Exercise Sheet 2 : Costing and Modelling

Assumptions

- The secondary storage is HDD and it has a seek time while performing random access. We assume that all pages of a relation are stored in contiguous regions on the disk and that different relations are stored in different regions. The smallest unit of transfer from a disk is a page.
- $\|R\|$ denotes the number of tuples in relation R , $|R|$ denotes the number of disk pages required to store R and ρ_R denotes the number of tuples per page in R. $\|R\| = |R| * \rho_R$.
- Whenever caches are used, we assume it is a write-back cache. Therefore, writes go to caches and they are transferred to memory asynchronously. We ignore memory costs for writes, and consider only cache cost.

Exercise 1: Joins

1. Give pseudo-code for Simple Nested Loop Join between two relations R and S and analyse its cost in terms of number of disk seeks and pages transferred.

   Solution: The algorithm reads every tuple of R one after the other, and for each tuple of R, it iterates over every tuple of S. The algorithm uses two buffer pages in memory, one for each relation. In order to obtain the cost through coarsening, we split the loop into two parts - transferring page into buffer and processing tuples from the buffer. For the outer relation R, there is a seek and a transfer once every tuple of a page gets processed in order to fetch the next page. We assume that the tuples of S are stored in contiguous regions and that the algorithm processes tuples of S stored in the buffer page faster than the speed with which the disk transfers one page into the buffer. Thus, after transferring one page of S, the disk transfers the next page without waiting or seeking. Thus, there is only one seek for scanning the relation S. If it had been the case that either the pages of S were not contiguous or that the computations were not finished by the time the disk finished transferring one page, a seek would be required to fetch the each page. We assume the output is simply being printed on screen or discarded and we ignore CPU and memory costs involved in the computations.

```
1:  procedure SimpleNestedLoopJoin
2:      for all page p_R in R do                          ▷ Repeat |R| times
3:          read p_R into buffer for R
4:          seeks += 1
5:          transfers += 1
6:          for all tuple t_R in p_R do                   ▷ Repeat ρ_R times
7:              seeks += 1                                ▷ Only once to go to start of S
8:              for all page p_S in S do                  ▷ Repeat |S| times
9:                  read p_S into buffer for S
10:                 transfers += 1                        ▷ for S
11:                 for all tuple t_S in p_S do           ▷ Repeat ρ_S times
12:                     if t_R and t_S match condition θ then
13:                         output(t_R, t_S)
```

$$
\begin{aligned}
\text{Total seeks} &= |R| * (1 + \rho_R * 1) = |R| + \|R\| \\
\text{Seek cost} &= \text{Total seeks} * 10 \text{ ms} = (|R| + \|R\|) * 10 \text{ ms} \\
\text{Total transfers} &= |R| * (1 + \rho_R * |S|) = |R| + \|R\| * |S| \\
\text{Transfer cost} &= \text{Total transfers} * 0.1 \text{ ms} = (|R| + \|R\| * |S|) * 0.1 \text{ ms}
\end{aligned}
$$

2. Rewrite the algorithm to make use of the same buffer pages more efficiently and evaluate the cost.

   Solution: We swap the loop order to load one page of S before iterating over tuples of R stored in the buffer. In the outer loop, the disk seeks to the next page of R, transfers it and then seeks to the start of S. In the inner loop, each page of S is read into the buffer one after the other. We assume that the time taken for transferring one page from disk is higher than the computational cost of processing the tuples in it. Therefore, pages of S are read continuously and there is only seek for the entire inner loop.

```
1: procedure PageOrientedNestedLoopJoin
2:     for all page p_R in R do                                  ▷ Repeat |R| times
3:         read p_R into buffer
4:         seeks += 1
5:         transfers +=1
6:
7:         seeks += 1                                            ▷ Only once to go to start of S
8:         for all page p_S in S do                              ▷ Repeat |S| times
9:             read p_S into buffer
10:            transfers += 1
11:            for all tuple t_R in p_R do
12:                for all tuple t_S in p_S do
13:                    if t_R and t_S match condition θ then
14:                        output(t_R, t_S)
```

$$\begin{aligned}\text{Total seeks} &= |R| * 2 \\ \text{Total transfers} &= |R| * (1 + |S|) \\ &= |R| + |R| * |S|\end{aligned}$$

3. Generalize the algorithm for buffer size B > 2 and evaluate the new cost.

   Solution: We assign one page to buffer the relation S and all remaining pages to buffer the outer relation R. By maximizing the number of pages of R in memory, we reduce the total number of iterations over the S relation. While costing, we discard the output as usual.

```
1: procedure BlockNestedLoopJoin
2:     for all blocks of (B − 1) pages b_R in R do               ▷ Repeat ⌈|R| / (B − 1)⌉ times
3:         read b_R into buffer
4:         seeks += 1
5:         transfers += (B-1)
6:
7:         seeks += 1                                            ▷ Only once to go to start of S
8:         for all page p_S in S do                              ▷ Repeat |S| times
9:             read p_S into buffer
10:            transfers += 1
11:            for all tuple t_R in b_R do
12:                for all tuple t_S in p_S do
13:                    if t_R and t_S match condition θ then
14:                        output(t_R, t_S)
```

$$\begin{aligned}\text{Total seeks} &= \left\lceil \frac{|R|}{B-1} \right\rceil * 2 \\ \text{Total transfers} &= \left\lceil \frac{|R|}{B-1} \right\rceil * ((B-1) + |S|) \\ &= |R| + \left\lceil \frac{|R|}{B-1} \right\rceil * |S|\end{aligned}$$

4. Given a hash function h_1 that hashes tuples of both relations uniformly into partitions of appropriate sizes, explain how you can reduce the disk I/O for equi-joins.

   Solution: We scan each relation and partition the tuples according to their hash value. Then, since the join is based on equality, we only need to compare tuples in i^{th} partition of R with those in the i^{th} partition of S because equal tuples hash to the same value. Assume h_1 hashes tuples into $k = (B-1)$ hash partitions. We use one page as input buffer and (B-1) pages for partitions. Whenever the partitions gets full, we flush them to disk. Assuming that the hash function partitions tuples uniformly, on average, one buffer page gets written out to disk

after processing every tuple from one page of the relation. Based on this assumption, during the partition phase, on average, one input page is loaded, processed and one hash buffer page is written before loading the next input page. Thus, two seeks and two transfers are required for every page in the relation. During the joining phase each partition of R and S are loaded into memory and apply BlockNestedLoop algorithm.

```
1:  procedure HashJoin
2:      //Partition Phase
3:      for all page p_R in R do                                   ▷ Repeat |R| times
4:          read p_R into buffer
5:          seeks += 1
6:          transfers += 1
7:          for all tuple t_R in p_R do                            ▷ Repeat for ρ_R times
8:              write t_R to buffer h_1(t_R)
9:              if buffer h_1(t_R) is full then                    ▷ probability = 1/ρ_R
10:                 Flush h_1(t_R) to disk
11:                 seeks += 1
12:                 transfers += 1
13:     for all page p_S in S do                                   ▷ Repeat |S| times
14:         read p_S into buffer
15:         seeks += 1
16:         transfers += 1
17:         for all tuple t_S in p_S do                            ▷ Repeat for ρ_S times
18:             write t_S to buffer h_1(t_S)
19:             if buffer h_1(t_S) is full then                    ▷ probability = 1/ρ_S
20:                 Flush h_1(t_S) to disk
21:                 seeks += 1
22:                 transfers += 1
23:     //Joining Phase
24:     for all partitions P_R and P_S of R and S do               ▷ Repeat k = (B − 1) times
25:         for all block of B − 1 pages b_R in P_R do             ▷ Repeat ⌈1/(B−1) * |R|/k⌉ times
26:             read b_R into B − 1 pages of buffer
27:             seeks += 1
28:             transfers += (B-1)
29:
30:             seek += 1                                          ▷ Only once to go to start of S
31:             for all page p_S in P_S do                         ▷ Repeat ⌈|S|/k⌉ times
32:                 read p_S into buffer
33:                 transfers += 1
34:                 for all tuple t_R in b_R do
35:                     for all tuple t_S in p_S do
36:                         if t_R and t_S match condition θ then
37:                             output(t_R, t_S)
```

$$\begin{aligned}
\text{Total seeks} &= |R| * \left(1 + \frac{\rho_R}{\rho_R}\right) + |S| * \left(1 + \frac{\rho_S}{\rho_S}\right) + (B-1) * \left(\left\lceil \frac{|R|}{(B-1)*(B-1)} \right\rceil * (1+1)\right) \\
&= 2 * (|R| + |S|) + 2 * \left\lceil \frac{|R|}{B-1} \right\rceil \qquad \triangleright \text{partitioning + joining} \\
\text{Total transfers} &= |R| * \left(1 + \frac{\rho_R}{\rho_R}\right) + |S| * \left(1 + \frac{\rho_S}{\rho_S}\right) + (B-1) * \left(\left\lceil \frac{|R|}{(B-1)*(B-1)} \right\rceil * \left((B-1) + \left\lceil \frac{|S|}{B-1} \right\rceil\right)\right) \\
&= 2 * (|R| + |S|) + \left(|R| + \left\lceil \frac{|R|}{B-1} \right\rceil * \left\lceil \frac{|S|}{B-1} \right\rceil\right) \qquad \triangleright \text{partitioning + joining}
\end{aligned}$$

5. Analyse the memory costs in terms of cache misses during the phase for in-memory joins after partitioning in algorithm (4). Assume that the cache is large enough to hold more than one tuple of each relation and that

there is no pre-fetching. For what values of ρ_R, ρ_S, etc. is it possible to perform the join computations faster than loading the next page from disk?

Solution: Assuming that the hash function partitioned tuples uniformly into $k = (B-1)$ partitions, each partition of R and S has a size of $\frac{|R|}{k}$ and $\frac{|S|}{k}$ pages respectively. We have a cache miss in every iteration over the tuples t_R in the page loaded in the buffer memory pages. After the initial miss, accessing t_R while iterating over tuples of S will be cache hits. We also have a cache miss while iterating over tuples t_S as well.

```
1:  procedure InMemoryJoin(buffer)
2:      for all partitions P_R and P_S of R and S do                    ▷ Repeat k = B − 1 times
3:          for all blocks of (B − 1) pages b_R in P_R do               ▷ Repeat for ⌈|R| / ((B − 1) ∗ k)⌉ times
4:              seeks += 1
5:              transfers += B-1
6:
7:              seeks += 1                                              ▷ Only once to go to the start of S
8:              for all page p_S in P_S do                              ▷ Repeat for ⌈|S| / k⌉ times
9:                  transfers += 1
10:                 for all tuple t_R in b_R do                         ▷ Repeat for ρ_R ∗ (B − 1) times
11:                     cache miss += 1                                 ▷ access t_R
12:                     for all tuple t_S in p_S do                     ▷ Repeat for ρ_S times
13:                         cache miss += 1                             ▷ access t_S
14:                         if t_R and t_S match condition θ then       ▷ t_R already in cache
15:                             output(t_R, t_S)
```

We analyse the cost for every iteration that loads a page from S and processes it.

$$\begin{aligned}\text{Disk transfer cost} &= 100\ \mu s\\ \text{Cache misses} &= \rho_R * (B-1) * (1 + \rho_S * 1)\\ \text{Memory cost} &= \text{Cache misses} * 100\text{ ns}\\ &= 100 * \rho_R * (B-1) * (1+\rho_S)\text{ ns}\end{aligned}$$

Memory cost is less than transfer cost only if

$$\begin{aligned}100 * \rho_R * (B-1) * (1+\rho_S) &< 100,000\\ \rho_R * (1+\rho_S) * (B-1) &< 1000.\end{aligned}$$

6. Given a hash function h_2 defined for tuples of one of the relations, explain how the in-memory join can be optimized in case of algorithm (4). Can h_1 be same as h_2 ? Why / Why not?

   Solution: Assume h_2 partitions into $k_2 = B-1$ buckets. We use one page to buffer the input from partitions of R and $B-1$ for hash table using h_2. We assume that every partition of R fits in the buffer and that no page allocated to the hash table overflows while hashing any partition of R $\left(\frac{|R|}{k_1} < B-1\right)$ where k_1 is the number of partitions created in the first phase by h_1.

   While building hash table for each partition of R, since we assumed that there are no page overflows, and therefore no flushes to disk, all pages can be read one after the other with only one seek in total. Similarly, while scanning the partition of S, as long as the join computation over tuples of one page occurs finishes before the next page of S is loaded, there is only one seek.

```
1: procedure GraceHashJoinProbe(buffer)
2:     for all partitions P_R and P_S of R and S do                    ▷ Repeat k_1 = B − 1 times
3:         seeks += 1
4:         for all page p_R in P_R do                                  ▷ Repeat |R|/k_1 times
5:             read p_R into buffer
6:             transfers += 1
7:             for all tuple t_R in p_R do                             ▷ Repeat ρ_R times
8:                 write t_R to buffer h_2(t_R)                        ▷ partitions into k_2 = B − 1 buckets with no overflow
9:         seeks += 1
10:        for all page p_S in P_S do                                  ▷ Repeat |S|/k_1 times
11:            read p_S into buffer
12:            transfers += 1
13:            for all tuple t_S in p_S do
14:                for all tuple t_R in buffer h_2(t_S) do
15:                    if t_R and t_S match condition θ then
16:                        output(t_R, t_S)
```

We cannot use h_2 same as h_1 as we are using h_2 for every bucket partitioned by h_1. If we use h_1 again, all tuples will return same hash value, making the second hash pointless.

$$\begin{aligned}
\text{Total seeks in probe phase} &= (B-1)*(1+1) = 2*(B-1) \\
\text{Total transfers in probe phase} &= (B-1)*\left(\frac{|R|}{B-1}+\frac{|S|}{B-1}\right) \\
&= |R|+|S|
\end{aligned}$$

Grace Hash Join:

$$\begin{aligned}
\text{Total seeks} &= 2*(|R|+|S|)+2*(B-1) && \triangleright \text{partitioning + probing} \\
&= 2*(|R|+|S|+B-1) \\
\text{Total transfers} &= 2*(|R|+|S|)+(|R|+|S|) && \triangleright \text{partitioning + probing} \\
&= 3*(|R|+|S|)
\end{aligned}$$

Exercise 2: Sorting

Consider the algorithm used for sorting databases larger than the memory as seen in the lecture - External Merge Sort.

1. Give a pseudo-code for the algorithm that uses 2-way merge and 3 buffer pages in memory. Also estimate its cost in terms of the total number of pages transferred from disk as well as disk seeks.

   Solution: The algorithm builds runs of size 1 by reading a page, sorting it in memory and writing it back to disk. Since read and write is involved to different pages, a seek is required for both operations for every page. Then, during the merge phase, while merging runs, the algorithm reads each run one page at a time and writes back the merged run. In the worst case, a seek would be required before fetching every page to be read as well as before writing the merged page. In every pass of the loop, the number of runs gets halved and the size of each run gets doubled and their product remains the same with value $|R|$. To simplify the cost calculation, the cost of writing the final output to disk is included in the cost.

```
1:  procedure ExternalMergeSort
2:      for all page p_R ∈ R do                                   ▷ Repeat |R| times
3:          Read p_R into buffer
4:          seeks += 1
5:          transfers += 1
6:          Sort p_R in memory
7:          Write back to disk as runs of size 1
8:          seeks += 1
9:          transfers += 1
10:     numRuns = |R|
11:     runSize = 1
12:     while numRuns > 1 do                                      ▷ Repeat ⌈log_2(|R|)⌉ times
13:         for r = 1 to numRun; r+=2 do                          ▷ Repeat numRun/2 times
14:             Read run r and run r + 1 to buffer one page at a time
15:             seek += (1 + 1) * runSize                         ▷ worst case
16:             transfers += (1 +1) * runSize
17:             Merge runs and write to output buffer
18:             Write output buffer one page at a time as run of double size
19:             seek += 1 * (2 * runSize)                         ▷ worst case
20:             transfer += 1 * (2 * runSize)
21:         numRuns /= 2
22:         runSize *= 2
```

$$\text{Total transfers} = \text{Total seeks} = |R| * 2 + \sum_{\substack{numRuns=2 \\ \text{multiply by 2}}}^{|R|} \frac{numRuns}{2} * \left(2 * \frac{|R|}{numRuns} * 2\right) = |R| * 2 + \sum_{\substack{numRuns=2 \\ \text{multiply by 2}}}^{|R|} (2 * |R|)$$

$$= 2 * |R| * (1 + \log_2 |R|)$$

2. Modify the algorithm to make use of B buffer pages in memory to minimize disk I/Os. What is the total I/O cost (seeks and transfers) ?

   Solution: During the build phase, we can load B pages at the same time and sort them in memory. Thus, the initial size of the run is B. During the merge phase, we only need to keep one page for output and therefore can perform (B-1) way merge of runs. To simplify the cost calculation, the cost of writing the final output to disk is included in the cost.

```
1:  procedure ExternalMergeSortWithBuffer
2:      for all block of B pages b_R ∈ R do                ▷ Repeat ⌈|R|/B⌉ times
3:          Read b_R into buffer
4:          seeks += 1
5:          transfers += B
6:          Sort it in memory
7:          Write back to disk as runs of size B
8:          seeks += 1
9:          transfers += B
10:     numRuns = ceil(|R|/B)
11:     runSize = B
12:     while numRuns > 1 do                               ▷ Repeat ⌈log_{B-1}(|R|/B)⌉ times
13:         for r = 1 to numRun; r+= (B − 1) do            ▷ Repeat numRun/(B − 1) times
14:             Read run r, r + 1, ..., (r + B − 2) to (B − 1) buffer pages each one page at a time
15:             seek += (B − 1)∗ runSize                   ▷ worst case
16:             transfers += (B − 1)∗ runSize
17:             Merge runs and write to output buffer
18:             Write output buffer one page at a time as run of (B − 1) times previous size
19:             seeks += (B − 1)∗ runSize                  ▷ worst case
20:             transfers += (B − 1)∗ runSize
21:         numRuns / = (B − 1)
22:         runSize ∗ = (B − 1)
```

$$
\begin{aligned}
\text{Total seeks} &= \left\lceil \frac{|R|}{B} \right\rceil * 1 + \sum_{\substack{numRuns=(B-1) \\ \text{multiply by (B-1)}}}^{\left\lceil \frac{|R|}{B} \right\rceil} \frac{numRuns}{B-1} * \left((B-1) * \frac{|R|}{numRuns} * 2 \right) \\
&= \left\lceil \frac{|R|}{B} \right\rceil + \sum_{\substack{numRuns=(B-1) \\ \text{multiply by (B-1)}}}^{\left\lceil \frac{|R|}{B} \right\rceil} (2 * |R|) \\
&= \left\lceil \frac{|R|}{B} \right\rceil + 2 * |R| * \left(\log_{B-1} \left\lceil \frac{|R|}{B} \right\rceil \right) \\
\text{Total transfers} &= \left\lceil \frac{|R|}{B} \right\rceil * (2 * B) + \sum_{\substack{numRuns=(B-1) \\ \text{multiply by (B-1)}}}^{\left\lceil \frac{|R|}{B} \right\rceil} \frac{numRuns}{B-1} * \left((B-1) * \frac{|R|}{numRuns} * 2 \right) \\
&= |R| * 2 + \sum_{\substack{numRuns=(B-1) \\ \text{multiply by (B-1)}}}^{\left\lceil \frac{|R|}{B} \right\rceil} (2 * |R|) \\
&= 2 * |R| * \left(1 + \log_{B-1} \left\lceil \frac{|R|}{B} \right\rceil \right)
\end{aligned}
$$

3. Analyse the cost of memory operations in terms of cache misses after the pages are loaded into memory. The cache has enough capacity to hold at least B tuples of the relation along with some meta-data. The cost of a cache access is 4 ns and that of memory access is 100 ns. For what values of B do these costs become non-negligible ($> 10\%$) compared to disk seek and transfer costs? Assume that each tuple in the relation is of size 64 bytes and that page size is 8 KiB.

   Solution:

   In every round of merge, we take $B - 1$ runs at a time and merge them. Only one page of each run resides in the memory buffer page, and we compare only one tuple from each run. The first time a tuple is compared with any other tuple, it has to be fetched from memory, and after that for every comparison until it is written to output, it is fetched from cache. The algorithm stores the index of the tuple being processed from each run in

```
1: procedure MERGE(buffer) // One pass of k-way merge
2:     for r = 1 to numRun; r+= (B − 1) do                ▷ Repeat numRun/(B − 1) times
3:         for i = 1 to B do cursor[i] = 1
4:         repeat
5:             (val,k′) = min(buffer[k][cursor[k]] for k = 1 to (B − 1))
6:                                  ▷ All values are in cache except for the tuple replacing previous min
7:             memory access += 1   ▷ On average, reading one tuple first time from memory
8:             cache access += 2 * (B-1)                  ▷ cursor[k], buffer[k][cursor[k]]
9:             cursor[k′] += 1
10:            cache access += 2                          ▷ read and write
11:            if cursor[k′] > ρ_R then                   ▷ prob = 1/ρ_R
12:                read new page into buffer[k′]
13:                seeks += 1
14:                transfers += 1
15:                cursor[k′] = 1
16:                cache access += 1
17:
18:            buffer[B][cursor[B]] = val
19:            cache access += 2                          ▷ cursor[B], buffer[B][cursor[B]]
20:            cursor[B] += 1
21:            cache access += 2                          ▷ read and write
22:            if cursor[B] > ρ_R then                    ▷ prob = 1/ρ_R
23:                write output buffer to disk
24:                seeks += 1
25:                transfers += 1
26:                cursor[B] = 1
27:                cache access += 1
28:        until All pages in every run has been processed   ▷ ≈ ρ_R * (B − 1) * runSize iterations
```

the `cursor` array which resides in the cache as it is used very frequently. We know that each page of R is read and written back once in the whole round.

$$
\begin{aligned}
\text{Disk seeks} = \text{Disk transfers} &= \frac{numRuns}{B-1} * (\rho_R * (B-1) * runSize) * \frac{1}{\rho_R} * 2 \\
&= 2 * numRuns * runSize = 2 * |R| \\
\text{Memory operations} &= \frac{numRuns}{B-1} * (\rho_R * (B-1) * runSize) \\
&= numRuns * runSize * \rho_R = |R| * \rho_R \\
\text{Cache operations} &= \frac{numRuns}{B-1} * \left(\rho_R * (B-1) * runSize * \left(2 * (B-1) + 4 + \frac{1}{\rho_R} * 2\right)\right) \\
&= numRuns * runSize * \rho_R * \left(2 * (B-1) + 4 + \frac{1}{\rho_R} * 2\right) \\
&\approx |R| * (2 * \rho_R * B) \\
\text{Disk cost} &= \text{Disk seeks} * 10 \text{ ms} + \text{Disk transfers} * 0.1 \text{ ms} \\
&= 2 * |R| * 10.1 \text{ ms} \\
\text{Memory + Cache cost} &= \text{Memory operations } * 100 \text{ ns} + \text{Cache operations} * 4 \text{ ns} \\
&= |R| * \rho_R * 100 \text{ ns} + |R| * 2 * B * 4 \text{ ns} \\
&= |R| * \rho_R * (100 + 8 * B) \text{ ns} \\
\rho_R &= 8 * 1024/64 = 128 \\
\text{Ratio of memory+cache cost to disk cost } &= \frac{20.2 * 10^6}{\rho_R * (100 + 8 * B)} = \frac{128 * (100 + 8 * B)}{20.2 * 10^6} \\
\frac{128 * (100 + 8 * B)}{20.2 * 10^6} &> 0.1 \\
12800 + 1024 * B &> 20.2 * 10^5 \\
B &\gtrapprox 2000
\end{aligned}
$$

Exercise 3: Matrix Multiplication

Consider multiplication of two floating point square matrices each of dimension 4,096 on a single core. The RAM has an access latency of 100 ns.

1. Write pseudo-code for an naive in-memory matrix multiplication and estimate the total time when there are no caches.

   Solution:

```
1: procedure MMul(A,B,C)
2:     Initialize C with 0
3:     for i = 0 to 4095 do
4:         for j = 0 to 4095 do
5:             for k = 0 to 4095 do
6:                 C[i][j] += A[i][k] * B[k][j]
7:                 mem += 4                          ▷ read A,B,C write C
```

$$\text{Total time} = \text{mem} * 100 \text{ ns} = 4096 * 4096 * 4096 * 4 * 100 \text{ ns} = 100 * 2^{38} \text{ ns} \approx 27487 \text{ s}$$

2. Suppose an L1 read-cache of size 96 KiB was introduced with access latency of 4 ns that uses LRU eviction policy. Rewrite the above algorithm to be cache aware and estimate total running time.

   Solution: 96 KiB cache can accommodate 96 * 1024/ 8 = 12288 floating point elements in total = 4096 elements per matrix (A, B, and C) = 64×64. We divide the matrices into tiles of 64×64. We multiply tiles of A and B to get tiles of C. When multiplying two tiles, the first time an element is accessed, it is not in cache, and has to

be fetched from memory into cache. Successive accesses to the element will hit the cache. For each of the tile, only 64 * 64 memory accesses are made in 64 * 64 * 64 iterations of times, or once in 64 iterations. We assume that the cache is write back, and we only consider the cost of cache access while writing and ignore the memory cost.

```
1:  procedure MMul2(A,B,C)
2:      Initialize C with 0
3:      for I = 0 to 4095; I+= 64 do                          ▷ Repeat 4096/64 times
4:          for J = 0 to 4095; J += 64 do                     ▷ Repeat 4096/64 times
5:              for K = 0 to 4095; K +=64  do                 ▷ Repeat 4096/64 times
6:                  for i = I to I + 64 do                    ▷ Repeat 64 times
7:                      for j = J to J + 64 do                ▷ Repeat 64 times
8:                          for k = K to K + 64 do            ▷ Repeat 64 times
9:                              C[i][j] += A[i][k] * B[k][j]
10:                             Memory += 3 * 1 / 64          ▷ access to memory first time
11:                             L1 cache + = 4                ▷ read A,B,C write C
```

$$\begin{aligned}\text{Memory} &= (64*64*64)*(64*64*64)*1/64*3 = 3*2^{30}\\ \text{L1} &= (64*64*64)*(64*64*64)*4 = 2^{38}\\ \text{Total time} &= \text{mem}*100\text{ ns} + L1*4\text{ ns} = 3*2^{30}*100\text{ ns} + 2^{38}*4\text{ ns}\\ &\approx 1421\text{ s}\end{aligned}$$

3. Suppose another level of cache of size 24 MiB with access latency of 20 ns and similar eviction policy is introduced. Rewrite the algorithm to consider this cache as well and estimate total running time

   Solution: 24 MiB cache size can accommodate 24 * 1024 * 1024/ 8 = 3 * 1024 * 1024 floating point elements in total = 1024 * 1024 elements per matrix (A, B, and C). We first divide the matrices into tiles of size 1024 × 1024 first before subdividing into 64 × 64. Data is loaded into L1 cache from L2 cache now instead of the main memory. For the bigger tile, 1024 * 1024 elements need to be loaded into L2 cache from memory in 16 * 16 * 16 * 64 * 64 * 64 iterations, or once in every 1024 iterations.

```
1:  procedure MMul3(A,B,C)
2:      Initialize C with 0
3:      for X = 0 to 4095; X+= 1024 do                        ▷ Repeat 4096/1024 times
4:          for Y = 0 to 4095; Y +=1024 do                    ▷ Repeat 4096/1024 times
5:              for Z = 0 to 4095; Z +=1024  do               ▷ Repeat 4096/1024 times
6:                  for I = X to X+1024; I+= 64 do            ▷ Repeat 1024/64 times
7:                      for J = Y to Y + 1024; J += 64 do     ▷ Repeat 1024/64 times
8:                          for K = Z to Z + 1024; K +=64  do ▷ Repeat 1024/64 times
9:                              for i = I to I + 64 do        ▷ Repeat 64 times
10:                                 for j = J to J + 64 do    ▷ Repeat 64 times
11:                                     for k = K to K + 64 do ▷ Repeat 64 times
12:                                         C[i][j] += A[i][k] * B[k][j]
13:                                 Memory += 1/1024 * 3      ▷ first time accessing memory
14:                                 L2 cache += 1/64 * 3      ▷ what used to be memory now becomes L2
15:                                 L1 cache + = 4            ▷ read A,B,C write C
```

$$\begin{aligned}\text{Memory} &= (4*4*4)*(16*16*16)*(64*64*64)*1/1024*3 = 3*2^{26}\\ \text{L2} &= (4*4*4)*(16*16*16)*(64*64*64)*1/64*3 = 3*2^{30}\\ \text{L1} &= (4*4*4)*(16*16*16)*(64*64*64)*4 = 2^{38}\\ \text{Total time} &= \text{Memory}*100\text{ ns} + L2*20\text{ ns} + L1*4\text{ ns} = 3*2^{26}*100\text{ ns} + 3*2^{30}*40\text{ ns} + 2^{38}*4\text{ ns}\\ &\approx 1184\text{ s}\end{aligned}$$