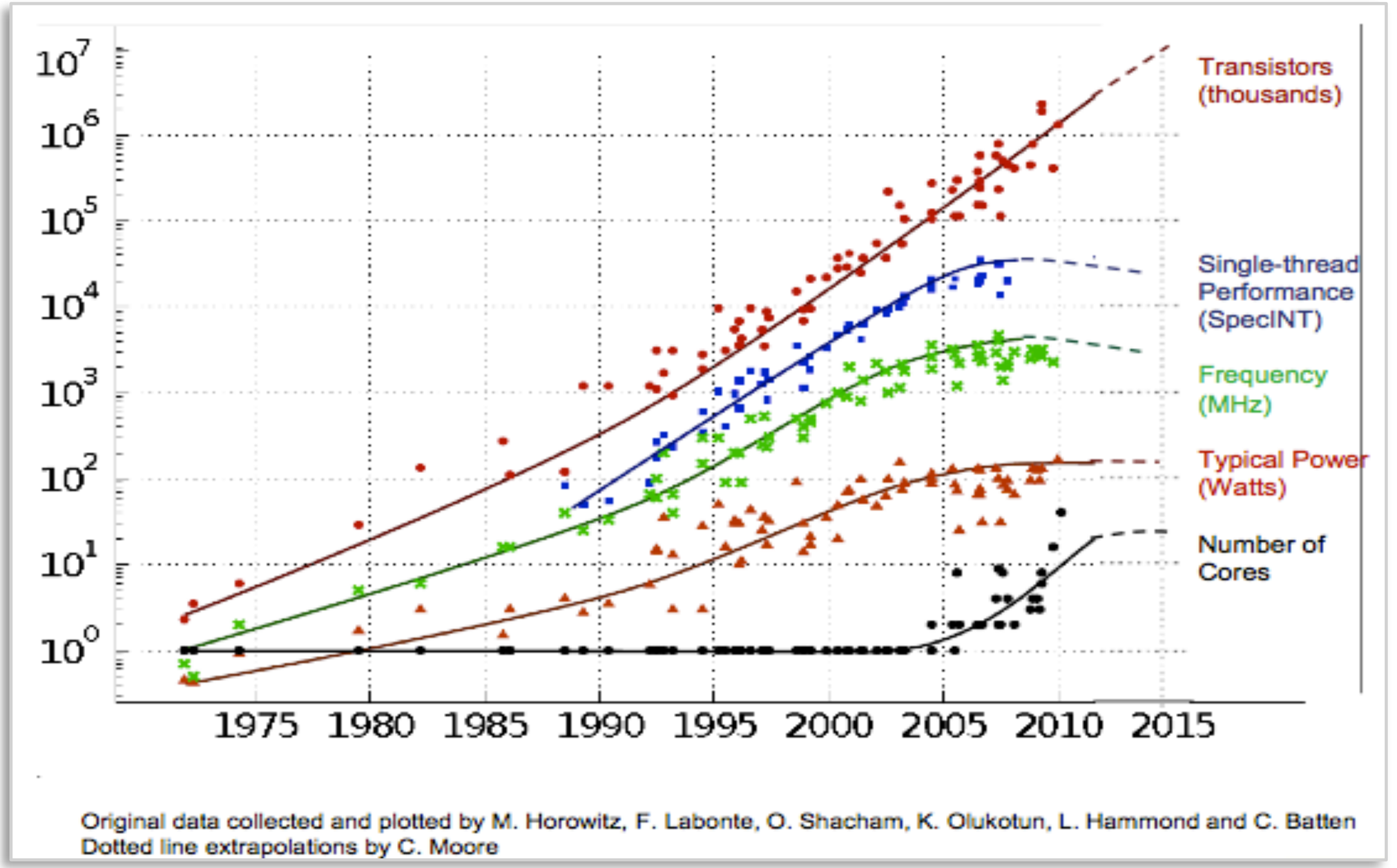
QCD TO THE EXASCALE

Kate Clark, August 2nd 2016

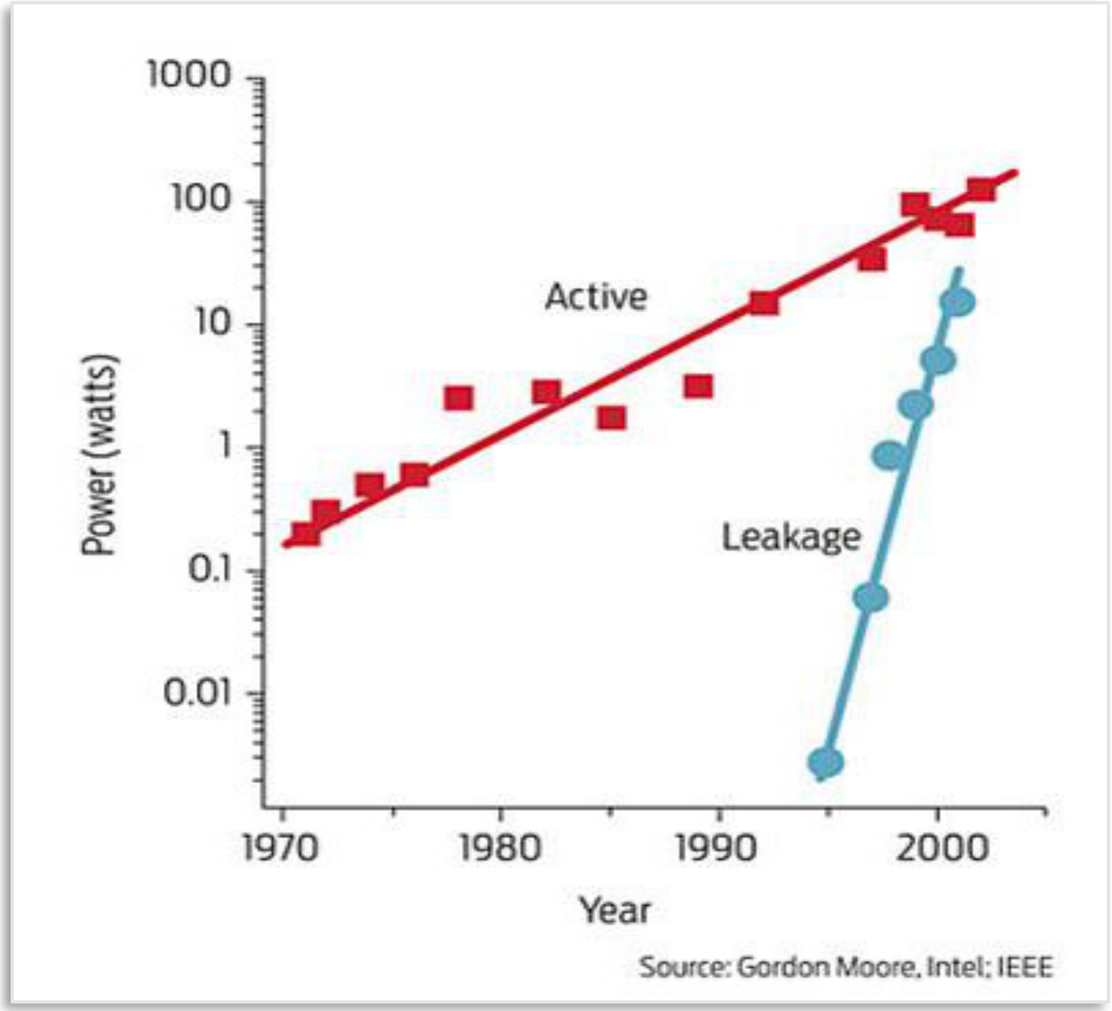
CONTENTS

CHARTING OUR WAY TO THE EXASCALE

HPC's Biggest Challenge: Power


10^7
10^6
10^5
10^4
10^3
10^2
10^1
10^0
1975 1980 1985 1990 1995 2000 2005 2010 2015
Transistors (thousands)
Single-thread Performance (SpecINT)
Frequency (MHz)
Typical Power (Watts)
Number of Cores
Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore

THE RISE OF LEAKAGE



LEAKAGE IS KILLING VOLTAGE SCALING

The Good Old Days
Leakage not important,
voltage scaled with feature size

$L' = L/2$
$V' = V/2$
$E' = CV^2 = E/8$
$f' = 2f$
$D' = 1/L^2 = 4D$
$P' = P$

L/2 => 4x transistors
=> 8x capability
=> same power

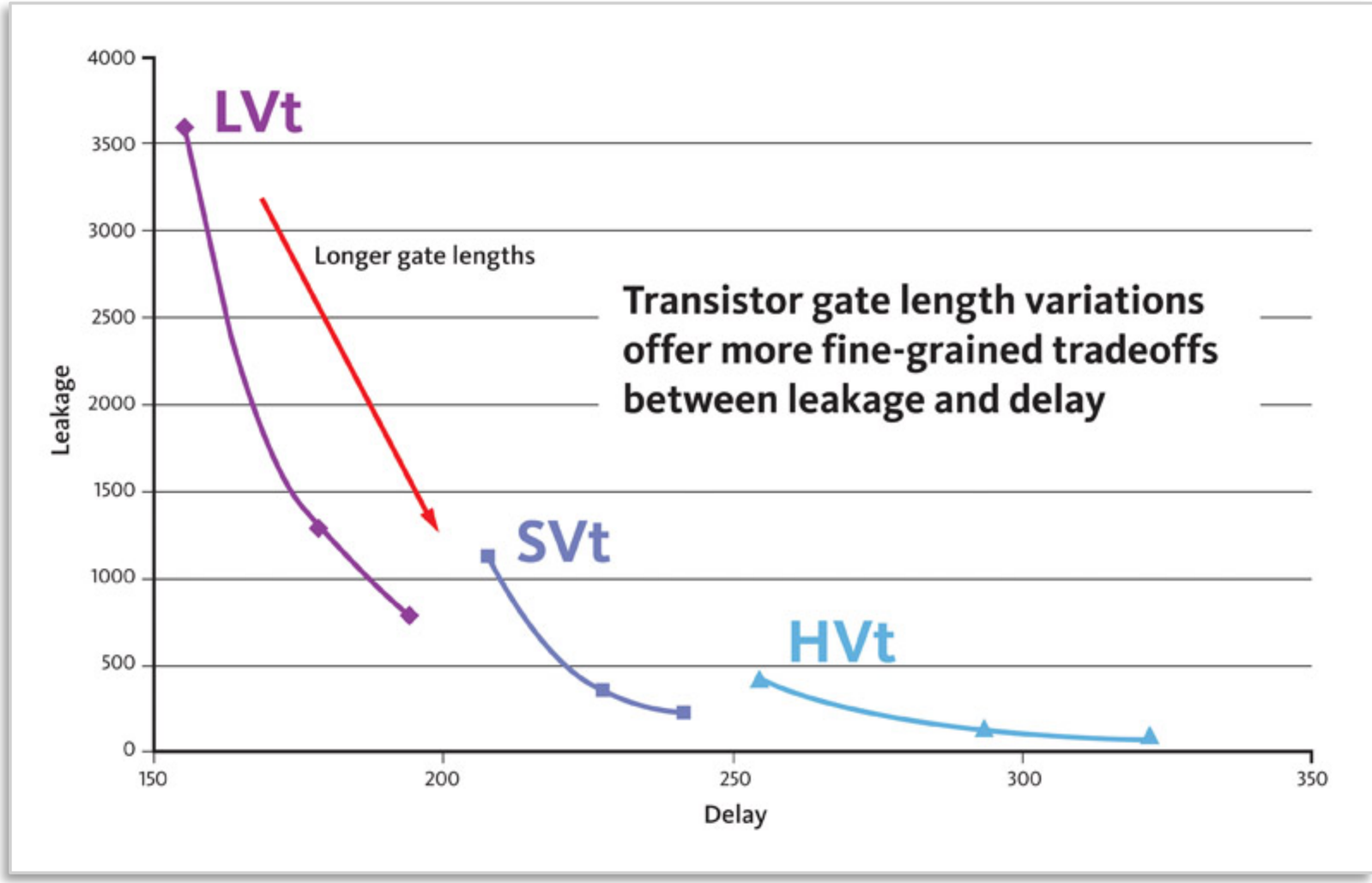
The New Reality
Leakage limiting threshold voltage,
voltage scaling dying

$L' = L/2$
$V' = \sim V$
$E' = CV^2 = E/2$
$f' = 2f$
$D' = 1/L2 = 4D$
$P' = 4P$

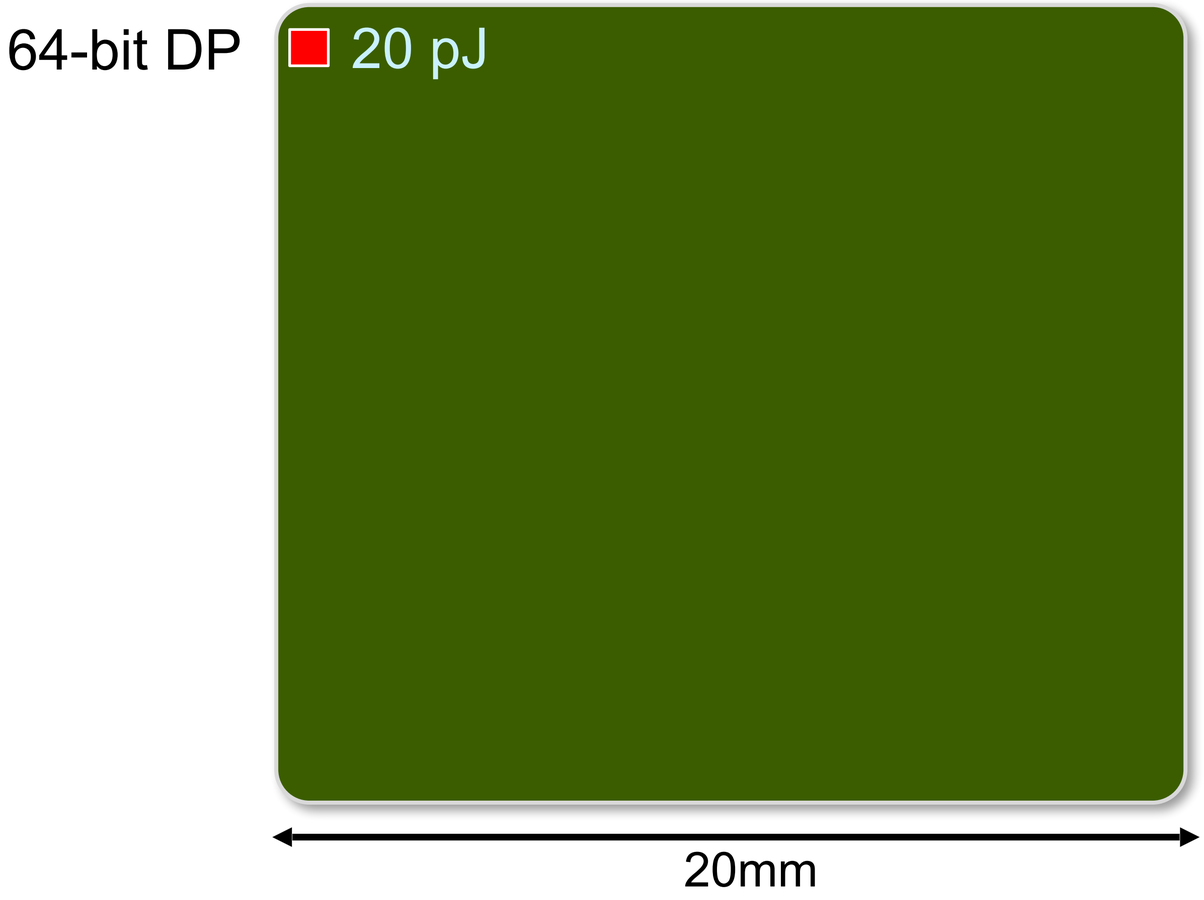
L/2 => 4x transistors
=> 8x capability
=> 4x power, or…

2x capability,
same power, ¼ area

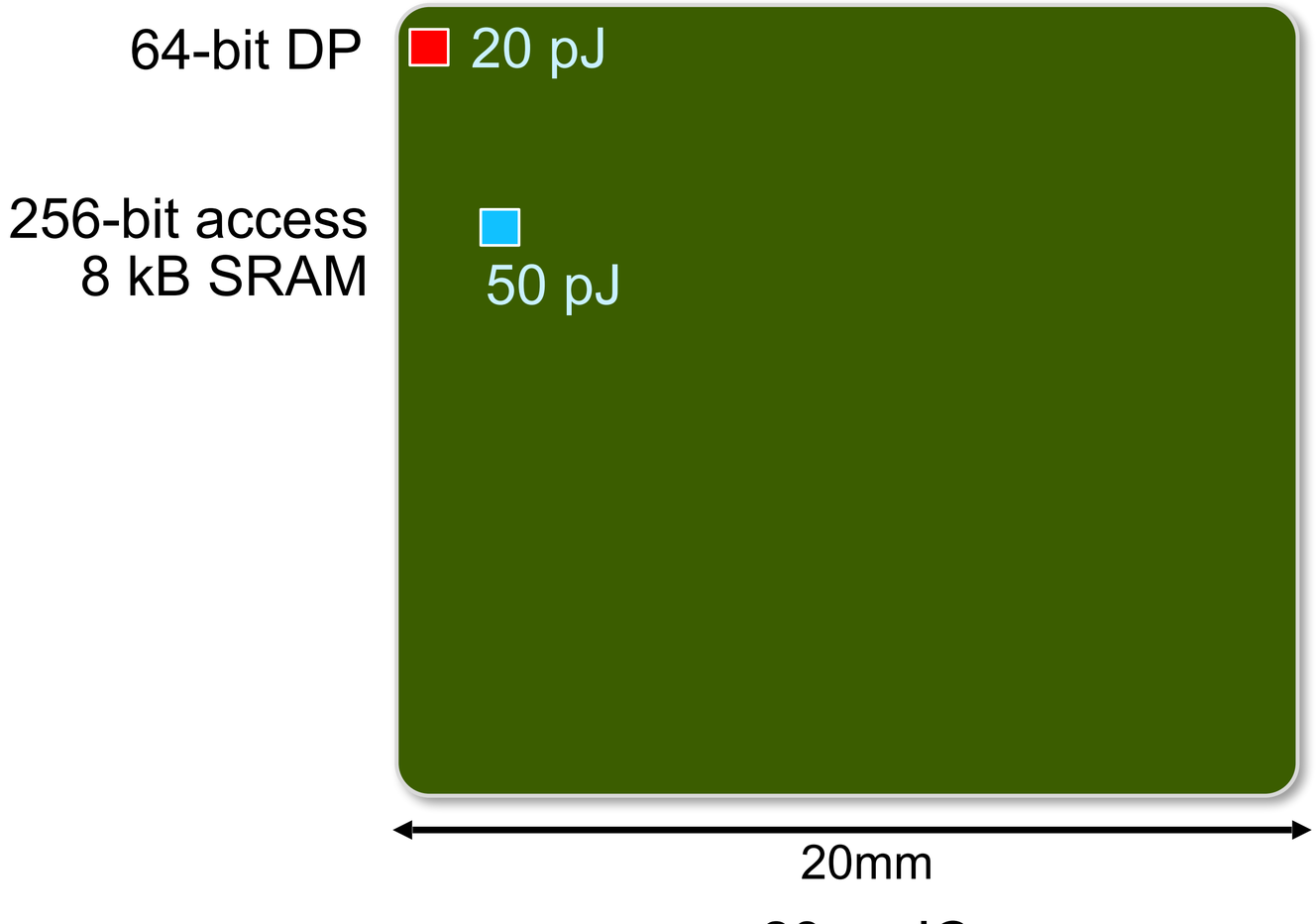
LEAKAGE SOLUTION: GO SLOW



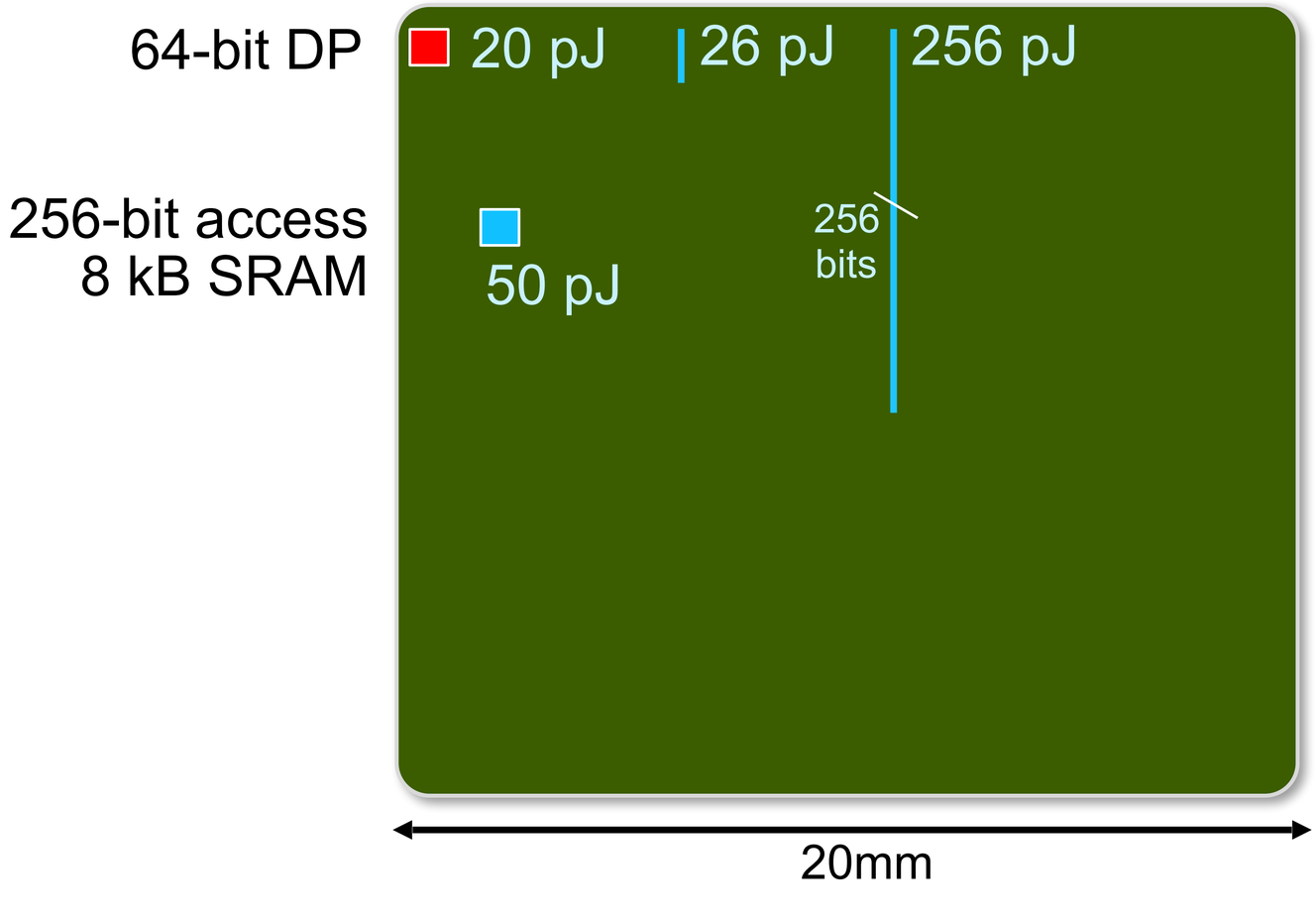
ENERGY EFFICIENCY DRIVES LOCALITY



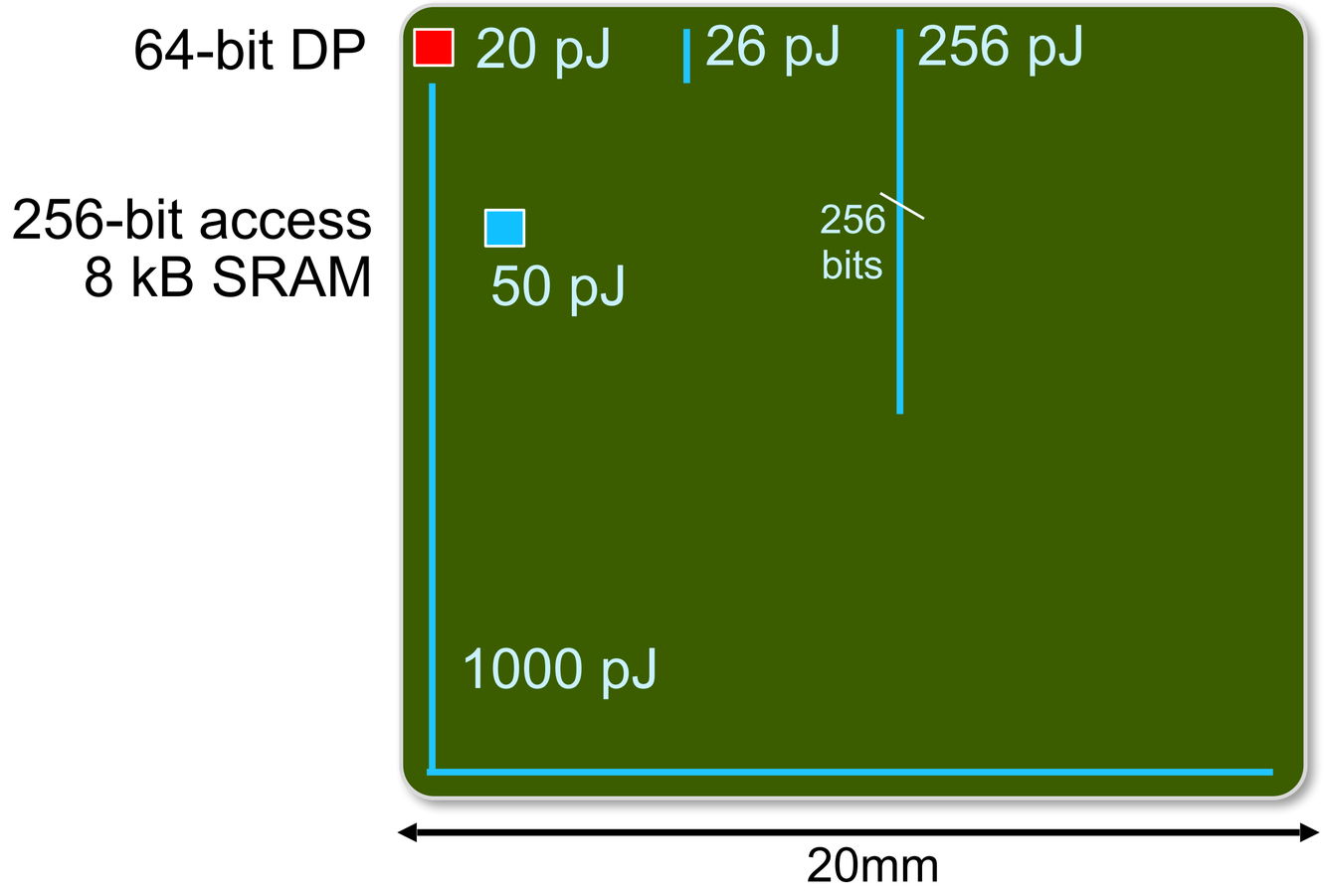
ENERGY EFFICIENCY DRIVES LOCALITY



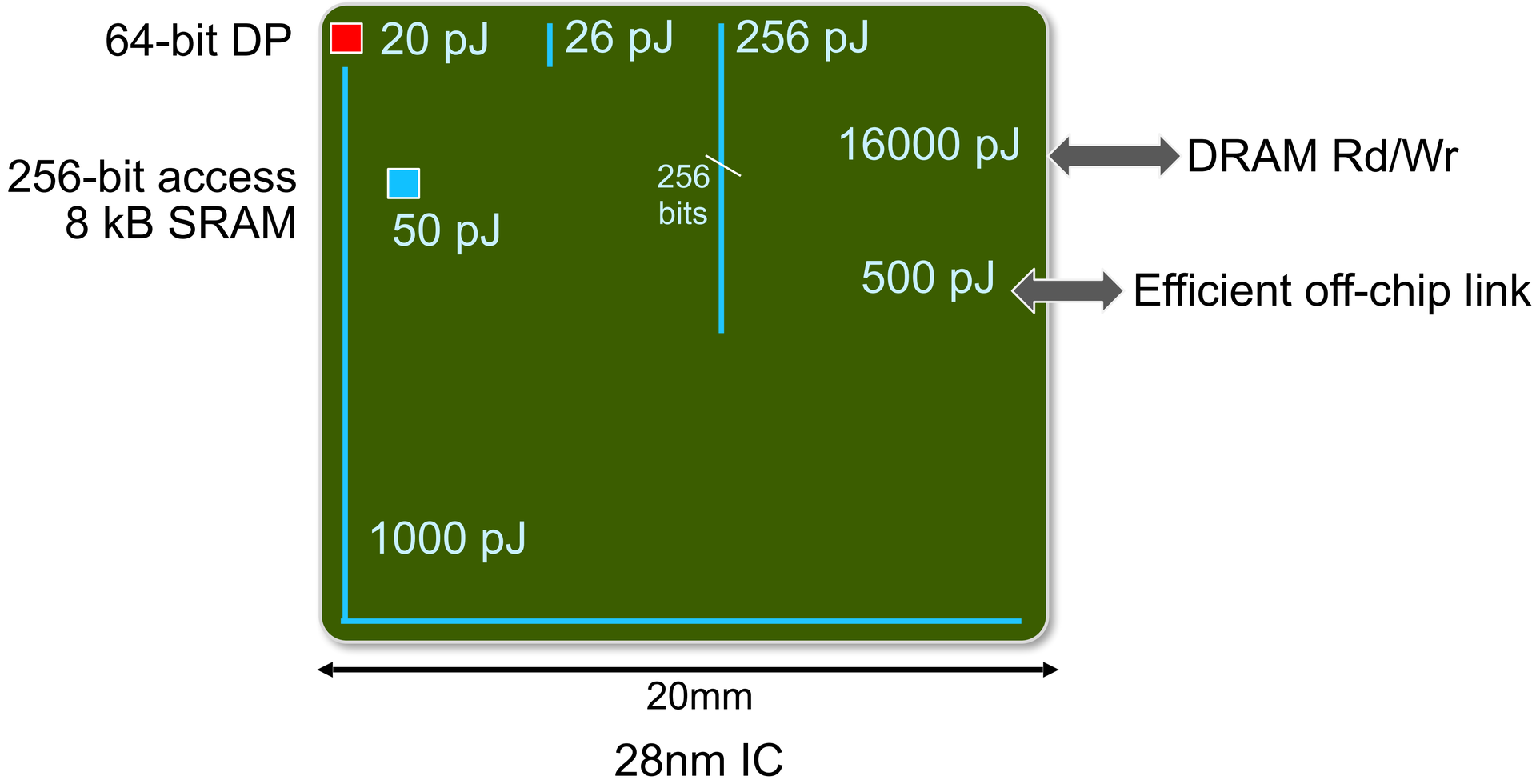
ENERGY EFFICIENCY DRIVES LOCALITY



ENERGY EFFICIENCY DRIVES LOCALITY



ENERGY EFFICIENCY DRIVES LOCALITY

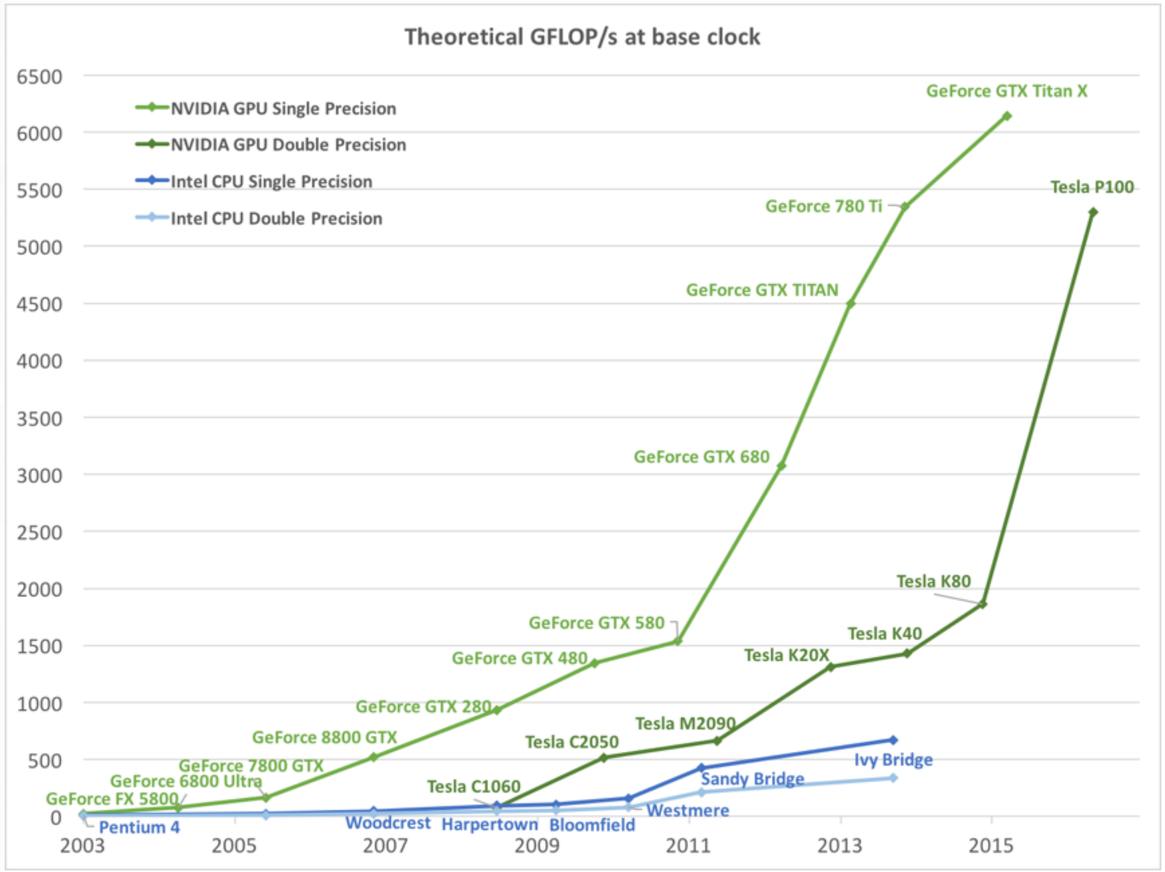


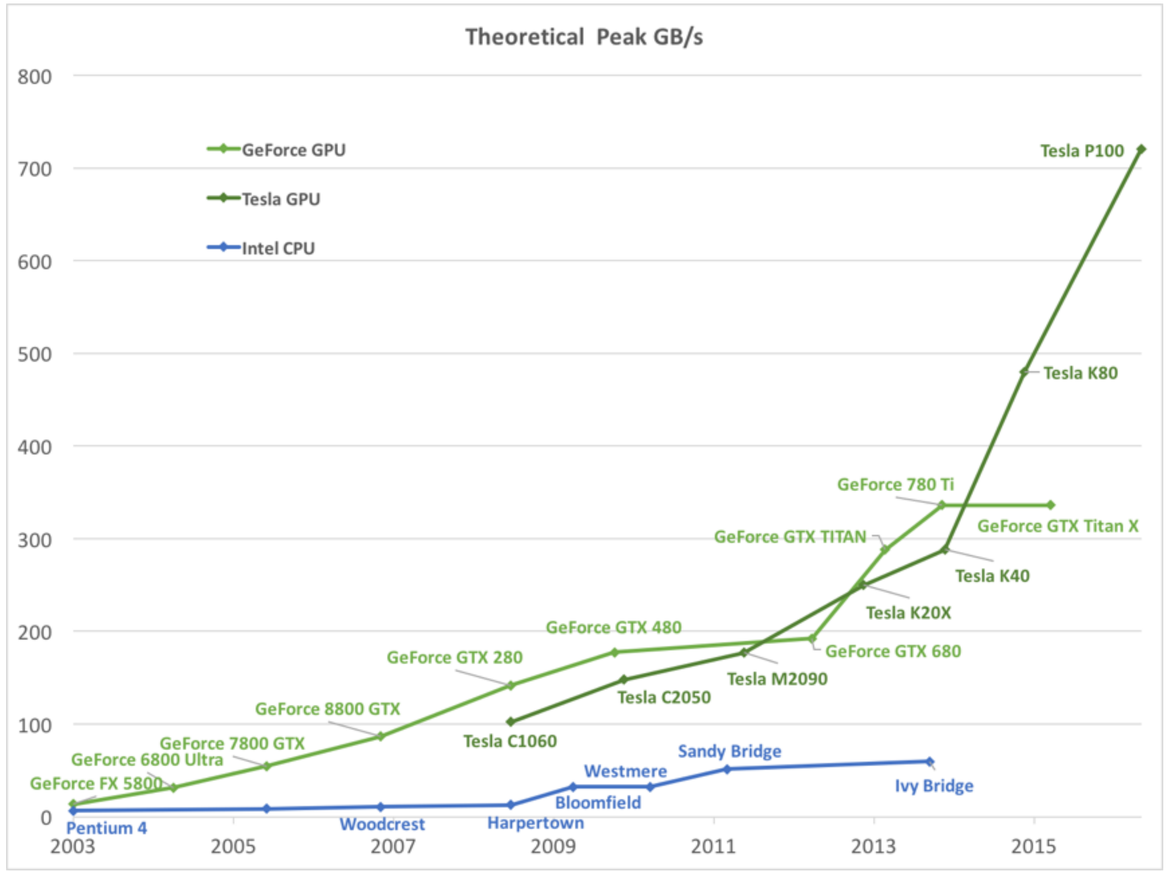
SOFTWARE IMPLICATIONS

- Need to expose massive concurrence
  - Exaflop at O(GHz) clocks ⇒ O(billion-way) parallelism!
- Need to expose and exploit locality
  - Data *motion* more expensive than computation
  - > 100:1 global v. local energy

GPUS

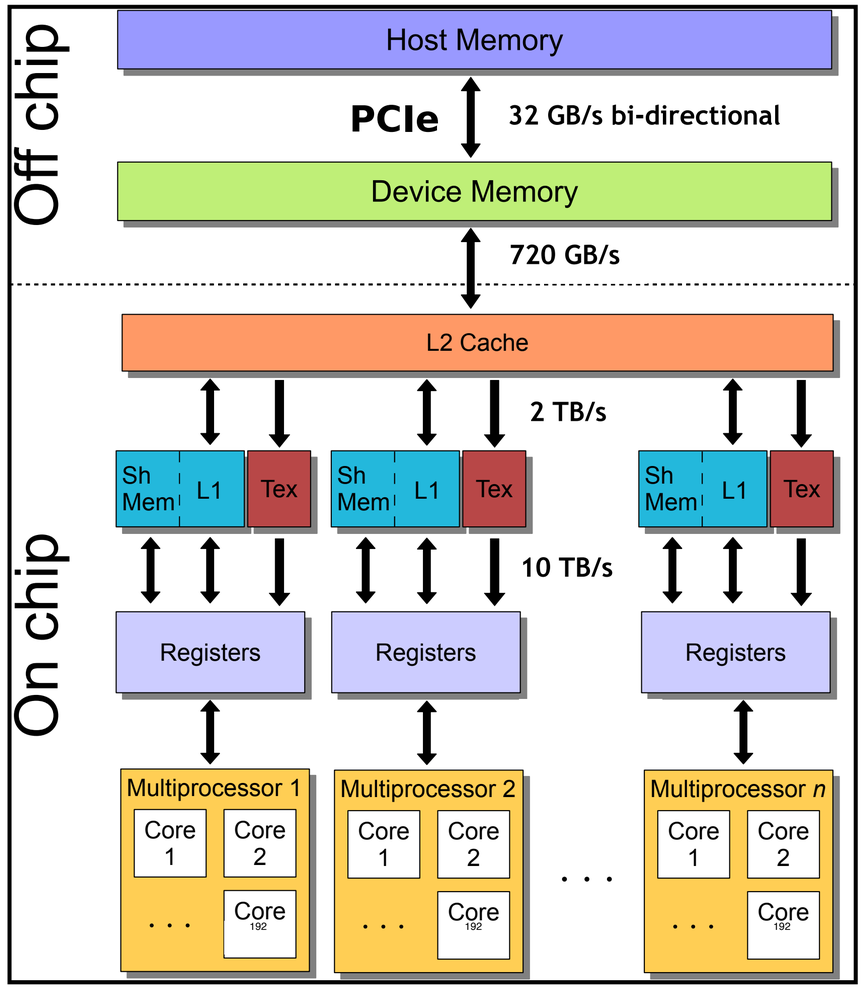
THE RISE OF THE GPU





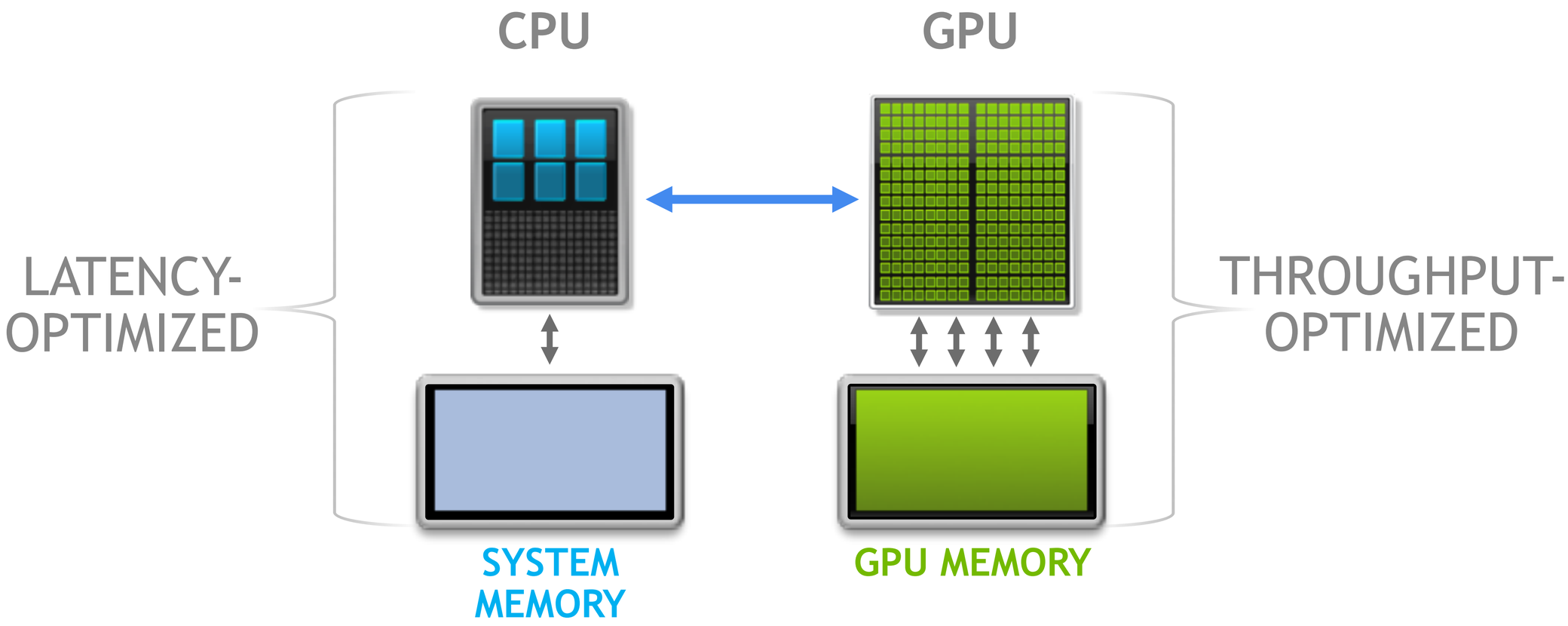
WHAT IS A GPU?

- Pascal P100 (2016)
  - 3584 FP32 processing elements
  - 10.6 TFLOPS FP32 / 5.3 TFLOPS FP64 peak
- Deep memory hierarchy
- Programmed using a thread model
  - Architecture abstraction is CUDA
  - Fine-grained parallelism is required
- Diversity of programming languages
  - CUDA C / C++ / Fortran
  - C / C++ / Fortran + OpenACC / OpenMP
  - Python, Java, etc.



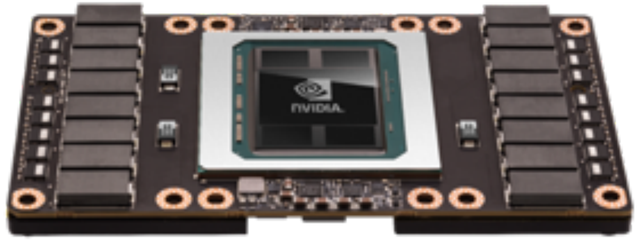
HETEROGENOUS NODE PHILOSOPHY

Optimize for Efficiency



TESLA PASCAL P100

Tesla P100
for NVLink-enabled Servers



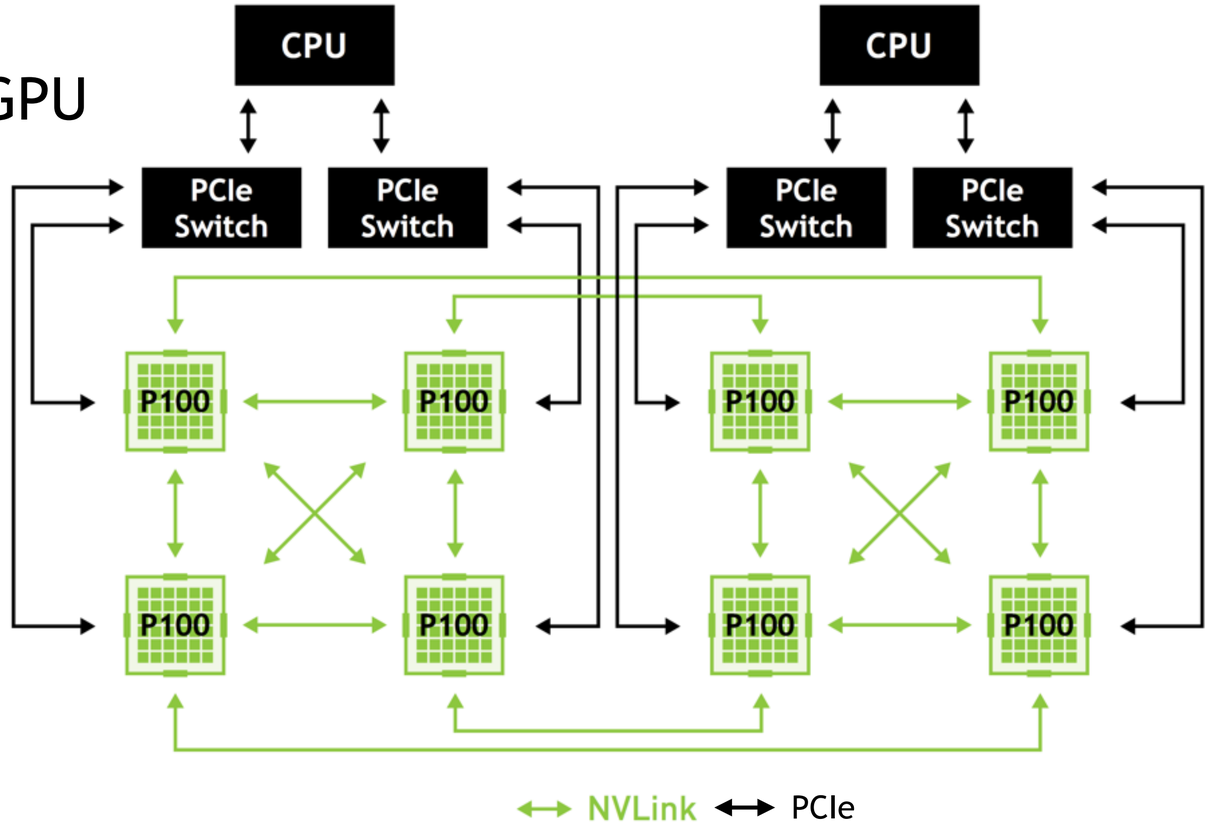
5.3 TF DP · 10.6 TF SP · 21 TF HP
720 GB/sec Memory Bandwidth
16 GB HBM2

Tesla P100
for PCIe-Based Servers

4.7 TF DP · 9.3 TF SP · 18.7 TF HP
Config 1: 16 GB (HBM2), 720 GB/sec
Config 2: 12 GB (HBM2), 540 GB/sec

DGX-1

- Pascal includes NVLink interconnect technology
  - 4x NVLink connections per GPU
  - 160 GB/s bi-directional bandwidth per GPU
- NVIDIA DGX-1
  - 8x P100 GPUs
  - 2^3 NVLink topology
  - 4x EDR IB (via PCIe switches)
  - 2x Intel Xeon hosts
- 1 DGX-1 equivalent inter-GPU bandwidth to 64 nodes of Titan



US to Build Two Flagship Supercomputers

Powered by the Tesla Platform

OAK RIDGE National Laboratory

Lawrence Livermore National Laboratory

IBM

NVIDIA

100-300 PFLOPS Peak

10x in Scientific App Performance

IBM POWER9 CPU + NVIDIA Volta GPU

NVLink High Speed Interconnect

40 TFLOPS per Node, >3,400 Nodes

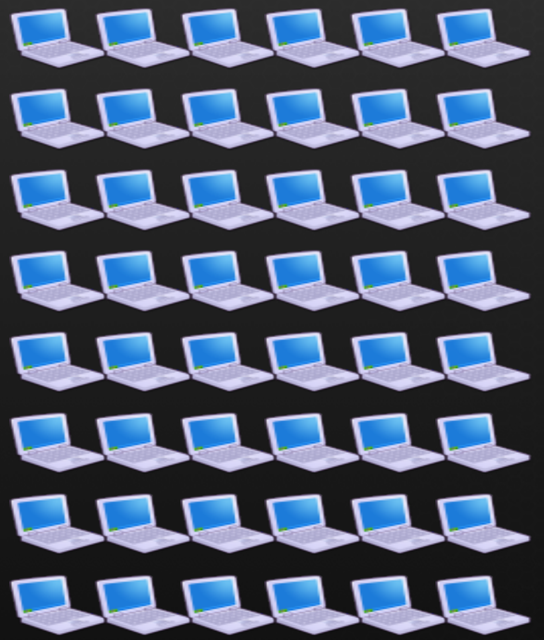
2017

Major Step Forward on the Path to Exascale

TOP500®
SUPERCOMPUTER SITES

Just 4 nodes in Summit
would make the Top500 list of supercomputers today

Similar Power as Titan
5-10x Faster
1/5th the Size



150 PF = 3M Laptops
One laptop for Every Resident in State of Mississippi

QUDA

- “QCD on CUDA” – http://lattice.github.com/quda (open source, BSD license)
- Effort started at Boston University in 2008, now in wide use as the GPU backend for BQCD, Chroma, CPS, MILC, TIFR, tmLQCD, etc.
  - Latest release 0.8.0 (8th February 2016)
- 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
  - Domain-decomposed (Schwarz) preconditioners for strong scaling
  - Eigenvector and deflated solvers (Lanczos, EigCG, GMRES-DR)
  - Multi-source solvers
  - Multigrid solvers for optimal convergence
- A research tool for how to reach the exascale

QUDA CONTRIBUTORS

(multigrid collaborators in green)

Ron Babich (NVIDIA)
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)
Dean Howarth (Rensselaer Polytechnic Institute)
Bálint Joó (Jlab)
Hyung-Jin Kim (BNL -> Samsung)
Claudio Rebbi (Boston University)
Guochun Shi (NCSA -> Google)
Mario Schröck (INFN)
Alexei Strelchenko (FNAL)
Alejandro Vaquero (INFN)
Mathias Wagner (NVIDIA)
Frank Winter (Jlab)

QUDA - LATTICE QCD ON GPUS

http://lattice.github.com/quda

lattice / quda

Watch 36 | Unstar 45 | Fork 29

Code | Issues 107 | Pull requests 2 | Wiki | Pulse | Graphs | Settings

QUDA is a library for performing calculations in lattice QCD on GPUs. http://lattice.github.com/quda — Edit

4,621 commits | 49 branches | 19 releases | 16 contributors

Branch: develop | New pull request | Create new file | Upload files | Find file | Clone or download

mathiaswagner committed on GitHub Merge pull request #487 from lattice/hotfix/checkerboard-reference ... Latest commit f3e2aa7 a day ago

include	In ColorSpinorParam, if staggered fermions then set field dimension t...	11 days ago
lib	Correctly set volumeCB for parity subset references - need to check p...	a day ago
tests	Requesting --test 1 with staggered_dslash_test now tests MdagM operator	11 days ago
.gitignore	Updates to .gitignore and renamed multigrid_benchmark to multigrid_be...	3 months ago
CMakeLists.txt	added some comments to CMakeLists.txt	3 months ago

MAPPING THE DIRAC OPERATOR TO CUDA

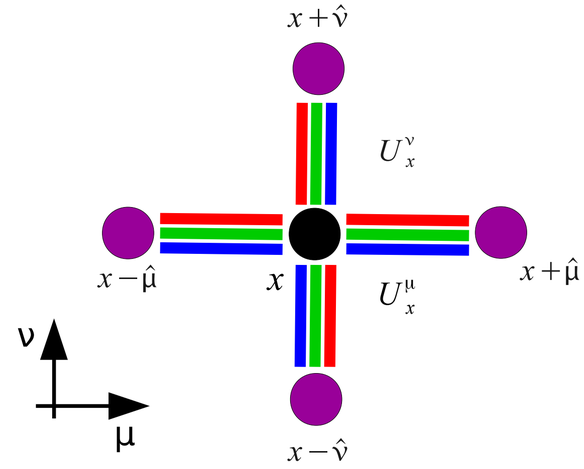
Assign a single space-time point to each thread

- V = XYZT threads, e.g., V = 24^4 => $3.3\text{x}10^6$ 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)

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



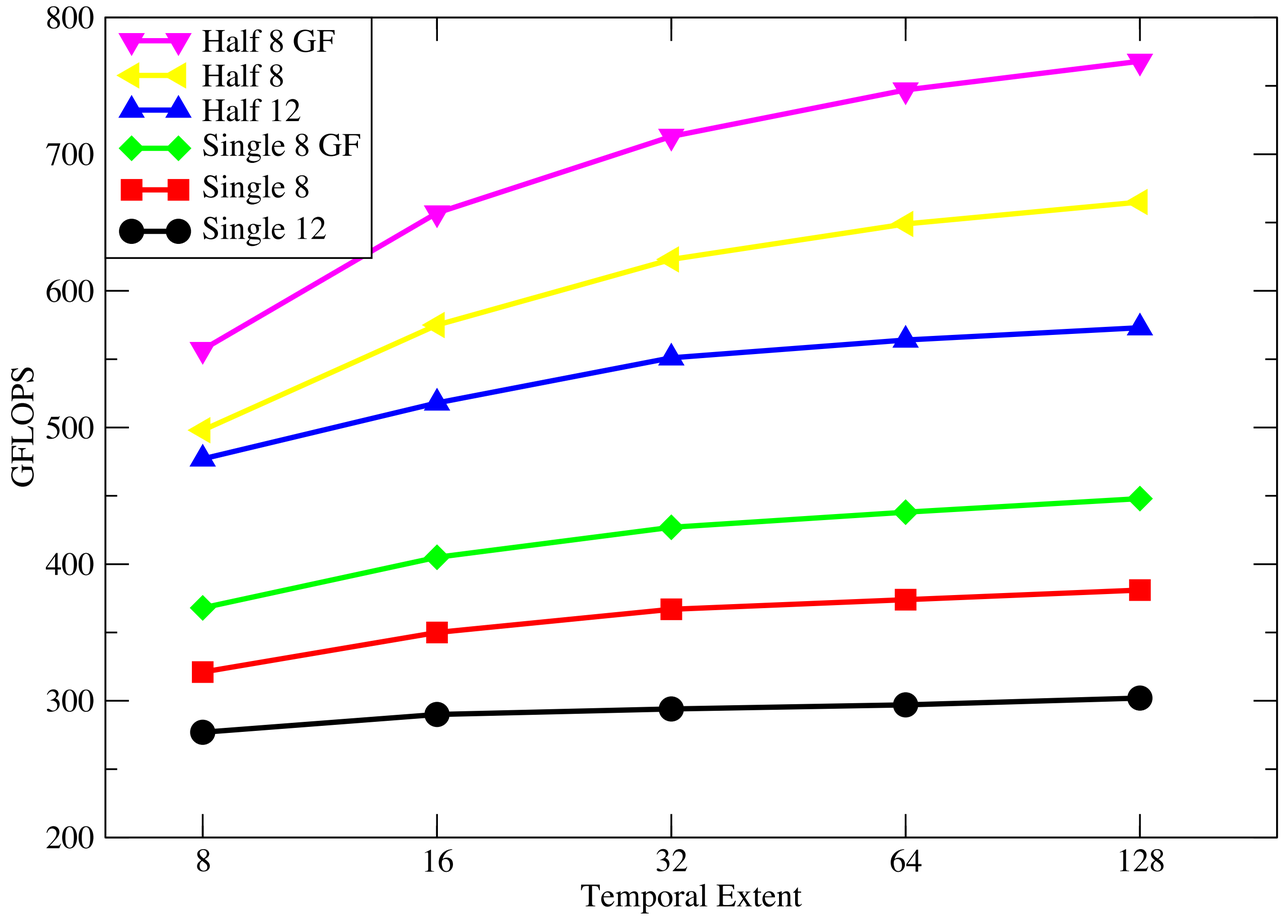
Each thread has (Wilson Dslash) 0.92 naive arithmetic intensity

QUDA reduces memory traffic

- Exact SU(3) matrix compression (18 => 12 or 8 real numbers)
- Use 16-bit fixed-point representation with mixed-precision solver

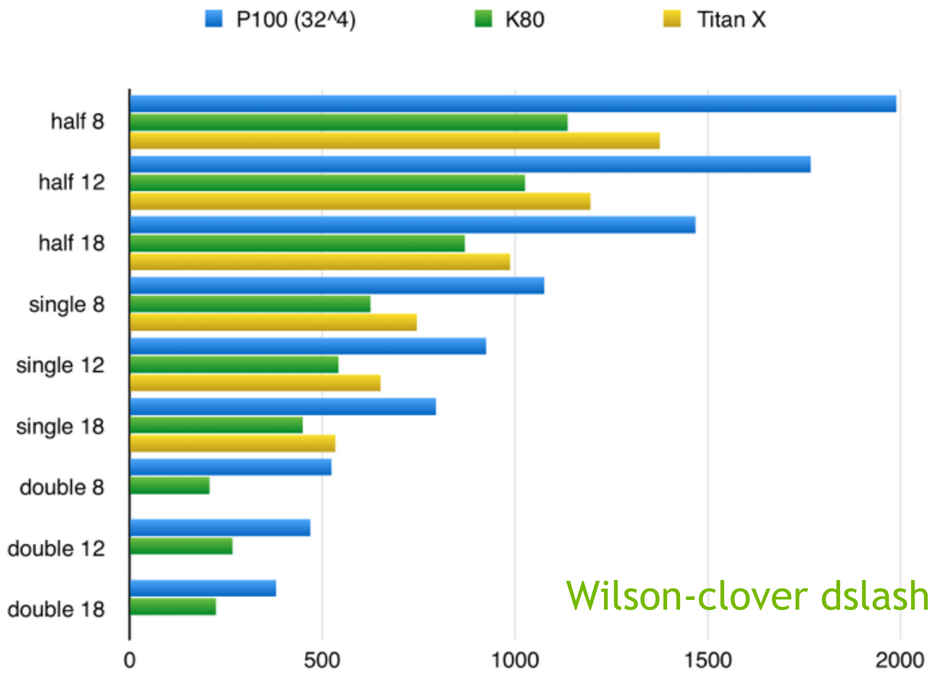
WILSON-DSLASH PERFORMANCE

K20X (2012), ECC on, V = 24^3xT

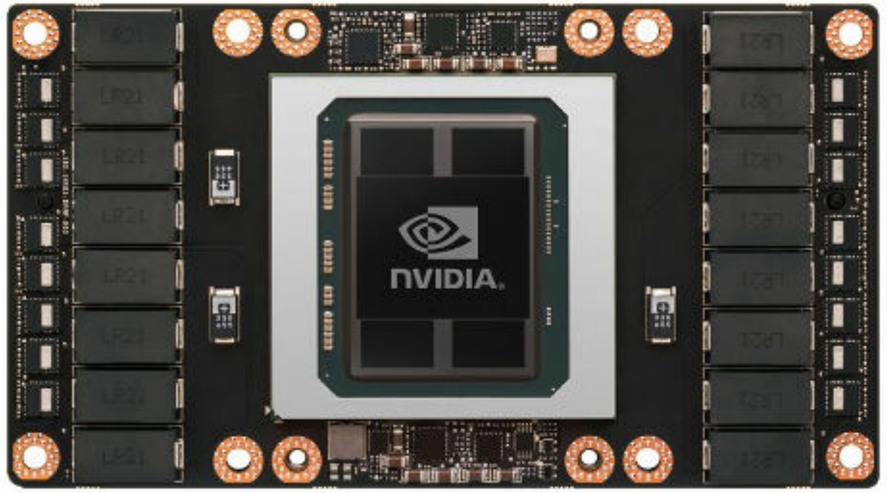


PASCAL

- Newly launched Pascal P100 GPU provides significant performance uplift through stacked HBM memory
- Wilson-clover dslash 3x speedup vs K40
  - double ~500 GFLOPS
  - single ~1000 GFLOPS
  - half ~2000 GFLOPS



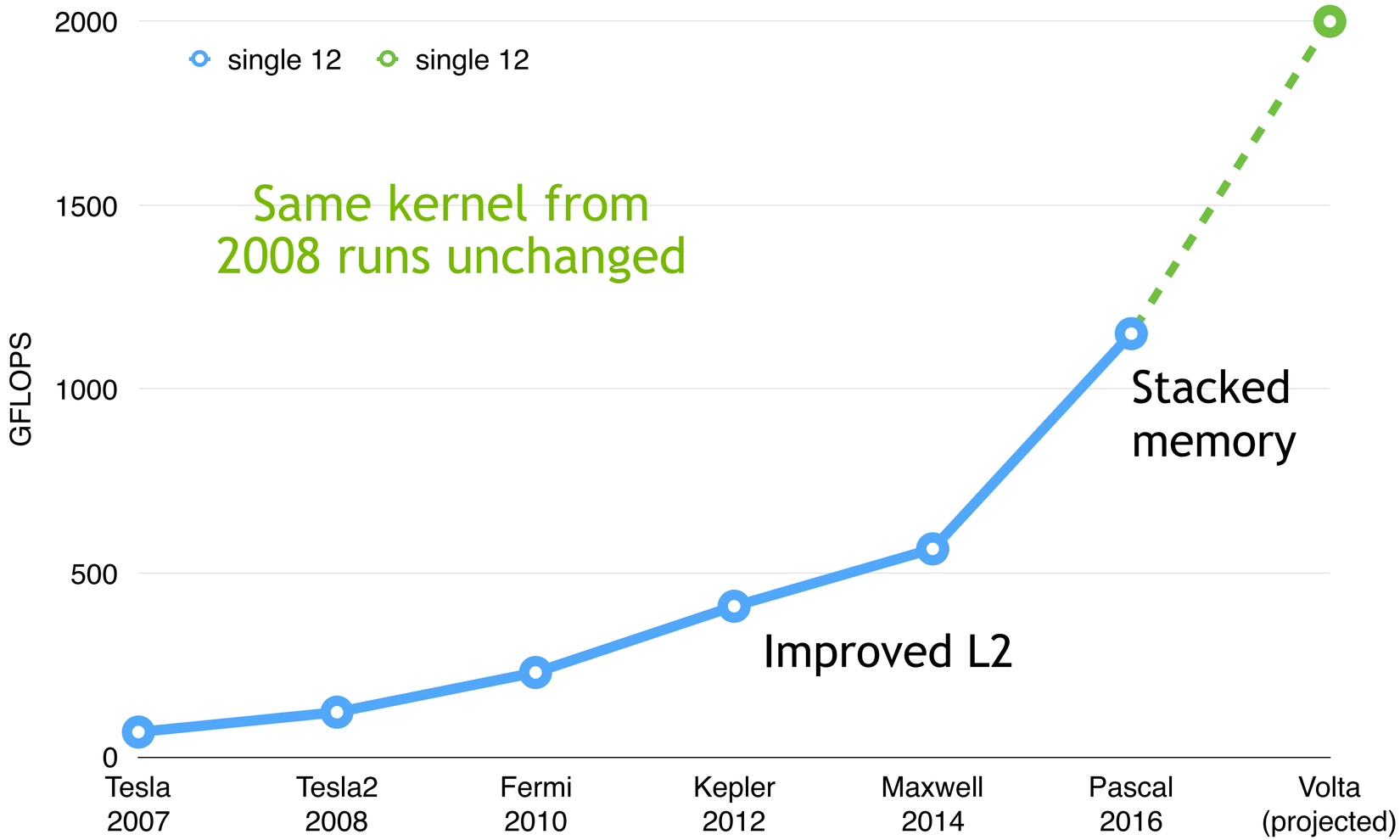
Wilson-clover dslash



P100

LQCD WITH GPU GENERATION

Single precision Wilson-dslash performance



SOLVERS FOR MULTIPLE RIGHT HAND SIDES

CONJUGATE GRADIENT

just as a reminder

procedure CG

$r^{(0)} = b - Ax^{(0)}$

$p^{(0)} = r^{(0)}$

for $k = 1, 2, \ldots$ until converged **do**

$z^{(k-1)} = Ap^{(k-1)}$

$\alpha^{(k-1)} = \frac{\left|r^{(k-1)}\right|^2}{\left\langle (p^{(k-1)}), z^{(k-1)} \right\rangle}$

$x^{(k)} = x(k-1) + \alpha^{(k-1)} p^{(k-1)}$

$r^{(k)} = r^{(k-1)} - \alpha^{(k-1)} z^{(k-1)}$

$\beta^{(k-1)} = \frac{\left|r^{(k-1)}\right|^2}{\left|r^{(k)}\right|^2}$

$p^{(k)} = r^{(k)} + \beta^{(k-1)} p^{(k-1)}$

end for

end procedure

QCD PERFORMANCE LIMITERS

QCD is **memory bandwidth bound**

Dslash arithmetic intensity for HISQ ~ 0.7

QCD PERFORMANCE LIMITERS

QCD is **memory bandwidth bound**

Dslash arithmetic intensity for HISQ ~ 0.7

exploit SU(3) symmetry:

reconstruct gauge field from 8/12 floats

QCD PERFORMANCE LIMITERS

QCD is **memory bandwidth bound**

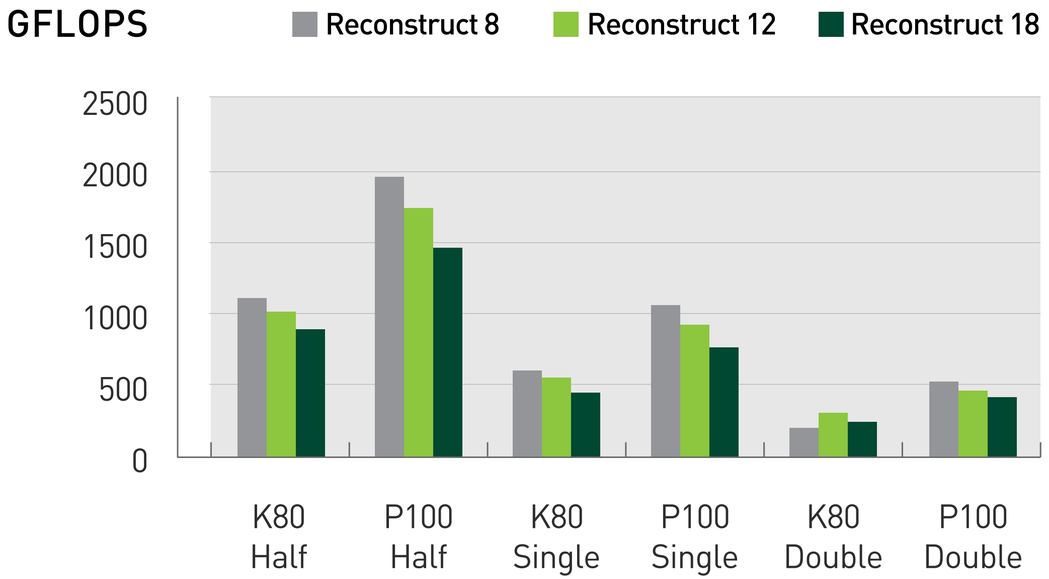
Dslash arithmetic intensity for HISQ ~ 0.7

exploit SU(3) symmetry:

reconstruct gauge field from 8/12 floats

WILSON CLOVER DSLASH

Volume = 32^4



QCD PERFORMANCE LIMITERS

QCD is **memory bandwidth bound**

Dslash arithmetic intensity for HISQ ~ 0.7

exploit SU(3) symmetry:

reconstruct gauge field from 8/12 floats

Smearing kills symmetry: stuck with 18 floats

QCD PERFORMANCE LIMITERS

QCD is **memory bandwidth bound**

Dslash arithmetic intensity for HISQ ~ 0.7

exploit SU(3) symmetry:

reconstruct gauge field from 8/12 floats

Smearing kills symmetry: stuck with 18 floats

Reuse gauge field for multiple rhs

QCD PERFORMANCE LIMITERS

QCD is **memory bandwidth bound**

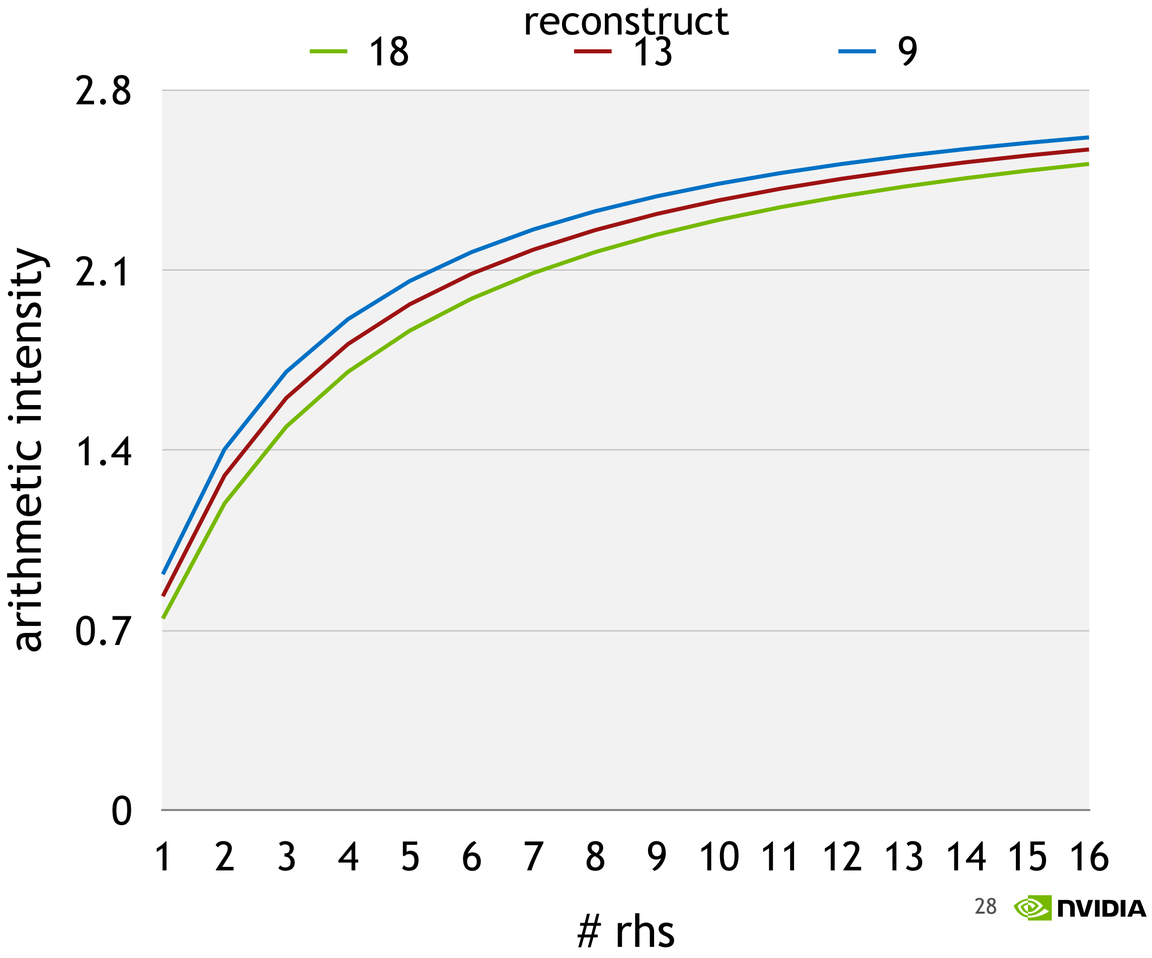
Dslash arithmetic intensity for HISQ ~ 0.7

exploit SU(3) symmetry:

reconstruct gauge field from 8/12 floats

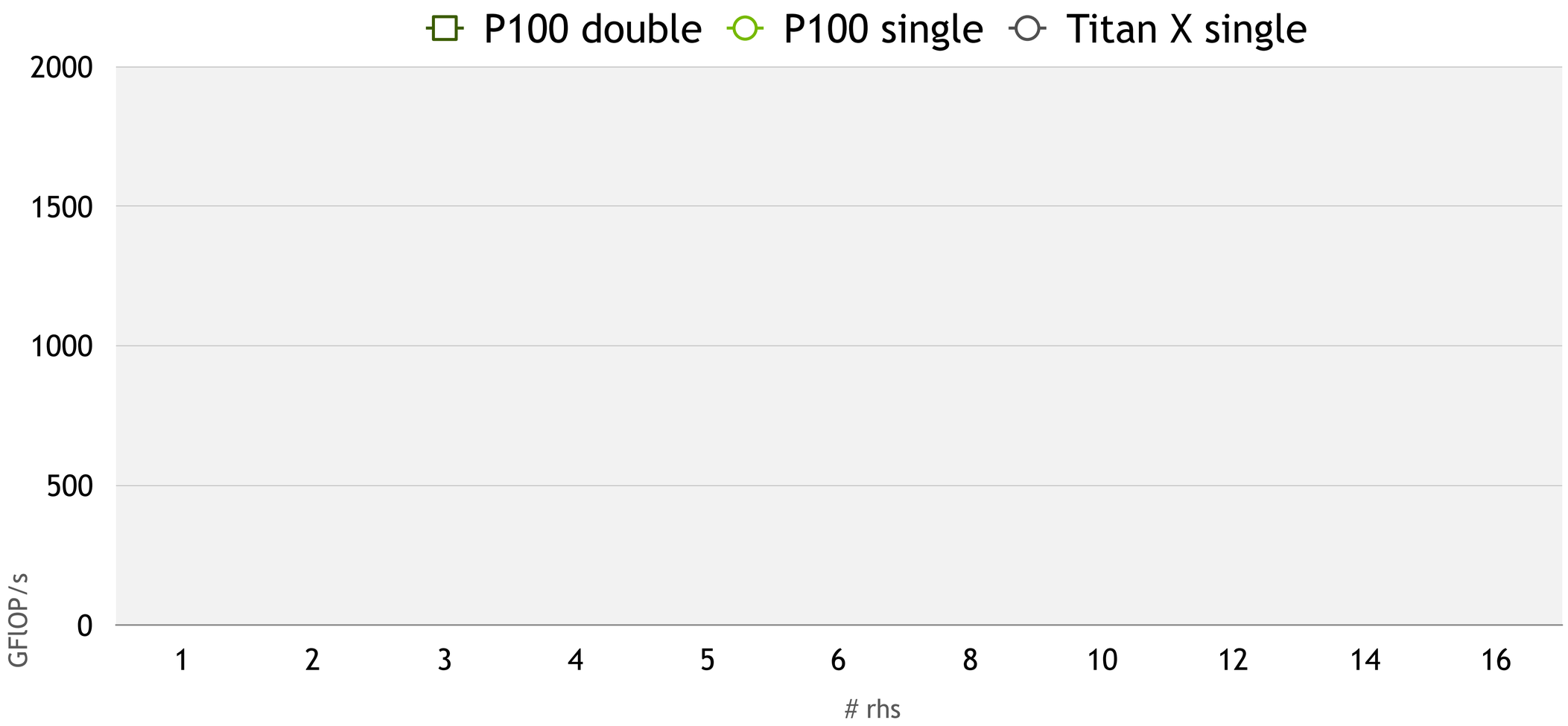
Smearing kills symmetry: stuck with 18 floats

Reuse gauge field for multiple rhs



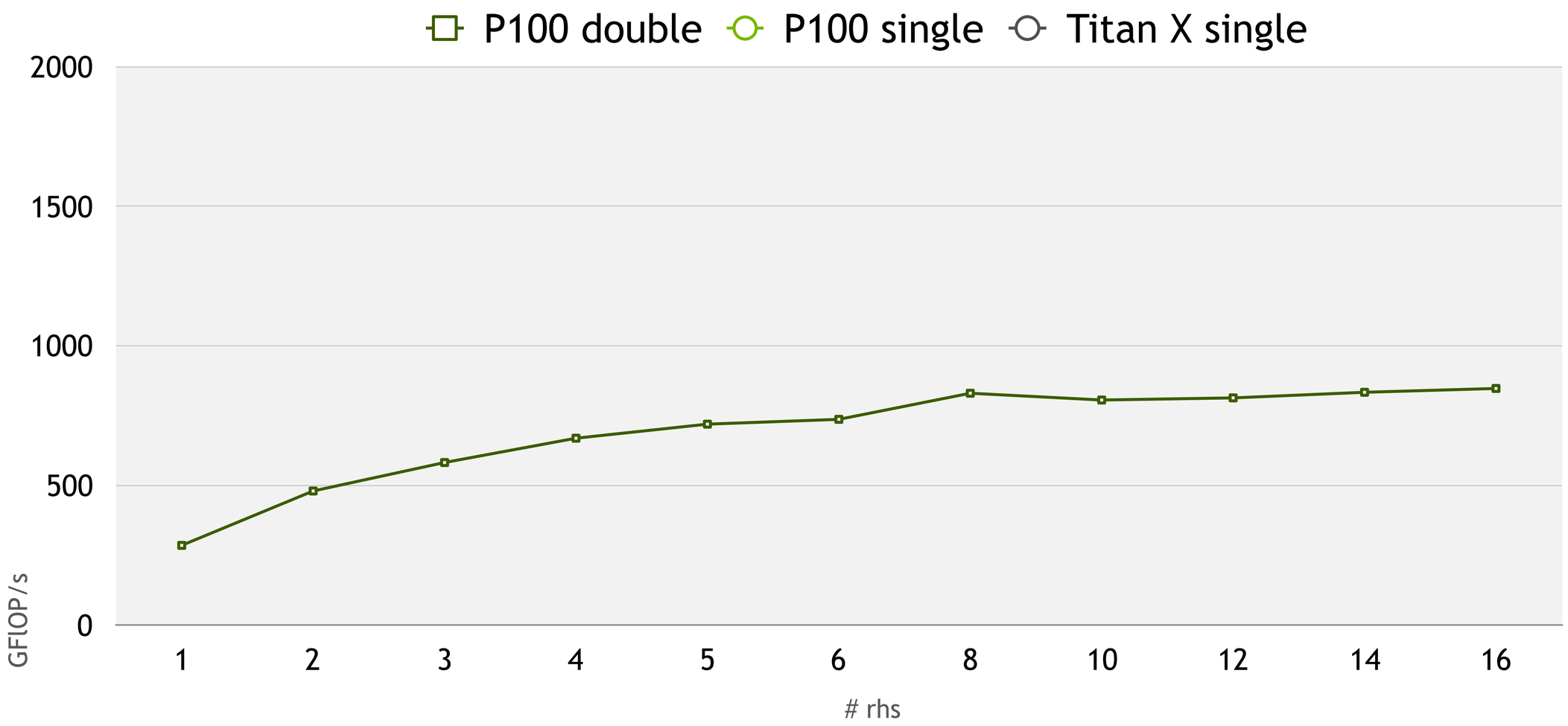
MULTI-SRC DSLASH ON PASCAL

Volume 24^4, HISQ, tuned gauge reconstruct



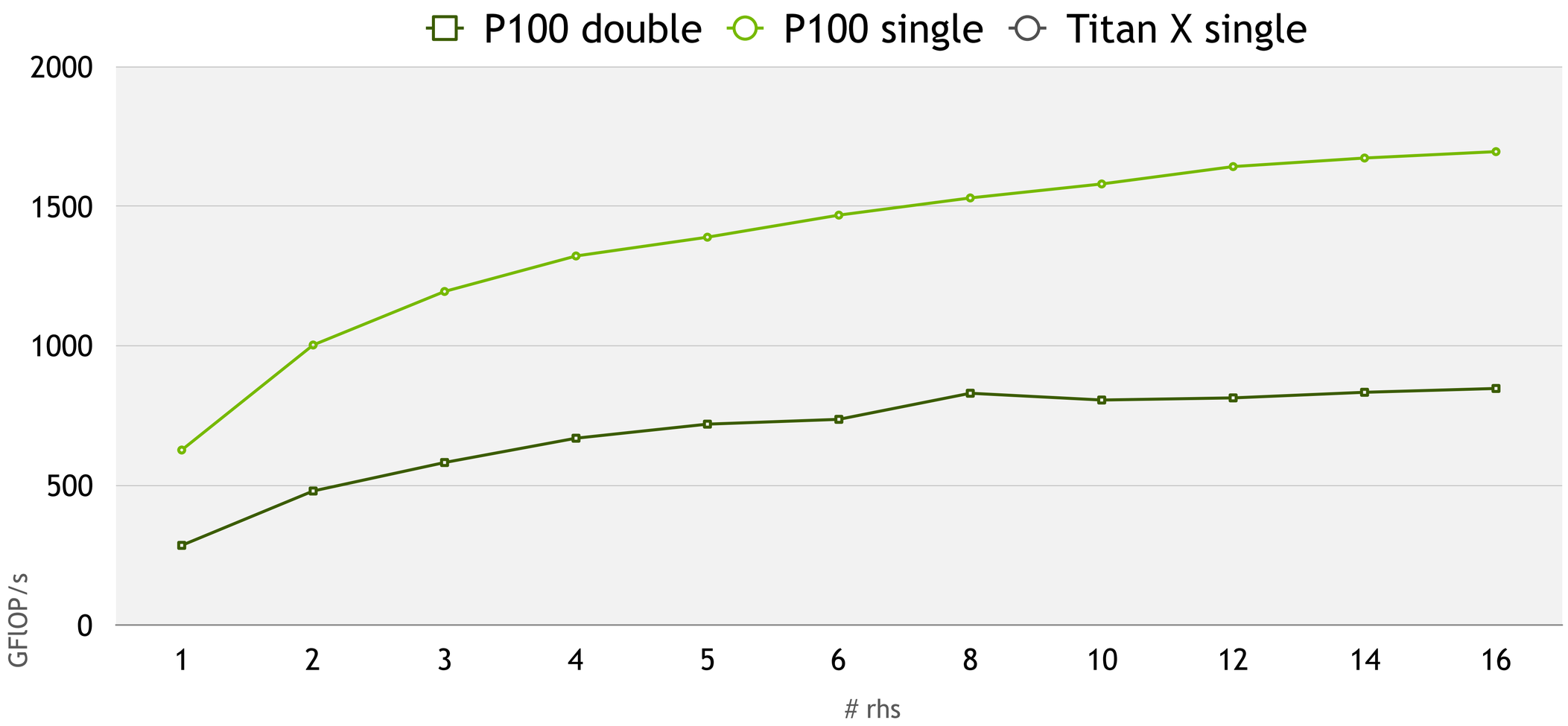
MULTI-SRC DSLASH ON PASCAL

Volume 24^4, HISQ, tuned gauge reconstruct



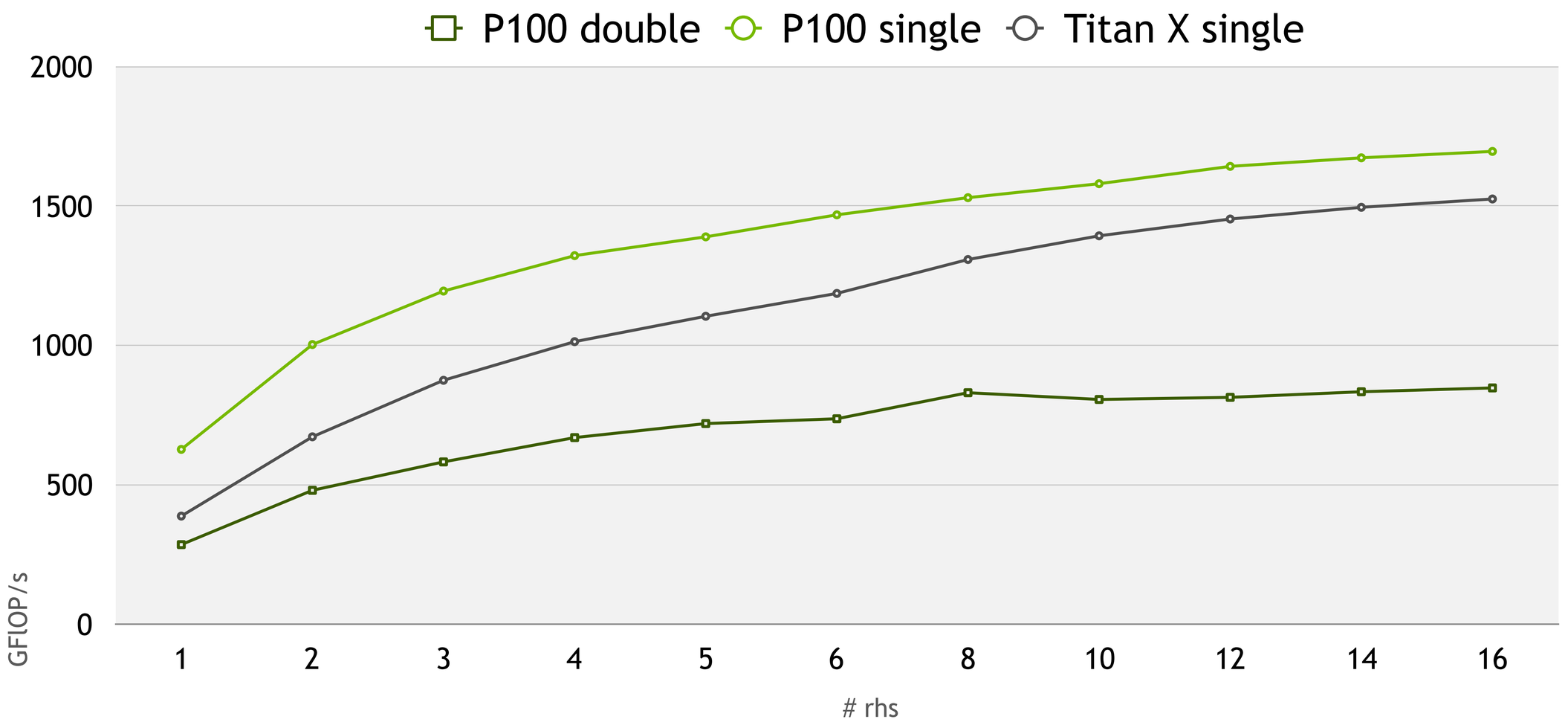
MULTI-SRC DSLASH ON PASCAL

Volume 24^4, HISQ, tuned gauge reconstruct



MULTI-SRC DSLASH ON PASCAL

Volume 24^4, HISQ, tuned gauge reconstruct



CONJUGATE GRADIENT

using multi-src Dslash

procedure CG WITH MULTI-SRC DSLASH

$r_i = b_i - Ax_i^{(0)}$

$p_i^{(0)} = r_i^{(0)}$

for $k = 1, 2, \ldots$ until converged **do**

$\left\{ z_i^{(k-1)} \right\} = A \left\{ p^{(k-1)} \right\}$

$\alpha_i^{(k-1)} = |r_i^{(k-1)}|^2 / \langle (p_i^{(k-1)}), z_i^{(k-1)} \rangle$

$x_i^{(k)} = x_i^{(k-1)} + \alpha_i^{(k-1)} p_i^{(k-1)}$

$r_i^{(k)} = r_i^{(k-1)} - \alpha_i^{(k-1)} z_i^{(k-1)}$

$\beta_i^{(k-1)} = |r_i^{(k-1)}|^2 / |r_i^{(k)}|^2$

$p_i^{(k)} = r_i^{(k)} + \beta_i^{(k-1)} p_i^{(k-1)}$

end for

end procedure

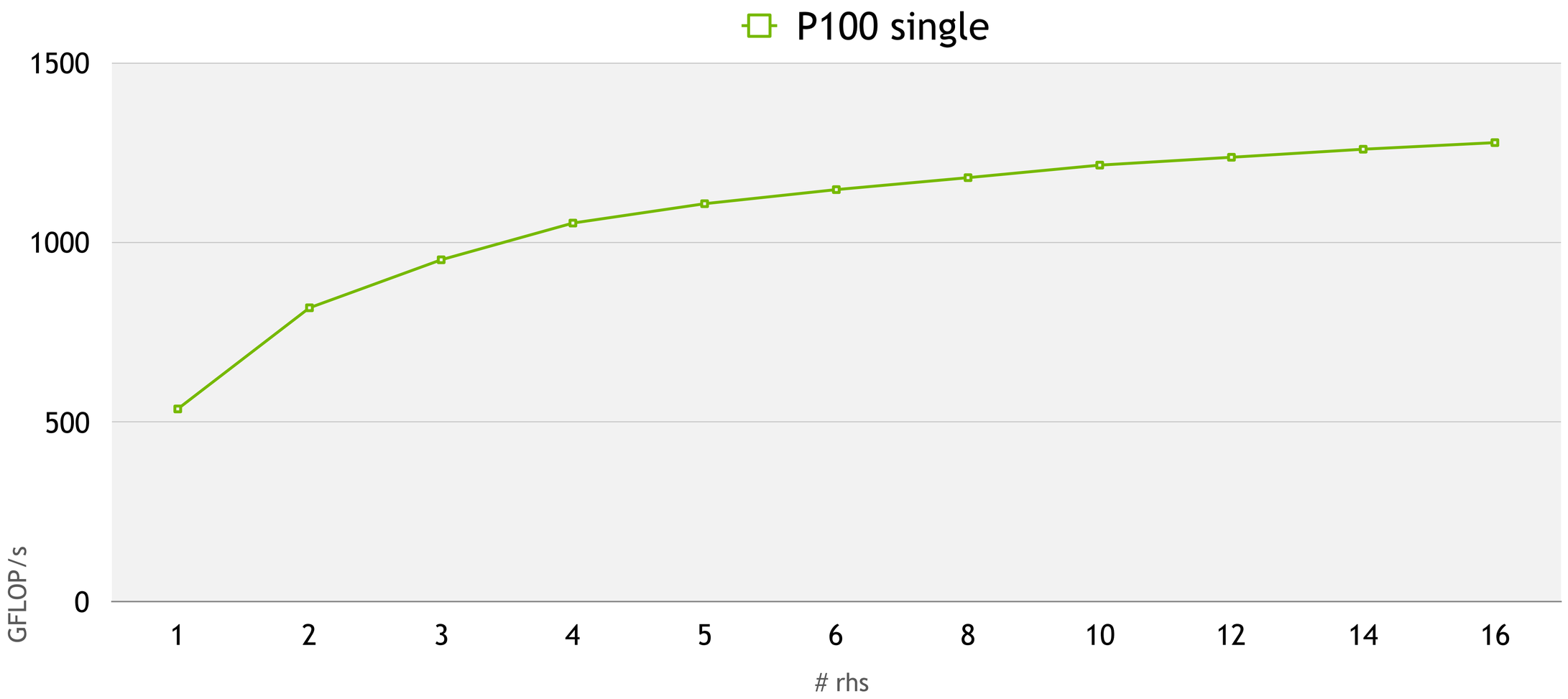
exploit multi-src Dslash performance

do all the linear algebra for each rhs

same iteration count as CG

MULTI-SRC CG ON PASCAL

Volume 24^4, HISQ



BLOCK KRYLOV SOLVERS

Share the Krylov space

BlockCG solver suggested by O'Leary in early 80's

retooled BlockCG by Dubrulle 2001

In exact precision converges in N / rhs iterations

BLOCK KRYLOV SOLVERS

Share the Krylov space

BlockCG solver suggested by O'Leary in early 80's

retooled BlockCG by Dubrulle 2001

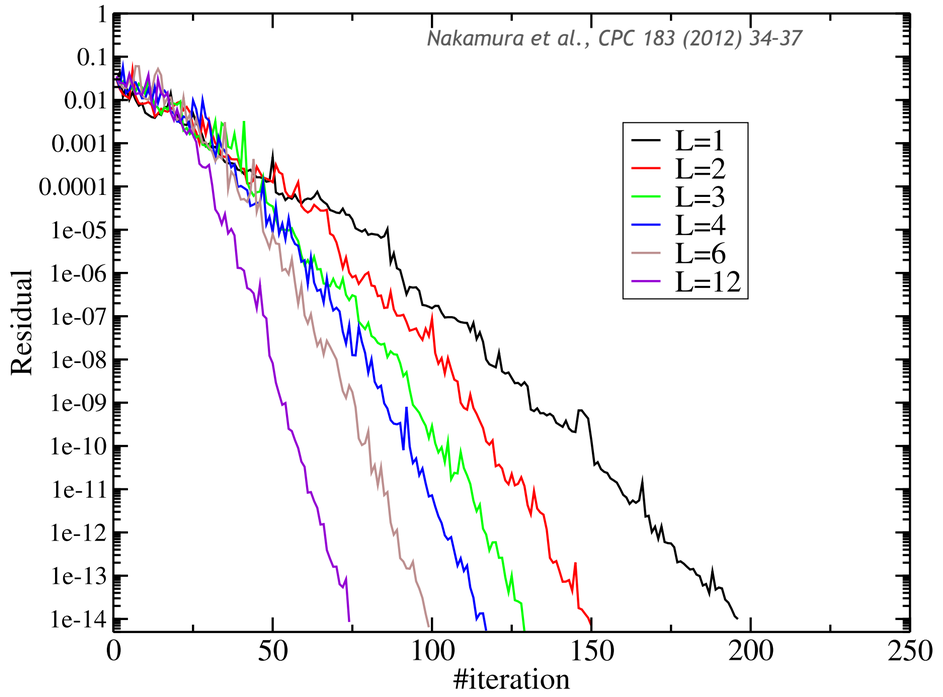
In exact precision converges in N / rhs iterations

Application in QCD:

Nakamura et. (modified block BiCGStab)

Birk and Frommer (block methods,

including block methods for multi shift)



BLOCK CG

share Krylov space between multiple rhs

procedure BLOCKCG

$R^{(0)} = B - AX^{(0)}$

$P^{(0)} = R^{(0)}$

for $k = 1, 2, \ldots$ until converged **do**

$Z^{(k-1)} = AP^{(k-1)}$

$\alpha^{(k-1)} = \left[(P^{(k-1)})^H Z^{(k-1)} \right]^{-1} (R^{(k-1)})^H R^{(k-1)}$

$X^{(k)} = X^{(k-1)} + P^{(k-1)} \alpha^{(k-1)}$

$R^{(k)} = R^{(k-1)} - Z^{(k-1)} \alpha^{(k-1)}$

$\beta^{(k-1)} = \left[(R^{(k-1)})^H R^{(k-1)} \right]^{-1} (R^{(k)})^H R^{(k)}$

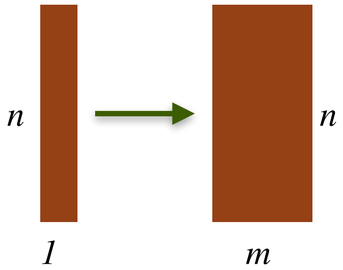
$P^{(k)} = R^{(k)} - P^{(k-1)} \beta^{(k-1)}$

end for

end procedure

BLOCK CG

share Krylov space between multiple rhs



procedure BLOCKCG

$$R^{(0)} = B - AX^{(0)}$$
$$P^{(0)} = R^{(0)}$$

for $k = 1, 2, \ldots$ until converged **do**

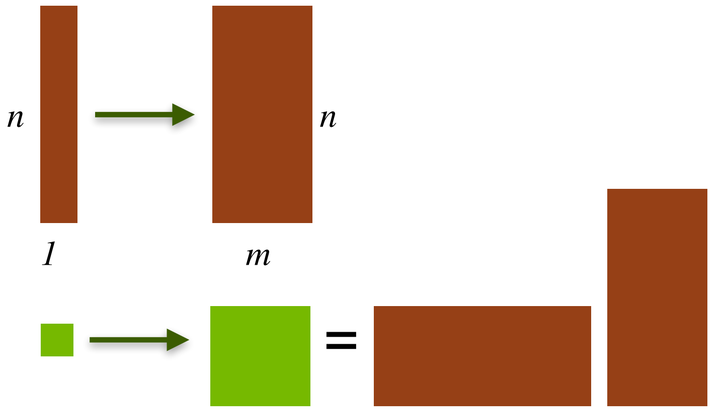
$$Z^{(k-1)} = AP^{(k-1)}$$
$$\alpha^{(k-1)} = \left[(P^{(k-1)})^H Z^{(k-1)} \right]^{-1} (R^{(k-1)})^H R^{(k-1)}$$
$$X^{(k)} = X^{(k-1)} + P^{(k-1)} \alpha^{(k-1)}$$
$$R^{(k)} = R^{(k-1)} - Z^{(k-1)} \alpha^{(k-1)}$$
$$\beta^{(k-1)} = \left[(R^{(k-1)})^H R^{(k-1)} \right]^{-1} (R^{(k)})^H R^{(k)}$$
$$P^{(k)} = R^{(k)} - P^{(k-1)} \beta^{(k-1)}$$

end for

end procedure

BLOCK CG

share Krylov space between multiple rhs



procedure BLOCKCG

$$R^{(0)} = B - AX^{(0)}$$
$$P^{(0)} = R^{(0)}$$

for $k = 1, 2, \ldots$ until converged **do**

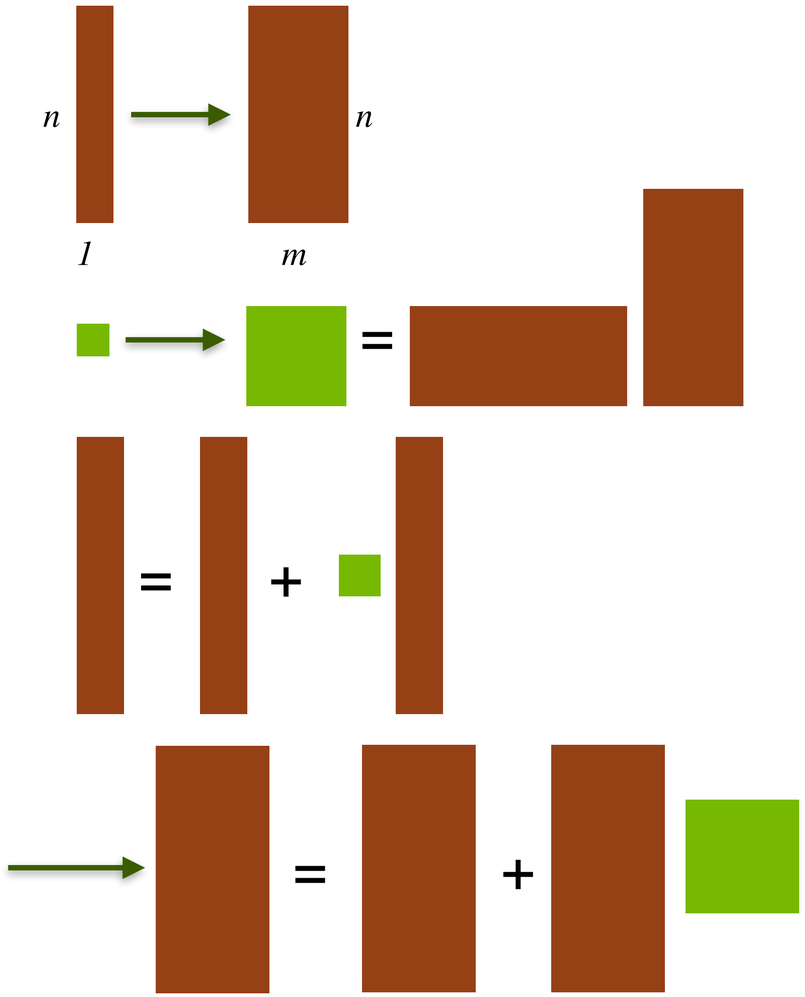
$$Z^{(k-1)} = AP^{(k-1)}$$
$$\alpha^{(k-1)} = \left[(P^{(k-1)})^H Z^{(k-1)} \right]^{-1} (R^{(k-1)})^H R^{(k-1)}$$
$$X^{(k)} = X^{(k-1)} + P^{(k-1)} \alpha^{(k-1)}$$
$$R^{(k)} = R^{(k-1)} - Z^{(k-1)} \alpha^{(k-1)}$$
$$\beta^{(k-1)} = \left[(R^{(k-1)})^H R^{(k-1)} \right]^{-1} (R^{(k)})^H R^{(k)}$$
$$P^{(k)} = R^{(k)} - P^{(k-1)} \beta^{(k-1)}$$

end for

end procedure

BLOCK CG

share Krylov space between multiple rhs



procedure BLOCKCG

$R^{(0)} = B - AX^{(0)}$

$P^{(0)} = R^{(0)}$

for $k = 1, 2, \ldots$ until converged **do**

$Z^{(k-1)} = AP^{(k-1)}$

$\alpha^{(k-1)} = \left[(P^{(k-1)})^H Z^{(k-1)} \right]^{-1} (R^{(k-1)})^H R^{(k-1)}$

$X^{(k)} = X^{(k-1)} + P^{(k-1)} \alpha^{(k-1)}$

$R^{(k)} = R^{(k-1)} - Z^{(k-1)} \alpha^{(k-1)}$

$\beta^{(k-1)} = \left[(R^{(k-1)})^H R^{(k-1)} \right]^{-1} (R^{(k)})^H R^{(k)}$

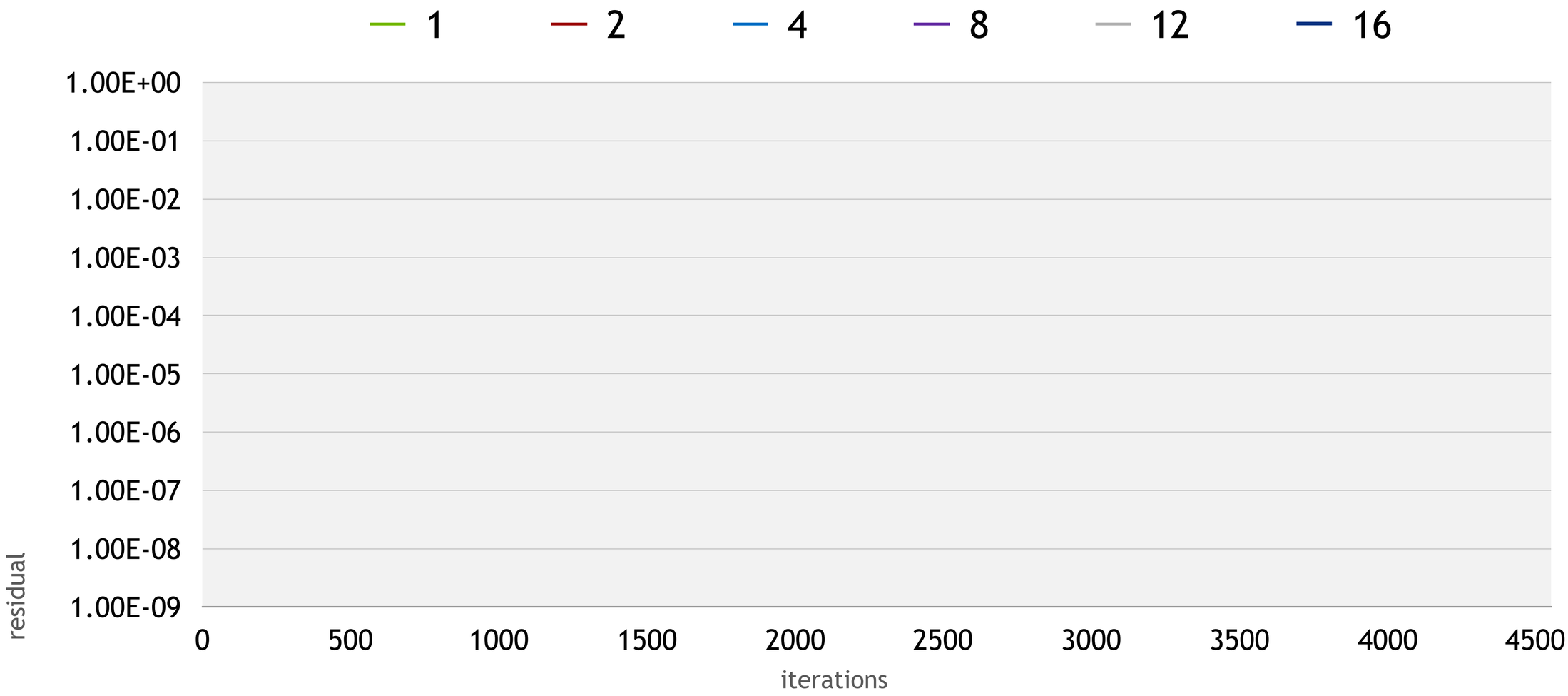
$P^{(k)} = R^{(k)} - P^{(k-1)} \beta^{(k-1)}$

end for

end procedure

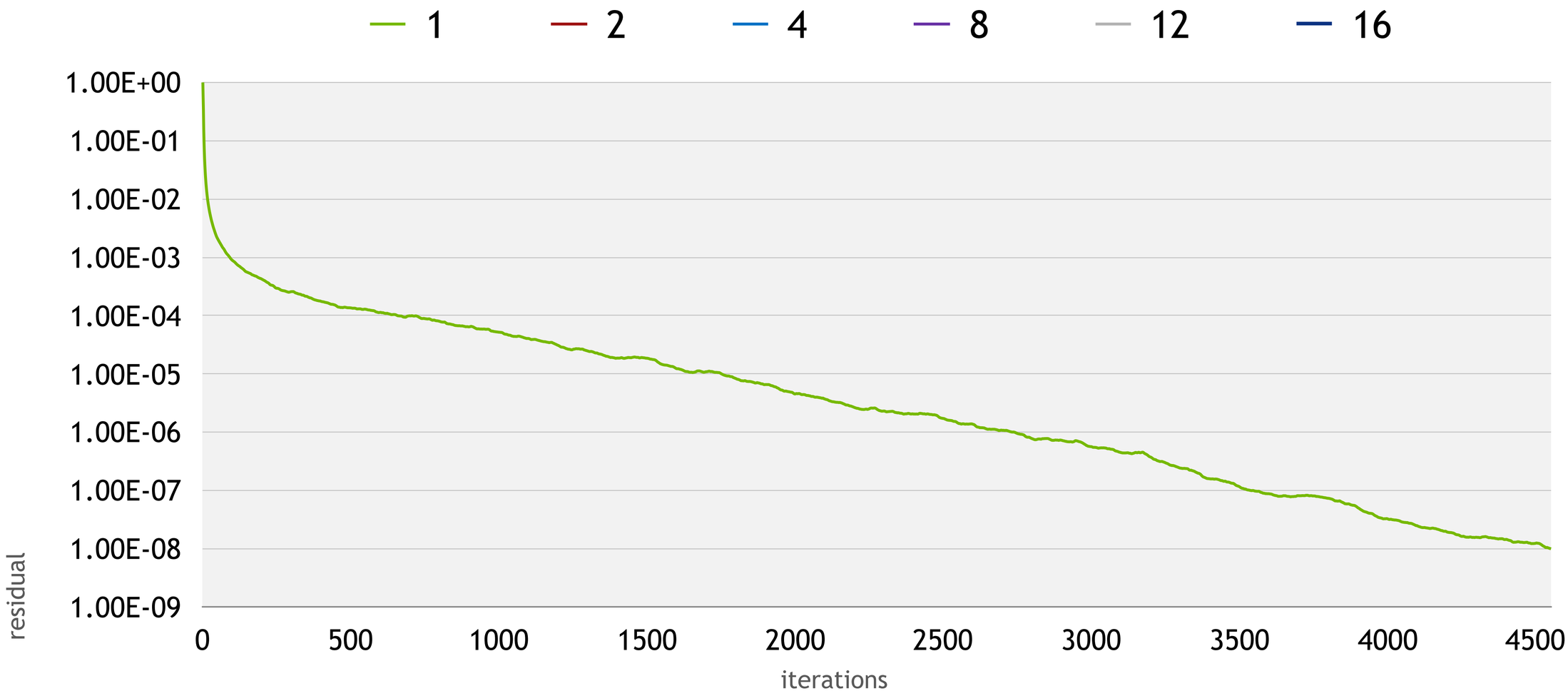
REDUCED ITERATION COUNT

HISQ, 32^3x8, Gaussian random source



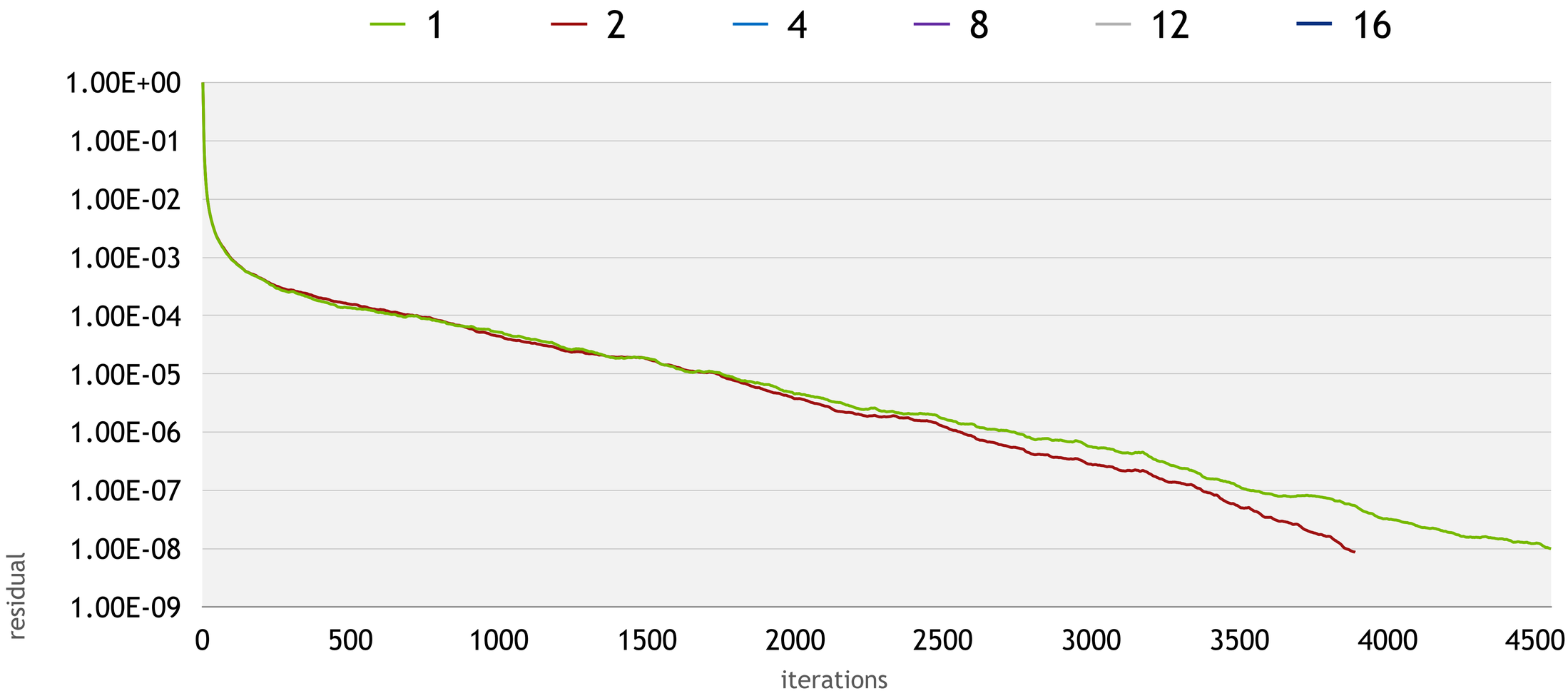
REDUCED ITERATION COUNT

HISQ, 32^3x8, Gaussian random source



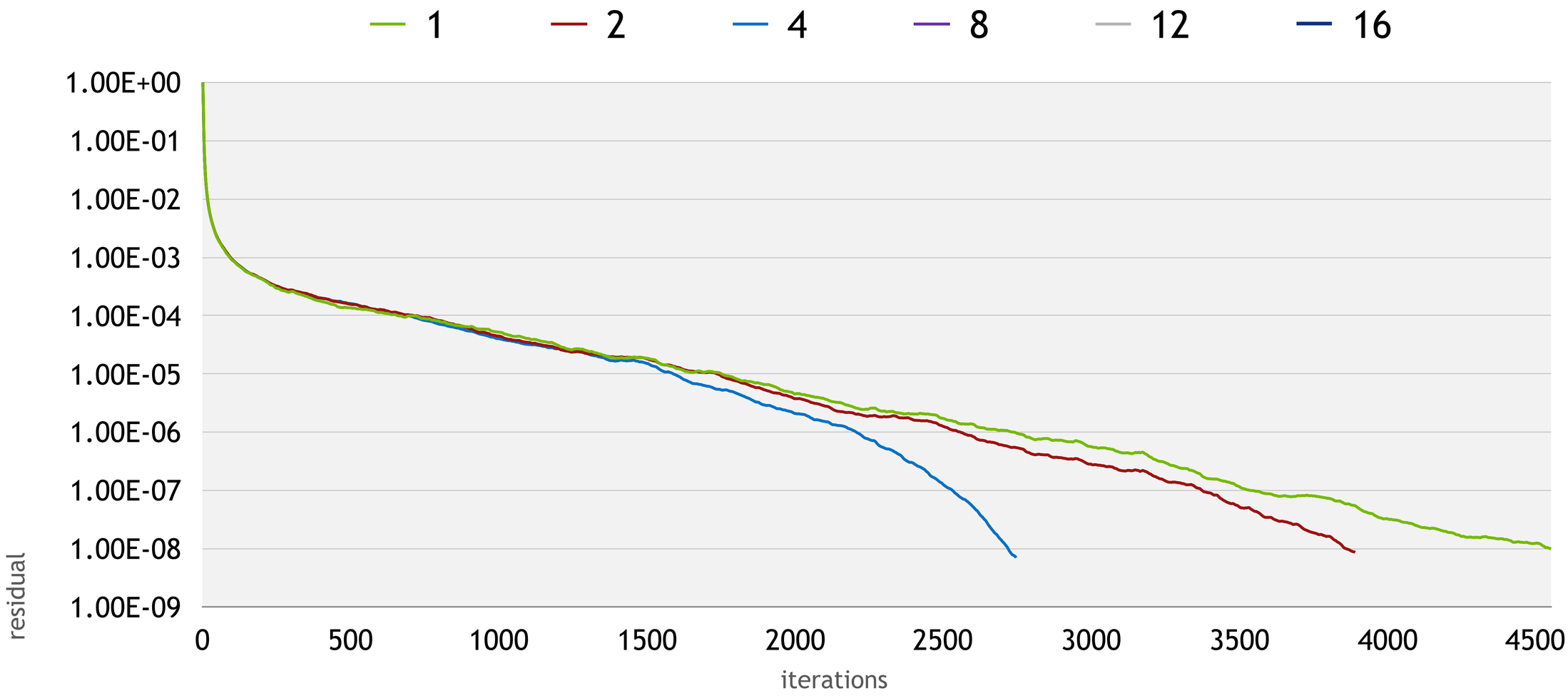
REDUCED ITERATION COUNT

HISQ, 32^3x8, Gaussian random source



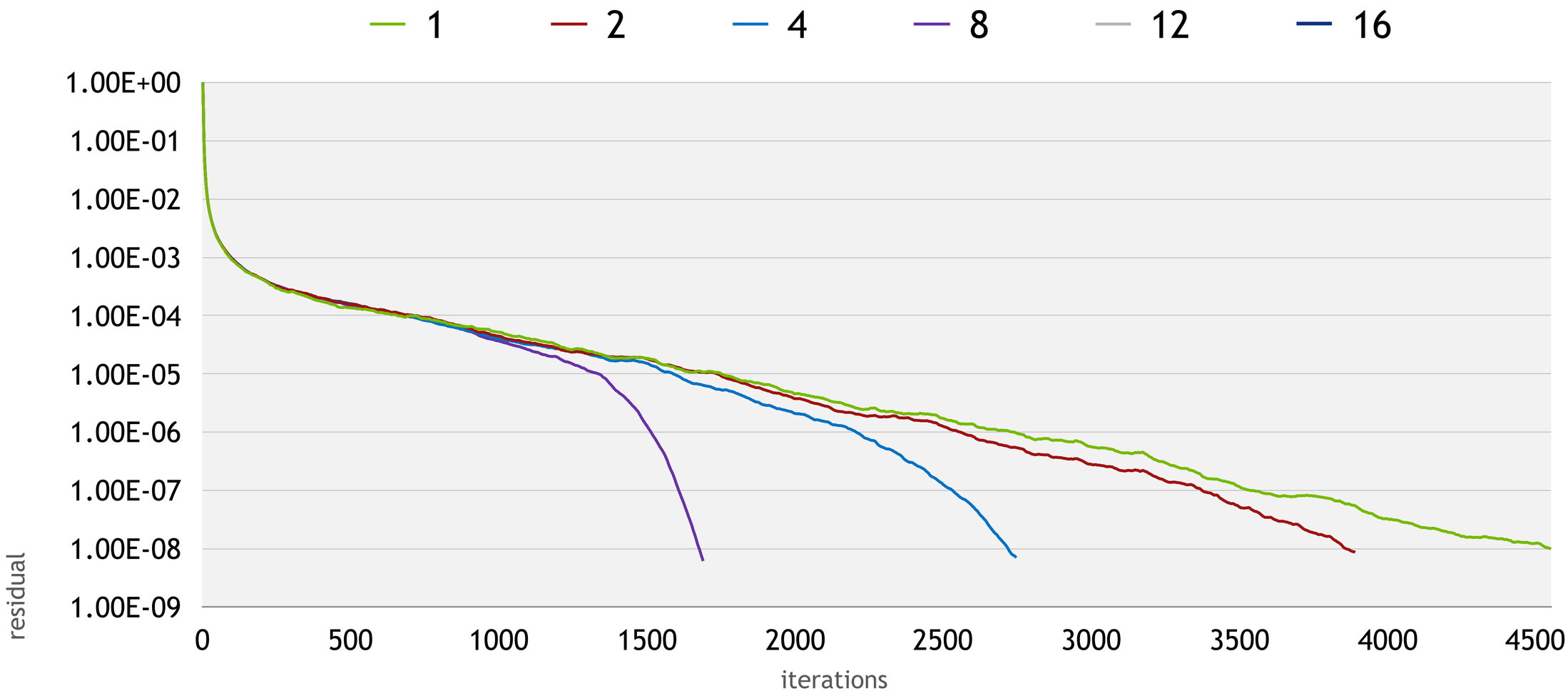
REDUCED ITERATION COUNT

HISQ, 32^3x8, Gaussian random source



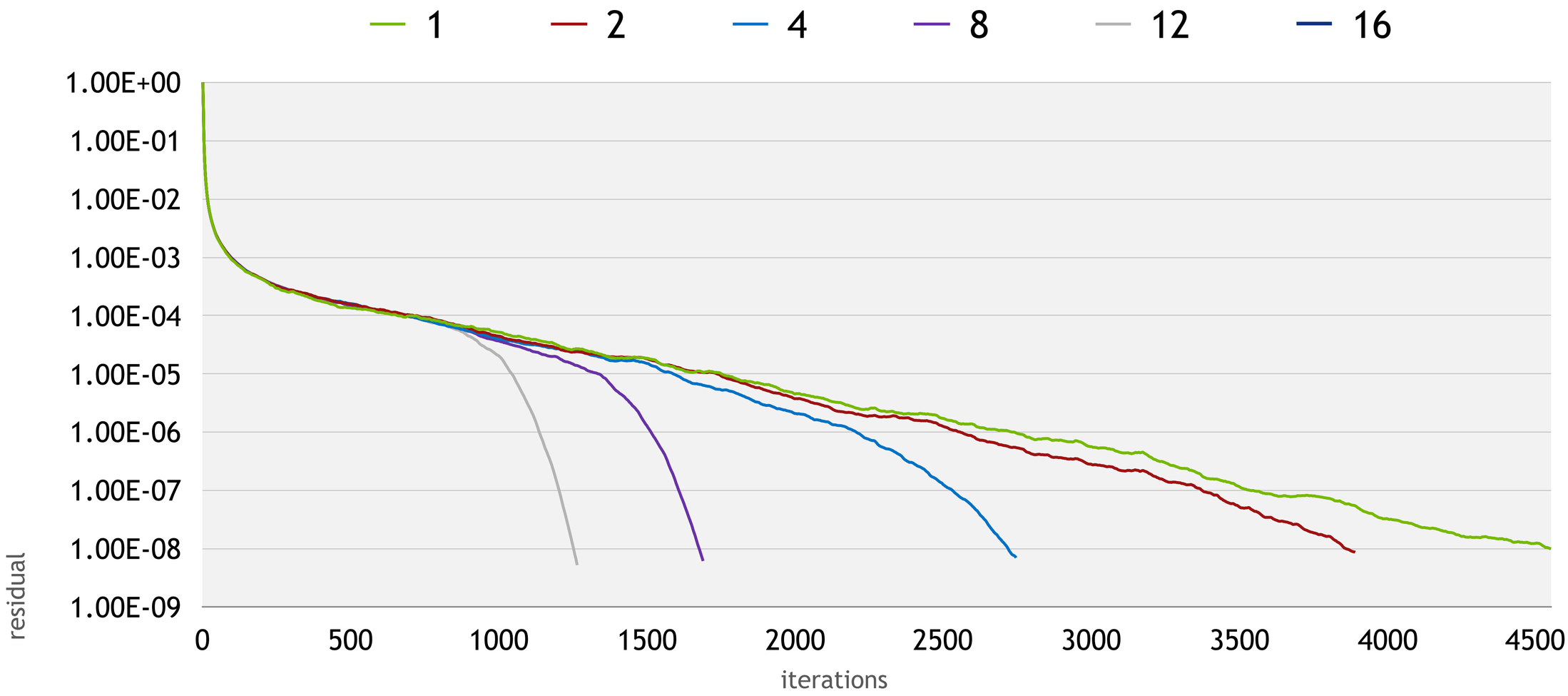
REDUCED ITERATION COUNT

HISQ, 32^3x8, Gaussian random source



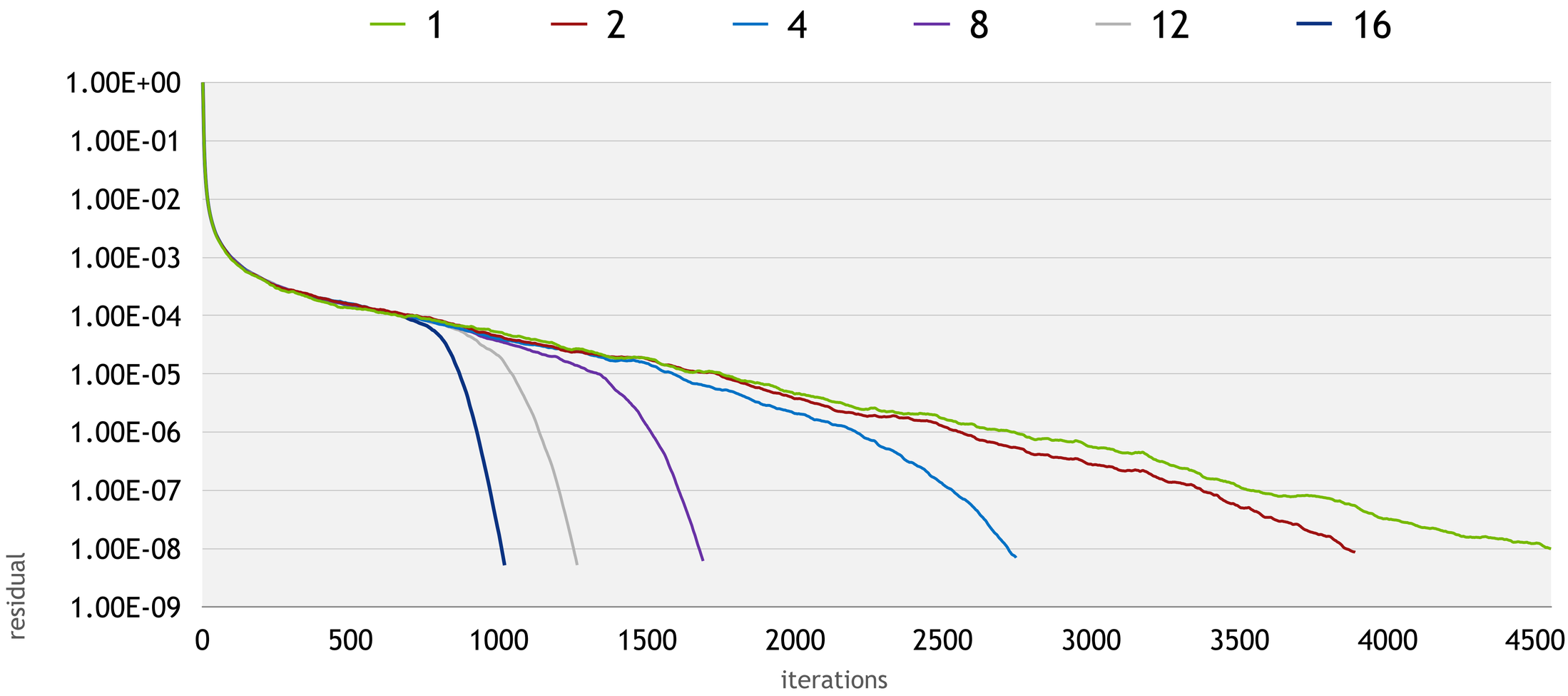
REDUCED ITERATION COUNT

HISQ, 32^3x8, Gaussian random source



REDUCED ITERATION COUNT

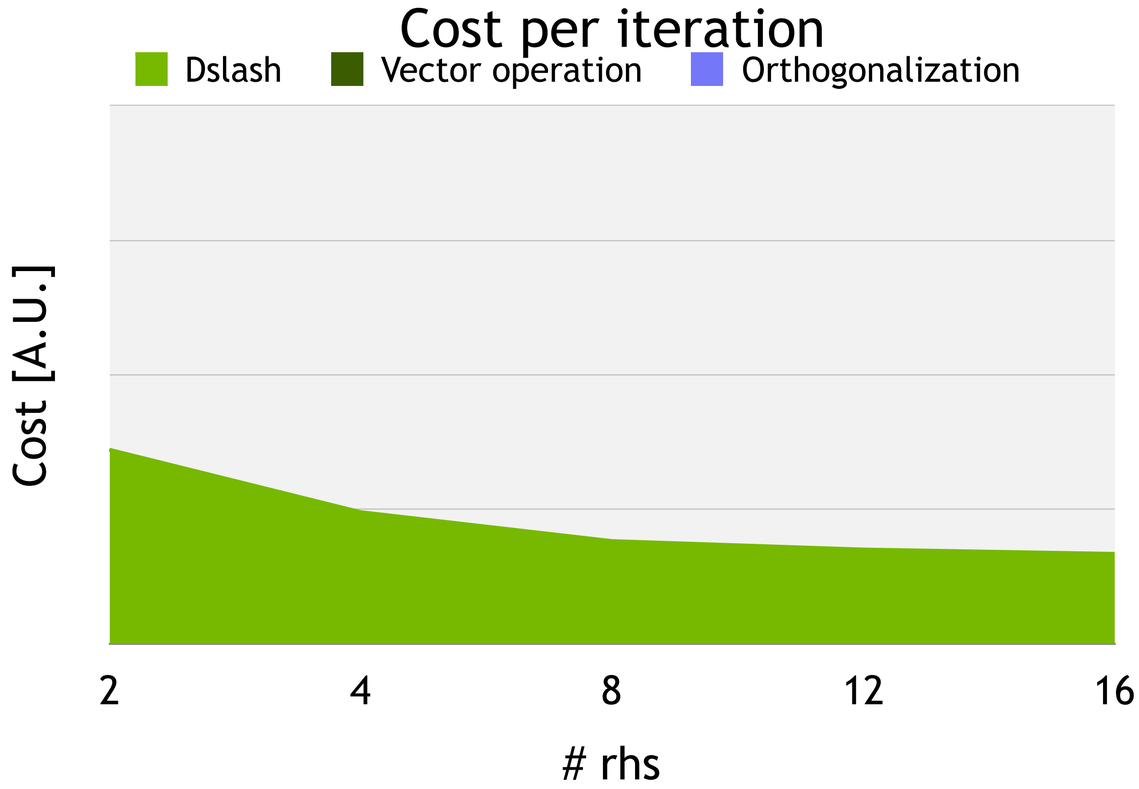
HISQ, 32^3x8, Gaussian random source



WHY DOESN'T EVERYONE USE IT?

BlockCG is not always numerically stable

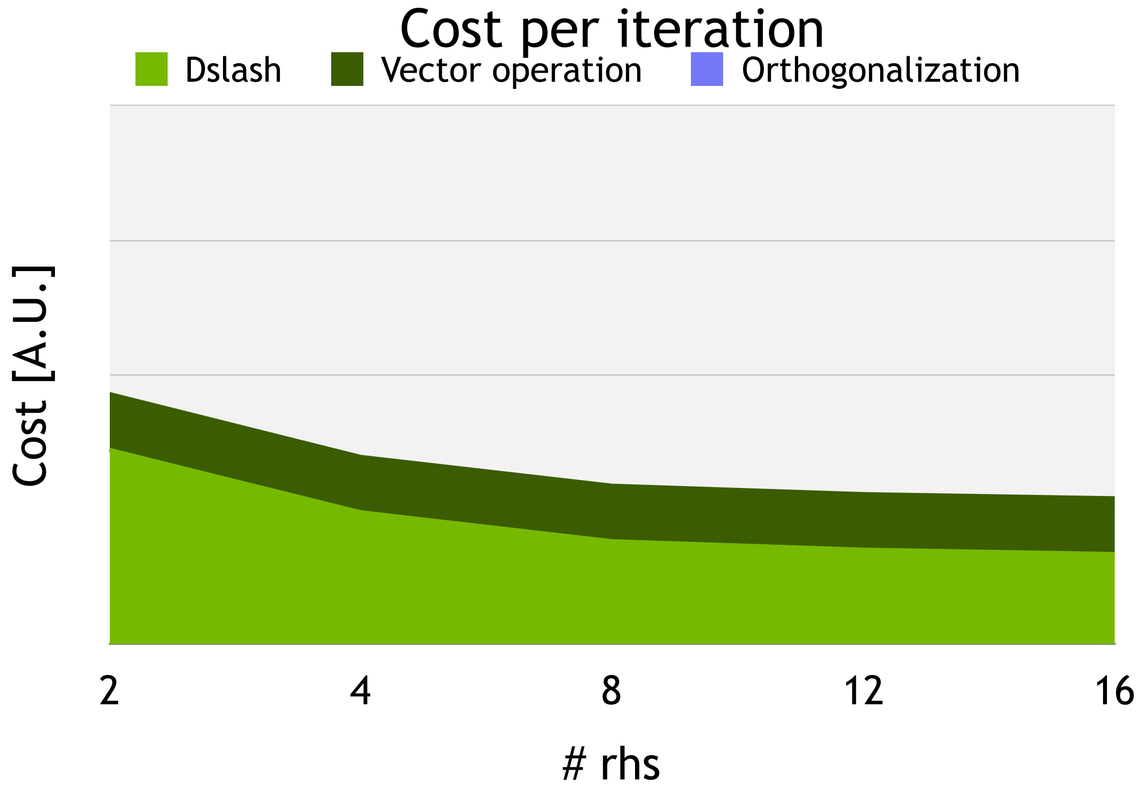
- Remedy with orthogonalization: Gram-Schmidt or modified Gram-Schmidt
  - Quadratic scaling with # rhs
  - Becomes prohibitive



WHY DOESN'T EVERYONE USE IT?

BlockCG is not always numerically stable

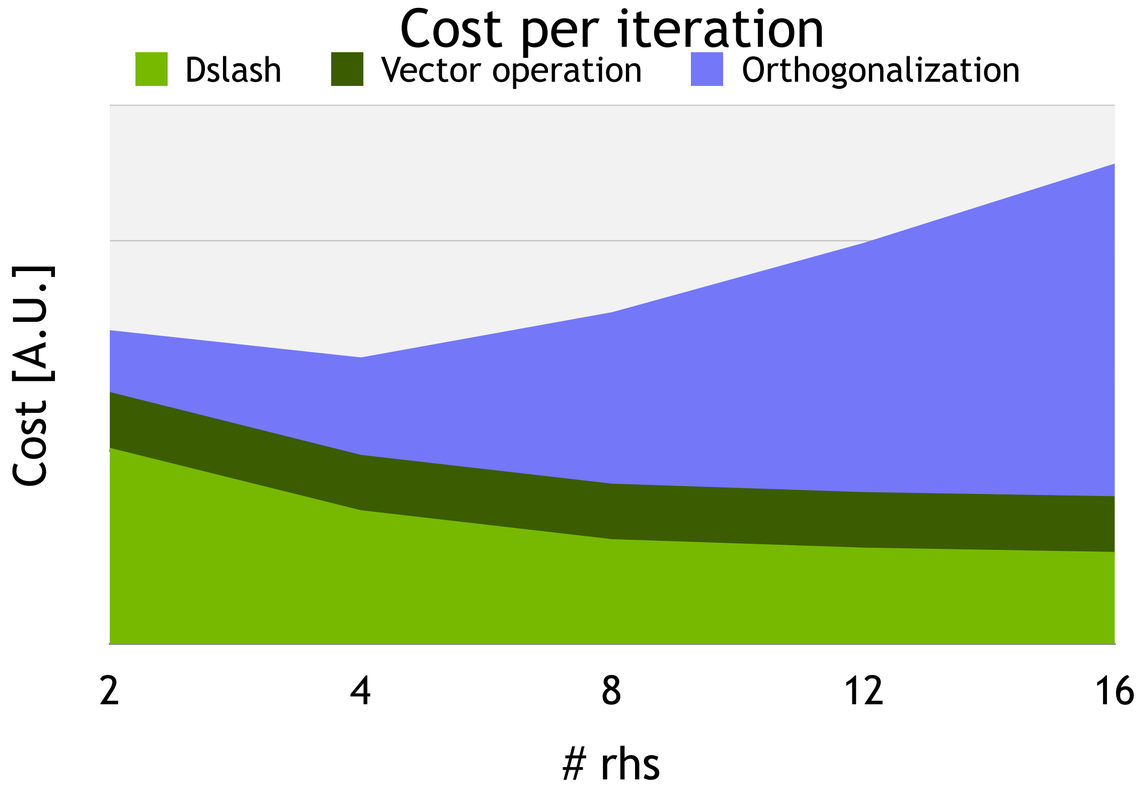
- Remedy with orthogonalization: Gram-Schmidt or modified Gram-Schmidt
  - Quadratic scaling with # rhs
  - Becomes prohibitive



WHY DOESN'T EVERYONE USE IT?

BlockCG is not always numerically stable

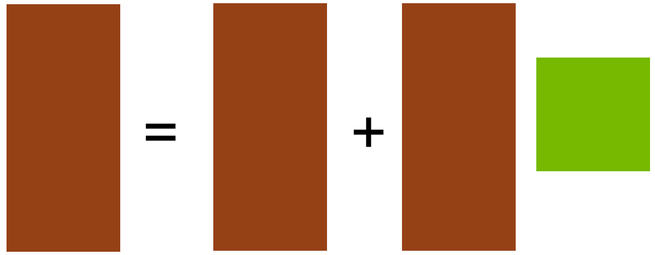
- Remedy with orthogonalization: Gram-Schmidt or modified Gram-Schmidt
  - Quadratic scaling with # rhs
  - Becomes prohibitive



EXPLOIT GPU ARCHITECTURE

to overcome quadratic scaling

$$y_i = \sum a_{ij} x_j + y_i$$



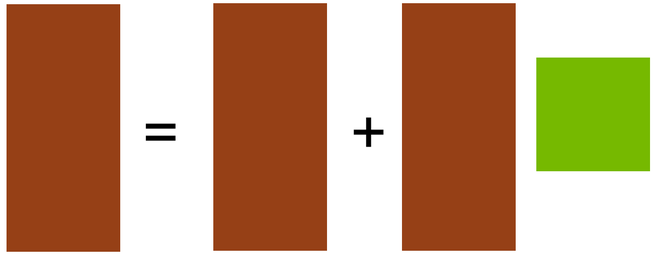
CUDA supports two dimensional grid blocks:

easy to exploit locality for texture cache / shared memory

EXPLOIT GPU ARCHITECTURE

to overcome quadratic scaling

$$y_i = \sum a_{ij} x_j + y_i$$



CUDA supports two dimensional grid blocks:

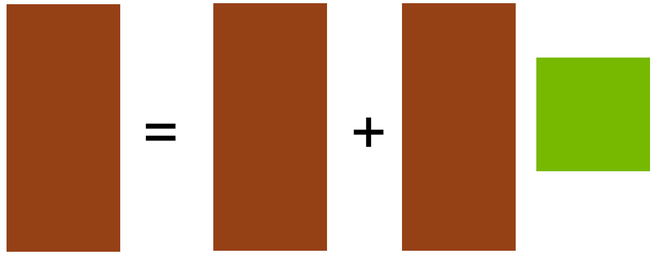
easy to exploit locality for texture cache / shared memory

(0,0)	(1,0)	(2,0)	(3,0)
(0,1)	(1,1)	(2,1)	(3,1)
(0,3)	(1,2)	(2,2)	(3,2)
(0,4)	(1,3)	(2,3)	(3,4)

EXPLOIT GPU ARCHITECTURE

to overcome quadratic scaling

$$y_i = \sum a_{ij} x_j + y_i$$



CUDA supports two dimensional grid blocks:

easy to exploit locality for texture cache / shared memory

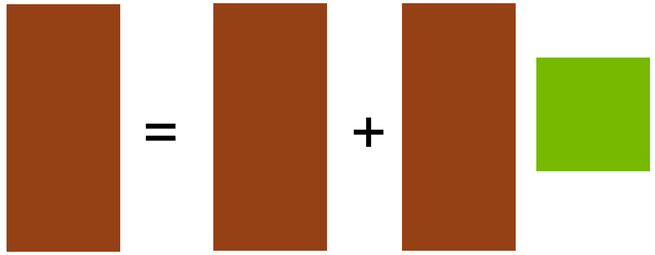
$$y_0(0) = a_{00}x_0(0) + a_{01}x_1(0) + \ldots$$
$$y_1(0) = a_{10}x_0(0) + a_{11}x_1(0) + \ldots$$
$$y_2(0) = a_{20}x_0(0) + a_{21}x_1(0) + \ldots$$
$$y_3(0) = a_{30}x_0(0) + a_{31}x_1(0) + \ldots$$

(0,0)	(1,0)	(2,0)	(3,0)
(0,1)	(1,1)	(2,1)	(3,1)
(0,3)	(1,2)	(2,2)	(3,2)
(0,4)	(1,3)	(2,3)	(3,4)

EXPLOIT GPU ARCHITECTURE

to overcome quadratic scaling

$$y_i = \sum a_{ij} x_j + y_i$$



CUDA supports two dimensional grid blocks:

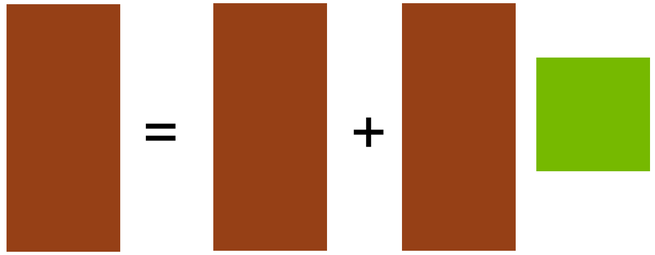
easy to exploit locality for texture cache / shared memory

$$y_0(0) = a_{00}x_0(0) + a_{01}x_1(0) + \ldots$$
$$y_1(0) = a_{10}x_0(0) + a_{11}x_1(0) + \ldots$$
$$y_2(0) = a_{20}x_0(0) + a_{21}x_1(0) + \ldots$$
$$y_3(0) = a_{30}x_0(0) + a_{31}x_1(0) + \ldots$$

EXPLOIT GPU ARCHITECTURE

to overcome quadratic scaling

$$y_i = \sum a_{ij} x_j + y_i$$



CUDA supports two dimensional grid blocks:

easy to exploit locality for texture cache / shared memory

$$y_0(0) = a_{00}x_0(0) + a_{01}x_1(0) + \ldots$$
$$y_1(0) = a_{10}x_0(0) + a_{11}x_1(0) + \ldots$$
$$y_2(0) = a_{20}x_0(0) + a_{21}x_1(0) + \ldots$$
$$y_3(0) = a_{30}x_0(0) + a_{31}x_1(0) + \ldots$$

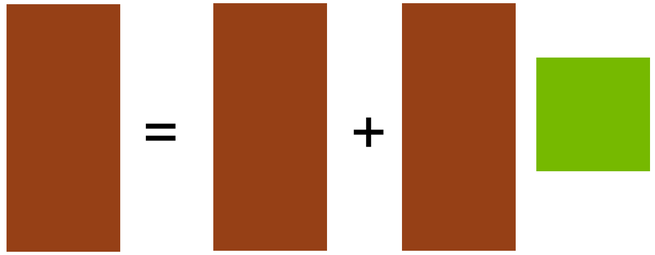
(0,0)	(1,0)	(2,0)	(3,0)
(0,1)	(1,1)	(2,1)	(3,1)
(0,3)	(1,2)	(2,2)	(3,2)
(0,4)	(1,3)	(2,3)	(3,4)

cache reuse

EXPLOIT GPU ARCHITECTURE

to overcome quadratic scaling

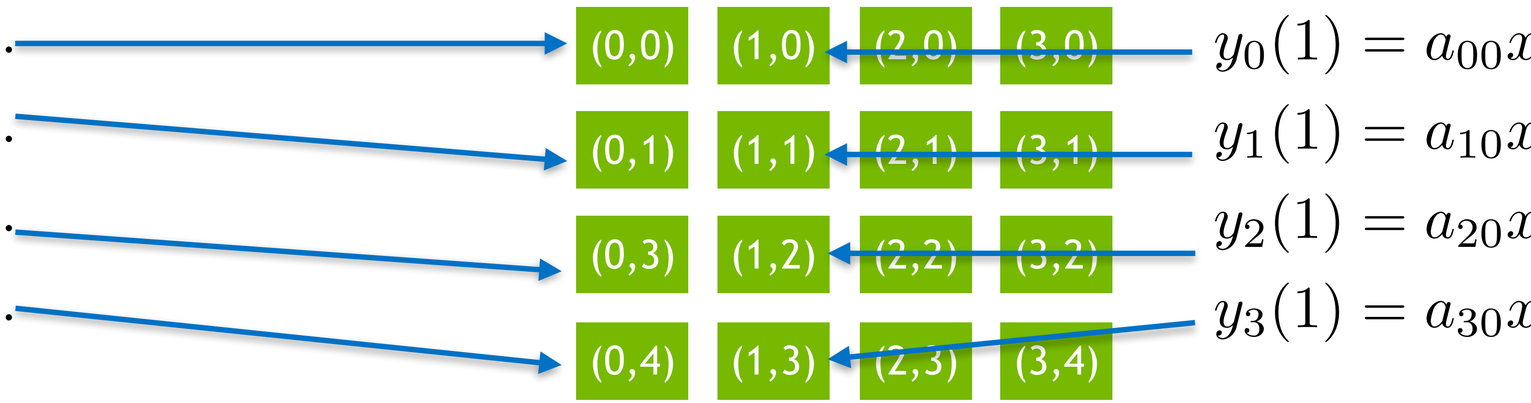
$$y_i = \sum a_{ij} x_j + y_i$$



CUDA supports two dimensional grid blocks:

easy to exploit locality for texture cache / shared memory

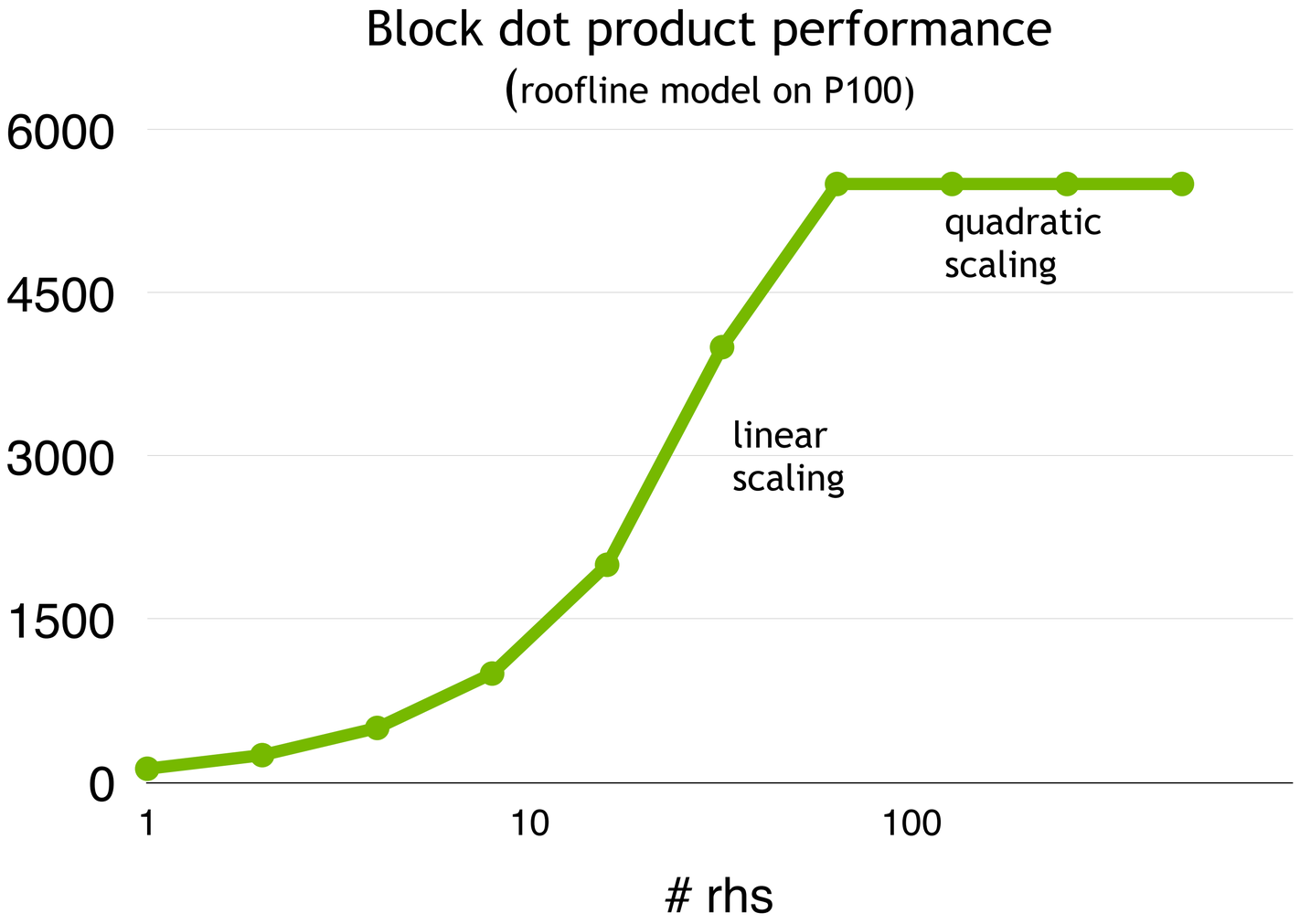
$$y_0(0) = a_{00}x_0(0) + a_{01}x_1(0) + \dots$$
$$y_1(0) = a_{10}x_0(0) + a_{11}x_1(0) + \dots$$
$$y_2(0) = a_{20}x_0(0) + a_{21}x_1(0) + \dots$$
$$y_3(0) = a_{30}x_0(0) + a_{31}x_1(0) + \dots$$



cache reuse

SCALABILITY

Proportional to peak flops not memory bandwidth

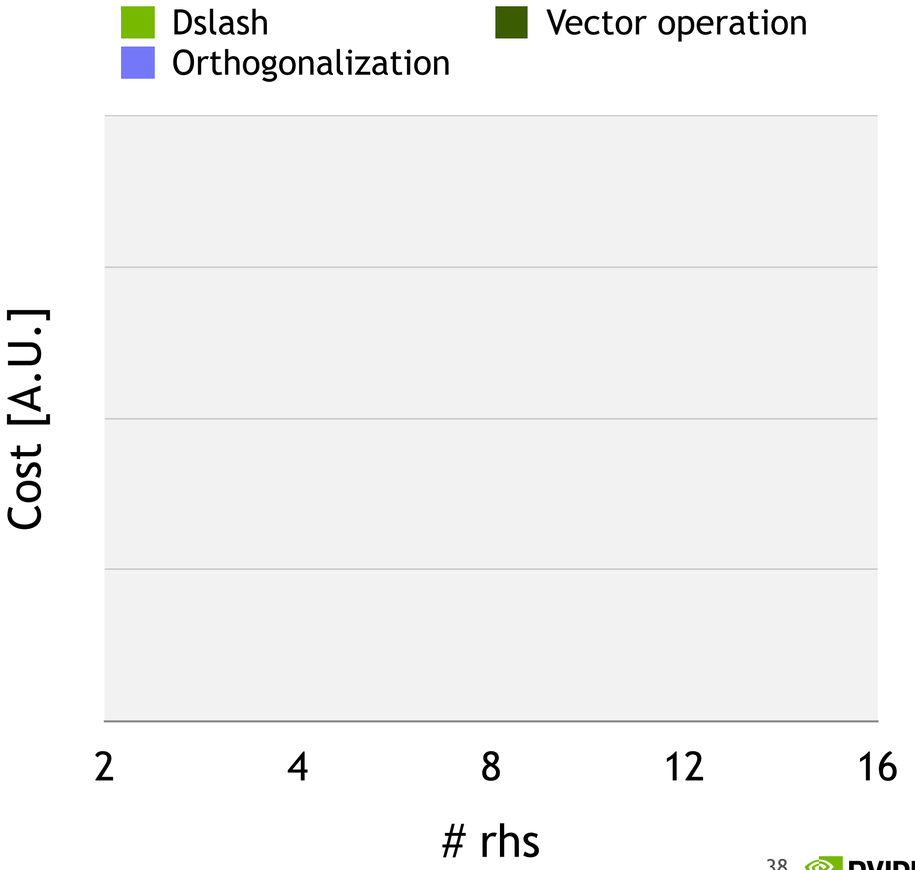


COST OF ONE ITERATION

CholQR

Gram-Matrix:	$B = R^H R$	$m \times m$ dot products of length n
Cholesky Decomposition	$S^H S = B$	of $m \times m$ matrix
apply to vectors	$Q = RS^{-1}$	axpy $m \times m$ (output, input)

- Linear algebra and orthogonalization stay constant
- Large benefits from multi-src Dslash but relative importance of Dslash reduces

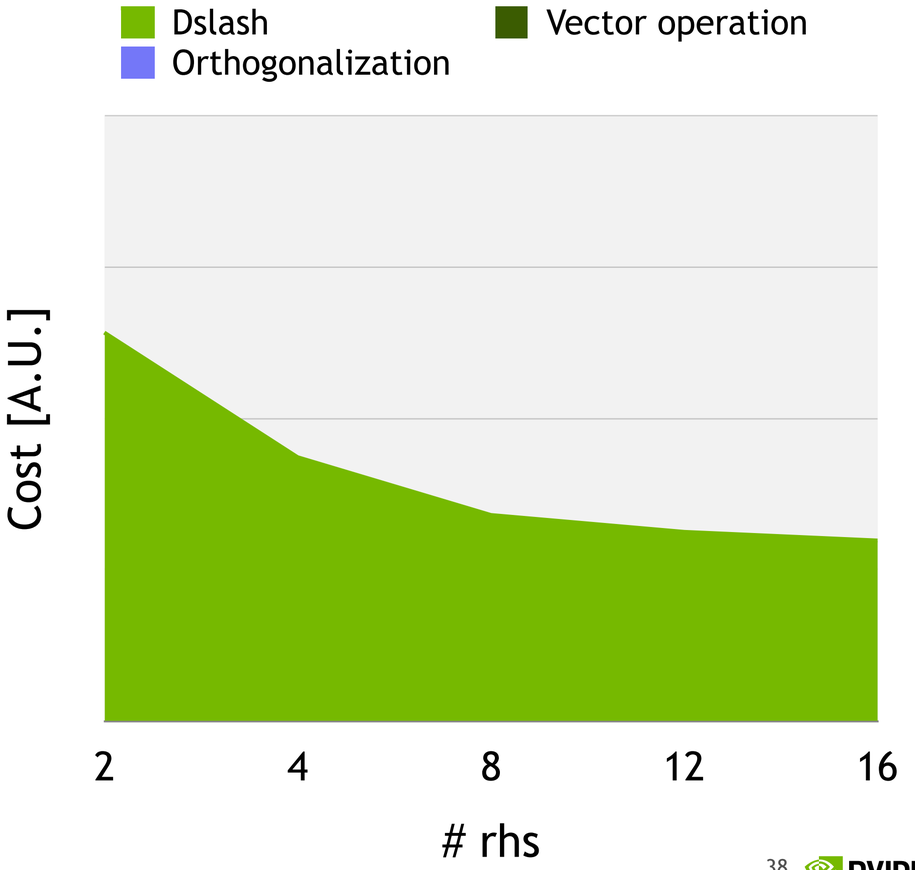


COST OF ONE ITERATION

CholQR

Gram-Matrix:	$B = R^H R$	$m \times m$ dot products of length n
Cholesky Decomposition	$S^H S = B$	of $m \times m$ matrix
apply to vectors	$Q = RS^{-1}$	axpy $m \times m$ (output, input)

- Linear algebra and orthogonalization stay constant
- Large benefits from multi-src Dslash but relative importance of Dslash reduces

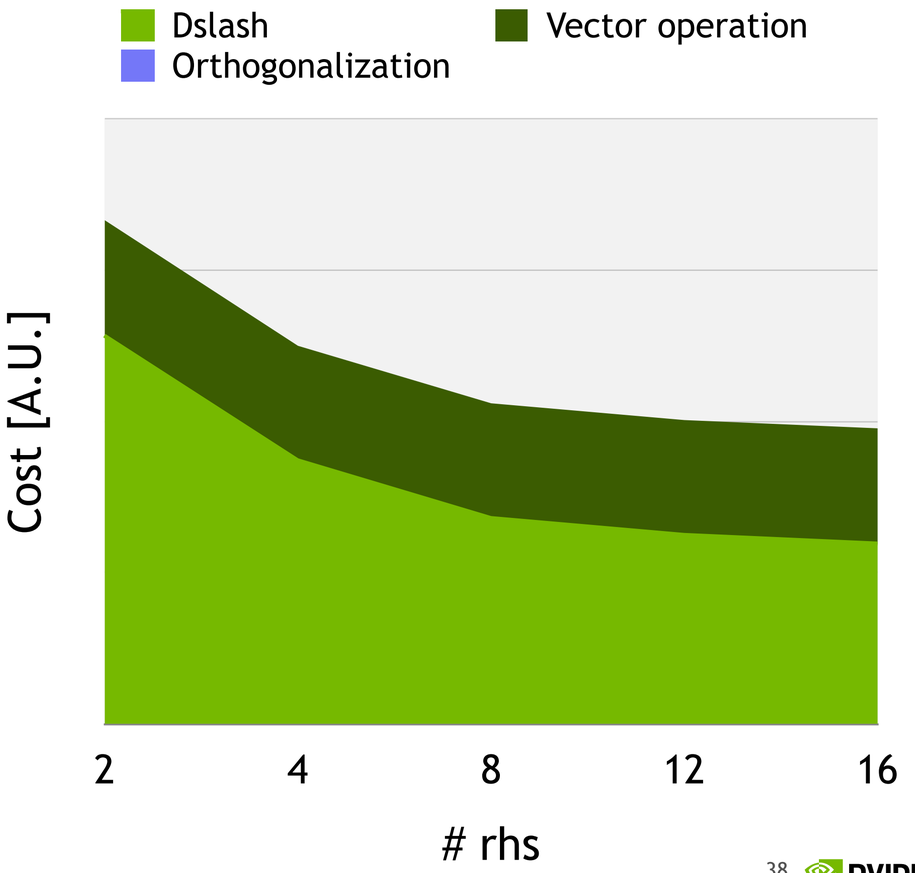


COST OF ONE ITERATION

CholQR

Gram-Matrix:	$B = R^H R$	$m \times m$ dot products of length n
Cholesky Decomposition	$S^H S = B$	of $m \times m$ matrix
apply to vectors	$Q = RS^{-1}$	axpy $m \times m$ (output, input)

- Linear algebra and orthogonalization stay constant
- Large benefits from multi-src Dslash but relative importance of Dslash reduces

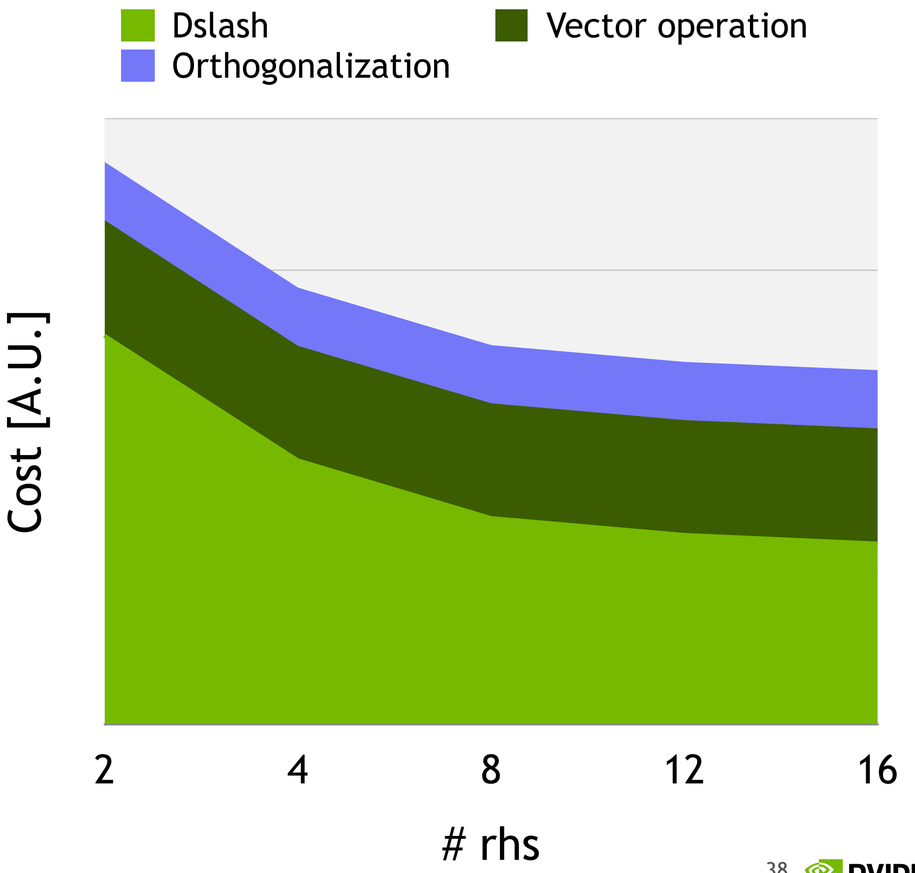


COST OF ONE ITERATION

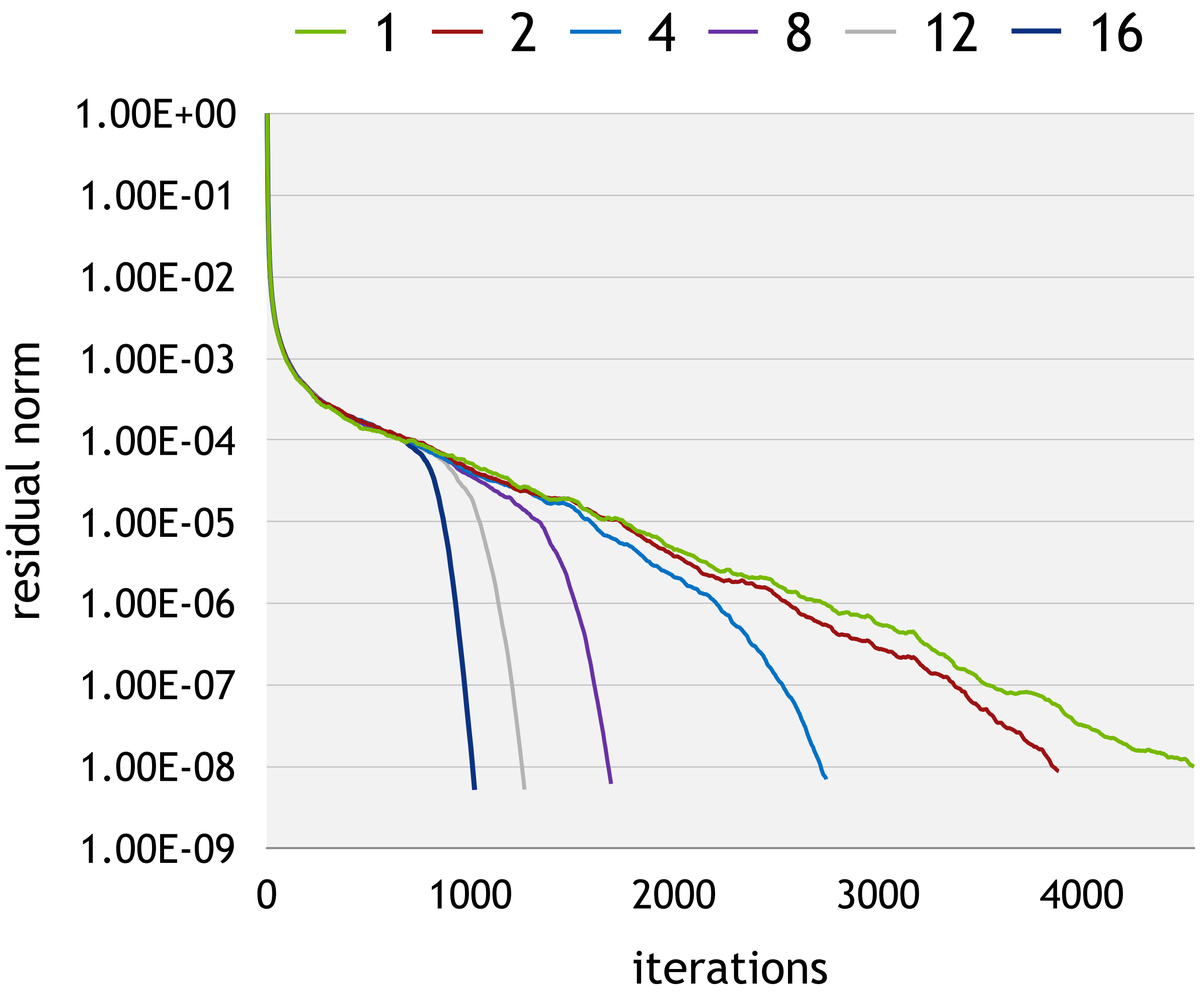
CholQR

Gram-Matrix:	$B = R^H R$	$m \times m$ dot products of length n
Cholesky Decomposition	$S^H S = B$	of $m \times m$ matrix
apply to vectors	$Q = RS^{-1}$	axpy $m \times m$ (output, input)

- Linear algebra and orthogonalization stay constant
- Large benefits from multi-src Dslash but relative importance of Dslash reduces



WORK TO BE DONE



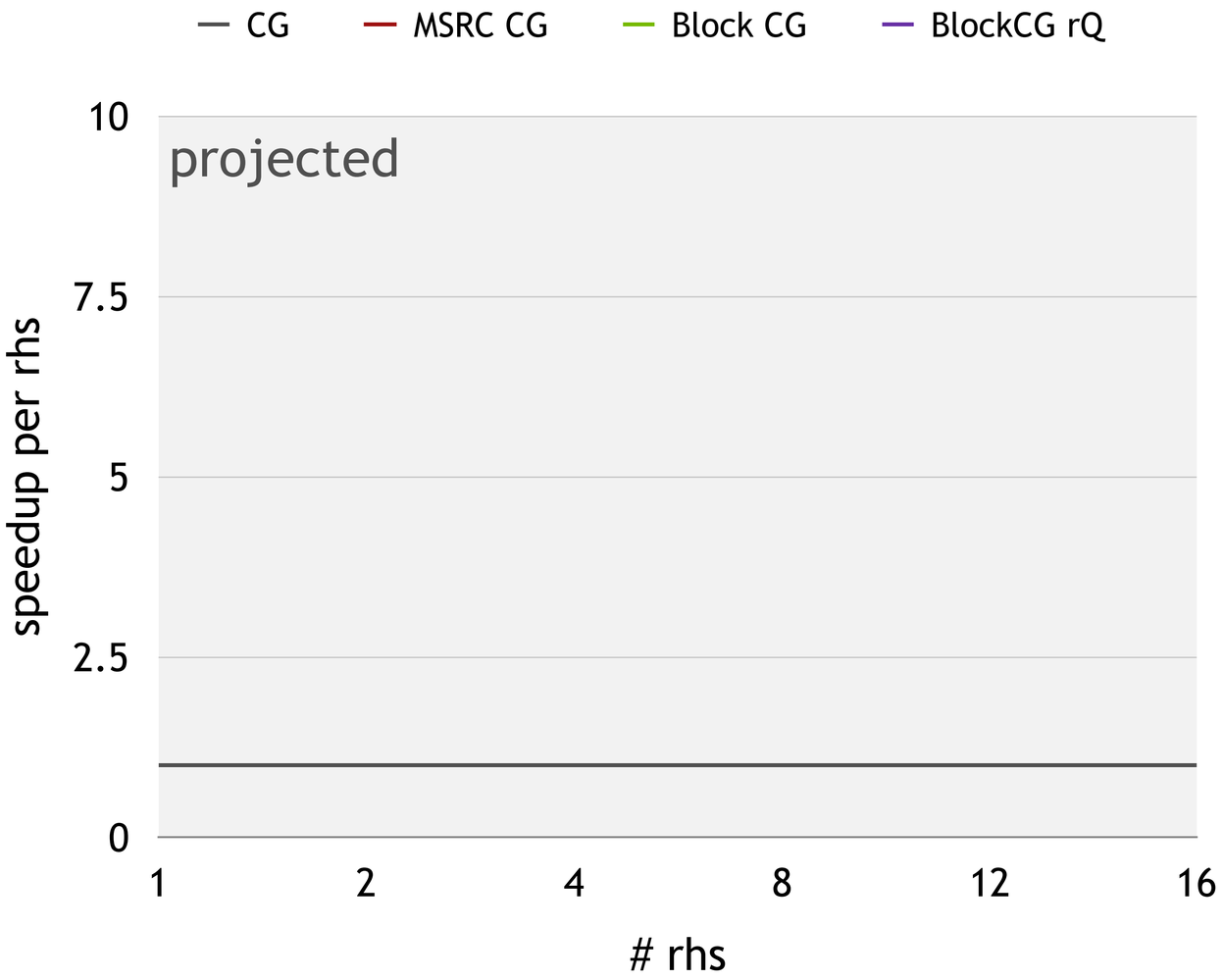
stability needs real world testing
orthogonalization might be necessary

iteration count improvement may depend on gauge field and sources

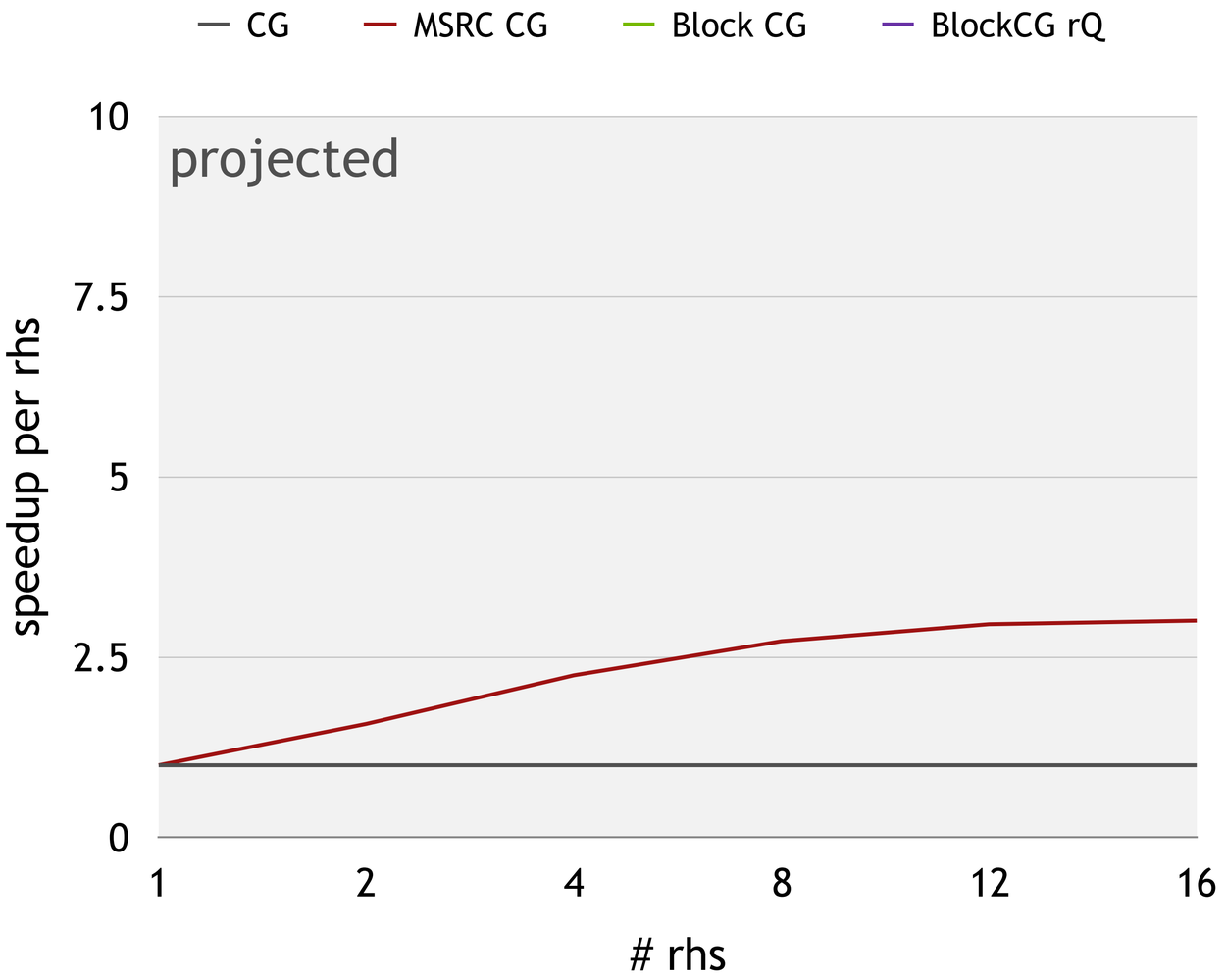
need to finish up implementation

add mixed precision

SPEEDUP OVER CG

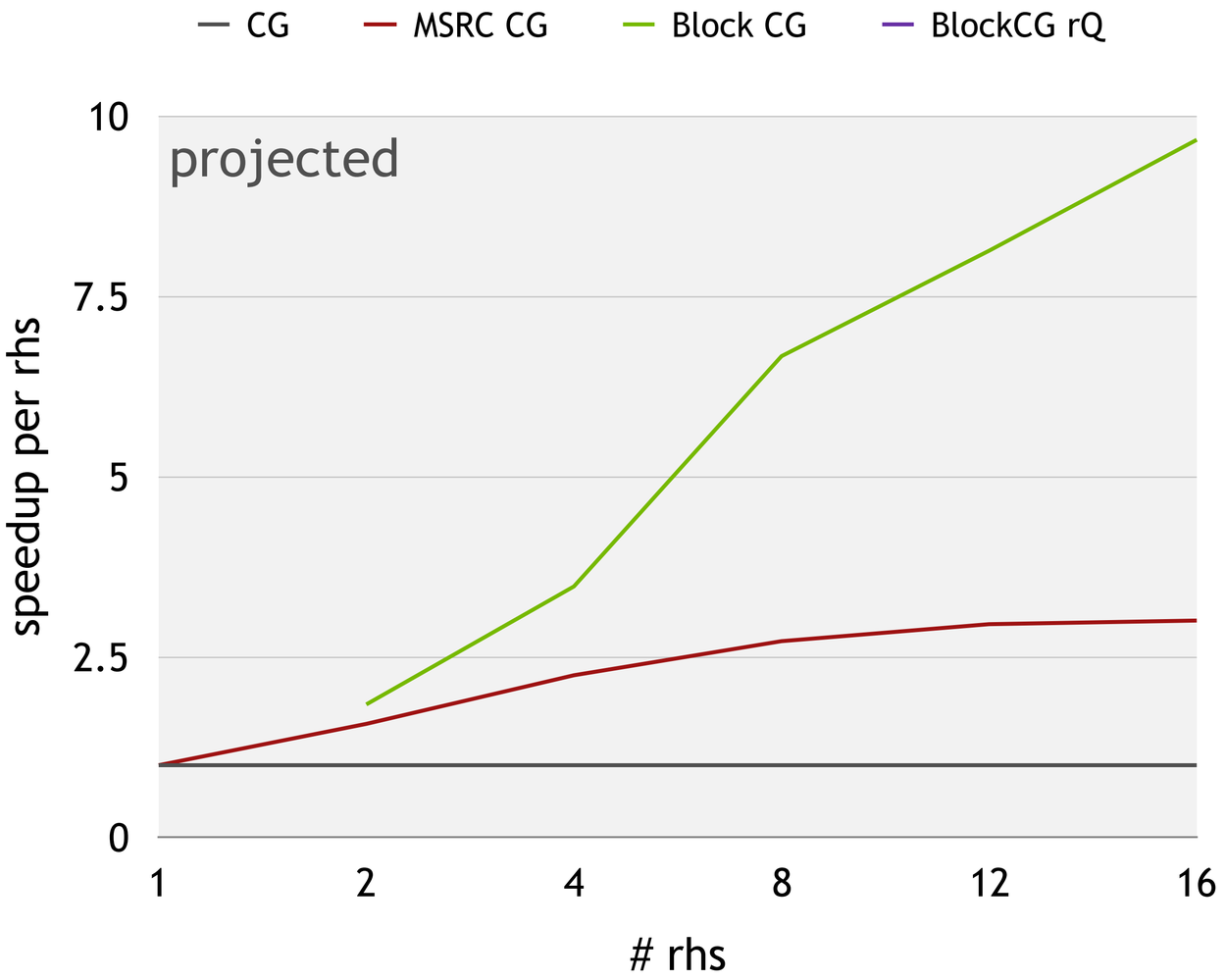


SPEEDUP OVER CG



Reuse gauge field for Dslash

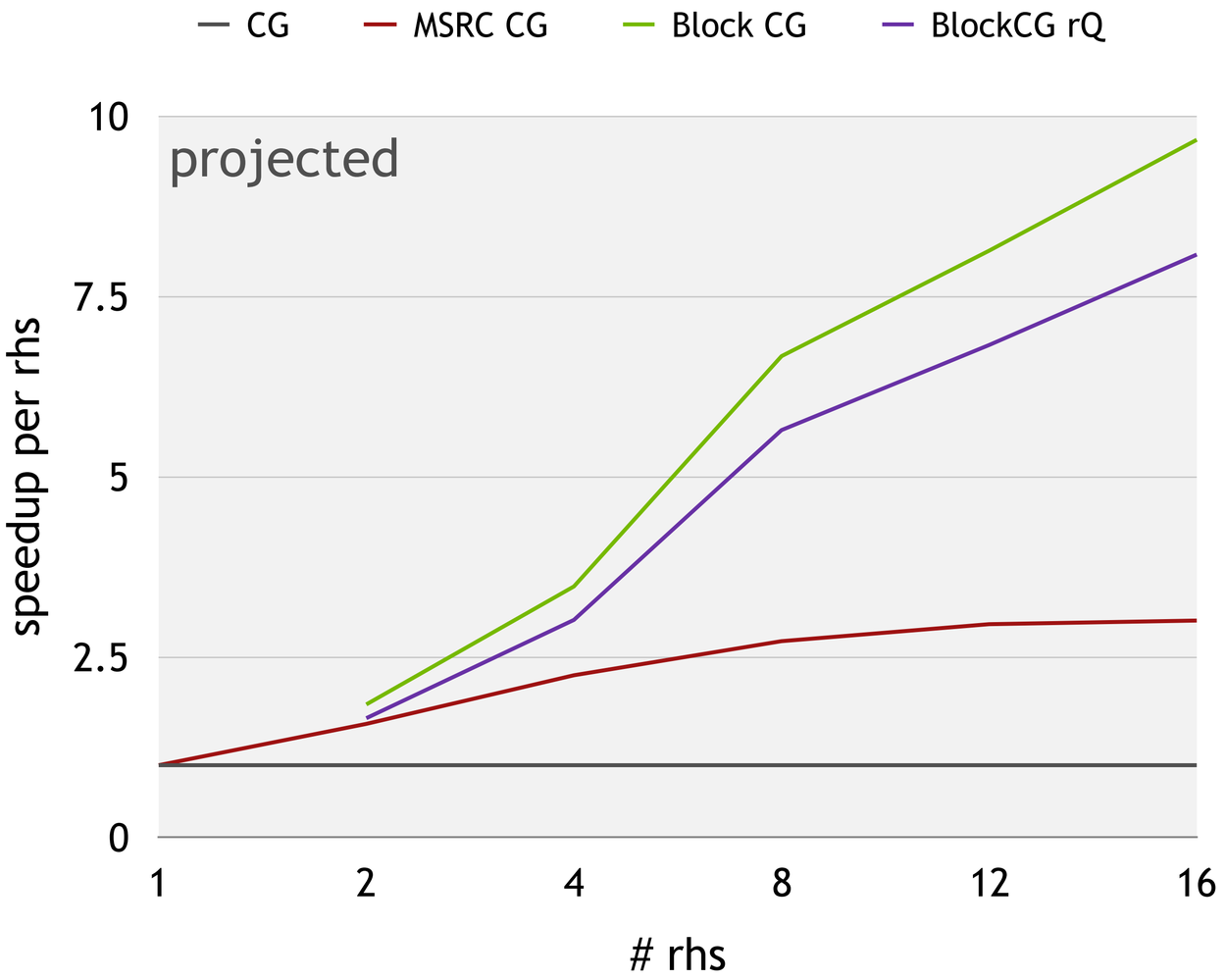
SPEEDUP OVER CG



Reuse gauge field for Dslash

Reduced iteration count

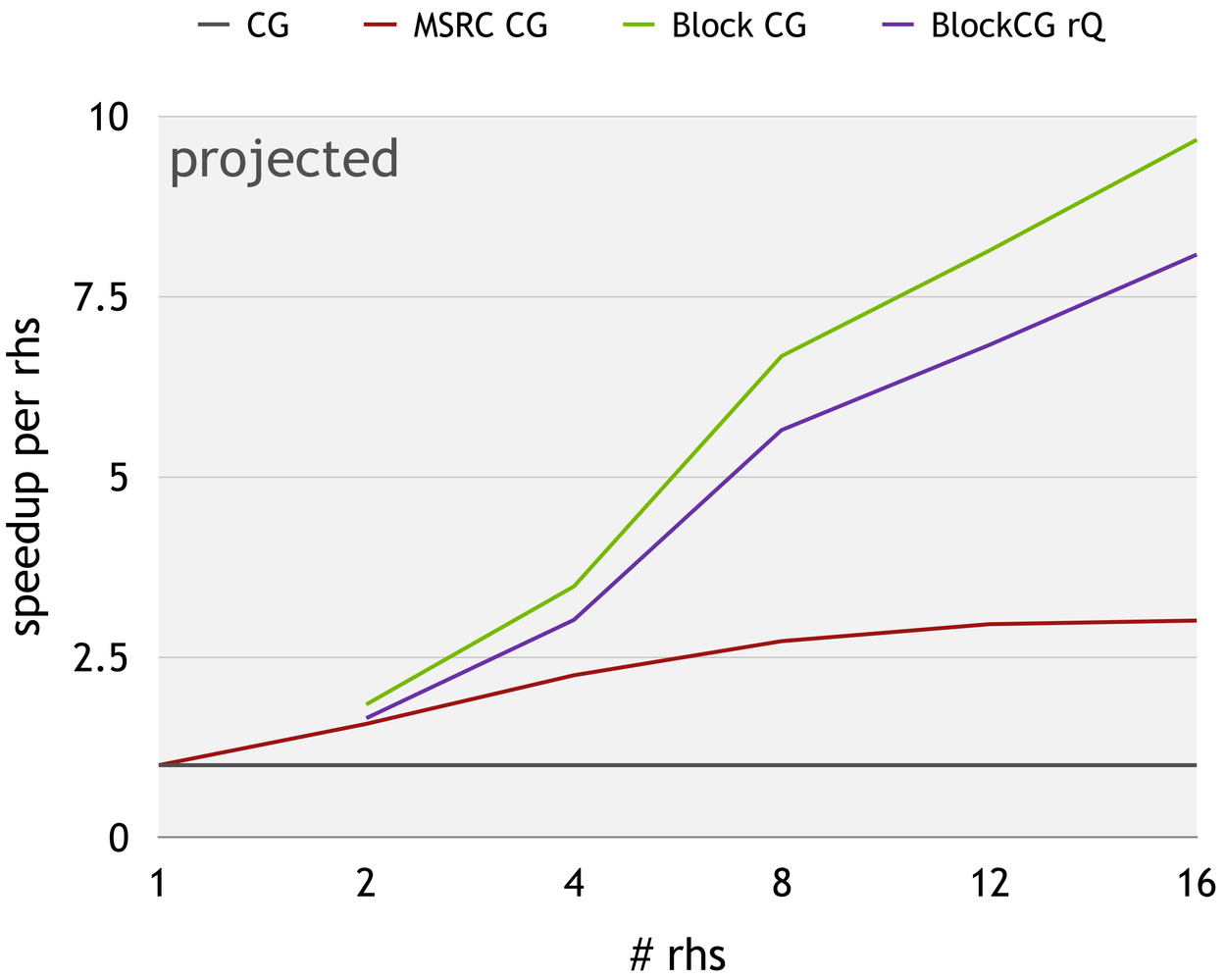
SPEEDUP OVER CG



Reuse gauge field for Dslash

Reduced iteration count

SPEEDUP OVER CG



Reuse gauge field for Dslash

Reduced iteration count

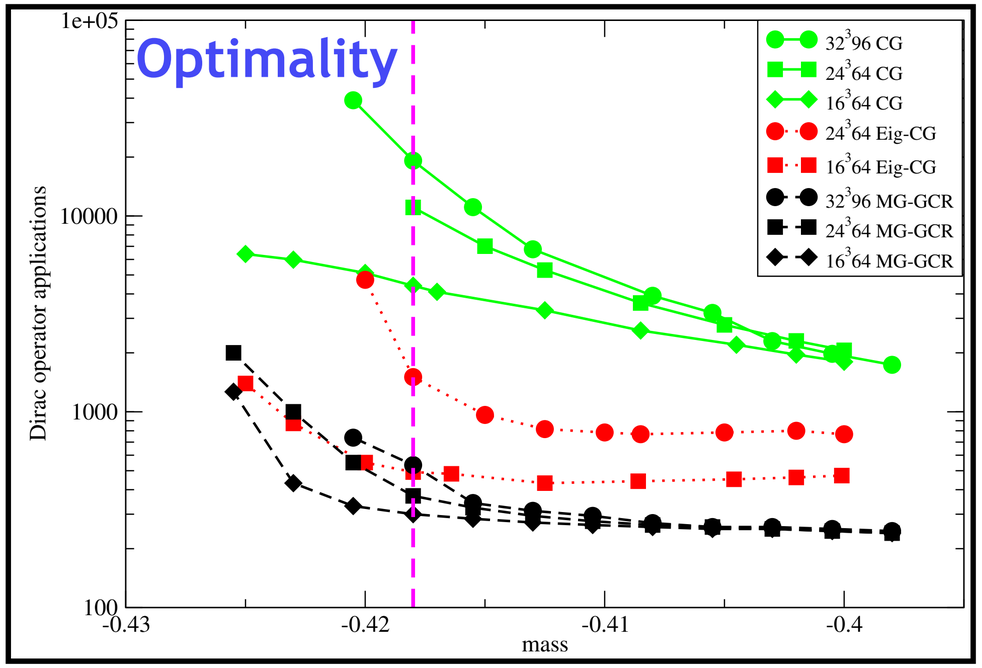
Avoid quadratic scaling in LA

Speedups up to 10x

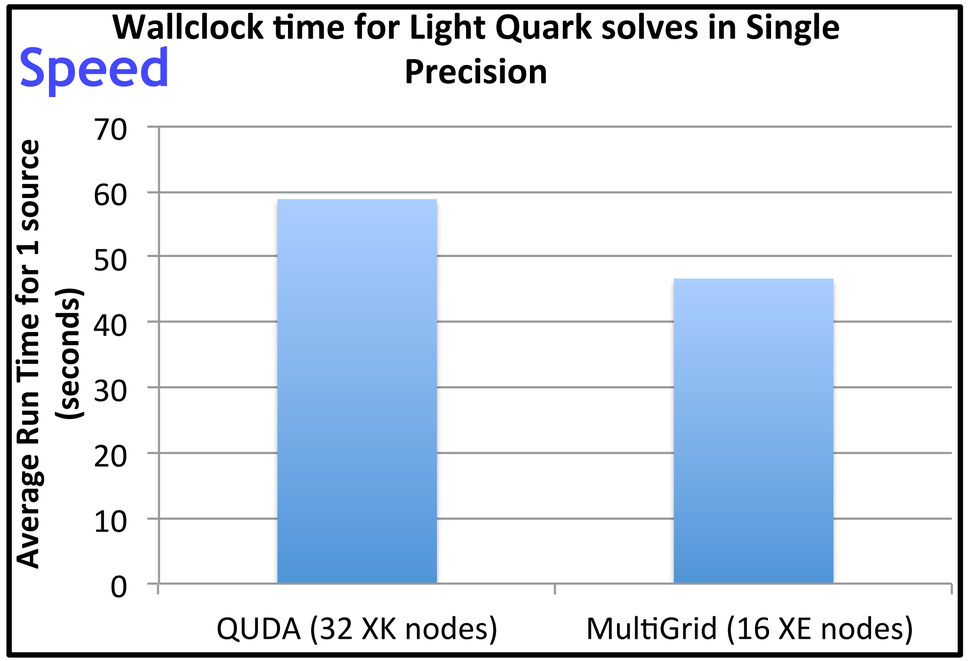
Scalable algorithm for future architectures

MULTIGRID

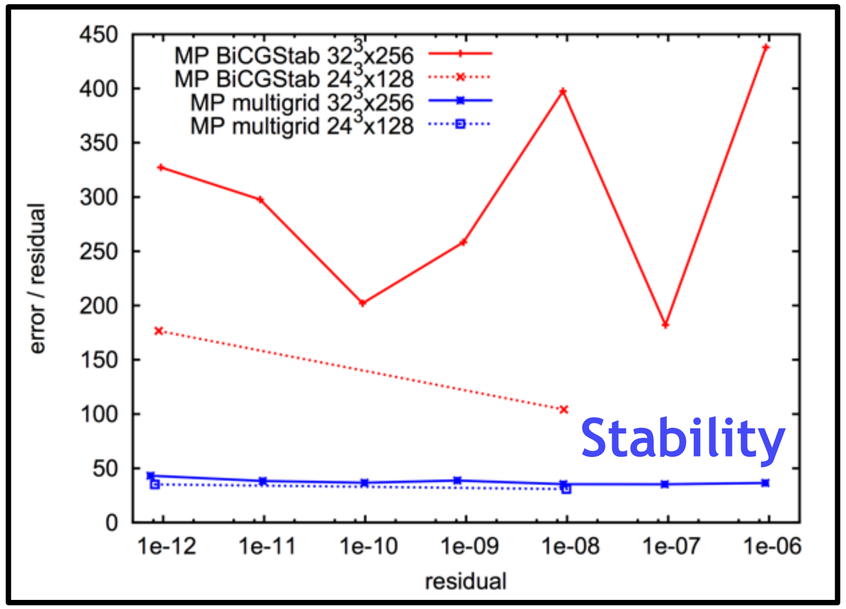
WHY MULTIGRID?



Babich *et al* 2010



Chroma propagator benchmark
Figure by Balint Joo
MG Chroma integration by Saul Cohen
MG Algorithm by James Osborn



Osborn *et al* 2010

ADAPTIVE GEOMETRIC MULTIGRID

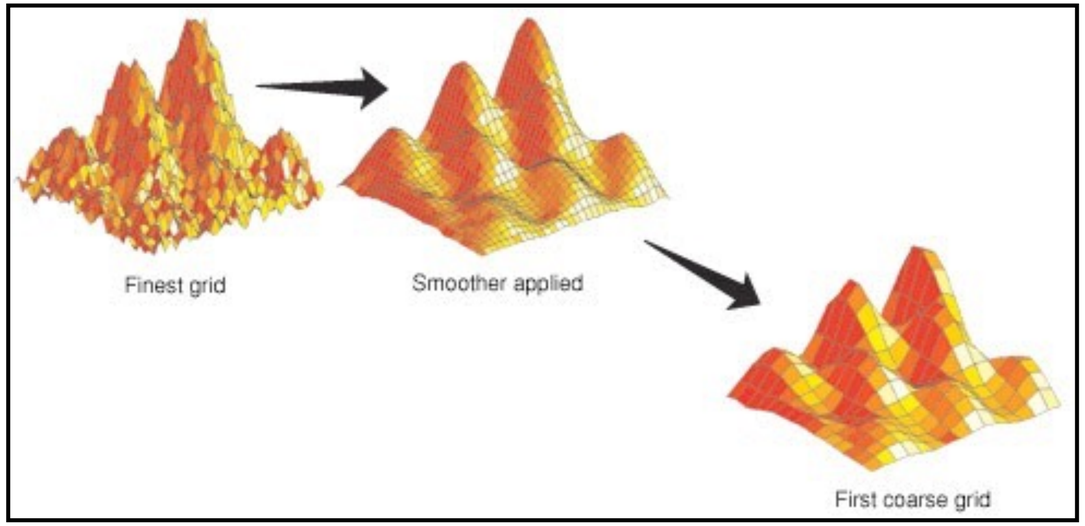
Adaptively find candidate null-space vectors

- Dynamically learn the null space and use this to define the prolongator
- Algorithm is self learning

Setup

1. Set solver to be simple smoother
2. Apply current solver to random vector $v_i = P(D)\ \eta_i$
3. If convergence good enough, solver setup complete
4. Construct prolongator using fixed coarsening $(1 - P\ R)\ v_k = 0$
   - ➡ Typically use 4^4 geometric blocks
   - ➡ Preserve chirality when coarsening $R = \gamma_5\ P^\dagger\ \gamma_5 = P^\dagger$
5. Construct coarse operator ($D_c = R\ D\ P$)
6. Recurse on coarse problem
7. Set solver to be augmented V-cycle, goto 2

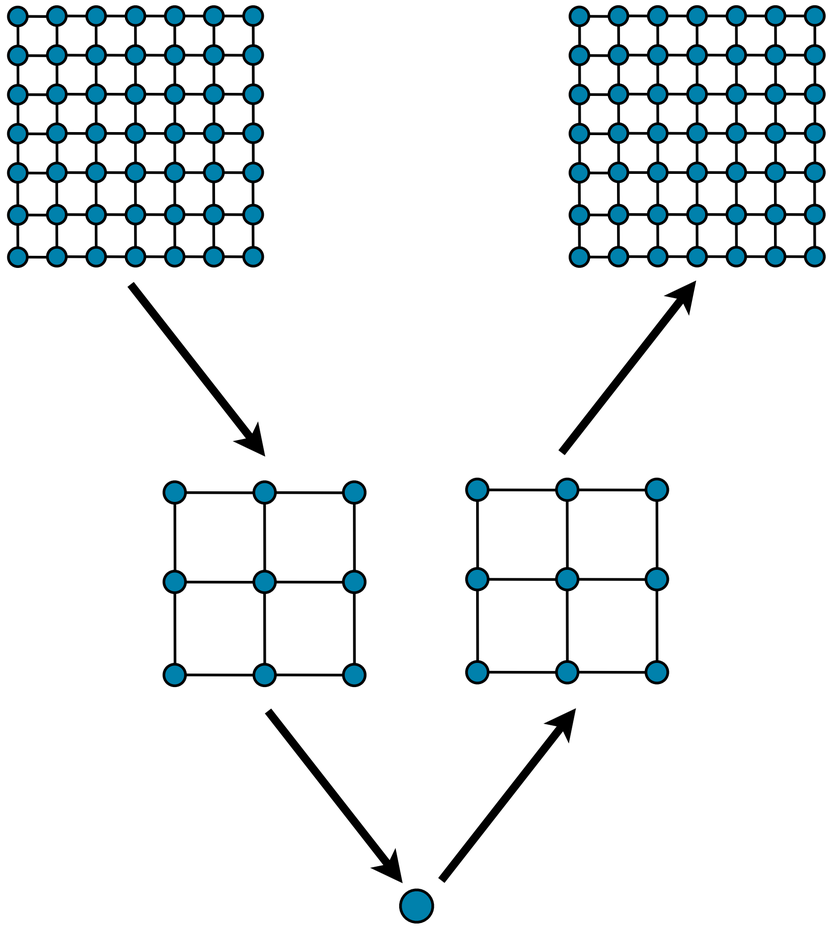
Babich *et al* 2010



Falgout

see also Inexact Deflation (Lüscher, 2007)
Local coherence = weak approximation theory

THE CHALLENGE OF MULTIGRID ON GPU



GPU requirements very different from CPU

- Each thread is slow, but O(10,000) threads per GPU

Fine grids run very efficiently

- High parallel throughput problem

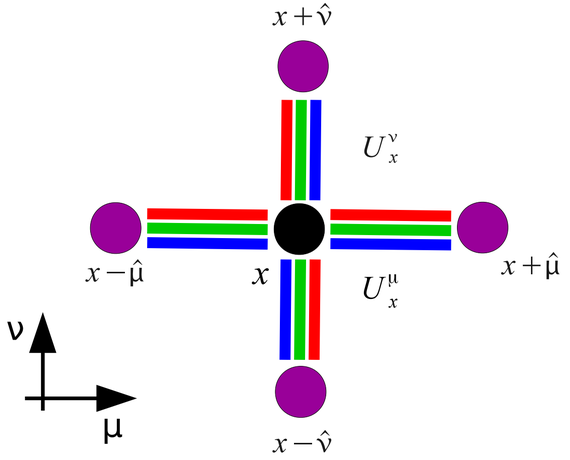
Coarse grids are worst possible scenario

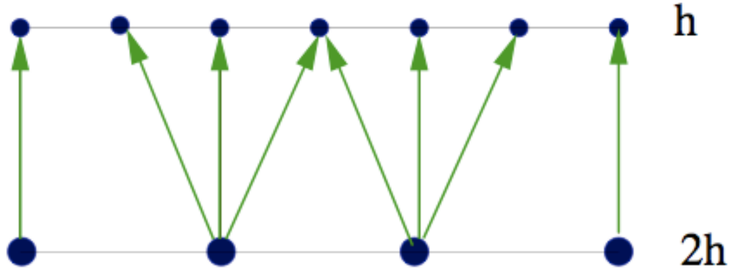
- More cores than degrees of freedom
- Increasingly serial and latency bound
- Little's law (bytes = bandwidth * latency)
- Amdahl's law limiter

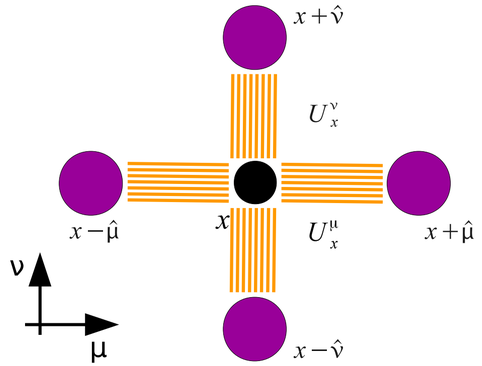
Multigrid exposes many of the problems expected at the Exascale

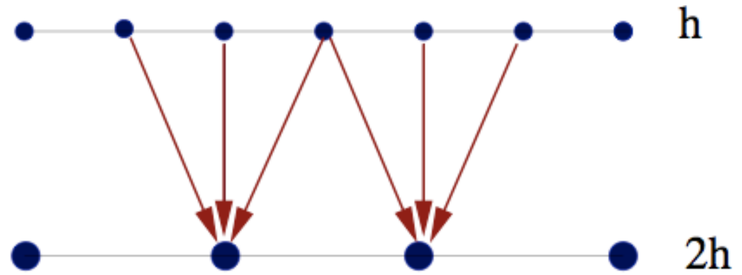
INGREDIENTS FOR PARALLEL ADAPTIVE MULTIGRID

- Multigrid setup
  - Block orthogonalization of null space vectors
  - Batched QR decomposition
- Smoothing (relaxation on a given grid)
  - Repurpose existing solvers
- Prolongation
  - interpolation from coarse grid to fine grid
  - one-to-many mapping
- Restriction
  - restriction from fine grid to coarse grid
  - many-to-one mapping
- Coarse Operator construction (setup)
  - Evaluate $R\,A\,P$ locally
  - Batched (small) dense matrix multiplication
- Coarse grid solver
  - Need optimal coarse-grid operator

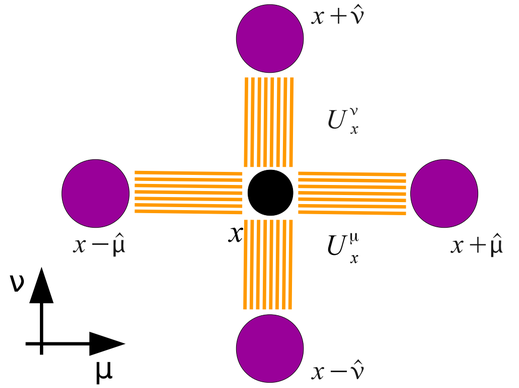








COARSE GRID OPERATOR

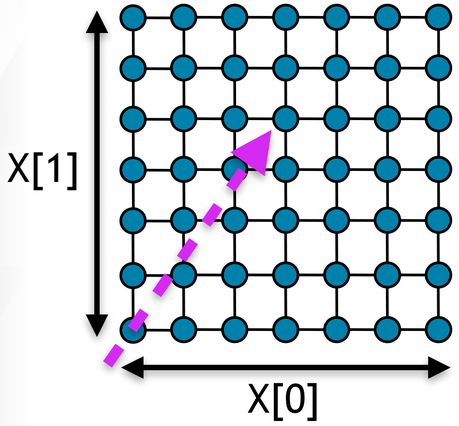


- Coarse operator looks like a Dirac operator (many more colors)
  - Link matrices have dimension $2N_v$ x $2N_v$ (e.g., 48 x 48)

$$\hat{D}_{\mathbf{i}\hat{s}\hat{c},\mathbf{j}\hat{s}'\hat{c}'} = -\sum_{\mu}\left[Y^{-\mu}_{\mathbf{i}\hat{s}\hat{c},\mathbf{j}\hat{s}'\hat{c}'}\delta_{\mathbf{i}+\mu,\mathbf{j}} + Y^{+\mu\dagger}_{\mathbf{i}s\hat{c},\mathbf{j}s'\hat{c}'}\delta_{\mathbf{i}-\mu,\mathbf{j}}\right] + \left(M - X_{\mathbf{i}\hat{s}\hat{c},\mathbf{j}\hat{s}'\hat{c}'}\right)\delta_{\mathbf{i}\hat{s}\hat{c},\mathbf{j}\hat{s}'\hat{c}'}.$$

- Fine vs. Coarse grid parallelization
  - Fine grid operator has plenty of grid-level parallelism
    - E.g., 16x16x16x16 = 65536 lattice sites
  - Coarse grid operator has diminishing grid-level parallelism
    - first coarse grid 4x4x4x4= 256 lattice sites
    - second coarse grid 2x2x2x2 = 16 lattice sites
- Current GPUs have up to 3840 processing elements
- Need to consider finer-grained parallelization
  - Increase parallelism to use all GPU resources
  - Load balancing

SOURCE OF PARALLELISM



1. Grid parallelism

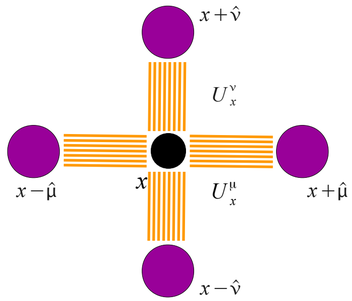
Volume of threads

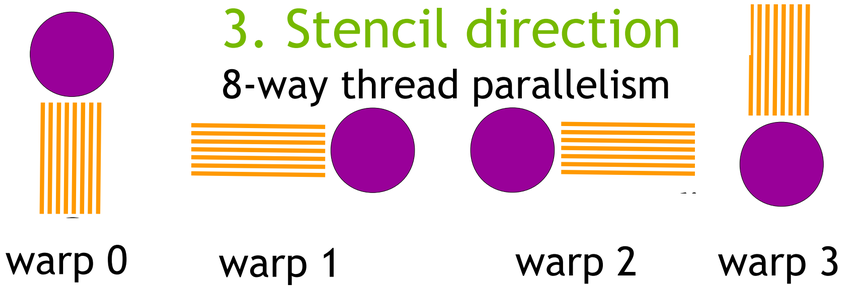
thread y index

$$\begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix} += \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

2. Link matrix-vector partitioning

2 N_{vec}-way thread parallelism (spin * color)





3. Stencil direction

8-way thread parallelism

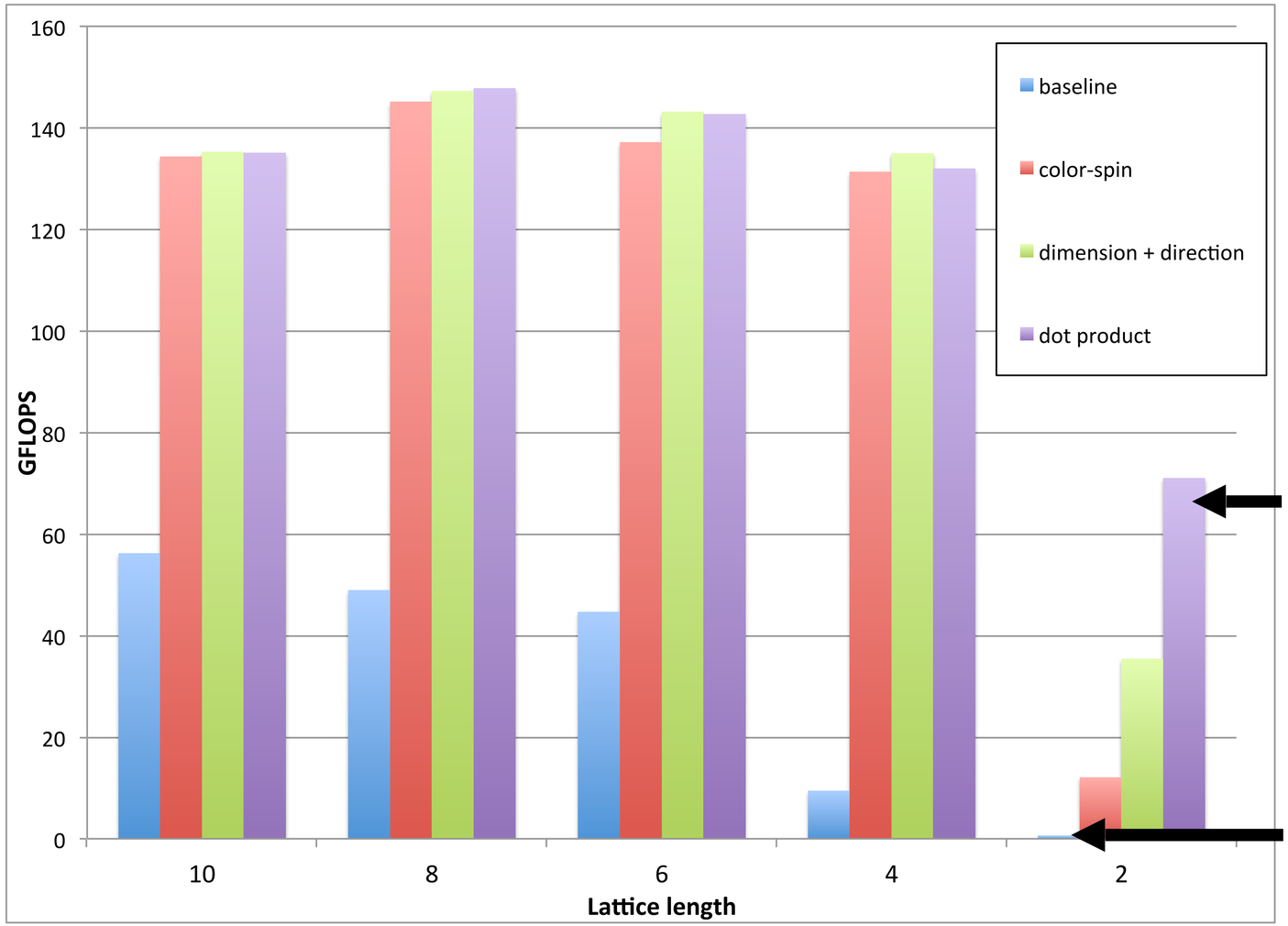
$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} \Rightarrow \begin{pmatrix} a_{00} & a_{01} \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \end{pmatrix} + \begin{pmatrix} a_{02} & a_{03} \end{pmatrix} \begin{pmatrix} b_2 \\ b_3 \end{pmatrix}$$

4. Dot-product partitioning

4-way thread parallelism + ILP

COARSE GRID OPERATOR PERFORMANCE

Tesla K20X (Titan), FP32, N_{vec} = 24



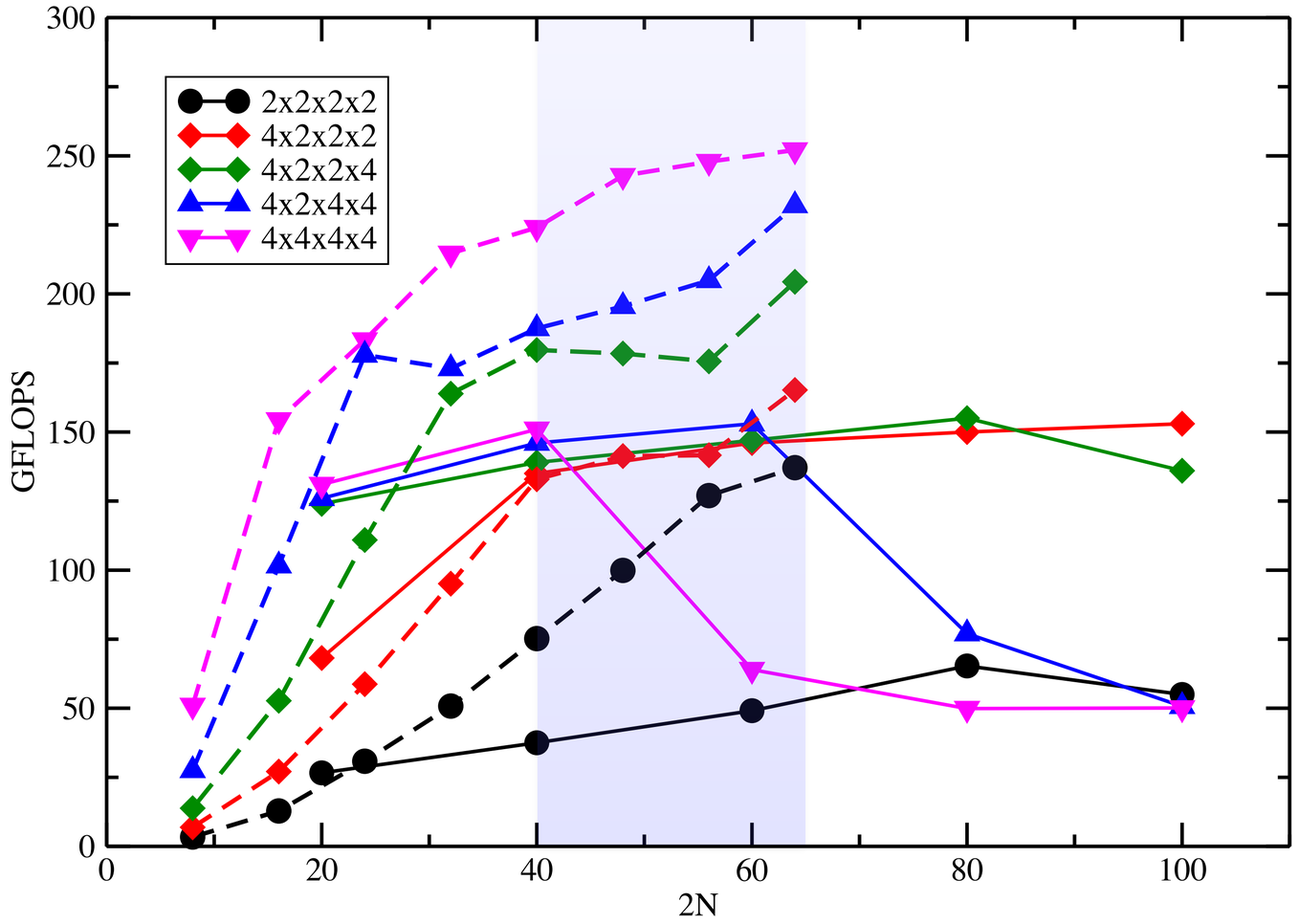
24,576-way parallel

16-way parallel

NVIDIA

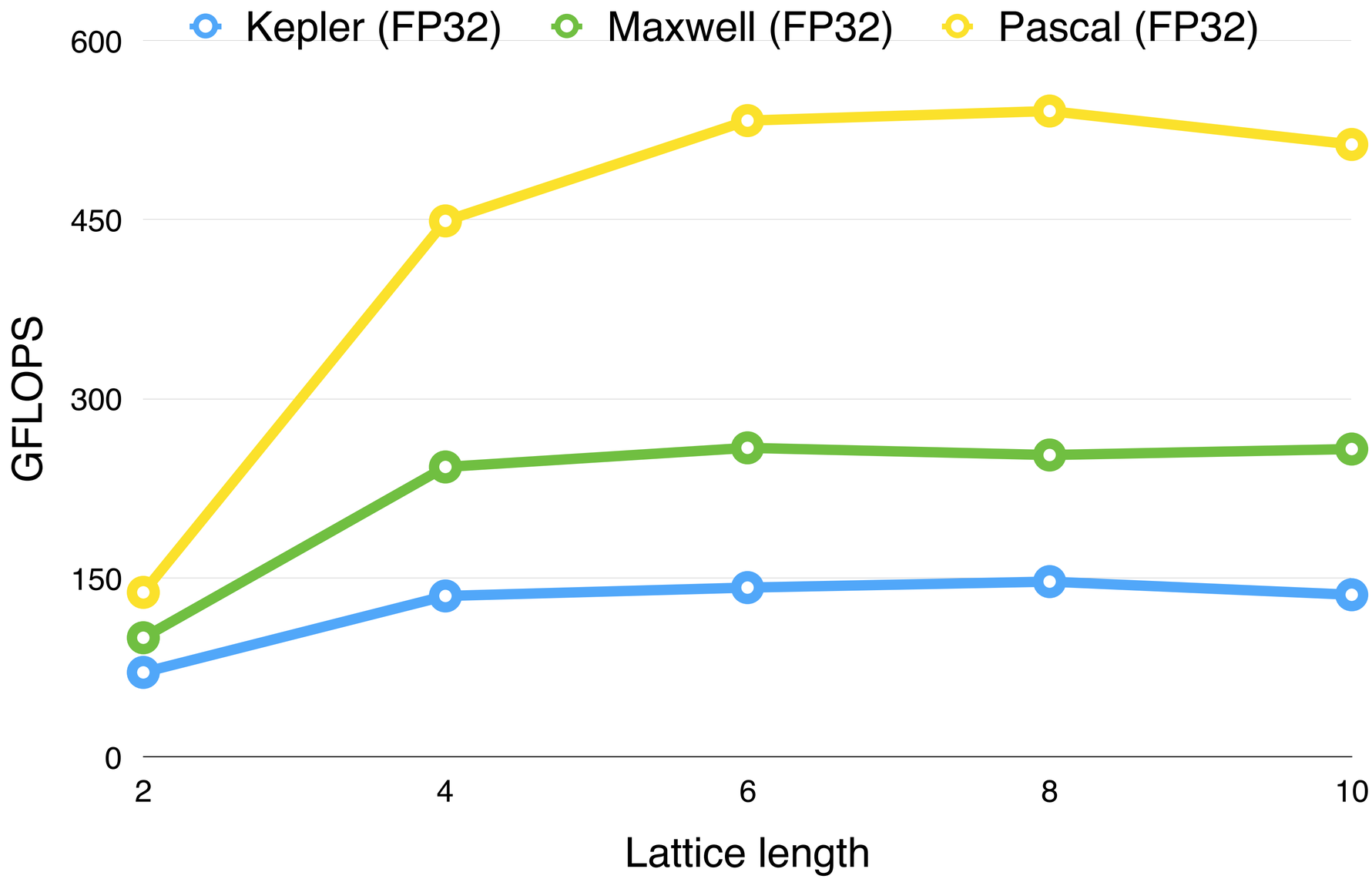
COARSE GRID OPERATOR PERFORMANCE

8-core Haswell 2.4 GHz (solid line) vs M6000 (dashed lined), FP32

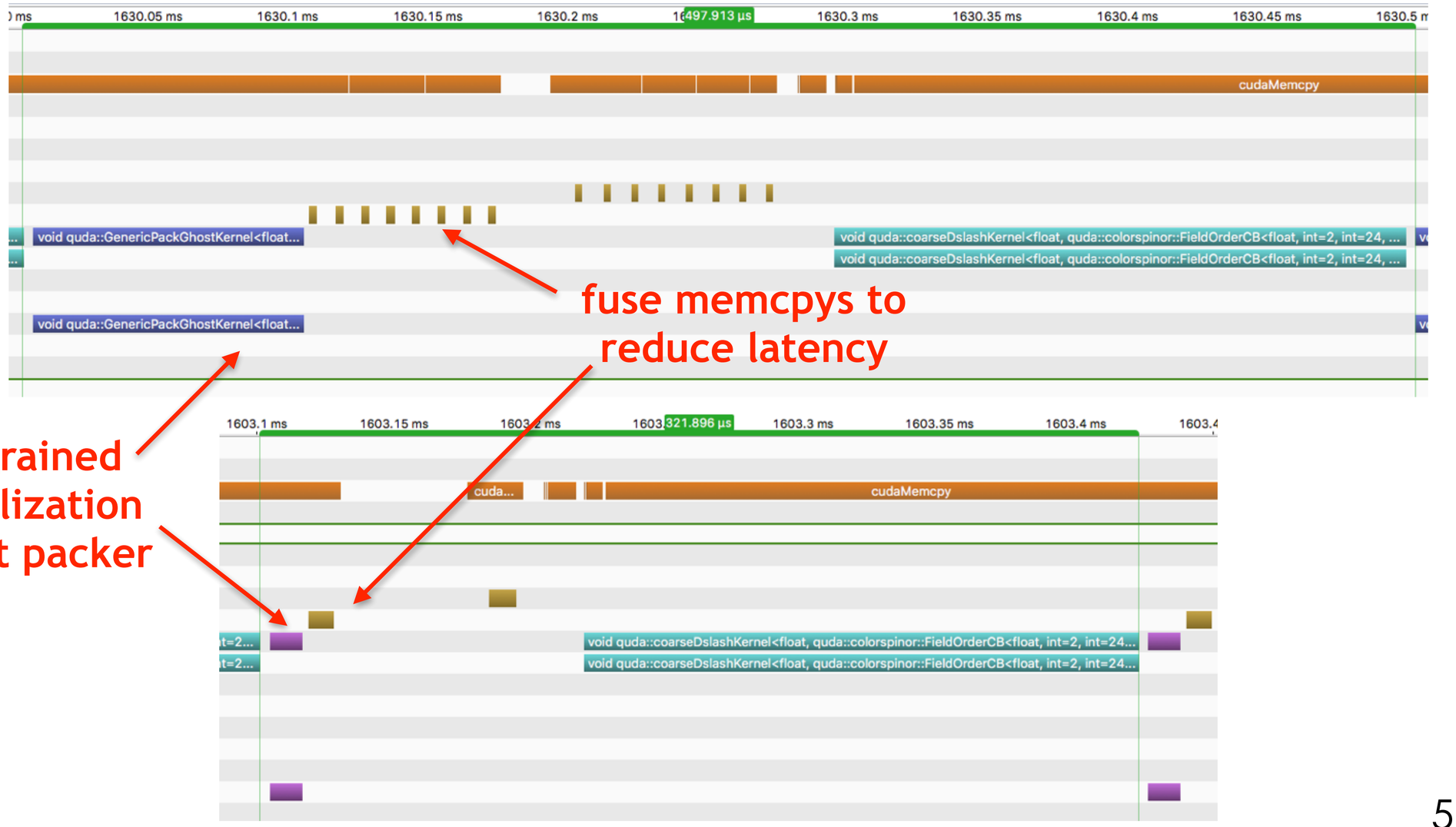


- Autotuner finds optimum degree of parallelization
  - Larger grids favor less fine grained
  - Coarse grids favor most fine grained
- GPU is nearly always faster than CPU
- Expect in future that coarse grids will favor CPUs
- For now, use GPU exclusively

COARSE GRID OPERATOR PERFORMANCE

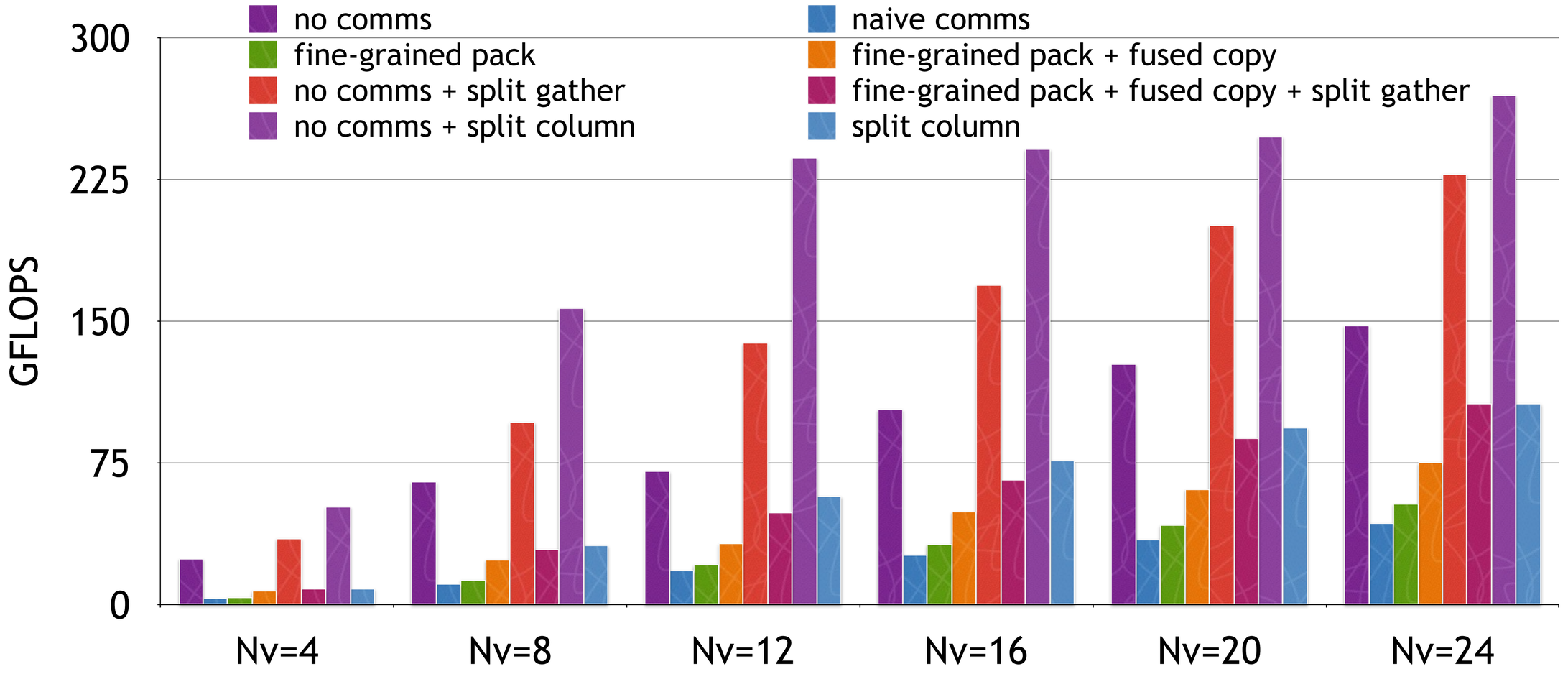


IMPROVING STRONG SCALING



IMPROVING STRONG SCALING

$V_{coarse} = 4^4$, 8-way communication, FP32, Quadro M6000



NVIDIA

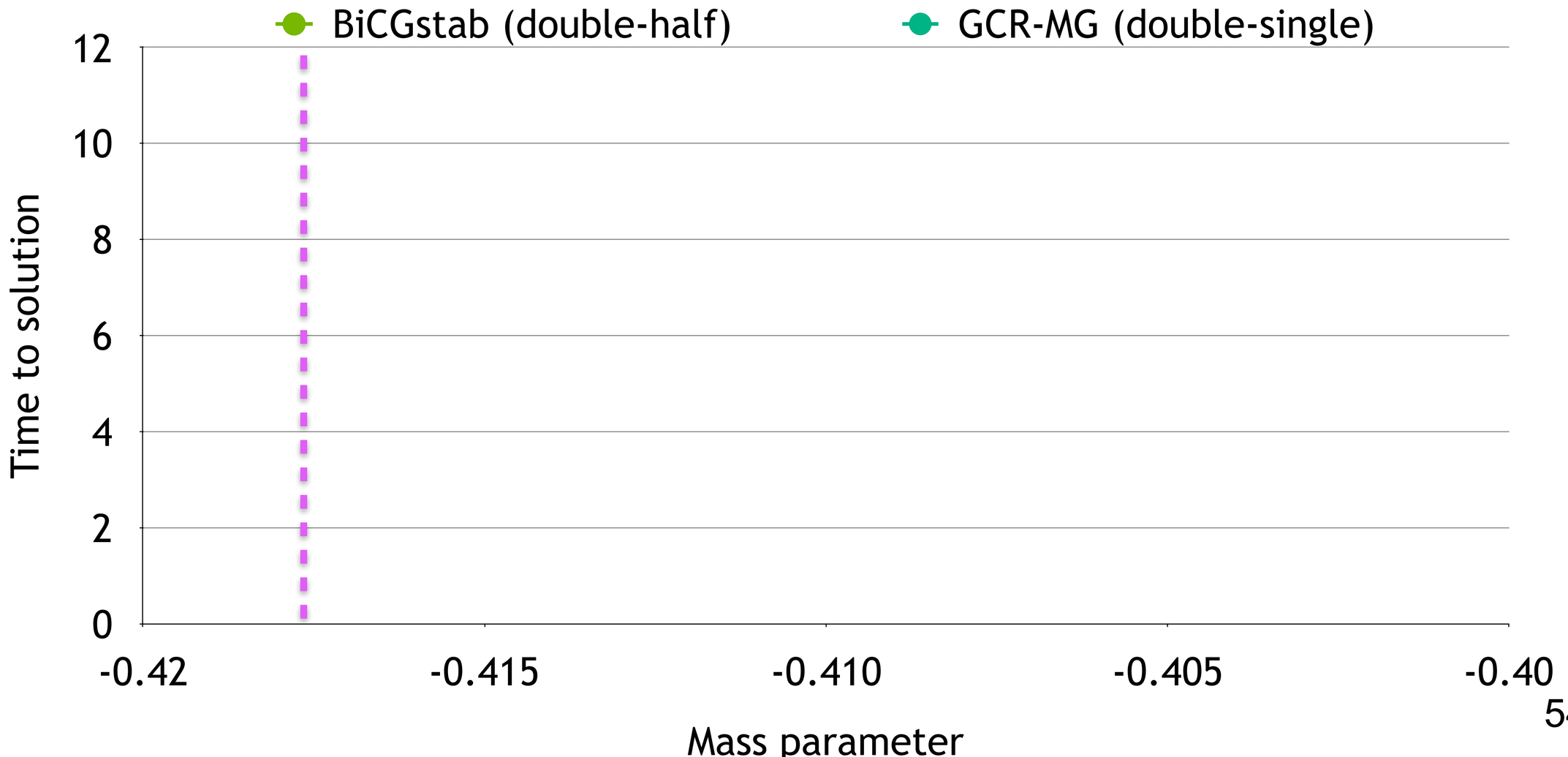
MULTIGRID VERSUS BICGSTAB

- Compare MG against the state-of-the-art traditional Krylov solver on GPU
  - BiCGstab in double/half precision with reliable updates
  - 12/8 reconstruct
  - Symmetric red-black preconditioning $\hat{M}_{ee} = 1_{ee} - \kappa^2 A_{ee}^{-1} D_{eo} A_{oo}^{-1} D_{oe}$

- Adaptive Multigrid algorithm
  - GCR outer solver wraps 3-level MG preconditioner
  - GCR restarts done in double, everything else in single
  - 24 null-space vectors on fine grid
  - Minimum Residual smoother
  - Symmetric red-black preconditioning on each level
  - GCR coarse-grid solver

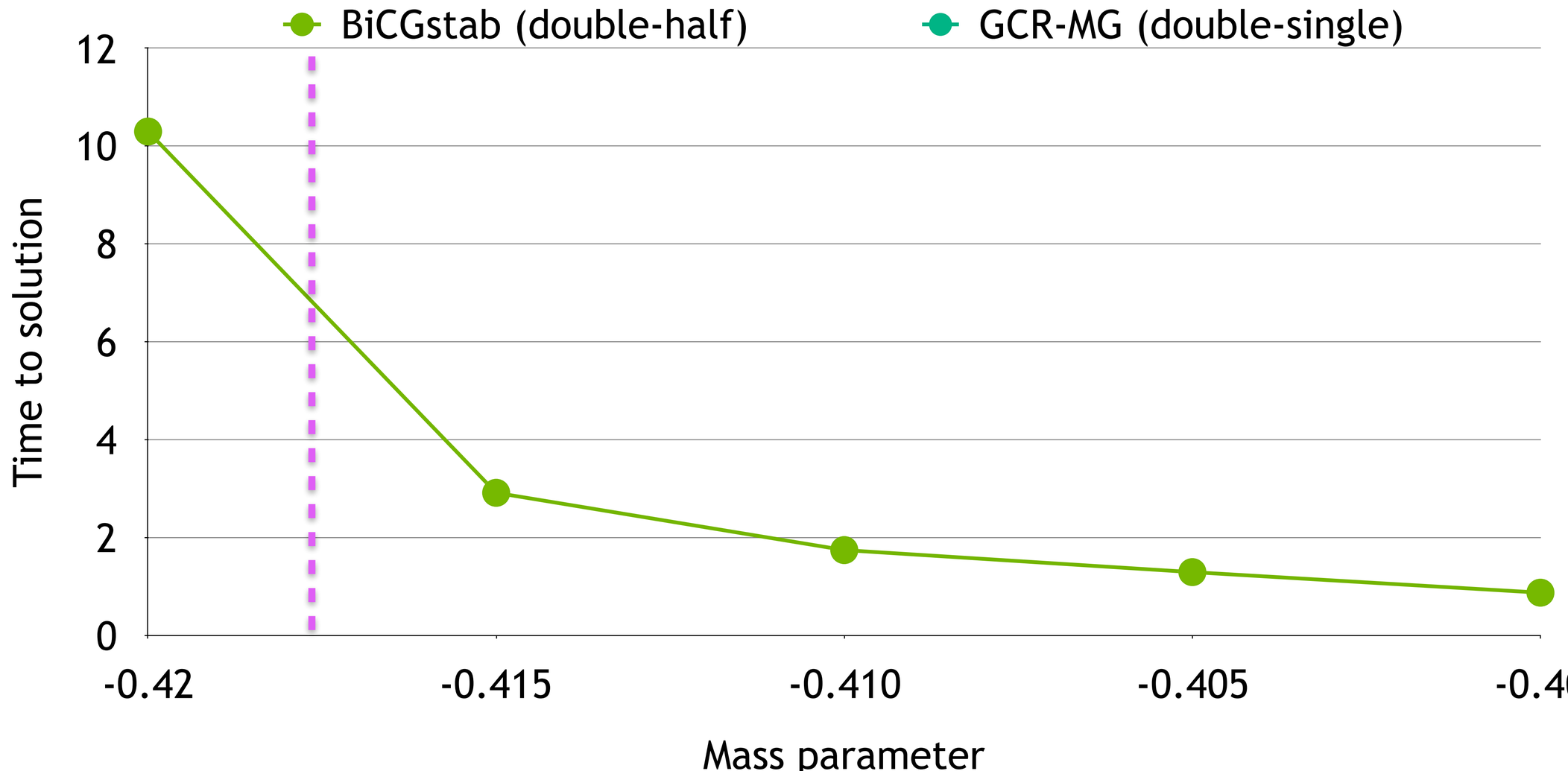
MULTIGRID VERSUS BICGSTAB

Wilson, V = 24^3x64, single workstation (3x M6000)



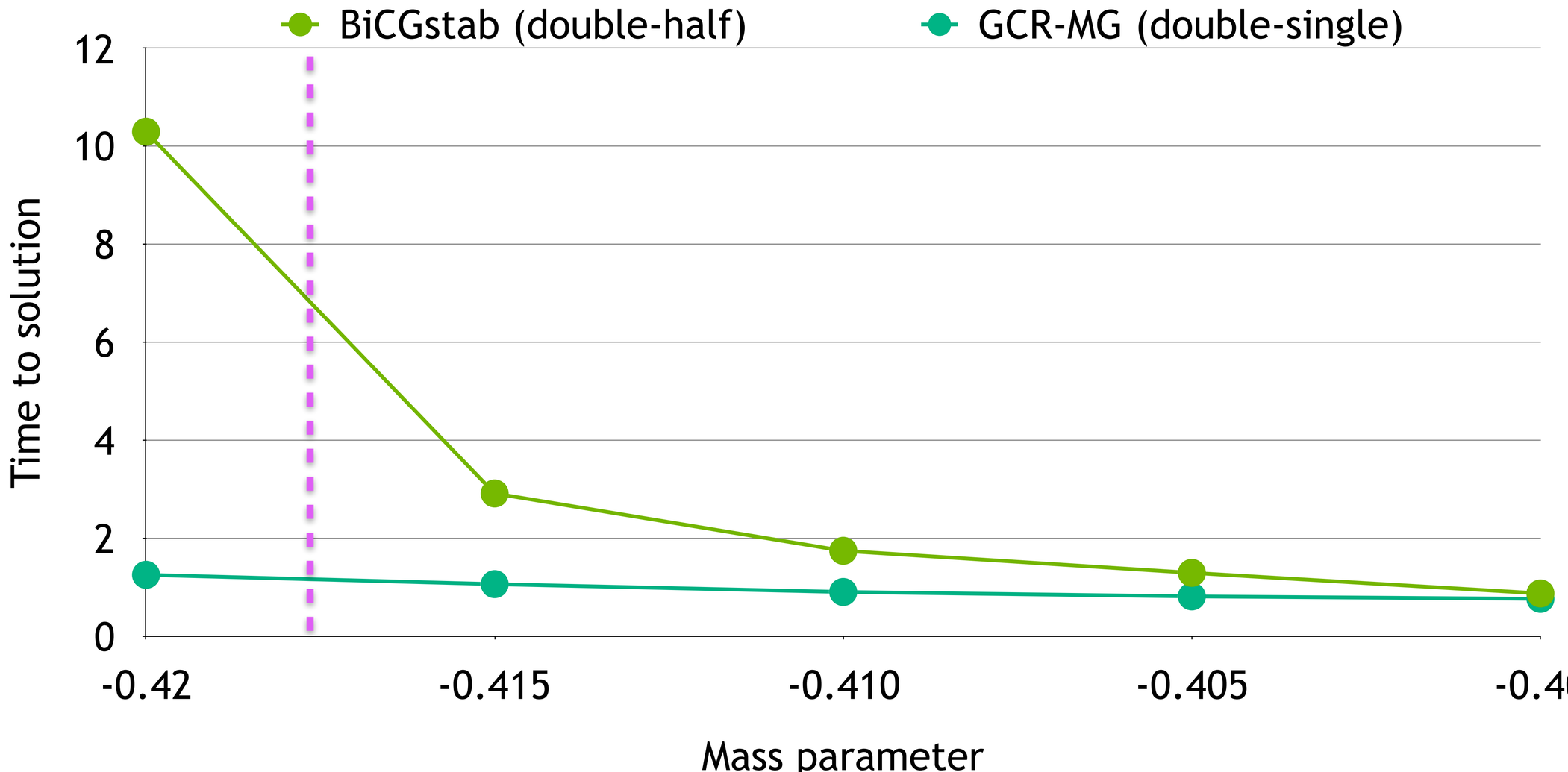
MULTIGRID VERSUS BICGSTAB

Wilson, V = 24^3x64, single workstation (3x M6000)



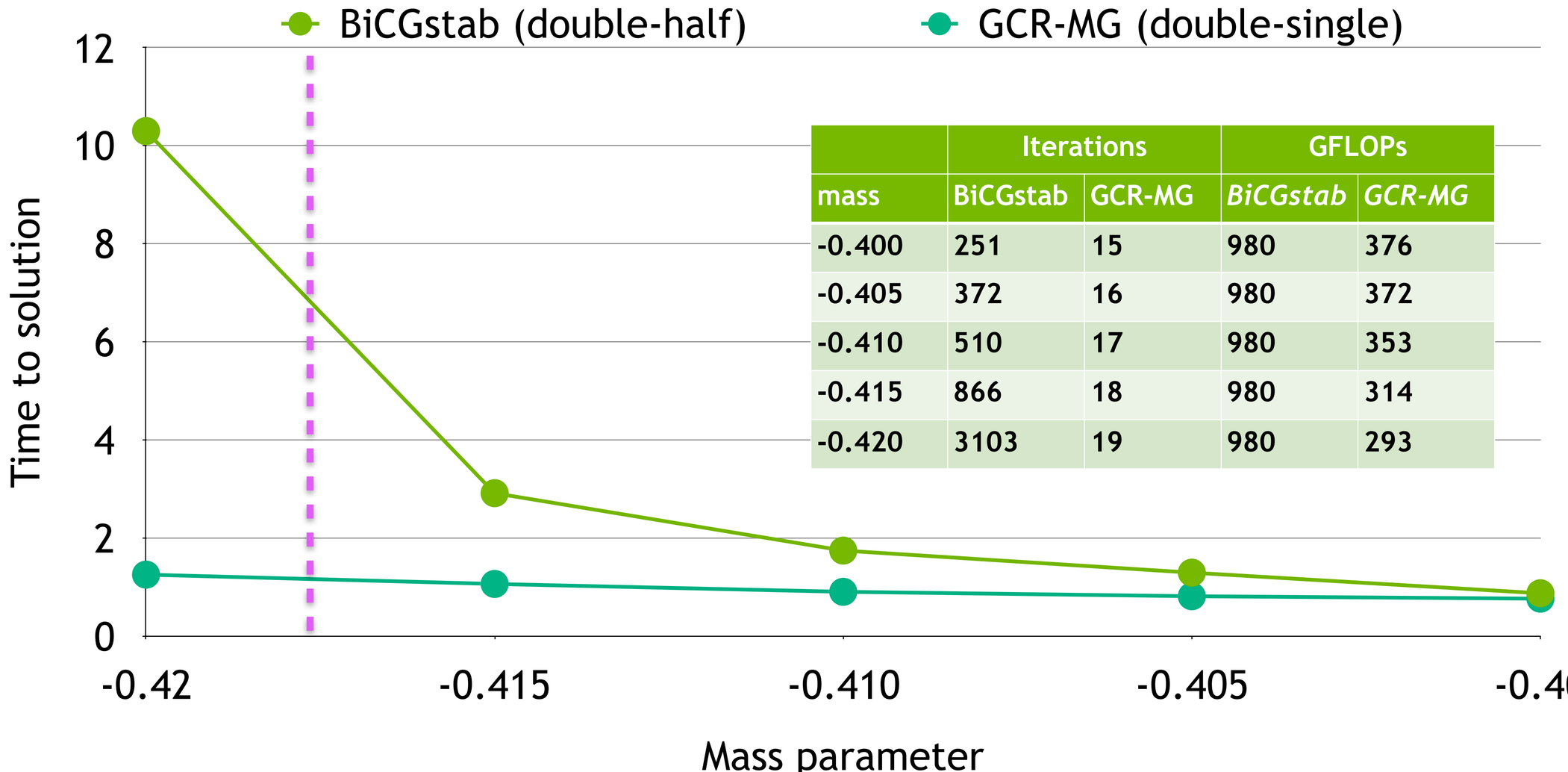
MULTIGRID VERSUS BICGSTAB

Wilson, V = 24^3x64, single workstation (3x M6000)

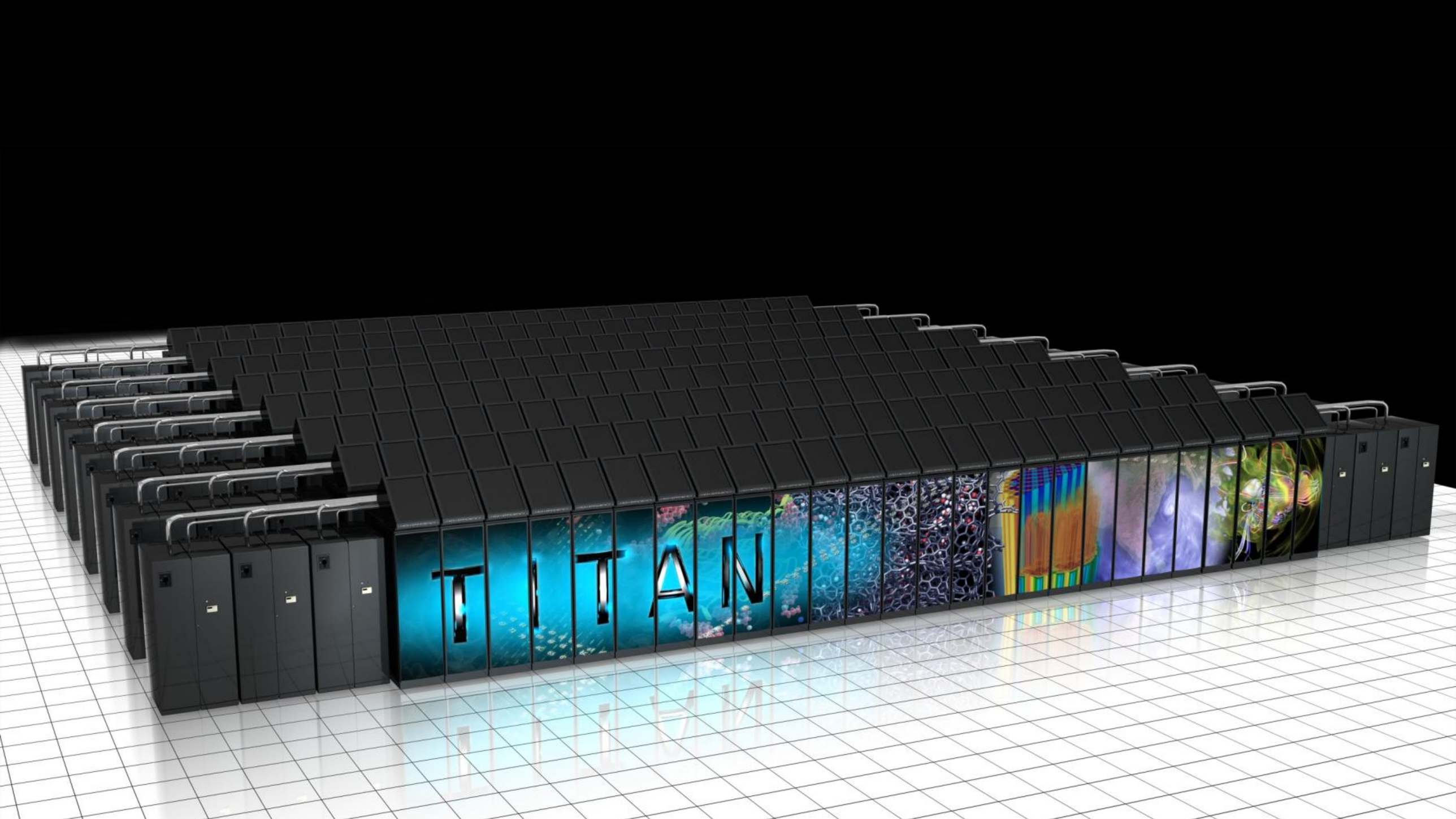


MULTIGRID VERSUS BICGSTAB

Wilson, V = 24^3x64, single workstation (3x M6000)

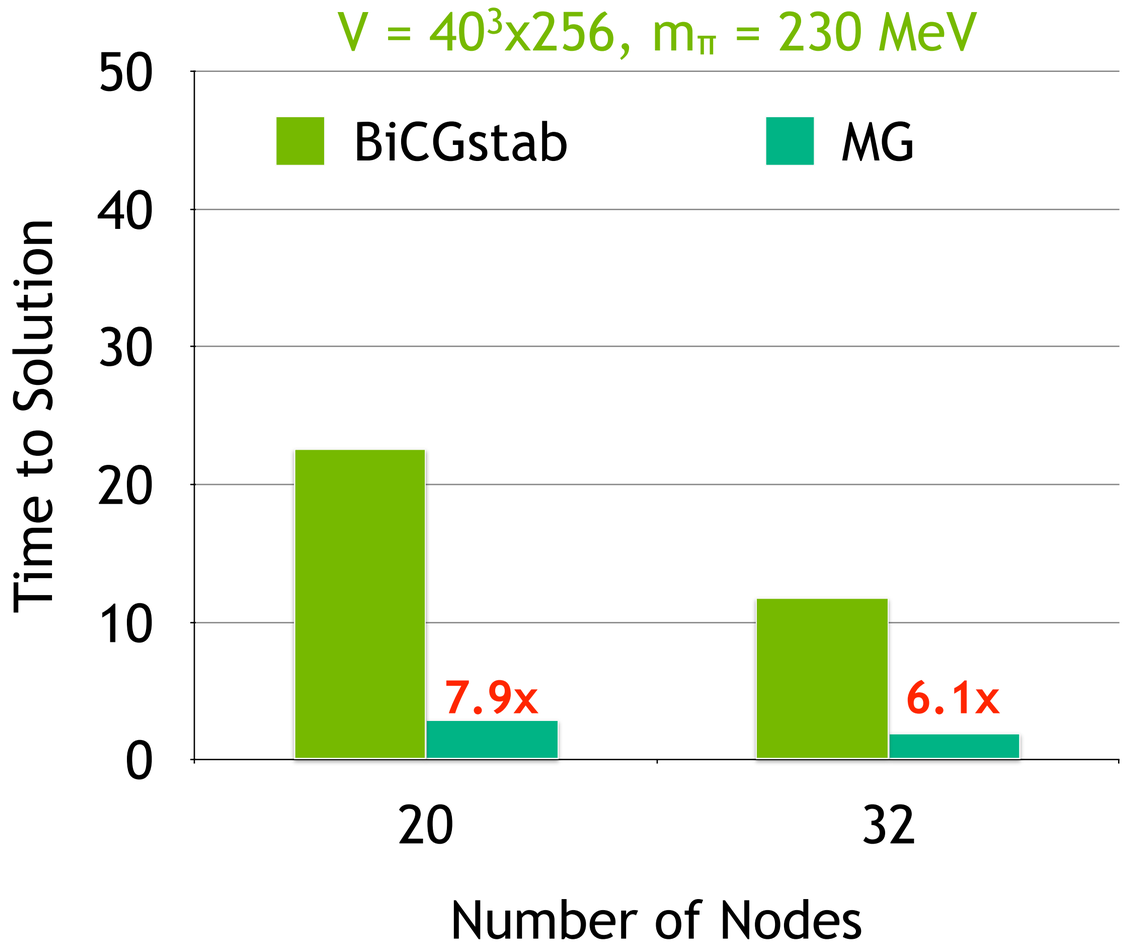


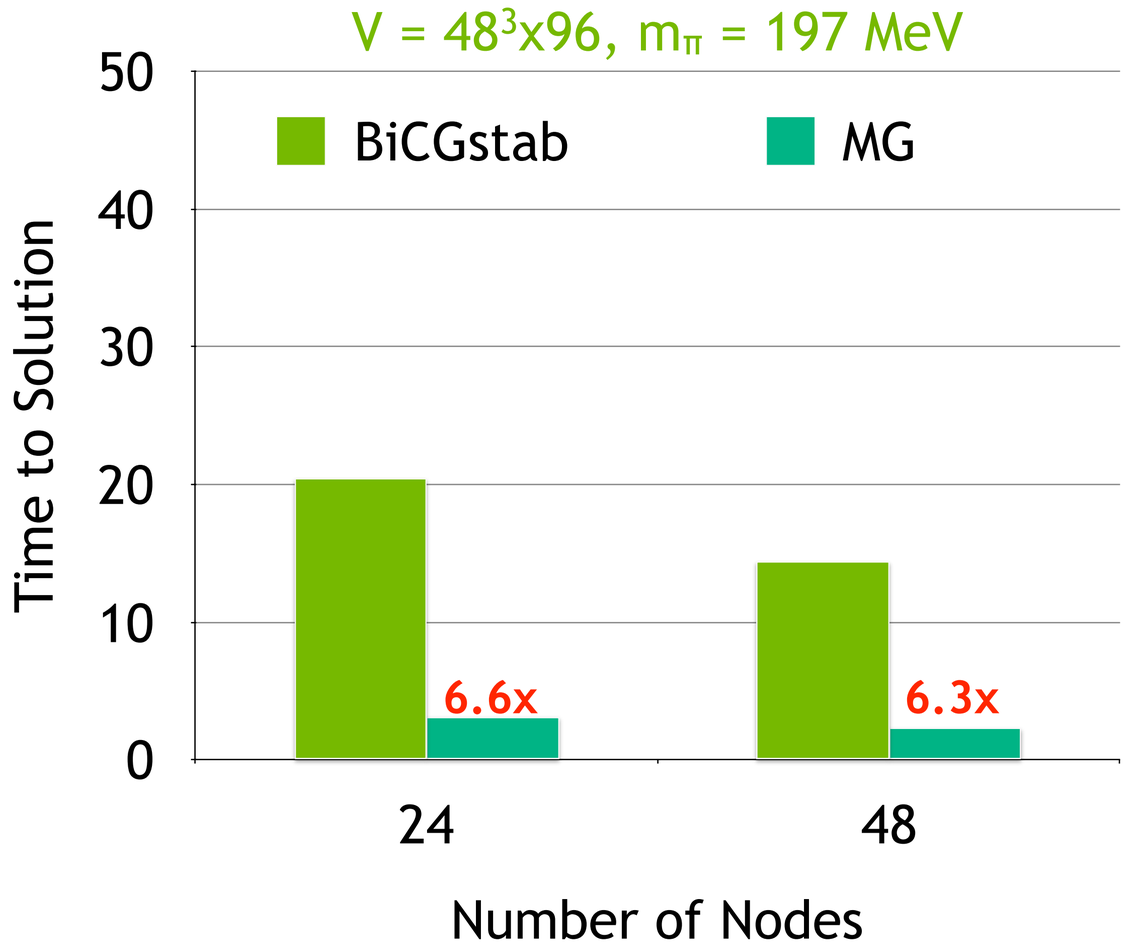
	Iterations		GFLOPs	
mass	BiCGstab	GCR-MG	*BiCGstab*	*GCR-MG*
-0.400	251	15	980	376
-0.405	372	16	980	372
-0.410	510	17	980	353
-0.415	866	18	980	314
-0.420	3103	19	980	293


TITAN

MULTIGRID VERSUS BICGSTAB

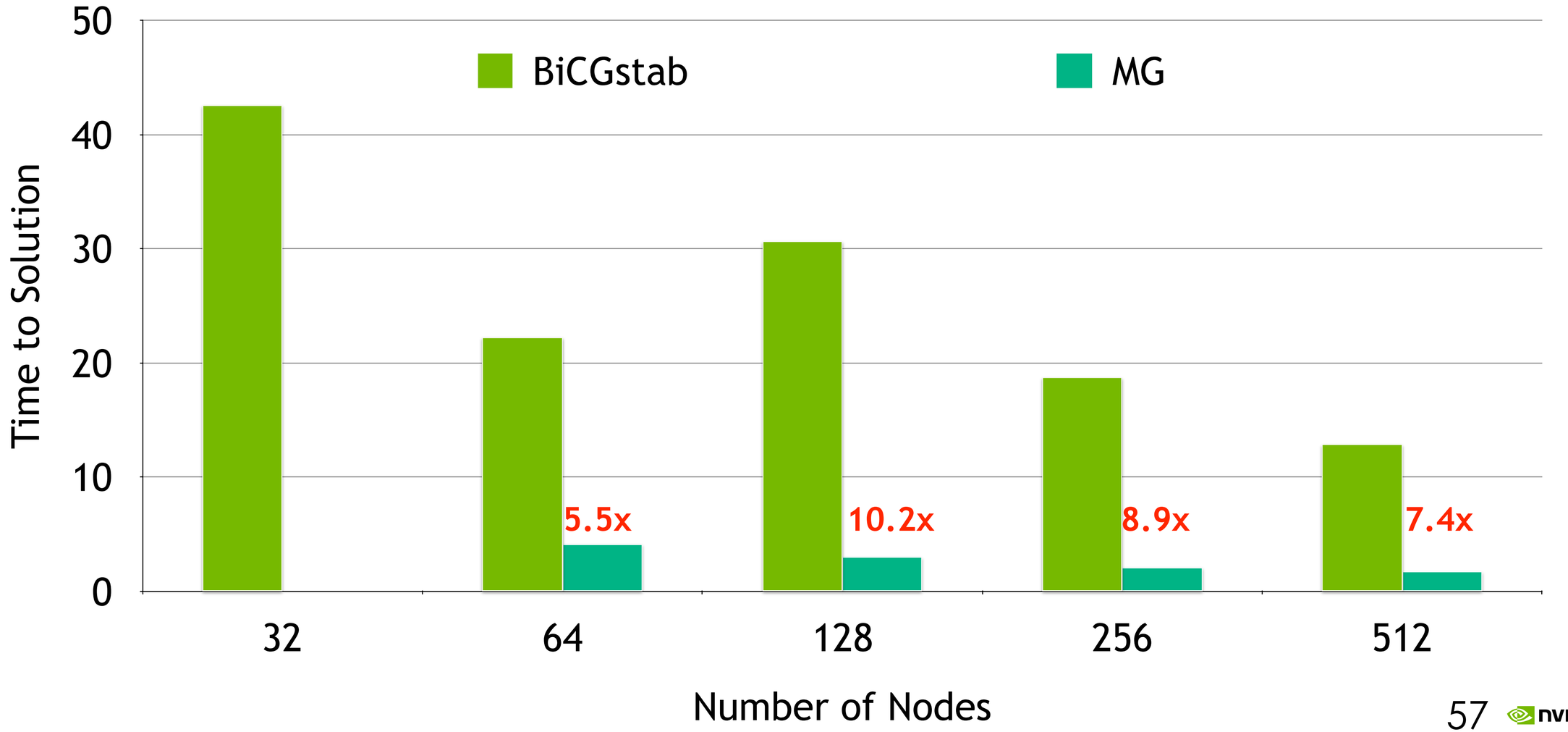
Wilson-clover, Strong scaling on Titan (K20X)





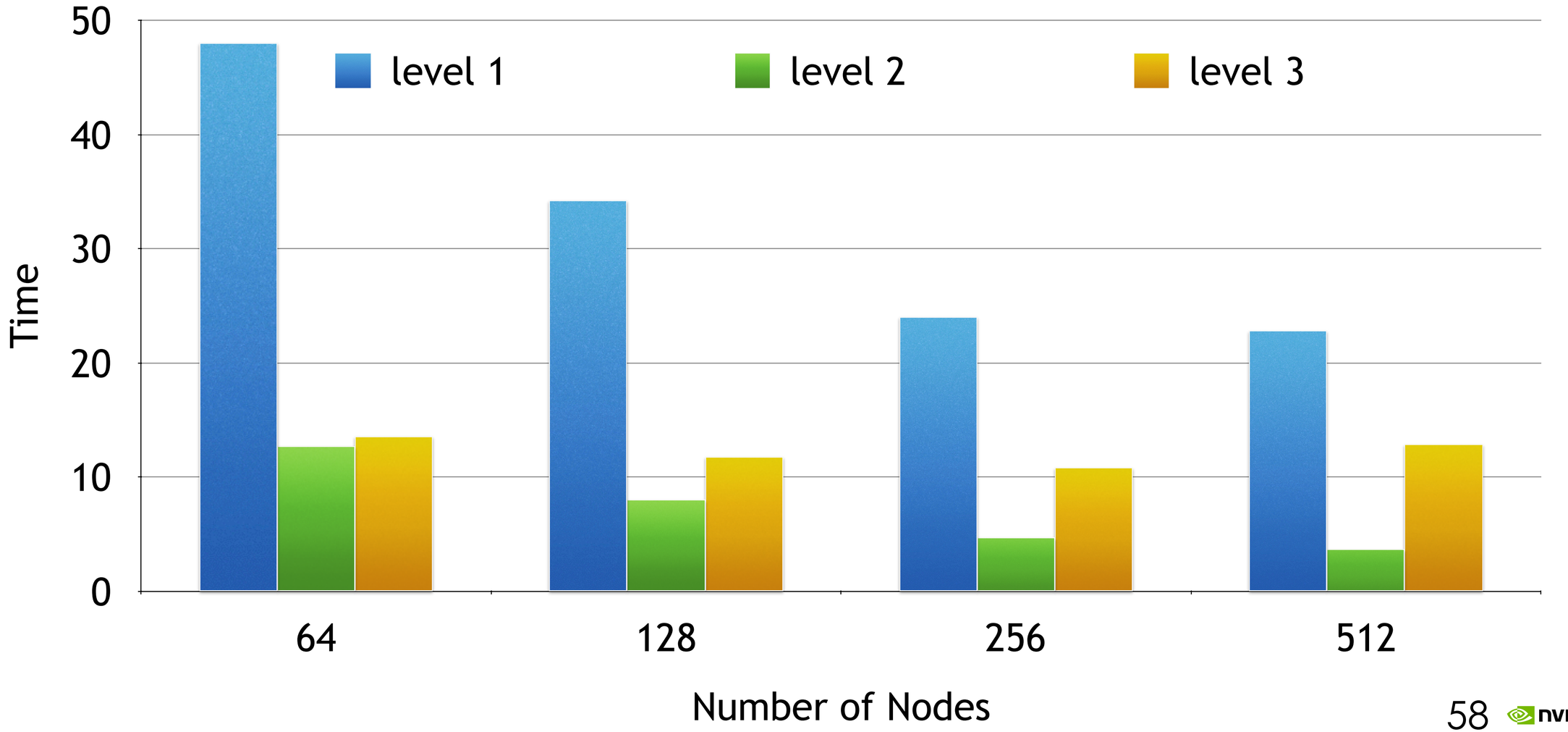
MULTIGRID VERSUS BICGSTAB

Wilson-clover, Strong scaling on Titan (K20X), V = 64^3x128, m_π = 197 MeV



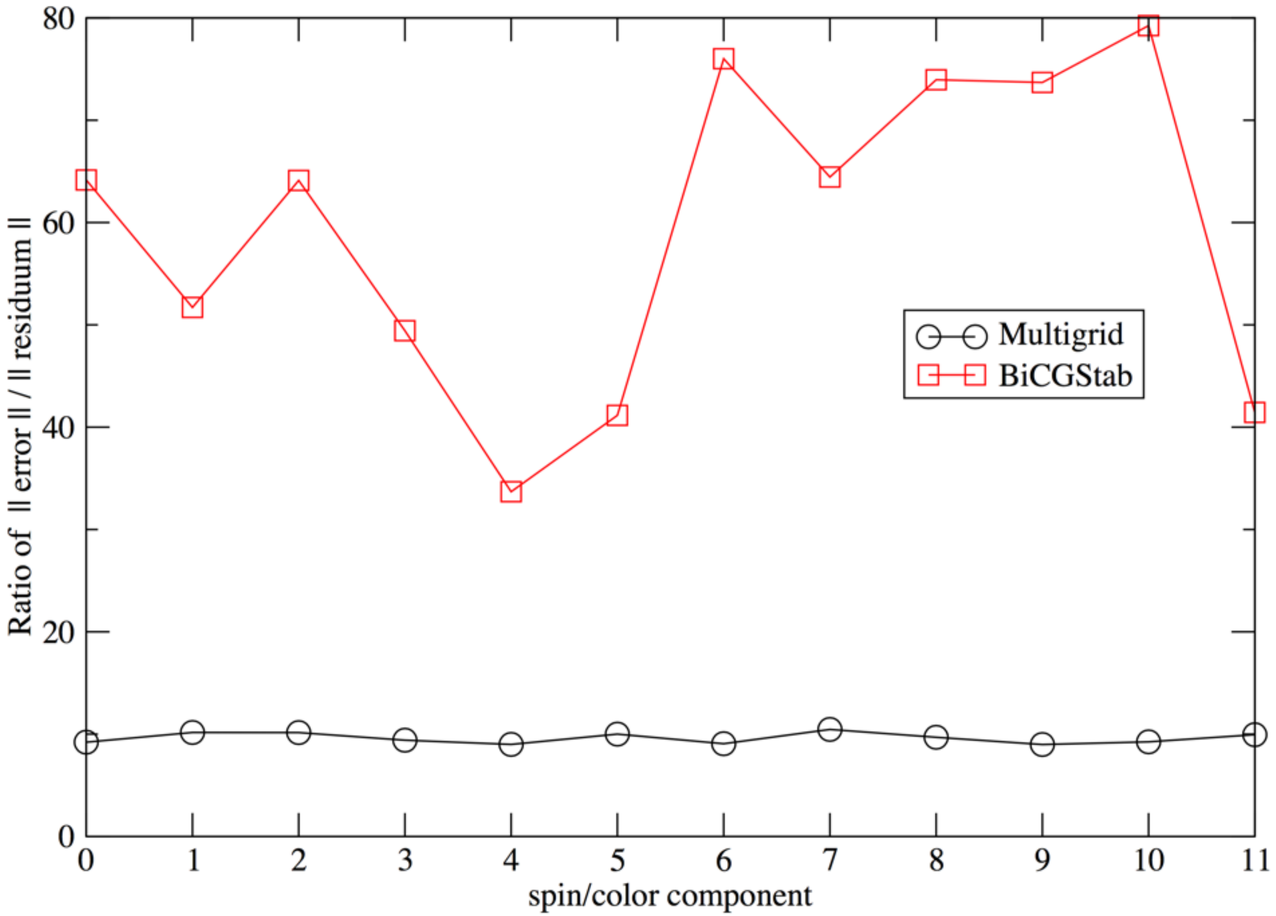
MULTIGRID TIMING BREAKDOWN

Wilson-clover, Strong scaling on Titan (K20X), V = 64^3x128, 12 linear solves



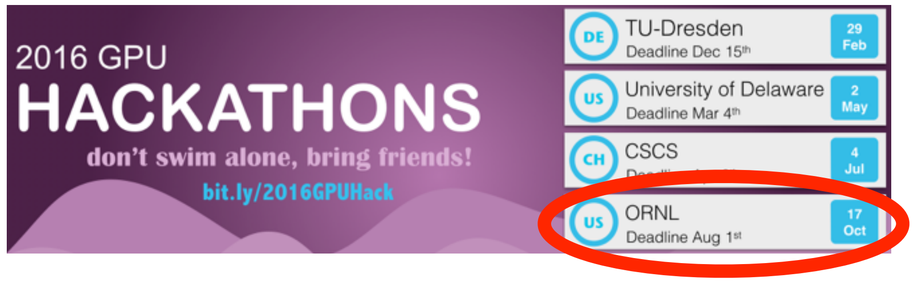
ERROR REDUCTION AND VARIANCE

$V = 40^3 x 256$, $m_\pi = 230$ MeV

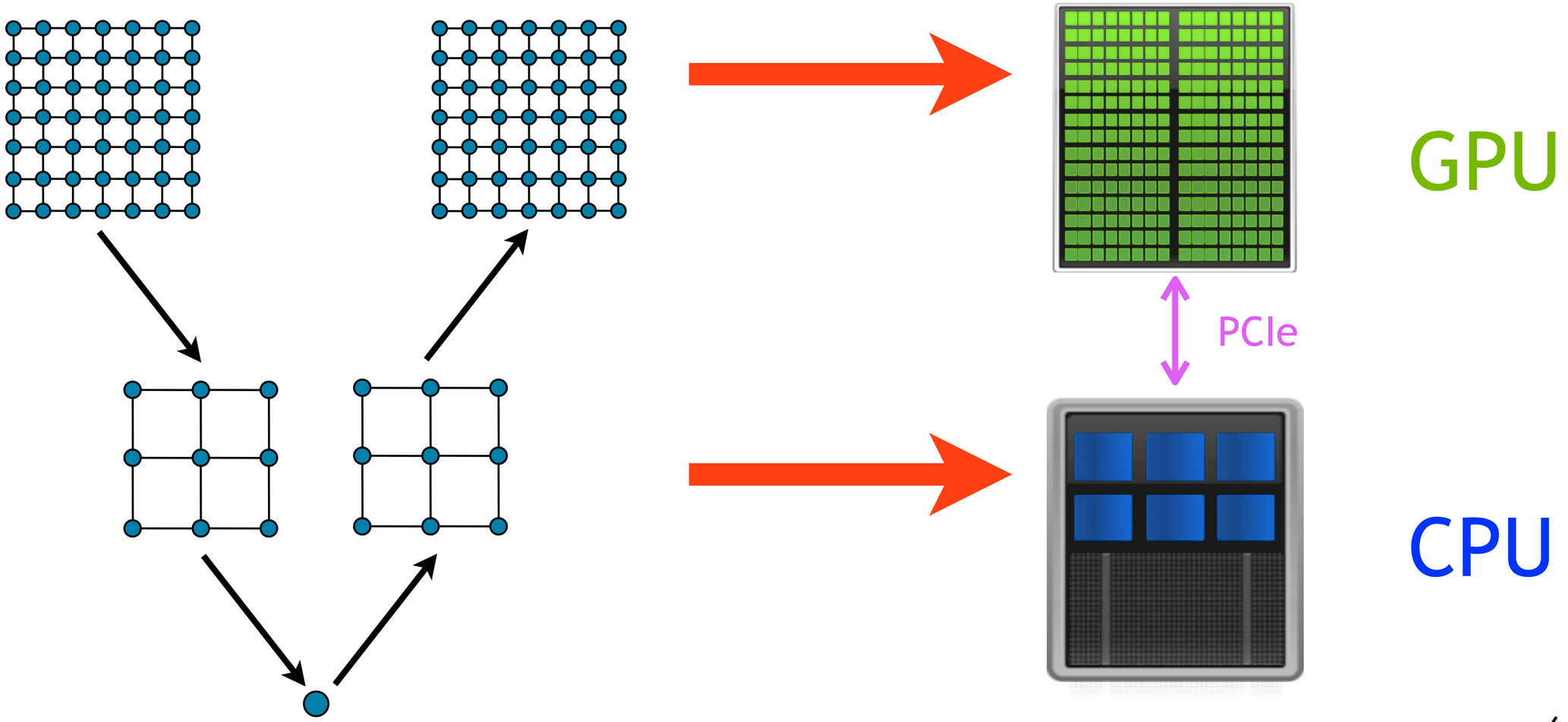


MULTIGRID FUTURE WORK

- Absolute Performance tuning, e.g., half precision on coarse grids
- Strong scaling improvements:
  - Combine with Schwarz preconditioner
  - Accelerate coarse grid solver: CA-GMRES instead of GCR, deflation
  - More flexible coarse grid distribution, e.g., redundant nodes
- Investigate off load of coarse grids to the CPU
  - Use CPU and GPU simultaneously using additive MG
- Full off load of setup phase to GPU - required for HMC

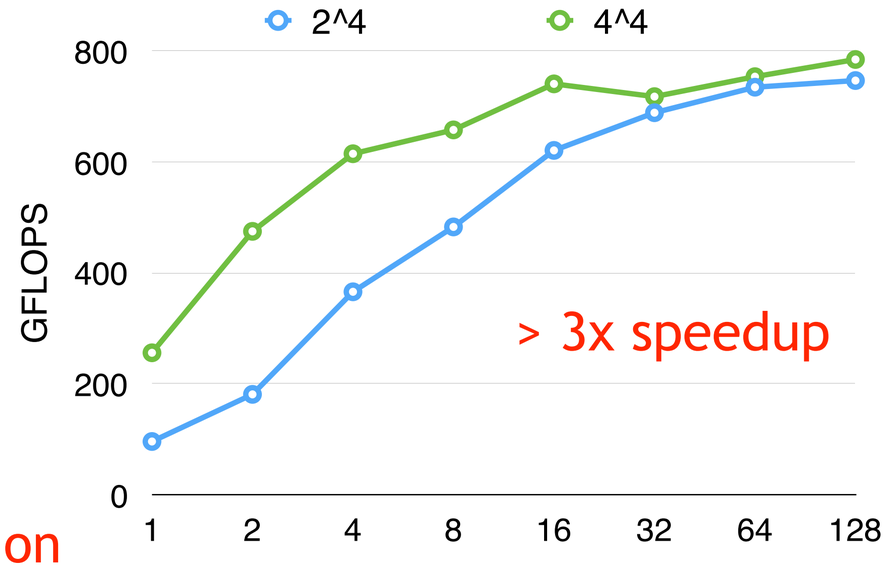


HIERARCHICAL ALGORITHMS ON HETEROGENEOUS ARCHITECTURES



MULTI-SRC SOLVERS

- Multi-src solvers increase locality through link-field reuse
- Multi-grid operators even more so since link matrices are 48x48
  - Coarse Dslash / Prolongator / Restrictor
- Coarsest grids also latency limited
  - Kernel level latency
  - Network latency
- Multi-src solvers are a solution
  - More parallelism
  - Bigger messages



Coarse dslash on M6000 GPU vs #rhs

TO THE EXASCALE AND BEYOND

SOFTWARE AND ALGORITHMS

- Algorithms continue to innovate rapidly inside and outside of LQCD
  - E.g., Krylov solvers
    - Communication avoiding solvers (Demmel *et al*)
    - Cooperative Krylov methods (Bhaya *et al*)
    - Enlarged Krylov space methods (Grigori *et al*)
- Software can be the problem
  - Hierarchical / overlapping grids break most LQCD frameworks
  - Used to calling solvers in serial fashion
  - Precision often baked in

FINE-GRAINED PARALLELISM AND THE IMPLICATIONS FOR DLLS

- Traditional DSL approach is to abstract the grid parallelism

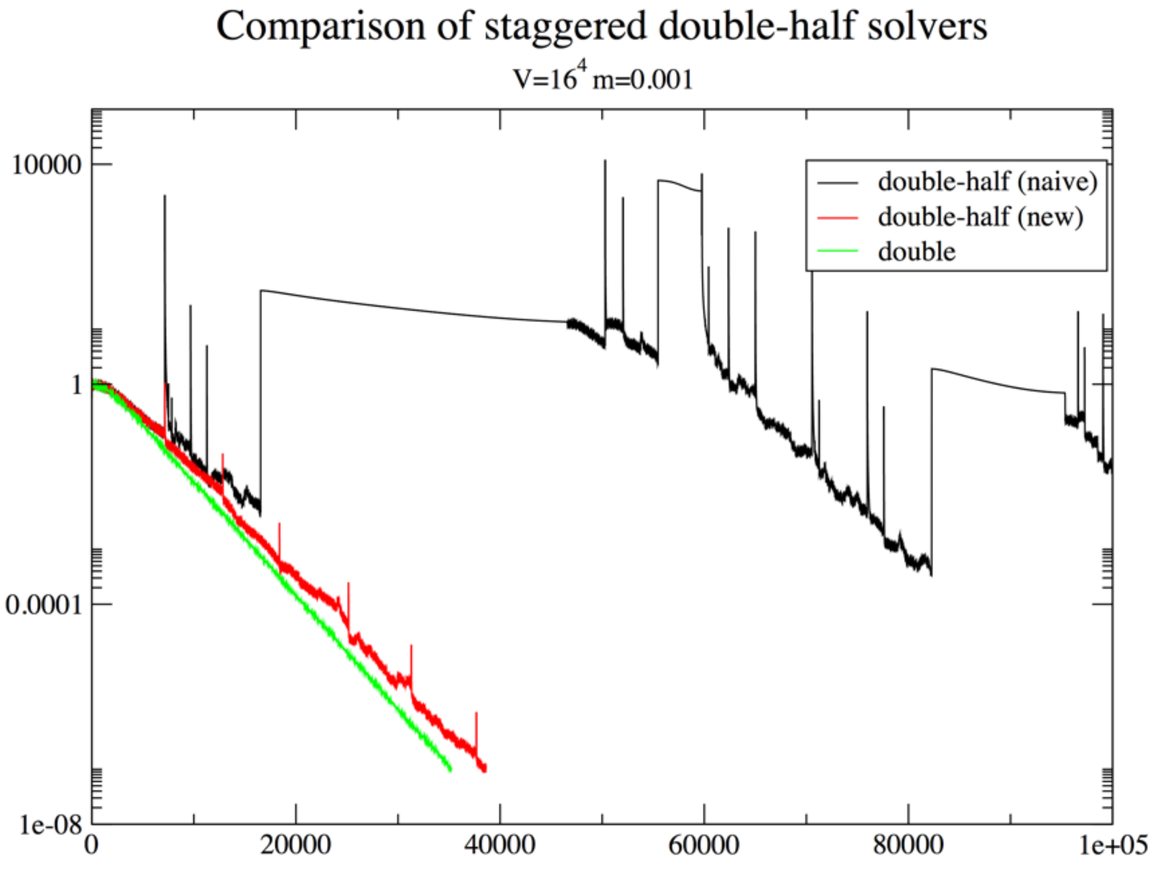
```
Matrix u;
Vector x, y;
y = u * x;
```

- Compiler / front end will then transform this expression into a data parallel operation using OpenMP / CUDA / C++ meta template magic, etc.
- This abstraction breaks with multigrid
  - Not enough grid parallelism
- Platform and algorithmic independent conjecture

"Fine-grained parallelization will becoming increasingly a requirement at the Exascale (and beyond)"

PRECISION

- How much precision is really required?
  - 16-bit solvers with high precision reliable updates (arXiv:0911.3191)
  - Truncate trailing mantissa bits in halo exchange (P. Boyle, arXiv:1402.2585)
  - Use fp16 for coarse grid solve (Heybrock *et al*, arXiv:1512.04506)
- Large HMC runs with tolerance ~10^{-13}
  - What happens when the volumes get bigger?
- Precision optimization an important part of the puzzle



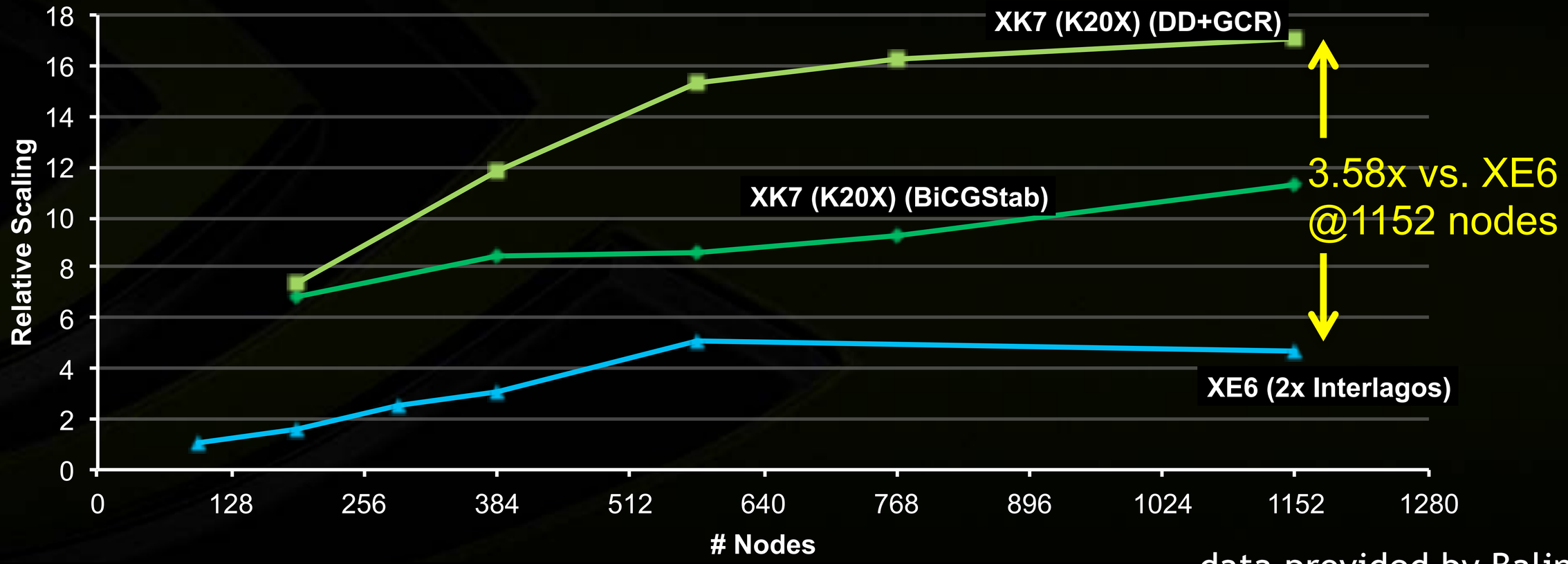
COMMUNICATION

Chroma

48^3x512 lattice

Relative Scaling (Application Time)

"XK7" node = XK7 (1x K20X + 1x Interlagos)
"XE6" node = XE6 (2x Interlagos)



data provided by Balint Joo

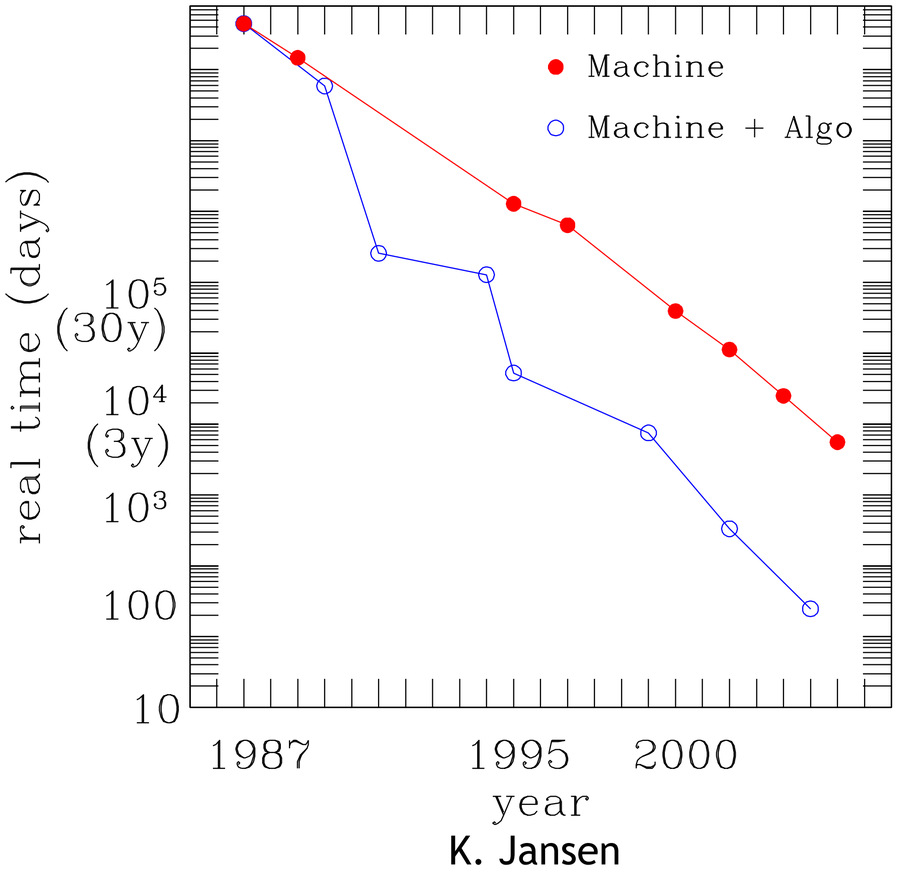
TO THE EXASCALE AND BEYOND

- (At least) four challenges to overcome
  - Parallelism
  - Locality
  - Communication
  - Latency

- What's the answer to the above?

Algorithms

*and the ability to express those algorithms



K. Jansen

HMC AND STREAM PARALLELISM

- Network bandwidth increasing becoming a limiting factor
- Starting to see hierarchy of network bandwidths
  - CORAL: fat nodes with thin network
- HMC requires strong scaling
  - HMC has a flat communications profile
  - Limited by slowest connection
- Split determinants and impose task parallelism
  - Each fat node computes one determinant contribution
  - Eliminates slow connection from fermion solver

$$\int dU e^{-S_g(U)} \det(\mathcal{M}) = \int dU e^{-S_g(U)} \prod_{i=1}^{n} \det(\mathcal{M}^{1/n})$$