Introduction To Scalable Systems Assignment 1

By Muttaqi Ahmad Alladin

Objectives: To create the best program for multiplying 1024x1024 double matrices on a single processor core by only using loop interchange and blocking/tiling.

Materials and Methods:

Materials:

1. A 13- inch late 2011 dual-core MacBook Pro with the following machine properties:

Model Name	MacBook Pro
Model Identifier	MacBookPro8,1
Processor Name	Dual-Core Intel Core i5
Processor Speed	2.4 GHz
Number of Processors	1
Total Number of Cores	2
L2 Cache (per Core)	256 KB
L3 Cache	3 MB
Hyper-Threading Technology	Enabled
Memory	4 GB
Boot ROM Version	87.0.0.0.0
SMC Version (system)	1.68f99
Serial Number (system)	C17H89DFDV13
Hardware UUID	8C2E3E26-56E8-51C3-B625-0A008C54886E
L1 cache*	32768 bytes

2. Various standard C libraries like stdio.h, stdlib.h etc were used.
3. gettimeofday() was used to calculate the time taken for execution.
4. rand() was used to initialize the matrices.

*Note: l1 cache was found using the command

```
muttaqialladin@muttaqis-MacBook-Pro ~ % sysctl -a | grep -i l1
hw.l1icachesize: 32768
```

Methods:
The c program justmat.c was created to multiply the matrices (source code appended herewith). Microsoft Visual Studio Code was used to type the program.

Note: The various loops in the source code are inspired by the slides of Prof. Matthew Jacob Thazhuthaveetil and the published work of Bryant and O'Hallaron.

Methodology:
First, matrices MA and MB were initialized using the rand() function. Next, matrices MC and MD were initialized to zero. Thereafter, matrix multiplication was carried out between matrices MA and MB and the result was stored in matric MC. This result in MC will be used to check if the matrix multiplication is accurate in subsequent operations. The following operations were carried out thereafter:

1) Loop Interchange
2) Blocking

Note: the matrix multiplication results from these operations(loop interchange/ blocking) were stored in matrix MD. After validating the matrix multiplication results using MC, the matrix MD was reinitialized to zero before moving the next matrix multiplication.

Loop Interchange:
A typical matrix multiplication loop is shown in Figure 1:

```
for (i=0;i<1024;i++){
    for(j=0;j<1024;j++){
        for(k=0;k<1024;k++){
            MD[i][j]+=MA[i][k]*MB[k][j];

        }
    }
}
```

Figure 1: Generic matrix multiplication

As can be seen from Figure 1, matrix multiplication requires the use of 3 nested loops(loop I, loop J and loop K) and the 6 possible permutations in which these loops may be arranged are:

1. Order K,J,I
2. Order I,J,K
3. Order I,K,J
4. Order J,K,I
5. Order J,I,K
6. Order K,I,J

All these six possible permutations were evaluated 21 times and their execution times, in microseconds, were stored in an array. This array was used to calculate, the mean, median, range and standard deviation.(to see all the results refer to the Raw Results towards the end of the assignment)

Blocking/Tiling:
The code was used to multiply matrices using blocking is shown in figure 2

```
for (kk = 0; kk < en; kk += bsize) {
    for (jj = 0; jj < en; jj += bsize) {
        for (i = 0; i < 1024; i++) {
            for (j = jj; j < jj + bsize; j++) {
                sum = MD[i][j];
                for (k = kk; k < kk + bsize; k++) {
                    sum += MA[i][k]*MB[k][j];
                }
                MD[i][j] = sum;
            }
        }
    }
}
```

Figure 2: Multiplying matrices through blocking

While blocking, we use the assumption that the matrix size is an integral multiple of the block size. Considering this, for and our matrix of size 1024x1024, the possible block sizes are 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 and 1024. Each possible block size was evaluated 21 times, and its execution time, in microseconds, was stored in an array. We use this array to calculate the mean, median, range and standard deviation. (refer Raw Results)

Results and Discussion:
For exact execution times and various calculated statistics like mean, median, range and standard deviation, please refer to the **Raw Results** section.

Loop Interchange:
The various execution times for matrix multiplication, and the loop orders used for these multiplications were plotted in a box plot in figure 3.(see figure in next page) This figure shows that loop orders used and their respective execution times tend to exist in pairs. i.e. time taken to compute matrix multiplication using loop order K,J,I is similar to the time taken to calculate matrix multiplication using loop order J,K,I. This same relationship holds true for loop orders I,J,K and J,I,K and for loop orders I,K,J and K,I,J.

The results show that the most unoptimized loop orders were K,J,I and J,K,I. These loop orders had a median execution time of 28.39 and 28.82 seconds, respectively (refer to raw results section).

The next poorly optimized loop orders were I,J,K and J,I,K. These loop orders had a median execution time of 19.66 and 19.99 seconds, respectively (refer to raw results section). This is an average improvement of around 30% compared to the previous loop orders.

The most optimized loop orders were I,K,J and K,I,J. These loop orders had a median execution time of 4.12 and 4.38 seconds, respectively (refer to raw results section). This is an improvement of around 80% compared to the previous loop orders (order I,J,K and J,I,K).

Figure 3: Time vs Loop order boxplot.

From the results, it is clear that the most essential factor in determining the execution time of a particular loop order is the inner-most loop. In other words, if the inner-most loop is I, the performance is the worst. Similarly, if the inner-most loop is J, the performance is the best. The order of the outermost 2 loops does not seem to matter that much.

Keeping in mind the concepts of cache, spacial and temporal locality of reference, and the fact that C stores its matrices in row-major form, let us try to understand as to why certain loop orders are better than others.

Consider a matrix multiplication:

MD[I][J]+=MA[I][K]*MB[K][J];

In loop order I,J,K, since MD[I][J] is not going to change as the inner-most loop K executes, we can assume that MD[I][J] will be in the register or, at the very least, be a cache hit most of the time due to temporal locality of reference. So the other variables that we need to access to perform this task are:
MA[0][0], MB[0][0], MA[0][1], MB[1][0], MA[0][2], MB[2][0],…….
MA[1][0], MB[0][1], MA[1][1], MB[1][1], MA[1][2], MB[2][1],…….
…….

The values that are colored in green are those that are likely to be cache hits due to spacial locality of reference, and the values colored red are those that are likely to be cache misses. (because c stores matrices in the row-major form)

Note: The variable access pattern of loop order J,I,K is also very similar to that of loop order I,J,K and hence justifies the similarity in execution times.

For loop order K,J,I, since MB[K][J] is not going to change as the inner-most loop I executes, we can assume that MB[K][J] is going to be in the register or, at the very least, be a cache hit most of the time due to temporal locality of reference. So, the other variables that are needed to be accessed for matrix multiplication are:
MD[0][0], MA[0][0], MD[1][0], MA[1][0], MD[2][0],……
MD[0][1], MA[0][0], MD[1][1], MA[1][0], MD[2][1],……
……..

Note that there are no variables above that are likely to be cache hits due to spacial locality of reference, and this is responsible for its really long execution times.

As in the previous case, the variable access pattern of loop order J,K,I is very similar to that of loop order K,J,I, and hence justifies the similarity in execution times.

For loop order I,K,J, since MA[I][K] is not going to change as the inner-most loop J executes, we can assume that MA[I][K] is going to be in the register or, at the very least, be a cache hit most of the time due to temporal locality of reference. So, the other variables that are needed to be accessed for matrix multiplication are:
MD[0][0], MB[0][0], MD[0][1], MB[0][1]……
MD[0][0], MB[1][0], MD[0][1], MB[1][1]…….
……..

Note that all the variables above are likely to be cache hits due to spacial locality of reference and this is responsible for the optimal execution times.

The variable pattern of loop order K,I,J is going to be very similar to that of I,K,J, and hence, it justifies the similarity in execution times.

Conclusion:
The above experiment underscores the importance of keeping cache in mind while designing a program as we can see that just by a simple loop interchange, we can get the performance that is almost an order of magnitude faster. In the above example, If a matrix multiplication problem would have taken a month using loop order K,J,I, that same problem can be done well under a working week using loop order K,I,J.
Another critical point to note is that we need to keep into account the way the particular language stores matrices as these results would have been different if I had used another programming like Fortran. (I would expect loop orders K,J,I and J,K,I to be the most optimal in Fortran as it stores matrices in column-major order) In other words, the programming logic that is most optimal in one language need not be optimal in another language.

Blocking :
The various execution times for block sizes 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 and 1024 are plotted in Figure 4.

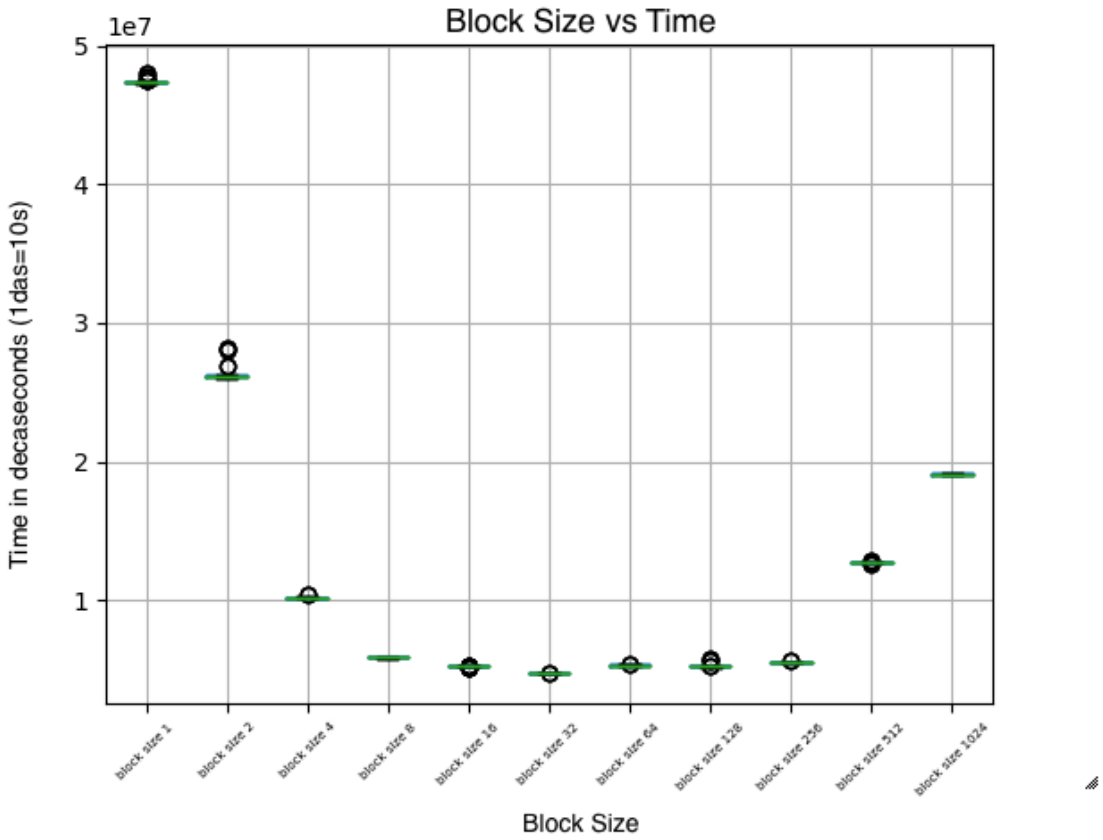

Figure 4: Block Size vs Time boxplot

From the figure, we can see that the most unoptimized size for blocking is 1. Its median execution time is around 47 seconds. Block size 2 is quite an improvement over the previous block size and can its median execution time is around 26.18 seconds. Block size 4 is an improvement over block size 2 and has a median execution time of around 10.17 seconds. This trend of improving times continues until the block size of 32 with its median time of 4.75 seconds. After that, the execution times get worse, and the median execution time of block size of 1024 is 19.10 seconds. (for all the exact values, please refer to Raw Results).

Let us try to understand as to why we see such a trend(as in Figure 4). The basic idea of blocking is to improve the temporal locality of the inner loops. This is done by organizing the data structures into large chunks called blocks. The program tries to load blocks into the L1 cache. All the required operations are carried out on the data in the blocks (in this case matrix multiplication), then the blocks are discarded, and the new blocks are loaded. Since the entire block is in the cache, this significantly improves performance by increasing temporal reference.
Now, since, the program tries to load the entire block into the L1 cache, the optimal block size will depend heavily on the size of the L1 cache.

In my particular local machine, I have an L1 cache of 32,768 bytes so the most optimal block size will be the one that ideally fits perfectly in the cache.
Let us evaluate the cache memory used for different block sizes, taking into account that for any matrix multiplication operation, we have to load three blocks into the L1 cache. (matrices MD, MA, MB)

- Block size of 8 -> 8x8x3x8 bytes= 1,536 bytes
- Block size of 16 -> 16x16x3x8 bytes = 6,144 bytes
- Block size of 32 -> 32x32x3x8 bytes=24,576 bytes
- Block size of 64 -> 64x64x3x8 bytes = 98,304 bytes
- Block size of 126 -> 128x128x3x8 bytes= 393,216 bytes
- Block size of 256 -> 256x256x3x8 bytes =1,572,864 bytes
  Similarly, the sizes of other block sizes may be calculated.
  Note: The 8 in the above calculations is due to the fact the double precision has a size of 8 bytes in c.

Considering that my local machine has an L1 cache of 32,768 bytes we can try to understand as to why the block size of 32 is the most optimal. Any block size above 32 and the data will not fit into the L1 cache causing significant cache misses, and any block size below will be sub-optimal as it will not utilize the cache to the fullest. This explains why the block size of 32 is the most optimal in my particular case.

Conclusion:
The above discussions help us understand the importance of taking into account the cache memory available. If we could adequately design our program around the cache, significant improvements in execution times are possible.
Given that we can find the optimal block size, this approach is more generalizable when compared to loop interchange, as it depends on the size of the L1 cache and not on how a particular language stores its values. In other words, If I had run the same program to calculate the most efficient block size in Fortran, I would likely still get the answer to be 32.

Raw Results:

- All values given below are in microseconds.
- Each loop order and block size was executed 21 times.

Loop interchange:

1. loop order(k,j,i):
   The execution times of this operation (in microseconds) are as follows:
   28177028, 28183212, 28218513, 28236984, 28243686, 28260677, 28274694, 28300563, 28310240, 28350342, 28383336, 28397790, 28624996, 28783198, 28834904, 28836704, 28839185, 28899468, 28914397, 28950539, 29002222,
   its range is 825194 (29002222 - 28177028)
   median is 28397790
   mean is 28524889.428571
   standard deviation is 297526.028650
2. loop order (i,j,k):
   The execution times of this operation (in microseconds) are as follows:
   19492285, 19522820, 19532118, 19612314, 19613177, 19614357, 19614912, 19627050, 19637096, 19647510, 19661262, 19661869, 19681270, 19683140, 19689453, 19690705, 19696769, 19705139, 19707672, 19716015, 19725004,

its range is 232719 (19725004 - 19492285)
median is 19661869
mean is 19644377.952381
standard deviation is 63378.252888

3. loop order(i,k,j):
   The execution times of this operation (in microseconds) are as follows:
   4037549, 4075081, 4078956, 4089889, 4092698, 4100887, 4106290, 4108223, 4108237, 4113865, 4120122, 4123970, 4126102, 4126194, 4141395, 4142500, 4143848, 4145492, 4149160, 4156424, 4175711,
   its range is 138162 (4175711 - 4037549)
   median is 4123970
   mean is 4117266.333333
   standard deviation is 31181.709174
4. loop order(j,k,i):
   The execution times of this operation (in microseconds) are as follows:
   28651249, 28682844, 28750638, 28760325, 28788518, 28792603, 28798051, 28801693, 28808668, 28814200, 28824464, 28828176, 28852651, 28874252, 28879359, 28894939, 28895024, 28902726, 28988818, 29121856, 29929212,
   its range is 1277963 (29929212 - 28651249)
   median is 28828176
   mean is 28887631.714286
   standard deviation is 252417.078164
5. loop order(j,i,k):
   The execution times of this operation (in microseconds) are as follows:
   19940263, 19951167, 19962290, 19963890, 19969113, 19979167, 19980029, 19981706, 19982526, 19995707, 19997407, 19998408, 20000618, 20005747, 20027817, 20033391, 20052697, 20057649, 20062722, 20078063, 20097730,
   its range is 157467 (20097730 - 19940263)
   median is 19998408
   mean is 20005624.142857
   standard deviation is 42578.234243
6. loop order(k,i,j):
   The execution times of this operation (in microseconds) are as follows:
   4272339, 4339148, 4339252, 4343141, 4353200, 4364717, 4365790, 4371462, 4379370, 4379497, 4381062, 4383614, 4386963, 4387647, 4392918, 4393227, 4396437, 4396646, 4410270, 4428560, 4436125,
   its range is 163786 (4436125 - 4272339)
   median is 4383614
   mean is 4376256.428571
   standard deviation is 34258.364544

Blocking:

1. for block size of 1
   The execution times of this operation (in microseconds) are as follows:
   47325293, 47337048, 47350661, 47359842, 47368286, 47370382, 47371334, 47374897, 47376498, 47383234, 47383746, 47385397, 47387116, 47396290, 47410312, 47412520, 47440160, 47508251, 47582689, 47788077, 47993213,

its range is 667920 (47993213 - 47325293)
median is 47385397
mean is 47443106.952381
standard deviation is 158695.444846

2. for block size of 2
   The execution times of this operation (in microseconds) are as follows:
   26018784, 26036140, 26058587, 26067263, 26090484, 26090674, 26127490, 26127599, 26148190, 26149766, 26175439, 26182322, 26185544, 26199148, 26201833, 26213062, 26217724, 26219976, 26954484, 28097873, 28182160,
   its range is 2163376 (28182160 - 26018784)
   median is 26182322
   mean is 26368787.714286
   standard deviation is 603198.739919
3. for block size of 4
   The execution times of this operation (in microseconds) are as follows:
   10134578, 10142608, 10158013, 10158634, 10162593, 10166667, 10169439, 10172529, 10174165, 10174379, 10177358, 10177659, 10177949, 10180341, 10185059, 10201423, 10201521, 10210209, 10218326, 10245748, 10367820,
   its range is 233242 (10367820 - 10134578)
   median is 10177659
   mean is 10188429.428571
   standard deviation is 47090.567396
4. for block size of 8
   The execution times of this operation (in microseconds) are as follows:
   5803667, 5805523, 5821008, 5824342, 5835476, 5837169, 5838112, 5843823, 5843836, 5857152, 5863087, 5864571, 5868361, 5868943, 5869674, 5871014, 5872191, 5885015, 5895708, 5896364, 5904706,
   its range is 101039 (5904706 - 5803667)
   median is 5864571
   mean is 5855702.000000
   standard deviation is 27982.595038
5. for block size of 16
   The execution times of this operation (in microseconds) are as follows:
   5169881, 5177526, 5191735, 5198781, 5204399, 5205919, 5209532, 5212715, 5212733, 5213901, 5214480, 5214934, 5215933, 5216020, 5216161, 5218479, 5226134, 5228950, 5238439, 5244830, 5250404,
   its range is 80523 (5250404 - 5169881)
   median is 5214934
   mean is 5213423.142857
   standard deviation is 18841.825784
6. for block size of 32
   The execution times of this operation (in microseconds) are as follows:
   4675527, 4721896, 4729544, 4730573, 4737026, 4738489, 4752686, 4753003, 4754197, 4754324, 4754592, 4759555, 4764268, 4768551, 4771705, 4771938, 4773941, 4774106, 4774892, 4788954, 4789532,
   its range is 114005 (4789532 - 4675527)
   median is 4759555

mean is 4754252.333333
standard deviation is 25422.262635

7. for block size of 64
   The execution times of this operation (in microseconds) are as follows:
   5259690, 5267044, 5268171, 5269592, 5277900, 5281550, 5283957, 5284104, 5285023, 5294009, 5297018, 5300923, 5308416, 5309196, 5310477, 5312174, 5313062, 5316239, 5321889, 5332933, 5405282,
   its range is 145592 (5405282 - 5259690)
   median is 5300923
   mean is 5299935.666667
   standard deviation is 30669.637095
8. for block size of 128
   The execution times of this operation (in microseconds) are as follows:
   5172136, 5172208, 5177284, 5178360, 5181890, 5186228, 5186447, 5186768, 5188181, 5197516, 5200435, 5207191, 5214530, 5214716, 5216088, 5217730, 5226685, 5247230, 5277382, 5609995, 5799240,
   its range is 627104 (5799240 - 5172136)
   median is 5207191
   mean is 5250392.380952
   standard deviation is 152346.114569
9. for block size of 256
   The execution times of this operation (in microseconds) are as follows:
   5488060, 5496866, 5508007, 5510195, 5510605, 5511105, 5514677, 5515659, 5516747, 5519785, 5520773, 5521495, 5528660, 5531017, 5532157, 5532699, 5542179, 5552959, 5553047, 5562689, 5566580,
   its range is 78520 (5566580 - 5488060)
   median is 5521495
   mean is 5525521.952381
   standard deviation is 20178.717158
10. for block size of 512
    The execution times of this operation (in microseconds) are as follows:
    12621826, 12664958, 12685893, 12689002, 12692343, 12695814, 12698522, 12699487, 12699854, 12709240, 12715845, 12716546, 12717128, 12723666, 12727110, 12730198, 12741349, 12744899, 12754205, 12784484, 12789007,
    its range is 167181 (12789007 - 12621826)
    median is 12716546
    mean is 12714351.238095
    standard deviation is 36675.375995
11. for block size of 1024
    The execution times of this operation (in microseconds) are as follows:
    19049478, 19052519, 19057832, 19070618, 19077035, 19078067, 19080260, 19082510, 19091597, 19094433, 19101511, 19107407, 19113392, 19116014, 19118656, 19122093, 19135324, 19142541, 19143028, 19147043, 19162213,
    its range is 112735 (19162213 - 19049478)
    median is 19107407
    mean is 19102074.809524
    standard deviation is 31985.930726