Forschungszentrum Jülich
in der Helmholtz-Gemeinschaft

BERGISCHE
UNIVERSITÄT
WUPPERTAL

Performance and optimization of LQCD codes on BlueGene and Cell

Outline

("JUBL") The 8 rack Blue Gene/L in Jülich

- Wilson Dirac kernel
- Serial code
  - Using GCC inline assembly
  - Using the IBM XLC compiler
  - Memory Issues
- Comms & Performance
  - MPI
  - IBM QCD API & MPI
- Mixed Precision
- Outlook: LQCD on Blue Gene/P
- Outlook: LQCD on Cell

The Wilson Dirac kernel

$$\begin{aligned} D_{nm}\Psi_m &= (4+m_0)\,\Psi_n \\ &+ U_\mu(n)\,(1-\gamma^\mu)\,\Psi_{n+\hat{\mu}} \\ &+ U_\mu^\dagger(n-\hat{\mu})\,(1+\gamma^\mu)\,\Psi_{n-\hat{\mu}} \end{aligned}$$

- U is a SU(3) “gauge” matrix, that is a 3x3 matrix with complex entries
- $(1\pm\gamma^\mu)$ 4x4 matrix, acting as a “spin” projector
- Ψ is the “spinor” (quarkfield), it contains 3x4 complex numbers (color*spin) per lattice site
- The Kernel connects nearest neighbours only.

The Wilson Dirac kernel: Spin projection

$$(1+\gamma^z)\Psi = \begin{pmatrix} 1 & 0 & i & 0 \\ 0 & 1 & 0 & -i \\ -i & 0 & 1 & 0 \\ 0 & i & 0 & 1 \end{pmatrix} \begin{pmatrix} \Psi_0 \\ \Psi_1 \\ \Psi_2 \\ \Psi_3 \end{pmatrix} = \begin{pmatrix} \Psi_0 + i\Psi_2 \\ \Psi_1 - i\Psi_3 \\ -i(\Psi_0 + i\Psi_2) \\ i(\Psi_2 - i\Psi_3) \end{pmatrix}$$

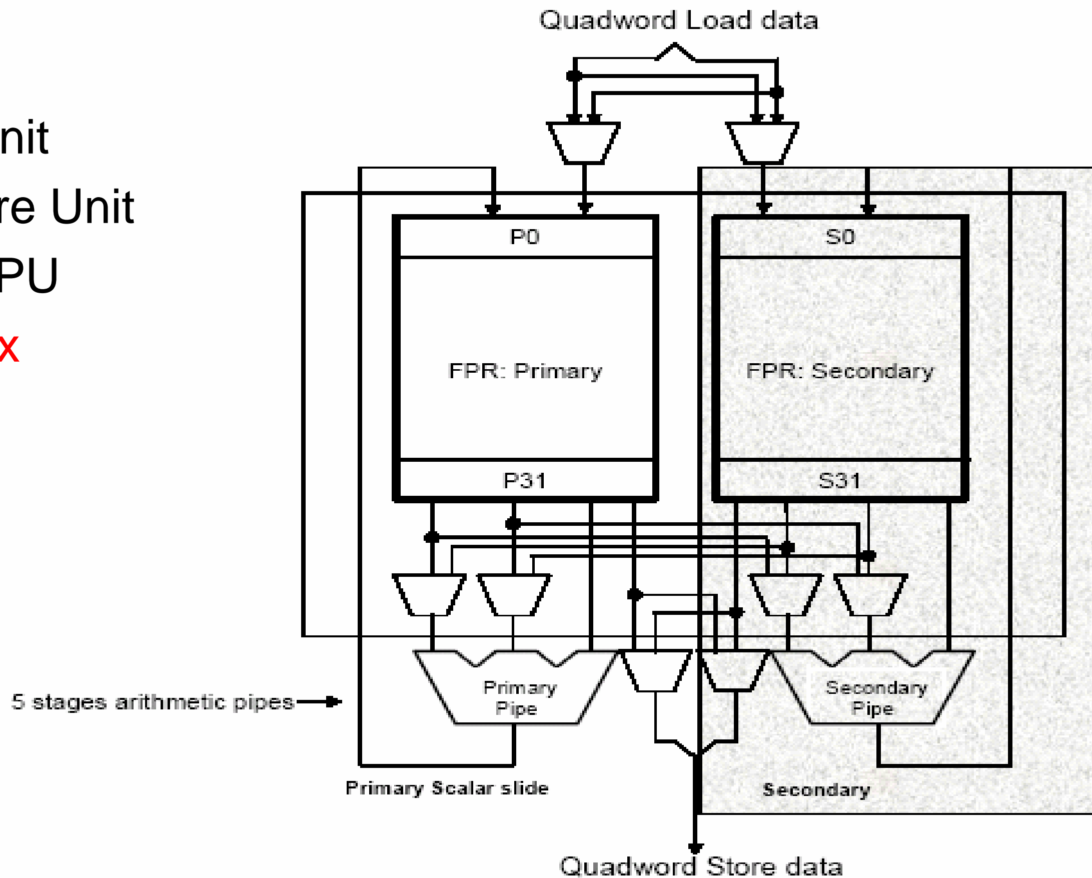
- Spin projection halves the number of independent spin components
- Half the number of multiplications with the SU(3) matrix need to be performed
- Half the data needs to be communicated when spin projection is applied before communication (“2spinor comms”)

Serial code

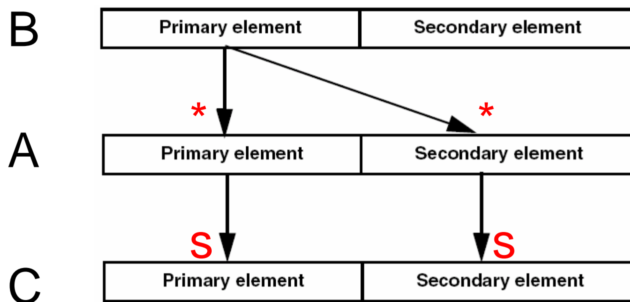
```
QuadLoadU(a, 10, o);
    asm volatile ( "fpmadd    6, 20, 0, 10");
QuadStoreU(g, 6, o);
    asm volatile ( "fpmadd    7, 21, 1, 11");
QuadLoadU(b, 11, o);
    asm volatile ( "fpmadd    8, 22, 2, 12");
QuadStoreU(g, 7, o);
    asm volatile ( "fpmadd    9, 23, 3, 13");
QuadLoadU(c, 12, o);
    asm volatile ( "fxcxnsma  5, 24, 4, 14");
QuadStoreU(g, 8, o);
    asm volatile ( "fxcxnsma 15, 16, 17, 18");
QuadLoadU(d, 13, o);
    asm volatile ( "fxcxnsma 25, 26, 27, 28");
QuadStoreU(g, 9, o);
    asm volatile ( "fxcxnsma 19, 29, 30, 31");
QuadLoadU(e, 14, o);
    asm volatile ( "fpmadd    6, 20, 0, 10");
QuadStoreU(g, 9, o);
    asm volatile ( "fpmadd    7, 21, 1, 11");
QuadLoadU(e, 14, o);
```

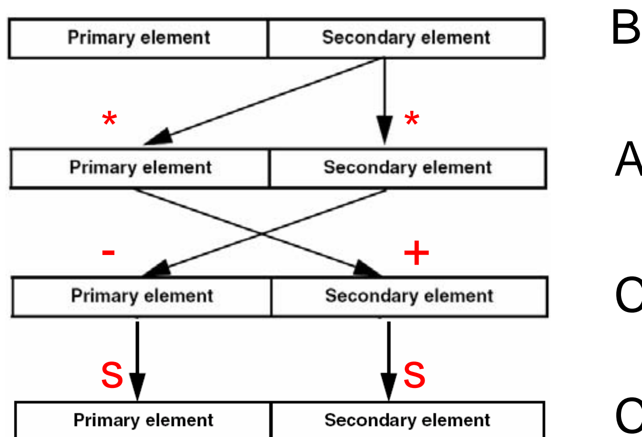
ppc440d “Double Hummer” FPU

- CPU contains
- One Integer Unit
- One Load/Store Unit
- One special FPU
- Secial complex algebra SIMD instructions



ppc440d "Double Hummer" FPU





Complex multiply (A*B=C):

Re(C) =(Re(A)*Re(B) – Im(A)*Im(B))

Im(C) =(Im(A)*Re(B) + Re(A)*Im(B))

Required:

- a cross copy primary multiplication (2+1 register operands)
- a cross mixed negative secondary multiply – subtraction (3+1 register operands)

Intermezzo: Ugly code on Intel & Co

movapd xmm0, <mem_X>	Load complex number X into xmm0
movapd xmm1, <mem_Y>	Load complex number Y into xmm1
movapd xmm2, <mem_Y>	Load complex number Y into xmm1
unpcklpd xmm1, xmm1	{Re Y, Im Y} → {Re Y, Re Y}
unpckhpd xmm2, xmm2	{Re Y, Im Y} → {Im Y, Im Y}
mulpd xmm1, xmm0	{Re X * Re Y, Im X * Re Y}
mulpd xmm2, xmm0	{Re X * Im Y, Im X * Im Y}
xorpd xmm2, xmm7	{Re X * Im Y, (-1) * Im X * Im Y}
shufpd xmm2, xmm2, 0x1	{(-1) * Im X * Im Y, Re X * Im Y}
addpd xmm2, xmm1	{…done…}
movapd <mem_Z>, xmm2	Store result to memory

ppc440d “Double Hummer” FPU

- Complex number multiply requires 2 instructions
  (1 cycle each, 5 cycle pipeline latency)

  fxpmul A, B, C
  …
  fxcxnpma E, B, C, A

- SU(3) matrix vector multiply requires 2*9 instructions

  … fxpmul … fxcxnpma … (fxcpmadd … fxcxnpma …)

  Matrix multiply has theoretical performance of 91.7% peak.

Using the GCC inline assembly

- A good way to generate Double Hummer code with good performance
- The programmer has to take care of everything:
  selecting the instruction, selecting the registers, scheduling the instructions
- The CPU does not necessarily obey the order of instructions
  → scheduling loads and prefetches can be more trail and error than …
- However this way one can use the QCD APIs provided by IBM
  (→ Pavlos Vranas IBM Watson, see Talk Jun Doi IBM Japan)

Using the IBM XLC compiler

- The XLC attempts to simdize code if –qarch=440d is used
- However: Simdization of generic C-code is complicated due to alignment issues
- The Double Hummer FPU can only operate on quad-aligned data (address on 16 byte boundary)
- Inside a function the alignment of a pointer may be unknown
- XLC will try to use primary FPU for padding
- Resulting performance is often poor and below –qarch=440, that is when only one of the FPUs is used
- Possible gains by helping the compiler understand the code:
  → use #pragma alignx , #pragma unroll to help avoiding pipeline stalls

Using the IBM XLC compiler

- Alternative: Write code with built-in floating point instructions ("intrinsics"), compile with –qarch=440d flag
- Intrinsics are peudo-assembly instructions
- User has to select the Double Hammer assembly instruction (see IBM Redbook BGL Application Development, chapter IV)
- Compiler will do scheduling of instructions
- #pragma's are still required!
- Performance of code written using the intrinsics is usually much higher than performance of c
- The ppc440 cache manipulation instructions like DCBT are also available and their usage improves performance significantly

Memory issues

- No Blocking used in case of the BGL kernel
- However allocating a chunk of memory with store without allocate is useful for temporary spinors.
- SWOA: When creating the temporaries avoid spoiling L1 cache contents by writing to L3 right away

Useful RTS calls, V1R3 RTS version

(see the updated version of the IBM Redbook “Application Development”)

- New RTS release offers new system calls. Two of these are particularly interesting:
- **rts_get_dram_window(…)**: Allows to select the caching policy for a chunk of memory. However:
  - a total of 14 MB is available
  - Blocks can only be allocated in 1MB chunks
  - The application has to be linked differently (see Redbook for Details)

Environment Variables

There are few more environment variables that influence the performance of the scalar part of the code

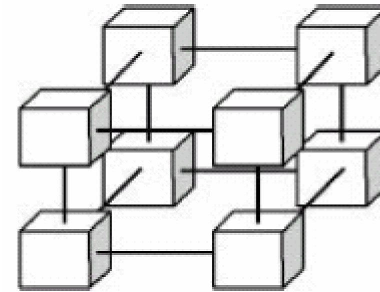
- **BGL_APP_L1_SWOA**=0/1
  - Caching strategy. Switch on/off "store without allocate" for all nodes.
- **BGL_APP_L1_WRITETHROUGH**=0/1
  - Performance on/off

Optimizing the communications

Blue Gene/L network overview

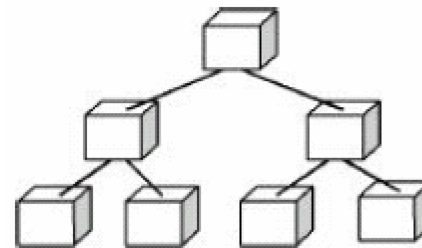
Blue Gene/L has 5 independent networks:

- Torus Network:
  12 links at 1.4 Gb/s
- Tree Network at 2.4 Gb/s



And:

- Tree Network for barriers and interrupts
- Gigabit Ethernet for file i/o
- Control Network



Torus network

- A Blue Gene/L node has 2*(3+1)+2*7 torus fifos
- (send 0/1: A, B, C, P; recv 0/1: ±X, ±Y, ±Z, P)
- Fifos are memory mapped: Writing to a fifo is storing to a *fixed* address
- All fifos are serviced simultaneously by the hardware
- Status registers can be polled to indicate send/receive status:
  - Check for enough space in a send fifo
  - Check for arrived data in a recv fifo
- Packet destination is defined in a 16Byte header
- All nodes in a partition can be addressed directly
- (RTS an) hardware routes packages

Communication strategy on BGL

1. Do spin projection and gauge multiplication

2. Communicate buffers in all directions simultaneously (persistent)

3. Reconstruct and do remaining gauge multiplications

Communication with MPI

- The MPI performance is very good, however this is more true for bandwidth than for latency (surprise…)
- However they can obviously only be used to communicate buffers
- LQCD kernel: For small volumes the Kernel using APIs may prove to be as much as twice as fast as the MPI version
- For typical lattice sizes the difference is smaller.
- MPI Wilson Matrix multiplication peaks at >18% double, >24% single (using persistent sends)
- Remarks: Make sure to use the right communicator (compare bgl_get_personality with MPI Cartesian coordinates)
  → Performance penalty may be severe.

Communication with MPI

There are a few of useful environment variables that can improve MPI performance:

- **BGLMPI_MAPPING=XYZT**
  - Influence the placing of the MPI processes to the nodes.
- **BGLMPI_PACING**=Y/N
  - Use/Don't use packet pacing (rendezvous protocol). Rendezvous protocol implies a node only sending when requested to do so by destination node.
- **BGLMPI_EAGER**=1000(default)
  - Set threshold message size from where MPI will switch from eager to rendezvous protocol.

Standard but important RTS calls

rts_get_timebase():

- Get the value of the clock counter register. Usefull for timings.

rts_get_personality(…):

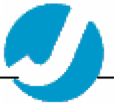
- Get node coordinates in partition (machine/physical coordinates)
- Get partition size (X-Y-Z)
- etc …

rts_get_processor_id(…):

- Get the id of the core on the node

Communication with IBM QCD APIs

- Strategies for communication:
- Communicate buffers:
  - Collect data in buffer, communicate buffers while making sure to feed all torus injection fifos simultaneously
  - Use time while packets on the fly to communicate local directions
- Communicate directly from registers after spin projection (and color multiplication)
  - Each fifo has 1KB buffer space
  - Communication can be better hidden behind calculations
- In each case: To maximize bandwidth have one CPU communicating in +, the other in – direction, then swap their roles.



Communication with IBM QCD APIs

- The IBM QCD APIs are available in Juelich since the update to V1R3
- They allow using the full capabilities of the network:
  (see Talk of J. Doi, Boston conference:)
  - API function to prepare packet header
  - API macros to send/recv data from/to FPU register
    - Communication between node (XYZ) and between CPU (T) can be handled in same way
    - These macros are used with inline assembly to optimize instruction pipeline with computations
  - API function for internal barrier between 2 CPUs
  - API functions to send / recv through user buffer
    - These functions are used if we do not want to use inline assembly

Useful RTS calls, V1R2 & V1R3 RTS version

- **rts_get_scratchpad_window(…)**:
  - Can be used for comms (Possible conflicts with MPI!)
  - Get a window to a cache inhibited area in local memory

- **rts_malloc_sram(…)**: Get a chunk of the shared SRAM. However:
  - Only a total of 8KB is available and MPI uses at least half of it
  - Can be allocated in chunks of 32Bytes only.

- **rts_get_virtual_process_window (…)**: See the other cores memory in virtual node mode

  →This might be a rather easy way to avoid the local comms…
  if one keeps in mind that the 1st level caches are not coherent

IBM QCD API performance (double)

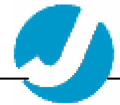
%of peak	2^4	$4x\ 2^3$	4^4	8×4^3	$8^2 \times 4^2$	16×4^3
w/o comms	31.5	28.2	25.9	27.1	27.1	27.8
production	12.6 (16.2)	15.4	15.6	19.5	19.7	20.3
Inverter	13.1	15.3	15.4	18.7	18.8	19.0

Numbers by Pavlos Vranas, IBM Watson

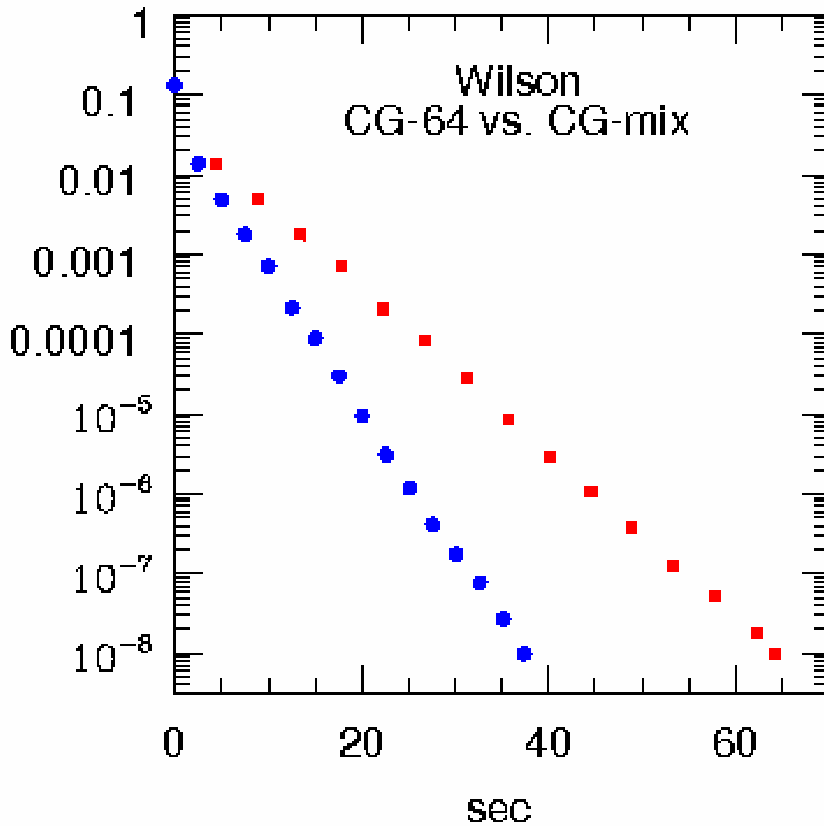
Single precision peaks at 25.5%

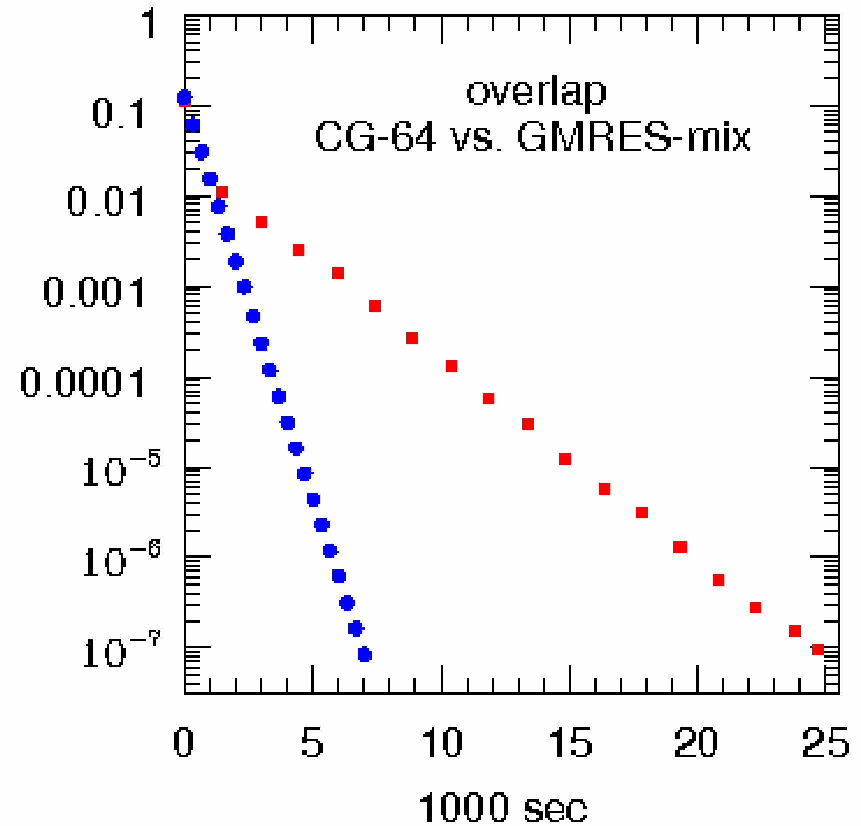
Double precision results, single precision speed

- The Simulation performance is dominated by an inverter (solver)
- The solver performance is dominated by the kernel performance
- Single precision kernel is faster than double precision, and *has a smaller cache footprint*
- Use adaptive precision techniques to utilize this advantage
- Our own method allows this: relGMRESR(SUMR) with single precision SUMR; but CG works as well
- Minimal losses: Total number of matrix multiplication increases by 10%, single precision kernel can gain more than 100%
- Of course: Result (solution vector) is double precision precise.



Double precision results, single precision speed





Outlook: QCD on Blue Gene/P

- Blue Gene/P features
  - a 4 way SMP node (with double FPUs)
  - higher clock frequency
  - Increased torus bandwidth to keep up with the increased compute performance
  - A fullblown DMA for communication (comms can be offloaded)
- Roughly speaking: Everything gets scaled by a factor of 2.4
- Kernel ported to BGP (with DMA comms) already reaches higher performance than BGL (measured in percent of peak)
- BGP QCD kernel reaches **4.3** (**3.3**) GFlop/s = 31.5 (24.2)% Peak compared to BGL kernel **1.3** (**1.1**) GFlop/s = 25.5 (20.2)% Peak

with room for improvement!

Outlook: QCD on Cell

This is very much work in progress

- The Cell Broadband Engine has peak of >200GF / CPU
- It also features a high BW I/O connectivity via the EIB

There is a study under way for a Cell based QCD computer (3d torus) with the next generation Cell BE (with improved double precision performance)

Performance of the single precision kernel:

- DMA bw tests suggest that a performance of 27% of peak, theoretical peak is 33% (see talk by H. Simma @ Cell Workshop @ FZJ-ZAM)
- Tests of serial code written with intrinsics run at >90% of peak

→ Writing the kernel in intrinsics alone might be sufficient

Thank you for your attention!