S9330: Lattice QCD with Tensor Cores

Kate Clark[3], Chulwoo Jung[2], Robert Mawhinney[1], and Jiqun Tu[1]*

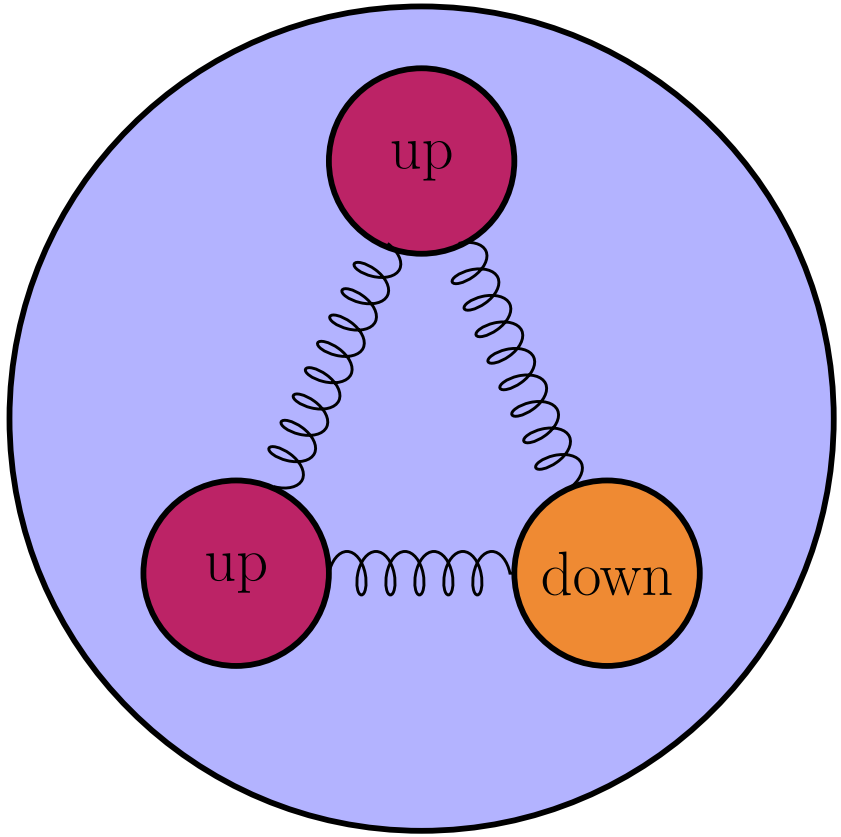
[1] Columbia University
[2] Brookhaven National Labratory
[3] Nvidia Corporation

GPU Technology Conference, Silicon Valley, March 2019

*speaker

Some Background

What's inside a proton?



$$m_u = \quad 2.2 \text{ MeV}$$
$$m_d = \quad 4.7 \text{ MeV}$$
$$m_p = 938.3 \text{ MeV}$$

Quantum Chromodynamics (QCD)

- QCD is an $\mathrm{SU}(3)$ local gauge theory with 6 flavors of dynamic quarks.

Quantum Chromodynamics (QCD)

- QCD is an ~~$\mathrm{SU}(3)$ local gauge~~ theory with ~~6 flavors of dynamic~~ quarks.

Quantum Chromodynamics (QCD)

- QCD is an ~~$\mathrm{SU}(3)$ local gauge~~ theory with ~~6 flavors of dynamic~~ quarks.
- Shows *confinement*: not possible to extract individual quarks from their bound states,

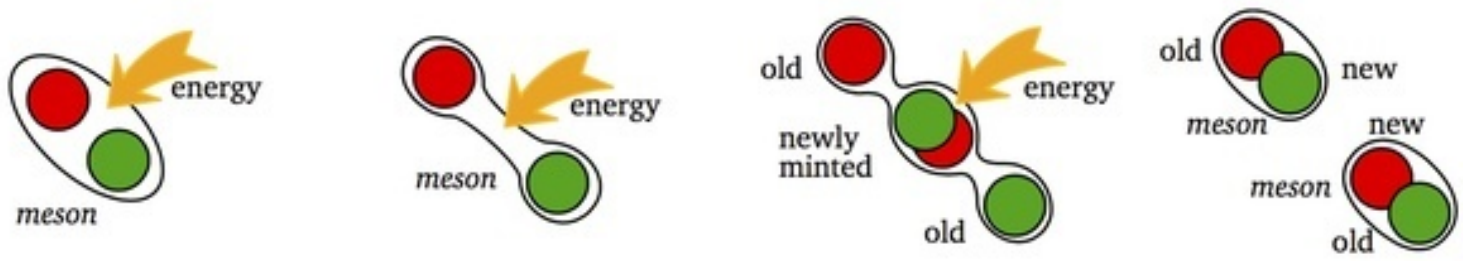


Figure –1.5: Inseparability of quarks and antiquarks in spite of investing ever more energy

Quantum Chromodynamics (QCD)

- QCD is an ~~$\mathrm{SU}(3)$ local gauge~~ theory with ~~6 flavors of dynamic~~ quarks.
- Shows *confinement*: not possible to extract individual quarks from their bound states,

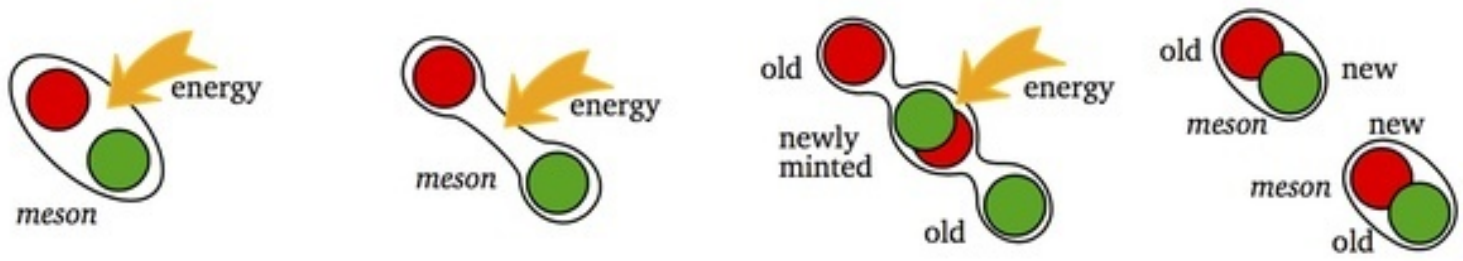
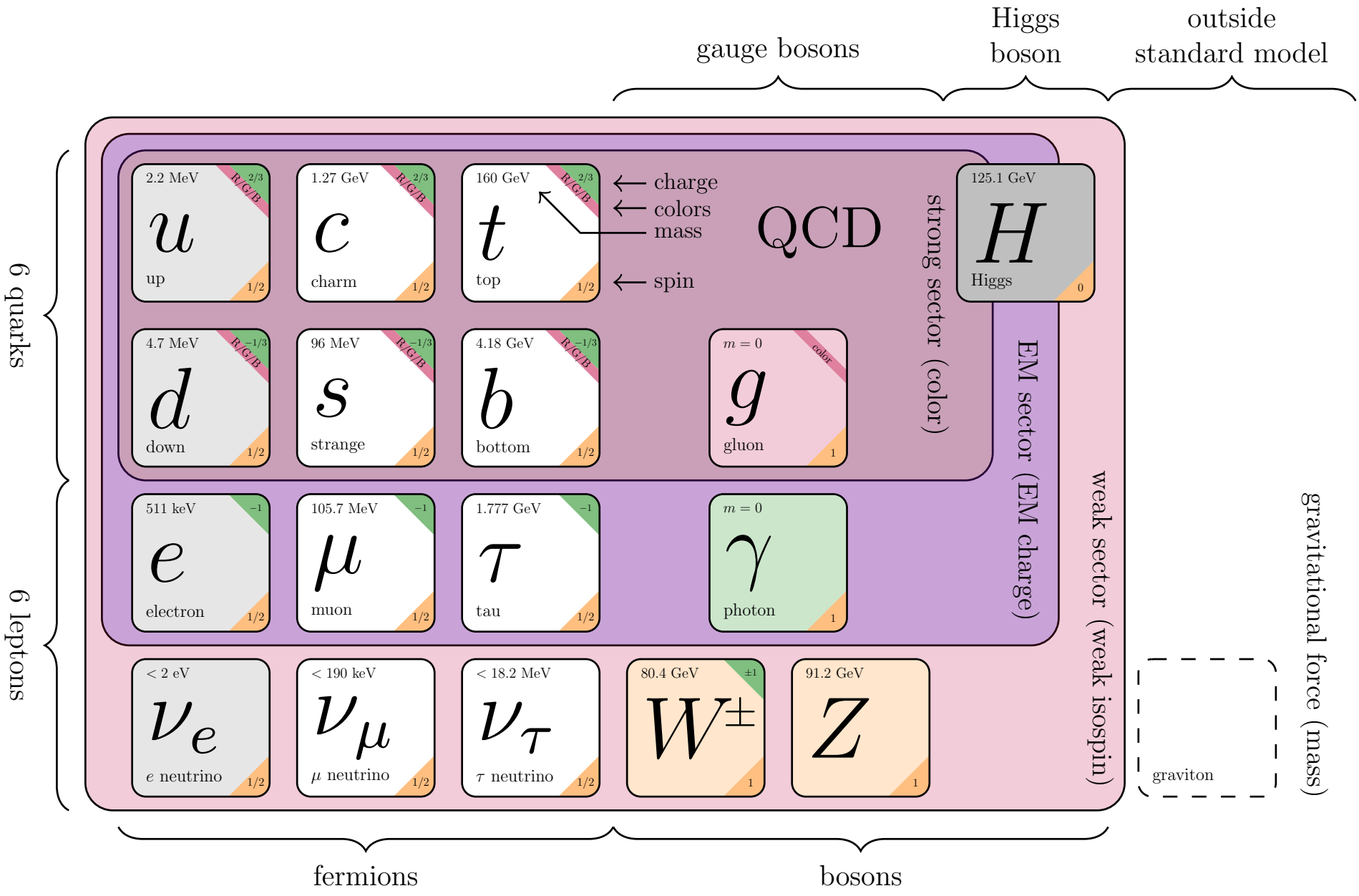
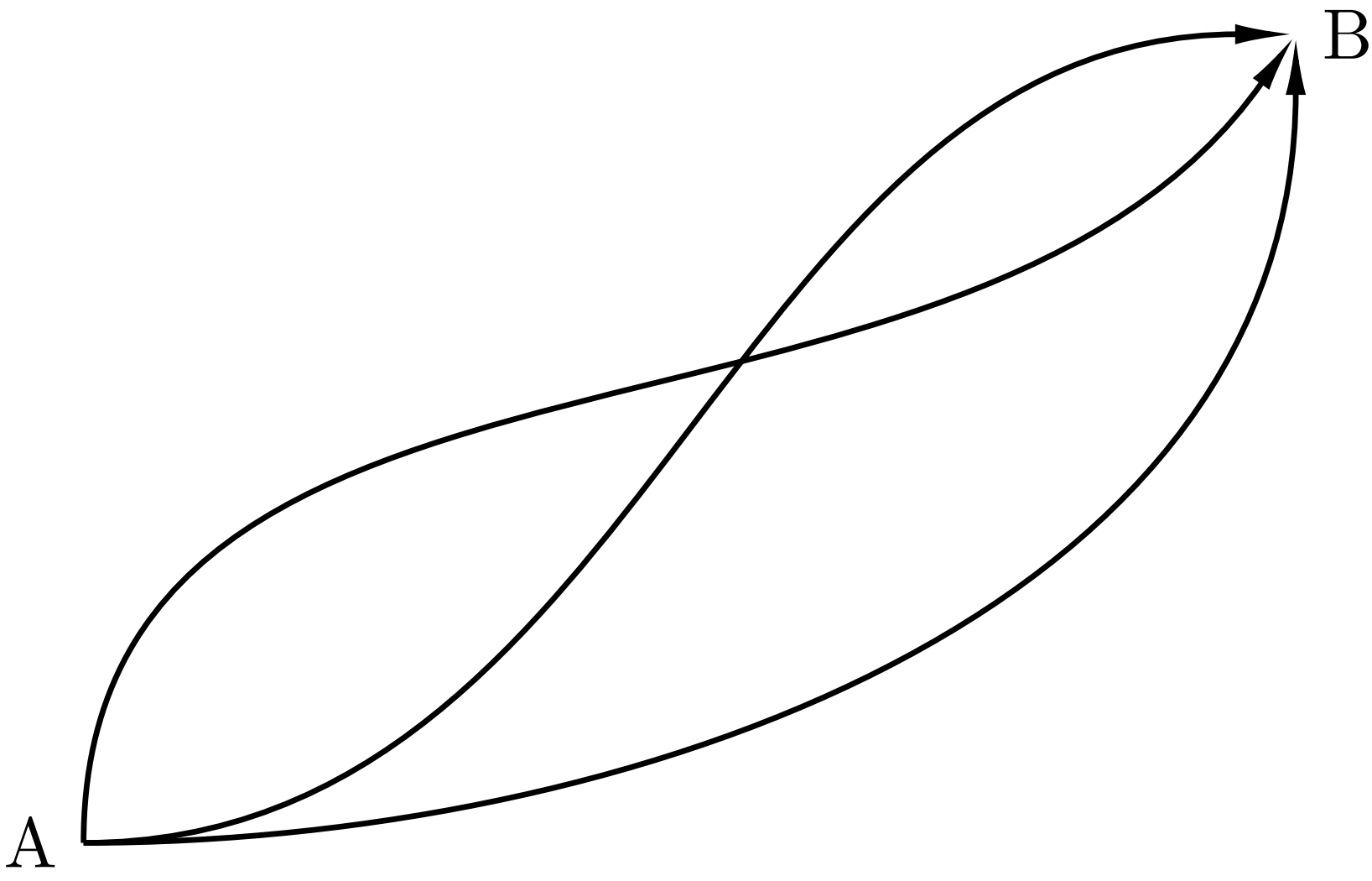


Figure –1.5: Inseparability of quarks and antiquarks in spite of investing ever more energy

- Non-perturbative: need numerical simulations to calculate measurable quantities from the theory to be measured in real world laboratories.

The Standard Model of Particles




B
A

- Path Integral formulation

$$\langle \mathcal{O} \rangle = \frac{\int [d\bar{\psi}][d\psi][dU]\mathcal{O}[U,\bar{\psi},\psi]e^{-S[U,\bar{\psi},\psi]}}{\int [d\bar{\psi}][d\psi][dU]e^{-S[U,\bar{\psi},\psi]}}$$

$$S[U,\bar{\psi},\psi] = (\text{gauge action}) + \bar{\psi}D[U]\psi$$

Path Integral: Monte Carlo

- Path Integral formulation

$$\langle \mathcal{O} \rangle = \frac{\int [d\bar{\psi}][d\psi][dU]\mathcal{O}[U,\bar{\psi},\psi]e^{-S[U,\bar{\psi},\psi]}}{\int [d\bar{\psi}][d\psi][dU]e^{-S[U,\bar{\psi},\psi]}}$$

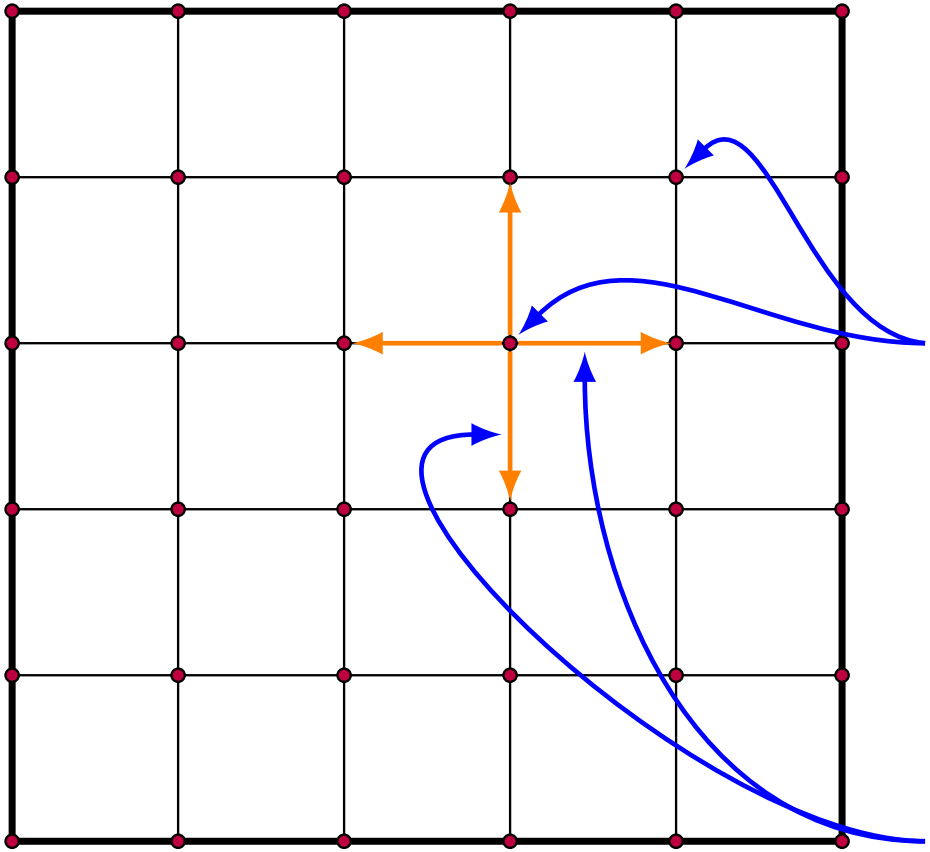
$$S[U,\bar{\psi},\psi] = (\text{gauge action}) + \bar{\psi}D[U]\psi$$

- Monte Carlo: Draw samples of U (gauge field configurations) according to the weight e^{-S} and measure some observable $\mathcal{O}$ on these samples.

$$\langle \mathcal{O} \rangle \simeq \frac{1}{n}\sum_i \mathcal{O}_i$$

From theory to simulation

discretize the theory and put it onto a 4 dimensional *lattice*



quark (fermion) fields ψ

gauge fields U

Quarks cause (at least) two problems. :(

1. They are anti-commutating fermion fields/numbers,

$$\psi(x)\psi(y) = -\psi(y)\psi(x),$$

how to simulate anti-commutating numbers on a traditional computer?

Quarks cause (at least) two problems. :(

1. They are anti-commutating fermion fields/numbers,

$$\psi(x)\psi(y) = -\psi(y)\psi(x),$$

   how to simulate anti-commutating numbers on a traditional computer?

2. Upon discretization they break chiral symmetry (the doubler problem): this symmetry says the nature uses left and right hand equally frequently.

Solution to the first problem:

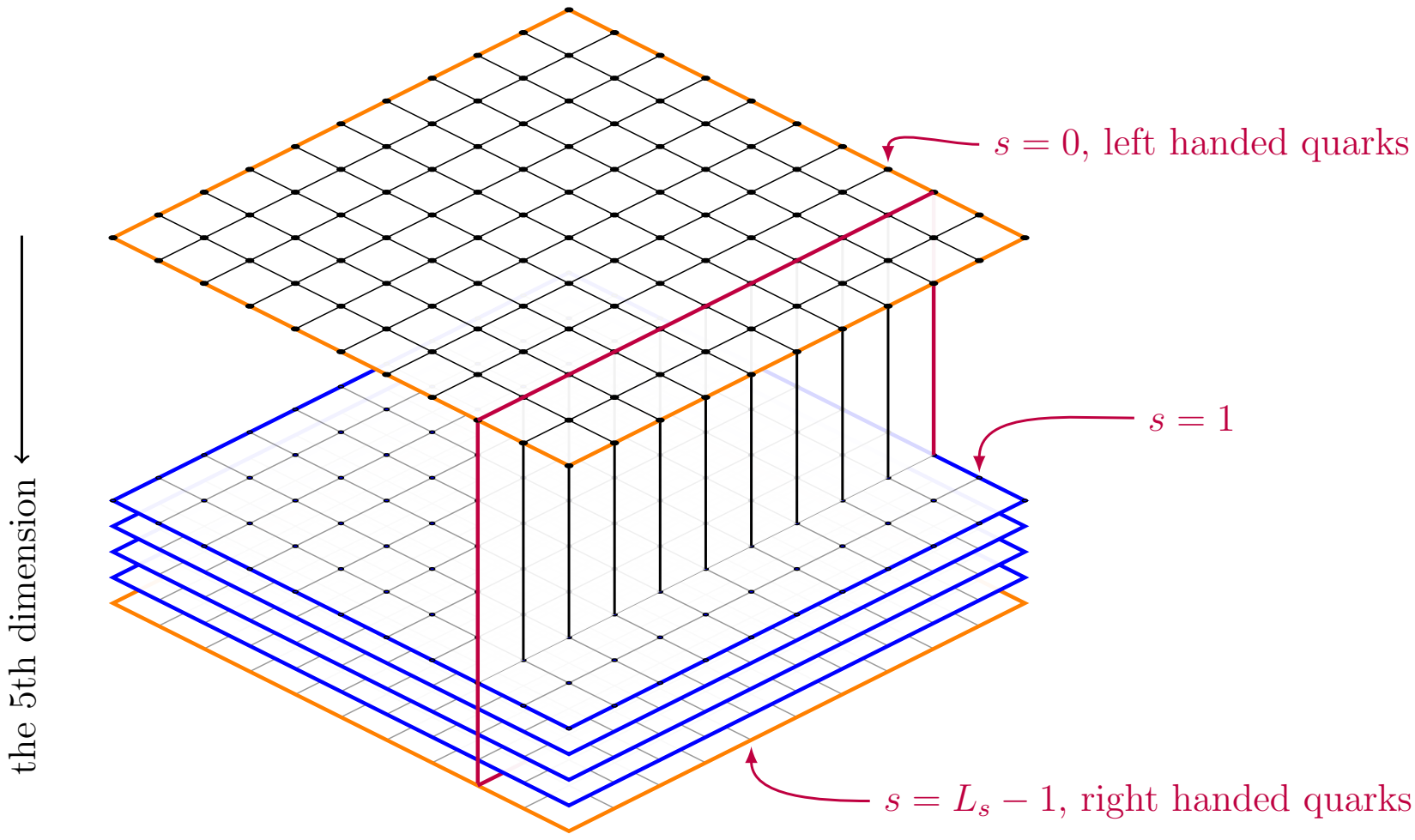
$$\int [d\bar{\psi}][d\psi] e^{-\bar{\psi} D[U] \psi} = \det D[U] = \int [d\phi^\dagger][d\phi] e^{-\phi^\dagger D[U]^{-1} \phi}$$

The anti-commuting grassmann variables ψ's are replaced with the usual commuting complex number ϕ's, at the expense of inverting the Dirac matrix $D[U]$. Typically we want to solve Dirac equations of the form

$$Dx = y, x = D^{-1} y$$

From theory to simulation

One solution to the second problem: we use the formulation of domain wall fermion (DWF) and add a 5th dimension



From theory to simulation

- Our numerical problem is as the following: Need to solve for a large sparse matrix of size $\sim 10^{10}$ by $\sim 10^{10}$ that spans over several different dimensions:

From theory to simulation

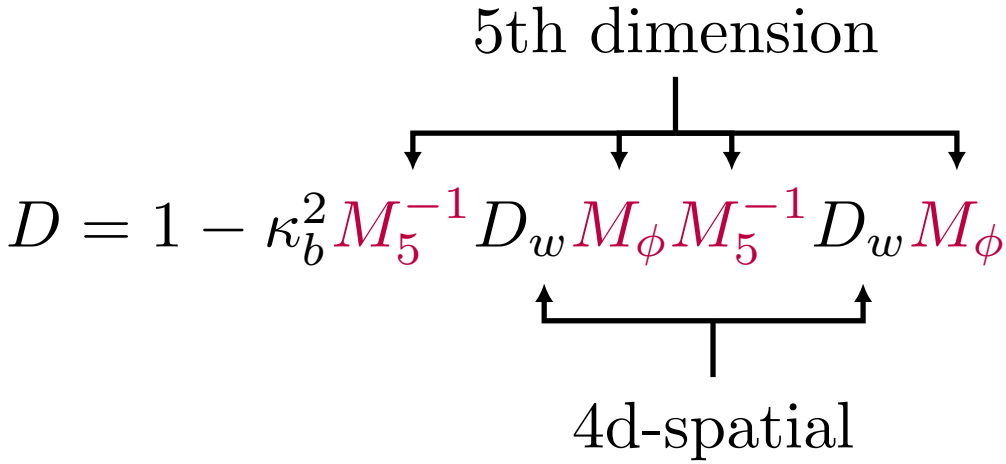
- Our numerical problem is as the following: Need to solve for a large sparse matrix of size $\sim 10^{10}$ by $\sim 10^{10}$ that spans over several different dimensions:
- 4d-spatial dimensions ($\sim 10^8$) entangled with spin(4) and color(3)

From theory to simulation

- Our numerical problem is as the following: Need to solve for a large sparse matrix of size $\sim 10^{10}$ by $\sim 10^{10}$ that spans over several different dimensions:
- 4d-spatial dimensions ($\sim 10^8$) entangled with spin(4) and color(3)
- the 5th dimension ($\sim 10^1$) entangled with spin(4)

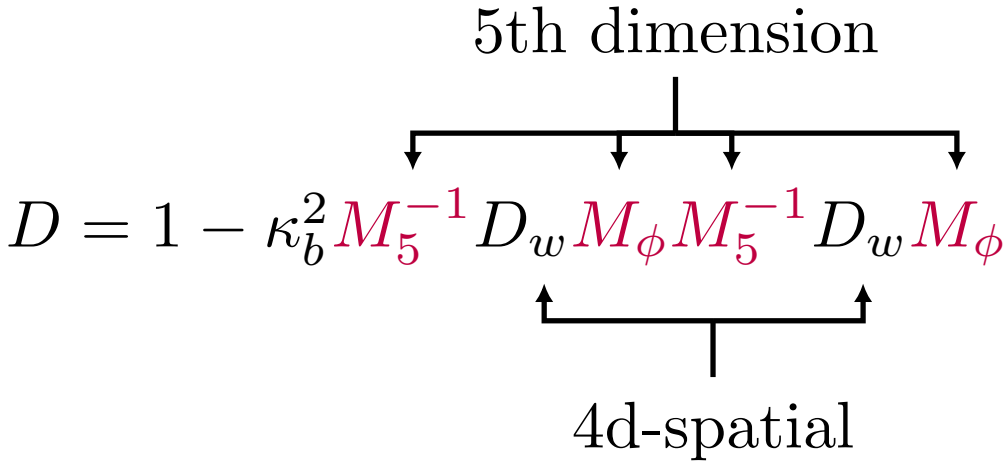
The Matrix

- The Dirac matrix,



The Matrix

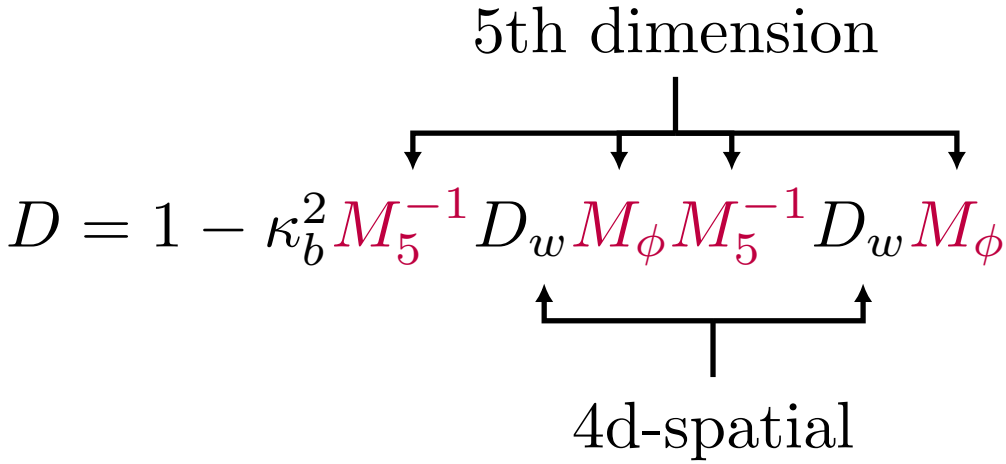
- The Dirac matrix,



- The 4d-spatial operation (D_w) involves only the nearest neighbor with the same 5th dimension index, i.e. only operates horizontally.

The Matrix

- The Dirac matrix,



- The 4d-spatial operation (D_w) involves only the nearest neighbor with the same 5th dimension index, i.e. only operates horizontally.
- The 5th dimension operations (M_5^{-1} and M_ϕ) involves only elements with the same 4d-spatial index, i.e. only operates vertically.

- Conjugate gradient (CG): a Krylov space iterative algorithm that minimizes the residual in each iteration.

The Matrix Inversion

- Conjugate gradient (CG): a Krylov space iterative algorithm that minimizes the residual in each iteration.
- Each iteration applies the normal operator dslash once.

The Matrix Inversion

- Conjugate gradient (CG): a Krylov space iterative algorithm that minimizes the residual in each iteration.
- Each iteration applies the normal operator dslash once.
- Use CG to solve the normal operator problem instead of the original one, ($\dagger$ = transpose + complex conjugate)

$$Dx = y \rightarrow D^\dagger D x = D^\dagger y.$$

The Matrix Inversion

- Conjugate gradient (CG): a Krylov space iterative algorithm that minimizes the residual in each iteration.
- Each iteration applies the normal operator dslash once.
- Use CG to solve the normal operator problem instead of the original one, ($\dagger$ = transpose + complex conjugate)

$$Dx = y \rightarrow D^\dagger D x = D^\dagger y.$$

- The convergence rate decreases as the condition number κ (=max eigenvalue/min eigenvalue) of the matrix $D^\dagger D$ increases.

The Matrix Inversion

- Conjugate gradient (CG): a Krylov space iterative algorithm that minimizes the residual in each iteration.
- Each iteration applies the normal operator dslash once.
- Use CG to solve the normal operator problem instead of the original one, ($\dagger$ = transpose + complex conjugate)

$$Dx = y \rightarrow D^\dagger D x = D^\dagger y.$$

- The convergence rate decreases as the condition number κ (=max eigenvalue/min eigenvalue) of the matrix $D^\dagger D$ increases.
- For our matrix $\kappa \sim 10^8$, i.e. extremely ill-conditioned.

The Matrix Inversion

- Conjugate gradient (CG): a Krylov space iterative algorithm that minimizes the residual in each iteration.
- Each iteration applies the normal operator dslash once.
- Use CG to solve the normal operator problem instead of the original one, ($\dagger$ = transpose + complex conjugate)

$$Dx = y \rightarrow D^\dagger D x = D^\dagger y.$$

- The convergence rate decreases as the condition number κ (=max eigenvalue/min eigenvalue) of the matrix $D^\dagger D$ increases.
- For our matrix $\kappa \sim 10^8$, i.e. extremely ill-conditioned.
- This matrix inversion accounts for over 90% of the simulation time.

- In the measurement phase of the Monte Carlo for one gauge field configuration typically a large number of Dirac equations with the same Dirac matrix but different RHS are solved.

$$D^\dagger D x = D^\dagger y.$$

The Matrix Inversion

- In the measurement phase of the Monte Carlo for one gauge field configuration typically a large number of Dirac equations with the same Dirac matrix but different RHS are solved.

$$D^\dagger D x = D^\dagger y.$$

- Several eigen-space methods, including the implicitly restarted Lanczos algorithm with Chebyshev polynomial [Y. Saad, SIAM J. Numer. Anal. 17, 687 (1980)], have been developed. Low-lying eigenvectors of the matrix are computed to deflate the inversions, the cost of which is amortized by the large number of RHS.

The Matrix Inversion

- In the measurement phase of the Monte Carlo for one gauge field configuration typically a large number of Dirac equations with the same Dirac matrix but different RHS are solved.

$$D^\dagger D x = D^\dagger y.$$

- Several eigen-space methods, including the implicitly restarted Lanczos algorithm with Chebyshev polynomial [Y. Saad, SIAM J. Numer. Anal. 17, 687 (1980)], have been developed. Low-lying eigenvectors of the matrix are computed to deflate the inversions, the cost of which is amortized by the large number of RHS.
- Sheer size of the eigenvectors poses challenge to both the generation (not enough memory) and the storage (not enough disk space, easily reaching peta-byte scale).

Compression

A compression algorithm, which exploits the physical correlation between the eigenvecors, has been developed and applied. For the gauge configurations generated in this work the memory and disk space usage is expected to be reduced by a factor of 30.
[arXiV:1710.06884]

The eigen-space methods does NOT work in the sample generation phase of lattice QCD since during this phase for the same Dirac matrix we only solve very few (typically one) Dirac equations.

Simulation on GPUs

Now finally let's put this problem onto the GPUs

- Code in this work is developed under the framework of QUDA (https://github.com/lattice/quda), a library for performing calculations in lattice QCD on GPUs.

Simulation on GPUs

Now finally let's put this problem onto the GPUs

- Code in this work is developed under the framework of QUDA (https://github.com/lattice/quda), a library for performing calculations in lattice QCD on GPUs.
- Matrix: Even though a sparse matrix, calculating the entries on the fly from the gauge field beats directly loading the whole matrix.

Now finally let's put this problem onto the GPUs

- Code in this work is developed under the framework of QUDA (https://github.com/lattice/quda), a library for performing calculations in lattice QCD on GPUs.
- Matrix: Even though a sparse matrix, calculating the entries on the fly from the gauge field beats directly loading the whole matrix.
- Vector: storage strategy that meets GPUs' global memory coalescing requirement.

Now finally let's put this problem onto the GPUs

- Code in this work is developed under the framework of QUDA (https://github.com/lattice/quda), a library for performing calculations in lattice QCD on GPUs.
- Matrix: Even though a sparse matrix, calculating the entries on the fly from the gauge field beats directly loading the whole matrix.
- Vector: storage strategy that meets GPUs' global memory coalescing requirement.
- Solver: Lower precisions (half/single) are used to improve the performance and higher precisions (double) are used to systematically correct the inaccuracy.

Move Over, China: U.S. Is Again Home to World's Speediest Supercomputer

June 8, 2018

SUMMIT at ORNL

- The SUMMIT machine at Oak Ridge National Laboratory (ORNL) is currently the fastest supercomputer in the world. 6 Tesla V100 GPUs per node with a total of 4608 nodes. Our proposed jobs run on 1024 nodes (6096 GPUs) at a time.

	we have ...	for one iter. we need
memory bandwidth	850 GB/s	69 GB
network bandwidth	8.3 GB/s	21 GB
network/memory	0.010	0.304

SUMMIT at ORNL

- The SUMMIT machine at Oak Ridge National Laboratory (ORNL) is currently the fastest supercomputer in the world. 6 Tesla V100 GPUs per node with a total of 4608 nodes. Our proposed jobs run on 1024 nodes (6096 GPUs) at a time.
- Distribute our 4d-spatial lattice to the GPUs. Consequence: need to communicate between the nodes; network bandwidth largely determines the scaling.

	we have ...	for one iter. we need
memory bandwidth	850 GB/s	69 GB
network bandwidth	8.3 GB/s	21 GB
network/memory	0.010	0.304

Preconditioned CG

- Precondition the original matrix A to reduce the condition number κ.

with preconditioning	we have ...	for one iter. we need
memory bandwidth	850 GB/s	69×12.2 GB
network bandwidth	8.3 GB/s	21 GB
network/memory	0.010	0.025

Preconditioned CG

- Precondition the original matrix A to reduce the condition number κ.
- Solve $(AP^{-1})(Px) = y$ instead of $Ax = y$. Choose some preconditioner P that requires no communication.

with preconditioning	we have ...	for one iter. we need
memory bandwidth	850 GB/s	69×12.2 GB
network bandwidth	8.3 GB/s	21 GB
network/memory	0.010	0.025

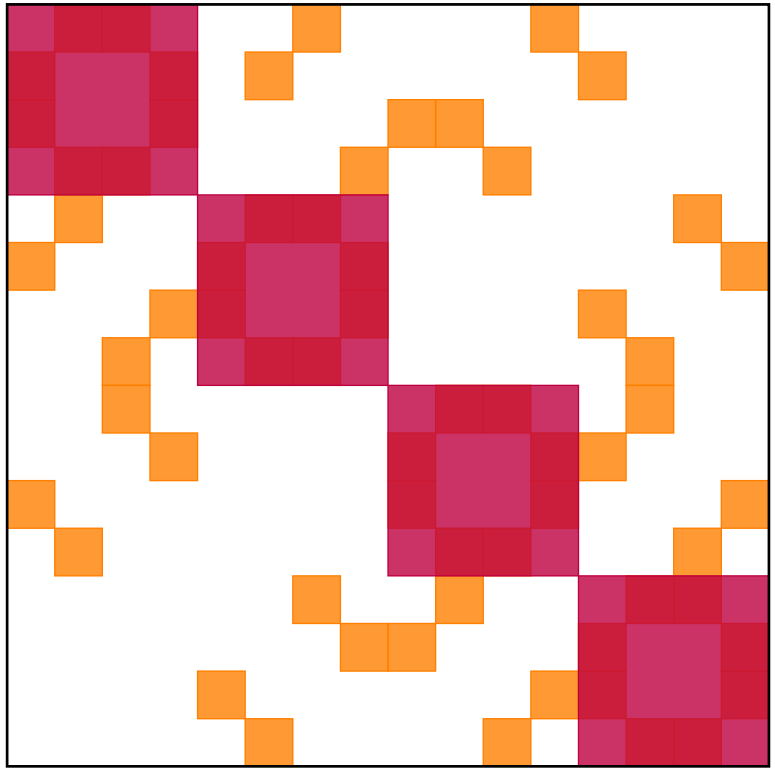
Preconditioned CG

- Precondition the original matrix A to reduce the condition number κ.
- Solve $(AP^{-1})(Px) = y$ instead of $Ax = y$. Choose some preconditioner P that requires no communication.
- Consequence: Less iterations, less communication needed; Needs more local computation on each GPU.

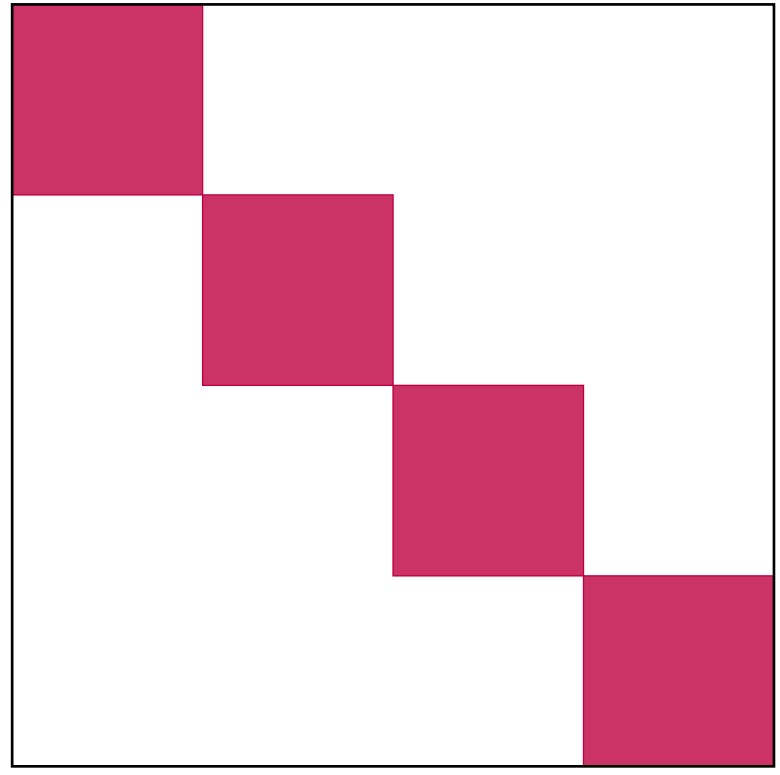
with preconditioning	we have ...	for one iter. we need
memory bandwidth	850 GB/s	69×12.2 GB
network bandwidth	8.3 GB/s	21 GB
network/memory	0.010	0.025

Preconditioned CG

Choose the predicontioner with the *correct* boundary condition. [arXiv:1811.08488]



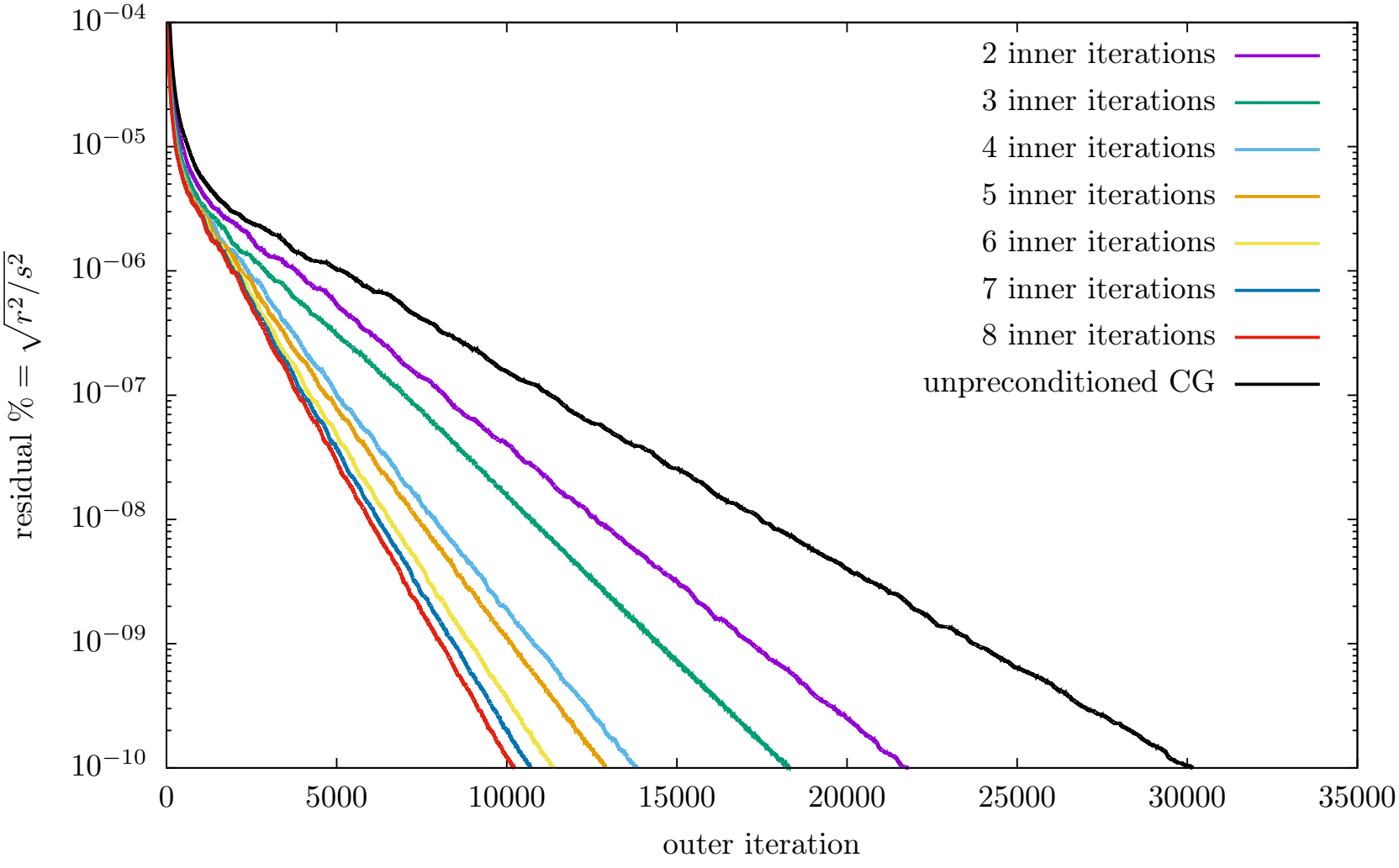
A



$P = \bigoplus_s A_s$

Preconditioned CG

The preconditioner inversion P^{-1} does not need to be solved to arbitrarily high precision for the algorithm to work.



More Problems

- Without preconditioning network bandwidth is the bottleneck due to large communication demand.

More Problems

- Without preconditioning network bandwidth is the bottleneck due to large communication demand.
- With preconditioning the balance is shifted. In order to achieve a speed up in terms of time to solution we need to do better on utilizing the memory bandwidth and compute flops available.

More Problems

- Without preconditioning network bandwidth is the bottleneck due to large communication demand.
- With preconditioning the balance is shifted. In order to achieve a speed up in terms of time to solution we need to do better on utilizing the memory bandwidth and compute flops available.
    - Memory bandwidth: kernel fusion;

More Problems

- Without preconditioning network bandwidth is the bottleneck due to large communication demand.
- With preconditioning the balance is shifted. In order to achieve a speed up in terms of time to solution we need to do better on utilizing the memory bandwidth and compute flops available.
  - Memory bandwidth: kernel fusion;
  - Compute flops: tensor core for the 5th dimension operations.

Kernel Fusion

The 4d-volume ($\sim 10^8$) is much larger than the size of the 5th dimension ($\sim 10^1$). Fusing the 4d-spatial operations with the 5th dimension operations right after reduces global memory traffic.

$$\left[1-\kappa_b^2 \underbrace{M_\phi^\dagger D_w^\dagger}_{\text{fuse}} \underbrace{M_5^{-\dagger} M_\phi^\dagger D_w^\dagger}_{\text{fuse}} \underbrace{M_5^{-\dagger}}_{\text{fuse}}\right]\left[1-\kappa_b^2 \underbrace{M_5^{-1} D_w}_{\text{fuse}} \underbrace{M_\phi M_5^{-1} D_w}_{\text{fuse}} M_\phi\right]$$

This goes as the following:

- For each block apply the 4d-spatial operation D_w. Here we rely on the GPU cache system to achieve data reuse. The theoretical limit is 8-way (nearest neighbors in 4 directions) and we are getting around 50-60% of that.

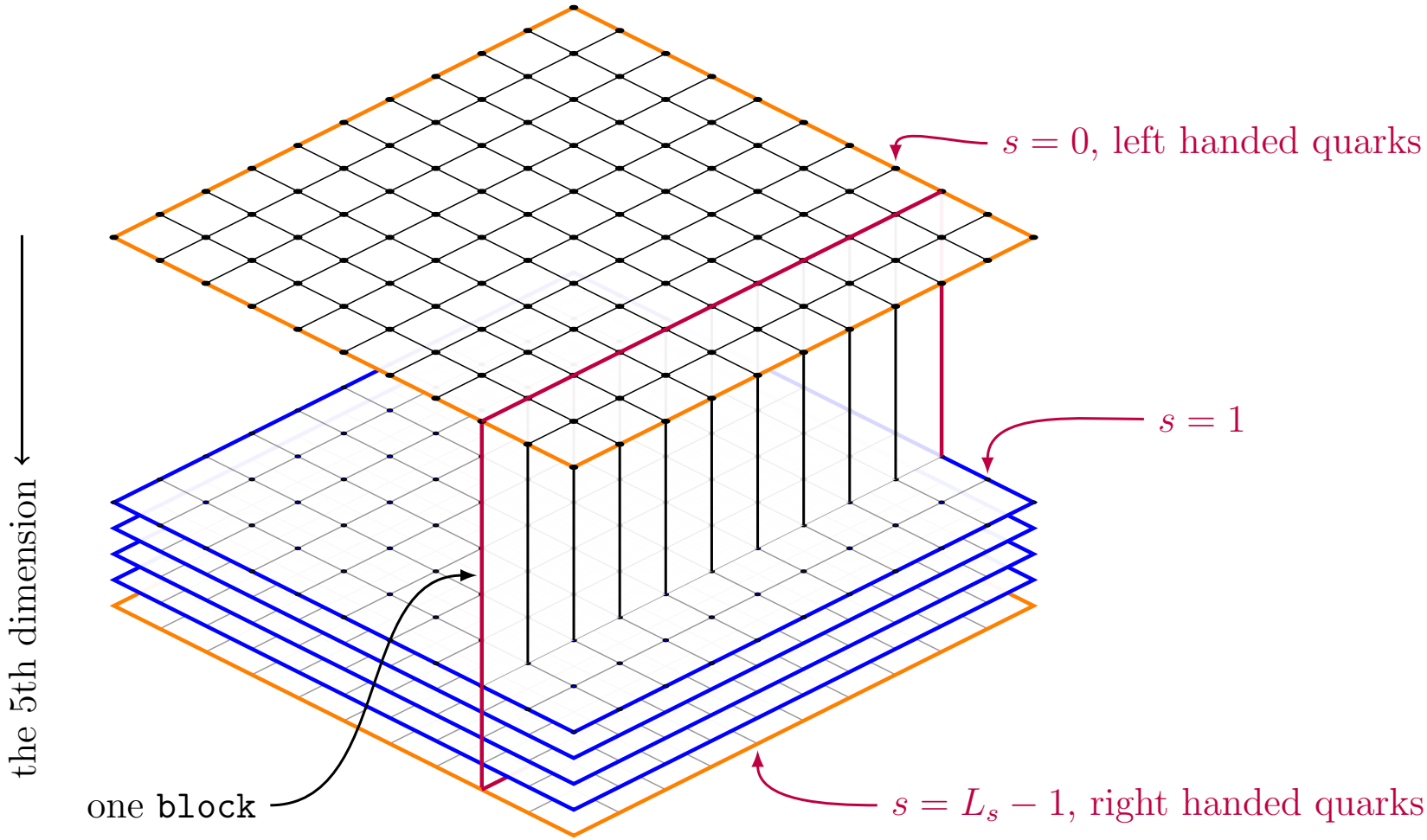
This goes as the following:

- For each block apply the 4d-spatial operation D_w. Here we rely on the GPU cache system to achieve data reuse. The theoretical limit is 8-way (nearest neighbors in 4 directions) and we are getting around 50-60% of that.
- The results are stored in the shared memory.

This goes as the following:

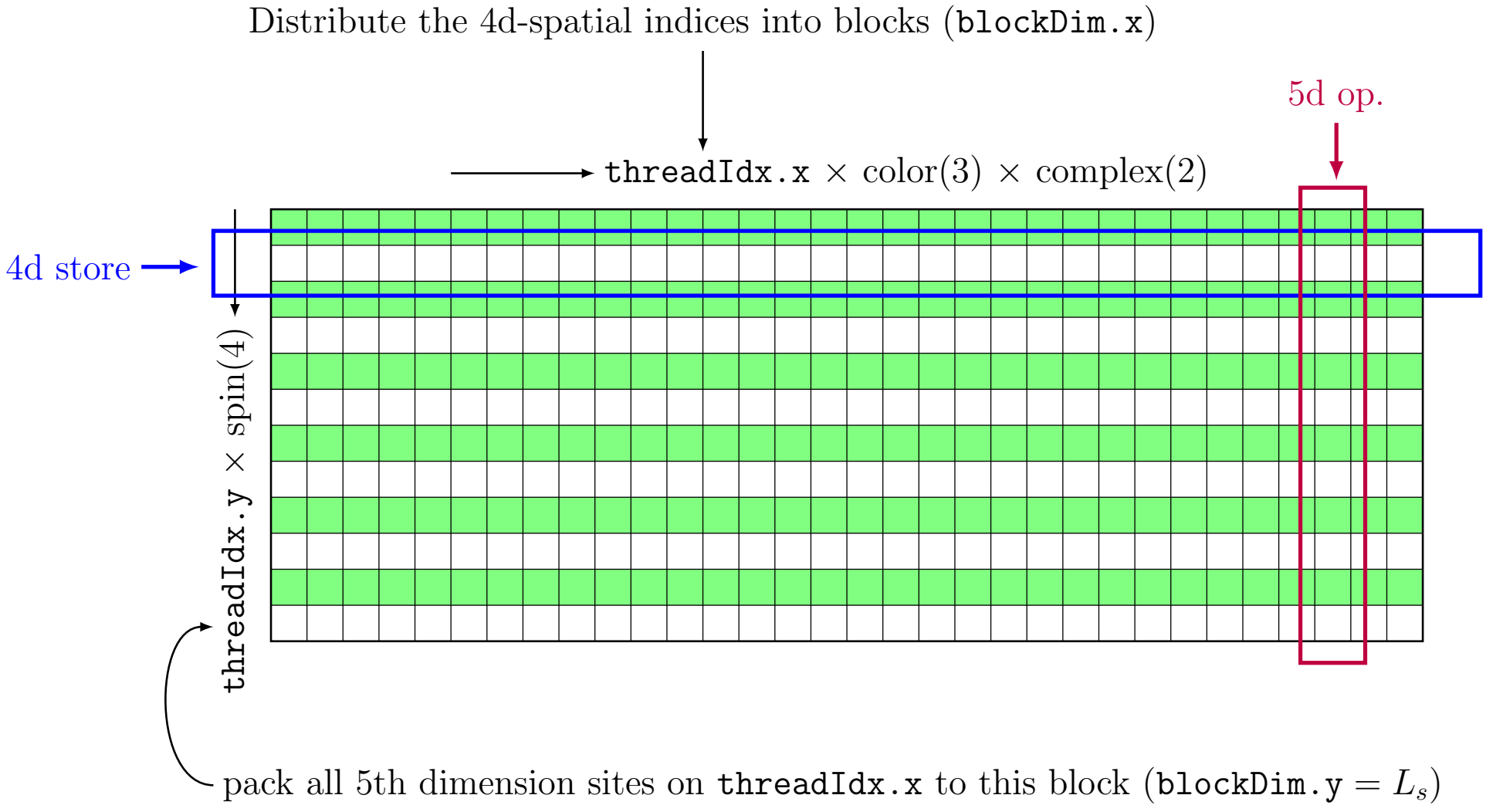
- For each block apply the 4d-spatial operation D_w. Here we rely on the GPU cache system to achieve data reuse. The theoretical limit is 8-way (nearest neighbors in 4 directions) and we are getting around 50-60% of that.
- The results are stored in the shared memory.
- the 5th dimension operators are applied on the shared memory. We are not at the mercy of the cache system here: 100% data reuse.

Kernel Fusion



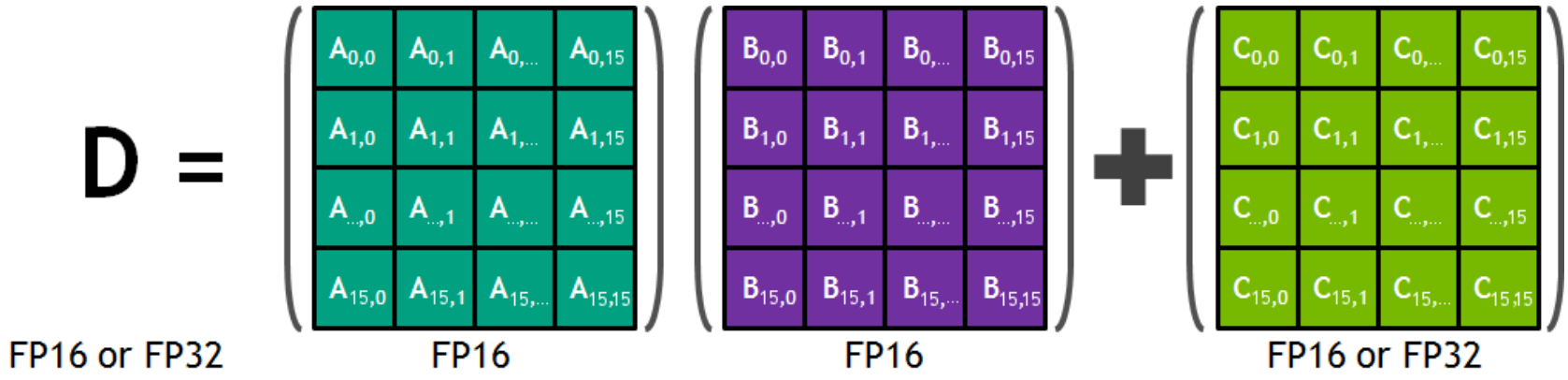
Kernel Fusion: Shared Memory

Take a vertical slice ...



Tensor Cores

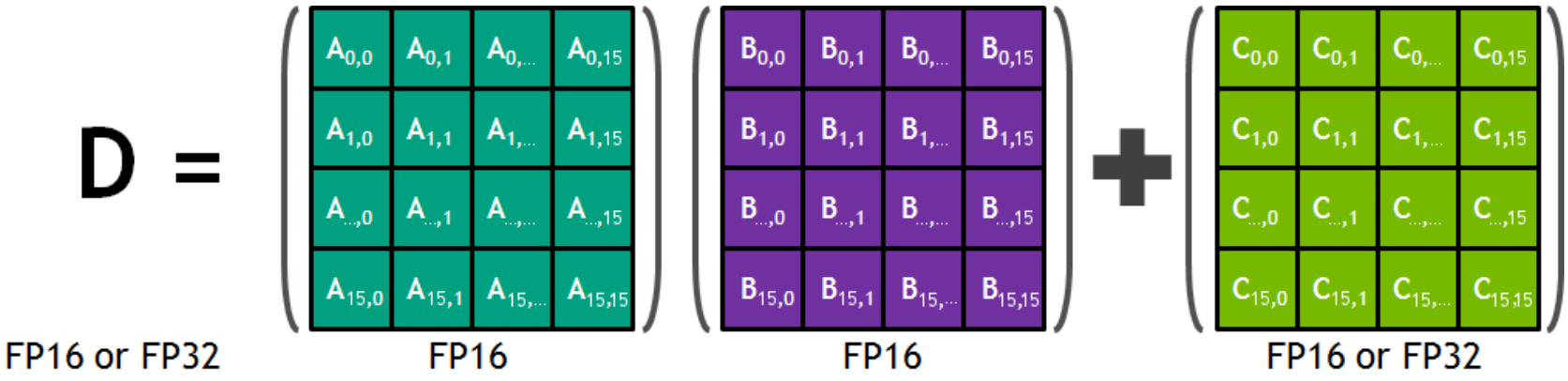
"Tensor cores provide a huge boost to convolutions and matrix operations."



- Half precision? No problem, we are already using half precision.

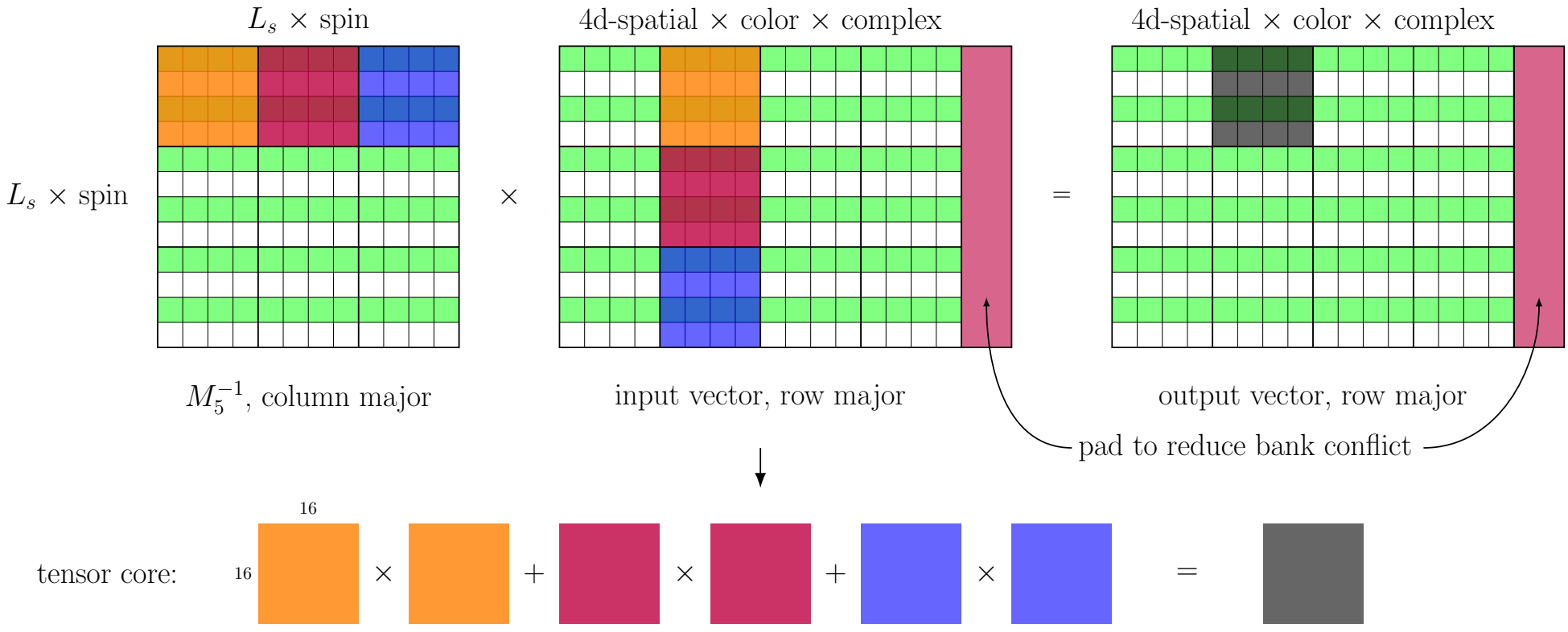
Tensor Cores

"Tensor cores provide a huge boost to convolutions and matrix operations."



- Half precision? No problem, we are already using half precision.
- Memory bandwidth? No problem, the numbers are already in shared memroy.

Memory Layout for Tensor Cores

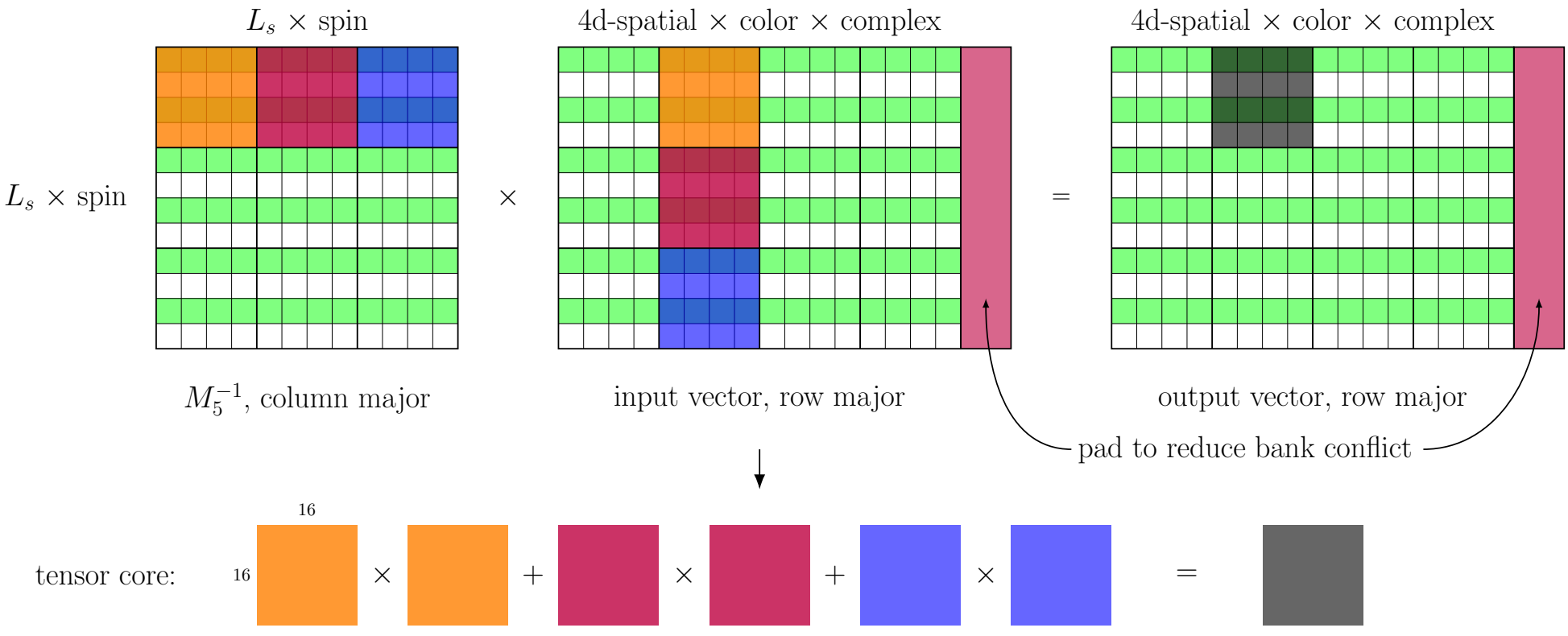


- Due to the way the `wmma` functions are called it is not necessary to have two separate pieces of memory for the input and output vector. We load from and store into the same shared memory location in each block.

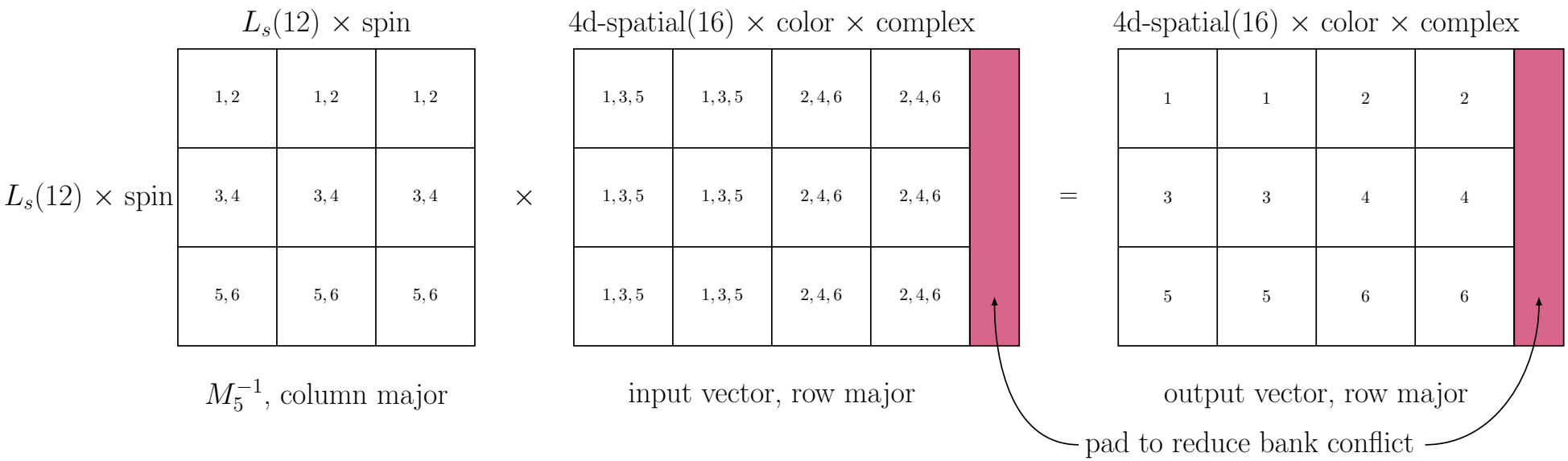
```cpp
#include <mma.h>

using namespace nvcuda::wmma;

__global__ void wmma_ker(half *a, half *b, float *c) {
  // Declare the fragments
  fragment<matrix_a, 16, 16, 16, half, col_major> a_frag;
  fragment<matrix_b, 16, 16, 16, half, row_major> b_frag;
  fragment<accumulator, 16, 16, 16, float> c_frag;

  // Initialize the output to zero
  fill_fragment(c_frag, 0.0f);

  // Load the inputs
  load_matrix_sync(a_frag, a, 16);
  load_matrix_sync(b_frag, b, 16);

  // Perform the matrix multiplication
  mma_sync(c_frag, a_frag, b_frag, c_frag);

  // Store the output
  store_matrix_sync(c, c_frag, 16, wmma::mem_row_major);
}
```

Further optimizations: Template Everything

- Template everything: maximize compile time optimization.

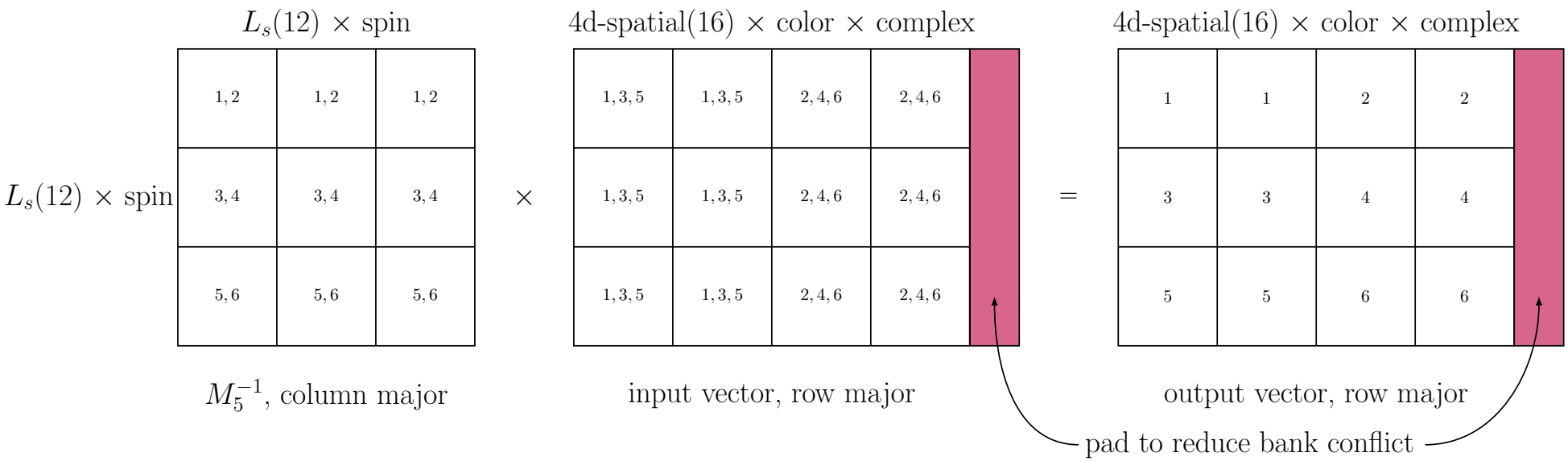


Further optimizations: Reload or Not?



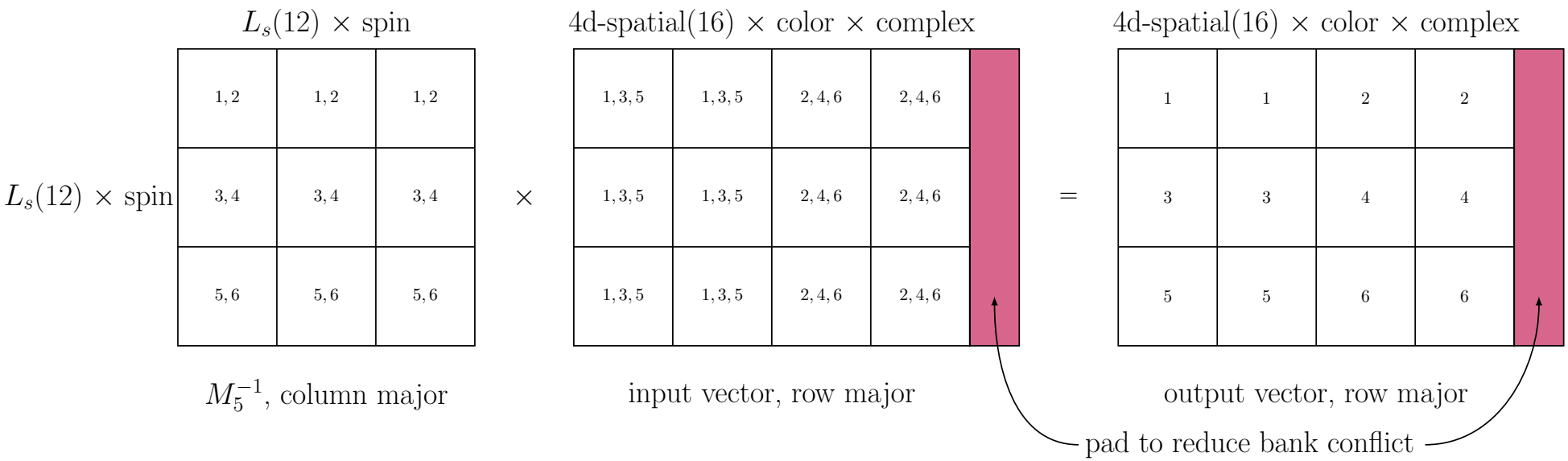
- $12 \times 16 = 192$ threads in the block; $192/32 = 6$ *virtual* warps.

Further optimizations: Reload or Not?



- $12 \times 16 = 192$ threads in the block; $192/32 = 6$ *virtual* warps.
- The numbers in each tile shows the warps that need to load it.

Further optimizations: Reload or Not?



- $12 \times 16 = 192$ threads in the block; $192/32 = 6$ *virtual* warps.
- The numbers in each tile shows the warps that need to load it.
- We can reload the matrix when ever needed, or preload them at the beginning.

Further optimizations: Reload or Not?

- "To reload, or not to reload, that is the question." – William Shakespeare
- Preloading naturally saves the loading time. Reloading saves register usage: need only one `wmma:fragment` object.
- We use QUDA's intrinsic autotuning feature to decide which strategy wins: just try both and pick the faster one.

Further optimizations: Spill or Not?

- Since register usage is a problem, we also tune register usage using `__launch_bounds__(maxThreadsPerBlock, minBlocksPerMultiprocessor)`.

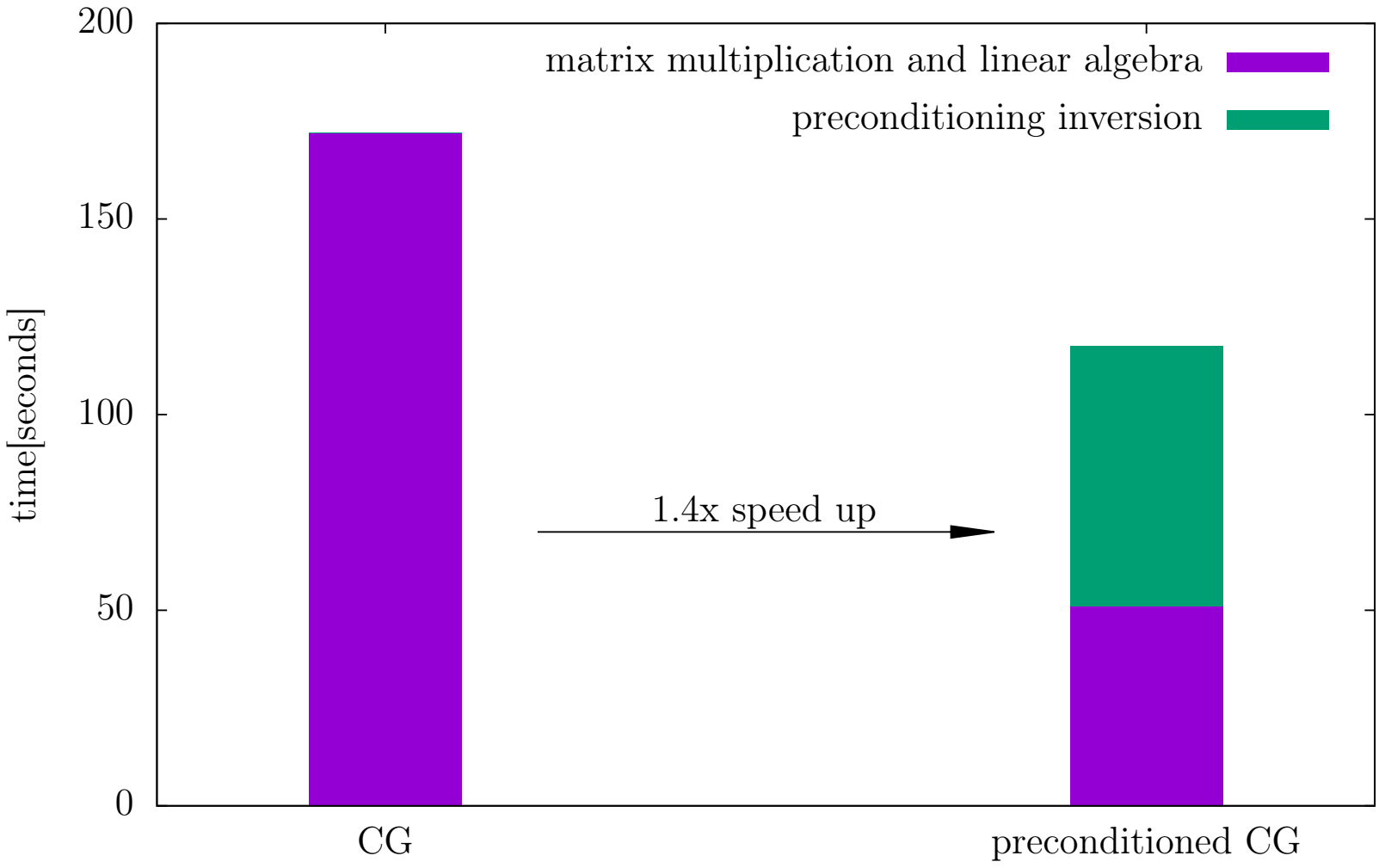
Further optimizations: Spill or Not?

- Since register usage is a problem, we also tune register usage using `__launch_bounds__(maxThreadsPerBlock, minBlocksPerMultiprocessor)`.
- Kernel variables could spill into device memory, potentially leads to latency.

Further optimizations: Spill or Not?

- Since register usage is a problem, we also tune register usage using `__launch_bounds__(maxThreadsPerBlock, minBlocksPerMultiprocessor)`.
- Kernel variables could spill into device memory, potentially leads to latency.
- But why worry? It won't do any harm since we can also use autotuning to do the dirty work for us ... :)

Results


200
150
100
50
0
time[seconds]
matrix multiplication and linear algebra
preconditioning inversion
1.4x speed up
CG
preconditioned CG

Conclusion

- With a carefully chosen preconditioner we are able to shift the balance between network bandwidth, GPU memory bandwidth and compute flops when solving a large sparse ill-conditioned matrix on a large scale supercomputer. The scaling is improved and we are able to achieve a speed up.

Conclusion

- With a carefully chosen preconditioner we are able to shift the balance between network bandwidth, GPU memory bandwidth and compute flops when solving a large sparse ill-conditioned matrix on a large scale supercomputer. The scaling is improved and we are able to achieve a speed up.
- Tensor cores are used, possibly for the first time, to speed up the 5th dimension operations of domain wall fermion in lattice QCD simulations.