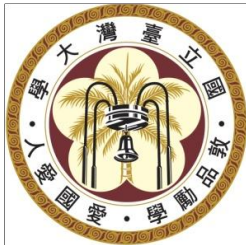


Lattice 2010, Villasimius, Italy

GPU-Based Conjugate Gradient Solver for Lattice QCD with Domain-Wall Fermions



TWQCD Collaboration
Ting-Wai Chiu, Yao-Yuan Mao, Kenji Ogawa
Dept. of Physics, National Taiwan University



Lattice 2010, June 18

Outline

- ▶ Introduction
- ▶ Our problem to solve
 - ▶ CG solver for optimal domain-wall fermion operator
- ▶ Implementation on CUDA architecture
- ▶ Performance
- ▶ Summary

Introduction

- ▶ With CUDA/OpenCL, GPU enables large-scale parallel computation for general purpose with a much lower price when comparing to a CPU cluster.
- ▶ For lattice QCD, much of the computation during HMC simulation is matrix-vector multiplication, which can be parallelized well.
- ▶ GPU-based codes for Wilson/staggered/overlap fermions have been developed by other groups.
- ▶ We develop GPU-based codes for (optimal) domain-wall fermions, which provide exact chiral symmetry at finite lattice spacing.

Optimal Domain-Wall Fermions

- ▶ The optimal domain-wall fermion proposed by T.-W. Chiu can maintain optimal chiral symmetry on lattice.

Different fermions can be obtained by adjusting these two weights

$$[\mathcal{D}(m_q)]_{xx';ss'} = (\omega_s D_w + 1)_{xx'} \delta_{ss'} + (\sigma_s D_w - 1)_{xx'} L_{ss'}$$

$$(D_w)_{xx'} = -\frac{1}{2} \sum_{\mu} [(1 - \gamma_{\mu}) U_{\mu}(x) \delta_{x+a\hat{\mu},x'} + (1 + \gamma_{\mu}) U_{\mu}^{\dagger}(x') \delta_{x-a\hat{\mu},x'}] \\ + (d - m_0)$$

$$L = P_+ L_+ + P_- L_-, \quad P_{\pm} = (1 \pm \gamma_5)/2$$

$$(L_+)_{ss'} = \begin{cases} \delta_{s-1,s'}, & 1 < s \leq N_s \\ -(m_q/2m_0) \delta_{N_s,s'}, & s = 1 \end{cases}, \quad L_- = (L_+)^{\dagger}$$

Even-Odd Preconditioning

- ▶ We separate even and odd sites. $\begin{pmatrix} \text{EE} & \text{EO} \\ \text{OE} & \text{OO} \end{pmatrix}$
- ▶ Let D^{OE} and D^{EO} contain only 4D hopping terms.

$$\begin{aligned}
 \mathcal{D}(m_q) &= D_w(\omega + \sigma L) + (1 - L) \\
 &= \begin{pmatrix} d - m_0 & D_w^{\text{EO}} \\ D_w^{\text{OE}} & d - m_0 \end{pmatrix} (\omega + \sigma L) + (1 - L) \\
 &= \begin{pmatrix} (d - m_0)(\omega + \sigma L) + (1 - L) & D_w^{\text{EO}}(\omega + \sigma L) \\ D_w^{\text{OE}}(\omega + \sigma L) & (d - m_0)(\omega + \sigma L) + (1 - L) \end{pmatrix} \\
 &= \begin{pmatrix} X & D_w^{\text{EO}}Y \\ D_w^{\text{OE}}Y & X \end{pmatrix},
 \end{aligned}$$

$$X \equiv (d - m_0)\omega(1 + cL) + (1 - L), \quad Y \equiv \omega(1 + cL)$$

$$\omega_{ss'} = \omega_s \delta_{ss'}, \quad \sigma_{ss'} = \sigma_s \delta_{ss'}, \quad c_{ss'} = (\sigma_s / \omega_s) \delta_{ss'}$$

Further Preconditioning

- ▶ We further make the expression more symmetric.

$$M_5 \equiv \sqrt{\omega}^{-1} Y X^{-1} \sqrt{\omega} = \left[(d - m_0) + \sqrt{\omega}^{-1} (1 - L)(1 + cL)^{-1} \sqrt{\omega}^{-1} \right]^{-1}$$

$$S_1 \equiv \sqrt{\omega}^{-1} Y X^{-1} = M_5 \sqrt{\omega}^{-1}, \quad S_2 \equiv Y^{-1} \sqrt{\omega}$$

$$\mathcal{D}(m_q) = S_1^{-1} \begin{pmatrix} 1 & M_5 D_w^{\text{EO}} \\ M_5 D_w^{\text{OE}} & 1 \end{pmatrix} S_2^{-1}$$

Schur decomposition



$$\mathcal{D}(m_q) = S_1^{-1} \begin{pmatrix} 1 & 0 \\ M_5 D_w^{\text{OE}} & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & C \end{pmatrix} \begin{pmatrix} 1 & M_5 D_w^{\text{EO}} \\ 0 & 1 \end{pmatrix} S_2^{-1}$$

$$C \equiv 1 - M_5 D_w^{\text{OE}} M_5 D_w^{\text{EO}}$$

Conjugate Gradient Method (CG)

- ▶ Conjugate Gradient is an iterative method for solving the inverse of a positive-definite Hermitian matrix.

$$Ax = b, \quad A = CC^\dagger$$

$x_0 :=$ initial guess

$$r_0 := b - Ax$$

$$p_0 := r_0$$

CG is used for calculating fermion force and quark propagator.

It is the most time-consuming part in the simulation.

Iteration to convergence

$$\alpha_k = \frac{(r_k, r_k)}{(p_k, Ap_k)} = \frac{(r_k, r_k)}{(C^\dagger p_k, C^\dagger p_k)}$$

$$r_{k+1} = r_k - \alpha_k Ap_k$$

$$\beta_{k+1} = \frac{(r_{k+1}, r_{k+1})}{(r_k, r_k)}$$

$$x_{k+1} = x_k + \alpha_k p_k$$

$$p_{k+1} = r_{k+1} + \beta_{k+1} p_k$$

Mixed-Precision CG (1)

Single-precision operations are much faster than double-precision ones on GPU

High precision

$$\hat{A}\hat{x} = \hat{b}, \quad \hat{A} = \hat{C}\hat{C}^\dagger$$

$\hat{x} :=$ initial guess

$$\hat{r} := \hat{b} - \hat{A}\hat{x}$$

while ($|\hat{r}|^2 > \hat{\epsilon}$) {

$$r := \hat{r}$$

$$p := \hat{r}$$

$$x := 0$$

Low-precision CG

$$\hat{x} := \hat{x} + x$$

$$\hat{r} := \hat{b} - \hat{A}\hat{x}$$

}

Low precision

$$Ax = r(= \hat{r}), \quad A = CC^\dagger$$

$$\rho := (r, r)$$

while ($\beta_0 > \epsilon$) {

$$v_0 := C^\dagger p$$

$$\alpha := \rho / (v_0, v_0)$$

$$r := r - \alpha C v_0$$

$$\rho' := \rho$$

$$\rho := (r, r)$$

$$x := x + \alpha p$$

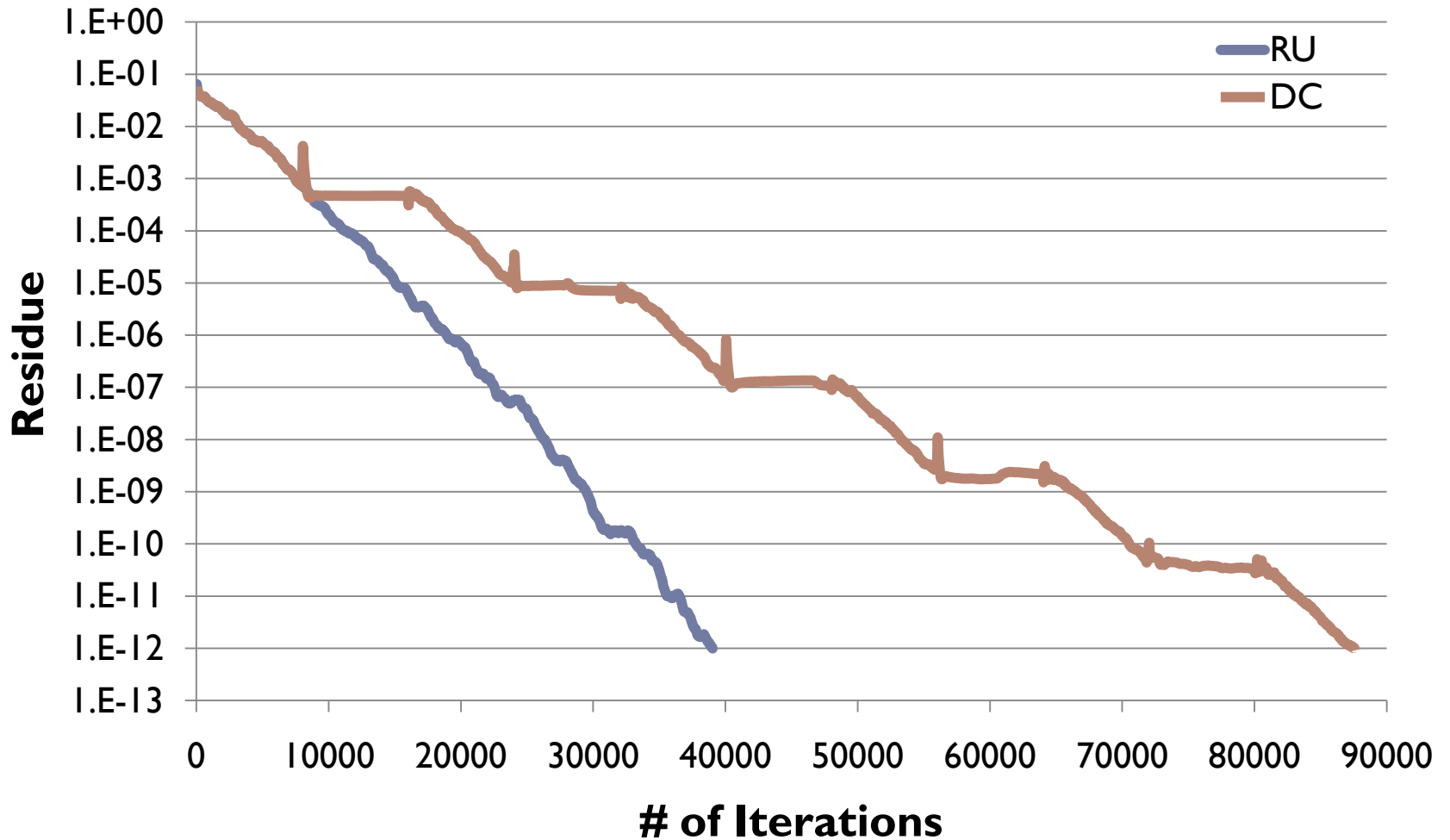
$$p := r + (\rho / \rho') p$$

}

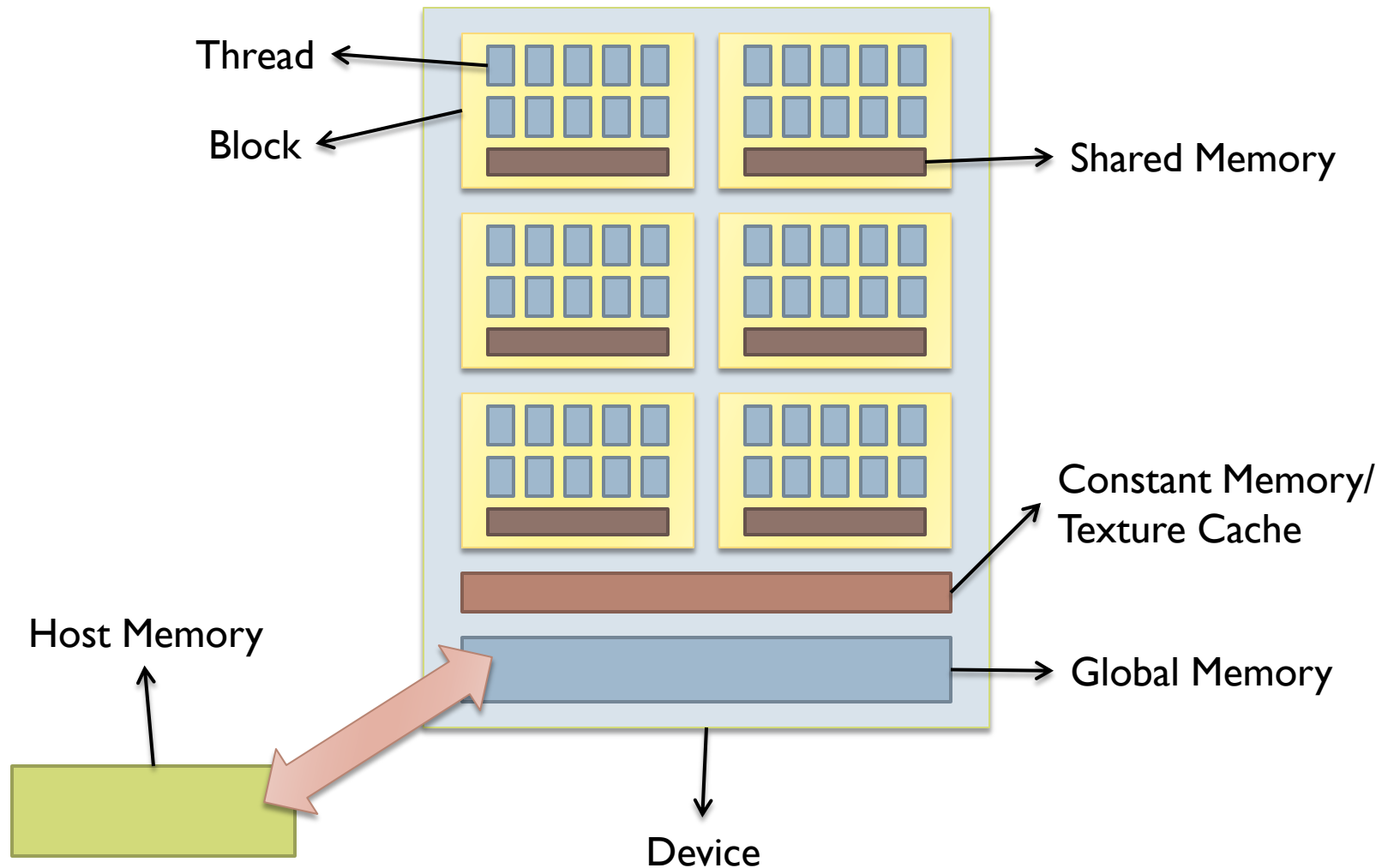
Mixed-Precision CG (2)

Defect Correction	Reliable Updates
Set $p = \hat{r}$ every time when starting low-precision iteration	Keep the previous p when starting low-precision iteration
$(p_k, r_l) = 0, \forall 0 < k < l$	$(p_k, r_l) \sim 0, \forall 0 < k < l$
Discard previous information Takes longer to convergence	Keep previous information Take shorter to convergence
Strict stopping criterion should be used for low-precision CG	Loose stopping criterion should be used for low-precision CG

Mixed-Precision CG (3)



CUDA Architecture



Memory Architecture

- ▶ Basic ideas about the memory architecture:

	size	access	bandwidth
Global	Large	r/w by all threads and host	Slow
Constant	Small	Read only by all threads	Fast
Texture	Small cache	Read only by all threads	Fast
Shared	Very small	r/w by all threads within one block	Very fast
Register	Very small	r/w by only one thread	Very fast

- ▶ Shared memory may have bank conflict
- ▶ Texture can take care of the locality.
- ▶ GPU computing is **memory bandwidth bound!**

Thread/Block Management

- ▶ Parallelize a loop by designating the value of loop counter to each thread.
- ▶ Number of threads per block
 - ▶ Should be tested to find the best value (may be limited by resource in one block)
 - ▶ Must be a multiple of **half-warp**.
- ▶ **Memory bandwidth bound** → Try to **reuse data**.
 - ▶ Larger number of blocks does NOT mean better performance!
 - ▶ Using loop inside kernel to reduce the number of blocks sometimes runs faster.

CG Kernels Overview (single-prec.)

$$C \equiv 1 - M_5 D_w^{\text{OE}} M_5 D_w^{\text{EO}}$$

The multiplication of M_5 and D_w are implemented in different kernels.

$$v_0 := C^\dagger p$$

$$\alpha := \rho / (v_0, v_0)$$

$$r := r - \alpha C v_0$$

$$\rho' := \rho$$

$$\rho := (r, r)$$

$$x := x + \alpha p$$

$$p := r + (\rho / \rho') p$$

Each line below is implemented in one kernel.

$$v_1 := M_5^\dagger p$$

$$v_0 := (D_w^{\text{OE}})^\dagger v_1$$

$$v_1 := M_5^\dagger v_0$$

$$v_0 := p - (D_w^{\text{EO}})^\dagger v_1$$

$$\alpha := \rho / (v_0, v_0)$$

$$v_1 := D_w^{\text{EO}} v_0, \quad r := r - \alpha v_0$$

$$v_0 := M_5 v_1$$

$$v_1 := D_w^{\text{OE}} v_0$$

$$r := r + \alpha M_5 v_1$$

$$\rho' := \rho_0, \quad \rho_0 := (r, r)$$

$$x := x + \alpha p, \quad p := r + (\rho / \rho') p$$

CG Kernels Overview (double-prec.)

$$A \equiv CC^\dagger$$

$$C \equiv 1 - M_5 D_w^{\text{OE}} M_5 D_w^{\text{EO}}$$

The multiplication of M_5 and D_w are implemented in different kernels.

$$\hat{x} := \hat{x} + x$$

$$\hat{r} := \hat{b} - \hat{A}\hat{x}$$

Each line below is implemented in one kernel.

$$v_1 := M_5^\dagger p$$

$$v_0 := (D_w^{\text{OE}})^\dagger v_1$$

$$v_1 := M_5^\dagger v_0$$

$$v_0 := (D_w^{\text{EO}})^\dagger v_1$$

$$v_1 := p - v_0$$

$$v_2 := D_w^{\text{EO}} v_1$$

$$v_0 := M_5 v_2$$

$$v_2 := D_w^{\text{OE}} v_0$$

$$v_1 := v_1 - v_2$$

$$r := b - v_1$$

Dw Multiplication Implementation

$$(D_w^{\text{OE}})_{xx'} = -\frac{1}{2} \sum_{\mu} \left[(1 - \gamma_{\mu}) U_{\mu}(x) \delta_{x+a\hat{\mu},x'} + (1 + \gamma_{\mu}) U_{\mu}^{\dagger}(x') \delta_{x-a\hat{\mu},x'} \right]$$

▶ Hopping terms

- ▶ Texture is used for caching data
- ▶ Internal loop is used to reuse read-in data

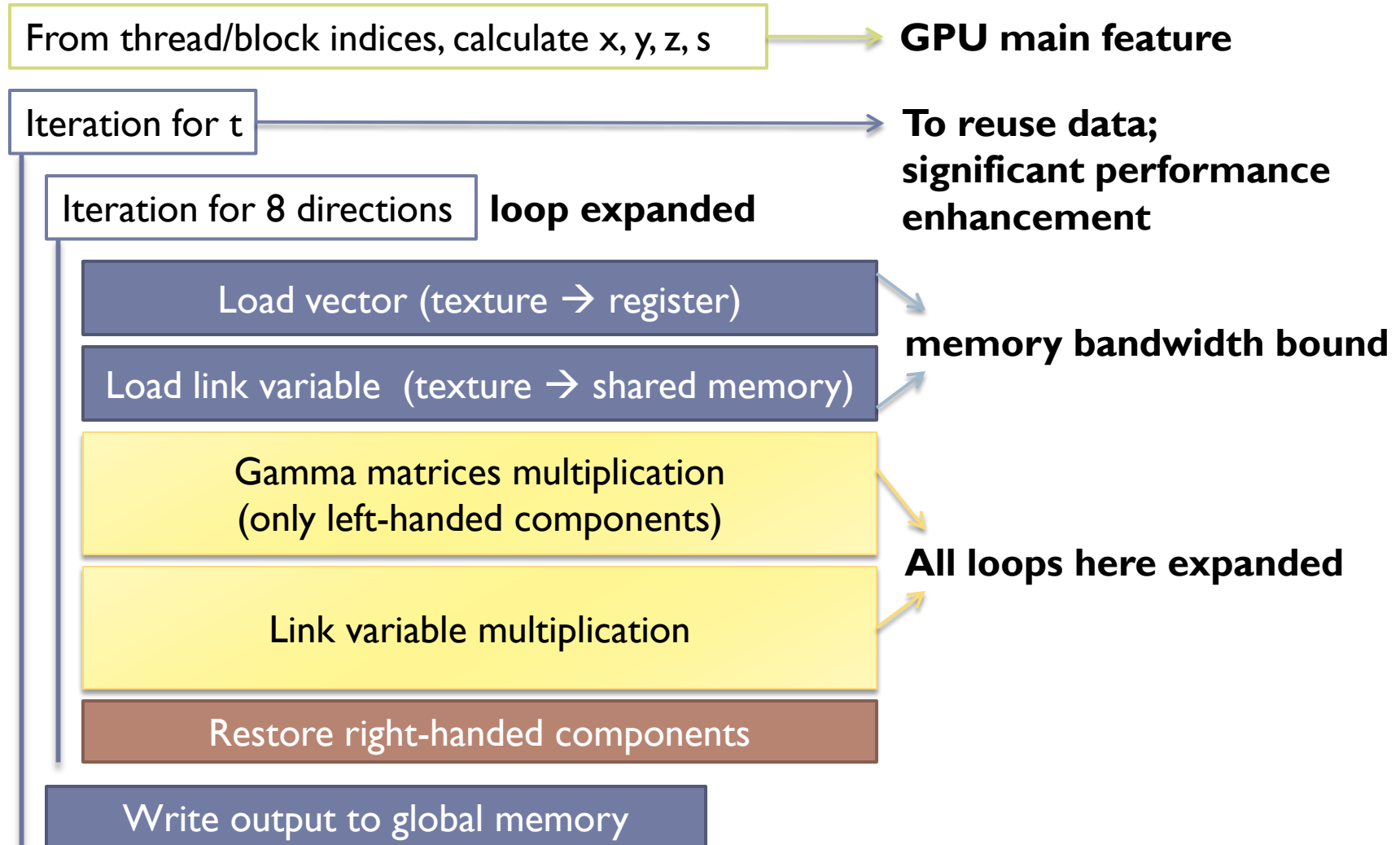
▶ Link variables multiplication

- ▶ For a given μ , U is the same for all $s = 1 \dots N_s$
→ use shared memory

▶ Gamma matrices multiplication

- ▶ Only left-handed Dirac indices are calculated

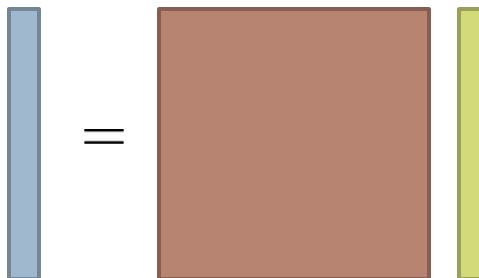
Dw Multiplication Block Diagram



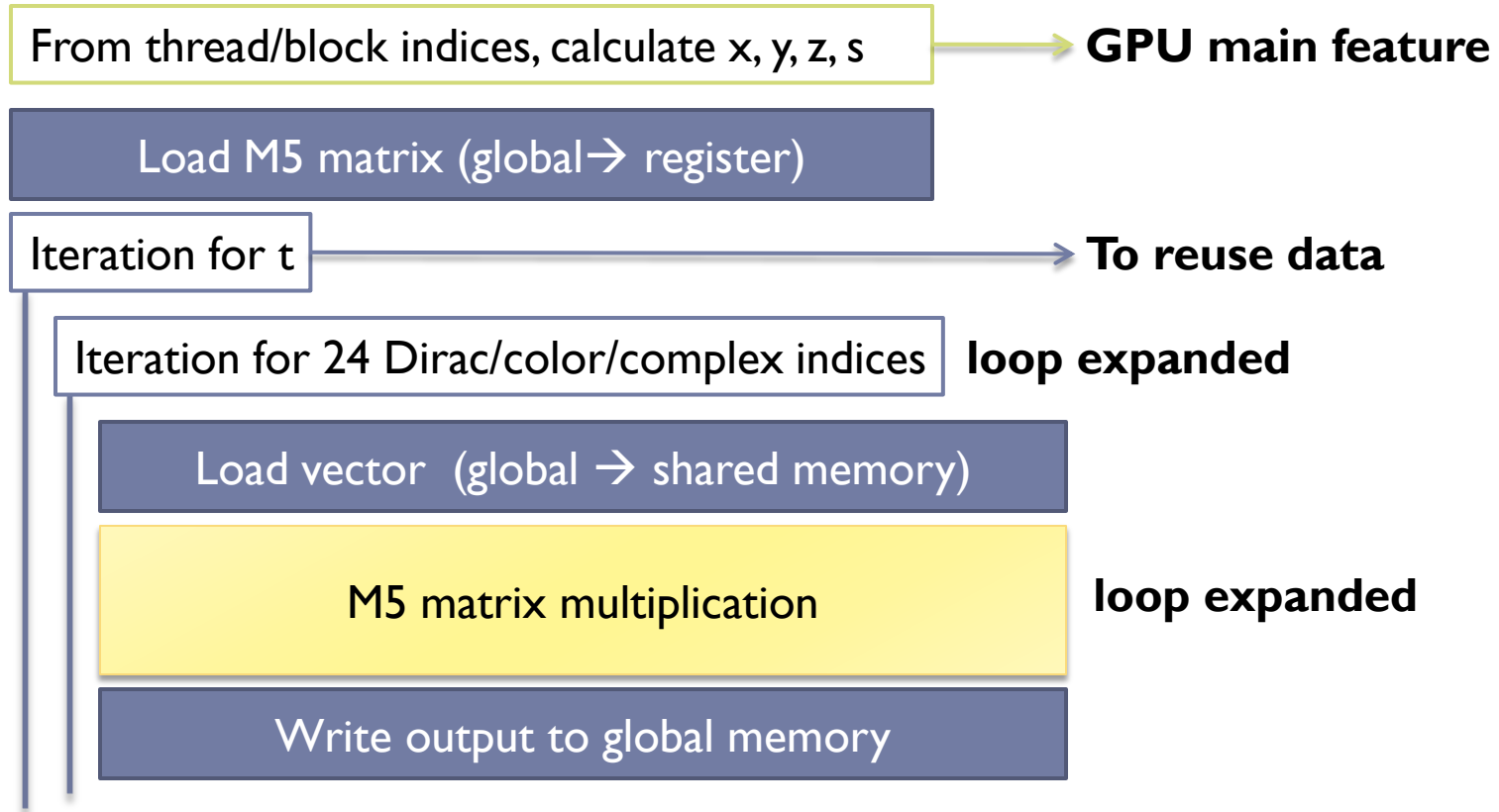
M5 Multiplication Implementation

$$M_5 = \left[(d - m_0) + \sqrt{\omega}^{-1} (1 - L)(1 + cL)^{-1} \sqrt{\omega}^{-1} \right]^{-1}$$

- ▶ **Block diagonal in the chiral basis.**
- ▶ Does not depend on x, y, z, t , or color index.
- ▶ So it is a **constant matrix-vector multiplication** in the 5th-dim space.
- ▶ Use shared memory for storing **source vector**

$$v_{s'} = \sum_s (M_5)_{s' s} v_s$$


M5 Multiplication Block Diagram



Some More Tuning Methods for Kernels

- ▶ To calculate the norm of a vector, we need to sum up all the components → **Parallel Reduction**
To maximize the number of working threads!

- ▶ Try to **reuse data** as much as possible!
- ▶ When doing parallel reduction (calculating norm), do partly in the pervious kernel:

$v_0 := p - (D_w^{\text{EO}})^\dagger v_1$ → Do a “pre parallel reduction” within each block

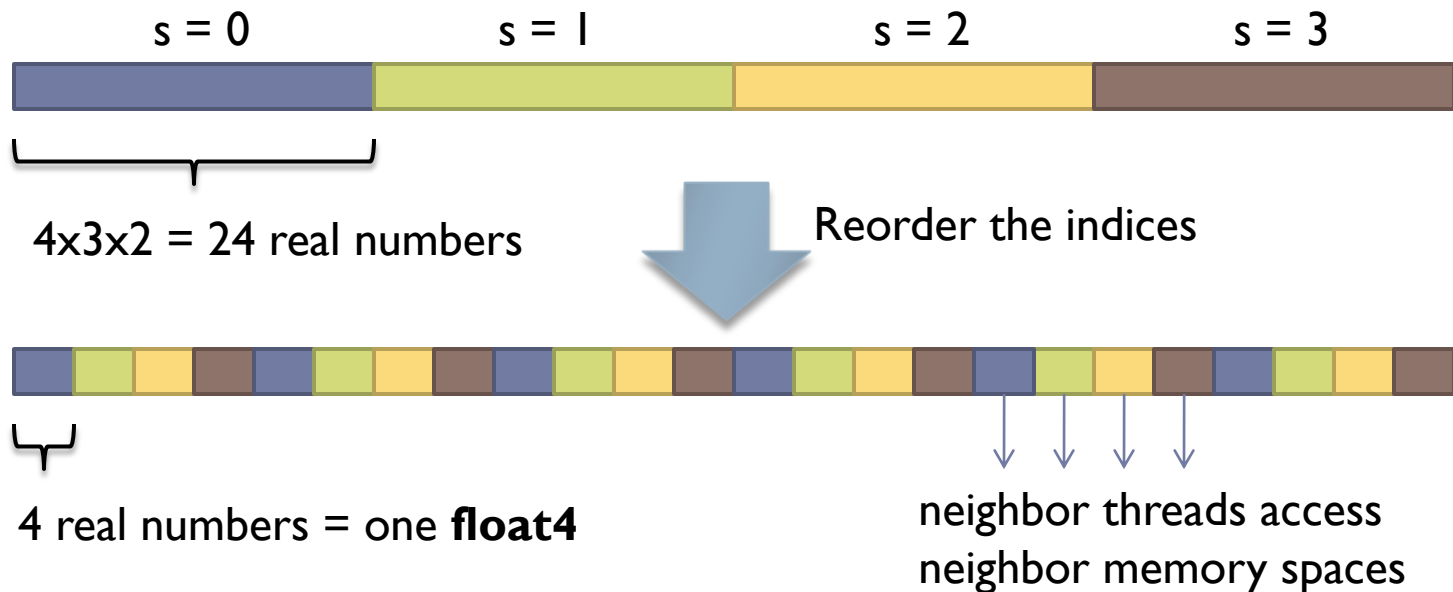
$\alpha := \rho / (v_0, v_0)$ → Parallel reduction of v_0

- ▶ **Addition/subtraction: try to combine these simple operation with existing multiplication kernels, for example:** $v_1 := D_w^{\text{EO}} v_0, \quad r := r - \alpha v_0$

Tuning for Memory Access

- ▶ **Reorder array indices** such that adjacent threads will access adjacent memory spaces.
→ **Better coalesce!**

When $N_s = 4$, for a given point (x, y, z, t) :



Performance

- ▶ Our current kernels can obtain 233 Gflops on NVIDIA GeForce GTX 480.

	Dw (Single)	M5 (Single)	Dw (Double)	M5 (Double)	CG (Mixed)
GTX 285	177	346	33	69	181
GTX 480 *	248	331	32	116	233
C1060	128	290	29	61	132
C2050 *	160	239	22	100	156

* Our code is not yet well-tuned for Fermi.
All numbers in the table are effective GFlops.
Tested with a $16^3 \times 32$ lattice.

- ▶ The bottleneck is Dw single-precision multiplication.

Different DWF Performance Comparison

- ▶ Plaquette action on a $16^3 \times 32$ lattice, $m_\pi \sim 300$ MeV
- ▶ One column of quark propagator is calculated
(One CG with reliable updates)

	ODWF Ns = 16	Borici Ns = 16	DWF Ns = 16	ODWF Ns = 32	Borici Ns = 32	DWF Ns = 32
GTX 285	180	173	164	--	--	--
GTX 480 *	231	224	216	--	--	--
C2050 *	146	140	133	170	167	163
GTX 285	308	87	47	--	--	--
GTX 480 *	241	67	35	--	--	--
C2050 *	379	107	57	1220	478	281

upper: Gflops / lower: time(s)

Different DWF Performance Comparison

► Sign Function Error

$$\max_{\forall Y} \left| \frac{Y^\dagger [1 - S^2(H)] Y}{Y^\dagger Y} \right|, \quad S(H) \simeq \frac{H}{\sqrt{H^2}}$$

Sign Func Error	$\sim 10^{-7}$	$\sim 10^{-4}$	$\sim 10^{-4}$	$\sim 10^{-10}$	$\sim 10^{-6}$	$\sim 10^{-6}$
	ODWF Ns = 16	Borici Ns = 16	DWF Ns = 16	ODWF Ns = 32	Borici Ns = 32	DWF Ns = 32
GTX 285	180	173	164	--	--	--
GTX 480 *	231	224	216	--	--	--
C2050 *	146	140	133	170	167	163
GTX 285	308	87	47	--	--	--
GTX 480 *	241	67	35	--	--	--
C2050 *	379	107	57	1220	478	281

upper: Gflops / lower: time(s)

Summary

- ▶ We have implemented an efficient GPU-based CG solver for generalized domain-wall fermions.
- ▶ On NVIDIA GeForce GTX 480, our CG solver attains 233 Gflops (sustained).
- ▶ CUDA kernels are tuned in several ways, including to separate Dw and M5, to reorder indices, to reuse data, and to expand short loops etc.
- ▶ Optimal domain-wall fermion provides a viable framework to simulate lattice QCD with optimal chiral symmetry, especially with our GPU-based CG solver.

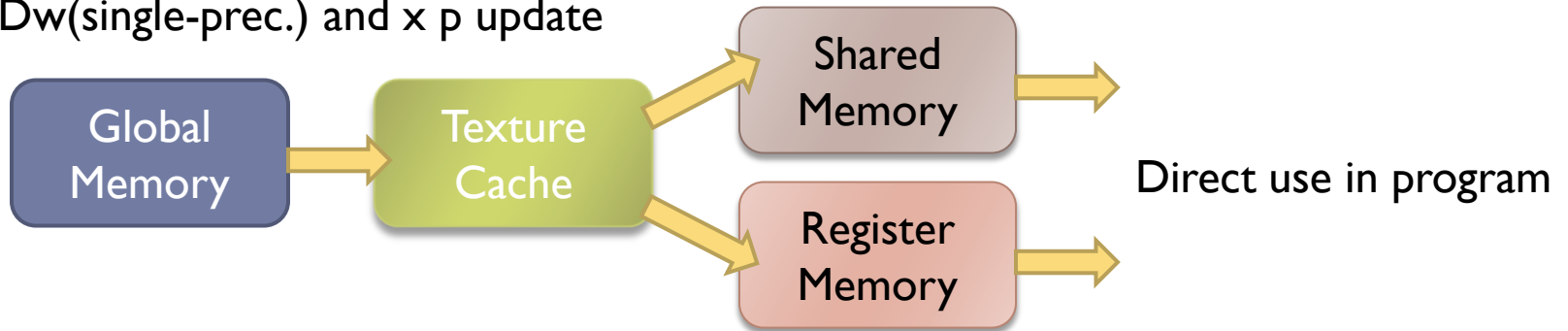
Thanks for your attention!

Lattice 2010, June 18, 2010

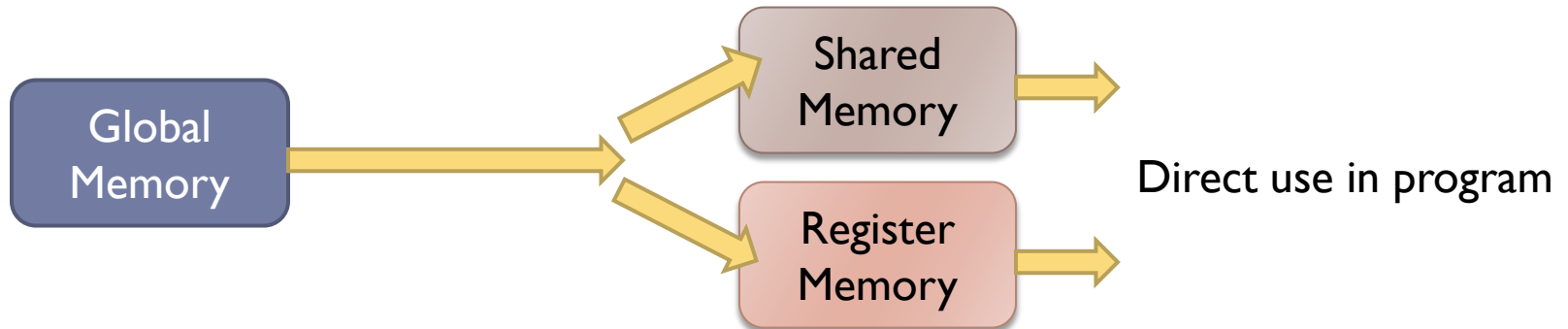
Tuning for Memory Access (2)

► Use **texture** and shared memory

Use by Dw(single-prec.) and x p update



Use by M5(single-prec.), and all double prec. kernels



Kernel Code Generator

- ▶ Python is used to generate the source code of some kernels because
 1. Expanding some short loops can enhance the performance.
 2. When doing parallel reduction, number of addition operations can be controlled.
 3. Much easier to maintain the code and to change lattice size.

Tuning for “Fermi”

- ▶ Recently NVIDIA released a new CUDA compute architecture “Fermi.”
- ▶ Some important changes include:
 1. One can choose between 16KB/48KB shared memory/L1 cache or vice versa.
 2. Global memory is now cached, and accesses are processed per warp.
 3. Shared memory now has 32 banks and accesses are processed per warp.

16 KB Shared m.
64 KB L1 cache

OR

64 KB Shared m.
16 KB L1 cache

[Tuning CUDA™ Applications for Fermi™, Version 1.0]

“Fermi” – L1 Cache

- ▶ All the 3 points mentioned are related to L1 cache. L1 cache has following properties:
 1. Same on-chip memory is used for both L1 and shared memory . (one is 16KB, another is 48KB)
 2. Global memory is cached in L1.
(can be disabled in compiler option)
 3. Local memory is also cached in L1.
 4. L1 cache has higher bandwidth than texture cache.

[Tuning CUDA™ Applications for Fermi™, Version 1.0]

“Fermi” – L1 Cache

▶ L1 cache benchmark on GTX 480

Global cache	L1 cache	Dw (Single)	M5 (Single)	Dw (Double)	M5 (Double)
L1/L2	48 KB	266	313	36	127
L1/L2	16 KB	183	313	32	95
L2 only	48 KB	266	313	38	132
L2 only	16 KB	183	313	32	98

All numbers in the table are estimated effective Gflops.

- ▶ Local resource used by Dw kernel is very much.
→ Larger L1 cache gives better performance.
- ▶ Texture has been used for single-precision Dw.
→ Global cache does not affect.

Further Possible Tuning Methods

▶ Half-Precision

- ▶ The precision is too low if using CUDA built-in function `__float2half_rn()`. A home-made function is needed.

▶ Link variable

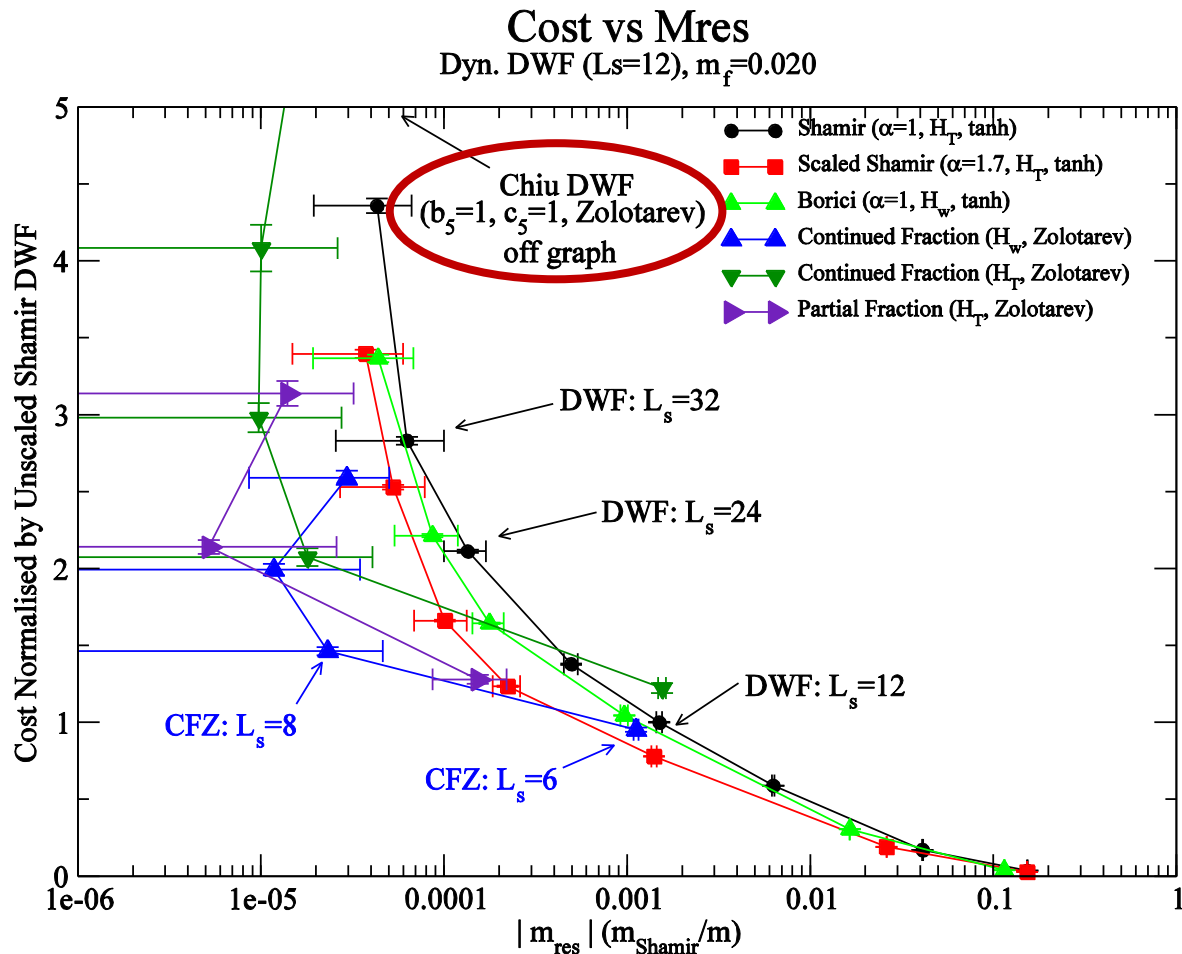
- ▶ Gauge fixing
- ▶ Reducing I/O (use only 12 or 8 real numbers to store one SU(3) matrix)

▶ Dynamical switched mixed-precision CG

- ▶ Reliable updates scheme is faster for most of cases, but sometimes it does not converge correctly.

Different DWF Performance Comparison

► In Lattice 2005: [R. G. Edwards et al., arXiv:hep-lat/0510086v2]



How GPU
enables us to
use ODWF

