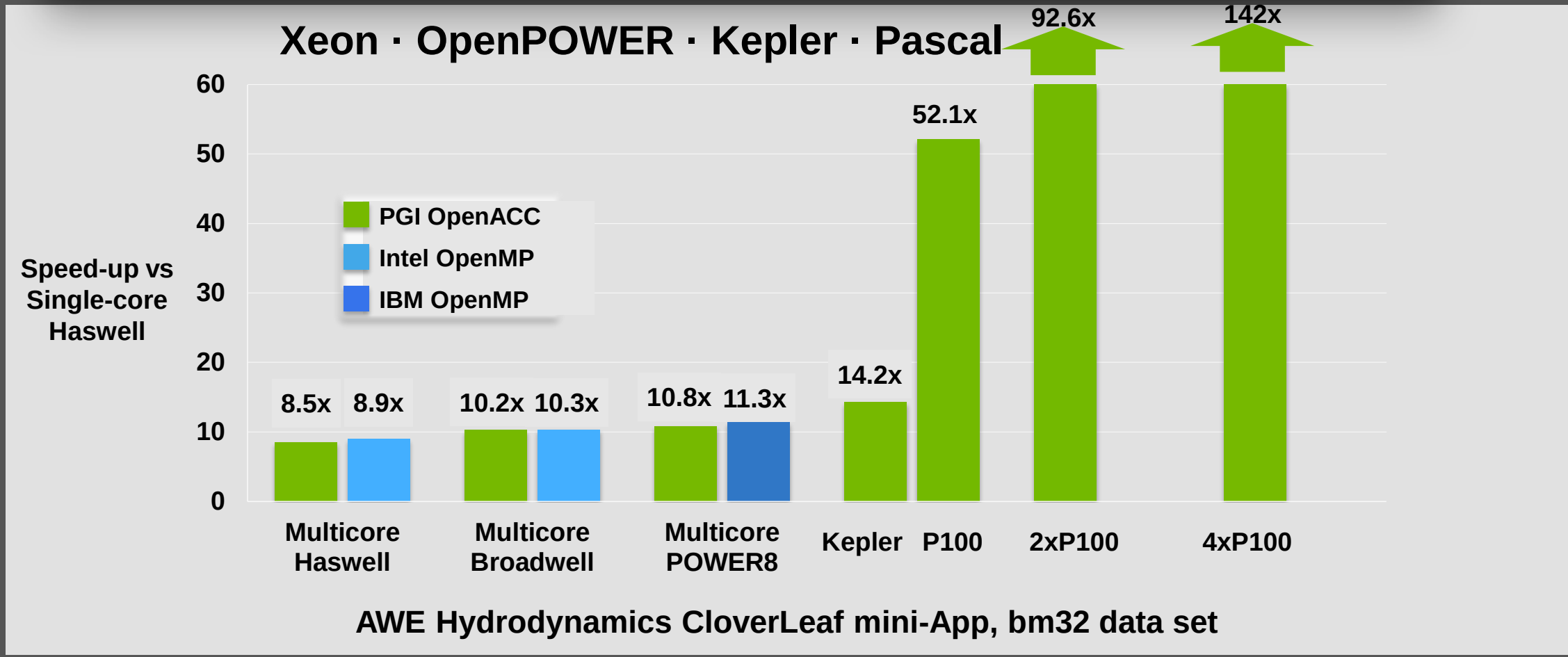
Powerful

PORTING THE OPENACC VERSION OF GTC FROM XEON TO OPENPOWER



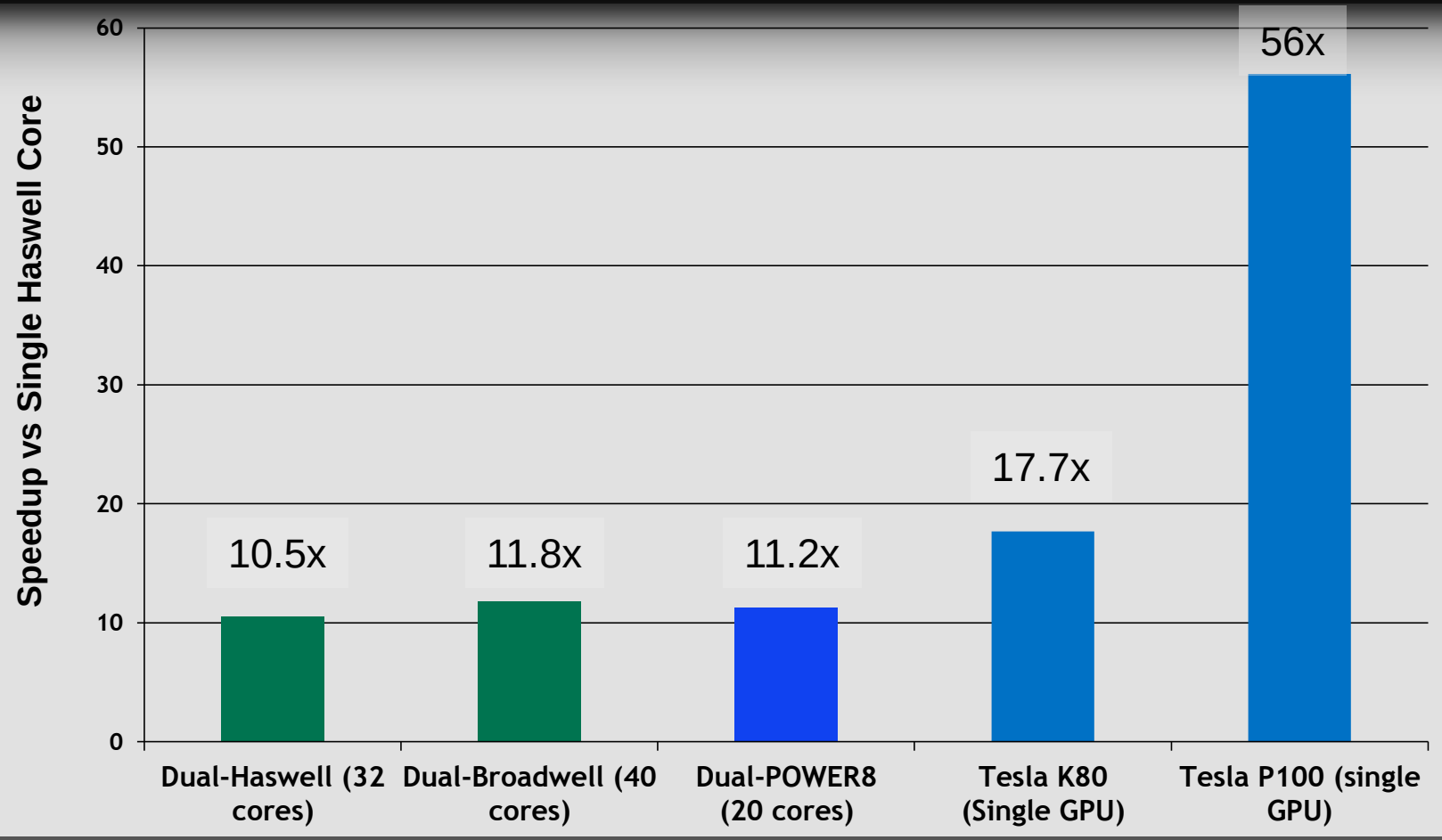


OPENACC PERFORMANCE PORTABILITY - CLOVERLEAF 1.3



OPENACC SPEC ACCEL 1.0 BENCHMARKS

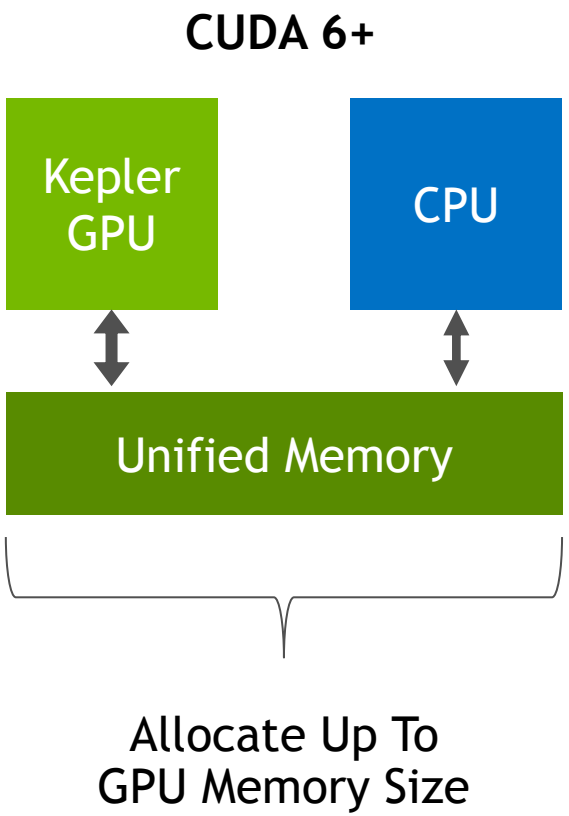
Geometric mean across all 15 benchmarks



Performance measured November, 2016 and are considered estimates per SPEC run and reporting rules. SPEC® and SPEC ACCEL® are registered trademarks of the Standard Performance Evaluation Corporation (www.spec.org).

UNIFIED MEMORY

KEPLER/MAXWELL UNIFIED MEMORY



Simpler Programming & Memory Model

- Single allocation, single pointer, accessible anywhere
- Eliminate need for *explicit copy*
- Greatly simplifies code porting

Performance Through Data Locality

- Migrate data to accessing processor
- Guarantee global coherency
- Still allows explicit hand tuning

PAGE MIGRATION ENGINE

Support Virtual Memory Demand Paging

49-bit Virtual Addresses

- Sufficient to cover 48-bit CPU address + all GPU memory

GPU page faulting capability

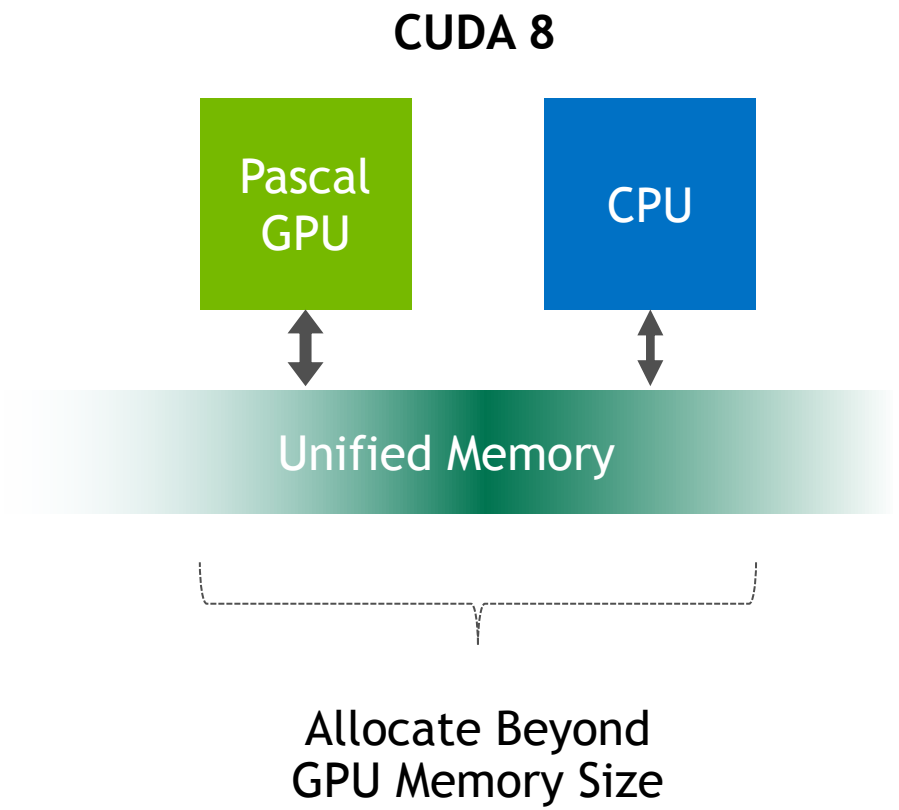
- Can handle thousands of simultaneous page faults

Up to 2 MB page size

- Better TLB coverage of GPU memory

PASCAL UNIFIED MEMORY

Large datasets, simple programming, High Performance



Enable Large Data Models	Oversubscribe GPU memory Allocate up to system memory size
Tune Unified Memory Performance	Usage hints via cudaMemAdvise API Explicit prefetching API
Simpler Data Access	CPU/GPU Data coherence Unified memory atomic operations

CUDA 8 UNIFIED MEMORY — EXAMPLE

Allocating 4x more than P100 physical memory

```
void foo() {

  // Allocate 64 GB
  char *data;
  size_t size = 64*1024*1024*1024;
  cudaMallocManaged(&data, size);
}
```

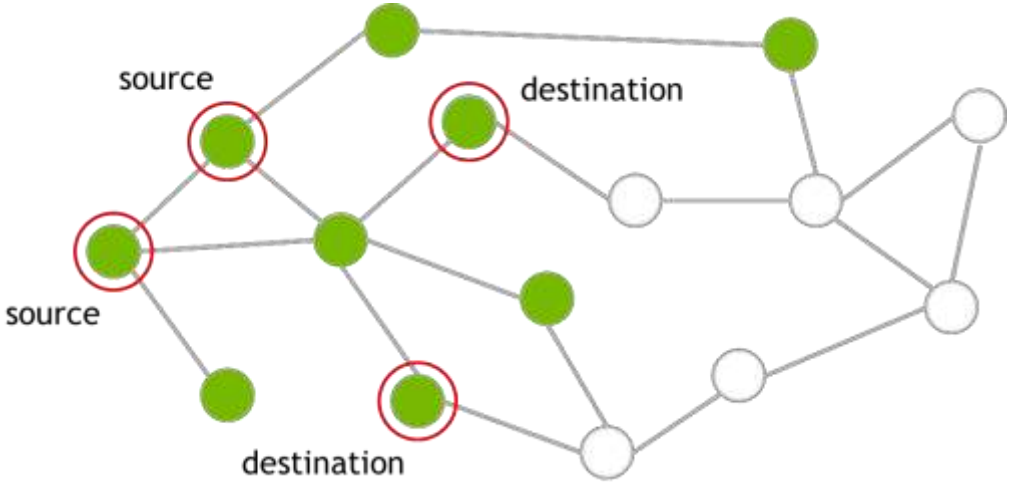
64 GB unified memory allocation on P100 with 16 GB physical memory

Transparent – No API changes

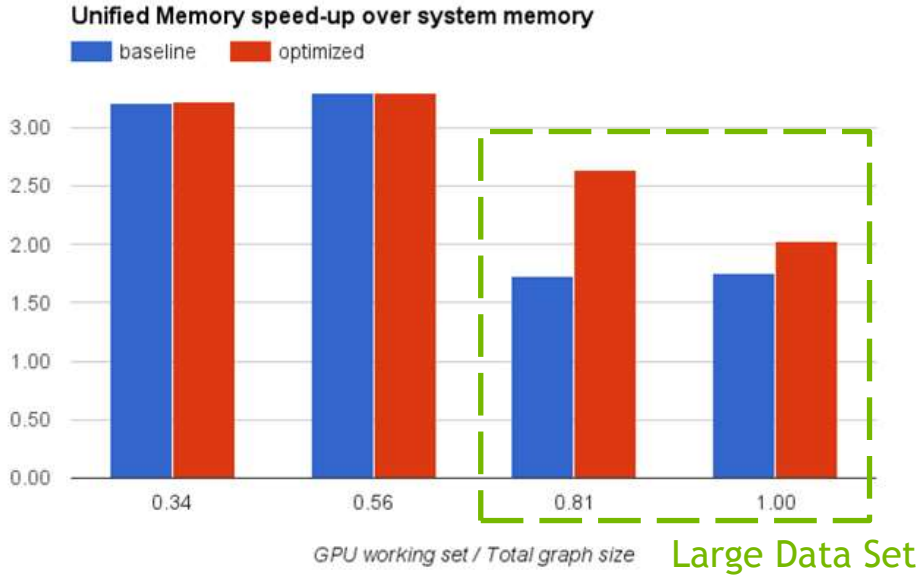
Works on Pascal & future architectures

USE CASE: ON-DEMAND PAGING

Graph Algorithms

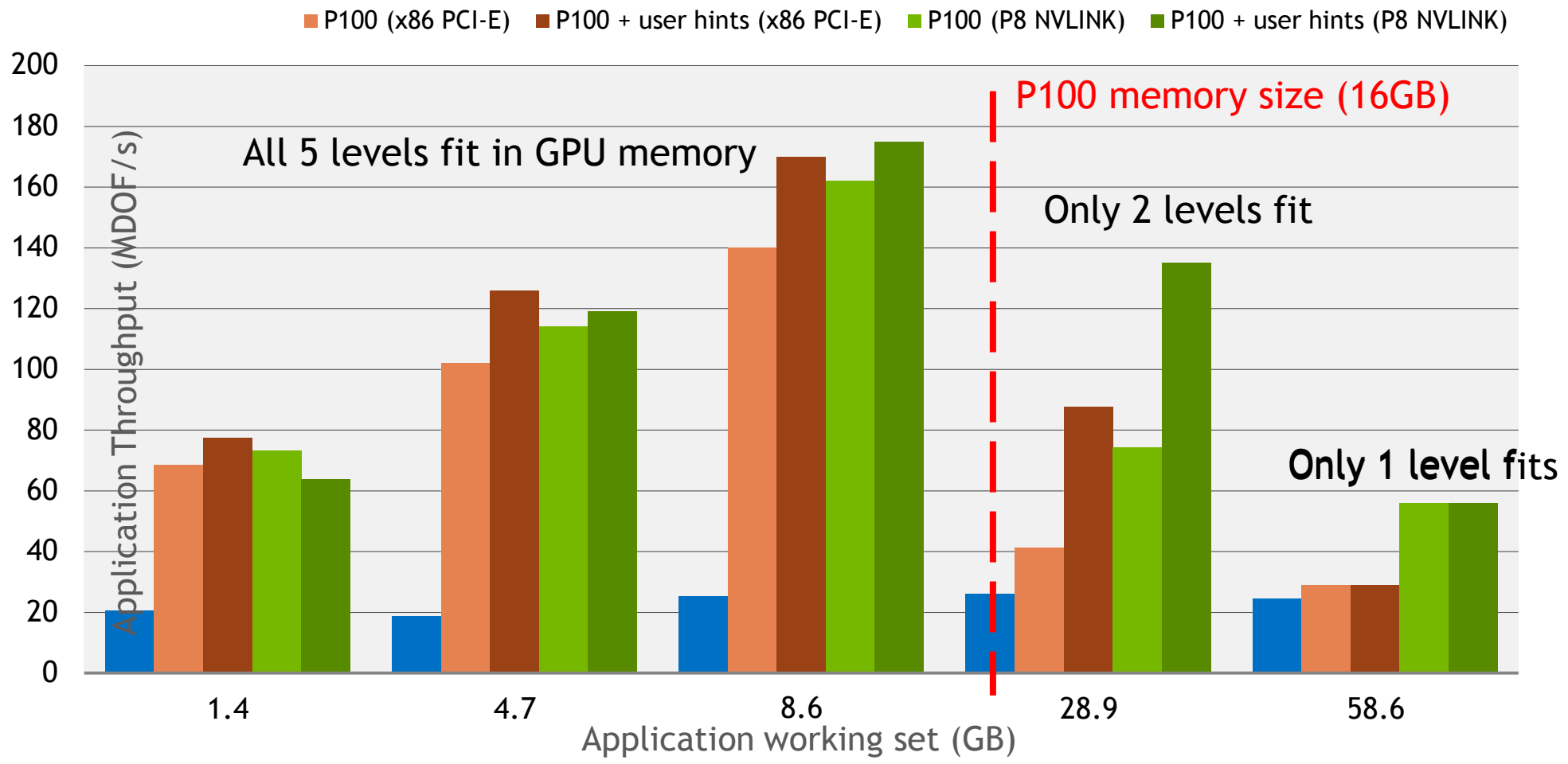


Performance over GPU directly accessing host memory (zero-copy)



Baseline: migrate on first touch
Optimized: best placement in memory

OUT-OF-CORE AMR COMPUTATIONS WITH UNIFIED MEMORY ON P100



x86 CPU: Intel E5-2630 v3, 2 sockets of 10 cores each with HT on (40 threads)

GREAT PERFORMANCE WITH UNIFIED MEMORY

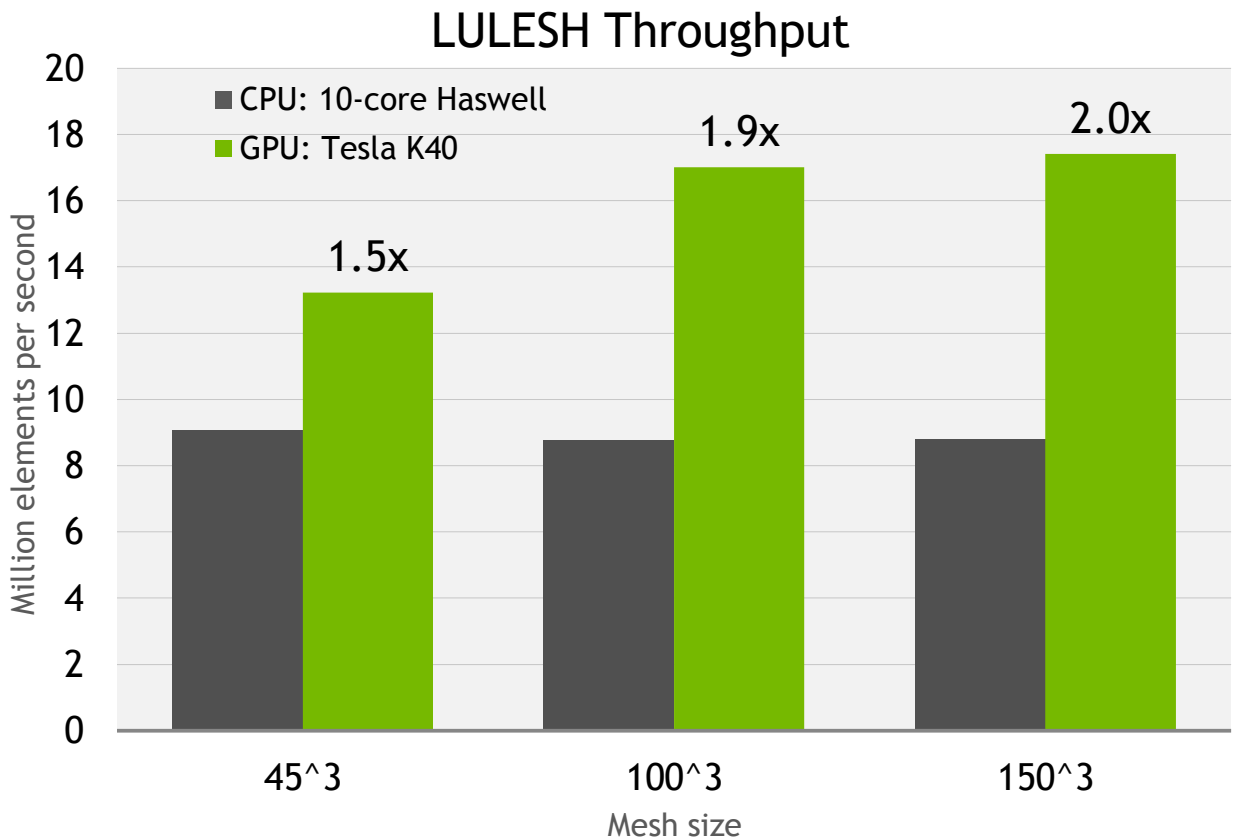
RAJA: Portable C++ Framework for parallel-for style programming

RAJA uses Unified Memory for heterogeneous array allocations

Parallel forall loops run on device

"Excellent performance considering this is a "generic" version of LULESH with no architecture-specific tuning."

-Jeff Keasler, LLNL



GPU: NVIDIA Tesla K40, CPU: Intel Haswell E5-2650 v3 @ 2.30GHz, single socket 10-core

HOW UNIFIED MEMORY WORKS ON PASCAL

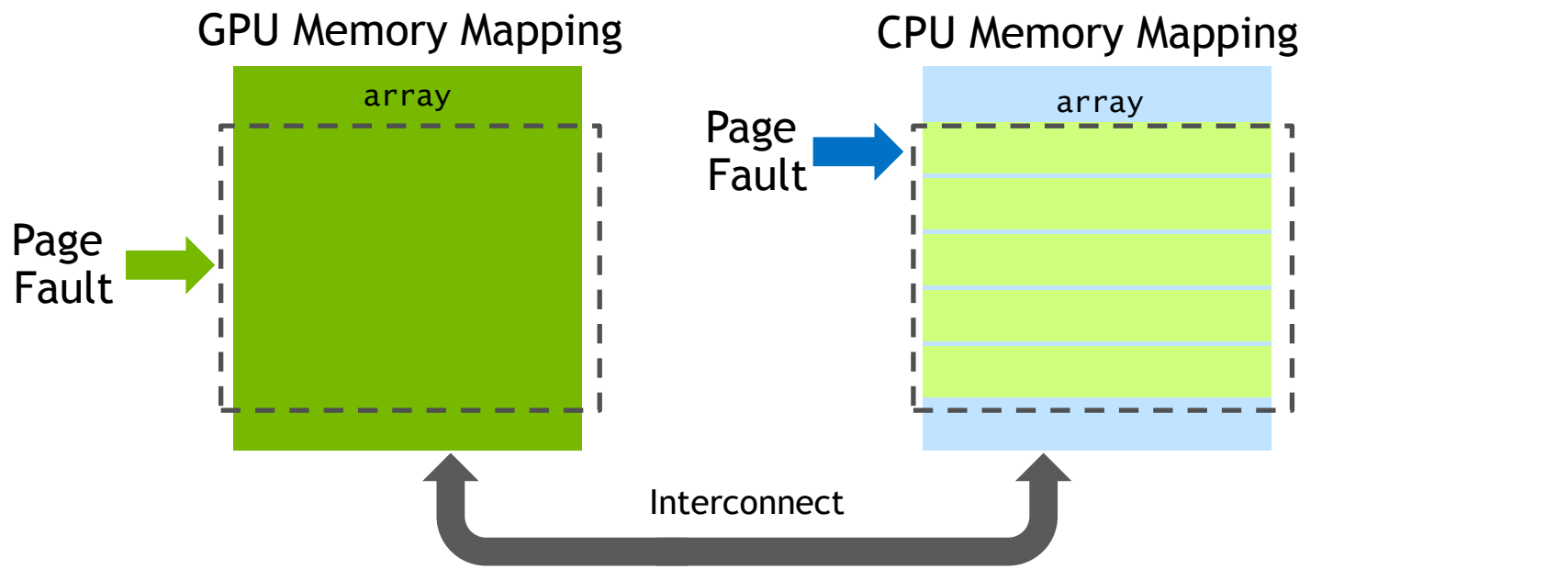
Servicing CPU *and* GPU Page Faults

GPU Code

```
__global__
Void setValue(char *ptr, int index, char val)
{
  ptr[index] = val;
}
```

CPU Code

```
cudaMallocManaged(&array, size);

memset(array, size);

setValue<<<...>>>(array, size/2, 5);
```



OPENACC AND CUDA UNIFIED MEMORY

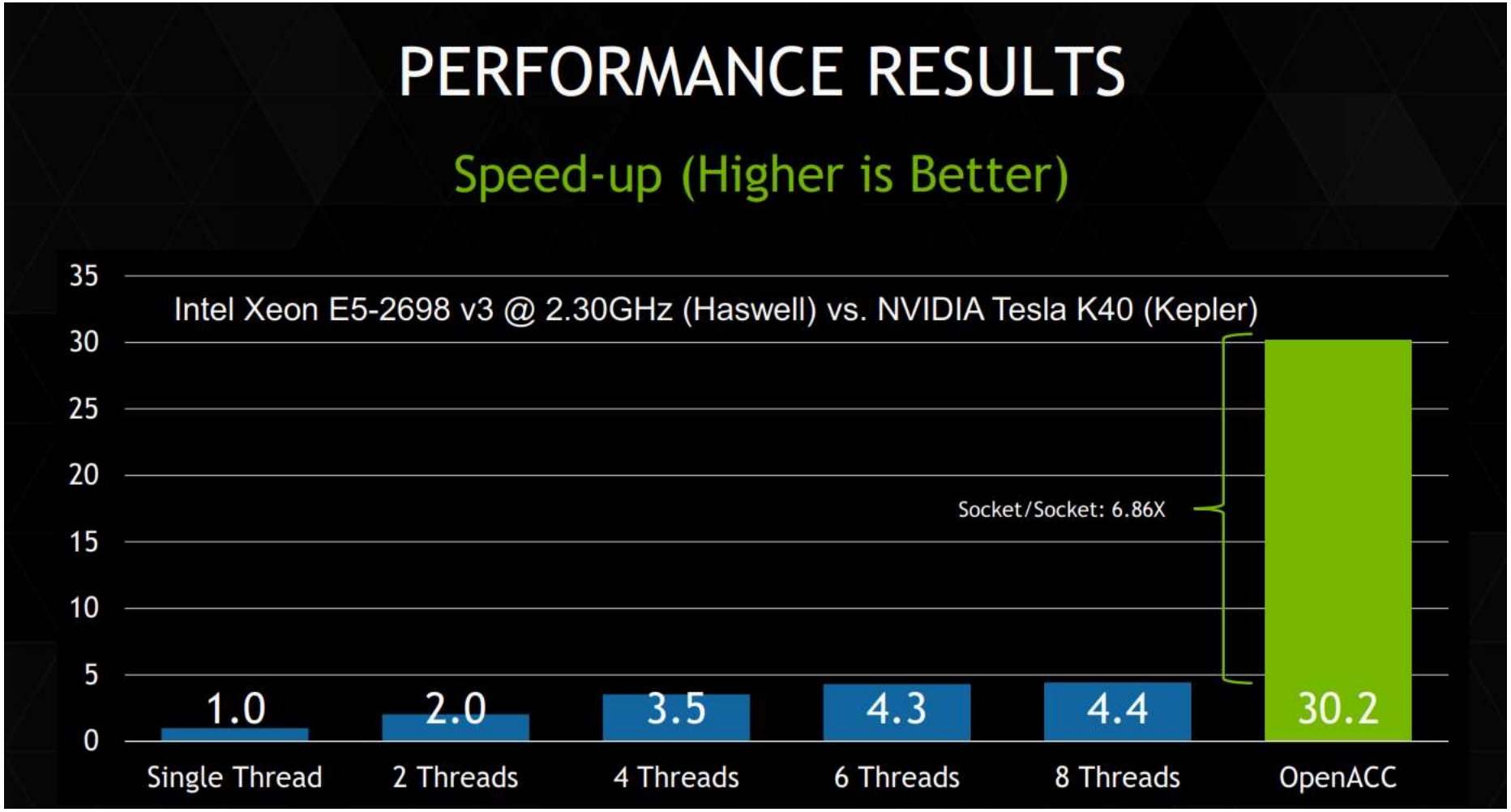
INDEPENDENT CLAUSE

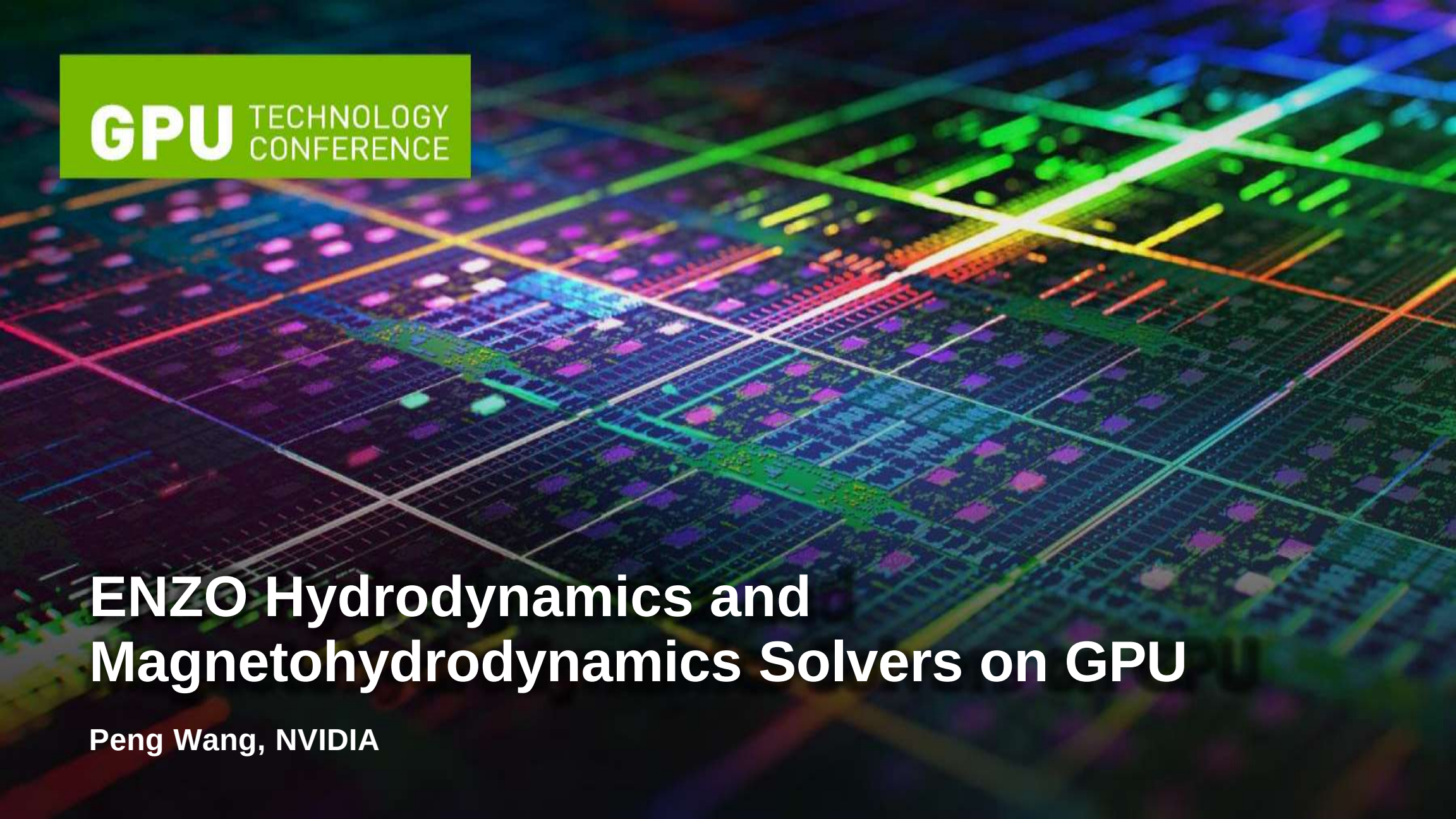
```
while ( err > tol && iter < iter_max ) {
  err=0.0;

#pragma acc kernels
{
  #pragma acc loop independent
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {

      Anew[j*m+i] = 0.25 * (A[j*m+i+1] + A[j*m+i-1] +
                            A[(j-1)*m+i] + A[(j+1)*m+i]);

      err = max(err, abs(Anew[j*m+i] - A[j*m+i]));
    }
  }

  #pragma acc loop independent
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j*m+i] = Anew[j*m+i];
    }
  }
}

  iter++;
}
```

Tell compiler that it's safe to parallelize

Tell compiler that it's safe to parallelize

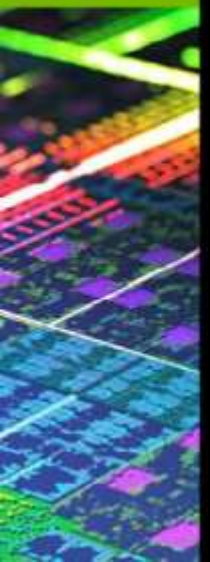
OPENACC AND CUDA UNIFIED MEMORY





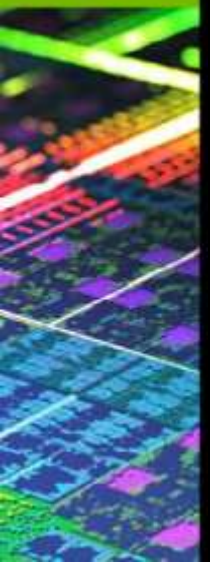
ENZO Hydrodynamics and Magnetohydrodynamics Solvers on GPU

Peng Wang, NVIDIA



Overview

- Introduction to ENZO
- How to port ENZO to GPU
- Performance and analysis

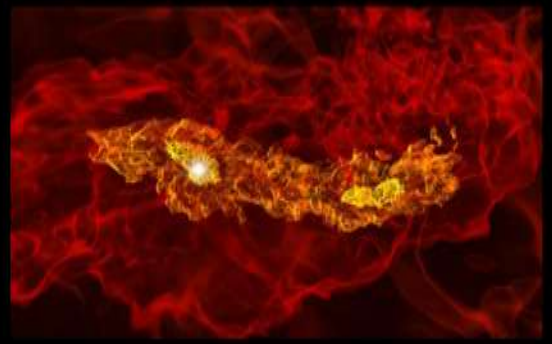


The ENZO Multi-Physics Code

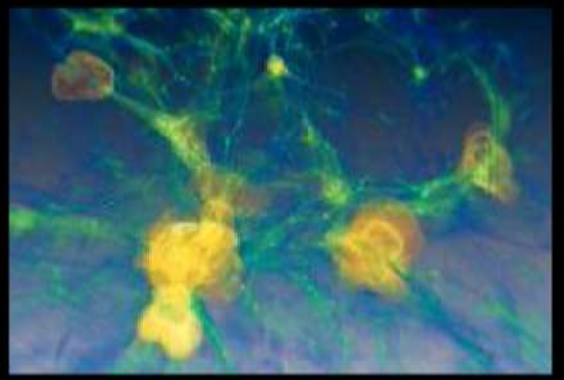
- Block-structured AMR
- Finite volume hydrodynamics and magnetohydrodynamics solver
- Multigrid Poisson solver
- Particle-mesh N-body solver
- Particle-fluid interaction: particle formation, accretion, feedback
- Radiative transfer using adaptive ray-tracing
- Chemistry network
- MPI-parallelized

http://code.google.com/p/enzo

ENZO Applications



First stars

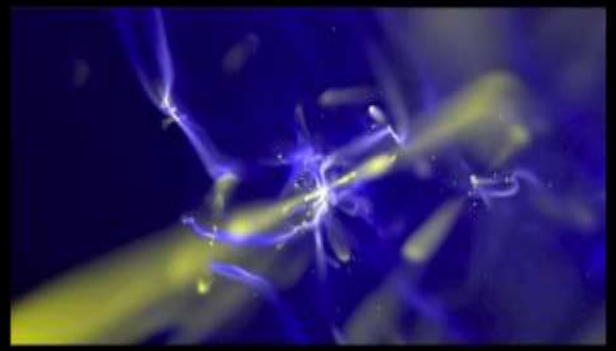


Galaxies

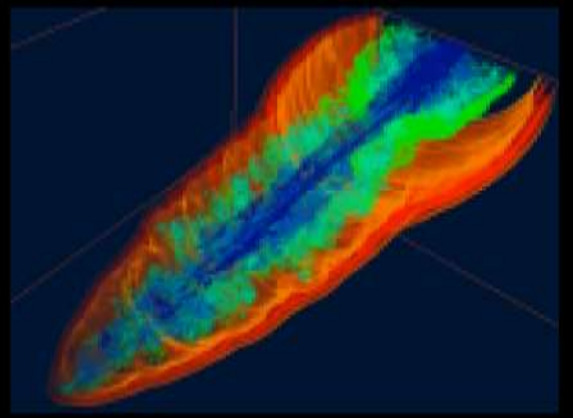
Black Hole

HII Region

Star formation

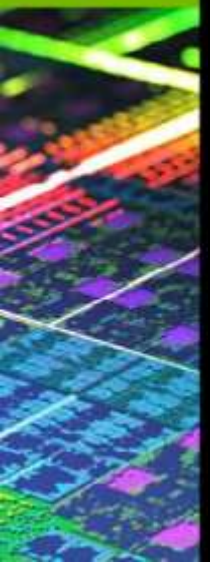


Relativistic Jets



ENZO GPU Porting

- Collaborators: Tom Abel, Ralf Kaehler (Stanford)
- First phase project finished in 2009
  - HLL-PLM hydro solver
  - New Astronomy 15 (2010) 581-589
- Second phase project started in 2012
  - PPM and MHD solvers
- Focus on MHD solver in this talk as
  - PPM solver's implementation is similar
  - MHD solver is a "newer" feature in ENZO

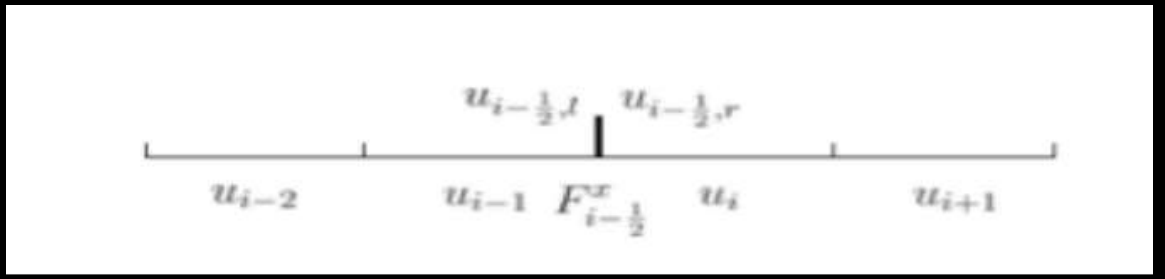


Supported MHD Feature on GPU

- HLL-PLM solver
- Chemistry
  - MultiSpecies=1,2,3
- External sources
  - Gravity
  - Driving force
- Comoving coordinates

ENZO MHD Solver

- Finite volume solver
  - Riemann solver: HLL
  - Reconstruction: PLM
- Solver algorithm
  - Input: field values at cell centers
  - Output: flux at cell interfaces
  - Highly parallel: each flux depending on the neighboring 4 inputs
    - $Flux_{i-1/2}=F(u_{i-2},u_{i-1},u_i,u_{i+1})$



Why Focusing on Solver

- One of the most time consuming parts
  - DrivenTurbulence Problem: ~90% of the total time
- So we want to speedup it up using GPU!

	1 CPU core	16 CPU cores
Total Time	**1328.7**	**87.4**
Solver Time	**1277.8**	**78**

Problem:
- Grid: 256^3
- Run for 10 time steps

Benchmarking system:
- Cray XK7
- AMD 16 core Opteron 6270
- NVIDIA K20X GPU
- Cray Linux Environment
- GNU compilers (OPT=aggressive)

GPU Solver

- Basically the same between CPU and GPU solver
- Still room for further speedup.
  - This simple port gives enough speedup that solver is no longer the bottleneck
  - Single source for CPU/GPU solver

ENZO CPU MHD solver code

```
for (int i=0; i < NCell; i++)
  flux(i) = HLL_PLM(u(i-2),u(i-1),u(i),u(i+1));
```

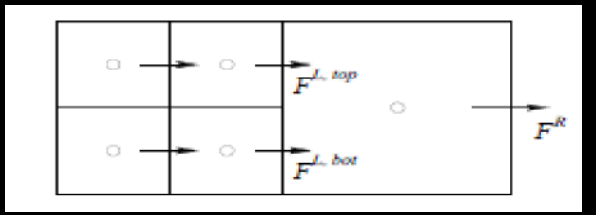
ENZO GPU MHD solver code

```
int i=blockIdx.x*blockDim.x + threadIdx.x;

flux(i) = HLL_PLM(u(i-2),u(i-1),u(i),u(i+1));
```

```
#pragma kernels
for (int i=0; i < NCell; i++)
  flux(i) = HLL_PLM(u(i-2),u(i-1),u(i),u(i+1));
```

Integrating GPU Solver with AMR

- Each grid is an independent initial-boundary value problem
  - — Just call GPU solver on each grid seperately
- Flux correction: ensure consistency between grids
  - — Use fine grids fluxes to update coarse grid fluxes



Flux Correction

- Problem: fluxes now calculated on GPU
- Two possible solutions:
  - Transfer the whole flux back to CPU and call CPU flux correction routine
    - Very simple to implement but wastes a lot of data transfer time: small fraction of flux is actually needed
  - Collect needed flux on GPU and transfer only them to CPU
    - What we implemented

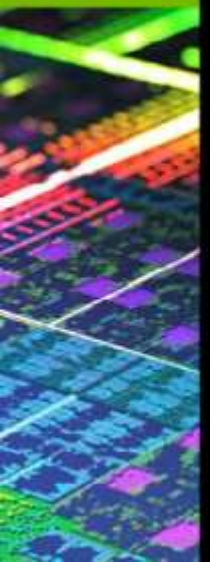
ENZO GPU MHD Solver

```
for each level do:
  for each grid in this level do:
    call MHD solver on this grid:
      transfer fields to GPU call
      GPU HLL-PLM MHD solver update
      chemistry field on GPU update
      source terms on GPU flux
      correction on GPU
      transfer updated fields to CPU
    call other modules on CPU(chemistry, gravity, etc)
```

Performance

- 1core + 1GPU vs 16 cores
  - — Solver: 7.2x
  - — Overall: 1.4x
- Amdahl's law
  - — 16 core: non-solver 9.4 sec
  - — 1 core+1GPU: non-solver 57.8 sec
- Solution: launch multiple MPI processes per GPU

#MPI=#cores	16 CPU cores	1 CPU core+1GPU	Speedup
Total Time	87.4	61.6	1.4
Solver Time	78	10.8	7.2



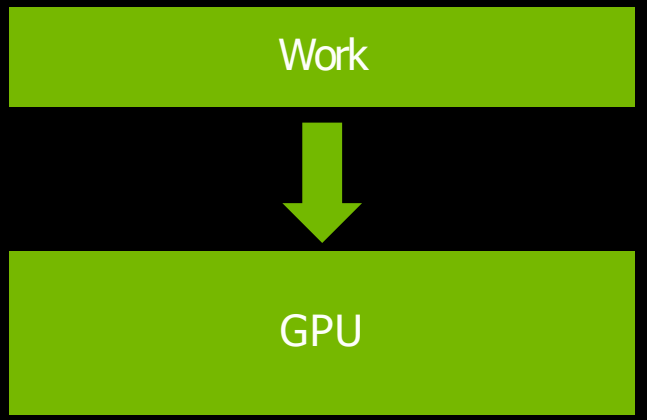
GPU Profiling

- Data transfer ~50% of the total GPU time
- Overlapping data transfer with kernel computation?
  - — CUDA stream
  - — Many independent grids in AMR
  - — Modifying high level AMR code structure

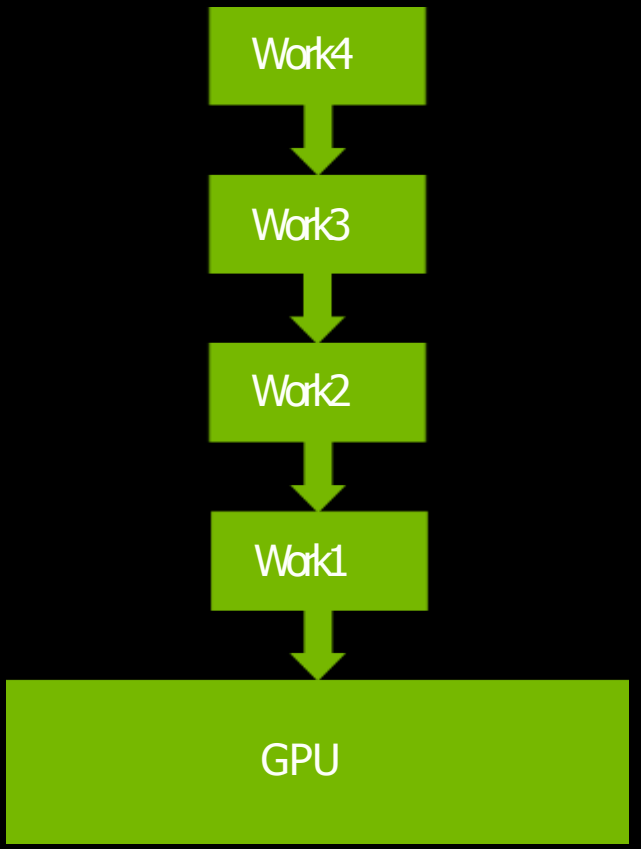
Multi MPI Processes Per GPU Before Kepler

- Serialized use of the GPU
  - Potentially under-utilization

1 MPI per GPU

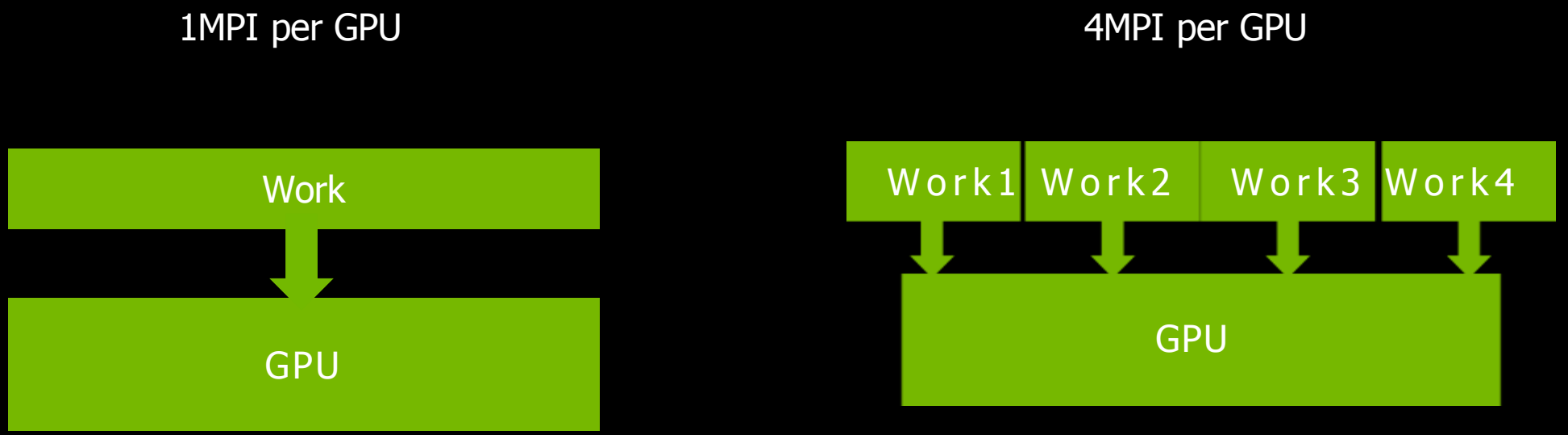


4MPI per GPU



Simultaneous Multiprocess (MPS)

- Simultaneous processes on Kepler+ GPU
- Automatic overlap between data transfer and kernel computation



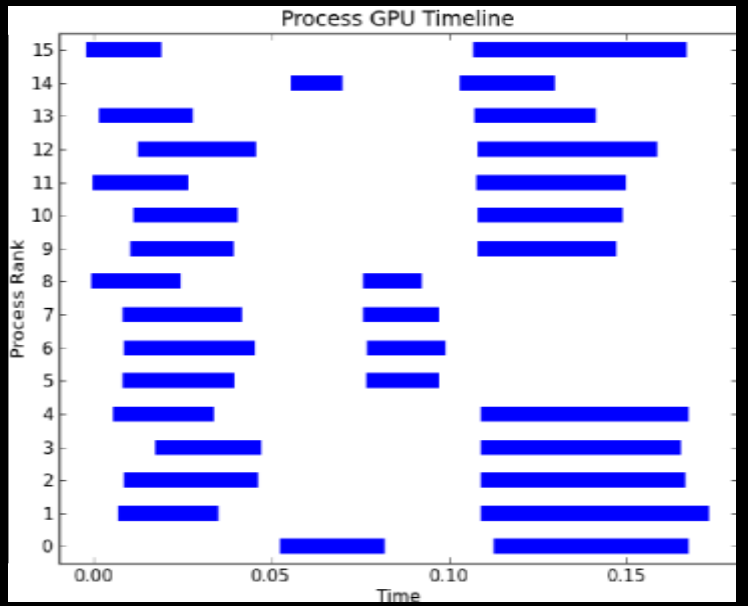
Final Performance

- 16 CPU cores + 1 GPU (proxy on) vs 16 CPU cores
  - Overall: 6x
  - Solver: 16x

#MPI=#cores	16 CPU cores	1 CPU core +1GPU	16 CPU cores+ 1GPU
Total Time	87.4	61.6	14.3
Solver Time	78	10.8	4.7

Proxy does Work!

Rank



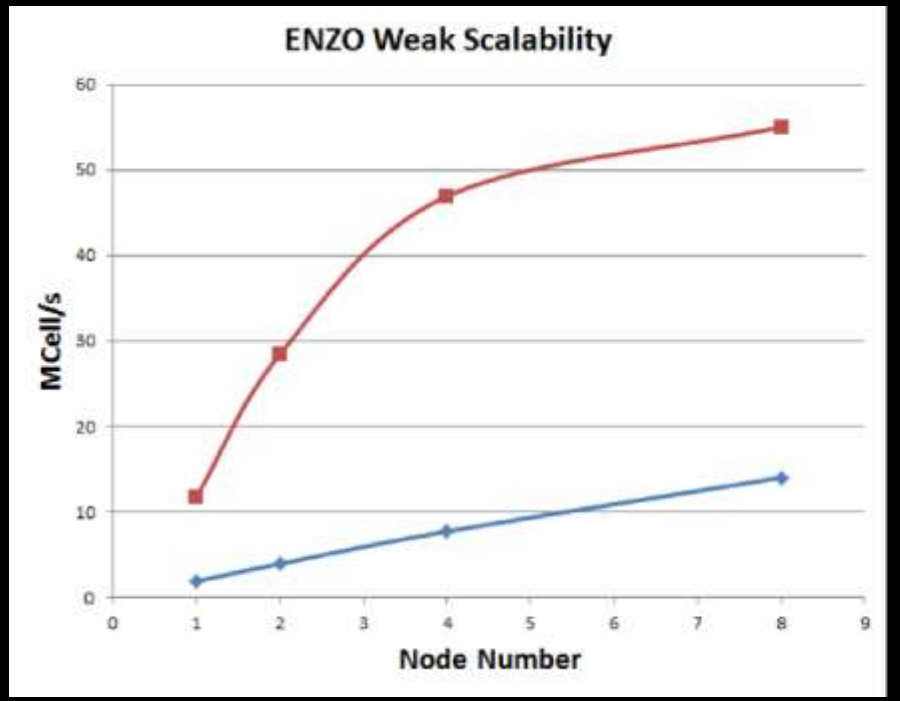
Time

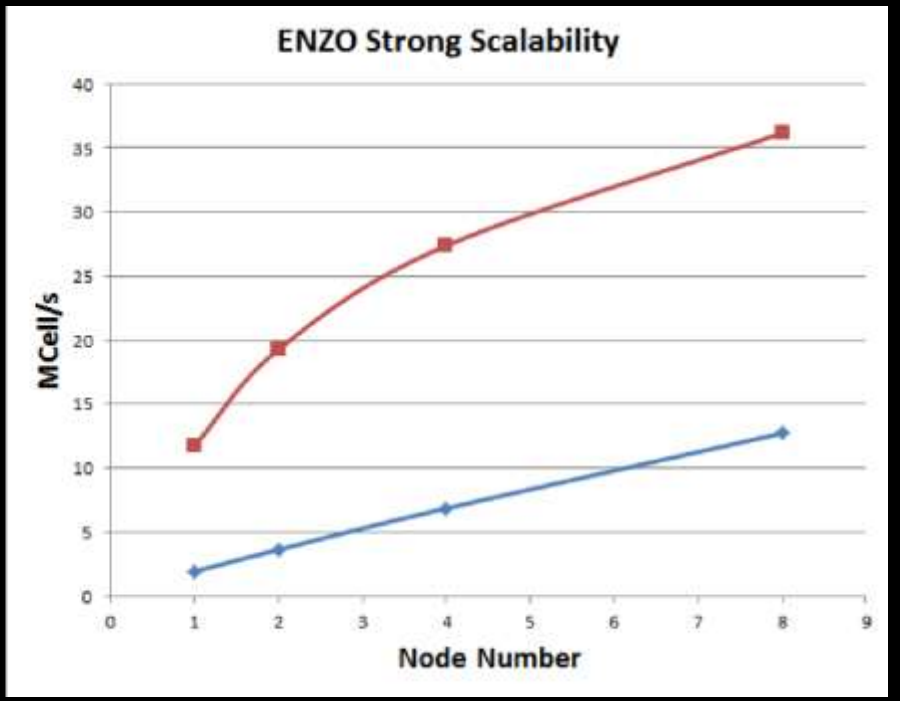
Proxy does work!

- Ubuntu PC with 4 core Nehalem and K20c
- Proxy performance advantage:
  - 30% in the solver
  - 10% overall

#MPI=#cores	1 CPU core +1GPU	4 CPU cores+ 1GPU, proxy off	4 CPU cores+ 1GPU, proxy on
Total Time	51.8	24.2	21.7
Solver Time	10.64	8.6	5.9

Scalability





GPU

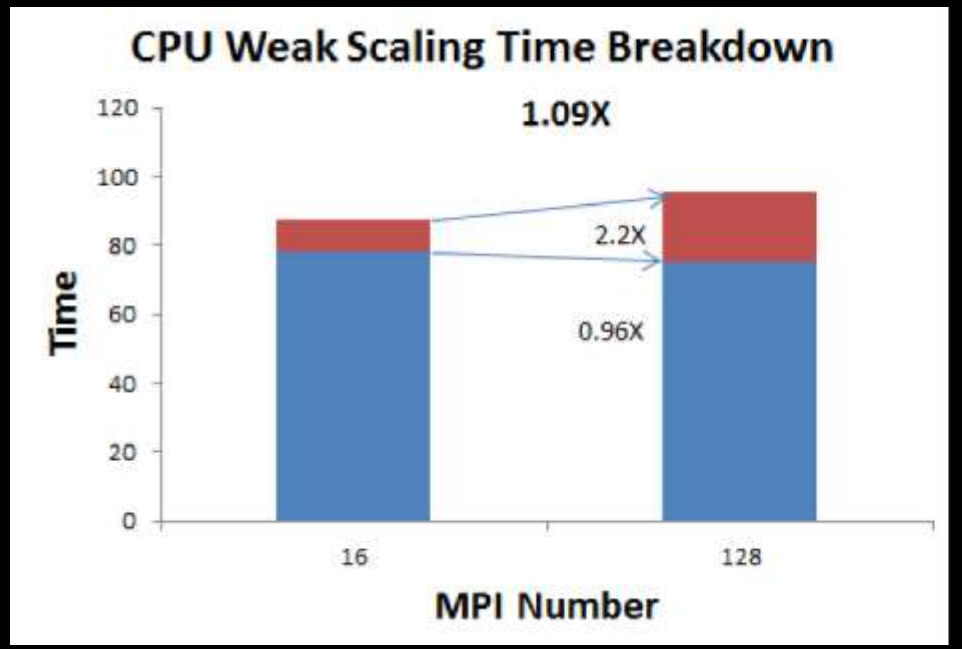
CPU

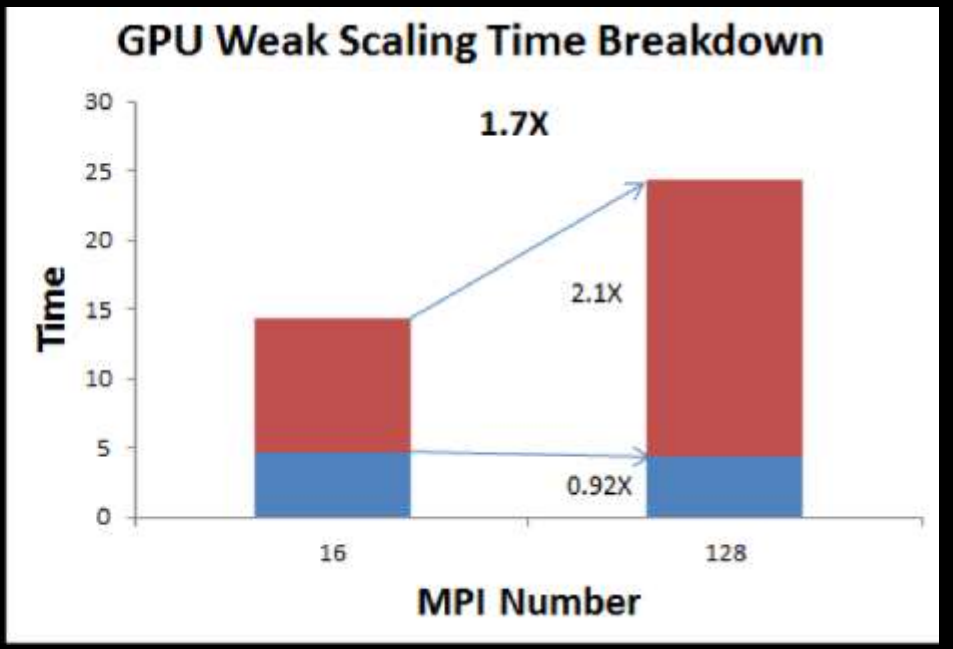
Cray XK6
1 Node: 1 16-core CPU+1 K20X

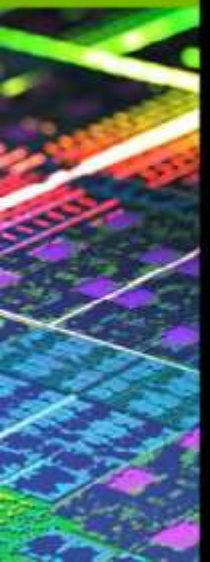
Mcell/s=#cell/time_per_step

What's Limiting GPU's scalability?

- Amdahl's law: non-solver part now the dominant factor in GPU version and it doesn't scale Non-Solver
  - — Mostly MPI Solver



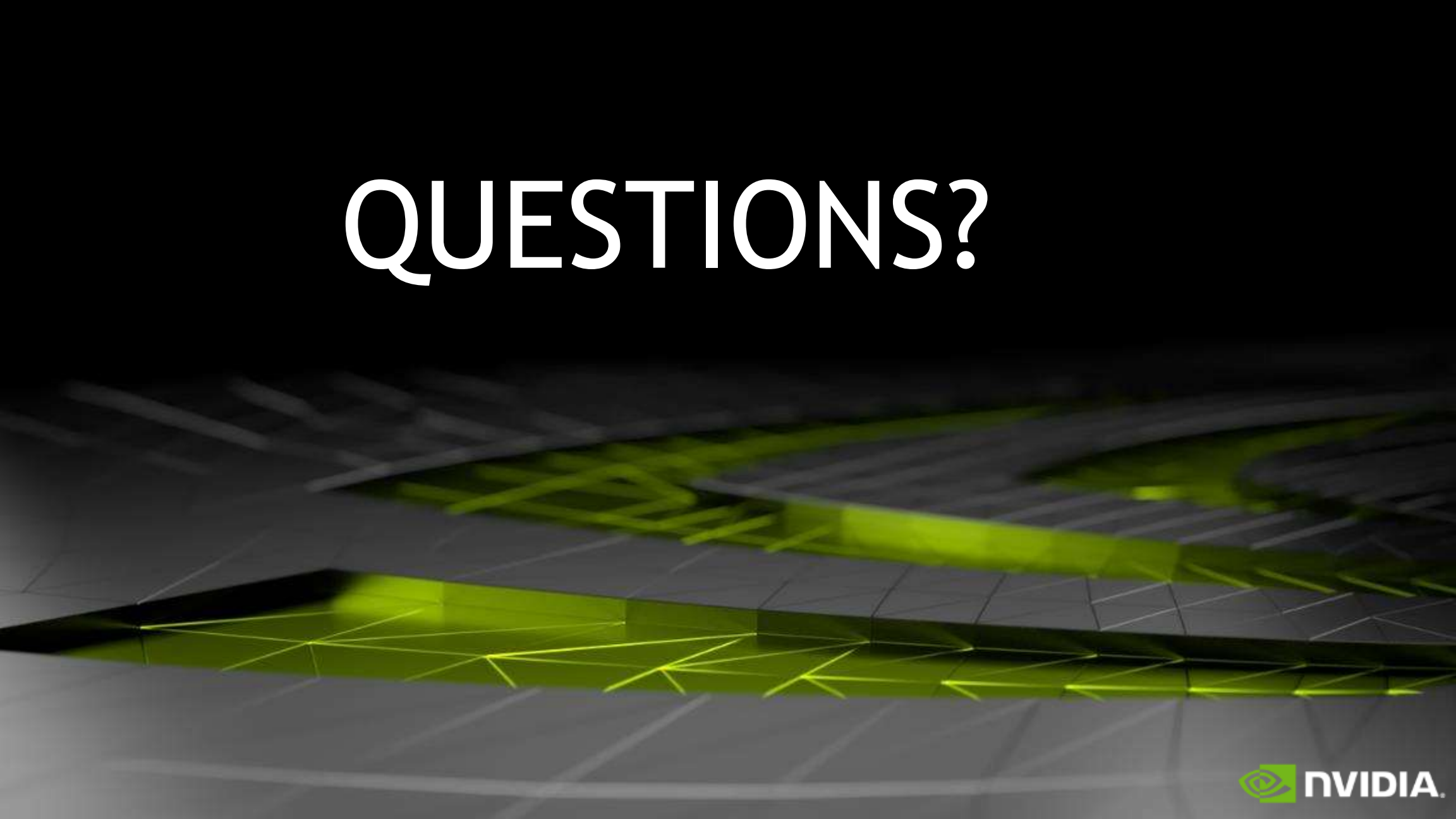


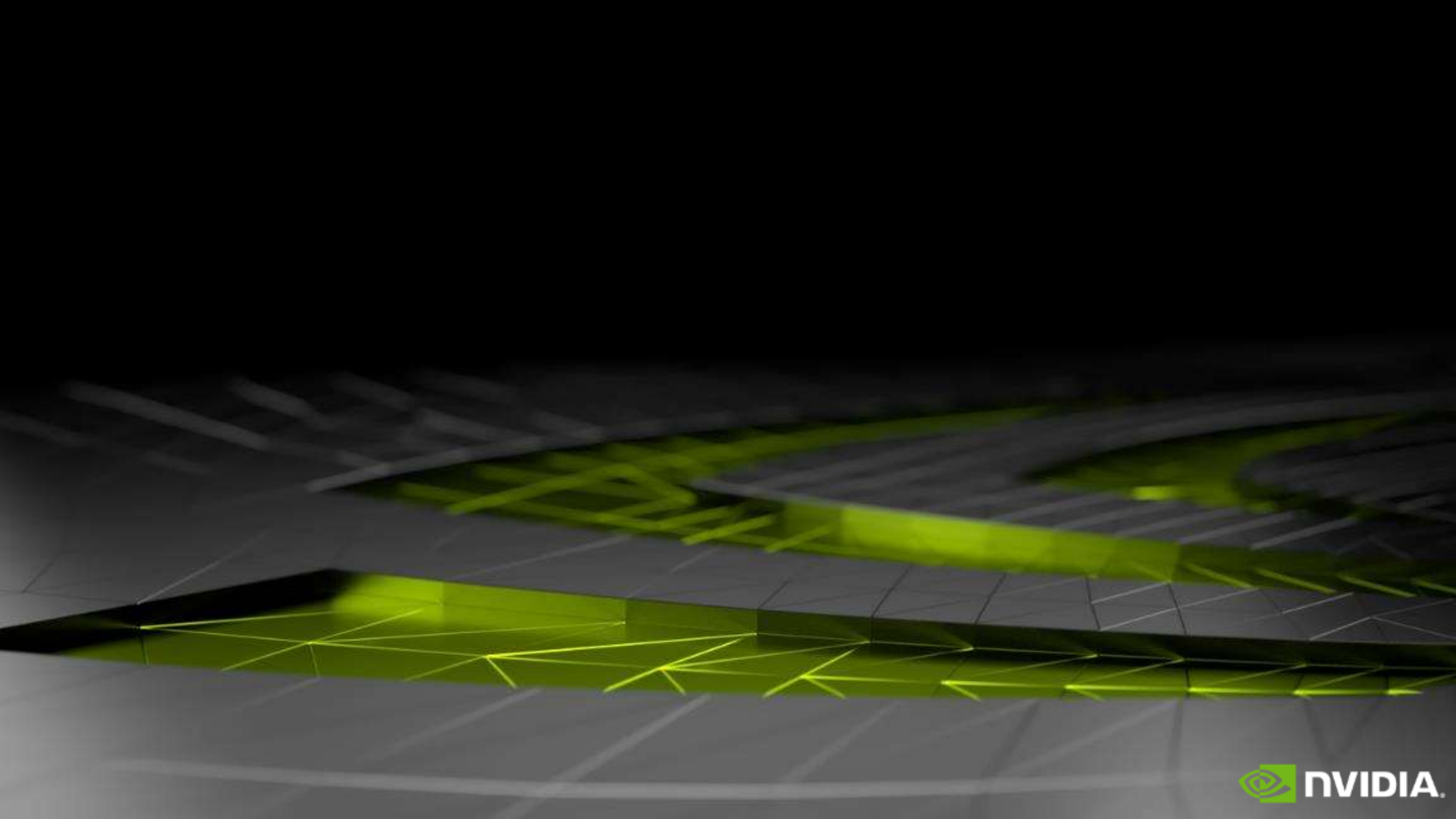


Summary

- Easy to integrate GPU into block-structured AMR code
  - Similar to integrating OpenMP with MPI
- GPU gives good speedup on AMR solvers
- MPS proxy makes it easy to achieve high performance for AMR codes

QUESTIONS?




NVIDIA.