ETH 263-2210-00L Computer Architecture, Fall 2019

HW 5: GPU Programming, Emerging Memory Technologies, Prefetching, Asymmetric Multiprocessors, Consistency, and Coherence (SOLUTIONS)

Instructor: Prof. Onur Mutlu
TAs: Mohammed Alser, Rahul Bera, Geraldo Francisco De Oliveira Junior, Can Firtina, Juan Gomez Luna, Jawad Haj-Yahya, Hasan Hassan, Konstantinos Kanellopoulos, Jeremie Kim, Nika Mansouri Ghiasi, Lois Orosa Nogueira, Jisung Park, Minesh Hamenbhai Patel, Abdullah Giray Yaglikci

Given: Friday, Dec 6, 2019
Due: **Thursday, Dec 19, 2019**

- **Handin - Critical Paper Reviews (1).** You need to submit your reviews to `https://safari.ethz.ch/review/architecture19/`. Please, check your inbox, you should have received an email with the password you should use to login. If you didn't receive any email, contact comparch@lists.inf.ethz.ch. In the first page after login, you should click in "Architecture - Fall 2019 Home", and then go to "any submitted paper" to see the list of papers.
- **Handin - Questions (2-7).** You should upload your answers to the Moodle Platform (`https://moodle-app2.let.ethz.ch/mod/assign/view.php?id=391622`) as a single PDF file.

1. Critical Paper Reviews [300 points]

Please read the guidelines for reviewing papers and check the sample reviews. You may access them by *simply clicking on the QR codes below or scanning them.* We will give out extra credit that is worth 0.5% of your total grade for each good review.

Guidelines

Sample reviews

Write an approximately one-page critical review for each of the following papers. A review with bullet point style is more appreciated. Try not to use very long sentences and paragraphs. Keep your writing and sentences simple. Make your points bullet by bullet, as much as possible.

- Suleman et al., "Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures" in Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2009. `https://people.inf.ethz.ch/omutlu/pub/acs_asplos09.pdf`
- Srinath et al., "Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers" in Proceedings of the 13th International Symposium on High-Performance Computer Architecture (HPCA), 2006. `https://people.inf.ethz.ch/omutlu/pub/srinath_hpca07.pdf`
- Vijaykumar et al., "A Case for Core-Assisted Bottleneck Acceleration in GPUs: Enabling Flexible Data Compression with Assist Warps" in Proceedings of the 42nd International Symposium on Computer Architecture (ISCA), 2015. `https://people.inf.ethz.ch/omutlu/pub/caba-gpu-assist-warps_isca15.pdf`

2. GPU Programming and Performance Analysis [150 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 $\frac{ty*bdx+tx}{warp\text{-}size}$.

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) 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) 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) 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) 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) 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 $NumBlocks = \lceil \frac{rows \times cols}{bdx \times bdy} \rceil$. 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.

3. Emerging Memory Technologies [150 points]

Computer scientists at ETH developed a new memory technology, ETH-RAM, which is non-volatile. The access latency of ETH-RAM is close to that of DRAM while it provides higher density compared to the latest DRAM technologies. ETH-RAM has one shortcoming, however: it has limited endurance, i.e., a memory cell stops functioning after 10^6 writes are performed to the cell (known as cell wear-out).

A bright ETH student has built a computer system using 1 GB of ETH-RAM as main memory. ETH-RAM exploits a perfect wear-leveling mechanism, i.e., a mechanism that equally distributes the writes over all of the cells of the main memory.

(a) This student is worried about the lifetime of the computer system she has built. She executes a test program that runs special instructions to bypass the cache hierarchy and repeatedly writes data into different words until **all** the ETH-RAM cells are worn-out (stop functioning) and the system becomes useless. The student's measurements show that ETH-RAM stops functioning (i.e., all its cells are worn-out) in one year (365 days). Assume the following:

- The processor is in-order and there is no memory-level parallelism.
- It takes 5 ns to send a memory request from the processor to the memory controller and it takes 28 ns to send the request from the memory controller to ETH-RAM.
- ETH-RAM is word-addressable. Thus, each write request writes 4 bytes to memory.

What is the write latency of ETH-RAM? Show your work.

$t_{wear\_out} = \frac{2^{30}}{2^2} \times 10^6 \times (t_{write\_MLC} + 5 + 28)$
$365 \times 24 \times 3600 \times 10^9 \text{ns} = 2^{28} \times 10^6 \times (t_{write\_MLC} + 33)$
$t_{write\_MLC} = \frac{365 \times 24 \times 3600 \times 10^3}{2^{28}} - 33 = 84.5\text{ns}$

Explanation:

- Each memory cell should receive 10^6 writes.
- Since ETH-RAM is word addressable, the required amount of writes is equal to $\frac{2^{30}}{2^2} \times 10^6$ (there is no problem if 1 GB is assumed to be equal to 10^9 bytes).
- The processor is in-order and there is no memory-level parallelism, so the total latency of each memory access is equal to $t_{write\_MLC} + 5 + 28$.

(b) ETH-RAM works in the multi-level cell (MLC) mode in which each memory cell stores 2 bits. The student decides to improve the lifetime of ETH-RAM cells by using the single-level cell (SLC) mode. When ETH-RAM is used in SLC mode, the lifetime of each cell improves by a factor of 10 and the write latency decreases by 70%. What is the lifetime of the system using the SLC mode, if we repeat the experiment in part (a), with everything else remaining the same in the system? Show your work.

$t_{wear\_out} = \frac{2^{29}}{2^2} \times 10^7 \times (25.35 + 5 + 28) \times 10^{-9}$
$t_{wear\_out} = 78579686.3\text{s} = 2.49 \text{ year}$

Explanation:

- Each memory cell should receive $10 \times 10^6 = 10^7$ writes.
- The memory capacity is reduced by 50% since we are using SLC: $Capacity = 2^{30}/2 = 2^{29}$
- The required amount of writes is equal to $\frac{2^{29}}{2^2} \times 10^7$.
- The SLC write latency is $0.3 \times t_{write\_MLC}$: $t_{write\_SLC} = 0.3 \times 84.5 = 25.35\,\text{ns}$

4. Prefetching [150 points]

A processor is observed to have the following access pattern to cache blocks. Note that the addresses are **cache block addresses**, not byte addresses. This pattern is repeated for a large number of iterations.

```
Access Pattern P:   A,   A + 3,   A + 6,   A,   A + 5
```

Each cache block is 8KB. The hardware has a fully associative cache with LRU replacement policy and a total size of 24KB.

None of the prefetchers mentioned in this problem employ confidence bits, but they all start out with empty tables at the beginning of the access stream shown above. Unless otherwise stated, assume that 1) each access is separated long enough in time such that all prefetches issued can complete before the next access happens, and 2) the prefetchers have large enough resources to detect and store access patterns.

(a) You have a stream prefetcher (i.e., a next-N-block prefetcher), but you donâĂŹt know the prefetch degree (N) of it. However, you have a magical tool that displays the coverage and accuracy of the prefetcher. When you run a large number of repetitions of access pattern *P*, you get 40% coverage and 10% accuracy. What is the degree of this prefetcher (how many next blocks does it prefetch)?

Next 4 blocks.
40% coverage with a stream prefetcher for this pattern means blocks A+3 and A+6 are prefetched. Possible N at this point are 3 and 4. Accuracy 10% = 2/(N*5), so N is 4.

(b) You didnâĂŹt like the performance of the stream prefetcher, so you switched to a PC-based stride prefetcher that issues prefetch requests based on the stride detected for each memory instruction. Assume all memory accesses are incurred by the *same* load instruction (i.e., the same PC value) and the initial stride value for the prefetcher is set to 0.
Circle which of the cache block addresses are prefetched by this prefetcher:

```
A,   A + 3,   A + 6,   A,   A + 5
A,   A + 3,   A + 6,   A,   A + 5
A,   A + 3,   A + 6,   A,   A + 5
A,   A + 3,   A + 6,   A,   A + 5
```

Explain:

This prefetcher remembers the last stride and applies that to prefetch the next block from the current access.

(c) Stride prefetcher couldnâĂŹt satisfy you either. You changed to a Markov prefetcher with a correlation table of 12 entries (assume each entry can store a single address to prefetch, and remembers the most recent correlation). When all the entries are filled, the prefetcher replaces the entry that is least-recently accessed.
Circle which of the cache block addresses are prefetched by this prefetcher:

```
A,   A + 3,   A + 6,   A,   A + 5
A,   A + 3,   A + 6,   A,   A + 5
A,   A + 3,   A + 6,   A,   A + 5
A,   A + 3,   A + 6,   A,   A + 5
```

Explain:

All entries are filled after the first repetition, except the entry for A+5. Accesses to A+3 and A+5 thrash each other for block AâĂŹs next-block entry, so they cannot be correctly prefetched.

(d) Just in terms of coverage, after how many repetitions of access pattern P does the Markov prefetcher from part (c) start to outperform the stream prefetcher from part (a), if it can at all? Show your work.

5 repetitions.
Coverage for Markov prefetcher from part (c) is (0+2+3*(N-2))/(5N). This is equal to 40% when N=4, so it outperforms 40% at the 5th repetition. We gave full credit if you wrote either 4 or 5, depending on the work shown

(e) You think having a correlation table of 12 entries makes the hardware too costly, and want to reduce the number of correlation table entries for the Markov prefetcher. What is the minimum number of entries that gives the same prefetcher performance as 12 entries? Similar to the last part, assume each entry can store a single next address to prefetch, and remembers the most recent correlation. Show your work.

4 entries.
With 4 different accesses, we need at least 4 entries.

(f) Your friend is running the same program on a different machine that has a Markov prefetcher with 2 entries. The same assumptions from part (e) apply.
Circle which of the cache block addresses are prefetched by the prefetcher:

```
A,   A + 3,   A + 6,   A,   A + 5
A,   A + 3,   A + 6,   A,   A + 5
A,   A + 3,   A + 6,   A,   A + 5
A,   A + 3,   A + 6,   A,   A + 5
```

Explain:

NONE. Not enough entries. Entries will be evicted before they are used again.

(g) As an avid computer architect, you decide to update the processor by increasing the cache size to 32KB with the same cache block size. Assume you will be only running a program with the same access pattern P for a large number of iterations (i.e., one trillion), describe a prefetcher that provides

smaller memory bandwidth consumption than the baseline without a prefetcher.
Explain:

No prefetcher. Since all lines will sit in the cache after the first iteration, the prefetcher needs to achieve 100% accuracy on the first iteration in order to consume the same amount of memory bandwidth as the baseline, which is not possible without any training on the prefetcher beforehand.
Note: We leave it up to you to think whether or not you can ever design a prefetcher that leads to less memory consumption than the baseline without a prefetcher.

5. Asymmetric Multicore [150 points]

A microprocessor manufacturer asks you to design an asymmetric multicore processor for modern workloads. You should optimize it assuming a workload with 80% of its work in the parallel portion. Your design contains one large core and several small cores, which share the same die. Assume the total die area is 32 units.

- *Large core*: For a large core that is n times faster than a single small core, you will need n^3 units of die area (n is a positive integer). The dynamic power of this core is $6 \times n$ Watts and the static power is n Watts.
- *Small cores*: You will fit as many small cores as possible, after placing the large core. A small core occupies 1 unit of die area. Its dynamic power is 1 Watt and its static power is 0.5 Watts.

The parallel portion executes *only* on the small cores, while the serial portion executes *only* on the large core.

Please answer the following questions. Show your work. Express your equations and solve them. You can approximate some computations, and get partial or full credit.

(a) What configuration (i.e., number of small cores and size of the large core) results in the best performance?

One large core and 24 small cores. The large core will occupy 8 units of die area.

Explanation:
Given that the large core occupies n^3 units, the number of small cores will be $32 - n^3$. Thus, the speedup can be calculated as:
$Speedup = \frac{1}{\frac{0.2}{n} + \frac{0.8}{32-n^3}}$.

Without loss of generality, we assume that the total execution time is:
$t_{total} = t_{serial} + t_{parallel} = \frac{0.2}{n} + \frac{0.8}{32-n^3}$ seconds.

n	$\#small$	t_{serial}	$t_{parallel}$	t_{total}
1	31	0.20	0.03	0.23
2	24	0.10	0.03	**0.13**
3	5	0.07	0.16	0.23

These calculations can be approximated without a calculator:

n	$\#small$	t_{serial}	$t_{parallel}$	t_{total}
1	31	0.20 / 1 = 0.20	0.02 < 0.80 / 31 < 0.03	> 0.22
2	24	0.20 / 2 = 0.10	0.03 < 0.80 / 24 < 0.04	< **0.14**
3	5	0.20 / 3 = 0.07	0.80 / 5 = 0.16	> 0.22

(b) The energy consumption should also be a metric of reference in your design. Compute the energy consumption for the best configuration in part (a).

$E_{total} = 26 \times t_{serial} + 38 \times t_{parallel} = 3.74$ Joules.

Explanation:
We can calculate the energy consumption as:
$E_{total} = E_{large} + E_{small} =$
$(P_{large\_dynamic} + P_{large\_static}) \times t_{serial} + P_{large\_static} \times t_{parallel}$
$+ (P_{small\_static} \times t_{serial} + (P_{small\_dynamic} + P_{small\_static}) \times t_{parallel}) \times (32 - n^3) =$
$7 \times n \times t_{serial} + n \times t_{parallel} + (0.5 \times t_{serial} + 1.5 \times t_{parallel}) \times (32 - n^3) =$
$14 \times t_{serial} + 2 \times t_{parallel} + 12 \times t_{serial} + 36 \times t_{parallel} =$
$26 \times t_{serial} + 38 \times t_{parallel} = 3.74$ Joules.

This result can be approximated without a calculator:
$E_{total} < 26 \times 0.10 + 38 \times 0.04 = 2.6 + 1.52 = 4.12$ Joules.

(c) For the best configuration obtained in part (a), you are considering to use the large core to collaborate with the small cores on the execution of the parallel portion.

(i) What is the overall performance improvement, compared to the performance obtained in part (a), if the large core collaborates on the parallel portion?

If the large core collaborates with the small cores in the parallel portion, the best-case speedup can be calculated as:
$Speedup = \frac{1}{\frac{0.2}{n} + \frac{0.8}{32-n^3+n}}$.

Without loss of generality, we assume that the total execution time is:
$t_{total} = t_{serial} + t_{parallel} = \frac{0.2}{n} + \frac{0.8}{32-n^3+n}$ seconds.

The execution time of the serial part t_{serial}, which takes significantly longer than the parallel part (about 3 times longer), does not change. By using the large core to collaborate in the parallel portion, the execution time of the parallel part $t_{parallel}$ decreases from $\frac{0.8}{24}$ to $\frac{0.8}{24+2}$, i.e., a speedup of $\frac{13}{12}$, which is less than 10%. Thus, the overall performance improvement from using the large core to collaborate in the parallel portion is negligible.

(ii) What is the overall energy change, compared to the energy obtained in part (b), if the large core collaborates on the parallel portion?

If the large core collaborates in the parallel portion, we calculate the energy consumption as:
$E_{total} = E_{large} + E_{small} =$
$(P_{large\_dynamic} + P_{large\_static}) \times t_{serial} + (P_{large\_dynamic} + P_{large\_static}) \times t_{parallel}$
$+ (P_{small\_static} \times t_{serial} + (P_{small\_dynamic} + P_{small\_static}) \times t_{parallel}) \times (32 - n^3) =$
$7 \times n \times t_{serial} + 7 \times n \times t_{parallel} + (0.5 \times t_{serial} + 1.5 \times t_{parallel}) \times (32 - n^3) =$
$14 \times t_{serial} + 14 \times t_{parallel} + 12 \times t_{serial} + 36 \times t_{parallel} =$
$26 \times t_{serial} + 50 \times t_{parallel} \simeq 2.6 + 2.0 = 4.6$ Joules.

We assume that $t_{parallel}$ has a very small change, as discussed above. If we compare this equation to the energy equation in part (b), we observe that the energy consumption increases by $P_{large\_dynamic} \times t_{parallel} = 6 \times n \times t_{parallel} = 12 \times t_{parallel}$ Joules. Since the energy consumption of the parallel portion is $38 \times t_{parallel}$ Joules in part (b), there is an energy increase in the parallel portion of more than 30% (i.e., $\frac{12}{38}$). The overall energy increase is more than 11%.

(iii) Discuss whether it is worth using the large core to collaborate with the small cores on the execution of the parallel portion.

It is not really worth using the large core in the parallel part. While the performance improvement is negligible, the overall energy consumption increases by more than 11%.

(d) Now assume that the serial portion can be optimized, i.e., the serial portion becomes smaller. This gives you the possibility of reducing the size of the large core, and still improving performance. For a large core with an area of $(n-1)^3$, where n is the value obtained in part (a), what should be the fraction of serial portion that would lead to better performance than in part (a)?

10%.

Explanation:
We call t_{total} the total execution time with a large core with $n = 2$, as obtained in part (a), and t'_{total} for a smaller core with $n = 1$. We can obtain the new parallel fraction p from the following equation:

$t_{total} > t'_{total}$;

$0.13 > \frac{1-p}{n-1} + \frac{p}{32-(n-1)^3}$;

$0.13 > \frac{1-p}{1} + \frac{p}{31}$;

$p > 0.90$.

The serial portion should be *at most* 10%.

(e) Your design is so successful for desktop processors that the company wants to produce a similar design for mobile devices. The power budget becomes a constraint. For a maximum of total power of 20W, how much would you need to reduce the dynamic power consumption of the large core, if at all, for the best configuration obtained in part (a)? Assume again that the parallel fraction is 80% of the workload. (Hint: Express the dynamic power of the large core as $D \times n$ Watts, where D is a constant).

We have to reduce the dynamic power consumption of the large core by *at least* 20×.

Explanation:
We calculate the total power as the total energy divided by the total execution time:
$P_{total} = \frac{E_{total}}{t_{total}}$ Watts;

$P_{total} = \frac{E_{large}+E_{small}}{t_{total}} \leq 20$ Watts;

We express the dynamic power of the large core as $D \times n$. From part (a) we know n, t_{serial}, $t_{parallel}$ and t_{total}, from part (b) we know E_{small}:

$\frac{(D+1)\times n\times t_{serial}+n\times t_{parallel}+E_{small}}{t_{total}} = \frac{(D+1)\times 2\times 0.10+n\times 0.03+2.00}{0.13} \leq 20$ Watts;

$D \leq 0.3$.

In mobile devices, the dynamic power of the large core has to be $\leq 0.3 \times n$ Watts (given the assumptions in the question). Since the dynamic power of the large core is $6 \times n$ Watts in the desktop processor, we have to reduce the dynamic power consumption of the large core by *at least* 20× for mobile devices.

6. Memory Consistency [150 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) 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) 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 $\rightarrow$ T0.1 - Final value = 1.
Ordering 2: T0.1 $\rightarrow$ 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) 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 $\rightarrow$ T1.1 $\rightarrow$ T0.3 - Final value = 2.
Ordering 2 (from the perspective of T0): T1.1 $\rightarrow$ T1.0 $\rightarrow$ T0.3 - Final value = 2.
Ordering 3 (from the perspective of T0): T1.1 $\rightarrow$ T0.3 $\rightarrow$ 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) 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		**Thread T1**
Instr. T0.0	`a = X[0];`	Instr. T1.0	`Y[0] = 1;`
Instr. T0.1	`b = a + Y[0];`		**`memory_fence();`**
Instr. T0.2	`while(*flag == 0);`	Instr. T1.1	`*flag = 1;`
	`memory_fence();`	Instr. T1.2	`X[1] *= 2;`
Instr. T0.3	`Y[0] += 1;`	Instr. T1.3	`a = 0;`

7. Cache Coherence [150 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) 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) 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) 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) 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) 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.