JLAB-THY-01-29

Optimized Lattice QCD Kernels for a Pentium 4 Cluster

Chris McClendon
JLab Summer High School Intership Program
JLab High Performance Computing Group and JLab Theory Group

September 15, 2001

Abstract

Soon, a new cluster of parallel Pentium 4 machines will be set up at JLAB to run Lattice QCD calculations. I discuss the rationale for optimized Lattice QCD routines, and how the features of the Pentium 4 enable new optimized routines to run much faster than normal C routines. I describe the optimization strategies used in SU(3) linear algebra routines, and in both single-node and parallel implementations of the Wilson-Dirac Operator. Finally, I show single node performance timings for the parallel version of the Wilson-Dirac operator.

Introduction

The U.S. Lattice QCD Collaboration consisting of various institutions and universities in the U.S., has recently been awarded grant money out of SciDAC (Scientific Discovery through Advanced Computing) for new hardware and software. Fermilab and JLAB will be the home of two new clusters of Pentium 4's, each linked together in a parallel architecture. These clusters will run computationally-intensive Lattice QCD calculations.

In lattice QCD, a four-dimensional box of points is used to simulate a region of the space-time continuum inside and surrounding a hadron. Quantum fields representing quarks are associated with the lattice sites, and quantum fields representing gluons have values at the links between sites. For the most accurate results, lattice spacing should be small. However, for smaller lattice spacing, more computational time is needed. A technique that eases this conflict is using higher-order derivatives of the field. Using these higher-order derivatives is faster for a given accuracy than decreasing the lattice spacing. A widely used simplification that reduces the computational time immensely is called the quenched approximation, in which quarks propagate in a background gluon field, but do not influence the behavior of the gluons. The propagation of the quarks is represented by extremely large, spare matrices, which typically contain two million rows and columns. Inverting the matrices, as required in the calculations, takes a very long time. Thus, improving the efficiency of such calculations is very important [1].

At Jefferson Lab National Accelerator Facility, physicists are experimentally measuring the masses of hadrons in the N* spectrum. Physicists are attempting to theoretically determine why the nature and the origin of the mass spectrum. Lattice QCD is the only method for calculating the masses of the hadrons from first principles, and is therefore being used to investigate the nature of the spectra of hadron masses.

Several features make parallel computing ideal for lattice QCD. The interactions are short-range, meaning that that for each lattice site, only several adjacent sites must be considered in the calculations. Also, the same operations have to be performed at each lattice site. The four-dimensional lattice is subdivided among a group of processors. Periodic boundary conditions are generally used to control surface effects at the edges of the box. These periodic boundary conditions make the communications cyclic on the grid of processors. Communication between processors of data on the surfaces of each processor's mini-lattice happens while the data points on the interior of the lattice are processed. However, parallel processing introduces challenges to efficiency. One such challenge is communication speed, a bottleneck in parallel computing. If communication speed is slower than processor speed, processors will be forced to wait for the communication to complete before continuing. Due to the communication between processors, synchronization is especially important; because of the synchronization, many processors must often wait in an idle state until all processors have reached a specified step in the calculations. Yet, parallel computing is still a good vehicle for lattice QCD [2]

SIMD on the Pentium 4

Martin Luescher at DESY has demonstrated the high efficiency obtainable for Lattice QCD kernel routines using the SSE vector registers on the Pentium 3 and 4 [3]. His use of inline assembly macros added an extra layer of abstraction onto the vector abstraction already made possible by the vector registers, a heartening feature for RISC assembly programmers. The eight 128-bit "xmm" vector registers allow SIMD (Single Instruction, Multiple Data) optimization for 32-bit and 64-bit data. In 32-bit SIMD, one instruction, an addition for instance, can take four floats in a source xmm register, add them to four other floats in the destination xmm register, and then place the four results in the destination register. Thus, 32-bit SIMD can perform operations with two complex numbers at a time. In 64-bit arithmetic, only two doubles fit in each register, so the SIMD instructions can only work with one complex number at a time (half of the register contains the real part, the other half, the imaginary part) [4].

Since Lattice QCD relies heavily on complex linear algebra, the 128-bit SIMD vector registers and SSE2 (Streaming SIMD Extensions) instructions are well suited to the task of producing high-performance Lattice QCD kernels. However, GCC does not even use SIMD instructions or registers in its generated code, and Intel's Pentium C/C++ optimizing compiler itself generally only vectorizes for simple loops with predictable data patterns [5]. Therefore, use of GCC inline assembly instructions facilitates the construction of optimized Lattice QCD kernels by providing a modular and abstract way to build up SIMD assembly code, while letting the C compiler take care of offsets and address calculations.

The QCD API

The QCD API (Application Protocol Interface) defines a flexible set of C-callable Lattice QCD functions that can be used by virtually any high-level lattice hadron physics code [6]. This API is composed of various levels which sufficiently abstract the basic operations of QCD. At the highest levels, a data-parallel paradigm is presented to the user which hide all architectural details of the computing system.

SZIN is a data-parallel object oriented Lattice QCD software system written by Robert Edwards (JLAB) and Tony Kennedy (Edinburgh) with two levels: architecture independent high level physics code and a macro-based system of data-parallel operations suitable for lattice QCD. These low level routines encapsulate the details of the architecture and have been implemented over a wide class of computing platforms including the QCDSP at JLab, single node and threaded-smp workstations, clusters of workstations, Cray, CM-2 and CM-5. C-callable routines similar to the ones derived from the generic macro-based system will form the core of the QCD API. Therefore, SZIN is an excellent software system in which to test the routines that will become a part of the QCDAPI.

SZIN Dirac operator: part of the Level 3 API

$$\chi(x) = \sum_{\mu=0}^{Nd-1} U_\mu(x)(1-\gamma_\mu)\psi(x+\widehat{\mu}) + \sum_{\mu=0}^{Nd-1} U_\mu(x-\widehat{\mu})(1+\gamma_\mu)\psi(x-\widehat{\mu})$$

The Wilson-Dirac operator is the most computationally intensive function in Lattice QCD. The computing time required for high-level physics code is proportional to the time required to compute this operator, because it is called multiple times for each iteration of the conjugate gradient inverter. Mathematically, the Wilson-Dirac operator, acting on each site, involves a sum over Euclidean directions of a spin-projected quark field multiplied by a gluon gauge link, taking quark fields from adjacent sites and gluon fields associated with the current and adjacent sites.

The Wilson-Dirac operator for the new Pentium 4 cluster is written in C and GNU inline assembly macros that take advantage of SIMD vector registers unknown to the GCC optimizing compiler. SZIN supports single-node and parallel versions of the Wilson-Dirac operator (dslash) in 32bit and 64bit arithmetic. In each version, the lattice is divided into even and odd checkerboards to allow red/black preconditioning. The dslash loops over the sites on, for example, the red checkerboard, and uses spinors on the black checkerboard and gauge fields on the red and black checkerboards

Figure 1. QCD API (*)

```
                                 level 3
   -------------------------------------------------------------------------
   |             Dirac Operators, CG Routines etc.                          |
   |                                                                        |
   |           (Organized by MILC or SZIN or CPS etc.)                      |
   |                                                                        |
   ==========================================================================

                                level 2
   --------------------------------------------------------------------------
   |                Data Parallel QCD Lattice Operations                    |
   |                 (overlapping Algebra and Messaging)                    |
   |                                                                        |
   |               A = SHIFT(B, mu) * C;  Global sums, etc                  |

   --------------------------------------------------------------------------
   |  Lattice Wide Lin Alg            || Lattice Wide Data Movement         |
   |   (No Communication)             || (Pure Communiation, non-blocking)  |
   |                                  ||                                    |
   |  e.g.   A = B * C                || e.g  Atemp = SHIFT(A, mu-dir)      |
   |                                  ||                                    |
   ==========================================================================

                                 level 1
   --------------------------------------------------------------------------
   | Single Site Linear Algebra API ||     Message Passing API              |
   |                                ||                                      |
   | SU(3), gamma algebra etc.      ||    (Know about mapping of Lattice    |
   |                                ||     into Network Geomentry)          |
   --------------------------------------------------------------------------


 (*)Ideally each level only calls the level directly below it. When this is
 not possible because of efficiency, this level structure should still be
 maintained in the SEMANTICS.
```

C Preprocessor Abstractions

The code created makes heavy use of C Preprocessor #define's and macros. These overlays create an abstract means of expressing the necessary operations and facilitates porting the code to different systems of data layouts, lattice geometries, etc., and were very helpful in producing the hundreds of generic SU(3) algebra routines. These macros are located at the top of each dslash code, for ease of reference.

Single-Node Implementation of the SU(3) algebra routines for the Level 1 API

The general-purpose SU(3) linear algebra routines optimally walk over a number of sites and perform the multiplication of an SU(3) matrix with a spinor, another SU(3) matrix, a halfspinor, or a lone SU(3) vector. In 32-bit arithmetic, for the case of the multiplication of an odd number of columns, a new macro multiplies SU(3) matrices by SU(3) color vectors from two sites at the same time. In the 64-bit arithmetic, the SU(3) multiplication macro only can operate on one SU(3) color vector at a time, because of the reduced number of words per vector register.

A myriad of possibilities are supported:

$$\begin{array}{ll}
r = ab & r = r - ab \\
r = ab^{\dagger} & r = r - ab^{\dagger} \\
r = a^{\dagger}b & r = r - a^{\dagger}b \\
r = a^{\dagger}b^{\dagger} & r = r - a^{\dagger}b^{\dagger} \\
r = -ab & r = r + ab \\
r = -ab^{\dagger} & r = r + ab^{\dagger} \\
r = -a^{\dagger}b & r = r + a^{\dagger}b \\
r = -a^{\dagger}b^{\dagger} & r = r + a^{\dagger}b^{\dagger}
\end{array}$$

Also, the routines support any combination of gather/scatter operations on the input/output operands via shift tables. The gather and scatter possibilities can be used by the Level 2 API to parallelize these routines across multiple nodes and/or threads, depending on the machine configuration used.

SZIN Dirac Operator Optimization for the Pentium 4: General Considerations

The optimization strategies used are conditioned by several features of the Pentium 4 processor. The Pentium 4 processor has a 256KB level 2 cache with an 128-byte cache line. This is the only cache that can be software-prefetched into on the Pentium 4 processor [5]. This implementation is very sensitive to the physical data layout of the spinors and gauge fields because of the high cost of cache misses. Also, it is important to make sure that the data fall out of cache as little as possible to reduce memory bandwidth. Prefetching does not take up memory bandwidth if the data are already in cache. Working with few cache lines at a time is more efficient than using bits and pieces from many different cache lines. Optimizations that facilitate loop blocking, an efficient data layout technique, include splitting larger lattices into smaller hypercubes and bringing the boundaries in one direction closer together in physical memory. However, due to the dependency of higher-level code on a canonical data layout of spinors, such optimizations could not be included. Static memory-efficient layouts of gauge fields are possible in quenched Lattice QCD because the gauge fields stay the same. Therefore, a “packed” copy of the gauge fields is created for use by the Dirac operator.

Single-Node Implementation of the SZIN Dirac Operator, Part of the Level 3 API

SZIN Dirac operator: 32 bit

Several key features give the single-node dslash an extremely high performance. Instead of looping over the lattice one site at a time, the dslash walks along the lattice two sites at a time, taking advantage of the long, 128-byte prefetchable cache line. In addition, it features an SU(3) multiplication that operates on two spin components at a time without spilling or reloading data.

SZIN Dirac operator: 64 bit

Due to the larger data sizes and reduced words per vector register (2 instead of 4), the 64 bit code only delivers about half the performance in Megaflops of the 32-bit code. The 64 bit dslash walks over each site separately, and can not work with two spin components at a time in the manner that the 32 bit code can.

Data-Parallel Implementation of the SZIN Dirac Operator, Part of the Level 3 API

The parallel implementation splits up a large lattice into sublattices. Each sublattice lives in a different node. Since the dslash only involves data from nearest neighbor sites, the only data that need to be communicated are data from the sublattice boundaries.

In the 32-bit case, since the loop over sites is unrolled into a loop over two sites at a time and since the SU(3) multiplication is always performed with the halfspinors and gauge fields associated with the current site, the gauge fields can be (and are assumed to be by default) laid out in memory such that each loop works with two adjacent lattice sites' worth of data, and the second sites' worth of gauge fields vary faster than the direction associated with the gauge fields. The actual lexical mapping can easily be seen in the macro overlays.

The implementation must overlap as much computation and communication as reasonably possible in order to maximize performance.

A simple way to do this is as follows: first, do the spin decomposition. Next, communicate the spin projected halfspinors in the "forward" direction while performing the Hermitian conjugate multiplication. Synchronize (e.g., wait for the communications), and then communicate the multiplied halfspinors in the "backward" direction while performing the normal multiplication. Synchronize, and finally reconstruct the other two spin components while summing over the four directions:

$$(1-\gamma_\mu) = \begin{pmatrix} I^{2x2} \\ M^{2x2} \end{pmatrix} (I^{2x2} \quad N^{2x2})$$

$$(1+\gamma_\mu) = \begin{pmatrix} I^{2x2} \\ -M^{2x2} \end{pmatrix} (I^{2x2} \quad -N^{2x2})$$

Communication..*Computation*

....$\chi_\mu^a(x+\hat{\mu}) = \begin{pmatrix} I^{2x2} & N^{2x2} \end{pmatrix}\psi(x)$

....$\chi_\mu^b(x-\hat{\mu}) = \begin{pmatrix} I^{2x2} & -N^{2x2} \end{pmatrix}\psi(x)$

$Communicate(boundaries(\chi_\mu^a, +\hat{\mu}), +\hat{\mu})$.........$\chi_\mu^c = U_\mu^\dagger(x)\chi_\mu^b$

$Synchronize()$..

$Communicate(boundaries(\chi_\mu^a, -\hat{\mu}), -\hat{\mu})$.........$\chi_\mu^d = U_\mu(x)\chi_\mu^a$

$Synchronize()$..

....$\Psi(x) = \sum_{\mu=0}^{Nd-1} \begin{pmatrix} I^{2x2} \\ M^{2x2} \end{pmatrix}\chi_\mu^d + \begin{pmatrix} I^{2x2} \\ -M^{2x2} \end{pmatrix}\chi_\mu^c$

However, the parallel SZIN dslash goes a step further: only the half of the spin projection that is needed for the first communication (the "backwards" direction) is done outside the communication call, and only half of the spin reconstruction is done outside the last synchronization barrier (the half that involves terms that were just communicated in the "forward" direction):

Communication..*Computation*

....$\chi_\mu^a(x+\hat{\mu}) = \begin{pmatrix} I^{2x2} & N^{2x2} \end{pmatrix}\psi(x)$ (1)

$Communicate(boundaries(\chi_\mu^a, +\hat{\mu}), +\hat{\mu})$.............$\chi_\mu^c(x) = U_\mu^\dagger(x)\begin{pmatrix} I^{2x2} & -N^{2x2} \end{pmatrix}\psi(x)$ (2)

$Synchronize()$..

$Communicate(boundaries(\chi_\mu^c, -\hat{\mu}), -\hat{\mu})$.............$\Psi(x) = \sum_{\mu=0}^{Nd-1} \begin{pmatrix} I^{2x2} \\ M^{2x2} \end{pmatrix} U_\mu(x)\chi_\mu^a(x)$ (3)

$Synchronize()$..

....$\Psi(x) = \Psi(x) + \sum_{\mu=0}^{Nd-1} \begin{pmatrix} I^{2x2} \\ -M^{2x2} \end{pmatrix}\chi_\mu^c(x)$ (4)

The first lattice-sized region of each temporary, χ, contains information for non-boundary sites. The boundary values are placed in "TAIL1", a lattice-sized region following the first region. A third lattice-sized region, "TAIL2", is used to receive the boundaries sent from the adjacent nodes. For optimal communications, all of the boundaries for each direction must be placed contiguous in memory. A set of scatter/gather lookup tables was created to lexically map the lattice boundaries to these "tail" buffers for variable lattice sizes and parallel machine geometries.

This parallel implementation in general uses the same set of assembly macros for spin projection, SU(3) multiplication, and spin reconstruction as the single-node implementation. While routine (1) is heavily memory-bandwidth-bound, the contiguous nature of the loads and the use of prefetching leads to few stalls. Routine (2) looks up the addresses in which to scatter the output and prefetches from it before the multiplication for better performance. Routine (3) hides the latency of a gather on the input lattice temporary with prefetching and computationally-intensive SU(3) multiplications of halfspinors. Routine (4) loops over output spin components and uses three of the“xmm” registers to accumulate the three colors of two output components at a time. While it prefetches several iterations ahead, it remains heavily memory-bandwidth-bound.

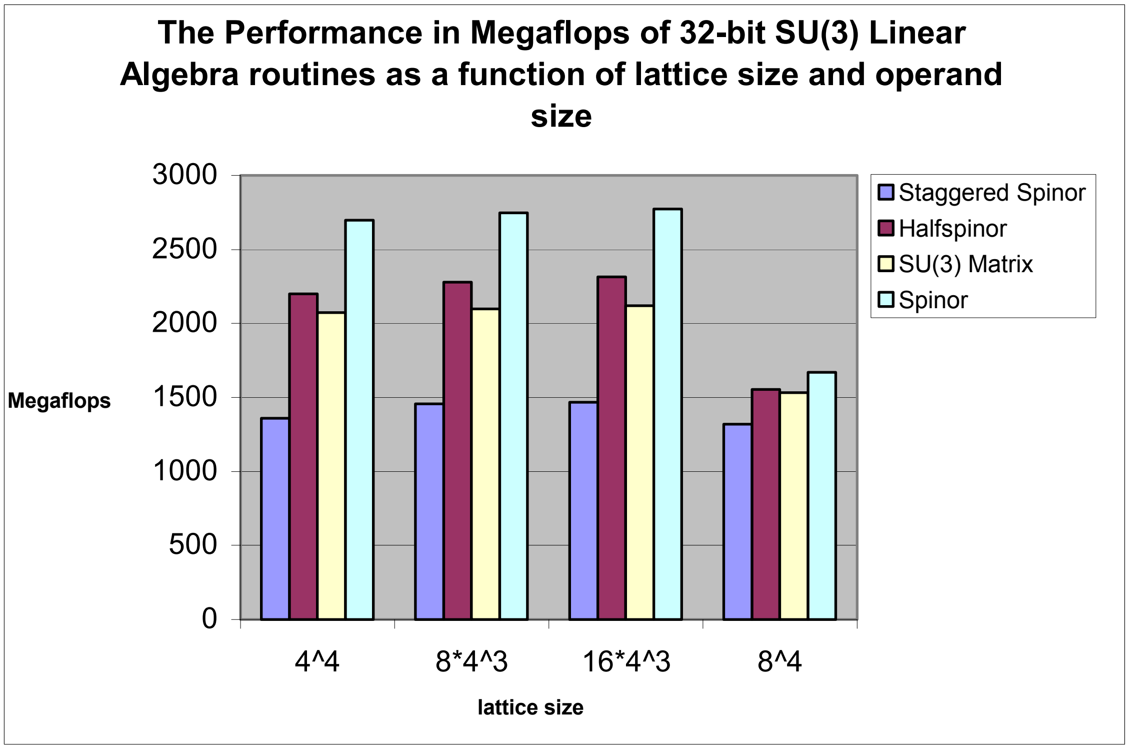
The most optimal approach, though difficult, would apparently involve only spin projecting the boundaries before communicating, and likewise reconstructing all of the sites except the boundaries before the final synchronization. However, there are different boundaries for each direction, and treating each direction separately involves a separate loop over sites for each direction, resulting in a performance hit incurred by reloading the input spinors. Thus, it is not known at this time whether this more difficult approach is worth pursuing.

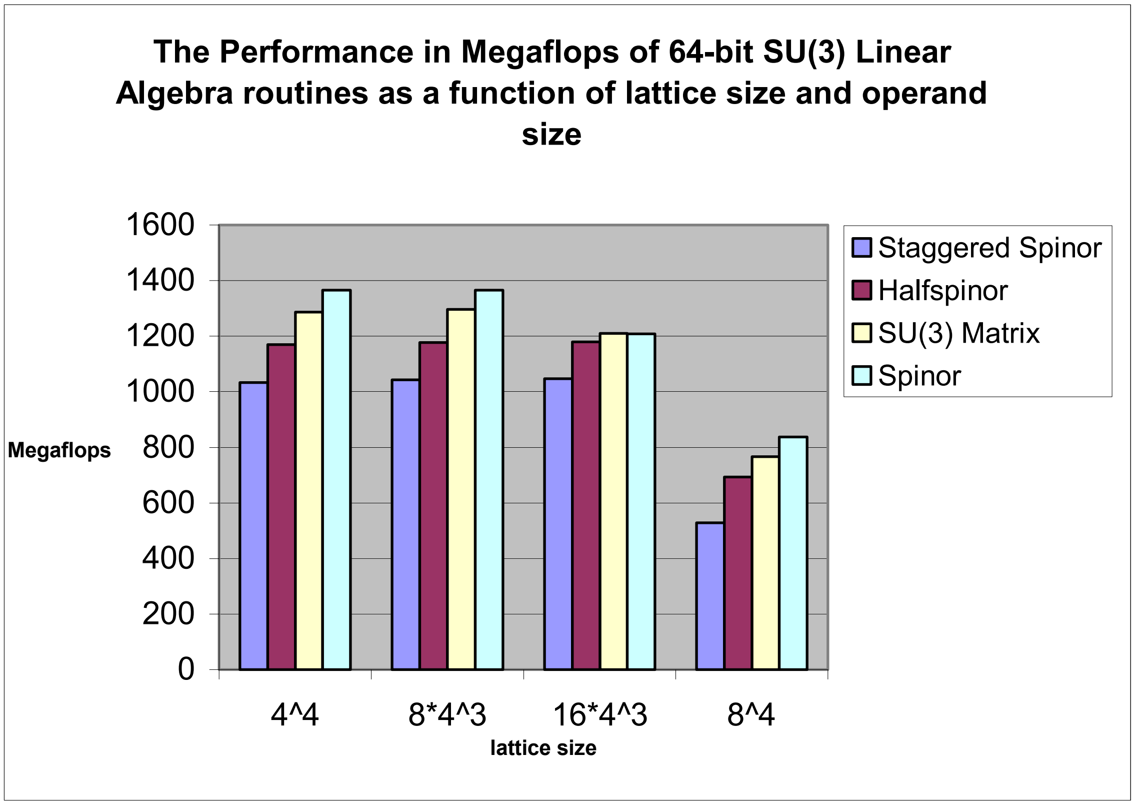
Additional Routines: part of the Level 1 API

Also needed for a conjugate gradient inversion are two additional routines: one that computes (for a number of sites) the sum of a spinor and a constant multiplied by another spinor, and one that computes the square of the norm of spinors. These are memory-bandwidth bound (heavily memory bound in the x =x + a*y case), but are benefited from the use of SIMD vector registers and cache prefetches.

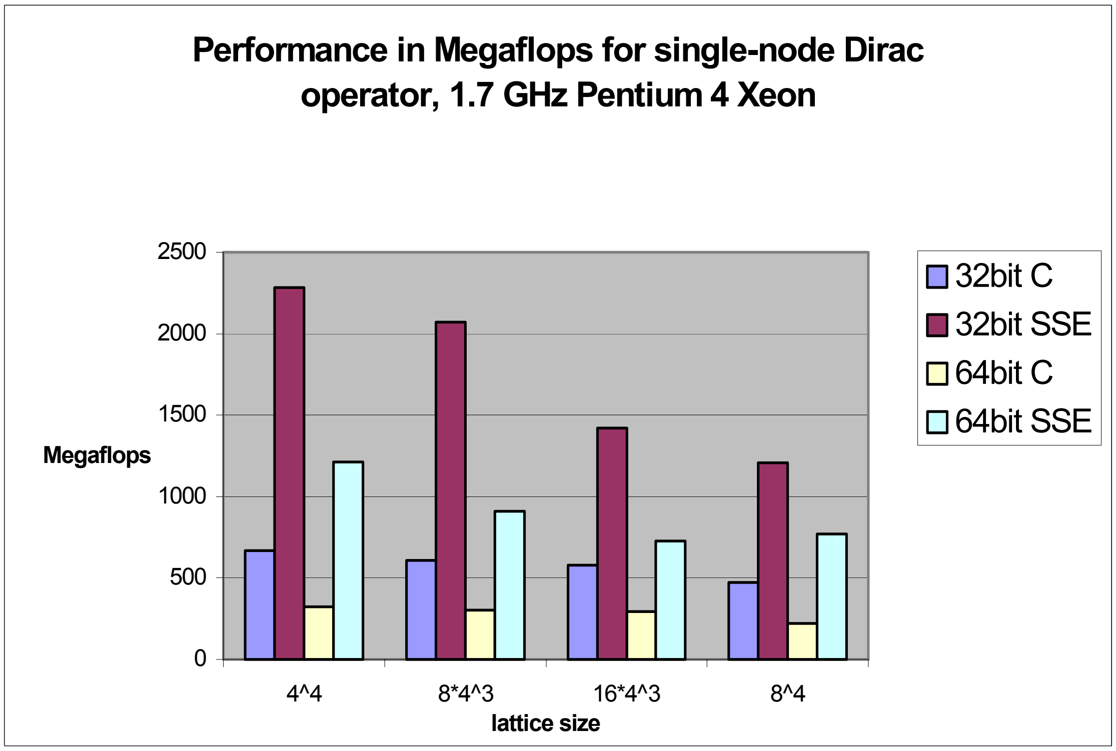
Results

Shown below are the single-node performance data for the generic SU(3) linear algebra routines, in terms of Megaflops, as a function of the number of SU(3) vectors per lattice site in the operand and the lattice size.

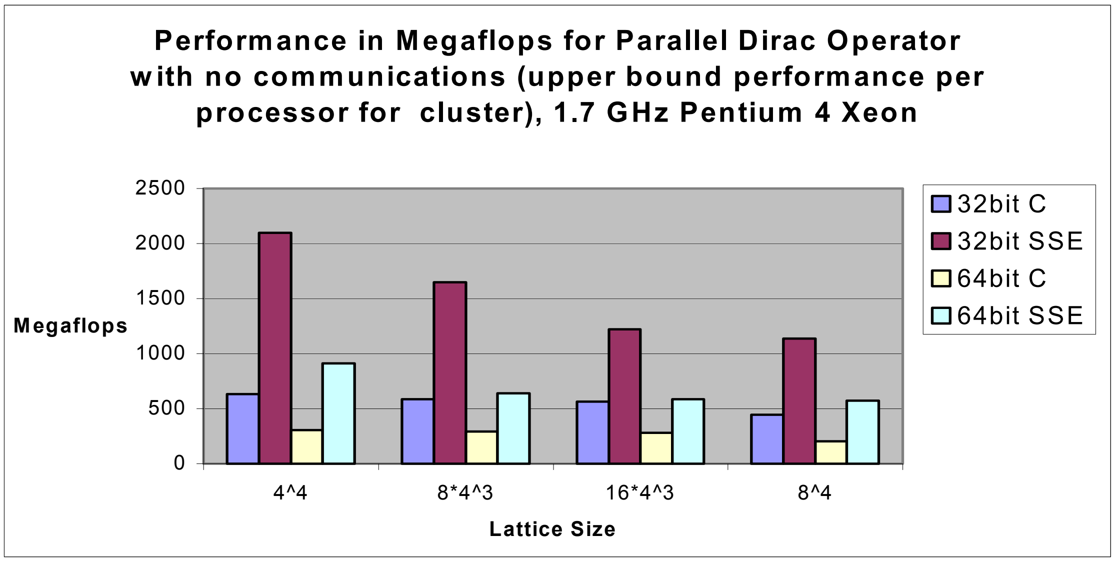




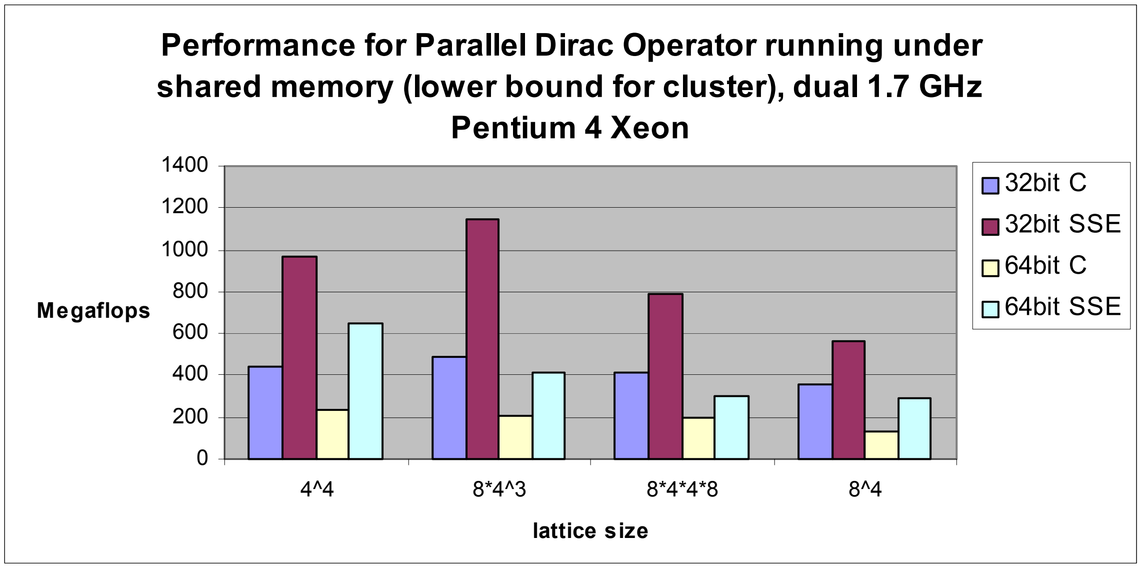
Show below is the 32-bit and 64-bit single-node performance of the canonical C code SZIN Dirac operator and the new optimized Dirac operator.



The next graph shows the performance of the 32-bit and 64-bit parallel performance of the canonical C code SZIN Dirac operator and the new optimized Dirac operator with "fake" communications (since no directions involve communications between nodes, the temporaries only end up using the first (local) region of the temporaries and not the "TAIL" buffers): this sets the upper bound on actual parallel performance.



Now for completeness we show the performance of the 32-bit and 64-bit parallel performance of the canonical C code SZIN Dirac operator and the new optimized Dirac operator with communications using MPI shared memory over a dual Pentium 4 Xeon: this (hopefully) sets the lower bound on actual parallel performance, because the communication is handled across the memory bus, which will compete with the computations running concurrently.



Conclusions

This use of SSE instructions double, triple, and in some routines almost quadruple the performance of C code alone, and will enable Lattice QCD calculations to run much faster than they ordinarily would with C code alone. The fast parallel implementation of the Dirac operator in many cases removes the computational throughput bottleneck that existed and encourages the development of faster communications.

However, I was unable to obtain timings using multiple threads SMP on dual-processor Pentium 4 Xeon machines because of an operating-system bug. Using multiple threads could allow a performance increase per node of perhaps 50%. However, multiple processor machines share the same 400 MHz memory bus, and with the introduction of the 2.0 GHz Pentium 4, dual processor machines may not show significantly higher performance than single processor machines.

Overall, the Pentium 4 processor was shown to be very powerful with small lattice sizes. However, it is extremely sensitive to physical data layout, as the good data layout in the parallel implementation enabled the performance of the parallel implementation (without communications) to approach the performance of the single-node implementation for the smallest lattice size, in which cache misses were rare. Each way of level 2 cache can only hold 256 cache lines' worth of data, so it seems reasonable that lattice sizes greater than 4^4 for the 32-bit parallel implementation and 8*4^3 for the single-node implementation would show a drop in performance corresponding with level 2 cache misses, because of the use of gauge fields from both checkerboards. Even though prefetching is helpful, if a prefetch results in a cache miss, the memory bus is used, and in memory-bandwidth-bound cases, prefetching can decrease performance [5].

Acknowledgements

I would like to thank Dr. Robert Edwards for his time, for getting me started with Pentium 4 SIMD, for shedding light on various tricks, for much advice when I was stuck, and for helping me to integrate my code into his. I would like to thank Dr. Martin Luescher for his prototype code, which was very helpful in revealing some of the optimization tricks for the Pentium 4,as a starting point for the single-node SZIN mod, and as a simple software package in which to make a prototype of the parallel implementation. I would also like to thank Dr. David Richards for his advice, support, and for his good editing of this paper. This work was supported by a grant from the DOE SciDAC program and by DOE contract DE-AC05-84ER40150 under which the Southeastern Universities Research Association (SURA) operates the Thomas Jefferson National Accelerator Facility (TJNAF).

References:

1. C. Davies and S. Collins, 2000. Getting to grips with the strong force. Physics World 13(8):35-40.

2. R. Gupta, 1999. General physics motivations for numerical simulations of quantum field theory. <http://xxx.lanl.gov/abs/hep-lat/9905027>.

3. Martin Luescher. Personal communication. April 2001.

4. *IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture*. Accessed September 12, 2001. ftp: // download.intel.com /design /Pentium4 / manuals/24547004.pdf

5. *Intel® Pentium® 4 Processor Optimization Reference Manual*. Accessed Septermer 12, 2001. ftp:// download.intel.com/ design/ Pentium4/ manuals/ 24896604.pdf

6. "QCD API". Accessed September 12, 2001. http://www.jlab.org/~edwards/ qcdapi/QCDAPI.htm

7. Christopher L. McClendon. "The Performance of UKQCD on Parallel Alpha Architectures." May 2001.

Appendix A. Spin Basis

This Appendix is included from the SZIN manual:

The spin conventions are contained in the macro file spinor.mh . The default spin conventions used are the same as the MILC code and also the Columbia Physics System. The basis is a chiral one. The gamma matrices are below.

Here is the basis. Note, they are labeled gamma_{1,2,3,4} for x,y,z,t in that order. You could also think of x as the 0th directorion, so they are from 0 to 3, inclusive.

```
# gamma(   1)   0  0  0  i           # ( 0001 )  --> 1
#               0  0  i  0
#               0 -i  0  0
#              -i  0  0  0

# gamma(   2)   0  0  0 -1           # ( 0010 )  --> 2
#               0  0  1  0
#               0  1  0  0
#              -1  0  0  0

# gamma(   3)   0  0  i  0           # ( 0100 )  --> 4
#               0  0  0 -i
#              -i  0  0  0
#               0  i  0  0

# gamma(   4)   0  0  1  0           # ( 1000 )  --> 8
#               0  0  0  1
#               1  0  0  0
#               0  1  0  0
```

Appendix B. Options for SU(3) Algebra Library

In total, all of the options listed above for the SU(3) algebra routines add up to a few hundred procedures, generated using four template procedures and lots of C Preprocessor tricks. The function calls look like: Function_name(src1, src2, dest, # of sites, stride1, stride2, stride3 <,shift1> <,shift2> <,shift3>), where shift1, shift2, and shift3 are the shift tables used to gather or scatter from src1, src2, and dest, respectively.

Note: the library does support strides other than 1, but VARIABLE_STRIDE must be defined…otherwise, stride lengths other than 1 will be treated as though they are stride lengths of 1.

The SU(3) matrix * SU(3) matrix multiplication showed less performance in the 32-bit case than the SU(3) matrix * spinor or the SU(3) matrix * halfspinor multiplications because 1/3 of that routine uses two gauge fields to multiply two sites' worth of columns at a time, and the two loads into an "xmm" register before the shuffling of the two matrix elements in the register into a R^4 or I^4 single color vector creates a dependency chain that cannot be avoided if we wish to carry out four operations at once; there simply are not enough "xmm" registers.

Appendix B. 32-bit Spin-Projection, SU(3) Multiplication, and Spin-Reconstruction Macros

The spin basis-specific macros have been defined outside of the dslash to provide a not-too-difficult way to change the spin basis. Basically, 1 (+ or -) gamma(mu^) is expressed as the product of a 4x2 matrix and a 2x4 matrix, and when each of these is partitioned into two 2x2 matrices, the 2x2 matrices are related by simple invertible operations. Then, the action of these matrices on a spinor can be easily expressed in terms of vector registers and instructions. Here's an example:

```
#define _sse_42_gamma0_minus()   _sse_vector_xch_i_sub()
```

This means that the 2x4 spin projection matrix looks like this :

$$(I^{2x2} \quad N^{2x2})\psi(x) = \begin{vmatrix} 1 & 0 & 0 & -i \\ 0 & 1 & -i & 0 \end{vmatrix}$$

First, before this macro is called, the two-component spin projection takes (all 3 colors of) the first and second components of the input into 3 xmm registers, and the third and fourth into another 3 xmm registers. Now to form the 2x2 submatrix on the right from the one on the left, we reflect it vertically, and multiply by i. So this inline asm macro swaps the third and fourth components around in their 3 xmm registers, and then performs three subtractions of xmm registers for the three colors, and then the halfspinor is in the right registers for the SU(3) multiplication.

The SU(3) multiplication of a halfspinor is denoted by the macro "_sse_su3_multiply()", and its Hermitian conjugate denoted by the macro "_sse_su3_inverse_multiply()". These SU(3) multiplication macros take a halfspinor in three of the "xmm" registers and do the multiplication, neither reloading matrix elements of the input SU(3) matrix nor spilling temporaries to memory, and leave the output halfspinor in three of the "xmm" registers.

After the SU(3) multiplication, the 4x2 matrix in the factorization describes how the two remaining spinor components can be reconstructed from the two components that are the output of the SU(3) multiplication. The following statement defines the appropriate matrix:

#define _sse_24_gamma0_minus_set() _sse_vector_xch_i_mul_up()

$$\begin{pmatrix} I^{2x2} \\ M^{2x2} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & i \\ i & 0 \end{pmatrix}$$

This will swap the two components of the half-spinor and multiply them by i to form the bottom two components.

The "_set" here means that this is the first part of the summation of the result spinor, so no addition or subtraction is needed. An "_add" suffix (present for only the first direction) would denote that the partial sum of the output spinor is to be added or subtracted from instead of "_set". The dslash routine spills the output spinor to memory, loading it later when it is needed in the sum over directions. At the end of the loop, the output spinor, the sum of both terms in the sum over directions, is stored to memory.

The complete list of spin basis-specific macros used by the 32-bit dslash:

```
/* gamma 0 */
#define _sse_42_gamma0_minus()       _sse_vector_xch_i_sub()
#define _sse_42_gamma0_plus()        _sse_vector_xch_i_add()
#define _sse_24_gamma0_minus_set()   _sse_vector_xch_i_mul_up()
#define _sse_24_gamma0_plus_set()    _sse_vector_xch_i_mul_neg_up()
#define _sse_24_gamma0_minus_add()   _sse_vector_xch_i_add()
#define _sse_24_gamma0_plus_add()    _sse_vector_xch_i_sub()
 /* gamma 1 */
#define _sse_42_gamma1_minus()        _sse_vector_xch();   _sse_vector_addsub()
#define _sse_42_gamma1_plus()        _sse_vector_xch();   _sse_vector_subadd()
#define _sse_24_gamma1_minus()       _sse_vector_xch();    _sse_vector_subadd()
#define _sse_24_gamma1_plus()        _sse_vector_xch();    _sse_vector_addsub()
/* gamma 2 */
#define _sse_42_gamma2_minus()       _sse_vector_i_subadd()
#define _sse_42_gamma2_plus()        _sse_vector_i_addsub()
#define _sse_24_gamma2_minus()       _sse_vector_i_addsub()
#define _sse_24_gamma2_plus()        _sse_vector_i_subadd()
/* gamma 3 */
#define _sse_42_gamma3_minus()            _sse_vector_sub()
#define _sse_42_gamma3_plus()             _sse_vector_add()
#define _sse_24_gamma3_minus()            _sse_vector_sub()
```

```
#define _sse_24_gamma3_plus()            _sse_vector_add()
#define _sse_24_gamma3_minus_rows12()    _sse_vector_add()
#define _sse_24_gamma3_plus_rows12()     _sse_vector_add()
```

Appendix C. 64-bit spin-projection, SU(3) multiplication, and spin-reconstruction macros

The spin basis-specific macros have been defined outside of the dslash to provide a not-too-difficult way to change the spin basis. Since the 64 bit dslash can not work with two spinor components at a time, each component must be treated separately. Also, the loads and stores used are part of the spin-basis specific macros. For instance, following the example above:

```
#define _sse_42_1_gamma0_minus(sp) \
    _sse_load((sp)_c1__); \
    _sse_load_up((sp)_c4__);\
    _sse_vector_i_mul();\
    _sse_vector_sub()

#define _sse_24_1_gamma0_minus_set() \
    _sse_store_up(rs_c1__);\
    _sse_vector_i_mul_up();\
    _sse_store_up(rs_c4__)

#define _sse_24_1_gamma0_minus_add() \
    _sse_load(rs_c1__);\
    _sse_vector_add();\
    _sse_store(rs_c1__);\
    _sse_load(rs_c4__);\
    _sse_vector_i_mul();\
    _sse_vector_add();\
    _sse_store(rs_c4__)
```

For example, in _sse_42_1_gamma0_minus(sp), spinor components 1 and 4 are loaded into registers. Then, component 4 is multiplied by i, and then subtracted from component 1. This corresponds to the top row of 2x4 spin projection matrix in the above representation. Then, the resulting half of a halfspinor is passed to the SU(3) multiplication.

The 64-bit SU(3) multiplication of a SU(3) matrix by one spin component of a halfspinor is denoted by the macro _sse_su3_multiply(), and its Hermitian conjugate denoted by the macro _sse_su3_inverse_multiply(). These SU(3) multiplication macros take one spin component of a halfspinor in three of the “xmm” registers and do the multiplication, neither reloading matrix elements of the input SU(3) matrix nor spilling temporaries to memory, and leave the output one spin component of a halfspinor in three of the “xmm” registers.

After the SU(3) multiplication, the 4x2 matrix in the factorization describes how the two remaining spinor components can be reconstructed from the two components that are the output of the SU(3) multiplication. So, _sse_24_1_gamma0_minus_add() or _sse_24_1_gamma0_minus_set() will reconstruct one of the two spinor components that were not spin projected--using the first component of the halfspinor--and then perform one direction's worth of the sum over directions to two spinor components. Directions 0 and 3 are special cases, because in the "set" case, the term is the first or last in the sum over directions and needs to be treated as such; the "add" suffix denotes that the term should be added to the partial sum over directions thus far. At the end of the loop, the output spinor, the sum of both terms in the sum over directions, is stored to memory.

The spin basis-specific macros for the 64-bit code have not been included here because they are not as concise, elegant, or abstract as the 32-bit-specific macros.