Name: SOLUTIONS Student ID:

Final Exam

Computer Architecture (263-2210-00L)

ETH Zürich, Fall 2017

Prof. Onur Mutlu

Problem 1 (80 Points):	
Problem 2 (80 Points):	
Problem 3 (60 Points):	
Problem 4 (65 Points):	
Problem 5 (90 Points):	
Problem 6 (60 Points):	
Problem 7 (65 Points):	
Problem 8 (BONUS: 45 Points):	
Problem 9 (BONUS: 75 Points):	
Total (620 (500 + 120 bonus) Points):	

Examination Rules:

1. Written exam, 180 minutes in total.
2. No books, no calculators, no computers or communication devices. 6 pages of handwritten notes are allowed.
3. Write all your answers on this document, space is reserved for your answers after each question. Blank pages are available at the end of the exam.
4. Clearly indicate your final answer for each problem. Answers will only be evaluated if they are readable.
5. Put your Student ID card visible on the desk during the exam.
6. If you feel disturbed, immediately call an assistant.
7. Write with a black or blue pen (no pencil, no green or red color).
8. Show all your work. For some questions, you may get partial credit even if the end result is wrong due to a calculation mistake.
9. Please write your initials at the top of every page.

Tips:

- **Be cognizant of time.** Do not spend too much time on one question.
- **Be concise.** You may be penalized for verbosity.
- **Show work when needed.** You will receive partial credit at the instructors' discretion.
- **Write legibly.** Show your final answer.

This page intentionally left blank

1 DRAM Refresh [80 points]

A memory system is composed of eight banks, and each bank contains 32K rows. The row size is 8 KB.

Every DRAM row refresh is initiated by a command from the memory controller, and it refreshes a single row. Each refresh command keeps the command bus busy for 5 ns.

We define *command bus utilization* as a fraction of the total time during which the command bus is busy due to refresh.

The retention time of each row depends on the temperature (T). The rows have different retention times, as shown in the following Table 1:

Retention Time	Number of rows
$(128 - T)$ ms, $0°C \leq T \leq 128°C$	2^8 rows
$2 * (128 - T)$ ms, $0°C \leq T \leq 128°C$	2^{16} rows
$4 * (128 - T)$ ms, $0°C \leq T \leq 128°C$	all other rows
$8 * (128 - T)$ ms, $0°C \leq T \leq 128°C$	2^8 rows

Table 1: Retention time

1.1 Refresh Policy A [20 points]

Assume that the memory controller implements a refresh policy where all rows are refreshed with a fixed refresh interval, which covers the worst-case retention time (Table 1).

(a) [5 points] What is the maximum temperature at which the DRAM can operate reliably with a refresh interval of 32 ms?

T=96°C.

Explanation. Looking at the first column of Table 1, it is obvious that the DRAM rows with the least retention time (i.e., the first row of the table) can operate at temperatures up to 96°C (128 - 96) when refreshed at fixed 32 ms refresh period.

(b) [15 points] What command bus utilization is directly caused by DRAM refreshes (with refresh interval of 32 ms)?

4.096%.

Explanation. The time that the command bus spends on refresh commands in 32ms is 2^{18}*5ns, where 2^{18} is the number of rows (8 banks, 32K rows per banks), and 5ns is the time that the bus is busy for each refresh command.

Utilization = $\frac{2^{18} * 5ns}{32ms}$ = 4.096%

1.2 Refresh Policy B [15 points]

Now assume that the memory controller implements a refresh policy where all rows are refreshed only *as frequently as required* to correctly maintain their data (Table 1).

(a) [15 points] How many refreshes are performed by the memory controller during a 1.024 second period? (with T=64°C)

The number of refreshes in 1.024 seconds is 1313280.

Explanation. 2^8 rows are refreshed every 64ms $\Longrightarrow$ $(1024/64) * 2^8 = 2^{12}$ refresh commands in 1.024 seconds.
2^{16} rows are refreshed every 128ms $\Longrightarrow$ $(1024/128) * 2^{16} = 2^{19}$ refresh commands in 1.024 seconds.
2^8 rows are refreshed every 512ms $\Longrightarrow$ $(1024/512) * 2^8 = 2^9$ refresh commands in 1.024 seconds.

The remaining rows are $2^{18} - (2^8 + 2^{16} + 2^8) = 2^{18} - 2^{16} - 2^9$, so: $2^{18} - 2^{16} - 2^9$ rows are refreshed every 256ms $\Longrightarrow$ $(1024/256) * (2^{18} - 2^{16} - 2^9) = 2^{20} - 2^{18} - 2^{11}$ refresh commands in 1.024 second.

Total: $2^{12} + 2^{19} + 2^9 + 2^{20} - 2^{18} - 2^{11} = 1313280$ refresh commands in 1.024 second.

1.3 Refresh Policy C [25 points]

Assume that the memory controller implements an even smarter policy to refresh the memory. In this policy, the refresh interval is fixed, and it covers the worst-case retention time (64ms), as the refresh policy in part 1.1. However, as an optimization, a row is refreshed only if it has *not* been **accessed** during the past refresh interval. For maintaining correctness, if a cell reaches its maximum retention time without being refreshed, the memory controller issues a refresh command.

(a) [5 points] Why does a row *not* need to be refreshed if it was accessed in the past refresh interval?

The charge of a DRAM cell is restored when it is accessed.

(b) [20 points] A program accesses all the rows repeatedly in the DRAM. The following table shows the access interval of the rows, the number of rows accessed with the corresponding access interval, and the retention times of the rows that are accessed with the corresponding interval.

Access interval	**Number of rows**	**Retention times**
1ms	2^{16} rows	64ms, 128ms or 256ms
60ms	2^{16} rows	64ms, 128ms or 256ms
128ms	all other rows	128ms or 256ms

What command bus utilization is directly caused by DRAM refreshes?

Command bus utilization: 0.512%.

Explanation. The rows that are accessed every 1 ms and every 60 ms do *not* need to be refreshed.

The remaining rows (2^{17}) are accessed once every two refresh intervals (128 ms). So, the command bus utilization is $\frac{2^{17} * 5ns * 100}{128ms} = 0.512\%$

1.4 Refresh Policy D [20 points]

Assume that the memory controller implements an approximate mechanism to reduce refresh rate using Bloom filters, as we discussed in class. For this question we assume the retention times in Table 1 with a constant temperature of $64°C$.

One Bloom filter is used to represent the set of all rows that require a 64 ms refresh rate.

The memory controller's refresh logic is modified so that on every potential refresh of a row (every 64 ms), the refresh logic probes the Bloom filter. If the Bloom filter probe results in a "hit" for the row address, then the row is refreshed. Any row that does *not* hit in the Bloom filter is refreshed at the default rate of once per 128 ms.

(a) [20 points] The memory controller performs 2107384 refreshes in total across the channel over a time interval of 1.024 seconds. What is the false positive rate of the Bloom filter? (NOTE: False positive rate = Total number of false positives / Total number of accesses).

- Hint: $2107384 = 2^3 * (2^{18} + 2^{11} - 2^{10} + 2^9 - 2^8 - 1)$

False Positive rate = $(2^{10} - 1)/2^{18}$.

Explanation. First, there are 2^8 rows that need to be refreshed every 64ms. All the other rows ($2^{18} - 2^8$) will be refreshed every 128ms.

At 64 ms, we have 2^8 rows which actually require this rate, plus P out of the $2^{18} - 2^8$ other rows which will be false-positives. In a period of 1.024s, this bin is refreshed 16 times. The number of refreshes for this bin is $16 * (2^8 + P * (2^{18} - 2^8))$.

At 128 ms we have the rest of the rows which are $2^{18} - 2^8$ minus the false positives refreshed in the 64ms bin. So, the number of rows are $2^{18} - 2^8 - P * (2^{18} - 2^8)$. In a period of 1.024s, this bin is refreshed 8 times. The number of refreshes for this bin is $8 * ((2^{18} - 2^8) - (P * (2^{18} - 2^8))$

Thus, in total we have $16 * (2^8 + P * (2^{18} - 2^8)) + 8 * ((2^{18} - 2^8) - P * (2^{18} - 2^8)) = 2107384$. Solving the equation, P=2^{-8}.

Finally, False positive rate = Total number of false positives / Total number of accesses = $P * (2^{18} - 2^8))/2^{18} = 2^{-8} * (2^{18} - 2^8)/2^{18} = (2^{10} - 1)/2^{18}$

2 DRAM Scheduling and Latency [80 points]

You would like to understand the configuration of the DRAM subsystem of a computer using reverse engineering techniques. Your current knowledge of the particular DRAM subsystem is limited to the following information:

- The physical memory address is 16 bits.
- The DRAM subsystem consists of a single channel and 4 banks.
- The DRAM is byte-addressable.
- The most-significant 2 bits of the physical memory address determine the bank.
- The DRAM command bus operates at 500 MHz frequency.
- The memory controller issues commands to the DRAM in such a way that *no command* for servicing a *later* request is issued before issuing a READ command for the current request, which is the oldest request in the request buffer. For example, if there are requests A and B in the request buffer, where A is the older request and the two requests are to different banks, the memory controller does *not* issue an ACTIVATE command to the bank that B is going to access *before* issuing a READ command to the bank that A is accessing.

You realize that you can observe the memory requests that are waiting to be serviced in the request buffer. At a particular point of time, you take the snapshot of the request buffer and you observe the following requests in the request buffer.

Requests in the request buffer (in descending order of request age, where the oldest request is on the top):

time ↓

```
Read 0x4C80
Read 0x0140
Read 0x4EC0
Read 0x8000
Read 0xF000
Read 0x803F
Read 0x4E80
```

At the same time you take the snapshot of the request buffer, you start probing the DRAM command bus. You observe the DRAM command type and the cycle (relative to the first command) at which the command is seen on the DRAM command bus. The following are the DRAM commands you observe on the DRAM bus while the requests above are serviced.

```
Cycle 0 --- PRECHARGE
Cycle 6 --- ACTIVATE
Cycle 10 --- READ
Cycle 11 --- READ
Cycle 21 --- PRECHARGE
Cycle 27 --- ACTIVATE
Cycle 31 --- READ
Cycle 32 --- ACTIVATE
Cycle 36 --- READ
Cycle 37 --- READ
Cycle 38 --- READ
Cycle 42 --- PRECHARGE
Cycle 48 --- ACTIVATE
Cycle 52 --- READ
```

Answer the following questions using the information provided above.

(a) [15 points] What are the following DRAM timing parameters used by the memory controller, in terms of nanoseconds?

i) ACTIVATE-to-READ latency

8 ns.

Explanation. After issuing the ACTIVATE command at cycle 6, the memory controller waits until cycle 10, which indicates that the ACTIVATE-to-READ latency is 4 cycles. The command bus operates at 500 MHz, so it has 2 ns clock period. Thus, the ACTIVATE-to-READ is $4 * 2 = 8$ ns.

ii) ACTIVATE-to-PRECHARGE latency

30 ns.

Explanation. The bank activated at cycle 6 is precharged at cycle 21. Although the memory controller is idle after cycle 11, it waits until cycle 21, which means the ACTIVATE-to-PRECHARGE latency restricts the memory controller from issuing the PRECHARGE command earlier. Thus, the ACTIVATE-to-PRECHARGE latency is 15 cycles = 30 ns.

iii) PRECHARGE-to-ACTIVATE latency

12 ns.

Explanation. The PRECHARGE-to-ACTIVATE latency can be easily seen in the first two commands at cycles 0 and 6. The PRECHARGE-to-ACTIVATE latency is 6 cycles = 12 ns.

(b) [20 points] What is the row size in bytes? Explain your answer.

64 bytes.

Explanation. The Read request to address 0x803F (to Bank 2) does not require an ACTIVATE command, which means there is a row hit for that access. The open row was activated by the command issued for the request to the address 0x8000. That means the target rows of both requests should be the same. When we look at the binary form of those addresses, we see that the least significant 6 bits are different (000000 for 0x8000 and 111111 for 0x803F). That means at least 6 of the least significant bits should be the column bits.
Later, when we look at the commands issued for the requests to 0x4ECO and 0x4E80, we see that for both of those requests, the memory controller has opened a new row. Thus, the target rows of those requests should be different. Since only the 6th bit (assuming the least significant bit is the 0th bit) is different between the two addresses, the 6th bit should be part of the row address. So, of the 6th bit is part of the row address, the number of columns bits should be 6 or less. As we previously found from the requests to 0x8000 and 0x803F that the number of column bits should be at least 6, combining those two findings we can say that the number of column bits should be exactly 6. Thus, the row size is $2^6 = 64$ bytes.

(c) [20 points] What is the status of the banks *prior* to the execution of any of the the above requests? In other words, which rows from which banks were open immediately prior to issuing the DRAM commands listed above? Fill in the table below indicating whether a bank has an open row, and if there is an open row, specify its address. If there is not enough information to infer the open row address, write *unknown*.

	Open or Closed?	**Open Row Address**
Bank 0	Open	5
Bank 1	Open	Unknown
Bank 2	Closed	—
Bank 3	Open	192

Explanation. By decoding the accessed addresses we can find which bank and row each access targets. Looking at the commands issued for those requests, we can determine which requests needed PRECHARGE (row buffer conflict, the initially open row is unknown in this case), ACTIVATE (the bank is initially closed), or directly READ (the bank is initially open and the open row is the same as the one that the request targets).

time

```
0x4C80 → Bank:  1, Row:  50 (PRECHARGE first. Any row other than 50 could be opened.)
0x0140 → Bank:  0, Row:  5 (Row hit, so Bank 0 must have row 5 open.)
0x4EC0 → Bank:  1, Row:  59
0x8000 → Bank:  2, Row:  0 (ACTIVATE is issued first. That means the bank was already closed.)
0xF000 → Bank:  3, Row:  192 (Row hit, so Bank 3 must have row 192 open.)
0x803F → Bank:  2, Row:  0
0x4E80 → Bank:  1, Row:  58
```

(d) [25 points] To improve performance, you decide to implement the idea of Tiered-Latency DRAM (TL-DRAM) in the DRAM chip. Assume that a bank consists of a single subarray. With TL-DRAM, an entire bank is divided into a near-segment and far-segment. When accessing a row in the near-segment, the ACTIVATE-to-READ latency *reduces* by 2 cycles and the ACTIVATE-to-PRECHARGE latency reduces by 5 cycles. When accessing a row in the far-segment, the ACTIVATE-to-READ latency *increases* by 1 cycle and the ACTIVATE-to-PRECHARGE latency increases by 2 cycles.

Assume that the rows in the near-segment have smaller row ids compared to the rows in the far-segment. In other words, physical memory row addresses 0 through $N-1$ are the near-segment rows, and physical memory row addresses N through $M-1$ are the far-segment rows.

If the above DRAM commands are issued 5 cycles faster with TL-DRAM compared to the baseline (the last command is issued in cycle 47), how many rows are in the near-segment? Show your work.

59 rows have to be in the near segment.

Explanation. There should 59 rows in the near-segment (rows 0 to 58) since rows until row id 58 need to be accessed with low latency to get 5 cycle reduction. Rows 59 and 192 are in the far-segment, thus latency for accessing them increases slightly.

Here is the new command trace:

```
Cycle 0 -- PRECHARGE - Bank 1
Cycle 6 -- ACTIVATE - Bank 1, Row 50, near segment
Cycle 8 -- READ - Bank 1
Cycle 9 -- READ - Bank 0
Cycle 16 -- PRECHARGE - Bank 1
Cycle 22 -- ACTIVATE - Bank 1, Row 59, far segment
Cycle 27 -- READ - Bank 1
Cycle 28 -- ACTIVATE - Bank 2, Row 0
Cycle 30 -- READ - Bank 2
Cycle 31 -- READ - Bank 3
Cycle 32 -- READ - Bank 2
Cycle 39 -- PRECHARGE - Bank 1
Cycle 45 -- ACTIVATE - Bank 1, Row 58, near segment
Cycle 47 -- READ - Bank 1
```

3 Cache Reverse Engineering [60 points]

Consider a processor using a 4-block LRU-based L1 data cache. Starting with an empty cache, an application accesses three L1 cache blocks in the following order, where consecutive numbers (e.g., n, $n+1$, $n+2, ...$) represent the starting addresses of consecutive cache blocks in memory:

$$n \rightarrow n+2 \rightarrow n+4$$

3.1 Part I: Vulnerability [35 points]

A malicious programmer realizes she can reverse engineer the number of sets and ways in the L1 data cache by issuing *just* two more accesses and observing *only* the cache hit rate across these two accesses. Assume that she can insert the malicious accesses only after the above three accesses of the program.

1. [15 points] What are the next two cache block she should access? (e.g., [n+?, n+?])

There are two possible answers:

- $\boxed{\textbf{[n} \rightarrow \textbf{n+4]}}$
- $\boxed{\textbf{[n} \rightarrow \textbf{n+2]}}$

Explanation. There are three possible set/way configurations, shown below labeled by their respective sets/ways. Each configuration shows a drawing of the cache state after the three initial accesses. Rows and columns represent sets and ways, respectively, and the LRU address is shown for each occupied set:

(a) **(4 sets, 1 way)**

$(n+4)_{LRU}$
-
$(n+2)_{LRU}$
-

(b) **(2 sets, 2 ways)**

$n+4$	$(n+2)_{LRU}$
-	-

(c) **(1 set, 4 ways)**

n_{LRU}	$n+2$	$n+4$	-

At this point, all three caches have a 100% miss rate since they started cold. In order to differentiate the three cases with *just two* more accesses, we need to induce *different* hit/miss counts in each of the three types of caches. The only way this is possible is if one cache type experiences two hits, another two misses, and the last one hit and one miss.

Only two solutions exist to produce this case:

- $\boxed{\textbf{[n} \rightarrow \textbf{n+4]}}$
  (a) n miss, $n+4$ miss = 100% miss rate
  (b) n miss, $n+4$ hit = 80% miss rate
  (c) n hit, $n+4$ hit = 60% miss rate
- $\boxed{\textbf{[n} \rightarrow \textbf{n+2]}}$
  (a) n miss, $n+2$ hit = 80% miss rate
  (b) n miss, $n+2$ miss = 100% miss rate
  (c) n hit, $n+2$ hit = 60% miss rate

2. [10 points] How many L1 sets/ways would there be if the cache hit rate over the 2 extra instructions was:

There are two possible solutions due to the two possible solutions to part (1). They directly encode the hit/miss rates for the last two accesses shown in the solution above:

- Solution 1:

L1 hit rate	# sets	# ways
100%	1	4
50%	2	2
0%	4	1

- Solution 2:

L1 hit rate	# sets	# ways
100%	1	4
50%	4	1
0%	2	2

3. [10 points] What should the next two accesses be if the replacement policy had been Most Recently Used (MRU)? (e.g., [n+?, n+?])

There is no solution for just two more accesses because with an MRU policy, no permutation of two more accesses is able to assign a unique L1 hit rate to each of the three types of caches.

3.2 Part II: Exploitation [25 points]

Assuming the original cache design (i.e., with an LRU replacement policy) is using a 1-set (4-way) configuration, the malicious programmer decides to disrupt a high-security banking application, allowing her to transfer large amounts of foreign assets into her own secure Swiss account. By using a carefully designed concurrently running process to interfere with the banking application, she would like to induce slowdown, thereby exposing opportunity for her mischief.

Assume that the unmodified banking application issues the following access pattern, where each number represents X in $n + X$:

$$0 \rightarrow 6 \rightarrow 1 \rightarrow 7 \rightarrow 6 \rightarrow 1 \rightarrow 4 \rightarrow 0 \rightarrow 5 \rightarrow 0 \rightarrow 7 \rightarrow 4 \rightarrow 2 \rightarrow 7 \rightarrow 2 \rightarrow 4$$

1. [10 points] What is the unmodified banking application's cache hit rate?

$\frac{7}{16}$

2. [15 points] Now, assume that the malicious programmer knows the access pattern of the banking application. Using this information, she is able to inject a **single** extra cache access in between each of the banking application's accesses (i.e., she can interleave a malicious access pattern with the normal application's execution).

What is the minimum cache hit rate (not counting extra malicious accesses) that the malicious programmer can induce for the banking application?

$\boxed{\frac{2}{16}}$

Explanation. Since the malicious programmer wants to cause every cache access by the banking application to miss, she must ensure that any block brought into the cache by an application access is evicted before it is accessed again. The (1 set, 4 way) configuration causes a set conflict between any two possible blocks, so in order to force an eviction of a given line, she must ensure that *four* cache accesses to different, unique cache lines are performed. This guarantees that accessing the given line again results in a miss.

Because she can only insert one extra cache access in between each pair of normal application accesses, she cannot force an eviction of the first access in the two sequences "...0 → 5 → 0..." and "...2 → 7 → 2...". In these two sequences, accesses to the same cache block are separated by only one access. This means that the malicious programmer can only cause a total of three distinct cache block accesses in between the accesses to the same cache block, and therefore, she cannot prevent a cache hit.

Thus, despite the malicious programmer's best effort, the two cache accesses shown below in bold will still hit in the cache, resulting in an application cache hit rate of $\frac{2}{16}$.

0 → 6 → 1 → 7 → 6 → 1 → 4 → 0 → 5 → **0** → 7 → 4 → 2 → 7 → **2** → 4

Here is an example cache access sequence resulting in the solution hit rate, with the malicious injected accesses shown in blue:

0 → 2 → 6 → 3 → 1 → 4 → 7 → 5 → 6 → 2 → 1 → 3 → 4 → 6 → 0 → 7 →
5 → 1 → **0** → 2 → 7 → 3 → 4 → 5 → 2 → 6 → 7 → 0 → **2** → 1 → 4 → 3

Note that this is one of many possible such sequences.

4 Runahead Execution [65 points]

Assume an in-order processor that employs Runahead execution, with the following specifications:

- The processor enters Runahead mode when there is a cache miss.
- There is no penalty for entering and leaving the Runahead mode.
- There is a 64KB data cache. The cache block size is 64 bytes.
- Assume that the instructions are fetched from a separate dedicated memory that has zero access latency, so an instruction fetch never stalls the pipeline.
- The cache is 4-way set associative and uses the LRU replacement policy.
- A memory request that hits in the cache is serviced instantaneously.
- A cache miss is serviced from the main memory after X cycles.
- A cache block for the corresponding fetch is allocated *immediately* when a cache miss happens.
- The cache replacement policy does *not* evict the cache block that triggered entry into Runahead mode until after the Runahead mode is exited.
- The victim for cache eviction is picked at the same time a cache miss occurs, i.e., during cache block allocation.
- ALU instructions and Branch instructions take one cycle.
- Assume that the pipeline *never stalls* for reasons *other than data cache misses*. Assume that the conditional branches are always correctly predicted and the data dependencies do not cause stalls (except for data cache misses).

Consider the following program. Each element of Array A is one byte.

```
for(int i=0;i<100;i++){ \\ 2 ALU instructions and 1 branch instruction
    int m = A[i*16*1024]+1; \\ 1 memory instruction followed by 1 ALU instruction
    ... \\ 26 ALU instructions
}
```

(a) [20 points] After running this program using the processor specified above, you find that there are 66 data cache hits. What are **all** the possible values of the cache miss latency X? You can specify all possible values of X as an inequality. Show your work.

$61 < X < 93$.

Explanation. The program makes 100 memory accesses in total. To have 66 cache hits, a cache miss needs to be followed by 2 cache hits. Hence, the Runahead engine needs to prefetch 2 cache blocks. After getting a cache miss and entering Runahead mode, the processor needs to execute 30 instructions to reach the next LD instruction. To reach the LD instruction 2 times, the processor needs to execute at least 62 instructions (2*(29 ALU + 1 Branch + 1 LD)) in Runahead mode. If the processor spends more than 92 cycles in Runahead mode, then it will prefetch 3 cache lines instead of two, which will cause the number of cache hits to be different. Thus, the answer is $61 < X < 93$.

(b) [20 points] Is it possible that *every* memory access in the program misses in the cache? If so, what are **all** possible values of X that will make all memory accesses in the program miss in the cache? If not, why? Show your work.

Yes, for $X < 31$ and $X > 123$.

Explanation. When X is equal to or smaller than 30 cycles, the processor will be in Runahead mode for insufficient amount of time to reach the next LD instruction (i.e., the next cache miss). Thus, none of the data will be prefetched and all memory accesses will get cache miss.
When X is larger than 123 cycles, the processor will prefetch 4 cache blocks. Since the prefetched cache blocks will map to the same cache set, the latest prefetched cache block will evict the first prefetched cache block in Runahead mode (note that the cache block that triggered Runahead execution remains in the cache due to the cache block replacement policy). This will cause a cache miss in the next iteration after leaving the Runahead mode. Thus, the accesses in the program will always miss in the cache.

(c) [25 points] What is the *minimum* number of cache misses that the processor can achieve by executing the above program? Show your work.

25 cache misses.

Explanation. When $92 < X < 124$, the Runahead engine will prefetch exactly 3 cache blocks that will be accessed after leaving the Runahead mode. It is the minimum number of misses that could be achieved since all cache blocks accessed by the program map to the same cache set and the cache is 4-way associative.

5 Interconnection Networks [90 points]

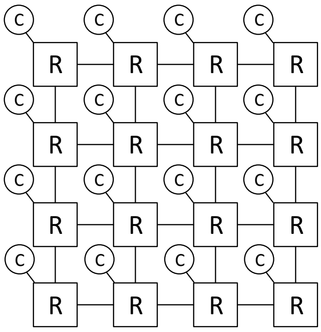
Suppose you would like to connect 2^N processors, and you are considering four different topologies:

- $\sqrt{2^N} \times \sqrt{2^N}$ 2D mesh
- $\sqrt{2^{N-2}} \times \sqrt{2^{N-2}}$ 2D concentrated mesh (Cmesh), where each router serves four processors
- $\sqrt{2^N} \times \sqrt{2^N}$ 2D torus
- Hypercube

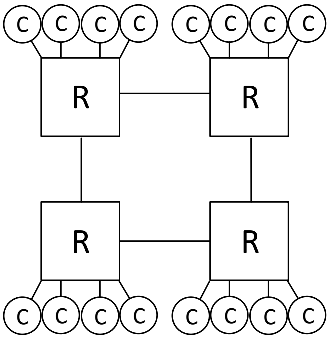
Please answer the following questions. Show your work.

(a) [20 points] For N = 4, please draw how each network looks like. You can use ... (three dots) to avoid repeated patterns.

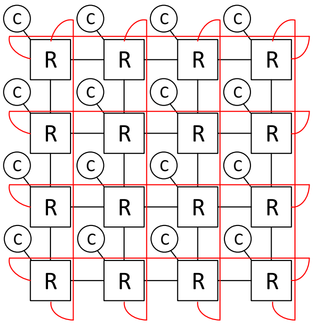
2D mesh



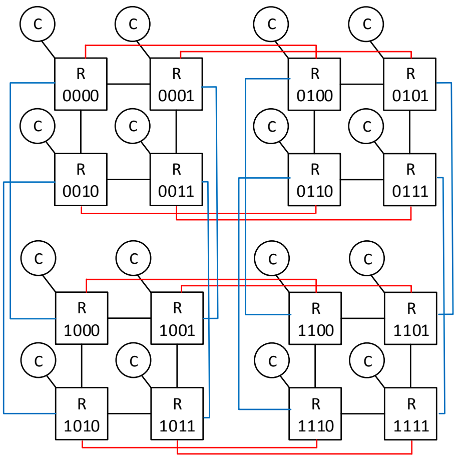
Cmesh



2D torus



Hypercube



For the remaining questions, *assume N = 8.*

(b) [20 points] For N = 8, calculate the number of network links for each network. (Hint: a single network link is bi-directional)

2D mesh: $2 \times (\sqrt{2^N} - 1)(\sqrt{2^N})$ links → $2 \times 15 \times 16 = 480$ links
Cmesh: $2 \times (\sqrt{2^{N-2}} - 1)(\sqrt{2^{N-2}})$ links → $2 \times 7 \times 8 = 112$ links
2D torus: $2 \times (\sqrt{2^N})(\sqrt{2^N})$ links → $2 \times 16 \times 16 = 512$ links
Hypercube: $2^N \times N/2$ links → $256 \times 8/2 = 1024$ links

(c) [25 points] For N = 8, calculate the number of input/output ports including the injection/ejection ports for *each router* in these topologies (Hint: give answer to all types of routers that exist in an irregular network).

2D mesh: (4+1) inputs/outputs, (3+1) inputs/outputs, and (2+1) inputs/outputs
Cmesh: (4+4) inputs/outputs, (3+4) inputs/outputs, and (2+4) inputs/outputs
2D torus: (4+1) inputs/outputs
Hypercube: N+1 inputs/outputs → 9 inputs/outputs

(d) [25 points] Assume a network link can be faulty. For each topology, what is the minimum possible number of faulty links that are needed to make at least one processor unreachable from any other processor?

2D mesh: 2 links
Cmesh: 2 links
2D torus: 4 links
Hypercube: $Nlinks$ → 8 links

6 Cache Coherence [60 points]

We have a system with 4 byte-addressable processors. Each processor has a private 256-byte, direct-mapped, write-back L1 cache with a block size of 64 bytes. Coherence is maintained using the Illinois Protocol (MESI), which sends an invalidation to other processors on writes, and the other processors invalidate the block in their caches if *the block is present* (NOTE: On a write hit in one cache, a cache block in Shared state becomes Modified in that cache).

Accessible memory addresses range from 0x50000000 − 0x5FFFFFFF. Assume that the offset within a cache block is 0 for all memory requests. We use a snoopy protocol with a shared bus.

Cosmic rays strike the MESI state storage in your coherence modules, causing the state of a *single* cache line to instantaneously change to another state. This change causes an inconsistent state in the system. We show below the initial tag store state of the four caches, *after* the inconsistent state is induced.

Inital State

Cache 0		
	Tag	*MESI state*
Set 0	0x5FFFFF	M
Set 1	0x5FFFFF	E
Set 2	0x5FFFFF	S
Set 3	0x5FFFFF	I

Cache 1		
	Tag	*MESI state*
Set 0	0x522222	I
Set 1	0x510000	S
Set 2	0x5FFFFF	S
Set 3	0x533333	S

Cache 2		
	Tag	*MESI state*
Set 0	0x5F111F	M
Set 1	0x511100	E
Set 2	0x5FFFFF	S
Set 3	0x533333	S

Cache 3		
	Tag	*MESI state*
Set 0	0x5FF000	E
Set 1	0x511100	S
Set 2	0x5FFFF0	I
Set 3	0x533333	I

(a) [15 points] What is the inconsistency in the above initial state? Explain with reasoning.

Cache 2, Set 1 should be in S state. Or Cache 3, Set 1 should be in I state.

Explanation. If the MESI protocol performs correctly, it is *not* possible for the same cache line to be in S and E states in different caches.

(b) [20 points] Consider that, after the initial state, there are several paths that the program can follow that access different memory instructions. In b.1-b.4, we will examine whether the followed path can potentially lead to incorrect execution, i.e., an incorrect result.

b.1) [10 points] Could the following path potentially lead to incorrect execution? Explain.

order	**Processor 0**	**Processor 1**	**Processor 2**	**Processor 3**
1^{st}			ld 0x51110040	
2^{nd}	st 0x5FFFFF40			
3^{rd}				st 0x51110040
4^{th}		ld 0x5FFFFF80		
5^{th}		ld 0x51110040		
6^{th}		ld 0x5FFFFF40		

No.

Explanation. The 3^{rd} instruction (st 0x51110040 in Processor 3) will invalidate the same line in Processor 2, and the whole system will be back to a consistent state (only one valid copy of 0x51110040 in the caches). Thus, the originally-inconsistent state does not affect the architectural state.

b.2) [10 points] Could the following path potentially lead to incorrect execution? Explain.

order	**Processor 0**	**Processor 1**	**Processor 2**	**Processor 3**
1^{st}				ld 0x51110040
2^{nd}	ld 0x5FFFFF00			
3^{rd}			ld 0x51234540	
4^{th}	st 0x5FFFFF40			
5^{th}				ld 0x51234540
6^{th}	ld 0x5FFFFF00			

Yes.

Explanation. The 1^{st} instruction could read invalid data. This would be the case if the cosmic-ray-induced change was from I to S in Cache 3 for cache line 0x51110040.

After some time executing a particular path (which could be a path *different* from the paths in parts b.1-b.4) and with no further state changes caused by cosmic rays, we find that the final state of the caches is as follows.

Final State

Cache 0		
	Tag	*MESI state*
Set 0	0x5FFFFF	M
Set 1	0x5FFFFF	E
Set 2	0x5FFFFF	S
Set 3	0x5FFFFF	E

Cache 1		
	Tag	*MESI state*
Set 0	0x5FF000	I
Set 1	0x510000	S
Set 2	0x5FFFFF	S
Set 3	0x533333	I

Cache 2		
	Tag	*MESI state*
Set 0	0x5F111F	M
Set 1	0x511100	E
Set 2	0x5FFFFF	S
Set 3	0x533333	I

Cache 3		
	Tag	*MESI state*
Set 0	0x5FF000	M
Set 1	0x511100	S
Set 2	0x5FFFF0	I
Set 3	0x533333	I

(c) [25 points] What is the *minimum* set of memory instructions that leads the system from the initial state to the final state? Indicate the set of instructions in order, and clearly specify the access type (ld/st), the address of each memory request, and the processor from which the request is generated.

The minimum set of instructions is:

(1) st 0x533333C0 // Processor 0

(2) ld 0x5FFFFFC0 // Processor 0

(3) ld 0x5FF00000 // Processor 1

(4) st 0x5FF00000 // Processor 3

Alternatively, as instructions (1)(2) and instructions (3)(4) touch different cache lines, we just need to keep the order between (1)(2), and between (3)(4). These are valid reorderings: (3)(4)(1)(2), (1)(3)(2)(4), (3)(1)(4)(2), (1)(3)(4)(2) or (3)(1)(2)(4).

Explanation.

(1) The instruction sets the line 0x533333C0 to M state in Cache 0, and invalidates the line 0x533333C0 in Cache 1 and Cache 2.

(2) The instruction evicts 0x533333C0 from Cache 0, and sets the line 0x5FFFFFC0 to E state in Cache 0.

(3) The instruction sets the line 0x5FF00000 to S state in Cache 1, as well as in Cache 3.

(4) The instruction sets the line 0x5FF00000 to M state in Cache 3, and it invalidates the line 0x5FF00000 in Cache 1.

7 Memory Consistency [65 points]

A programmer writes the following two C code segments. She wants to run them concurrently on a multicore processor, called SC, using two different threads, each of which will run on a different core. The processor implements *sequential consistency*, as we discussed in the lecture.

	Thread T0		Thread T1
Instr. T0.0	`a = X[0];`	Instr. T1.0	`Y[0] = 1;`
Instr. T0.1	`b = a + Y[0];`	Instr. T1.1	`*flag = 1;`
Instr. T0.2	`while(*flag == 0);`	Instr. T1.2	`X[1] *= 2;`
Instr. T0.3	`Y[0] += 1;`	Instr. T1.3	`a = 0;`

`X`, `Y`, and `flag` have been allocated in main memory, while `a` and `b` are contained in processor registers. A read or write to any of these variables generates a single memory request. The initial values of all memory locations and variables are 0. Assume each line of the C code segment of a thread is a *single* instruction.

(a) [15 points] What is the final value of `Y[0]` in the SC processor, after both threads finish execution? Explain your answer.

2.

Explanation. `Y[0]` is set equal to 1 by instruction T1.0. Then, it will be incremented by instruction T0.3. The sequential consistency model ensures that the operations of each individual thread are executed in the order specified by its program. Across threads, the ordering is enforced by the use of `flag`. Thread 0 will remain in instruction T0.2 until `flag` is set by T1.1, i.e., after `Y[0]` is initialized. So, instruction T0.3 must be executed after instruction T1.0, causing `Y[0]` to be first set to 1 and then incremented.

(b) [15 points] What is the final value of `b` in the SC processor, after both threads finish execution? Explain your answer.

0 or 1.

Explanation. There are *at least* two possible sequentially-consistent orderings that lead to *at most* two different values of `b` at the end:
Ordering 1: T1.0 → T0.1 - Final value = 1.
Ordering 2: T0.1 → T1.0 - Final value = 0.

With the aim of achieving higher performance, the programmer tests her code on a new multicore processor, called RC, that implements *weak consistency*. As discussed in the lecture, the weak consistency model has no need to guarantee a strict order of memory operations. For this question, consider a very weak model where there is *no* guarantee on the ordering of instructions as seen by different cores.

(c) [15 points] What is the final value of `Y[0]` in the RC processor, after both threads finish execution? Explain your answer.

1 or 2.

Explanation. Since there is no guarantee of a strict order of memory operations, as seen by different cores, instruction T1.1 could complete before or after instruction T1.0, from the perspective of the core that executes T0. If instruction T1.1 completes before instruction T1.0, from the perspective of the core that executes T0, instruction T0.3 could complete before or after instruction T1.0. Thus, there are three possible weakly-consistent orderings that lead to different values of `Y[0]` at the end:
Ordering 1 (from the perspective of T0): T1.0 → T1.1 → T0.3 - Final value = 2.
Ordering 2 (from the perspective of T0): T1.1 → T1.0 → T0.3 - Final value = 2.
Ordering 3 (from the perspective of T0): T1.1 → T0.3 → T1.0 - Final value = 1.

After several months spent debugging her code, the programmer learns that the new processor includes a `memory_fence()` instruction in its ISA. The semantics of `memory_fence()` is as follows for a given thread that executes it:

1. Wait (stall the processor) until *all* preceding memory operations from the thread complete in the memory system and become visible to other cores.
2. Ensure *no* memory operation from any later instruction in the thread gets executed before the `memory_fence()` is retired.

(d) [20 points] What *minimal* changes should the programmer make to the program above to ensure that the final value of `Y[0]` on RC is the same as that in part (a) on SC? Explain your answer.

Use memory fences before T1.1 and after T0.2.

Explanation. The memory fence before instruction T1.1 stalls thread 1 until instruction T1.0 has completed, i.e., ensures that `Y[0]` is initialized to 1 before the flag is set. Thread 0 waits in the loop T0.2 until the flag is set. The memory fence after instruction T0.2 ensures that instruction T0.3 will not happen until the memory fence is retired. Thus, instruction T0.3 will also complete *after* the flag is set. The modified code will be as follows:

	Thread T0
Instr. T0.0	`a = X[0];`
Instr. T0.1	`b = a + Y[0];`
Instr. T0.2	`while(*flag == 0);`
	memory_fence();
Instr. T0.3	`Y[0] += 1;`

	Thread T1
Instr. T1.0	`Y[0] = 1;`
	memory_fence();
Instr. T1.1	`*flag = 1;`
Instr. T1.2	`X[1] *= 2;`
Instr. T1.3	`a = 0;`

8 Prefetching [45 points]

An ETH student writes two programs A and B and runs them on 3 different toy machines, M1, M2, and M3, to determine the type of the prefetching mechanism used in each of these 3 machines. She observes programs A and B to have the following access patterns to cache blocks. Note that the addresses are *cache block addresses*, not byte addresses.

Program A: 27 accesses

```
a, a + 1, a + 2, a + 3, a + 4, a + 8, a + 16, a + 32, a + 64,
a, a + 1, a + 2, a + 3, a + 4, a + 8, a + 16, a + 32, a + 64,
a, a + 1, a + 2, a + 3, a + 4, a + 8, a + 16, a + 32, a + 64
```

Program B: 501 accesses

```
b, b + 2, b + 4, ...., b + 998, b + 1000
```

The student is able to measure the accuracy and coverage of the prefetching mechanism in each of the machines. The following table shows her measurement results:

	Machine M1		**Machine M2**		**Machine M3**	
	Coverage	Accuracy	Coverage	Accuracy	Coverage	Accuracy
Program A	6/27	6/27	0	0	1/3	9/26
Program B	499/501	499/501	0	0	499/501	499/500

The student knows the following facts about M1, M2, and M3 machines:

- The prefetcher prefetches into a fully-associative cache whose size is 8 cache blocks. The cache employs the FIFO (First-In First-Out) replacement policy.
- The prefetchers have large enough resources to detect and store access patterns.
- Each cache block access is separated long enough in time such that all prefetches issued can complete before the next access happens.
- There are 5 different possible choices for the prefetching mechanism:
  1) Markov prefetcher with a correlation table of 4 entries
  2) Markov prefetcher with a correlation table of 10 entries
  3) 1st-next-block prefetcher (degree = 1) – prefetches block N + 1 after seeing block N
  4) 4th-next-block prefetcher (degree = 1) – prefetches block N + 4 after seeing block N
  5) stride prefetcher
- None of the above-mentioned prefetchers employ confidence bits.
- The prefetchers start out with an empty table when each program A and B start execution.
- The prefetcher sends only one prefetch request after a program access (i.e., prefetch degree = 1).

Determine what type of prefetching mechanism each of the above-mentioned machines use:

Machine M1: 4th-next-block prefetcher

Machine M2: Markov prefetcher with a correlation table of 4 entries

Machine M3: Stride prefetcher

Extra space for explanation:

We calculate the accuracy and coverage for all 5 types of prefetchers, and then we can answer what prefetcher each machine is using:

The 5 prefetechers work in the following ways when running Application A:

Markov, table size=4: Coverage: 0, Accuracy: 0
a, a + 1, a + 2, a + 3, a + 4, a + 8, a + 16, a + 32, a + 64,
a, a + 1, a + 2, a + 3, a + 4, a + 8, a + 16, a + 32, a + 64,
a, a + 1, a + 2, a + 3, a + 4, a + 8, a + 16, a + 32, a + 64,

Markov, table size=10: Coverage: 17/27, Accuracy: 17/18
a, a + 1, a + 2, a + 3, a + 4, a + 8, a + 16, a + 32, a + 64,
a, $\underline{a + 1}$, $\underline{a + 2}$, $\underline{a + 3}$, $\underline{a + 4}$, $\underline{a + 8}$, $\underline{a + 16}$, $\underline{a + 32}$, $\underline{a + 64}$,
$\underline{a}$, $\underline{a + 1}$, $\underline{a + 2}$, $\underline{a + 3}$, $\underline{a + 4}$, $\underline{a + 8}$, $\underline{a + 16}$, $\underline{a + 32}$, $\underline{a + 64}$ |unused: a

1st-next-block: Coverage: 4/9, Accuracy: 4/9
a, $\underline{a + 1}$, $\underline{a + 2}$, $\underline{a + 3}$, $\underline{a + 4}$, a + 8, a + 16, a + 32, a + 64, |unused: a + 5, a + 9, a + 17, a + 33, a + 65,
a, $\underline{a + 1}$, $\underline{a + 2}$, $\underline{a + 3}$, $\underline{a + 4}$, a + 8, a + 16, a + 32, a + 64, |unused: a + 5, a + 9, a + 17, a + 33, a + 65,
a, $\underline{a + 1}$, $\underline{a + 2}$, $\underline{a + 3}$, $\underline{a + 4}$, a + 8, a + 16, a + 32, a + 64 |unused: a + 5, a + 9, a + 17, a + 33, a + 65

4th-next-block: Coverage: 6/27, Accuracy: 6/27
a, a + 1, a + 2, a + 3, $\underline{a + 4}$, $\underline{a + 8}$, a + 16, a + 32, a + 64,|unused: a + 5, a + 6, a + 7, a + 12, a + 20, a + 36, a + 68,
a, a + 1, a + 2, a + 3, $\underline{a + 4}$, $\underline{a + 8}$, a + 16, a + 32, a + 64,|unused: a + 5, a + 6, a + 7, a + 12, a + 20, a + 36, a + 68,
a, a + 1, a + 2, a + 3, $\underline{a + 4,}$ $\underline{a + 8}$, a + 16, a + 32, a + 64|unused: a + 5, a + 6, a + 7, a + 12, a + 20, a + 36, a + 68

Stride: Coverage: 1/3, Accuracy: 9/26
a, a + 1, $\underline{a + 2}$, $\underline{a + 3}$, $\underline{a + 4}$, a + 8, a + 16, a + 32, a + 64, |unused: a + 5, a + 12, a + 24, a + 48, a + 96,
a, a + 1, $\underline{a + 2}$, $\underline{a + 3}$, $\underline{a + 4}$, a + 8, a + 16, a + 32, a + 64, |unused: a - 64, a + 5, a + 12, a + 24, a + 48, a + 96,
a, a + 1, $\underline{a + 2}$, $\underline{a + 3}$, $\underline{a + 4}$, a + 8, a + 16, a + 32, a + 64 |unused: a - 64, a + 5, a + 12, a + 24, a + 48, a + 96

The 5 prefetechers work in the following ways when running Application B:

Markov, table size=4: Coverage: 0, Accuracy: 0
b, b + 2, b + 4, b + 6, b + 8, b + 10, ... , b + 998, b + 1000

Markov, table size=10: Coverage: 0, Accuracy: 0
b, b + 2, b + 4, b + 6, b + 8, b + 10, ... , b + 998, b + 1000

1st-next-block: Coverage: 0, Accuracy: 0
b, b + 2, b + 4, b + 6, b + 8, b + 10, ... , b + 998, b + 1000 |unused: b + 1, b + 3, ..., b + 999, b + 1001

4th-next-block: Coverage: 499/501, Accuracy: 499/501
b, b + 2, $\underline{b + 4}$, $\underline{b + 6}$, $\underline{b + 8}$, $\underline{b + 10}$, ..., $\underline{b + 998}$, $\underline{b + 1000}$ |unused: b +1002, b + 1004

Stride: Coverage: 499/501, Accuracy: 499/500
b, b + 2, $\underline{b + 4}$, $\underline{b + 6}$, $\underline{b + 8}$, $\underline{b + 10}$, ..., $\underline{b + 998}$, $\underline{b + 1000}$ |unused: b + 1002

9 BONUS: GPU Programming and Performance Analysis [75 points]

The following two program segments are executed on a GPU with C compute units. In each compute unit, one or more thread-blocks can run. Each thread-block is composed of threads that are grouped into warps of W threads.

In both programs, 2D thread-blocks are used. Each thread-block is identified by its block indices (bx, by), and each thread is identified by its thread indices (tx, ty). The size of a thread-block is bdx * bdy. Consider that a thread-block is decomposed into warps in a way that threads with consecutive tx and equal ty belong to the same warp. More specifically, the warp number for a thread (tx, ty) is $\frac{\texttt{ty*bdx+tx}}{\texttt{warp-size}}$.

The entire input size is rows * cols integers. The size of an integer element is 4 bytes. The input is divided into tiles that are assigned to the thread-blocks.

local_data is an array in *local memory*, a fast on-chip memory that is used as a software-programmable cache. The amount of local memory per compute unit is S bytes. The threads of a thread-block can load data from *global memory* (i.e., the GPU off-chip memory) into local memory. The size of a global memory transaction is equal to the warp size times 4 bytes.

Program A:

```
__gpu_kernel_a(int* data, int rows, int cols){

  int* local_data[bdx * bdy];

  const int g_row = by * bdy + ty;
  const int g_col = bx * bdx + tx;
  const int l_row = ty;
  const int l_col = tx;
  const int pos   = g_row * cols + g_col;

  local_data[l_row * bdx + l_col] = data[pos];

  // Compute using local_data
}
```

Program B:

```
__gpu_kernel_b(int* data, int rows, int cols){

  int* local_data[bdx * bdy];

  const int g_row = bx * bdx + tx;
  const int g_col = by * bdy + ty;
  const int l_row = tx;
  const int l_col = ty;
  const int pos   = g_row * cols + g_col;

  local_data[l_row * bdy + l_col] = data[pos];

  // Compute using local_data
}
```

Please answer the questions on the next page.

(a) [15 points] What is the maximum number of thread-blocks that run in *each* compute unit for programs A and B?

$\lfloor \frac{S}{bdx \times bdy \times 4} \rfloor$.

Explanation. Given that each thread loads *one* integer value into local memory, the amount of local memory needed per thread-block is $bdx \times bdy \times 4$ (i.e., the number of integer elements of the array in local memory $bdx \times bdy$ times the size of an integer). Thus, the number of thread-blocks per compute unit is: $\lfloor \frac{S}{bdx \times bdy \times 4} \rfloor$.

(b) [15 points] Assuming that the GPU does *not* have caches, which program will execute faster? Why?

Program A.

Explanation. Program A will be faster, because it performs coalesced memory accesses (i.e., consecutive threads in the same warp access consecutive elements in memory), which ensure the minimum possible number of memory transactions.

(c) [15 points] Assume that the GPU has a single level of cache shared by all compute units. What will be the effect of this cache on the execution time of programs A and B?

Program B will be much faster than before (i.e., part (b)), while program A will not experience any improvement.

Explanation. There will be no significant change in the performance of program A, because the coalesced memory accesses already ensured the minimum possible number of memory transactions. However, program B will be much faster, because many accesses will hit the cache. For instance, if the threads with `ty` = 0 cause cache misses, the corresponding cache blocks will be loaded into the cache. Later accesses by the threads with `ty` = 1 will likely hit in the cache.

(d) [15 points] Assume that the access latency to the shared cache in part (c) is negligible. What should be the minimum size of the shared cache to guarantee that programs A and B have the same (or very similar) performance? (NOTE: The solution is independent of the warp scheduling policy).

$C \times \lfloor \frac{S}{bdx \times bdy \times 4} \rfloor \times (bdx \times bdy \times 4)$ bytes.

Explanation. Each thread-block loads one tile in local memory. Thus, the size of the shared cache per thread-block should be the size of the tile (`bdx * bdy`). Taking into account that there are $\lfloor \frac{S}{bdx \times bdy \times 4} \rfloor$ thread-blocks in each of the `C` compute units, the amount shared cache needed to keep all tiles in the cache is $C \times \lfloor \frac{S}{bdx \times bdy \times 4} \rfloor \times (bdx \times bdy \times 4)$ bytes.

(e) [15 points] Now assume that *only one* thread-block is executed in each compute unit. Each thread-block in program A needs always `T` ms to complete its work, because the computation is very regular. What will be the total execution time of program A?

$NumBatches \times T$ ms.

Explanation. The total number of thread-blocks is $NumBlocks = \lfloor \frac{rows \times cols}{bdx \times bdy} \rfloor$. The number of concurrent thread-blocks is $NumConcBlocks = C$. Thus, the total number of thread-blocks will be executed in a number of batches ($NumBatches$) that is equal to:

$NumBatches = \lceil \frac{NumBlocks}{NumConcBlocks} \rceil$

The total execution time is the $NumBatches \times T$ ms.