Efficient (and power efficient) computing in particle physics

Peter Boyle, University of Edinburgh

- **The Lattice QCD challenge**
- Optimising for BlueGene/Q
- BG/Q performance
- Optimising for x86 multi-core (Archer...)
- Future: Optimising for Knights series

Wilson Dirac Operator

Usual Wilson matrix is

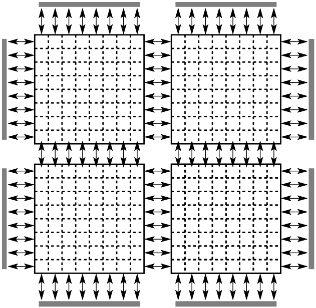
$$D_W(M) = M + 4 - \frac{1}{2} D_{\mathrm{hop}},$$

where

$$D_{\mathrm{hop}} = (1 - \gamma_\mu) U_\mu(x) \delta_{x+\mu,y} + (1 + \gamma_\mu) U_\mu^\dagger(y) \delta_{x-\mu,y} \tag{1}$$

Dirac equation is a classic sparse matrix problem

- Geometrical decomposition on multiple nodes
- Halo exchange communication [4d]
- Time to solution critical
- Matrix is regular, structured, block band-diagonal
  Dense 3×3 complex blocks
  Different coefficients in each block



(Simplified) sparse matrix performance analysis

Model time to apply Wilson operator as $t_{Wilson} = Max\{t_{comm}, t_{fpu}, t_{memory}, t_{cache}\}$

Wilson operator D_W

- $2 \times 24L^4$ words to memory
- $9 \times 24L^4$ words to cache [1]
- $16 \times 12L^3$ words bidi comms

FPU

- $1320 \times L^4$ flops: 480 MADDS, 96 MULS, 264 ADDS

Challenge: design network and memory bandwidth so $t_{cache}, t_{comm}, t_{memory} \approx t_{fpu}$

Assumptions

- When coded right these will take place concurrently. The longest will determine time
- loop order will maximise cache reuse; count *compulsory* memory traffic
- Inverter working set does not fit in cache

[1] "cache" really means the highest level of memory at which reuse can occur. This may be some form of local memory on certain systems.

How fast can a computer go?

$B_N/B_M/B_C$ are Network/Memory/Cache bandwidths (fp words/sec)

- Scalability limited when t_{comm} large $\Rightarrow$ minimum sensible local volume L_{min}

$$t_{comm} \leq t_{cache} \iff \frac{192L^3}{B_N} \leq \frac{216L^4}{B_C}$$
$$\Rightarrow L_{min} \sim \frac{B_C}{B_N}$$

- D_W scalability determined by ratio of network bandwidth to cache & memory bandwidth [2]
- Maximum performance on a given total problem size N then determined by L_{min}. e.g.

$$\text{Performance} \sim \frac{1320 \times N^4}{t_{comm}} = \frac{1320 \times N^4 B_N^4}{192 \times B_C^3}$$

- Maximum performance and scalability fall as *fourth power* of network bandwidth.

[2] or floating point processing rate – whichever is rate limiter – usually bandwidth

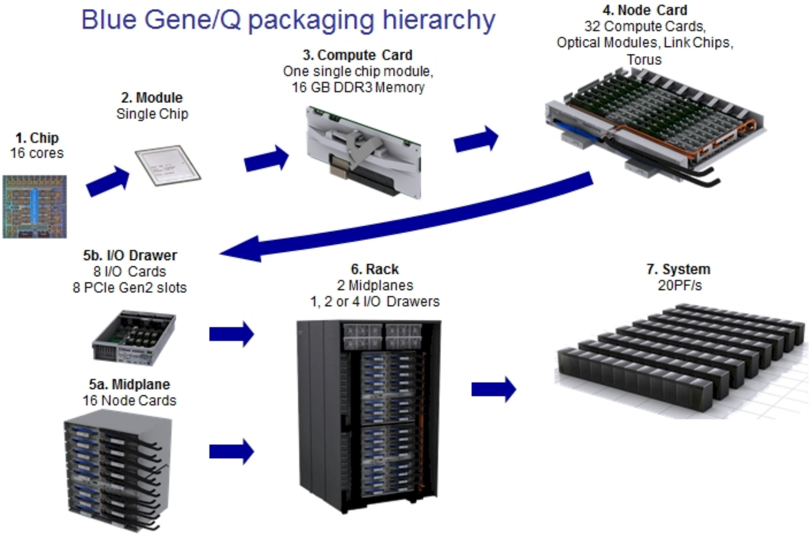
An example

- Consider 64^4 on an nominal 100Gflop/s and 100W node

$\frac{B_C}{B_N}$	Number of nodes	Electrical power	Max System Performance	
32	16	1.6kW	1.6Tflop/s	
16	256	25.6kW	25.6 Tflop/s	
8	4096	410kW	409.6 Tflop/s	←Edinburgh
4	64k	6.4MW	6 Pflop/s	
2	1M	100MW	96 Pflop/s	← DOE

- Conclusion: integrate network controller in the memory system so that $B_N \sim O(B_C)$

Machines: BlueGene/Q



BlueGene/Q overview

- 45nm, 360mm^2, 1.6GHz, 55W
- 16 $\times$ PowerPC 64 bit compute cores (+1 O/S +1 yield)
- In order core
- 16 KB L1 data cache, 4KB L1p prefetch engine, 32 MB L2 cache
- 16GB DDR3 1333 memory (dual controller : 2 $\times$ 128 bit I/F)
- 4 threads per core, 64 threads per chip
- Quad double precision short vector (SIMD) fpu
  QCD is limited to 78% of peak
- FP/Memory/Network bandwidths

GFlop/s	L1 GB/s	L2 GB/s	DDR GB/s	Torus GB/s
204.8	820GB/s	563(448)	42.7	40

- SoC integrates huge cache, huge MPI bandwidth ($\equiv$ O(10) Mellanox cards) within modest area and power budget
  $\Rightarrow$ scalable *and* power efficient

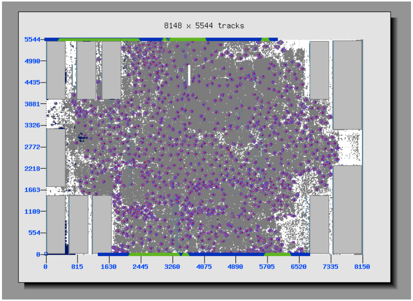
Edinburgh/Columbia/IBM Collaboration

Dec 2007 IBM Research, Edinburgh U., Columbia U. formed a collaboration agreement to jointly develop next generation of BlueGene.

2007-2011 PAB (UoE), Christ (CU), and Changhoan Kim (CU, now IBM) designed adaptive memory prefetch engine (L1P) as contractors.
VHDL logic design, clock tree, test structures, timing closure and placement

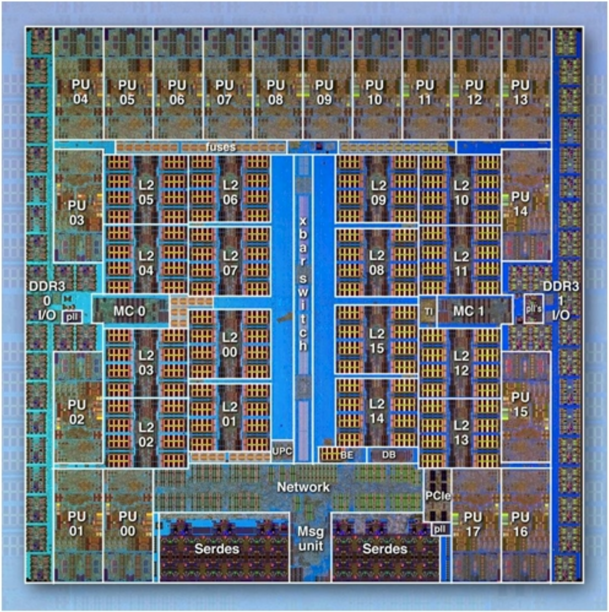
- *QCD assembler and hardware prefetcher jointly developed*
  → The *design* element of codesign is truly important

Can you find L1p in the next slide's die photo?



Hint: SRAMS are the rectangular blocks - match the SRAM pattern

BlueGene/Q die photo



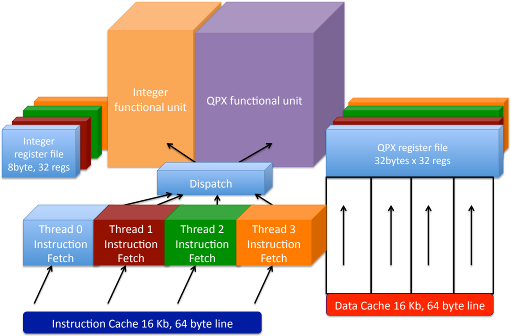
Efficient (and power efficient) computing in particle physics

Peter Boyle, University of Edinburgh

- The Lattice QCD challenge
- **Optimising for BlueGene/Q**
- BG/Q performance
- Optimising for x86 multi-core (Archer...)
- Future: Optimising for Knights series

BlueGene/Q processor core

- Most processors in the world spend 95% of their time idle stalled on memory
- If fetch *independent* instructions from another thread they can be executed
- Replicate instruction fetch, register files. Share the big functional units



- QPX loads, stores and operates on four consecutive double prec. words in parallel

SIMD optimisation

QPX supports paired complex SIMD operations (quad double)

- Develop BAGEL domain specific compiler for BG/Q QPX support
  Human performs register allocation
  Compiler automates pipeline scheduling
- Remember why SIMD was *easy* on the Connection Machine!
  - Subdivide node volume into smaller *virtual nodes*
  - Spread virtual nodes across SIMD lanes (these were memory banks in CM5)
  - Modifies data layout to align data parallel operations to SIMD hardware
- Data parallel operation on both virtual nodes is now simple
  - Crossing between SIMD lanes restricted to during cshifts between virtual nodes
  - Code to treat 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

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

SIMD made easy

- Sequence of operations remains the same as on BG/Q after BAGEL layout transformation
- O(100%) SIMD efficiency

Optimised sequence of operations is *identical* for scalar complex and SIMD operation
BG/L(left, scalar complex) and BG/Q(right vector complex) assembler comparison

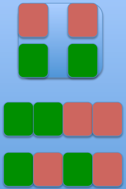
```
bt gt,  __lab3
addi.  %r9 , %r13 , 0
__lab3:
fxcxnpma  0 , 30 , 29 , 26
dcbt  %r18,%r9
fxcxnpma  1 , 30 , 22 , 24
stfpdx  9,%r21,%r17
fxcxnpma  2 , 30 , 7 , 23
stfpdx  10,%r22,%r17
fxcxnpma  3 , 30 , 28 , 27
dcbt  %r20,%r9
fxcxnpma  4 , 30 , 21 , 25
stfpdx  11,%r23,%r17
fxcxnpma  5 , 30 , 6 , 31
la  %r16, -1(%r16)
fxpmul  7 , 15 , 0
dcbt  %r22,%r9
fxpmul  6 , 12 , 0
```

```
        bt gt,  __lab3
         addi  %r9 , %r13 , 0
__lab3:
         qvfxxnpmadd  0 , 29 , 30 , 26
         dcbt    %r18,%r9
         qvfxxnpmadd  1 , 22 , 30 , 24
         qvstfdx  9,%r21,%r17
         qvfxxnpmadd  2 , 7 , 30 , 23
         qvstfdx  10,%r22,%r17
         qvfxxnpmadd  3 , 28 , 30 , 27
         dcbt    %r20,%r9
         qvfxxnpmadd  4 , 21 , 30 , 25
         qvstfdx  11,%r23,%r17
         qvfxxnpmadd  5 , 6 , 30 , 31
         la  %r16,      -1(%r16)
         qvfxmul  7 , 15 , 0
         dcbt    %r22,%r9
         qvfxmul  6 , 12 , 0
```

Path to wider SIMD?

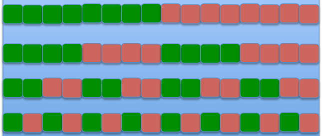
Generalises to wider SIMD

- 2x2 (SSE single, Altivec)



Permute/insert/extract stencils simple

- 2x2x2x2 (MIC)



- Permutation/insert/extract masks required for 16 way SIMD & 2x2x2x2

- 50% efficiency face operations till 16 way SIMD
- 25% efficiency for up to 4^4 = 256 way SIMD

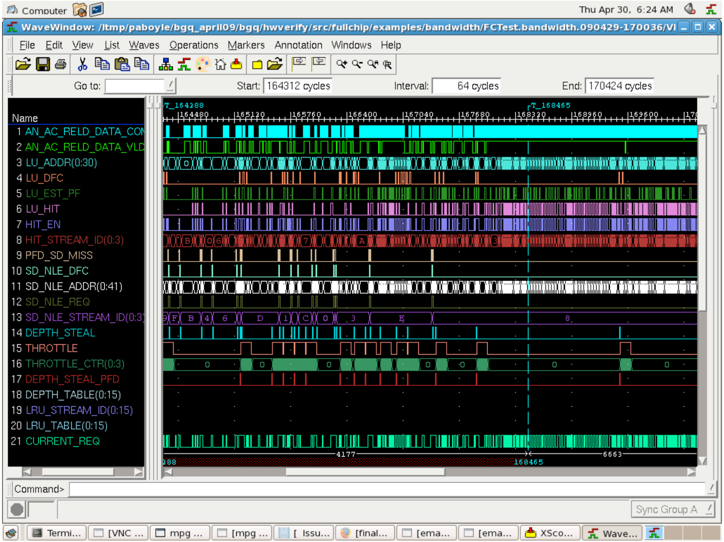
- Same transformation required to both exploit SIMD, gain read coalescence in GPU's.
- Language support for layout transformation is way forward
  World needs to resurrect CMfortran layout statements & conformable array operations
  - target threads & SIMD lanes instead of processing elements and memory banks
- Conformable array operations automatically map to independent threads and independent SIMD ops.

Efficient (and power efficient) computing in particle physics

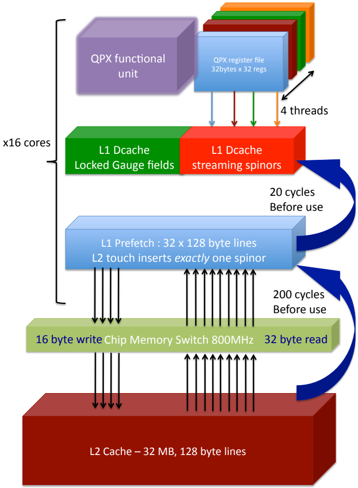
Peter Boyle, University of Edinburgh

- The Lattice QCD challenge
- Optimising for BlueGene/Q
- **BG/Q performance**
- Optimising for x86 multi-core (Archer...)
- Future: Optimising for Knights series

Adaptive prefetch

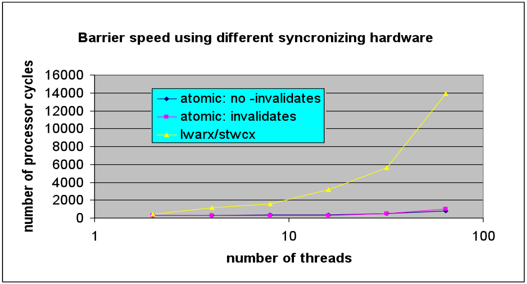


Dataflow



QPX functional unit
QPX register file
32bytes x 32 regs
4 threads
x16 cores
L1 Dcache
Locked Gauge fields
L1 Dcache
streaming spinors
20 cycles
Before use
L1 Prefetch : 32 x 128 byte lines
L2 touch inserts exactly one spinor
200 cycles
Before use
16 byte write
Chip Memory Switch 800MHz
32 byte read
L2 Cache – 32 MB, 128 byte lines

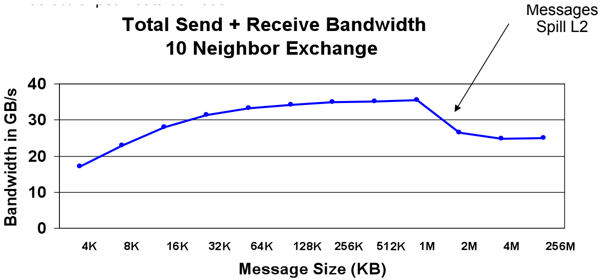
Hybrid OpenMP/MPI code



Bagel uses 64 threads and one MPI process per node

- Long lived threads – duration of solver
- Barrier synchronisation
  - minimises fork/join overhead
  - External packet size is maximised giving best MPI bandwidth
  - Internal copying for MPI within node is eliminated
- Use L2 atomic operations to obtain best performance

Network performance

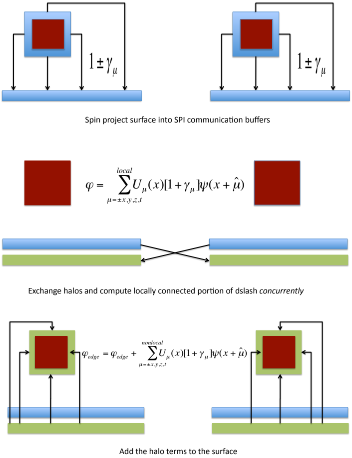


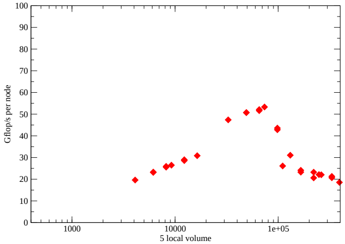
90% link saturation; delivered network bandwidth *exceeds* DDR memory bandwidth
⇒ designed for scaling

600ns latency available through SPI

$\not{D}$ implementation

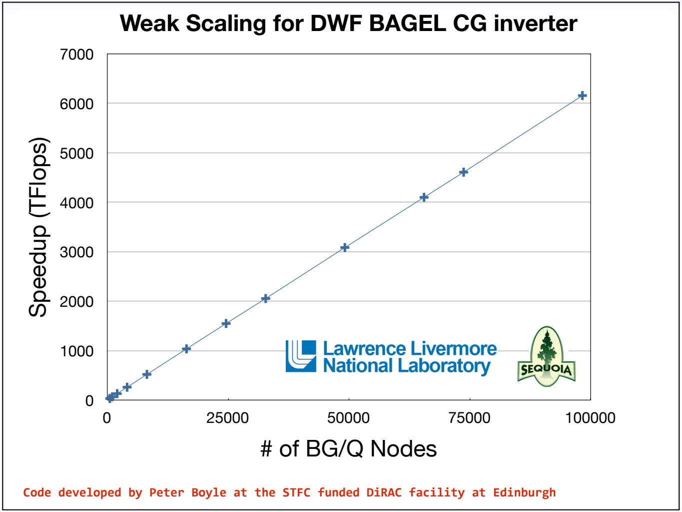
- Single-node, double precision get 110Gflop/s (65% pipeline usage) within L2 cache
- Multi-node cache optimal loop order forces two pass approach to overlap comms & compute (interior/exterior)





Multi-node double precision DWF dslash performance

Bagel DWF CG performance on Sequoia (48 racks, 50% machine)



Weak Scaling on $8^4 \times 16$ local volume

Thanks to Michael Buchoff, Pavlos Vranas, Joseph Wasem, Christopher Schroeder, Thomas Luu and Ron Soltz at Lawrence Livermore National Laboratory.

Sustained 7.2 Pflop/s on 1.6 Million cores (Gordon Bell finalist 2013)

Efficient (and power efficient) computing in particle physics

Peter Boyle, University of Edinburgh

- The Lattice QCD challenge
- Optimising for BlueGene/Q
- BG/Q performance
- **Optimising for x86 multi-core (Archer...)**
- Future: Optimising for Knights series

Optimising for Archer (x86 multicore)

Ivy-Bridge is aggressively out of order and compilers are improving.

- Good news! – I do NOT recommend
  - Compiler development work
  - Optimising to the transistor level
- Bad news! – I DO recommend targetting AVX.
  - For heaven's sake do this in a *general* way!!

Performance portability: targetting variable width SIMD

- *I credit useful discussions with Codeplay, Edinburgh compiler company*
- Develop a general short vector class of variable (compile time determined) width.
- Transform legacy code from array-of-structs (AoS) $\rightarrow$ Struct-of-array (SoA).
  - Strictly Array-of-structs-of-short-vectors [AoSoSV]
- Parameterise this transformation [layout opaque containers etc...].

Define performant classes *vfloat*, *vdouble*, *vfcomplex*, *vzcomplex*.

```
#if defined (AVX1) || defined (AVX2)
    typedef __m256 dvec;
#endif
#if defined (SSE2)
    typedef __m128 dvec;
#endif
#if defined (AVX512)
    typedef __m512 dvec;
#endif
#if defined (QPX)
    typedef vector4double dvec;
#endif
    class vdouble  {
         dvec v;
      // Define arithmetic operators
        friend inline vdouble operator + (vdouble a, vdouble b);
        friend inline vdouble operator - (vdouble a, vdouble b);
        friend inline vdouble operator * (vdouble a, vdouble b);
        friend inline vdouble operator / (vdouble a, vdouble b);
        static int Nsimd(void) { return sizeof(dvec)/sizeof(double);}
    }
```

Code examples & performance analysis

Define performant classes *vfloat*, *vdouble*, *vfcomplex*, *vzcomplex*.

```
   friend inline vdouble operator + (vdouble a, vdouble b)  {
     vdouble ret;
#if defined (AVX1)|| defined (AVX2)
          ret.v = _mm256_add_pd(a.v,b.v);
#endif
...
     return ret;
    };

    friend inline vdouble operator * (vdouble a, vdouble b) {
      vdouble ret;
#if defined (AVX1)|| defined (AVX2)
      ret.v = _mm256_mul_pd(a.v,b.v);
#endif
...
      return ret;
    };

   friend inline void fmac (vdouble &y,vdouble a, vdouble x){
#if defined (AVX1) || defined (SSE2)
     y = a*x+y;
#endif
#ifdef AVX2     // AVX 2 introduced FMA support. FMA4 eliminates a copy, but AVX only has FMA3
     // accelerates multiply accumulate, but not general multiply add
     y.v = _mm256_fmadd_pd(a.v,x.v,y.v);
#endif
   }
```

Code examples & performance analysis

Apply Nsimd() small dense matrix multiplies in parallel:

```
// L1 resident
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]);
        }
    }
}

// Memory resident
template<int N,class simd>
void matmul_vec(int nmat, simd * __restrict__ x,simd * __restrict__ y, simd *__restrict__ z)
{
    for(int m=0;m<nmat;m++){
        for(int i=0;i<N;i++){
            for(int j=0;j<N;j++){
                x[i]= x[i]+y[i*N+j]*z[j];
            }
        }
        y+= N*N;
        x+= N;
        z+= N;
    }
}
```

- Template parameter matrix size; known at compile time
- Generates *very* efficient AVX/AVX2 code with clang

Code examples & performance analysis

```
Ltmp4:
        .cfi_def_cfa_register %rbp
        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
        .align  4, 0x90
LBB0_1:                                 ## %.preheader
                                        ## =>This Inner Loop Header: Depth=1
        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
        jne     LBB0_1
```

- Template parameter matrix size ¡8¿ ; known at compile time
- Generates *very* efficient AVX/AVX2 code with clang
- retains column vec x in registers ymm0-7; dependent chain accumulated in

Performance analysis

Test system

- FP pipeline
  - dual issue 8 wide single precision 2.3GHz.
  - Peak 16x2.3 = 36.8 Gflop/s per core single
  - Peak 8x2.3 = 18.4 Gflop/s per core double
- Memory system
  - Streams bandwidth benchmark reports 13GB/s.
  - Peak memory bandwidth 25.6GB/s.
- L1 resident results (should saturate FP pipe)
  - matmul with N=12 : 32Gflop/s
- DRAM resident results (78MB footprint - should saturate memory bus)
  - 32 bit arithmetic: 6.9 Gflop/s $\leftrightarrow$ 16.2 Gbyte/s
  - 64 bit arithmetic: 3.0 Gflop/s $\leftrightarrow$ 14.0 Gbyte/s

Performance analysis

Conclusion:

- Correct use of AVX *through clang compiler*
  - saturates FP pipe from L1
  - and exceeds streams bandwidth from DRAM
- Dependent chains of register use by consecutive instructions *relies* on OoO execution
- Key Question: Will this be sufficient for Knights Corner/Knights landing???
  - Since XeonPhi is *in order*, I am pursuing *both* evolution of BAGEL approach and this compiler approach
  - Intel expanded Register File to 32 entries and four threads $\Rightarrow$ 8KB
  - XeonPhi is in-order execution for power reasons. Likely must programme around pipeline

Efficient (and power efficient) computing in particle physics

Peter Boyle, University of Edinburgh

- The Lattice QCD challenge
- Optimising for BlueGene/Q
- BG/Q performance
- Optimising for x86 multi-core (Archer...)
- **Future: Optimising for Knights series**

Future plans

- How to get 10x improvement in machine power within fixed, affordable recurrent budget ?

Power envelope

- Requires 10x improvement in power performance from current 2.1 Gflop/s per watt.
  - *accompanied* with adequately performant memory and network subsystems
  - Rules out conventional x86 processors
- Knights landing and Nvidia Volta have 2015 projections in 16/32/64 Gflop/s per Watt range (dp/sp/hp).

Memory bandwidth

- Require 1TB/s cache bandwidth per Tflop/s
- HMC 3D chipstack memory is disruptive technology, 1/10th Energy/bit and 10x bandwidth.
- NVIDIA and Intel plan to use micron HMC's in their Volta and KNL products
- I have ported my compiler to KNL

Network bandwidth

- No breakthrough's for several years
  20-30 GB/s all we can hope for from either EDR PCIe-3.0 IB cards, or proprietary interconnects

Performance modelling

Architecture	Cache read BW/size	Memory BW/size	Network BW	$L_{min} \sim \frac{B_C}{B_N}$
BG/Q	410GB/s , 32MB	43GB/s, 16GB	40GB/s (30)	10 (8)
K-computer	??/6MB	64GB/s, 16GB	100GB/s 64GB/s	4
Cray XK6 (twin GPU)	??	354GB/s , 12GB	20 GB/s	18
GPU+infiniband 1:1	??	150GB/s , 6GB	5GB/s	30
GPU+infiniband 4:1	??	600GB/s , 24GB	5GB/s	120

- GPUs + IB (1:1) will allow modest scaling on big volumes
- GPUs + IB (4:1) will not scale beyond one node on any reasonable lattice

Broadly two models emerging:

- Coherent many-core nodes: MPI $\otimes$ OpenMP $\otimes$ SIMD
- Accelerator nodes: MPI $\otimes$ CUDA/OpenCL/OpenAcc/OpenMP 4.0

Conjugate gradient optimisation

2014: Developed new adaptive aggregate multigrid deflation algorithm (HDCG) arXiv:1402.2585

- 14x runtime algorithmic acceleration
- 30x saving in matrix multiplies.