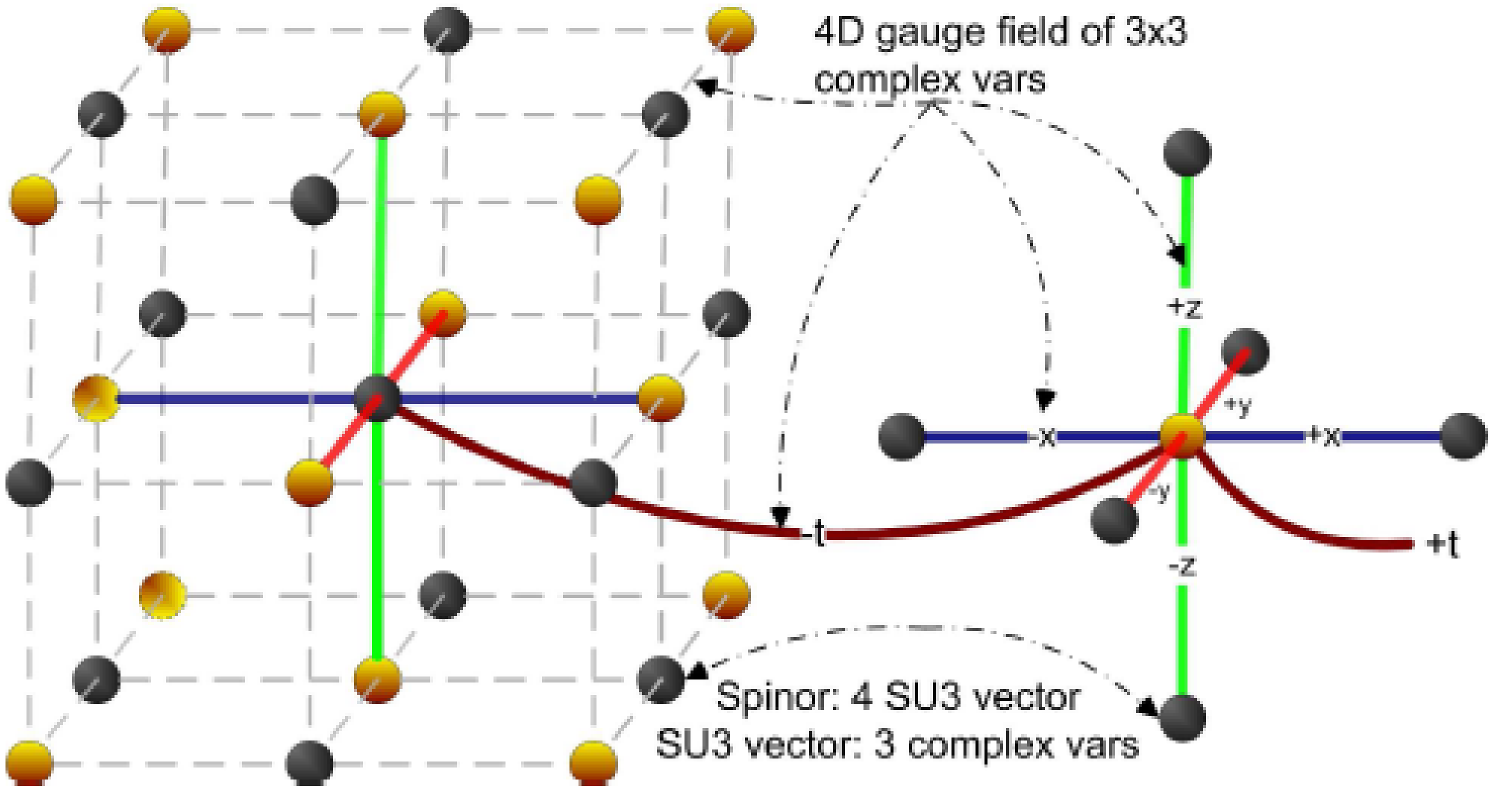
GPU Implementation of CG solver for MILC

Guochun Shi

Innovative Systems Laboratory



Four dimensional space-time Lattice QCD.

Lattice QCD: Solving the following linear system

$$M\phi = b$$

where $\phi_{i,x}$ and $b_{i,x}$ are complex vectors carrying a color index $i = 1, 2, 3$ and a four-dimensional lattice coordinate x. The matrix M is given by

$$M = 2maI + D$$

where I is the identity matrix, $2ma$ is a constant, and the matrix D (called "D slash") is given by

$$D_{j,y;i,x} = \sum_{\mu=1}^{4} [U_{j,i,x,\mu} \delta_{x,y+\hat{\mu}} - U^{\dagger}_{j,i,x,\mu} \delta_{y,x+\hat{\mu}}]$$

The linear system (3) is solved using a conjugate gradient method after recasting it in the positive definite form

$$M^{\dagger} M \phi = M^{\dagger} b.$$

where

$$M^{\dagger} M = (2ma)^2 I + D^{\dagger} D$$

The wilson dslash operator

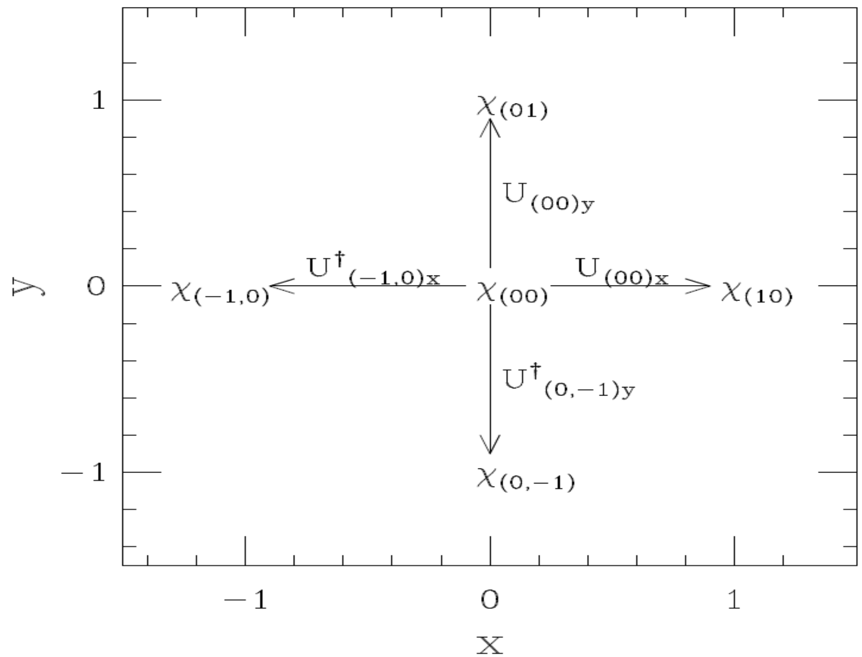


Figure 1: Gathers to a site at the origin in the `dslash` operation. Two dimensions are shown for simplicity. The full problem requires four dimensions.

$$b_1 = U^{\dagger}_{(-1,0)x} \chi_{(-1,0)} \quad , \quad b_2 = U^{\dagger}_{(0,-1)\,y} \chi_{(0,-1)} .$$

$$a = U_{(0,0)x} \chi_{(1,0)} + U_{(0,0)y} \chi_{(0,1)}$$

$$\psi(0,0) = a - b_1 - b_2$$

BU code: dslash reference implementation in CPU

```
template <typename sFloat, typename gFloat>
void dslashReference(sFloat *res, gFloat **gaugeFull, sFloat *spinorField, int oddBit, int daggerBit) {
  for (int i=0; i<Vh*4*3*2; i++) res[i] = 0.0;

  gFloat *gaugeEven[4], *gaugeOdd[4];
  for (int dir = 0; dir < 4; dir++) {
    gaugeEven[dir] = gaugeFull[dir];
    gaugeOdd[dir]  = gaugeFull[dir]+Vh*gaugeSiteSize;
  }

  for (int i = 0; i < Vh; i++) {
    for (int dir = 0; dir < 8; dir++) {
      gFloat *gauge = gaugeLink(i, dir, oddBit, gaugeEven, gaugeOdd);
      sFloat *spinor = spinorNeighbor(i, dir, oddBit, spinorField);

      sFloat projectedSpinor[4*3*2], gaugedSpinor[4*3*2];
      int projIdx = 2*(dir/2)+(dir+daggerBit)%2;
      multiplySpinorByDiracProjector(projectedSpinor, projIdx, spinor);

      for (int s = 0; s < 4; s++) {
        if (dir % 2 == 0)
          su3Mul(&gaugedSpinor[s*(3*2)], gauge, &projectedSpinor[s*(3*2)]);
        else
          su3Tmul(&gaugedSpinor[s*(3*2)], gauge, &projectedSpinor[s*(3*2)]);
      }

      sum(&res[i*(4*3*2)], &res[i*(4*3*2)], gaugedSpinor, 4*3*2);
    }
  }
}
```

BU code: GPU kernel code (x+ direction)

```
{
    // Projector P0-
    // 1 0 0 -i
    // 0 1 -i 0
    // 0 i 1 0
    // i 0 0 1

    int sp_idx = ((x1==X1m1) ? X-X1m1 : X+1) >> 1;
    int ga_idx = sid;

    // read gauge matrix from device memory
    READ_GAUGE_MATRIX(GAUGE0TEX, 0);

    // read spinor from device memory
    READ_SPINOR(SPINORTEX);

    // reconstruct gauge matrix
    RECONSTRUCT_GAUGE_MATRIX(0);

    // project spinor into half spinors
    spinorFloat a0_re = +i00_re+i30_im;
    spinorFloat a0_im = +i00_im-i30_re;
    spinorFloat a1_re = +i01_re+i31_im;
    spinorFloat a1_im = +i01_im-i31_re;
    spinorFloat a2_re = +i02_re+i32_im;
    spinorFloat a2_im = +i02_im-i32_re;

    spinorFloat b0_re = +i10_re+i20_im;
    spinorFloat b0_im = +i10_im-i20_re;
    spinorFloat b1_re = +i11_re+i21_im;
    spinorFloat b1_im = +i11_im-i21_re;
    spinorFloat b2_re = +i12_re+i22_im;
    spinorFloat b2_im = +i12_im-i22_re;

    // multiply row 0
    spinorFloat A0_re = + (g00_re * a0_re - g00_im * a0_im) + (g01_re * a1_re - g01_im * a1_im) + (g02_re * a2_re - g02_im * a2_im);
    spinorFloat A0_im = + (g00_re * a0_im + g00_im * a0_re) + (g01_re * a1_im + g01_im * a1_re) + (g02_re * a2_im + g02_im * a2_re);
    spinorFloat B0_re = + (g00_re * b0_re - g00_im * b0_im) + (g01_re * b1_re - g01_im * b1_im) + (g02_re * b2_re - g02_im * b2_im);
    spinorFloat B0_im = + (g00_re * b0_im + g00_im * b0_re) + (g01_re * b1_im + g01_im * b1_re) + (g02_re * b2_im + g02_im * b2_re);

    // multiply row 1
    spinorFloat A1_re = + (g10_re * a0_re - g10_im * a0_im) + (g11_re * a1_re - g11_im * a1_im) + (g12_re * a2_re - g12_im * a2_im);
    spinorFloat A1_im = + (g10_re * a0_im + g10_im * a0_re) + (g11_re * a1_im + g11_im * a1_re) + (g12_re * a2_im + g12_im * a2_re);
    spinorFloat B1_re = + (g10_re * b0_re - g10_im * b0_im) + (g11_re * b1_re - g11_im * b1_im) + (g12_re * b2_re - g12_im * b2_im);
    spinorFloat B1_im = + (g10_re * b0_im + g10_im * b0_re) + (g11_re * b1_im + g11_im * b1_re) + (g12_re * b2_im + g12_im * b2_re);

    // multiply row 2
    spinorFloat A2_re = + (g20_re * a0_re - g20_im * a0_im) + (g21_re * a1_re - g21_im * a1_im) + (g22_re * a2_re - g22_im * a2_im);
    spinorFloat A2_im = + (g20_re * a0_im + g20_im * a0_re) + (g21_re * a1_im + g21_im * a1_re) + (g22_re * a2_im + g22_im * a2_re);
    spinorFloat B2_re = + (g20_re * b0_re - g20_im * b0_im) + (g21_re * b1_re - g21_im * b1_im) + (g22_re * b2_re - g22_im * b2_im);
    spinorFloat B2_im = + (g20_re * b0_im + g20_im * b0_re) + (g21_re * b1_im + g21_im * b1_re) + (g22_re * b2_im + g22_im * b2_re);

    o00_re += A0_re;
    o00_im += A0_im;
    o10_re += B0_re;
    o10_im += B0_im;
    o20_re -= B0_im;
    o20_im += B0_re;
    o30_re -= A0_im;
    o30_im += A0_re;
```

Disclaimer

- The source code is from Bosten University's Quda package.
- The diagrams/formulas are from two papers
  - C. Bernarda, C. DeTarb, S. Gottliebc, U.M. Hellerd, J. Hetricke, N. Ishizukaa, L. K¨arkk¨ainenf , S.R. Lantzg, K. Rummukainenc, R. Sugarh, D. Toussainte and M. Wingatei, "**Lattice QCD on the IBM Scalable POWERParallel Systems SP2"**
  - K. Z. Ibrahim, F. Bodin, "**Efficient SIMDization and Data Management of the Lattice QCD Computation on the Cell Broadand Engine**"

CG in BU code

Standard CG procedure from "A*n Introduction* to the *Conjugate Gradient Method*. *Without the Agonizing Pain*" : **A x = b**

δ_{new} ➔ r2

δ_{old} ➔ r2_old

d ➔ p

A ➔ $\tilde{M}^\dagger \tilde{M}$. (where $\tilde{M}$ is the preconditioned matrix)

$$M = \begin{pmatrix} \mathbf{1}_{E \leftarrow E} & -\kappa D\!\!\!/_{E \leftarrow O} \\ -\kappa D\!\!\!/_{O \leftarrow E} & \mathbf{1}_{O \leftarrow O} \end{pmatrix}$$

$$L = \begin{pmatrix} \mathbf{1}_{E \leftarrow E} & \mathbf{0} \\ -\kappa D\!\!\!/_{O \leftarrow E} & \mathbf{1}_{O \leftarrow O} \end{pmatrix} \quad U = \begin{pmatrix} \mathbf{1}_{E \leftarrow E} & -\kappa D\!\!\!/_{E \leftarrow O} \\ \mathbf{0} & \mathbf{1}_{O \leftarrow O} \end{pmatrix}$$

$$\tilde{M} = L^{-1} M U^{-1} = \begin{pmatrix} \mathbf{1}_{E \leftarrow E} & \mathbf{0} \\ \mathbf{0} & \mathbf{1}_{O \leftarrow O} - \kappa^2 D\!\!\!/_{O \leftarrow E} D\!\!\!/_{E \leftarrow O} \end{pmatrix}$$

```
i ⇐ 0
r ⇐ b − Ax
d ⇐ r
δ_new ⇐ r^T r
δ_0 ⇐ δ_new
While i < i_max and δ_new > ε^2 δ_0 do
    q ⇐ Ad
    α ⇐ δ_new / (d^T q)
    x ⇐ x + αd
    If i is divisible by 50
        r ⇐ b − Ax
    else
        r ⇐ r − αq
    δ_old ⇐ δ_new
    δ_new ⇐ r^T r
    β ⇐ δ_new / δ_old
    d ⇐ r + βd
    i ⇐ i + 1
```

CG code in BU code

```
while (r2 > stop && k<perf->maxiter) {
  MatPCDagMatPCCuda(Ap, gaugeSloppy, p, perf->kappa, tmp_sloppy, perf->matpc_type);

  pAp = reDotProductCuda(p, Ap);

  alpha = r2 / pAp;
  r2_old = r2;
  r2 = axpyNormCuda(-alpha, Ap, r_sloppy);

  // reliable update conditions
  rNorm = sqrt(r2);
  if (rNorm > maxrx) maxrx = rNorm;
  if (rNorm > maxrr) maxrr = rNorm;
  int updateX = (rNorm < delta*r0Norm && r0Norm <= maxrx) ? 1 : 0;
  int updateR = ((rNorm < delta*maxrr && r0Norm <= maxrr) || updateX) ? 1 : 0;

  if (!updateR) {
    beta = r2 / r2_old;
    axpyZpbxCuda(alpha, p, x_sloppy, r_sloppy, beta);
  } else {
    axpyCuda(alpha, p, x_sloppy);

    if (x.precision != x_sloppy.precision) copyCuda(x, x_sloppy);

    MatPCDagMatPCCuda(r, gaugePrecise, x, invert_param->kappa,
                      tmp, invert_param->matpc_type);

    r2 = xmyNormCuda(b, r);
    if (x.precision != r_sloppy.precision) copyCuda(r_sloppy, r);
    rNorm = sqrt(r2);

    maxrr = rNorm;
    rUpdate++;

    if (updateX) {
      xpyCuda(x, y);
      zeroCuda(x_sloppy);
      copyCuda(b, r);
      r0Norm = rNorm;

      maxrx = rNorm;
      xUpdate++;
    }

    beta = r2 / r2_old;
    xpayCuda(r_sloppy, beta, p);
  }
```

Different solution types solve different equations

QUDA_MAT_SOLUTION

$Mx = b$ ➔ $\tilde{M}^\dagger \tilde{M} x' = b'$ *where* $b' = \tilde{M}^\dagger L b$ $x = U x'$

QUDA_MATPC_SOLUTION

$\tilde{M}^\dagger x = b$ ➔ $\tilde{M}^\dagger \tilde{M} x = b'$ *where* $b' = \tilde{M}^\dagger b$

QUDA_MATPCDAG_MATPC_SOLUTION

$\tilde{M}^\dagger \tilde{M} x = b$ ➔ the same

Staggered Dslash reference Implementation

```
template <typename sFloat, typename gFloat>
void dslashReference_st(sFloat *res, gFloat **fatlink, gFloat** longlink, sFloat *spinorField, int oddBit, int daggerBit)
{
    for (int i=0; i<Vh*1*3*2; i++) res[i] = 0.0;

    gFloat *fatlinkEven[4], *fatlinkOdd[4];
    gFloat *longlinkEven[4], *longlinkOdd[4];

    for (int dir = 0; dir < 4; dir++) {
        fatlinkEven[dir] = fatlink[dir];
        fatlinkOdd[dir] = fatlink[dir] + Vh*gaugeSiteSize;
        longlinkEven[dir] =longlink[dir];
        longlinkOdd[dir] = longlink[dir] + Vh*gaugeSiteSize;
    }
    for (int i = 0; i < Vh; i++) {
        for (int dir = 0; dir < 8; dir++) {
            gFloat* fatlnk = gaugeLink_st(i, dir, oddBit, fatlinkEven, fatlinkOdd, 1);
            gFloat* longlnk = gaugeLink_st(i, dir, oddBit, longlinkEven, longlinkOdd, 3);

            sFloat *first_neighbor_spinor = spinorNeighbor(i, dir, oddBit, spinorField, 1);
            sFloat *third_neighbor_spinor = spinorNeighbor(i, dir, oddBit, spinorField, 3);

            sFloat gaugedSpinor[spinorSiteSize];

            if (dir % 2 == 0){
                su3Mul(gaugedSpinor, fatlnk, first_neighbor_spinor);
                sum(&res[i*spinorSiteSize], &res[i*spinorSiteSize], gaugedSpinor, spinorSiteSize);
                su3Mul(gaugedSpinor, longlnk, third_neighbor_spinor);
                sum(&res[i*spinorSiteSize], &res[i*spinorSiteSize], gaugedSpinor, spinorSiteSize);
            }
            else{
                su3Adjmul(gaugedSpinor, fatlnk, first_neighbor_spinor);
                sub(&res[i*spinorSiteSize], &res[i*spinorSiteSize], gaugedSpinor, spinorSiteSize);
                su3Adjmul(gaugedSpinor, longlnk, third_neighbor_spinor);
                sub(&res[i*spinorSiteSize], &res[i*spinorSiteSize], gaugedSpinor, spinorSiteSize);
            }
        }
    }
}
```

Staggered Dslash GPU implementation

- Similar to the Wilson Dslash
- Link representation is the same
  - Use 5 float4 to represent 3x3 complex matrix (18 floats used, 2 floats ununsed)
  - But staggered Dslash has 2 links, wilson has 1 link only

- Spinor representation differ slightly
  - Wilson: 6 float4 to represent 4x3 complex matrix (total 24 floats)
  - Stagger: 3 float2 to represent 1x3 complex vector (total 6 floats)

Preliminary Results

- GPU results and CPU reference does not match (yet)
- Flops per site: CM(complex multiplication)=6, CA(complex addtion)=2
  - 3*(3*CM+2*CA)*8*2 + 15*(3*CA) = 1146 flops
- In/Out bytes (12 construct):
  - ((8*6*2) + 6 + (8*12*2)) * sizeof(float) = 1176 bytes
- Nvidia GTX 280
  - GFLOPS: 106.7
  - Bandwidth: 102.0 GB/s

Preliminary Results (single precision only)

- GPU and CPU results agree
  - Fixed small errors in both CPU and GPU code
- Conjugate Gradient (CG) works
  - Solves $\tilde{M}^{\dagger}\tilde{M}x = b$ where $\tilde{M} = L^{-1}MU^{-1}$

$$= \begin{pmatrix} \mathbf{1}_{E \leftarrow E} & \mathbf{0} \\ \mathbf{0} & \mathbf{1}_{O \leftarrow O} - \kappa^2 \not{D}_{O \leftarrow E} \not{D}_{E \leftarrow O} \end{pmatrix}$$

  - 93 Gflops with GTX280

What’s next

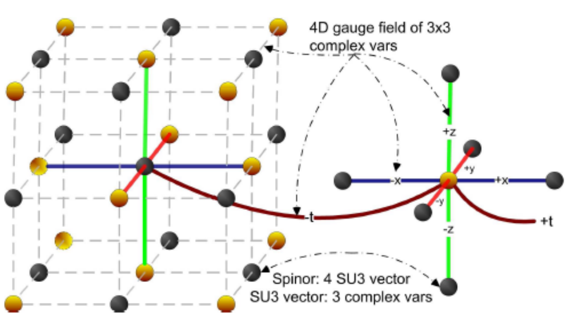
- Optimizing the single precision version in GPU
- Make other flavors work
  - 8 reconstruct
  - Double precision/half precision, especially double precision because of next GPU architecture
- Multi-gpu / multi-node implementation for large lattice size
- Incorporating the code into MILC (?)

Staggered Dslash
CPU data layout

- Each site contains:
  - 1 spinor (1x3 complex)
  - 4 fatlink (3x3 complex)
  - 4 longlink (3x3 complex)

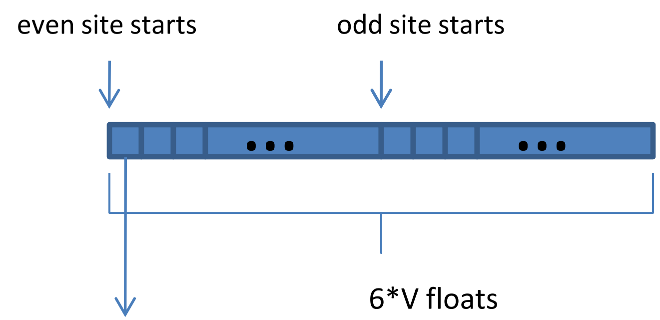
- Sites are divided into even and odd sites. For site (x,y,z,t)
  - (x+y+z+t)%2 == 0 → even site
  - (x+y+z+t)%2 ==1 → odd site

- Total number of sites
  - V = dimX * dimY * dimZ * dimT
  - Half of total sites Vh = V/2



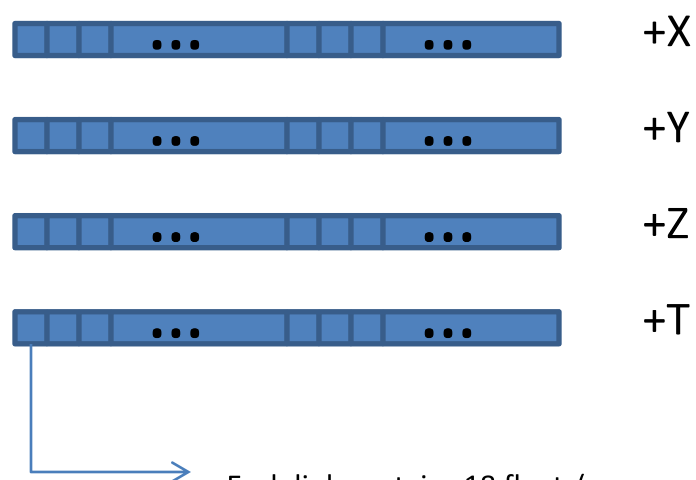
Four dimensional space-time Lattice QCD.

spinor



Each spinor contains 6 (3*2) floats

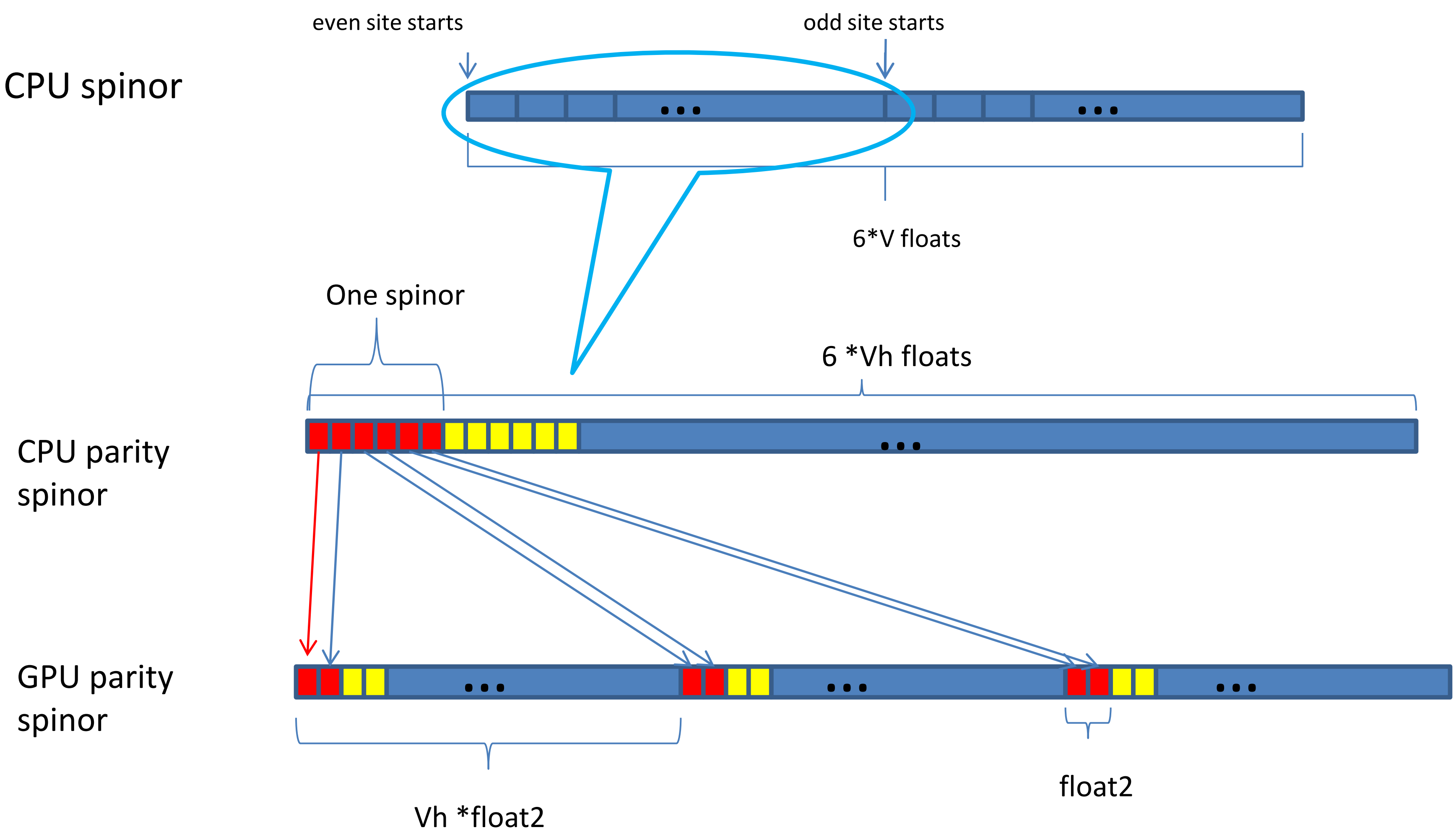
Fatlink:
Array of pointers



Each link contains 18 floats(3x3x2)

Longlink:
Same as fatlink

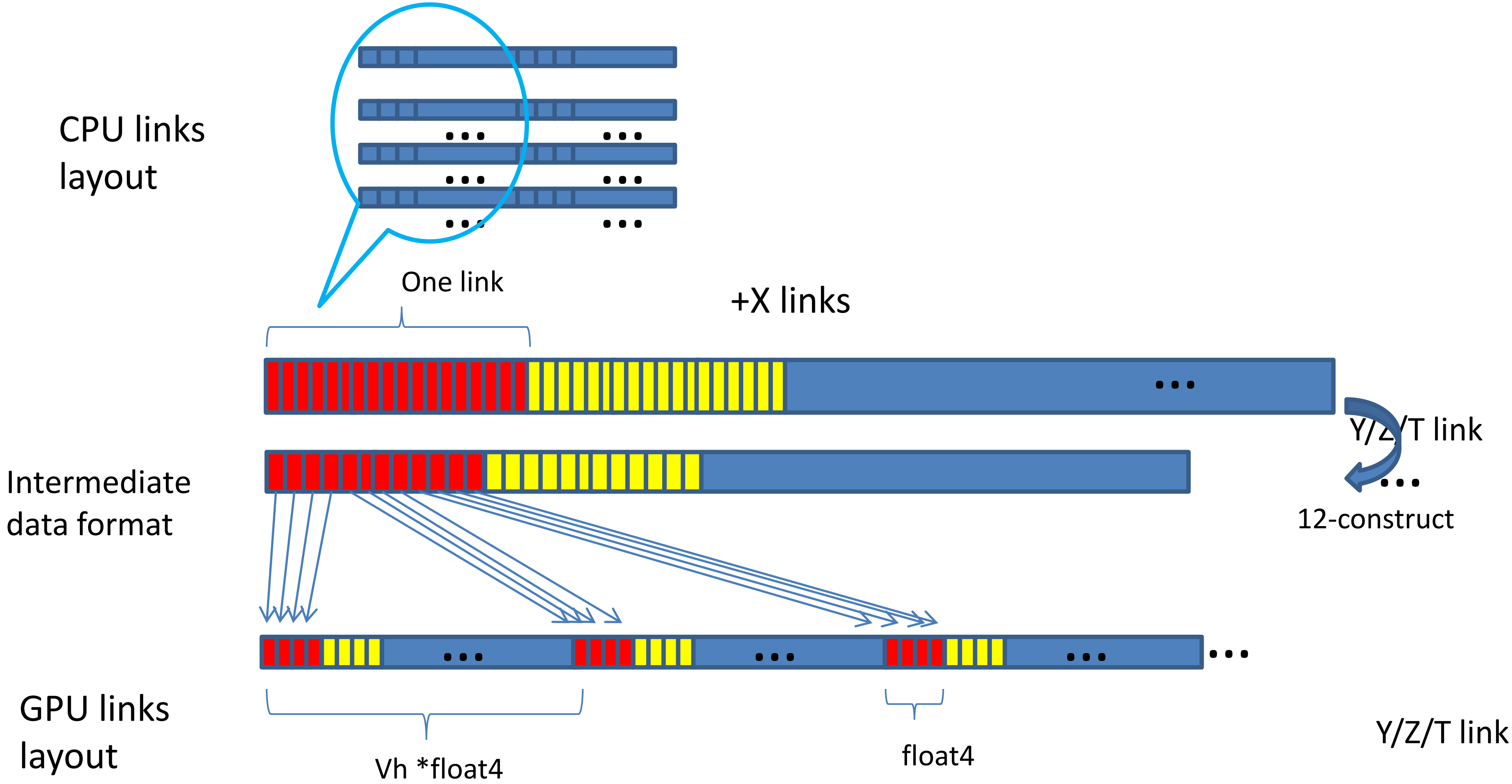
Spinor CPU-> GPU mapping



GPU kernel code to read spinor

```
#define READ_SPINOR_SINGLE(spinor)                        \
  float2 I0 = tex1Dfetch((spinor), sp_idx + 0*Vh);    \
  float2 I1 = tex1Dfetch((spinor), sp_idx + 1*Vh);    \
  float2 I2 = tex1Dfetch((spinor), sp_idx + 2*Vh);
```

Link CPU-> GPU mapping



GPU kernel code to read link

```
#define READ_FAT_MATRIX_12_SINGLE(gauge, dir, idx)                    \
  float4 FAT0 = tex1Dfetch((gauge), idx + ((dir/2)*3+0)*Vh);     \
  float4 FAT1 = tex1Dfetch((gauge), idx + ((dir/2)*3+1)*Vh);     \
  float4 FAT2 = tex1Dfetch((gauge), idx + ((dir/2)*3+2)*Vh);     \
  float4 FAT3 = make_float4(0,0,0,0);                            \
  float4 FAT4 = make_float4(0,0,0,0);
```

Progress in last week

- 8 reconstruct works (for long link), full load for fat link works
  - Long link is loaded using n (n=2 or 3) float4
  - Fat link is loaded using m(m=9) float2, no bandwidth wasted
- Performance (8 reconstruct for long link, full load with fat link)
  - Dslash
    - 97 Gflops, bandwidth achieved 97.9 GB/s
  - CG
    - 86.7 Gflops

optimization using shared memory

- Link is not shared in the Dslash computation
- Each spinor is shared 16 times
  - Since the majority of the bandwidth requirement comes from links, there is an upper limit even we share the spinor perfectly, i.e. each spinor is only loaded once
  - Normal data requirement for each site (12-reconstruct):
    - (8*6*2+6)+ 8*18+8*12= 342 bytes
  - The “best” spinor shared stragety can reduce that to
    - (1*6+6)+8*18+8*12= 252 bytes, leading to 26.3% improvement
  - Shared memory size is limited ➔the number of spinor in shared memory is limited (16KB can hold 682 spinors, approximately 6^4/2)
    - Need to rearrange data
    - Probably need to use 4-D “tile” to scan through spinors
    - Implementation nontrivial
  - Low priority task

Progress in last week

- Double/single/half precision all works
  - Need to know the range of long link values in order to implement half precision, now assume [-1, 1]
  - Mixed precision for spinor/gauge should work, not tested completely yet
  - The sloppy precision should also work in CG but not tested completely yet
  - Bug fix: feedback to BU

Dslash performance (GFLOPS and bandwidth)

	Double	Single	half
8-reconstruct	17.4 (35.1)	97.1(97.9)	152.5(76.9)
12-reconstruct	32(71.1)	87.6(97.4)	143.8(80)

CG performance (GFLOPS)

	Double	Single	half
8-reconstruct	16.6	87.8	134.7
12-reconstruct	30	78.3	126.5
Converge steps	63	64	90

All tests running with 24^3 * 32 lattice with GTX280

CG performance

- (spinor, link, recon, spinor_sloppy, link_sloppy, recon_sloppy): total 108 combinations
- Some typical flavor performance is shown in the table below
  - Residual is determined by higher accuracy spinor/link/recon
  - Gflops and iterations are determined by sloppy spinor/link/recon

Spinor	link	recon	Spinor sloppy	Link sloppy	Recon sloppy	residual	gflops	iterarions
double	double	12	double	double	12	1.88e-12	29.97	63
double	double	12	single	single	8	1.88e-12	79.58	64
double	double	12	half	half	8	2.02e-12	116.46	69
single	single	8	single	single	8	3.29e-07	86.68	64
single	single	8	half	half	8	3.30e-07	130.61	72
half	half	8	half	half	8	1.6e-03	134.91	90

CG in MILC

Standard CG procedure from "A*n Introduction* to the *Conjugate Gradient Method. Without the Agonizing Pain*" : **A x = b**

δ_{new} ➔ rsq

δ_{old} ➔ oldrsq

d ➔ cg_p

A ➔ $M^{\dagger}M$ (where $M = \not{D} + 2m$)

q ➔ - ttt

r ➔ resid

α ➔ a

β ➔ beta

x ➔ dest

b ➔ src

$i \Leftarrow 0$
$r \Leftarrow b - Ax$
$d \Leftarrow r$
$\delta_{new} \Leftarrow r^T r$
$\delta_0 \Leftarrow \delta_{new}$
While $i < i_{max}$ and $\delta_{new} > \varepsilon^2 \delta_0$ do
    $q \Leftarrow Ad$
    $\alpha \Leftarrow \frac{\delta_{new}}{d^T q}$
    $x \Leftarrow x + \alpha d$
    If i is divisible by 50
        $r \Leftarrow b - Ax$
    else
        $r \Leftarrow r - \alpha q$
    $\delta_{old} \Leftarrow \delta_{new}$
    $\delta_{new} \Leftarrow r^T r$
    $\beta \Leftarrow \frac{\delta_{new}}{\delta_{old}}$
    $d \Leftarrow r + \beta d$
    $i \Leftarrow i + 1$

$$M^{\dagger}Mx = b$$

```
/* main loop - do until convergence or time to restart */
    /*
       oldrsq <- rsq
       ttt <- (-1)*M_adjoint*M*cg_p
       pkp <- (-1)*cg_p.M_adjoint*M.cg_p
       a <- -rsq/pkp
       dest <- dest + a*cg_p
       resid <- resid + a*ttt
       rsq <- |resid|^2
       b <- rsq/oldrsq
       cg_p <- resid + b*cg_p
    */
do{
    oldrsq = rsq;
    pkp = 0.0;
    /* sum of neighbors */

    if(special_started==0){
        dslash_fn_field_special( cg_p, ttt, l_otherparity, tags2, 1 );
        dslash_fn_field_special( ttt, ttt, l_parity, tags1, 1);
        special_started=1;
    }
    else {
        dslash_fn_field_special( cg_p, ttt, l_otherparity, tags2, 0 );
        dslash_fn_field_special( ttt, ttt, l_parity, tags1, 0);
    }

    /* finish computation of M_adjoint*m*p and p*M_adjoint*m*Kp */
    /* ttt  <- ttt - msq_x4*cg_p     (msq = mass squared) */
    /* pkp  <- cg_p.(ttt - msq*cg_p) */
    pkp = 0.0;
    FORSOMEPARITY(i,s,l_parity){
      if( i < loopend-FETCH_UP ){
        prefetch_VV( &ttt[i+FETCH_UP], &cg_p[i+FETCH_UP] );
      }
      scalar_mult_add_su3_vector( &ttt[i], &cg_p[i], -msq_x4,
                                  &ttt[i] );
      pkp += (double)su3_rdot( &cg_p[i], &ttt[i] );
    } END_LOOP
    g_doublesum( &pkp );
    iteration++;
    total_iters++;

    a = (Real) (-rsq/pkp);

    /* dest <- dest - a*cg_p */
    /* resid <- resid - a*ttt */
    rsq=0.0;
    FORSOMEPARITY(i,s,l_parity){
      if( i < loopend-FETCH_UP ){
        prefetch_VVVV( &t_dest[i+FETCH_UP],
                       &cg_p[i+FETCH_UP],
                       &resid[i+FETCH_UP],
                       &ttt[i+FETCH_UP] );
      }
      scalar_mult_add_su3_vector( &t_dest[i], &cg_p[i], a, &t_dest[i] );
      scalar_mult_add_su3_vector( &resid[i], &ttt[i], a, &resid[i]);
      rsq += (double)magsq_su3vec( &resid[i] );
    } END_LOOP
        g_doublesum(&rsq);
#ifdef CG_DEBUG
        if(mynode()==0){printf("iter=%d, rsq= %e, pkp=%e\n",
           iteration,(double)rsq,(double)pkp);fflush(stdout);}
#endif

        if( rsq <= rsqstop ){
          /* copy t_dest back to site structure */
          FORSOMEPARITY(i,s,l_parity){
                  *(su3_vector *)F_PT(s,dest) = t_dest[i];
          } END_LOOP
            /* if parity==EVENANDODD, set up to do odd sites and go back */
            if(parity == EVENANDODD) {
                l_parity=ODD; l_otherparity=EVEN;
                parity=EVEN;    /* so we won't loop endlessly */
                iteration = 0;
#ifdef CG_DEBUG
                node0_printf("normal goto start\n");
#endif
                goto start;
            }
            *final_rsq_ptr=(Real)rsq;
            if(special_started==1) {
              cleanup_gathers(tags1,tags2);
              special_started = 0;
            }

#ifdef CG_DEBUG
            node0_printf("normal return\n"); fflush(stdout);
#endif
#ifdef CGTIME
 dtimec += dclock();
if(this_node==0){
printf("CONGRAD5: time = %e (fn) iters = %d mflops = %e\n",
dtimec,iteration,(double)(nflop*volume*iteration/(1.0e6*dtimec*numnodes())) );
fflush(stdout);}
#endif
            cleanup_dslash_temps();
            free(ttt); free(cg_p); free(resid); free(t_dest); first_congrad = 1;
             return (iteration);
        }

        b = (Real)rsq/oldrsq;
        /* cg_p  <- resid + b*cg_p */
        FORSOMEPARITY(i,s,l_parity){
           scalar_mult_add_su3_vector( &resid[i],
                                       &cg_p[i] , b , &cg_p[i]);
        } END_LOOP

    } while( iteration%niter != 0);
```

Interface function to MILC

```
int ks_congrad( field_offset src, field_offset dest, Real mass,
                int niter, int nrestart, Real rsqmin, int prec,
                int parity, Real *final_rsq_ptr )
```

- b in the @src
- Guess solution in @dest
- Solve

$$M^{\dagger}Mx = b$$

Direct CG performance(I)

- Solve $M^{\dagger}Mx = b$ instead of $\tilde{M}^{\dagger}\tilde{M}x = b$
- Some typical flavor performance is shown in the table below
  - Some combination does not converge after maximum(9999) iterations, e.g. (--sprec double --gprec double --recon 12 --sprec_sloppy half --gprec_sloppy double --recon_sloppy 12) .
  - All non-converging runs involve half precision

CPU precision	Spinor	link	recon	Spinor sloppy	Link sloppy	Recon sloppy	residual	gflops	iterarions
double	double	double	12	double	double	12	8.34e-13	29.98	88
	double	double	12	single	single	8	9.96e-13	78.94	88
	double	double	12	half	half	8	1.13e-12	130.04	1808
	single	single	8	single	single	8	1.70e-07	83.60	88
	single	single	8	half	half	8	2.93e-07	131.09	1999
	half	half	8	half	half	8	9.63e-04	131.65	3534

Direct CG performance (II)

- CPU in single precision
  - The cpu precision has no effect on the accuracy

CPU precision	Spinor	link	recon	Spinor sloppy	Link sloppy	Recon sloppy	Residual	Gflops	Iterarions
single	single	single	8	single	single	8	4.27e-07	83.66	88
	single	single	8	half	half	8	4.27e-07	130.74	1692
	half	half	8	half	half	8	9.62e-4	131.41	2508

Interface function to MILC

– int ks_congrad_parity_gpu(su3_vector *t_src, su3_vector *t_dest,
  quark_invert_control *qic, Real mass,
  ferm_links_t *fn)
– Replace the function ks_congrad_parity() in MILC 7.6.3
– The program runs
– Results doe not match with CPU
– Reason:
  - the long link is not reconstructed correctly
  - How to do it correctly?

Multi mass CG solver

- Standalone test program works for all precisions
  - All solution precisions are good
- Mixed precision CG solver
  - Only the first solution's accuracy is good, the rest of solutions are as good as the sloppy precision
- Interface function to MILC written but untested
  - Small test input needed