# ECE563 – Project 3

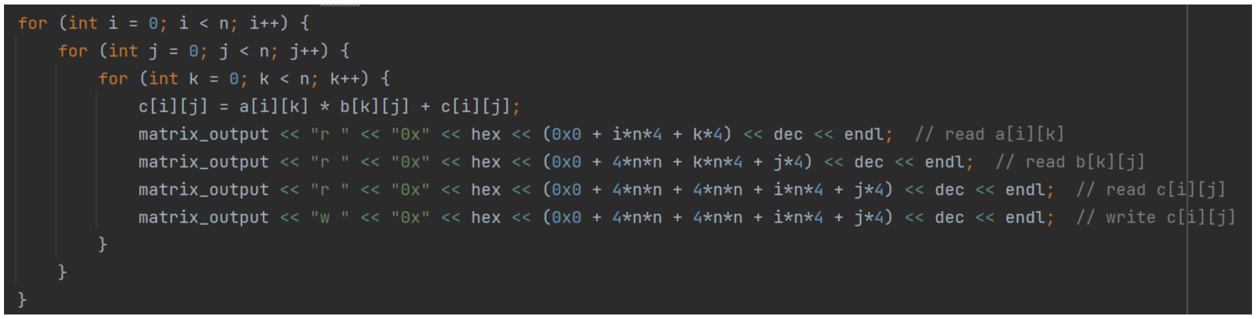
# L1 Cache Simulator

Prepared by:

Shiva Kaushik

## 1. Naive Matrix Multiplication

The following code is used for generating memory access traces for the naive matrix multiplication algorithm. The size of all the 3 matrices is set to 128 x 128.



```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        for (int k = 0; k < n; k++) {
            c[i][j] = a[i][k] * b[k][j] + c[i][j];
            matrix_output << "r " << "0x" << hex << (0x0 + i*n*4 + k*4) << dec << endl;  // read a[i][k]
            matrix_output << "r " << "0x" << hex << (0x0 + 4*n*n + k*n*4 + j*4) << dec << endl;  // read b[k][j]
            matrix_output << "r " << "0x" << hex << (0x0 + 4*n*n + 4*n*n + i*n*4 + j*4) << dec << endl;  // read c[i][j]
            matrix_output << "w " << "0x" << hex << (0x0 + 4*n*n + 4*n*n + i*n*4 + j*4) << dec << endl;  // write c[i][j]
        }
    }
}
```

Figure 1: Naive Matrix Multiplication Code

The memory accesses are executed on the multiple cache configurations as given in Table 1 (detailed configuration combinations used for testing are in given in table 3). The miss rate for the naive matrix multiplication for all the cache configurations is given in Figure 2. The write hit/miss Write Back/Write Allocate is used for testing. (In this algorithm, we can never have a write miss because C is the only variable that's being written which read before being written)

| | |
|---|---|
| Cache Size | 16KB, 32KB, 64KB |
| Cache Associativity | 1-way, 2-way, 4-way, 16-way |
| Block Size | 32B, 64B, 128B, 256B |

Table 1: Cache configurations used

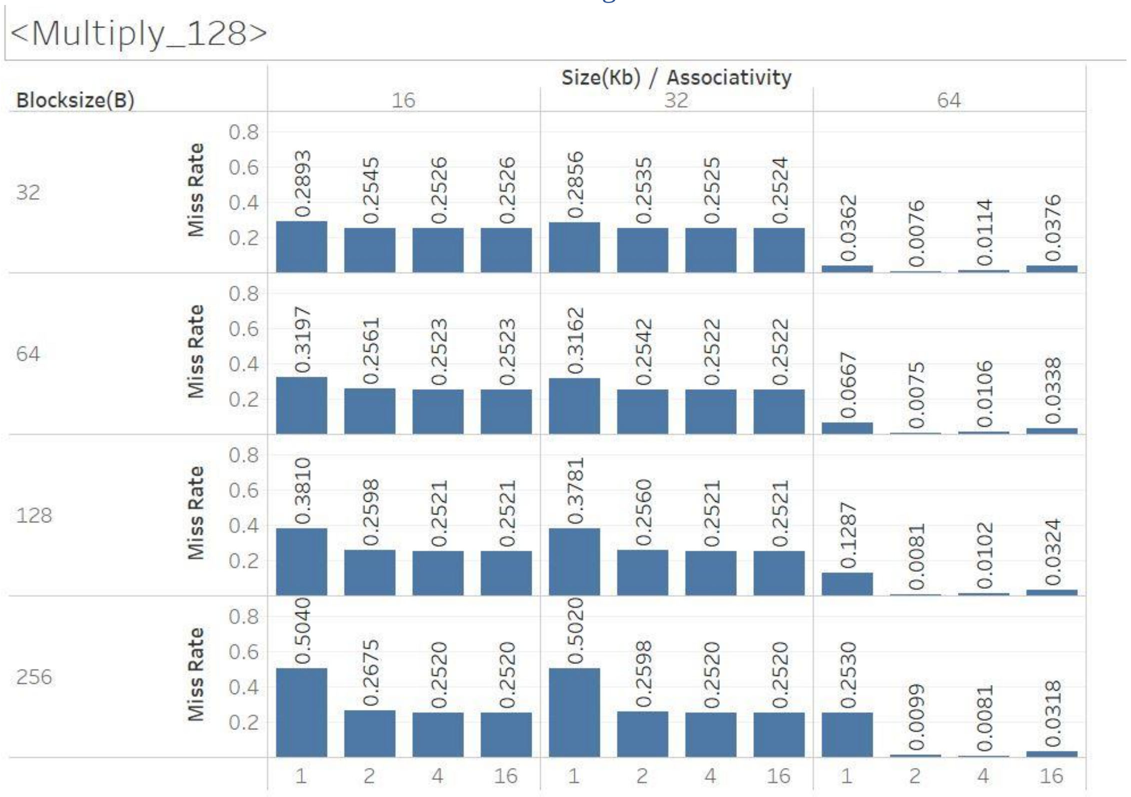


Figure 2: Miss Rate of multiple cache configurations for naive matrix multiplication

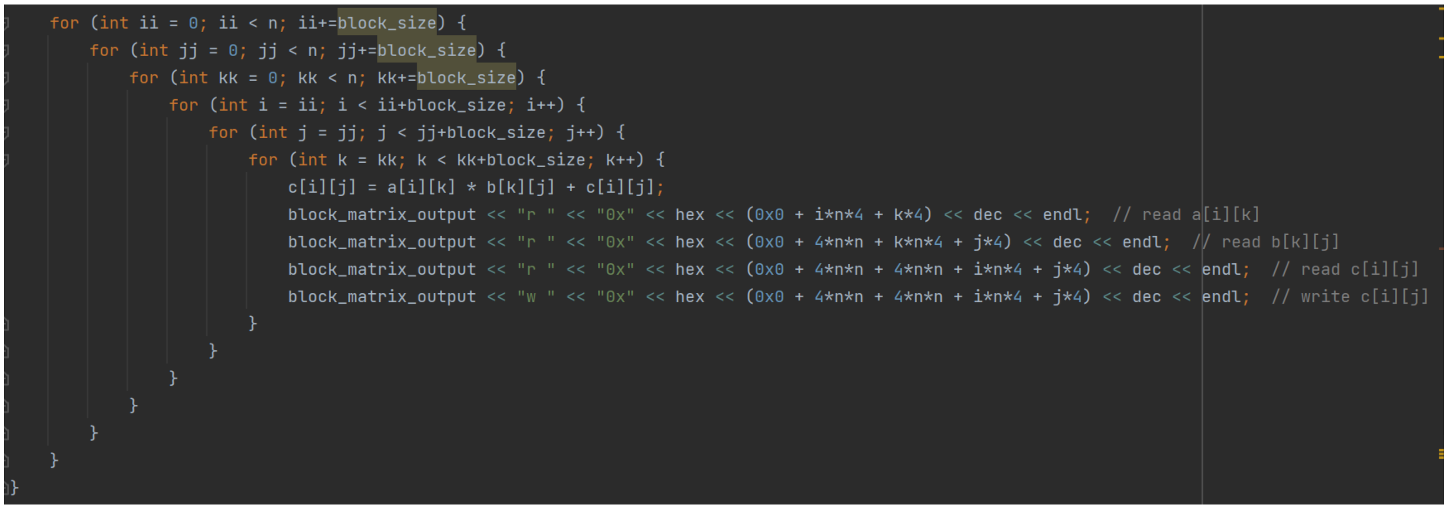
The address ranges of the variables used are as follows,

A – 0x0 – 0x0FFFF; B – 0x10000 – 0x1FFFF; C – 0x20000 – 0x2FFFF

The variables A, C content for the same cache set during some of the loops. So, we can see that for any Cache Size/Block Size, configurations with 1-Way have a higher miss rate and the miss-rate of 2, 4 and 16 is almost the same because only 2 variables content for a cache line most of the time while having 4 – ways has little or no benefit over 2 -ways. Also, for caches with 1-way associativity, the miss rate increases if the block size is increased. Having more bits for set index i.e having more sets is beneficial because the higher bytes of the addresses of variables differ, so if these bytes overlap with set index bits, we can space out the variables thereby have less evictions. So, increasing the cache size helps to lower the miss rate. So, from Figure-2, we can see that an optimal cache configuration for this program is 64KB, 64B, 2-way. Increasing block size beyond 64B has little or no benefit in caches with associativity greater than or equal to 2.

## 2. Cache Friendly Matrix Multiplication

The following code is used for generating memory access traces for the cache friendly matrix multiplication algorithm. The size of all the 3 matrices is set to 128 x 128. The memory access traces for block_size 2, 4, 8, 16, 32, 64 are computed and compared.



```
    for (int ii = 0; ii < n; ii+=block_size) {
        for (int jj = 0; jj < n; jj+=block_size) {
            for (int kk = 0; kk < n; kk+=block_size) {
                for (int i = ii; i < ii+block_size; i++) {
                    for (int j = jj; j < jj+block_size; j++) {
                        for (int k = kk; k < kk+block_size; k++) {
                            c[i][j] = a[i][k] * b[k][j] + c[i][j];
                            block_matrix_output << "r " << "0x" << hex << (0x0 + i*n*4 + k*4) << dec << endl;  // read a[i][k]
                            block_matrix_output << "r " << "0x" << hex << (0x0 + 4*n*n + k*n*4 + j*4) << dec << endl;  // read b[k][j]
                            block_matrix_output << "r " << "0x" << hex << (0x0 + 4*n*n + 4*n*n + i*n*4 + j*4) << dec << endl;  // read c[i][j]
                            block_matrix_output << "w " << "0x" << hex << (0x0 + 4*n*n + 4*n*n + i*n*4 + j*4) << dec << endl;  // write c[i][j]
                        }
                    }
                }
            }
        }
    }
}
```

Figure 3: Cache Friendly Matrix Multiplication

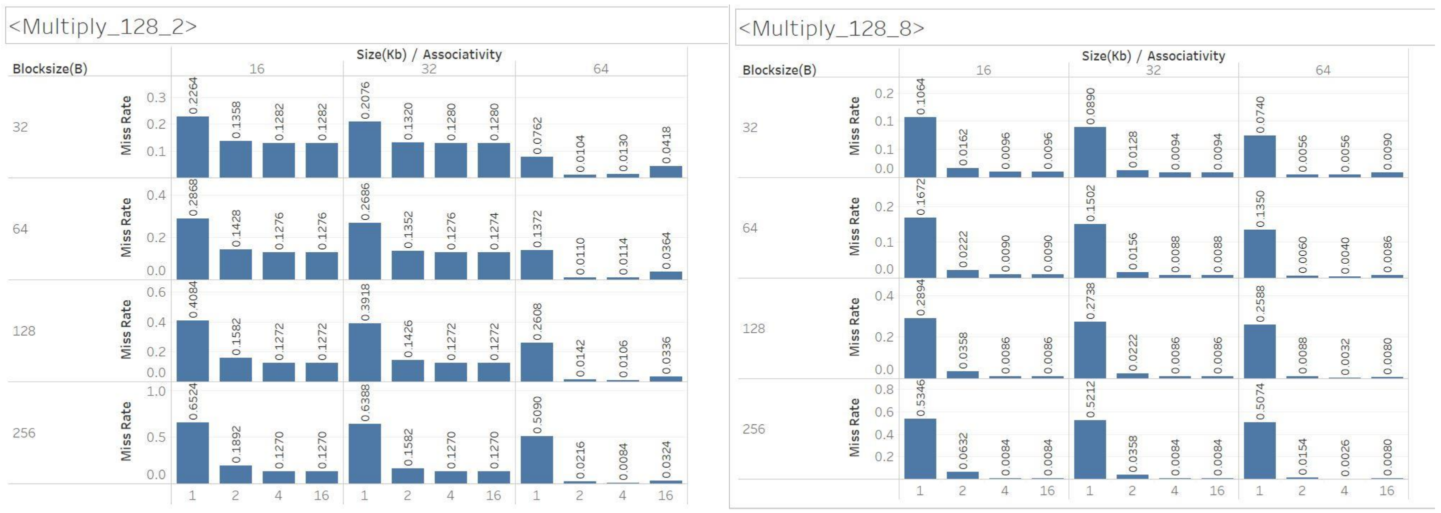


Figure 4: Miss Rate of multiple cache configurations for cache friendly multiplication with block offset set to 2 and 8

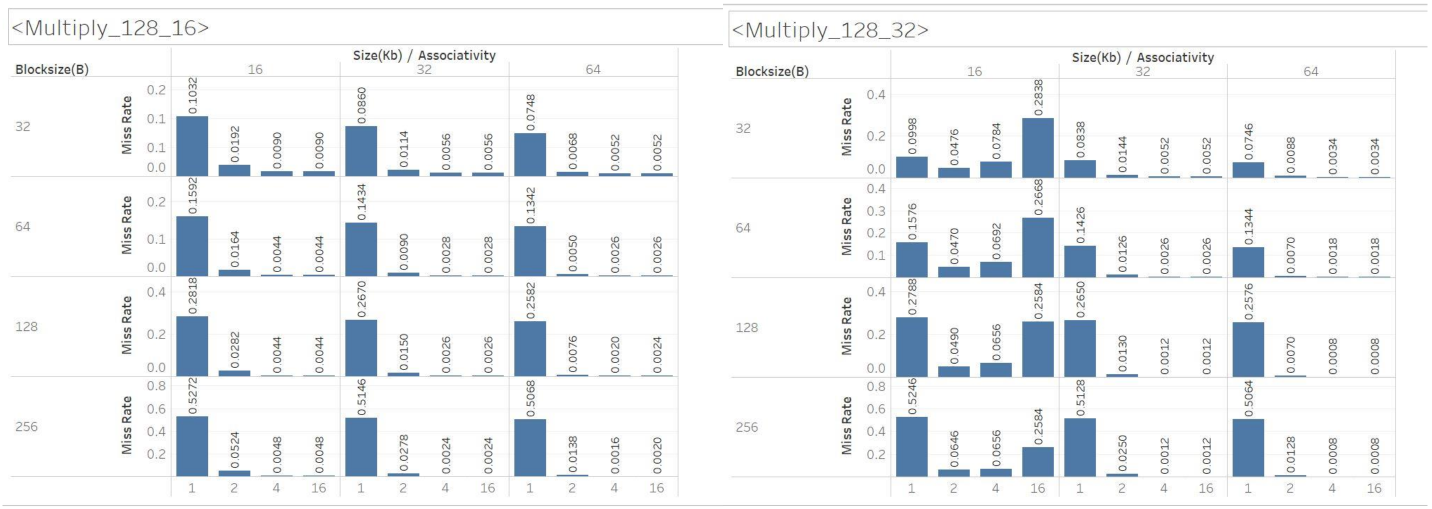


Figure 5: Miss Rate of multiple cache configurations for cache friendly multiplication with block offset set to 16 and 32

For any given cache configuration, the block_size parameter in the loop set to the cache block size/4 has the least miss rate value. For example, let's consider the cache configuration 32KB, 64B, 2-way.

| Block_Size Value | Miss Rate |
|---|---|
| 2 | 0.1352 |
| 8 | 0.0156 |
| 16 | 0.0090 |
| 32 | 0.0126 |
| 64 | 0.026 |

Table 2: Block_Size value parameter and Miss Rate for cache config 32KB, 64B, 2-way

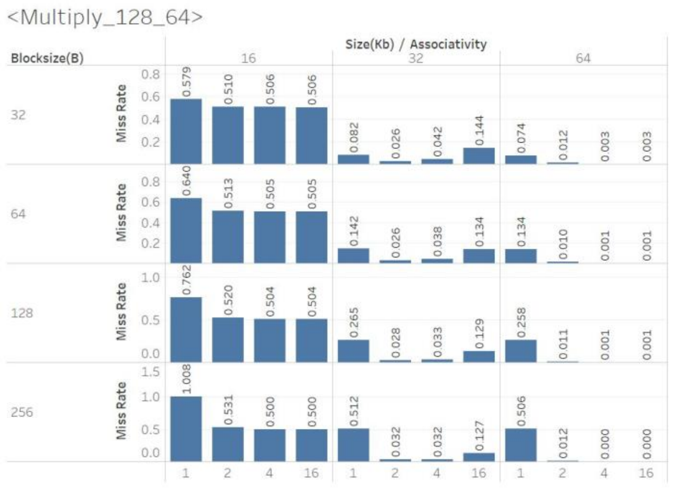


Figure 6: Miss Rate of multiple cache configurations for cache friendly multiplication with block offset set to 64

So, from above table, we can see that the miss rate is minimum for this cache configuration when block_size parameter is set to 16 which is Block Size (64) / 4. (It's divided by 4 because each integer is assumed to be 4 bytes. So, BlockSize/4 is the total number of integers a single cache block can hold). Figures 4, 5, 6 shows the miss rates of different loop block_size parameters (2, 8, 16, 32, 64) for all the cache configurations. The cache friendly implementation utilizes the cache in a more optimized way than the naïve implementation because the lower 3 loops ensure that variables that are loaded to cache are fully utilized before being evicted. For example, when a[0][0] is read, if the cache block size 64 bytes, the cache fetches a[0][0] to a[0][15], when b[0][0] is read, b[0][0] to [0][15] are fetched, in the next cycle, a[0][1] is used and b[1][0] is which is not present in cache so again b[1][0] to b[1][15] are fetched and if the block_size is set to 16, b is fetched every cycle and assuming there enough unique sets, the cache will contain b[0][0] to b[15][15] available in cache. So, if we continue fetching b[16][0], b[17][0],……. b[127][0] like in naive code implementation, we might evict the already cache blocks containing b[0][0] to b[15][15], instead we could use these to compute all the required values and then fetch the subsequent entries of b. In this way, even if the existing entries of b are evicted, it wouldn't effect the performance because all the values than need these entries have already been computed so these entries wouldn't be required anymore.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| A_0_0 | A_0_1 | A_0_2 | … | A_0_15 | A_0_16 | ………. | A_0_127 |
| A_1_0 | A_1_1 | A_1_2 | … | A_1_15 | A_1_16 | ………. | A_1_127 |
| A_2_0 | A_2_1 | A_2_2 | … | A_2_15 | A_2_16 | ………. | A_2_127 |
| … | | | | | | | |
| A_15_0 | A_15_1 | A_15_2 | … | A_15_15 | A_15_16 | ………. | A_15_127 |
| … | | | | | | | |
| A_127_0 | A_127_1 | A_127_2 | … | A_127_15 | A_127_16 | ………. | A_127_127 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| B_0_0 | B_0_1 | B_0_2 | … | B_0_15 | B_0_16 | ………. | B_0_127 |
| B_1_0 | B_1_1 | B_1_2 | … | B_1_15 | B_1_16 | ………. | B_1_127 |
| B_2_0 | B_2_1 | B_2_2 | … | B_2_15 | B_2_16 | ………. | B_2_127 |
| … | | | | | | | |
| B_15_0 | B_15_1 | B_15_2 | … | B_15_15 | B_15_16 | ………. | B_15_127 |
| … | | | | | | | |
| B_127_0 | B_127_1 | B_127_2 | … | B_127_15 | B_127_16 | ………. | B_127_127 |

Under the conditions described above, after executing the lowest for loop (k), the data marked in green would be available in cache but only data highlighted is used. If we continue fetching subsequent elements marked in red we might evict the cache lines holding the green data even before they're utilized. The cache friendly implementation ensures all the green data that's already fetched is utilized before fetching the next elements. So, later when the new elements are fetched and then if the green data is evicted, it wouldn't have any effect on performance because this data is not needed anymore as all the computation that require this data has already been done.

| Cache Size | Block Size | Ways | #Blocks | #Sets | #Offset Bits | #Index Bits | #Tag Bits |
|---|---|---|---|---|---|---|---|
| 16 | 32 | 1 | 512 | 512 | 5 | 9 | 18 |
| 16 | 32 | 2 | 512 | 256 | 5 | 8 | 19 |
| 16 | 32 | 4 | 512 | 128 | 5 | 7 | 20 |
| 16 | 32 | 16 | 512 | 32 | 5 | 5 | 22 |
| 16 | 64 | 1 | 256 | 256 | 6 | 8 | 18 |
| 16 | 64 | 2 | 256 | 128 | 6 | 7 | 19 |
| 16 | 64 | 4 | 256 | 64 | 6 | 6 | 20 |
| 16 | 64 | 16 | 256 | 16 | 6 | 4 | 22 |
| 16 | 128 | 1 | 128 | 128 | 7 | 7 | 18 |
| 16 | 128 | 2 | 128 | 64 | 7 | 6 | 19 |
| 16 | 128 | 4 | 128 | 32 | 7 | 5 | 20 |
| 16 | 128 | 16 | 128 | 8 | 7 | 3 | 22 |
| 16 | 256 | 1 | 64 | 64 | 8 | 6 | 18 |
| 16 | 256 | 2 | 64 | 32 | 8 | 5 | 19 |
| 16 | 256 | 4 | 64 | 16 | 8 | 4 | 20 |
| 16 | 256 | 16 | 64 | 4 | 8 | 2 | 22 |
| 32 | 32 | 1 | 1024 | 1024 | 5 | 10 | 17 |
| 32 | 32 | 2 | 1024 | 512 | 5 | 9 | 18 |
| 32 | 32 | 4 | 1024 | 256 | 5 | 8 | 19 |
| 32 | 32 | 16 | 1024 | 64 | 5 | 6 | 21 |
| 32 | 64 | 1 | 512 | 512 | 6 | 9 | 17 |
| 32 | 64 | 2 | 512 | 256 | 6 | 8 | 18 |
| 32 | 64 | 4 | 512 | 128 | 6 | 7 | 19 |
| 32 | 64 | 16 | 512 | 32 | 6 | 5 | 21 |
| 32 | 128 | 1 | 256 | 256 | 7 | 8 | 17 |
| 32 | 128 | 2 | 256 | 128 | 7 | 7 | 18 |
| 32 | 128 | 4 | 256 | 64 | 7 | 6 | 19 |
| 32 | 128 | 16 | 256 | 16 | 7 | 4 | 21 |
| 32 | 256 | 1 | 128 | 128 | 8 | 7 | 17 |
| 32 | 256 | 2 | 128 | 64 | 8 | 6 | 18 |
| 32 | 256 | 4 | 128 | 32 | 8 | 5 | 19 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 32 | 256 | 16 | 128 | 8 | 8 | 3 | 21 |
| 64 | 32 | 1 | 2048 | 2048 | 5 | 11 | 16 |
| 64 | 32 | 2 | 2048 | 1024 | 5 | 10 | 17 |
| 64 | 32 | 4 | 2048 | 512 | 5 | 9 | 18 |
| 64 | 32 | 16 | 2048 | 128 | 5 | 7 | 20 |
| 64 | 64 | 1 | 1024 | 1024 | 6 | 10 | 16 |
| 64 | 64 | 2 | 1024 | 512 | 6 | 9 | 17 |
| 64 | 64 | 4 | 1024 | 256 | 6 | 8 | 18 |
| 64 | 64 | 16 | 1024 | 64 | 6 | 6 | 20 |
| 64 | 128 | 1 | 512 | 512 | 7 | 9 | 16 |
| 64 | 128 | 2 | 512 | 256 | 7 | 8 | 17 |
| 64 | 128 | 4 | 512 | 128 | 7 | 7 | 18 |
| 64 | 128 | 16 | 512 | 32 | 7 | 5 | 20 |
| 64 | 256 | 1 | 256 | 256 | 8 | 8 | 16 |
| 64 | 256 | 2 | 256 | 128 | 8 | 7 | 17 |
| 64 | 256 | 4 | 256 | 64 | 8 | 6 | 18 |
| 64 | 256 | 16 | 256 | 16 | 8 | 4 | 20 |

Table 3: Cache Configurations used for testing