Brookhaven National Laboratory

Computational and algorithmic advances using Grid

Peter Boyle, Brookhaven National Laboratory

- RBC-UKQCD simulation plans
- Field Transformation HMC implementation and status
- Multiple RHS multigrid for MDWF
- Grid and GPU portability
  - Data layout: SIMD vs SIMT
  - Offload primitives
  - Data motion: O(1) software managed cache
  - Performance on Exascale systems

SciDAC-5 personnel

https://scidac5-fastmath.lbl.gov/
https://petsc.org/release/

SciDAC:

- MIT - Youssef Marzouk (FastMath Uncertainty Quantification), Jan Glaubitz
- LBNL - Mark Adams (FastMath PETSc)
- SUNY Buffalo - Matt Knepley (PETSc), Joe Pusztay, Duncan Clayton

USQCD:

- ANL - James Osborne, Xiaoyong Jin
- BNL - Peter Boyle, Taku Izubuchi, Chulwoo Jung, Christopher Kelly, Shuhei Yamamoto, Patrick Oare
- FNAL - Andreas Kronfeld
- Boston University - Rich Brower, Nobu Matsumoto
- Columbia - Norman Christ, Yikai Huo
- Indiana - Steve Gottlieb, Leon Hostetler
- MSU - Alexei Bazavov
- UIUC - Aida El-Khadra, Michael Lynch
- Utah - Carleton Detar, David Clarke

Brookhaven National Laboratory

Future of RBC-UKQCD simulation program?

- How can we go beyond present lattice spacings and lattice sizes and exploit available GPU opportunities?
- FTHMC: 1/a = 3.5 GeV, 128^3x288 lattice volumes, high GPU efficiency
  - Accomplished because Grid provided access to GPU performance for algorithm development
  - Topological sampling is now adequate with FTHMC and longer trajectories on this lattice
- Large volumes challenge eigenmode deflation in valence analysis
  - Problem is solved by multiple right hand side multigrid
- Entire physics work flow (valence and HMC) runs fast at scale because of a portable, highly efficient GPU implementation
- Will enable 4 lattice spacing continuum limit for many quantities at physical quark mass with MDWF.

Field Transformation HMC (SciDAC-5)

- Unpublished quenched implementation and demonstration in Qlat, by Luchang Jin
  - Decorrelates DBW2 more efficiently
  - Follows Luscher's Wilson flow approach, but does NOT take continuous flow limit
    - Just take one big, cheap step

- arXiv:2212.11387 Nobu Matsumoto, Taku Izubuchi Lattice 2022

- arXiv:2401.16620 Peter Boyle (Grid implementation), Lattice 2023
  - Reimplemented FTHMC for plaquette flow
  - Runs with 2+1f dynamical DWF
  - Store intermediate gauge fields to avoid inverse flow (as with stout implementation)

- Shuhei Yamamoto, Lattice 2024 (proceedings soon!)
  - Effect on autocorrelation with large rho near bound
  - Code optimisation

- Christoph Lehner, 2024
  - GPT implementation
  - Broad algorithm optimisation study
  - Combine with long trajectories

- Much activity based on Luscher's Wilson flowed HMC (arXiv:1009.5877)

Two possible directions to address critical slowing down:

- Complex IR flow: learned generative/trivialising maps;
- Simple UV flow: retain momentum based local update with field transformation FT-HMC
  - Might be substantially easier to map QCD to QCD than trivialising to strong coupling limit

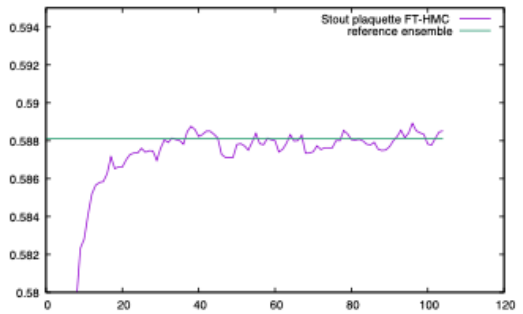
UV smearing function $U(V)$ brings tunable Fourier acceleration with *incomplete trivialisation*

$$\int dU e^{-S[U]} = \int dV \left| \frac{dU}{dV} \right| e^{-S[U(V)]}$$

- FTHMC :
  Gaussian momentum distribution
  Covariant smearing $\Rightarrow$ wavelength dependent transformation to physical gauge field
  Computable exact log det Jacobian
  - Quenched FT-HMC; general Wilson loops (Matsumoto, Jin, Izubuchi, Tomiya et al)

Grid implementation and demonstration, Lattice 2023 (PB)

- (masked) stout plaquette smeared FT-HMC in Grid: enables Fermion simulations (PB, Jin)
  - Reimplements Luchang Jin's Qlat FT-HMC; adds many options for fermions
- $L_s = 16$ Domain wall fermions + Iwasaki gauge action
- $16^3 \times 48$, $\beta = 2.13$, $m_{ud} = 0.01$, $m_s = 0.04$ 2+1 flavour
  - TWQCD's exact one flavor algorithm for strange (arXiv:1403.1683)
- $2 \times Nd$ subsets for plaquette stout smearing, $\rho = 0.1$
  - Developed under SciDAC-5 WP2 ; single node of 4xAMD GPUs (Lumi-G)
    - *Field transformation overhead significant but sub-dominant*
  - Reproduces reference plaquette in smeared links
  - Plan to investigate critical slowing down on 32^3 at 3 GeV
- Exascale consideration: the Jacobian force parallelises; Fermion solvers do not.
  - FT-HMC overhead is scalable.



Trajectory	Jacobian	Force smearing	Fermion action + force
1530s	295s	166s	900s

Improved autocorrelation times (Shuhei Yamamoto, Lattice 2024); 4x code speedup.

Look at the master-field / stochastic locality site-wise autocorrelation of Wilson flowed energy densities

Autocorrelation

$$\rho(\tau) = \frac{\sum_x (\mathcal{O}_n(x) - \bar{\mathcal{O}}_n)(\mathcal{O}_{n+\tau}(x) - \bar{\mathcal{O}}_{n+\tau})}{\sqrt{\sum_x (\mathcal{O}_n(x) - \bar{\mathcal{O}}_n)^2}\sqrt{\sum_y (\mathcal{O}_{n+\tau(y)} - \bar{\mathcal{O}}_{n+\tau})^2}}$$

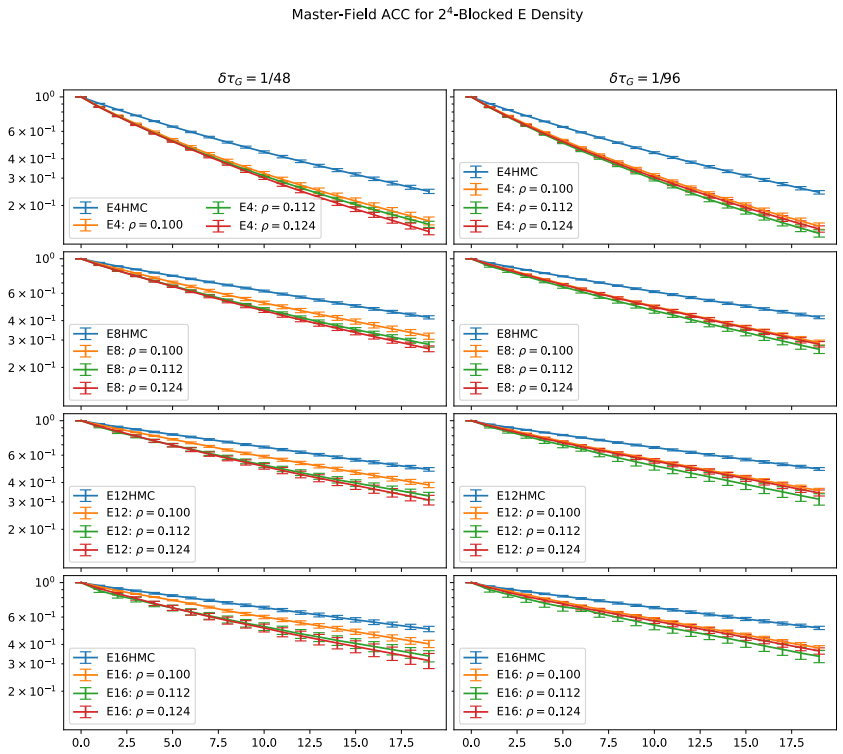


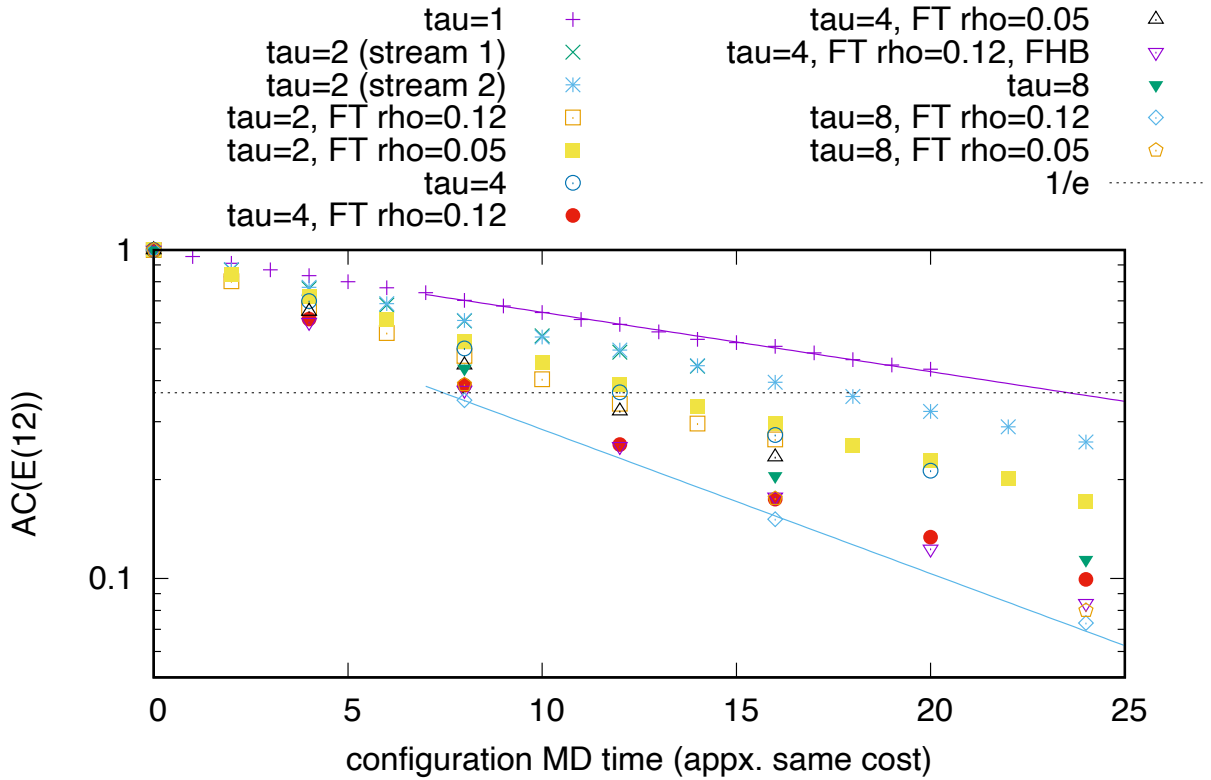
Figure: Plots of local ACC. Blue line is for HMC, and orange, greed, and red lines for FTHMC with $\rho = 0.1, 0.12, 0.124$, respectively

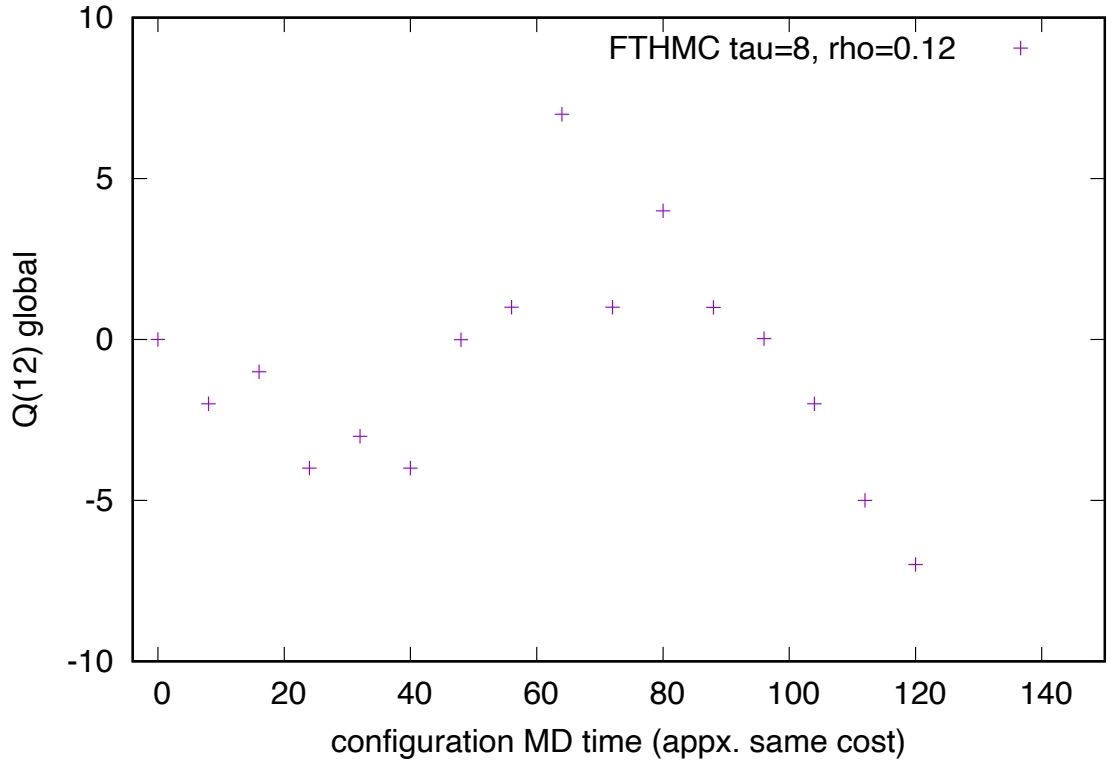
- Field transformation reduces autocorrelation
- The effect of smearing step size is not resolved in this figure

Christoph Lehner (2024):

GPT implementation

Algorithm survey at scale on 64^3 x 96 (left) and final 128^3 x 288 (right) physical point DWF 3.5 GeV 2+1f Lattice





3.5x speed up in decorrelation of subvolumes after second last doubling; consistent with observed thermalization on final volume

Topological sampling is restored

Combination of tau=8 trajectories AND field transformation is additive

Domain Wall Multigrid (SciDAC-5)

MDWF Multigrid: principally discuss: arXiv:2409.03904

- Multiple right hand side multigrid for domain wall fermions
  - Uses GPU matrix hardware efficiently
  - Introduces multigrid preconditioned block CG for increased algorithm efficiency
  - Over 20x speed up on test system at physical quark masses c.f. red-black CGNE

Multiple right hand side multigrid for domain wall fermions with a multigrid preconditioned block conjugate gradient algorithm.

Peter Boyle

[a] *Physics Department, Brookhaven National Laboratory, Upton, 11777, NY, USA*

Abstract

We introduce a class of efficient multiple right-hand side multigrid algorithm for domain wall fermions. The simultaneous solution for a modest number of right hand sides allows for a significant reduction in the time spent solving the coarse grid operator in a multigrid preconditioner. We use a preconditioned block conjuate gradient with a multigrid preconditioner, giving additional algorithmic benefit from the multiple right hand sides. There is also a very significant computational rate gain from multiple right hand sides. This both increases the arithmetic intensity in the coarse space and increases the amount of work being performed in each subroutine call, leading to excellent performance on modern GPU architectures. The software implementation makes use of vendor linear algebra routines (batched GEMM) that can make use of high throughput tensor hardware on recent Nvidia, AMD and Intel GPUs. The cost of the coarse space is made sub-dominant in this algorithm, and benchmarks from the Frontier supercomputer system show up to a factor of twenty speed up over the standard red-black preconditioned conjugate gradient algorithm on a large system with physical quark masses.

Keywords: Multigrid, Linear Solver, Lattice QCD, Algorithms, BlockCG

Edinburgh-2014/03

Hierarchically deflated conjugate gradient

P. A. Boyle[1]

(RBC and UKQCD Collaborations)

[1] *SUPA, School of Physics and Astronomy,*

The University of Edinburgh, Edinburgh EH9 3JZ, UK

(Dated: February 12, 2014)

PACS numbers: 11.15.Ha, 11.30.Rd, 12.15.Ff, 12.38.Gc 12.39.Fe

Abstract

We present a multi-level algorithm for the solution of five dimensional chiral fermion formulations, including domain wall and Mobius Fermions. The algorithm operates on the red-black preconditioned Hermitian operator, and directly accelerates conjugate gradients on the normal equations. The coarse grid representation of this matrix is next-to-next-to-next-to-nearest neighbour and multiple algorithmic advances are introduced, which help minimise the overhead of the coarse grid. The treatment of the coarse grids is purely four dimensional, and the bulk of the coarse grid operations are nearest neighbour. The intrinsic cost of most of the coarse grid operations is therefore comparable to those for the Wilson case. We also document the implementation of this algorithm in the BAGEL/Bfm software package and report on the measured performance gains the algorithm brings to simulations at the physical point on IBM BlueGene/Q hardware.

DWF: multigrid enfant terrible

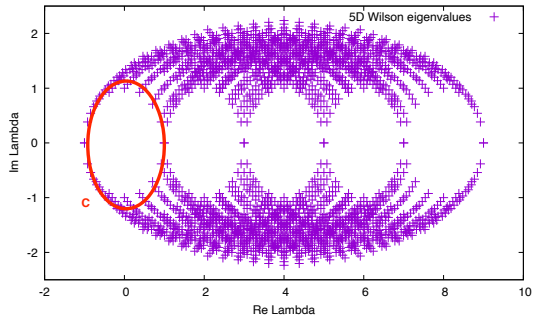
- Spectrum of DWF presents problems to non-Hermitian Krylov solvers
- Why? Krylov space is the span of polynomials of matrix M.
  Let $|i\rangle$ be the set of right eigenvectors, $\mathscr{P}(x) = c_n x^n$ a polynomial

$$\begin{aligned} M|i\rangle &= \lambda_i |i\rangle \\ \eta &= \eta_i |i\rangle \\ \psi^{\text{Krylov}} &= \mathscr{P}(M)\eta = (c_n \lambda_i^n)\eta_i |i\rangle \\ \psi^{\text{True}} &= \frac{1}{\lambda_i}\eta_i |i\rangle \end{aligned}$$

- There exists a contour C contained entirely within the (dense in large/infinite volume) spectrum such that

$$\oint_C \mathscr{P}^{\text{Krylov}}(z)dz = 0$$

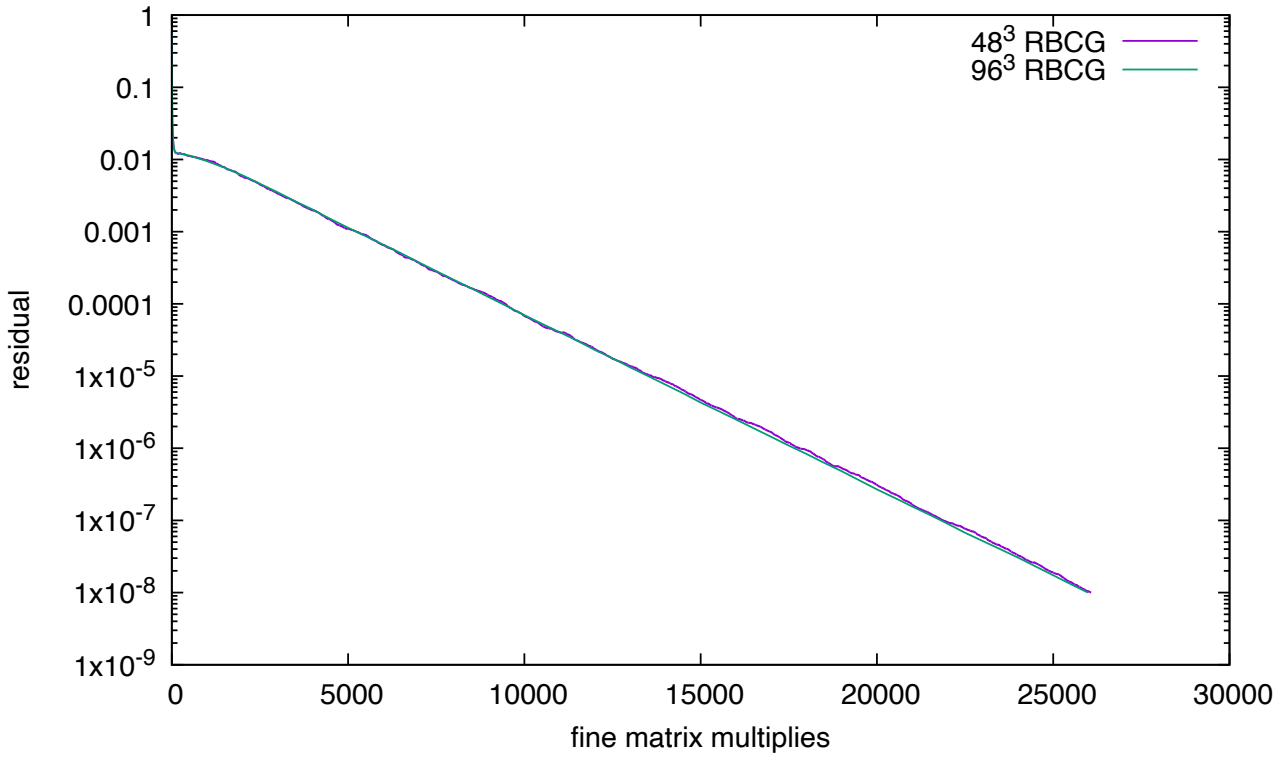
$$\oint_C F^{\text{True}}(z)dz = \oint_C \frac{1}{z}dz = 2\pi i$$



- Thus the Krylov polynomial and true solution must differ *within the domain of the spectrum*
- Must differ from solution *between* discrete eigenvalues. Low order polynomial is inadequate
- Manifests as slow convergence, perhaps of order system size

CG (and other Krylov solvers) are a function only of the spectrum:

Changing volume 16x didn't affect convergence.
But deflating on 16x bigger volume is a 256 times more expensive (!)



Measured convergence rate in tail: $\sigma_{cg} = .999451$

Chebyshev bound: $\sigma = \frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1} = 0.999445.$

$\lambda_{\max} = 89.757 \quad \lambda_{\min} = 6.92238 \times 10^{-6}$

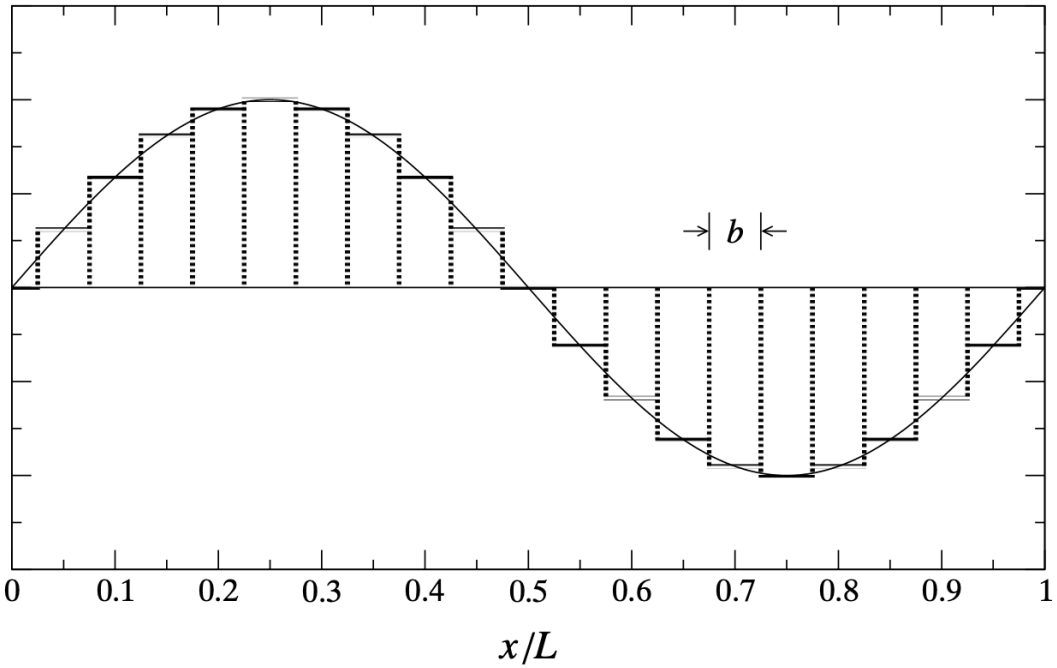
We are in large volume limit with dense spectrum
Orders of magnitude more EV's than order of polynomial
Saturates worst case minimax bound

Multigrid algorithms

Multigrid Dirac solvers:

- *Learn near null space of Dirac matrix* (~60 vectors): non-trivial in gauge theory
- Break vectors into local wavelet basis chunks
- Calculate a representation of Dirac operator within this critical subspace
- Use as an approximate near-null space multigrid preconditioner

- For DWF fine operator is $M_{pc}^{\dagger} M_{pc}$; where $M_{pc} = \left(M_{ee} - M_{eo} M_{oo}^{-1} M_{oe}\right)$
  - 81 point stencil !



Algorithm	Operator	Iterations	Full Matmuls	Time (s)
CGNR	$M^{\dagger}M$	9541	19082	183s
BiCGSTAB	$M_{PV}^{\dagger}M$	4140	8280	79s
prec-CGNR	$(M_{ee} - M_{eo}M_{oo}^{-1}M_{oe})^{\dagger}(M_{ee} - M_{eo}M_{oo}^{-1}M_{oe})$	3224	6448	62s
prec-CGNR	$(1 - M_{ee}^{-1}M_{eo}M_{oo}^{-1}M_{oe})^{\dagger}(1 - M_{ee}^{-1}M_{eo}M_{oo}^{-1}M_{oe})$	3880	7760	77s
GCR(32,32)	$M_{PV}^{\dagger}M$	8693	17386	474s

(Figure ``borrowed'' from Martin Lüscher's paper)

CERN-PH-TH/2007-096

Local coherence and deflation of the low quark modes in lattice QCD

Martin Lüscher

CERN, Physics Department, TH Division
CH-1211 Geneva 23, Switzerland

Multigrid preconditioners

Low mode subspace vectors ϕ generated in some way

- Inverse iteration, Inverse iteration with self refinement, Chebyshev filters

$$\phi_k^b(x) = \begin{cases} \phi_k(x) & ; \quad x \in b \\ 0 & ; \quad x \notin b \end{cases} \tag{1}$$

$$\text{span}\{\phi_k\} \subset \text{span}\{\phi_k^b\}. \tag{2}$$

$$P_S = \sum_{k,b} |\phi_k^b\rangle\langle\phi_k^b| \qquad ; \qquad P_{\bar{S}} = 1 - P_S \tag{3}$$

$$M = \begin{pmatrix} M_{\bar{S}\bar{S}} & M_{S\bar{S}} \\ M_{\bar{S}S} & M_{SS} \end{pmatrix} = \begin{pmatrix} P_{\bar{S}} M P_{\bar{S}} & P_S M P_{\bar{S}} \\ P_{\bar{S}} M P_S & P_S M P_S \end{pmatrix} \tag{4}$$

We can represent the matrix M exactly on this subspace by computing its matrix elements, known as the *little Dirac operator* (coarse grid matrix in multi-grid)

$$A_{jk}^{ab} = \langle\phi_j^a|M|\phi_k^b\rangle \qquad ; \qquad (M_{SS}) = A_{ij}^{ab}|\phi_i^a\rangle\langle\phi_j^b|. \tag{5}$$

the subspace inverse can be solved by Krylov methods and is:

$$Q = \begin{pmatrix} 0 & 0 \\ 0 & M_{SS}^{-1} \end{pmatrix} \qquad ; \qquad M_{SS}^{-1} = (A^{-1})_{ij}^{ab}|\phi_i^a\rangle\langle\phi_j^b| \tag{6}$$

It is important to note that A inherits a sparse structure from M because well separated blocks do *not* connect through M.

2 level ADEF-2 preconditioned CG algorithm

Flexible ADEF2 Preconditioned Conjugate Gradient

1. x arbitrary
2. $M^{-1} := (P_R M_{IRS}(\mathcal{H}, \Lambda) + Q)$
3. $x_0 = Qb + P_R x$
4. $r_0 = b - \mathcal{H} x_0$
5. $z_0 = M^{-1} r_0$; $p_0 = z_0$
6. for iteration $k \in 0, 1, 2 \ldots$
7. $\quad d_k = \mathcal{H} p_k$
8. $\quad w_k = d_k - \sum_{i=k-m_{max}}^{k-1} \frac{(d_k, \mathcal{H} w_i)}{(w_i, \mathcal{H} w_i)} w_i$
9. $\quad \alpha_k = (r_k, y_k)/(p_k, w_k)$
10. $\quad x_{k+1} = x_k + \alpha_k p_k$
11. $\quad r_{k+1} = r_k - \alpha_k w_k$
12. $\quad z_{k+1} = M^{-1} r_{k+1}$
13. $\quad \beta_k = (r_{k+1}, z_{k+1})/(r_k, z_k)$
14. $\quad p_{k+1} = z_{k+1} + \beta_k p_k$
15. end for
16. return x_{k+1}

Figure 1: Flexible ADEF2 preconditioned conjugate gradient algorithm following Tang et al[41] for solving a Hermitian system $\mathcal{H}x = b$, with an additonal flexible search direction orthogonalisation (step 8). We have implemented "inexact preconditioned CG"[43] and "flexible CG" [44] variants of the ADEF2 algorithm to address the variability in the preconditioner when it is composed of Krylov processes. The matrix Q is a coarse grid correction, and the matrix $M_{IRS}(\mathcal{H}, \Lambda)$ is a smoother in multigrid nomenclature. The Galerkin projector P_R is defined in the text.

Multigrid cycle is introduced as a CG preconditioner

Flexible algorithm: orthogonalise to previous search directions to tolerate preconditioner variability - if A is not constant, maintaining A-orthogonality is hard!

Coarse grid correction

Deflate this with O(200) coarse grid eigenvectors

Compute using block Lanczos

Smoother - **use CG but force it to look at high end of spectrum**

7 iterations of CG ~ $\frac{1}{\mathcal{H} + \lambda}$

First step: reimplement HDCG in Grid to run on modern GPUs

- Coarse grid was expensive and difficult to optimize on GPUs as not enough work/latency dominated
- Rapidly concluded I needed multiple right hand side implementation

Why design algorithms for multiple right hand sides?

Faster code

- Applying the coarse operator is dominated by cost of loading n^2_{basis}coefficients A_{ij} and applying to the n_{basis} coarse vector ϕ_j
- We can apply to several vectors at once, loading n^2_{basis} coefficient matrices and $n_{\text{basis}} \times n_{rhs}$
  - When $n_{rhs} \ll n_{basis}$ the additional right hand sides are free
  - Promoting to matrix matrix operations increases arithmetic intensity
  - Using multiple right hand sides increases the parallelism in the coarse operations: very GPU friendly

$$\begin{pmatrix} A & & & \\ & A & & \\ & & A & \\ & & & A \end{pmatrix} \begin{pmatrix} \psi_1 \\ \psi_2 \\ \psi_3 \\ \psi_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix}$$

Faster algorithms

- Block algorithms, such as BlockCG, can share the Krylov space between multiple solves
  - Known to accelerate staggered fermions; advocated as alternative to multigrid for staggered (Clark et al, 1710.09745)
    - Linear algebra scales as n^2_{rhs}
    - Matrix multiply scales as n_{rhs}

BlockCGrQ

1. $\mathcal{H} \in \mathbb{C}^{n \times n}$
2. B source $\in \mathbb{C}^{n \times n_{rhs}}$
3. X arbitrary $\in \mathbb{C}^{n \times n_{rhs}}$
4. $R_0 = B - \mathcal{H} X_0$
5. $Q_0 C_0 = R_0$
6. $D_0 = Q_0$
7. for iteration $k \in 0, 1, 2 \ldots$
8. $\quad Z_k = \mathcal{H} D_k$
9. $\quad M_k = [D_k^\dagger Z_k]^{-1}$
10. $\quad X_{k+1} = X_k + D_k M_k$
11. $\quad Q_{k+1} S_{k+1} = Q_k - Z_k M_k$
12. $\quad D_{k+1} = Q_{k+1} + D_k S_{k+1}^\dagger$
13. $\quad C_{k+1} = S_{k+1} C_k$

Preconditioned Block CG

- To combine multi-grid with block algorithms had to solve two problems
  - Introduce a preconditioned block CG
    - This did not exist in the literature, despite being relatively obvious
  - Address preconditioner variability: $n_{rhs}^2 \times m_{max}$ orthogonalisation is expensive
    - Replace CG based smoother and coarse solver with a fixed polynomial
- Perform CG in composite matrix $M^{-1}A$ and with M-inner product
  - $M^{-1}A$ is only Hermitian in the M-inner product $\langle x \,|\, y \rangle_M = \langle x \,|\, M \,|\, y \rangle_E$
  - The M-inner products are re-written as $\langle Z \,|\, Z \rangle_M = \langle Z \,|\, z \rangle_E$

Preconditioned BlockCGrQ

1. $z_0 = B - \mathcal{H} X_0$
2. $Z_0 = M^{-1} z_0$
3. $C_0^\dagger C_0 = Z_0^\dagger \cdot z_0 = \langle Z_0 | Z_0 \rangle_M$
4. $q_0 = z_0 C_0^{-1} \quad ; \quad Q_0 = Z_0 C_0^{-1}$
5. $D_0 = Q_0$
6. for iteration $k \in 0, 1, 2 \ldots$
7. $\quad z_k = \mathcal{H} D_k$
8. $\quad Z_k = M^{-1} z_k$
9. $\quad N_k = [D_k^\dagger \cdot z_k]^{-1} = \langle D_k | Z_k \rangle_M^{-1}$
10. $\quad X_{k+1} = X_k + D_k N_k$
11. $\quad t_k = q_k - z_k N_k \quad ; \quad T_k = Q_k - Z_k N_k$
12. $\quad S_k^\dagger S_{k+1} = T_k^\dagger \cdot t_k = \langle T | T \rangle_M$
13. $\quad q_{k+1} = t_k S_k^{-1} \quad ; \quad Q_{k+1} = T_k S_k^{-1}$
14. $\quad D_{k+1} = Q_{k+1} + D_k S_k^\dagger$
15. $\quad C_{k+1} = S_k C_k$

Stationary coarse grid deflated Chebyshev inverter

Break source into pieces parallel and perpendicular to low mode space
Solve with deflation (parallel) and polynomial approximation (perp)

$$
\begin{aligned}
b &= b^{\|} + b^{\perp}, \\
b^{\|} &= \left(\sum_{i=1}^{n_{ev}} |i\rangle\langle i| \right) b, \\
b^{\perp} &= \left(1 - \sum_{i=1}^{n_{ev}} |i\rangle\langle i| \right) b, \\
Qb^{\|} &= \left(\sum_{i=1}^{n_{ev}} \frac{|i\rangle\langle i|}{\lambda_i} \right) b.
\end{aligned}
$$

$$
\begin{aligned}
Q^{Cheby+defl} &= \mathrm{diag}\left(\frac{1}{\lambda_{min}}, \ldots, \frac{1}{\lambda_{n_{ev}}}, \mathbb{P}(^{\mathrm{Cheby}}\lambda_{n_{ev}+1}), \ldots, \mathbb{P}^{\mathrm{Cheby}}(\lambda_{max})\right) \\
Q^{Cheby+defl} b &= \mathrm{Defl}(b^{\|}) + \mathbb{P}^{\mathrm{Cheby}}(b^{\perp}).
\end{aligned}
$$

Stationary Fine grid smoother : fixed order Chebyshev inverse over spectral range

Defining the coarse operator: Low mode subspace setup and fidelity

Several multigrid subspace setup options:

- Recursive Chebyshev low pass filters (HDCG, HDCR)

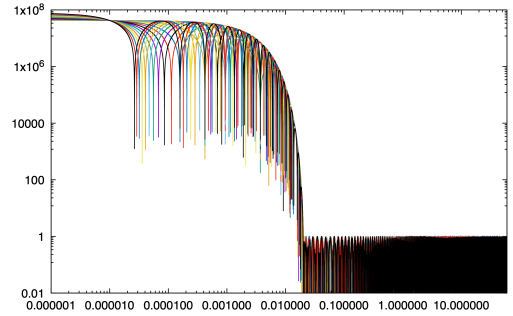


FIG. 2. Overlay of Chebyshev low-pass filter functions used to create the subspace. An initial O(500) low pass filter enhances the low mode content from Gaussian noise by 10^8. This is refiltered with further, recursive Chebyshev functions of the fine operator, with a total of 1500 further matrix multiplies for 16 vectors. Plotted is the absolute value of the filter response for each of the 16 filtered vectors versus eigenvalue. The filter parameters have $\lambda_{max} = 60$, $m = 500$, $\Delta = 100$ and $\lambda_0 = 0.02$, and are taken from the optiminum from a $16^3 \times 32$ domain wall fermion test case. The downward drops (obviously) correspond to the roots of the polynomials and bands within the target spectral range have relative signs inserted for different order polynomials, creating independent vectors with an efficient recursion.

- Rational function lowpass via single multishift CG

$$f(\lambda) = \frac{1}{(\lambda + \Lambda_1)(\lambda + \Lambda_2)(\lambda + \Lambda_3)(\lambda + \Lambda_4)}$$

- Demonstrated use of Lanczos Eigenvectors is superior

Measure completeness of Eigenvectors in coarse basis

- BFM Filter scheme: 64x (rational 4th order low pass + shifted CG refine)
- Grid Filter scheme: Chebyshev low pass + shifted CG refine
- Grid: Lanczos eigenvectors (4x more costly - or more)

- re-expand 200 lowest modes in the Filter scheme coarse grid
  - projectCoarse, promoteFine, take difference
  - projectCoarse, promoteFine, 7step smoother, take difference

$$E_i = \sqrt{||(1 - PP^\dagger)|i\rangle||}.$$

$$E_i^{smoothed} = \sqrt{||M_{IRS}(1 - PP^\dagger)|i\rangle||}.$$

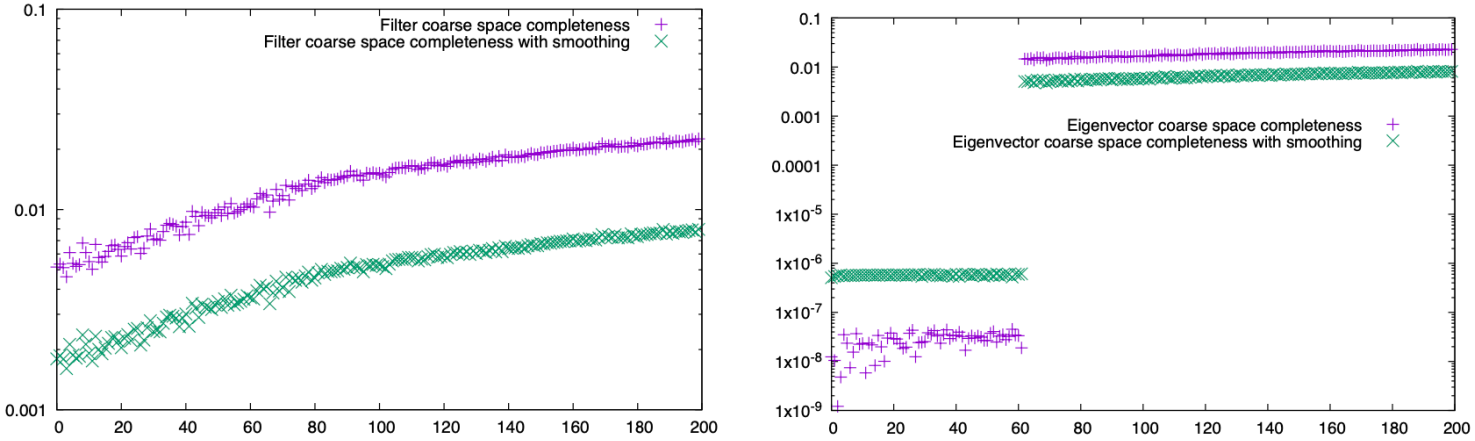


Figure 4: On our $48^3 \times 96$ test volume we can assess the completeness of the low mode by computing $E_i = \sqrt{||(1 - PP^\dagger)|i\rangle||}$ and $E_i^{smoothed} = \sqrt{||M_{IRS}(1 - PP^\dagger)|i\rangle||}$ for each eigenpair, here with the Filter based subspace setup (left) and Lanczos based setup (right). Post-smoothing reduces the error and may be indicative of a relative cheap way to improve coarse operator based low mode variance reduction. The exact eigenvectors included in the coarse basis (right) are faithfully reproduced to numerical precision, while higher modes have a percent scale error.

Compare coarse eigenspectrum to fine eigenspectrum

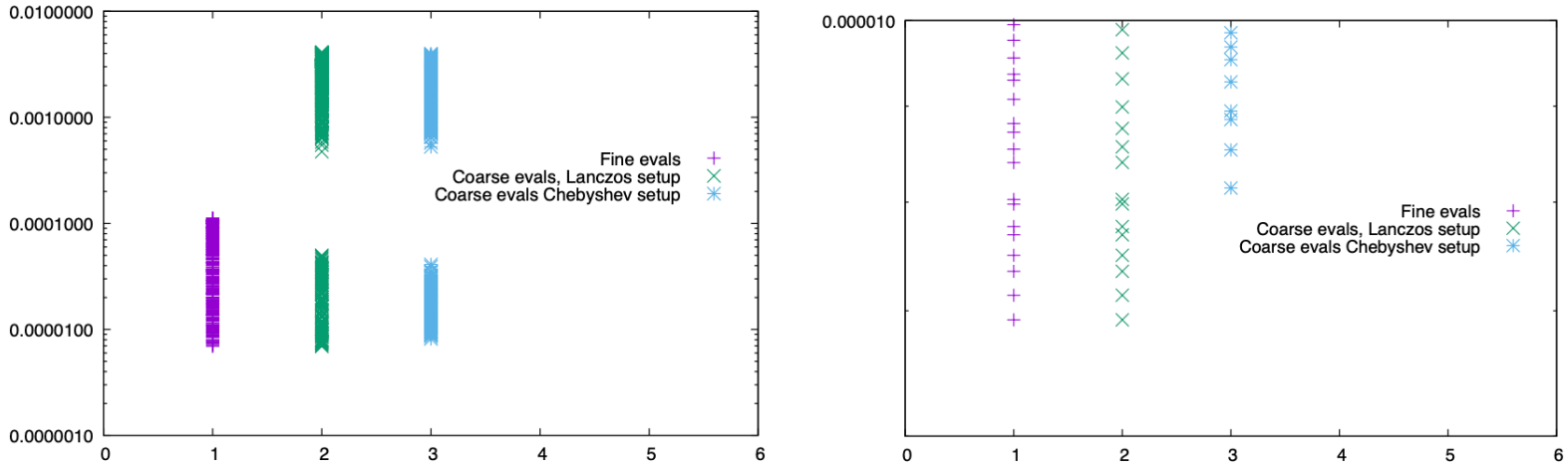


Figure 5: Spectrum of the lowest 200 eigenmodes of the fine operator $\mathcal{H}$ and the coarse operator with both Lanczos (green) and Chebyshev filter (blue) setup schemes with 62 near null basis vectors on the 48^3 lattice. A cluster of exactly $n_{basis} = 62$ very low eigenvalues are seen in the coarse operator, corresponding to the maximal diagonalisation of the fine operator within this set of near null vectors, whereas directions that involve a non-trivial coarse coordinate dependence in the coarse eigenvector necessarily incur spectral leakage at the boundaries between blocks and this lifts the coarse eigenvalue by an order of magnitude. The upper

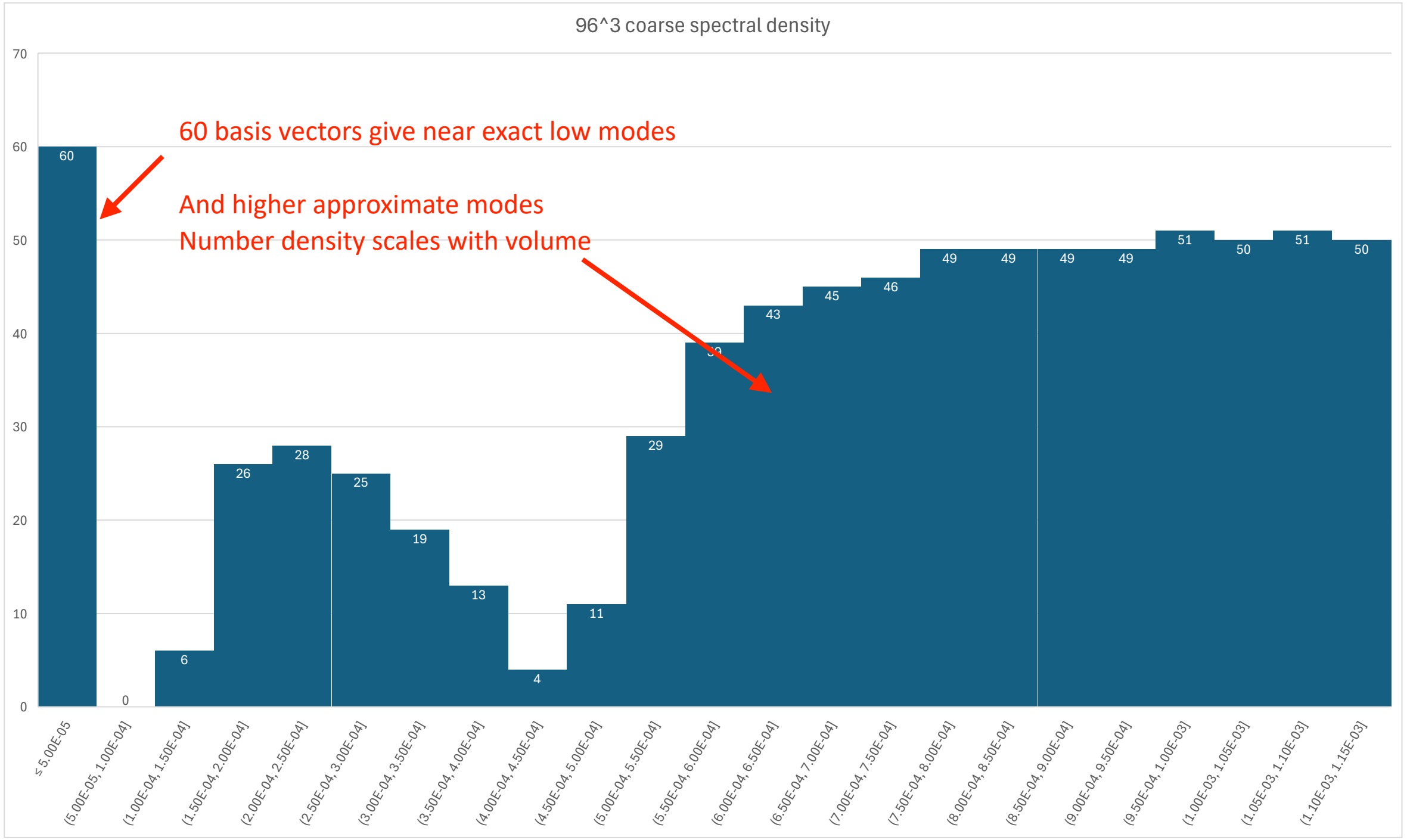
There are exactly n_{basis} *very* low modes. Could probably avoid coarse Lanczos

Many more approximate low modes with eigenvalues 10^{-4} or higher

- Diagonalising within the original vectors preserves near null property
- Moving out this pace raises eigenvalue due to sharp edges!

Very lowest modes *exactly* preserved in Lanczos subspace coarsening (60 vecs)

- Fairly rapidly loses accuracy, and gap in spectrum present in both


96^3 coarse spectral density
60 basis vectors give near exact low modes
And higher approximate modes
Number density scales with volume
70
60
50
40
30
20
10
0
60
0
6
26
28
25
19
13
4
11
29
43
45
46
49
49
49
49
51
50
51
50
≤ 5.00E-05
(5.00E-05, 1.00E-04]
(1.00E-04, 1.50E-04]
(1.50E-04, 2.00E-04]
(2.00E-04, 2.50E-04]
(2.50E-04, 3.00E-04]
(3.00E-04, 3.50E-04]
(3.50E-04, 4.00E-04]
(4.00E-04, 4.50E-04]
(4.50E-04, 5.00E-04]
(5.00E-04, 5.50E-04]
(5.50E-04, 6.00E-04]
(6.00E-04, 6.50E-04]
(6.50E-04, 7.00E-04]
(7.00E-04, 7.50E-04]
(7.50E-04, 8.00E-04]
(8.00E-04, 8.50E-04]
(8.50E-04, 9.00E-04]
(9.00E-04, 9.50E-04]
(9.50E-04, 1.00E-03]
(1.00E-03, 1.05E-03]
(1.05E-03, 1.10E-03]
(1.10E-03, 1.15E-03]

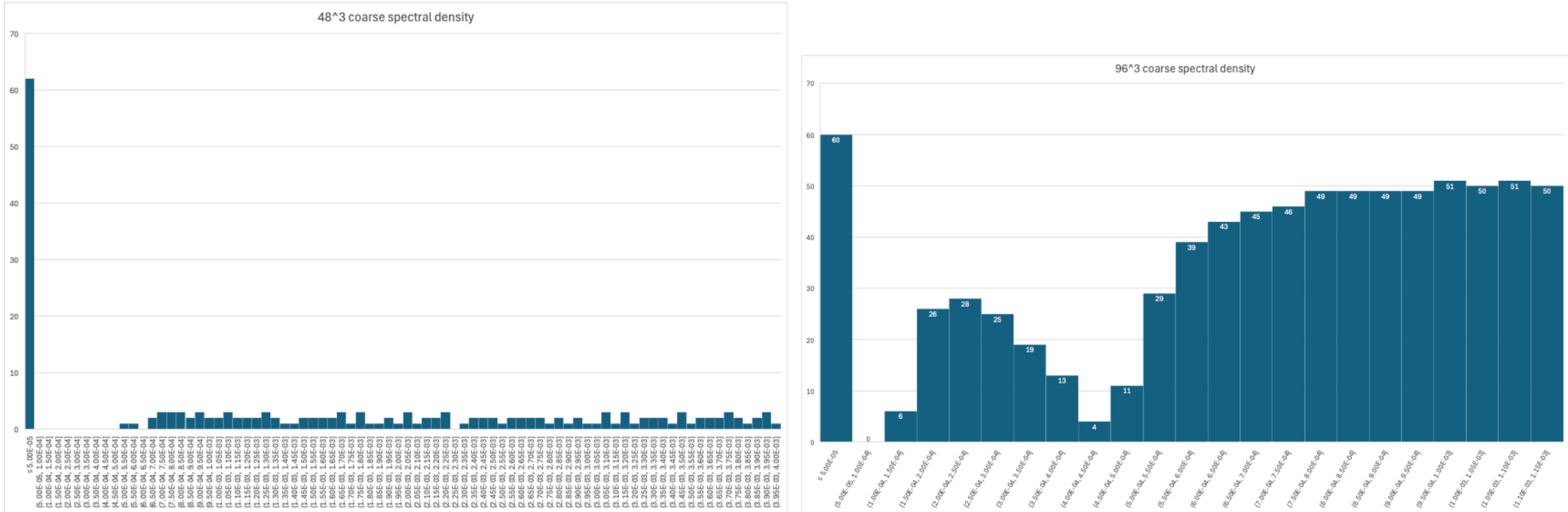


Figure 6: Spectral density (modes per bin, bin width 5×10^5) for the coarse operator on the 48^3 volume (left) and 96^3 volume (right). There is a peak of $n_{basis} = 62$ modes in the correct physical low mode region, corresponding to the dimension of the input set of near null vectors, with the eigenvectors in practice found by diagonalising mainly within this basis. Directions outside this sub-space necessarily induce upwards spectral leakage leaving this cluster clearly detached from the bulk spectrum with about 2 or 3 modes per bin. If we compare these, we see that the detached cluster again corresponds to the number of basis vectors defining the coarsening, but the density of modes in the higher, bulk spectrum grows linearly in the volume by exactly the expected factor of 16.

Multigrid solver performance

All tests run on Frontier@ORNL, 18 or 288 nodes
AMD GPU supercomputer, similar to Lumi-G
Mobius DWF with Ls=24 and physical quark mass and 1.78GeV lattice spacing

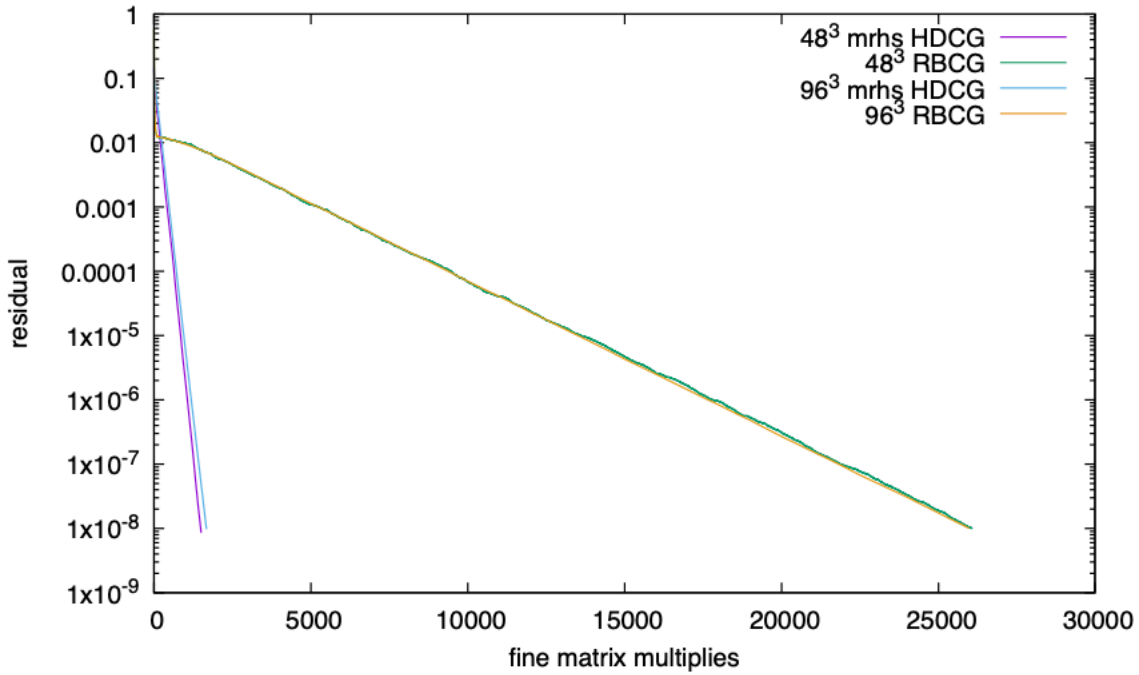
	$48^3 \times 96$	$48^3 \times 96$	$96^3 \times 192$
Algorithm	Flex-ADEF2	PrecBlockCGrQ	Flex-ADEF2
n_{basis}	62	62,64	60
n_{rhs}	12	12	12
block	4.4.6.4	4.4.6.4	4^4
Smoother	$M_{IRS}(2.0, 7)$	$\mathbb{P}^{cheby}(2.0, 92.0, 8)$	$M_{IRS}(2.0, 7)$
Coarse solve	Defl CG tol 0.04	$\mathbb{P}^{cheby+defl}(0.02, 40, 120)$	Defl CG tol 0.04
Coarse EVs	192	192	1000
Setup	Lanczos/Cheby	Lanczos	Cheby
Speedup	12x - 15x	20x	10.5x

Factor of > twenty speed up

- N-lowest eigenvectors was the best setup
- BlockCGrQ for 12 RHS was best (20% gain)
- Flexible ADEF2 with multiple RHS was still good
- Filter setup was competitive and substantially cheaper

Setup	n_{basis}	Algorithm	Outer Iters	Fine matrix multiplies	12 solves (s)	Speed up
$48^3 \times 96 \times 24$						
EV	64	PrecBlockCGrQ	108	1080	460	20.2
EV	62	PrecBlockCGrQ	113	1130	480	19.3
EV	62	FlexADEF2*	131	1300	570	16.3
EV	62	FlexADEF2	141	1269	602	15.4
Filter	62	PrecBlockCGrQ	123	1230	499	18.6
Filter	62	FlexADEF2	167	1503	720	12.9
EV (sum-2)	62	FlexADEF2	213	1917	934	9.9
EV $(i\|2 = 0)$	62	FlexADEF2	255	2295	1090	8.5
EV $(i\|3 = 0)$	62	FlexADEF2	268	2412	1182	7.8
-	-	RedBlackCG	26075	26075	9288	1

Setup	n_{basis}	Algorithm	Outer Iters	Fine matrix multiplies	12 solves (s)	Speed up
$96^3 \times 192 \times 24$						
Filter	60	FlexADEF2	187	1683	1060	10.5
-	-	RedBlackCG	25984	25984	12720	1



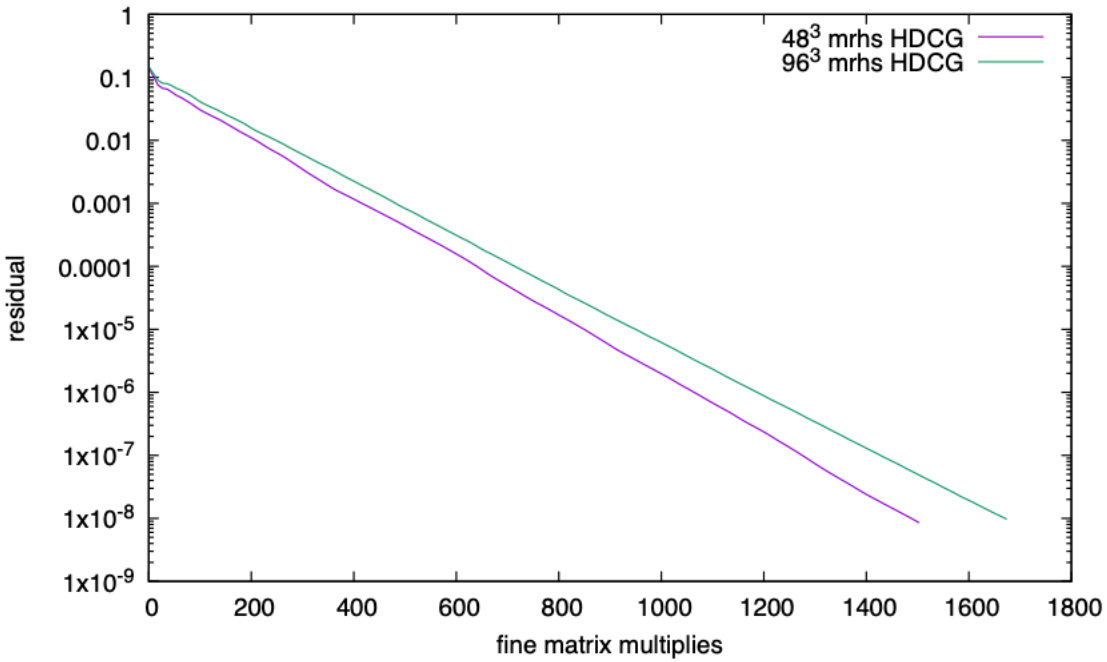


Figure 7: (Left) Convergence of mrhs-HDCG and red-black preconditioned Conjugate Gradient on sample $48^3 \times 96$ and $96^3 \times 192$ configurations. (Right) Zoomed comparison between the two volumes on the HDCG convergence history.

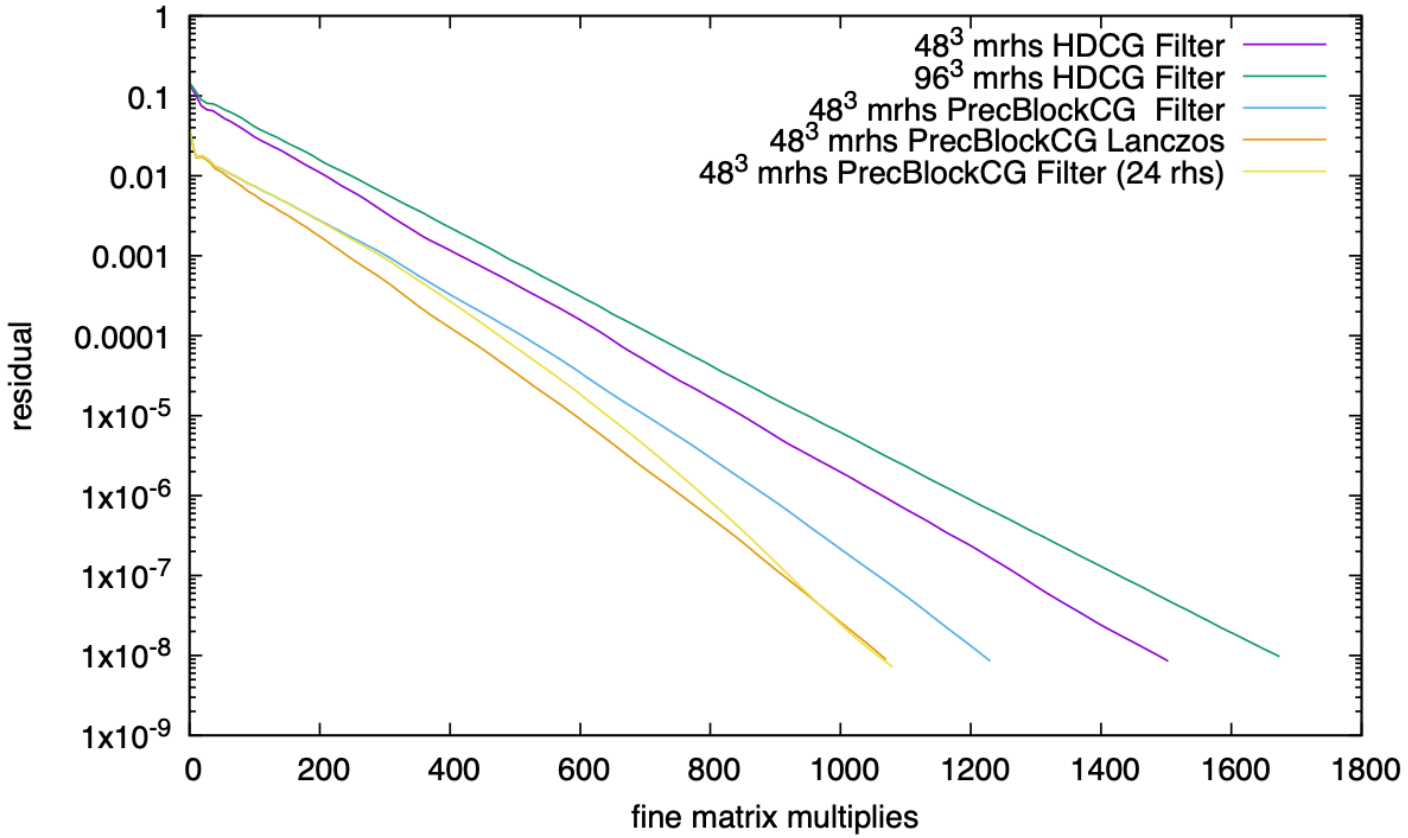


Figure 8: Convergence of Flexible ADEF2 and preconditioned BlockCGrQ on the 48^3 configurations with 12 right hand sides and 62 basis vectors (either eigenvectors or filtered noise vectors). The Block algorithm substantially reduces the difference between the eigenvector and filtered vector basis creation choices. The Block algorithm appears to better tolerate an imperfect setup than preconditioned CG alone. With 24 right hand sides and the Filter setup there is clearer evidence of the superlinear convergence property of BlockCG, however with the current software implementation the linear algebra overhead is too large to make this beneficial.

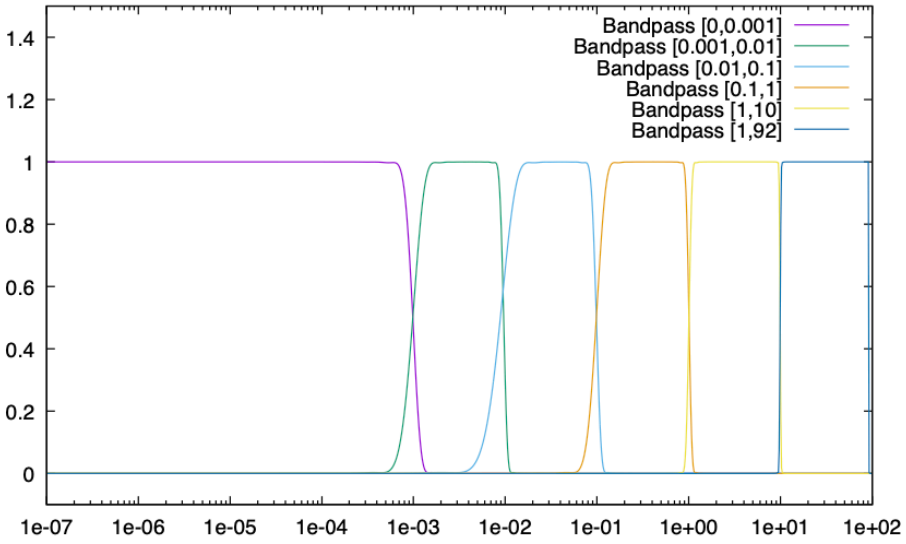
Power spectrum analysis (with Chebyshev filters)

Solution error is low mode dominated

Residual is high mode dominated

Smoother and coarse grid correction act in the spectral bans you expect

Multigrid V-cycle is active in low modes and below smoother edge



Eigenvalue range	Solution Error	Residual	r	$M_{IRS}r$	Qr	$z = M^{-1}r$
$[0, 0.001]$	9.98E-01	3.95E-05	1.54E-06	3.71E-05	8.19E-01	5.08E-01
$[0.001, 0.01]$	1.32E-03	1.59E-04	1.25E-05	3.01E-04	1.04E-01	6.81E-02
$[0.01, 0.1]$	2.03E-04	2.55E-03	3.62E-04	8.57E-03	3.98E-02	3.84E-02
$[0.1, 1]$	2.24E-05	2.14E-02	1.27E-02	2.50E-01	1.11E-02	1.15E-01
$[1, 10]$	1.03E-06	5.35E-02	1.11E-01	6.19E-01	2.44E-03	2.13E-01
$[10, 100]$	5.98E-08	9.20E-01	8.71E-01	9.36E-02	1.74E-03	3.06E-02
Total	0.99943	0.99761	0.99545	0.97072	0.97797	0.97413

Table 5: To illustrate the spectral behaviour of the multigrid preconditioner, we take the power spectrum of six different vectors with respect to the red-black preconditioned Dirac operator. These vectors are (left to right) the error on the solution vector when the solver is only converged to 10^{-4} accuracy and the residual at this 10^{-4} iteration. The error in the solution vector is predomninantly in the lowest bin, while the residual is dominated by the highest few bins. The final four vectors are the final residual, the smoother applied to this final residual (reduces power in the highest mode bin), the coarse grid inverse applied to the residual (dominated by lowest spectral bin) and the composite multigrid preconditioner applied to this residual, with quite an interesting spectral shape.

Multigrid software performance

MultiRHS coarse operator implemented using Batched BLAS calls

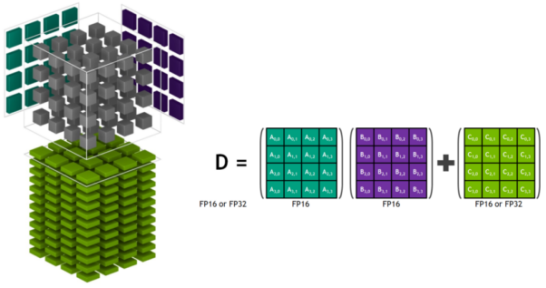
Coarse operations implemented using “Batched GEMM” APIs in HIP, CUDA and OneAPI (and Eigen for CPU)

Using machine learning tensor hardware in GPUs to speed up multi-right-hand side solvers in QCD

Grid / Grid / algorithms / **blas** /

paboyle Batched blas, but not working yet on OneAPI

Name
..
BatchedBlas.cc
BatchedBlas.h



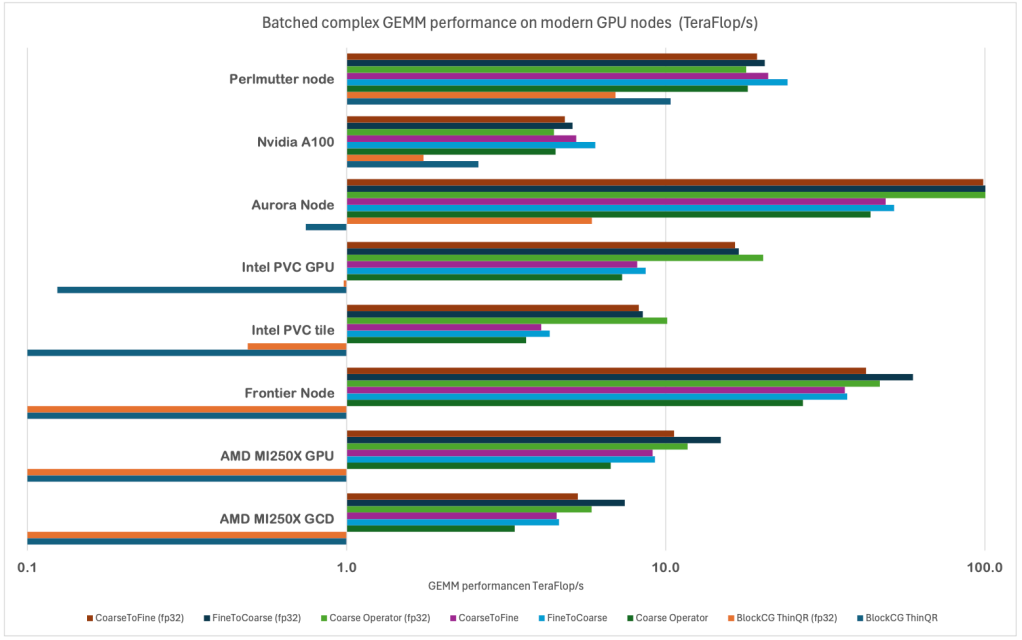


Figure 11: Performance of batched GEMM operations in TF/s as used in mrhs-HDCG across a variety of modern GPUs and some of the most significant current supercomputing platforms. The Matrix dimensions correspond to the application of the coarse grid operator, the projection of data from the fine grid to the coarse grid, the promotion from the coarse grid to the fine grid and the QR rotation that enters the Block conjugate gradient algorithms. These include the Frontier supercomputer at ORNL (four AMD MI250X GPUs and eight logical GCD’s), the Sunspot/Aurora supercomputer at ANL (six Intel Pontevecchio GPUs and 12 logical tiles) and the Perlmutter supercomputer at NERSC. Multiple TF/s per logical GPU is easily obtained on most of the relevant matrix ranks, with the exception of the ThinQR factorisation in BlockCG on the fine grid on AMD and Intel libraries. This is relatively easily addressed in our implementation by using a batched call to execute many shorter K matrix multiplications and then summing manually the resulting 12×12 matrices, and then yields several TF/s per GPU, but the vendor library delivers less than 10 GF/s performance without this approach.

Project (fine to coarse)

$$\forall_{x^c} \quad : \quad C_{N_{basis} \times N_{rhs}}(x^c) = B^{\dagger}_{N_{basis} \times V_{block}}(x^c) \times F_{V_{block} \times N_{rhs}(x^c)}$$

Promote (coarse to fine)

$$\forall_{x^c} \quad : \quad F_{V_{block} \times N_{rhs}(x^c)} = B_{V_{block} \times N_{basis}}(x^c) C_{N_{basis} \times N_{rhs}}(x^c)$$

Coarse operator

$$\forall_{x^c} \quad : \quad C_{N_{basis} \times N_{rhs}}(x^c) = C_{N_{basis} \times N_{rhs}}(x^c) + A^p_{N_{basis} \times N_{basis}}(x^c) \times B_{N_{basis} \times N_{rhs}}(x^c + \delta_p).$$

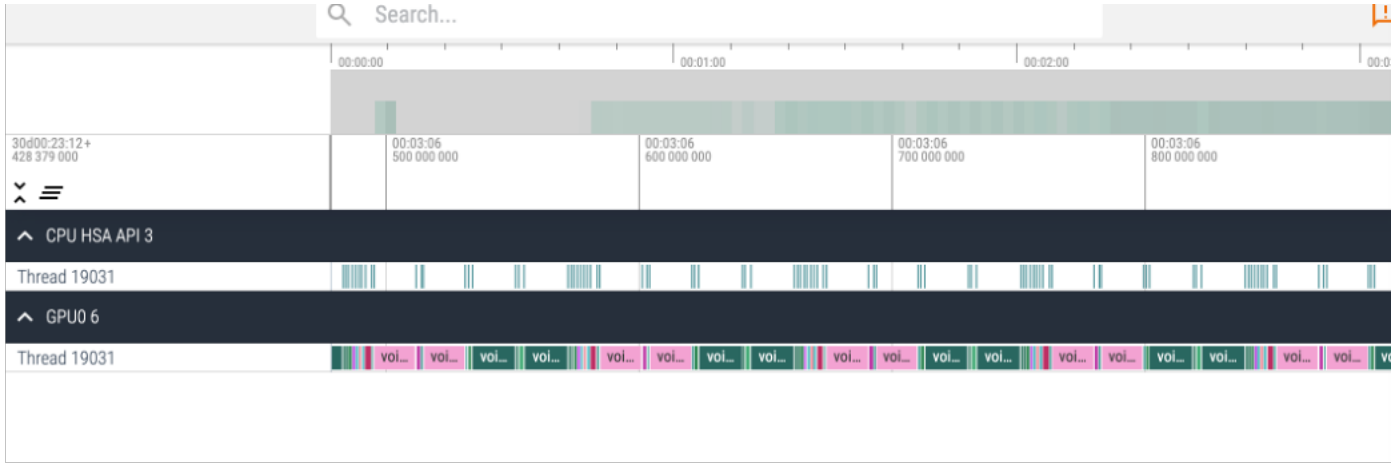
					Frontier AMD MI250X			Aurora Intel PVC			Perlmutter Nvidia A100	
	M	N	K	Batch	GCD	GPU	Node	Tile	GPU	Node	GPU	Node
FP64												
CoarseOp	64	12	64	4096	3.4	6.7	26.9	3.7	7.3	43.8	4.5	18.1
Project	64	12	256	4096	4.6	9.3	37.1	4.3	8.7	52.0	6.0	24.1
Promote	12	256	64	4096	4.6	9.1	36.4	4.1	8.1	48.9	5.2	21.0
ThinQR	12	12	1M	1	0.0	0.0	0.1	0.1	0.1	0.7	2.6	10.4
FP32												
CoarseOp	64	12	64	4096	5.9	11.7	46.9	10.1	20.2	121.2	4.5	17.9
Project	64	12	256	4096	7.4	14.9	59.5	8.5	17.0	101.7	5.1	20.4
Promote	12	256	64	4096	5.3	10.6	42.5	8.2	16.5	98.9	4.8	19.3
ThinQR	12	12	1M	1	0.0	0.0	0.0	0.5	1.0	5.9	1.7	7.0

Conjugate Gradient is:

1. Matrix
2. norm
3. linalg
4. Goto 1.

mrhs-Multigrid has

1. 12xCG smoothers
2. BatchedGemm blockProject
3. BatchedGemm deflate
4. MultiRHS coarse CG solve
5. BatchGemm blockPromote
6. Linalg
7. Goto 1.



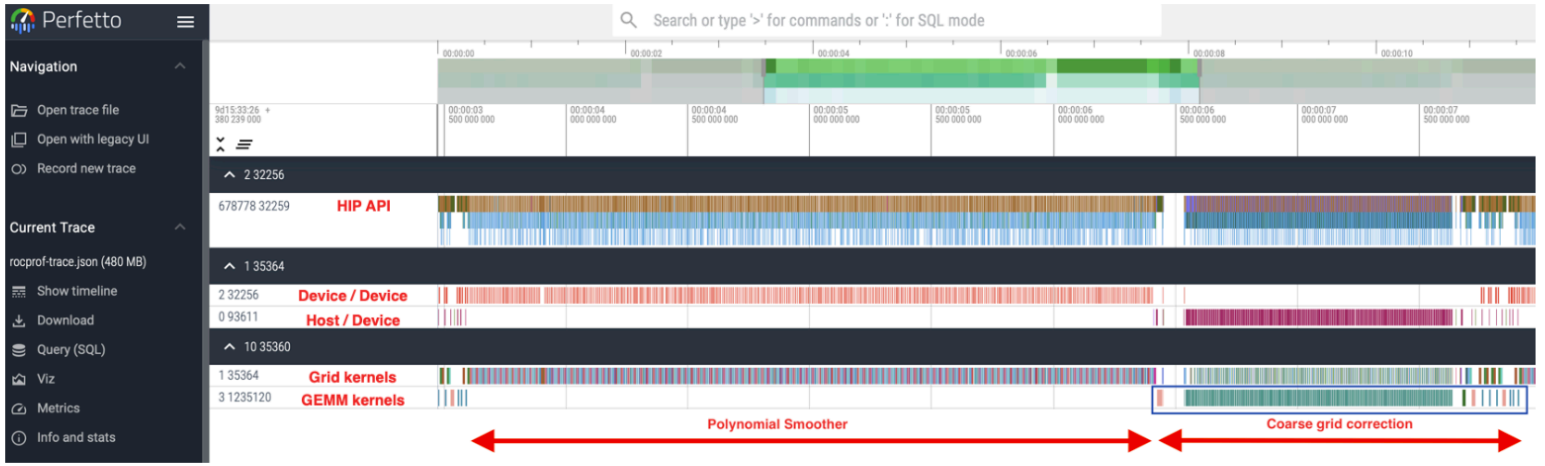


Figure 12: AMD Rocprof obtained profile from Frontier of the multigrid iteration on 18 nodes on the 48^3 test problem after careful optimisation. The general Grid software kernels are shown executing along side GEMM kernels, used by projection to coarse, deflation, coarse Chebyshev solver, promotion to fine and then a thinQR rotation. Broadly it is possible to perform almost the entire multigrid preconditioner in BLAS routines using optimised hardware, except for the relatively modest overhead of data layout changes and of course halo-exchange routines.

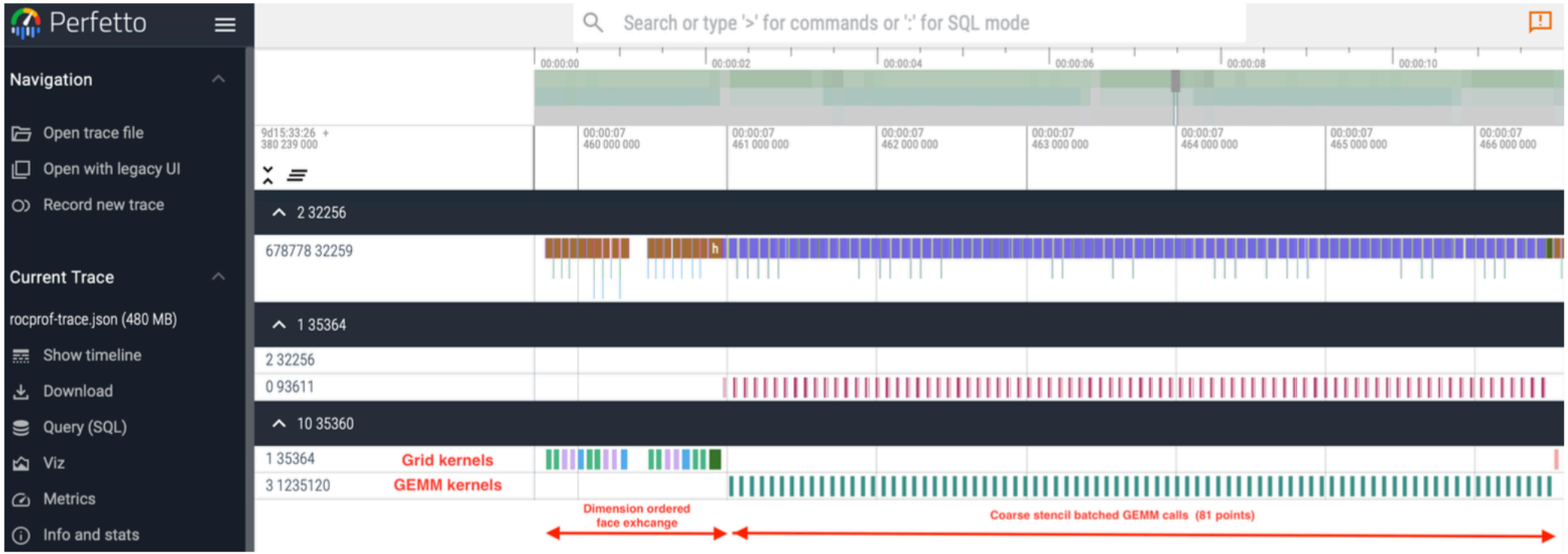
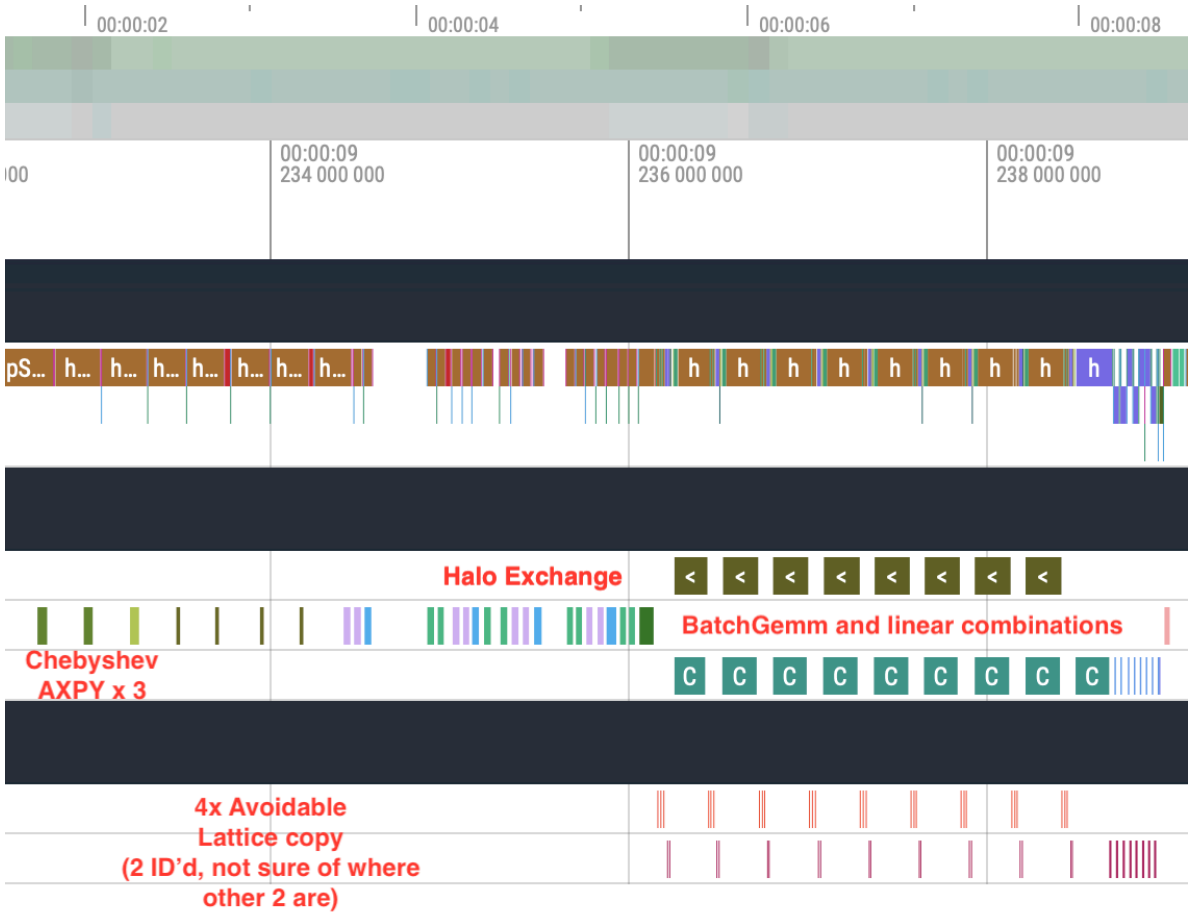


Figure 13: AMD Rocprof obtained profile from Frontier of the Coarse Grid operator on 18 nodes on the 48^3 test problem after careful optimisation. The general Grid software kernels are shown executing along side GEMM kernels. There remains some scope for further optimisation in the coarse space operator as GPU synchronisation overhead remains a 50% overhead in that routine by fusing together larger batched operations, perhaps an estimated 10% effect on the overall solver performance.

As was anticipated in paper, further improved the coarse operator with bigger GEMM batches since

- 81 point stencil as 9 batches of (coarse-vol x 9 stencil terms)
  - Sum 9 temporaries at end
  - Significant additional speed up (4.4ms -> 2.5ms)



Operation	Time for component
Linear Algebra	16s
Fine residual	30s
Multigrid preconditioner	368s
Total	417s
Restriction	3.5s
Prolongation	2.9s
Coarse deflation	6s
Coarse solve	100s
Smoother	223s

Table 4: In the fastest Preconditioned BlockCGrQ run from the 48^3 ensemble we give the breakdown of the total time into the component operations. The operations that are accelerated using the (batched) GEMM interfaces are colored red, and consume around 30% of the run time, illustrating how machine learning and artificial intelligence focussed matrix and tensor hardware can be used directly in multi-rhs multigrid solvers. Of this overhead, the coarse grid deflation is around 1.5% but is the only element of the algorithm that scales as $O(\text{volume}^2)$

Only the coarse deflation is volume^2

Will declare problem successfully deferred for my lifetime (!)

Further gains in the coarse operator remain possible, but are 10% level

Software and Exascale machines

(Finally! The novel computing doo-dah bit)

- All major US supercomputers are GPU accelerated
  - There are multiple, software incompatible platforms
- Majority of recent EU supercomputers are GPU accelerated

- Code must be highly parallel, regular (lots work items executing same software paths)

- Data must be structured and laid out to create locality of access between work items
  - Used to be called vectorization, but the software interfaces complicate this a little.

- Huge factors of computer performance will be lost if algorithms/software do not face these facts.

- Not all algorithms will naturally suit GPUs. Life is tough.

If the internal abstraction is defined right, porting is easy !

- Performed the HIP and SYCL ports on the SAME DAY

Portability 101: Use an internal abstraction

- Grid contains a data parallel API, expression template engine, general QFT data types

- D-dimensional Lattice<T> containers have opaque layout and partial vector layout transformation

- **Internal** offload API in ``Accelerator.h''
  - accelerator_for() macro captures a loop body in a C++11 Lambda function and executes it.
  - acceleratorCopyToDevice etc….
  - deviceVector<T>

- Software managed cache of Lattice objects is maintained on device.
  - Lookup in cache is O(1) overhead (hash table - cost does NOT grow with volume of data)
  - True LRU Q (least recently used) eviction algorithm when a high-watermark is exceeded
  - Transient / discard next facility
  - Entries are “CpuDirty”, “AccDirty”, “Consistent” ; “Views” can be open for CPU or GPU but not both
    - Opening views updates state and triggers data motion

- API covers OpenMP, HIP, SYCL and CUDA: vector intrinsics for most CPUs and most major GPUs.

- Most users do not see this API because the Lattice ET engine sits above it.
  - API has also been used directly by C. Lehner (GPT), C. Kelly (CPS), Luchang Jin (Qlat)
  - It’s open source and can be mined for ideas

```
#define accelerator_for2dNB( iter1, num1, iter2, num2, nsimd, ... )     \
  {                                                                     \
    int nt=acceleratorThreads();                                        \
    typedef uint64_t Iterator;                                          \
    auto lambda = [=] accelerator                                       \
      (Iterator iter1,Iterator iter2,Iterator lane) mutable {           \
      __VA_ARGS__;                                                      \
    };                                                                  \
    dim3 cu_threads(nsimd,acceleratorThreads(),1);                      \
    dim3 cu_blocks ((num1+nt-1)/nt,num2,1);                             \
    LambdaApply<<<cu_blocks,cu_threads,0,computeStream>>>(num1,num2,nsimd,lambda);      \
  }

template<typename lambda>  __global__
void LambdaApply(uint64_t num1, uint64_t num2, uint64_t num3, lambda Lambda)
{
  // Weird permute is to make lane coalesce for large blocks
  uint64_t x = threadIdx.y + blockDim.y*blockIdx.x;
  uint64_t y = threadIdx.z + blockDim.z*blockIdx.y;
  uint64_t z = threadIdx.x;
  if ( (x < num1) && (y<num2) && (z<num3) ) {
    Lambda(x,y,z);
  }
}
```

CUDA/Nvidia

```
accelerator_inline int acceleratorSIMTlane(int Nsimd) {
#ifdef GRID_SIMT
 return __spirv::initLocalInvocationId<3, cl::sycl::id<3>>()[2];
#else
 return 0;
#endif
} // SYCL specific

#define accelerator_for2dNB( iter1, num1, iter2, num2, nsimd, ... )     \
  theGridAccelerator->submit([&](cl::sycl::handler &cgh) {              \
    unsigned long nt=acceleratorThreads();                              \
    if(nt < 8)nt=8;                                                     \
    unsigned long unum1 = num1;                                         \
    unsigned long unum2 = num2;                                         \
    unsigned long unum1_divisible_by_nt = ((unum1 + nt - 1) / nt) * nt; \
    cl::sycl::range<3> local {nt,1,nsimd};                              \
    cl::sycl::range<3> global{unum1_divisible_by_nt,unum2,nsimd};       \
    cgh.parallel_for(                                                   \
                     cl::sycl::nd_range<3>(global,local),               \
                     [=] (cl::sycl::nd_item<3> item) /*mutable*/        \
                     [[intel::reqd_sub_group_size(16)]]                 \
                     {                                                  \
                       auto iter1    = item.get_global_id(0);           \
                       auto iter2    = item.get_global_id(1);           \
                       auto lane     = item.get_global_id(2);           \
                       { if (iter1 < unum1){ __VA_ARGS__ } };           \
                     });                                                \
  });

#define accelerator_barrier(dummy) { theGridAccelerator->wait(); }

inline void *acceleratorAllocShared(size_t bytes){ return malloc_shared(bytes,*theGridAccelerator);};
inline void *acceleratorAllocDevice(size_t bytes){ return malloc_device(bytes,*theGridAccelerator);};
inline void acceleratorFreeShared(void *ptr){free(ptr,*theGridAccelerator);};
inline void acceleratorFreeDevice(void *ptr){free(ptr,*theGridAccelerator);};

inline void acceleratorCopySynchronise(void) {  theCopyAccelerator->wait(); }
inline void acceleratorCopyDeviceToDeviceAsynch(void *from,void *to,size_t bytes)  {  theCopyAccelerator->memcpy(to,from,bytes);}
inline void acceleratorCopyToDevice(void *from,void *to,size_t bytes)  { theCopyAccelerator->memcpy(to,from,bytes); theCopyAccelerator->wait();}
inline void acceleratorCopyFromDevice(void *from,void *to,size_t bytes){ theCopyAccelerator->memcpy(to,from,bytes); theCopyAccelerator->wait();}
inline void acceleratorMemSet(void *base,int value,size_t bytes) { theCopyAccelerator->memset(base,value,bytes); theCopyAccelerator->wait();}
```

SYCL/Intel

Device lambda (same as Kokkos): Capture loop body in … and inject it into offloaded code as __VA_ARGS__

```
extern hipStream_t copyStream;
extern hipStream_t computeStream;
/*These routines define mapping from thread grid to loop & vector lane indexing */
accelerator_inline int acceleratorSIMTlane(int Nsimd) {
#ifdef GRID_SIMT
  return hipThreadIdx_x;
#else
  return 0;
#endif
} // HIP specific

#define accelerator_for2dNB( iter1, num1, iter2, num2, nsimd, ... )     \
  {                                                                     \
    typedef uint64_t Iterator;                                          \
    auto lambda = [=] accelerator                                       \
      (Iterator iter1,Iterator iter2,Iterator lane ) mutable {          \
      { __VA_ARGS__;}                                                   \
    };                                                                  \
    int nt=acceleratorThreads();                                        \
    dim3 hip_threads(nsimd, nt, 1);                                      \
    dim3 hip_blocks ((num1+nt-1)/nt,num2,1); \
    if(hip_threads.x * hip_threads.y * hip_threads.z <= 64){ \
      hipLaunchKernelGGL(LambdaApply64,hip_blocks,hip_threads,          \
                         0,computeStream,                               \
                         num1,num2,nsimd, lambda);                      \
    } else { \
      hipLaunchKernelGGL(LambdaApply,hip_blocks,hip_threads,            \
                         0,computeStream,                               \
                         num1,num2,nsimd, lambda);                      \
    } \
  }
```

HIP/AMD

```
#define accelerator
#define accelerator_inline strong_inline
#define accelerator_for(iterator,num,nsimd, ... )   thread_for(iterator, num, { __VA_ARGS__ });
#define accelerator_forNB(iterator,num,nsimd, ... ) thread_for(iterator, num, { __VA_ARGS__ });
#define accelerator_barrier(dummy)
#define accelerator_for2d(iter1, num1, iter2, num2, nsimd, ... ) thread_for2d(iter1,num1,iter2,num2,{ __VA_ARGS__ });

accelerator_inline int acceleratorSIMTlane(int Nsimd) { return 0; } // CUDA specific


#define thread_for( i, num, ... )                       DO_PRAGMA(omp parallel for schedule(static)) for ( uint64_t i=0;i<num;i++) { __VA_ARGS__ } ;
#define thread_for2d( i1, n1,i2,n2, ... )  \
  DO_PRAGMA(omp parallel for collapse(2))  \
  for ( uint64_t i1=0;i1<n1;i1++) {        \
  for ( uint64_t i2=0;i2<n2;i2++) {        \
  { __VA_ARGS__ } ;                        \
  }}
```

OpenMP

Low level code can differentiate “on the device” or “on the host” with GRID_SIMT macro

SIMD vs SIMT is the main programming model difference

The GPU premise is to provide hardware for executing 3D (6D) loops

Each loop iteration appears as a distinct ‘software thread’
Each thread is told which 3D iteration tuple it must work on.

Loop iterations that are ‘close by’ other can communicate with each other and MUST coordinate memory accesses for efficiency.

SIMD = Single Instruction Multiple Data = CPU vectorization

SIMT = Single Instruction Multiple Thread = GPU hidden vectorization

SIMT in many ways makes vectorization easier and more automatic
but you need to get your head around what is happening and penetrate some jargon

SIMD / CPU model

Single Instruction

Single address/ integer variables

Vector Floating point variables

CPU memory

Aligned contiguous load/store

Vector Floating point Arithmetic

Aligned vector operations are cumbersome.

Grid uses translational symmetry to ENSURE this

SIMT / GPU model

Single Instruction

Mask

Vector address/ integer variables

"Global" GPU memory
All threads can address any word

Fast load store to 'global' memory
IF addresses contiguous (coalesced read)

Vector Floating point variables

"Local" GPU memory
Threads live in parallel universes

Fast load store to 'local' memory
IF same location on respective stack

Vector Floating point Arithmetic

Local "physical" memory address
= (relative address<<shift)|threadID

Automatically interleaves local memory if the threads are doing the same thing

Covariant programming : capturing the variation between SIMD and SIMT in a single code

The struct-of-array (SoA) portability problem:

- Scalar code: CPU needs struct memory accesses struct calculation
- SIMD vectorisation: CPU needs SoA memory accesses and SoA calculation
- SIMT coalesced reading: GPU needs SoA memory accesses struct calculation
- GPU data structures in memory and data structures in thread local calculations *differ*

Model	Memory	Thread
Scalar	Complex Spinor[4][3]	Complex Spinor[4][3]
SIMD	Complex Spinor[4][3][N]	Complex Spinor[4][3][N]
SIMT	Complex Spinor[4][3][N]	Complex Spinor[4][3]
Hybrid?	Complex Spinor[4][3][Nm][Nt]	Complex Spinor[4][3][Nt]

How to program portably?

- Use operator() to transform memory layout to per-thread layout.
- Two ways to access for read
- operator[] returns whole vector
  - operator() returns SIMD lane threadIdx.y in GPU code
  - operator() is a trivial identity map in CPU code
- Use coalescedWrite to insert thread data in lane threadIdx.y of memory layout.

Wilson Kernels

```
#define GENERIC_STENCIL_LEG(Dir,spProj,Recon)                    \
  SE = st.GetEntry(ptype, Dir, sF);                              \
  if (SE->_is_local) {                                           \
    int perm= SE->_permute;                                      \
    auto tmp = coalescedReadPermute(in[SE->_offset],ptype,perm,lane);  \
    spProj(chi,tmp);                                             \
  } else {                                                       \
    chi = coalescedRead(buf[SE->_offset],lane);                  \
  }                                                              \
  acceleratorSynchronise();                                            \
  Impl::multLink(Uchi, U[sU], chi, Dir, SE, st);                 \
  Recon(result, Uchi);
```

```
template <class Impl> accelerator_inline
void WilsonKernels<Impl>::GenericDhopSiteDag(StencilView &st, DoubledGaugeFieldView &U,
                                             SiteHalfSpinor *buf, int sF,
                                             int sU, const FermionFieldView &in, FermionFieldView &out)
{
  typedef decltype(coalescedRead(buf[0]))   calcHalfSpinor;
  typedef decltype(coalescedRead(in[0])) calcSpinor;
  calcHalfSpinor chi;
  //  calcHalfSpinor *chi_p;
  calcHalfSpinor Uchi;
  calcSpinor result;
  StencilEntry *SE;
  int ptype;
  const int Nsimd = SiteHalfSpinor::Nsimd();
  const int lane=acceleratorSIMTlane(Nsimd);
  GENERIC_STENCIL_LEG(Xp,spProjXp,spReconXp);
  GENERIC_STENCIL_LEG(Yp,spProjYp,accumReconYp);
  GENERIC_STENCIL_LEG(Zp,spProjZp,accumReconZp);
  GENERIC_STENCIL_LEG(Tp,spProjTp,accumReconTp);
  GENERIC_STENCIL_LEG(Xm,spProjXm,accumReconXm);
  GENERIC_STENCIL_LEG(Ym,spProjYm,accumReconYm);
  GENERIC_STENCIL_LEG(Zm,spProjZm,accumReconZm);
  GENERIC_STENCIL_LEG(Tm,spProjTm,accumReconTm);
  coalescedWrite(out[sF],result,lane);
};
```

Recovers parallel execution after thread divergence in if/else for boundary / interior link.

Multiplying by gauge link is provided by a “policy” template class.

Share kernel source for BOTH flavored G-parity / C* and periodic

These datatypes change covariantly
With the architecture for SIMD vs SIMT

Contrast to most portability approaches where software is invariant

Thread Divergence - GPU's HATE branchy code

The thread enable mask is important!

If some threads say "IF" and others say "ELSE" then both IF and ELSE are computed with enable masks set accordingly

- Parallel execution rapidly becomes serial execution in branchy code
  - A program can be thought of as a binary tree of branch decisions
    - Highly divergent code will have to execute down to ALL leaves of the b-tree
  - Bad news for a lot of HEP event processing code, GEANT etc…
    - (Added to problem of object orientation and virtual pointers do not work in GPU memory systems)

GPU register files are huge.

- Allows to schedule loads earlier and cover latency
- Many threads sharing same execution pipes increases latency tolerance
  - Reminder / humble-brag : I designed the adaptive prefetch engine on IBM BlueGene/Q - this also covers latency
- Register files are multi-banked (unlike CPU) with access combination rules that only the compiler really handles well

Performance

Benchmark_usqcd - produces a CSV spreadsheet of results

- Dslash DWF/ Wilson/ Staggered for various local volumes per MPI rank
- Communication bandwidth (intranode and internode)
- Memory bandwidth
- Batched CGEMM/ZGEMM performance (c.f. multiRHS multigrid, later)

A	B
Memory Bandwidth	
Bytes	GB/s per node
6291456	254.227647
100663296	3500.053754
509607936	10351.75926
1610612736	14353.04629

Communications		
Packet bytes	direction	GB/s per node
4718592	2	207
4718592	3	208
4718592	6	190
4718592	7	204
15925248	2	308
15925248	3	288
15925248	6	296
15925248	7	301
37748736	2	332
37748736	3	339
37748736	6	333
37748736	7	336

ZGEMM				
M	N	K	BATCH	GF/s per rank
16	8	16	256	5.787642
16	16	16	256	275.941053
16	32	16	256	256.925207
32	8	32	256	652.809961
32	16	32	256	180.69161
32	32	32	256	2346.463776
64	8	64	256	2338.287944
64	16	64	256	4660.337778
64	32	64	256	6563.214083
16	8	256	256	1950.83907
16	16	256	256	3647.22087
16	32	256	256	5604.080501
32	8	256	256	2874.041285
32	16	256	256	5651.272758
32	32	256	256	9905.36738
64	8	256	256	4444.295629
64	16	256	256	8589.934592
64	32	256	256	14851.20089
8	256	16	256	1928.415632
16	256	16	256	3717.942604
32	256	16	256	4652.260936
8	256	32	256	2995.931429
16	256	32	256	5122.814046
32	256	32	256	8310.695232
8	256	64	256	4350.655689
16	256	64	256	8146.75132
32	256	64	256	12874.60221

Per node summary table				
L	Wilson	DWF4	Staggered	GF/s per node
8	192	2295	64	
12	974	8668	323	
16	2959	17458	910	
24	9732	28906	3605	
32	17661	34400	7477	

Frontier MI250X4 | SDCC-A100x4 | SDCC-H100x4 | **SDCC-H100x8** | Aurora PVCx6 | SPR Xeon+HBM | SPR+DDR | AMD CPU Genoa - 32x2

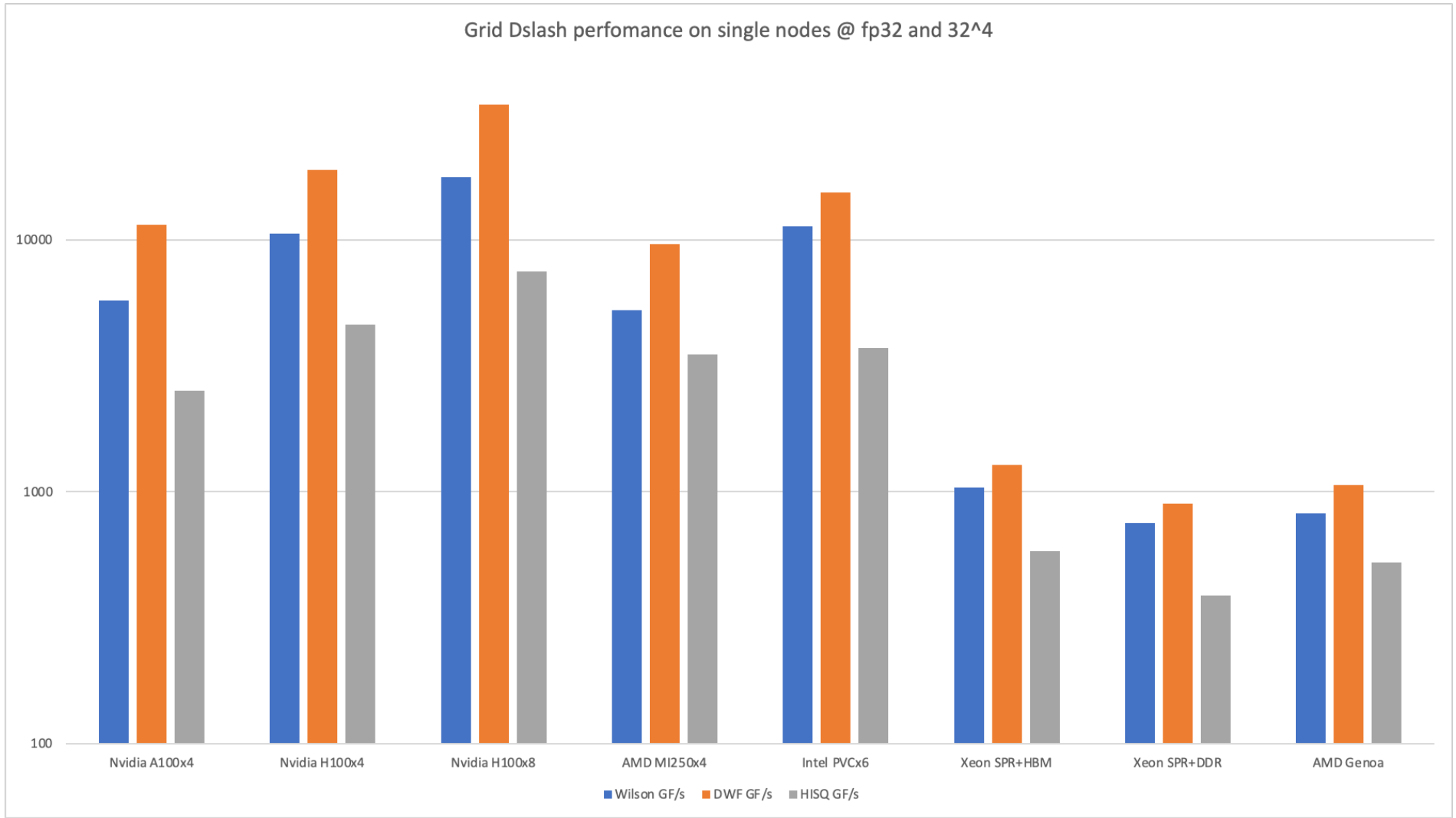
Nvidia Hopper 8x H100 @ SDCC

Multi-GPU nodes give between 11TF/s and 40TF/s on AMD, Nvidia and Intel GPU nodes

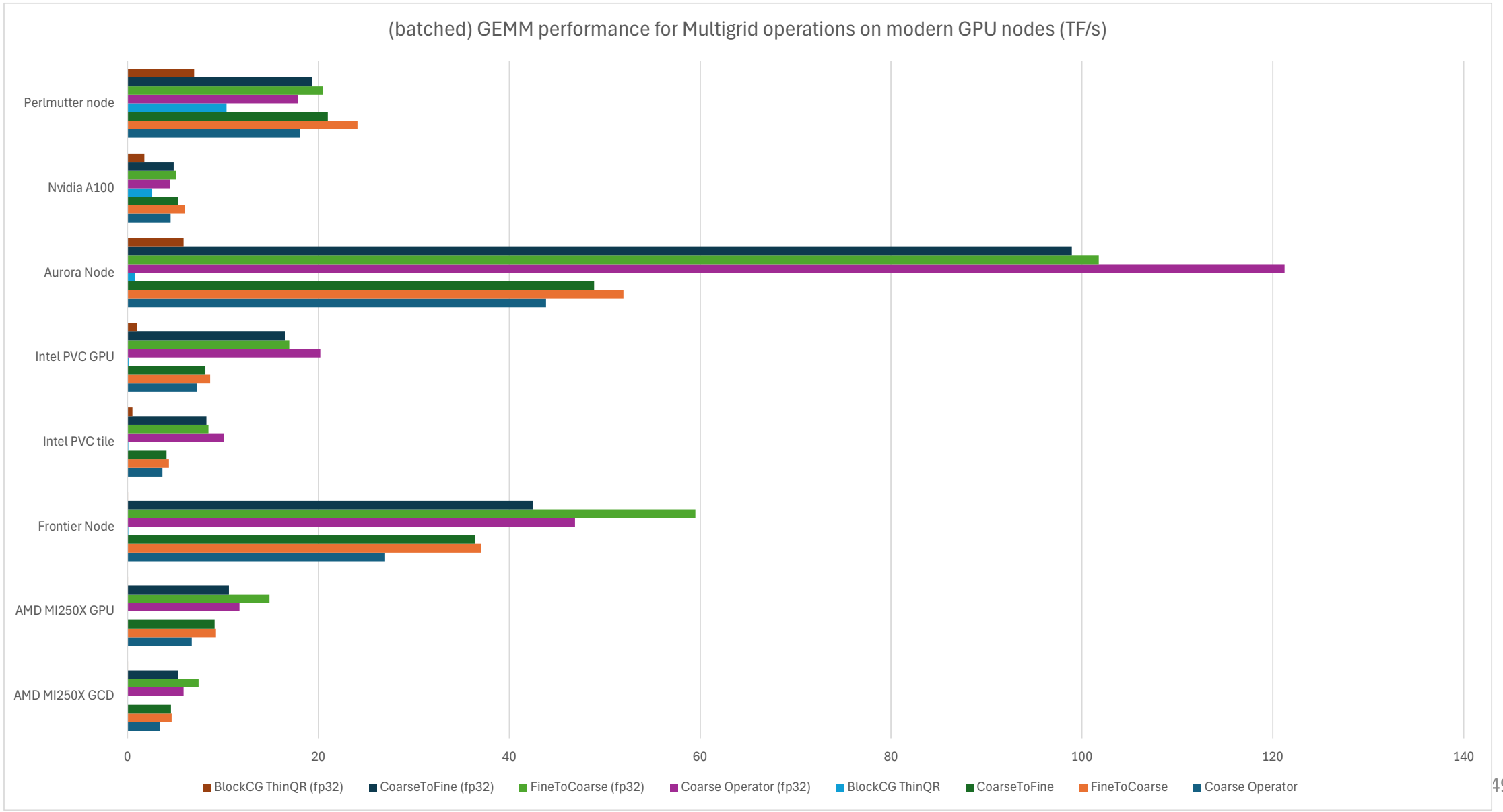
Intel PVC: Aurora@ANL

AMD MI250: Frontier@ORNL, Lumi-G@CSC

A100: Perlmutter@NERSC, Booster@Juelich, Tursa@Edinburgh, Leonard@Cineca

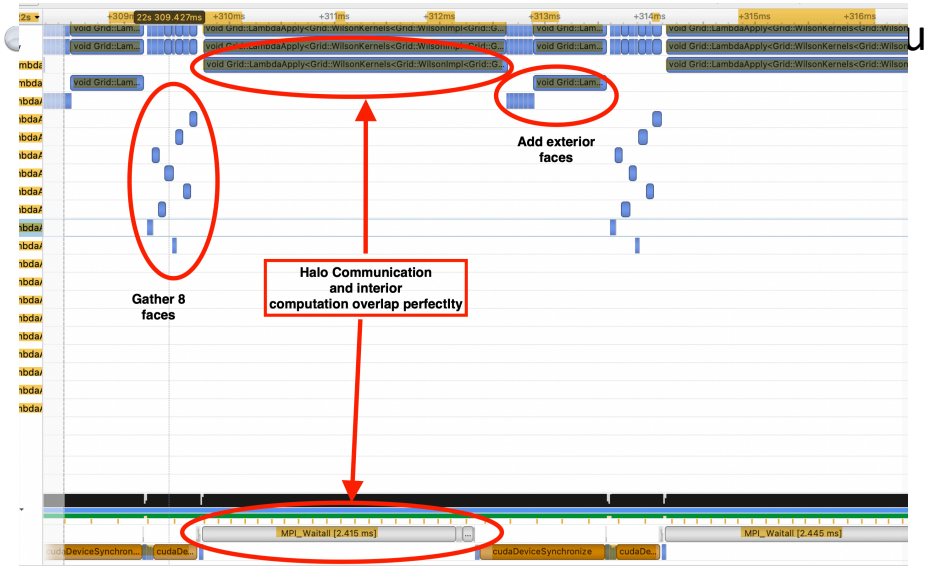


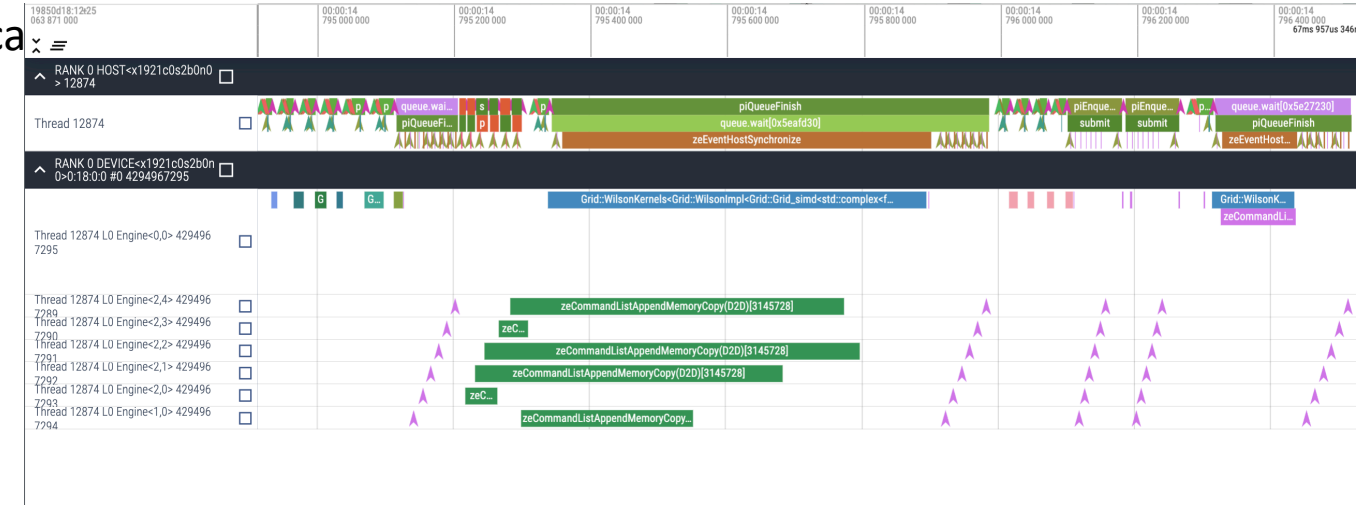
Computation part of Coarse grid operators for multiple RHS can perform VERY well



Communications performance is key to scalability

- Overlap communication and computation efficiently from 16^4 - 24^4 per GPU with 200GBit/s HFI per GPU
- 180GB/s bidirectional HaloExchange on 4x HPE Slingshot-11-200 OR 4x HDR-200
  - Booster, Tursa, Perlmutter, Frontier all scale very well
- Aurora scales very well with up to 270GB/s bidirectional BW when placed in CPU's HBM (6 HFI's Slingshot-11)
  - Aurora is preproduction, has early software, and results are subject to change.
- Systems that under provision network rapidly hit scaling problems



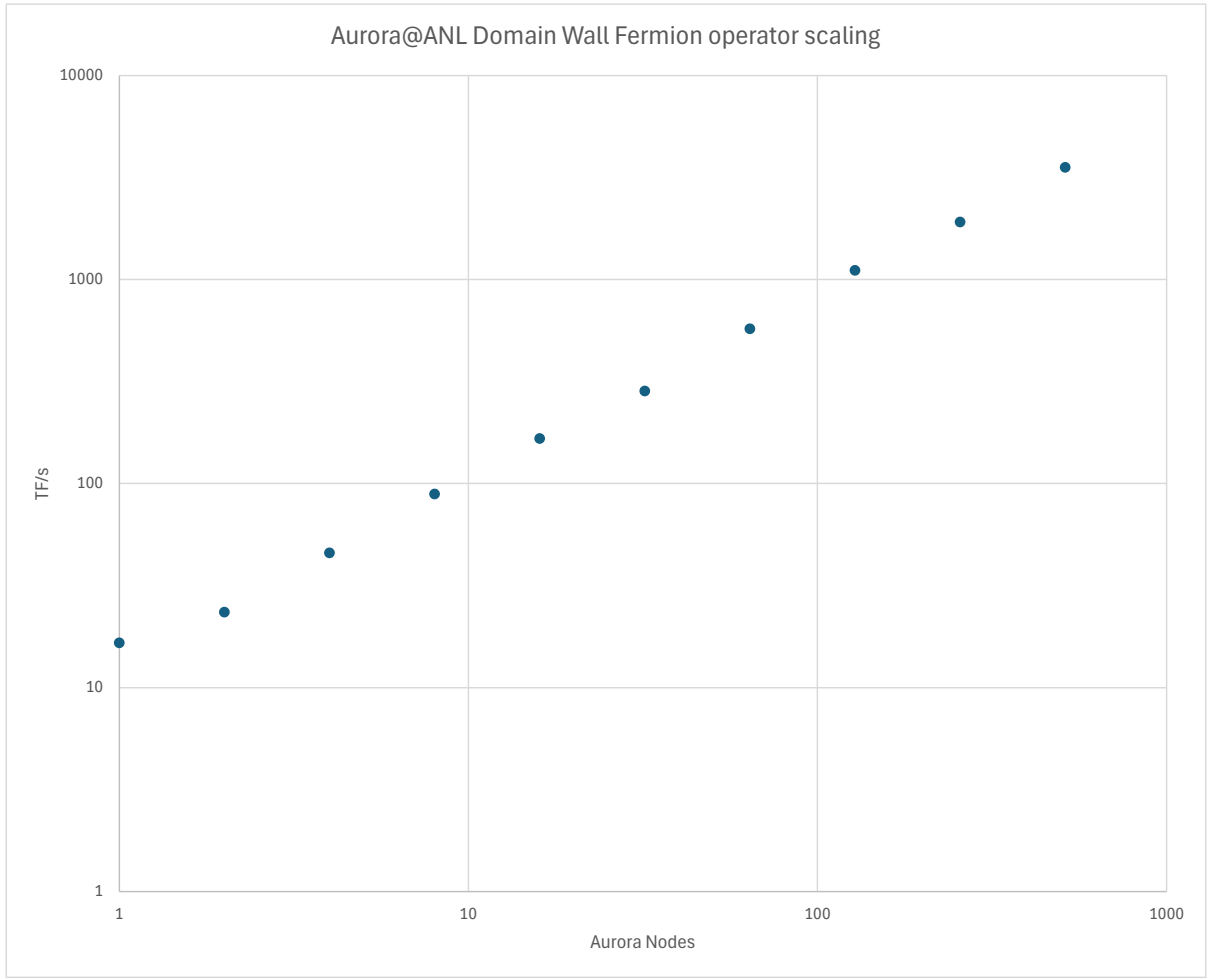


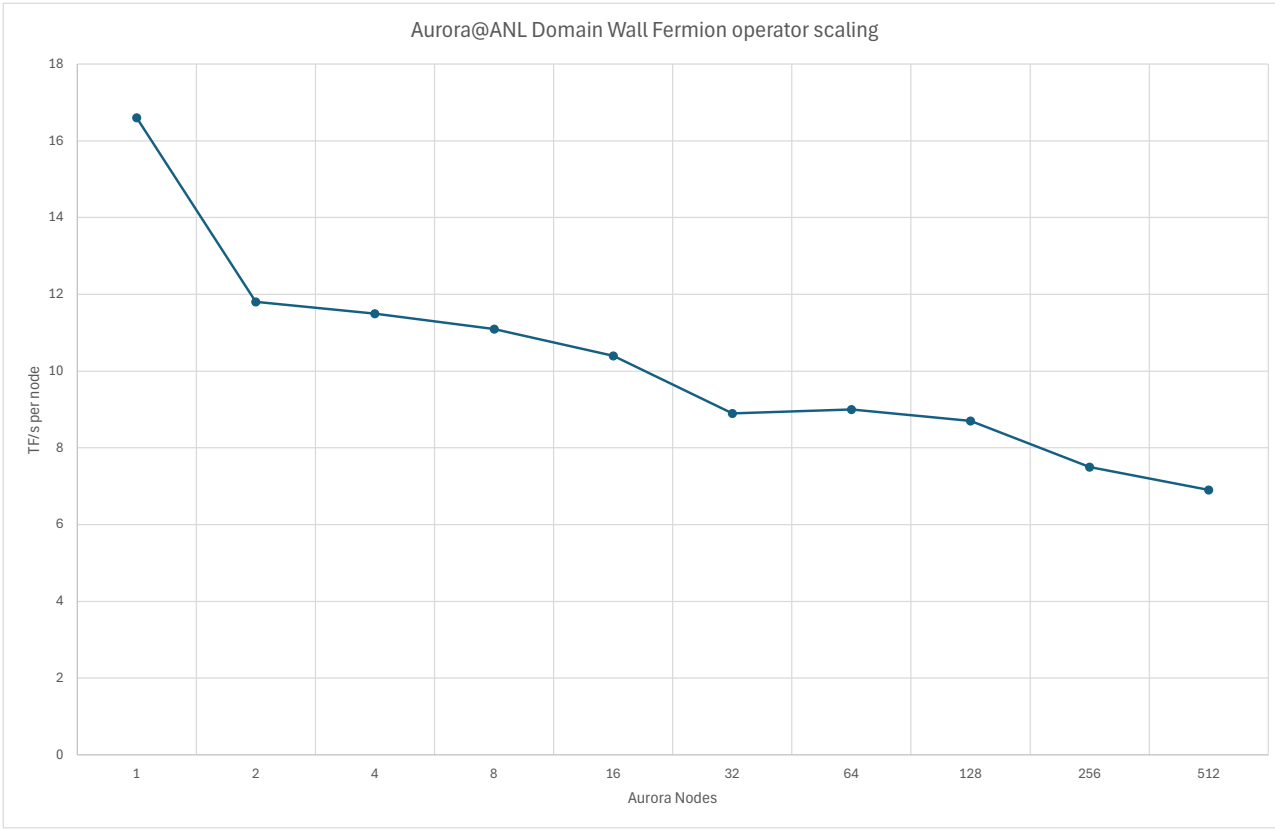
Tursa & Booster 32^4 per GPU
A100 x 4 + HDR-200 x 4
11TF/s per node fp32

Aurora, 16^3 x 32 per tile
A100 x 4 + HDR-200 x 4
16-22 TF/s per node fp32 (preliminary)

Preliminary scaling on Aurora.

The system is preproduction, has early software and results are subject to change.





Summary

- Grid Software is portable and performant on a broad range of architectures
  - Software design is important
  - After a lot of work, we run efficiently on many different high end computers around the world

- Multiple right hand side multigrid can use tensor core hardware very efficiently. Up to 100 TF/s per node !
  - > 20x speed up on MDWF propagator compared to red-black CG baseline
  - Alternative to Volume^2 eigenvector deflation. Makes largest volumes practical.
  - Will now looks at HMC appropriate multigrid algorithms:
    - direct coarsen PVdagM, use spectral improvement noted by Weinberg, Brower; avoid squaring condition number!

- Jacobian computable stout-like field transformation HMC is running in production on 128^3 x 288 3.5 GeV lattice
  - Combined with long trajectories obtain 3.5x autocorrelation improvement
  - Topological sampling is restored (for now)
  - Need to investigate how and why it works in more detail (may lead to further gains)

- **The level of optimization required to accompany algorithm development to make practical is genuinely *distressing***