Roll no.: 241110406

Programming For Performance Assignment 1:

Problem 1:

Cache block length = 16 words(floats)
Cache blocks in array: 32k/16 = 2k
Cache size = 256KB
Array size = 128KB
⇒ Capacity miss not possible

Stride 1 access:

Cache misses: 32k/16 = **2k misses**. (all cold misses)
Why: Cache size is 256KB and array size is 128KB, so complete cache will be loaded in the cache and for later values of **it** all array accesses will result in cache hit.

Stride 4 access:

Array blocks accessed: 32k/16 = 2k
Per block 1 miss
Cache misses: **2k misses** (all cold misses)
Why: same as above

Stride 16 access:

Array blocks accessed: 32k/16 = 2k
Per block 1 miss
Cache misses: **2k misses** (all cold misses)
Why: same as above

Stride 64 access:

Array blocks accessed: 2k*I*(64/16) = 512
Cache misses: **512 misses** (all cold misses)
Why: Although not every block will be loaded (every 4th cache block will be loaded), the associativity is large enough that there will be no conflict misses. Every 512th block will be assigned the same cache set and only 2k cache blocks. So only 4 out of 8 lines will be used in each set and later accesses to those array blocks will cause cache hits.

Stride 2k access:

Array blocks accessed: 2k**/**(2k/16) = 16
Cache misses: **16 misses** (cold misses)
Why: Since every 512th block or 8k-th array element is assigned different lines in the same 8-way associative set while performing sequential access to array elements, and access stride is 2k, only 4 of 512 sets will be used and in each of those only 4 out of 8 cache lines will be populated. So no conflict misses and later accesses to those elements will cause cache hits.

Stride 8k access:

Array blocks accessed: 2k/(8k/16) = 4
Cache misses: **4 misses** (cold misses)
Why: Only 4 cache blocks will be loaded and all of them will populate only the first 4 lines of the first set. No conflict misses will occur and later accesses to those elements will result in cache hits.

Stride 16k access:

Array blocks accessed: 2k/(16k/16) = 2
Cache misses: **2 misses** (cold misses)
Why: Only first two lines of the 8 way associative set will be populated so no conflict misses are possible and later accesses will be cache hits.

Stride 32k access:

Array blocks accessed: 2k/(32k/16) = 1
Cache misses: **1 misses** (cold miss)
Why: Only the first cache block is accessed in a single iteration of **it** and later iterations of **it** cause cache hits on array access.

Problem 2:

N = 1024
Length of each block (BL) = 8 words
Cache size = 64k words
No. of lines in cache = 8k

kij form of loops

Listing 1: kij form

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

For direct mapped cache:

	A	B	C
k	1024 (N) ***(##)***	1024 (N)	1024 (N) ***(^)***
i	1024 (N)	1 ***(#)***	1024 (N)
j	1	128 (N/BL)	128 (N/BL)
Total	1M	128K	128M

Total cache miss = 1M + 128K + 128M = **135,397,376**

- **(#)** For one complete iteration cycle of **i** from 0 to N-1 to access the array **B** the value of **k** doesn't change so the same row is accessed multiple times unless the **k** changes. One row of **B** is small enough (1k words) to fit in the cache (64k words), so when the value of **i** changes later accesses to the same array element will result in cache hits.

- **(##)** For every 64th row (because each row has 1k words, cache size is 64k words and the array is arranged in row major order) the cache line assigned for an array block would be the same. Now, every (n > 63)th row accessed in a single column of **A**[i][k] will cause a miss and eviction of an existing cache block. So the next iteration of k (accessing the next column) will not find the required block in cache and result in cold miss.

- **(^)** For each iteration of **k** to access **C**, the complete array of size 1024x1024 words is loaded which is bigger than the whole cache of 64k words, so spatial locality is destroyed and every access to **C** for a new iteration of **k** results in a cache miss.

For fully associative cache:

	A	B	C
k	128 (N/BL) ***(#)***	1024 (N)	1024 (N)
i	1024 (N)	1	1024 (N)
j	1	128 (N/BL)	128 (N/BL)
Total	128K	128K	128M

Total cache miss = 128K + 128K + 128M = **134,742,016**

- **(#)** In case of Direct mapped cache there were evictions caused due to same cache lines being assigned to multiple cache blocks regardless of the available space in cache. A fully associative cache would rectify this and preserve every cache block loaded while accessing the columns of A, so that the adjacent column access would result in cache hits.

jik form of loops

Listing 2: jik form

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

For direct mapped cache:

	A	B	C
j	1024 (N) ***(##)***	1024 (N) ***(^)***	1024 (N) ***(^)***
i	1024 (N)	1024 (N) ***(#)***	1024 (N)
k	128(N/BL)	1024 (N)	1
Total	128M	1G	1M

Total cache miss = 128M + 1G + 1M = **1,209,008,128**

- ***(#)*** For a given iteration of **i** (say **i** = 0), the value of **j** remains the same and the value of **k** goes from 0 to 1023 loading **B[0 to 1023][0]**. Now, the column elements of **B** don't enjoy spatial locality. Moreover, considering the fact that DM cache has 8k cache lines and a single row of array makes 128 cache blocks of size 8 words, column-wise access to elements of **B** can accommodate only 64 elements in the cache at once. Since a single column of **B** contains 1024 elements there will be evictions of previously loaded cache blocks and in the next iteration of **i** all of the elements will be loaded again.

- ***(##)*** For any given value of **j**, the nested loops of **i** and **k** load the complete 1024x1024 array **A** which is bigger than the whole cache so there will be evictions in loading the array and next iteration of **j** will again load complete array from memory.

- ***(^)*** In both cases for a given value/iteration of **j**, **B[k][j]** and **C[i][j]** loads a column, but due to small size of cache and row-wise storage of array elements, column-wise access of elements causes eviction of loaded cache blocks whose spatial locality can't be used in next iteration of **j** to quickly access adjacent columns causing process to load the complete column again.

For fully associative cache:

	A	B	C
j	1024 (N) ***(##)***	128 (N) ***(^)***	128(N) ***(^)***
i	1024 (N)	1 ***(#)***	1024 (N)
k	128(N/BL)	1024 (N)	1
Total	128M	128K	128K

Total cache miss = 128M + 128K + 128K = **134,742,016**

- ***(#)*** For any given value of i, j remains fixed and only k completes its full cycle from 0 to 1023. That means for every value of i program loads the same column of B using B[k][j]. But unlike direct mapped cache, fully associative cache can store all the column-wised accessed blocks at the same time and for next time access of the same column there are no cache misses.

- ***(##)*** For any given value of **j**, the nested loops of **i** and **k** load the complete 1024x1024 array **A** which is bigger than the whole cache so there will be evictions in loading the array and next iteration of **j** will again load complete array from memory.

- ***(^)*** In both cases for a given value/iteration of **j**, **B[k][j]** and **C[i][j]** loads the **j**th column, but unlike direct mapped cache, column-wise access of elements in fully associative cache doesn't causes eviction of loaded cache blocks, whose spatial locality can be used in next iteration of **j** to quickly access adjacent columns in event of a cache hit.

Problem 3:

Given program:

```
#define N (4096)
double y[N], X[N][N], A[N][N];
for (k = 0; k < N; k++)
  for (j = 0; j < N; j++)
    for (i = 0; i < N; i++)
      y[i] = y[i] + A[i][j] * X[k][j];
```

Direct Mapped Cache specifications:

Cache size = 16MB
Line size = 32B
Word size = 8B
No. of cache lines = 512K
Block Length (BL) = 4 words

Cache miss estimation table:

	A	X	y
k	4096 (N)	4096 (N)	1 ***(##)***
j	4096 (N) ***(#)***	1024 (N/BL)	1 ***(##)***
i	4096 (N)	1	1024 (N/BL)
Total	64G	4M	1024

Total cache miss = 64G+ 4M+ 1K = **68,723,672,064**

- **(#)** For every iteration of **j**, the program loads **j**th column of matrix **A** but columnwise access of elements assigns each block 1024 lines after the previously assigned block in the direct mapped cache (because each row of the 4096x4096 array can hold 1024 blocks). So, the cache can hold only 512 blocks from column-wise access at the same time, but column size is 4096 elements so cache evictions will occur while loading the column from memory. Cache evictions will not allow spatial locality to be used to access adjacent columns for the next iteration of **j** causing 100% cache miss for every iteration of **j**.

- ***(##)*** For the first iteration of **j** and **k** both the array **Y** will be loaded in the cache with some cache misses, but for later iterations of **j** and **k** the same array **Y** has to be used again and is already loaded in cache, so it will be a cache hit for all later iterations.

Problem 4:

Function for sequential matrix multiplication:

***Inputs**: Pointers for array A, B and C*

```
void matmul_ijk(const uint32_t *A, const uint32_t *B, uint32_t *C,
const int SIZE) {
  for (int i = 0; i < SIZE; i++) {
    for (int j = 0; j < SIZE; j++) {
      uint32_t sum = 0.0;
      for (int k = 0; k < SIZE; k++) {
        sum += A[i * SIZE + k] * B[k * SIZE + j];
      }
      C[i * SIZE + j] += sum;
    }
  }
}
```

Function of blocked matrix multiplication:

***Inputs:** Pointers for array A, B and C and an integer array pointer blk_size*

```
void matmul_ijk_blocking(const uint32_t *A, const uint32_t *B,
uint32_t *C, const int SIZE, int* blk_size) {
  for(int i = 0; i < SIZE; i += blk_size[0])
    for(int j = 0; j < SIZE; j += blk_size[1])
      for(int k = 0; k < SIZE; k += blk_size[2])


        for(int ii = i; ii < min(N, i+blk_size[0]); ii++)
          for(int jj = j; jj < min(N, j+blk_size[1]); jj++)
            for(int kk = k; kk < min(N, k+blk_size[2]); kk++)
              C[ii*SIZE + jj] += A[ii*SIZE + kk]*B[kk*SIZE + jj];
}
```

Performance gains:

Size of Matrices: ***2048x2048***

Word size: ***32 bit***

Block size			Average Speed Up (time_seq / time_blk)	Average L2 cache miss reductions by blocking code
A	B	C		
2	2	2	0.41x	-18904109
4	4	4	1.06x	1624265456
8	8	8	2.20x	1824321411
16	16	16	1.52x	1977513973
32	32	32	0.10x	2121534526
64	64	64	0.89x	1192977918
4	4	8	1.17x	1624091482
4	8	8	1.80x	1560218873
8	8	16	1.30x	1732444795
16	8	16	1.45x	1912129836
8	8	32	1.05x	1917303745

- **Best Performing block size (in terms of time)**: A => 8, B => 8, C => 8
- **Best Performing block size**
  (in terms of L2 cache miss reductions): A => 32, B => 32, C => 32

System Cache Hierarchy:

Cache Information.

L1 Data Cache:
Total size: 32 KB
Line size: 64 B
Number of Lines: 512
Associativity: 8

L1 Instruction Cache:

Total size: 64 KB
Line size: 64 B
Number of Lines: 1024
Associativity: 4

L2 Unified Cache:

Total size: 512 KB
Line size: 64 B
Number of Lines: 8192
Associativity: 8

L3 Unified Cache:

Total size: 4096 KB
Line size: 64 B
Number of Lines: 65536
Associativity: 16