SIGCSE **inroads** submission

Program Optimization: Enforcement of Local Access and Array Access *via* Pointers

Timothy J. Rolfe

Computer Science Department
Eastern Washington University
202 Computer Sciences Building
Cheney, WA 99004-2412 USA
Timothy.Rolfe@mail.ewu.edu
http://penguin.ewu.edu/~trolfe/

Abstract

Matrix multiplication in linear algebra provides a useful problem through which one can investigate optimizations based on local access to memory rather than scattered access, and on the use of pointers in places of array subscripting. Benchmarking results favor a pointer-based implementation with a reordering of the three loops in the definition.

Algorithms

Given matrices A[N_1][N_2], B[N_2][N_3], and C[N_1][N_3], matrix multiplication is defined as the generation of the matrix C from the matrices A and B: for each cell c_{ij} in C,

$$c_{ij} = \sum_{k=0}^{N_2-1} a_{ik} \cdot b_{kj}$$

The typical implementation is as a simple triple loop.

```
// Straight computation (i.e., ALL of C[i][j] at once)
void MatMult00(double A[n1][n2], double B[n2][n3],
               double C[n1][n3])
{  for ( int i = 0; i < n1; i++ )
      for ( int j = 0; j < n3; j++ )
      {  double sigma = 0;

         for ( int k = 0; k < n2; k++ )
         {  sigma += A[i][k] * B[k][j]; }
         C[i][j] = sigma;
      }
}
```

The C language uses row-major storage, so that the rows of A are accessed sequentially, but the columns in B require scattered access within the matrix, stepping by N_2, the row dimension of B. As noted, each cell in C is completely calculated by the innermost loop. Also, the cells of C are also accessed sequentially, given that the outermost loop drives the row subscript in C.

For small matrix dimensions, this non-local access in B may not pose a problem, but for large dimensions (sufficiently large that all of the data cannot reside in the program's working set), the point comes at which each access to `B[k][j]` causes a page fault.

There are two possible optimizations, each with its own cost.

The first optimization is to re-order the loops to enforce local access in all of the matrices. The penalty is that the cells of C are computed in a scattered fashion, requiring explicit zeroing of each row in C before the cells of that row are computed. The cells of C, though, *are* accessed in storage order, as are the cells of B.

```
// Scattered computation (i.e., row-wise in B and C)
void MatMult01(double A[n1][n2], double B[n2][n3],
               double C[n1][n3])
{  int i, j, k;

   for ( i = 0; i < n1; i++ )
   {  double Aik = A[i][0];
      for ( j = 0; j < n3; j++ )
         C[i][j] = Aik * B[0][j];
      for ( k = 1; k < n2; k++ )
      {  Aik = A[i][k];
         for ( j = 0; j < n3; j++ )
            C[i][j] += Aik * B[k][j];
      }
   }
}
```

This rearrangement is discussed in Alan Jennings' *Matrix Computation for Engineers and Scientists*.[1] Jennings notes it as a strategy that allows all operations involving A[i][k] to be performed together — and all skipped together on a zero entry in the A matrix. Perhaps reflecting the state of computers twenty-five years ago, he states:

> If the entire matrix multiplication can be performed within the available main store then there will be little difference in efficiency between the two multiplication strategies. The standard procedure will probably be preferred for fully populated matrices.[2]

Jennins does, however, recognize the potential for thrashing:

> In cases where block storage transfers between the main store and the backing store have to be carried out during matrix multiplication, either explicitly or implicitly, the choice of multiplication scheme may affect execution times very substantially.[3]

The second optimization is to require that the B matrix be transposed, for an $O(n^2)$ set-up penalty to generate the transpose matrix Bt, but the cells of Bt are accessed sequentially and the cells of C are completely computed within the innermost loop.

```
// Straight computation (i.e., ALL of C[i][j] at once)
// except that B has been transposed for local access
void MatMult02(double A[n1][n2], double Bt[n3][n2],
               double C[n1][n3])
{  for ( int i = 0; i < n1; i++ )
      for ( int j = 0; j < n3; j++ )
      {  double sigma = 0;

         for ( int k = 0; k < n2; k++ )
         {  sigma += A[i][k] * Bt[j][k]; }
         C[i][j] = sigma;
      }
}
```

In the benchmarking program used for the above three implementation, matrices were passed as pointers to one-dimensional arrays along with the three dimensions N_1, N_2 and N_3. The standard mapping was then used from a two-dimensional array onto a one-dimensional space. (Explicit measurement with 500x500 matrices showed that the

times needed for the above three algorithms were the same in the explicit two-dimensional code as in the one-dimensional code.) Thus the first algorithm was in fact coded thus:

```
void MatMult00(double *A, double *B, double *C,
               int n1, int n2, int n3)
{  for ( int i = 0; i < n1; i++ )
      for ( int j = 0; j < n3; j++ )
      {  double sigma = 0;

         for ( int k = 0; k < n2; k++ )
         {  sigma += A[i*n2 + k] * B[k*n3 + j]; }
         C[i*n3 + j] = sigma;
      }
}
```

This makes explicit the fact that subscripting in a two-dimensional array requires integer multiplication as well as addition. Replacing subscripting with explicit pointer access allows the elimination of these multiplications. For instance, the pointer into the A matrix can be initialized before the inner-most loop above and then advanced by one for each pass of that loop, while the pointer into the B matrix, after initialization, is advanced by `n3` for each pass. The resulting code follows:

```
void MatMult10(double *A, double *B, double *C,
               int n1, int n2, int n3)
{  double *pC = C;

   for ( int i = 0; i < n1; i++ )
      for ( int j = 0; j < n3; j++ )
      {  double sigma = 0;
         double *pA = A + i*n2,
                *pB = B + j;

         for ( int k = 0; k < n2; k++ )
         {  sigma += *pA * *pB;
            pA++; pB += n3;
         }
         *pC++ = sigma;
      }
}
```

This eliminates $O(n^3)$ integer multiplications, while retaining only $O(n^2)$ multiplications for the initialization of the pointer into A; in the subscripting code that was the expense for the access into C.

Similar optimizations can be made to the two local-access optimizations.

Benchmark Results

There are thus six procedures: three different algorithms implemented in two different fashions.

	Scattered B access	Sequential B access	B passed as transpose
Subscripts	`MatMult00`	`MatMult01`	`MatMult02`
Pointers	`MatMult10`	`MatMult11`	`MatMult12`

The benchmark program allows specification of the three matrix dimensions N_1, N_2 and N_3, as well as the number of random matrices to be multiplied to get average execution times. It was run on 2.0 GHz Dell computers under Linux after being compiled by gcc with full optimization specified. The computers were otherwise unloaded. Specifically, N_1 and N_2 were held at 100, while N_3 went through a range of values. For each value of N_3, 1000

random matrix pairs were generated and multiplied. The expectation is that time required should increase linearly, with a change of slope at the point where page thrashing begins for MatMult00 and MatMult01.

The predicted behavior was in fact observed. Both MatMult00 and MatMult01 show a distinct change in slope at the point where B becomes larger than 100x300. In addition, the pointer implementation of all of the algorithms runs significantly faster than the array subscripting implementation. Figure One shows the results up to $N_3 = 500$, with significantly more points captured in the range [300..500] than in the initial range [50..300].

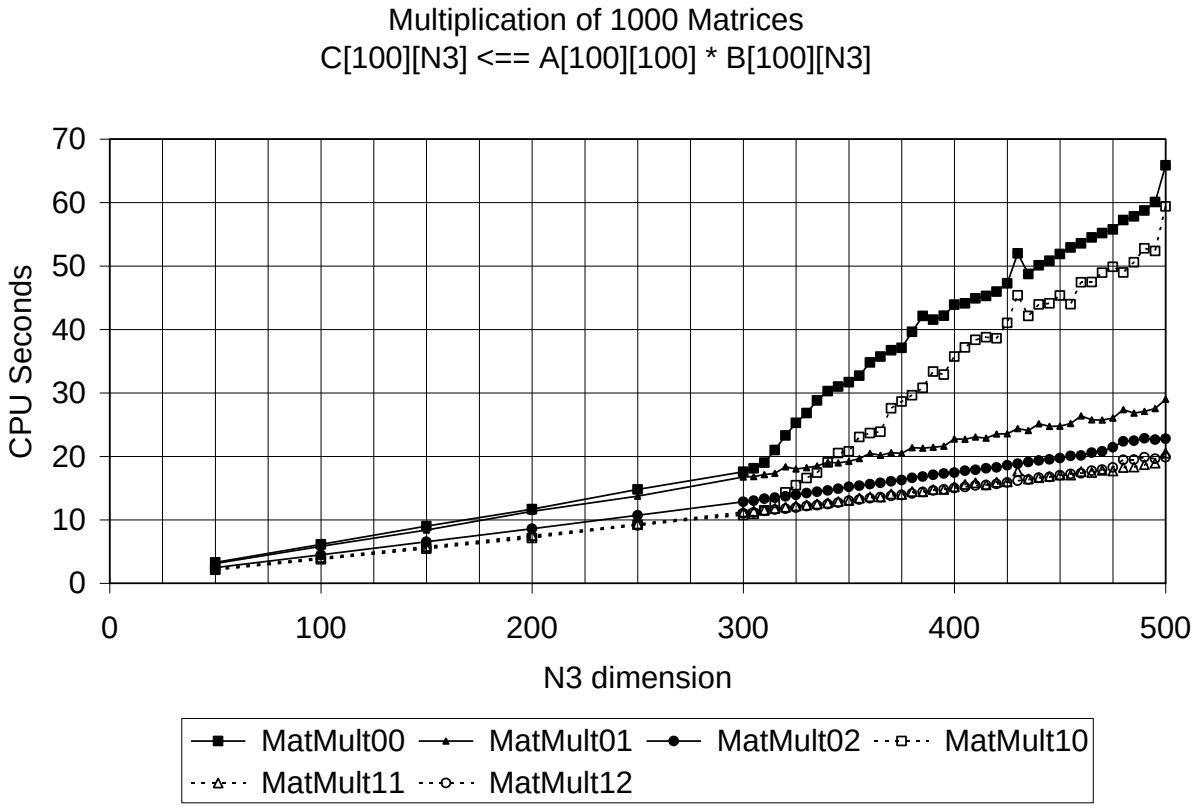

Figure One: Benchmark Results, All Algorithms

The sharpness of the change in slope does seem surprising. For instance, one might expect to reach a point where every *other* access within B causes a page fault, followed by the point at which every three accesses within B cause two page faults, etc. This suggests that something more is happening past B[100][300] than just page faults.

Extending the benchmark program to include the range [500..1000], Figure Two shows the behavior of the two local-access implementations.

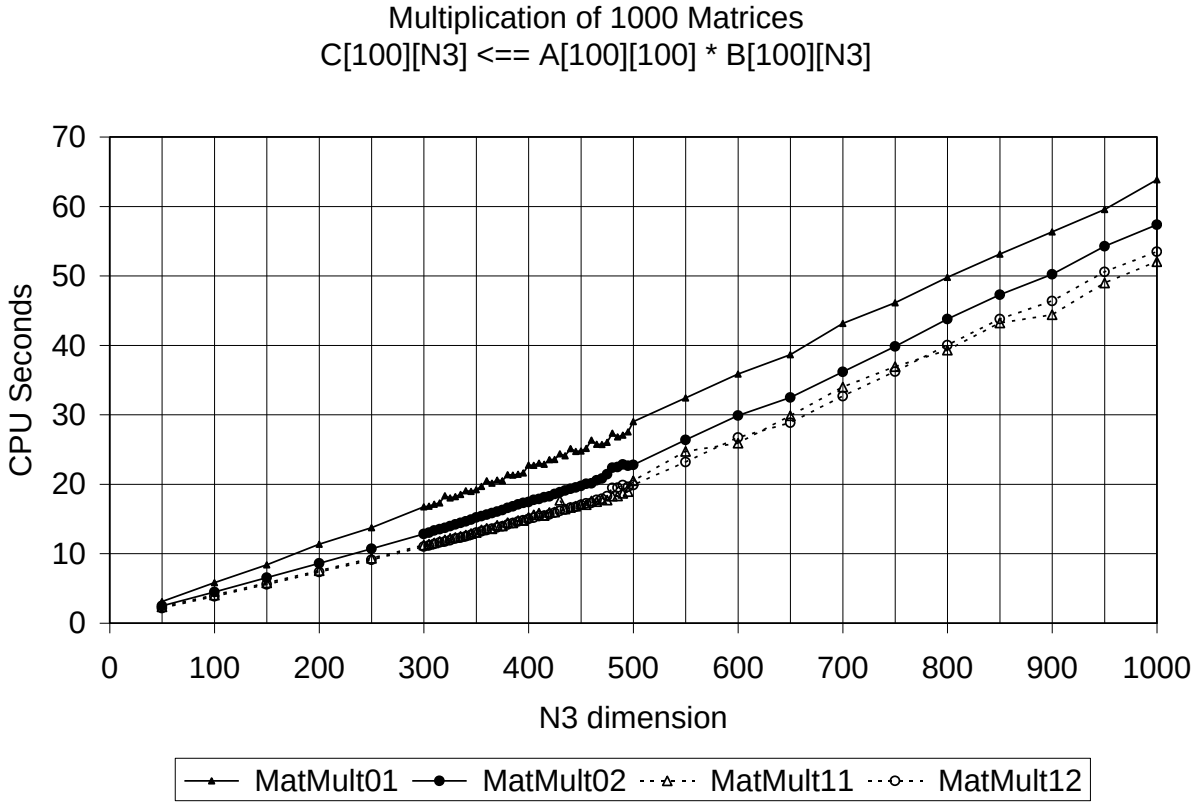

Figure Two: Benchmark Results, Excluding Thrashing Algorithms

The benchmark program is available from the author's web site, along with the Excel workbook that generated the above two figures: http://penguin.ewu.edu/~trolfe/MatMult/

Summary

For matrices of sufficient size, straight implementation of matrix multiplication taken from its definition causes page thrashing, significantly slowing the computation. Avoiding this thrashing can markedly speed up computation. In addition, one can speed the code further by avoiding matrix subscripting and accessing data explicitly through pointers.

The pointer implementations of the two non-thrashing implementations of matrix multiplication require approximately the same time. Consequently the less intrusive algorithm (reordering the loops for the calculation) is probably the better choice since it does not require changing the procedure interface.

References

[1] Alan Jenings, *Matrix Computation for Engineers and Scientists* (John Wiley & Sons, 1977), pp. 77-78.
[2] Ibid., p. 78.
[3] Ibid., pp. 78-79.