BROOKHAVEN NATIONAL LABORATORY

INDIANA UNIVERSITY

Universität Bielefeld

Conjugate gradient solvers

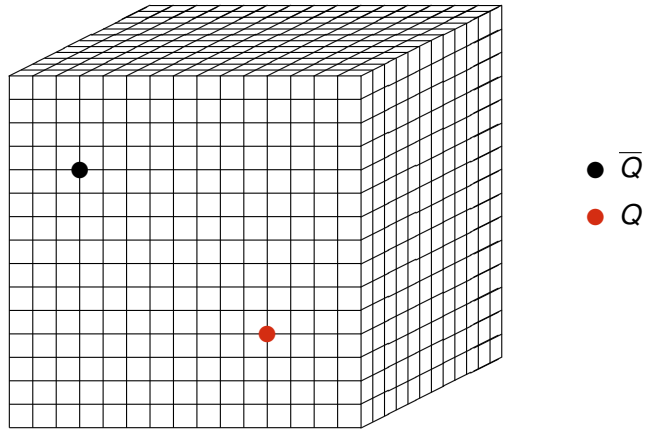
on Intel® Xeon Phi™ and NVIDIA® GPUs

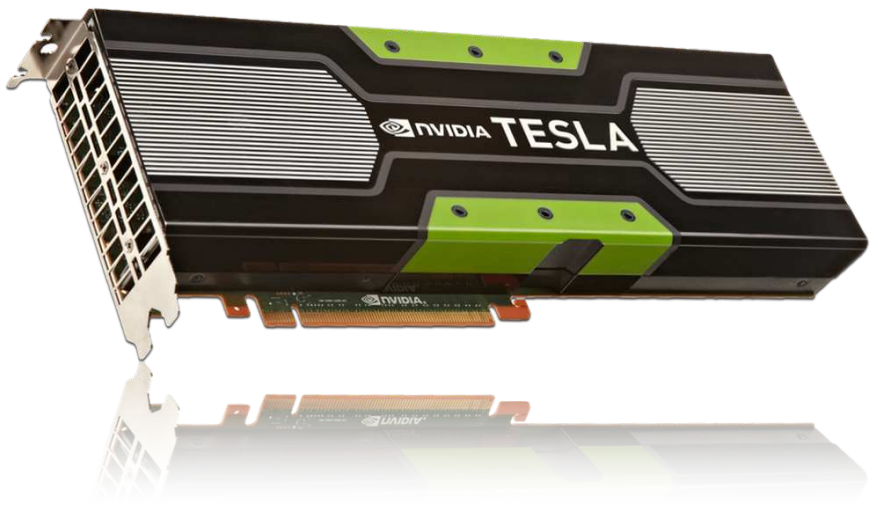
September 10, 2014 | Patrick Steinbrecher |

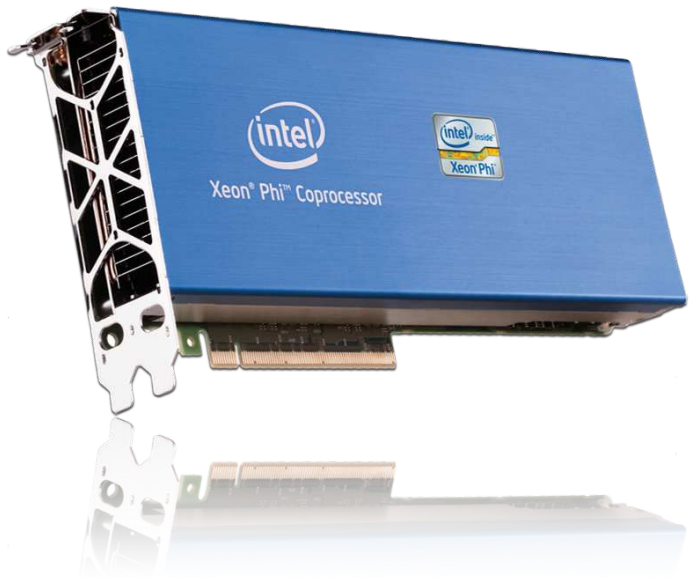
Olaf Kaczmarek
Swagato Mukherjee
Christian Schmidt
Mathias Wagner

Outline

- Lattice QCD
- Implementation
- Comparison

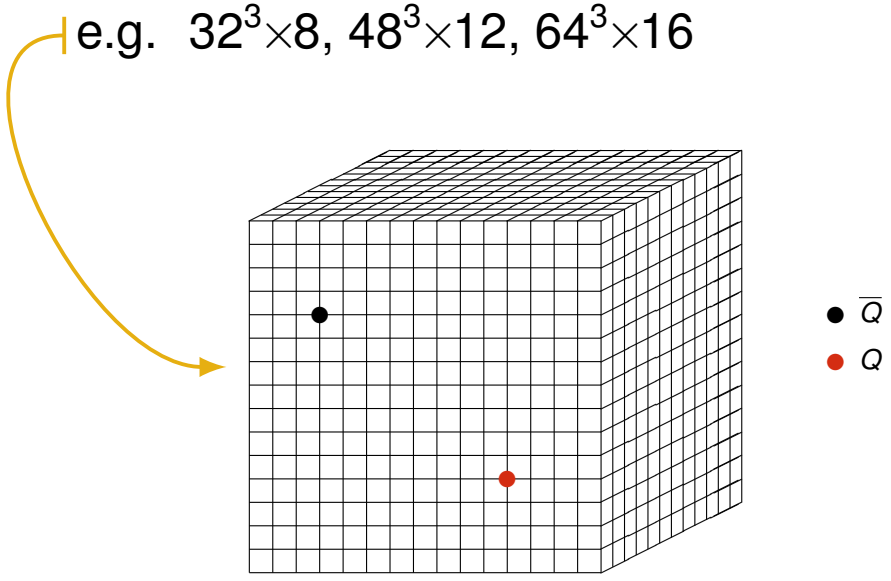


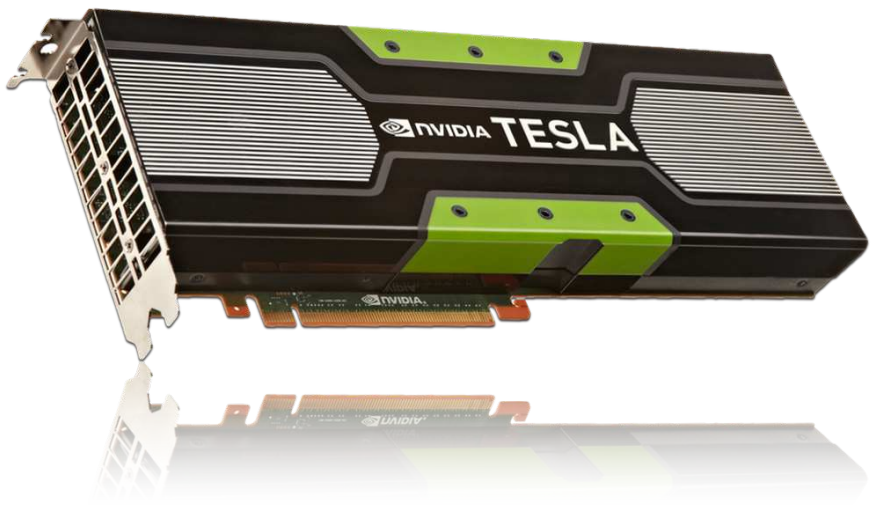


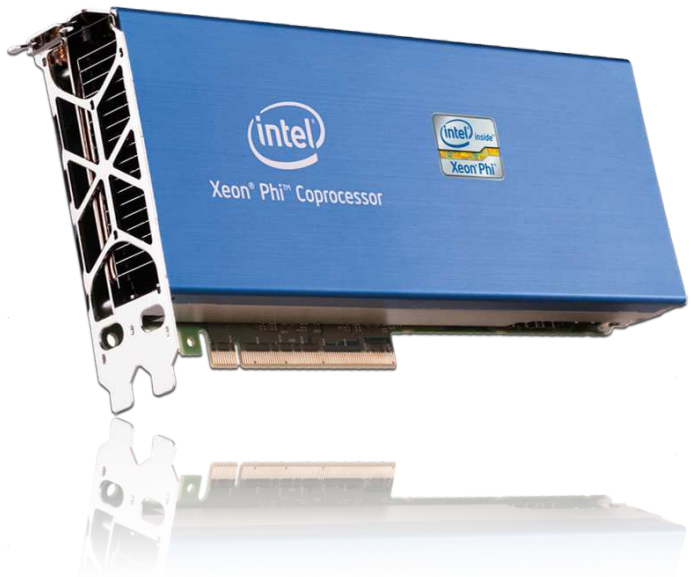


Outline

- Lattice QCD
- Implementation
- Comparison







Lattice QCD measurements

- Chiral Condensate

$$\langle \overline{\psi}\psi \rangle = \frac{\partial \ln \mathcal{Z}}{\partial m_q} = \frac{n_f}{4} \left\langle \mathrm{Tr}\ M^{-1} \right\rangle$$

Lattice QCD measurements

- Chiral Condensate

$$\langle \overline{\psi}\psi \rangle = \frac{\partial \ln \mathcal{Z}}{\partial m_q} = \frac{n_f}{4} \left\langle \mathrm{Tr}\ M^{-1} \right\rangle$$

Conjugate Gradient

solve:

$$M x = y$$

$$x = M^{-1} y$$

Lattice QCD measurements

- Chiral Condensate

$$\langle \overline{\psi}\psi \rangle = \frac{\partial \ln \mathcal{Z}}{\partial m_q} = \frac{n_f}{4} \left\langle \mathrm{Tr}\, M^{-1} \right\rangle$$

Conjugate Gradient

solve:

$$M x = y$$

$$x = M^{-1} y$$

- derivatives of $\ln \mathcal{Z}$ w.r.t. the chemical potential μ

$$\frac{\partial^2 \ln \mathcal{Z}}{\partial \mu^2} = \left\langle \frac{n_f}{4} \frac{\partial^2 \left(\ln \det M\right)}{\partial \mu^2} \right\rangle + \left\langle \left(\frac{n_f}{4} \frac{\partial \left(\ln \det M\right)}{\partial \mu} \right)^2 \right\rangle$$

$$\mathrm{Tr}\left(M^{-1} \frac{\partial^2 M}{\partial \mu^2} \right) - \mathrm{Tr}\left(M^{-1} \frac{\partial M}{\partial \mu} M^{-1} \frac{\partial M}{\partial \mu} \right) \qquad \mathrm{Tr}\left(M^{-1} \frac{\partial M}{\partial \mu} \right)$$

Random noise method

condition:

$$\lim_{N\to\infty} \frac{1}{N} \sum_{k=1}^{N} \eta_{ki}^{*} \eta_{kj} = \delta_{ij}$$

(simplest case)

$$\mathrm{Tr}\, M^{-1} = \lim_{N\to\infty} \frac{1}{N} \sum_{k=1}^{N} \eta_k^{\dagger} M^{-1} \eta_k$$

Random noise method

condition:

$$\lim_{N\to\infty} \frac{1}{N} \sum_{k=1}^{N} \eta_{ki}^{*} \eta_{kj} = \delta_{ij}$$

(simplest case)

$$\mathrm{Tr}\, M^{-1} = \lim_{N\to\infty} \frac{1}{N} \sum_{k=1}^{N} \eta_k^{\dagger} M^{-1} \eta_k$$

- evaluate traces using N random noise vectors η error $\sim N^{-1/2}$

Random noise method

condition:

$$\lim_{N\to\infty} \frac{1}{N} \sum_{k=1}^{N} \eta_{ki}^{*} \eta_{kj} = \delta_{ij}$$

(simplest case)

$$\mathrm{Tr}\, M^{-1} = \lim_{N\to\infty} \frac{1}{N} \sum_{k=1}^{N} \eta_k^{\dagger} M^{-1} \eta_k$$

- evaluate traces using N random noise vectors η — error $\sim N^{-1/2}$
  - up to 1500 η per gauge field configuration

Random noise method

condition:

$$\lim_{N\to\infty} \frac{1}{N} \sum_{k=1}^{N} \eta_{ki}^{*} \eta_{kj} = \delta_{ij}$$

(simplest case)

$$\mathrm{Tr}\, M^{-1} = \lim_{N\to\infty} \frac{1}{N} \sum_{k=1}^{N} \eta_k^{\dagger} M^{-1} \eta_k$$

- evaluate traces using N random noise vectors η — error $\sim N^{-1/2}$
  - up to 1500 η per gauge field configuration
- up to 20000 configurations for each temperature
  - implies $\sim 10^7$ inversions / temperature

Random noise method

condition:

$$\lim_{N\to\infty} \frac{1}{N} \sum_{k=1}^{N} \eta_{ki}^{*} \eta_{kj} = \delta_{ij}$$

(simplest case)

99% of runtime

$$\mathrm{Tr}\, M^{-1} = \lim_{N\to\infty} \frac{1}{N} \sum_{k=1}^{N} \eta_k^{\dagger} M^{-1} \eta_k$$

- evaluate traces using N random noise vectors η — error $\sim N^{-1/2}$
  - up to 1500 η per gauge field configuration
- up to 20000 configurations for each temperature
  - implies $\sim 10^7$ inversions / temperature

Random noise method

(simplest case)

99% of runtime

condition:

$$\lim_{N\to\infty} \frac{1}{N} \sum_{k=1}^{N} \eta_{ki}^{*} \eta_{kj} = \delta_{ij}$$

$$\mathrm{Tr}\, M^{-1} = \lim_{N\to\infty} \frac{1}{N} \sum_{k=1}^{N} \eta_k^{\dagger} M^{-1} \eta_k$$

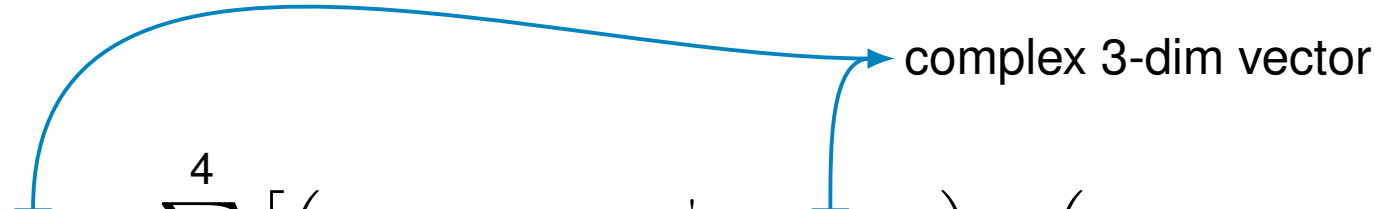
- evaluate traces using N random noise vectors η — error $\sim N^{-1/2}$
  - up to 1500 η per gauge field configuration
- up to 20000 configurations for each temperature
  - implies $\sim 10^7$ inversions / temperature
    - CG
      - 90% Dslash
      - 10% Linear Algebra

($64^3 \times 16$, Xeon Phi™ 5110P)

HISQ Dslash $w = \not{D} v$

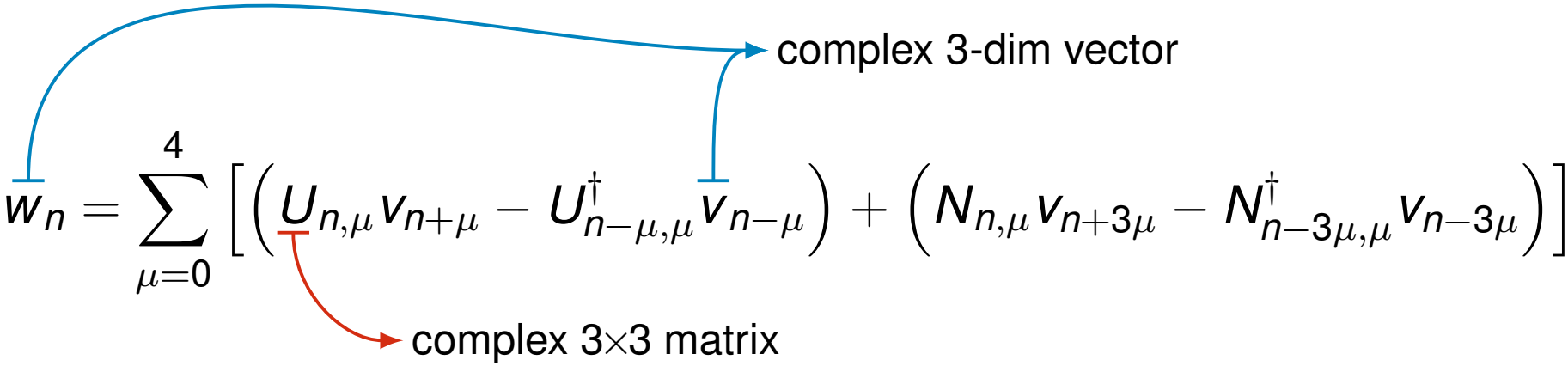
$$w_n = \sum_{\mu=0}^{4} \left[\left(U_{n,\mu} v_{n+\mu} - U^{\dagger}_{n-\mu,\mu} v_{n-\mu} \right) + \left(N_{n,\mu} v_{n+3\mu} - N^{\dagger}_{n-3\mu,\mu} v_{n-3\mu} \right) \right]$$

HISQ Dslash $w = \not{D} v$

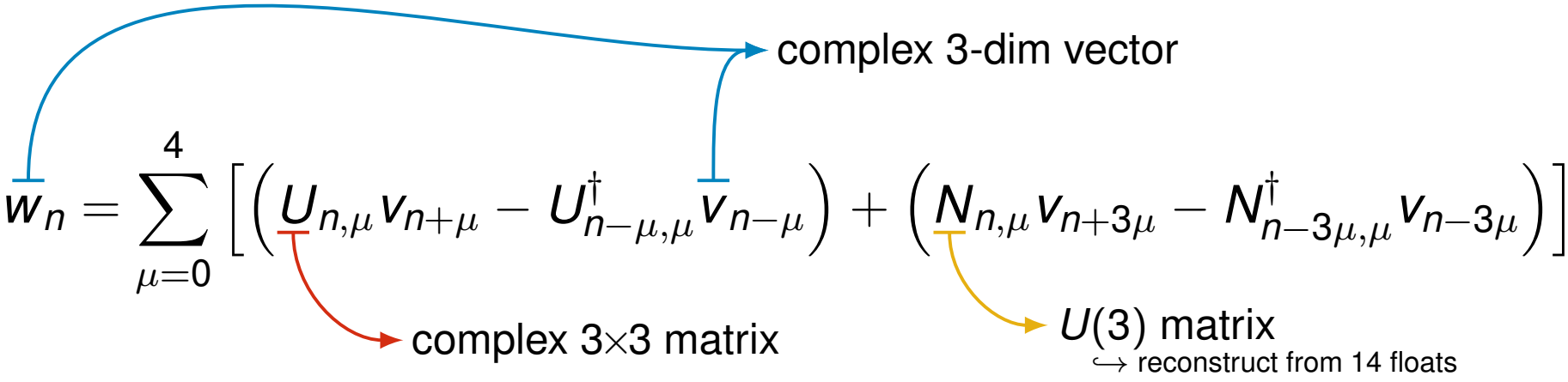


$$w_n = \sum_{\mu=0}^{4} \left[\left(U_{n,\mu} v_{n+\mu} - U^{\dagger}_{n-\mu,\mu} v_{n-\mu} \right) + \left(N_{n,\mu} v_{n+3\mu} - N^{\dagger}_{n-3\mu,\mu} v_{n-3\mu} \right) \right]$$

HISQ Dslash $w = \not{D} v$



HISQ Dslash $w = \not{D} v$



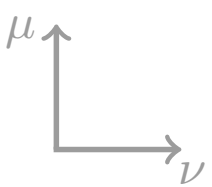
HISQ Dslash $w = \not{D} v$

complex 3-dim vector

$$w_n = \sum_{\mu=0}^{4} \left[\left(U_{n,\mu} v_{n+\mu} - U^{\dagger}_{n-\mu,\mu} v_{n-\mu} \right) + \left(N_{n,\mu} v_{n+3\mu} - N^{\dagger}_{n-3\mu,\mu} v_{n-3\mu} \right) \right]$$

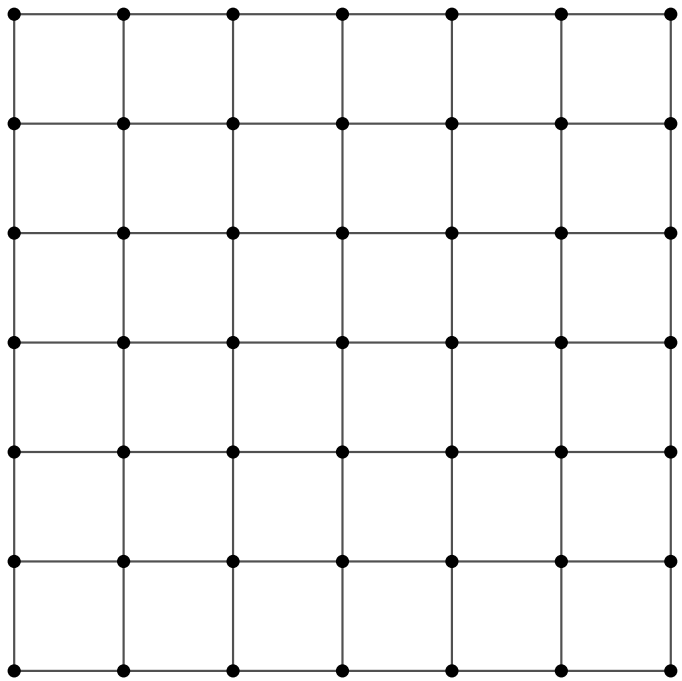
complex 3×3 matrix

$U(3)$ matrix
↪ reconstruct from 14 floats



matrix

vector



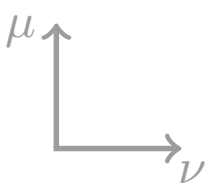
HISQ Dslash $w = \not{D} v$

complex 3-dim vector

$$w_n = \sum_{\mu=0}^{4} \left[\left(U_{n,\mu} v_{n+\mu} - U^{\dagger}_{n-\mu,\mu} v_{n-\mu} \right) + \left(N_{n,\mu} v_{n+3\mu} - N^{\dagger}_{n-3\mu,\mu} v_{n-3\mu} \right) \right]$$

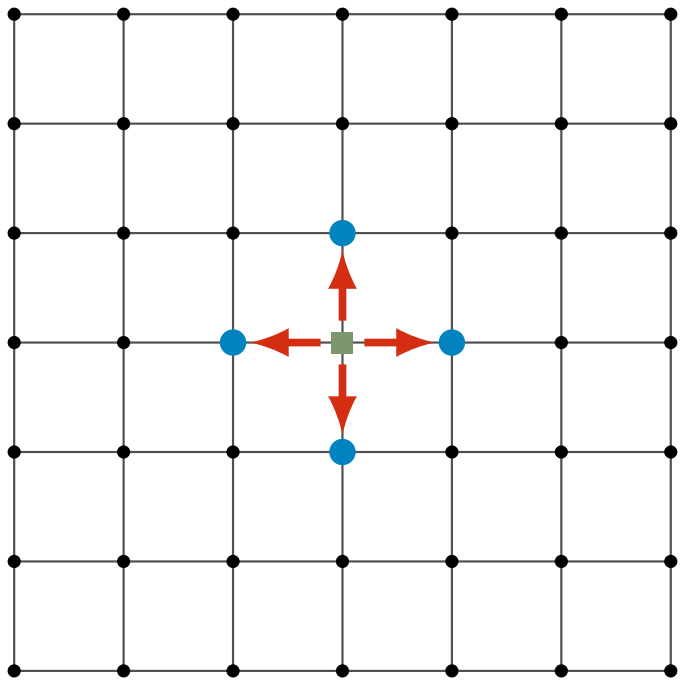
complex 3×3 matrix

$U(3)$ matrix
↪ reconstruct from 14 floats



matrix

vector



$w_{\blacksquare} =$ standard

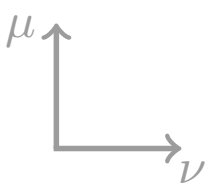
HISQ Dslash $w = \not{D} v$

complex 3-dim vector

$$w_n = \sum_{\mu=0}^{4} \left[\left(U_{n,\mu} v_{n+\mu} - U^{\dagger}_{n-\mu,\mu} v_{n-\mu} \right) + \left(N_{n,\mu} v_{n+3\mu} - N^{\dagger}_{n-3\mu,\mu} v_{n-3\mu} \right) \right]$$

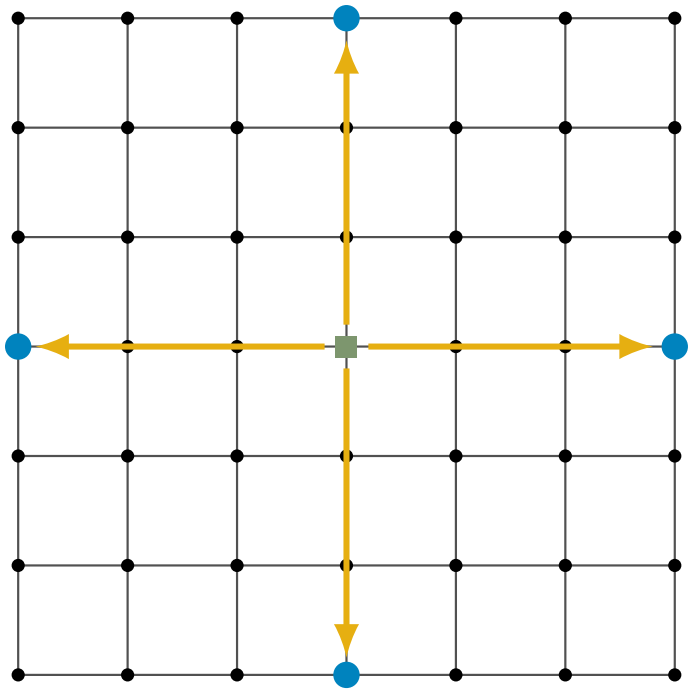
complex 3×3 matrix

$U(3)$ matrix
↪ reconstruct from 14 floats



matrix

vector



$w_{\blacksquare}$ = standard

\+ naik

↳ precalculated three-link term

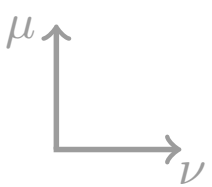
HISQ Dslash $w = \not{D} v$

complex 3-dim vector

$$w_n = \sum_{\mu=0}^{4} \left[\left(U_{n,\mu} v_{n+\mu} - U^{\dagger}_{n-\mu,\mu} v_{n-\mu} \right) + \left(N_{n,\mu} v_{n+3\mu} - N^{\dagger}_{n-3\mu,\mu} v_{n-3\mu} \right) \right]$$

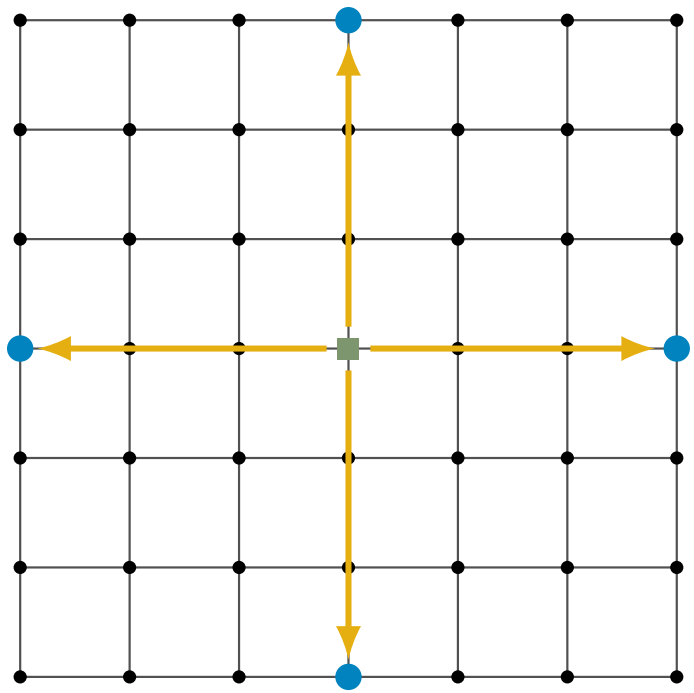
complex 3×3 matrix

$U(3)$ matrix
↪ reconstruct from 14 floats



matrix

vector



1146 Flop/site

0.8 Flop/byte
↪ single-precision

$w_{\blacksquare}$ = standard
+ naik
↳ precalculated three-link term

Multiple right-hand sides

Memory

Memory

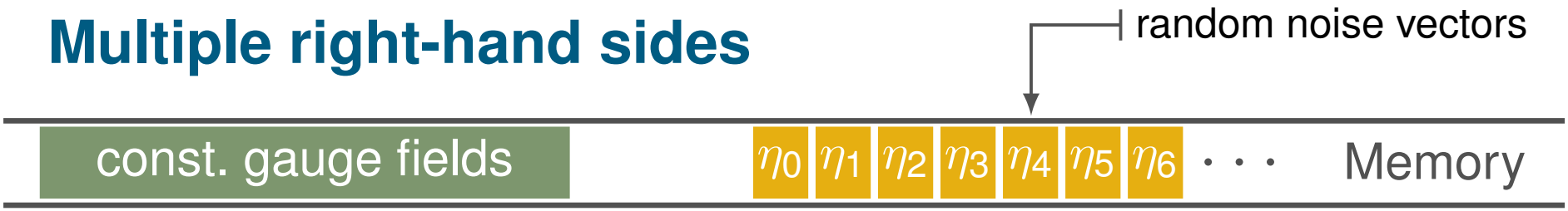
Multiple right-hand sides

const. gauge fields

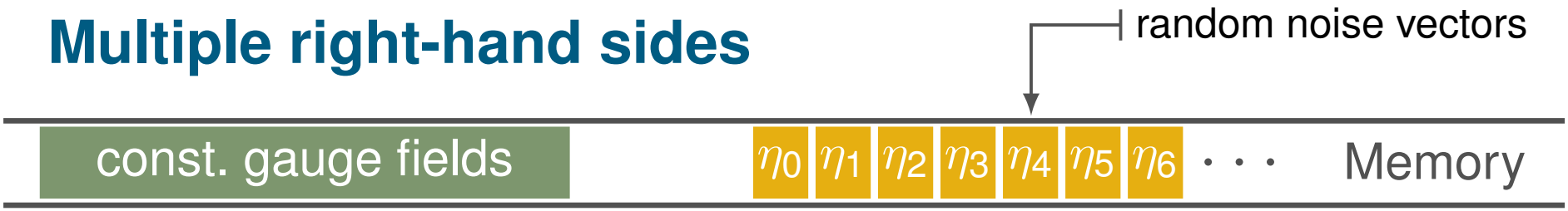
Memory

Memory

Multiple right-hand sides



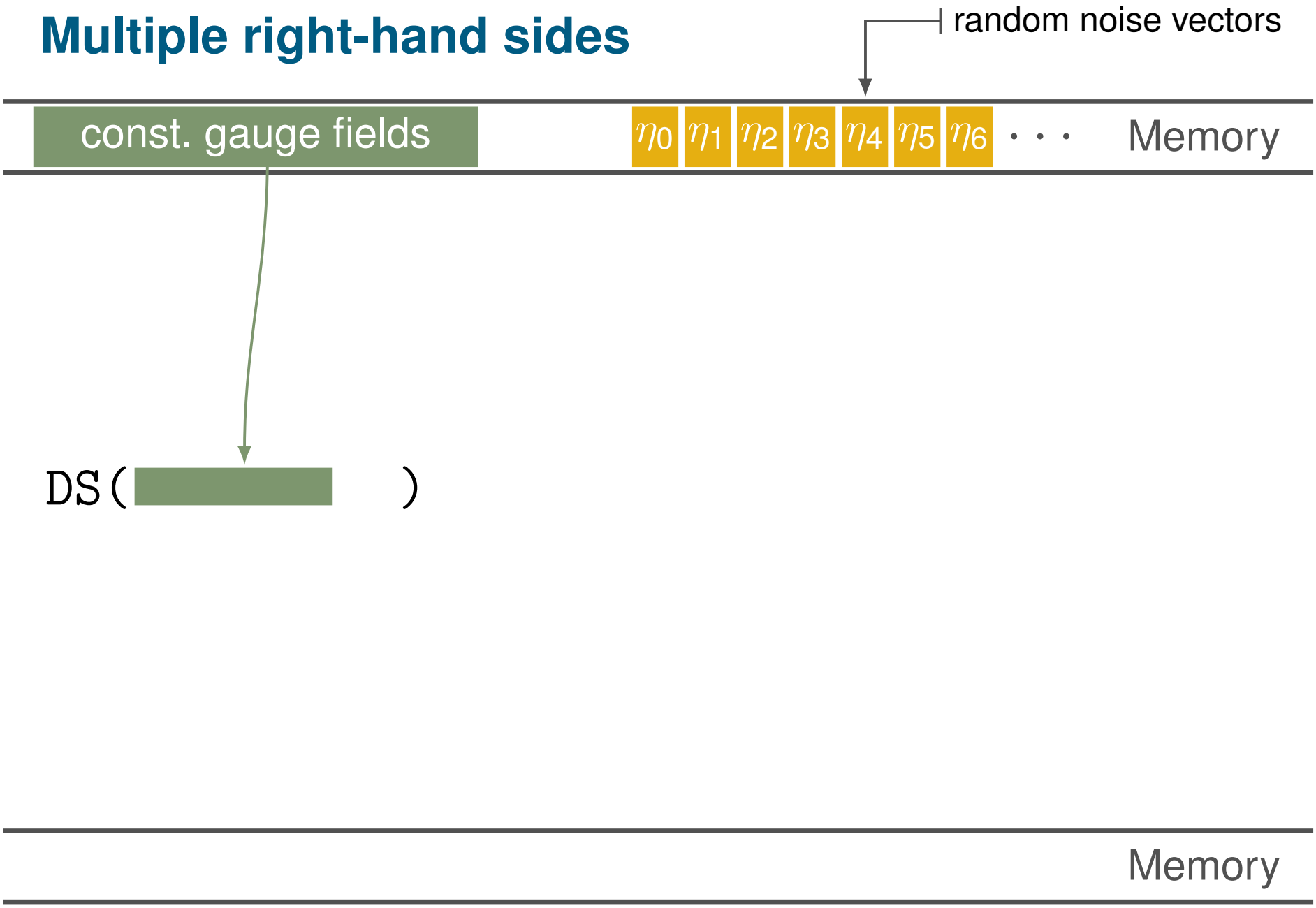
Multiple right-hand sides



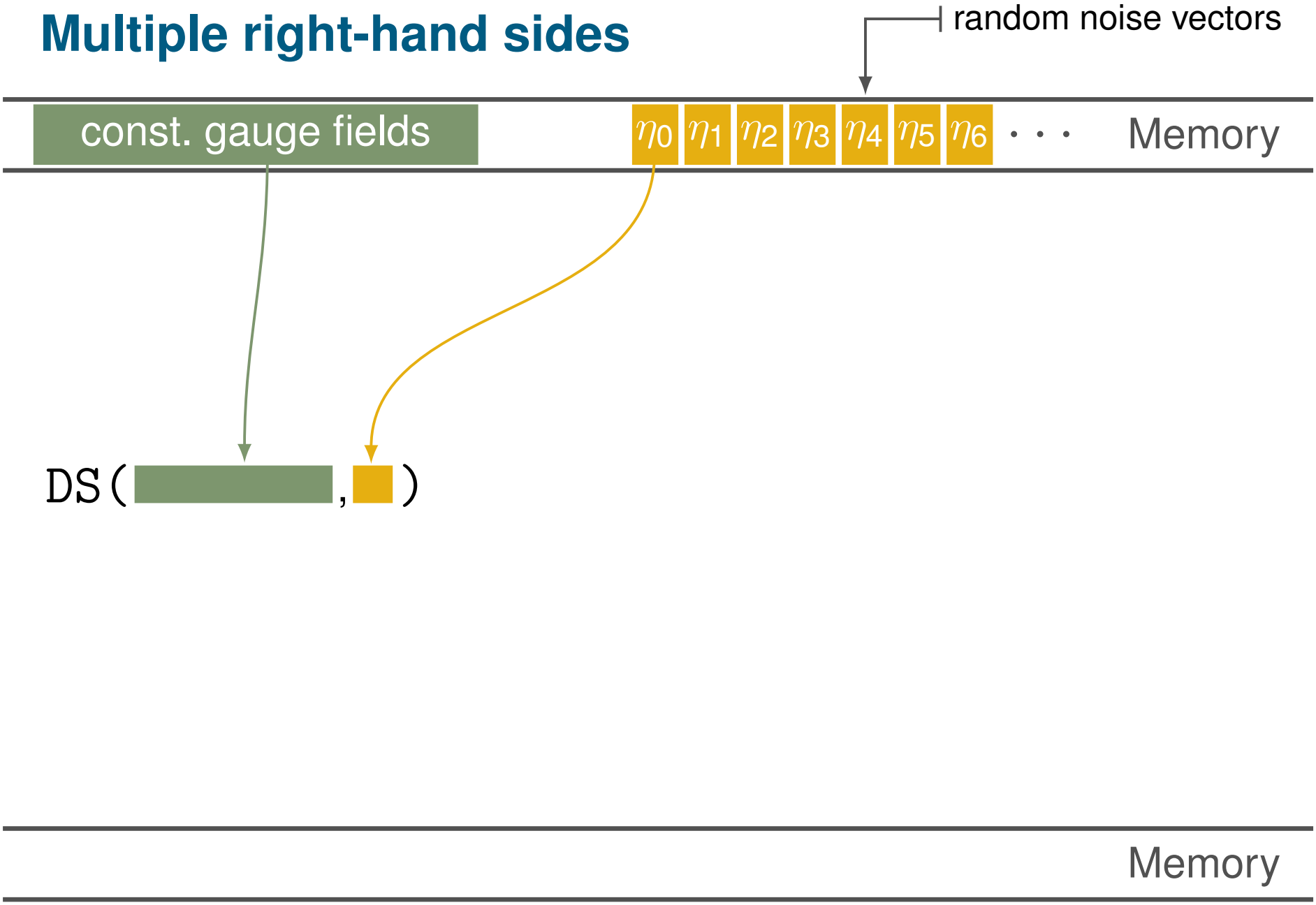
DS( )

Memory

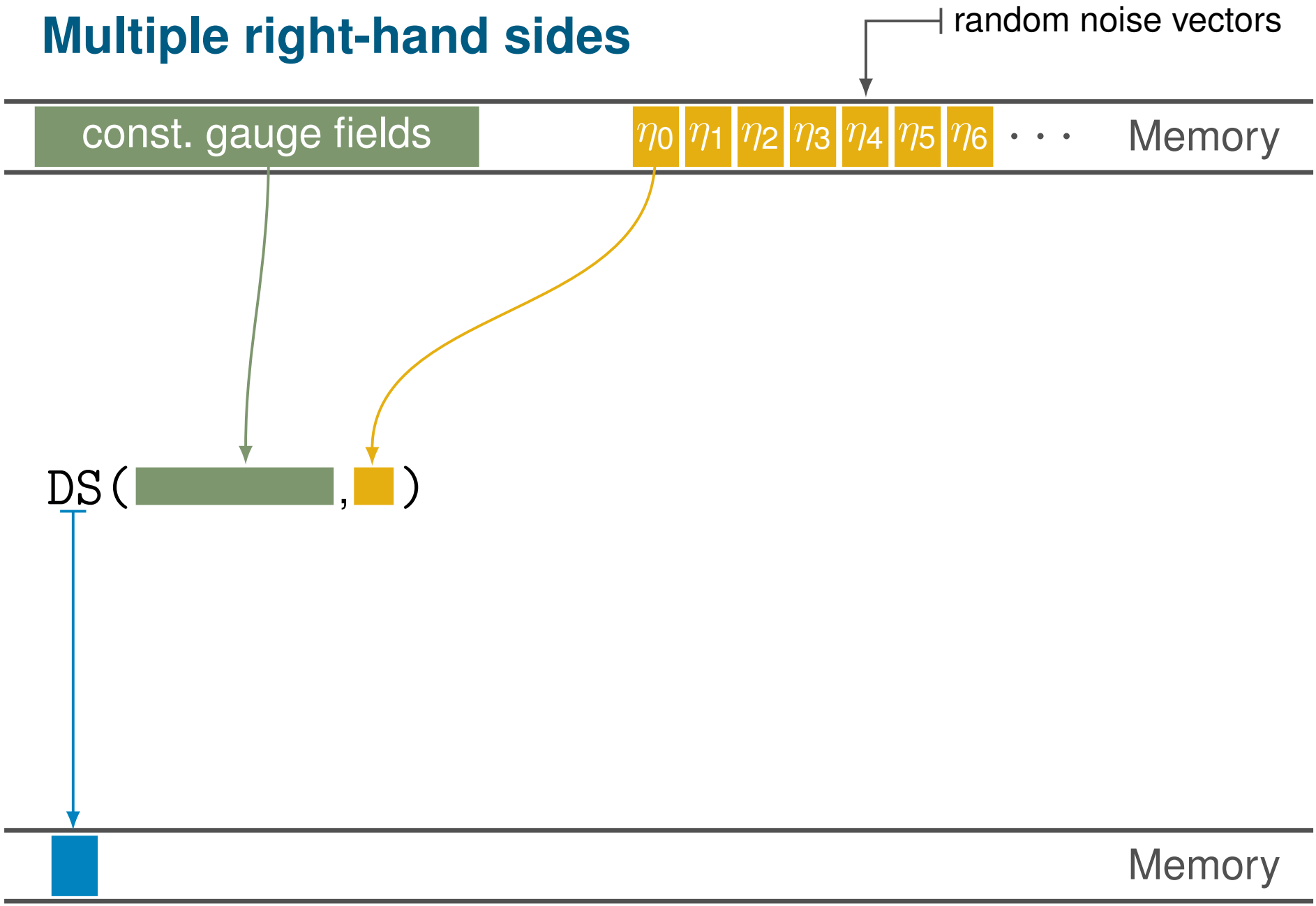
Multiple right-hand sides



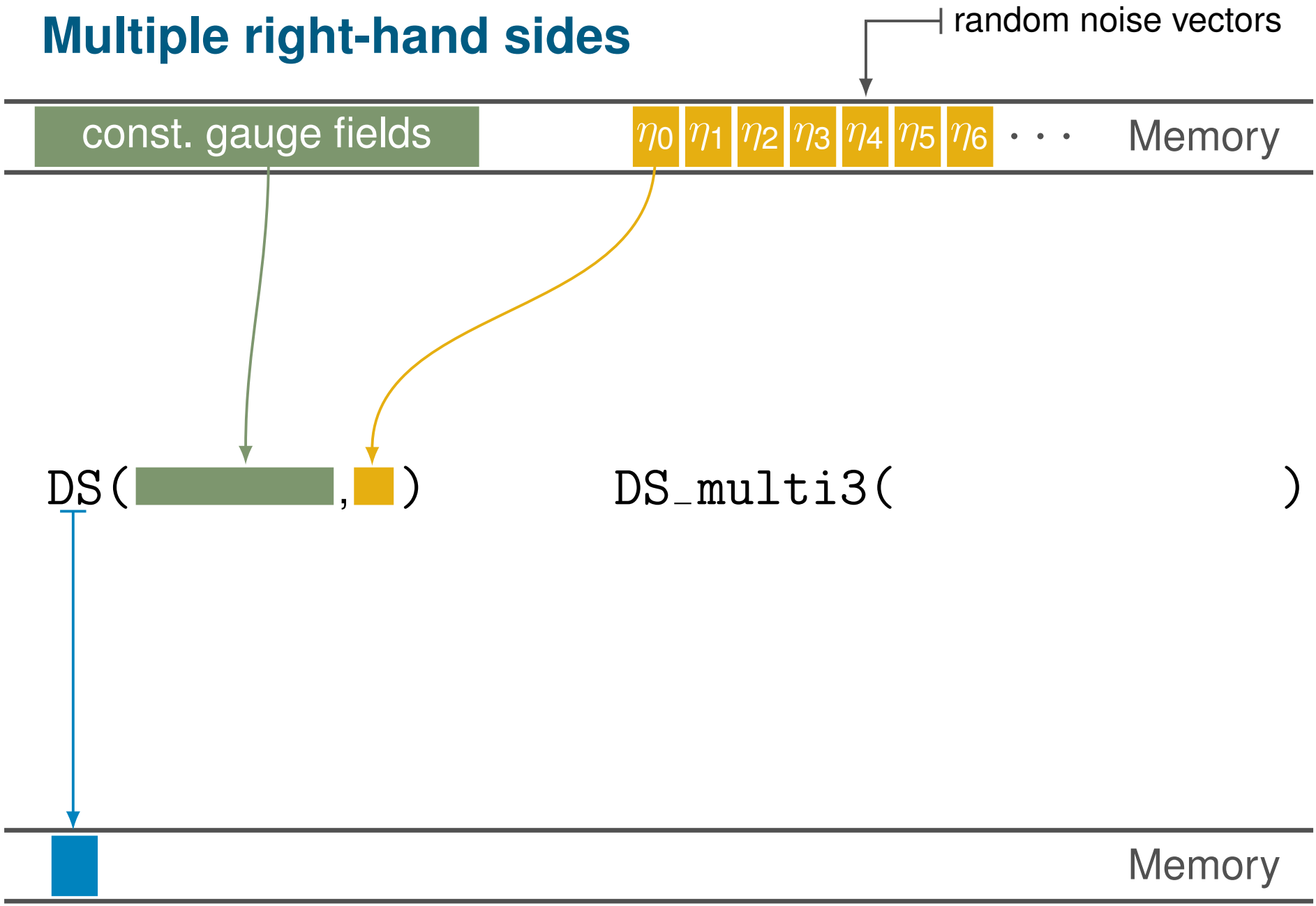
Multiple right-hand sides



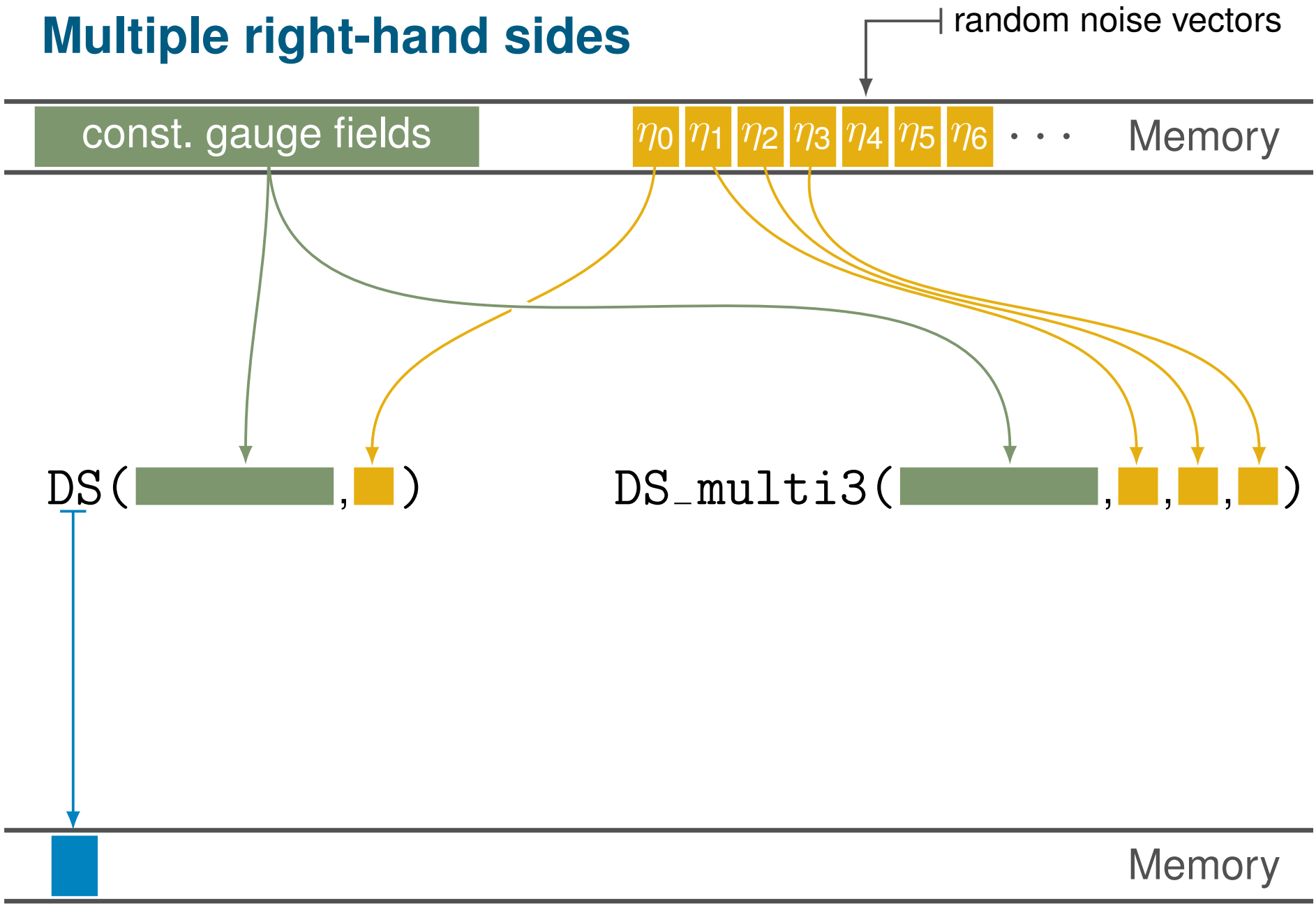
Multiple right-hand sides



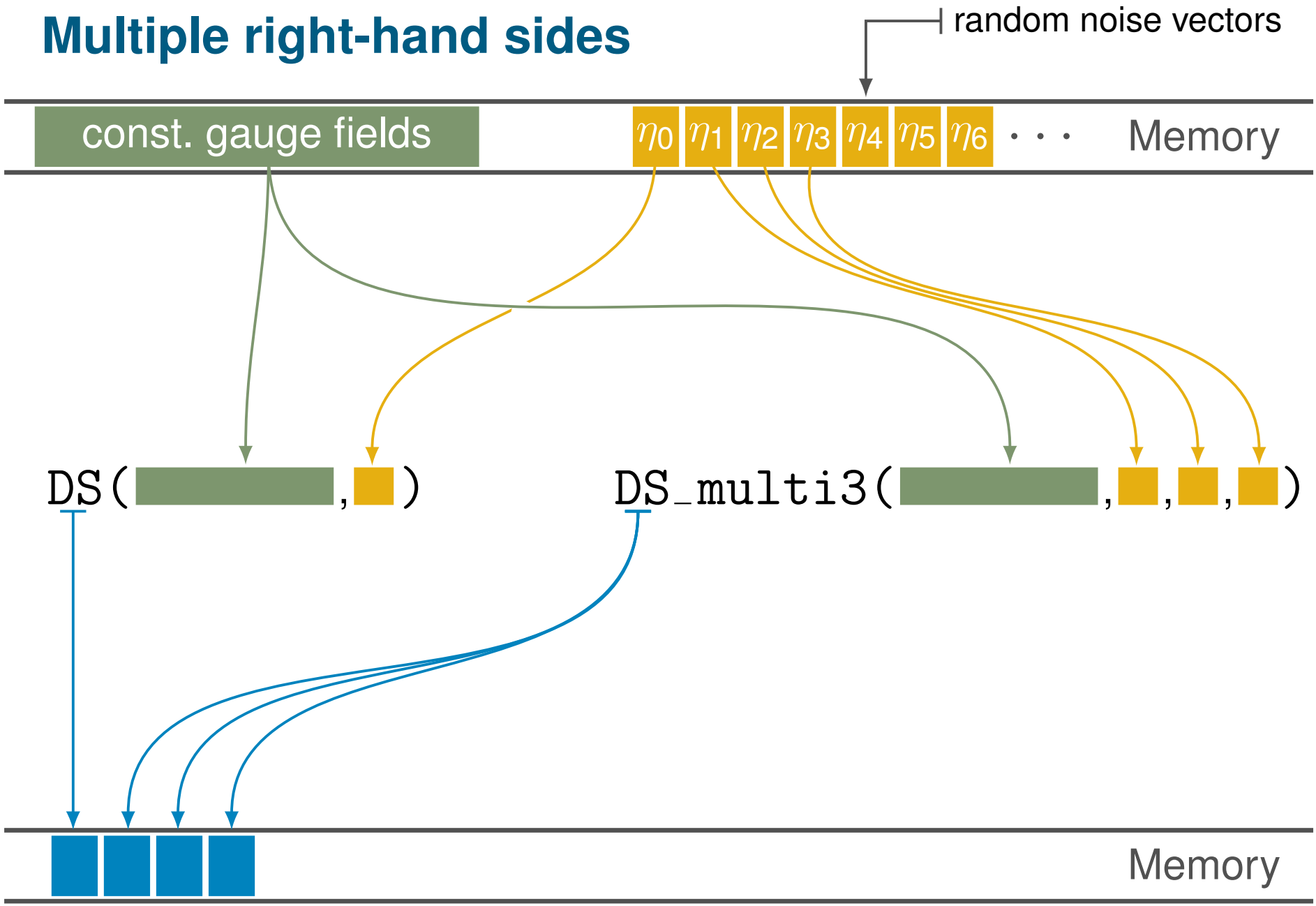
Multiple right-hand sides



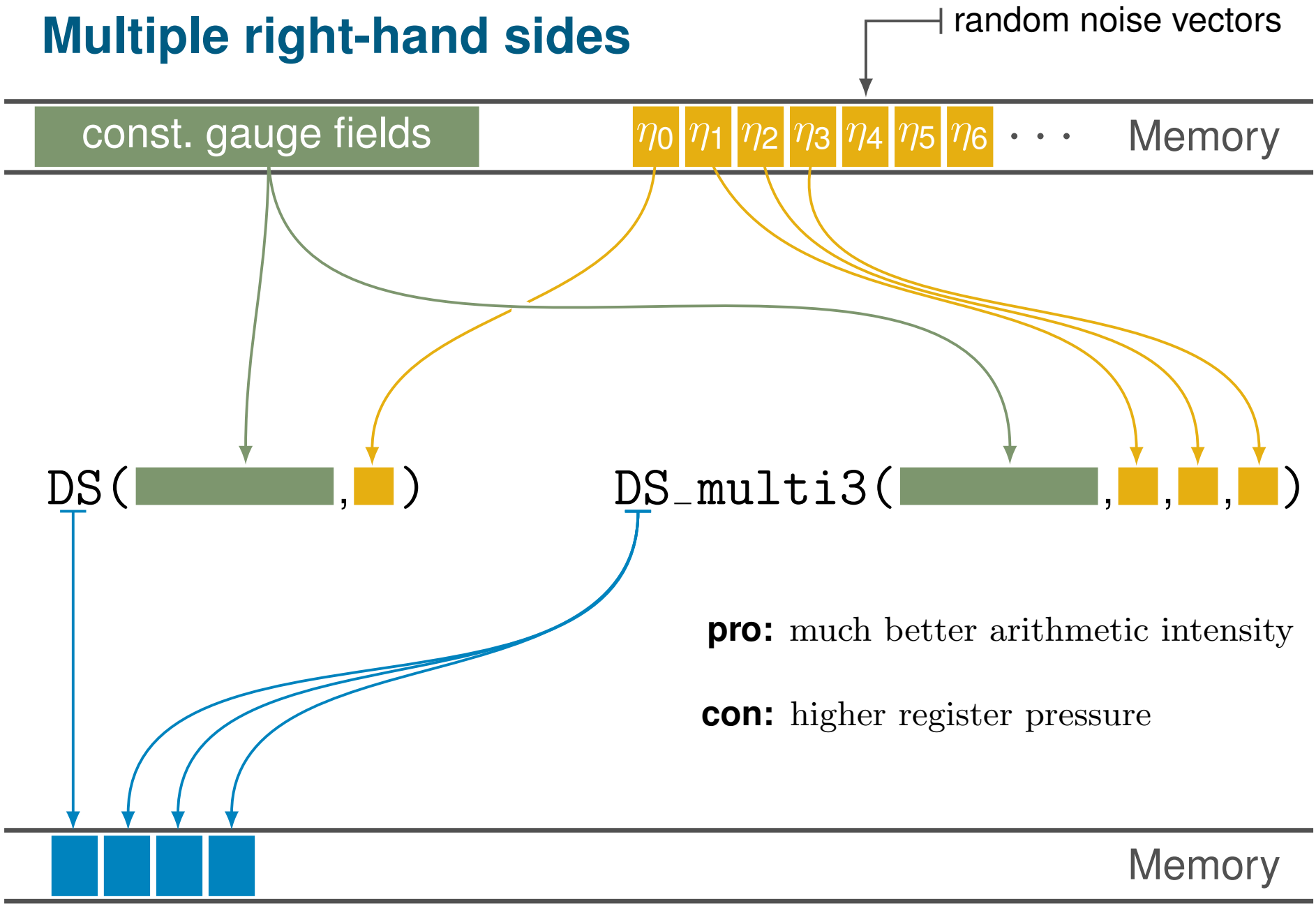
Multiple right-hand sides



Multiple right-hand sides



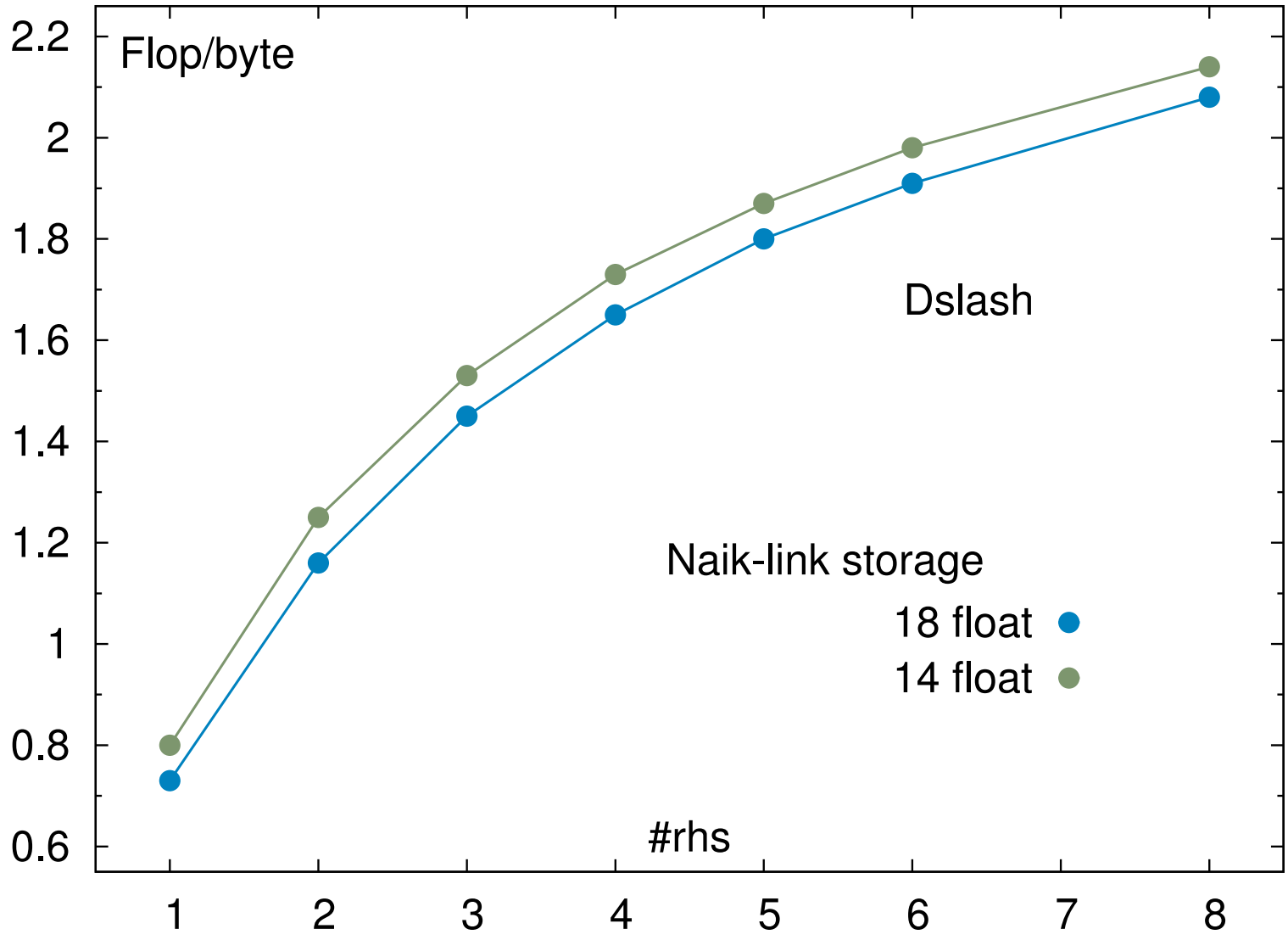
Multiple right-hand sides



pro: much better arithmetic intensity

con: higher register pressure

Multiple right-hand sides (rhs)



Multiple right-hand sides (rhs)

- gauge compression gives no significant perf. increase
- multiple rhs improves only Dslash operator
  - Linear Algebra still scales linearly

($64^3 \times 16$, Xeon Phi™ 5110P)

Implementation

GPU: since 2009

code `CUDA`

cards M2075
GTX™ 580
GTX™ Titan
K20
K40 ...

used for

A. Bazavov et al., arXiv:1404.4043 [hep-lat]
A. Bazavov et al., arXiv:1404.6511 [hep-lat]
A. Bazavov et al., Phys. Rev. Lett. **111**, 082301 (2013)
A. Bazavov et al., Phys. Rev. Lett. **109**, 192302 (2012)
⋮

MIC: since 2014
↪ more to explore
more to improve

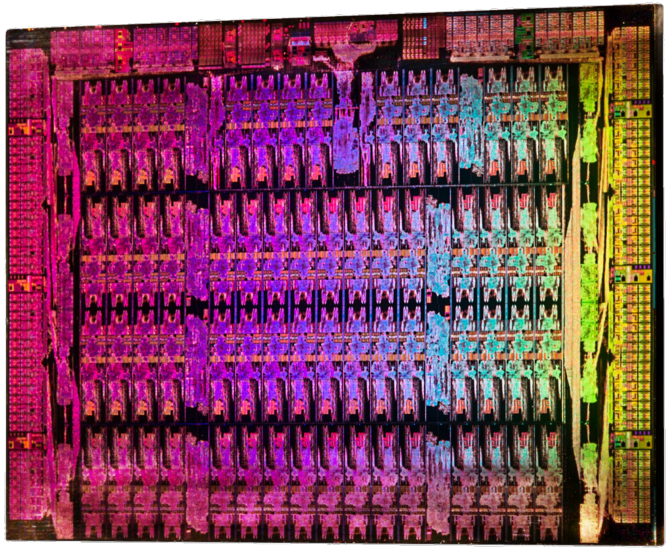
code `C++, OpenMP`
`Intrinsics`

cards Xeon Phi™ 5110P
Xeon Phi™ 7120P

used for
↪ will be used on larger clusters

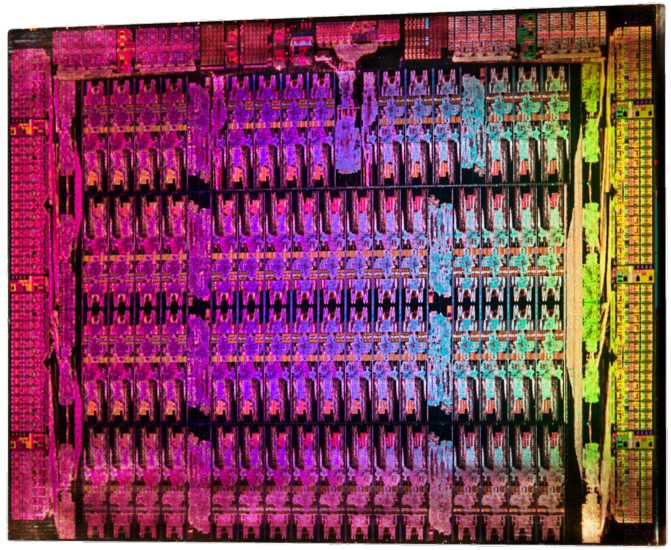
Intel® Xeon Phi™

- x86 based many-core processor
  - runs a Linux μOS
  - in-order execution
  - up to 61 cores at 1.238 GHz
  - core boost for 7120 series
  - theoretical bandwidth 352 $\mathrm{GB/s}$



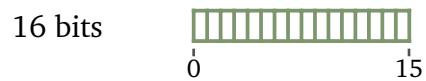
Intel® Xeon Phi™

- x86 based many-core processor
  - runs a Linux μOS
  - in-order execution
  - up to 61 cores at 1.238 GHz
  - core boost for 7120 series
  - theoretical bandwidth 352 GB/s



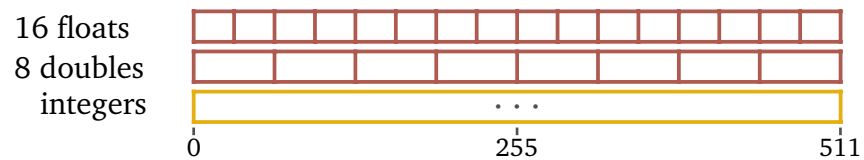
- parallelism through
  - large SIMD registers

Mask Registers:



Eight 16-bit k registers per core.

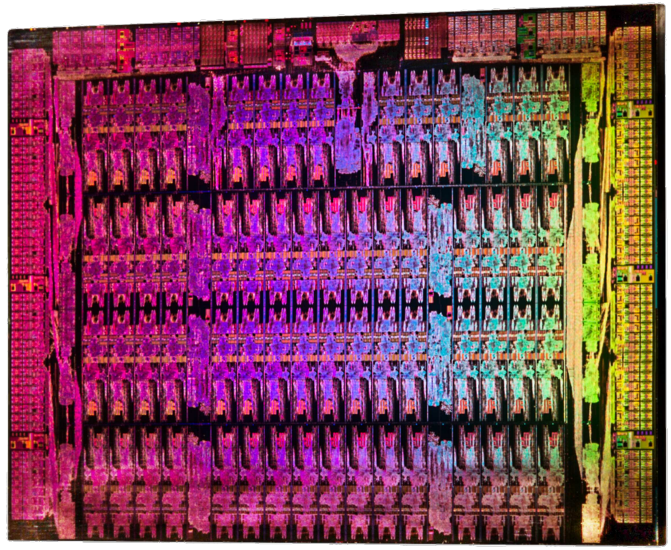
Vector Registers:



Thirty-two 512-bit zmm registers per core.

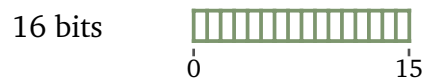
Intel® Xeon Phi™

- x86 based many-core processor
  - runs a Linux μOS
  - in-order execution
  - up to 61 cores at 1.238 GHz
  - core boost for 7120 series
  - theoretical bandwidth 352 GB/s



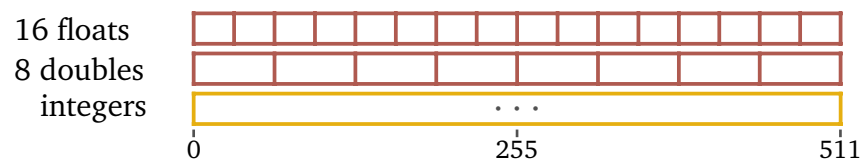
- parallelism through
  - large SIMD registers
  - 4 hardware threads per core

Mask Registers:



Eight 16-bit k registers per core.

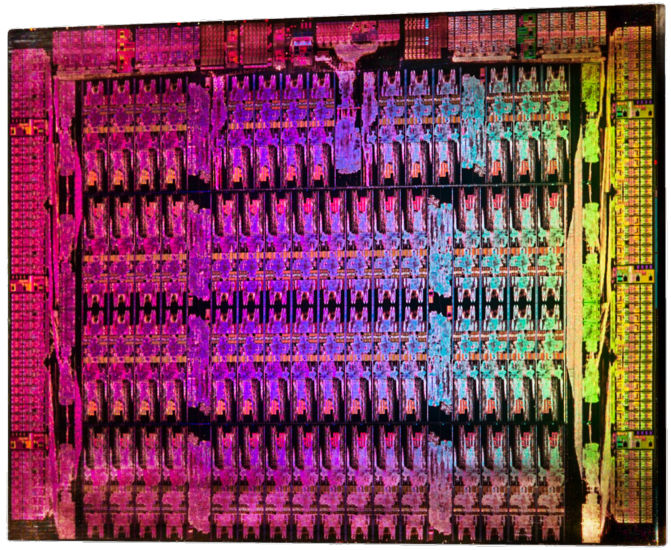
Vector Registers:



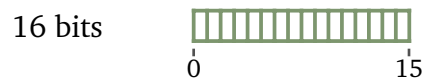
Thirty-two 512-bit zmm registers per core.

Intel® Xeon Phi™

- x86 based many-core processor
  - runs a Linux μOS
  - in-order execution
  - up to 61 cores at 1.238 GHz
  - core boost for 7120 series
  - theoretical bandwidth 352 GB/s
- parallelism through
  - large SIMD registers
  - 4 hardware threads per core
- peak fp32/fp64
  - 2.42/1.21 TFlop/s

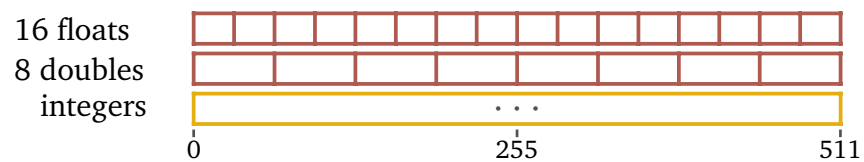


Mask Registers:



Eight 16-bit k registers per core.

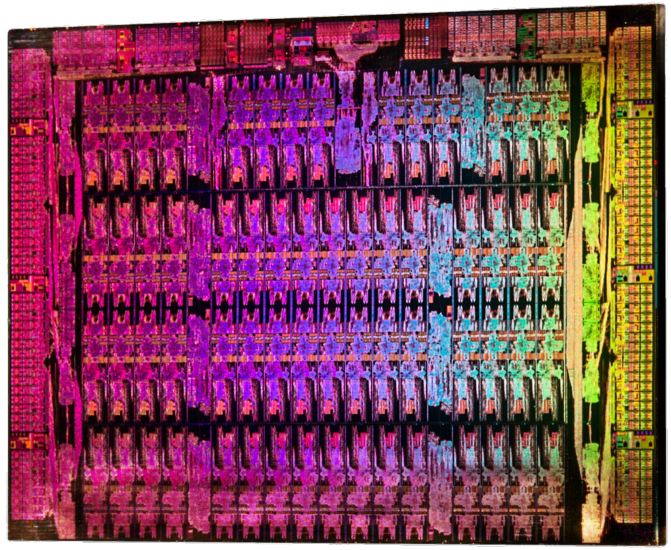
Vector Registers:



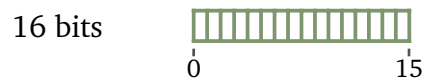
Thirty-two 512-bit zmm registers per core.

Intel® Xeon Phi™

- x86 based many-core processor
  - runs a Linux μOS
  - in-order execution
  - up to 61 cores at 1.238 GHz
  - core boost for 7120 series
  - theoretical bandwidth 352 GB/s
- parallelism through
  - large SIMD registers
  - 4 hardware threads per core
- peak fp32/fp64
  - 2.42/1.21 TFlop/s
- offload / native mode

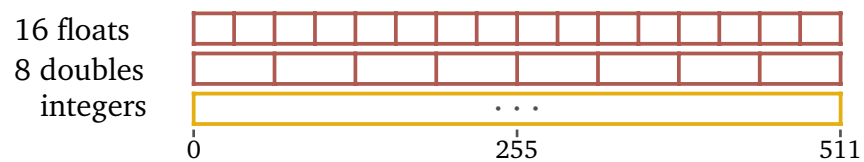


Mask Registers:



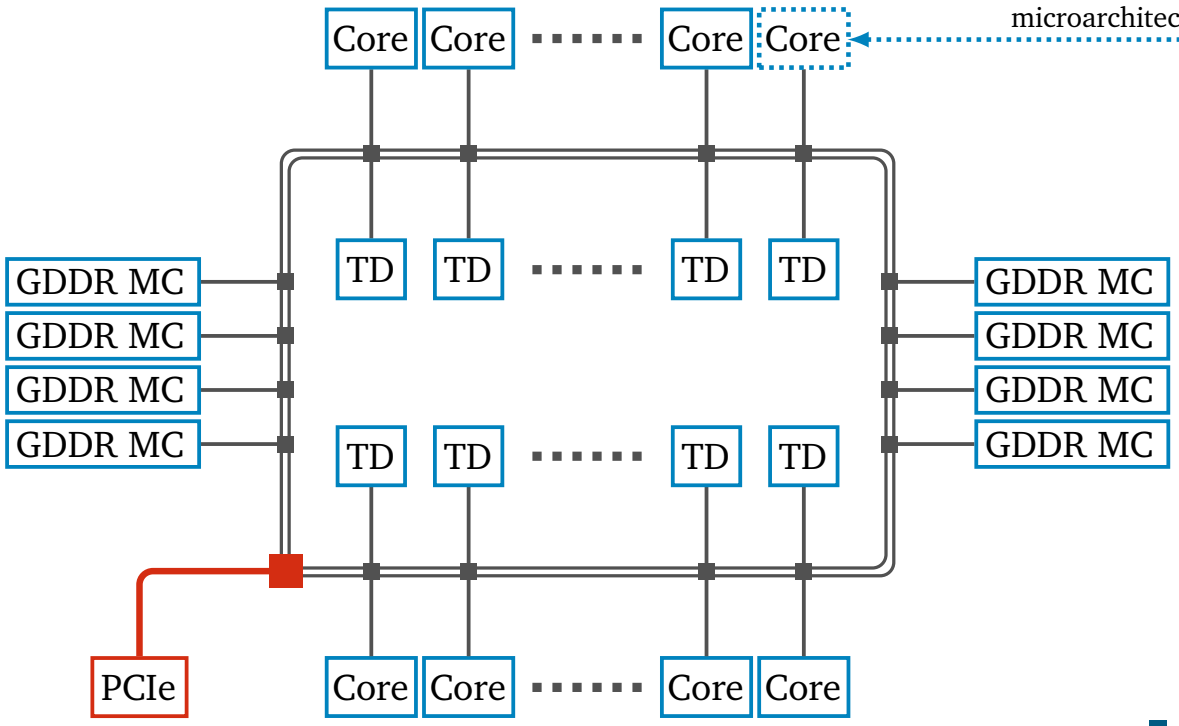
Eight 16-bit k registers per core.

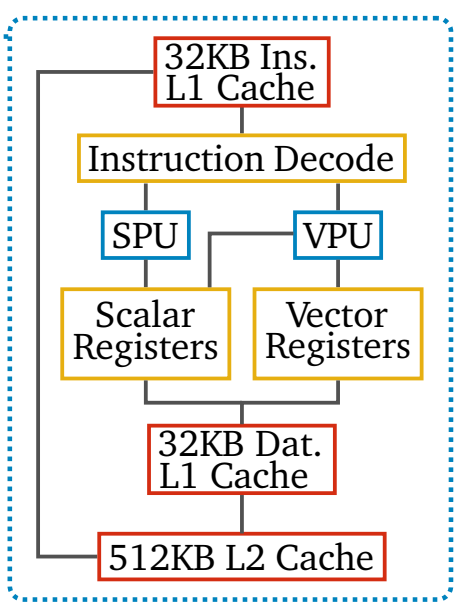
Vector Registers:



Thirty-two 512-bit zmm registers per core.

Knights Corner architecture





- Stream Triad (ECC, 4 threads/core)

7120P	149 GB/s
5110P	140 GB/s

- fully coherent cache
- two pipelines
  - scalar
  - vector + scalar

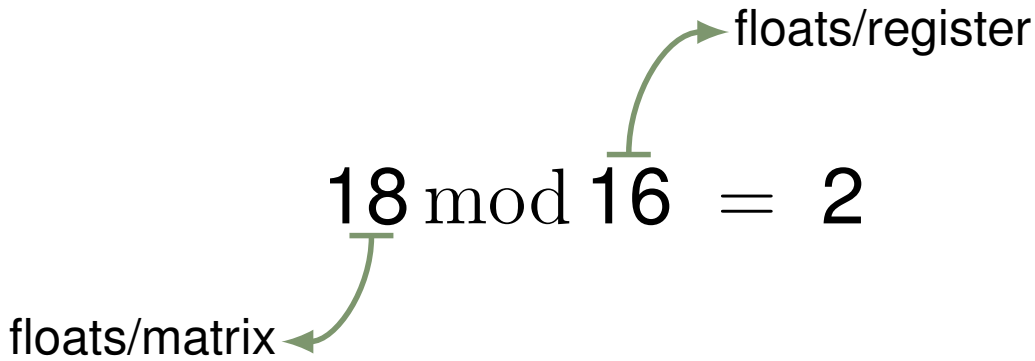
Vectorizing QCD

Problem:

$$18 \bmod 16 = 2$$

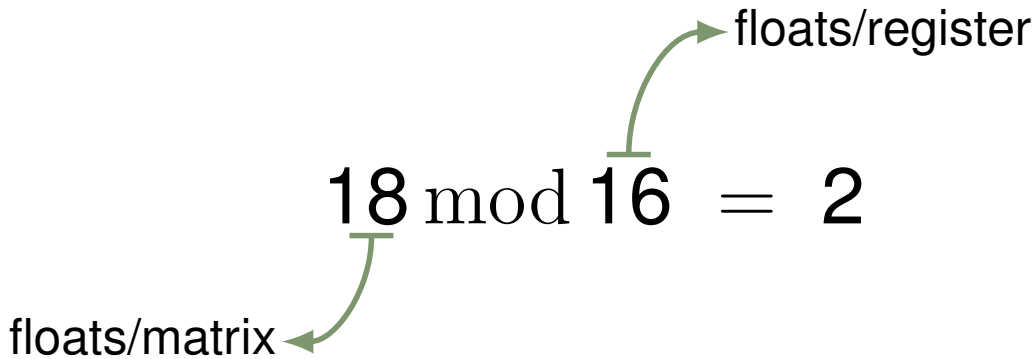
Vectorizing QCD

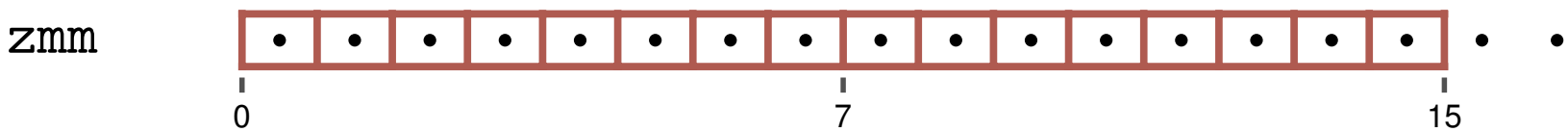
Problem:



Vectorizing QCD

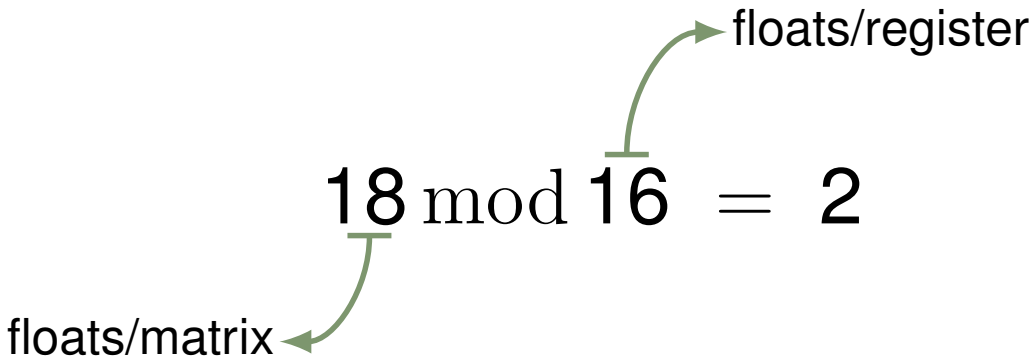
Problem:

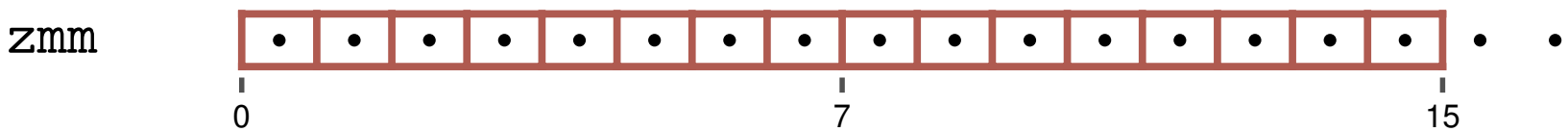




Vectorizing QCD

Problem:



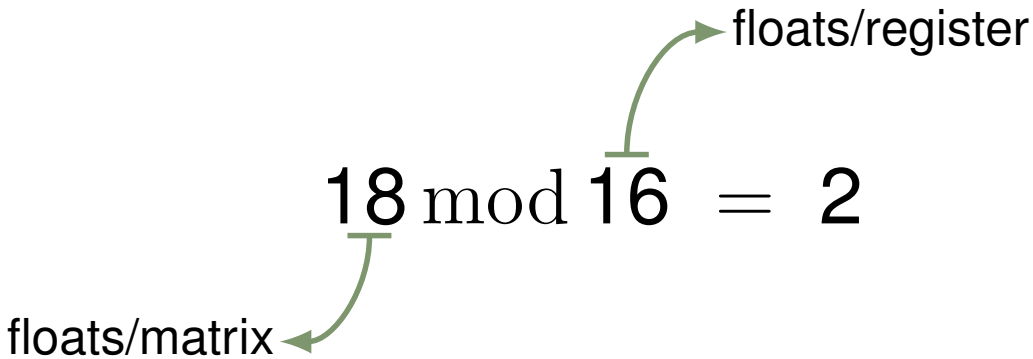


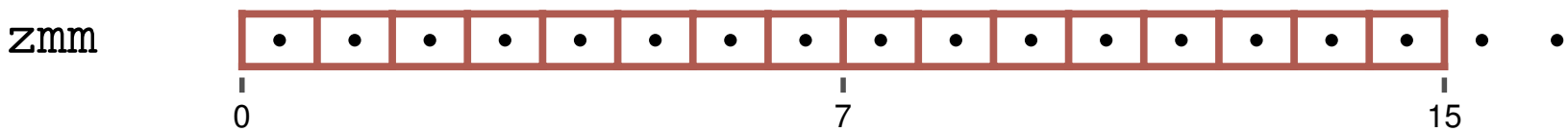
Solution:

mix color components of matrices

Vectorizing QCD

Problem:



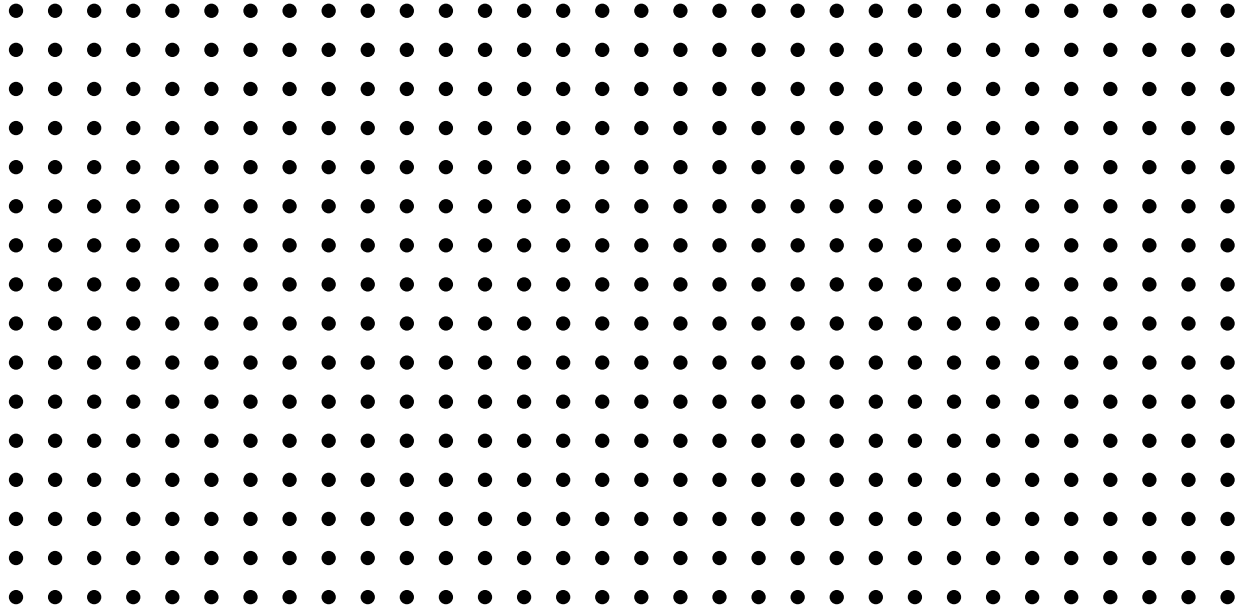


Solution:

mix color components of matrices

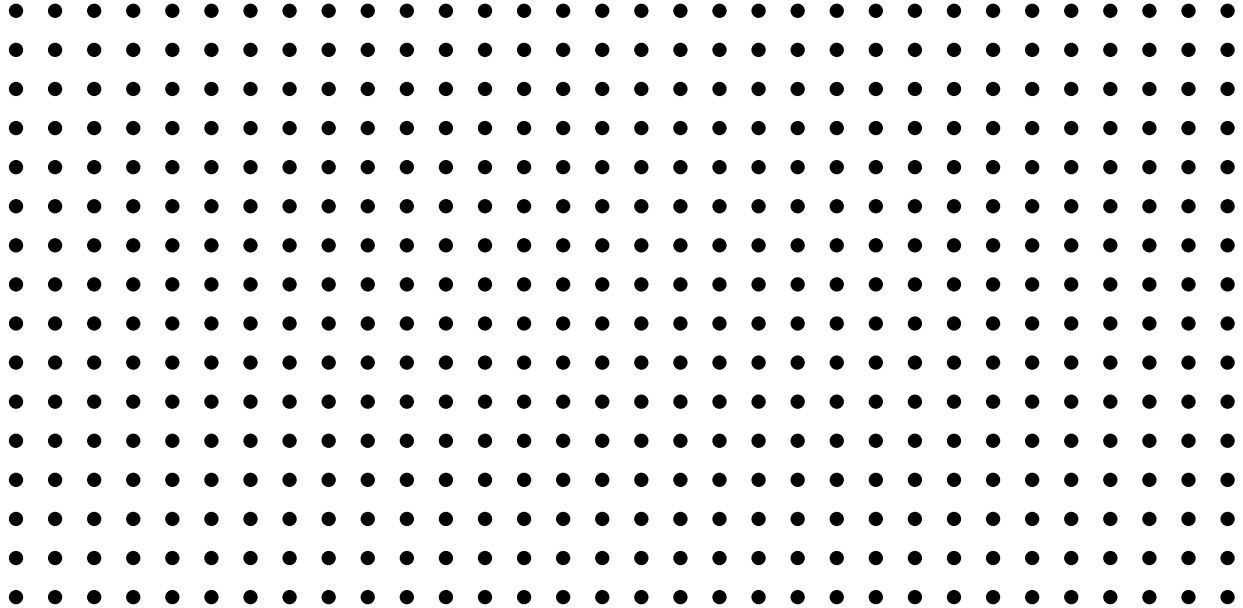
easiest single-precision approach
"16-fold site fusion"

Site fusion



Site fusion

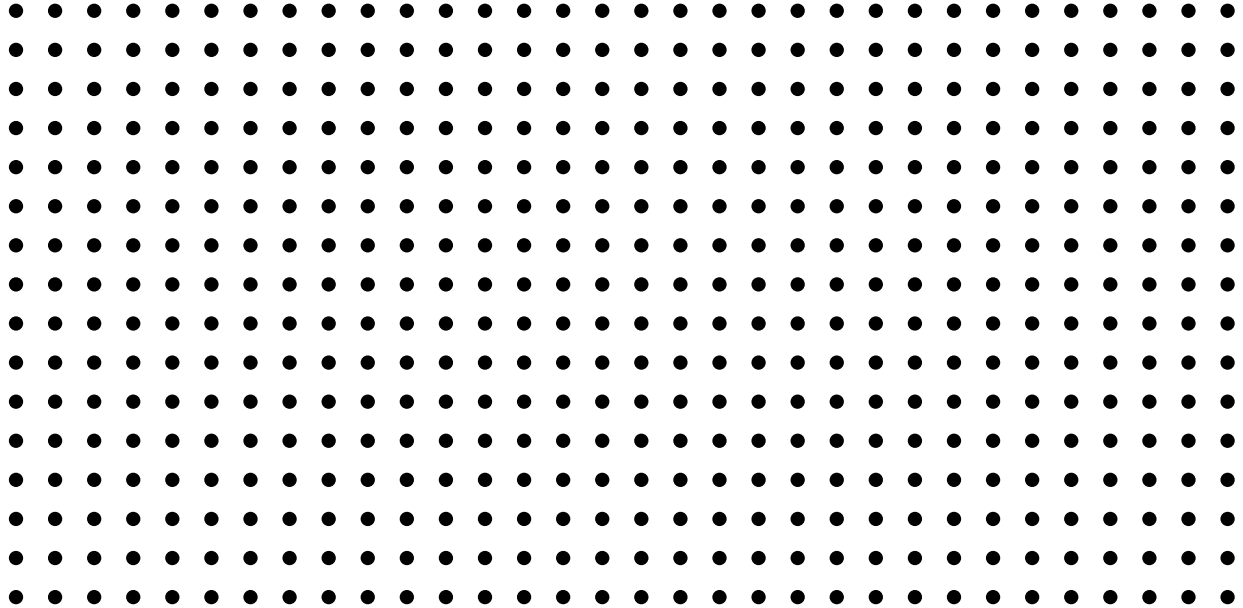
- even
- odd



- even-odd preconditioning
  - need to fuse sites of the same parity

Site fusion

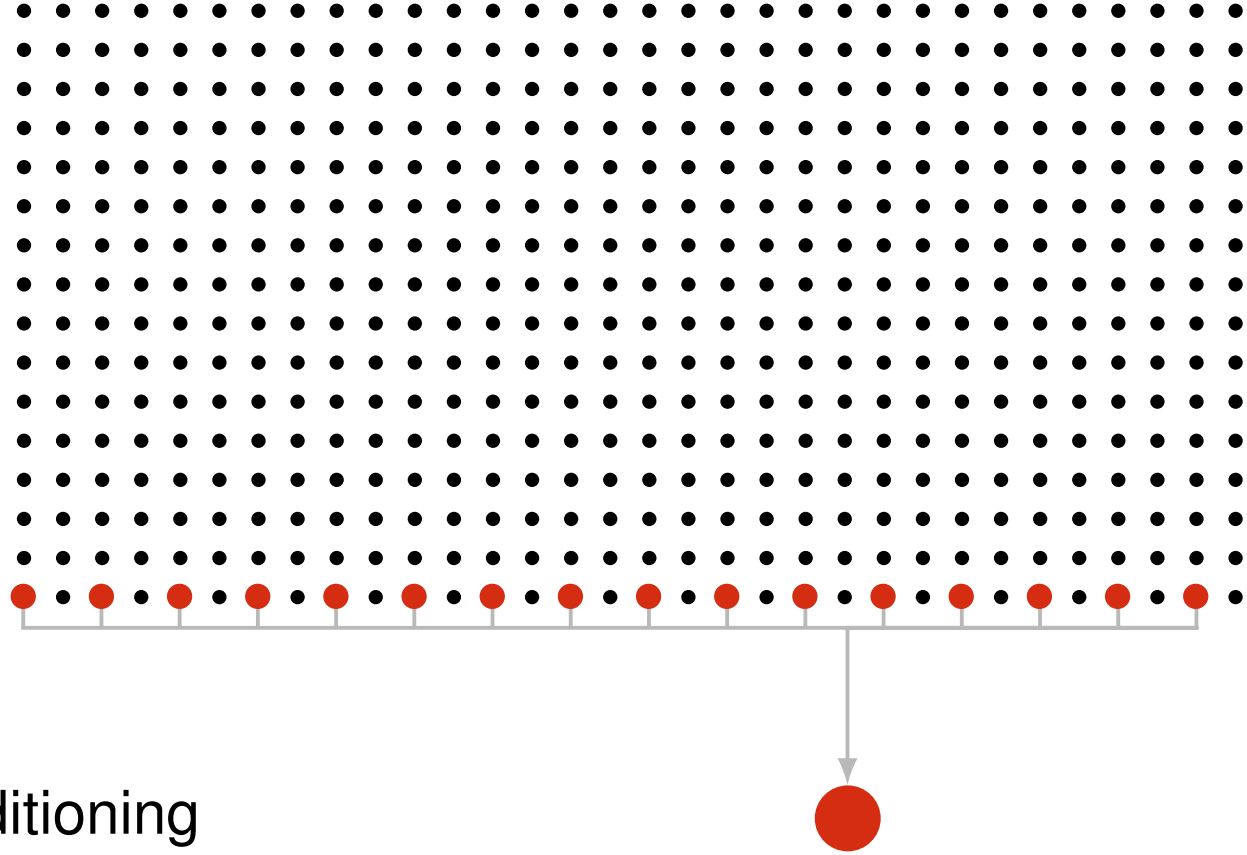
- even
- odd



- even-odd preconditioning
  - need to fuse sites of the same parity
- fuse 16 sites for single-precision

Site fusion

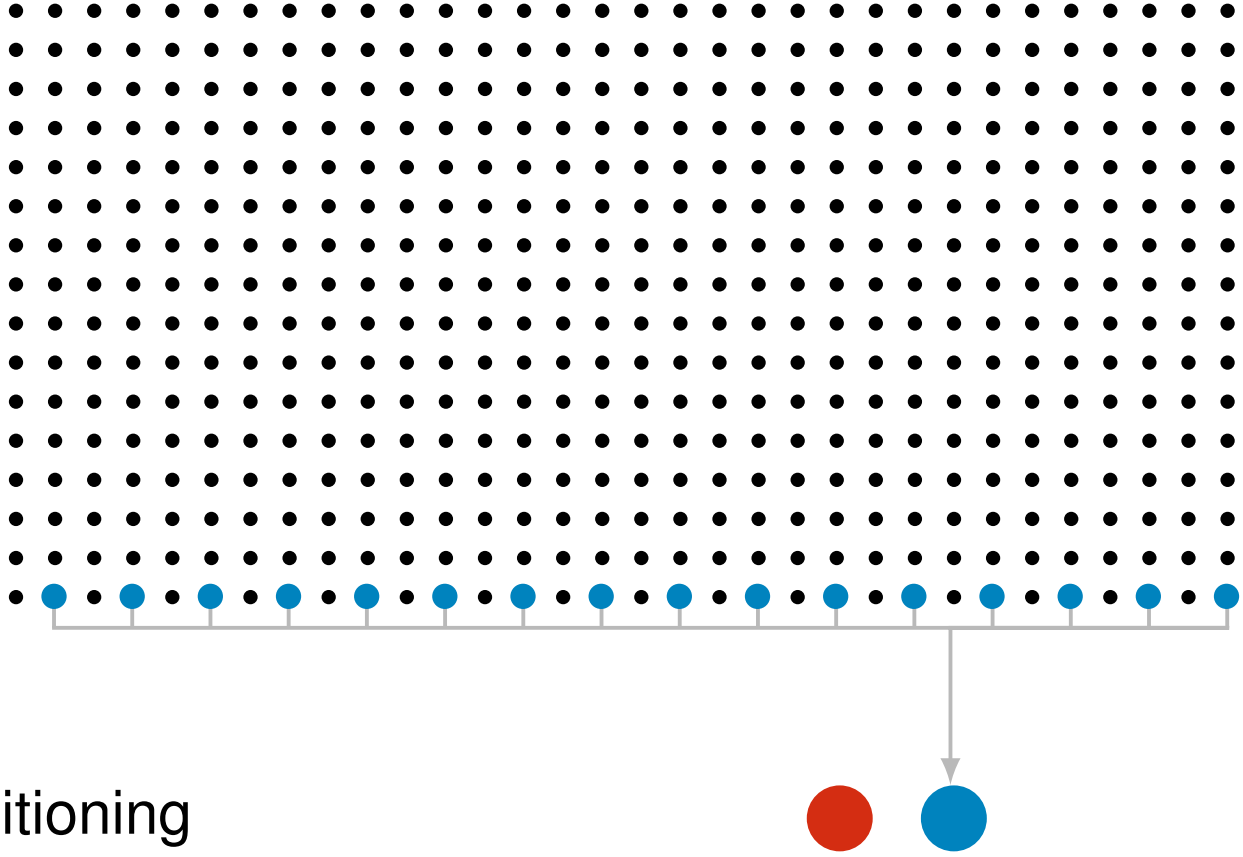
- even
- odd



- even-odd preconditioning
  - need to fuse sites of the same parity
- fuse 16 sites for single-precision

Site fusion

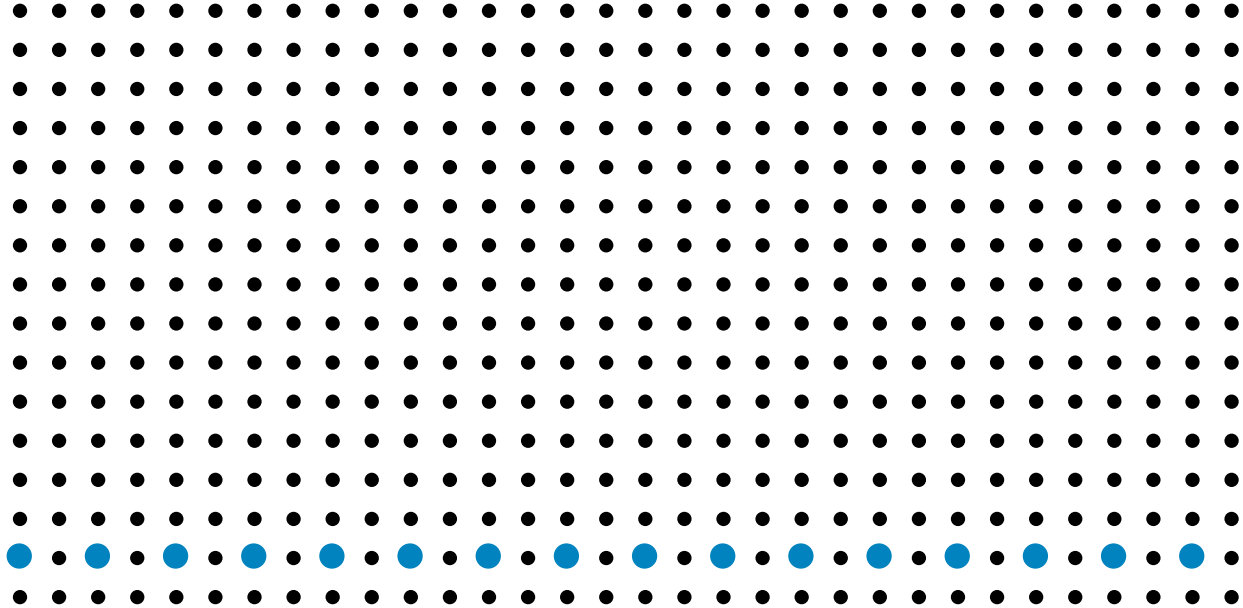
- even
- odd



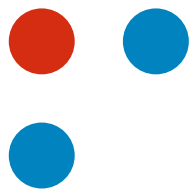
- even-odd preconditioning
  - need to fuse sites of the same parity
- fuse 16 sites for single-precision

Site fusion

- even
- odd



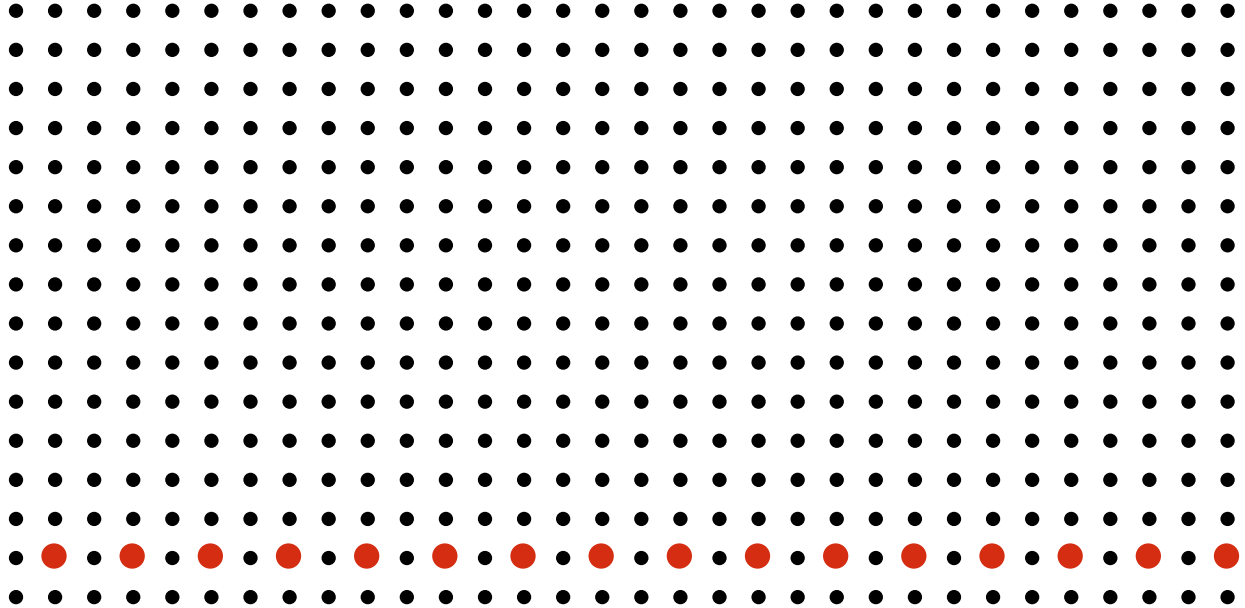
- even-odd preconditioning
  - need to fuse sites of the same parity



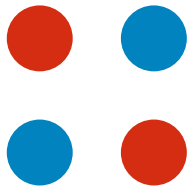
- fuse 16 sites for single-precision

Site fusion

- even
- odd



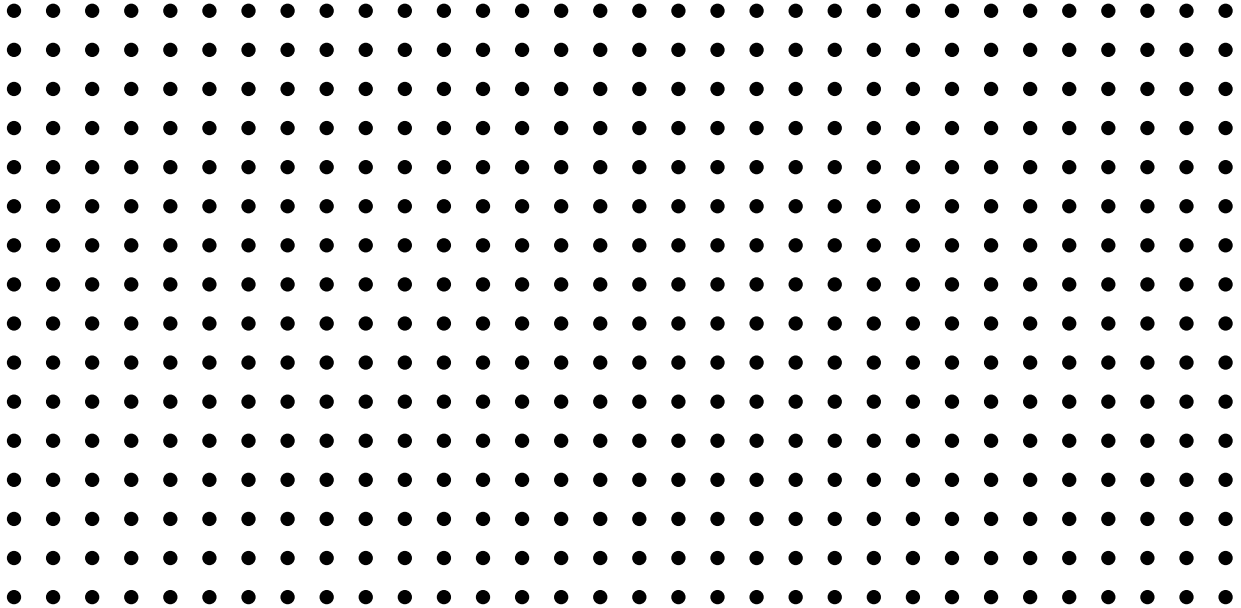
- even-odd preconditioning
  - need to fuse sites of the same parity



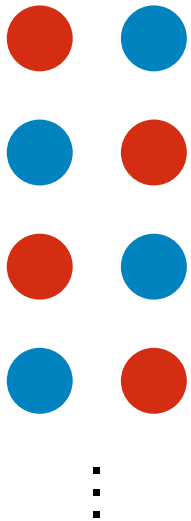
- fuse 16 sites for single-precision

Site fusion

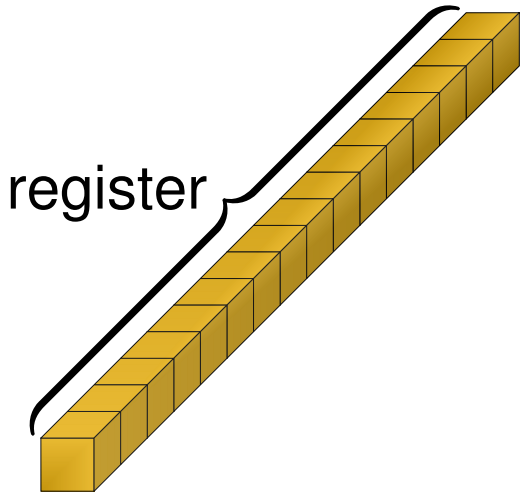
- even
- odd



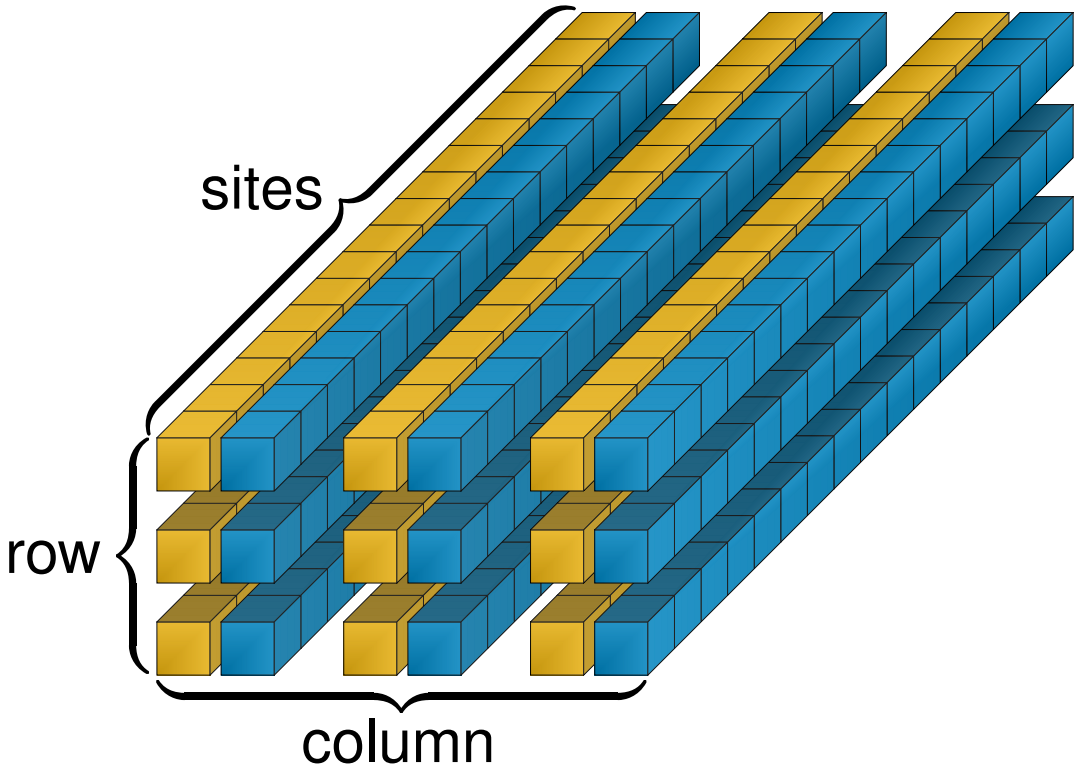
- even-odd preconditioning
  - need to fuse sites of the same parity
- fuse 16 sites for single-precision
  - $32^3 \times 8 \longrightarrow \mathbf{2} \times 32^2 \times 8$ (less indexing)
  - constraints on lattice size
  - high register pressure



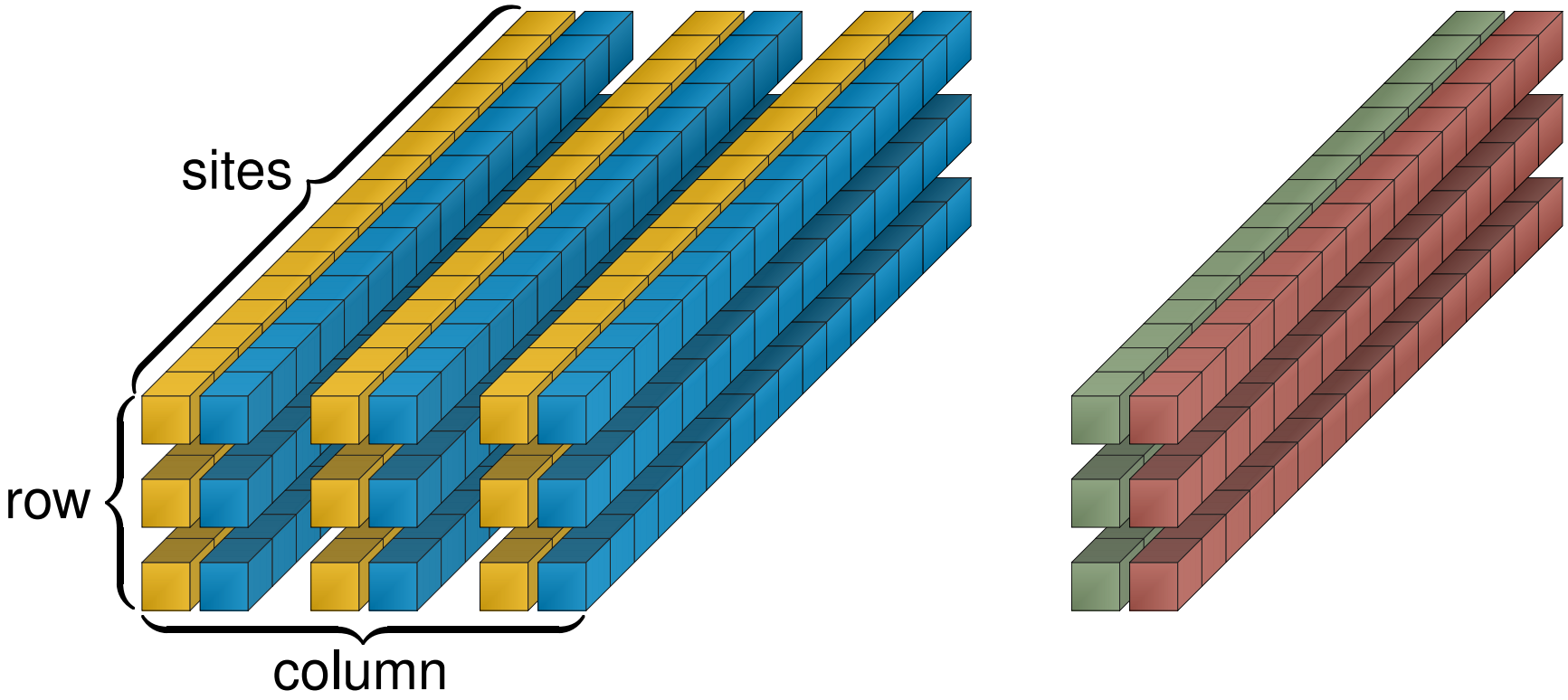
naive 16-fold site fusion



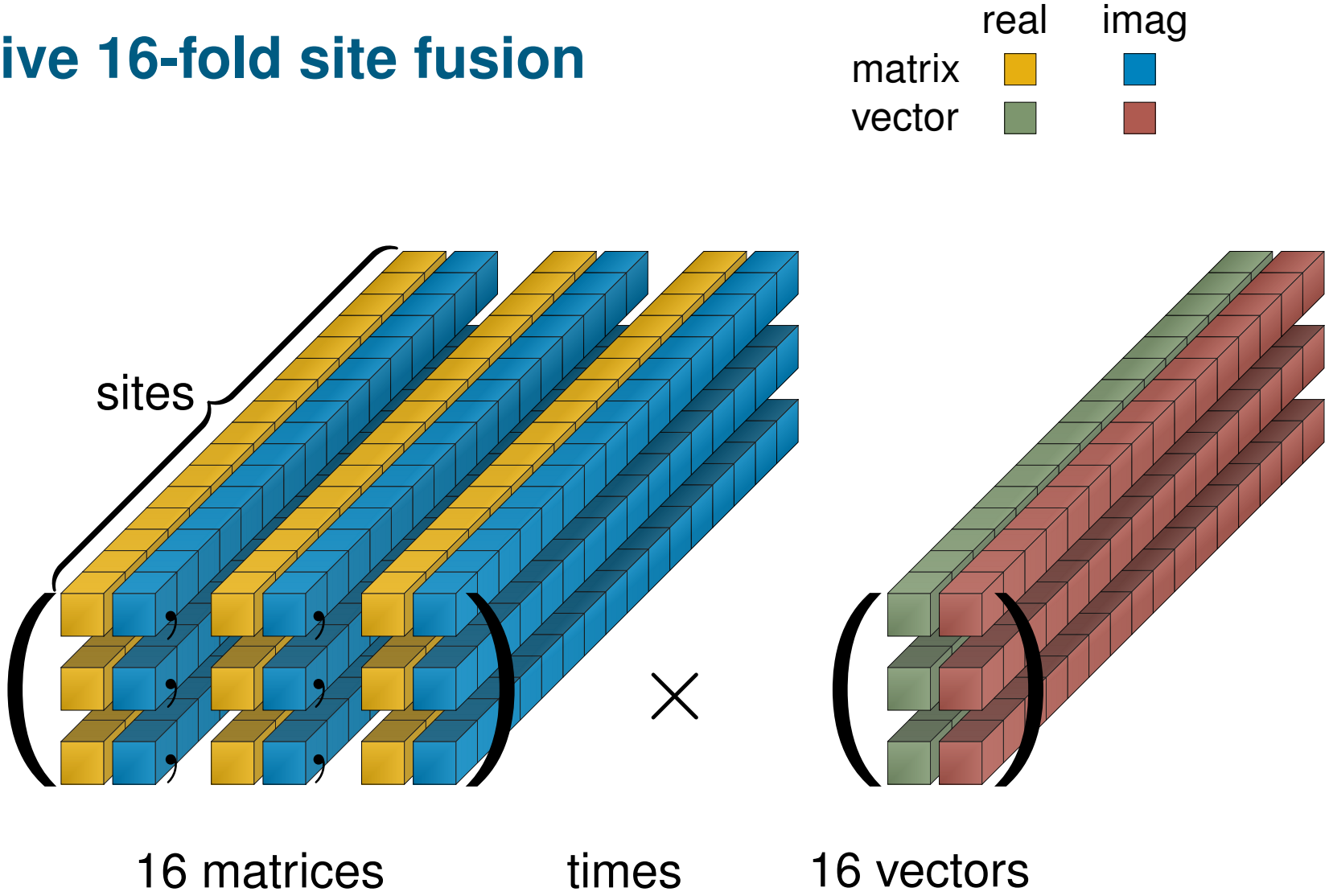
naive 16-fold site fusion



naive 16-fold site fusion



naive 16-fold site fusion



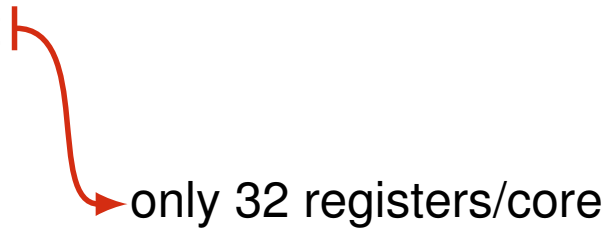
- very natural implementation $\longrightarrow$ use SIMD registers like scalars

Site fusion

- **16-fold**
  - easy to implement
  - very good performance for 1 right-hand side
  - high register pressure
    - 30 registers for a fused matrix-vector product
  - perfect arithmetic vectorization

Site fusion

- **16-fold**
  - easy to implement
  - very good performance for 1 right-hand side
  - high register pressure
    - 30 registers for a fused matrix-vector product
  - perfect arithmetic vectorization



Site fusion

- **16-fold**
  - easy to implement
  - very good performance for 1 right-hand side
  - high register pressure
    - 30 registers for a fused matrix-vector product
  - perfect arithmetic vectorization

only 32 registers/core

- **8-fold**
  - requires align/blend/mask instructions
    - some “padding” Flops
  - very good performance for 1 right-hand side
  - reduced register pressure
    - 16 registers for a fused matrix-vector product

we don’t count the $U(3)$ reconstruction and padding Flops for the performance!

Site fusion

- **16-fold**
  - easy to implement
  - very good performance for 1 right-hand side
  - high register pressure
    - 30 registers for a fused matrix-vector product
  - perfect arithmetic vectorization

only 32 registers/core

- **8-fold**
  - requires align/blend/mask instructions
    - some “padding” Flops
  - very good performance for 1 right-hand side
  - reduced register pressure
    - 16 registers for a fused matrix-vector product

55% faster for 4 right-hand sides compared to 16-fold

we don’t count the $U(3)$ reconstruction and padding Flops for the performance!

Prefetching

- in-order cores are very sensitive to cache misses

Prefetching

- in-order cores are very sensitive to cache misses
  - 1 thread/core

Prefetching

- in-order cores are very sensitive to cache misses
  - 1 thread/core $\rightarrow$ cache miss

Prefetching

- in-order cores are very sensitive to cache misses
  - 1 thread/core $\rightarrow$ cache miss $\rightarrow$ stall

Prefetching

- in-order cores are very sensitive to cache misses
  - 1 thread/core $\rightarrow$ cache miss $\rightarrow$ stall
  - need to use 4 threads per core

Prefetching

- in-order cores are very sensitive to cache misses
  - 1 thread/core $\rightarrow$ cache miss $\rightarrow$ stall
  - need to use 4 threads per core
- only L2 hardware prefetcher (no L1)
  - does a good job for a linear memory access

Prefetching

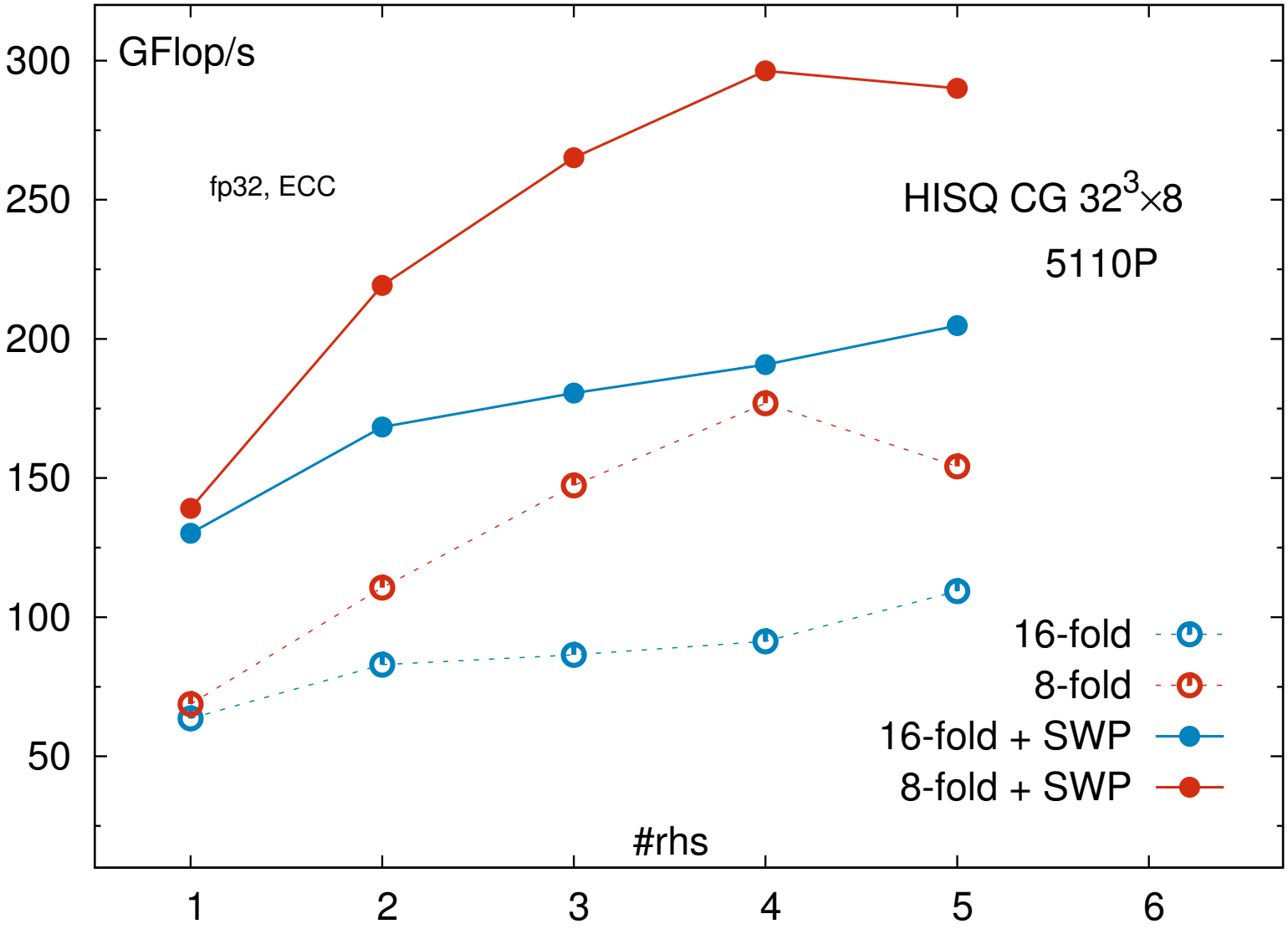
- in-order cores are very sensitive to cache misses
  - 1 thread/core $\rightarrow$ cache miss $\rightarrow$ stall
  - need to use 4 threads per core
- only L2 hardware prefetcher (no L1)
  - does a good job for a linear memory access
- access pattern of Dslash is too complicated for the compiler
  - insert software prefetches (SWP) by hand
  - hard part is to figure out the prefetch distance

Prefetching

- in-order cores are very sensitive to cache misses
  - 1 thread/core $\rightarrow$ cache miss $\rightarrow$ stall
  - need to use 4 threads per core
- only L2 hardware prefetcher (no L1)
  - does a good job for a linear memory access
- access pattern of Dslash is too complicated for the compiler
  - insert software prefetches (SWP) by hand
  - hard part is to figure out the prefetch distance

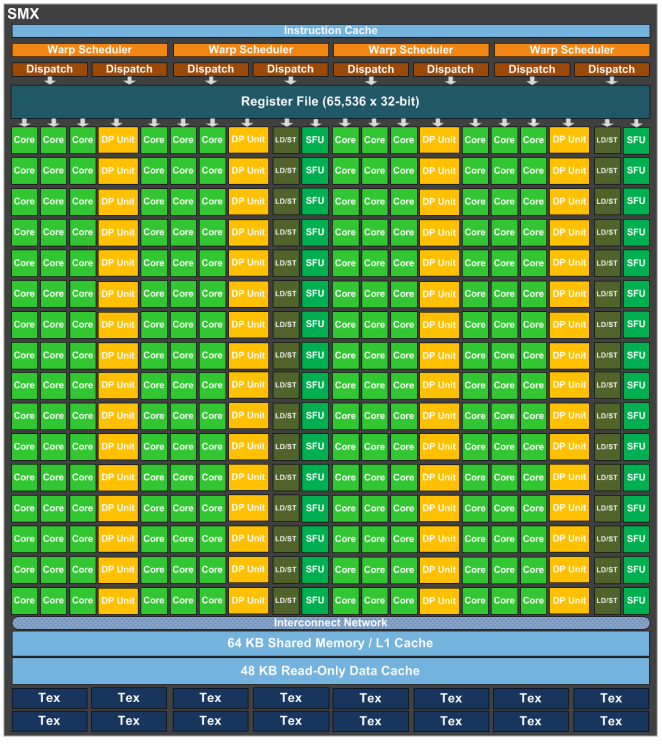
HISQ inverter runs 2$\times$ faster with software prefetches

Performance gains on Xeon Phi™

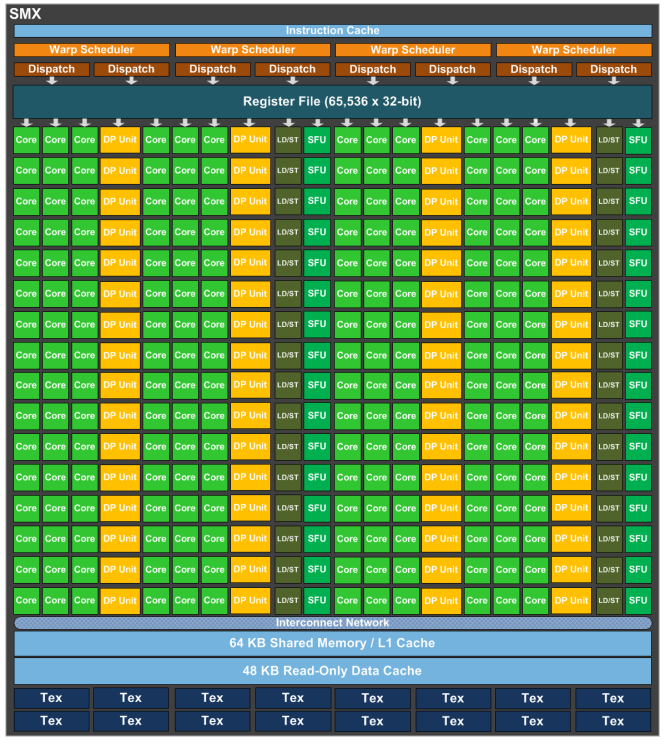


GPU implementation

- coalesced memory layout $\rightarrow$ SoA
- 1 threads calculates 1 lattice site
  - automatic tuning of block size

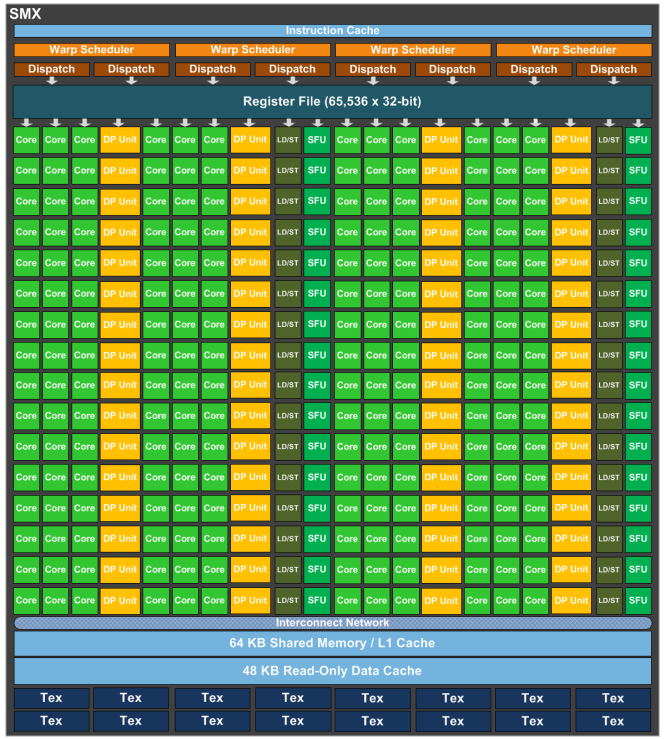


GPU implementation



- coalesced memory layout $\rightarrow$ SoA
- 1 threads calculates 1 lattice site
  - automatic tuning of block size
- memory access
  - exploit texture cache for gauge links
  - load vectors through the standard load path (L1)

GPU implementation

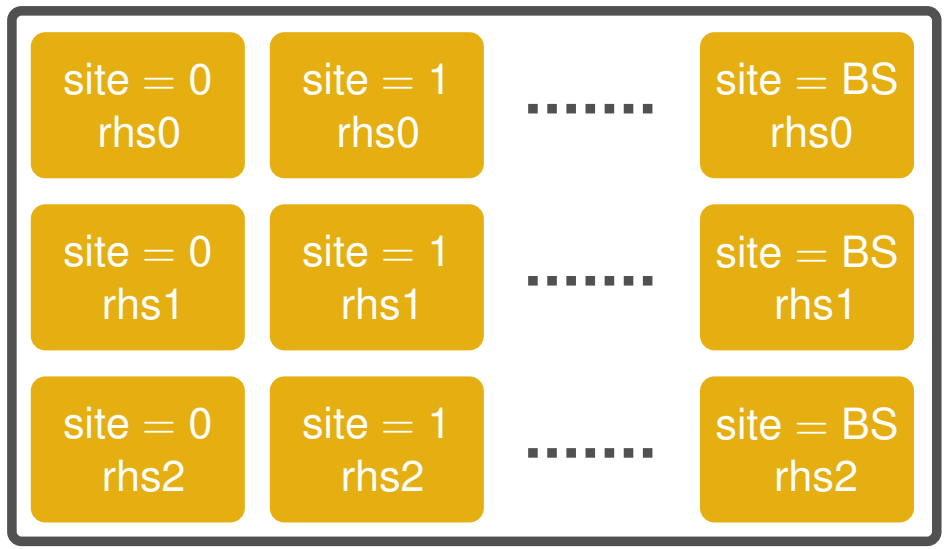


- coalesced memory layout $\rightarrow$ SoA
- 1 threads calculates 1 lattice site
  - automatic tuning of block size
- memory access
  - exploit texture cache for gauge links
  - load vectors through the standard load path (L1)
- multiple right-hand sides optimizations
  - register blocking

$$1 \text{ thread} = 1 \text{ site}(\texttt{\#n rhs})$$

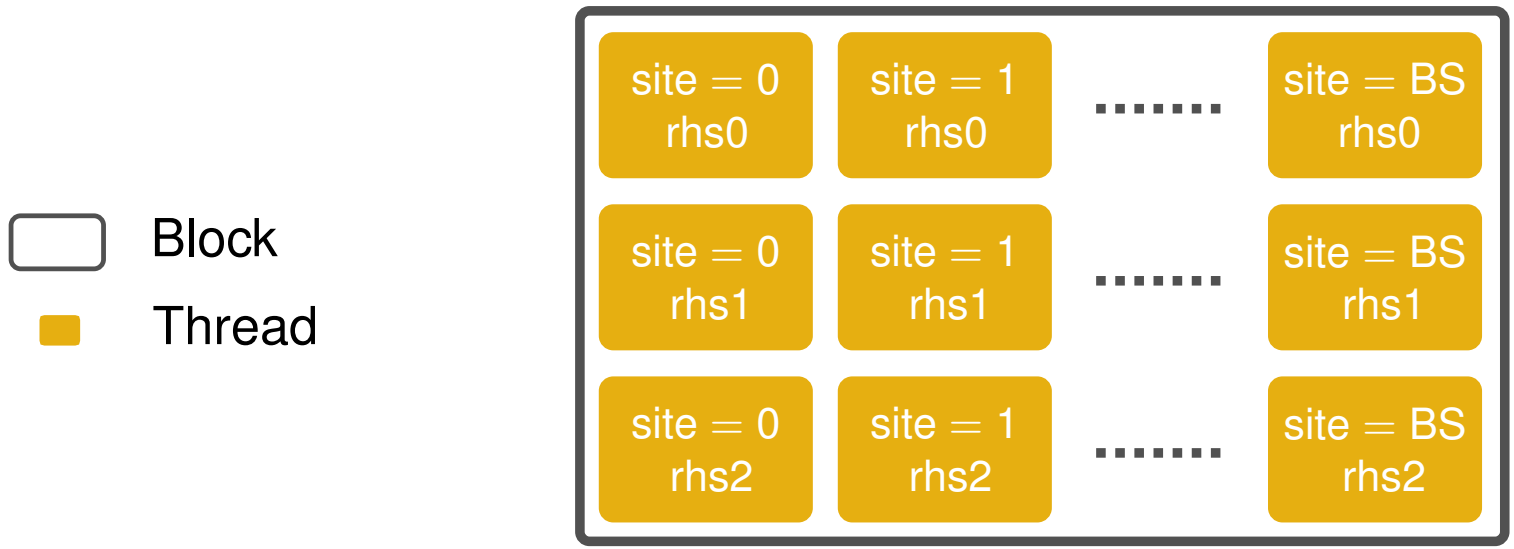
GPU implementation

- multiple right-hand sides optimizations
  - texture cache blocking



GPU implementation

- multiple right-hand sides optimizations
  - texture cache blocking



explore best possible combination
of cache and register blocking

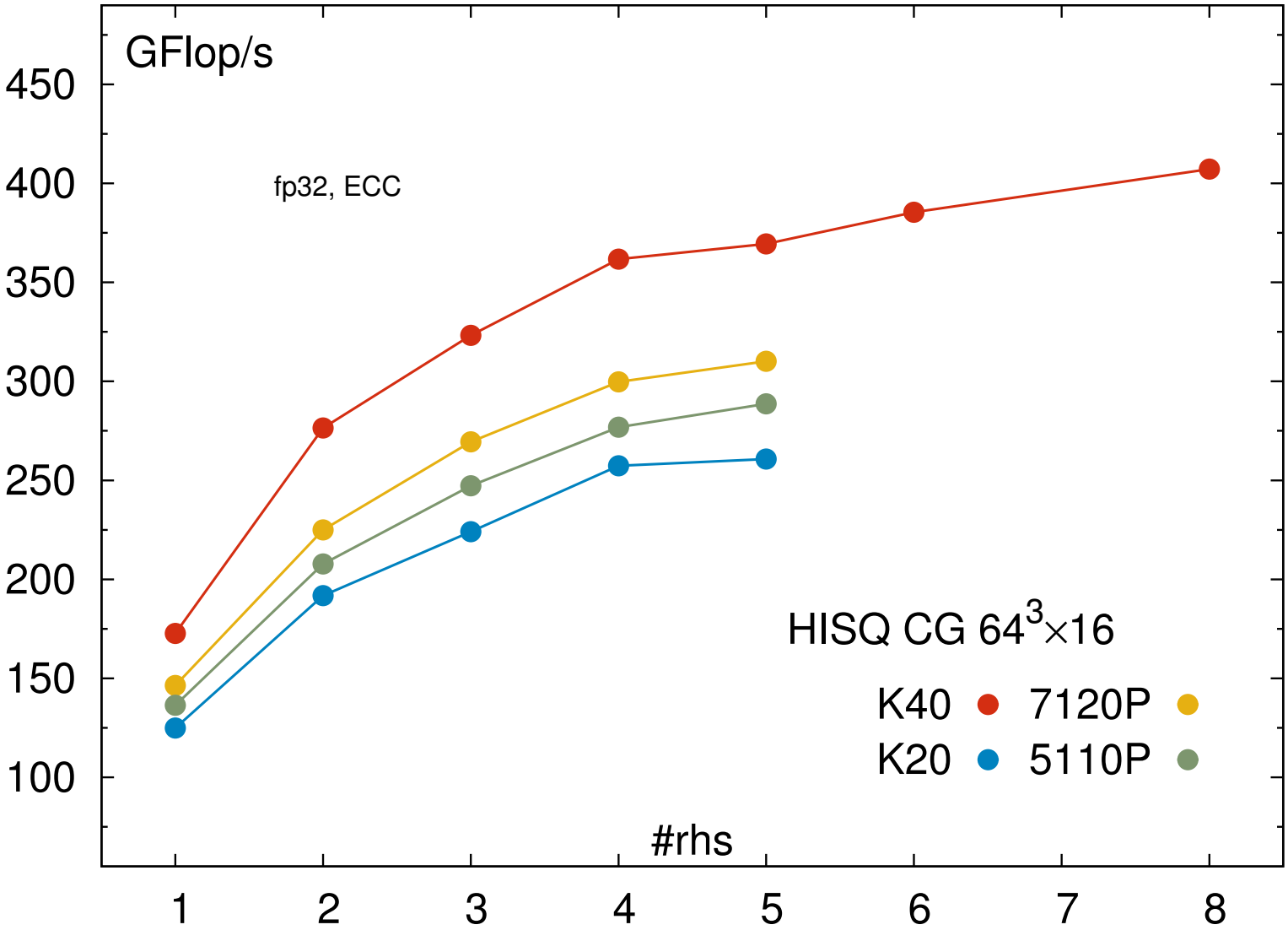
GPU implementation

- use GPU boost on K40 $\longrightarrow$ 875 MHz
  - memory-bound problems stay well within thermal limits

GPU talk by Mathias Wagner

GTC 2014 $\longrightarrow$ `http://goo.gl/AglZBM`

Comparison



K40[1]:

4.29 TFlop/s
288 GB/s

K20[1]:

3.52 TFlop/s
208 GB/s

7120P[2]:

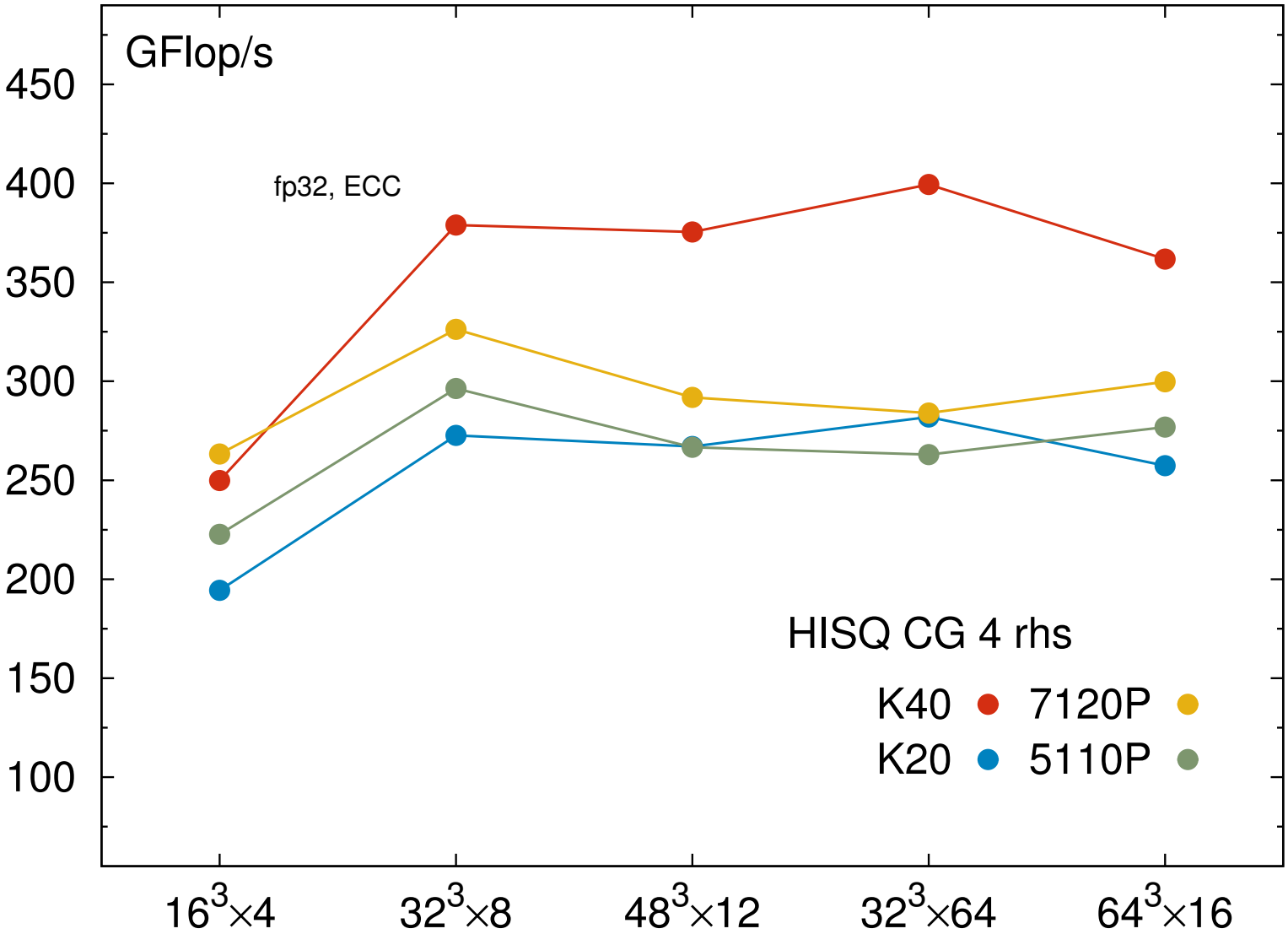
2.42 TFlop/s
149 GB/s

5110P[2]:

2.02 TFlop/s
140 GB/s

[1]Theoretical bandwidth, ECC off
[2]Stream Triad, 4 threads/core, ECC on

Comparison



K40[1]:

4.29 TFlop/s
288 GB/s

K20[1]:

3.52 TFlop/s
208 GB/s

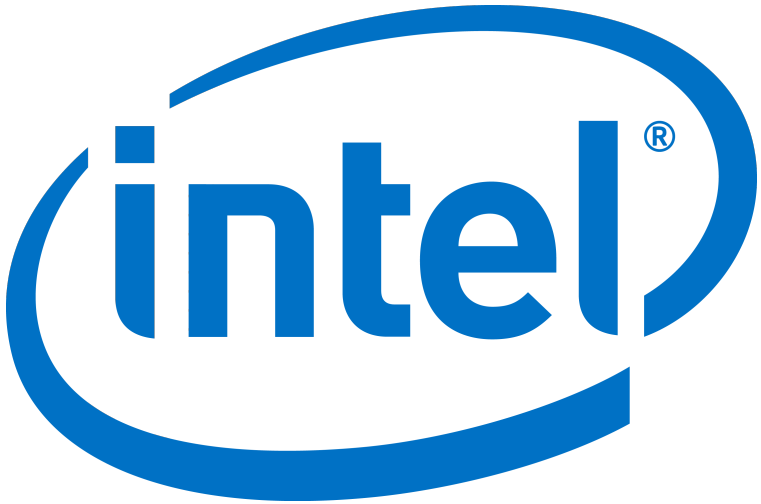
7120P[2]:

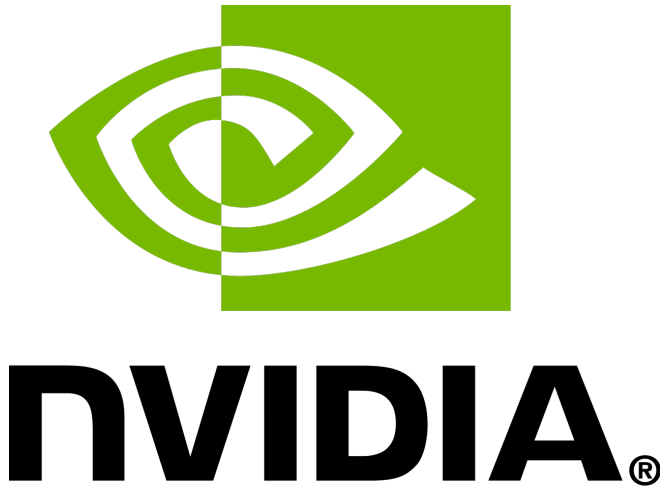
2.42 TFlop/s
149 GB/s

5110P[2]:

2.02 TFlop/s
140 GB/s

[1]Theoretical bandwidth, ECC off
[2]Stream Triad, 4 threads/core, ECC on





Thank you for your attention!

Things to improve / test on the Xeon Phi™

- divide lattice into L2 friendly blocks
  - let 4 threads work as a team in one block
  - synchronization at the end of each block

better cache locality

less TLB pressure

- run Linear Algebra with less than 4 threads per core?
  - better Stream Triad bandwidth for 1 thread/core

Intrinsics

Definition

Intrinsics are inlined assembly-coded `C` functions, which can be optimized by the compiler. They do not require explicit instruction or register scheduling through the programmer as in `asm`.

Intrinsics

Definition

Intrinsics are inlined assembly-coded `C` functions, which can be optimized by the compiler. They do not require explicit instruction or register scheduling through the programmer as in `asm`.

- full control over SIMD instructions
- MIC Intrinsics are very similar to ISA on Intel® CPUs
  - mostly easier on the Xeon Phi™

Intrinsics

Definition

Intrinsics are inlined assembly-coded `C` functions, which can be optimized by the compiler. They do not require explicit instruction or register scheduling through the programmer as in `asm`.

- full control over SIMD instructions
- MIC Intrinsics are very similar to ISA on Intel® CPUs
  - mostly easier on the Xeon Phi™