Techniques for Optimizing FEM/MoM Codes

Y. Ji, T. H. Hubing, and H. Wang
Electromagnetic Compatibility Laboratory
Department of Electrical & Computer Engineering
University of Missouri-Rolla
Rolla, MO 65401
yji@umr.edu, hubing@umr.edu, haoh@umr.edu

***Abstract* - This paper presents techniques for optimizing the performance of FEM/MoM codes. The process of computing MoM matrices can be accelerated by choosing the order of Gaussian Quadrature calculations adaptively and taking advantage of the symmetric nature of the impedance (or admittance) matrix. Optimizing loop index order and buffering loop indices can significantly improve the performance of matrix operations. Unrolling techniques can be utilized to exploit the power of high-speed cache memory and reduce data access time. Preconditioning techniques can enhance the spectral properties of the FEM/MoM matrix equations and dramatically improve the convergence rate of iterative solvers.**

***Index Terms* - Hybrid FEM/MoM, code performance, loop index order, unrolling, buffering, preconditioned iterative solvers.**

I. Introduction

The hybrid FEM/MoM method combines the finite element method (FEM) and the method of moments (MoM) to model inhomogeneous unbounded problems. These two methods are coupled by enforcing field continuity on the boundary that separates the FEM and MoM regions. Figure 1 shows the structure of EMAP5, a FEM/MoM code developed at University of Missouri-Rolla [1,2]. Table 1 lists the computation time required for EMAP5 to model a sample geometry with 579 FEM unknowns and 1,155 MoM unknowns. The performance bottlenecks are computing the MoM matrices, assembling and solving the final matrix equation. The time spent on other tasks such as reading the input file and generating output is minor, thus is not listed in Table 1.

This paper presents techniques that can be used to optimize FEM/MoM codes. Section II discusses how to optimize routines. Section III describes how to optimize the computation of the MoM matrices. Section IV shows that adjusting loop-index order, buffering loop indices, and unrolling techniques can significantly improve the code efficiency. Section V presents preconditioning techniques that can greatly improve the efficiency of iterative solvers. Finally, conclusions are drawn in Section VI. Sample problems analyzed using the EMAP5 code running on a Sun workstation are used to illustrate the improvements in run time that can be expected.

II. Optimizing Routines

Numerical codes are usually composed of routines (modules, functions). The length of a routine is not only a maintenance concern, but also an efficiency concern. A very short routine is not efficient because calling a routine involves expensive context switching operations (save the status of the current routine, then load another routine). On the other hand, a long routine is not efficient if it does not fit in the code cache. Therefore, it is best to divide codes into proper-size routines. The code length of each routine should not exceed 200 lines, excluding comments and blank lines [3]. The most error-prone routines are those larger than 500 lines of code [4].

Crucial routines must be examined, re-written and carefully tested to improve code performance. In FEM/MoM codes, a large portion of computation time is spent on matrix operations. Matrix operations should be put in separate routines that are optimized for the hardware they will run on. Although many compilers support code optimization, well-written codes can be further optimized as shown in latter sections.

III. Optimizing Computation of the MoM Matrices

The Gaussian Quadrature technique is generally used to numerically evaluate integrands over two surface patches in MoM. The choice of Gaussian Quadrature order (1-point, 3-point, or 7-point) can be determined adaptively depending on the distance between two patches. For example, suppose R_{cp} is the distance between the centroids of the triangle pair in Figure 2, and *MaxEdgeLen* is the maximum edge length of the source (or observing) triangular patch. The following procedure can be used to adaptively determine the Gaussian Quadrature order of the source (or observing) triangular patch,

```
if ( Rcp/MaxEdgeLen > 10 )
    QuadType = 1;        /* use 1-point Gaussian Quadrature */
else  if ( Rcp/MaxEdgeLen >5 )
    QuadType = 3;        /* use 3-point Gaussian Quadrature */
else  QuadType = 7;      /* use 7-point Gaussian Quadrature */
```

Furthermore, the impedance (or admittance) matrix in MoM is symmetric. Only half of the entries need to be computed.

IV. Optimizing Matrix Operations

The structure of computer memory is typically organized in a hierarchy [5] as shown in Figure 3. The higher-level media supports faster access but is more expensive. Performance data reported in this paper is measured on a Sun Ultra-II 250MHz workstation in units of MFlops (million floating-point operations per second). Figure 4 shows the code performance versus the size of problem being solved. The code performs best when data fits in the data cache. The code performance drops when data is stored in the primary cache and the first-level memory (DRAM/SDRAM). The code performance is 75% higher when data is stored in primary cache than in DRAM. In typical numerical applications, most data is stored in DRAM. The following two sub-sections present how to improve the performance of matrix operations by utilizing cache memory efficiently.

A. Optimizing inner-product operations in the complex bi-conjugate gradient method

Iterative solvers are widely used to solve large-scale matrix equations. The complex bi-conjugate gradient method (CBCG) introduced by Jacobs [6] can be used to solve the final complex matrix equation generated by FEM/MoM, which is of the form

$$[A][x]=[b] \qquad (1)$$

where [A] is an N×N complex matrix, and [b] and [x] are N×1 complex column vectors. Both [A] and [b] are known while [x] is unknown. As Table 1 indicates, solving the final matrix equation is one of the bottlenecks of the EMAP5 code performance.

The most time-consuming portion of CBCG are the operations computing the product of Ap_k and $A^H \underline{p_k}$, where [A] is the coefficient matrix, A^H is the Hermitian of matrix A, p_k is the direction vector, and $\underline{p_k}$ is the bi-direction vector. Both of these operations involve N^2 complex number multiplications. All other operations in CBCG only require O(N) complex multiplications. A natural implementation of $[C]_N = [A]_{N,N^*}[P]_N$ in the C programming language is shown below,

```
for ( i=0; i<N; i++ )
    for ( j=0; j<N; j++ )
```

```
            C[i] = A[i][j]*P[j] ;
```

The above loop index order is defined as (*i*, *j*). This implementation works fine. A natural implementation of $[C]_N = [A]^H_{N,N*}[P]_N$ is as follows,

```
for (  i=0;  i<N;  i++ )
    for ( j=0;  j<N;  j++ )
        C[i] = Conjugate( A[j][i] ) *P[j] ;
```

The above implementation is inefficient because data in the C language is row-majored (row-indexed). Whenever index *j* increases by one, the address of *A[j][i]* will jump N address spaces. If the next *A[j][i]* is not in the data cache, a cache miss occurs. The next *A[j][i]* and its neighbor units (a cache block) will be swapped from DRAM into the data cache. Frequent cache misses can significantly degrade the code performance. To better utilize the data-caching feature, it is desirable to access data in a cache block continuously to reduce the number of cache misses. For the above example, the loop index can be changed from *(i, j)* to *(j, i)* (the loop body is the same) as shown below,

```
for ( j=0;  j<N;  j++ )
    for ( i=0;  i<N;  i++ )
        C[i] = Conjugate( A[j][i] ) *P[j] ;
```

As shown in Table 2, this change of loop index order increases the performance from 1.52 MFlops to 2.94 MFlops, (i.e. by 93%).

Buffering the loop indices *i* and *j* in registers can improve the code performance even further. The following code (in the C language) stores two integers in registers,

```
register int i, j;
```

This ensures that the loop indices *i* and *j* will not be swapped out from cache memory to DRAM. Many C compilers do not optimize codes by buffering temporary variables. Programmers must decide how to buffer variables by themselves. It can be difficult to decide how to buffer temporary variables in a large routine because computers have a limited number of registers. Therefore, crucial codes (in terms of computational cost) should be put into separate routines of proper size and tuned to optimize performance. As shown in Table 2, buffering loop indices *i* and *j* improves the code performance by 90% (from 2.94 MFlops to 5.32MFlops) without compiler optimization and 11% (from 10.42MFlops to 11.6 MFlops) with compiler optimization. Choosing the proper loop index and buffering loop indices improves the overall code performance by a factor of 2.5 without compiler optimization and 2.6 with compiler optimization.

B. Optimizing matrix multiplication

Hybrid FEM/MoM codes perform many matrix multiplications [1]. This process is numerically intensive, involving $O(N^3)$ floating point operations. Choosing the proper loop index order is critical to the code performance. For typical matrix multiplication $[A]_{N,N} = [B]_{N,N}*[C]_{N,N}$, loop index (*i*, *j*, *k*) refers to the following loop order,

```
for ( i=0; i<N; i++)
    for(  j=0;  j<N;  j++)
        for( k=0; k<N; k++)
            A[i][j] += B[i][k]*C[k][j];
```

Table 3 shows the code performance of all six possible combinations of loop indices. It is clear that *(k, i, j)* and *(i, k, j)* are the best loop index orders. Without compiler optimization, the best/worst performance ratio is 1.88. With compiler optimization, the best/worst performance ratio is 5.0. This indicates that codes that are more efficient can be better optimized by compilers.

Unrolling techniques [5] can efficiently utilize cache blocks to reduce data access time. The basic idea of unrolling is to reduce the number of iterative loops, but add statements to loop bodies to do the missing iterations. For the above matrix multiplication codes, a four-way unrolling implementation is shown below,

```
for ( i=0; i<N; i++ )
   for ( k=0; k<N; k+=4 )
       for( j=0; j<N; j++ ) {
           A[i][j] += B[i][k]*C[k][j];
           A[i][j] += B[i][k+1]*C[k+1][j];
           A[i][j] += B[i][k+2]*C[k+2][j];
           A[i][j] += B[i][k+3]*C[k+3][j];
       }
```

Table 4 shows the code performance improvement obtained by using this 4-way unrolling technique and buffering loop indices. The time required for matrix multiplication is reduced by 68%.

V. Preconditioned Iterative Solvers

The coefficient matrices generated by the application of hybrid FEM/MOM codes to complex geometries often have very large condition numbers. It is difficult to apply iterative solvers to these matrix equations. However, in general, a matrix equation can be transformed into another matrix equation so that the new matrix equation has the same solution as the original one, but has better spectral properties. For instance, both sides of Equation (1) can be multiplied by a square matrix P^{-1},

$$P^{-1}Ax = P^{-1}b \tag{2}$$

where P has the following properties,

(1) $K(P^{-1}A) \ll K(A)$

(2) $\det(P^{-1}A) \neq 0$

(3) it is inexpensive to solve $Px = b$

and where $K(\bullet)$ and $\det(\bullet)$ are the condition number and the determinant of a matrix, respectively. Such a matrix P is called a *preconditioner*. This technique is called *preconditioning*. Condition (1) assures favorable spectral properties for the new linear system. Condition (2) guarantees that the new system, Equation (2), has the same non-trivial solution as Equation (1). Condition (3) is essential to ensure the efficiency of preconditioned iterative solvers. In preconditioned iterative algorithms, it is not necessary to solve P^{-1} explicitly. Instead, a linear system of the form $Px = b$ is solved at each step.

A new equation solver has been developed for EMAP5 based on the preconditioned Bi-Conjugated Gradient Stabilized method (BiCGSTAB) [2]. Two printed circuit board (PCB) problems and one scattering problem have been chosen to test the new solver. As shown in Table 5, the new solver has improved the efficiency of the matrix equation solver by a factor of 48 in Problem 1 (from 206.10 sec to 3.55 sec), by a factor of 150 in Problem 2 (from 6,037.90 sec to 40.10 sec), and by a factor of 48 in Problem 3 (from 386.77 sec to 8.10 sec). The overall improvements for the three problems are 221%, 783% and 636%, respectively. This shows that preconditioning techniques can dramatically improve the convergence rate of iterative solvers and the overall performance of FEM/MoM codes.

VI. Conclusions

This paper presents techniques for optimizing FEM/MoM codes. Numerical codes should be divided into routines of appropriate length for the sake of maintenance and performance. Crucial routines should be well tuned to the hardware they will run on. The performance bottlenecks of FEM/MoM codes are computing the MoM matrices and assembling and solving the final matrix equation. The process of computing the MoM matrices can be accelerated by using an adaptive Gaussian Quadrature technique and by taking advantage of the symmetric nature of the impedance (admittance) matrix. The performance of matrix operations can be significantly improved by optimizing loop index order and buffering loop indices. Unrolling techniques can be utilized to reduce data access time. The hybrid FEM/MoM method often generates matrix equations with very large condition numbers. It can be difficult to apply iterative solvers to these matrix equations directly. Preconditioning techniques can be used to improve the spectral properties of matrix equations and dramatically accelerate the convergence rate of iterative solvers. Together, these optimization techniques can greatly reduce the overall run time of hybrid FEM/MoM codes analyzing large problems.

References:

[1] M. Ali, T. H. Hubing, and J. L. Drewniak, "A hybrid FEM/MoM technique for electromagnetic scattering and radiation from dielectric objects with attached wires," *IEEE Trans. on Electromagnetic Compatibility*, vol. 39, pp. 304-314, Nov. 1997.

[2] Y. Ji, M. Ali, and T. H. Hubing, "EMC applications of the EMAP5 hybrid FEM/MoM code," *Proceedings of the IEEE 1998 International Symposium on Electromagnetic Compatibility*, vol. 1, pp. 177-181, Denver, Colorado, Aug. 1998.

[3] S. McConnell, *Code Complete*, Washington: Microsoft Press, 1993.

[4] C. Jones, *Programming Productivity*, New York: McGraw-Hill, 1986.

[5] D. Loshin, *High Performance Computing Demystified*, Boston: Academic Press, 1994.

[6] D. A. H. Jacobs, "A generalization of the conjugate gradient method to solve complex system," *IMA J. Numerical Anal.* vol. 6, pp 447-452, 1986.

[7] R. Barrett, M. Berry, T. F. Chan, F. Demmel, J. M. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, Philadelphia: SIAM, 1994.

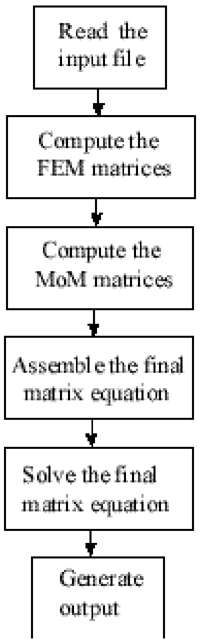

Figure 1. The EMAP5 code structure.

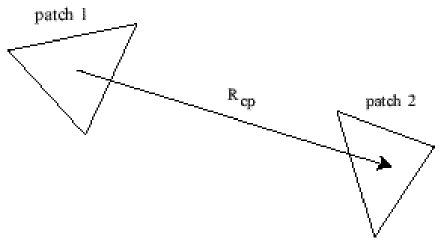

Figure 2. Center-to-center distance between two surface patches.

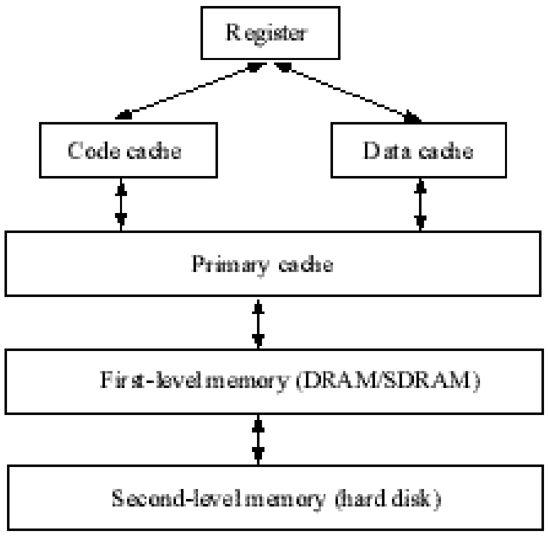
Register
Code cache
Data cache
Primary cache
First-level memory (DRAM/SDRAM)
Second-level memory (hard disk)

Figure 3. Memory hierarchies.

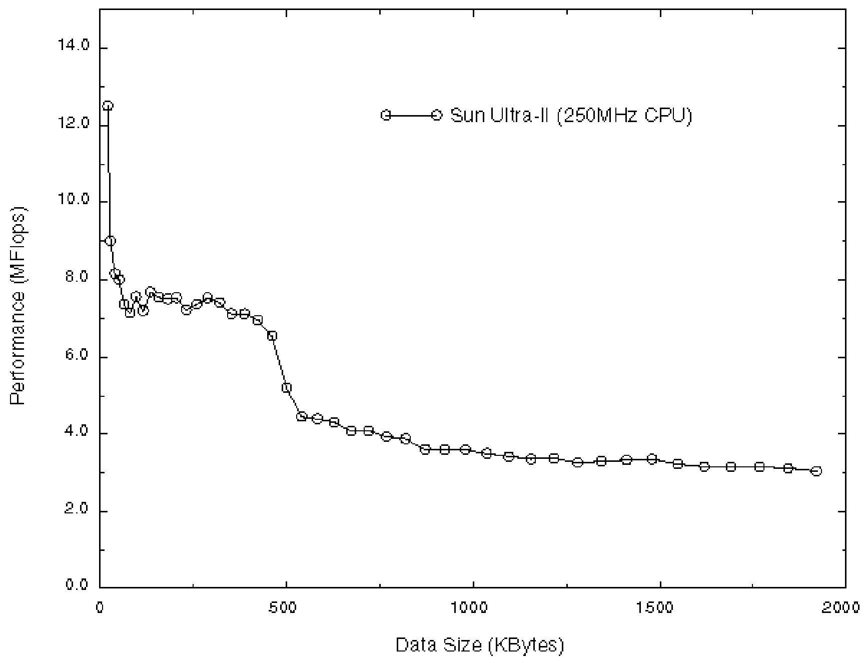

Figure 4. Code performance verses data size.

Table 1. Time required to model a problem with 579 FEM and 1,155 MoM unknowns.

Total time (min.)	Compute the FEM matrices (min.)	Compute the MoM matrices (min.)	Assemble the final matrix equation (min.)	Solve the final matrix equation (min.)
174	0.1	20	92	63

Table 2. Performance of $[C]_N = [A]^H_{N,N} * [P]_N$ versus loop index permutation when N= 400.

	Loop index	Use registers to buffer variables (Yes/No)	Performance (MFlops)	
			No compiler optimization	Using compiler optimization
Case 1	(i, k)	No	1.52	3.23
Case 2	(k, i)	No	2.94	10.42
Case 3	(k, i)	Yes	5.32	11.60
Improvement (best/worst)			250%	260%

Table 3. Performance of $[A]_{N,N} = [B]_{N,N}*[C]_{N,N}$ versus loop index permutation when N=400.

	Loop index	Performance (MFlops)	
		No compiler optimization	Using compiler optimization
Case 1	(k, j, i)	0.90	1.57
Case 2	(j, k, i)	0.95	3.64
Case 3	(j, i, k)	1.36	3.64
Case 4	(i, j, k)	1.31	3.71
Case 5	(k, i, j)	1.78	7.72
Case 6	(i, k, j)	1.79	7.72
Improvement (best/worst)		188%	500%

Table 4. Performance of $[A]_{N,N} = [B]_{N,N}*[C]_{N,N}$ using unrolling techniques when N=400.

	Use registers to buffer variables	Performance (MFlops)	
		No compiler optimization	Using compiler optimization
Case 1	No	1.31	3.71
Case 2	Yes	1.85	3.70
Unrolling	Yes	2.20	6.26

Table 5. Time required to solve three sample problems.

	Compute FEM and MoM matrices (sec)	Forming the final matrix equation (sec)	Original		Preconditioned		Overall improvement (%)
			Solving the final equation (sec)	Total (sec)	Solving the final equation (sec)	Total (sec)	
Problem 1	48.00	40.23	206.10	294.22	3.55	91.78	221%
Problem 2	287.20	438.60	6,037.90	6,763.7	40.10	765.90	783%
Problem 3	40.12	11.33	386.77	438.22	8.10	59.55	636%