CSE 621
Assignment 2

1.
```
#define N 512
double A[N][N];
 for (j=0; j<N; j++)
    for(i=0; i<N; i++)
        s+= A[i][j];
```

Capacity (C) : 256 Kbyte (256 * 1024) = 2^{18} Bytes
Linesize (B) : 128 Bytes = 2^{7} Bytes
Word Size (w): 8 Bytes = 2^{3} Bytes

(a) A direct-mapped cache:

Associativity (E) = 1
No. of lines in cache/No. Of sets in cache (L) = $2^{18}/2^{7} = 2^{11}$
No. of words that can be stored in each line = Linesize/word size = $2^{7}/2^{3} = 2^{4}$

Cache mapping:

Line No	**First Element in Line**		**Last element in Line**
1	A[0][0]	…..	A[0][15]
2	A[0][16]	…..	A[0][31]
..		…..	...
33	A[1][0]	…..	A[1][15]
2048	A[63][496]	…..	A[63][511]

As we can see for a direct cache map, the number of lines present is not enough to store all the rows of the matrix A. And since in the inner loop, the element of A are accessed row-wise, every access will be a *cache-miss.* Hence, we have a total of 512 cache-misses in the inner loop.

In the outer-loop, the process begins again, since the elements that are required have been evicted due to *conflict-misses.*

Hence we have a total of 512 * 512 misses.

Total number of misses is 2^{18}

(b) A 4-way set-associative cache
Associativity (E) = 4
No. Of sets in cache (L) = $2^{18}/(2^2 * 2^{7)} = 2^9$
No. of words that can be stored in each line = Linesize/word size = $2^7/2^3 = 2^4$

Cache mapping:

Set No	**First Element in Line**		**Last element in Line**
S 1	A[0][0]	…..	A[0][15]
….	….	…..	…
S 2	A[0][16]	…..	A[0][31]
..		…..	...
S 33	A[1][0]	…..	A[1][15]
S 512	A[15][496]	…..	A[15][511]

Using this mapping we see that, A[0][0], A[16][0], A[32][0], A[48][0], …. A[448][0] all map to the Set 1. But each set has an associativity of 4. Hence once the first 4 Lines are filled with the respective blocks, accesses to the other 4 Blocks will cause the already present lines in cache to be evicted.

Hence we see that for the inner loop, all the accesses will cause a *cold-miss* and when it comes to the outer loop, since the elements in the inner loop all cause the previous elements to be evicted, every iteration of the outer-loop cause cold-misses in the inner loop.

Hence total no. of misses = Total number of iterations of the outer-loop * Total no. of misses in the inner loop.

=> 512 * 512

Total number of misses is 2^{18}

(c) A fully associative cache:

Number of Sets = 1
No. of lines in cache = $2^{18}/2^7 = 2^{11}$
No. of words that can be stored in each line = Linesize/word size = $2^7/2^3 = 2^4$

Since it is a fully associative cache, the blocks are not mapped to any particular line on the cache. An empty line in the cache can be filled with any block from memory.

Hence in one complete run of the inner-loop, 512 Rows of A are stored in the array.
On the next iteration of the outer-loop, the elements being accessed are a different column of the same rows in the cache. This can be seen in the following cache-mapping:

Cache mapping:

Line No	**First Element in Line**		**Last element in Line**
1	A[0][0]	…..	A[0][15]
2	A[1][0]	…..	A[1][15]
..		…..	...
512	A[511][0]	…..	A[511][15]

Hence this reduces to the lower-bound of N^2/B

Therefore total cache-misses can be calculated as:
=> Total No. of cache-misses in the inner-loop * Total No. of iterations of the outer-loop/No. of words in a line.

=> 512 * (512/16)
=> $2^9 * (2^9 / 2^4)$
=> 2^{14}

(d) Largest value of N for which the number of misses will be no more than the lower bound for the computation, if the cache is direct-mapped

Lower Bound is given by N^2/B
We know that :
$C = 2^{18}$ Bytes
No. of lines => $2^{18}/2^7 = 2^{11}$ Lines

Each Line contains => 128/8 = 16 Elements.

Therefore 2^{11} Lines can contain $2^{11} * 16 => 2^{11+4} = 2^{15}$ elements

This essentially needs to be equal to or greater than the total number of elements in the array A. Which gives us:

=> $N^2 <= 2^{15}$
= $N^2 <= 32768^{(½)}$
= $N^2 <= 181.019$

We also know that this number should be divisible by the Block/Line Size.

Hence we get the largest N = 176

(e) (d) for a 4-way associative cache

For the number of misses to be lower bound, it is essential that all the columns of the rows be accessed before they are evicted from the cache.
This essential reduces to the same as having a direct-mapped cache as above.

Hence we get the largest N = 176

(f) (d) for a fully associative cache

For a fully associative cache, we can have as many rows as we have lines in the cache, for the cache-misses to be lower bound.

Since the number of lines is 2^{11}

Hence we get the largest N = 2048

(g) For N=512, how large must a direct-mapped cache be in order that the number of cache misses equal the lower bound for the computation?

For the above algorithm to be lower-bound, it is required for a cache to be large enough to store all the Block size elements from each Row without any of them being evicted.

In order for this to happen, we need a cache that has a size equal to the size of the entire array.

We get,
No. of elements = $2^9 * 2^9 = 2^{18}$ elements (2^{21} Bytes)
Lower bound = N^2/B where B = 128 Bytes

Size of cache required = 2^{21} Bytes. Or 2 MB

(h) For N=512, what is the minimum degree of associativity needed to minimize the number of cache misses (to the lower bound for the computation)?

As mentioned above, to minimize the No. of misses, we need to have in cache all the rows of the array A, without getting it evicted. For this to happen, we need to have 512 lines in each set.

Hence minimum degree of associativity = 512

2.
```
double A[N][N+1]
  for (j=0; j<N; j++)
   for(i=0; i<N; i++)
     s+= A[i][j];
```

(a) For a direct-mapped cache:
Associativity (E) = 1
No. of lines in cache/No. Of sets in cache (L) = $2^{18}/2^{7} = 2^{11}$
No. of words that can be stored in each line = Linesize/word size = $2^{7}/2^{3} = 2^{4}$

Cache mapping:

Line No	**First Element in Line**		**Last element in Line**
1	A[0][0]	…..	A[0][15]
2	A[0][16]	…..	A[0][31]
..	...	…..	...
33	A[0][512]	A[1][0] …..	A[1][14]
34	A[1][15]		A[1][30]
...	...	...	...
65	A[1][511]	A[1][512], A[2][0], …....	A[2]13]
...	...	...	...
...	...	...	…..
321	…	… A[10][0], A[10][1], A[10][2], A[10][3], A[10][4]	A[10][5]
...	….	….	...
...	….	….	...
449	….	…......................... A[13][511], A[13][512], A[14][0]	A[14][1]
...	….	…..	
481	….	…..	A[15][0]
512	….	…..	A[15][496]
513	A[15][497]	…..	A[15][512]
514	A[16][0]	…..	A[16][15]
2048	A[63][433]	…..	A[63][448]
2049 = L1	A[63][449]		A[63][464]
2050 = L2	A[63][465]		A[63][480]
2051 = L3	A[63][481]		A[63][496]
2052 = L4	A[63][497]		A[63][512]
2053 = L5	A[64][0]		A[64][15]

As denoted by the cache-mapping,
A[0][0] maps to Line no. 1
A[1][0] maps to Line no. 33 and so on, in offsets of 32 lines.
When we get to A[64][0] because of the extra padding of one word per row, it gets mapped

to Line no. 5. (instead of Line no. 1 as was the case without padding)
This way, all the rows of the matrix A get stored in the cache. Since the size of the cache is large enough to store all rows, and there will be *no eviction* due to ***conflict-misses.***

This therefore reduces to the lower-bound number of misses.

Total number of misses is $N^2/B = 2^{18} / (128 / 8) = 2^{18-4} = 2^{14}$

(b) For a 4-way associative cache:

Associativity (E) = 4
No. Of sets in cache (L) = $2^{18}/(2^2 * 2^{7)} = 2^9$
No. of words that can be stored in each line = Linesize/word size = $2^7/2^3 = 2^4$

Cache mapping:

Set No	First Element in Line		Last element in Line
S 1	A[0][0]	…..	A[0][15]
….	….	…..	…
S 2	A[0][16]	…..	A[0][31]
..		…..	...
S 33	A[0][512]	A[1][0]…..	A[1][14]
S 512	….	…..	A[15][496]
S 513 = S 1	A[15][497]	…..	A[15][512]
S 514 = S 2	A[16][0]	…..	A[16][15]

As denoted by the cache-mapping,
A[0][0] maps to Set no. 1
A[1][0] maps to Set no. 33 and so on, in offsets of 32 Sets.
When we get to A[16][0] because of the extra padding of one word per row, it gets mapped to Set no. 2. (instead of Set no. 1 as was the case without padding)
This way, all the rows of the matrix A get stored in the cache. Since the size of the cache is large enough to store all rows, and there will be *no eviction* due to ***conflict-misses.***

This therefore reduces to the lower-bound number of misses.

Total number of misses is $N^2/B = 2^{18} / (128 / 8) = 2^{18-4} = 2^{14}$

(c) For a fully-associative cache:

Number of Sets = 1
No. of lines in cache = $2^{18}/2^7 = 2^{11}$

No. of words that can be stored in each line = Linesize/word size = $2^7/2^3 = 2^4$

Since it is a fully associative cache, the blocks are not mapped to any particular line on the cache. An empty line in the cache can be filled with any block from memory.

Hence in one complete run of the inner-loop, 512 Rows of A are stored in the array.
On the next iteration of the outer-loop, the elements being accessed are a different column of the same rows in the cache. This can be seen in the following cache-mapping:

Cache mapping:

Line No	**First Element in Line**		**Last element in Line**
1	A[0][0]	…..	A[0][15]
2	A[0][512]	A[1][0]…..	A[1][14]
..		…..	...
512	A[510][498]	…..	A[511][0]
513	A[0][16]	…..	A[0][31]

Hence this reduces to the lower-bound of N^2/B

Therefore total cache-misses can be calculated as:
=> Total No. of cache-misses in the inner-loop * Total No. of iterations of the outer-loop/No. of words in a line.

=> 512 * (512/16)
=> $2^9 * (2^9 / 2^4)$
=> 2^{14}

3.
```
#define N 64
double A[N][N][N], B[N][N][N], C[N][N];
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    for (k=0; k<N; k++)
      for (l=0; l<N; l++)
        C[i][j] += A[l][i][k]*B[k][l][j];
```

(a) Stride analysis considering each as inner loop:

Array	Inner-Loop As			
	i	**j**	k	l
A	N	**0**	1	N^2
B	0	**1**	N^2	N

C	N	**1**	0	0

We would use a permutation that has "j" as the inner most loop. As seen from the stride analysis above, it has the lowest stride, and hence it would be optimal to have it as the inner-most-loop.

(b)

i. Analysis for Array A

For a fixed I, j, k as l is varied, the elements of A are accessed in the 3rd Dimension. These elements are separated by a stride of $N^2(64^2 = 4096)$ and will be in blocks that are N^2/B apart where B is the linesize in words.

We get 4096/(128/8) = 256 apart.

With a direct-mapped cache of 256 Kbytes, we have 2048 sets. After 2048/256 = 8 successive elements, have been accessed in the 3rd dimension of A, the next access to A[8][0][0] will map again to set 0 causing the eviction of A[0][0][0], therefore when I is changed by one and a different row in the same 3rd dimension is being accessed, no reuse in cache is possible. Hence the total number of misses will be N * N * N * N
=> 64^4
=> 2^{24}

ii. Analysis for Array B:
For a fixed I, j, k as l is varied, the elements of B are accessed column wise. These elements are separated by a stride of N (64) and will be in blocks that are N/B apart where B is the linesize in words.

We get 64/(128/8) = 4 apart.

With a direct-mapped cache of 256 Kbytes, we have 2048 sets. These number of sets are sufficient to store all the rows (64) of a given column and constant third dimension. Therefore there will be N misses in the inner-most loop. (*cold miss*)

Now for the outer loops, ie, k, j and i , the elements will not be present and hence cache reuse is not possible.

Therefore total number of misses => N * N * N * N =>64^4

Total no. of misses = 2^{24}

iii. Analysis for Array C:

From the loop we see that the elements is array C have good temporal and spatial locality. In the inner-most 2 loops (k & l), the same element of C are being accessed. In the outer most two loops, the elements of C are being accessed with a stride of 1.

We have
C = 256 Kbytes = 2^{18} Bytes
B = 128 Bytes = 2^7 Bytes

Size of C = N * N = 64 * 64 words
= 2^{12} words = 2^{12} * 2^3 bytes = 2^{15} bytes.

No of elements in a line = $2^7 / 2^3 = 2^4$ (or 16 elements in a block)

We see that the size of the array C is smaller that the size of the cache. Hence the entire array can be stored in the cache, without the element getting evicted.

Therefore, this case reduces to a lower-bound on the number of misses.
=> N * N/B * 1 * 1=>$2^{15}/2^7 = 2^8$
Yields,
Total number of misses = 256

(c)
```
#define N 64
    double A[N][N][N], B[N][N][N], C[N][N];
    for (l=0; i<N; i++)
      for (i=0; j<N; j++)
        for (j=0; k<N; k++)
          for (k=0; l<N; l++)
            C[i][j] += A[l][i][k]*B[k][l][j];
```

i. Analysis of A:

For a fixed l, i, j, as k is varied elements of A are accessed column-wise, ie with a stride of N=1.
In our direct mapped cache, we have 2048 sets which is enough to store all the columns of a given row.
Hence the number of misses in the inner-most loop(k) = N/B =>64/16 = 4
Now when 'j' is varied, the same elements already present in the cache are being accessed. Hence there will be no misses.

Now, when 'i' is varied, the elements in A are being accessed row-wise and hence will all cause cold-misses.

Therefore the total number of misses can be given by

=> N * N * 1 * N/B = N^3/B
=> $64^4/2^4$ = **2^{20} misses**

ii. Analysis of B:

   For a fixed l,i,j as k is varied elements of B are accessed in the 3rd Dimension.
   These elements are separated by a stride of $N^2(64^2 = 4096)$ and will be in blocks that are N^2/B apart where B is the linesize in words.

   We get 4096/(128/8) = 256 apart.

   With a direct-mapped cache of 256 Kbytes, we have 2048 sets. After 2048/256 = 8 successive elements, have been accessed in the 3rd dimension of A, the next access to A[8][0][0] will map again to set 0 causing the eviction of A[0][0][0], therefore when j is changed by one and a different column in the same 3rd dimension is being accessed, no reuse in cache is possible. Hence the total number of misses will be N * N * N * N
   => 64^4
   => 2^{24} misses

iii. Analysis of C:

   Similar to the previous permutation case, array C has temporal and spatial locality in the l,i,j,k permutation as well.

   Therefore, this case reduces to a lower-bound on the number of misses.
   => N * N/B * 1 * 1=>$2^{15}/2^7$ = 28
   Yields,
   Total number of misses = 256

4.

5. Sequential phase takes : 100 * N^2 cycles
   Parallelizable phase takes: 10 * N^3 cycles

   (a) Maximum ideal speed up possible for:

   speedup = T_1 / T_n
   $T_1 = (100 * N^2) + (10 * N^3)$
   $T_n = (100 * N^2)$ In an ideal case, no. of processors is infinite.
   Therefore speedup = $T_1 / T_n = 1 + N/10$

   i. N = 10
      => Speedup = 1 + 10/10 = 2

   ii. N = 100
      => Speedup = 1 + 100/10 = 11

iii. N = 1000
=> Speedup = 1 + 1000/10 =101

(b) Maximum number processors for which efficiency is at least 0.5
Efficiency = Speedup / n
n = Speedup / Efficiency

i. N = 10
=> n = 2 / 0.5 = 4

ii. N = 100
=> n = 11 / 0.5 = 22

iii. N = 1000
=> n = 101 / 0.5 = 202