Grid

Peter Boyle, Azusa Yamaguchi, Guido Cossu, Antonin Portelli

Fine grid Dirac matrix bandwidth analysis

- L^4 local volume; 8/16 point stencil
  - Multi-RHS and DWF take $L_s = N_{\rm rhs}$. Suppresses gauge field overhead;
  - Cache reuse $\times N_{\rm stencil}$ on Fermion possible

- Accesses per 4d site of result
  - Fermion: $N_{\rm stencil} \times (N_s \in \{1,4\}) \times (N_c = 3) \times (N_{\rm rhs} \in \{1,16\})$ complex
  - Gauge: $2N_d \times N_c^2$ complex
- Flops
  - $N_{\rm stencil} \times N_{hs}$ SU(3) MatVec: $66 \times N_{\rm hs} \times N_{\rm stencil}$ (+ spin projection)

Action	Fermion Vol	Surface	N_s	N_{hs}	$N_{\rm rhs}$	Flops	Bytes	Bytes/Flops
HISQ	L^4	$3 \times 8 \times L^3$	1	1	1	1146	1560	1.36
Wilson	L^4	$8 \times L^3$	4	2	1	1320	1440	1.09
DWF	$L^4 \times N$	$8 \times L^3$	4	2	16	$N_{\rm rhs} \times 1320$	$N_{\rm rhs} \times 864$	0.65
Wilson-RHS	L^4	$8 \times L^3$	4	2	16	$N_{\rm rhs} \times 1320$	$N_{\rm rhs} \times 864$	0.65
HISQ-RHS	L^4	$3 \times 8 \times L^3$	1	1	16	$N_{\rm rhs} \times 1146$	$N_{\rm rhs} \times 408$	0.36

- $\sim \frac{1}{L}$ of data references come from off node

Scaling fine operator requires interconnect bandwidth

$$B_{network} \sim \frac{B_{memory}}{L} \times R$$

where R is the *reuse* factor obtained for the stencil in caches. $R \in [1, N_{stencil}]$

Growing on chip parallelism...

http://www.agner.org/optimize/

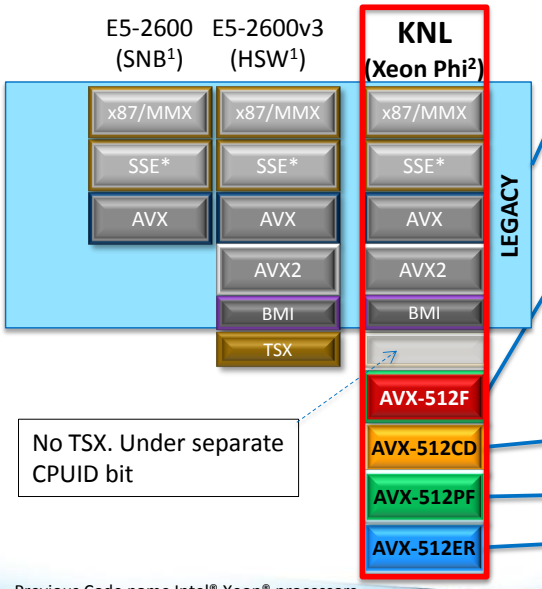
Core	simd	Year	Vector bits	SP flops/clock/core	cores	flops/clock
Pentium III	SSE	1999	128	3	1	3
Pentium IV	SSE2	2001	128	4	1	4
Core2	SSE2/3/4	2006	128	8	2	16
Nehalem	SSE2/3/4	2008	128	8	10	80
Sandybridge	AVX	2011	256	16	12	192
Haswell	AVX2	2013	256	32	18	576
KNC	IMCI	2012	512	32	64	2048
KNL	AVX512	2016	512	64	72	4608
Skylake	AVX512	2017(?)	512	64	28	1792

- Growth in core counts
- Growth in SIMD parallelism
- Interconnect performance failing to grow as fast as processor and memory performance

Standard industry solution is to dump it on the progammer!

Intel Knight's Landing Deep Dive

KNL ISA



KNL implements all legacy instructions

- Legacy binary runs w/o recompilation
- KNC binary requires recompilation

KNL introduces AVX-512 Extensions

- 512-bit FP/Integer Vectors
- 32 registers, & 8 mask registers
- Gather/Scatter

Conflict Detection: Improves Vectorization

Prefetch: Gather and Scatter Prefetch

Exponential and Reciprocal Instructions

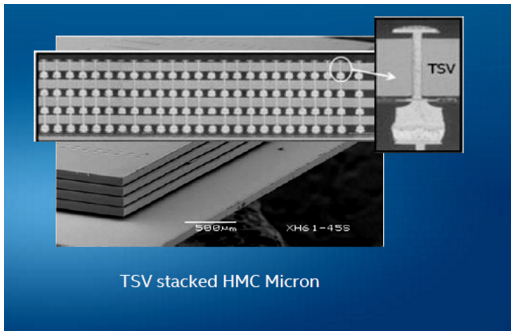
1. Previous Code name Intel® Xeon® processors
2. Xeon Phi = Intel® Xeon Phi™ processor

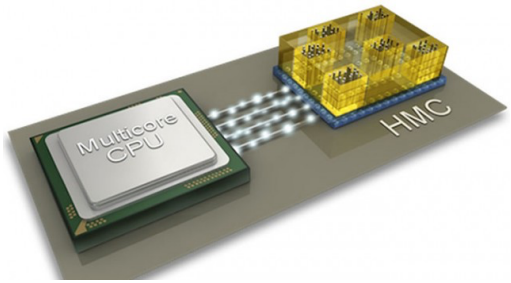
3D integration

- Apply to memory buses with through-silicon-via's (TSVs)!
- 2.5D : Integrate memory stacks on an *interposer* (Intel, Nvidia, AMD)
  In package memory: long thin wires → short broad fast wires
- 3D : Direct bond memory stacks to compute (PEZY, mobile, Broadcom)
  3D memory could grow the bus widths almost arbitrarily

Massive replica counts from silicon lithography compared to macroscopic assembly

There's plenty of room at the bottom (Feynman); Avagadro's number is big!



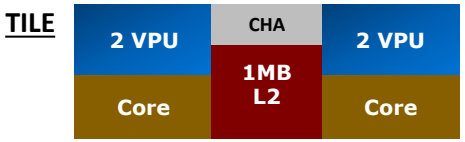


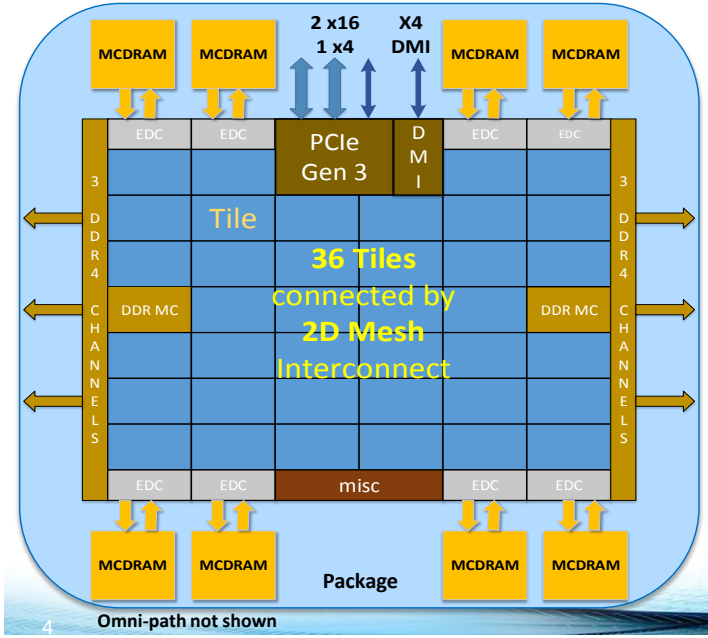
- This years tech:
  - 16 GB (AXPY 400 GB/s) Intel Knights Landing (KNL)
  - 16-32 GB (AXPY 600 GB/s) Nvidia Pascal P100
  - Regular Xeon ... when ?

Intel Knight's Landing Deep Dive

Intel HotChips Talk Hyperlink

Knights Landing Overview





Chip: 36 Tiles interconnected by **2D Mesh**
Tile: 2 Cores + 2 VPU/core + 1 MB L2

Memory: MCDRAM: 16 GB on-package; High BW
DDR4: 6 channels @ 2400 up to 384GB
IO: 36 lanes PCIe Gen3. 4 lanes of DMI for chipset
Node: 1-Socket only
Fabric: Omni-Path on-package (not shown)

Vector Peak Perf: 3+TF DP and 6+TF SP Flops
Scalar Perf: ~3x over Knights Corner
Streams Triad (GB/s): MCDRAM : 400+; DDR: 90+

Source Intel: All products, computer systems, dates and figures specified are preliminary based on current expectations, and are subject to change without notice. KNL data are preliminary based on current expectations and are subject to change without notice. 1Binary Compatible with Intel Xeon processors using Haswell Instruction Set (except TSX). 2Bandwidth numbers are based on STREAM-like memory access pattern when MCDRAM used as flat memory. Results have been estimated based on internal Intel analysis and are provided for informational purposes only. Any difference in system hardware or software design or configuration may affect actual performance.

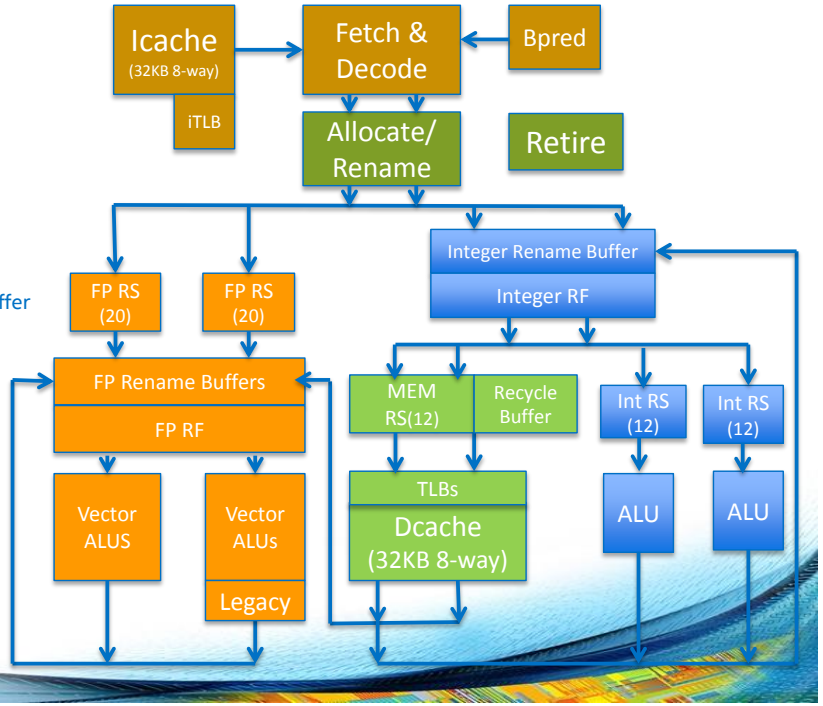
Core & VPU

- Out-of-order core w/ 4 SMT threads
- VPU tightly integrated with core pipeline

- 2-wide Decode/Rename/Retire
- ROB-based renaming. 72-entry ROB & Rename Buffers
- Up to 6-wide at execution
- Int and FP RS OoO.
- MEM RS inorder with OoO completion. Recycle Buffer holds memory ops waiting for completion.
- Int and Mem RS hold source data. FP RS does not.

- 2x 64B Load & 1 64B Store ports in Dcache.
- 1st level uTLB: 64 entries
- 2nd level dTLB: 256 4K, 128 2M, 16 1G pages

- L1 Prefetcher (IPP) and L2 Prefetcher.
- 46/48 PA/VA bits
- Fast unaligned and cache-line split support.
- Fast Gather/Scatter support



Grid performance portability strategy

Define performant classes *vRealF*, *vRealD*, *vComplexF*, *vComplexD*.

```
#if defined (AVX1) || defined (AVX2)
    typedef __m256 SIMD_Ftype;
#endif
#if defined (SSE2)
    typedef __m128 SIMD_Ftype;
#endif
#if defined (AVX512)
    typedef __m512 SIMD_Ftype;
#endif
...
template <class Scalar_type, class Vector_type>
class Grid_simd {
      Vector_type v;

      // Define arithmetic operators
      friend inline vRealD operator + (vRealD a, vRealD b);
      friend inline vRealD operator - (vRealD a, vRealD b);
      friend inline vRealD operator * (vRealD a, vRealD b);
      friend inline vRealD operator / (vRealD a, vRealD b);

      static int Nsimd(void) { return sizeof(Vector_type)/sizeof(Scalar_type);}
}

typedef Grid_simd<float, SIMD_Ftype> vRealF;
typedef Grid_simd<double, SIMD_Dtype> vRealD;
typedef Grid_simd<std::complex<float>, SIMD_Ftype> vComplexF;
typedef Grid_simd<std::complex<double>, SIMD_Dtype> vComplexD;
typedef Grid_simd<Integer, SIMD_Itype> vInteger;
```

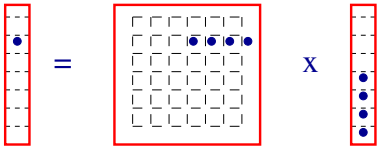
- Different implementations of vector classes require only 400 lines
- Implementation uses more sophisticated template metaprogramming in C++ than shown
- e.g. EnableIf differentiates complex, non-complex behaviour etc..

What is the best SIMD strategy?

SIMD most efficient for *independent but identical* work
e.g. apply N small dense matrix-vector multiplies in parallel:

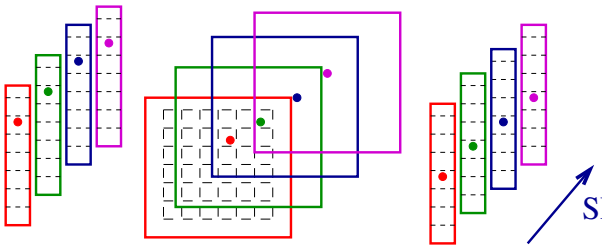
```
inline template<int N, class simd>
void matmul( simd * __restrict__ x,
             simd * __restrict__ y,
             simd * __restrict__ z)
{
    for(int i=0;i<N;i++){
        for(int j=0;j<N;j++){
            fmac(y[i*N+j],z[j],x[i]);
        }
    }
}
```

Vector = Matrix x Vector



Reduction of vector sum
is bottleneck for small N

Many vectors = many matrices x many vectors



No reduction or SIMD lane
crossing operations.

SIMD interleave

Code examples & performance analysis

```
        vmovaps (%rdx), %ymm0
        vmovaps 32(%rdx), %ymm1
        vmovaps 64(%rdx), %ymm2
        vmovaps 96(%rdx), %ymm3
        vmovaps 128(%rdx), %ymm4
        vmovaps 160(%rdx), %ymm5
        vmovaps 192(%rdx), %ymm6
        vmovaps 224(%rdx), %ymm7
        xorl    %eax, %eax
LBB0_1:                                 ## %.preheader
        vmulps  (%rsi,%rax,8), %ymm0, %ymm8
        vaddps  (%rdi,%rax), %ymm8, %ymm8
        vmulps  32(%rsi,%rax,8), %ymm1, %ymm9
        vaddps  %ymm9, %ymm8, %ymm8
        vmulps  64(%rsi,%rax,8), %ymm2, %ymm9
        vaddps  %ymm9, %ymm8, %ymm8
        vmulps  96(%rsi,%rax,8), %ymm3, %ymm9
        vaddps  %ymm9, %ymm8, %ymm8
        vmulps  128(%rsi,%rax,8), %ymm4, %ymm9
        vaddps  %ymm9, %ymm8, %ymm8
        vmulps  160(%rsi,%rax,8), %ymm5, %ymm9
        vaddps  %ymm9, %ymm8, %ymm8
        vmulps  192(%rsi,%rax,8), %ymm6, %ymm9
        vaddps  %ymm9, %ymm8, %ymm8
        vmulps  224(%rsi,%rax,8), %ymm7, %ymm9
        vaddps  %ymm9, %ymm8, %ymm8
        vmovaps %ymm8, (%rdi,%rax)
        addq    $32, %rax
        cmpq    $256, %rax              ## imm = 0x100
```

- Template parameter matrix size 8 ; known at compile time; AVX1/2
- Column vec in registers ymm0-7; accumulation in ymm8/9
- *Out of order execution* in Ivybridge runs four copies of loop in parallel (!!!)

Back to the Future

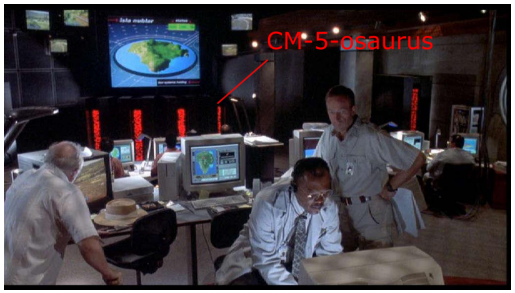
Q) How do we find copious *independent but identical* work?

A) Remember that SIMD was NOT hard in the 1980's (CM, APE...)

Connection Machine Model CM-2 and DataVault System

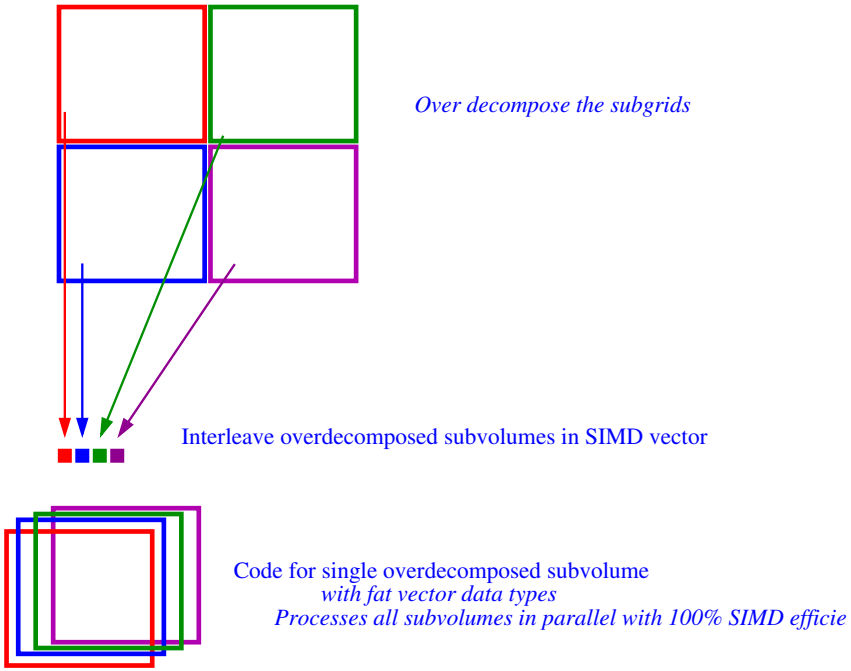
The Connection Machine Model CM-2 uses thousands of processors operating in parallel to achieve peak processing speeds of above 10 gigaflops. The DataVault mass storage system stores up to 60 gigabytes of data.

- Resurrect Jurassic data parallel programming techniques: **cmfortran, HPF**
- Address SIMD, OpenMP, MPI with *single* data parallel interface
  - Map arrays to virtual nodes with user controlled layout primitives
  - Conformable array operations proceed data parallel with 100% SIMD efficiency
  - CSHIFT primitives handle communications



GRID parallel library

- Geometrically decompose cartesian arrays across nodes (MPI)
- Subdivide node volume into smaller *virtual nodes*
- Spread virtual nodes across SIMD lanes
- Use OpenMP+MPI+SIMD to process conformable array operations
- Same instructions executed on many nodes, each node operates on four virtual nodes



GRID data parallel CSHIFT details

- Crossing between SIMD lanes restricted to during cshifts between virtual nodes
- Code for N-virtual nodes is identical to scalar code for one, except datum is N fold bigger

$$\underbrace{(A,B,C,D)}_{\text{virtual subnode}} \quad \underbrace{(E,F,G,H)}_{\text{virtual subnode}} \rightarrow \underbrace{(AE,BF,CG,DH)}_{\text{Packed SIMD}}$$

- CSHIFT involves a CSHIFT of SIMD, and a permute *only* on the surface

$$(AE,BF,CG,DH) \rightarrow \underbrace{(BF,\ CG,DH,AE)}_{\text{cshift bulk}}$$

$$\downarrow$$

$$\underbrace{(BF,\ CG,DH,EA)}_{\text{permute face}} \rightarrow \underbrace{(B,C,D,E)}_{\text{virtual subnode}} \quad \underbrace{(F,G,H,A)}_{\text{virtual subnode}}$$

- Shuffle overhead is *suppressed* by surface to volume ratio

GRID data parallel template library

Ordering	Layout	Vectorisation	Data Reuse
Microprocessor	Array-of-Structs (AoS)	Hard	Maximised
Vector	Struct-of-Array (SoA)	Easy	Minimised
Bagel	Array-of-structs-of-short-vectors (AoSoSV)	Easy	Maximised

- www.github.com/paboyle/Grid
- PAB, Cossu, Portelli, Yamaguchi: arXiv:1512.03487; Poster 184
- Automatically transform layout of arrays of mathematical objects using vSIMD template parameter
- Conformable array operations are data parallel on the *same* Grid layout

```
vRealF, vRealD, vComplexF, vComplexD

template<class vtype> class iScalar
{
    vtype _internal;
};
template<class vtype,int N> class iVector
{
    vtype _internal[N];
};
template<class vtype,int N> class iMatrix
{
    vtype _internal[N][N];
};

typedef Lattice<iMatrix<vComplexD> > LatticeColourMatrix;
typedef iMatrix<ComplexD>  ColourMatrix;
```

- Internal type can be SIMD vectors *or* scalars

```
LatticeColourMatrix A(Grid);
LatticeColourMatrix B(Grid);
LatticeColourMatrix C(Grid);
LatticeColourMatrix dC_dy(Grid);

C = A*B;

const int Ydim = 1;

dC_dy = 0.5*Cshift(C,Ydim, 1 )
      - 0.5*Cshift(C,Ydim,-1 );
```

- *High-level* data parallel code gets 65% of peak on AVX2
- Single data parallelism model targets BOTH SIMD and threads efficiently.

C++11 implementation

- Grid types composable to arbitrary nested tensors:

```
Vector<Vector<Vector<RealF,Ncolour>, Nspin >, Nflavour >
```

1. Arbitrary depth tensor products supported
2. the return type of the same operation on the $n-1$ index nested object (recursive)
3. the multiplication rules for scalar/vector/matrix on the n-th index
4. pass return pointer to elide object copies (copy elision/RVO)

L	op	R	→ ret	L	op	R	→ ret	L	op	R	→ ret
S	×	S	→S	S	×	V	→ V	S	×	M	→ M
V	×	S	→ V	V	×	V	→ S	V	×	M	→ V
M	×	S	→ M	M	×	V	→ V	M	×	M	→ M

```
template<class l,class r,int N> inline
auto operator * (const iMatrix<l,N>& lhs,const iVector<r,N>& rhs)
       -> iVector<decltype(lhs._internal[0][0]*rhs._internal[0]),N>
{
 typedef decltype(lhs._internal[0][0]*rhs._internal[0]) ret_t;
 iVector<ret_t,N> ret;
 for(int c1=0;c1<N;c1++){
   mult(&ret._internal[c1],&lhs._internal[c1][0],&rhs._internal[0]);
   for(int c2=1;c2<N;c2++){
     mac(&ret._internal[c1],&lhs._internal[c1][c2],&rhs._internal[c2]);
   }
 }
 return ret;
}
```

- QDP++/PETE generates over 50k LOC enumerating the cases for oLattice ⊗ Spin ⊗ Colour ⊗ Reality ⊗ iLattice
- Good example of less code and more general enabled by C++11.

C++11 implementation

Vector loop fusion obtained with *very* compact homegrown expression template engine

- Operator + returns an expression object referencing arguments
- Operator = loops over lattice evaluating expression object on each site

Template meta-programming recurses down expression tree inlining at compile time

```
template <typename Op, typename T1, typename T2>
auto inline eval(const unsigned int ss, const LatticeBinaryExpression<Op,T1,T2> &expr) // eval two operands
  -> decltype(expr.op.func(eval(ss,expr.arg1),eval(ss,expr.arg2)))
{
  return expr.op.func(eval(ss,expr.arg1),eval(ss,expr.arg2))
}
```

Code examples, cshift

```
  namespace PeriodicBC {

    template<class covariant,class gauge> Lattice<covariant> CovShiftForward(const Lattice<gauge> &Link,
                                                                             int mu,
                                                                             const Lattice<covariant> &field)
    {
      return Link*Cshift(field,mu,1);// moves towards negative mu
    }
  }

  namespace ConjugateBC {
  template<class covariant,class gauge> Lattice<covariant> CovShiftForward(const Lattice<gauge> &Link,
                                                                            int mu,
                                                                            const Lattice<covariant> &field)
  {
    GridBase * grid = Link._grid;

    int Lmu = grid->GlobalDimensions()[mu]-1;

    conformable(field,Link);

    Lattice<iScalar<vInteger> > coor(grid);    LatticeCoordinate(coor,mu);

    Lattice<covariant> field_bc = Cshift(field,mu,1);// moves towards negative mu;

    field_bc = where(coor==Lmu,conjugate(field_bc),field_bc);
    //    std::cout<<"Gparity::CovCshiftForward mu="<<mu<<std::endl;
    return Link*field_bc;
  }
  }
// Common wilson loop observables
template <class Gimpl> class WilsonLoops : public Gimpl {
public:
  INHERIT_GIMPL_TYPES(Gimpl);

  ////////////////////////////////////////////////
  // directed plaquette oriented in mu,nu plane
  ////////////////////////////////////////////////
  static void dirPlaquette(GaugeMat &plaq, const std::vector<GaugeMat> &U,
                           const int mu, const int nu) {
    plaq =
      Gimpl::CovShiftBackward(U[mu], mu, Gimpl::CovShiftBackward(U[nu], nu, Gimpl::CovShiftForward(U[mu], mu, U[nu])));
  }
```

Stencil operators

Stencil support eases the pain of optimised matrix multiplies

```
int npoint;
std::vector<int> directions   ;
std::vector<int> displacements;
CartesianStencil Stencil(&CoarseGrid,npoint,Even,directions,displacements)
```

Pass the stencil a list of directions and displacements

```
void M (const CoarseVector &in, CoarseVector &out){
  conformable(_grid,in._grid);
  conformable(in._grid,out._grid);

  SimpleCompressor<siteVector> compressor;
  Stencil.HaloExchange(in,comm_buf,compressor);

PARALLEL_FOR_LOOP
  for(int ss=0;ss<Grid()->oSites();ss++){
    siteVector res = zero;
    siteVector nbr;
    int offset,local,perm,ptype;

    for(int point=0;point<geom.npoint;point++){
      offset = Stencil._offsets [point][ss];
      local  = Stencil._is_local[point][ss];
      perm   = Stencil._permute [point][ss];
      ptype  = Stencil._permute_type[point];

      if(local&&perm) {
        permute(nbr,in._odata[offset],ptype);
      } else if(local) {
        nbr = in._odata[offset];
      } else {
        nbr = comm_buf[offset];
      }
      res = res + A[point]._odata[ss]*nbr;
    }
    vstream(out._odata[ss],res);
  }
  return norm2(out);
};
```

Coarse grid operator in Grid

Stencil organises halo exchange for any vector type; compressor can do spin proj for Wilson fermions.

Stencil provides index of each neighbour (knows the geometry)

User dictates how to treat the internal indices in operator

Wilson Dirac Kernel

```
void WilsonKernels<Impl>::DiracOptGenericDhopSiteDag(StencilImpl &st,LebesgueOrder &lo,DoubledGaugeField &U,
                                                     std::vector<SiteHalfSpinor,alignedAllocator<SiteHalfSpinor> >  &buf,
                                                     int sF,int sU,const FermionField &in, FermionField &out)
{
  SiteHalfSpinor  tmp;
  SiteHalfSpinor  chi;
  SiteHalfSpinor *chi_p;
  SiteHalfSpinor Uchi;
  SiteSpinor result;
  StencilEntry *SE;
  int ptype;

  ///////////////////////////
  // Xp
  ///////////////////////////
  SE=st.GetEntry(ptype,Xp,sF);

  if (SE->_is_local ) {
    chi_p = &chi;
    if ( SE->_permute ) {
      spProjXp(tmp,in._odata[SE->_offset]);
      permute(chi,tmp,ptype);
    } else {
      spProjXp(chi,in._odata[SE->_offset]);
    }
  } else {
    chi_p=&buf[SE->_offset];
  }

  Impl::multLink(Uchi,U._odata[sU],*chi_p,Xp,SE,st);
  spReconXp(result,Uchi);

  ///////////////////////////
  // Yp
  ///////////////////////////
  SE=st.GetEntry(ptype,Yp,sF);

  if ( SE->_is_local ) {
    chi_p = &chi;
    if ( SE->_permute ) {
      spProjYp(tmp,in._odata[SE->_offset]);
      permute(chi,tmp,ptype);
    } else {
      spProjYp(chi,in._odata[SE->_offset]);
    }
  } else {
    chi_p=&buf[SE->_offset];
  }

  Impl::multLink(Uchi,U._odata[sU],*chi_p,Yp,SE,st);
  accumReconYp(result,Uchi);
```

Aside on serialisation

Object reflection/automatic serialisation has been a problem for C, C++ for decades.
Solved this using variadic macros; substantially better implementation than Boost HANA.
Sun should have done this in 1984 and avoided decades of IDL's (RPCGEN, CORBA, SOAP...).

```
class myclass {
public:
  GRID_DECL_CLASS_MEMBERS(myclass,
                          int, x,
                          double, y,
                          bool , b,
                          std::string, name,
                          std::vector<double>, array);
};

{
  myclass obj;
  XmlWriter WR("bother.xml");
  write(WR,"obj",obj);
}
```

Multi-Grid for Chiral Fermions

- Using Grid as a rapid development environment
- multi-grid transfer interface

```
template<class vobj,class CComplex,int nbasis>
inline void blockProject(Lattice<iVector<CComplex,nbasis > > &coarseData,
                         const            Lattice<vobj>   &fineData,
                         const std::vector<Lattice<vobj> > &Basis)

template<class vobj,class CComplex>
inline void blockInnerProduct(Lattice<CComplex> &CoarseInner,
                              const Lattice<vobj> &fineX,
                              const Lattice<vobj> &fineY)

template<class vobj,class CComplex>
inline void blockNormalise(Lattice<CComplex> &ip,Lattice<vobj> &fineX)

template<class vobj,class CComplex>
inline void blockOrthogonalise(Lattice<CComplex> &ip,std::vector<Lattice<vobj> > &Basis)

template<class vobj,class CComplex,int nbasis>
inline void blockPromote(const Lattice<iVector<CComplex,nbasis > > &coarseData,
                         Lattice<vobj>   &fineData,
                         const std::vector<Lattice<vobj> > &Basis)

inline void subdivides(GridBase *coarse,GridBase *fine)
```

Grid performance

Architecture	Cores	GF/s (Ls x Dw)	peak
Intel Knight's Landing 7250	68	770	6100
Intel Knight's Corner	60	270	2400
Intel Broadwellx2	36	800	2700
Intel Haswellx2	32	640	2400
Intel Ivybridgex2	24	270	920
AMD Interlagosx4	32 (16)	80	628

Grid performance

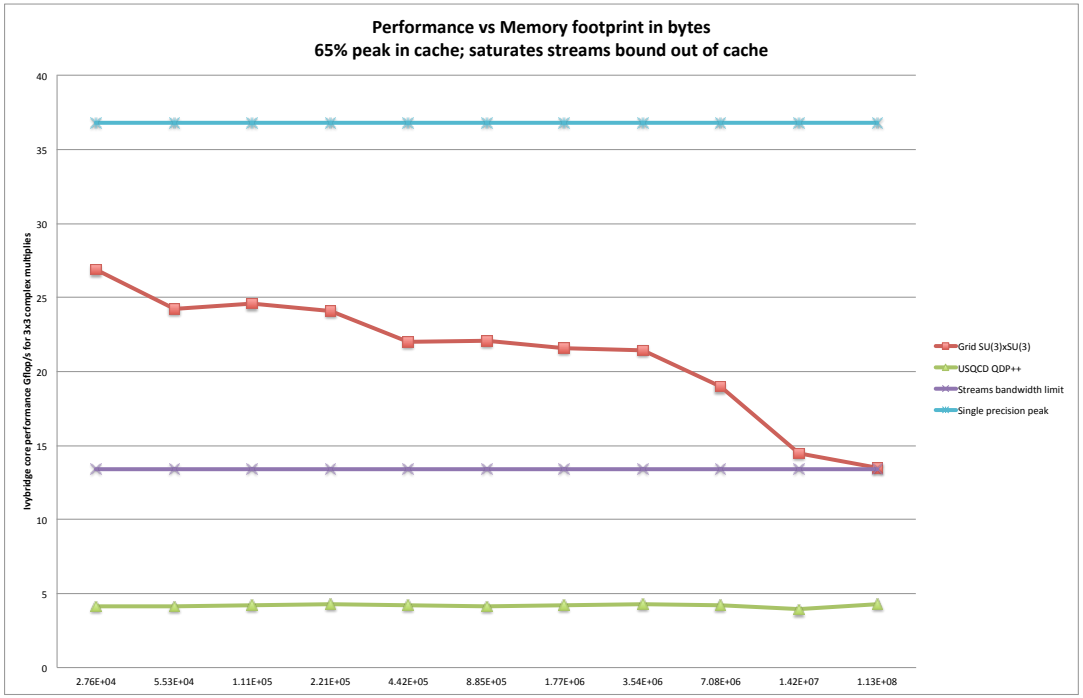
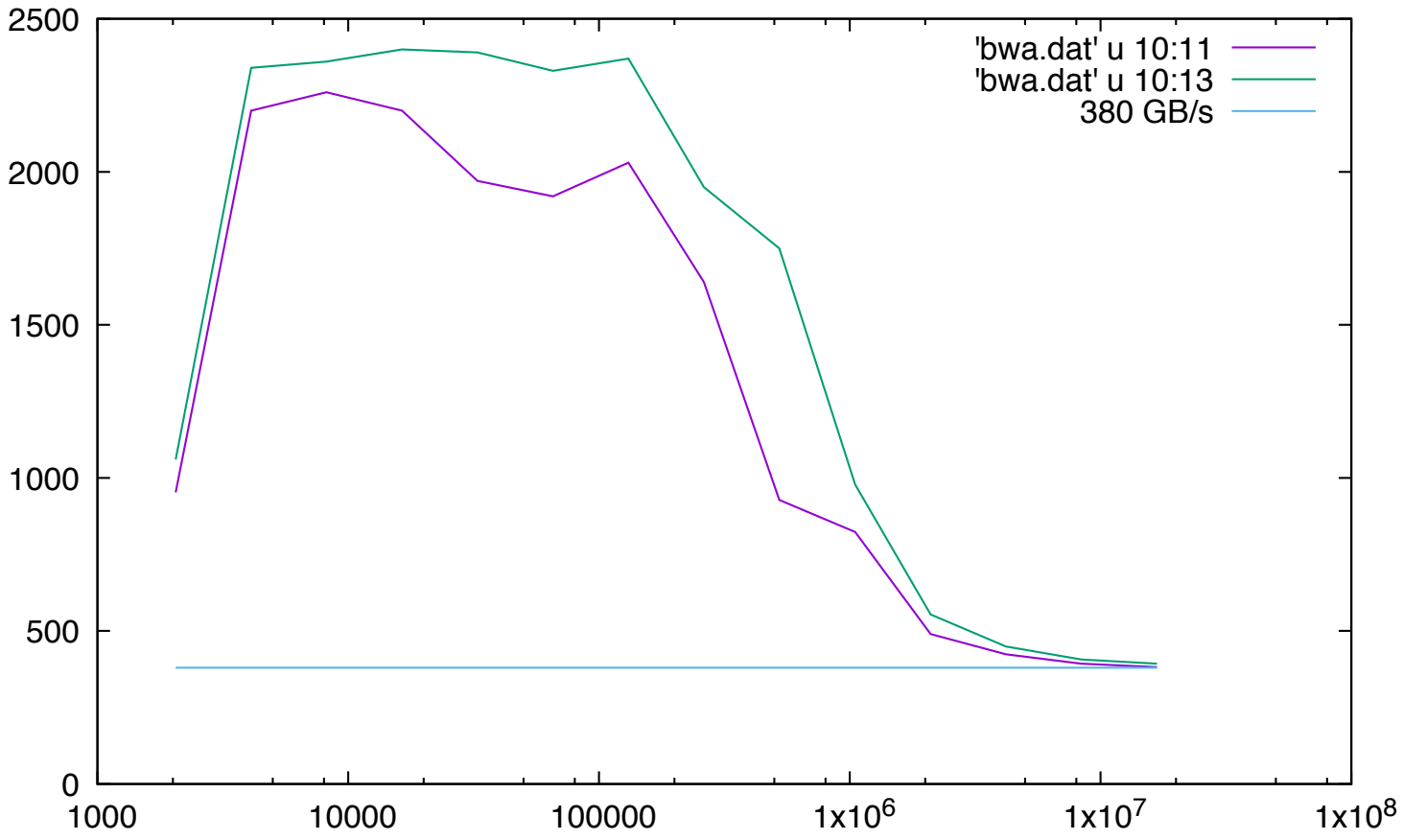


Figure 4: We compare the performance of Grid (red) on SU(3)×SU(3) matrix multiplication to peak (blue), the limit imposed by memory bandwidth (purple), and to that of the QDP++ code system (green).

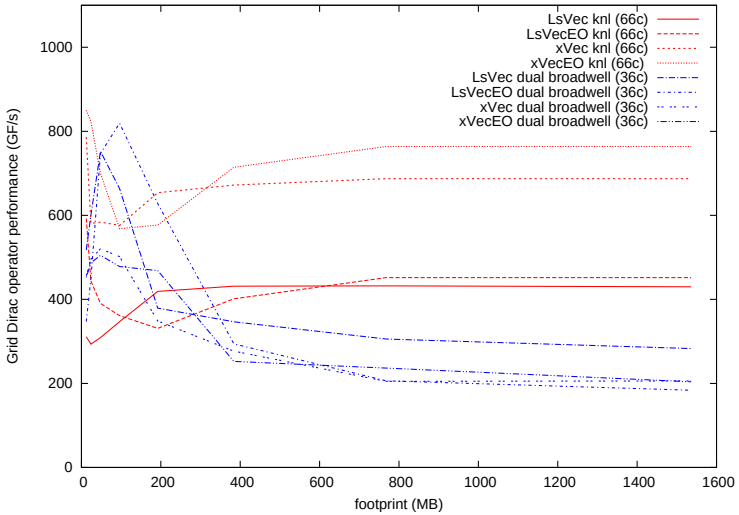
SU3 x SU3 example

Grid performance



Knight's Landing memory system profile (GB/s vs footprint bytes/core)

Grid multi-RHS Wilson Dslash and DWF



- Grid single node, single precision performance for multiRHS Wilson term
- Knight's Landing 7250, 68 core
  - Used 66 cores - a few empty cores usually faster
- One KNL substantially faster than two Broadwell's (18+18) out of cache

- 1 thread per core fastest after writing in assembler (*not* intrinsics)
  - Macro system and mixed C++/asm minimises pain
  - Hand allocation of registers evades stack eviction, cache more deterministic
  - Hand prefetch to L2 and to L1
  - 8.2.2.2 cache blocking
  - Less reuse than I hoped for
- Single core instructions-per-cycle is 1.7 (85% of theoretical)
- Multi-core L1 hit rate is 99% (perfect SFW prefetching)
- Multi-core MCDRAM bandwidth 97% (370GB/s)
- Provably unimprovable ?

Wilson Dslash Kernel

C++ implementation

```
////////////////////////////
// Yp
////////////////////////////
SE=st.GetEntry(ptype,Yp,sF);

if ( SE->_is_local ) {
  chi_p = &chi;
  if ( SE->_permute ) {
    spProjYp(tmp,in._odata[SE->_offset]);
    permute(chi,tmp,ptype);
  } else {
    spProjYp(chi,in._odata[SE->_offset]);
  }
} else {
  chi_p=&buf[SE->_offset];
}

Impl::multLink(Uchi,U._odata[sU],*chi_p,Yp,SE,st);
accumReconYp(result,Uchi);
```

Wilson Dslash Kernel

Inline assembly implementation
Macro system reduces level of barbarism

```
#define MULT_2SPIN(ptr,pf)                                  \
  LOAD64(%r8,ptr)                                                \
  LOAD64(%r9,pf)                                                 \
  __asm__ (                                                      \
           VSHUF(Chi_00,T1)      VSHUF(Chi_10,T2)                \
           VMULIDUP(0,%r8,T1,UChi_00) VMULIDUP(0,%r8,T2,UChi_10)  \
           VMULIDUP(3,%r8,T1,UChi_01) VMULIDUP(3,%r8,T2,UChi_11)  \
           VMULIDUP(6,%r8,T1,UChi_02) VMULIDUP(6,%r8,T2,UChi_12)  ....
etc..

  ////////////////////////////////
  // Yp
  ////////////////////////////////
  basep = st.GetPFInfo(nent,plocal); nent++;
  if ( local ) {
    LOAD64(%r10,isigns);
    YM_PROJMEM(base);
    MAYBEPERM(PERMUTE_DIR2,perm);
  } else {
    LOAD_CHI(base);
  }
  base  = st.GetInfo(ptype,local,perm,Zp,ent,plocal); ent++;
  PREFETCH_CHIMU(base);
  {
    MULT_2SPIN_DIR_PFYP(Yp,basep);
  }
  LOAD64(%r10,isigns);
  YM_RECON_ACCUM;
```

Implementation status

- Basic Grid type system essentially complete
  - QCD types, generic SU(N), arbitrary dimension
  - Sum, SliceSum
  - To do: Fast Fourier Transform
- Algorithms
  - Quenched heatbath
  - Hybrid Monte Carlo, Rational Hybrid Monte Carlo
  - CG, MCR, GCR, Two level CG/GCR
  - Chebyshev approx, Remez, Multishift CG
  - Chebyshev preconditioned Lanczos (Jung, Arthur)
- Fermion dirac operators
  - Even-odd and unpreconditioned have a single unified definition
  - Wilson, WilsonTM
  - {Wilson, Shamir, Mobius } - Kernel
    $\otimes$ {Zolotarev, Tanh } - Approximation
    $\otimes$ {Continued Fraction, Partial Fraction , Cayley } - Representation
  - DWF, Mobius as special cases
  - To do: (aniso) Clover, RHQ
- Smeared link evolution
- To do: Generic fermion reps of SU(N)

HDCR

Peter Boyle, Azusa Yamaguchi

Multigrid for 5d chiral Fermions

Aim to produce a true multi-grid algorithm for 5d Chiral Fermions

- Seek to coarsen a nearest neighbour operator
  - *but folklore says DWF only converges with conjugate gradients on normal equations !*

First attempt: Hierarchically deflated conjugate gradient (HDCG, arXiv:1402.2585)

- Based on CGNE on the squared, preconditioned operator
- Deflates $M_{pc}^{\dagger} M_{pc}$ next-next-next-to-nearest neighbour (320 point stencil!)
  - Suitable for valence analysis
  - enables 17x speed up over CGNE double precision with no loss of accuracy.
- Not a true multigrid as sparsity pattern of coarse space representation fills in
  - Not appropriate for use in HMC
  - Even if the subspace can be tracked, cost challenging
- Under constraint of blocks $\geq 4^4$, 81 point stencil
- Requires $\approx 81 \times 64 = 5184$ matrix multiplies to recompute little dirac operator.
- This is a consequence of basing algorithm on normal equations

Key question: why do 5d Chiral Fermions require CGNE ?

- Normal equations: for propagators use source η^0

$$M^\dagger M\psi = M^\dagger \eta^0$$

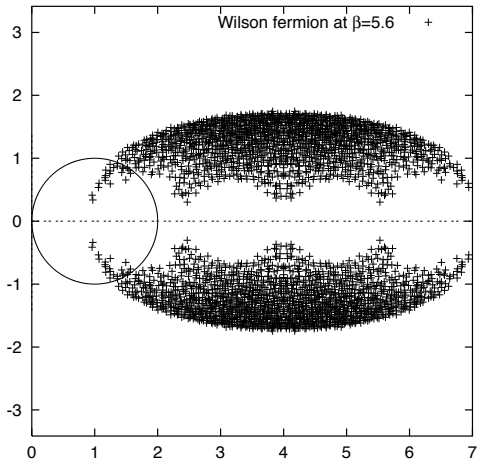
- Non-hermitian system "does not converge" with any known Krylov algorithm

$$M\psi = \eta$$

- Why? Krylov solvers form a *polynomial approximation* to M^{-1}

$$\psi = P(M)\eta$$

  where the polynomial coefficients are chosen via some algorithm, e.g. GMRES, GCR etc.
- What goes wrong with DWF ?



- Wilson spectrum lies in right half plane - the "Big Mac" plot [a]
- DWF spectrum shifted by negative massto place zero in centre of first opening
  - Violates the *folklore* present in numerical analysis of the *half-plane condition*.
  - c.f. *N. M. Nachtigal, S. C. Reddy, and L. N. Trefethen, How Fast are Nonsymmetric Matrix Iterations? SIAM. J. Matrix Anal. Appl., 13(3), 778795.*
- There is a fundamental explanation for this folklore

[a] reused from hep-lat/0007017

Spectral reason CGNE needed for DWF

- In the infinite volume the spectrum becomes dense
- Krylov solver must approximate $P(z) \to \frac{1}{z}$ over region in complex plane *encircling* the pole at zero
- It is *impossible* to reproduce the phase winding over this region with any polynomial

$$\oint z^{-1} dz = 2\pi i \neq \oint P(z) dz = 0$$

- Minimising L2 norm of error over a circle of fixed radius gives all polynomial coefficients as zero!

CGNE is the conventional way out: multiply by $\bar{z}$ to make the problem real, pos def.
There exists a good polynomial apprximation

$$P(\bar{z}z) \approx \frac{1}{\bar{z}z} \qquad ; \qquad \bar{z}z \in (0, \infty)$$

Then

$$P(\bar{z}z)\bar{z} \approx \frac{1}{z\bar{z}}\bar{z} = \frac{1}{z}$$

- Time to play the γ_5 joker!

Using γ_5 Hermiticity

Phase response is the problem: make the system real *indefinite* using γ_5, in two/three different ways.

Cayley form for DWF and $c = 0$ Mobius

- Domain wall fermions: $H_{dwf} = \gamma_5 R_5 D_{dwf} = \Gamma_5 D_{dwf}$
- Mobius fermions with $c = 0$, $b \neq 1$

Continued fraction form for H_W kernel is already Hermitian indefinite:

$$\begin{bmatrix} H & \frac{1}{\sqrt{\beta_0\beta_1}} & 0 & 0 & 0 \\ \frac{1}{\sqrt{\beta_1\beta_0}} & -H & \frac{1}{\sqrt{\beta_1\beta_2}} & 0 & 0 \\ 0 & \frac{1}{\sqrt{\beta_2\beta_1}} & H & \frac{1}{\sqrt{\beta_2\beta_3}} & 0 \\ 0 & 0 & \frac{1}{\sqrt{\beta_3\beta_2}} & -H & \frac{1}{\sqrt{\beta_3}} \\ 0 & 0 & 0 & \frac{1}{\sqrt{\beta_3}} & R\gamma_5 + \beta_0 H \end{bmatrix}$$

These operators are *nearest neighbour* and preserve sparsity in a coarse space.

Conjugate residual: a Hermitian indefinite solver

Conjugate residual iterates

```
a = rAr/pAAp;
axpy(psi,a,p,psi);
cp = axpy_norm(r,-a,Ap,r);

rArp=rAr;
Linop.HermOpAndNorm(r,Ar,rAr,rAAr);

b   =rAr/rArp;
axpy(p,b,p,r);
pAAp=axpy_norm(Ap,b,Ap,Ar);
```

- Creates a Krylov space that strictly contains the CGNE Krylov space

$$P_N(D^\dagger D)D^\dagger = P_N(\Gamma_5 D \Gamma_5 D)\gamma_5 D \gamma_5 \subset P_{2N+1}(H_{dwf})\gamma_5 = P_{2N+1}(\Gamma_5 D)\gamma_5$$

  - Solve conjugate residual on Hermitian operator

$$\gamma_5 D \psi = \gamma_5 \eta$$

  - Spectral density has odd symmetry in the infinite volume limit $\Rightarrow$ All even terms in polynomial must vanish
  - Converges in similar number of matrix multiplies
- Use this to construct a deflation algorithm for hermitian indefinite operator.
- This remains nearest neighbour in course space, giving a true multigrid

Hierarchically deflated conjugate residual

Variable preconditioned GCR:

```
      for(int k=0;k<MaxIterations;k++){
cp=GCRnStep(Linop,src,psi,rsq);
        // Check for convergence
      }
```

GCRnStep:

```
for(int k=0;k<nstep;k++){
int kp     = k+1;

rq= real(innerProduct(r,q[k])); // what if rAr not real?
a = rq/qq[k];

axpy(psi,a,p[k],psi);

cp = axpy_norm(r,-a,q[k],r);

if((k==nstep-1)||(cp<rsq)) return cp;

Preconditioner(r,z);

Linop.HermOpAndNorm(z,Az,zAz,zAAz);

q[kp]=Az;
p[kp]=z;

        // Orthogonalise q to previous q's.
}
```

Hierarchically deflated conjugate residual

Multigrid is introduced as a *Preconditioner*
Low mode subspace vectors ϕ generated in some way: tried

- Inverse iteration (c.f. Luscher)
- Lanczos vectors
- Chebyshev filters

$$\phi_k^b(x) = \begin{cases} \phi_k(x) & ; \quad x \in b \\ 0 & ; \quad x \notin b \end{cases} \tag{1}$$

$$\mathrm{span}\{\phi_k\} \subset \mathrm{span}\{\phi_k^b\}. \tag{2}$$

$$P_S = \sum_{k,b} |\phi_k^b\rangle\langle\phi_k^b| \qquad ; \qquad P_{\bar{S}} = 1 - P_S \tag{3}$$

$$M = \begin{pmatrix} M_{\bar{S}\bar{S}} & M_{S\bar{S}} \\ M_{\bar{S}S} & M_{SS} \end{pmatrix} = \begin{pmatrix} P_{\bar{S}} M P_{\bar{S}} & P_S M P_{\bar{S}} \\ P_{\bar{S}} M P_S & P_S M P_S \end{pmatrix} \tag{4}$$

We can represent the matrix M exactly on this subspace by computing its matrix elements, known as the *little Dirac operator* (coarse grid matrix in multi-grid)

$$A_{jk}^{ab} = \langle\phi_j^a|M|\phi_k^b\rangle \qquad ; \qquad (M_{SS}) = A_{ij}^{ab}|\phi_i^a\rangle\langle\phi_j^b|. \tag{5}$$

the subspace inverse can be solved by Krylov methods and is:

$$Q = \begin{pmatrix} 0 & 0 \\ 0 & M_{SS}^{-1} \end{pmatrix} \qquad ; \qquad M_{SS}^{-1} = (A^{-1})_{ij}^{ab}|\phi_i^a\rangle\langle\phi_j^b| \tag{6}$$

It is important to note that A inherits a sparse structure from M because well separated blocks do *not* connect through M.

Hierarchically deflated conjugate residual

We can Schur decompose the matrix

$$M = UDL = \begin{bmatrix} M_{\bar{s}\bar{s}} & M_{\bar{s}s} \\ M_{s\bar{s}} & M_{ss} \end{bmatrix} = \begin{bmatrix} 1 & M_{\bar{s}s} M_{ss}^{-1} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} S & 0 \\ 0 & M_{ss} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ M_{ss}^{-1} M_{s\bar{s}} & 1 \end{bmatrix}$$

Note that $P_L M = \begin{bmatrix} S & 0 \\ 0 & 0 \end{bmatrix}$ yields the Schur complement $S = M_{\bar{s}\bar{s}} - M_{\bar{s}s} M_{ss}^{-1} M_{s\bar{s}}$, and that the diagonalisation L and U are related to Luscher's projectors P_L and P_R (Galerkin oblique projectors in multi-grid)

$$P_L = P_{\bar{S}} U^{-1} = \begin{pmatrix} 1 & -M_{\bar{S}S} M_{SS}^{-1} \\ 0 & 0 \end{pmatrix} \quad ; \quad P_R = L^{-1} P_{\bar{S}} = \begin{pmatrix} 1 & 0 \\ -M_{SS}^{-1} M_{S\bar{S}} & 0 \end{pmatrix} \tag{7}$$

Hierarchically deflated conjugate residual

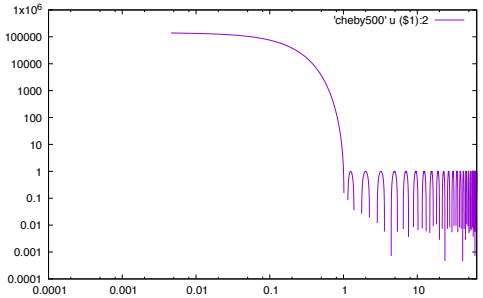
Multigrid

$$M_{chebyshev} P_L + P_R M_{chebyshev} + Q - M_{chebyshev} P_L \mathscr{H} M_{chebyshev}$$

```
SmootherOperator.AdjOp(in,vec1);// this is the G5 herm bit
ChebyAccu(fMdagMOp,vec1,out);   // solves  MdagM = g5 M g5M

                           // Update with residual for out
FineOperator.Op(out,vec1);// this is the G5 herm bit
vec1  = in - vec1;         // tmp  = in - A Min

Aggregates.ProjectToSubspace  (Csrc,vec1);
HermOp.AdjOp(Csrc,Ctmp);// Normal equations
CG(MdagMOp,Ctmp,Csol);
Aggregates.PromoteFromSubspace(Csol,vec1); // Ass^{-1} [in - A Min]s
                                           // Q = Q[in - A Min]
out = out+vec1;

// Three preconditioner smoothing -- hermitian if C3 = C1
FineOperator.Op(out,vec1);// this is the G5 herm bit
vec1  = in - vec1;   // tmp  = in - A Min

// Reapply smoother
SmootherOperator.Op(vec1,vec2);  // this is the G5 herm bit
ChebyAccu(fMdagMOp,vec2,vec1);    // solves  MdagM = g5 M g5M

out =out+vec1;
```

Hierarchically deflated conjugate residual

Test system: $16^3 \times 32 \times 16$. Set mass artificially low 0.001
16 nodes on Cori; 16 vectors O(900) polynomial



Algorithm	setup/vecs	Fine Matmuls	Time
CGNE	-	3221	110s
HDCR	Lanczos/16		45s
HDCR	$M^{-1}/16$		120s
HDCR	Cheby/16		40s
Coarse			13s
Fine		624	27s
Chebyshevs			300s
Ldop calc	4x gain possible		200s→50s

- Unable to use red black preconditioning in smoother (approx solving unpreconditioned $M^\dagger M$)
- Set up to solve unsquared system
$$\Gamma_5 D_{dwf} \psi = \Gamma_5 \eta$$
- HMC: ideally solve squared system
$$M_{pc}^\dagger M_{pc} \psi = \eta$$
- MG-HMC: people solve
$$M^\dagger M \psi = \eta$$
and drop red-black from the HMC
- NB not yet deflating the coarse space at all (gave 3x on coarse space for HDCG)

Summary

- First *true* multi-grid for Chiral fermions: DWF and Overlap covered
- Nearest neigbour coarsening admits many levels and cheaper reevaluation of coarse operator
- Wall clock speed up on 16^3 at around 2000 CG iterations
- New Chebyshev subspace initialisation is cheaper/better
- Significant work remains to integrate in two flavour HMC
- Code already in "Grid"; runs under SSE, AVX, AVX2, AVX512, IMCI on Xeons and Many-core
  Grid: A next generation data parallel C++ QCD library, Peter Boyle, Azusa Yamaguchi, Guido Cossu, Antonin Portelli; arXiv:1512.03487
  Hierarchically deflated conjugate gradient; P A Boyle; arXiv:1402.2585