UKQCD software map

CPS

Configurations
- Sharing Columbia code makes sense
  - Last thing we need is inconsistent RHMC codes!
- Significant UKQCD contributions
  - Bagel for DWF solver (Boyle)
  - RHMC & integrator framework (Mike Clark)
  - MPI port
- Programmer productivity for physics not as good as with QDP++ (PAB’s opinion)

UKvalence

Measurement
- UKhadron
  - (~scidac application)
    - High productivity using QDP++
    - Code is pretty inefficient
    - BAGEL used for matrix inversions
    - A few routines from CHROMA are also used
    - Avoids chroma XML challenge
- Ukfit
  - (~scidac level 4, predates)
    - Uses QDP++ in fit code!
    - Not worth fighting QDP++ I/O
    - JLAB use “ADAT” to translate!
    - Gives all the power of QDP++ data types, gammas to the fitter
      - Esp. for NPR

Analysis
- QDP++ I/O formats

BFM

UKfit

(Ukfit) (~scidac level 4, predates)

~Scidac level 3 + domain specific compiler

BAGEL

QDP++ (scidac level 2)
HPC basics

- Floating point small fraction of BG/P die!

Joint physical challenges:
- Motion of data
- Floating point density

Related challenges
- Cost
- Power consumption

Bandwidth - byte per second
Latency – seconds to first data

Increased bandwidth requires money
Latency reduction only requires speed of light increase!

In a parallel machine data is stored in the memories of multiple communicating processor chips
HPC basics (cont)

Best programming style depends on these parameters
All really techniques to mask memory and remote latency

• Caches hold recently used data hoping it will be reused data
• Prefetchers fetch ahead hoping it will be needed

Programmers should where possible reuse data maximally and make memory access patterns linear

Systems characterised by

Floating point performance
• Peak
• SIMD width

Memory performance
• L1, L2 cache, main memory
• Bandwidth
•Latency

Communication performance
• Topology
• Bandwidth
• Latency
• Concurrency
• Typically use MPI programming
HPC basics (cont)

Load/Store/Operate architecture

- The translation is of course normally done by a compiler

Processors are superscalar and pipelined. Typical code stalls a lot.

- Must ensure data is in L1 cache prior load
- Load early prior to use
- Ensure sequences of O(12) instructions are all independent

Norm of a vector

slow_norm:
  mtctr %r4
loop:
  lfd %f4,0(%r3)
  fmadd %f3,%f4,%f4,%f3
  addi %r3,%r3,8
  bdnz loop
  blr

Faster_norm:
  mtctr %r4
  loop:
    lfd %f4,0(%r3)
    lfd %f5,8(%r3)
    lfd %f6,16(%r3)
    lfd %f7,24(%r3)
    fmadd %f8,%f4,%f4,%f8
    fmadd %f9,%f5,%f5,%f9
    fmadd %f10,%f6,%f6,%f10
    fmadd %f11,%f7,%f7,%f11
    addi %r3,%r3,32
    bdnz loop
    fadd %f3,%f8,,%f9
    fadd %f3,%f10,%f3
    fadd %f3,%f11,,%f3
    blr

Fastest_norm:
  mtctr %r4
  loop:
    lfd %f4,0(%r3)
    lfd %f5,8(%r3)
    lfd %f6,16(%r3)
    lfd %f7,24(%r3)
    fmadd %f8,%f4,%f4,%f8
    fmadd %f9,%f5,%f5,%f9
    fmadd %f10,%f6,%f6,%f10
    fmadd %f11,%f7,%f7,%f11
    addi %r3,%r3,32
    addi %r4,%r4,96
dcbt %r0,%r4
    bdnz loop
Assembler is a pain

When a new machine comes my hard work is wasted => BAGEL

Abstracted assembler interface
Programmer declares memory pattern
Programmer decides register allocation

Processor model translates and reschedules code for each architecture and chip version

Current abstraction is complex simd + appropriate arithmetic operations

```
here = start_loop(counter)
for(int unroll=0; unroll<4; unroll++) {
    complex_load(U[i], UIMM[i], ptr);
    simd_madd(A[i], U[i], U[i].A[i]);
}
iterate_stream(mystream);
stop_loop(here, counter)
```
BAGEL performance

BAGEL-2 submitted to CPC last month

• Stream (qcdoc) & cache (BG/P) memory strategies
• Entire DWF solver in BAGEL now
• QDP++ & C++ interfaces (CHROMA and CPS)
• Extensive regression suite to CHROMA/QDP++
• Britney test used by IBM to shake Argonne BG/P
• Mixed precision single/single/double solves
• In use in UKhadron on QCDOC
• Third (write through L1) strategy implemented
• SSE support planned

New strategy for cached systems
<table>
<thead>
<tr>
<th>bfm/tests subdirectory</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>linalg</td>
<td>Linear combination, dot product etc...</td>
</tr>
<tr>
<td>wilson</td>
<td>Wilson fermion checkerboarded matrix application</td>
</tr>
<tr>
<td>dwf.mooee</td>
<td>DWF rb4d checkerboarded building block</td>
</tr>
<tr>
<td>dwf.mooeeinv</td>
<td>DWF rb4d checkerboarded building block</td>
</tr>
<tr>
<td>dwf.prec</td>
<td>DWF rb4d checkerboarded matrix</td>
</tr>
<tr>
<td>dwf.cg</td>
<td>DWF rb4d checkerboarded CG for 5d system</td>
</tr>
<tr>
<td>dwf.rb5d.dperp</td>
<td>DWF rb5d checkerboarded 5d hopping term</td>
</tr>
<tr>
<td>dwf.rb5d</td>
<td>DWF rb5d checkerboarded matrix</td>
</tr>
<tr>
<td>dwf.rb5d.cg</td>
<td>DWF rb5d checkerboarded CG for 5d system</td>
</tr>
<tr>
<td>dwf.rb5d.prop</td>
<td>12 component 4d propagator solution with $4d \leftrightarrow 5d$ projection.</td>
</tr>
<tr>
<td>majority_vote</td>
<td>Hardware test and diagnostic</td>
</tr>
</tbody>
</table>
L2 strategies

Apply Wilson operator to N vectors for both DWF (RBC-UKQCD-LHPC) & Overlap (KEK) actions.

- Two schemes: both have N-fold reuse of gauge field & L2 friendly “gather to a point” implementations

<table>
<thead>
<tr>
<th>Gauge link reuse in registers; results bounce off L1</th>
<th>Gauge link reuse in L1; result spinor in regs</th>
</tr>
</thead>
<tbody>
<tr>
<td>• For each direction load gauge link to registers</td>
<td>• Accumulate 4-spinor result in registers</td>
</tr>
<tr>
<td>– Spin-proj &amp; SU3 mult Ls 4-spinors</td>
<td>– Load neighbor 4-spinor &amp; spin-proj</td>
</tr>
<tr>
<td>– Store Ls 2 spinors to L1</td>
<td>– Load gauge link &amp; SU3 multiply</td>
</tr>
<tr>
<td>• Reload 2-spinors from L1 and sum</td>
<td>– Accumulate &amp; then store result</td>
</tr>
</tbody>
</table>

Pros:
- Multi-KB access patterns for prefetch
- Minimises number of loads&stores

Cons:
- High write traffic when write through L1
- (optimal for write back L1!)

Pros:
- Recognises stores impose 8x more switch requests than loads for same #bytes

Cons:
- ~384 byte sequential accesses
To claim future proof optimisation tempts fate…
Bagel now *combines* SIMD + threads + message passing

**Wider SIMD important:**
- intel SSE: 2,4 way simd
- intel AVX: 8 way SIMD
- intel Larrabee: 16 way SIMD
- BG/L,P: 2 way SIMD
- CELL: 2/4 way SIMD
- Etc..

**Multithreading important**
- Intel, AMD, BG/P etc..
- Threads+MP gives better network packet size & minimises intra-node traffic

**Message passing important**
- If one node is fast, 16384 are faster!
- I use QMP as gives MPI & QCDOC compatibility

**Cross platform:**
- BG/L,P,Q
- Power & PowerPC (pseries, QCDOC)
- Alpha 21064/21164/21264
- (ultra)SPARC – generalise sparc64viii?
- C++ (noarch)
- C++ with SSE intrinsics (experimental)

GPU is missing...
But...
GPU is missing ECC...
Single instruction multiple data

Developed BAGEL infrastructure to support ARBITRARILY wide vector SIMD.

SIMD vector, no gather/scatter reliance, but assume flexible permute
Tested on SSE port (unreleased, student project)

Take clues from connection machine fortran!
Divide single node into “virtual nodes” where virtual nodes are spread across SIMD lanes

Crossing between SIMD lanes restricted to comms between nodes

Code for one virtual node is almost identical to scalar code, except sizeof data type is increased

Neither “struct of arrays” nor “array of structs” but “array of structs of short arrays”

Retains locality benefits of microprocessor ordering
**SIMD**

- SIMD pipes participate in the distribution in the same was as the multiple CPU's.
- Basic datum size is increased (or from a data parallel view could be mult. By NSIMD * NCHIP)

- Assume 2x2x2x2 processor grid & 2 way SIMD ; global T-direction is

  ![Processor Diagram]

- Code identical down to register allocation & scheduling to non-simd code
  - Presence of second pipe is a minor disturbance that simply widens data
- Processor zero contains 8 sites of data (ABCDEFGH) if “simd-latt[t]” is 4
  - Pipe0 → ABCD
  - Pipe1 → EFGH
- Layout in memory as (AE),(BF),(CG),(DH)
- Hopping term shift “left” : remove the (AE) part & left shift
  - (BF)(CG)(DH)(??)
- (AE) treatment involves inter-pipeline operations & possible communication
  - Single node just permute: (BF)(CG)(DH)(EA) == BCDEFGHA
- Muti-node must send A, receive “I” & merge
  - (BF)(CG)(DH)(EI) == BCDEFGHI

- With flexible permute/merge type operations (altive/SSE/CELL) can do this in multiple dimensions and scale to arbitrarily wide SIMD
Generalises to wider SIMD

- 2x2 (SSE single, Altivec)

Permute/insert/extract stencils simple

- 2x2x2x2 (Larrabee)

- 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
Single Chip QCD SIMD – perhaps a future project?

• Were I not working on BG/Q I would probably be designing a wide SIMD architecture
  • Instruction broadcast killed SIMD – not a problem WITHIN a chip
  • Cmfortran proved data parallelism can be exploited.
    Cartesian “Layout” primitives decompose across SIMD direction
  • Use explicit message passing (killed HPF effort)

• Single integer core (ARM?), 256? complex arithmetic FPU’s, modest clock speed
  • Disjoint scalar and SIMD memory systems
  • Scalar <-> SIMD register move
  • Modest cache for SIMD data & prefetch
  • Wide GDDR memory system

• Advantages:
  • Large fraction of die in floating point – would have to recoup process disadvantage
  • Each memory load pulls 4KB of data, perhaps 32KB cacheline size
  • Low power
  • Memory system design much simpler than a multi-threaded chip
Multithreading

System dependencies and models differ wildly:

• Adopted neutral, minimal model for Bagel
  • All threads created outside of bagel & all threads call all routines
  • One thread per “hardware thread”
  • Only system dependency is the “barrier” routing
  • Can implement pthread, openMP or custom hardware barriers

• Long running threads + Barrier synchronisation
  • Minimises fork/join overhead
  • Removes system dependency on how to fork join

• Entire multithreaded process interfaces to communications layer
  • Maximises packet size, reduces latency sensitivity

```cpp
class ThreadModel {
    virtual void thread_init(int nthread);
    virtual int  thread_barrier(void);
    virtual void thread_sum(double & val, int me);
    void        thread_work(int nwork, int &me,
                            int &mywork, int &mystart);
}

int me, mywork, mystart;
int nwork = volume;

thread_work(nwork, me, mywork, mystart); // similar to OMP
for(int site = mystart; site<mystart+mywork; site++){
    do_work();
}
thread_barrier(); // incl MBAR for weak consistency
```
• BAGEL CODE WALK THROUGH!
  – Bagel
    • Processor
    • Instruction
    • Bgl.C
  – BFM
    • Bfm.h
    • Bfmbagel2.C
• Ukhadron CODE WALK THROUGH!
  – Meson trace
  – B_K
  – Wall Kl3
  – NPR

• Ukfit CODE WALK THROUGH
  – Distribution template
    • Matrix inverse specialisation
  – Fit
  – NPR amputation