Wilson Inverter using CUDA

Michael Clark

Centre for Computational science
boston university

with
R. Babich, K. Barros,
R. Brower and C. Rebbi

GPU Hardware

GTX 280 / 285

FP32: 0.933/1.06 Tflops, FP64: 78/89 Gflops

Memory 1GB, Bandwidth 141 / 159 GBs^{-1}

230 Watts, $224 / $250

Tesla S1070

FP32: 4 Tflops, FP64: 320 Gflops

Memory 16GB, Bandwidth 408 GBs^{-1}

900 Watts, $8000

Tesla C1060

FP32: 0.933 Tflop, FP64: 78 Gflops

Memory 4GB, Bandwidth 102 GBs^{-1}

230 Watts, $1000

GPU Programming

- Need to completely rethink algorithms/problems accordingly
  - Data parallelism
  - Flops/Bandwidth ratio huge : minimize memory traffic at all costs
  - Reorder data for memory coalescing
  - Limited register size requires each thread must be *light weight*
  - Use the shared memory to share data between threads
  - Fit problems within memory constraints
  - Single precision strong, double precision weak

GPU Wilson-Dirac Matrix-Vector

$$D = \delta_{x,y} - \varkappa \sum_{\mu} \left(P^{-\mu} U_{x,y}^{\mu} \delta_{x+\mu,y} + P^{+\mu} U_{y,x}^{\mu\dagger} \delta_{x-\mu,y} \right)$$

- Assign a single spacetime point to each thread
  - Lightweight threads
- (Spin-projected) dslash operator has
  - 1320 flops per site
  - 1440 bytes per site (32 bit)
- Extremely bandwidth limited
- Need to reduce memory bandwidth at all costs
  - Exploit all symmetries of Wilson-Dirac operator

Field Ordering

- Can order data optimally for any given hardware
  - Reorder field elements for coalescing (internal d.o.f. outside spacetime)
  - Usually store field of 24 vectors
  - Instead 6 fields of float4s
- Boundary conditions destroy coalescing -> use texture cache

γ Basis

- Usually use chiral basis
- γ_4 projector is $P^{\pm 4} = \begin{pmatrix} 1 & 0 & \pm 1 & 0 \\ 0 & 1 & 0 & \pm 1 \\ \pm 1 & 0 & 1 & 0 \\ 0 & \pm 1 & 0 & 1 \end{pmatrix}$
- Instead use non-relativistic basis

$$P^{+4} = \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad P^{-4} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix}$$

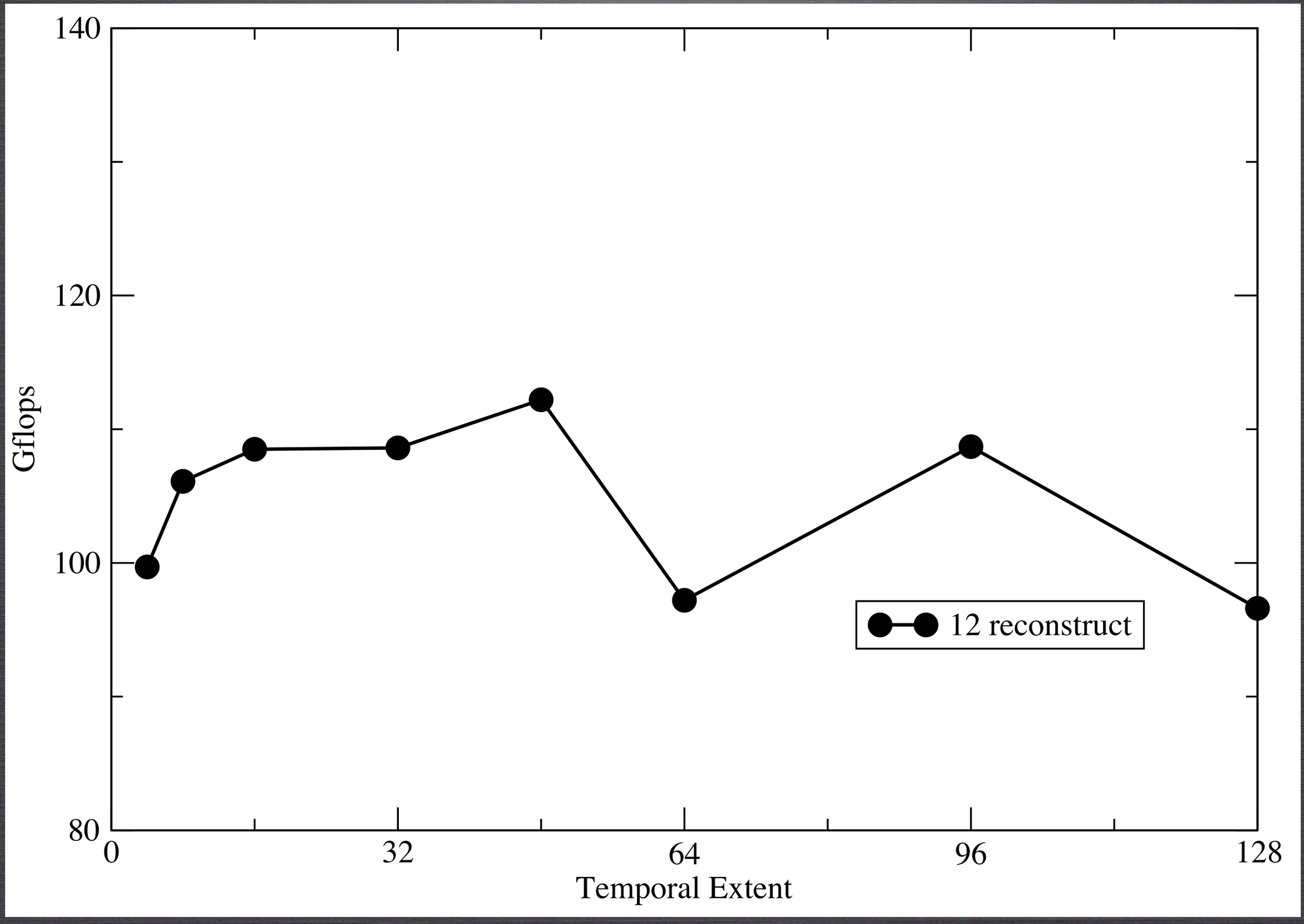
- Only need to load half-spinors in T direction
- Bytes per site 1440 -> 1248

SU(3) Representation

- SU(3) matrices are all unitary complex matrices with det = 1
  - 18 real numbers, but only 8 free parameters (generators)
- 12 number parameterization

$$\begin{pmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ c_1 & c_2 & c_3 \end{pmatrix} \longrightarrow \begin{pmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{pmatrix} \qquad \mathbf{c} = (\mathbf{a} \times \mathbf{b})^*$$

- Reconstruct full matrix on the fly
- 1152 Bytes per site (c.f. 1440)
- Additional 384 flops per site (c.f. 1368)



Wilson Matrix-Vector Performance

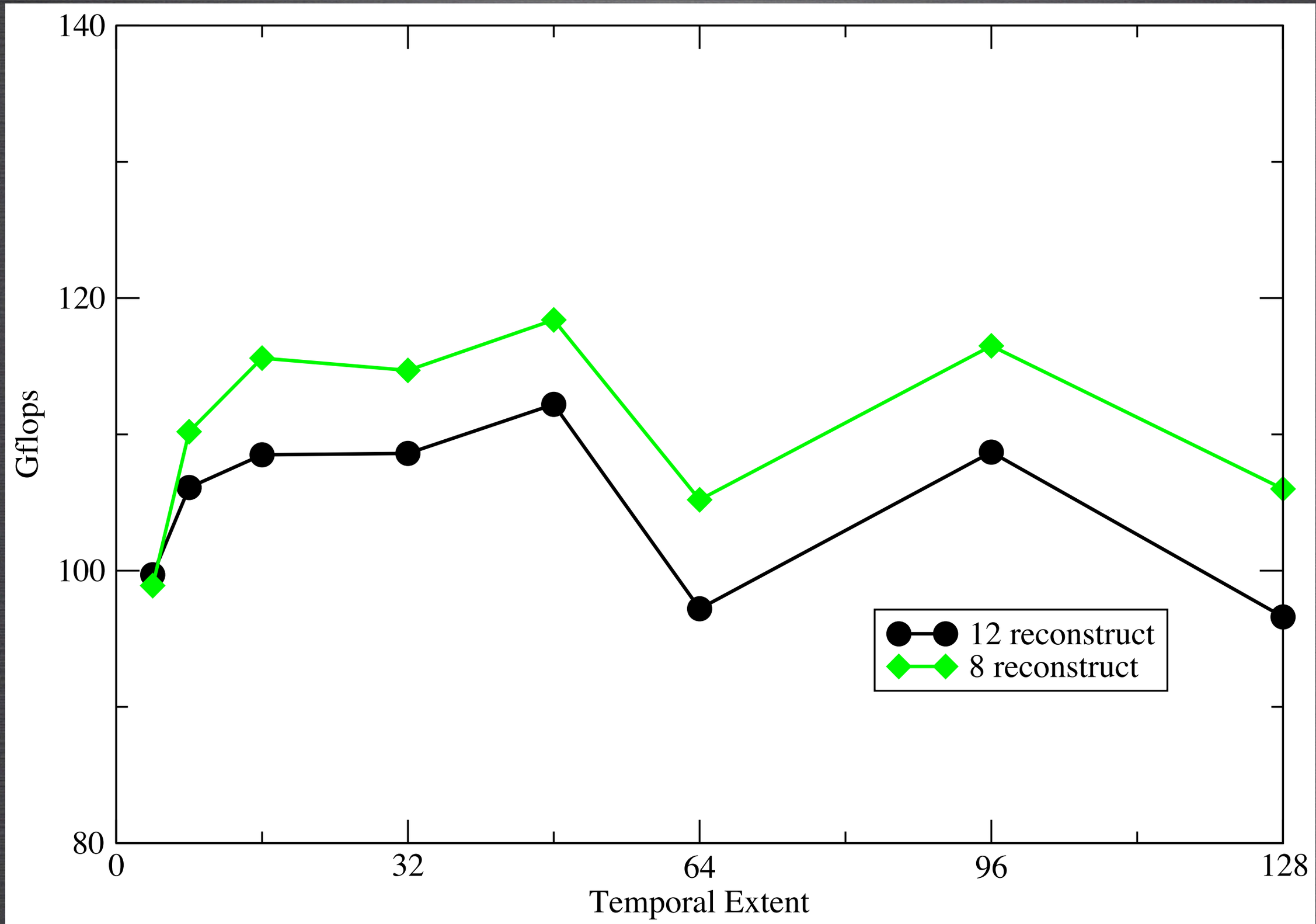
Single Precision ($V=24^3xT$)

SU(3) Representation

- Minimal 8 number parameterization

$$\begin{pmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ c_1 & c_2 & c_3 \end{pmatrix} \longrightarrow \begin{pmatrix} \arg(a_1) & \arg(c_1) & \mathrm{Re}(a_2) & \mathrm{Im}(a_2) \\ \mathrm{Re}(a_3) & \mathrm{Im}(a_3) & \mathrm{Re}(b_1) & \mathrm{Im}(b_1) \end{pmatrix}$$

  - Obtain a_1 and c_1 from normality
  - Reconstruct b_2, b_3, c_2, c_3 from SU(2) rotation
- 1024 Bytes per site (c.f. 1440)
- Additional 856 flops per site
  - Including 2 sqrt, 4 trigonometric, 2 divide

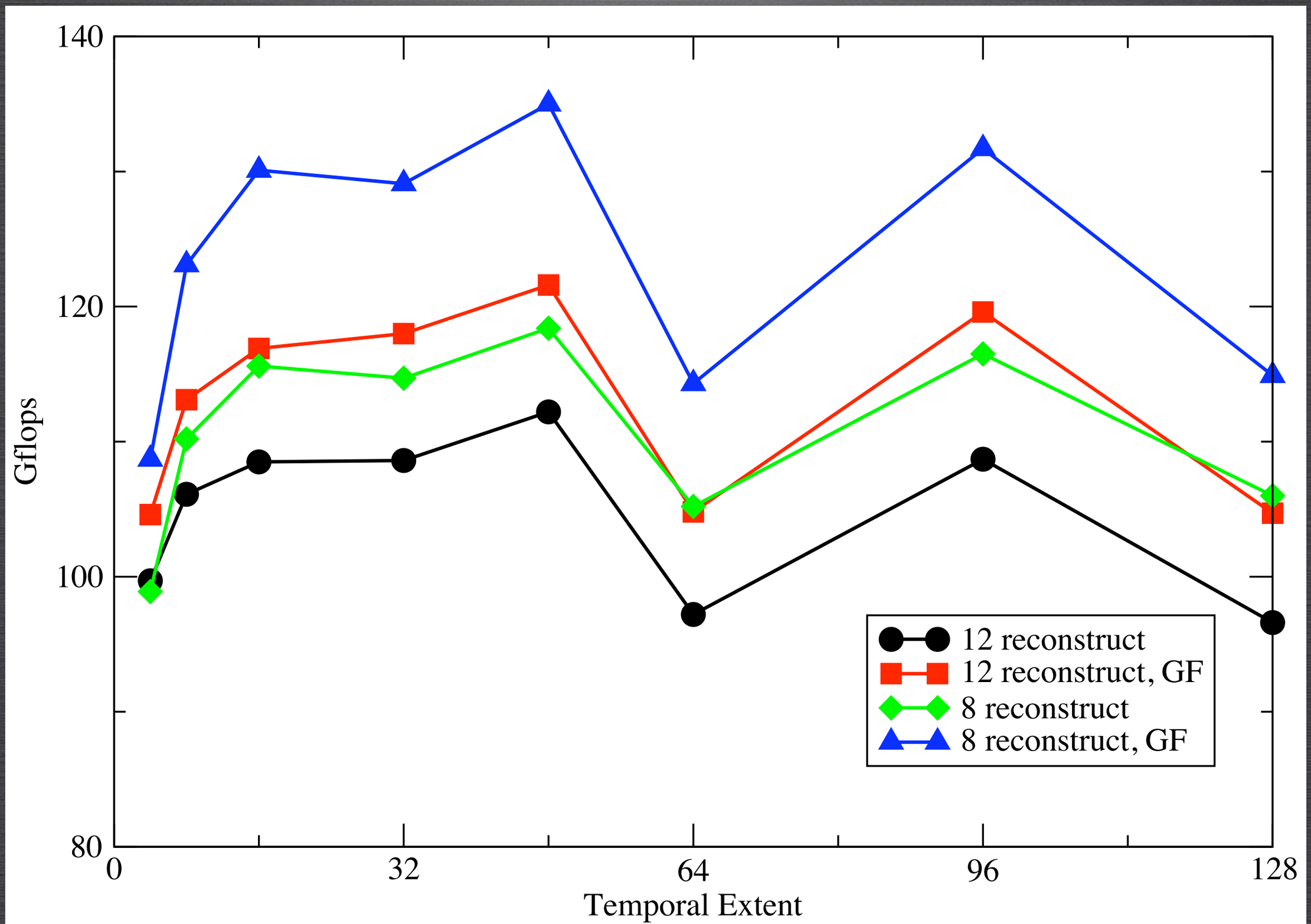


Wilson Matrix-Vector Performance

Single Precision ($V=24^3xT$)

More tricks

- Impose local gauge transformation -> temporal gauge
  - SU(3) field = unit matrix in temporal direction
  - Must calculate this transformation (done once only)
- 960 Bytes per site (c.f. 1440)
- In total: 33% bandwidth reduction



Wilson Matrix-Vector Performance

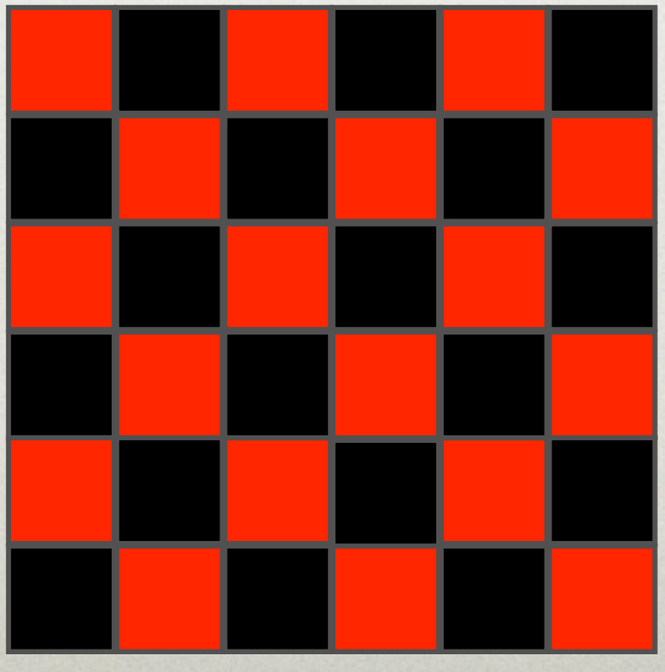
Single Precision ($V=24^3xT$)

Spinor Sharing

- Stencil methods on GPUs have large gains when vector components are shared between threads
  - Load spinor components into shared memory
  - Share spinors between threads
- Can we reuse some of the spinor loads?
  - Even-odd updating complicates things
  - 24 components per site -> Limited by shared memory

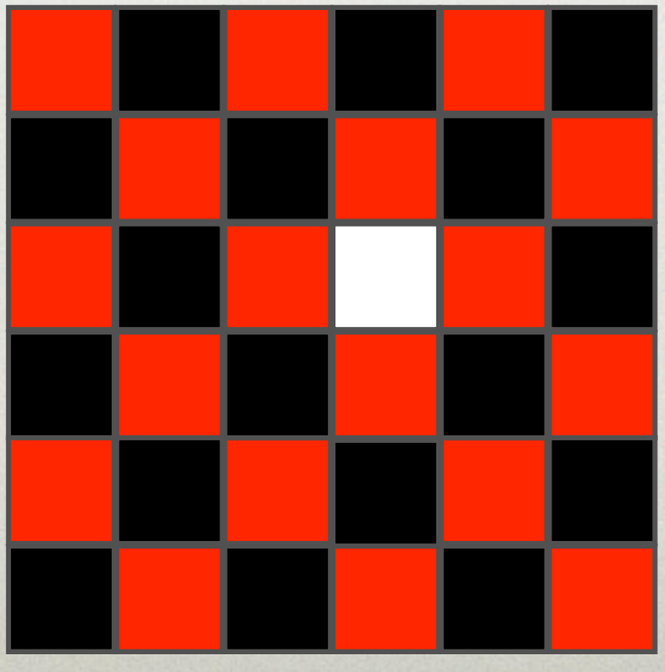
Spinor Sharing

- Stencil methods on GPUs have large gains when vector components are shared between threads
  - Load spinor components into shared memory
  - Share spinors between threads
- Can we reuse some of the spinor loads?
  - Even-odd updating complicates things
  - 24 components per site -> Limited by shared memory



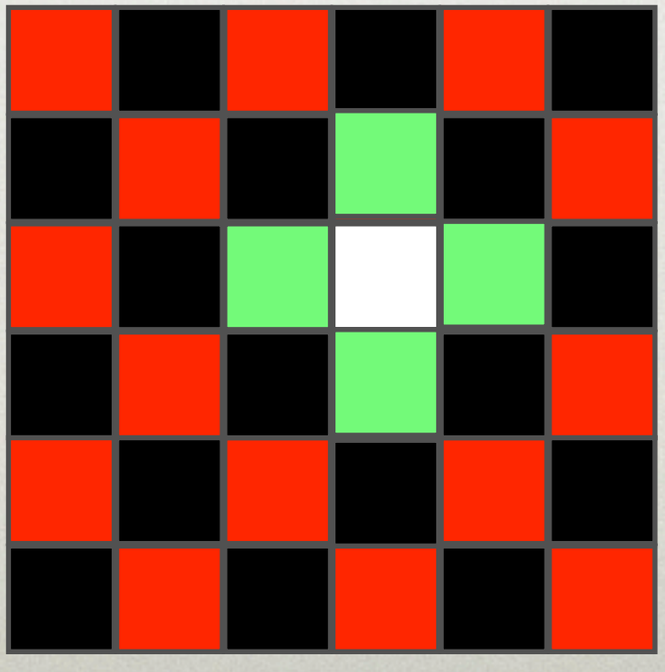
Spinor Sharing

- Stencil methods on GPUs have large gains when vector components are shared between threads
  - Load spinor components into shared memory
  - Share spinors between threads
- Can we reuse some of the spinor loads?
  - Even-odd updating complicates things
  - 24 components per site -> Limited by shared memory



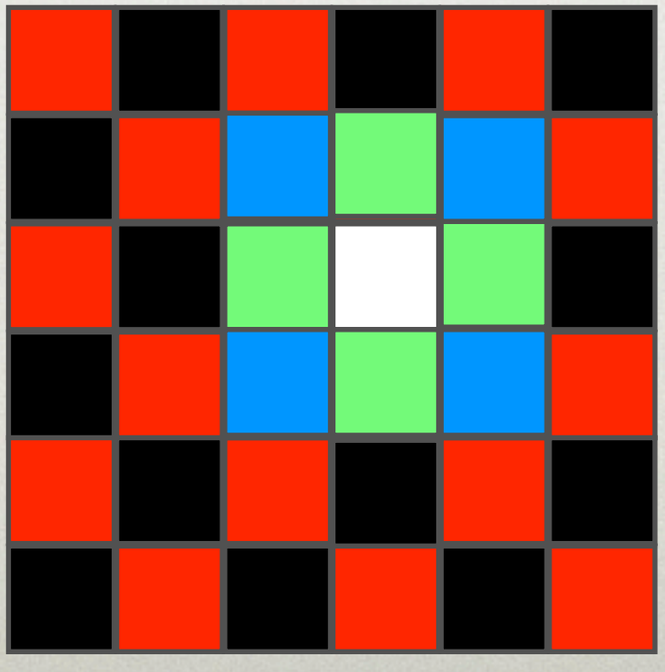
Spinor Sharing

- Stencil methods on GPUs have large gains when vector components are shared between threads
  - Load spinor components into shared memory
  - Share spinors between threads
- Can we reuse some of the spinor loads?
  - Even-odd updating complicates things
  - 24 components per site -> Limited by shared memory



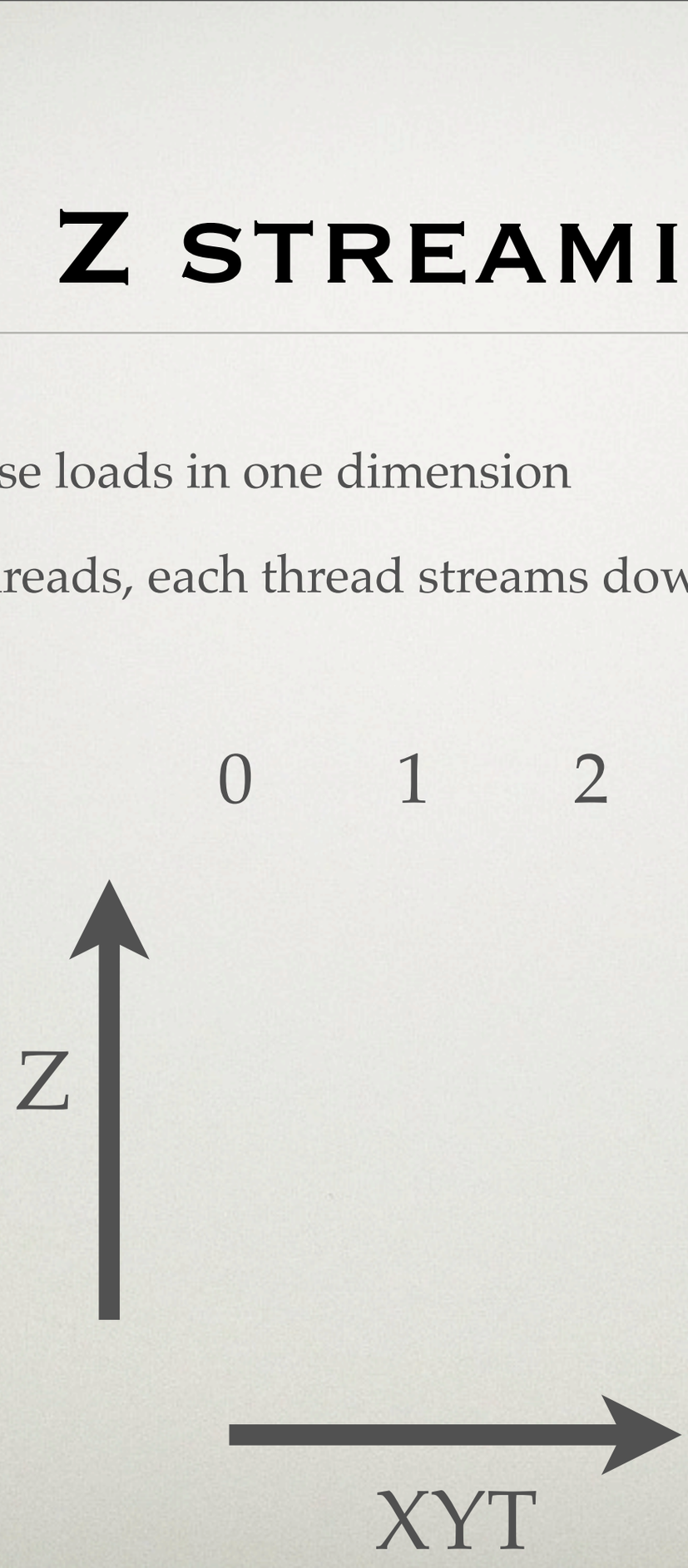
Spinor Sharing

- Stencil methods on GPUs have large gains when vector components are shared between threads
  - Load spinor components into shared memory
  - Share spinors between threads
- Can we reuse some of the spinor loads?
  - Even-odd updating complicates things
  - 24 components per site -> Limited by shared memory



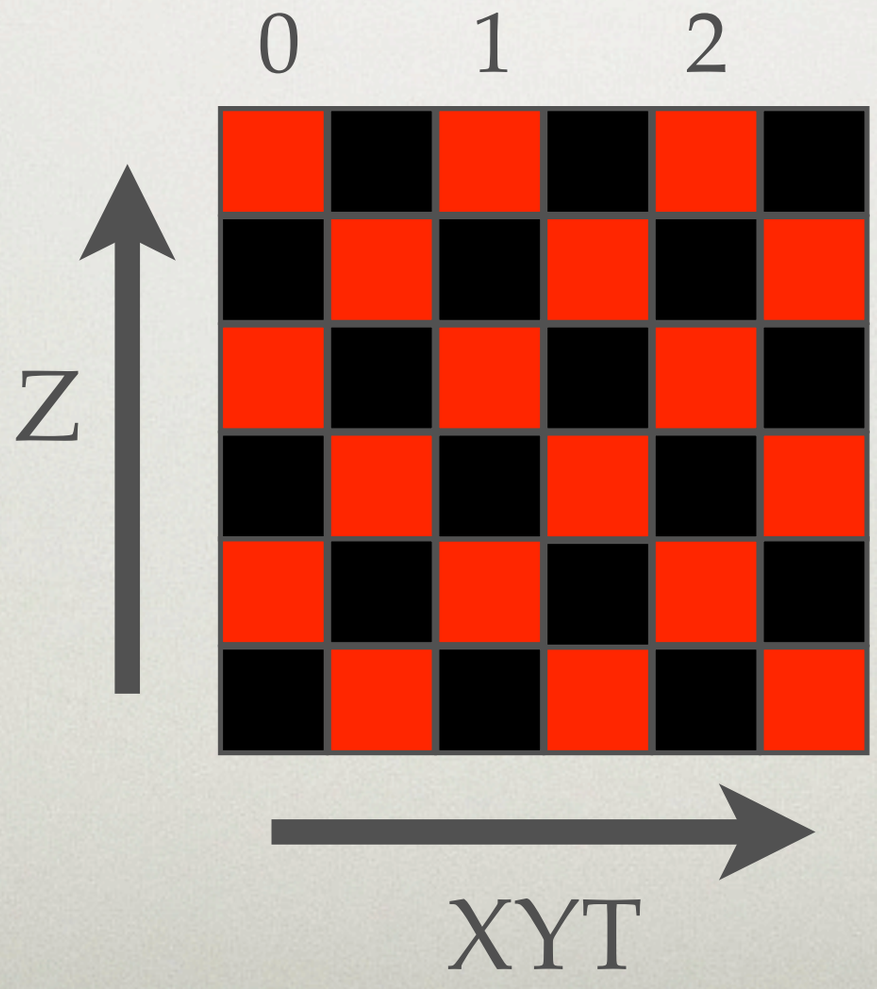
Z streaming

- Easy to reuse loads in one dimension
- Use XYT threads, each thread streams down Z dimension



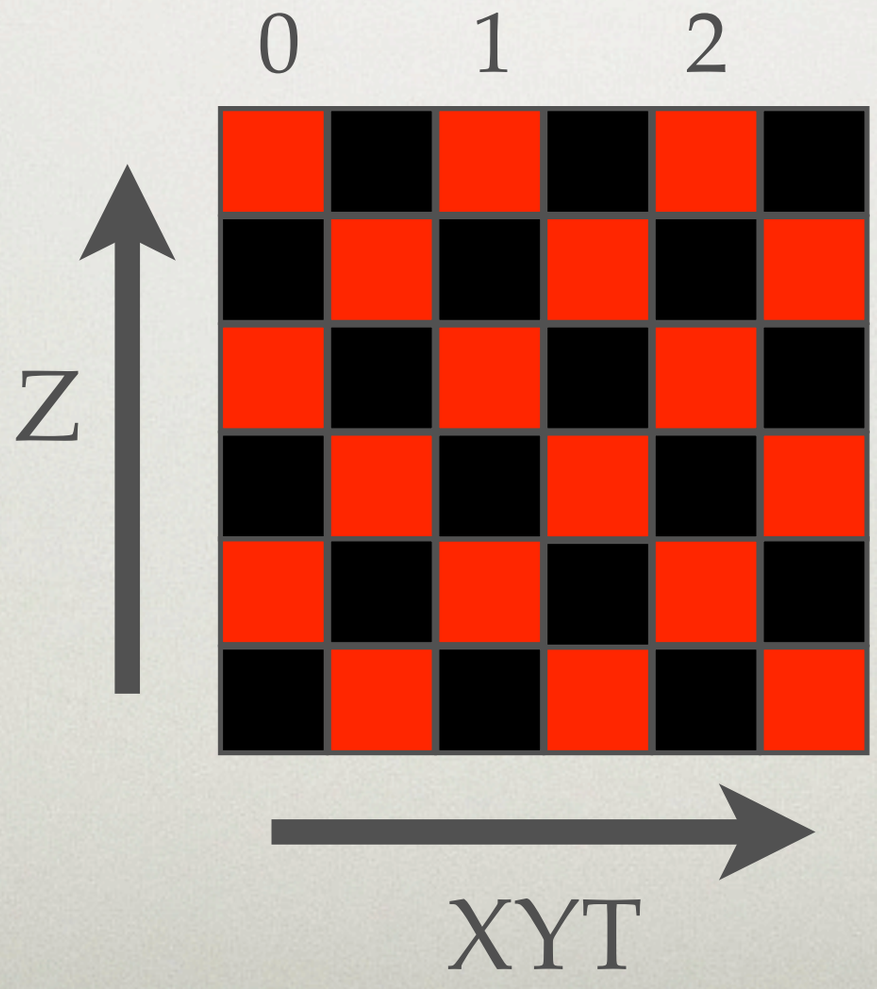
Z streaming

- Easy to reuse loads in one dimension
- Use XYT threads, each thread streams down Z dimension



Z streaming

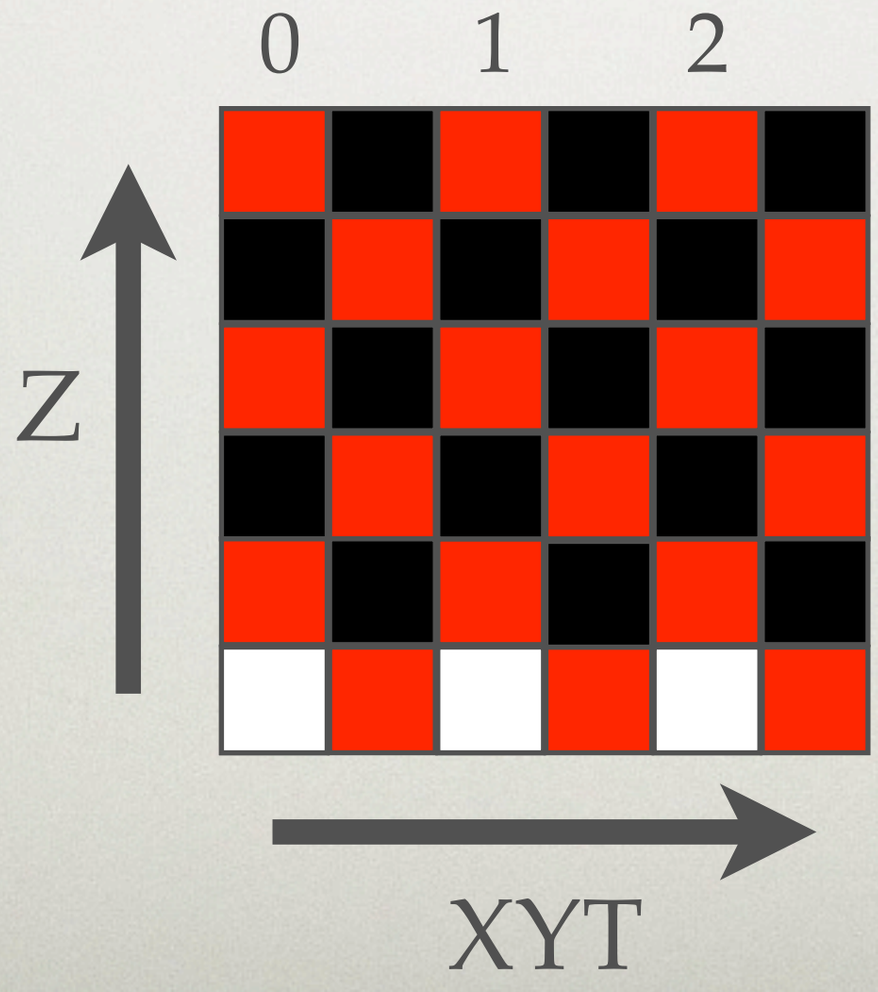
- Easy to reuse loads in one dimension
- Use XYT threads, each thread streams down Z dimension



Update z=0, load z=1

Z streaming

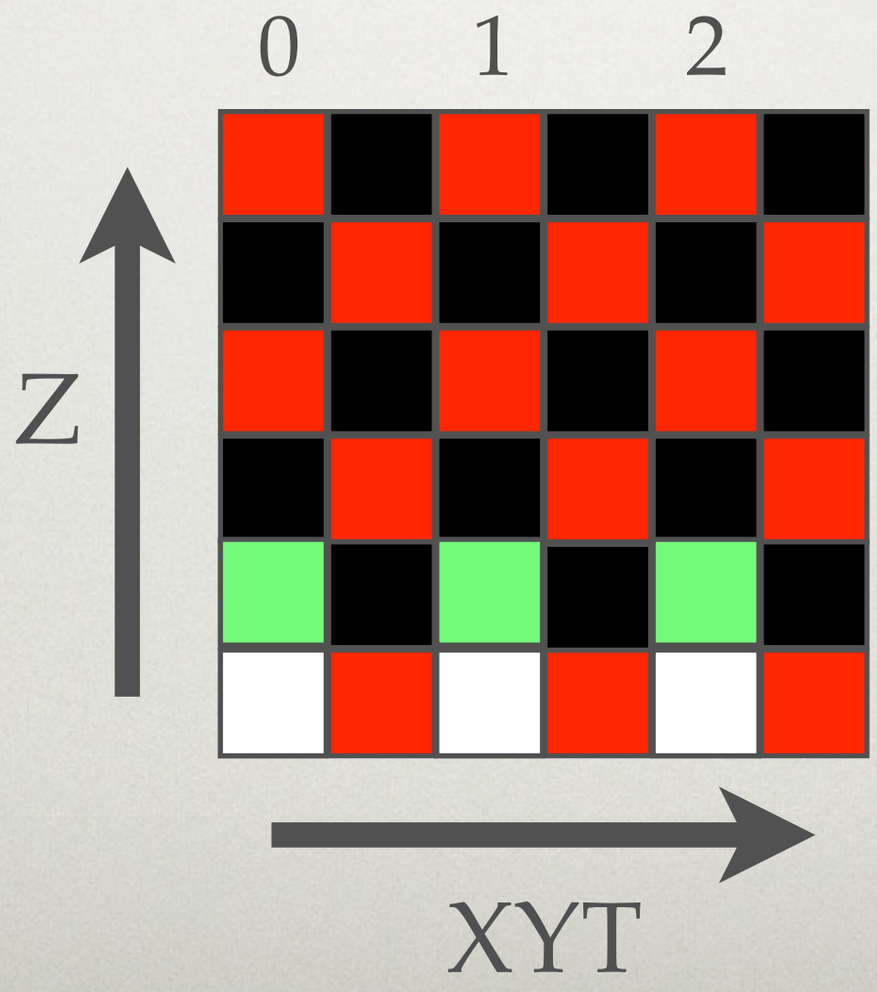
- Easy to reuse loads in one dimension
- Use XYT threads, each thread streams down Z dimension



Update z=0, load z=1

Z streaming

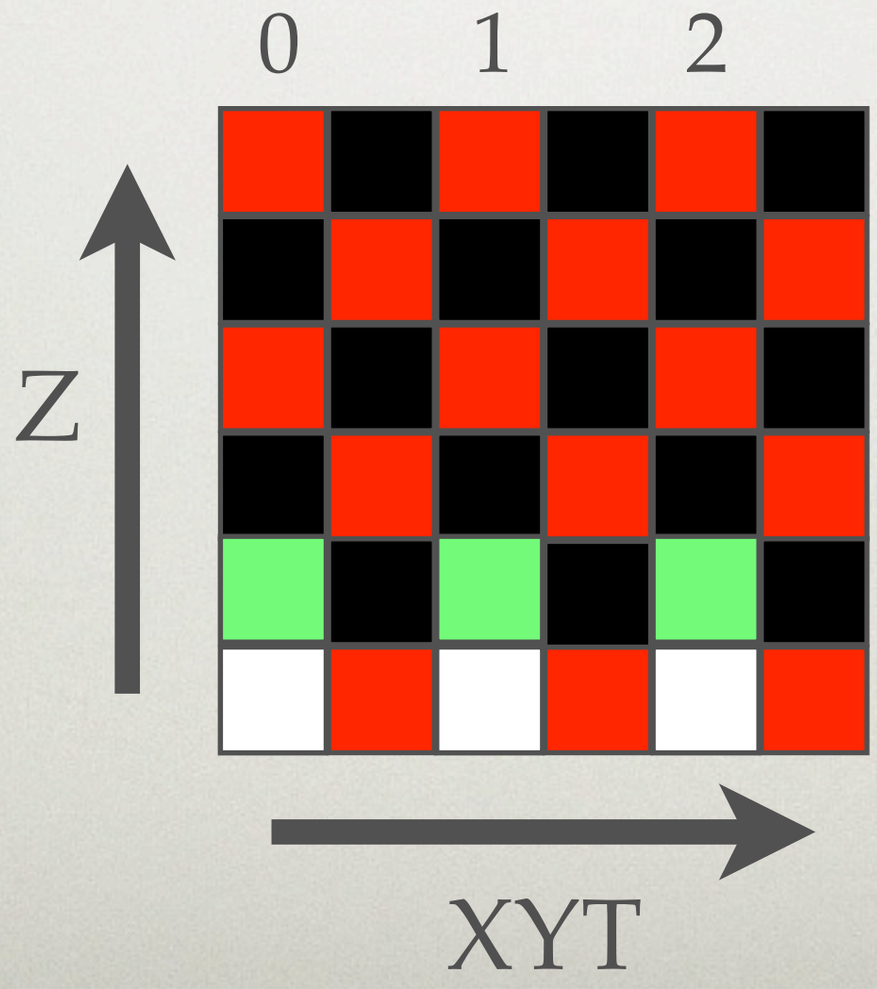
- Easy to reuse loads in one dimension
- Use XYT threads, each thread streams down Z dimension



Update z=0, load z=1

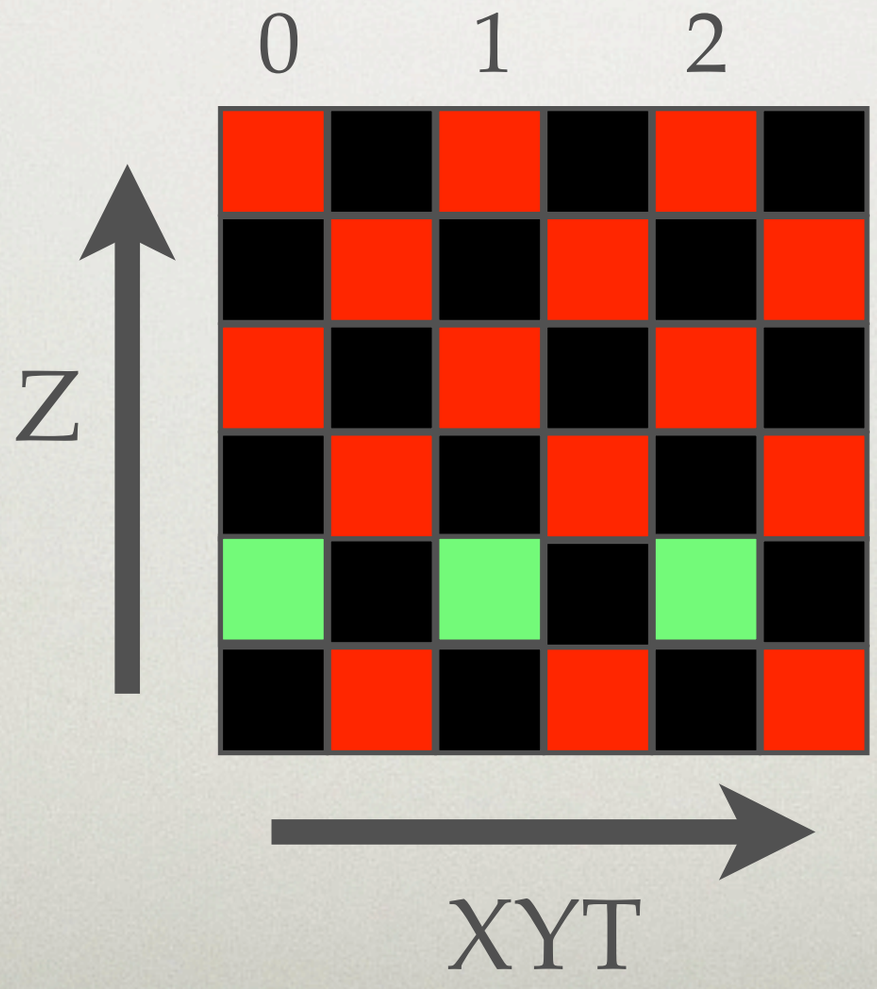
Z streaming

- Easy to reuse loads in one dimension
- Use XYT threads, each thread streams down Z dimension



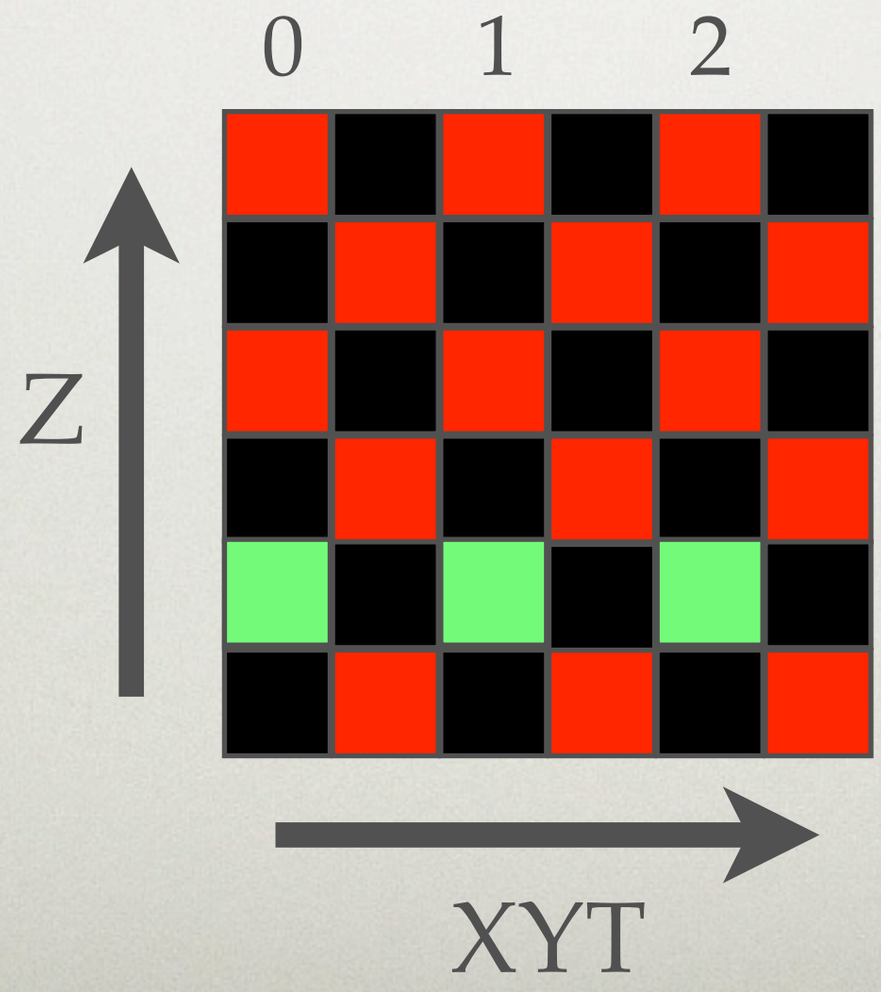
Z streaming

- Easy to reuse loads in one dimension
- Use XYT threads, each thread streams down Z dimension



Z streaming

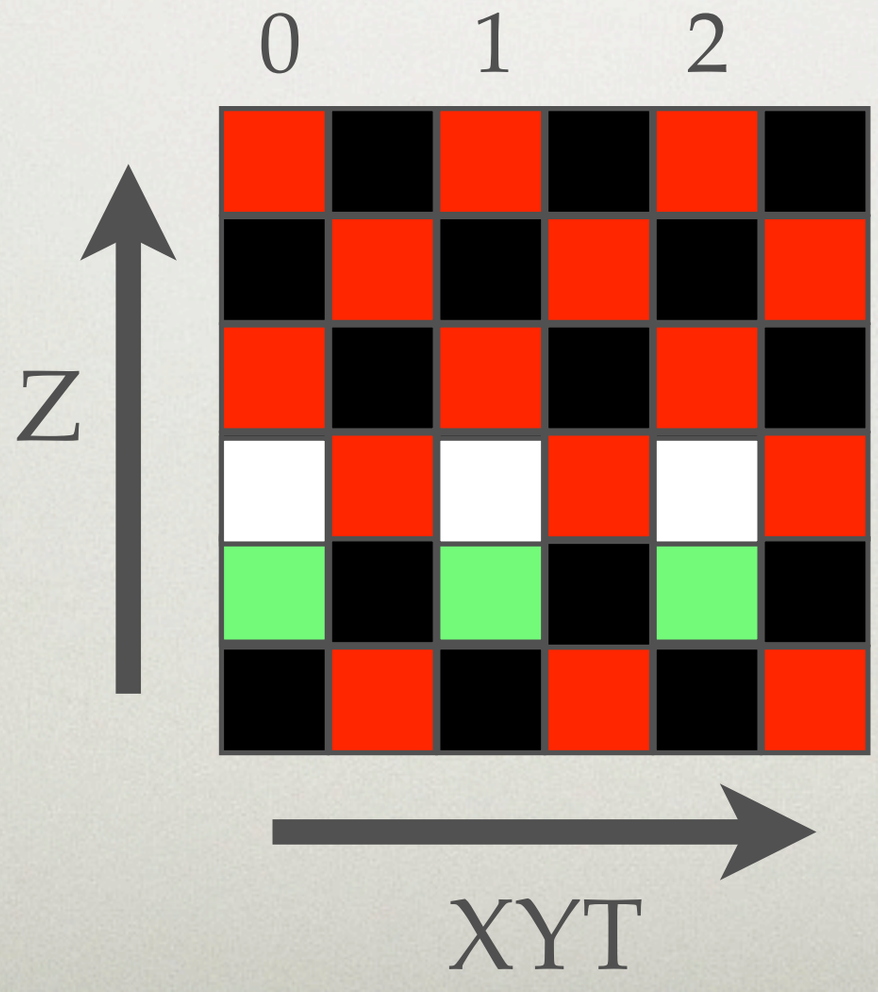
- Easy to reuse loads in one dimension
- Use XYT threads, each thread streams down Z dimension



Update z=2, reuse z=1

Z streaming

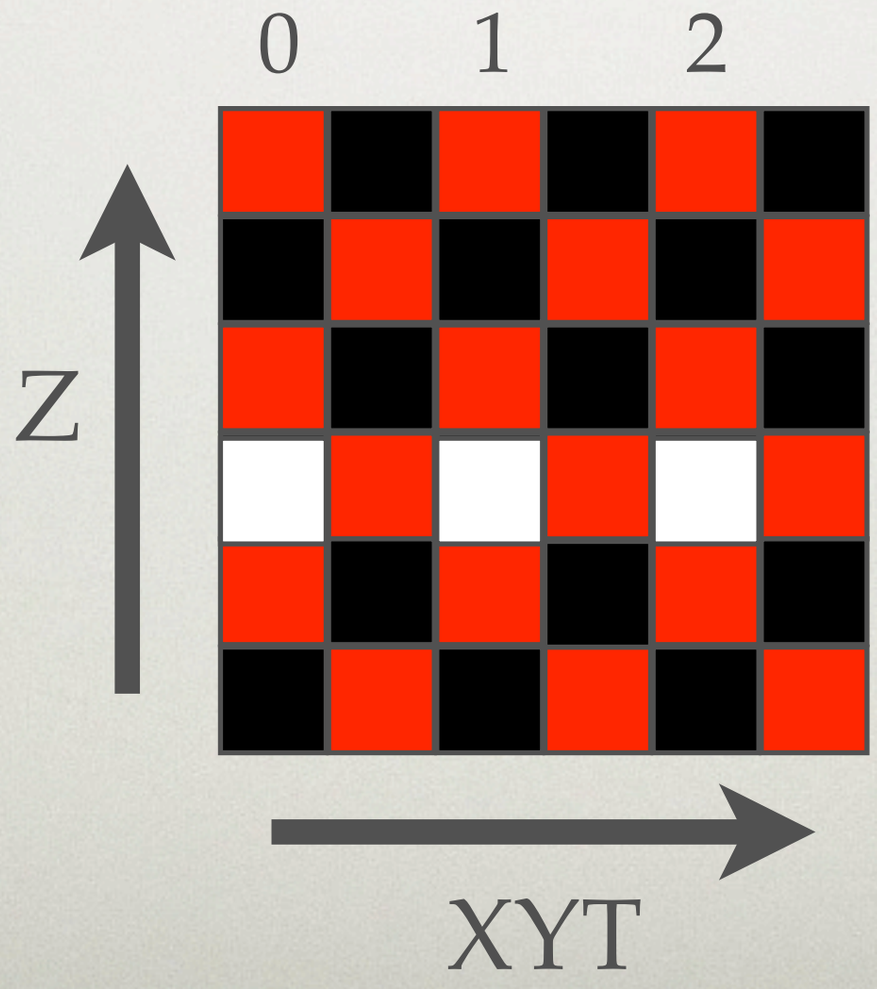
- Easy to reuse loads in one dimension
- Use XYT threads, each thread streams down Z dimension



Update z=2, reuse z=1

Z streaming

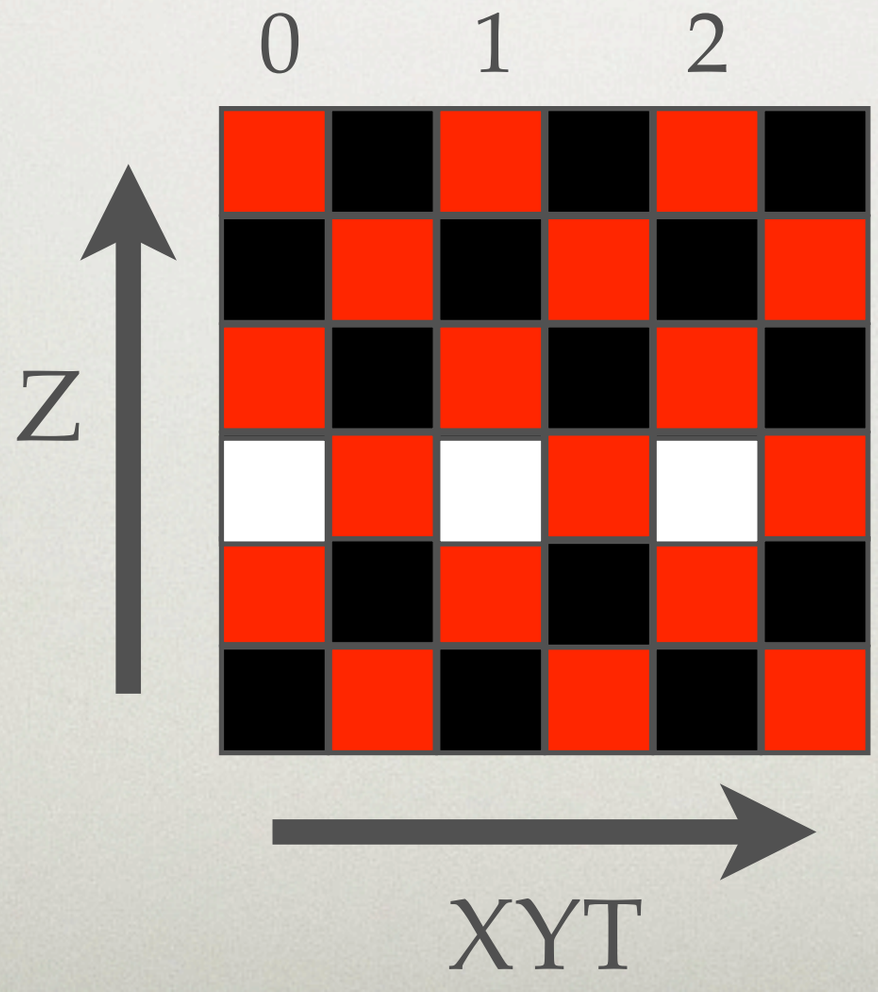
- Easy to reuse loads in one dimension
- Use XYT threads, each thread streams down Z dimension



Update z=2, reuse z=1

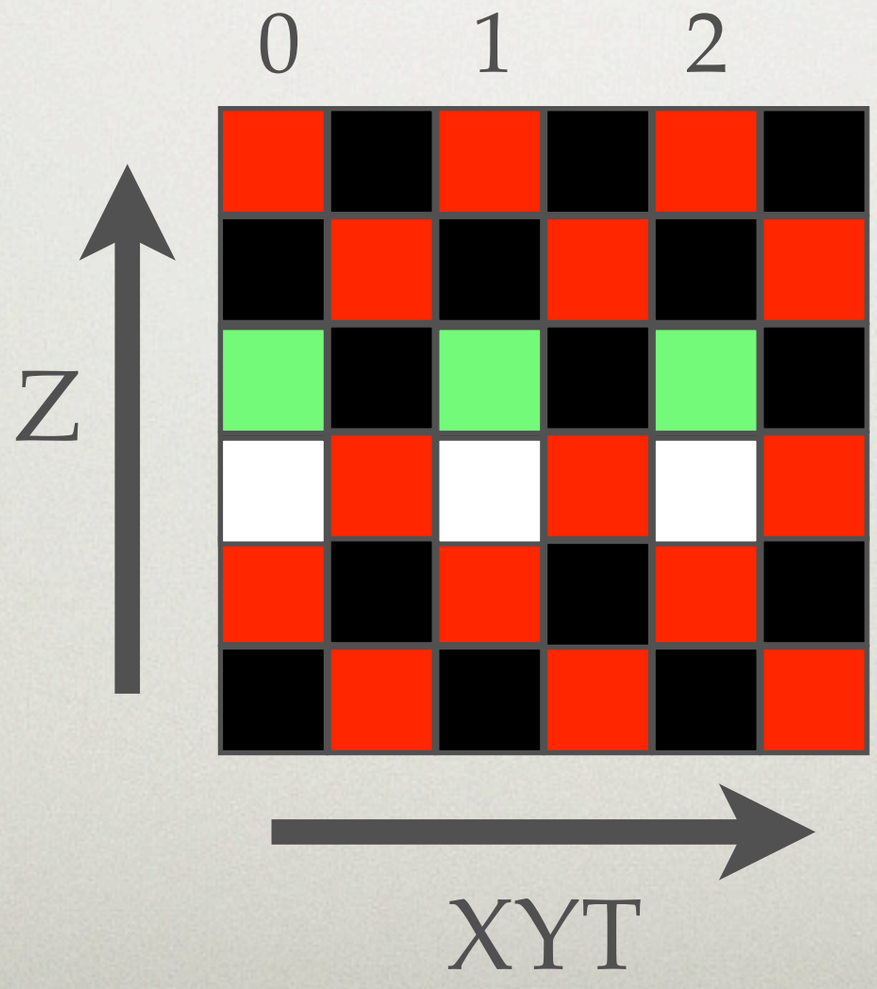
Z streaming

- Easy to reuse loads in one dimension
- Use XYT threads, each thread streams down Z dimension



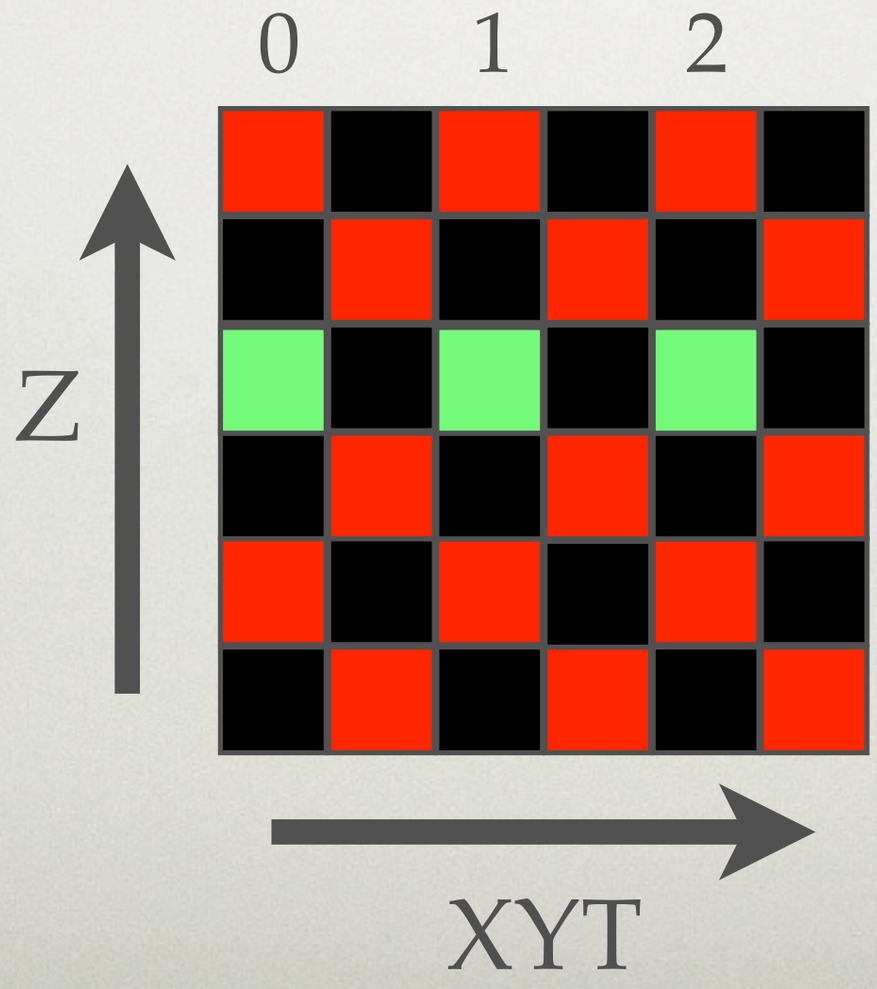
Z streaming

- Easy to reuse loads in one dimension
- Use XYT threads, each thread streams down Z dimension



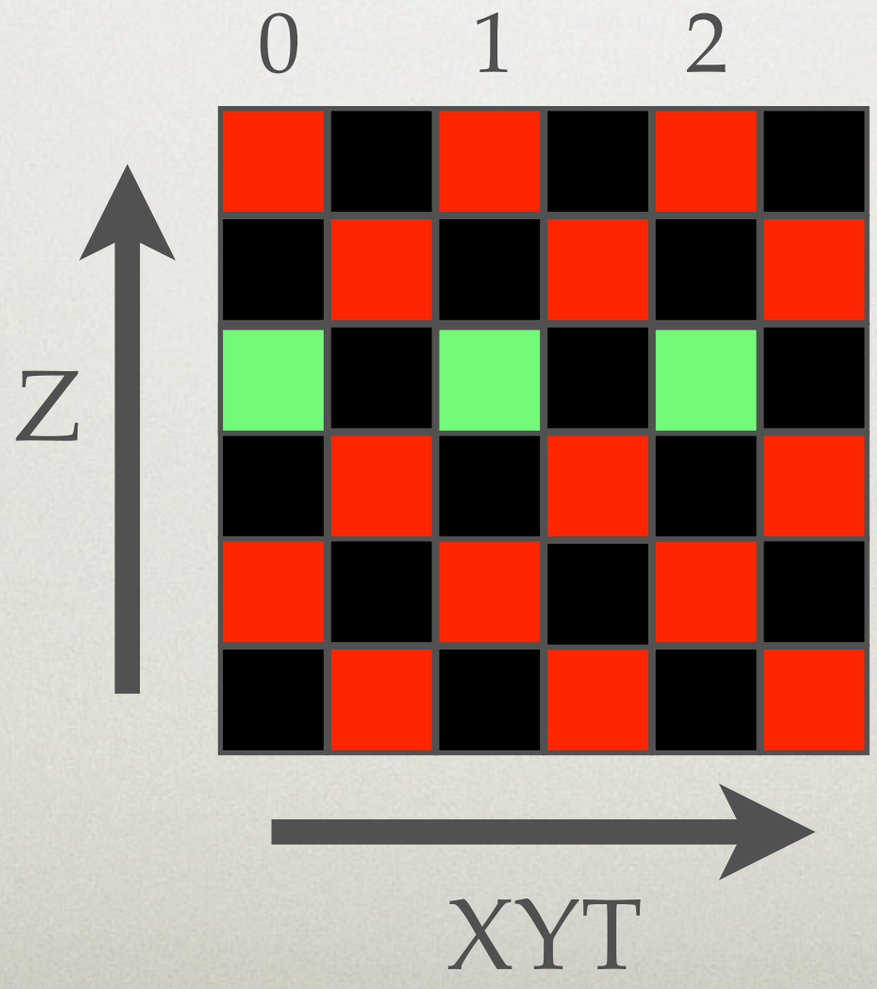
Z streaming

- Easy to reuse loads in one dimension
- Use XYT threads, each thread streams down Z dimension



Z streaming

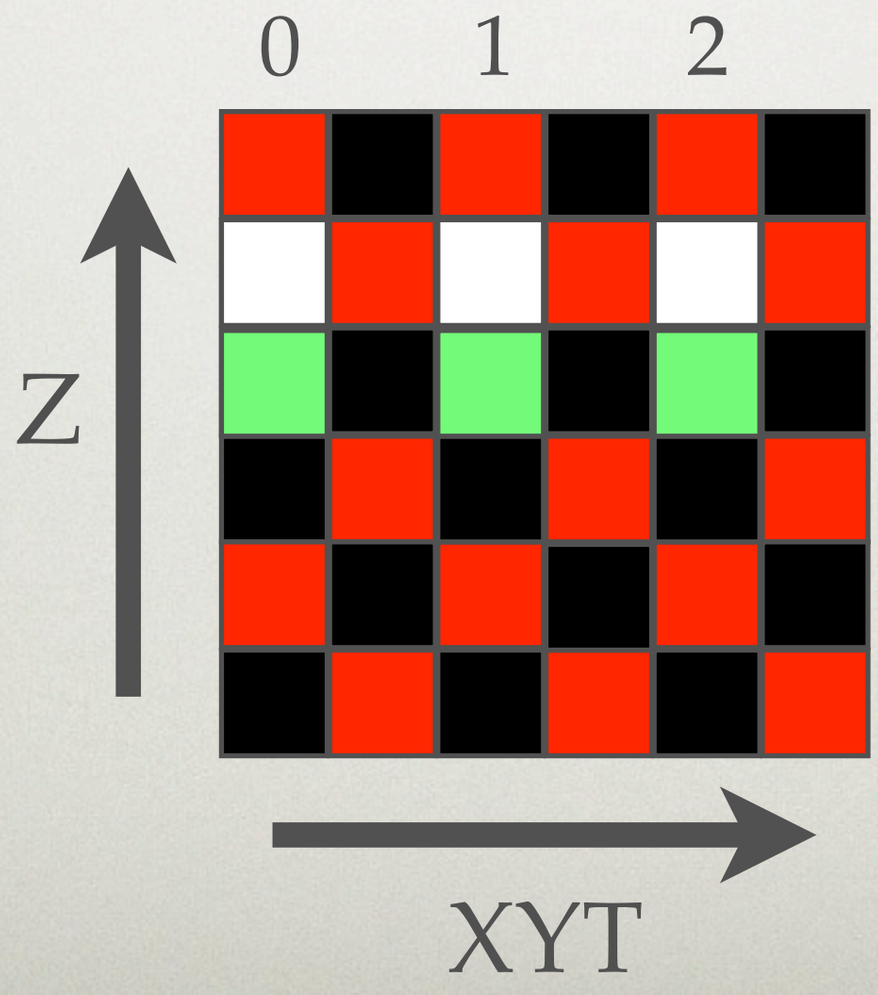
- Easy to reuse loads in one dimension
- Use XYT threads, each thread streams down Z dimension



Update z=2, load z=3

Z streaming

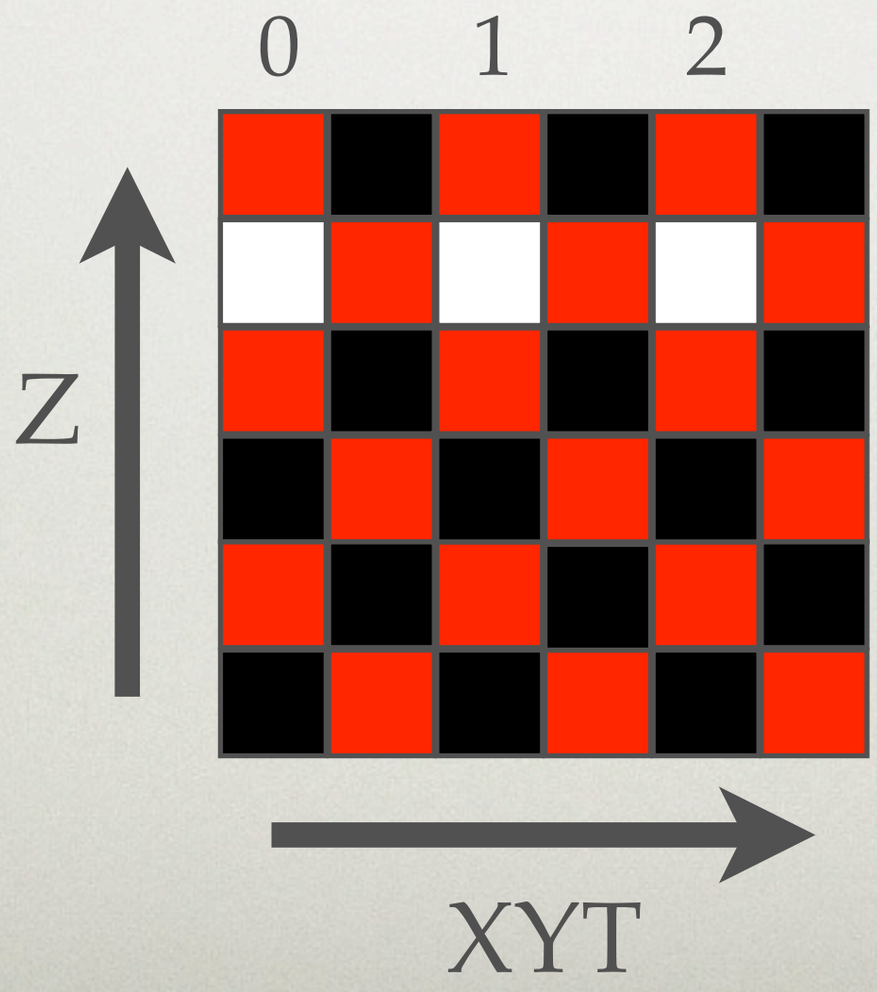
- Easy to reuse loads in one dimension
- Use XYT threads, each thread streams down Z dimension



Update z=2, load z=3

Z streaming

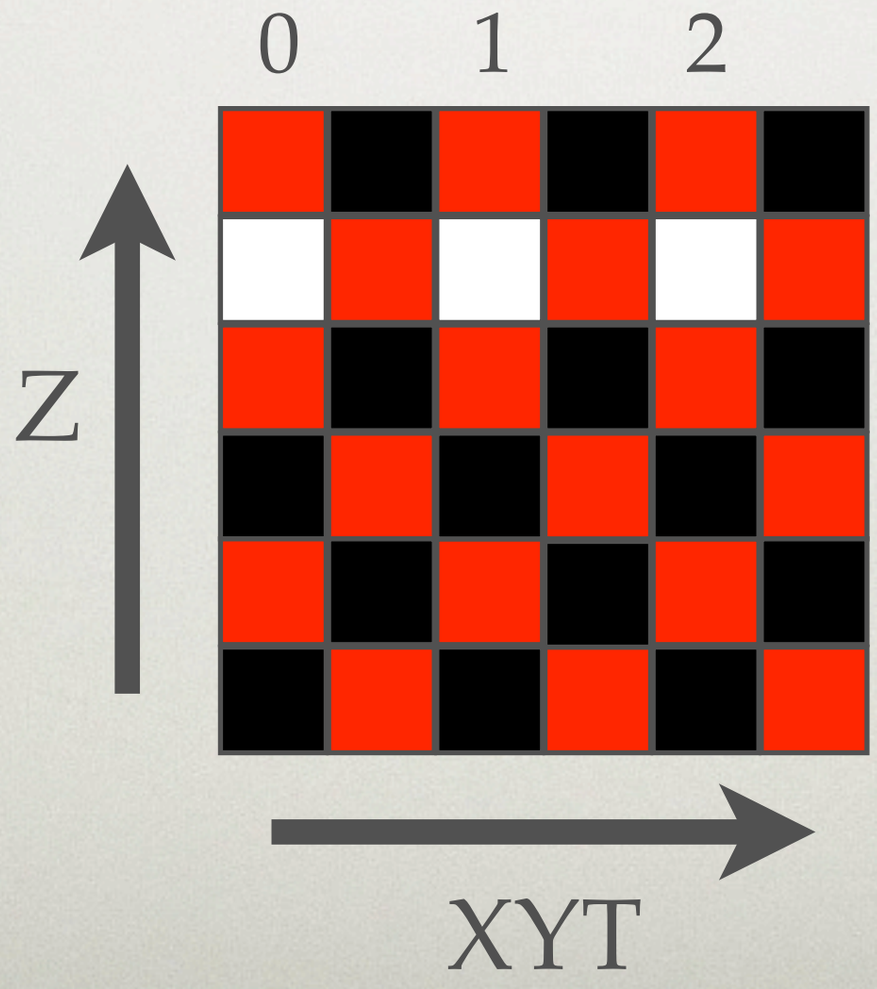
- Easy to reuse loads in one dimension
- Use XYT threads, each thread streams down Z dimension



Update z=2, load z=3

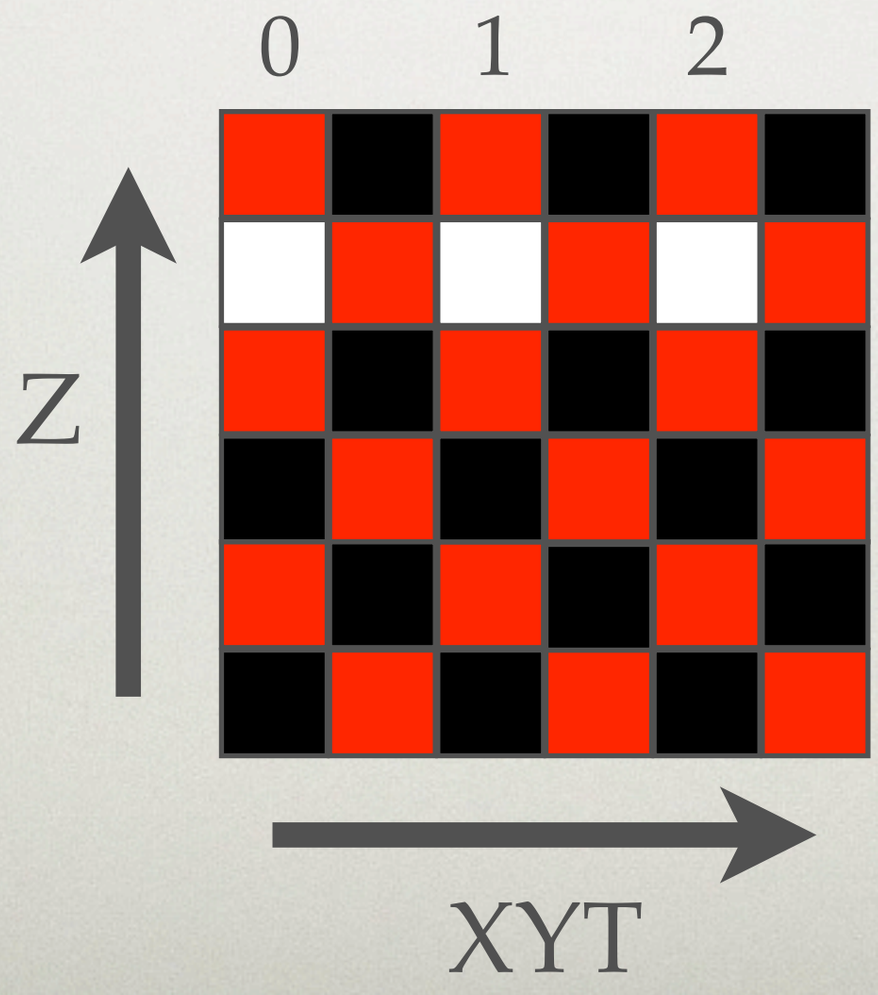
Z streaming

- Easy to reuse loads in one dimension
- Use XYT threads, each thread streams down Z dimension



Z streaming

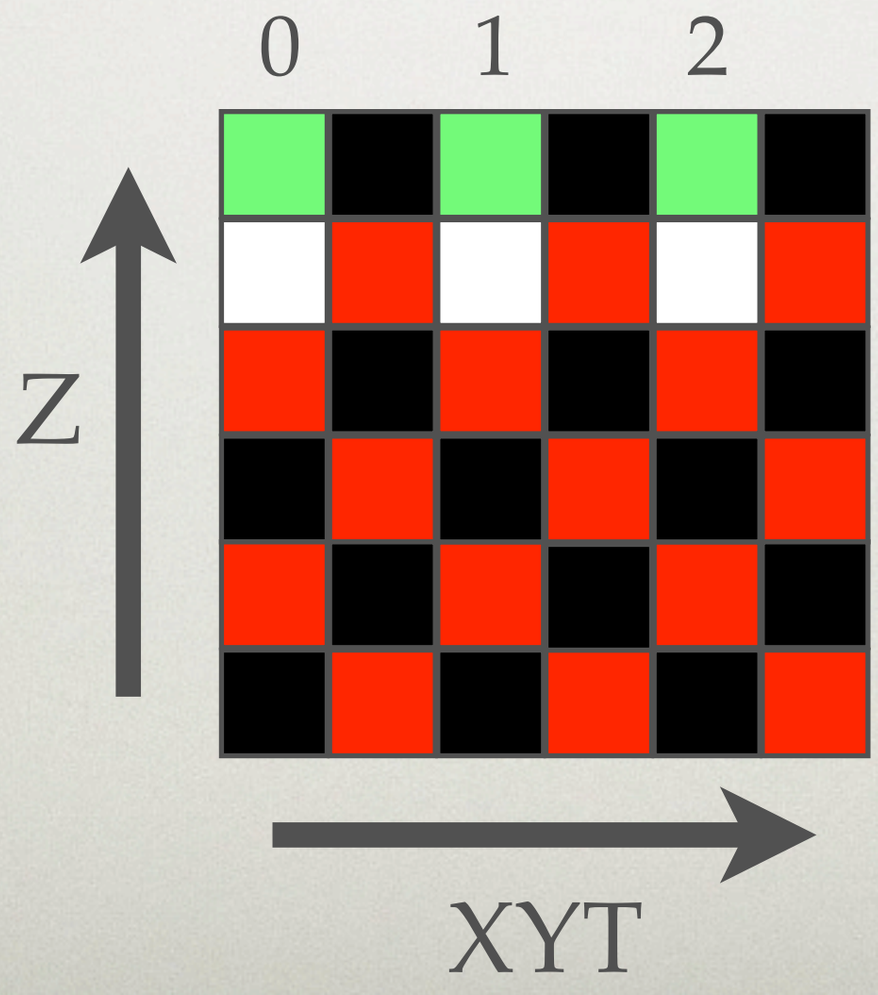
- Easy to reuse loads in one dimension
- Use XYT threads, each thread streams down Z dimension



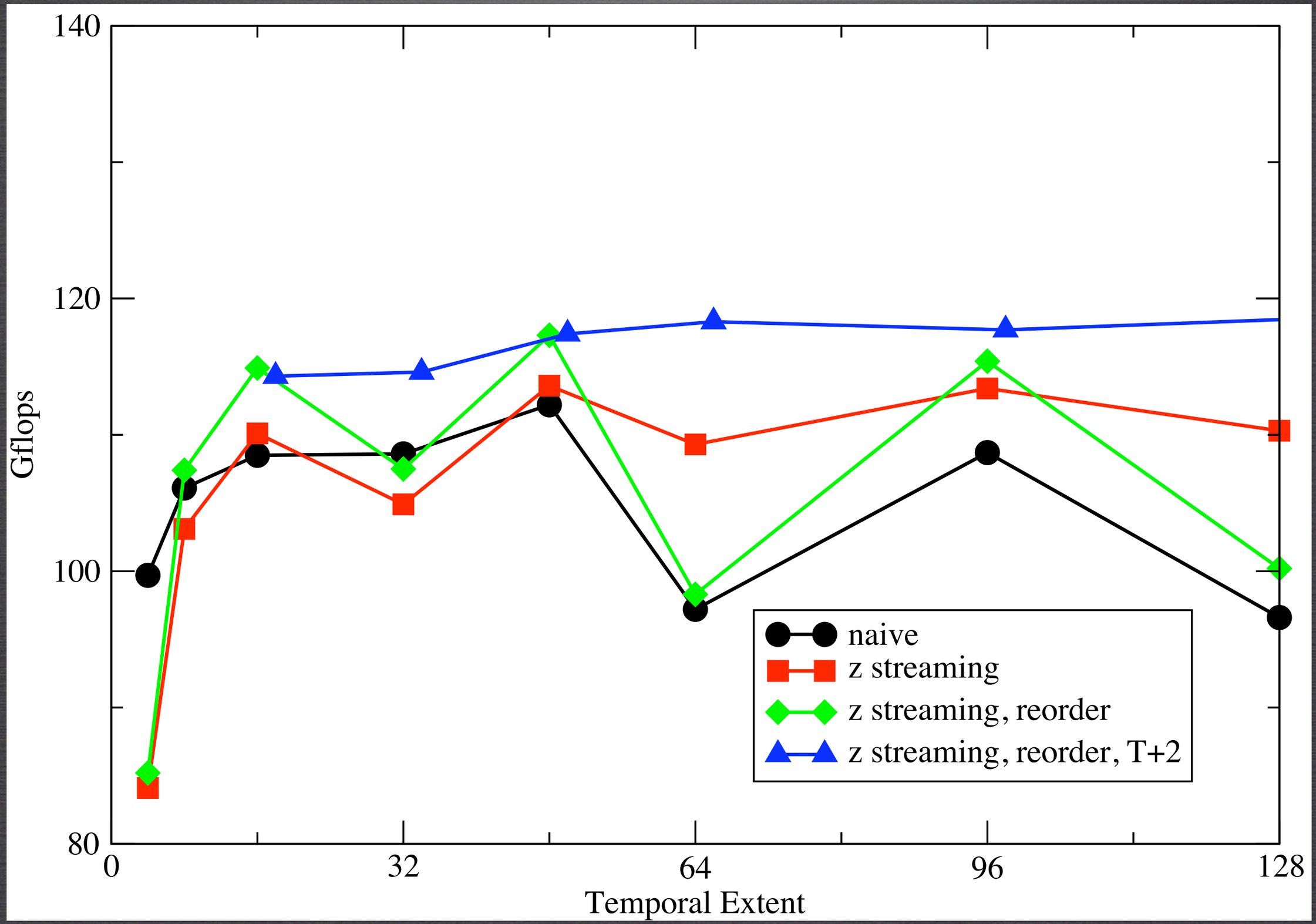
Update z=4, reuse z=3

Z streaming

- Easy to reuse loads in one dimension
- Use XYT threads, each thread streams down Z dimension

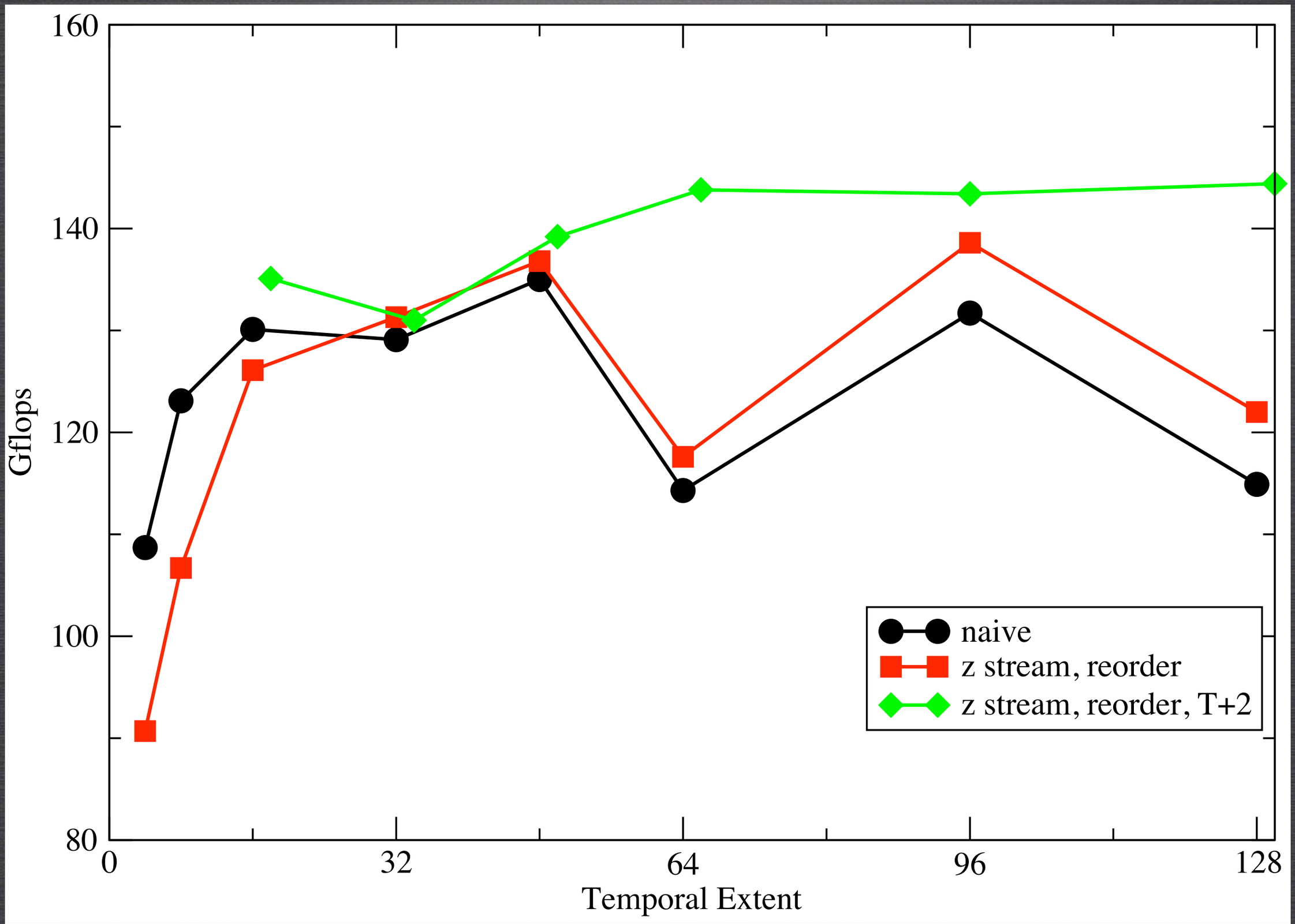


Update z=4, reuse z=3



Wilson Matrix-Vector Performance

Single Precision Z Streaming(12 Reconstruct, V=24³xT)

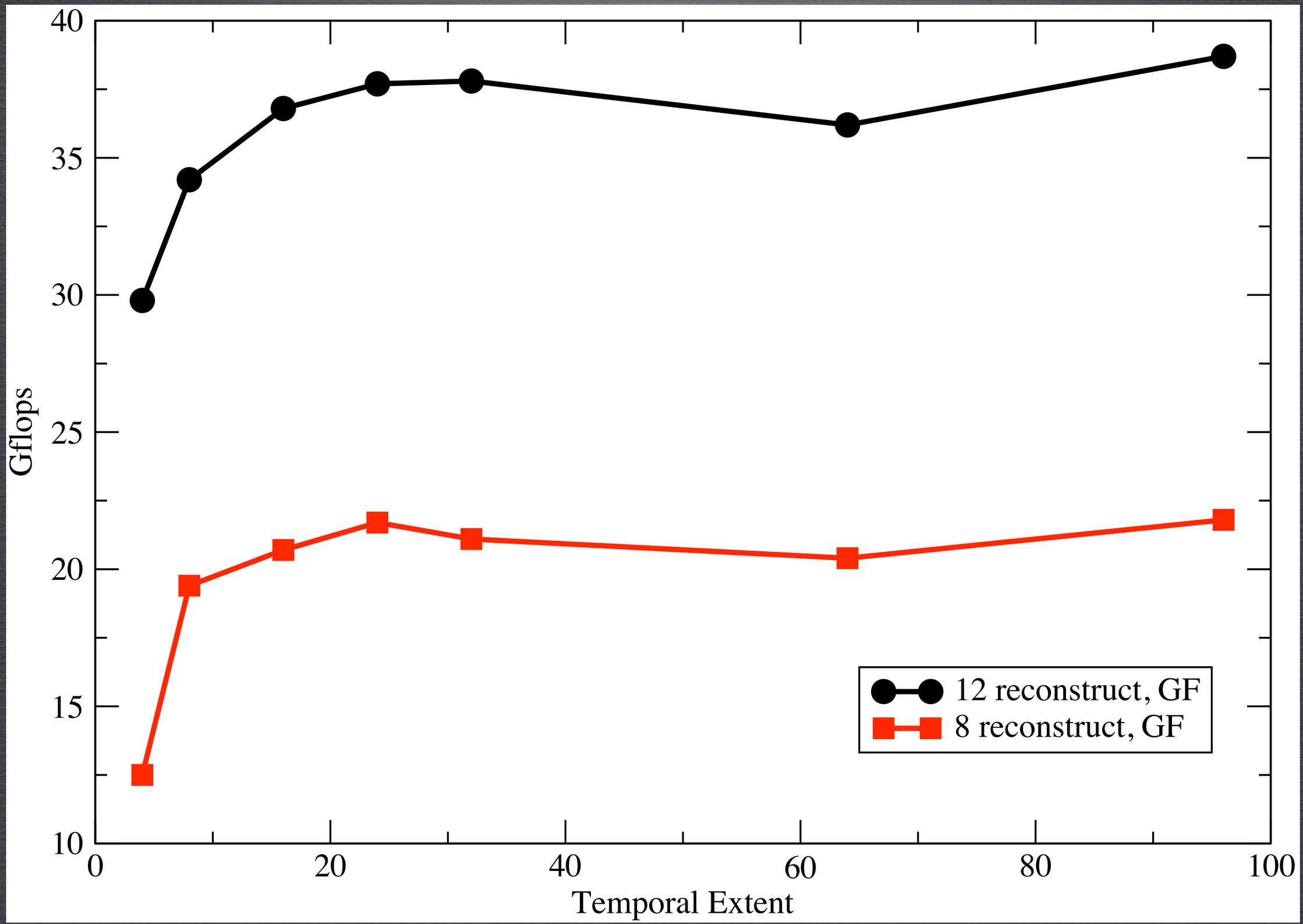


Wilson Matrix-Vector Performance

Single Precision Z Streaming (8 Reconstruct GF, V=24³xT)

Double Precision

- Double precision peak ~ 78 Gflops
  - Flop / Bandwidth ratio much more forgiving
    - Not all tricks are useful anymore...
- Half the amount of shared memory and registers
  - Simpler is better
- Performance penalty only a factor ~3 vs. single



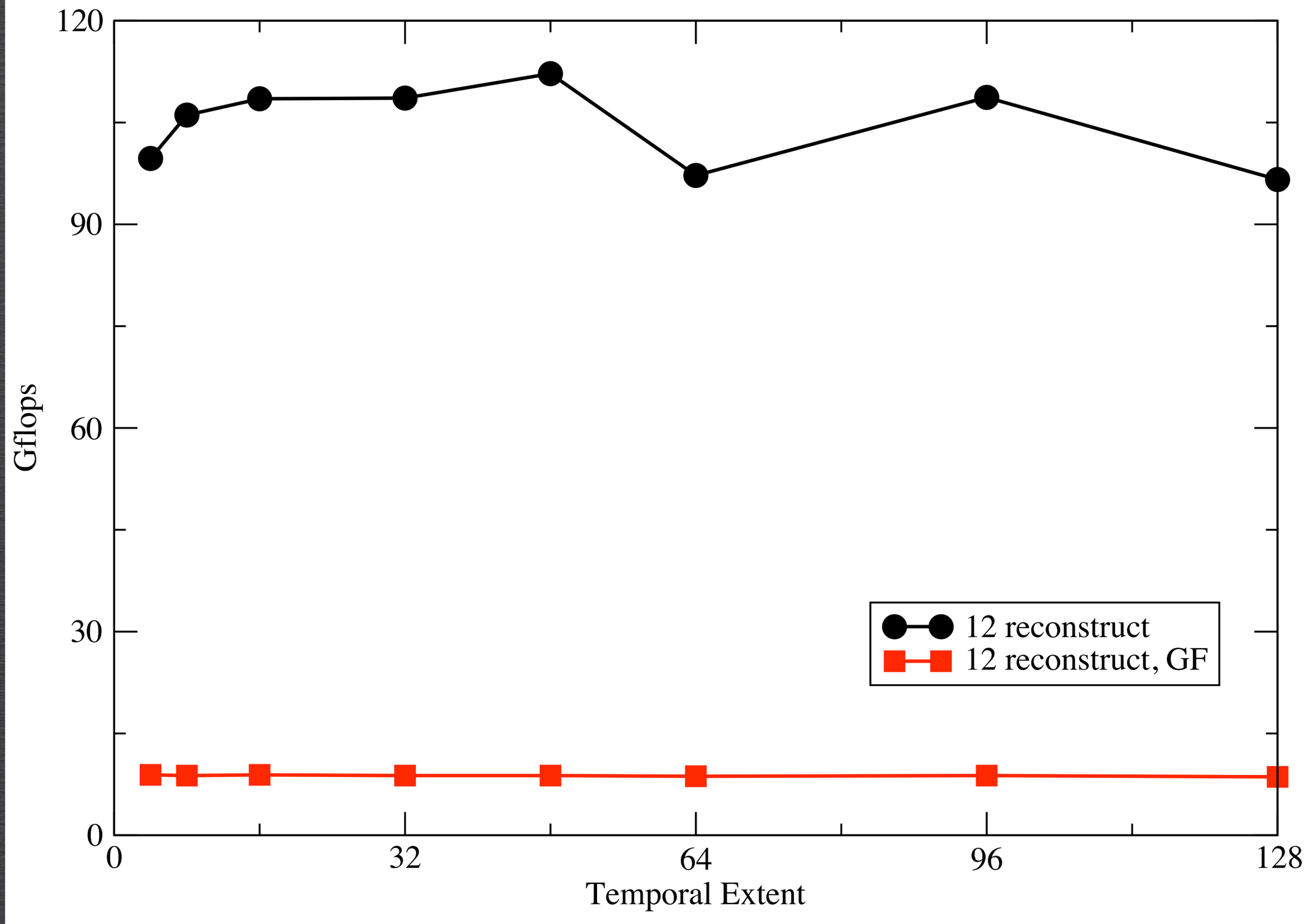
Wilson Matrix-Vector Performance

Double Precision ($V = 24^3 \times T$)

Krylov Solver Implentation

- Use GPU as an accelerator
- Replace matrix-vector in solver with GPU matrix-vector?
  - Transfer $\mathbf{p}_{k+1}$ to GPU
  - Perform computation
  - Transfer $A\mathbf{p}_{k+1}$ back to CPU
- Problem
  - Transfers and matrix-vector are O(N) operations

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

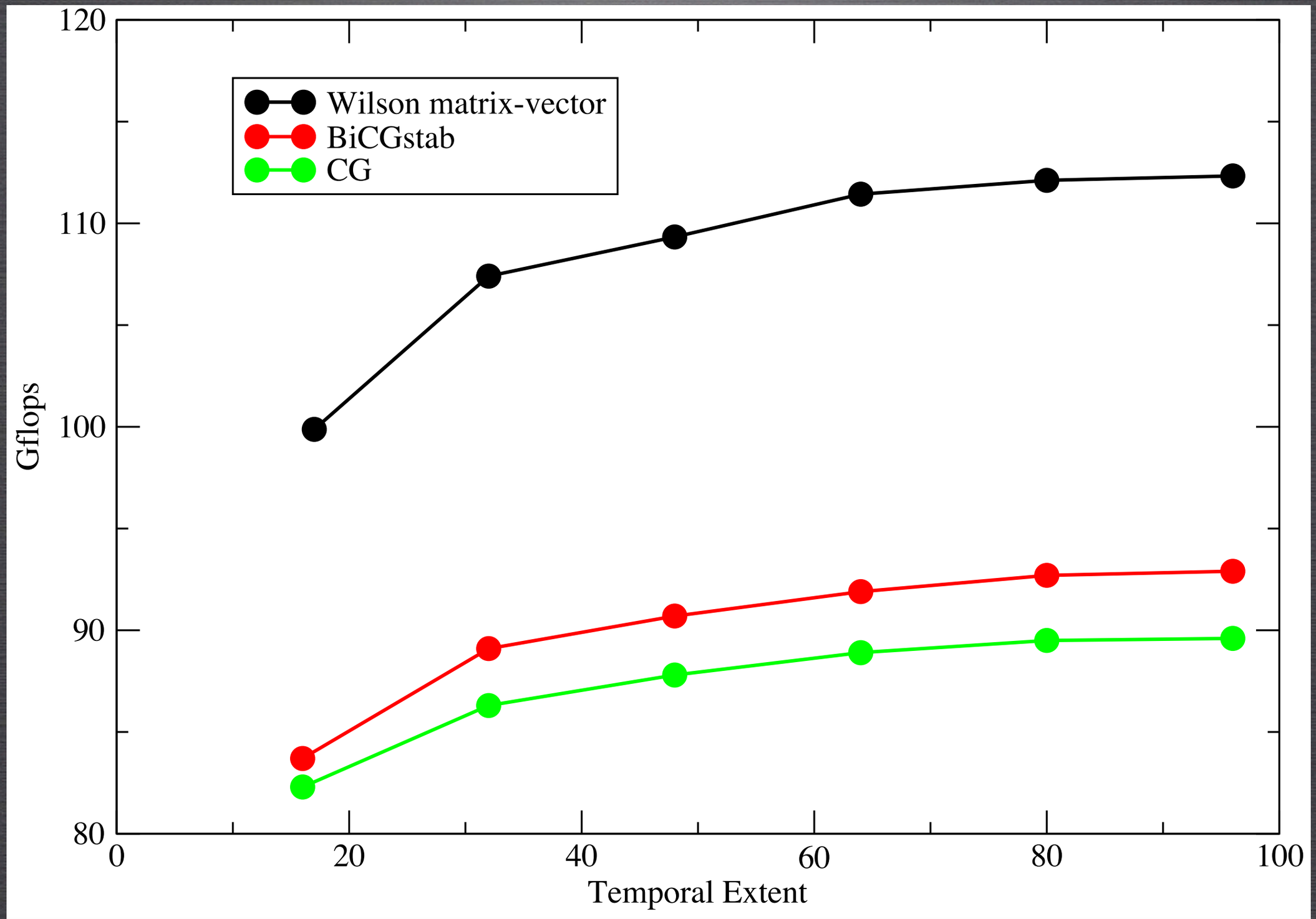


Wilson Matrix-Vector Performance

Single Precision Accelerator (V = 24^3xT)

Krylov Solver Implementation

- Complete solver must be on GPU
  - Transfer **b** to GPU
  - Solve A**x**=**b**
  - Transfer **x** to CPU
- Besides matrix-vector, require BLAS level 1 type operations
  - AXPY operations: **b** += a**x**
  - NORM operations: c = (**b**,**b**)
- CUBLAS library available
- Better to coalesce operations to minimize bandwidth
  - e.g., AXPY_NORM



Wilson Inverter Performance

Single Precision (12 reconstruct, V=24³xT)

Multi-Precision Solvers

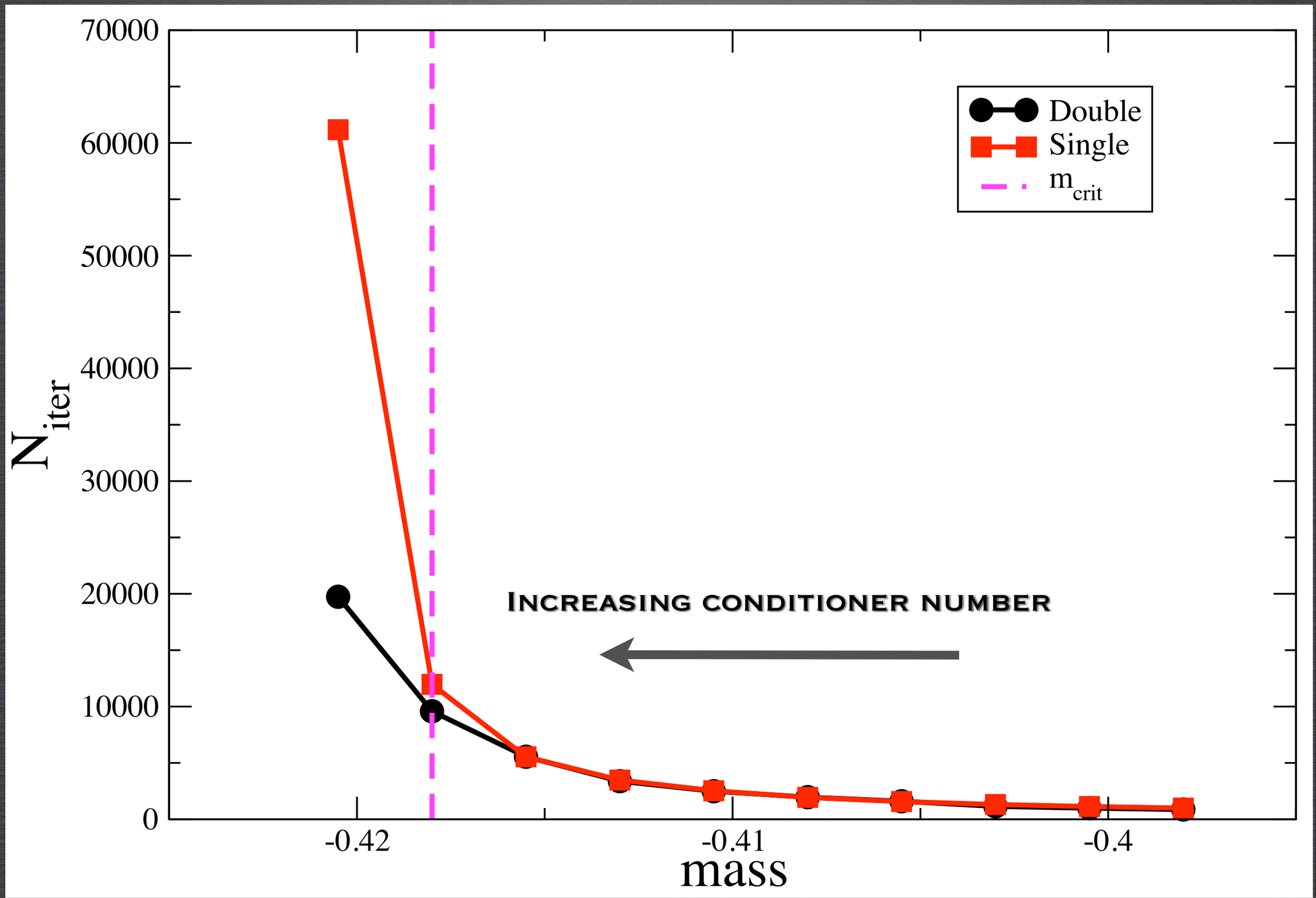
- Require solver tolerance beyond limit of single precision
- Use defect-correction

Double precision mat-vec and accumulate

```
while ( | r_k | > ε) {
    r_k = b - A x_k
    p_k = A^-1 r_k
    x_k+1 = x_k + p_k
}
```

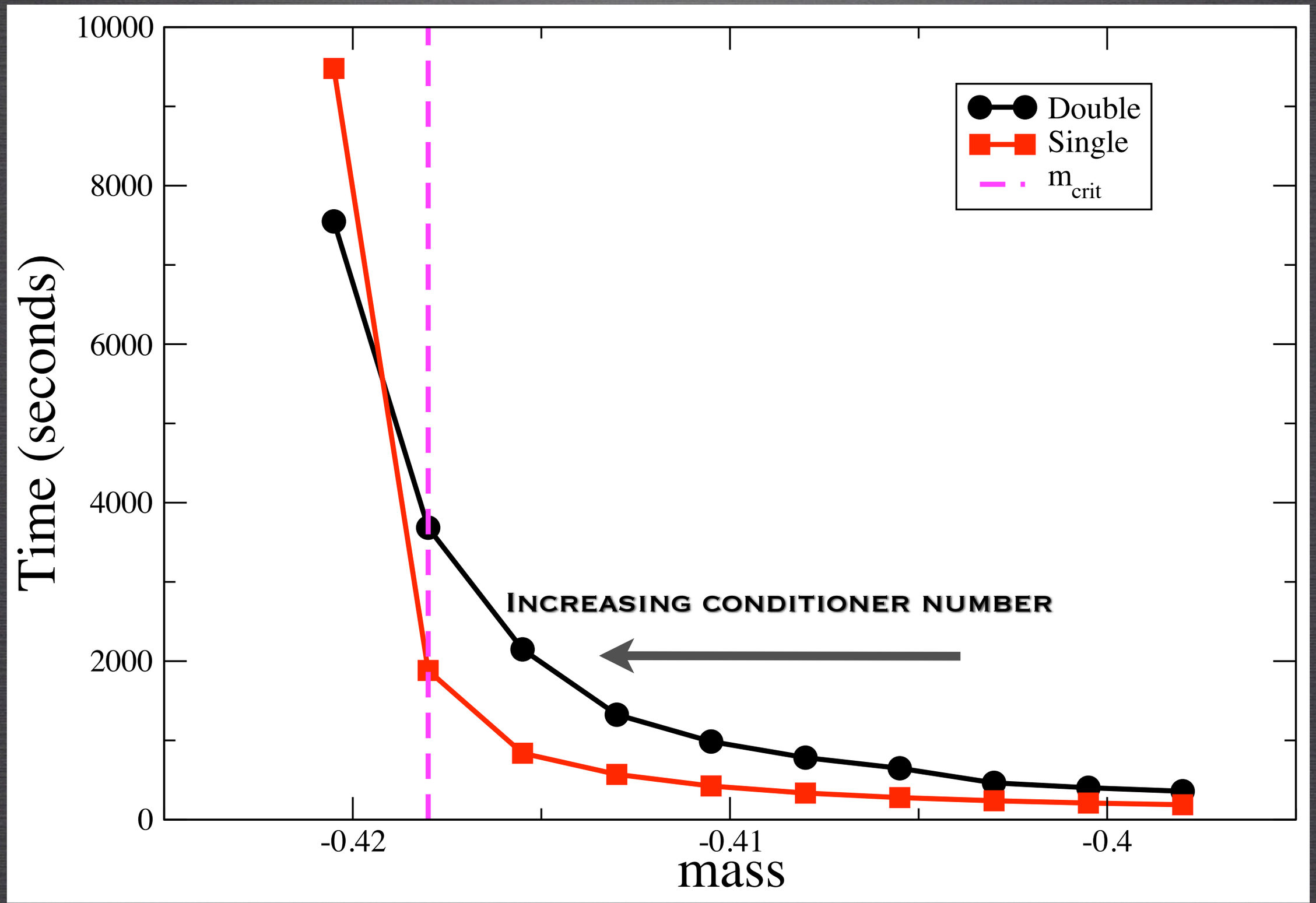
Inner single precision solve

- Double precision done can be done on CPU or GPU
- Can always check GPU gets correct answer
- Disadvantage is each new single precision solve is a restart



Wilson Inverter Iterations

($\varepsilon=10^{-8}$, $V=32^3x96$)

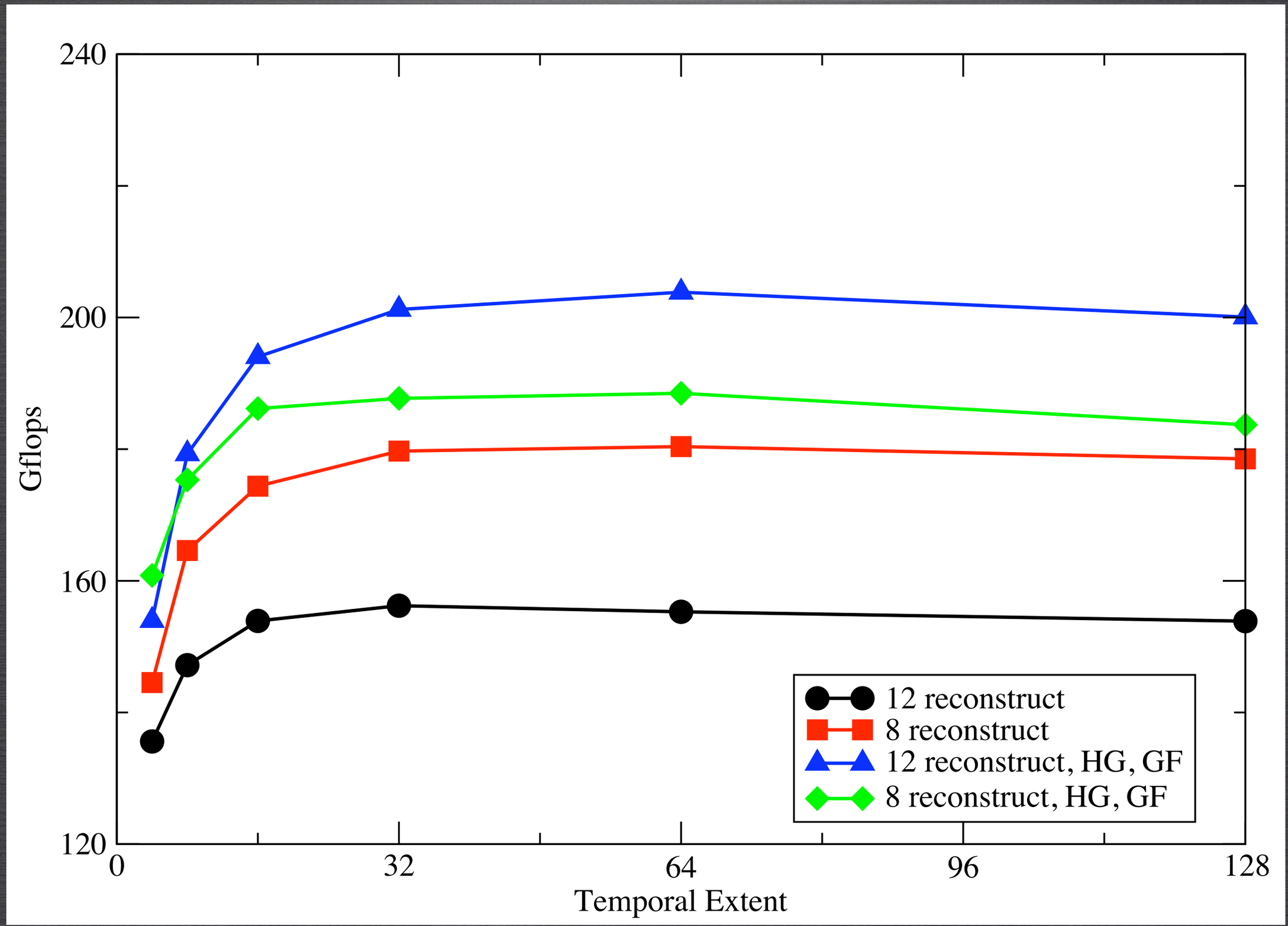


Wilson Inverter Time to Solution

($\varepsilon=10^{-8}$, $V=32^3x96$)

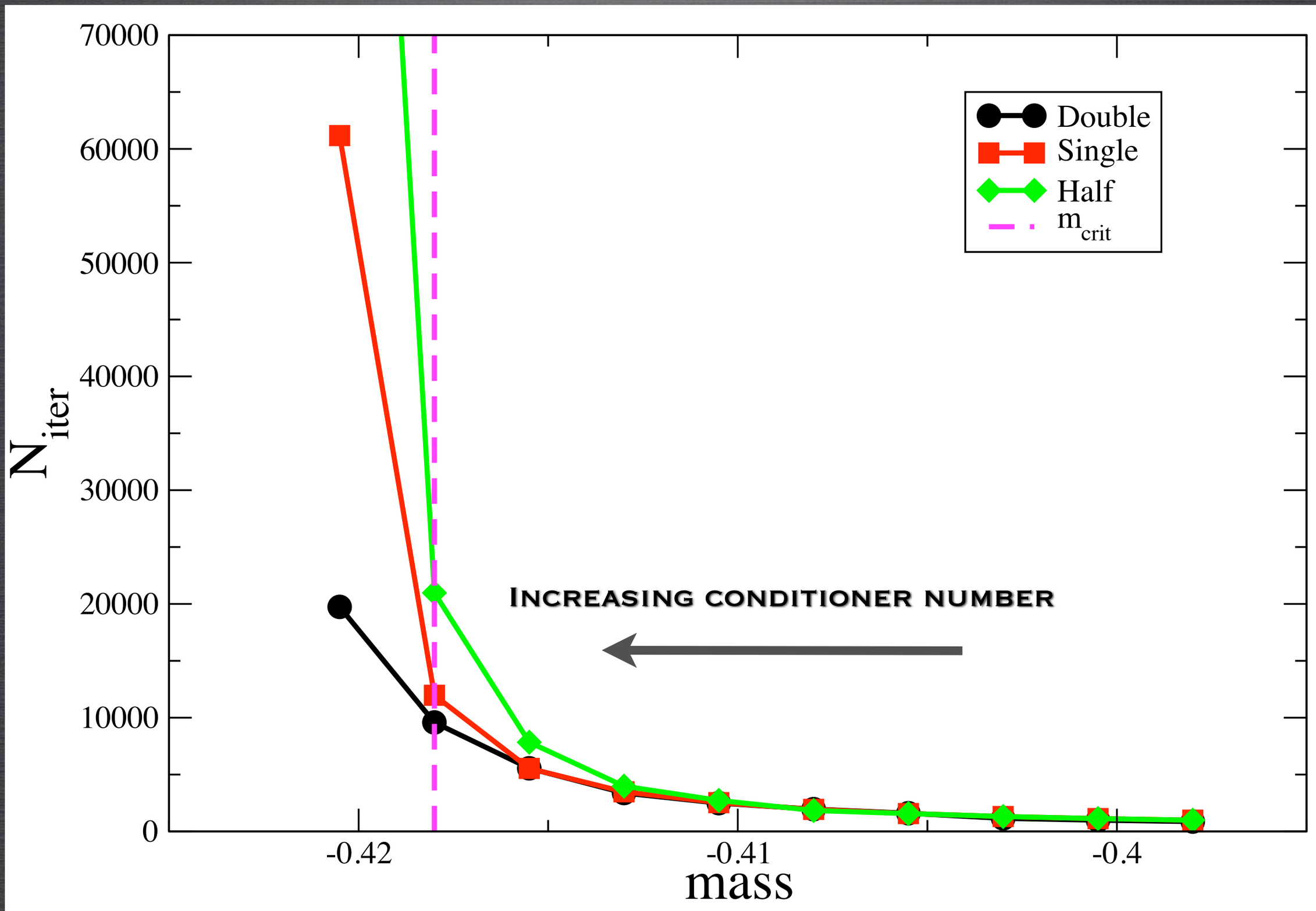
Half Precision

- Single-precision can be used to find double-precision result
- GPU kernel is still bandwidth bound
  - Use half precision for inner solve?
- ~~No support for FP16 in CUDA~~ (new in CUDA 2.3)
- Does support texture reading 16-bit integer -> float [-1,1]
  - Internal calculation done in 32-bit
- Natural format for SU(3) fields
- Vector field has no exponent restrictions
- Dirac operator off-diagonals are O(1) mixing of site-wise components
  - Scale all components at a site by maximum element
  - 24 floats per site -> 24 half-integers + 1 float



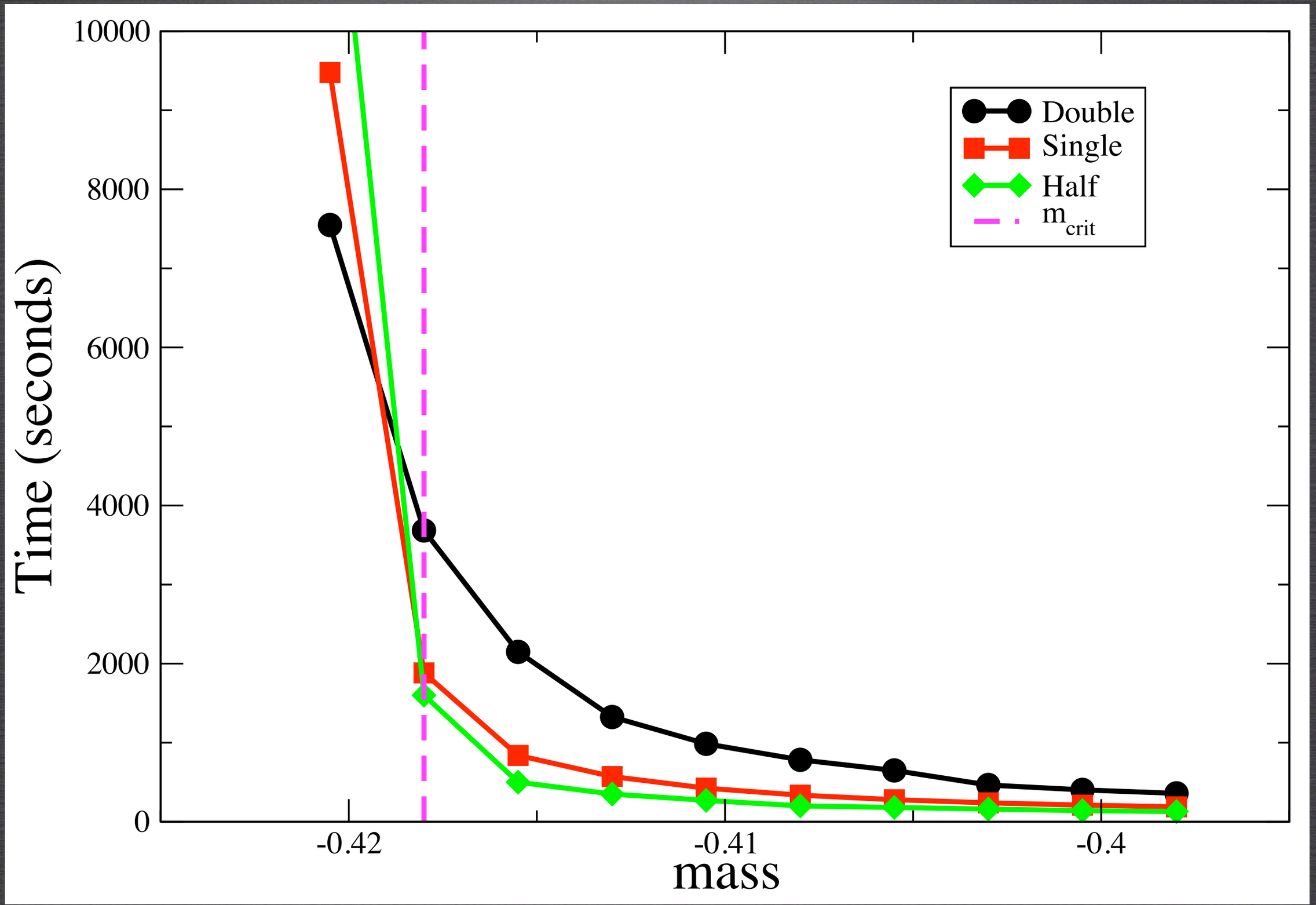
Wilson Matrix-Vector Performance

Half Precision



Wilson Inverter Iterations

($\varepsilon=10^{-8}$, $V=32^3x96$)

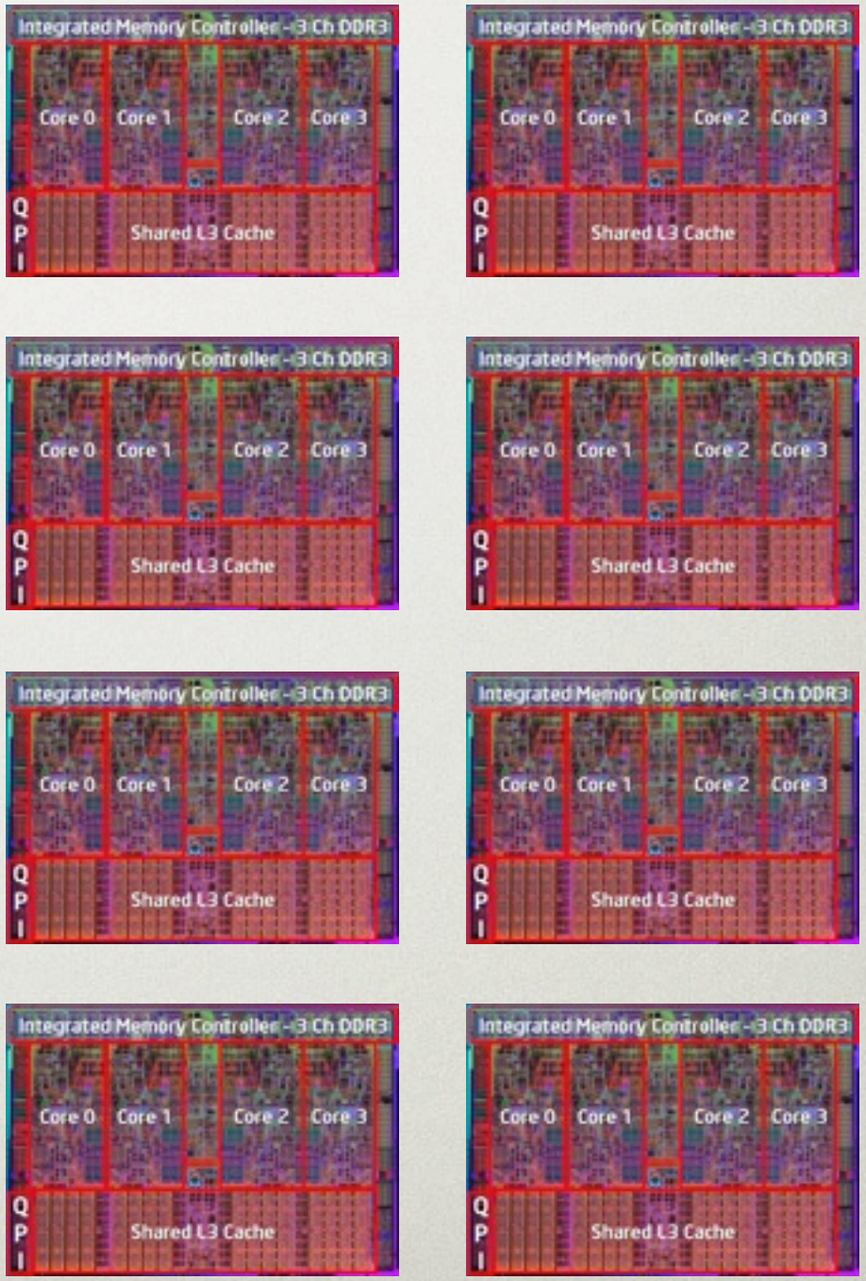


Wilson Inverter Time to Solution

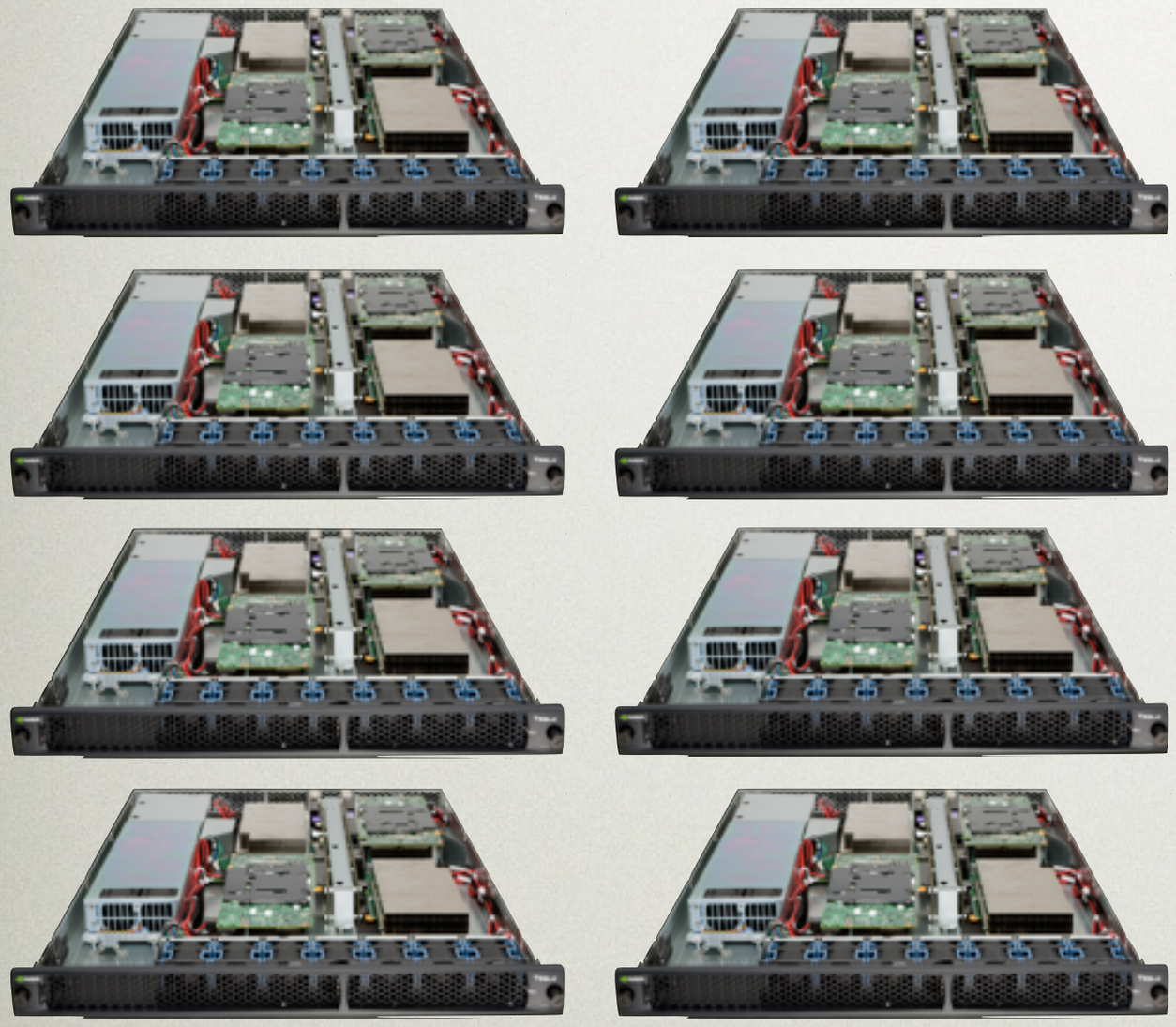
($\varepsilon=10^{-8}$, $V=32^3x96$)

Performance Per MFLOP

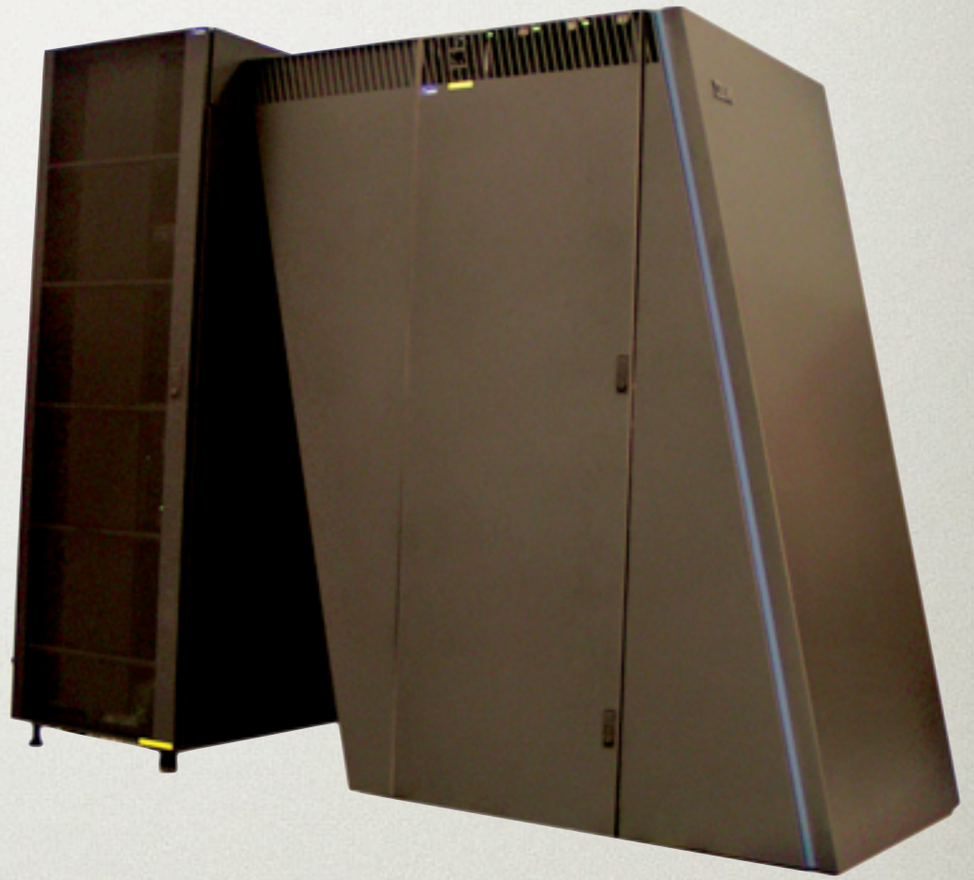
=



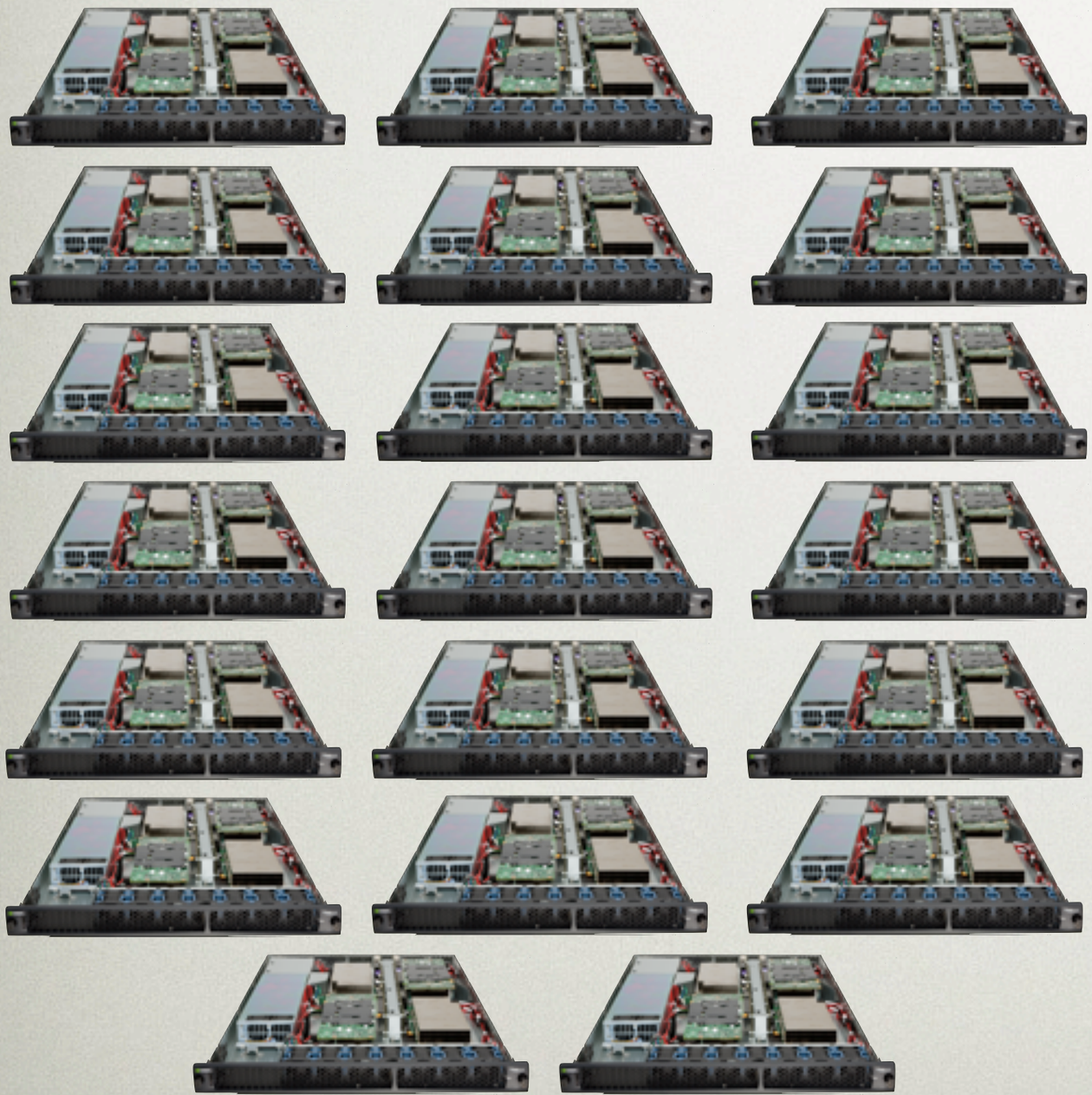
Performance Per MFLOP



=

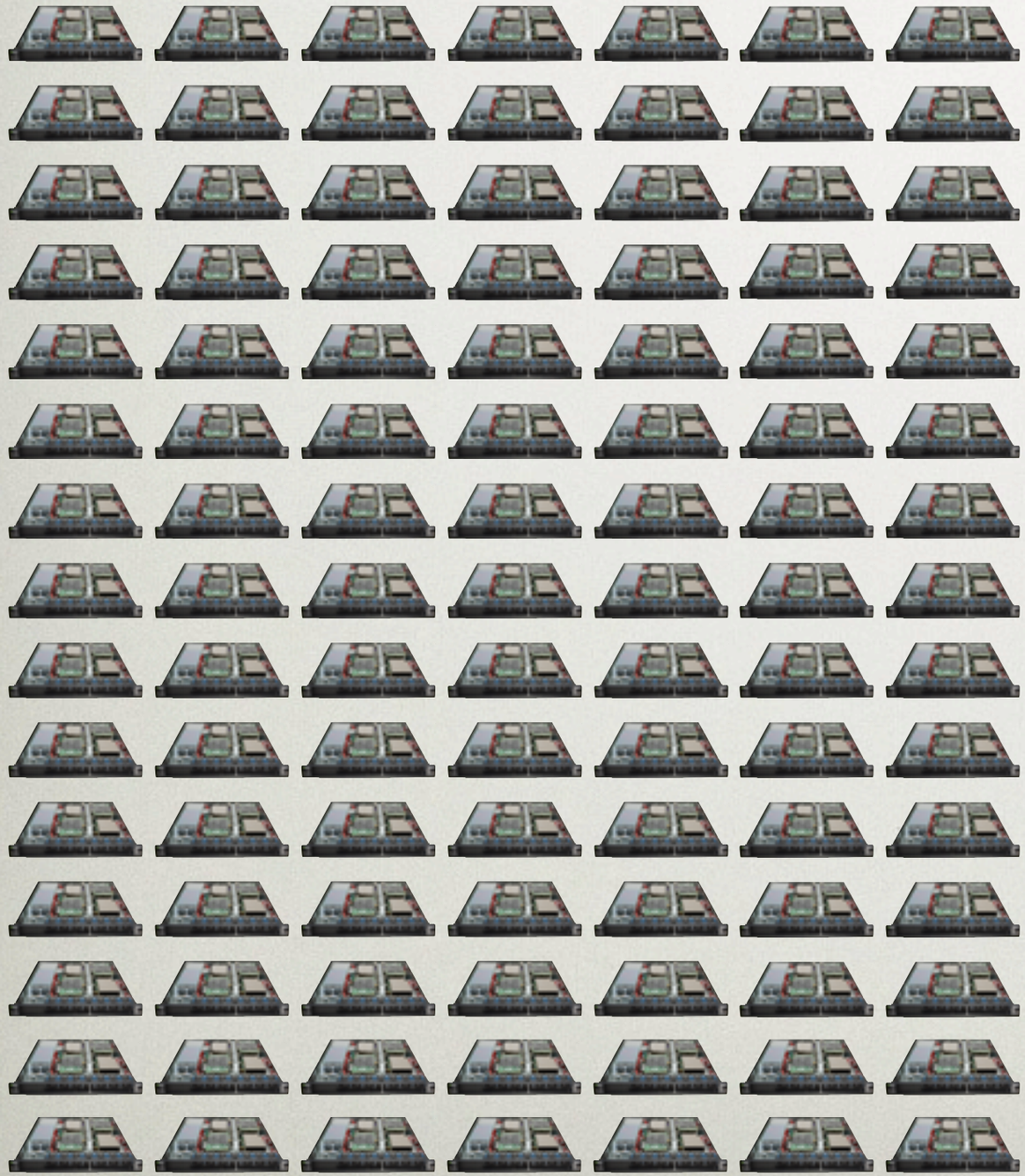


Performance Per Watt



=

Performance Per $



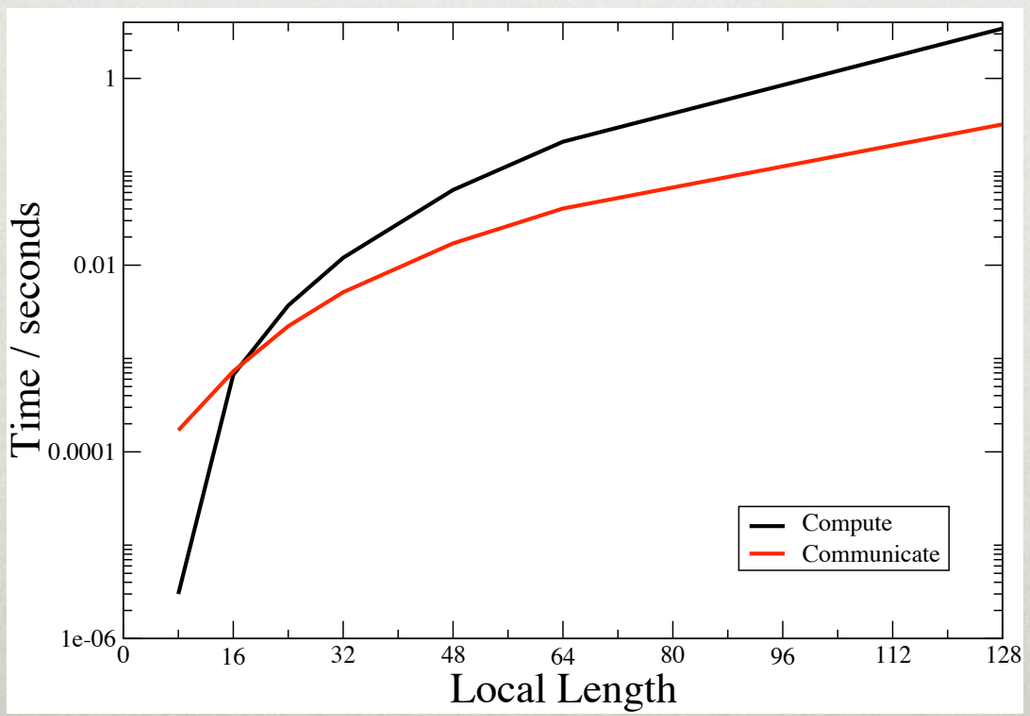
=

Multi-GPU

- Want to implement multi-GPU kernels
  - Overlap computation and communication
  - Limited by latency and inter-GPU bandwidth
- Multi-GPUs within a single PC or clusters with GPU in each?
- No-one's got hard numbers yet
- Model conservative performance scaling
  - Wilson dslash kernel @ 100 Gflops
  - PCIe bandwidth 5GBs^{-1}, latency = 5 μs
  - 8 way half-spinor communication

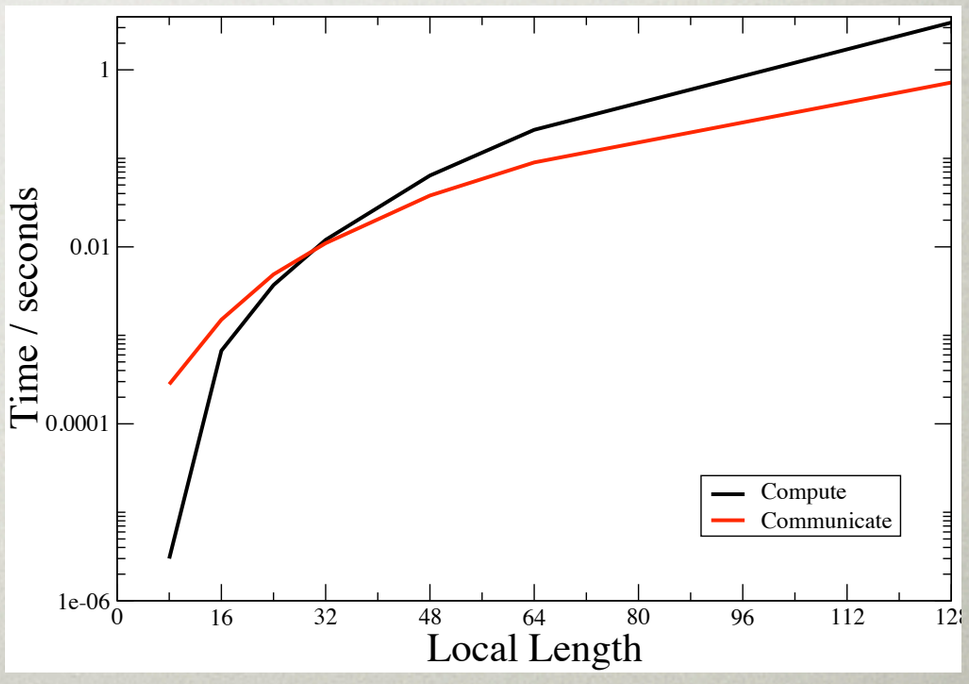
Possible Multi-GPU Configurations

- Multi-GPU within a box
  - Cheap and easy to set up
  - Should provide excellent scaling
  - O(1) Tflop under your desk!



Possible Multi-GPU Configurations

- True GPU cluster
  - Many nodes each with GPUs
- E.g., NCSA
  - 2x dual-core 2.4 GHz AMD Opterons, 8 GB of memory
  - 1xTesla S1070
  - 2GB/sec Infiniband connection
- Good scaling very challenging
  - Need large local volume
  - Better algorithms (DD solver?)
  - Custom hardware?



Isn't it a lot of work to write FAST GPU code?

- Many optimization parameters to consider
  - Loop unrolling
  - Depth of parallelism
  - Order of execution
- Also algorithm/physics parameters
  - Matrix transpose, SU(3) parameterization, gamma matrix basis
- Remove grunt work by writing code to write code
  - Python code generator creates CUDA code
  - Optimization parameters are just that, parameters
- Quickly find maximum in performance space

Conclusions

- Fantastic algorithmic performance obtained on today GPUs
  - Flops per Watt, Flops per $ -> new metric Time to solution per $
- Some work required to get best performance
- Algorithm design critical component
  - Bandwidth constraints force rethink of problem
  - New algorithms
- Many groups harnessing the power of GPUs
- Next step is multi-GPU
  - Case yet to be proven
  - Prediction: next year lots to report

QUDA Library

- QCD-GPU library
  - Wilson and Wilson-clover fermions
  - Inverter interface (SciDac level 3)
  - Dslash interface (accelerator mode)
- Hooks into QDP / CPS++ etc.
- Future...
  - Domain wall, twisted mass
  - Fermion and Gauge forces
  - Multi-GPU
  - Adaptive multigrid