Rearchitecting QUDA for multi-RHS computations

Kate Clark

Lattice 2024

Bálint Joó, Jiqun Tu, Mathias Wagner, Evan Weinberg

Motivation

HPC is Trending

- Three performance limiter trends are apparent in High Performance Computing
  - Memory bandwidth limited
  - Parallelism Limited
  - Energy Limited (more recent)

- This work seeks to address all of these limiters

- (Results are *extremely* preliminary)

**ECP benchmarks apps

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, etc. Provides solvers for all major fermionic discretizations, with multi-GPU support
- Maximize performance
  - Mixed-precision methods
  - Autotuning for high performance on all CUDA-capable architectures
  - Multigrid solvers for optimal convergence
  - NVSHMEM for improving strong scaling
- Portable: HIP (merged), SYCL (in review) and OpenMP (in development)
- **A research tool for how to reach the exascale (and beyond)**
  - Optimally mapping the problem to hierarchical processors and node topologies

QUDA CONTRIBUTORS

10+ years - lots of contributors

Buck Babich (NVIDIA)
Simone Bacchio (Cyprus)
Michael Balfhauf (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)
Marco Garofalo (Bonn)
Steve Gottlieb (Indiana University)
Kyriakos Hadjiyiannakou (Cyprus)
Ben Hoerz (Intel)
Dean Howarth (LBL)
Xiangyu Jiang (ITP, Chinese Academy of Sciences)
Xiao-Yong Jin (ANL)
Bálint Joó (Jlab)
Hyung-Jin Kim (BNL -> Samsung)
Bartek Kostrzewa (Bonn)
Damon McDougall (AMD)
Colin Morningstar (CMU)
James Osborn (ANL)
Ferenc Pittler (Cyprus)
Claudio Rebbi (Boston University)
Eloy Romero (William and Mary)
Hauke Sandmeyer (Bielefeld)
Aniket Sen (Bonn)
Guochun Shi (NCSA -> Google)
Mario Schröck (INFN)
Alexei Strelchenko (FNAL)
Jiqun Tu (NVIDIA)
Alejandro Vaquero (Utah University)
Michael Wagman (FNAL)
Mathias Wagner (NVIDIA)
André Walker-Loud (LBL)
Evan Weinberg (NVIDIA)
Frank Winter (Jlab)
Yi-bo Yang (CAS)

MAPPING THE DIRAC OPERATOR TO GPUS

Finite difference operator in LQCD is known as Dslash

Assign a single space-time point to each thread

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

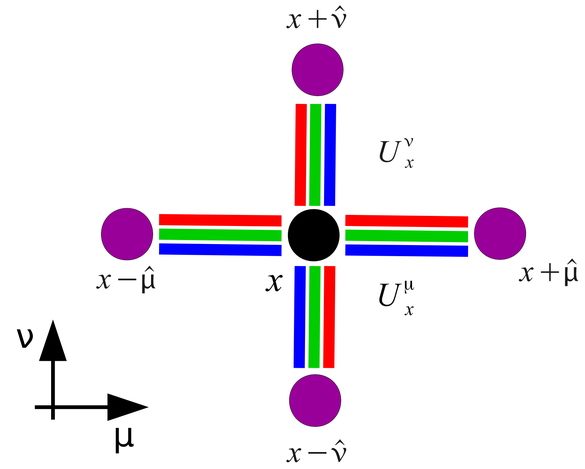
Each thread has (Wilson Dslash) 0.92 naive arithmetic intensity

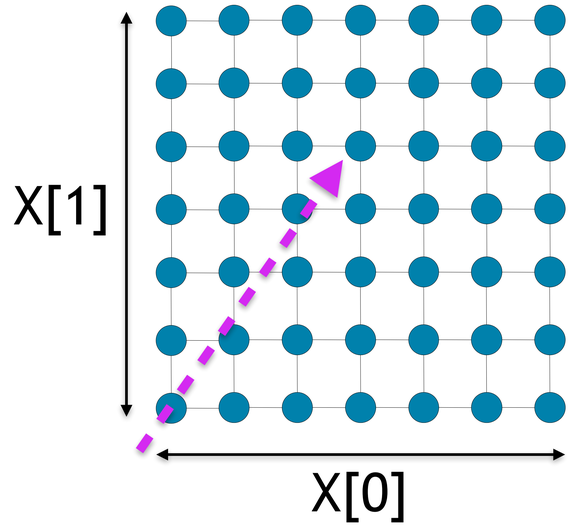
QUDA reduces memory traffic

Exact SU(3) matrix compression (18 => 12 or 8 real numbers)

Use 16-bit fixed-point representation with mixed-precision solver

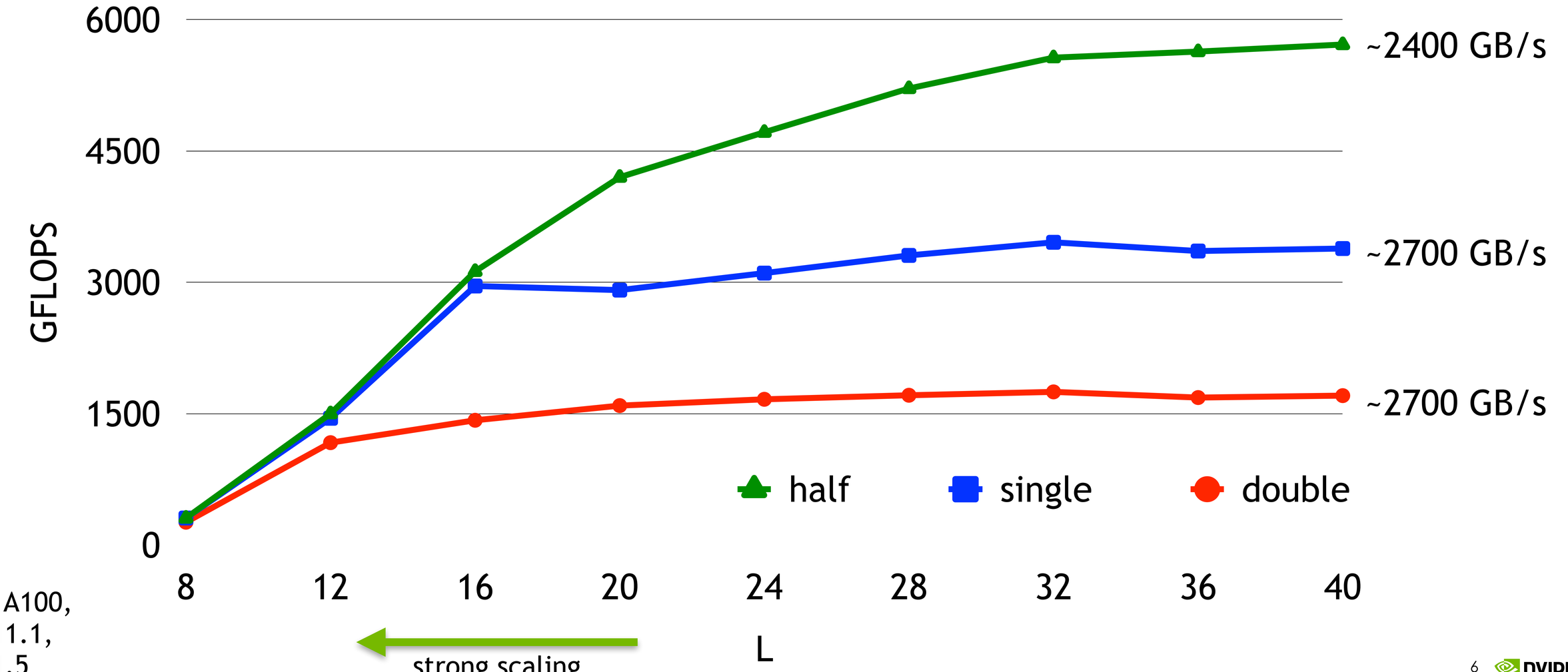
$$D_{x,x'} =$$





PARALLELISM ISN'T INFINITE

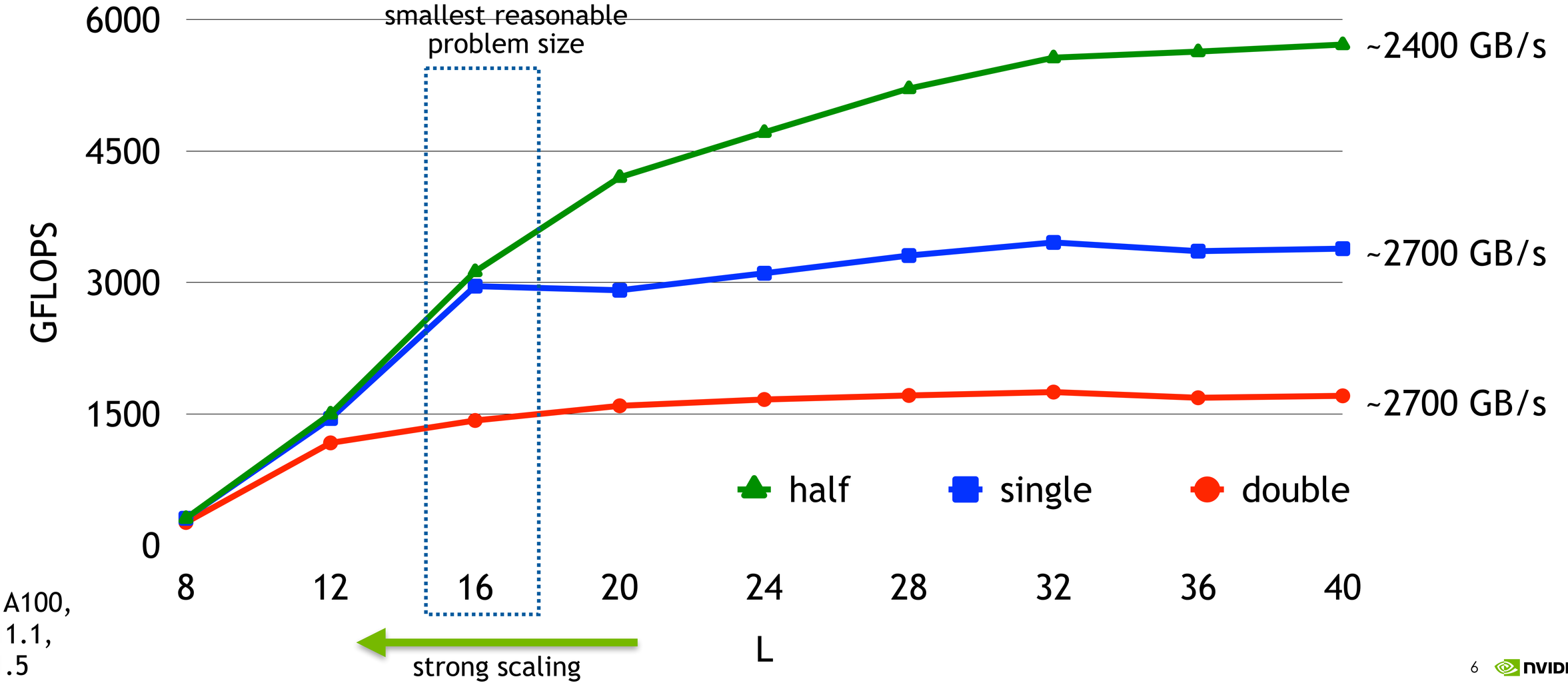
Wilson-clover stencil (Chroma, A100-80)



NVIDIA A100,
CUDA 11.1,
GCC 11.5

PARALLELISM ISN'T INFINITE

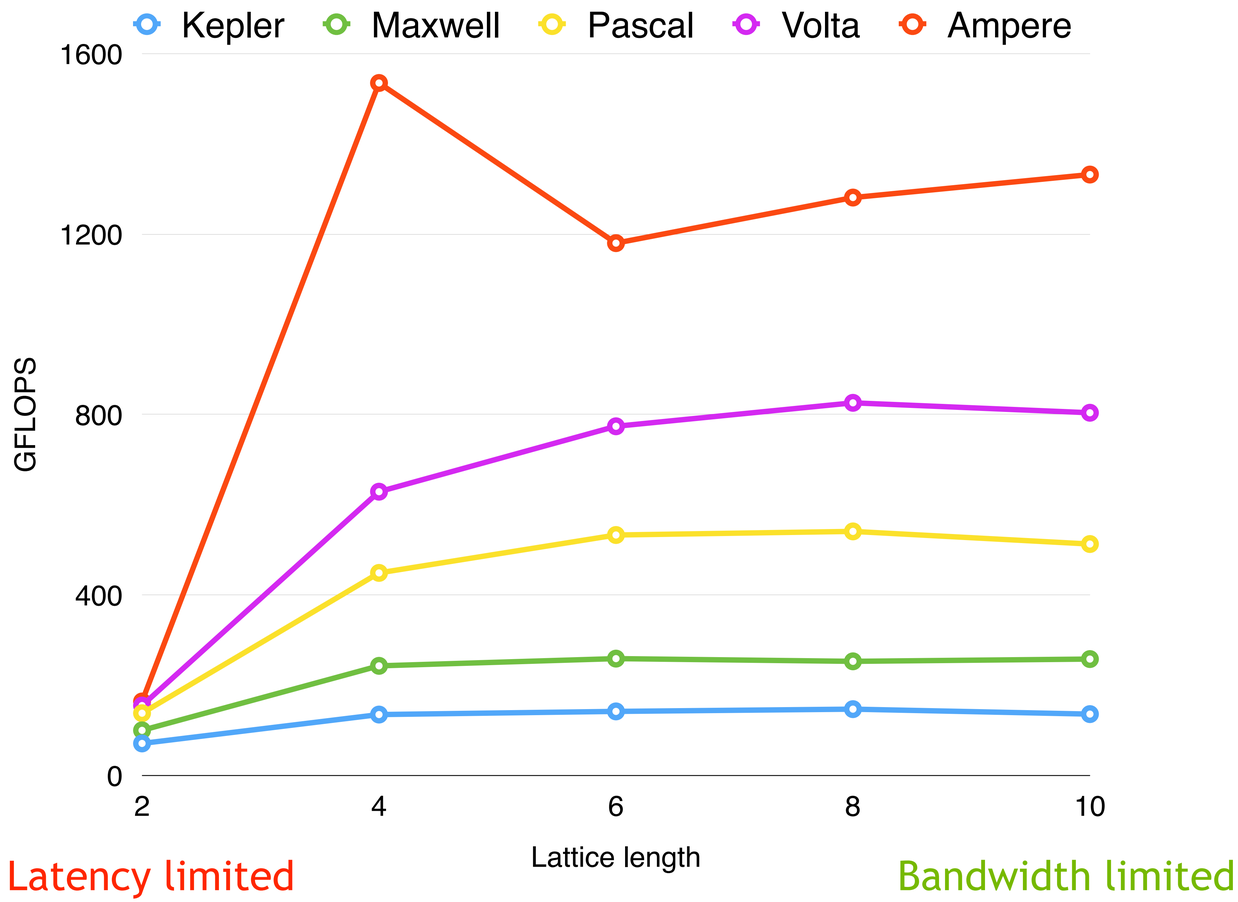
Wilson-clover stencil (Chroma, A100-80)



NVIDIA A100,
CUDA 11.1,
GCC 11.5

MULTIGRID IS EVEN WORSE

Gets harder with every generation

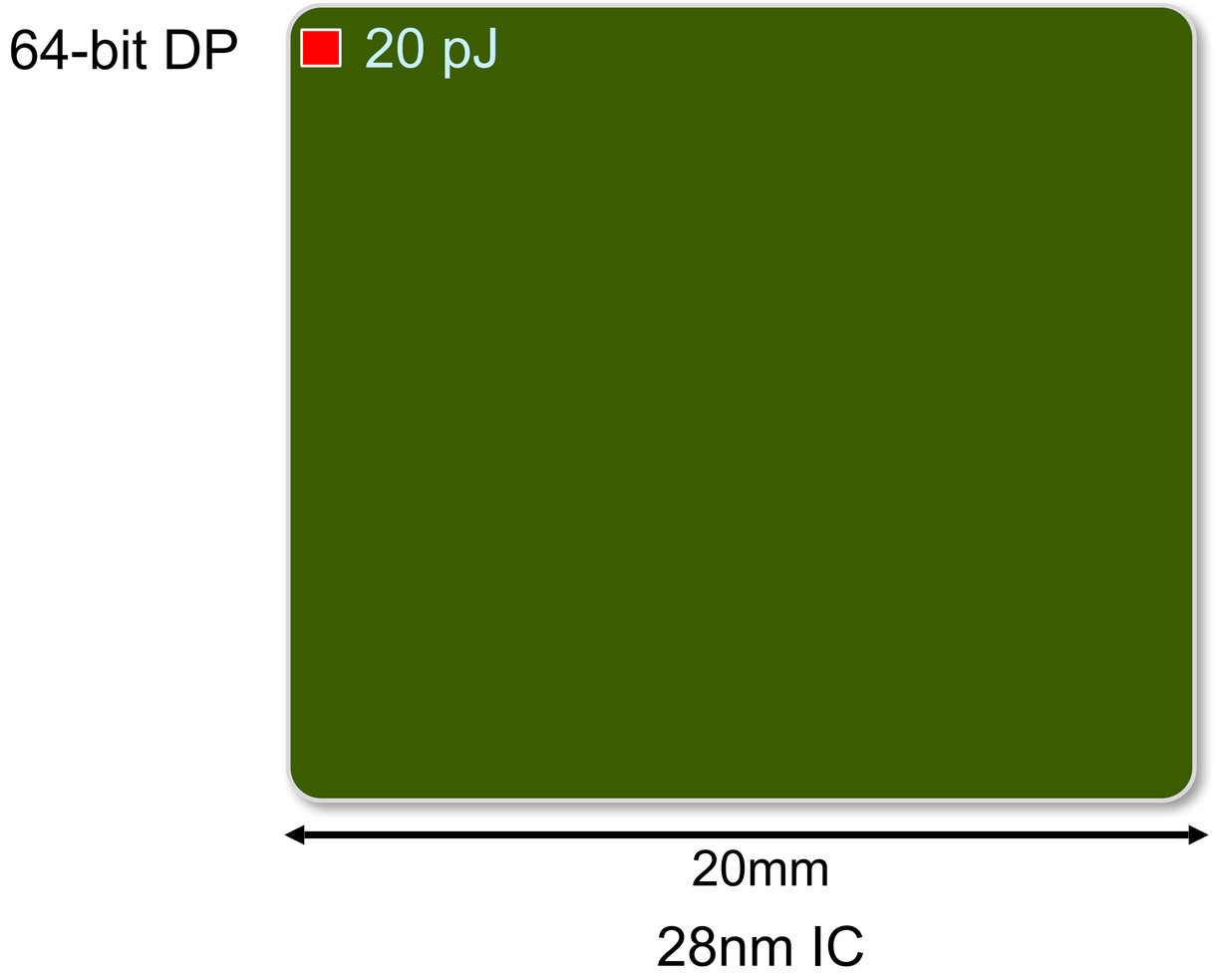


Latency limited

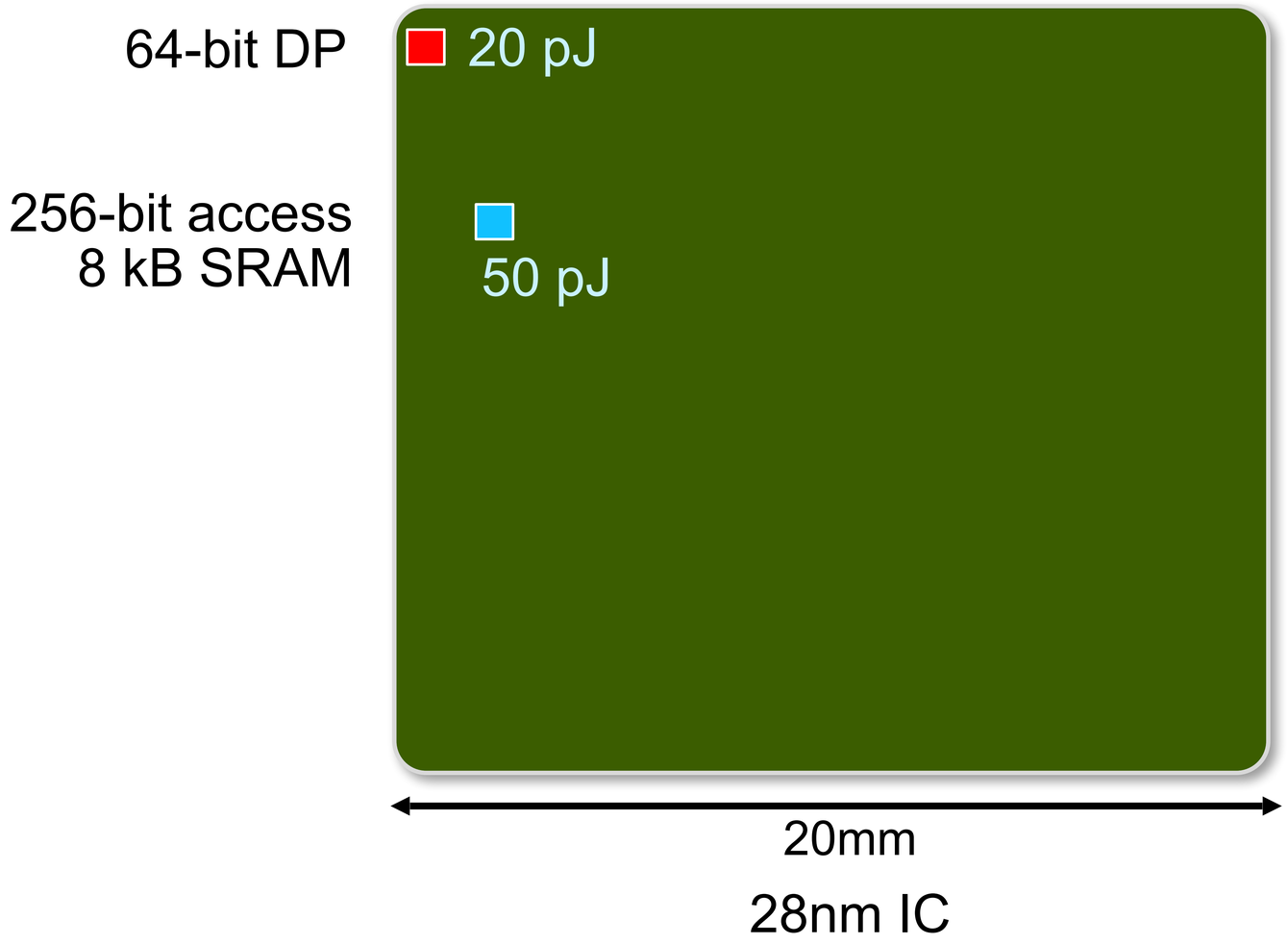
Bandwidth limited

Coarse operator performance

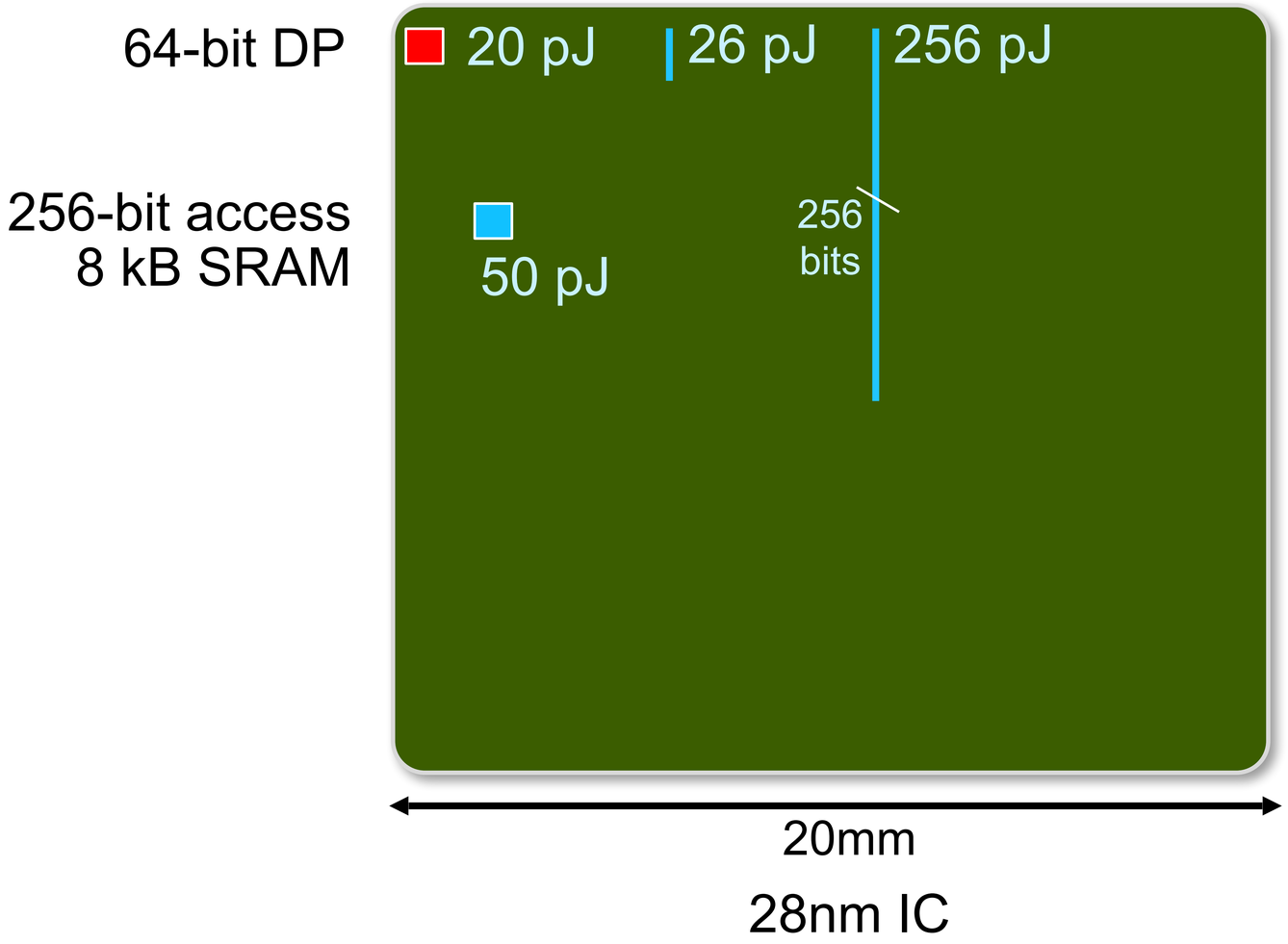
Energy Efficiency Drives Locality



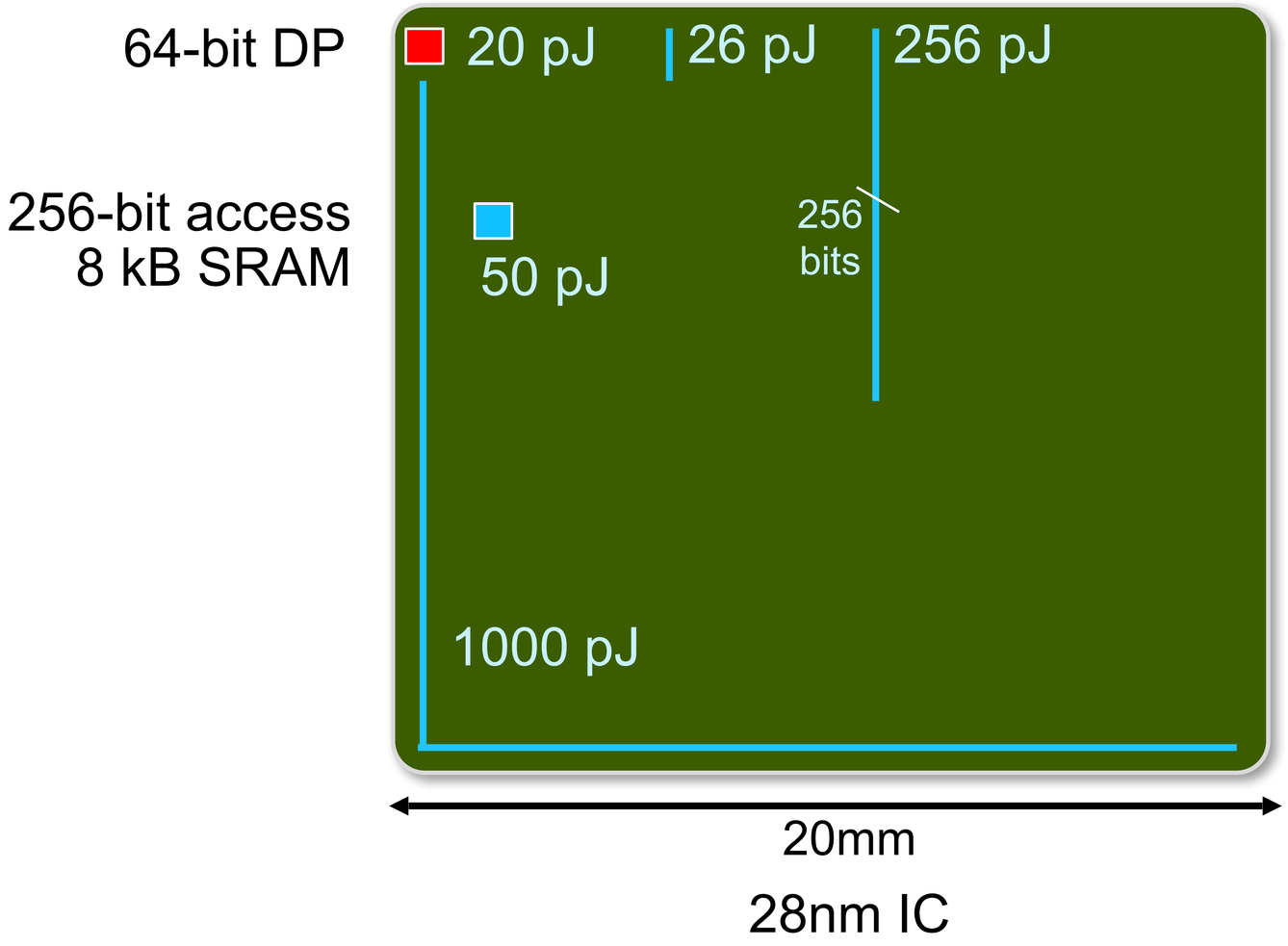
Energy Efficiency Drives Locality



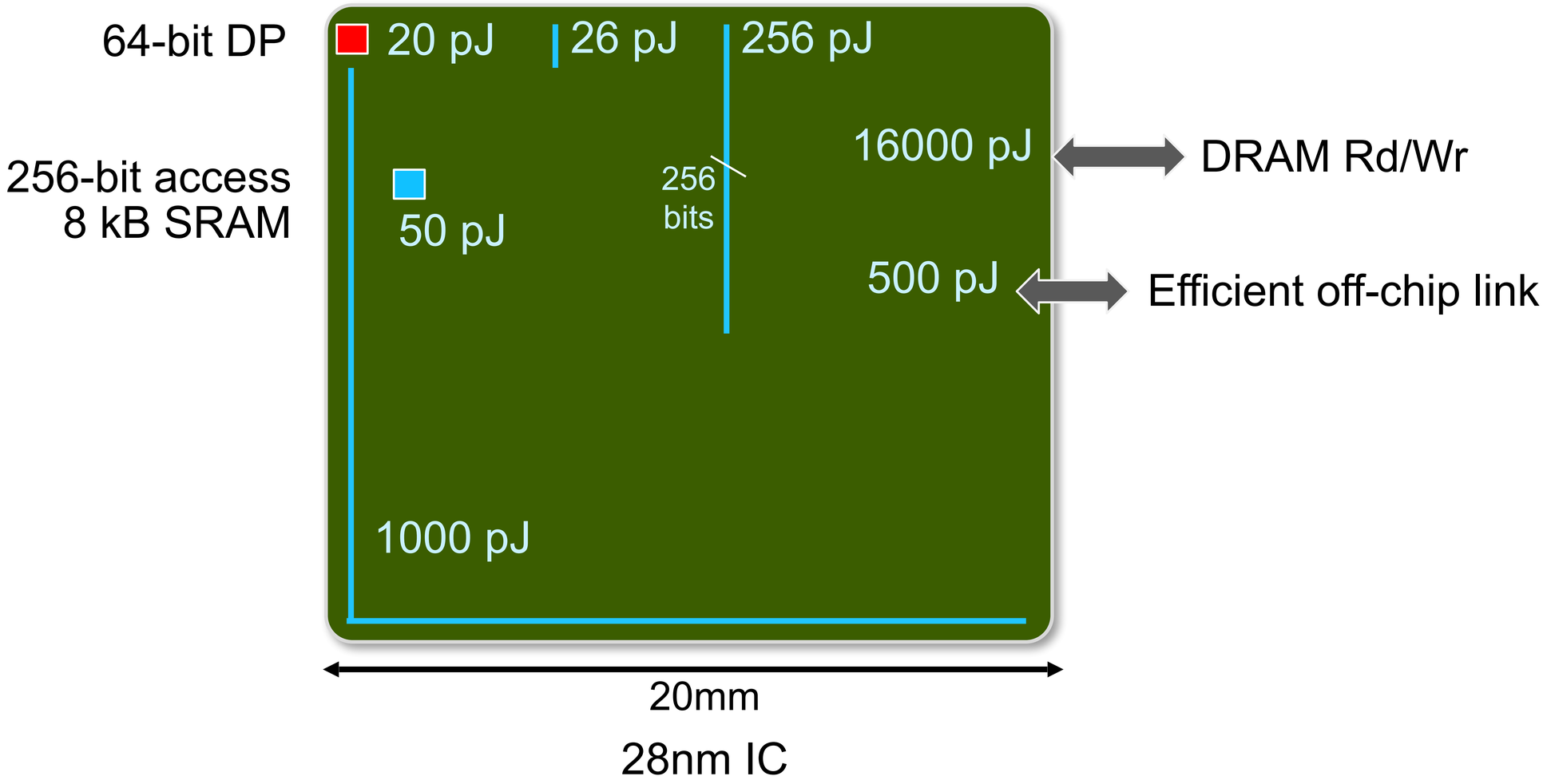
Energy Efficiency Drives Locality



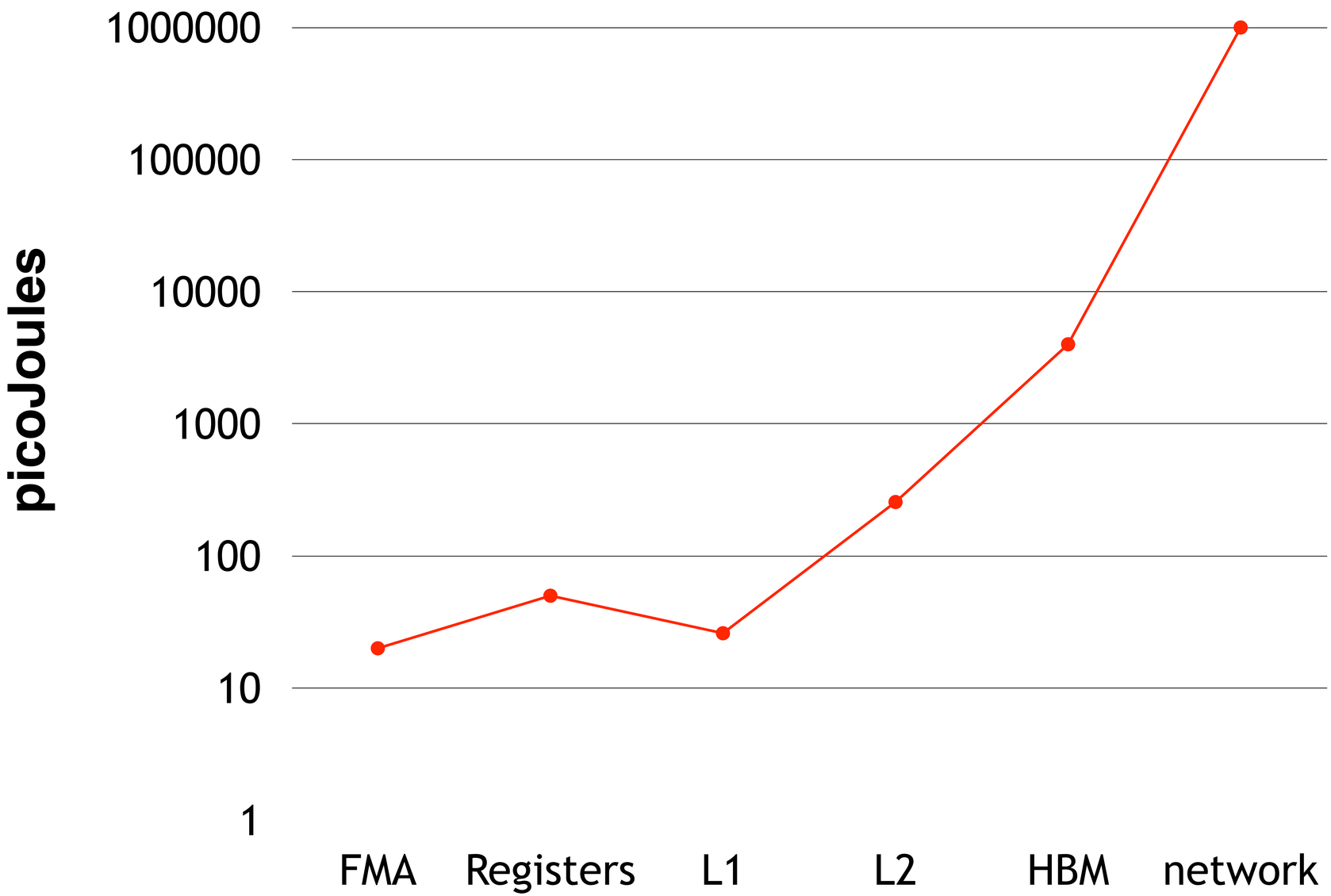
Energy Efficiency Drives Locality



Energy Efficiency Drives Locality



Locality Drives Energy Efficiency



MULTI RHS IS (SOME OF) THE SOLUTION

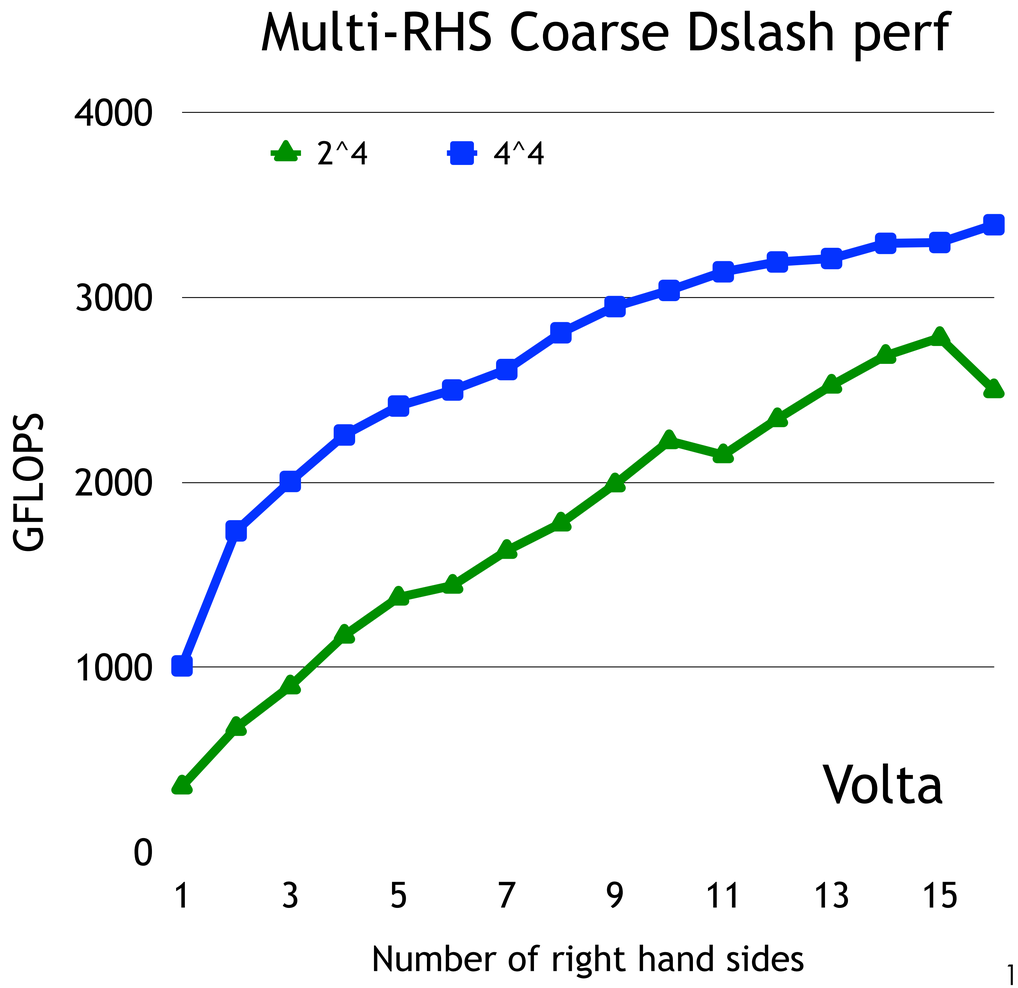
Locality, Parallelism, Energy

- Batch multiple RHS computation in a single kernel
- Memory traffic reduction
  - Gauge field load is shared across multiple RHS
  - Gauge field remains in cache after first touch
  - Traffic reduces as $\frac{1}{N_{rhs}}$
- Parallelism scales with number of RHS
- Energy reduces with decreased memory traffic
  - Power may go up due to faster rate of computation
  - Actual power efficiency will increase

MULTI RHS IS (SOME OF) THE SOLUTION

Locality, Parallelism, Energy

- Batch multiple RHS computation in a single kernel
- Memory traffic reduction
  - Gauge field load is shared across multiple RHS
  - Gauge field remains in cache after first touch
  - Traffic reduces as $\frac{1}{N_{rhs}}$
- Parallelism scales with number of RHS
- Energy reduces with decreased memory traffic
  - Power may go up due to faster rate of computation
  - Actual power efficiency will increase



Rearchitecting for MRHS

- Previously deployed block CG for staggered fermions in QUDA **arXiv:1710.09745**
  - Convenient to consider MRHS dimension as an “extra dimension” from architectural point of view
  - However restricts algorithmic flexibility, e.g., accessing subsets
  - Not suitable for library wide deployment

- Algorithmically might prefer to have a `std::vector<ColorSpinorField>`
  - Avoids requiring contiguous memory allocations
  - Disjoint communication buffers however would cause a significant latency overhead for halo communication
  - Arbitrary subsets will incur move / copy overheads

- Historically some of QUDA used `std::vector<ColorSpinorField*>`
  - No overhead for subsets, etc
  - Not desirable to rearchitect QUDA around passing raw pointers

Rearchitecting for MRHS

- Use `std::vector<std::reference_wrapper<ColorSpinorField>>` as the interface for all MRHS kernels?
  - Non-ownership of the fields
  - Zero overhead for taking subsets, supersets, etc.

- Extend `std::vector` to make it fit for purpose
  - `ColorSpinorField` methods available directly from `vector<std::reference_wrapper<ColorSpinorField>>`
    - e.g., querying the number of colors
    - Provides opportunity for set uniformity, parameter checking etc.
  - Auto construction of a vector container if a singleton is passed in
    - Compatibility with legacy code

- Use a single halo accessor for all RHS
  - Map RHS dimension to extra dimension for communication
  - All communication code, NVSHMEM etc., just works

Kernel Architecture

Wilson Dslash

- QUDA uses opaque “accessors” for all data access
- Implementation is simple: maintain an array of accessors, one per RHS
- Separate accessor for the ghost zones used by all RHS

```
template <typename Float, int nColor, int nDim, QudaReconstructType reconstruct>
struct WilsonArg : DslashArg<Float, nDim> {
 static constexpr int nSpin = 4;

  using F = typename colorspinor_mapper<Float, nSpin, nColor, spin_project, true>::type;
  F out[MAX_MULTI_RHS]; /** output vector field set */
  F in[MAX_MULTI_RHS];  /** input vector field set */

  using Ghost = typename colorspinor::GhostNOrder<Float, nSpin, nColor, spin_project, false>;
  Ghost halo;           /** halo accessor */
```

Parameter argument for driving the Wilson operator (abbreviated)

Array of accessors for the field bodies

Single accessor for the ghost zones

Forward derivative term

Single ghost buffer shared by all RHS
RHS index maps to the 5th dimension

Separate accessor for each RHS
RHS index maps to the accessor index

```
if (doHalo<kernel_type>(d) && ghost) {
  // we need to compute the face index if we are updating a face that isn't ours
  const int ghost_idx = (kernel_type == EXTERIOR_KERNEL_ALL && d != thread_dim) ?
    ghostFaceIndex<1, Arg::nDim>(coord, arg.dim, d, arg.nFace) : idx;

  Link U = arg.U(d, gauge_idx, gauge_parity);
  HalfVector in = arg.halo.Ghost(d, 1, ghost_idx + (src_idx * arg.Ls + coord.s) * arg.dc.ghostFaceCB[d],
                                 their_spinor_parity);

  out += fwd_coeff * (U * in).reconstruct(d, proj_dir);
} else if (doBulk<kernel_type>() && !ghost) {

  Link U = arg.U(d, gauge_idx, gauge_parity);
  Vector in = arg.in[src_idx](fwd_idx + coord.s * arg.dc.volume_4d_cb, their_spinor_parity);

  out += fwd_coeff * (U * in.project(d, proj_dir)).reconstruct(d, proj_dir);
}
```

Kernel Architecture

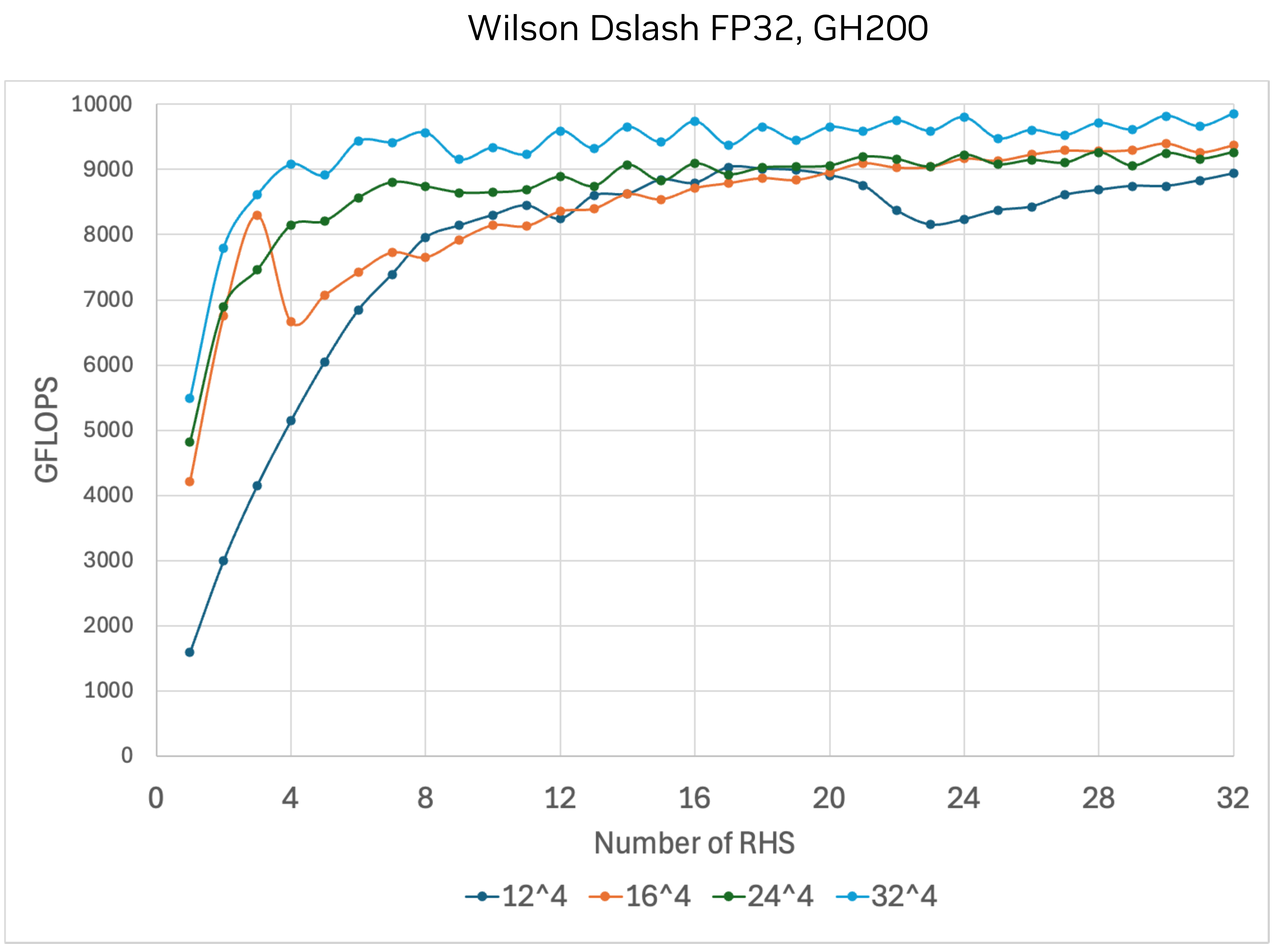
Mapping onto the hardware

- RHS index is mapped to y thread dimension
  - `src_idx = blockDim.y * blockIdx.y + threadIdx.y`
  - Autotuner will pick optimal block size, balancing locality against parallelism
  - Multiple RHS in same thread block will ensure L1 reuse of gauge field

- Maximum RHS per kernel instance controlled by **`MAX_MULTI_RHS`**
  - Exposed as a CMake parameter
  - Default is 64 on **green** team
  - Kernel argument footprint can be a problem on some non-green architectures

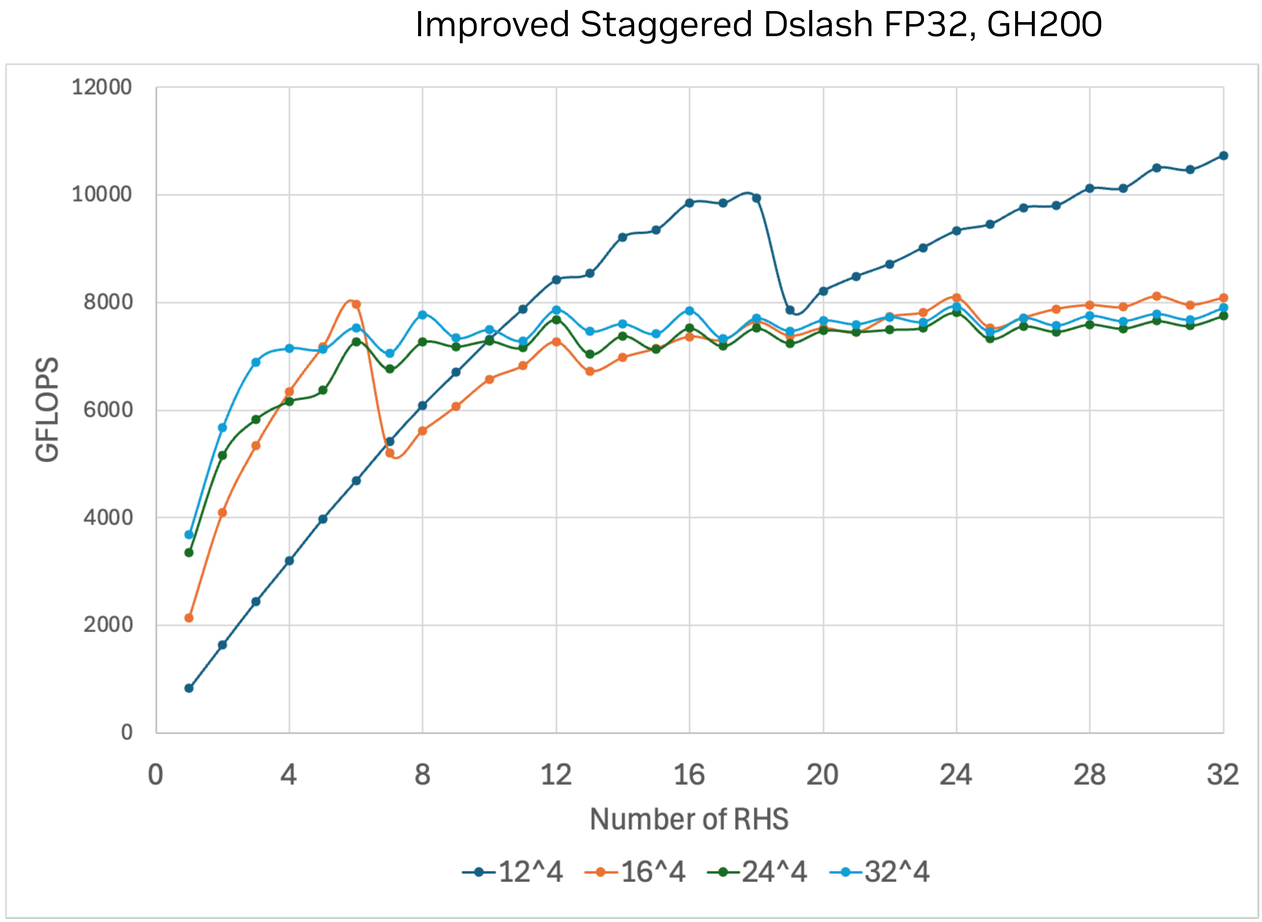
- All kernels deployed to run on arbitrary RHS
  - If set size exceeds **`MAX_MULTI_RHS`**, then split and recurse
  - Ensures that algorithms will run on any accelerator architecture

Wilson Dslash



- Smaller volumes see biggest boost in performance
  - Parallelism + Locality
- Larger volumes on see boost due to locality
- SRHS Performance model
  - Naïve $8 \times 24 + 18 \times 8 = 336$ words
  - Perfect caching $2 \times 24 + 18 \times 8 = 192$ words
- MRHS Performance model
  - Naïve asymptote $8 \times 24 = 192$ words
  - Perfect asymptote $2 \times 24 = 48$ words
- Expect speedup $\in [1.75, 4]$
  - Reality is somewhere in between

Improved Staggered



- Similar story for staggered
  - Larger speedups due to increased locality of staggered operator
- 12^4 has L1 cache quantization effects

- SRHS Performance model
  - Naïve $17 \times 6 + 36 \times 8 = 390$ words
  - Perfect caching $2 \times 6 + 36 \times 8 = 300$ words

- MRHS Performance model
  - Naïve asymptote $17 \times 6 = 102$ words
  - Perfect asymptote $2 \times 6 = 12$ words

Rewriting the Solvers

- All regular BLAS kernels rewritten to support batching

```
void axpy(double a, const ColorSpinorField &x,
          ColorSpinorField &y)
```

```
void axpy(cvector<double> &a,
          cvector_ref<const ColorSpinorField> &x,
          cvector_ref<ColorSpinorField> &y)
```

- Reductions return a vector of scalars

```
double b2 = blas::norm2(b);
```

```
auto b2 = blas::norm2(b);
```

- Solver interface promoted to batched
  - Changes required to solvers is modest and can be done incrementally

- Require convergence for all RHS before exiting solvers

- Block BLAS is not yet batch aware
  - For now performed as a serial loop over RHS
  - Impacts performance of some solvers, e.g., communication avoiding (CA) smoothers used in multigrid

QUDA uses the block TRLM algorithm
arXiv:1902.02064
Implemented by Dean Howarth

Block Lanczos + Block Deflation

HISQ Fermions

HotQCD V=48^3x12, m = 0.00167, β = 6.794

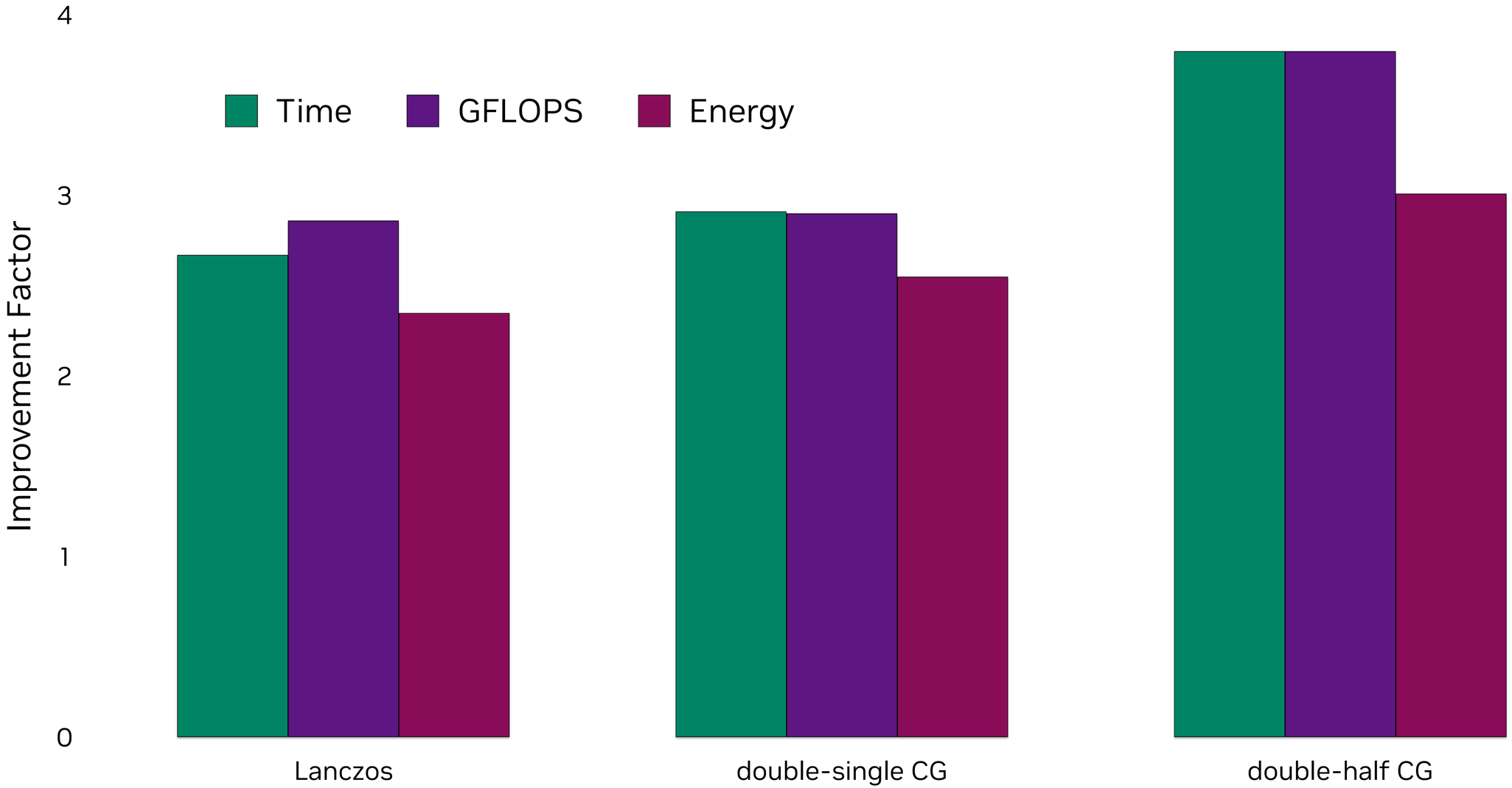
2x Quadro GV100, Gaussian sources, $\frac{||r||}{||b||} < 10^{-10}$

- Conventional deflation algorithm
  - Find eigenvectors of operator (Lanczos)
  - For each RHS
    - Deflate eigenvectors from residual (+ restart)
    - Run solver (CG)
- MRHS deflation algorithm
  - Find eigenvectors of operator (Block Lanczos)
  - Block deflate eigenvectors from set of RHS (+ restart)
  - Run MRHS solver (batch CG)
- Note energy number ignores non-GPU power
  - Energy reduction factor is *underestimated*

	SRHS	MRHS (B = 16) double-single	MRHS (B = 16) double-half
Lanczos time (sec)	155	58.0	
Lanczos GFLOPS	970	2775	
Lanczos energy (kJ)	47.0	20.0	
CG time (sec per source)	0.68	0.234	0.182
CG GFLOPS	890	2580	3390
CG energy J (per source)	220	86.4	73.1

Block Lanczos + Block Deflation

HISQ Fermions



Multigrid

- Multigrid has perhaps the greatest to benefit from MRHS
  - Coarse operator has more “colours” so more locality
  - Coarse grids are extremely parallelism challenged

- Both phases of MG can utilize MRHS
  - Batched null-space finding
  - MRHS deployment of the actual solver

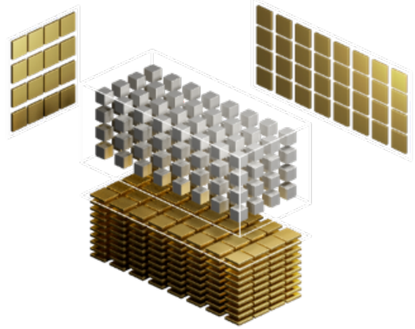
4x H100-80, tmLQCD V=32^3x64, $\kappa = 0.1373$, $c_{sw} = 1.57551$, N_{vec} = 32,64

MG Setup

Batch size	1/1	8/8	32/64
Time (sec)	13.3	6.35	5.90
TFLOPS	12.2	25.4	27.5
Speedup	1.0	2.08	2.25
Energy (kJ)	24.1	13.4	13.3

If you can’t beat them, join them

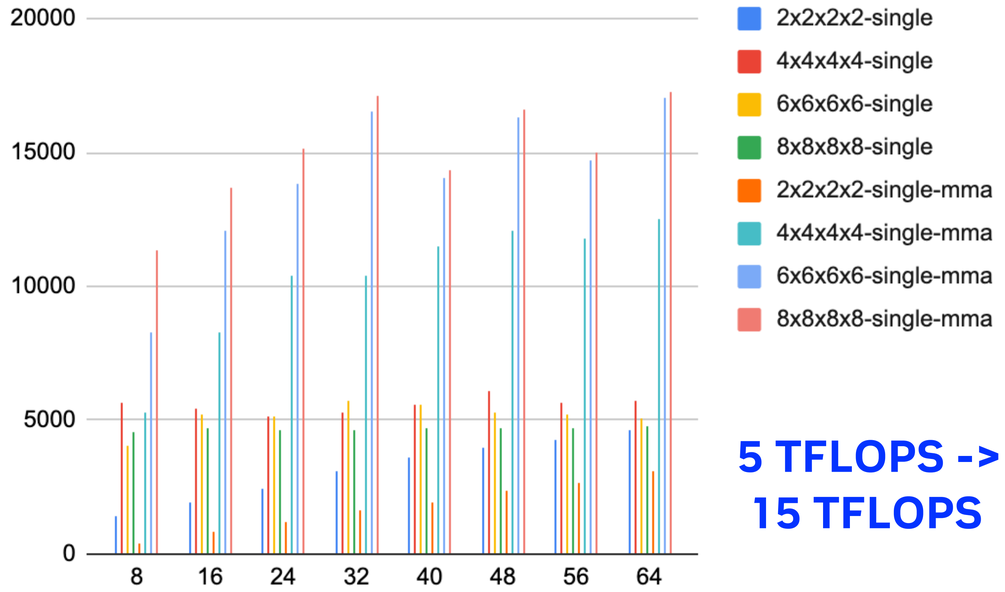
Tensor Cores



- Increasing proportion of GPU die area spent on AI
- Coarse grids have GEMM-like computations with tensor-core friendly dimensions (24, 32, 64, etc.)
- Combine multiple low-precision tensor-core operations to emulate higher precision

$$C = AB = (A_{hi} + A_{lo})(B_{hi} + B_{lo}) \sim (A_{hi}B_{hi} + A_{hi}B_{lo} + A_{lo}B_{hi})$$

  - FP32 ~ 3xTF32
  - QUDA half ~ 3x BF16
- Applying tensor cores to various MG kernels
  - Done: Coarse Dslash, link coarsening kernels
  - To do: prolongator, restrictor, block orthogonalization
- Continue to maintain non-tensor core variants in “portable QUDA”



Tensor-core accelerated multi-RHS coarse single-precision Dslash (A100)

5 TFLOPS -> 15 TFLOPS

NVIDIA

Multigrid

- Multigrid has perhaps the greatest to benefit from MRHS
  - Coarse operator has more “colours” so more locality
  - Coarse grids are extremely parallelism challenged

- Both phases of MG can utilize MRHS
  - Batched null-space finding
  - MRHS deployment of the actual solver

4x H100-80, tmLQCD V=32^3x64, $\kappa = 0.1373$, $c_{sw} = 1.57551$, N_{vec} = 32,64

MG Setup

Batch size	1/1	8/8	32/64	32/64TC
Time (sec)	13.3	6.35	5.90	3.91
TFLOPS	12.2	25.4	27.5	41.5
Speedup	1.0	2.08	2.25	3.4
Energy (kJ)	24.1	13.4	13.3	6.30

3.4x faster
and
3.8x less energy

Multigrid

4x H100-80, tmLQCD V=32^3x64, $\kappa = 0.1373$, $c_{sw} = 1.57551$, N_{vec} = 32,64, $\frac{||r||}{||b||} < 10^{-10}$

MG Solvers

Batch size	1	8	16	32	32TC
Time (sec per rhs)	0.157	0.125	0.0980	0.0889	0.0747
TFLOPS	10.7	12.8	17.8	19.4	22.8
Energy (J per rhs)	275	190	180	176	125

2.1x faster
and
2.2x less energy

- Speedups will only increase as optimization progresses

- MRHS motivates a retuning of algorithmic parameters
  - Significant cost reduction for setup provides scope to improve preconditioner quality
  - As we increase RHS, we can get a better solver at constant iteration cost

Drew Hanlon, Ben Hoerz,
Colin Morningstar, André Walker-Loud

Sink Projections

- Time-slice contraction of fermions with 3-d Laplace eigenvectors
  - Critical part of the stochastic LapH pipeline
  - CPU-based projections on Summit comparable to MG solves at physical masses (CLS E250)

- Traditionally run a serial loop over over eigenvectors and fermions

$$c_t^{s,i,j} = \sum_{\vec{x}} \psi^{s,i\,\dagger}_{\vec{x},t} \phi^{j}_{\vec{x},t}$$ s spin indices, i fermion index, j eigenvector index

- Instead deploy the calculation as a MRHS computation to increase parallelism and reuse of loads

- Use multi-level tiling to work around memory limitations and hide host <-> device transfers

- **Combination of CPU -> GPU and tiled computation ~100x speedup**
  - No longer any significant cost compared to MG solves

Summary

- Rearchitected QUDA for multi-RHS computation everywhere
  - Scalable for future architecture evolution
- MRHS solvers demonstrate significant speedup versus serial solvers
  - Speedups presently ~2-3x
  - Much more optimization coming (MG especially)
- MRHS significantly reduces energy of computation
  - Using tensor cores gives super-linear reduction
- Going forward, all stages of the LQCD pipeline should embrace this philosophy

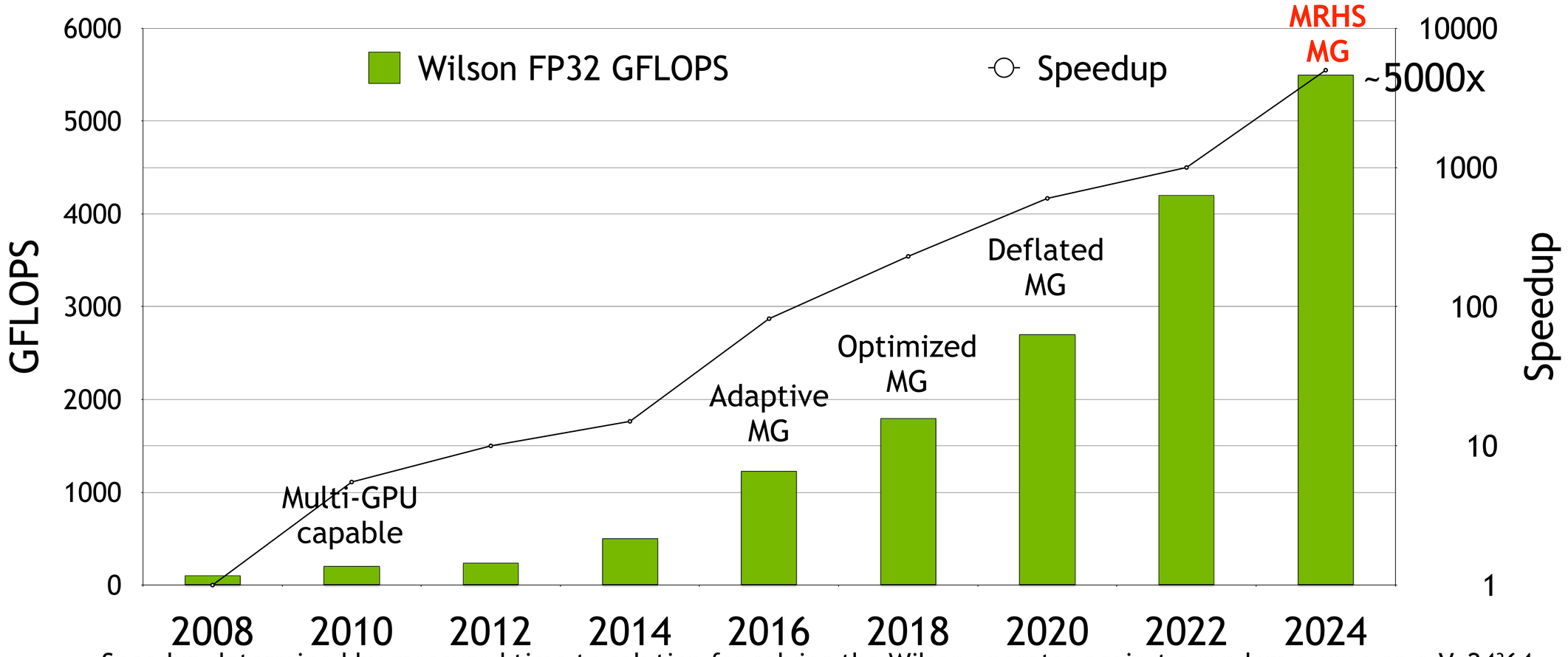
More details at the poster by **Evan Weinberg**

QUDA - Accelerated Batched Solvers for LQCD Workflows

- Split Grid + MRHS
- HISQ MG MRHS

QUDA NODE PERFORMANCE OVER TIME

Multiplicative speedup through software and hardware



Speedup determined by measured time to solution for solving the Wilson operator against a random source on a $V=24^3 64$ lattice, $\beta=5.5$, $M\pi=416$ MeV. One node is defined to be 3 GPUs.

REWORKING THE LQCD PIPELINE

slaphnn collaboration

2 nucleon (2 baryon) and 2 hadron (ππ, Kπ) and meson-baryon catering cross sections

	Classical approach	Parallelism / Intensity	Modern approach	Parallelism / Intensity
3-d Laplace eigenvectors	Lanczos	$T \times V_3$ AI ~ 1	Batched-Block-Lanczos	$B \times T \times V_3$ / AI ~ B
Clover-fermion solves	Sequential multigrid	V_4 AI ~ 1	Block multigrid	$N_\psi \times V_4$ / AI ~ N_{rhs}
Sink projections	Sequential inner products	$T \times V_3$ / AI ~ 1	Blocked inner productions => Matrix multiply	$N_\varphi \times N_\psi \times T \times V_3$ AI ~ $(N_\varphi \times N_\psi)/(N_l + N_\psi)$
Current Insertions	Sequential insertions (morally inner products)	$T \times V_3$ / AI ~ 1	Blocked insertions => Matrix multiply	$N_\psi^2 \times T \times V_3$ AI ~ $(N_\psi^2)/(2N_\psi)$