Computer Architecture (263-2210-00L), Fall 2018
HW 2: Caches, Memory Parallelism, DRAM Fundamentals
SOLUTIONS

Instructor: Prof. Onur Mutlu
TAs: Mohammed Alser, Can Firtina, Hasan Hassan, Jeremie Kim, Juan Gómez Luna, Geraldo Francisco de Oliveira, Minesh Patel, Giray Yaglikci

Assigned: Thursday, Oct 11, 2018
Due: **Thursday, Oct 25, 2018**

- **Handin - Critical Paper Reviews (1).** You need to submit your reviews to `https://safari.ethz.ch/review/architecture18/`. Please check your inbox. You should have received an email with the password you can use to login to the paper review system. If you have not received any email, please contact comparch@lists.ethz.ch. In the first page after login, you should click in "Architecture - Fall 2018 Home", and then go to "any submitted paper" to see the list of papers.
- **Handin - Questions (2-8).** Please upload your solution to the Moodle (`https://moodle-app2.let.ethz.ch/`) as a single PDF file. **Please use a typesetting software (e.g., LaTeX) or a word processor (e.g., MS Word, LibreOfficeWriter) to generate your PDF file. Feel free to draw your diagrams either using an appropriate software or by hand, and include the diagrams into your solutions PDF.**

1 Critical Paper Reviews [150 points]

Please read the following handout on how to write critical reviews. We will give out extra credit that is worth 0.5% of your total grade for each good review.

- Lecture slides on guidelines for reviewing papers. Please follow this format. `https://safari.ethz.ch/architecture/fall2018/lib/exe/fetch.php?media=onur-comparch-f18-how-to-do-the-paper-reviews.pdf`
- Some sample reviews can be found here: `https://safari.ethz.ch/architecture/fall2018/doku.php?id=readings`

(a) Write a one-page critical review for **two** of the following papers:

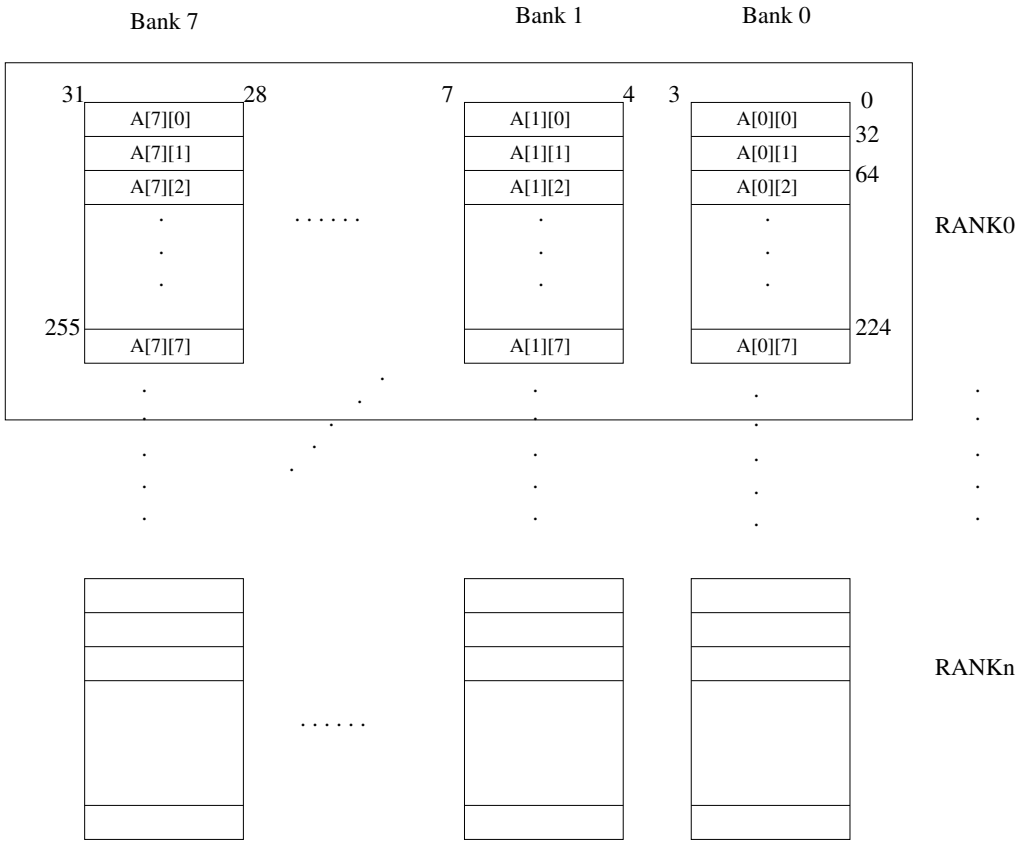
- M.K. Qureshi, D.N. Lynch, O. Mutlu, Y.N. Patt. "A Case for MLP-Aware Cache Replacement". ISCA 2006 `https://people.inf.ethz.ch/omutlu/pub/qureshi_isca06.pdf`
- D. Lee, Y. Kim, V. Seshadri, J. Liu, L. Subramanian, and O. Mutlu, "Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture," HPCA 2013 `https://people.inf.ethz.ch/omutlu/pub/tldram_hpca13.pdf`
- V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M.A. Kozuch, O. Mutlu, P.B. Gibbons, T.C. Mowry, "Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology," MICRO 2017 `https://people.inf.ethz.ch/omutlu/pub/ambit-bulk-bitwise-dram_micro17.pdf`

2 Main Memory Organization and Interleaving [150 points]

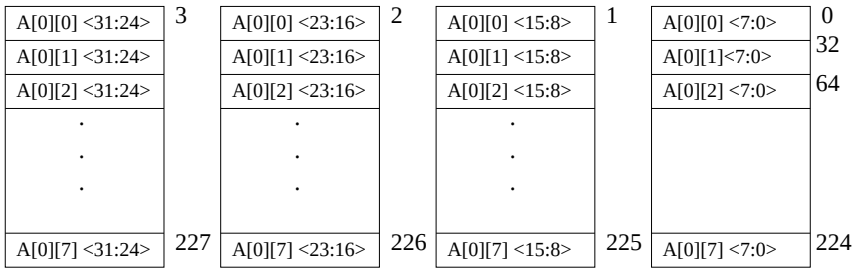
Consider the following piece of code:

```
for(i = 0; i < 8; ++i){
  for(j = 0; j < 8; ++j){
    sum = sum + A[i][j];
  }
}
```

The figure below shows an 8-way interleaved, byte-addressable memory. The total size of the memory is 4KB. The elements of the 2-dimensional array, A, are 4-bytes in length and are stored in the memory in column-major order (i.e., columns of A are stored in consecutive memory locations) as shown. The width of the bus is 32 bits, and each memory access takes 10 cycles.



A more detailed picture of the memory chips in Bank 0 of Rank 0 is shown below.



(a) Since the address space of the memory is 4KB, 12 bits are needed to uniquely identify each memory location, i.e., Addr[11:0]. Specify which bits of the address will be used for:

- Byte on bus
  Addr [1 : 0]
- Interleave/Bank bits
  Addr [4 : 2]
- Chip address
  Addr [7 : 5]
- Rank bits
  Addr [11 : 8]

(b) How many cycles are spent accessing memory during the execution of the above code? Compare this with the number of memory access cycles it would take if the memory were not interleaved (i.e., a single 4-byte wide array).

Assumptions: Requests are scheduled in order one after another and any number of requests can be outstanding at a time (limited only by bank conflicts).

As consecutive array elements are stored in the same bank, they have to be accessed serially. However, the memory access in the last iteration of the inner for loop (A[x][7]) and the memory access in the first iteration of the next time through the inner for loop (A[x+1][0]) go to different banks and can be accessed in parallel.
Therefore, number of cycles = $(10 \times 8 \times 8)$ - (9×7) = 577 cycles.

(c) Can any change be made to the current interleaving scheme to optimize the number of cycles spent accessing memory? If yes, which bits of the address will be used to specify the byte on bus, interleaving, etc. (use the same format as in part a)? With the new interleaving scheme, how many cycles are spent accessing memory? Remember that the elements of A will still be stored in column-major order.

Elements along a column are accessed in consecutive cycles. So, if they were mapped to different banks, accesses to them can be interleaved.
This can be done by using the following address mapping:

- Byte on bus
  Addr [1 : 0]
- Interleave/Bank bits
  Addr [7 : 5]
- Chip address
  Addr [4 : 2]
- Rank bits
  Addr [11 : 8]

The first iteration of the outer loop starts at time 0. The first access is issued at cycle 0 and ends at cycle 10. Accesses to consecutive banks are issued every cycle after that until cycle 7 and these accesses complete by cycle 17. The first access of the second iteration of the outer loop can start only in cycle 10, as bank 0 is occupied until cycle 10 with the first access of the previous outer loop iteration. The second iteration's accesses complete at cycle 27. Extending this, the eighth iteration's first access starts at cycle 70 and completes by cycle 87.
Therefore, number of cycles = 87 cycles.

(d) Using the original interleaving scheme, what small changes can be made to the piece of code to optimize the number of cycles spent accessing memory? How many cycles are spent accessing memory using the modified code?

The inner and outer loop can be reordered as follows:

```
for(j = 0; j < 8; ++j){
  for(i = 0; i < 8; ++i){
    sum = sum + A[i][j];
  }
}
```

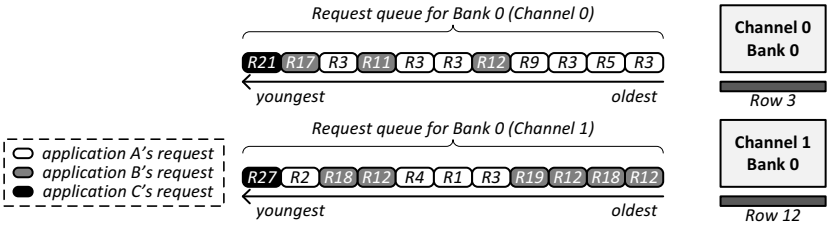
Thus, elements along a row, which are stored in consecutive banks (using the original interleaving scheme), will be accessed in consecutive cycles. The effect this has on the number of cycles is exactly the same as part (c).
Therefore, number of cycles = 87 cycles.

3 Memory Scheduling [150 points]

To serve a memory request, the memory controller issues one or multiple DRAM commands to access data from a bank. There are four different DRAM commands.

- `ACTIVATE`: Loads the row (that needs to be accessed) into the bank's row-buffer. This is called *opening* a row. **(Latency: 15ns)**
- `PRECHARGE`: Restores the contents of the bank's row-buffer back into the row. This is called *closing* a row. **(Latency: 15ns)**
- `READ/WRITE`: Accesses data from the row-buffer. **(Latency: 15ns)**

The following figure shows the snapshot of the memory request buffers (in the memory controller) at t_0. Each request is color-coded to denote the application to which it belongs (assume that all applications are running on separate cores). Additionally, each request is annotated with the address (or index) of the row that the request needs to access (e.g., *R3* means that the request is to the 3^{rd} row). Furthermore, assume that all requests are read requests.



A memory request is considered to be *served* when the `READ` command is complete (i.e., 15ns after the request's `READ` command has been issued). In addition, each application (A, B, or C) is considered to be **stalled** until *all* of its memory requests (across all the request buffers) have been served.

Assume that, initially (at t_0) each bank has the 3^{rd} and the 12^{th} row loaded in the row-buffer, respectively. Furthermore, no additional requests from any of the applications arrive at the memory controller.

3.1 Application-Unaware Scheduling Policies

(a) Using the **FCFS** scheduling policy, what is the **stall time** of each application?

App A: MAX(2H+7M, H+9M) = H+9M = 15+405 = 420ns

App B: MAX(2H+8M, H+8M) = 2H+8M = 30+360 = 390ns

App C: MAX(2H+9M, H+10M) = H+10M = 15+450 = 465ns

(b) Using the **FR-FCFS** scheduling policy, what is the stall time of each application?

App A: MAX(5H+2M, (4H+2M)+4M) = 4H+6M = 60+270 = 330ns

App B: MAX((5H+2M)+3M, 4H+2M) = 5H+5M = 75+225 = 300ns

App C: MAX(((5H+2M)+3M)+M, ((4H+2M)+4M)+M) = 4H+7M = 60 + 315 = 375ns

(c) What property of memory references does the *FR-FCFS* scheduling policy exploit? (Three words or less.)

Row-buffer locality

(d) Briefly describe the scheduling policy that would **maximize** the *request throughput* at any given bank, where request throughput is defined as the number of requests served per unit amount of time. (Less than 10 words.)

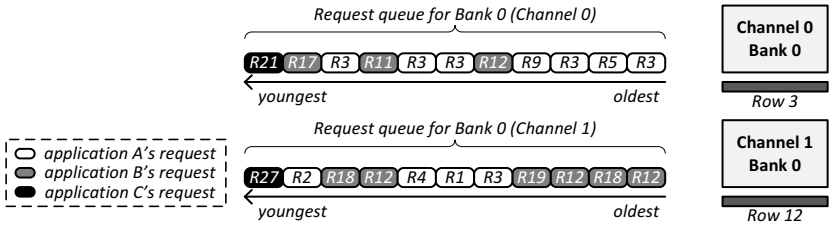
FR-FCFS

3.2 Application-Aware Scheduling Policies

Of the three applications, application C is the least memory-intensive (i.e., has the lowest number of outstanding requests). However, it experiences the largest stall time since its requests are served only after the numerous requests from other applications are first served. To ensure the shortest stall time for application C, one can assign its requests with the highest priority, while assigning the same low priority to the other two applications (A and B).

(a) **Scheduling Policy X:** When application C is assigned a high priority and the other two applications are assigned the same low priority, what is the stall time of each application? (Among requests with the same priority, assume that *FR-FCFS* is used to break ties.)

App A: MAX(M+(4H+3M), M+(3H+3M)+4M) = 3H+8M = 45+360 = 405ns App B: MAX(M+(4H+3M)+3M, M+(3H+3M)) = 4H+7M = 60+315 = 375ns App C: MAX(M, M) = M = 45ns



Can you design an even better scheduling policy? While application C now experiences low stall time, you notice that the other two applications (A and B) are still delaying each other.

(b) Assign priorities to the other two applications such that you minimize the average stall time across all applications. Specifically, list all **three** applications in the order of highest to lowest priority. (Among requests with the same priority, assume that *FR-FCFS* is used to break ties.)

C > B > A

(c) **Scheduling Policy Y:** Using your new scheduling policy, what is the stall time of each application? (Among requests with the same priority, assume that *FR-FCFS* is used to break ties.)

App A: MAX(M+(3M)+(4H+3M), M+(3H+3M)+4M) = 3H+8M = 45+360 = 405ns

App B: MAX(M+(3M), M+(3H+3M)) = 3H+4M = 45+180 = 225ns

App C: MAX(M, M) = M = 45ns

(d) Order the four scheduling policies (FCFS, FR-FCFS, X, Y) in the order of lowest to highest average stall time.

Y<X<FR-FCFS<FCFS

4 Main Memory Potpourri [100 points]

A machine has a 4 KB DRAM main memory system. Each row is refreshed every 64 ms.

Note: This question is open ended. We provide one possible set of solutions. There could be other possible right solutions.

(a) The machine's designer runs two applications A and B (each run alone) on the machine. Although applications A and B have a similar number of memory requests, application A spends a surprisingly larger fraction of cycles stalling for memory than application B does. What might be the reasons for this?

A large number of application A's memory requests are row-buffer conflicts, whereas a large number of application B's memory requests are row-buffer hits.
Hence, application A's requests take longer to service and it spends more time stalling for memory.

(b) Application A also consumes a much larger amount of memory energy than application B does. What might be the reasons for this?

Row-buffer conflicts also consume more energy, as they require a precharge, activate and a read/write, whereas row-buffer hits only require a read/write.
Hence, application A consumes more memory energy.

(c) When applications A and B are run together on the machine, application A's performance degrades significantly, while application B's performance doesn't degrade as much. Why might this happen?

When the applications are run together, they interfere with each other. Hence, both applications' performance degrades when they run together. However, if a memory scheduler that favor row-buffer hits over row-buffer conflicts (like FR-FCFS) is used, it would favor application B's requests over application A's requests.
Therefore, application A's performance degrades more.

(d) The designer decides to use a smarter policy to refresh the memory. A row is refreshed only if it has not been accessed in the past 64 ms. Do you think this is a good idea? Why or why not?

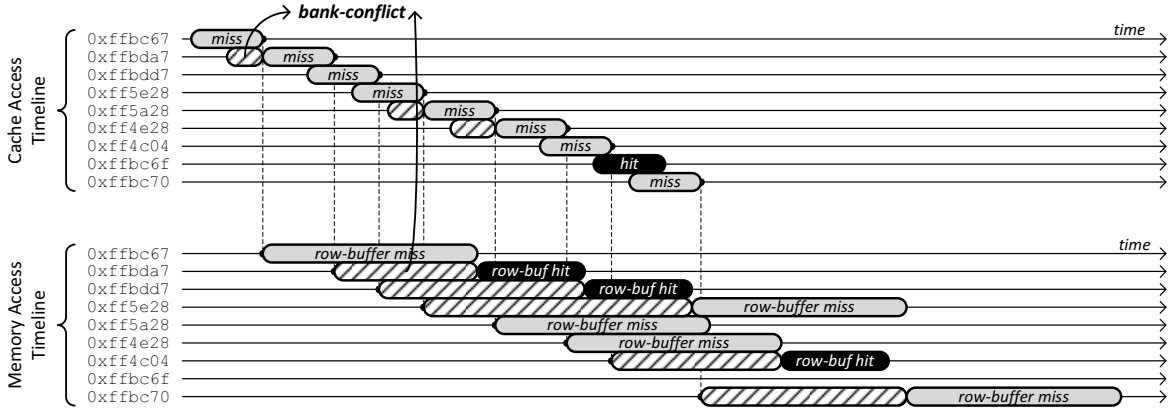
This can reduce refresh energy significantly if a large fraction of the rows in memory contain data and are accessed (within the 64 ms refresh window), as these rows do not have to be refreshed explicitly. However, if only a small number of rows contain data and only these rows are accessed, this policy will not provide much reduction in refresh energy as a large fraction of rows are still refreshed at the 64 ms rate.

(e) The refresh energy consumption when application B is run, drops significantly when this new refresh policy is applied, while the refresh energy when application A is run reduces only slightly. Is this possible? Why or why not?

This is possible. If application B has a large working set, it could access a large fraction of the memory rows (within the 64 ms refresh window) and hence these rows do not have to be refreshed explicitly. Application A on the other could have a much smaller working set and hence a large fraction of rows still have to be refreshed at the 64 ms rate.

5 Banks [150 points]

A processor's memory hierarchy consists of a small SRAM L1-cache and a large DRAM main memory, both of which are banked. The processor has a 24-bit physical address space and does not support virtual memory (i.e., all addresses are physical addresses). An application has just started running on this processor. The following figure shows the timeline of memory references made by that application and how they are served in the L1-cache or main memory.



For example, the first memory reference made by the application is to byte-address `0xffbc67` (assume that all references are byte-sized reads to byte-addresses). However, the memory reference misses in the L1-cache (assume that the L1-cache is intially empty). Immediately afterwards, the application accesses main memory, where it experiences a row-buffer miss (initially, assume that all banks in main memory each have a row opened that will never be accessed by the application). Eventually, the cache block (and only that cache block) that contains the byte-address `0xffbc67` is fetched from memory into the cache.

Subsequent memory references may experience *bank-conflicts* in the L1-cache and/or main memory, if there is a previous reference still being served at that particular bank. Bank-conflicts are denoted as hatched shapes in the timeline.

The following table shows the address of the memory references made by the application in both hexadecimal and binary representations.

(a) Memory address table

Hexadecimal	Binary
`ffbc67`	`1111 1111 1011 1100 0110 0111`
`ffbda7`	`1111 1111 1011 1101 1010 0111`
`ffbdd7`	`1111 1111 1011 1101 1101 0111`
`ff5e28`	`1111 1111 0101 1110 0010 1000`
`ff5a28`	`1111 1111 0101 1010 0010 1000`
`ff4e28`	`1111 1111 0100 1110 0010 1000`
`ff4c04`	`1111 1111 0100 1100 0000 0100`
`ffbc6f`	`1111 1111 1011 1100 0110 1111`
`ffbc70`	`1111 1111 1011 1100 0111 0000`

From the above timelines and the table, your job is to answer questions about the processor's cache and main memory organization. Here are some assumptions to help you along the way.

- Assumptions about the L1-cache
  - Block size: ? (Power of two, greater than two)
  - Associativity: ? (Power of two, greater than two)
  - Total data-store size: ? (Power of two, greater than two)
  - Number of banks: ? (Power of two, greater than two)
  - Initially empty
- Assumptions about main memory
  - Number of channels: 1
  - Number of ranks per channel: 1
  - Number of banks per rank: ? (Power of two, greater than two)
  - Number of rows per bank: ? (Power of two, greater than two)
  - Number of cache-blocks per row: ? (Power of two, greater than two)
  - Contains the entire working set of the application
  - Initially, all banks have their 0^{th} row open, which is never accessed by the application

5.1 First, let's cover the basics

(a) Caches and main memory are sub-divided into multiple banks in order to allow parallel access. What is an alternative way of allowing parallel access?

Multiporting, duplicating

(b) A cache that allows multiple cache misses to be outstanding to main memory at the same time is called what? (Two words or less. Hint: It's an adjective.)

Non-blocking (or lockup-free)

(c) While cache misses are outstanding to main memory, what is the structure that keeps bookkeeping information about the outstanding cache misses? This structure often augments the cache.

Miss status handling registers (MSHRs)

(d) Which is larger, an SRAM cell or a DRAM cell?

SRAM cell

(e) What is the number of transistors and/or capacitors needed to implement each cell, including access transistor(s)?

SRAM: 6T DRAM: 1T-1C

5.2 Cache and memory organization

NOTE: For the following questions, assume that all offsets and indexes come from contiguous address bits.

(a) What is the L1-cache's block size in bytes? Which bit positions in the 24-bit physical address correspond to the cache block offset? (The least-significant bit in the physical address has a bit position of 0.)

Block size: 16 bytes Bit positions of block offset: 0-3

(b) How many banks are there in the L1-cache? Which bit positions in the 24-bit physical address correspond to the L1-cache bank index? (The least-significant bit in the physical address has a bit position of 0.)

Number of L1-cache banks: 4 Bit positions of L1-cache bank index: 4-5

(c) How many banks are there in main memory? Which bit positions in the 24-bit physical address correspond to the main memory bank index? (The least-significant bit in the physical address has a bit position of 0.)

Number of main memory banks: 8 Bit positions of main memory bank index: 10-12

(d) What kind of interleaving is used to map physical addresses to main memory?

Row-interleaving

(e) To fully support a 24-bit physical address space, how many rows must each main memory bank have? Which bit positions in the 24-bit physical address correspond to the main memory row index? (The least-significant bit in the physical address has a bit position of 0.)

Number of rows per main memory bank: 2048 Bit positions of row index: 13-23

(f) Each cache block within a row is called a *column*. How many columns are there in a single row? Which bit positions in the 24-bit physical address correspond to the main memory column index? (The least-significant bit in the physical address has a bit position of 0.)

Number of columns per row: 64 Bit positions of column index: 4-9

6 Memory Hierarchy [100 points]

Assume you developed the next greatest memory technology, MagicRAM. A MagicRAM cell is non-volatile. The access latency of a MagicRAM cell is 2 times that of an SRAM cell but the same as that of a DRAM cell. The read/write energy of MagicRAM is similar to the read/write energy of DRAM. The cost of MagicRAM is similar to that of DRAM. MagicRAM has higher density than DRAM. MagicRAM has one shortcoming, however: a MagicRAM cell stops functioning after 2000 writes are performed to the cell.

(a) Is there an advantage of MagicRAM over DRAM other than its density? (Please do not repeat what is stated in the above paragraph.) Explain.

Yes. MagicRAM does not need refreshes, since it is nonvolatile. This can reduce dynamic power, bus utilization, and bank contention. MagicRAM is also nonvolatile, which can enable new uses or programming models.

(b) Is there an advantage of MagicRAM over SRAM? Explain.

Yes. MagicRAM has higher density and lower cost than SRAM.

(c) Assume you have a system that has a 64KB L1 cache made of SRAM, a 12MB L2 cache made of SRAM, and 4GB main memory made of DRAM.

Assume you have complete design freedom and add structures to overcome the shortcoming of MagicRAM. You will be able to propose a way to reduce/overcome the shortcoming of MagicRAM (note that you can design the hierarchy in any way you like, but cannot change MagicRAM itself).

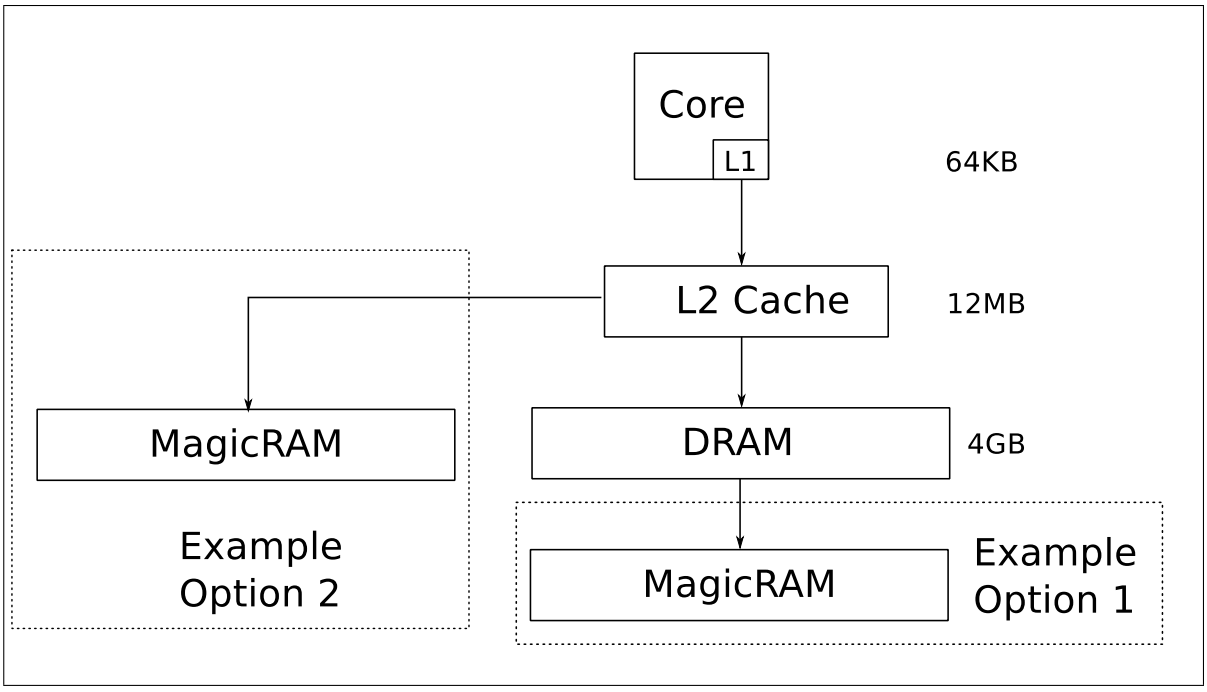
(i) Does it makes sense to add MagicRAM anywhere in this memory hierarchy given that you can potentially reduce its shortcoming?

Yes.

(ii) If so, where would you place MagicRAM? Describe it in terms of the figure above and describe why you made this choice.

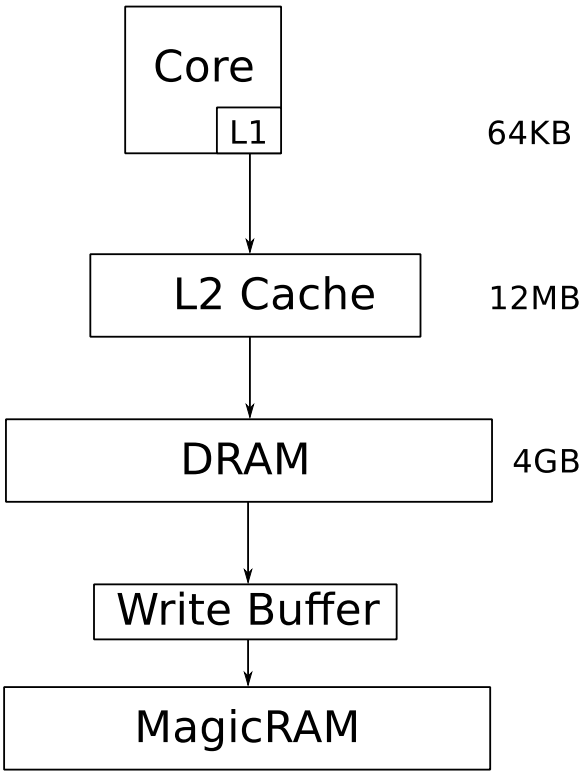
If not, why not? Explain below clearly and methodically

Many answers are possible. One option: Place MagicRAM below DRAM in the hierarchy, and use DRAM as a cache to MagicRAM. This way, DRAM performs write coalescing so that MagicRAM does not wear out as quickly. Another option could be to place MagicRAM "next to" DRAM (on the same or another channel), and use MagicRAM explicitly for read-only data.



(d) Propose a way to reduce/overcome the shortcoming of MagicRAM by modifying the given memory hierarchy. Be clear in your explanations and illustrate with drawings to aid understanding.

A write-combining/coalescing buffer could be added. The memory hierarchy could also perform wear-leveling, or it could predict which data is less likely to be modified and place only that data in MagicRAM.
Figure(s):



7 Instruction and Data Caches [100 points]

Consider the following loop is executed on a system with a small instruction cache (I-cache) of size 16 B. The data cache (D-cache) is fully associative of size 1 KB. Both caches use 16-byte blocks. The instruction length is 4 B. The initial value of register `$1` is 40. The value of `$0` is 0.

```
Loop: lw   $6, X($1)
      addi $6, $6, 1
      sw   $6, Y($1)
      subi $1, $1, 4
      beq $1, $0, Exit
      j    Loop
Exit:  ...
```

(a) Compute I-cache and D-cache miss rates, considering:

The I-cache can keep 4 instructions. Thus, in each iteration there will be 2 cache misses. The I-cache miss rate will be 2/6 = 0.33.

- `X` and `Y` are different arrays.
  If `X` and `Y` are different arrays, there will be 2 cache misses every 4 iterations, that is, one read miss and one write miss every 8 accesses. In that case, the D-cache miss rate will be 2/8 = 0.25.
- `X` and `Y` are the same array.
  If `X` and `Y` are the same array, the D-cache miss rate will be one half of the previous one (0.125).

(b) Compute the average number of cycles per instruction (CPI), using a baseline ideal CPI (ideal caches) equal to 2, and a miss latency equal to 10 clock cycles.

Since load and store instructions are one third of all the execute instructions, CPI is calculated as:
CPI = 2 + 0.33×10 + 0.33×0.25×10 = 6.1250 cycles, if `X` and `Y` are different arrays.
CPI = 2 + 0.33×10 + 0.33×0.125×10 = 5.7125 cycles, if `X` and `Y` are the same array.

(c) A compiler could unroll this loop for optimization. How would this affect CPI?

If the compiler unrolls the loop, the total number of executed instructions is reduced. As there will be one I-cache miss every four executed instructions, the I-cache miss rate will be 1/4 = 0.25. The D-cache miss rate remains the same, but the fraction of loads and stores changes. `beq` and `j` instructions are no longer necessary, so the fraction of load and stores is 0.50.
The new CPI is:
CPI = 2 + 0.25×10 + 0.50×0.25×10 = 5.75 cycles, if `X` and `Y` are different arrays.
CPI = 2 + 0.25×10 + 0.50×0.125×10 = 5.125 cycles, if `X` and `Y` are the same array.

(d) How would the previous results change with a 32-byte I-cache?

If the size of the I-cache is 32 B, the entire loop fits in it. There will be only two cold misses in the first iteration. Thus, the I-cache miss rate will be 2/60 = 0.033.

8 Prefetching [100 points]

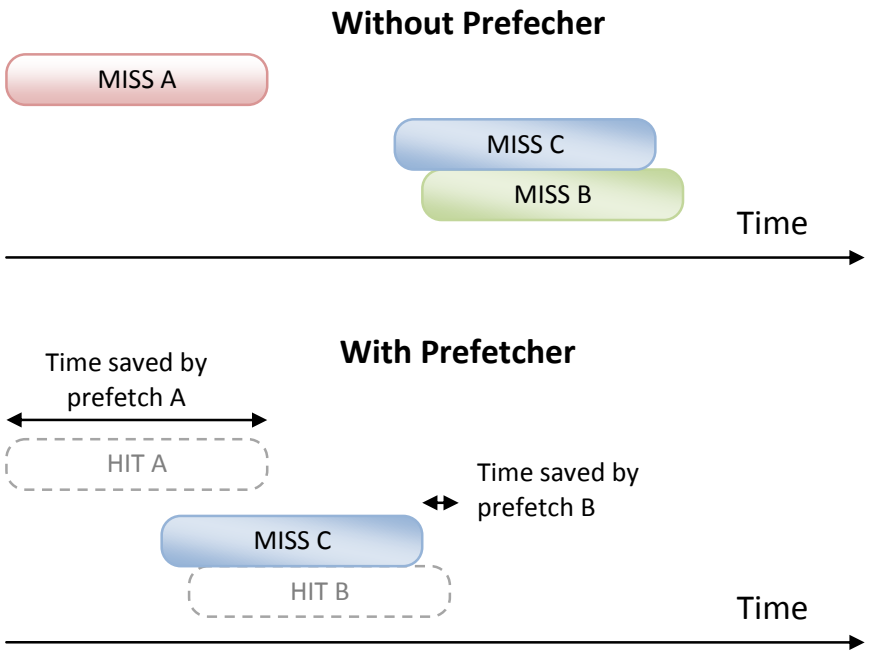
Suppose you have designed the next fancy hardware prefetcher for your system. You analyze its behavior and find the following:

(a) The prefetcher successfully prefetches block A into the cache before it is required by a load instruction. The prefetched block evicts a never-to-be-used block from the cache, so it does not cause cache pollution. Furthermore, you find that the prefetch request does not waste bus bandwidth needed by some other request.

(b) The prefetcher successfully prefetches block B into the cache before it is required by a load instruction. The prefetched block evicts a never-to-be-used block from the cache, so it does not cause cache pollution. Furthermore, you find that the prefetch request does not waste bus bandwidth needed by some other request.

Upon further analysis, you find that the prefetching of block A actually reduced execution time of the program whereas prefetching of block B did not reduce execution time significantly. Describe why this could happen. Draw two execution timelines, one with and one without the prefetcher, to illustrate the concept.

Given that both prefetch requests were used, no prefetch wasted bandwidth or caused cache pollution, but only A reduced execution time significantly, we must conclude that the miss of A was more expensive than the miss of B. Possible reason: Memory Level Parallelism. The miss of B was parallel with another miss (e.g. C) that wasn't prefetched (i.e., the processor still needs to stall for the parallel miss) whereas the miss of A was an isolated miss, eliminating which reduced the stall time.



9 GPUs and SIMD (I) [100 points]

We define the *SIMD utilization* of a program run on a GPU as the fraction of SIMD lanes that are kept busy with *active threads* during the run of a program. As we saw in lecture, the SIMD utilization of a program is computed across the *complete run* of the program.

The following code segment is run on a GPU. Each thread executes **a single iteration** of the shown loop. Assume that the data values of the arrays A, B, and C are already in vector registers so there are no loads and stores in this program. (Hint: Notice that there are 6 instructions in each thread.) A warp in the GPU consists of 64 threads, and there are 64 SIMD lanes in the GPU. Please assume that all values in array B have magnitudes less than 10 (i.e., $|B[i]| < 10$, for all i).

```
for (i = 0; i < 1024; i++) {
  A[i] = B[i] * B[i];
  if (A[i] > 0) {
    C[i] = A[i] * B[i];
    if (C[i] < 0) {
      A[i] = A[i] + 1;
    }
    A[i] = A[i] - 2;
  }
}
```

Please answer the following five questions.

(a) How many warps does it take to execute this program?

Warps = (Number of threads) / (Number of threads per warp)
Number of threads = 2^{10} (i.e., one thread per loop iteration).
Number of threads per warp = $64 = 2^6$ (given).
Warps = $2^{10}/2^6 = 2^4$

(b) What is the maximum possible SIMD utilization of this program?

100%

(c) Please describe what needs to be true about array B to reach the maximum possible SIMD utilization asked in part (b). (Please cover all cases in your answer)

For every 64 consecutive elements: every value is 0, every value is positive, or every value is negative. Must give all three of these.

(d) What is the minimum possible SIMD utilization of this program?

Answer: 132/384

(e) Please describe what needs to be true about array B to reach the minimum possible SIMD utilization asked in part (d). (Please cover all cases in your answer)

Exactly 1 of every 64 consecutive elements must be negative. The rest must be zero. This is the only case that this holds true.

10 GPUs and SIMD (II) [100 points]

We define the *SIMD utilization* of a program that runs on a GPU as the fraction of SIMD lanes that are kept busy with *active threads* during the run of a program. As we saw in lecture, the SIMD utilization of a program is computed across the *complete run* of the program.

The following code segment is run on a GPU. Each thread executes **a single iteration** of the shown loop. Assume that the data values of the arrays `A` and `B` are already in vector registers so there are no loads and stores in this program. (Hint: Notice that there are 3 instructions in each iteration.) A warp in the GPU consists of 32 threads, and there are 32 SIMD lanes in the GPU.

```
for (i = 0; i < 1025; i++) {
    if (A[i] < 33) {        // Instruction 1
        B[i] = A[i] << 1;   // Instruction 2
    }
    else {
        B[i] = A[i] >> 1;   // Instruction 3
    }
}
```

Please answer the following six questions.

(a) How many warps does it take to execute this program?

33 warps.

Explanation:
The number of warps is calculated as:
$\#Warps = \lceil \frac{\#Total\_threads}{\#Warp\_size} \rceil$,

where
$\#Total\_threads = 1025 = 2^{10} + 1$ (i.e., one thread per loop iteration),

and
$\#Warp\_size = 32 = 2^5$ (given).

Thus, the number of warps needed to run this program is:
$\#Warps = \lceil \frac{2^{10}+1}{2^5} \rceil = 2^5 + 1 = 33$.

(b) What is the maximum possible SIMD utilization of this program? (Hint: The warp scheduler does not issue instructions where no threads are active).

$\frac{1025}{1056}$.

Explanation:
Even though all active threads in a warp follow the same execution path, the last warp will only have one active thread.

(c) Please describe what needs to be true about array `A` to reach the maximum possible SIMD utilization asked in part (b). (Please cover all cases in your answer.)

For every 32 consecutive elements of `A`, every element should be lower than 33 (`if`), or greater than or equal to 33 (`else`). (NOTE: The solution is correct if both cases are given.)

(d) What is the minimum possible SIMD utilization of this program?

$\frac{1025}{1568}$.

Explanation:
`Instruction 1` is executed by every active thread ($\frac{1025}{1056}$ utilization).
Then, part of the threads in each warp executes `Instruction 2` and the other part executes `Instruction 3`. We consider that `Instruction 2` is executed by α threads in each warp (except the last warp), where $0 < \alpha \leq 32$, and `Instruction 3` is executed by the remaining $32 - \alpha$ threads.
The only active thread in the last warp executes either `Instruction 2` or `Instruction 3`. The other instruction is not issued for this warp.

The minimum SIMD utilization sums to $\frac{1025+\alpha\times32+(32-\alpha)\times32+1}{1056+1024+1024+32} = \frac{1025}{1568}$.

(e) Please describe what needs to be true about array `A` to reach the minimum possible SIMD utilization asked in part (d). (Please cover all cases in your answer.)

For every 32 consecutive elements of `A`, part of the elements should be lower than 33 (`if`), and the other part should be greater than or equal to 33 (`else`).

(f) What is the SIMD utilization of this program if `A[i] = i`? Show your work.

$\frac{1025}{1072}$.

Explanation:
`Instruction 1` is executed by every active thread ($\frac{1025}{1056}$ utilization).
`Instruction 2` is executed by the first 33 threads, i.e., all threads in the first warp and one thread in the second warp.
`Instruction 3` is executed by the remaining active threads.

The SIMD utilization sums to $\frac{1025+32+1+31+960+1}{1056+32+32+32+960+32} = \frac{2050}{2144} = \frac{1025}{1072}$.

11 SIMD Processing [50 points]

Suppose we want to design a SIMD engine that can support a vector length of 16. We have two options: a traditional vector processor and a traditional array processor.

(a) Which one is more costly in terms of chip area (circle one)? Please explain.

The traditional vector processor | **The traditional array processor** | Neither

An array processor requires 16 functional units for an operation whereas a vector processor requires only 1.

(b) Assuming the latency of an addition operation is five cycles in both processors, how long will a `VADD` (vector add) instruction take in each of the processors (assume that the adder can be fully pipelined and is the same for both processors)?

- For a vector length of 1:

  The traditional vector processor:
  5 cycles

  The traditional array processor:
  5 cycles

- For a vector length of 4:

  The traditional vector processor:
  8 cycles (5 for the first element to complete, 3 for the remaining 3)
  The traditional array processor:
  5 cycles

- For a vector length of 16:

  The traditional vector processor:
  20 cycles (5 for the first element to complete, 15 for the remaining 15)
  The traditional array processor:
  5 cycles