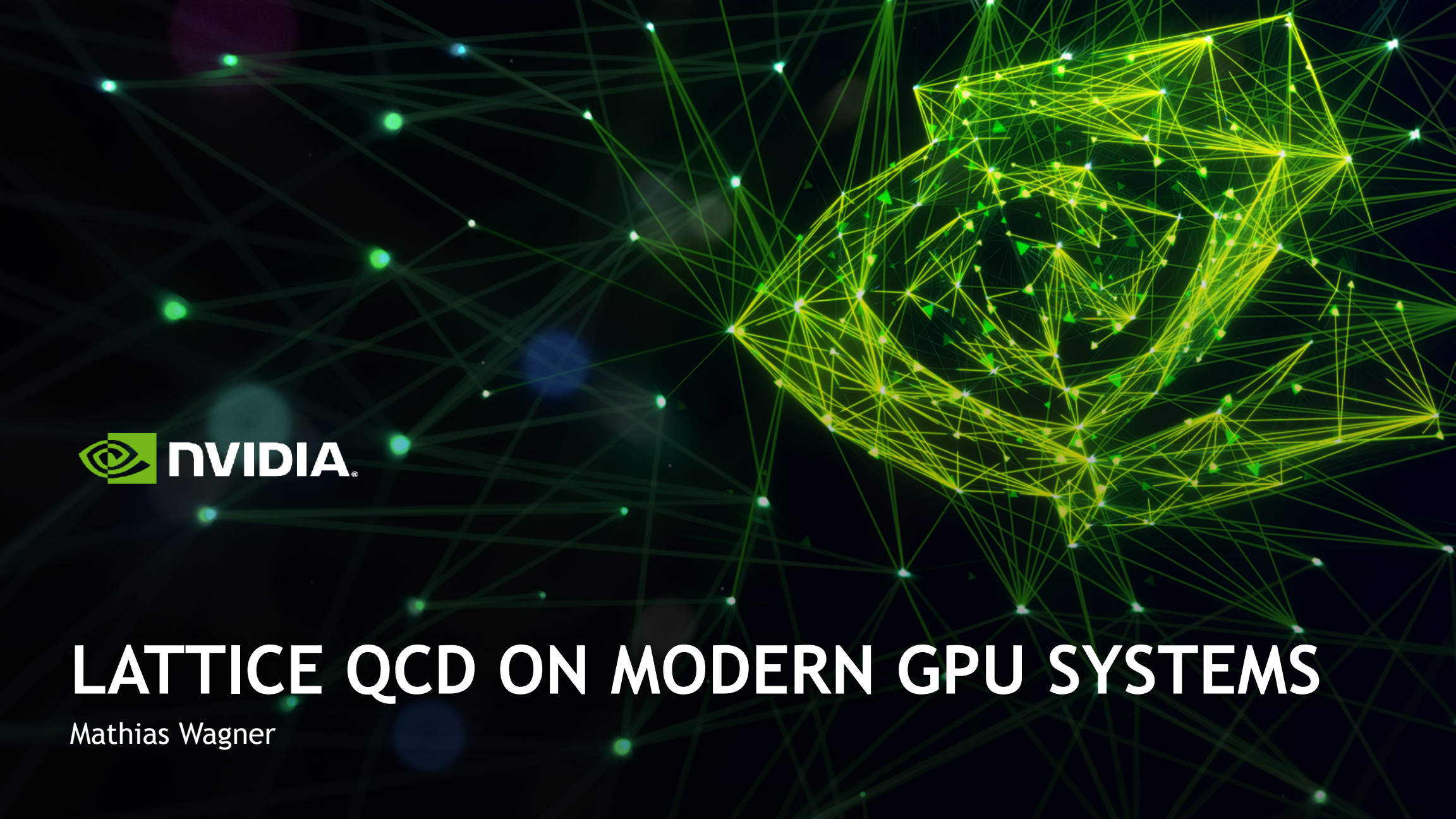


# LATTICE QCD ON MODERN GPU SYSTEMS

Mathias Wagner

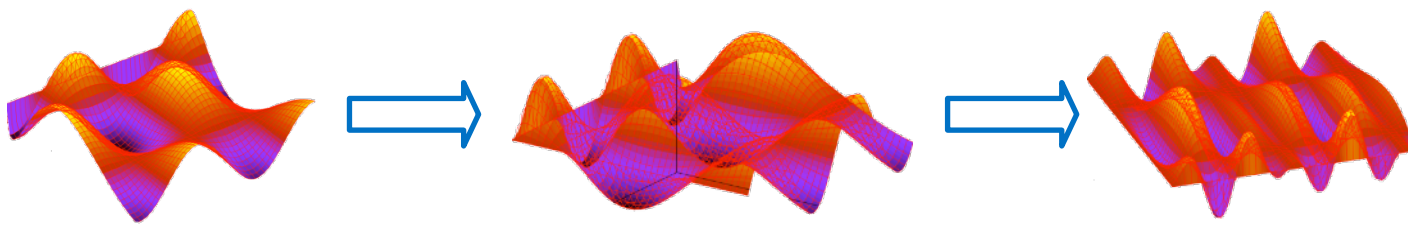
# STEPS IN AN LQCD CALCULATION

$$D_{ij}^{\alpha\beta}(x,y;U)\psi_j^{\beta}(y) = \eta_i^{\alpha}(x)$$

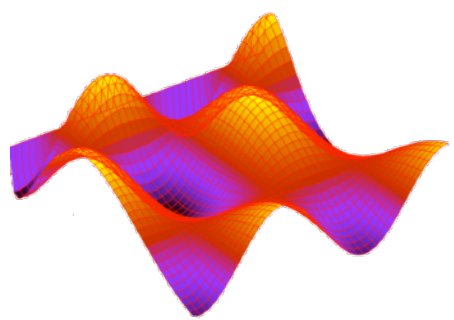
or Ax = b

1. Generate an ensemble of gluon field configurations “gauge generation”
   - Produced in sequence, with hundreds needed per ensemble
   - Strong scaling required with 100-1000 TFLOPS sustained for several months
   - 50-90% of the runtime is in the linear solver
   - O(1) solve per linear system
   - Target $16^4$ per GPU

Simulation Cost ~ $a^{-6}$ $V^{5/4}$



2. “Analyze” the configurations
   - Can be farmed out, assuming ~10 TFLOPS per job
   - Task parallelism means that clusters reign supreme here
   - 80-99% of the runtime is in the linear solver
     Many solves per system, e.g., O($10^6$)
   - Target $24^4$-$32^4$ per GPU



# MAPPING THE DIRAC OPERATOR TO CUDA

Finite difference operator in LQCD is known as Dslash

Assign a single space-time point to each thread

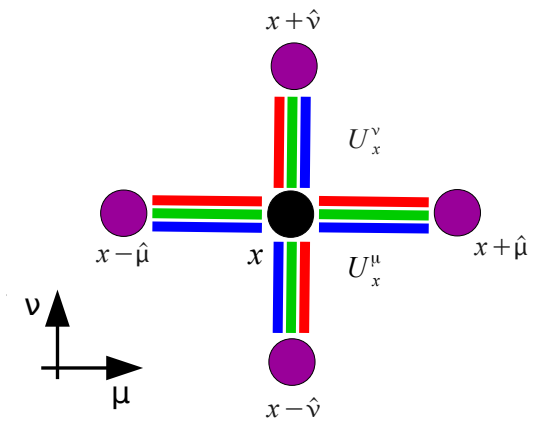
- V = XYZT threads, e.g., V = $24^4$ => $3.3 \times 10^6$ threads

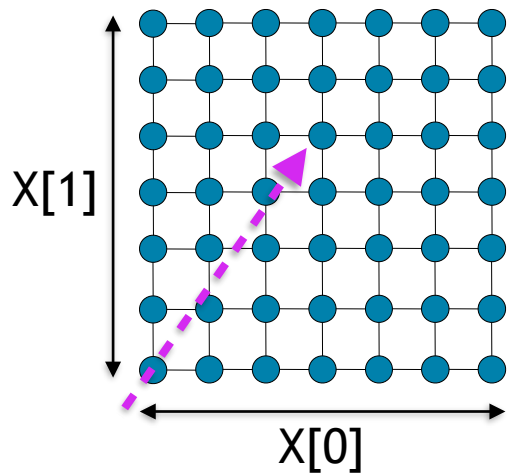
Looping over direction each thread must

- Load the neighboring spinor (24 numbers x8)
- Load the color matrix connecting the sites (18 numbers x8)
- Do the computation
- Save the result (24 numbers)

Each thread has (Wilson Dslash) 0.92 naive arithmetic intensity

$$D_{x,x'} =$$



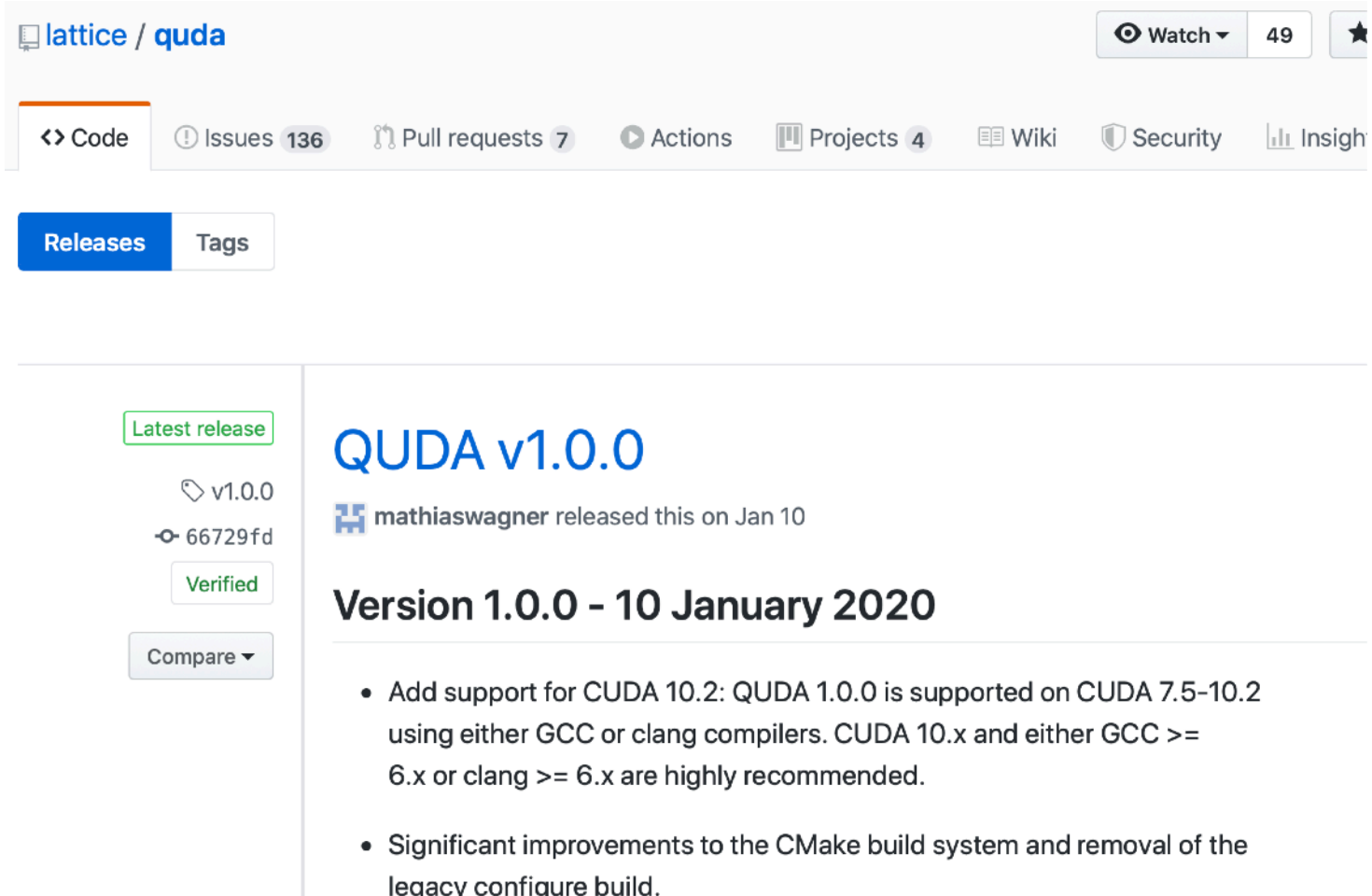




# QUDA

- Effort started at Boston University in 2008, now in wide use as the GPU backend for BQCD, Chroma, CPS, MILC, TIFR, tmLQCD, etc.
- Provides:
  - Various solvers for all major fermionic discretizations, with multi-GPU support
  - Additional performance-critical routines needed for gauge-field generation
- Maximize performance
  - Exploit physical symmetries to minimize memory traffic
  - Mixed-precision methods
  - Autotuning for high performance on all CUDA-capable architectures
  - Eigenvector and deflated solvers (Lanczos, EigCG, GMRES-DR)
  - Multigrid solvers for optimal convergence Multi-source solvers
  - Domain-decomposed (Schwarz) preconditioners for strong scaling
  - Strong-scaling improvements
- A research tool for how to reach the exascale

# QUDA - LATTICE QCD ON GPUS

http://lattice.github.com/quda, BSD license



# QUDA CONTRIBUTORS

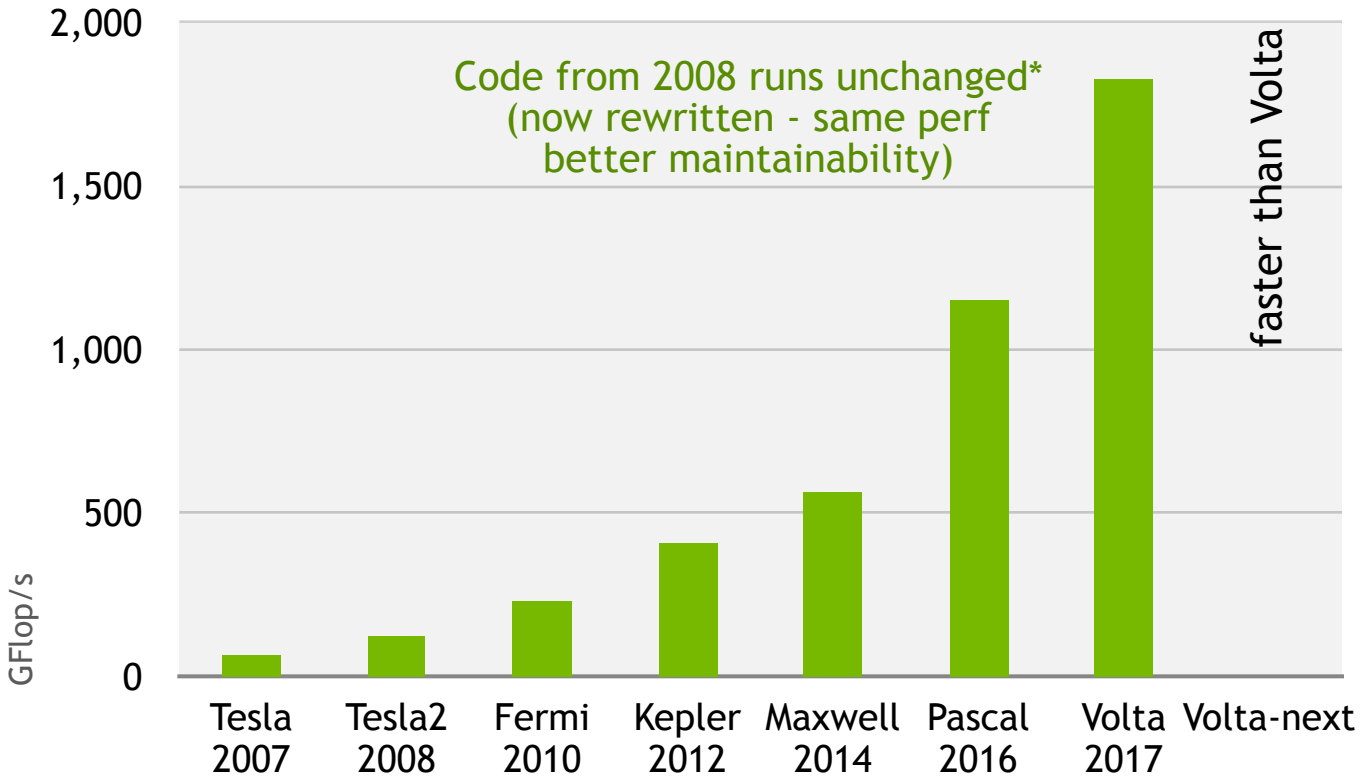
## 10+ years - lots of contributors

Ron Babich (NVIDIA)
Simone Bacchio (Cyprus)
Michael Baldhauf (Regensburg)
Kip Barros (LANL)
Rich Brower (Boston University)
Nuno Cardoso (NCSA)
Kate Clark (NVIDIA)
Michael Cheng (Boston University)
Carleton DeTar (Utah University)
Justin Foley (Utah -> NIH)
Joel Giedt (Rensselaer Polytechnic Institute)
Arjun Gambhir (William and Mary)
Steve Gottlieb (Indiana University)
Kyriakos Hadjiyiannakou (Cyprus)
Dean Howarth (LLNL)
Bálint Joó (Jlab)
Hyung-Jin Kim (BNL -> Samsung)
Bartek Kostrzewa (Bonn)
Claudio Rebbi (Boston University)
Hauke Sandmeyer (Bielefeld)
Guochun Shi (NCSA -> Google)
Mario Schröck (INFN)
Alexei Strelchenko (FNAL)
Jiqun Tu (Columbia -> NVIDIA)
Alejandro Vaquero (Utah University)
Mathias Wagner (NVIDIA)
Evan Weinberg (NVIDIA)
Frank Winter (Jlab)

# AUTOTUNING


# RECOMPILE AND RUN

## Autotuning provides performance portability



# QUDA'S AUTOTUNER

## ensuring optimal kernel performance

virtual C++ class "`Tunable`" that is derived for each kernel you want to autotune

By default `Tunable` classes will autotune 1-d block size, shared memory size, grid size

- Derived specializations do 2-d and 3-d block size tuning

Tuned parameters are stored in a `std::map` and dumped to disk for later reuse

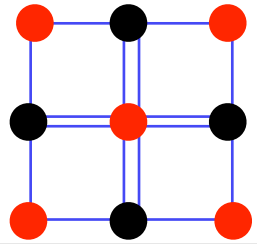
Built in performance metrics and profiling

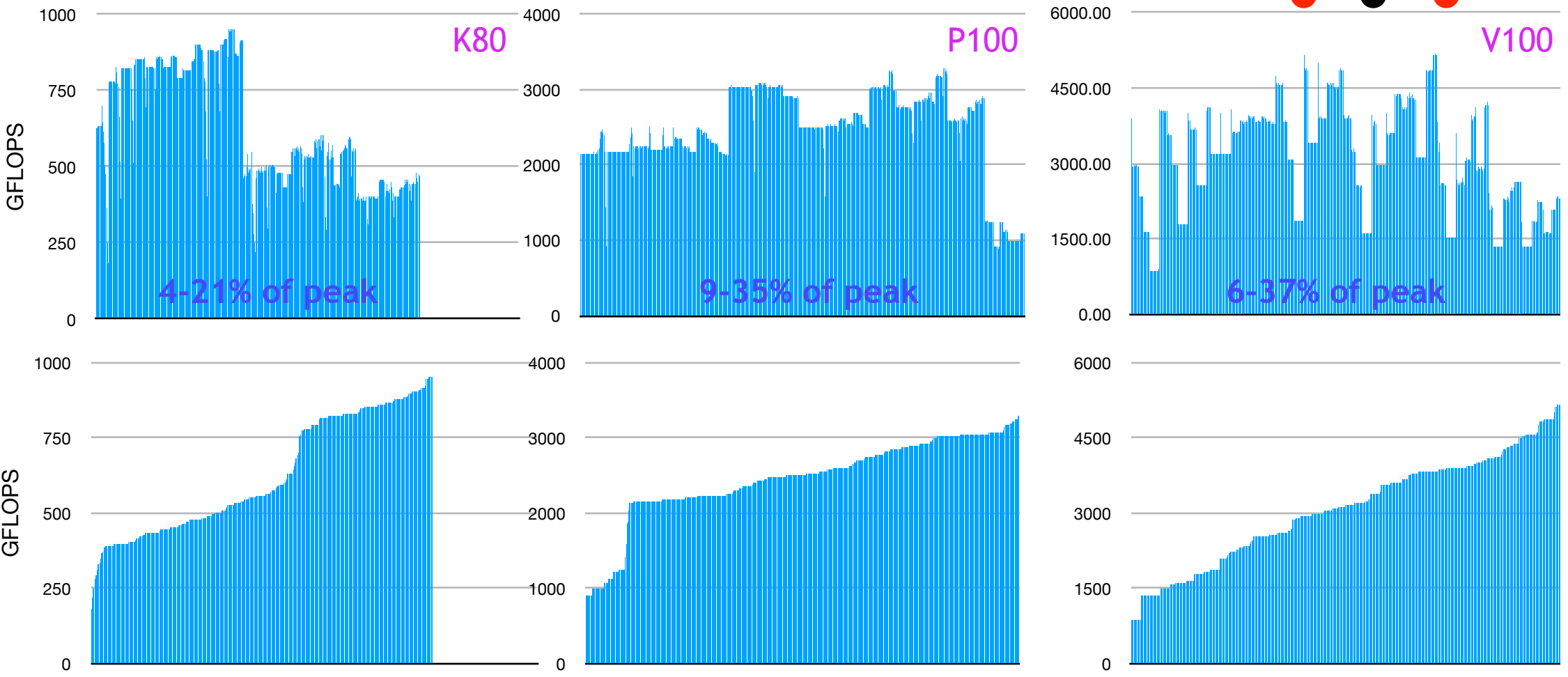
User just needs to

- State resource requirements: shared memory per thread and/or per block, total number of threads
- Specify a tuneKey which gives each kernel a unique entry and break any degeneracy

# GENERATIONAL COMPARISON

## $F_{\mu v}$ kernel - batched 3x3 multiplication





# MIXED PRECISION

# LINEAR SOLVERS

$$
\begin{aligned}
&\text{while } (|\mathbf{r}_k| > \varepsilon)\ \{\\
&\quad \beta_k = (\mathbf{r}_k, \mathbf{r}_k)/(\mathbf{r}_{k-1}, \mathbf{r}_{k-1})\\
&\quad \mathbf{p}_{k+1} = \mathbf{r}_k - \beta_k \mathbf{p}_k\\
&\quad \mathbf{q}_{k+1} = A\,\mathbf{p}_{k+1}\\
&\quad \alpha = (\mathbf{r}_k, \mathbf{r}_k)/(\mathbf{p}_{k+1}, \mathbf{q}_{k+1})\\
&\quad \mathbf{r}_{k+1} = \mathbf{r}_k - \alpha \mathbf{q}_{k+1}\\
&\quad \mathbf{x}_{k+1} = \mathbf{x}_k + \alpha \mathbf{p}_{k+1}\\
&\quad k = k+1\\
&\}
\end{aligned}
$$

conjugate gradient

QUDA supports a wide range of linear solvers

- CG, BiCGstab, GCR, Multi-shift solvers, etc.

Condition number inversely proportional to mass

- Light (realistic) masses are highly singular
- Naive Krylov solvers suffer from critical slowing down at decreasing mass

Entire solver algorithm must run on GPUs

- Time-critical kernel is the stencil application
- Also require BLAS level-1 type operations

# MIXED-PRECISION CG

Apply Dslash in sloppy precision (single, half)

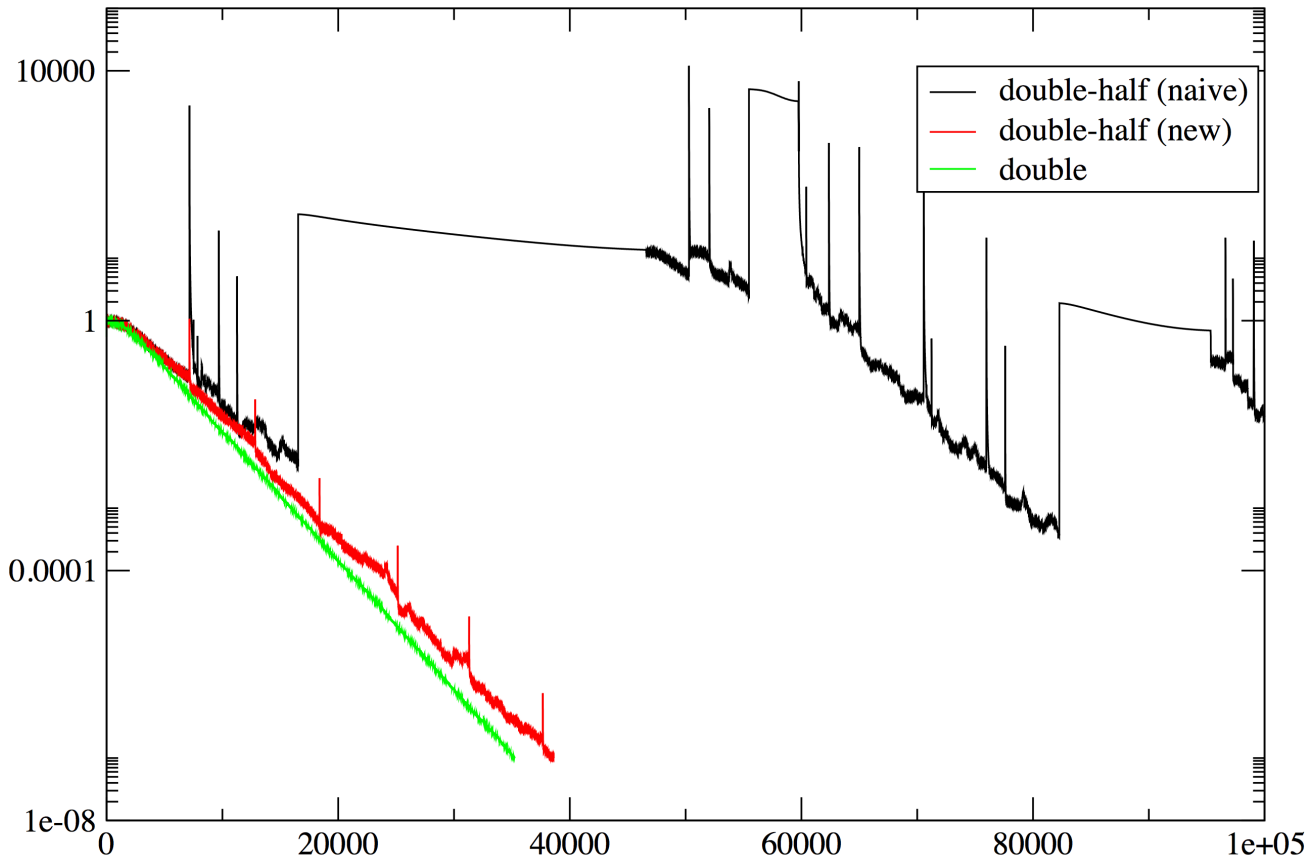
Reliable residual replacement in high precision

Ensures accuracy of final result

Half-precision **storage**:

- Stencil elements ∈ [-1,1] (Link):
  - 16-bit fixed point
- Grid elements (Spinor):
  - 16-bit fixed point (24 numbers)
  - float (exponent, 1 number)

Use fp32 for actual arithmetics



# MIXED-PRECISION CG

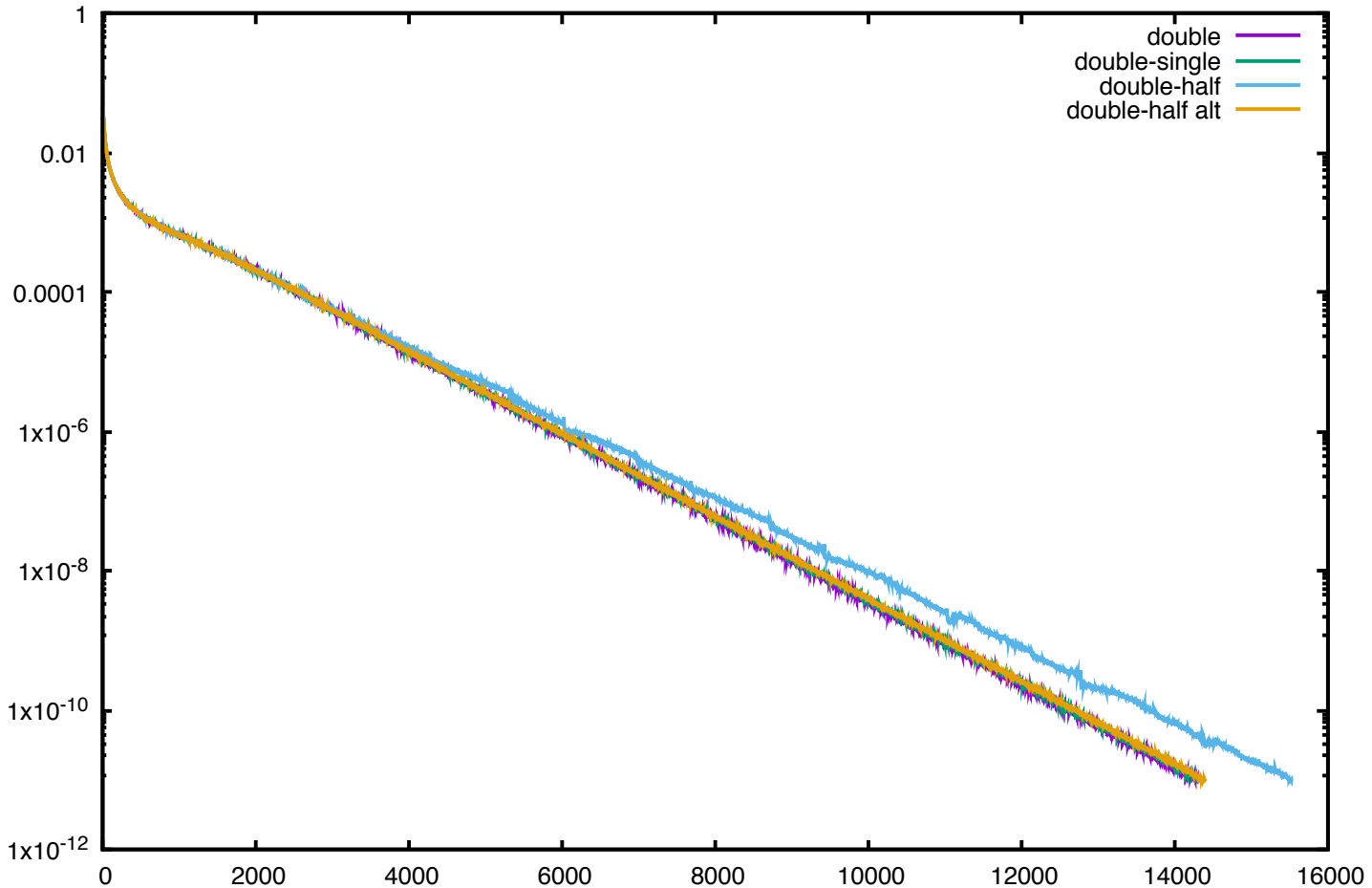
**double-half**

- Maintain solution vectors in high precision
  - Including the partial accumulator
- When true residual is injected, re-project the direction vector
- Use Polak-Ribière formula

$$\beta_k := \frac{\mathbf{z}_{k+1}^{\mathsf{T}} \left( \mathbf{r}_{k+1} - \mathbf{r}_k \right)}{\mathbf{z}_k^{\mathsf{T}} \mathbf{r}_k}$$

**double-half alt**

- Residual replacement strategy of van der Worst and Ye



# MIXED-PRECISION CG

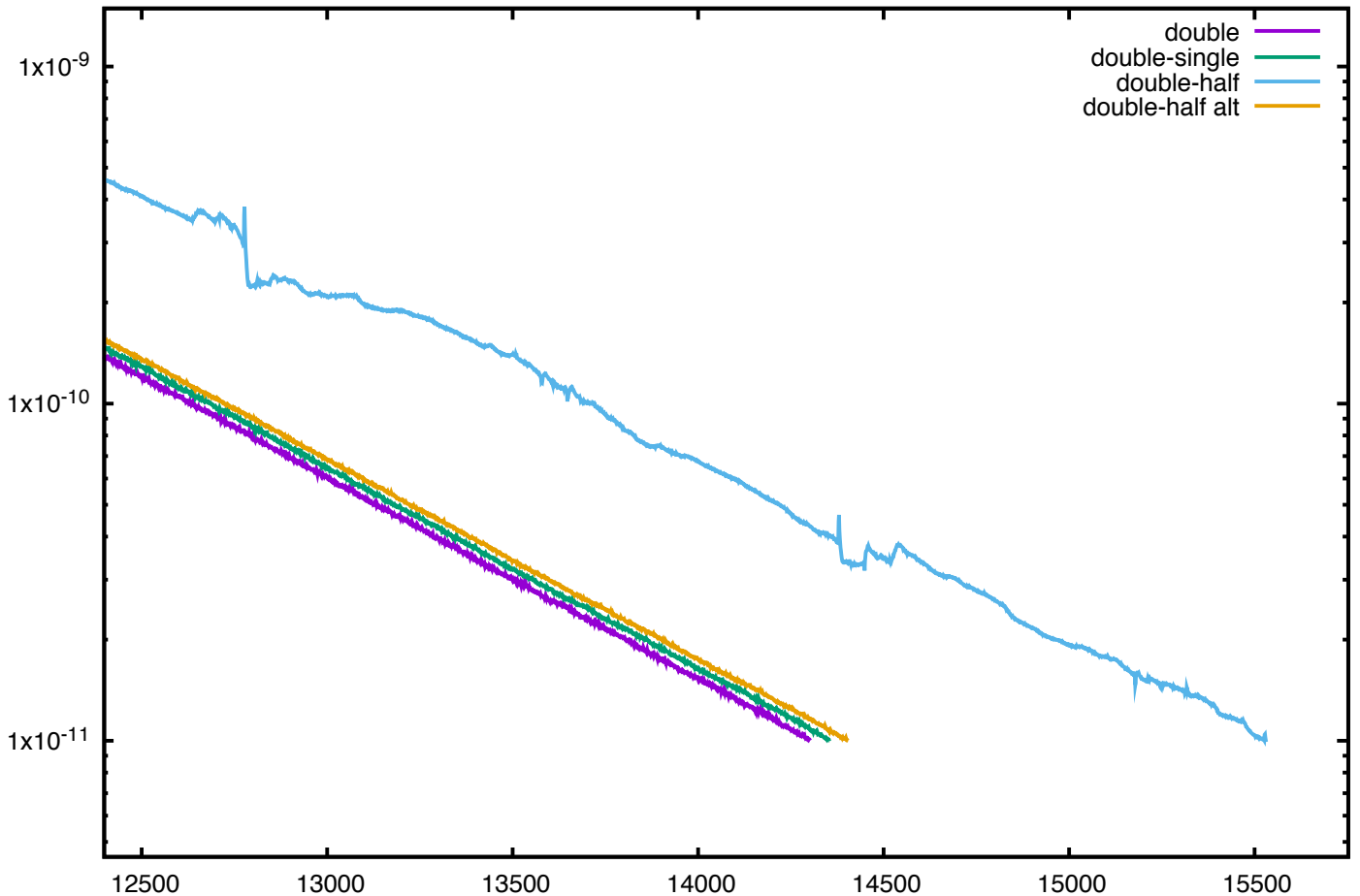
**double-half**

- Maintain solution vectors in high precision
  - Including the partial accumulator
- When true residual is injected, re-project the direction vector
- Use Polak-Ribière formula

$$\beta_k := \frac{\mathbf{z}_{k+1}^{\mathsf{T}} \left( \mathbf{r}_{k+1} - \mathbf{r}_k \right)}{\mathbf{z}_k^{\mathsf{T}} \mathbf{r}_k}$$

**double-half alt**

- Residual replacement strategy of van der Worst and Ye



# MIXED-PRECISION CG

**double-half**

- Maintain solution vectors in high precision
  - Including the partial accumulator

- When true residual is injected, re-project the direction vector

- Use Polak-Ribière formula

$$\beta_k := \frac{\mathbf{z}_{k+1}^{\mathsf{T}} \left( \mathbf{r}_{k+1} - \mathbf{r}_k \right)}{\mathbf{z}_k^{\mathsf{T}} \mathbf{r}_k}$$

**double-half alt**

- Residual replacement strategy of van der Worst and Ye

# MIXED-PRECISION CG

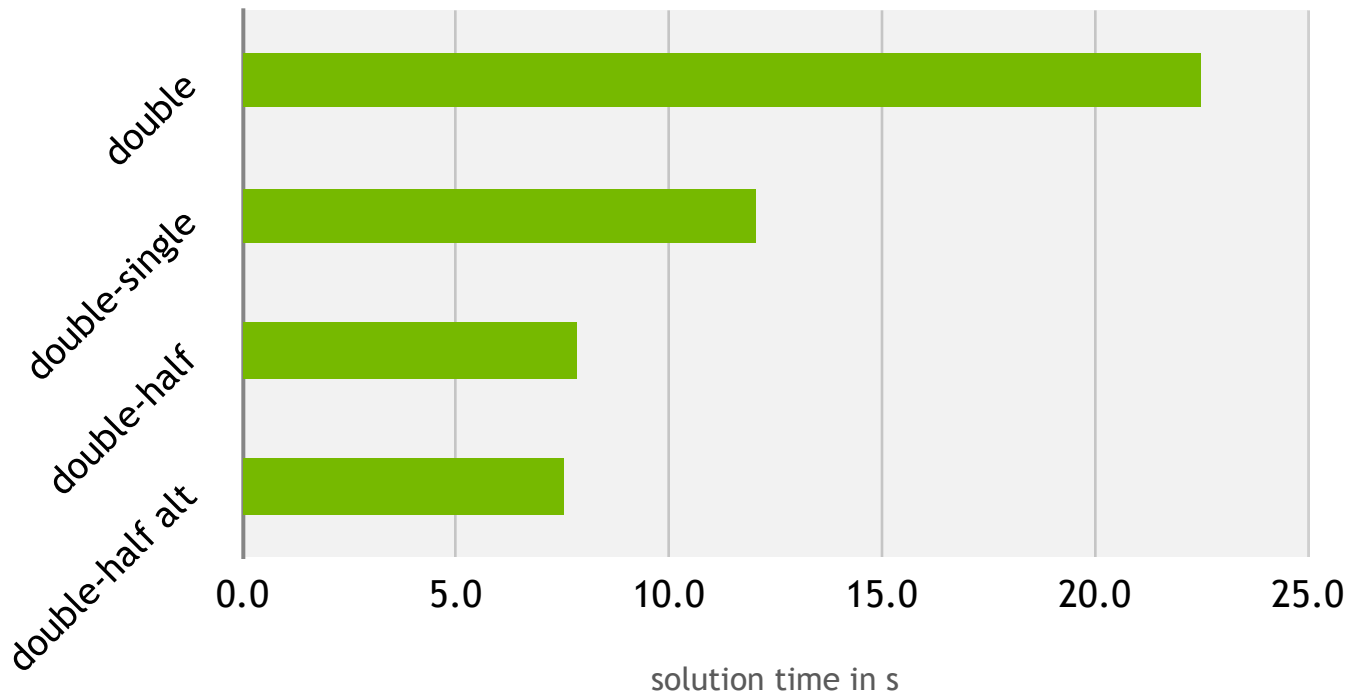
mixed precision:

apply Dslash in sloppy precision (single, half)

reliable residual replacement in high precision

ensures double-precision accuracy of final result

virtually identical iteration count



# EIGENSOLVERS

Multiple workflows require repeated solution with different RHS with the same matrix

Multigrid not amenable to all linear operators

Eigenvector deflation is a robust alternative applicable to all operators

- Deflate out low modes from linear operator to accelerate the solver
- Cost of eigensolver is amortized if we solve enough RHS
- Aside: also use deflation to accelerate multigrid

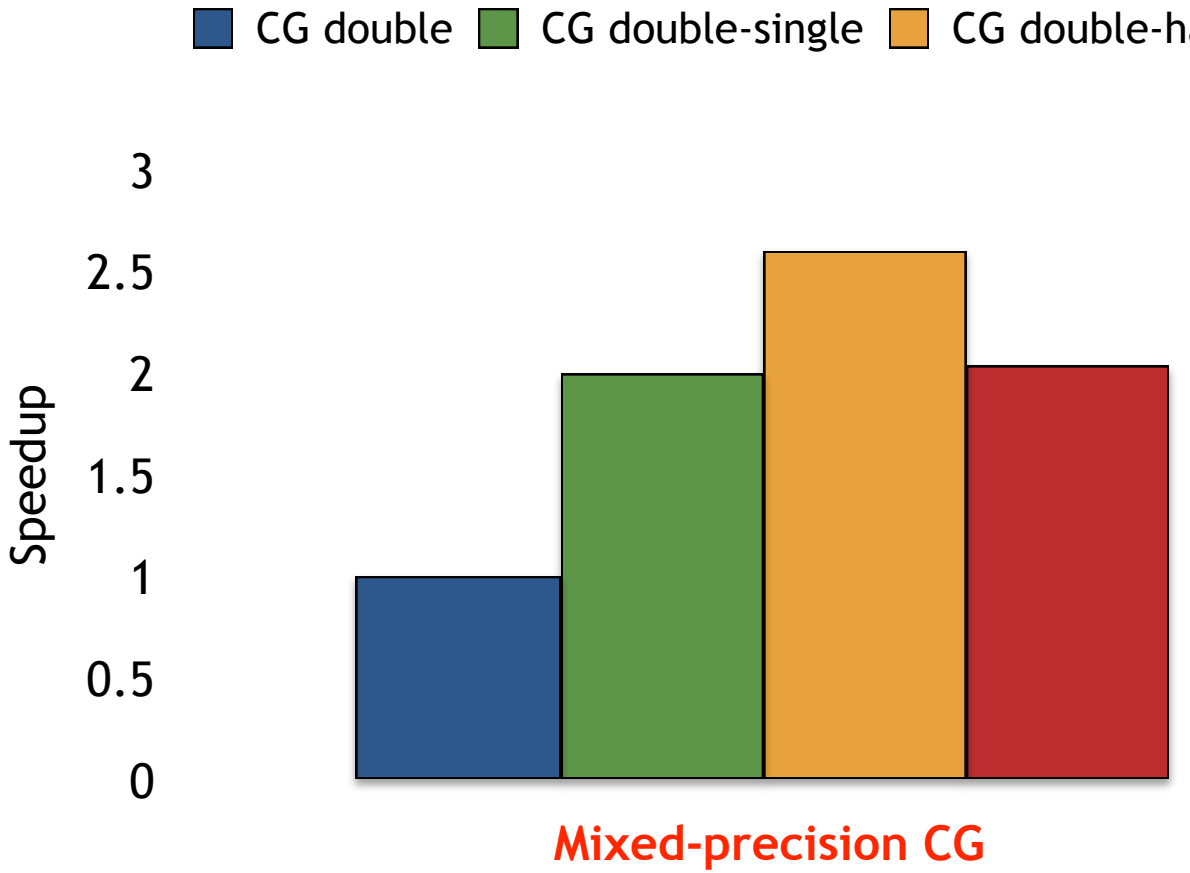
Memory overheads can be limiting factor

- May require storage of 1000s of vectors, ideally in fast memory

Configuration provided by
HotQCD collaboration (Mukherjee *et al*)

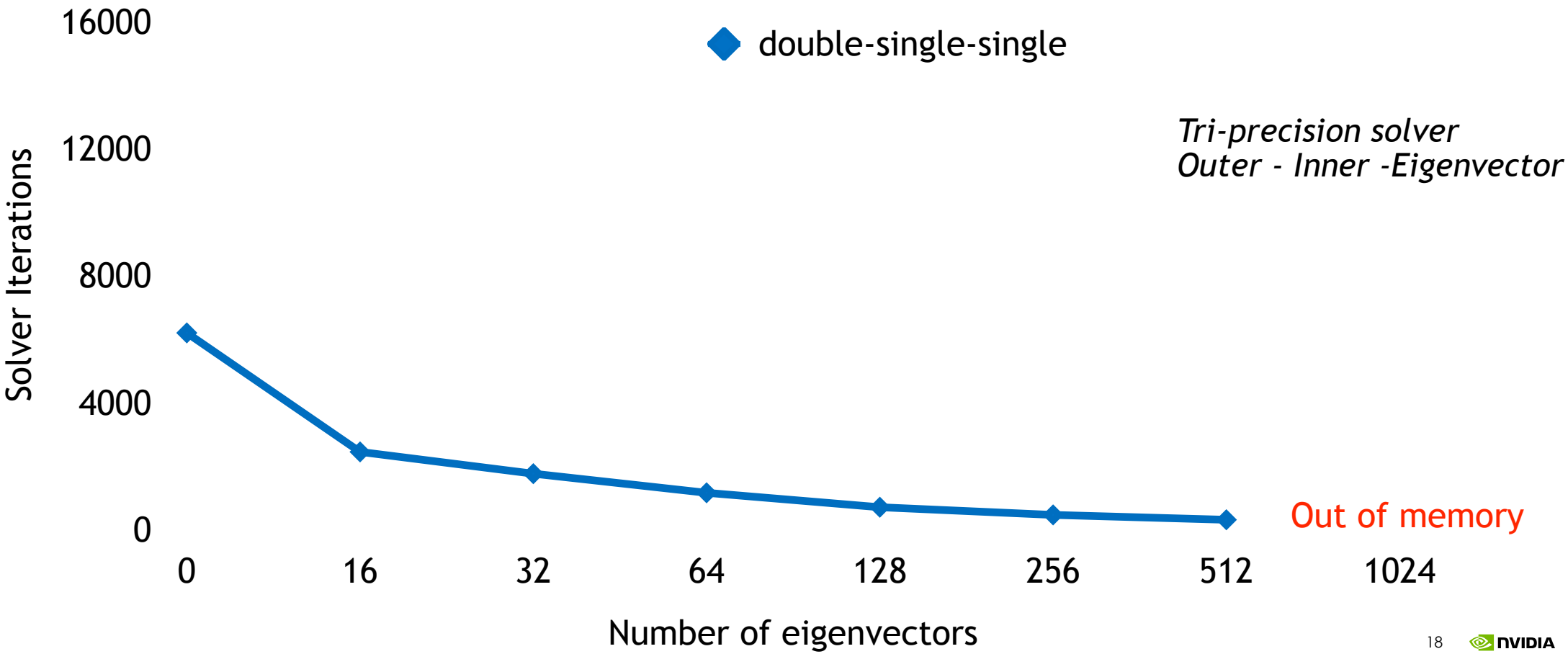
# MIXED-PRECISION DEFLATION

## V=$48^3$x12, HISQ operator, physical light quarks, tol $10^{-10}$, 2xV100



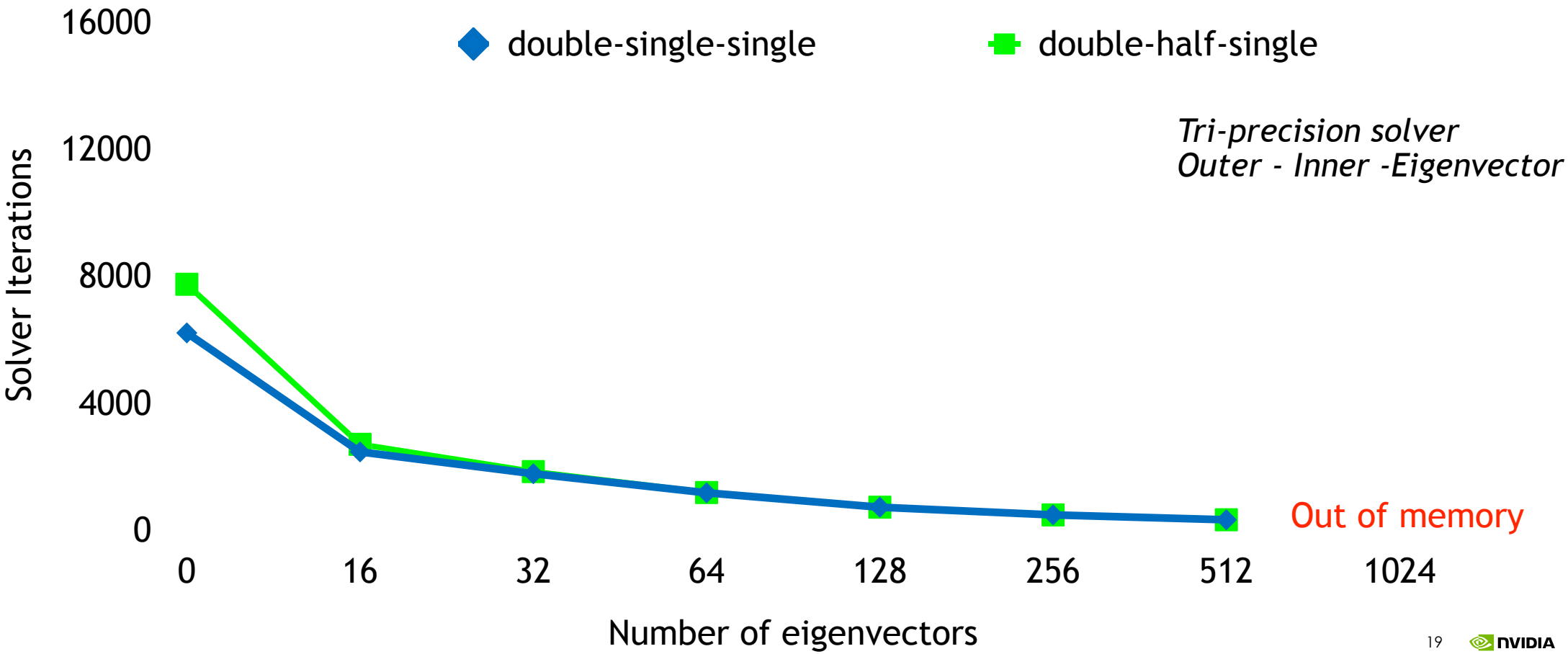
# DEFLATION STABILIZES LOW PRECISION

## V=$48^3$x12, HISQ operator, physical light quarks, tol $10^{-10}$



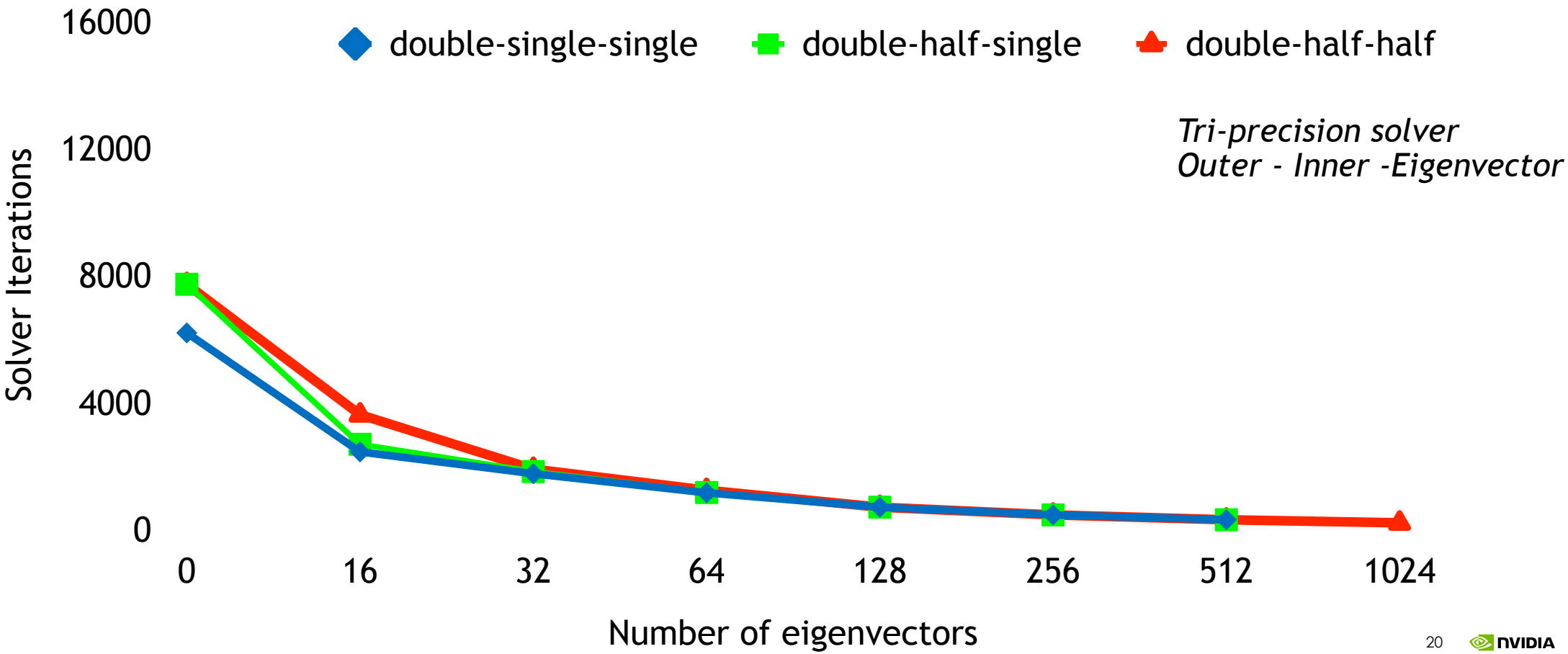
# DEFLATION STABILIZES LOW PRECISION

## V=$48^3$x12, HISQ operator, physical light quarks, tol $10^{-10}$



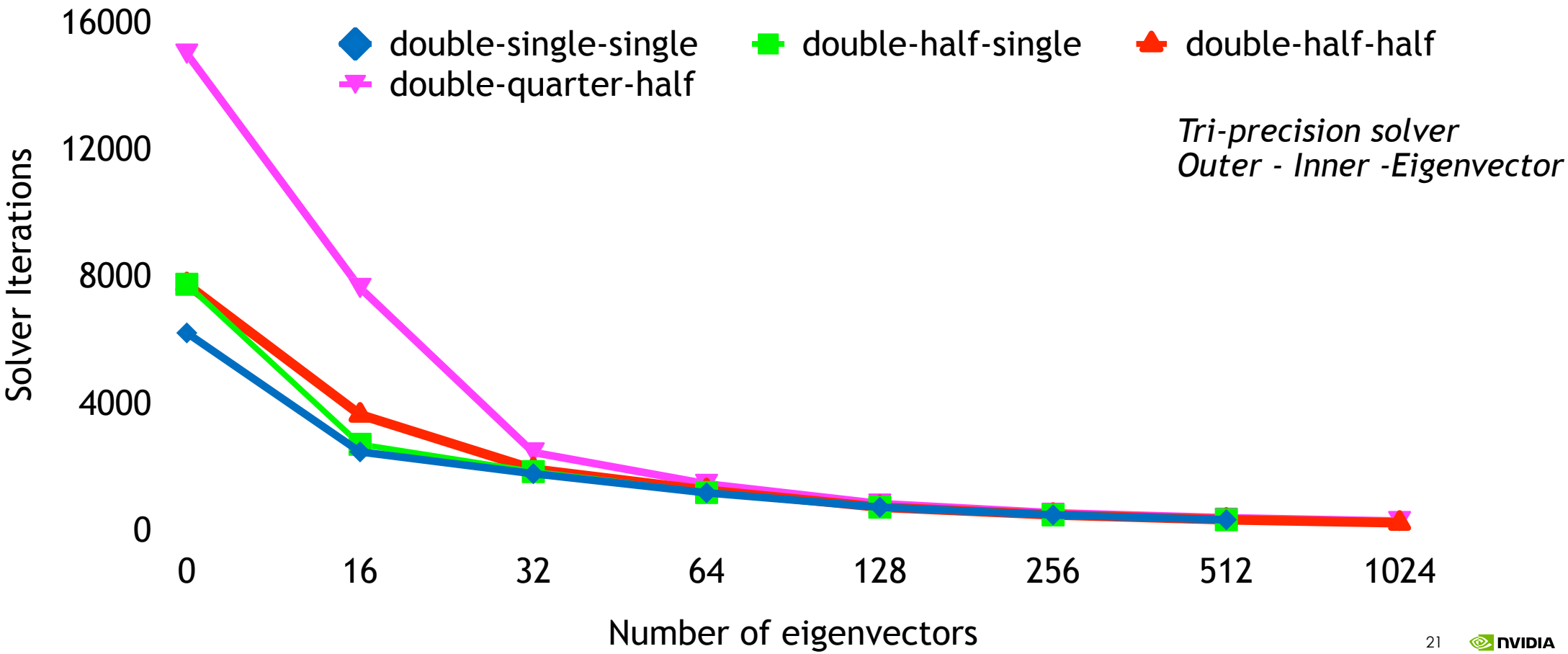
# DEFLATION STABILIZES LOW PRECISION

## V=$48^3$x12, HISQ operator, physical light quarks, tol $10^{-10}$



# DEFLATION STABILIZES LOW PRECISION

## V=$48^3$x12, HISQ operator, physical light quarks, tol $10^{-10}$



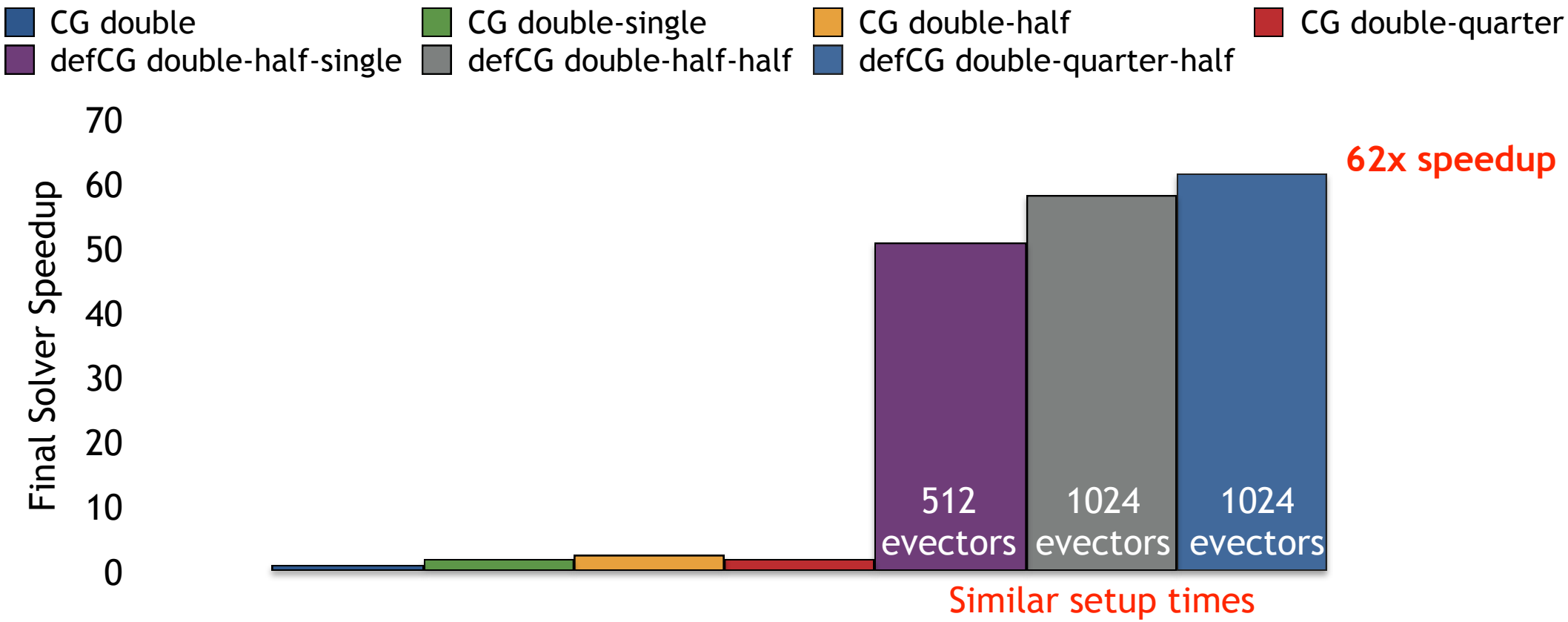



NVIDIA

*Configuration provided by*
*HotQCD collaboration (Mukherjee et al)*

# MIXED-PRECISION DEFLATION

## V=$48^3$x12, HISQ operator, physical light quarks, tol $10^{-10}$, 2xV100



# SCALING

# STRONG SCALING

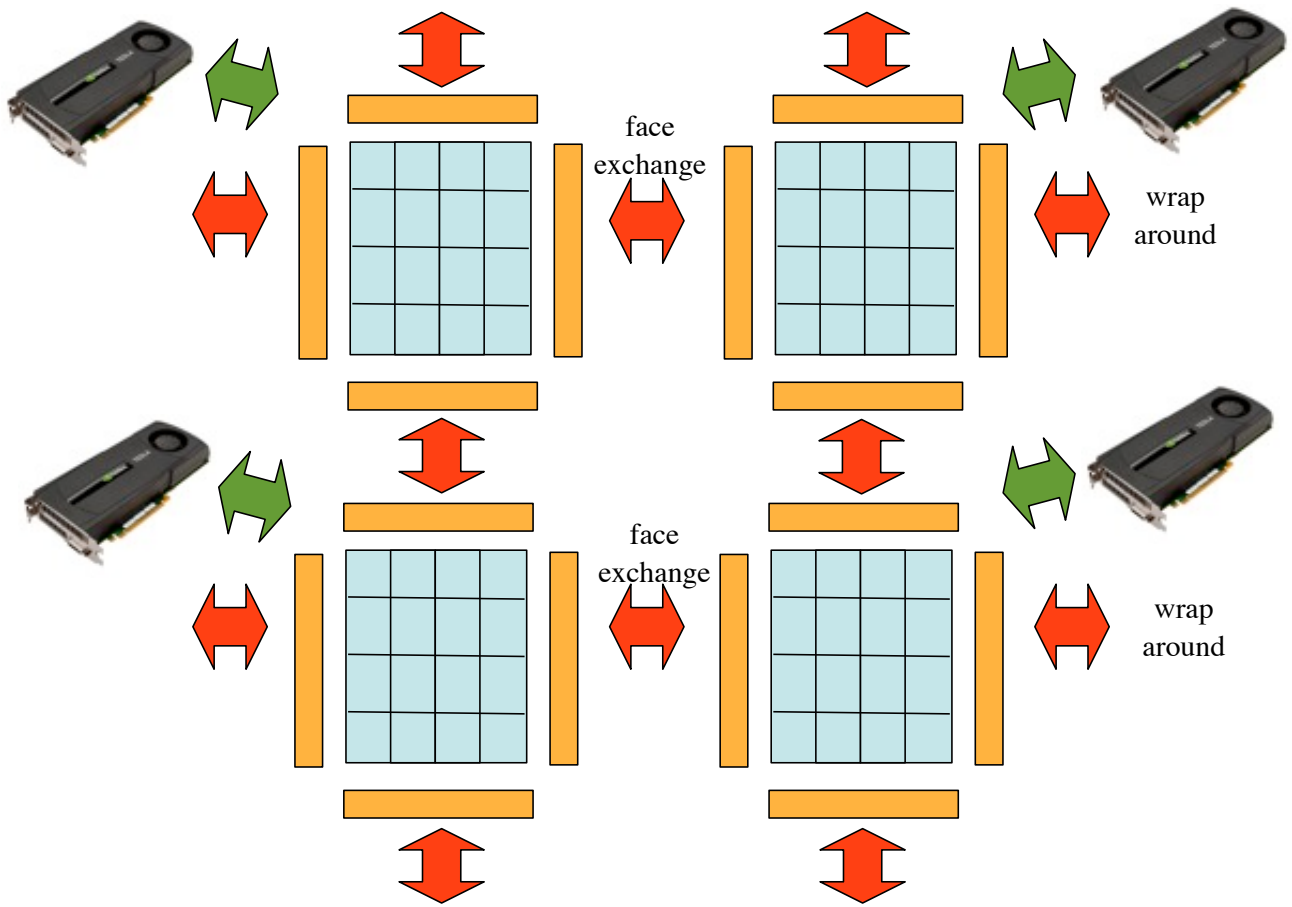
Multiple meanings

- Same problem size, more nodes, more GPUs
- Same problem, next generation GPUs
- Multigrid - strong scaling within the same run (not discussed here)

To tame strong scaling we have to understand the limiters

- Bandwidth limiters
- Latency limiters

# MULTI-GPU BUILDING BLOCKS




Halo packing Kernel

Interior Kernel

Halo communication

Halo update Kernel

# BENCHMARKING TESTBED

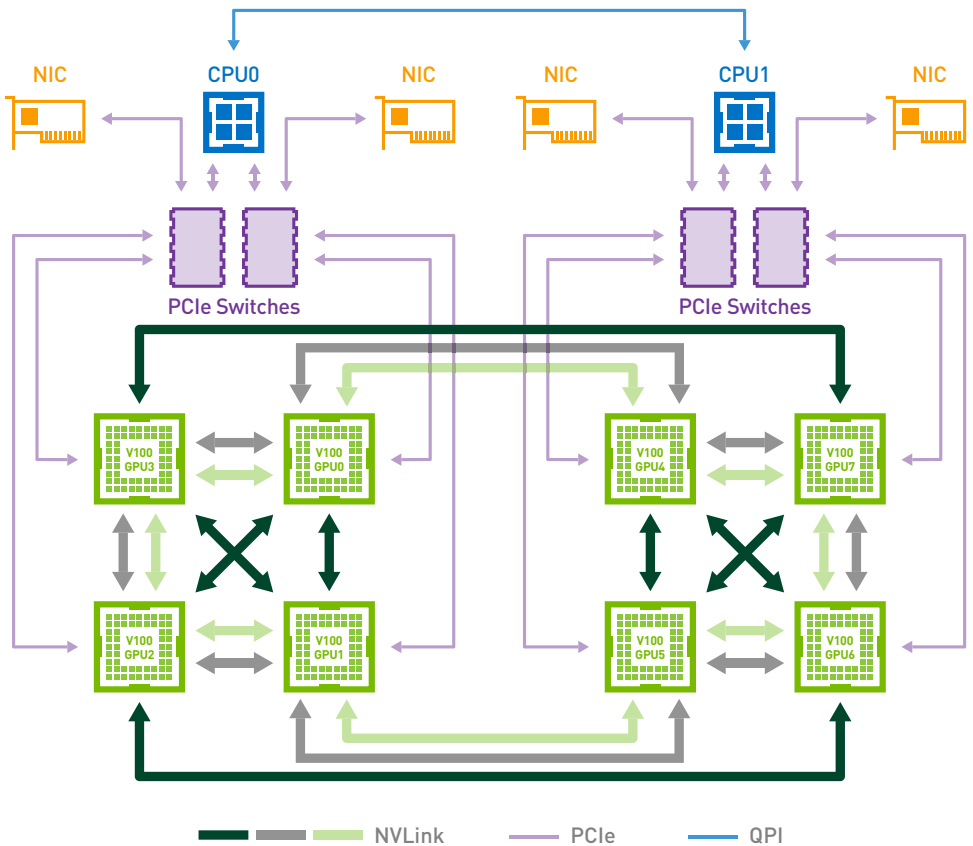
## NVIDIA Prometheus Cluster

DGX-1 nodes

- 8x V100 GPUs connected through NVLink
- 4x EDR for inter-node communication
- Optimal placement of GPUs and NIC

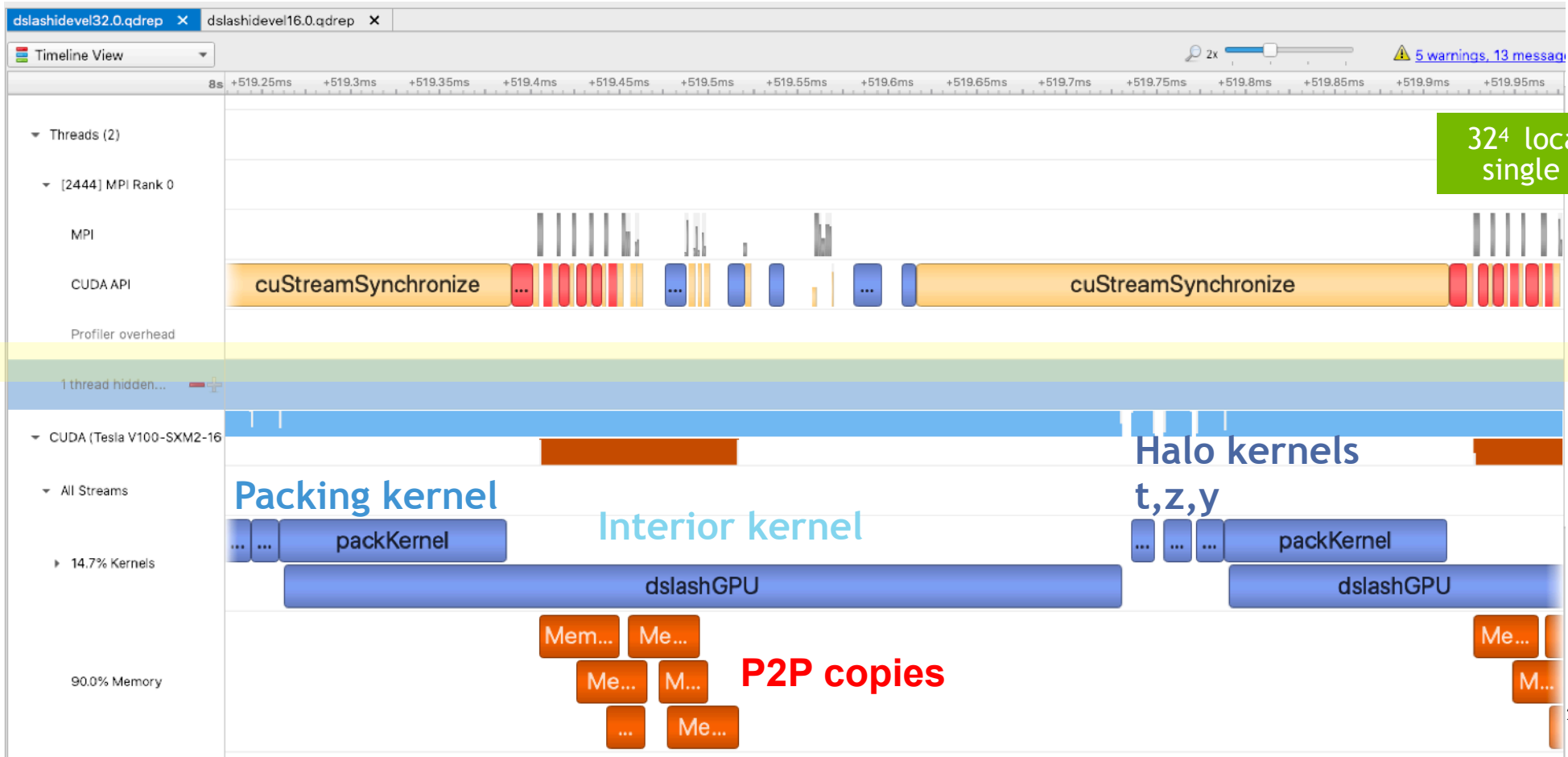
Balanced GPU / IB configuration

understand limiters
on a single node first



# MULTI-GPU PROFILE

## overlapping comms and compute

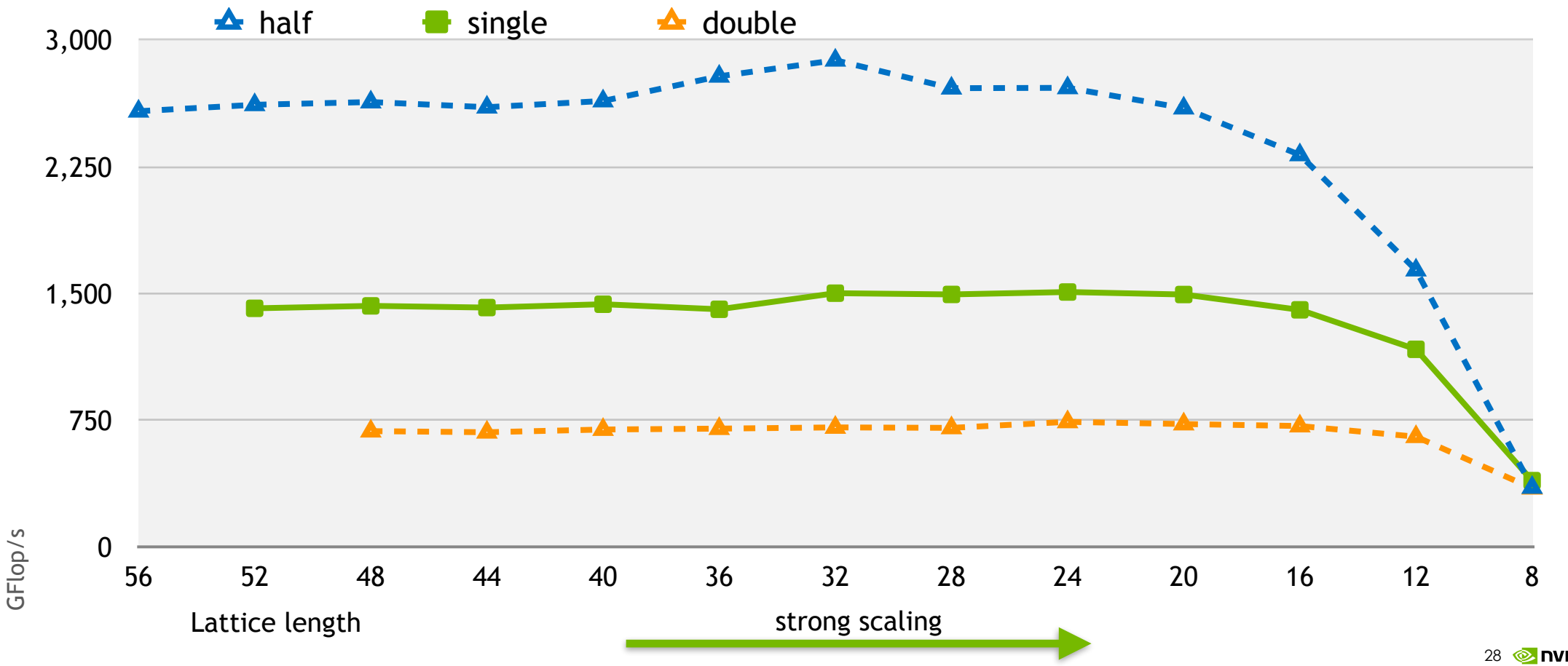



$32^4$ local volume, single precision

DGX-1, 1x2x2x2 partitioning

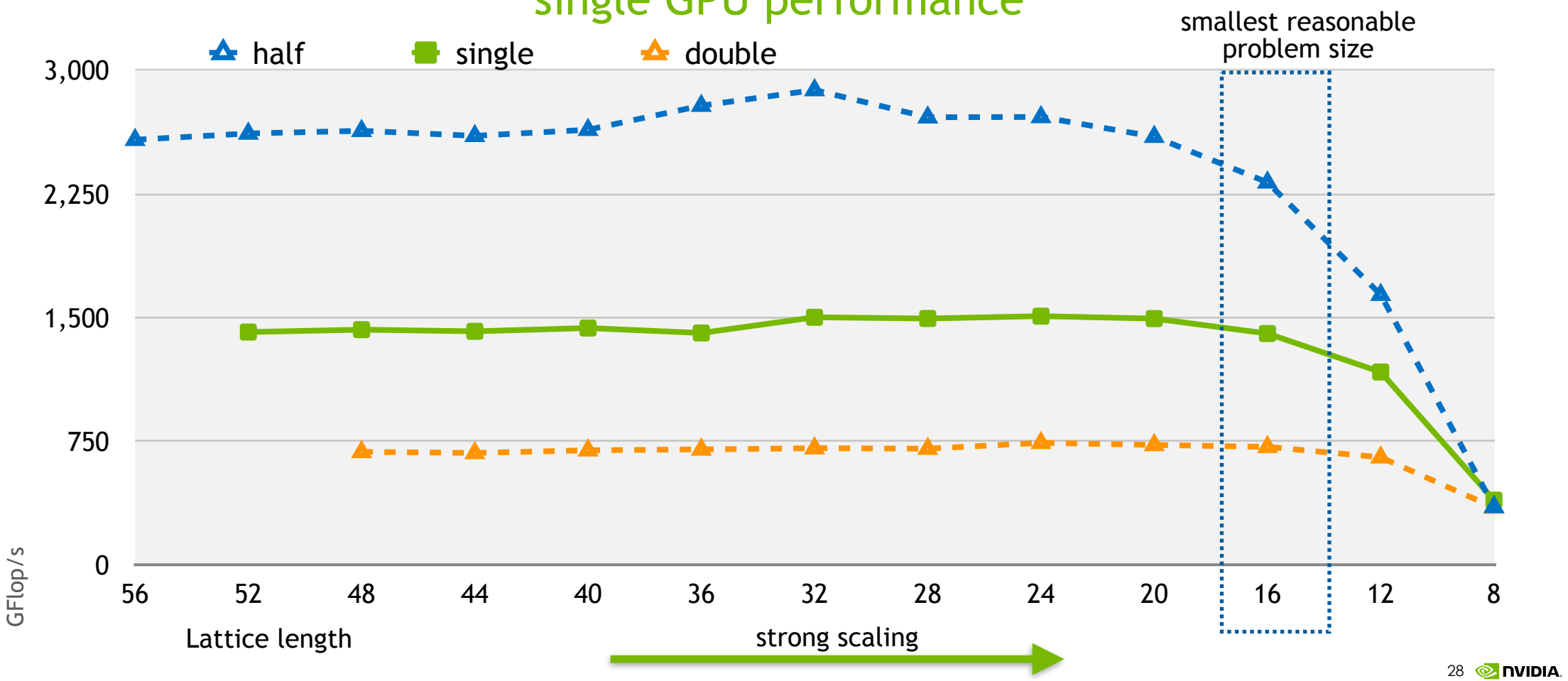
# REDUCING LOCAL PROBLEM SIZE

## single GPU performance



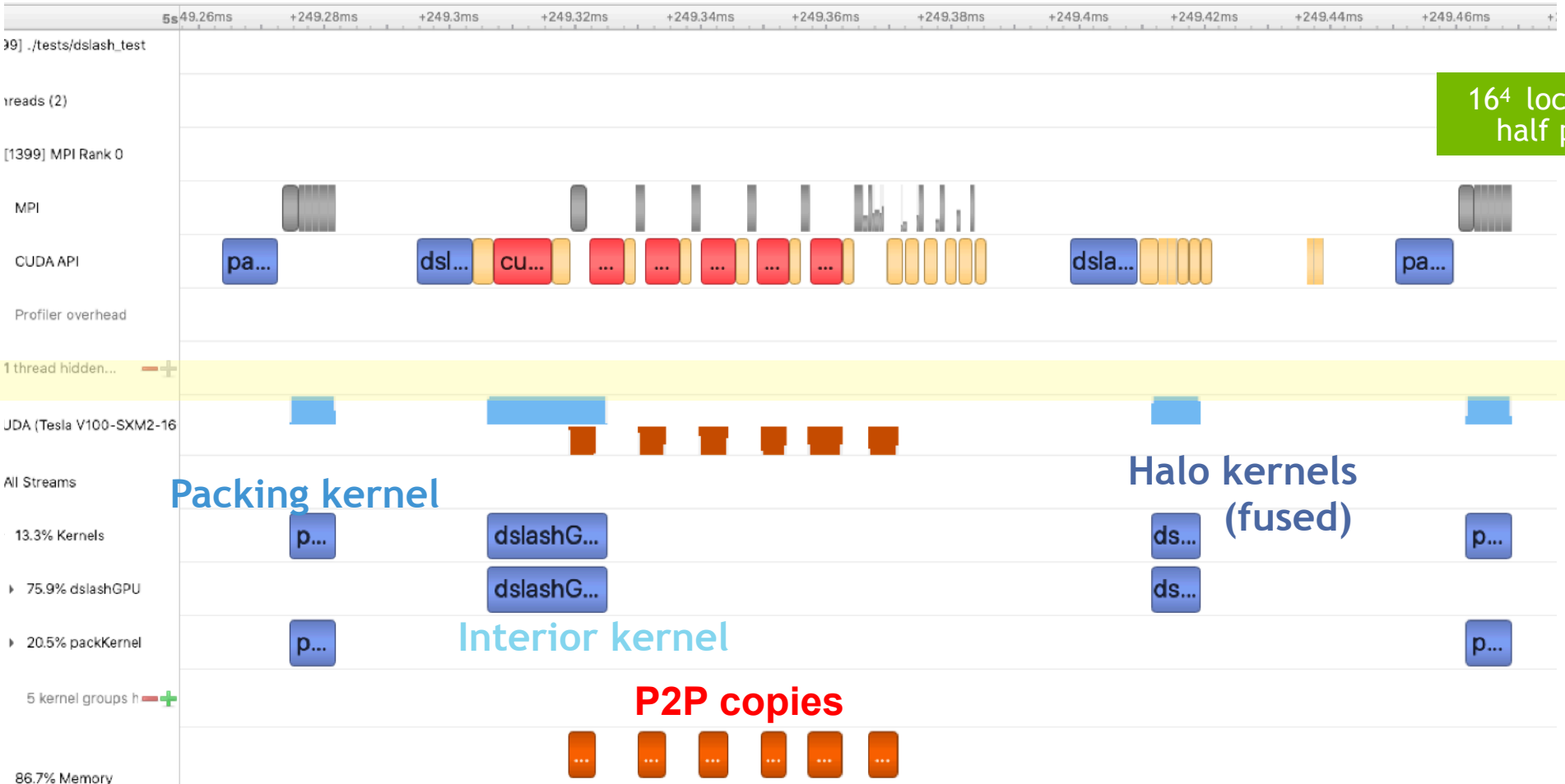
# REDUCING LOCAL PROBLEM SIZE

## single GPU performance



# STRONG SCALING PROFILE

## ~~overlapping~~ comms and compute

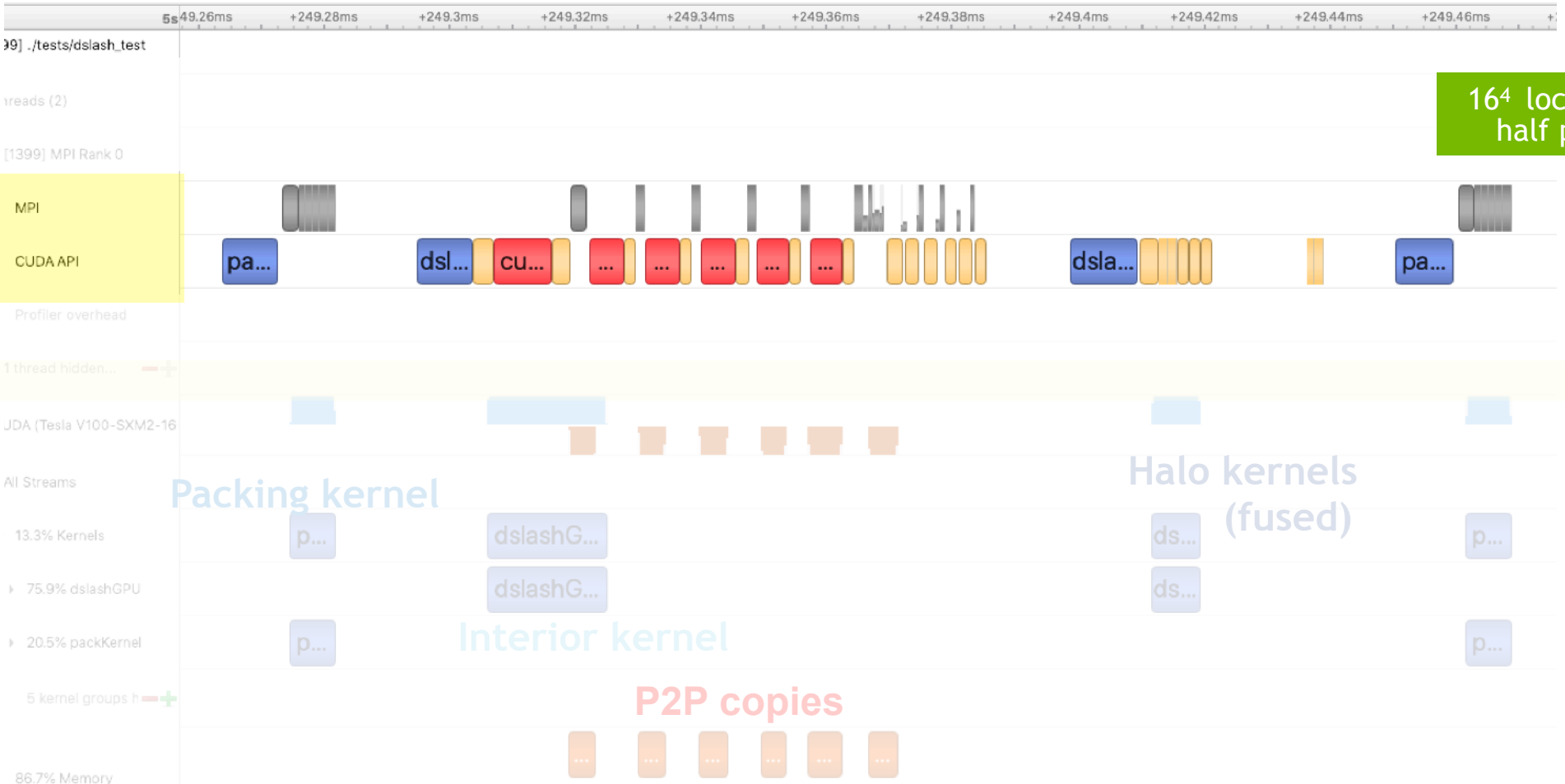



$16^4$ local volume, half precision

DGX-1, 1x2x2x2 partitioning

# STRONG SCALING PROFILE

## Latencies ate my scaling

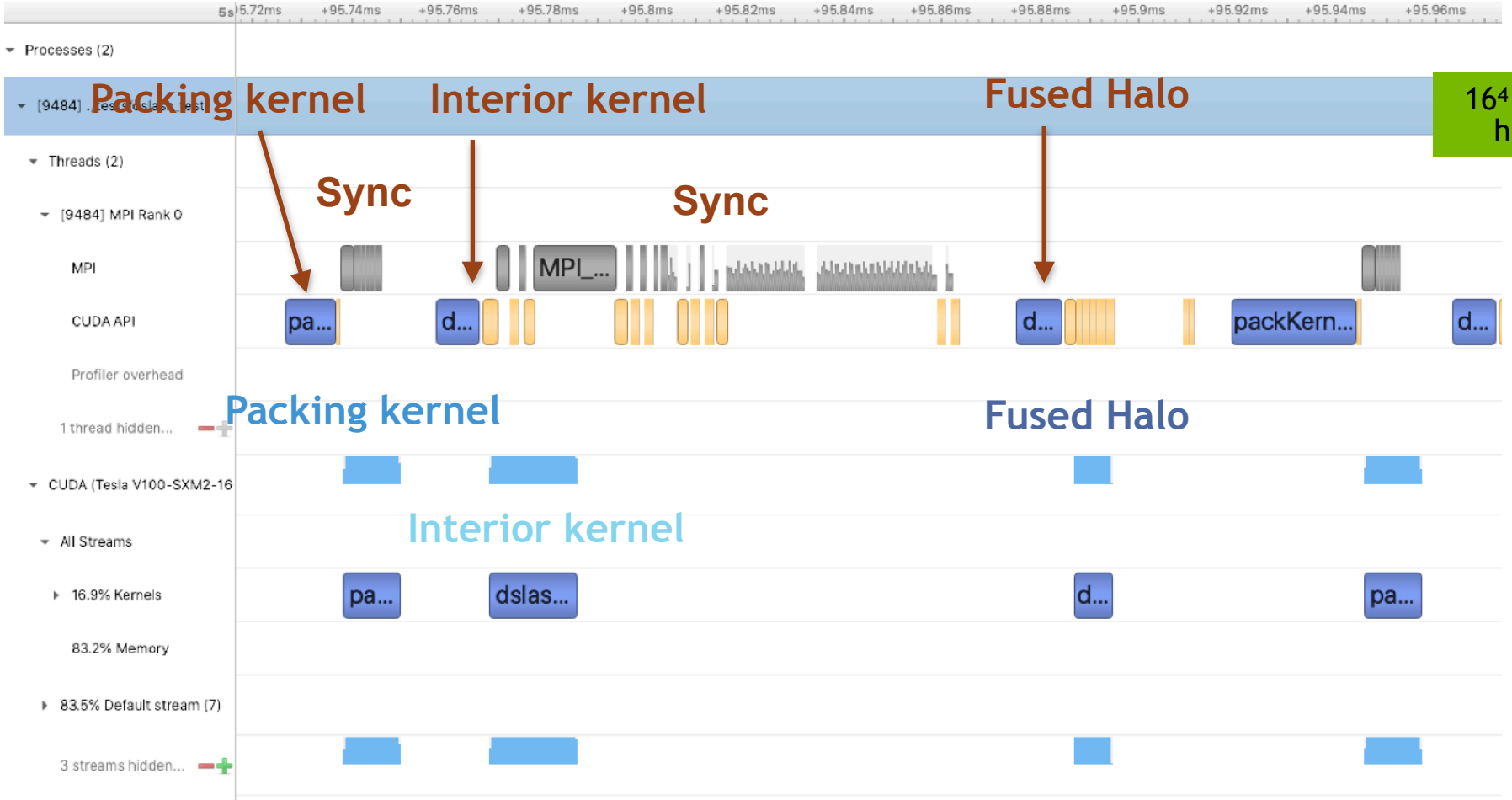



API Calls

$16^4$ local volume, half precision

# REDUCING API OVERHEADS

## Packing kernel writes to remote GPU using CUDA IPC




$16^4$ local volume, half precision

DGX-1, 1x2x2x2 partitioning

currently in early-access

# NVSHMEM

## Implementation of OpenSHMEM1, a Partitioned Global Address Space (PGAS) library

NVSHMEM features

- Symmetric memory allocations in device memory
- Communication API calls on CPU (standard and stream-ordered)
- Kernel-side communication (API and LD/ST) between GPUs

NVLink and PCIe support (intranode)

InfiniBand support (internode)

Interoperability with MPI and OpenSHMEM libraries

[1] SHMEM from Cray's "shared memory" library, https://en.wikipedia.org/wiki/SHMEM

# DSLASH NVSHMEM IMPLEMENTATION

## First exploration

Keep general structure of packing, interior and exterior Dslash

Use `nvshmem_ptr` for intra-node remote writes (fine-grained)

Packing buffer is located on remote device

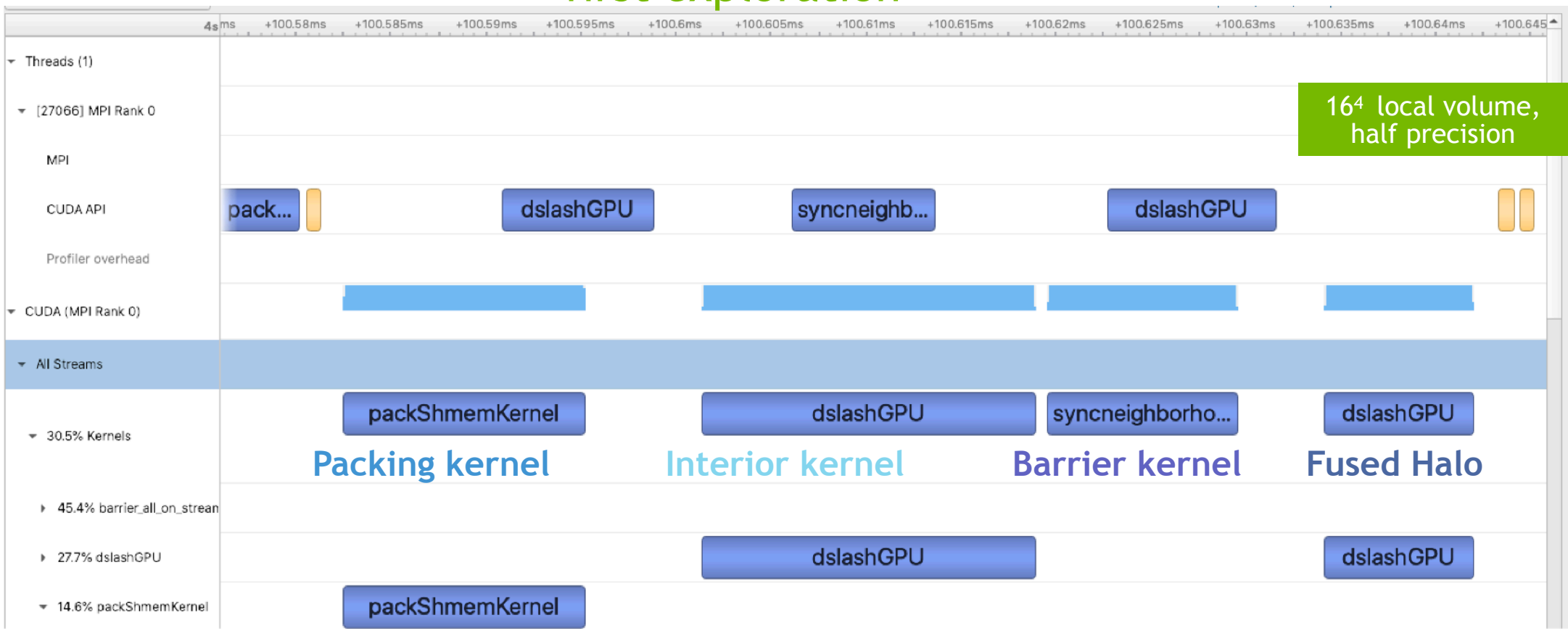
Use `nvshmem_putmem_nbi` to write to remote GPU over IB (1 RDMA transfer)

Need to make sure writes are visible: `nvshmem_barrier_all_on_stream`

or barrier kernel that only waits for writes from neighbors

# NVSHMEM DSLASH

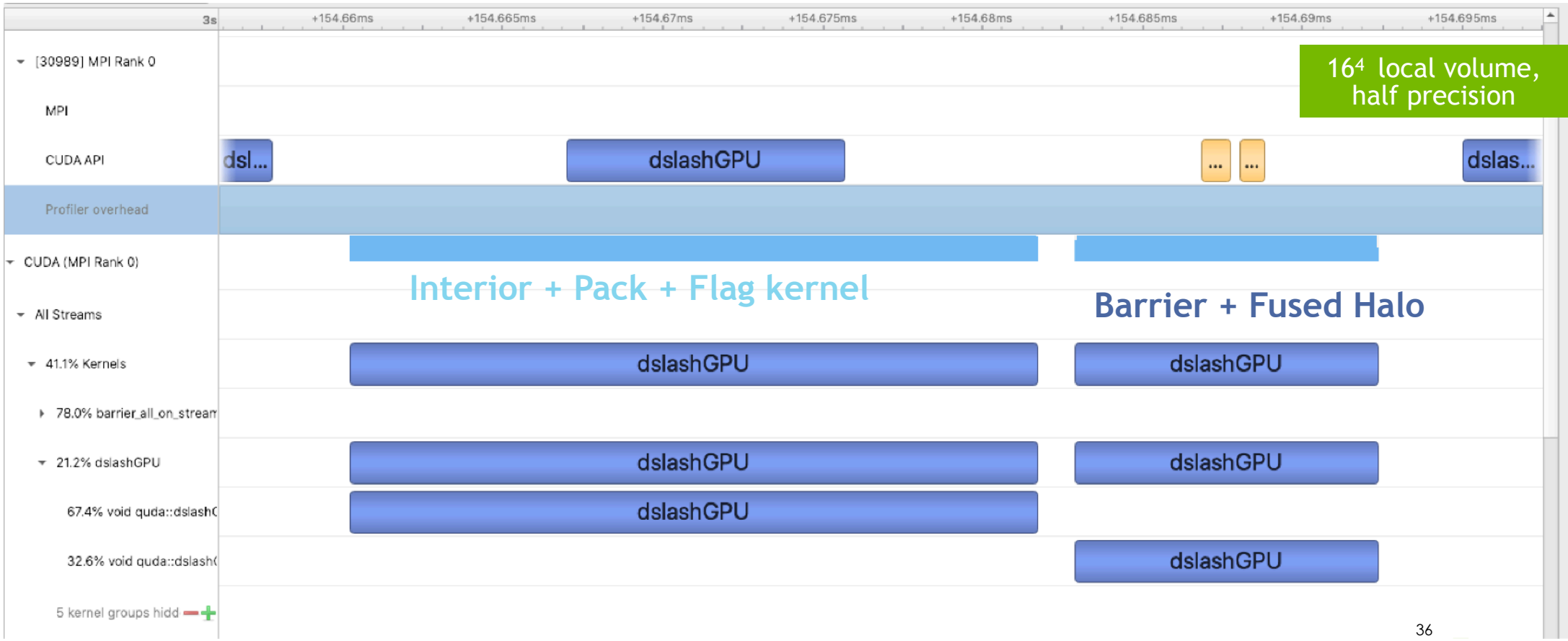
## first exploration




DGX-1, 1x2x2x2 partitioning

# NVSHMEM + FUSING KERNELS

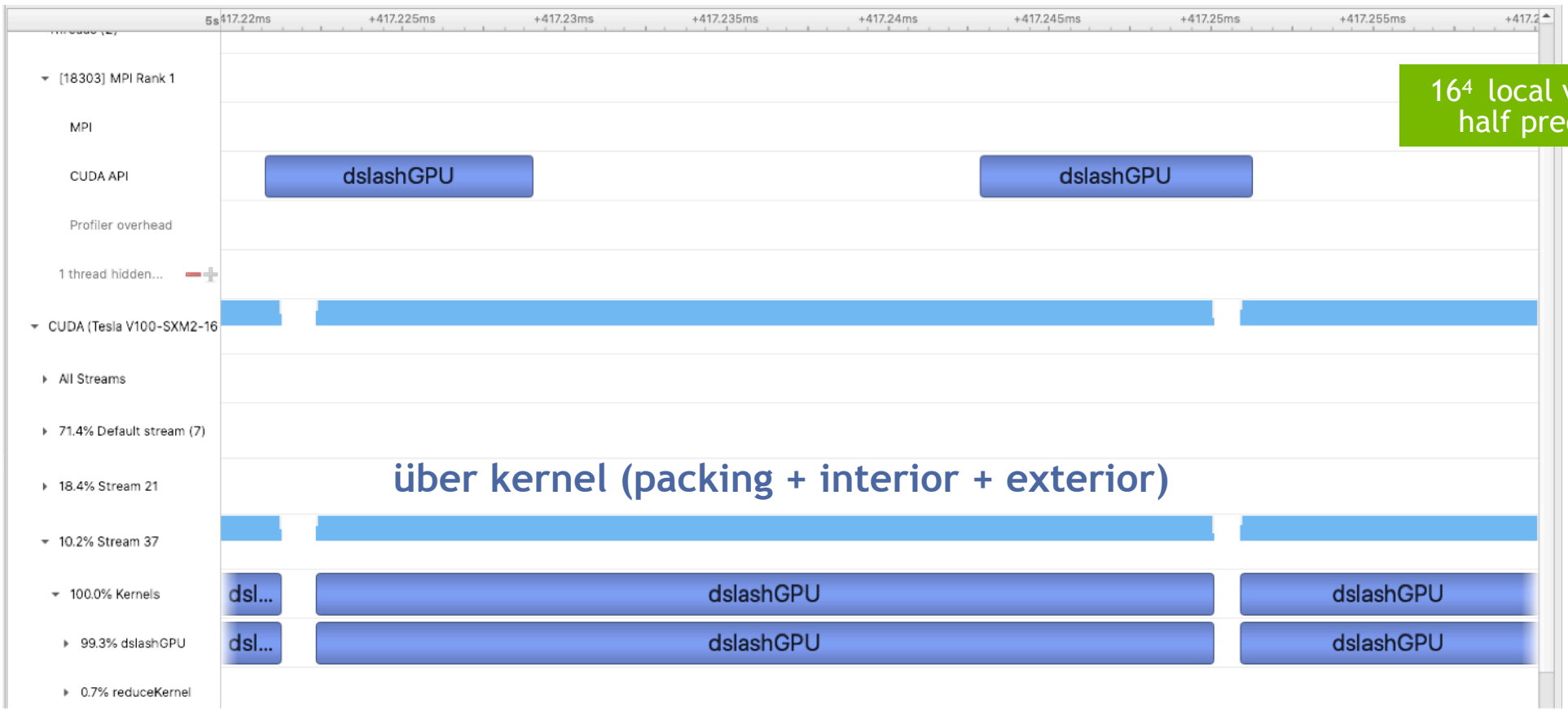
## no extra packing and barrier kernels needed




DGX-1, 1x2x2x2 partitioning

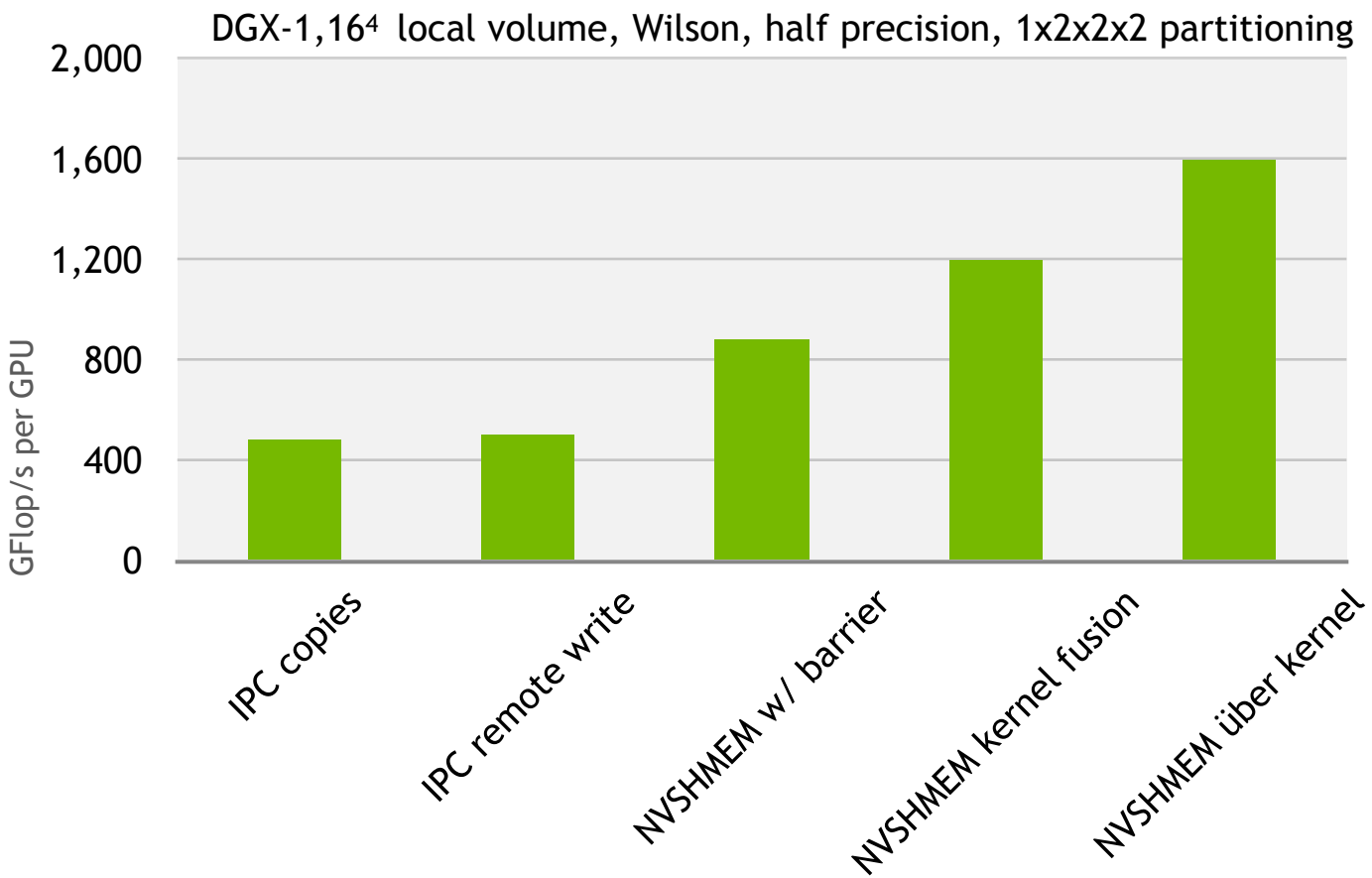
# ÜBER KERNEL

DGX-1, 1x2x2x2 partitioning



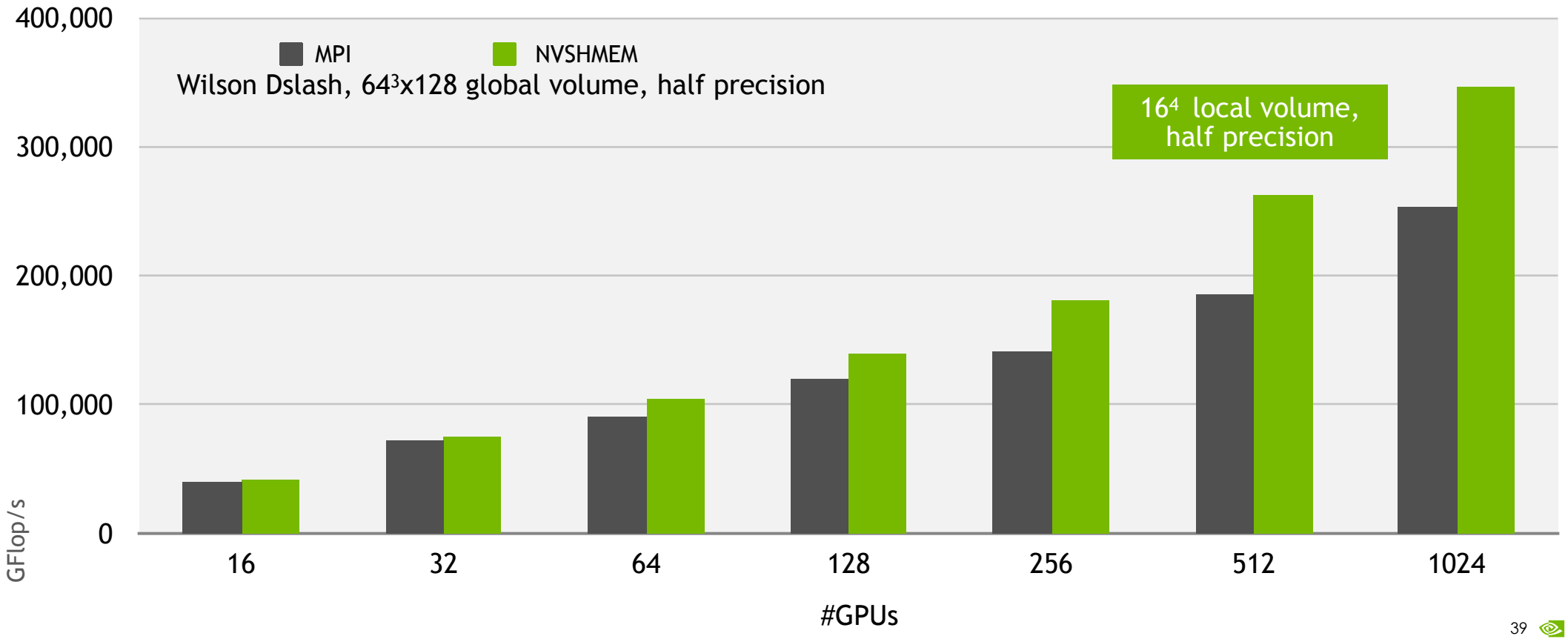

$16^4$ local volume, half precision

# LATENCY REDUCTIONS



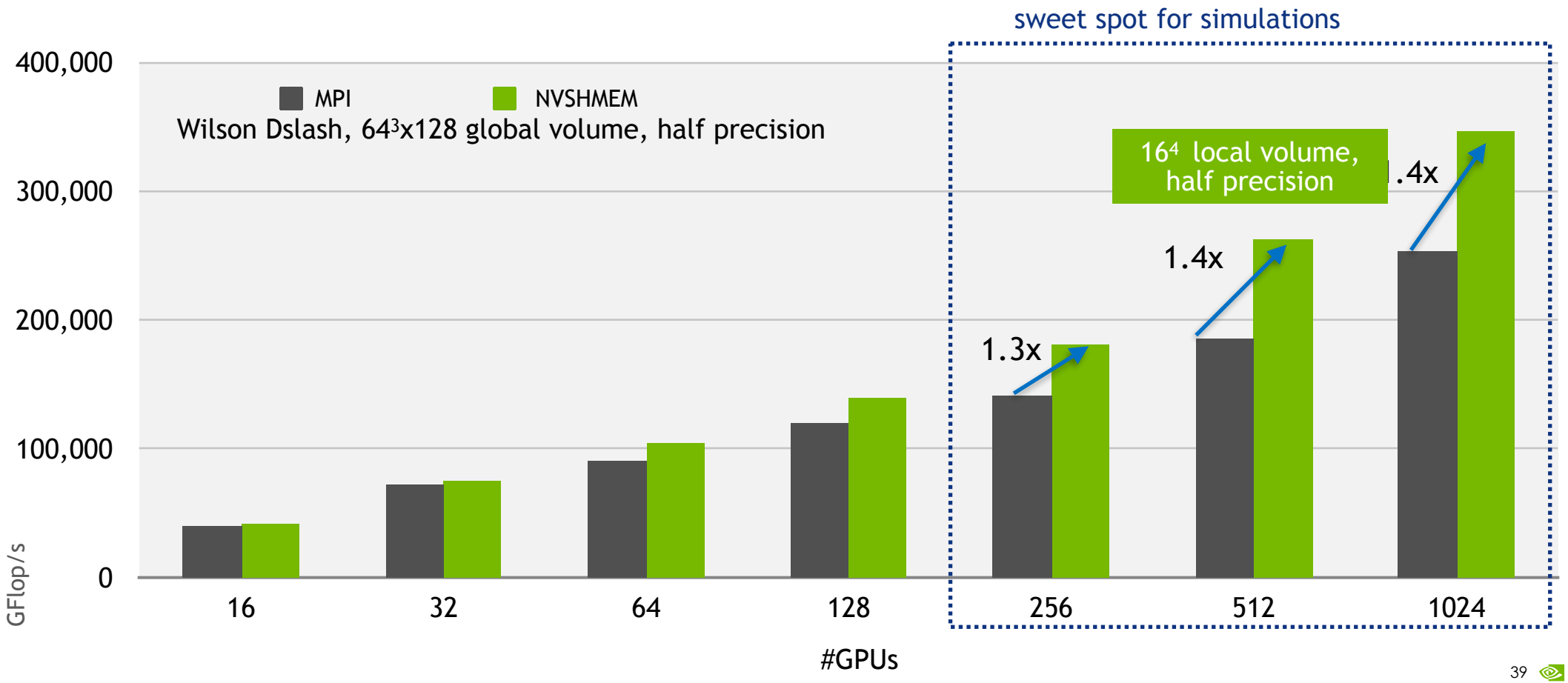
# MULTI-NODE STRONG SCALING

## DGX SuperPOD (DGX2 nodes: 16xV100 (32GB), 8xEDR IB)



# MULTI-NODE STRONG SCALING

## DGX SuperPOD (DGX2 nodes: 16xV100 (32GB), 8xEDR IB)



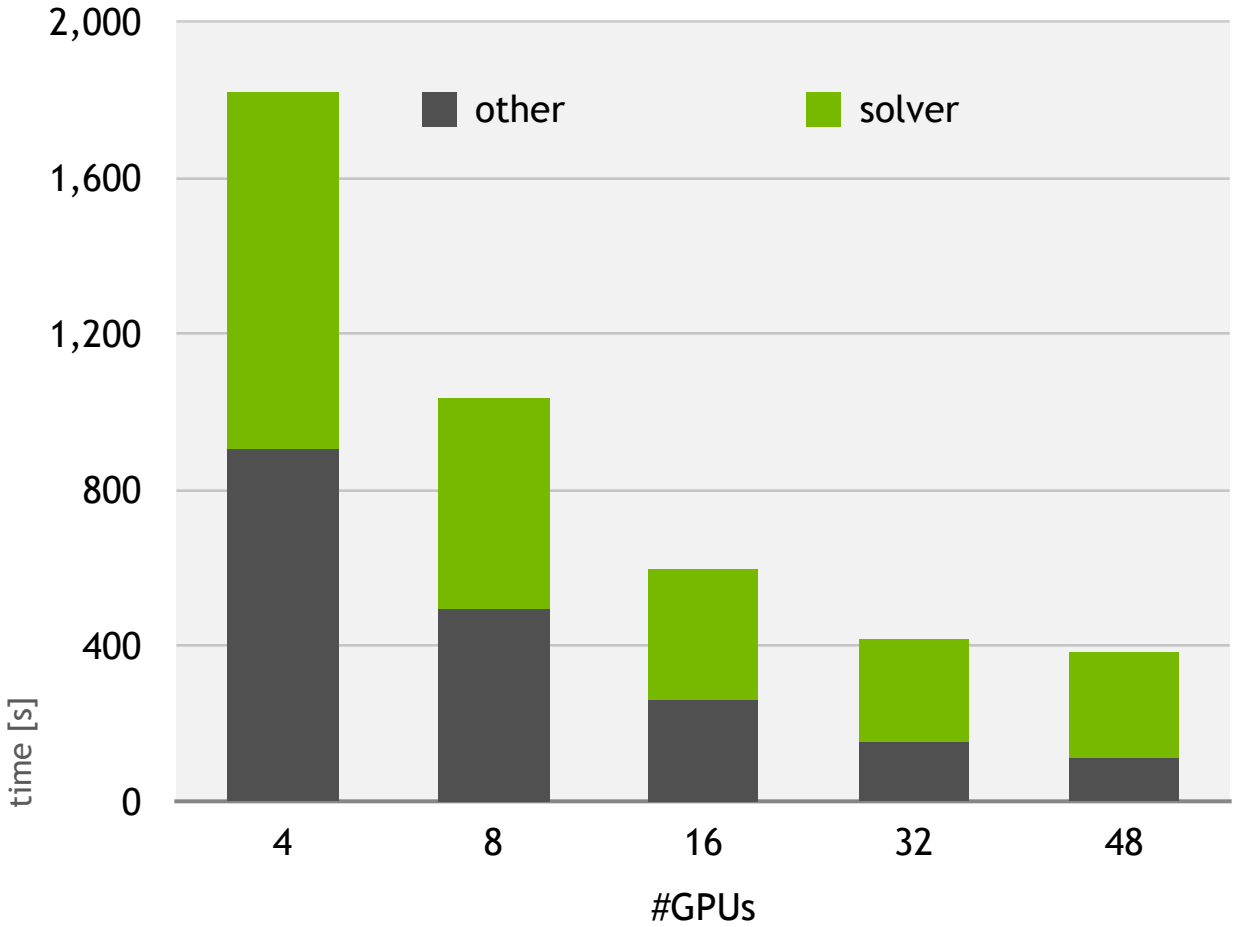
# APPLICATION SCALING

# MILC NERSC BENCHMARK OVERVIEW

- MILC NERSC Benchmark comes in 4 lattice sizes
  - small $18^3$x36, medium **$36^3$x72**, large $72^3$x144, x-large $144^3$x288
- Benchmark runs the RHMC algorithm
  - Dominated by the multi-shift CG sparse linear solver (stencil operator)
  - Also have auxiliary “Force” and “Link” computations
- Since 2012 MILC has built-in QUDA support
  - Enabled through a Makefile option
  - All time-critical functions off loaded to QUDA

# MILC HMC SCALING ON DGX-2

## NERSC MEDIUM BENCHMARK $36^3 x 72$



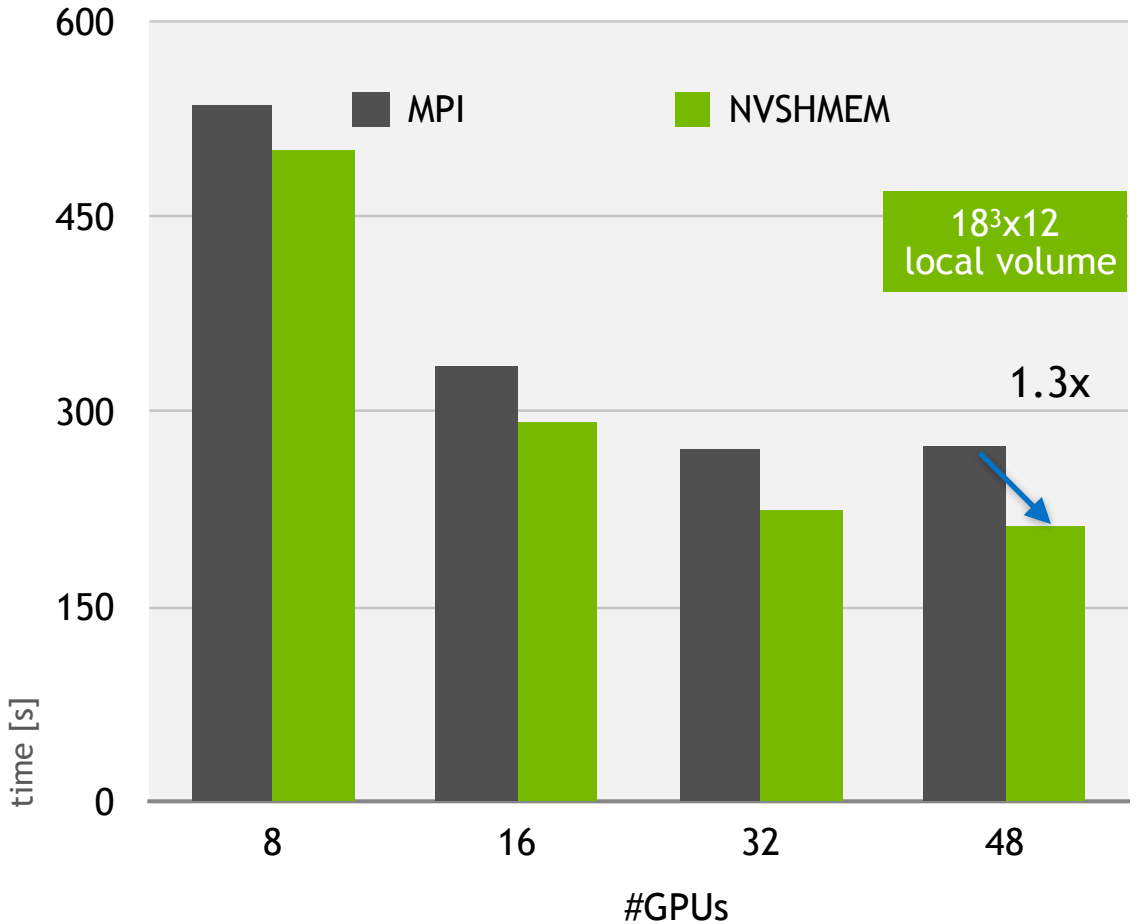

Running with MPI

other part scales reasonably
(not limited by communication)

solver part needs improvements

# MILC SOLVER SCALING ON DGX-2

## NERSC MEDIUM BENCHMARK $36^3x72$




multi-shift solver

mixed precision: double-single
refinement: double-half

MPI version overlaps BLAS + comms

NVSHMEM scales beyond 32 GPUs

# MULTIPLICATIVE SPEEDUP

# CHROMA HMC MULTIGRID

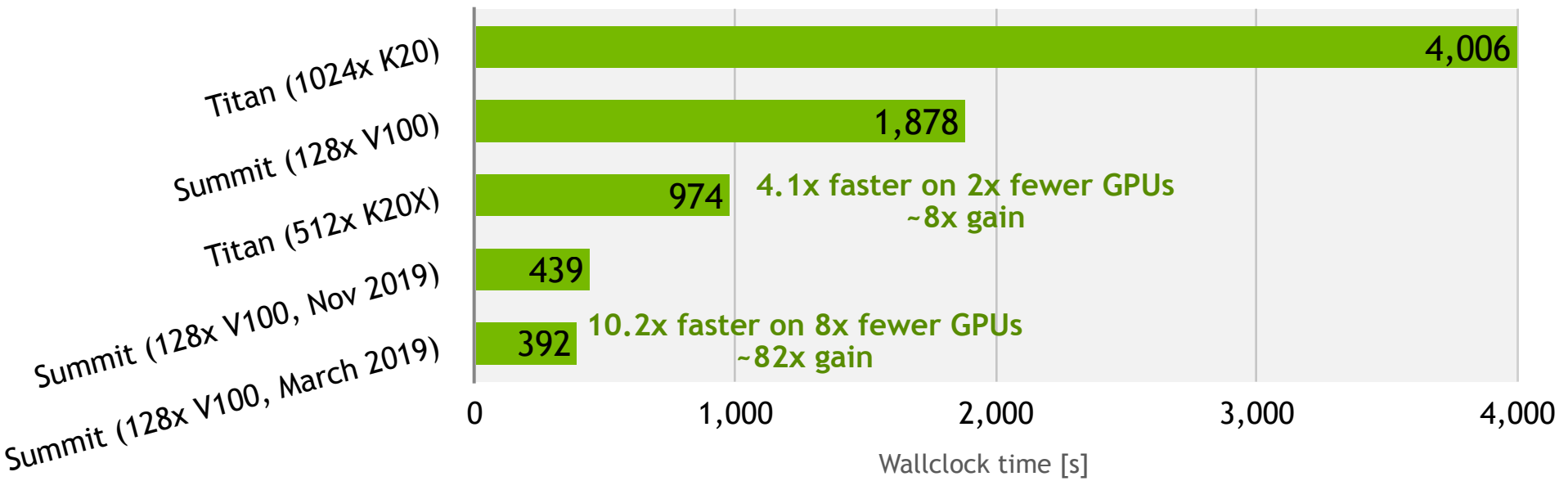
HMC typically dominated by solving the Dirac equation, but

- Few solves per linear system
- Can be bound by heavy solves (c.f. Hasenbusch mass preconditioning)

Multigrid setup must run at speed of light

- Reuse and evolve multigrid setup where possible
- Use the same null space for all masses (setup run on lightest mass)
- Evolve null space vectors as the gauge field evolves (Lüscher 2007)
- Update null space when the preconditioner degrades too much on lightest mass

# CHROMA HMC-MG ON SUMMIT




in collaboration with
Bálint Joó – Chroma
Boram Yoon - Force gradient integrator
Frank Winter – QDP-JIT

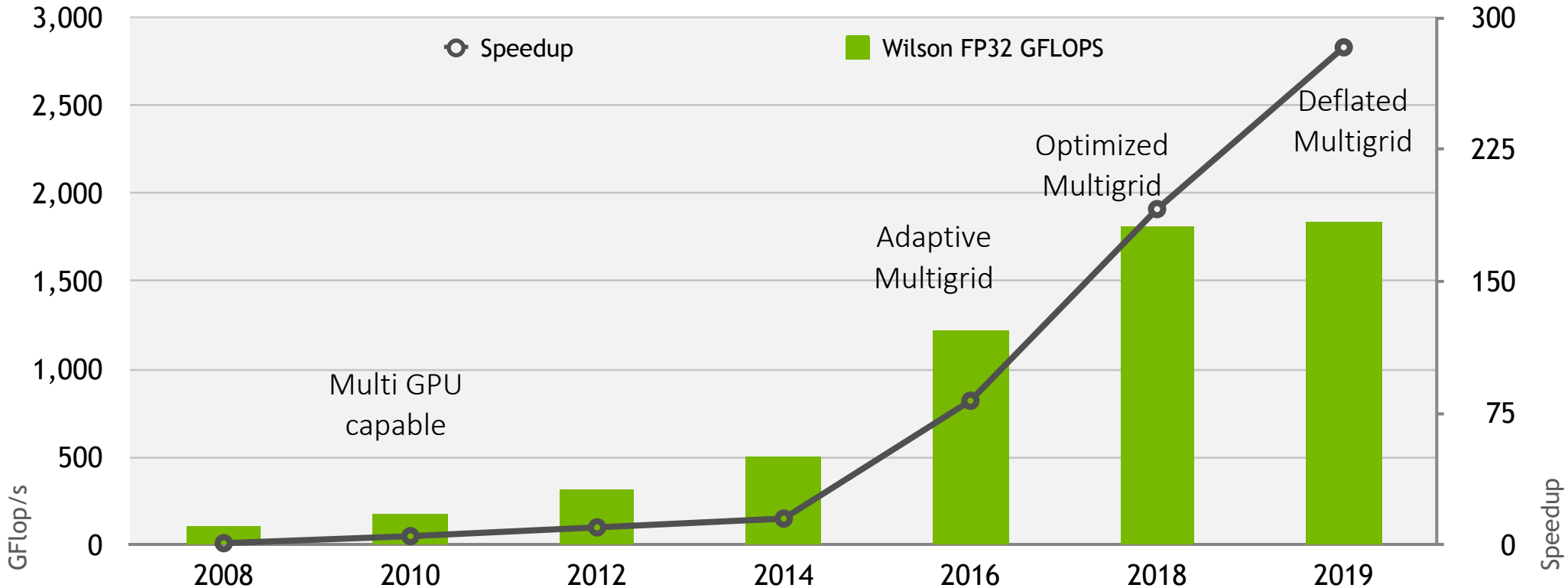
From Titan running 2016 code to Summit running 2019 code we see >82x speedup in HMC throughput

Multiplicative speedup coming from machine and algorithm innovation

Highly optimized multigrid for gauge field evolution

# NODE PERFORMANCE OVER TIME

## Multiplicative speedup through software and hardware




Time to solution is measured time to solution for solving the Wilson operator against a random source on a 24x24x24x64 lattice, β=5.5, $M_\pi$= 416 MeV. One node is defined to be 3 GPUs

# SUMMARIZING

# QUDA ROADMAP

## On to QUDA 2.0

Multi-rhs block solvers for all stencils

Contraction framework

Improved strong scaling through NVSHMEM

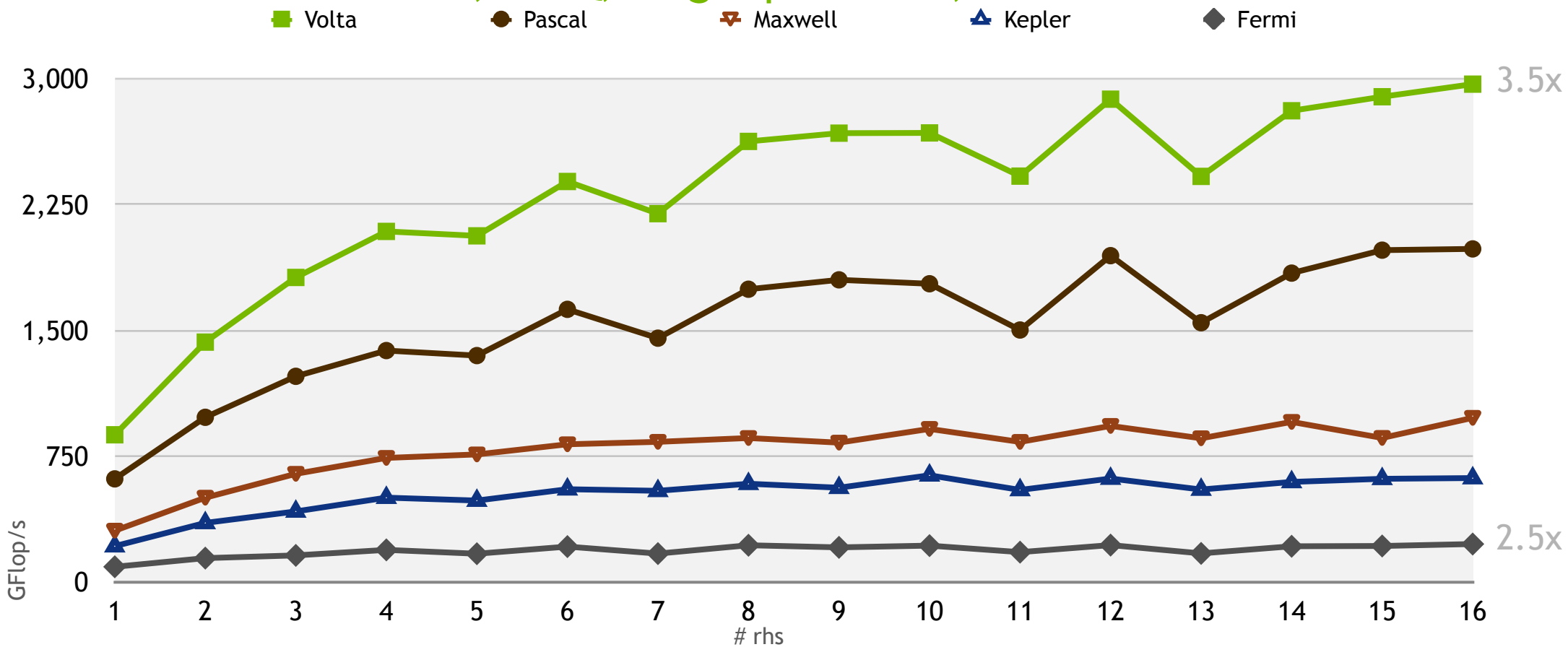
Beyond just regular QCD

Longer term: Investigate how well QUDA runs on C++17 pSTL

Post feature requests here: https://github.com/lattice/quda/issues

# MULTIPLE RIGHT-HAND SIDES

## $48^3x12$, HISQ, single precision, one code



# QUDA - LATTICE QCD ON GPUS

## Widely used for Lattice QCD applications on GPUs

State of the art solvers using mixed precision

- Multigrid
- Deflation
- Block-Krylov solver

All components for gauge field evolution

Portable high-performance kernels through auto-tuning and careful optimization
Tuned Multi-GPU scaling

- GPU centric communication with NVSHMEM takes CPU limitations out

**Multiplicative speedup from hardware and software: more science**


NVIDIA