Empirical Tuning of a Multiresolution Analysis Kernel Using a Specialized Code Generator*

Haihang You† Keith Seymour† Jack Dongarra‡ Shirley Moore†

March 5, 2007

1 Introduction

As part of our participation in the Performance Engineering Research Institute (PERI)*, we were made aware of an opportunity to apply empirical tuning techniques to a performance limiting kernel for the MADNESS† framework for adaptive multiresolution methods in multiwavelet bases [1]. Preliminary results using the General Code Optimization (GCO) system demonstrated effectiveness of empirical tuning of a C version of the code automatically translated from the Fortran version. In this report, we show that utilizing a directed empirical tuning method, in which we extract the computational kernel and use a more aggressive hand written code generator, achieves better performance than the previously applied GCO method.

2 Tuning the Multiresolution Analysis Kernel

The kernel we are tuning is encapsulated in the function `doitgen` in the original source code. The function `doitgen` is a good candidate for empirical tuning since it has a loop nest that can be optimized and the loops contain no external function calls. We translated the code to C, which is shown in Section 3.2. By studying the `doitgen` source code, we were able to reformulate the code in terms of matrix-vector multiplication. The modified code is shown in Section 3.3. Since our performance tuning task switched from tuning `doitgen` to tuning the matrix-vector kernel, we wrote a specific code generator for tuning matrix-vector multiplication (`dgemv` in BLAS terminology). Section 3.4 is an example of the generated dgemv code with block size of 2 and unrolling size of 2. In Section 3.5, the special comments at the top of the wrapper code are directives used by the tuning system to generate the appropriate testing driver and Makefile. These argument specification directives are written by hand. The function `doitgen` has a relatively small search space because the upper

*This work was supported by the Department of Energy SciDAC program under grant no. DE-FC02-06ER25761.

†Department of Computer Science, University of Tennessee, Knoxville

‡Department of Computer Science, University of Tennessee, Knoxville and Oak Ridge National Laboratory

*http://www.peri-scidac.org/

†http://www.csm.ornl.gov/ccsg/html/projects/madness.html

bound of each dimension of the input array is 31. Therefore, the more sophisticated search techniques were not necessary and a brute force search was used. For each matrix size from 1 to 31, we searched for the best `dgemv` code variant in terms of block size and unrolling amount. We tested all block sizes from 1 to 16 and for unrolling, we limited the maximum unroll amount to no greater than the selected block size. So far we have done experiments on Intel Pentium 4, Intel Core Duo (Woodcrest), and AMD Opteron. The specifications of these platforms are shown in Table 1. A comparison of the performance of `doitgen`, which includes the Fortran reference code, the Fortran hand-tuned code, C code translated from Fortran reference code, C code with simple `dgemv`, with ATLAS(P4 and Opteron) and MKL(Woodcrest), and with the generated `dgemv`, is shown in Figures 1, 2, and 3. The hand-tuned code (designated "Fortran doitgen RH tuned" in the graphs) was unrolled to varying depths by hand. The empirically tuned code is about 1.5 times faster than the hand-tuned version, and 2 to 3 times faster than the reference code.

Feature	Pentium 4	Woodcrest	Opteron
Processor Speed	1.7GHz	3.0GHz	1.8GHz
L1 Instruction	12KB	32KB	64KB
L1 Data	8KB	32KB	64KB
L2	256KB	128KB	1024KB
OS	Linux	Linux	Linux
C Compiler	gcc 3.4.4	icc 9.1.043	icc 9.1.043
Fortran Compiler	g77 3.4.4	ifort 9.1.037	ifort 9.1.037

Table 1: Processor Specifications

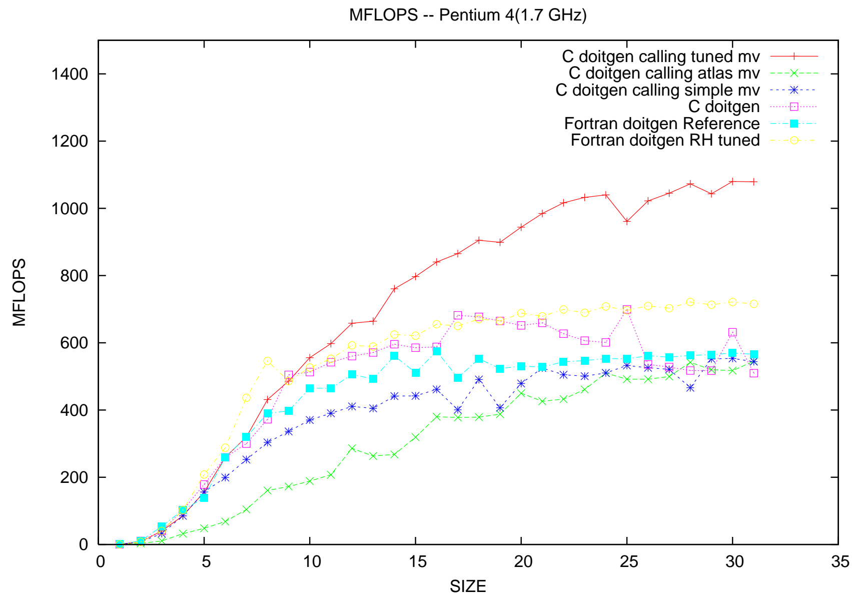

Figure 1: Pentium 4

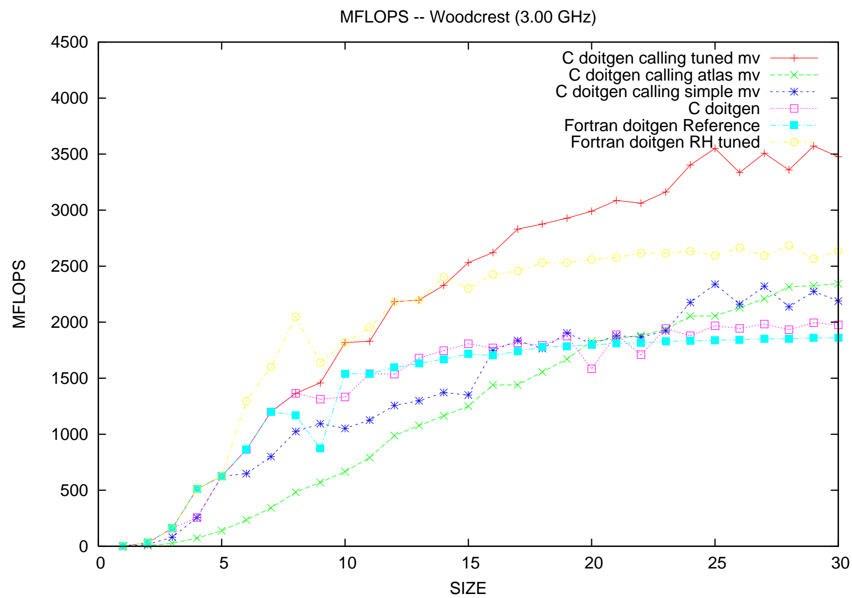

Figure 2: Woodcrest

3 Source Code

3.1 Reference doitgen in fortran

```
      subroutine doitgen_ref(a,ia1,ia2,ia3,x,ldx,np,nq,nr,ns)            doitgen_ref
      implicit none
      integer ia1,ia2,ia3,ldx,np,nq,nr,ns
      double precision a(*), x(ldx,*)
c
c     INPLACE transformation
c
c     a(p,q,r) <- sum(s) a(s,q,r) * x(s,p)
c
c     where ia1, ia2, ia3 are the increments between elements in the      10
c     respective dimensions of a
c
c     p = 1..np
c     q = 1..nq
c     r = 1..nr
c     s = 1..ns
c
c     TO DO (or done)
c
c     1) Test use of a DAXPY kernel so that can use the sparsity of the   20
c     operator x. Anticipate about 3-6x reduction in the flopcount by
c     doing this. Did this. There's lots of sparsity but the daxpy-like
```

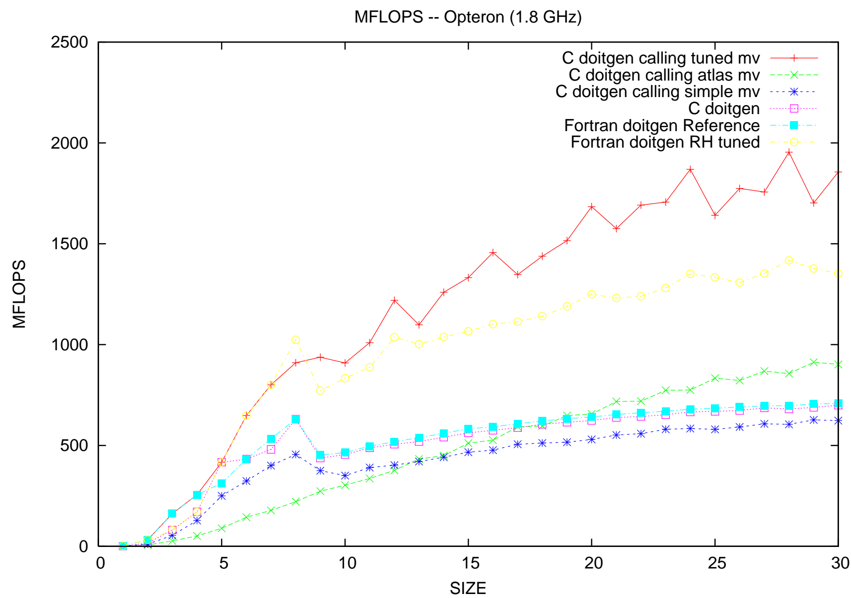

Figure 3: Opteron

```
c      kernel is much less efficient and the compiler does not optimize
c      as agressively with the if tests present.
c
c      2) Ditto using sparsity in the function.
c
c      3) Get rid of static data so can use multithreading
c
c      4) Tried special code for ia1=1 but no benefit                          30
c
c      Now try extensive hand unrolling to eliminate loops. For the zero
c      translation block, the matrices are nearly all full rank.  For the
c      unit translation, the matrices are about 1/3 rank. However, we
c      will always have one of the dimensions of x being n. So, try
c      unrolling one of s or p.
c
c       integer flops
c       common / cdoit2 / flops
c                                                                              40
c      ia64 and power4 unroll s/p =8 best
c      tried up to s=10 on power4 but no improvement
c
c       integer iflops
       integer p, q, r, s, pqr, qr, sqr
       double precision sum
       double precision t0(30)
```

```
*      static data not good for threading
*****      save t0, t1
c                                                                      50
c        iflops = np*nq*nr*ns
c        flops = flops + np*nq*nr*ns
c
c      This is the original version . . . a new port to another processor
c      could start from this.
c
      do r = 1, nr
      do q = 1, nq
      qr = 1 + (q−1)*ia2 + (r−1)*ia3
      do p = 1, np                                                     60
      sum = 0.0d0
      sqr = qr
      do s = 1, ns
      sum = sum + a(sqr)*x(s,p)
      sqr = sqr + ia1
      end do
      t0(p) = sum
      end do
      pqr = qr
      do p = 1, np                                                     70
      a(pqr) = t0(p)
      pqr = pqr + ia1
      end do
      end do
      end do
      end
```

3.2 Reference doitgen in C

```
int doitgen_ref_C(double *a, int ia1, int ia2,                         doitgen_ref_C
      int ia3, double *x, int ldx, int np, int nq,
      int nr, int ns)
{

      int p, q, r, s, pqr, qr, sqr;
      double sum;
      double t0[30];

      for(r = 0; r < nr; r++){                                         10
            for(q = 0; q < nq; q++){
                  qr = q * ia2 + r * ia3;
                  for(p = 0; p < np; p++){
                        sum = 0;
                        sqr = qr;
```

```
                        for(s = 0; s < ns; s++){
                                sum = sum + a[sqr] * x[s + p * ldx ];
                                sqr = sqr + ia1;
                        }
                        t0[p] = sum;
                }
                pqr = qr;
                for(p = 0; p < np; p++){
                        a[pqr] = t0[p];
                        pqr = pqr + ia1;
                }
        }
    }
    return 0;
}
```

3.3 doitgen with Matrix-Vector Kernel

```
inline void mv(int M, int N, double alpha, double *A, double *B, double *C, int lda)
{
    int i, j;
    double sum0;

    for(i=0;i<N;i++){
        sum0 = 0;
        for(j=0;j<M;j++){
            sum0 += A[i*lda+j]*B[j];
        }
        C[i] = sum0;
    }

}

int doitgen_ref_C_mv(double *a, int ia1, int ia2,
    int ia3, double *x, int ldx, int np, int nq,
    int nr, int ns)
{

    static int p, q, r, s, pqr, qr, sqr;
    static double t0[30];
    double a_copy[ns], *a_start;

    for(r = 0; r < nr; r++){
        r_ia3 = r * ia3;
        for(q = 0; q < nq; q++){
            qr = q * ia2 + r_ia3;
            a_start = a + qr;
```

```
                for(s = 0; s < ns; s++){                                        30
                        a_copy[s] = *a_start;
                        a_start += ia1;
                }

                for(p = 0; p < np; p++)
                        t0[p] = 0;

                mv(np, ns, 1, x, a_copy, t0, ldx);

                a_start = a + qr;                                               40
                for(p = 0; p < np; p++){
                        *a_start = t0[p];
                        a_start += ia1;
                }
        }
    }
    return 0;
}
```

3.4 Matrix-vector multiplication with bk=2, unroll=2

```
#include "mv.h"

inline void mv_bk2_unrl2(int M, int N, double alpha, double *A, double *B, double *C, int lda)
{
    register int i, j, i_lda, lda2, N_2, M_2, N_2_1, N_rest;
    register double sum_0, b0, *A_work_0;
    register double sum_1, b1, *A_work_1;

    M_2 = M − M % 2;
    N_rest = N % 2;                                                             10
    N_2 = N − N_rest;
    N_2_1 = N_2 − 2;
    lda2 = lda * 2;

    A_work_0 = A;
    A_work_1 = A + lda;
    i = 0;
    if(N_2){
        for(;i<N_2_1;i+=2){
            sum_0 = 0;                                                          20
            sum_1 = 0;
            for(j=0;j<M_2;j+=2){
                b0 = B[j];
                b1 = B[j+1];
                sum_0 += A_work_0[j] * b0;
```

```
                sum_1 += A_work_1[j] * b0;

                sum_0 += A_work_0[j+1] * b1;
                sum_1 += A_work_1[j+1] * b1;
                                                                        30
        }
        for(;j<M;j++){
                double b0 = B[j];
                sum_0 += A_work_0[j] * b0;
                sum_1 += A_work_1[j] * b0;
        }
        A_work_0 += lda2;
        A_work_1 += lda2;
        C[i] = sum_0;
        C[i+1] = sum_1;                                                 40
}
sum_0 = 0;
sum_1 = 0;
for(j=0;j<M_2;j+=2){
        double b0 = B[j];
        double b1 = B[j+1];
        sum_0 += A_work_0[j] * b0;
        sum_1 += A_work_1[j] * b0;

        sum_0 += A_work_0[j+1] * b1;                                    50
        sum_1 += A_work_1[j+1] * b1;

}
for(;j<M;j++){
        double b0 = B[j];
        sum_0 += A_work_0[j] * b0;
        sum_1 += A_work_1[j] * b0;
}
C[i] = sum_0;
C[i+1] = sum_1;                                                         60

switch(N_rest)
{
        case 0: return;
        case 1:
                A_work_0 += lda2;
                sum_0 = 0.;
                i += 2;
                for(j=0;j<M;j++){
                        sum_0 += A_work_0[j]*B[j];                      70
                }
```

```
                        C[i] = sum_0;
                        return;
                }
        }
        switch(N_rest)
        {
                case 0: return;
                case 1:
                        sum_0 = 0.;
                        for(j=0;j<M;j++){
                                sum_0 += A_work_0[j]*B[j];
                        }
                        C[i] = sum_0;
                        return;
        }
}
```

3.5 Wrapper doitgen

```
#include "f2c.h"

/*$ATLAS ROUTINE DOITGEN_REF */
/*$ATLAS SIZE 1:31:1 */
/*$ATLAS ARG IA1            IN     int     1       */
/*$ATLAS ARG IA2            IN     int     $size   */
/*$ATLAS ARG A[IA2][IA2][IA2] INOUT double $rand */
/*$ATLAS ARG X[IA2][IA2]    IN     double  $rand   */
/*$ATLAS ARG LDX            IN     int     $size   */
/*$ATLAS ARG NP             IN     int     $size   */
/*$ATLAS ARG NQ             IN     int     $size   */
/*$ATLAS ARG NR             IN     int     $size   */
/*$ATLAS ARG NS             IN     int     $size   */

extern int doitgen_ref__(double *a, int *ia1, int *ia2,
        int *ia3, double *x, int *ldx, int *np, int *nq,
        int *nr, int *ns);

/* Subroutine */ int doitgen_ref(int ia1, int ia2,
        double *a, double *x, int ldx, int np, int nq,
        int nr, int ns)
{
    int ia3=ia2*ia2, mu=1, xvt_dim1=ia2, xvt_dim2=ia2;
    double *xvt;
    int x_offset = 1 + xvt_dim1 * (1 + xvt_dim2);

    x -= x_offset;
    xvt=&x[(mu * xvt_dim2 + 1) * xvt_dim1 + 1];
```

```
    doitgen_ref__(a, &ia1, &ia2, &ia3, xvt, &ldx, &np, &nq, &nr, &ns);
                                                                        30

    return 0;
} /* doitgen_ref_wrap */
```

4 Conclusion

We have demonstrated an effective empirical tuning strategy for optimizing the `doitgen` computational kernel code. With less effort than it would take to tune the code by hand, we have achieved better performance than both the hand-tuned version and the version generated by a general-purpose optimizing compiler. Our future work will focus on writing specialized code generators for other MADNESS kernels and on collaborating with the MADNESS developers to incorporate our approach into their code development process.

References

[1] R.J. Harrison, I. Fann G, T. Yanai, and G. Beylkin. Multiresolution quantum chemistry in multiwavelet bases. In *Proc. International Conference on Computational Science (ICCS 2003)*, volume 2657-2660, pages 103–110, Melbourne, Australia, 2003. Springer-Verlag Lecture Notes in Computer Science.