

Domain Wall Fermions and SSE on Penitum 4

Andrew Pochinsky

October 1, 2003

1 INTRODUCTION

It seems that performance bottleneck on current versions of P-4 (and Xeon) lies in the memory bus bandwidth. This bottleneck manifests itself both in single processor code and when communication is present. Therefore, it is imperative to decrease amount of data needed per floating point operation. While very little could be done to improve FLOP/memory ratio for Wilson fermions, domain wall fermions (DWF) offer an extra data reuse opportunity, because the gauge field is replicated along the fifth dimension. This allows one to arrange lattice traversal in such a way as to achieve maximal reuse of gauge fields.

One should bear in mind, that the present code is not a complete DFW Dirac operator, but an illustration of possibilities in conserving memory bandwidth. We only study here effects of traversing the fifth dimension, though. This along allows on to save about 35.3% of memory bandwidth in the limit of infinite L_5 . Further savings are possible if 4-d data is arrange in memory to maximize cache utilization. However, these savings are uncertain at the moment, but could studing in depth if needed.

The code is written in `noweb` framework to preserve reasoning behind particular engineering decisions made throughout. For convenience, both original sources and processed `C++` files are provided in one tarball.

First, notice that DWF Dirac operator still admits the red-black decomposition:

$$D_5\psi = \begin{pmatrix} 1 & K_{eo} \\ K_{oe} & 1 \end{pmatrix} \begin{pmatrix} \psi_e \\ \psi_o \end{pmatrix}$$

For the sake of simplicity, the following γ -matrix basis is used:

$$\begin{aligned}
\gamma_1 &= 1 \otimes \sigma_2 = \begin{pmatrix} 0 & -i & 0 & 0 \\ i & 0 & 0 & 0 \\ 0 & 0 & 0 & -i \\ 0 & 0 & i & 0 \end{pmatrix} \\
\gamma_2 &= \sigma_1 \otimes \sigma_1 = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \\
\gamma_3 &= \sigma_2 \otimes \sigma_1 = \begin{pmatrix} 0 & 0 & 0 & -i \\ 0 & 0 & -i & 0 \\ 0 & i & 0 & 0 \\ i & 0 & 0 & 0 \end{pmatrix} \\
\gamma_4 &= \sigma_3 \otimes \sigma_1 = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & -1 & 0 \end{pmatrix} \\
\gamma_5 &= 1 \otimes \sigma_3 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}
\end{aligned}$$

This allows for some simplification in handling the fifth dimension.

In the following test code we implement periodic boundary conditions in the fifth dimension (so it is not a ready-to-use DWF Dirac operator, but close enough to get a measure of possible performance gains.) Boundary conditions in other four dimensions are best implemented via modifications to the gauge field.

2 LOW LEVEL SSE OPERATIONS

Let us start with a low level SSE operations required for LQCD code. As of `gcc 3.2` there are compiler's builtin functions covering some of functionality we need. Unfortunately, there some omissions too (notable, the shuffle operations.) In addition, in some cases `gcc` does not produce code conforming to the standard in some situations, so the implementation below may seem forced.

```

<qcdsse.hh>≡
#ifdef QCDSSSE_HH
#define QCDSSSE_HH
namespace QCDSSSE {
    <vector floating point type>
    <vector real numbers>
    <vector complex numbers>
};
#endif

```

GCC provides access to vector data types through a rather contoured facility. Fortunately, it only pains the library writer to suffer its odities.

```

<vector floating point type>≡
    typedef float VREAL __attribute__((mode(V4SF),aligned(16)));

```

Now, variable of type `VREAL` will correspond to 4-vectors of single precision floating point numbers packed into 128 bit. GCC provides us with assignment for this datatype. We now need to build a repertoire of operations on them. However, in order not to overstress gcc's typesystem, we package a 4-vector of real number into a structure and implement required operations on top of this abstraction.

```

<vector real numbers>≡
    struct vreal {
        VREAL v;
        <vreal constructors>
    };

```

The first constructor we need is a packager of `VREAL` into `struct vreal`. Though one would expect the compiler generate it automatically, this part is broken with gcc.

```

<vreal constructors>≡
    inline vreal() {}
    inline vreal(const VREAL &a): v(a) {} // -> a[3] a[2] a[1] a[0]

```

We also need a constructor populating all components of the 4-vector with the same value. Some experimentation shows that the best is produced when a constant reference is passed as an arguments. Yet another of gcc's little mysteries.

```

<vreal constructors>+≡
    inline vreal(const float &b) { // -> b b b b
        register VREAL x = __builtin_ia32_loadss((float *)&b);

        asm("shufps\t$0,%0,%0"
            : "+x" (x));
        v = x;
    }

```

By the way, `__builtin_is32_loadss` has a broken prototype: its arguments is clearly `const`. Here we have to cast the attribute away.

Now we are ready for arithmetics. Three operations we need are provided curtesy of Free Software Foundation, all we need to do is repackaging them into something more palatable:

```
<vector real numbers>+≡
inline vreal
operator+(const vreal &a, const vreal &b)
{
    return __builtin_ia32_addps(a.v, b.v);
}

inline vreal
operator-(const vreal &a, const vreal &b)
{
    return __builtin_ia32_subps(a.v, b.v);
}

inline vreal
operator*(const vreal &a, const vreal &b)
{
    return __builtin_ia32_mulps(a.v, b.v);
}
```

We also need two more operations to implement `cshift`'s along the fifth dimension. The shuffle operation allows us to do it in two instructions and places the result conveniently into an SSE register:

```

<vector real numbers>+≡
inline vreal
operator<<(const vreal &a, const vreal &b) // -> a[2] a[1] a[0] b[3]
{
    register VREAL x = a.v;
    register VREAL y = b.v;

    asm("shufps\t$3,%0,%1\n\t"    // y -> [*] a[0] [*] b[3]
        "shufps\t$152,%1,%0"      // x -> a[2] a[1] a[0] b[3]
        : "+x" (x), "+x" (y));
    return x;
}

inline vreal
operator>>(const vreal &a, const vreal &b) // -> a[0] b[3] b[2] b[1]
{
    register VREAL x = a.v;
    register VREAL y = b.v;

    asm("shufps\t$57,%1,%1\n\t"    // y -> [*] b[3] b[2] b[1]
        "shufps\t$36,%1,%0"      // x -> a[0] b[3] b[2] b[1]
        : "+x" (x), "+x" (y));
    return x;
}

```

Now, let us turn to the 4-vectors of complex numbers. First, the vector complex data type:

```

<vector complex numbers>≡
struct vcomplex {
    vreal re, im;
    <vcomplex constructors>
};

```

Making a `vcomplex` from two `vreal` is simple:

```

<vcomplex constructors>≡
inline vcomplex(const vreal &r, const vreal &i): re(r), im(i) {}

```

We also need a constructor from a couple of `float`'s (Again, a particular choice of `const` and reference arguments seems to make gcc happy).

```

<vcomplex constructors>+≡
inline vcomplex(const float &r, const float &i): re(r), im(i) {}

```

There is a bug in gcc that prevents it from generating the correct copy constructor. Here is a fix:

```
<vcomplex constructors>+≡
    inline vcomplex() {}
    inline void operator=(const vcomplex &z) { re = z.re; im = z.im; }
```

For complex operations, we only care about the ring structure (and conjugates)

```
<vector complex numbers>+≡
    inline vcomplex
    operator+(const vcomplex &a, const vcomplex &b)
    {
        return vcomplex(a.re + b.re, a.im + b.im);
    }

    inline vcomplex
    operator-(const vcomplex &a, const vcomplex &b)
    {
        return vcomplex(a.re - b.re, a.im - b.im);
    }

    inline vcomplex
    mul_nn(const vcomplex &a, const vcomplex &b)
    {
        return vcomplex(a.re * b.re - a.im * b.im,
                        a.re * b.im + a.im * b.re);
    }

    inline vcomplex
    mul_cn(const vcomplex &a, const vcomplex &b)
    {
        return vcomplex(a.re * b.re + a.im * b.im,
                        a.re * b.im - a.im * b.re);
    }

    inline vcomplex
    mul_nc(const vcomplex &a, const vcomplex &b)
    {
        return mul_cn(b,a);
    }

    inline vcomplex
    operator*(const vcomplex &a, const vcomplex &b)
    {
        return mul_nn(a,b);
    }
```

Next are up and down shifts in the fifth dimension:

$\langle \text{vector complex numbers} \rangle + \equiv$

```
inline vcomplex
operator<<(const vcomplex &a, const vcomplex &b) // -> a[2] a[1] a[0] b[3]
{
    return vcomplex(a.re << b.re, a.im << b.im);
}
```

```
inline vcomplex
operator>>(const vcomplex &a, const vcomplex &b) // -> a[0] b[3] b[2] b[1]
{
    return vcomplex(a.re >> b.re, a.im >> b.im);
}
```

One more thing. To implement $1 \pm \gamma\mu$ efficiently, we need a set of functions on complex numbers computing $a \pm b$ and $a \pm ib$:

```

<vector complex numbers>+≡
inline vcomplex
ap1b(const vcomplex &a, const vcomplex &b)
{
    return vcomplex(a.re + b.re, a.im + b.im);
}

inline vcomplex
am1b(const vcomplex &a, const vcomplex &b)
{
    return vcomplex(a.re - b.re, a.im - b.im);
}

inline vcomplex
apib(const vcomplex &a, const vcomplex &b)
{
    return vcomplex(a.re - b.im, a.im + b.re);
}

inline vcomplex
amib(const vcomplex &a, const vcomplex &b)
{
    return vcomplex(a.re + b.im, a.im - b.re);
}

inline void
set_ap1b(vcomplex &a, const vcomplex &b)
{
    a = ap1b(a,b);
}

inline void
set_am1b(vcomplex &a, const vcomplex &b)
{
    a = am1b(a,b);
}

inline void
set_apib(vcomplex &a, const vcomplex &b)
{
    a = apib(a,b);
}

```



```

inline void
set_amib(vcomplex &a, const vcomplex &b)
{
    a = amib(a,b);
}

```

3 LQCD ROUTINES

Here we define datatypes and operations dealing with LQCD objects.

```

<gcd.hh>≡
    #ifndef QCD_HH
    #define QCD_HH
    #include "qcdsse.hh"
    namespace QCDsse {
    <Gauge Fields>
    <Dirac Fermions>

    <QCD classes>
    };
    #endif

```

We need two kinds of gauge fields for efficiency. The 4-d fields are not SSE-aware and are stored in unpacked form:

```

<Gauge Fields>≡
    struct SU3 {
        float v[3][3][2];
    };

```

When applying the parallel transport to fermions, we need to fill SSE 4-vectors with the same values. Because gcc does register allocation for us, we introduce VSU3 structure:

```

<Gauge Fields>+≡
    struct VSU3 {
        vcomplex v[3][3];
        <Vector Gauge Field constructors>
    };

```

Constructing VSU3 is simple (but inadequate loop unrolling in gcc makes us use copy-paste):

```

<Vector Gauge Field constructors>≡
inline VSU3(const SU3 &x)
{
    v[0][0] = vcomplex(x.v[0][0][0], x.v[0][0][1]);
    v[0][1] = vcomplex(x.v[0][1][0], x.v[0][1][1]);
    v[0][2] = vcomplex(x.v[0][2][0], x.v[0][2][1]);

    v[1][0] = vcomplex(x.v[1][0][0], x.v[1][0][1]);
    v[1][1] = vcomplex(x.v[1][1][0], x.v[1][1][1]);
    v[1][2] = vcomplex(x.v[1][2][0], x.v[1][2][1]);

    v[2][0] = vcomplex(x.v[2][0][0], x.v[2][0][1]);
    v[2][1] = vcomplex(x.v[2][1][0], x.v[2][1][1]);
    v[2][2] = vcomplex(x.v[2][2][0], x.v[2][2][1]);
}

```

We also need an empty constructor for VSU3 because gcc is not very smart.

```

<Vector Gauge Field constructors>+≡
VSU3() {}

```

For fermions, only vector versions are needed. However, it is convenient to have both four-component and projected two-component variants.

```

<Dirac Fermions>≡
struct VDiracFermion {
    vcomplex r[4][3];
};

struct VHalfDiracFermion {
    vcomplex r[2][3];
};

```

Two multiplications of `VHalfDiracFermion` by `VSU3` are needed. The first computes $\chi = U\psi$ as follows.

$\langle \text{Dirac Fermions} \rangle + \equiv$

```
static void
mul_uh(VHalfDiracFermion &r, const VSU3 &u, const VHalfDiracFermion &p)
{
    r.r[0][0] = mul_nn(u.v[0][0], p.r[0][0])
               + mul_nn(u.v[0][1], p.r[0][1])
               + mul_nn(u.v[0][2], p.r[0][2]);
    r.r[0][1] = mul_nn(u.v[1][0], p.r[0][0])
               + mul_nn(u.v[1][1], p.r[0][1])
               + mul_nn(u.v[1][2], p.r[0][2]);
    r.r[0][2] = mul_nn(u.v[2][0], p.r[0][0])
               + mul_nn(u.v[2][1], p.r[0][1])
               + mul_nn(u.v[2][2], p.r[0][2]);

    r.r[1][0] = mul_nn(u.v[0][0], p.r[1][0])
               + mul_nn(u.v[0][1], p.r[1][1])
               + mul_nn(u.v[0][2], p.r[1][2]);
    r.r[1][1] = mul_nn(u.v[1][0], p.r[1][0])
               + mul_nn(u.v[1][1], p.r[1][1])
               + mul_nn(u.v[1][2], p.r[1][2]);
    r.r[1][2] = mul_nn(u.v[2][0], p.r[1][0])
               + mul_nn(u.v[2][1], p.r[1][1])
               + mul_nn(u.v[2][2], p.r[1][2]);
}
```

The second conjugates the gauge field before multiplication $\chi = U^\dagger \psi$ as follows.

```

<Dirac Fermions>+≡
static void
mul_ch(VHalfDiracFermion &r, const VSU3 &u, const VHalfDiracFermion &p)
{
    r.r[0][0] = mul_cn(u.v[0][0], p.r[0][0])
               + mul_cn(u.v[1][0], p.r[0][1])
               + mul_cn(u.v[2][0], p.r[0][2]);
    r.r[0][1] = mul_cn(u.v[0][1], p.r[0][0])
               + mul_cn(u.v[1][1], p.r[0][1])
               + mul_cn(u.v[2][1], p.r[0][2]);
    r.r[0][2] = mul_cn(u.v[0][2], p.r[0][0])
               + mul_cn(u.v[1][2], p.r[0][1])
               + mul_cn(u.v[2][2], p.r[0][2]);

    r.r[1][0] = mul_cn(u.v[0][0], p.r[1][0])
               + mul_cn(u.v[1][0], p.r[1][1])
               + mul_cn(u.v[2][0], p.r[1][2]);
    r.r[1][1] = mul_cn(u.v[0][1], p.r[1][0])
               + mul_cn(u.v[1][1], p.r[1][1])
               + mul_cn(u.v[2][1], p.r[1][2]);
    r.r[1][2] = mul_cn(u.v[0][2], p.r[1][0])
               + mul_cn(u.v[1][2], p.r[1][1])
               + mul_cn(u.v[2][2], p.r[1][2]);
}

```

All we need now are pieces for $(1 \pm \gamma_\mu)\psi$. Let us start from 4-d directions.
For

$$\gamma_1 = 1 \otimes \sigma_2 = \begin{pmatrix} 0 & -i & 0 & 0 \\ i & 0 & 0 & 0 \\ 0 & 0 & 0 & -i \\ 0 & 0 & i & 0 \end{pmatrix}$$

one has for the “plus” 1-direction:

$\langle \text{Dirac Fermions} \rangle + \equiv$

```
void inline
proj_plus1(VHalfDiracFermion &r, const VDiracFermion &a)
{
    r.r[0][0] = amib(a.r[0][0], a.r[1][0]);
    r.r[0][1] = amib(a.r[0][1], a.r[1][1]);
    r.r[0][2] = amib(a.r[0][2], a.r[1][2]);

    r.r[1][0] = amib(a.r[2][0], a.r[3][0]);
    r.r[1][1] = amib(a.r[2][1], a.r[3][1]);
    r.r[1][2] = amib(a.r[2][2], a.r[3][2]);
}

void inline
unpr_plus1(VDiracFermion &r, const VHalfDiracFermion &a)
{
    set_ap1b(r.r[0][0], a.r[0][0]);
    set_ap1b(r.r[0][1], a.r[0][1]);
    set_ap1b(r.r[0][2], a.r[0][2]);

    set_apib(r.r[1][0], a.r[0][0]);
    set_apib(r.r[1][1], a.r[0][1]);
    set_apib(r.r[1][2], a.r[0][2]);

    set_ap1b(r.r[2][0], a.r[1][0]);
    set_ap1b(r.r[2][1], a.r[1][1]);
    set_ap1b(r.r[2][2], a.r[1][2]);

    set_apib(r.r[3][0], a.r[1][0]);
    set_apib(r.r[3][1], a.r[1][1]);
    set_apib(r.r[3][2], a.r[1][2]);
}
```

And for the “minus” 1-direction:

$\langle \text{Dirac Fermions} \rangle + \equiv$

```
void inline
proj_minus1(VHalfDiracFermion &r, const VDiracFermion &a)
{
    r.r[0][0] = apib(a.r[0][0], a.r[1][0]);
    r.r[0][1] = apib(a.r[0][1], a.r[1][1]);
    r.r[0][2] = apib(a.r[0][2], a.r[1][2]);

    r.r[1][0] = apib(a.r[2][0], a.r[3][0]);
    r.r[1][1] = apib(a.r[2][1], a.r[3][1]);
    r.r[1][2] = apib(a.r[2][2], a.r[3][2]);
}

void inline
unpr_minus1(VDiracFermion &r, const VHalfDiracFermion &a)
{
    set_ap1b(r.r[0][0], a.r[0][0]);
    set_ap1b(r.r[0][1], a.r[0][1]);
    set_ap1b(r.r[0][2], a.r[0][2]);

    set_amib(r.r[1][0], a.r[0][0]);
    set_amib(r.r[1][1], a.r[0][1]);
    set_amib(r.r[1][2], a.r[0][2]);

    set_ap1b(r.r[2][0], a.r[1][0]);
    set_ap1b(r.r[2][1], a.r[1][1]);
    set_ap1b(r.r[2][2], a.r[1][2]);

    set_amib(r.r[3][0], a.r[1][0]);
    set_amib(r.r[3][1], a.r[1][1]);
    set_amib(r.r[3][2], a.r[1][2]);
}
```

For

$$\gamma_2 = \sigma_1 \otimes \sigma_1 = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

$\langle \text{Dirac Fermions} \rangle + \equiv$

```
void inline
proj_plus2(VHalfDiracFermion &r, const VDiracFermion &a)
{
    r.r[0][0] = ap1b(a.r[0][0], a.r[2][0]);
    r.r[0][1] = ap1b(a.r[0][1], a.r[2][1]);
    r.r[0][2] = ap1b(a.r[0][2], a.r[2][2]);

    r.r[1][0] = ap1b(a.r[1][0], a.r[3][0]);
    r.r[1][1] = ap1b(a.r[1][1], a.r[3][1]);
    r.r[1][2] = ap1b(a.r[1][2], a.r[3][2]);
}

void inline
unpr_plus2(VDiracFermion &r, const VHalfDiracFermion &a)
{
    set_ap1b(r.r[0][0], a.r[0][0]);
    set_ap1b(r.r[0][1], a.r[0][1]);
    set_ap1b(r.r[0][2], a.r[0][2]);

    set_ap1b(r.r[1][0], a.r[1][0]);
    set_ap1b(r.r[1][1], a.r[1][1]);
    set_ap1b(r.r[1][2], a.r[1][2]);

    set_ap1b(r.r[2][0], a.r[0][0]);
    set_ap1b(r.r[2][1], a.r[0][1]);
    set_ap1b(r.r[2][2], a.r[0][2]);

    set_ap1b(r.r[3][0], a.r[1][0]);
    set_ap1b(r.r[3][1], a.r[1][1]);
    set_ap1b(r.r[3][2], a.r[1][2]);
}
```

And for the “minus” 2-direction:

$\langle \text{Dirac Fermions} \rangle + \equiv$

```
void inline
proj_minus2(VHalfDiracFermion &r, const VDiracFermion &a)
{
    r.r[0][0] = am1b(a.r[0][0], a.r[2][0]);
    r.r[0][1] = am1b(a.r[0][1], a.r[2][1]);
    r.r[0][2] = am1b(a.r[0][2], a.r[2][2]);

    r.r[1][0] = am1b(a.r[1][0], a.r[3][0]);
    r.r[1][1] = am1b(a.r[1][1], a.r[3][1]);
    r.r[1][2] = am1b(a.r[1][2], a.r[3][2]);
}

void inline
unpr_minus2(VDiracFermion &r, const VHalfDiracFermion &a)
{
    set_ap1b(r.r[0][0], a.r[0][0]);
    set_ap1b(r.r[0][1], a.r[0][1]);
    set_ap1b(r.r[0][2], a.r[0][2]);

    set_ap1b(r.r[1][0], a.r[1][0]);
    set_ap1b(r.r[1][1], a.r[1][1]);
    set_ap1b(r.r[1][2], a.r[1][2]);

    set_am1b(r.r[2][0], a.r[0][0]);
    set_am1b(r.r[2][1], a.r[0][1]);
    set_am1b(r.r[2][2], a.r[0][2]);

    set_am1b(r.r[3][0], a.r[1][0]);
    set_am1b(r.r[3][1], a.r[1][1]);
    set_am1b(r.r[3][2], a.r[1][2]);
}
```


For

$$\gamma_3 = \sigma_{\otimes} \sigma_1 = \begin{pmatrix} 0 & 0 & 0 & -i \\ 0 & 0 & -i & 0 \\ 0 & i & 0 & 0 \\ i & 0 & 0 & 0 \end{pmatrix}$$

$\langle \text{Dirac Fermions} \rangle + \equiv$

```

void inline
proj_plus3(VHalfDiracFermion &r, const VDiracFermion &a)
{
    r.r[0][0] = amib(a.r[0][0], a.r[3][0]);
    r.r[0][1] = amib(a.r[0][1], a.r[3][1]);
    r.r[0][2] = amib(a.r[0][2], a.r[3][2]);

    r.r[1][0] = amib(a.r[1][0], a.r[2][0]);
    r.r[1][1] = amib(a.r[1][1], a.r[2][1]);
    r.r[1][2] = amib(a.r[1][2], a.r[2][2]);
}

void inline
unpr_plus3(VDiracFermion &r, const VHalfDiracFermion &a)
{
    set_ap1b(r.r[0][0], a.r[0][0]);
    set_ap1b(r.r[0][1], a.r[0][1]);
    set_ap1b(r.r[0][2], a.r[0][2]);

    set_ap1b(r.r[1][0], a.r[1][0]);
    set_ap1b(r.r[1][1], a.r[1][1]);
    set_ap1b(r.r[1][2], a.r[1][2]);

    set_apib(r.r[2][0], a.r[1][0]);
    set_apib(r.r[2][1], a.r[1][1]);
    set_apib(r.r[2][2], a.r[1][2]);

    set_apib(r.r[3][0], a.r[0][0]);
    set_apib(r.r[3][1], a.r[0][1]);
    set_apib(r.r[3][2], a.r[0][2]);
}

```

And for the “minus” 3-direction:

```

<Dirac Fermions>+≡
void inline
proj_minus3(VHalfDiracFermion &r, const VDiracFermion &a)
{
    r.r[0][0] = apib(a.r[0][0], a.r[3][0]);
    r.r[0][1] = apib(a.r[0][1], a.r[3][1]);
    r.r[0][2] = apib(a.r[0][2], a.r[3][2]);

    r.r[1][0] = apib(a.r[1][0], a.r[2][0]);
    r.r[1][1] = apib(a.r[1][1], a.r[2][1]);
    r.r[1][2] = apib(a.r[1][2], a.r[2][2]);
}

void inline
unpr_minus3(VDiracFermion &r, const VHalfDiracFermion &a)
{
    set_ap1b(r.r[0][0], a.r[0][0]);
    set_ap1b(r.r[0][1], a.r[0][1]);
    set_ap1b(r.r[0][2], a.r[0][2]);

    set_ap1b(r.r[1][0], a.r[1][0]);
    set_ap1b(r.r[1][1], a.r[1][1]);
    set_ap1b(r.r[1][2], a.r[1][2]);

    set_amib(r.r[2][0], a.r[1][0]);
    set_amib(r.r[2][1], a.r[1][1]);
    set_amib(r.r[2][2], a.r[1][2]);

    set_amib(r.r[3][0], a.r[0][0]);
    set_amib(r.r[3][1], a.r[0][1]);
    set_amib(r.r[3][2], a.r[0][2]);
}

```

For

$$\gamma_4 = \sigma_3 \otimes \sigma_1 = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

$\langle \text{Dirac Fermions} \rangle + \equiv$

```

void inline
proj_plus4(VHalfDiracFermion &r, const VDiracFermion &a)
{
    r.r[0][0] = ap1b(a.r[0][0], a.r[1][0]);
    r.r[0][1] = ap1b(a.r[0][1], a.r[1][1]);
    r.r[0][2] = ap1b(a.r[0][2], a.r[1][2]);

    r.r[1][0] = am1b(a.r[2][0], a.r[3][0]);
    r.r[1][1] = am1b(a.r[2][1], a.r[3][1]);
    r.r[1][2] = am1b(a.r[2][2], a.r[3][2]);
}

void inline
unpr_plus4(VDiracFermion &r, const VHalfDiracFermion &a)
{
    set_ap1b(r.r[0][0], a.r[0][0]);
    set_ap1b(r.r[0][1], a.r[0][1]);
    set_ap1b(r.r[0][2], a.r[0][2]);

    set_ap1b(r.r[1][0], a.r[0][0]);
    set_ap1b(r.r[1][1], a.r[0][1]);
    set_ap1b(r.r[1][2], a.r[0][2]);

    set_ap1b(r.r[2][0], a.r[1][0]);
    set_ap1b(r.r[2][1], a.r[1][1]);
    set_ap1b(r.r[2][2], a.r[1][2]);

    set_am1b(r.r[3][0], a.r[1][0]);
    set_am1b(r.r[3][1], a.r[1][1]);
    set_am1b(r.r[3][2], a.r[1][2]);
}

```

And for the “minus” 4-direction:

$\langle \text{Dirac Fermions} \rangle + \equiv$

```
void inline
proj_minus4(VHalfDiracFermion &r, const VDiracFermion &a)
{
    r.r[0][0] = am1b(a.r[0][0], a.r[1][0]);
    r.r[0][1] = am1b(a.r[0][1], a.r[1][1]);
    r.r[0][2] = am1b(a.r[0][2], a.r[1][2]);

    r.r[1][0] = ap1b(a.r[2][0], a.r[3][0]);
    r.r[1][1] = ap1b(a.r[2][1], a.r[3][1]);
    r.r[1][2] = ap1b(a.r[2][2], a.r[3][2]);
}

void inline
unpr_minus4(VDiracFermion &r, const VHalfDiracFermion &a)
{
    set_ap1b(r.r[0][0], a.r[0][0]);
    set_ap1b(r.r[0][1], a.r[0][1]);
    set_ap1b(r.r[0][2], a.r[0][2]);

    set_am1b(r.r[1][0], a.r[0][0]);
    set_am1b(r.r[1][1], a.r[0][1]);
    set_am1b(r.r[1][2], a.r[0][2]);

    set_ap1b(r.r[2][0], a.r[1][0]);
    set_ap1b(r.r[2][1], a.r[1][1]);
    set_ap1b(r.r[2][2], a.r[1][2]);

    set_ap1b(r.r[3][0], a.r[1][0]);
    set_ap1b(r.r[3][1], a.r[1][1]);
    set_ap1b(r.r[3][2], a.r[1][2]);
}
```

The fifth direction is special in two respects. First, the parallel transport is trivial. Second, we have cleverly chosen γ -matrix representation to avoid a few addition. For DWF we do not need the full complement of projection operators, hence this case is shorter than the previous four.

$$\gamma_5 = 1 \otimes \sigma_3 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$$

Checkerboarding requires two functions. Let us denote vector objects by bold letters as follows: $\mathbf{a} = [a_0, a_1, a_2, a_3]$. On even 4-d sites we need to compute

$$\mathbf{r} = (1 + \gamma_5)\mathbf{a} + (1 - \gamma_5)[a_2, a_1, a_0, b_3]$$

```

<Dirac Fermions>+≡
void inline
proj5_even(VDiracFermion &r, const VDiracFermion &a, const VDiracFermion &b)
{
    r.r[0][0] = a.r[0][0] + a.r[0][0];
    r.r[0][1] = a.r[0][1] + a.r[0][1];
    r.r[0][2] = a.r[0][2] + a.r[0][2];

    r.r[1][0] = a.r[1][0] + a.r[1][0];
    r.r[1][1] = a.r[1][1] + a.r[1][1];
    r.r[1][2] = a.r[1][2] + a.r[1][2];

    vcomplex r20 = a.r[2][0] << b.r[2][0]; r.r[2][0] = r20 + r20;
    vcomplex r21 = a.r[2][1] << b.r[2][1]; r.r[2][1] = r21 + r21;
    vcomplex r22 = a.r[2][2] << b.r[2][2]; r.r[2][2] = r22 + r22;

    vcomplex r30 = a.r[3][0] << b.r[3][0]; r.r[3][0] = r30 + r30;
    vcomplex r31 = a.r[3][1] << b.r[3][1]; r.r[3][1] = r31 + r31;
    vcomplex r32 = a.r[3][2] << b.r[3][2]; r.r[3][2] = r32 + r32;
}

```

On odd 4-d sites we need to compute

$$\mathbf{r} = (1 + \gamma_5)[a_0, b_3, b_2, b_1] + (1 - \gamma_5)\mathbf{b}$$

$\langle \text{Dirac Fermions} \rangle + \equiv$

```
void inline
proj5_odd(VDiracFermion &r, const VDiracFermion &a, const VDiracFermion &b)
{
    vcomplex r00 = a.r[0][0] >> b.r[0][0]; r.r[0][0] = r00 + r00;
    vcomplex r01 = a.r[0][1] >> b.r[0][1]; r.r[0][1] = r01 + r01;
    vcomplex r02 = a.r[0][2] >> b.r[0][2]; r.r[0][2] = r02 + r02;

    vcomplex r10 = a.r[1][0] >> b.r[1][0]; r.r[1][0] = r10 + r10;
    vcomplex r11 = a.r[1][1] >> b.r[1][1]; r.r[1][1] = r11 + r11;
    vcomplex r12 = a.r[1][2] >> b.r[1][2]; r.r[1][2] = r12 + r12;

    r.r[2][0] = b.r[2][0] + b.r[2][0];
    r.r[2][1] = b.r[2][1] + b.r[2][1];
    r.r[2][2] = b.r[2][2] + b.r[2][2];

    r.r[3][0] = b.r[3][0] + b.r[3][0];
    r.r[3][1] = b.r[3][1] + b.r[3][1];
    r.r[3][2] = b.r[3][2] + b.r[3][2];
}
```

This completes the set of inlined procedures. Now let us put together some serious procedures.

4 QCD INTERFACE

We simply provide interface to SSE based QCD code here.

For gauge field, we only need 0- and 4-dimensional versions:

$\langle QCD \text{ classes} \rangle \equiv$

```
class GaugeField {
    SU3 *v;
public:
    GaugeField(SU3 *r): v(r) {}
    SU3 &operator[](int i) { return v[i]; }
};

class GaugeField4d {
    SU3 *v;
public:
    GaugeField4d(SU3 *r): v(r) {}
    GaugeField operator[](int i) const { return GaugeField(v + 4 * i); }
};
```

For Dirac fermions, on the other hand, it is convenient to have 1- and 5-dimensional classes:

```

<QCD classes>+≡
class DiracFermion1d {
    VDiracFermion *f;
public:
    DiracFermion1d(): f(0) {}
    DiracFermion1d(VDiracFermion *x): f(x) {}
    VDiracFermion &operator[](int i) { return f[i]; }
};

class DiracFermion5d {
    VDiracFermion *f;
    int S8;
public:
    DiracFermion5d(VDiracFermion *x, int s8): f(x), S8(s8) {}
    DiracFermion1d operator[](int i) const { return DiracFermion1d(f + S8 * i); }
};

```

For the sake of exposition, let us collect all pieces describing lattice geometry into one class. It is likely, that an application will have only one variable of class Lattice5d, but what the heck.

```

<QCD classes>+≡
class Lattice5d {
    int X, Y, Z, T, S8;
    int XYZT;

    struct step {
        int d[4];
    };
    step *XYZTup;
    step *XYZTdown;
    step *Udown;
    int *Sup;
    int *Sdown;
    bool *parity;
public:
    Lattice5d(int x, int y, int z, int t, int s);
    DiracFermion5d create_fermion(void);
    GaugeField4d create_gauge(void);
    void Keo(DiracFermion5d &result,
             const GaugeField4d &U,
             const DiracFermion5d &psi);
    int KeoOps(void);
};

```

5 K_{eo} IMPLEMENTATION

```

<qcd.cc>≡
#include <stdlib.h>
#include "qcd.hh"
using namespace QCDSSE;
<QCD methods>

    Let us start with an implementation of  $K_{eo}$ .

<QCD methods>≡
void
Lattice5d::Keo(DiracFermion5d &result,
               const GaugeField4d &U,
               const DiracFermion5d &psi)
{
    <Keo local variables>
    for (int xyzt = XYZT; xyzt--;) {
        bool p = parity[xyzt];
        <Extract 1-d addresses and build SSE SU3 objects>
        for (int s = S8; s--;) {
            <Compute "one" site>
        }
    }
}

```


“One” site above is a string of 4 lattice sites along the fifth dimension. We have enough SSE machinery to compute part of K_{eo} efficiently.

$\langle \text{Compute “one” site} \rangle \equiv$

```

VDiracFermion &dfR(dfRR[s]);
if (p) {
    proj5_even(dfR, dfZZ[s], dfZZ[Sdown[s]]);
} else {
    proj5_odd(dfR, dfZZ[Sup[s]], dfZZ[s]);
}
proj_plus1(hp[0], dfp[0][s]);
mul_uh(vp[0], up[0], hp[0]);
unpr_plus1(dfR, vp[0]);

proj_minus1(hm[0], dfm[0][s]);
mul_ch(vm[0], um[0], hm[0]);
unpr_minus1(dfR, vm[0]);

proj_plus2(hp[1], dfp[1][s]);
mul_uh(vp[1], up[1], hp[1]);
unpr_plus2(dfR, vp[1]);

proj_minus2(hm[1], dfm[1][s]);
mul_ch(vm[1], um[1], hm[1]);
unpr_minus2(dfR, vm[1]);

proj_plus3(hp[2], dfp[2][s]);
mul_uh(vp[2], up[2], hp[2]);
unpr_plus3(dfR, vp[2]);

proj_minus3(hm[2], dfm[2][s]);
mul_ch(vm[2], um[2], hm[2]);
unpr_minus3(dfR, vm[2]);

proj_plus4(hp[3], dfp[3][s]);
mul_uh(vp[3], up[3], hp[3]);
unpr_plus4(dfR, vp[3]);

proj_minus4(hm[3], dfm[3][s]);
mul_ch(vm[3], um[3], hm[3]);
unpr_minus4(dfR, vm[3]);

```

Counting floating point operations above gives

```

<QCD methods>+≡
int
Lattice5d::KeoOps(void)
{
    return X * Y * Z * T * S8 * 4 * (24 + 4 * 2 * (12 + 24 + 12 * 11));
}

```

This is small potato, but, since we pretend to be writing fast code, let us not rely on a C++ compiler to optimize loop invariants:

```

<Extract 1-d addresses and build SSE SU3 objects>≡
dfRR = result[xyzt];
dfZZ = psi[xyzt];
for (int i = 0; i < 4; i++) {
    dfp[i] = psi[XYZTup[xyzt].d[i]];
    dfm[i] = psi[XYZTdown[xyzt].d[i]];
    up[i] = VSU3(U[xyzt][i]);
    um[i] = VSU3(U[Udown[xyzt].d[i]][i]);
}

```

Unfortunately, gcc generates awful code for local variables in some cases. K_{eo} is one of such bad cases. We need to sacrifice some of locality of variables to fix it.

```

<Keo local variables>≡
DiracFermion1d dfRR, dfZZ, dfp[4], dfm[4];
VSU3 up[4], um[4];
VHalfDiracFermion hp[4], hm[4], vp[4], vm[4];

```

Creating gauge field is simple. When `create_gauge` is called we already know size of the lattice, so we simply call `new` to create an appropriate array.

```

<QCD methods>+≡
GaugeField4d
Lattice5d::create_gauge(void)
{
    SU3 *v = new SU3[XYZT * 4];

    <Initialize Gauge Field>
    return v;
}

```

For now, we fill gauge field with random numbers:

```

<Initialize Gauge Field>≡
float *f = (float *)v;
for (int i = sizeof(SU3) * 4 * XYZT / sizeof(float); i--;) {
    f[i] = rand() / (double)RAND_MAX;
}

```

Equally simple is allocating a fresh red/black sublattice of `VDiracFermion`. Unfortunately, gcc's `new` is broken, we need to align SSE data manually.

```

<QCD methods>+≡
DiracFermion5d
Lattice5d::create_fermion(void)
{
    char *ptr = new char[sizeof (VDiracFermion) * (XYZT * S8 + 1)];
    unsigned long v = (unsigned long)ptr;
    unsigned long v_aligned = (v + 15) & ~15;

    <Initialize 5d Fermion>
    return DiracFermion5d((VDiracFermion *)v_aligned, S8);
}

```

Fermion field is also initialized with random numbers:

```

<Initialize 5d Fermion>≡
float *f = (float *)ptr;
for (int i = sizeof(VDiracFermion) / sizeof(f) * XYZT * S8; i--;) {
    f[i] = rand() / (double)RAND_MAX;
}

```

The only QCD method left is `Lattice5d` constructor. We use it to compute lattice sizes and build neighbor tables needed for K_{eo} .

```

<QCD methods>+≡
Lattice5d::Lattice5d(int x, int y, int z, int t, int s)
{
    X = x; Y = y; Z = z; T = t; S8 = s/8;
    XYZT = X * Y * Z * T;
    <Allocate indices>
    <Compute Sup and Sdown>
    <Compute 4-d indices>
}

```

Since we do not know the lattice size until runtime, indices are dynamically allocated. (Production code will need a destructor for `Lattice5d`, but here we can allow ourselves to be sloppy.)

```

<Allocate indices>≡
XYZTup    = new step[XYZT];
XYZTdown  = new step[XYZT];
Udown     = new step[XYZT];
parity    = new bool[XYZT];
Sup       = new int[S8];
Sdown     = new int[S8];

```

```

<Compute 4-d indices>≡
#define FourDidx(x,y,z,t) ((x) + X * ((y) + Y * ((z) + Z * (t))))
for (int x = 0; x < X; x++) {
    int xup = (x == (X-1)) ? 0: x+1;
    int xdown = (x == 0) ? X-1: x-1;
    for (int y = 0; y < Y; y++) {
        int yup = (y == (Y-1)) ? 0: y+1;
        int ydown = (y == 0) ? Y-1: y-1;
        for (int z = 0; z < Z; z++) {
            int zup = (z == (Z-1)) ? 0: z+1;
            int zdown = (z == 0) ? Z-1: z-1;
            for (int t = 0; t < T; t++) {
                int tup = (t == (T-1)) ? 0: t+1;
                int tdown = (t == 0) ? T-1: t-1;
                <Compute indices on one 4-d site>
            }
        }
    }
}

```

```

<Compute indices on one 4-d site>≡
int xyzt = FourDidx(x,y,z,t);
XYZTup[xyzt].d[0] = FourDidx(xup,y,z,t);
XYZTup[xyzt].d[1] = FourDidx(x,yup,z,t);
XYZTup[xyzt].d[2] = FourDidx(x,y,zup,t);
XYZTup[xyzt].d[3] = FourDidx(x,y,z,tup);

XYZTdown[xyzt].d[0] = FourDidx(xdown,y,z,t);
XYZTdown[xyzt].d[1] = FourDidx(x,ydown,z,t);
XYZTdown[xyzt].d[2] = FourDidx(x,y,zdown,t);
XYZTdown[xyzt].d[3] = FourDidx(x,y,z,tdown);

parity[xyzt] = ((x+y+z+t)&1) ? true: false;

```

Computing Sup and Sdown is simple. Here we implement periodic boundary conditions (which need to be changed for real domain wall fermion, but let us keep this code simple)

```

<Compute Sup and Sdown>≡
for (s = S8; s--;) {
    Sup[s] = s + 1;
    Sdown[s] = s - 1;
}
Sup[S8 - 1] = 0;
Sdown[0] = S8 - 1;

```

6 TEST CODE

The test is to loop K_{eo} enough time to get an estimate of performance.

```
<main.cc>≡
#include <sys/time.h>
#include <stdio.h>
#include <stdlib.h>
#include "qcd.hh"

int
main(int argc, char *argv[]) {
    using namespace QCDSSE;
    <check arguments>
    <create dynamic objects>
    struct timeval t0, t1;
    gettimeofday(&t0, NULL);
    <perform SSE operations>
    gettimeofday(&t1, NULL);
    <show time elapsed and performance>

    return 0;
}
```

Lattice size and number of iterations are arguments of `main()`:

```
<check arguments>≡
if (argc != 7) {
    error:
        fprintf(stderr, "usage: sse X Y Z T S iter\n");
        return 1;
}
```

From X, Y, Z and T we only require to be even and positive:

```
<check arguments>+≡
int X = atoi(argv[1]);
if ((X <= 0) || (X & 1 != 0)) {
    fprintf(stderr, "X must be even positive. (it is %d)\n", X);
    return 1;
}

<check arguments>+≡
int Y = atoi(argv[2]);
if ((Y <= 0) || (Y & 1 != 0)) {
    fprintf(stderr, "Y must be even positive. (it is %d)\n", Y);
    return 1;
}
```

```

<check arguments>+≡
    int Z = atoi(argv[3]);
    if ((Z <= 0) || (Z & 1 != 0)) {
        fprintf(stderr, "Z must be even positive. (it is %d)\n", Z);
        return 1;
    }

```

```

<check arguments>+≡
    int T = atoi(argv[4]);
    if ((T <= 0) || (T & 1 != 0)) {
        fprintf(stderr, "T must be even positive. (it is %d)\n", T);
        return 1;
    }

```

However, S must be a multiple of 8:

```

<check arguments>+≡
    int S = atoi(argv[5]);
    if ((S <= 0) || (S & 7 != 0)) {
        fprintf(stderr, "S must be positive multiple of 8 (it is %d)\n", S);
        return 1;
    }

```

Number of iteration is simply taken from the sixth argument:

```

<check arguments>+≡
    int count = atoi(argv[6]);

```

Next, we create Lattice5d object that will handle all other memory allocations.

```

<create dynamic objects>≡
    Lattice5d Lattice(X, Y, Z, T, S);

```

We need one gauge field on Lattice:

```

<create dynamic objects>+≡
    GaugeField4d U = Lattice.create_gauge();
    and two Dirac fermions (even and odd sublattices)

```

```

<create dynamic objects>+≡
    DiracFermion5d phi = Lattice.create_fermion();
    DiracFermion5d psi = Lattice.create_fermion();

```

Now we have all the pieces to compute $\phi = (K_{eo}\psi$:

```

<perform SSE operations>≡
    for (int i = count; i--;) {
        Lattice.Keo(phi, U, psi);
    }

```

Computing elapsed time is easy

```

<show time elapsed and performance>≡
    double dt = t1.tv_sec - t0.tv_sec + 10e-6*(t1.tv_usec - t0.tv_usec);

```

However, the elapsed time may be too short to compute performance. Check of overflow here to avoid embarasement

<show time elapsed and performance>+≡

```
if (dt > 0) {
    printf("Performance %g Mflops/sec. Lattice %d %d %d %d %d ."
           " %d Iterations. %g total seconds\n",
           (double)count * Lattice.KeoOps() / dt / 1e6,
           X, Y, Z, T, S, count, dt);
} else {
    printf("*** Keo ran too fast. Try increasing count\n");
    return 1;
}
```

7 TODO

First, let me note that many “obvious” optimizations do not work in this particular code and there are reasons to believe they will not work with gcc on any code of comparable size. The only path remaining is trial and error.

1. Compare to Intel’s compiler.
2. Try looping over a number of colors in multiplications and projection/unprojection operations.
3. It seems important to keep the inner loop (plus all functions it calls) under 16KB to fit into the trace cache. Keep this in mind when modifying the code.
4. Currently, too many extra memory accesses is generated by gcc. Try to fix it.