Performance Optimization Tutorial

Labs

Exercise 1

These exercises are intended to provide you with a general feeling about some of the issues involved in writing high performance code, both on a single processor and on a parallel machine.

To start, login in to the Origin 2000, pagh.wes.hpc.mil

Copy the exercises over to your area.

% cp -rf ~london/hpcug_lab .

% cd hpcug_lab/matmul1-f

The first exercise will use a simple matrix-matrix inner product multiplication to demonstrate various optimization techniques.

Matrix-Matrix Multiplication - Simple Optimization by Cache Reuse

Purpose: This exercise is intended to show how the reuse of data that has been loaded into cache by some previous instruction can save time and thus increase the performance of your code.

Information: Perform the matrix multiplication **A = A + B * C** using the code segment below as a template and ordering the ijk loops in to the following orders (**ijk**, **jki**, **kij**, and **kji**). In the file **matmul.f**, one ordering has been provided for you (**ijk**), as well as a high performance BLAS routine **dgemm** which does double precision general matrix multiplication. **dgemm** and other routines can be obtained from Netlib.

The variables in the matmul routine (reproduced on the next page) are chosen for compatibility with the BLAS routines and have the following meanings: the variables ii, jj, kk, reflect the sizes of the matrix A (ii by jj), B(ii by kk) and C(kk by jj); the variables lda, ldb and ldc are the leading dimensions of each of those matrices and reflect the total size of the allocated matrix, not just the part of the matrix used.

```
subroutine  ijk ( A, ii, jj, lda, B, kk, ldb, C, ldc )
double precision A(lda, *), B(ldb, *), C(ldc, *)
integer = i, j, k
do i = 1, ii
      do j = 1, jj
            do k = 1, kk
                  A(i,j) = A(i,j) + B(i,k) * C(k,j)
            enddo
      enddo
enddo
return
end
```

Instructions: For this exercise, use the files provided in the directory **matmul1-f**. You will need to work on the file **matmul.f**. Again, if you are extremely stressed, consult **matmul.f.ANS**, where there is one possible solution.

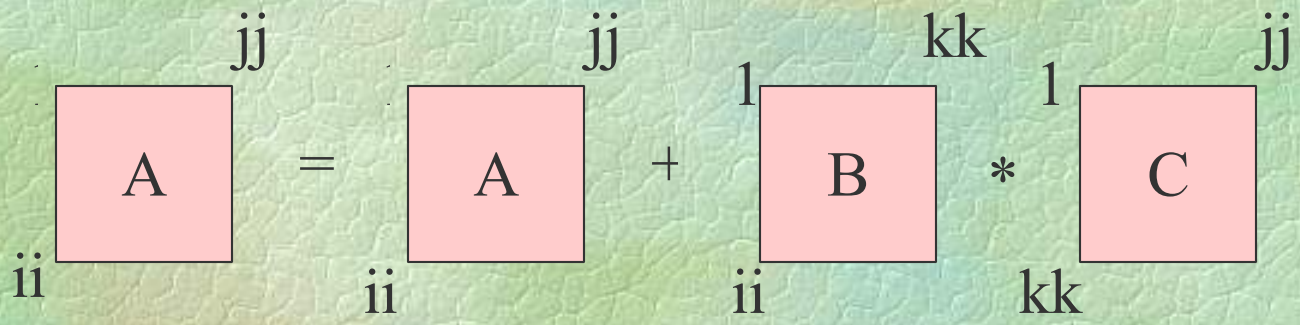
(a) Compile the code: **make matmul** and run the code, recording the Mflops values in a table like the one on the next page.

(b) Edit **matmul.f** and alter the orderings of the loops, **make**, run and repeat for the various loop orderings. Complete a table like the one below, which shows the orderings of the loops for each set of randomly generated matrices **A**, **B**, and **C**, or order Order. Enter the Mflops achieved while computing A = A + B * C using each of the four different ways to perform matrix multiplication..

Order	dgemm	ijk	jki	kij	kji
50					
100					
150					
200					
250					
300					

- Which loop ordering achieved the best performance and why?
- When you are done with this exercise, please **make clean** to remove the executable and object files.
- Note: **dgemm** only occupies one column of the table because its values should be very similar each time the program is executed.

Explanations: To explain the reason for these timing and performance figures, the multiplication operation needs to be examined more closely. The matrices are drawn below, with the dimensions of rows and columns indicated. The ii indicates the size of the dimension which is traveled when we do the i loop, the jj indicates the dimension traveled when we do the j loop and the kk indicates the dimension traveled when we do the k loop.



The pairs of routines with the same innermost loop (e.g. jki and kji) should have similar results. Let's look at jki and kji again. These two routines achieve the best performance, and have the i loop as the innermost loop. Looking at the diagram, this corresponds to traveling down the columns of 2 (A and B) of the 3 matrices that are used in the calculations. Since in Fortran, matrices are stored in memory column by column, going down a column simply means using the next contiguous data item, which usually will already be in the cache. Most of the data for the i loop should already be in the cache for both the A and B matrices when it is needed.

Some improvements to the simple loops approach to matrix multiplication which are implemented by dgemm include loop unrolling (some of the innermost loops are expanded so that not so many branch instructions are necessary), and blocking (data is used as much as possible while it is in cache). These methods will be explored later in Exercise 2.

Exercise 2 Matrix-Matrix Multiplication Optimization using Blocking and Unrolling of Loops

Purpose: This exercise is intended to show how to subdivide data into blocks and unroll loops. Subdividing data into blocks helps them to fit into cache memory better. Unrolling loops decreases the number of branch instructions. Both of these methods sometimes increase performance. A final example shows how matrix multiplication performance can be improved by combining methods of subdividing data into blocks, unrolling loops, and using temporary variables and controlled access patterns.

Information: The matrix multiplication **A = A + B * C** can be executed using the simple code segment below. This loop ordering **kji** should correspond to one of the best access ordering the six possible simple **i, j, k** style loops.

```
subroutine kji  ( A, ii, jj, lda, B, kk, ldb, C, ldc )
double precision A( lda, *), B(ldb, *), C(ldc, *)
integer i, j, k
do k = 1, kk
      do j = 1, jj
                  do i = 1, ii
                              A(i,j) = A(i,j) +B(i,k) * C(k,j)
                  enddo
      enddo
enddo
return
enddo
```

However, this is not the best optimization technique. Performance can be improved further by blocking and unrolling the loops. The first optimization will demonstrate the effect of loop unrolling. In the instructions, you will be asked to add code to unroll the j, k, and i loops by two, so that you have, for example, **do j = 1, jj, 2**, and add code to compensate for all the loops that you are skipping, for example, **A (i,j) = A(i,j) + B(i,k) *C(k,j) + B(i,k+1) * C(k+1, j)**. Think of multiplying a 2x2 matrix to figure out the unrolling.

The second optimization will demonstrate the effect of blocking, so that, as much as possible, the blocks that are being handled can be kept completely in cache memory. Thus each loop is broken up into blocks (ib, beginning of an i block, ie, end of an i block) and the variables travel from the beginning of the block to the end of the block for each i,j,k. Use blocks of size 32 to start with, if you wish you can experiment with the size of the block to obtain the optimal size.

The next logical step is to combine these two optimizations into a routine which is both blocked and unrolled and you will be asked to do this.

The final example tries to extract the core of the BLAS **dgemm** matrix-multiply routine. The blocking and unrolling are retained, but the additional trick here is to optimize the innermost loop. Make sure that it only references items in columns and that it does not reference anything that would not be in a column. To that end, B is copied and transposed into the temp matrix **T(k,i) = B(i,k)**. Then multiplying **B(i,k)*C(k,j)** is equivalent to multiplying **T(k,i)*C(k,j)** (notice the k index occurs only in the row). Also, we do not store the result in **A(i,j)=A(i,j)+B(i,k)*C(k,j)** but in a temporary variable **T1=T1+T(k,j)*C(k,j)**. The effect of this is the inner k-loop has no extraneous references. After the inner loop has executed, **A(i,j)** is set to its correct value.

mydgemm:

```
do kb = 1, kk, blk
        ke = min(kb+blk-1,kk)
        do ib = 1, ii, blk
                ie = min(ib+blk-1, ii)
                do i = ib,ie
                        do k = kb, ke
                                T(k-kb+1, i-ib+1) = B(i,k)
                        enddo
                enddo
                do jb = 1, jj, blk
                        je = min(jb+blk-1, jj)
                        do j = jb, je, 2
                                do i = ib, ie, 2
                                        T1 = 0.0d0
                                        T2 = 0.0d0
                                        T3 = 0.0d0
                                        T4 = 0.0d0
                                        do k = kb, ke
                                                T1 = T1 + T (k-kb+1,i-ib+1)*C(k,j)
                                                T2 = T2 + T(k-kb+1, i-ib+2)*C(k,j)
                                                T3 = T3 + T(k-kb+1, i-ib+1)*C(k,j+1)
                                                T4 = T4 +T(k-kb+1, i-ib+2)*C(k,j+1)
                                        enddo
                                        A(i,j) = A(i,j)+T1
                                        A(i+1,j) = A(i+1, j)+T2
                                        A(i,j+1) = A(i, j+1)+T3
                                        A(i+1, j+1) = A(i+1, j+1) +T4
                                enddo
                        enddo
                enddo
        enddo
enddo
```

Instructions: For this exercise, use the files provided in the directory matmul2-f. You will need to edit the file matmul.f. One possible solution has been provided in the file matmul.f.ANS.

•Compile by typing **make matmul** and execute **matmul**, recording the Mflops values returned for **kji, dgemm** and **mydgemm**. You will get some "0.000" values. Those are from areas where you are expected to edit the code and are not doing anything currently.

•Note: In order to speed up your execution, you can comment out each routine after you have finished recording its execution rates. For example, you could comment out the **kji, dgemm** and **mydgemm** routines now and you would not have to wait for them to execute in future runs.

•Edit **matmul.f** and uncomment and correct the routine **kjib** which should be a blocked version of kji (use blocks of size 32). Compile and execute the code, recording the Mflops values.

•Edit **matmul.f** and uncomment and correct the routine **kjiu** which should be an unrolled version of **kji.** Compile and execute the code, recording the Mflops values.

•Which optimizations achieved the best performance?

- Why was this performance achieved? (Review the information about **dgemm** and **mydgemm** for the answer)
- Why is the performance of **dgemm** worse than that of **mydgemm**? (**mydgemm** extracts the core of **dgemm** to make it somewhat simpler to understand. In doing so it throws away the parts of **dgemm** which are generic and applicable to any size matrix. Since **mydgemm** cannot handle arbitrary size matrices it is somewhat faster than **dgemm** but less useful).
- When you are done with this exercise, please **make clean** to remove the executable and object files.

Order	kji	kjib	kjiu	kjibu	dgemm	mydgemm
50						
150						
250						
350						
450						
550						