Lattice QCD on Intel Xeon Phi's

Peter Labus

in collaboration with Bartosz Kostrzewa and Martin Ueding
@ HISKP Universität Bonn

JSC Jülich
10.02.2016

LQCD & the rôle of the *dslash* stencil

- Calculate integrals of the form

$$\int \mathcal{D}\Phi \; f[\Phi] \; e^{-S_{QCD}[\Phi]}$$

  after *discretising* space-time through a lattice

- Algorithmic layers of LQCD:
  1. Hybrid Monte Carlo
  2. Efficient Krylov Solvers for $Mx = b$ in Molecular Dynamics
  3. BLAS linear algebra

- Most *expensive* part:

  Matrix times Vector (very large 10^9 squared & very sparse), described by (next-to) nearest-neighbour stencil

The QPhiX library (mainly: Balint Joo, Jefferson Lab)

- Aim: provide stencil operations for **general vector machines**
- Parallelised via QMP + OpenMP + SIMD (vector intrinsics)
- C++11 template library with external C++ code generator ($\sim 25k$ lines of code + $10k$ testing & timing)
- Implements
  1. Wilson / Wilson-Clover *dslash* stencils
  2. BLAS linear algebra
  3. Krylov Solvers (CG, BiCGStab, Mixed Precision, MultiCG)
- Common template parameters:
  1. `typename FT`
  2. `int VECLEN`
  3. `int SOALEN`
  4. `bool COMPRESS12`
- **Our goal: Integrate Twisted-Mass Fermions in QPhiX**
  This amounts to a change in the Fermion Matrix M …

The Fermion Matrix

Full (even-odd preconditioned) matrix is

$$M = A - \frac{1}{4} \not{D} A^{-1} \not{D} ,$$

where $\not{D}$ is the nearest-neighbour ***dslash* stencil**.

M can be constructed from two BLAS Matrx-Vector *kernels*:

$$(1) \;\; A^{-1} \not{D} x , \qquad\qquad (2) \;\; A x - \frac{1}{4} \not{D} y .$$

The vectors x and y are called *spinors*.

The **Clover Term** A includes the new *twisted-mass*:

$$A \equiv \alpha \mathbb{1} + c_{sw} T \pm i\mu\gamma_5 ,$$

where c_{sw} and μ are user parameters, while α is fixed.

The *dslash* stencil in action

$$
\begin{aligned}
\chi^a_\alpha(x) &= \sum_{y\in\Lambda}\sum_{b=0}^{2}\sum_{\beta=0}^{3} \not{D}^{ab}_{\alpha\beta}(x,y)\psi^b_\beta(y) \\
&= \sum_{b=0}^{2}\sum_{\beta=0}^{3}\sum_{\mu=0}^{3}\Big[U^{ab}(x,x+\hat{\mu})\ (\mathbb{1}-\gamma_\mu)_{\alpha\beta}\ \psi^b_\beta(x+\hat{\mu})+ \\
&\qquad\qquad\qquad U^{\dagger\,ab}(x,x-\hat{\mu})\ (\mathbb{1}+\gamma_\mu)_{\alpha\beta}\ \psi^b_\beta(x-\hat{\mu})\Big]
\end{aligned}
$$

- $x,\ y$ points/sites of the lattice Λ
- ψ input **spinor**: $3 \times 4 \times 2 = 24$ components per site
- χ output spinor
- U **gauge** or link variables: $3 \times 3 \times 2 = 18$ components per site
- γ_μ: 4 constant 4×4 complex matrices
- $\hat{\mu}$: 4 unit vectors (one for each space-time dimension)

The Clover Term

A is a 12×12 complex, block-diagonal matrix (each lattice site):

$$A = \begin{pmatrix} (\alpha \pm i\mu)\,\mathbb{1}_6 + c_{SW} B_0 & \mathbb{0}_6 \\ \mathbb{0}_6 & (\alpha \mp i\mu)\,\mathbb{1}_6 + c_{SW} B_1 \end{pmatrix}$$

QPhiX so far:

- $\mu = 0 \Rightarrow A$ and A^{-1} are hermitian (and block-diagonal)
- need to store only two blocks with
  - 6 real diagonal elements
  - 15 complex off-diagonal elements

We want: $\mu \neq 0$

- then A is not hermitian
- in particular A^{-1} is dense (but still block-diagonal)
- need to store two full 6×6 complex blocks

Arithmetic Intensities

$$I_A = \text{FLOPs per Byte of moved data}$$

***dslash* stencil:** $\not{D}\, x$ 1320 FLOPs per site

$$I_A = 1.06 \text{ FLOP/Byte}$$

combined multiplication with clover term: $A^{-1}\not{D}\, x$ 1872 FLOPs per site

$$\mu = 0 : \; I_A = 1.17 \text{ FLOP/Byte}$$
$$\mu \neq 0 : \; I_A = 0.98 \text{ FLOP/Byte}$$

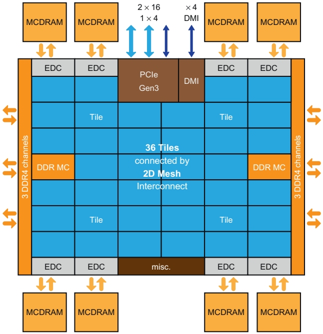
Theoretical Peaks on KNL / KNC:

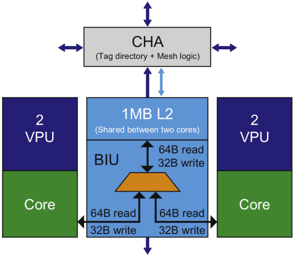
$$I_A = 15.4 \,/\, 6.3 \text{ FLOP/Byte}$$

ALL ROUTINES ARE MEMORY BANDWIDTH BOUND
Expect about 20% less performance with twisted-mass

Combining these Intensities with a simple hardware model and the (measured) memory bandwidth gives us a good *rooftop model* ...

Intel Xeon Phi Knights Landing (KNL)





- up to 72 cores @ 1.5 GHz (3.5 TFLOP/s)
- 2D Mesh: 700 GB/s
- 8 GB MCDRAM: 450 GB/s
- 384 GB DDR-4: 90 GB/s

- 2 cores / 4 VPUs
- 512-bit Vector Registers (AVX-512)
- 4 hardware hyper-threads
- 1 MB shared L2 cache

General Programming Implications

Three Guidelines...

1. **Scaling**
2. **Vectorisation**
3. **Data Locality & Cache Re-use**

... and how they are implemented in QPhiX

1. **OpenMP thread scheduling** for lattice traversal
2. Multi-ISA **C++ code generator** using Vector Intrinsics
3. **Cache blocking** into L2 and **Structures of Arrays** data layout (*tiles*)

Optimisations in QPhiX-codegen:
L1 & L2 software prefetches, dimensions of tiles (SOALEN), streaming stores, gather/scatter, ...

Data layout & Cache blocking

- Elementary vector elements are *Structures of Arrays* (SoA):

  `typedef FT FourSpinorBlock[3][4][2][SOALEN];`

- SOALEN may be a factor of VECLEN $= \mathtt{ngy} * \mathrm{SOALEN}$:

  form *tiles* in X-Y-plane (Xeon Phi SP: 16×1, 8×2, 4×4)

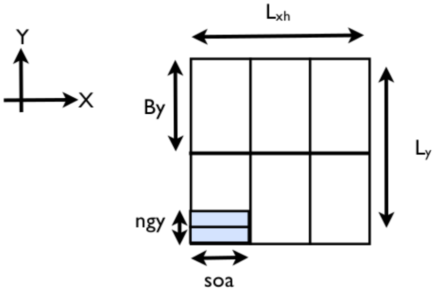
- Then divide lattice in blocks:

  X: SOALEN

  Y, Z: By, Bz

  such that 3 T-Slices fit into L2

- Distribute blocks to cores/threads & process in chunks of SIMD vectors, *i.e.* tiles



QPhiX-codegen

- Three abstract (class) objects:
  1. `Instruction`: FLOPs, Memory inst., scope delimiters, if-else, ...
  2. `Address`: Memory addresses to deal with pointers & offsets
  3. `FVec`: “The registers” which are fed to intrinsic calls, *i.e.* variables in C++ code (*e.g.* of type `__m512d`)
- First two classes: `std::string serialize(void)`
- Save Instructions into `InstVector`’s and dumb to file

```
for( auto ops : kernels ) {

  InstVector ivector;
  InstVector l2prefs;
  std::ostringstream filename;

  filename << ...;

  // Generate Instructions
  generateL2Prefetches(l2prefs, compress12, chi_prefetches, clover, twisted_mass);
  dslash_plain_body(ivector, compress12, clover, twisted_mass, isPlus);
  mergeIvectorWithL2Prefetches(ivector, l2prefs);
  dumpIVector(ivector, filename.str());

}
```

Code Example 1: NEW Clover Term Multiplication

```
typedef struct {
  FT diag1[6][VECLEN];
  FT off_diag1[15][2][VECLEN];
  FT diag2[6][VECLEN];
  FT off_diag2[15][2][VECLEN];
} Clover;
```

```
typedef struct {
  FT block1[6][6][2][VECLEN];
  FT block2[6][6][2][VECLEN];
} FullClover;
```

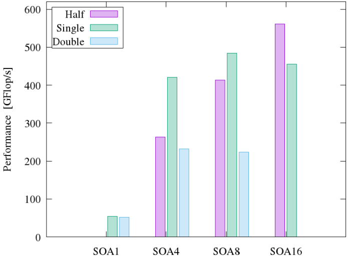
```
void full_clover_term(InstVector& ivector,FVec in_spinor[4][3][2],bool face,std::string mask)
{
  for(int block=0; block!=2; ++block) {

    PrefetchL1FullCloverFullBlockIn(ivector, clBase, clOffs, block);
    LoadFullCloverFullBlock(ivector, clov_full, clBase, clOffs, block);

    for(int sc1=0; sc1!=6; ++sc1) { // half-spin-colour row

      // Calculate out indices
      FVec *clout  = out_spinor[spin_out][col_out];

      for(int sc2=0; sc2!=6; ++sc2) { // half-spin-colour column

        // Calculate in indices
        FVec *clin  = in_spinor[spin_in][col_in];

        if(sc2 == 0 && !face)   mulCVec(ivector,clout,clov_full[sc1][sc2],clin,mask);
        else                    fmaddCVec(ivector,clout,clov_full[sc1][sc2],clin,clout,mask);
      }
    }
  }
}
```

Code Example 2: Extending the User Interface

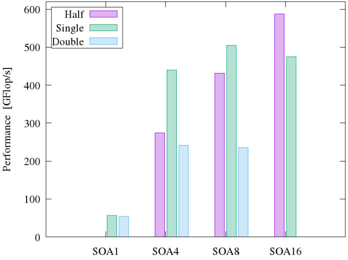
```
template <typename FT, int veclen, int soalen, bool compress12>
class TMClovDslash {
  public:
    // Constructor (padded and aligned data allocation)
    TMClovDslash(Geometry<FT,veclen,soalen,compress12>* geom_,
        double t_boundary_,
        double dslash_aniso_s_,
        double dslash_aniso_t_);

    // Destructor
    ~TMClovDslash();

    // Geometry Getter
    Geometry<FT,veclen,soalen,compress12>& getGeometry(void) { return (*s); }

    // A^{-1} * D * psi
    // (lattice traversal w/ pointer and offset calculation & MPI comm)
    void dslash(FourSpinorBlock* res,
        const FourSpinorBlock* psi,
        const SU3MatrixBlock* u,
        const FullCloverBlock* invclov[2],
        int isign,
        int cb);

    // A*chi - b*D*psi
    void dslashAChiMinusBDPsi(FourSpinorBlock* res,
        const FourSpinorBlock* psi,
        const FourSpinorBlock* chi,
        const SU3MatrixBlock* u,
        const FullCloverBlock* clov[2],
        const double beta,
        int isign,
        int cb);
}
```

Performance of $A^{-1} \not{D} x$ Kernels

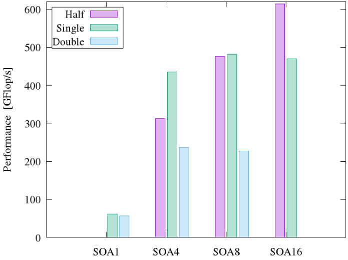
$c_{sw} = 0,\ \mu = 0$



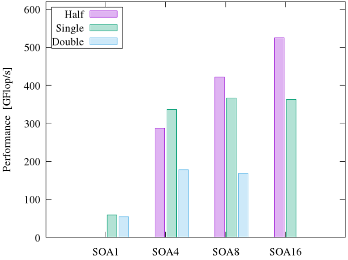
$c_{sw} = 0,\ \mu \neq 0$



$c_{sw} \neq 0,\ \mu = 0$



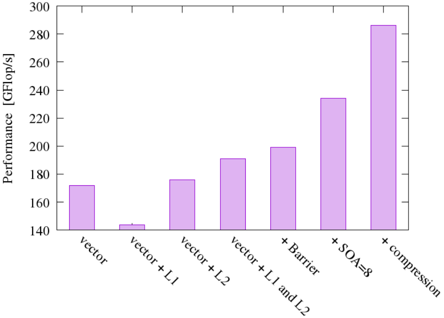
$c_{sw} \neq 0,\ \mu \neq 0$



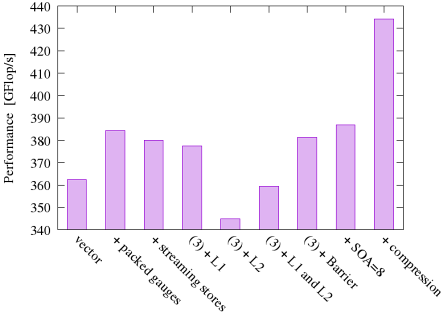
Optimisation Features on KNC vs. KNL

Displayed is the basic *dslash* stencil ($c_{sw} = \mu = 0$) in single precision with SOA $= 16$.

KNC:



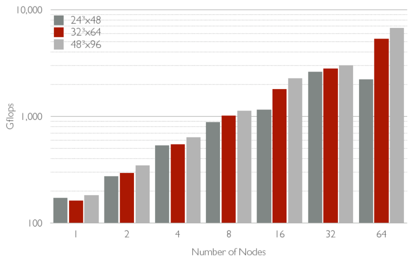
KNL:



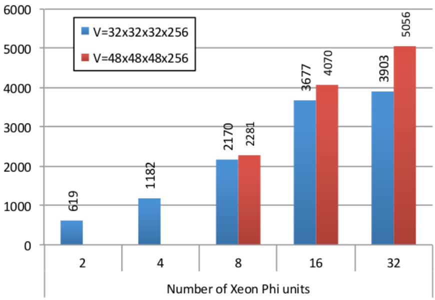
- KNL natively supports prefetches & streaming stores much better than KNC
- Visible benefits from data layout tuning (shown here: SoA length tuning)

Multi-node results

Dual-Socket Xeon (Haswell):



Xeon Phi (KNC):



- STRONG scaling up to 64 Xeon / 16 KNC nodes for various volumes
- WEAK scaling up to 64 KNC nodes

Conclusions & Outlook

- Added new data type & dynamic memory allocation facilities
- Implemented low-level Twisted-Mass-Clover kernels & hardware feature utility functions
- Extended User Interface, added Testing & Timing for new kernels, expanded autoconf/automake setup …
- Confirmed performance expectations on single KNL's processors

Conclusions & Outlook

- Added new data type & dynamic memory allocation facilities
- Implemented low-level Twisted-Mass-Clover kernels & hardware feature utility functions
- Extended User Interface, added Testing & Timing for new kernels, expanded autoconf/automake setup …
- Confirmed performance expectations on single KNL's processors

- Interface for tmLQCD
- Non-degenerate Twisted-Mass/Twisted-Mass-Clover *dslash*
- Twisted boundary conditions
- New Algorithms: Domain decomposition, Multi-grid, …?

Conclusions & Outlook

- Added new data type & dynamic memory allocation facilities
- Implemented low-level Twisted-Mass-Clover kernels & hardware feature utility functions
- Extended User Interface, added Testing & Timing for new kernels, expanded autoconf/automake setup …
- Confirmed performance expectations on single KNL's processors

- Interface for tmLQCD
- Non-degenerate Twisted-Mass/Twisted-Mass-Clover *dslash*
- Twisted boundary conditions
- New Algorithms: Domain decomposition, Multi-grid, …?

THANKS!