ETH 263-2210-00L Computer Architecture, Fall 2021
HW 2: Processing-in-Memory and Genome Analysis

Instructor: Prof. Onur Mutlu
TAs: Juan Gómez Luna, Mohammed Alser, Jisung Park, Lois Orosa Nogueira, Gagandeep Singh, Haiyu Mao, Behzad Salami, Nour Almadhoun Alserr, Mohammad Sadr, Hasan Hassan, Can Firtina Geraldo Francisco De Oliveira Junior, Abdullah Giray Yaglikci, Rahul Bera, Konstantinos Kanellopoulos, Nika Mansouri Ghiasi, Rakesh Nadig, João Dinis Ferreira, Haocong Luo, Roknoddin Azizibarzoki

Given: Mon, Oct 25, 2021
Due: **Sun, Nov 07, 2021**

- **Handin - Critical Paper Reviews (1).** You need to submit your reviews to `https://safari.ethz.ch/review/architecture21/`. Please, check your inbox, you should have received an email with the password you should use to login. If you did not receive any email, contact comparch@lists.inf.ethz.ch. In the first page after login, you should click in "Computer Architecture Home", and then go to "any submitted paper" to see the list of papers.
- **Handin - Questions (2-5).** You should upload your answers to the Moodle platform (`https://moodle-app2.let.ethz.ch/course/view.php?id=15536`) as a single PDF file.
- If you have any questions regarding this homework, please ask them the Moodle forum (`https://moodle-app2.let.ethz.ch/mod/moodleoverflow/view.php?id=657929`).
- Please note that the handin questions have a hard deadline. However, you can submit your paper reviews till the end of the semester.

1. Critical Paper Reviews [1,000 points]

Please read the guidelines for reviewing papers and check the sample reviews. We also assign you three **required readings** for this homework. 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. If you review a paper other than the REQUIRED papers, you will receive 250 BONUS points on top of 1,000 points you may get from paper reviews (i.e., each additional submission is worth 250 BONUS points with a possibility to get up to 5,500 points).

Guidelines

Sample reviews

Required Reading 1

Required Reading 2

Required Reading 3

Write an approximately one-page critical review for the following required readings (i.e., papers #1 to #3) **and** *at least* 1 more from the remaining 19 papers (i.e., papers #4 to #22). 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.

1. **(REQUIRED)** Seshadri et al., "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. **(REQUIRED)** Senol Cali el al., "GenASM: A High-Performance, Low-Power Approximate String Matching Acceleration Framework for Genome Sequence Analysis", MICRO, 2020. https://people.inf.ethz.ch/omutlu/pub/GenASM-approximate-string-matching-framework-for-genome-analysis_micro20.pdf
3. **(REQUIRED)** Ahn et al., "A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing", ISCA 2015, https://people.inf.ethz.ch/omutlu/pub/tesseract-pim-architecture-for-graph-processing_isca15.pdf
4. Mutlu et al., "A Modern Primer on Processing in Memory", Invited Book Chapter in Emerging Computing: From Devices to Systems - Looking Beyond Moore and Von Neumann, Springer https://people.inf.ethz.ch/omutlu/pub/ModernPrimerOnPIM_springer-emerging-computing-bookchapter21.pdf
5. Ahn et al., "PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture", ISCA 2015, https://people.inf.ethz.ch/omutlu/pub/pim-enabled-instructons-for-low-overhead-pim_isca15.pdf
6. Boroumand et al., "Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks", ASPLOS 2018, https://people.inf.ethz.ch/omutlu/pub/Google-consumer-workloads-data-movement-and-PIM_asplos18.pdf
7. Singh et al., "FPGA-based Near-Memory Acceleration of Modern Data-Intensive Applications", IEEE Micro 2021, https://arxiv.org/pdf/2106.06433.pdf
8. Seshadri et al., "RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization", MICRO 2013, https://people.inf.ethz.ch/omutlu/pub/rowclone_micro13.pdf
9. Seshadri et al., "Gather-Scatter DRAM: In-DRAM Address Translation to Improve the Spatial Locality of Non-unit Strided Accesses", MICRO 2015, https://people.inf.ethz.ch/omutlu/pub/GSDRAM-gather-scatter-dram_micro15.pdf
10. Wang et al., "FIGARO: Improving System Performance via Fine-Grained In-DRAM Data Relocation and Caching", MICRO 2020, https://people.inf.ethz.ch/omutlu/pub/FIGARO-fine-grained-in-DRAM-data-relocation-and-caching_micro20.pdf
11. Hajinazar and Oliveira et al., "SIMDRAM: An End-to-End Framework for Bit-Serial SIMD Computing in DRAM", ASPLOS 2021, https://people.inf.ethz.ch/omutlu/pub/SIMDRAM_asplos21.pdf
12. Alser et al., "Accelerating Genome Analysis: A Primer on an Ongoing Journey", IEEE Micro 2020 https://people.inf.ethz.ch/omutlu/pub/AcceleratingGenomeAnalysis_ieeemicro20.pdf
13. Lee et al., "Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology", ISCA 2021 https://ieeexplore.ieee.org/document/9499894
14. Harold Stone, "A Logic-in-Memory Computer", TC 1970 https://safari.ethz.ch/architecture/fall2020/lib/exe/fetch.php?media=stone_logic_in_memory_1970.pdf
15. Dunn and Sadasivan et al., "SquiggleFilter: An Accelerator for Portable Virus Detection", MICRO 2021, https://arxiv.org/pdf/2108.06610.pdf
16. Boroumand et al., "Google Neural Network Models for Edge Devices: Analyzing and Mitigating Machine Learning Inference Bottlenecks", PACT 2021 https://people.inf.ethz.ch/omutlu/pub/Google-neural-networks-for-edge-devices-Mensa-Framework_pact21.pdf
17. Giannoula et al., "SynCron: Efficient Synchronization Support for Near-Data-Processing Architectures", HPCA 2021 https://people.inf.ethz.ch/omutlu/pub/SynCron-synchronization-for-near-data-processing-systems_hpca21.pdf
18. Alser et al., "GateKeeper: A New Hardware Architecture for Accelerating Pre-Alignment in DNA Short Read Mapping", Bioinformatics 2017, https://people.inf.ethz.ch/omutlu/pub/gatekeeper_FPGA-genome-prealignment-accelerator_bionformatics17.pdf
19. Alser et al., "Shouji: A Fast and Efficient Pre-Alignment Filter for Sequence Alignment", Bioinformatics 2019, https://people.inf.ethz.ch/omutlu/pub/shouji-genome-prealignment-filter_bionformatics19.pdf
20. Alser et al., "SneakySnake: A Fast and Accurate Universal Genome Pre-Alignment Filter for CPUs, GPUs, and FPGAs", Bioinformatics 2020, https://people.inf.ethz.ch/omutlu/pub/SneakySnake_UniversalGenomePrealignmentFilter_bioinformatics20.pdf

21. Kim et al., "GRIM-Filter: Fast Seed Location Filtering in DNA Read Mapping Using Processing-in-Memory Technologies", APBC 2018, https://arxiv.org/pdf/1711.01177.pdf
22. Turakhia et al., "Darwin: A Genomics Co-processor Provides up to 15,000× acceleration on long read assembly", ASPLOS 2018, http://bejerano.stanford.edu/papers/p199-turakhia.pdf

0 0 0 0 0 0 0 0 -> 0 0 0 0

2. Processing-in-Memory [150 points]

You have been hired to accelerate ETH's student database. After profiling the system for a while, you found out that one of the most executed queries is to *"select the hometown of the students that are from Switzerland and speak German"*. The attributes *hometown*, *country*, and *language* are encoded using a four-byte binary representation. The database has 32768 (2^{15}) entries, and each attribute is stored contiguously in memory. The database management system executes the following query:

```
bool position_hometown[entries];
for(int i = 0; i < entries; i++){
    if(students.country[i] == "Switzerland" && students.language[i] == "German"){
        position_hometown[i] = true;
    }
    else{
        position_hometown[i] = false;
    }
}
```

(a) You are running the above code on a single-core processor. Assume that:

- Your processor has an 8 MB direct-mapped cache, with a cache line of 64 bytes.
- A hit in this cache takes one cycle and a miss takes 100 cycles for both load and store operations.
- All load/store operations are serialized, i.e., the latency of multiple memory requests cannot be overlapped.
- The starting addresses of *students.country*, *students.language*, and *position_hometown* are 0x05000000, 0x06000000, 0x07000000 respectively.
- The execution time of a non-memory instruction is zero (i.e., we ignore its execution time).

How many cycles are required to run the query? Show your work.

We need for each loop iteration 3 memory accesses (country, language, hometown)
Each space where they are stored is of size 2^{24} B
The DM cache is of size 2^{23} B and there are 2^{15} entries
Which means that at every request, the cache will need to reload as the tag doesn't match.
For each request, it will be a miss.

The number of cycles required is $N_c = 2^{15} \cdot 100 \cdot 3$

(b) Recall that in class we discussed AMBIT, which is a DRAM design that can greatly accelerate Bulk Bitwise Operations by providing the ability to perform bitwise AND/OR/XOR of two rows in a sub-array. AMBIT works by issuing back-to-back ACTIVATE (A) and PRECHARGE (P) operations. For example, to compute AND, OR, and XOR operations, AMBIT issues the sequence of commands described in the table below (e.g., $AAP(X, Y)$ represents double row activation of rows X and Y followed by a precharge operation, $AAAP(X, Y, Z)$ represents triple row activation of rows X, Y, and Z followed by a precharge operation).
In those instructions, AMBIT copies the source rows D_i and D_j to auxiliary rows (B_i). Control rows C_i dictate which operation (AND/OR) AMBIT executes. The DRAM rows with dual-contact cells (i.e., rows DCC_i) are used to perform the bitwise NOT operation on the data stored in the row. Basically, copying a source row to DCC_i flips all bits in the source row and stores the result in both the source row and DCC_i. Assume that:

- The DRAM row size is **8 Kbytes**.
- An ACTIVATE command takes 50 cycles to execute.
- A PRECHARGE command takes 20 cycles to execute.
- DRAM has a single memory bank.
- The syntax of an AMBIT operation is: *bbop_*[and/or/xor] *destination, source_ 1, source_ 2.*
- Addresses 0x08000000 and 0x09000000 are used to store partial results.
- The rows at addresses 0x0A000000 and 0x0B00000 store the codes for *"Switzerland"* and *"German"*, respectively, in each four bytes throughout the entire row.

$D_k = D_i$ **AND** D_j	$D_k = D_i$ **OR** D_j	$D_k = D_i$ **XOR** D_j
		AAP (D_i, B_0)
		AAP (D_j, B_1)
		AAP (D_i, DCC_0)
AAP (D_i, B_0)	AAP (D_i, B_0)	AAP (D_j, DCC_1)
AAP (D_j, B_1)	AAP (D_j, B_1)	AAP (C_0, B_2)
AAP (C_0, B_2)	AAP (C_1, B_2)	AAAP (B_0, DCC_1, B_2)
AAAP (B_0, B_1, B_2)	AAAP (B_0, B_1, B_2)	AAP (C_0, B_2)
AAP B_0, D_k	AAP B_0, D_k	AAAP (B_1, DCC_0, B_2)
		AAP (C_1, B_2)
		AAAP (B_0, B_1, B_2)
		AAP (B_0, D_k)

i) The following code aims to execute the query *"select the hometown of the students that are from Switzerland and speak German"* in terms of Boolean operations to make use of AMBIT. Fill in the blank boxes such that the algorithm produces the correct result. Show your work.

```
for(int i = 0; i < [ 16 ] ; i++){

  bbop_[ xor ] 0x08000000, 0x05000000 + i*8192, 0x0A000000;

  bbop_[ xor ] 0x09000000, 0x06000000 + i*8192, 0x0B000000;

  bbop_[ or ] 0x07000000, 0x08000000, 0x09000000;
}
```

0010
0001 → 0011

The limit for the condition in the for loop is $\frac{\text{number of bytes}}{\text{row size}} = \frac{2^{16} \cdot 4}{2^{13}} = 16.$

For the first two operations, we need xors because we'd like to find any bit difference between the language / country code and the input. If they match, the output rows will be filled by 0s. (addr 0x0600000 and 0x0900000)
The last command needs to be a OR, that will be all 0's expect where is doesn't match.

ii) How much speedup does AMBIT provide over the baseline processor when executing the same query? Show your work.

We compute the speedup like so:

AMBIT number of cycles needed:

N = number iterations · (N_A · cycles per activation + N_p · cycles per precharge)

= 16 · (2 XORs + 1 OR)

= 16 · (2 · (2 5A + 1 1P) + 11A + 5P)

= 16 · (61A + 27P) = 16 · (61 · 50 + 27 · 20)

= 57440 cycles

Speedup = $\frac{\text{BASELINE}}{\text{AMBIT}} = \frac{2^{15} \cdot 100 \cdot 3}{57440}$

3. In-DRAM Bit Serial Computation [150 points]

Recall that in class, we discussed Ambit, which is a DRAM design that can greatly accelerate bulk bitwise operations by providing the ability to perform bitwise AND/OR of two rows in a subarray and NOT of one row. Since Ambit is logically complete, it is possible to implement any other logic gate (e.g., XOR). To be able to implement arithmetic operations, bit shifting is also necessary. There is no way of shifting bits in DRAM with a conventional layout, but it can be done with a bit-serial layout, as Figure 1 shows. With such a layout, it is possible to perform bit-serial arithmetic computations in Ambit.

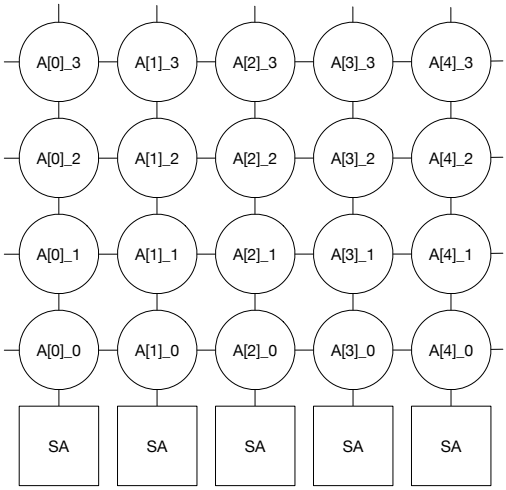


Figure 1. In-DRAM bit-serial layout for array `A`, which contains five 4-bit elements. DRAM cells in the same bitline contain the bits of an array element: `A[i]_j` represents bit `j` of element `i`.

We want to evaluate the potential performance benefits of using Ambit for arithmetic computations by implementing a simple workload, the element-wise addition of two arrays. Listing 1 shows a sequential code for the addition of two input arrays `A` and `B` into output array `C`.

Listing 1. Sequential CPU implementation of element-wise addition of arrays `A` and `B`.

```
for(int i = 0; i < num_elements; i++){
    C[i] = A[i] + B[i];
}
```

We compare two possible implementations of the element-wise addition of two arrays: a CPU-based and an Ambit-based implementation. We make two assumptions. First, we use the most favorable layout for each implementation (i.e., conventional layout for CPU, and bit-serial layout for Ambit). Second, both implementations can operate on array elements of any size (i.e., bits/element):

- *CPU-based implementation*: This implementation reads elements of `A` and `B` from memory, adds them, and writes the resulting elements of `C` into memory.
  Since the computation is simple and regular, we can use a simple analytical performance model for the execution time of the CPU-based implementation: $t_{cpu} = K \times num\_operations + \frac{num\_bytes}{M}$, where K represents the cost per arithmetic operation and M is the DRAM bandwidth. Note: $num\_operations$ should include only the operations for the array addition.
- *Ambit-based implementation*: This implementation assumes a bit serial layout for arrays `A`, `B`, and `C`. It performs additions in a bit serial manner, which only requires XOR, AND, and OR operations, as you can see in the 1-bit full adder in Figure 2.

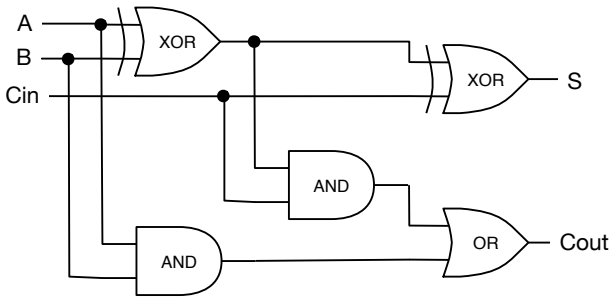


Figure 2. 1-bit full adder.

Ambit implements these operations by issuing back-to-back ACTIVATE (A) and PRECHARGE (P) operations. For example, to compute AND, OR, and XOR operations, Ambit issues the sequence of commands described in Table 1, where $AAP(X, Y)$ represents double row activation of rows X and Y followed by a precharge operation, and $AAAP(X, Y, Z)$ represents triple row activation of rows X, Y, and Z followed by a precharge operation.

In those instructions, Ambit copies the source rows D_i and D_j to auxiliary rows (B_i). Control rows C_i dictate which operation (AND/OR) Ambit executes. The DRAM rows with dual-contact cells (i.e., rows DCC_i) are used to perform the bitwise NOT operation on the data stored in the row. Basically, the NOT operation copies a source row to DCC_i, flips all bits of the row, and stores the result in both the source row and DCC_i. Assume that:

- The DRAM row size is 8 Kbytes.
- An ACTIVATE command takes 20ns to execute.
- A PRECHARGE command takes 10ns to execute.
- DRAM has a single memory bank.
- The syntax of an Ambit operation is: *bbop_*[and/or/xor] *destination, source_1, source_2.*
- The rows at addresses 0x00700000, 0x00800000, and 0x00900000 are used to store partial results. Initially, they contain all zeroes.
- The rows at addresses 0x00A00000, 0x00B00000, and 0x00C00000 store arrays `A`, `B`, and `C`, respectively.
- These are all byte addresses. All these rows belong to the same DRAM subarray.

Table 1. Sequences of ACTIVATE and PRECHARGE operations for the execution of Ambit's AND, OR, and XOR.

$D_k = D_i$ **AND** D_j	$D_k = D_i$ **OR** D_j	$D_k = D_i$ **XOR** D_j
		AAP (D_i, B_0)
		AAP (D_j, B_1)
		AAP (D_i, DCC_0)
AAP (D_i, B_0)	AAP (D_i, B_0)	AAP (D_j, DCC_1)
AAP (D_j, B_1)	AAP (D_j, B_1)	AAP (C_0, B_2)
AAP (C_0, B_2)	AAP (C_1, B_2)	AAAP (B_0, DCC_1, B_2)
AAAP (B_0, B_1, B_2)	AAAP (B_0, B_1, B_2)	AAP (C_0, B_2)
AAP B_0, D_k	AAP B_0, D_k	AAAP (B_1, DCC_0, B_2)
		AAP (C_1, B_2)
		AAAP (B_0, B_1, B_2)
		AAP (B_0, D_k)

(a) For the CPU-based implementation, you want to obtain K and M. To this end, you run two experiments. In the first experiment, you run your CPU code for the element-wise array addition for 65,536 4-bit elements and measure $t_{cpu} = 100$ us. In the second experiment, you run the STREAM-Copy benchmark for 102,400 4-bit elements and measure $t_{cpu} = 10$ us. The STREAM-Copy benchmark simply copies the contents of one input array `A` to an output array `B`. What are the values of K and M?

CPU $T = k \cdot n_{op} + \frac{nbytes}{M}$

- The stream-copy benchmark does not do any arithmetic operations, so we have $10\mu s = \frac{102400 \times 2 \times 4}{M \times 8} \Leftrightarrow M = \frac{102400 \times 2 \times 4}{10 \times 10^{-6} \times 8} = 10,24\ GB/s$

  As a side note, we need 2×4 bits per elements because we have two arrays.

- For k, we have $k = \left(100\mu s - \frac{65536 \times 4 \times 3}{10,24 \times 10^{9} \times 8}\right) / 65536$ (we need to access 3 arrays)

  $= 1.38 \times 10^{-9}$ seconds per operation

(b) Write the code for the Ambit-based implementation of the element-wise addition of arrays `A` and `B`. The resulting array is `C`.

8KB $= 2^3 \cdot 2^{10} = 2 \cdot 10^{12} = 0x2000$

```
for (int i = 0; i < n; i++)
{
    bbop_xor 0x00900000, 0x00A00000 + i * 0x2000, 0x00B00000 + i * 0x2000; //9 = A XOR B
    bbop_xor 0x00C00000 + i * 0x2000, 0x00700000, 0x00900000                //S
    bbop_and 0x00800000, 0x00900000, 0x00700000; //9 = 9 and Cin
    bbop_and 0x00900000, 0x00A00000 + i * 0x2000, 0x00B00000 + i * 0x2000 //Cin is 0x00800000
    bbop_or  0x00700000, 0x00900000, 0x00800000; //Cout
}
```

(c) Compute the maximum throughput (in arithmetic operations per second, OPS) of the Ambit-based implementation as a function of the element size (i.e., bits/element).

The throughput is given by the following formula,

$$C = (2\,XOR + 2\,AND + OR) \cdot m \quad \leftarrow \text{number of bits per element}$$

$$= ([50 + 22 + 11]A + [22 + 10 + 5]P) \cdot m$$

$$= [83A + 37P] \cdot m = 2030 \cdot m \; ns$$

The lowest Throughput (in OPS) is reacheable when the element size is 8 KB (the row size)

Thus the throughput $T = \frac{65536}{2030 \cdot m \cdot ns} = \frac{32.28}{m}$ GOPS

(d) Determine the element size (in bits) for which the CPU-based implementation is faster than the Ambit-based implementation (Note: Use the same array size as in the previous part).

We want that $T_{AMBIT} < T_{CPU}$ (operation / second)

For that, I write $t_{AMB} > t_{CPU}$ (second per op)

So the condition is $\frac{2030 \cdot m \cdot ns}{65536} > \left(1.38 \cdot ns - \frac{65536 \cdot m}{10.24 \times 10^{-9} \times 8}\right) / 65536$

$\Leftrightarrow$ There isn't a positive m that satisfies this condition.

In conclusion, AMBIT will always be faster than the CPU.

4. Caching vs. Processing-in-Memory [150 points]

We are given the following piece of code that makes accesses to integer arrays A and B. The size of each element in both A and B is 4 bytes. The base address of array A is $0x00001000$, and the base address of B is $0x00008000$.

```
movi R1, #0x1000 // Store the base address of A in R1
movi R2, #0x8000 // Store the base address of B in R2
movi R3, #0

Outer_Loop: x 16                                   A0   B0   A0   B1
    movi R4, #0
    movi R7, #0
    Inner_Loop: x 2
        add R5, R3, R4  // R5 = R3 + R4
        // load 4 bytes from memory address R1+R5
        ld R5, [R1, R5] // R5 = Memory[R1 + R5],        } x2   LOAD
        ld R6, [R2, R4] // R6 = Memory[R2 + R4]         }      LOAD
        mul R5, R5, R6  // R5 = R5 * R6
        add R7, R7, R5  // R7 += R5
        inc R4          // R4++
        bne R4, #2, Inner_Loop // If R4 != 2, jump to Inner_Loop

    //store the data of R7 in memory address R1+R3
    st [R1, R3], R7     // Memory[R1 + R3] = R7,   x1      STORE
    inc R3              // R3++
    bne R3, #16, Outer_Loop // If R3 != 16, jump to Outer_Loop
```

You are running the above code on a single-core processor. For now, assume that the processor *does not* have caches. Therefore, all load/store instructions access the main memory, which has a fixed 50-cycle latency, for both read and write operations. Assume that all load/store operations are serialized, i.e., the latency of multiple memory requests *cannot* be overlapped. Also assume that the execution time of a non-memory-access instruction is zero (i.e., we ignore its execution time).

(a) What is the execution time of the above piece of code in cycles?

Inner: 2 x 2 mem_acc
Outer: (Inner + 1 mem access) · 16
Total: Outer · 50 = 4000 cycles

$$\frac{128}{8} = \frac{2^7}{2^3} = 2^4 = 16$$

(b) Assume that a 128-byte private cache is added to the processor core in the next-generation processor. The cache block size is 8-byte. The cache is direct-mapped. On a hit, the cache services both read and write requests in 5 cycles. On a miss, the main memory is accessed and the access fills an 8-byte cache line in 50 cycles. Assuming that the cache is initially empty, what is the new execution time on this processor with the described cache? Show your work.

D-M Cache. M for Miss H for Hit

If we give names to the access (R_1+R_5, R_2+R_4, R_1+R_5, R_2+R_4, R_1+R_5 of the outer loop pattern) A_0 B_0 A_1 B_1 A_0

Iteration 0: A_0 B_0 A_1 B_1 A_0
M M M M M — A_0, A_1 in set 0

1: A_1 B_0 A_2 B_1 A_1 — A_2, A_3 in set 1, A_0, A_1 in set 0
H M M H M

2: A_2 B_0 A_3 B_1 A_2 — A_2, A_3 in set 1, B_0, B_1 in set 0
H M H H H

3: A_3 B_0 A_4 B_1 A_3 — A_4, A_5 in set 2
H H M H H

4: A_4 B_0 A_5 B_1 A_4
H H H H H

From there, 6, 8, 10, 12, 14, will have 5 hits each → 25 cc
and 5, 7, 9, 11, 13, 15 will have 4 hits, 4 miss → 370 cc

The hit count in total is $0+2+4+4+5\cdot6+4\cdot8$
" miss count " " is $5+3+4+1+1\cdot6$

The numb of cycles required is: 1120 cycles

(c) You are not satisfied with the performance after implementing the described cache. To do better, you consider utilizing a processing unit that is available *close to the main memory*. This processing unit can directly interface to the main memory with a *10-cycle* latency, for both read and write operations. How many cycles does it take to execute the same program using the in-memory processing units? (Assume that the in-memory processing unit does not have a cache, and the memory accesses are serialized like in the processor core. The latency of the non-memory-access operations is ignored.)

It is the same latency as when there's no cache, but w/ a reduced latency.

$N_C = 16 \cdot 5 \cdot 10 = 800$

(d) You friend now suggests that, by changing the cache capacity of the single-core processor (in part (b)), she could provide as good performance as the system that utilizes the memory processing unit (in part (c)). Is she correct? What is the minimum capacity required for the cache of the single-core processor to match the performance of the program running on the memory processing unit?

That's not correct because the two accesses Ao, Bo would still cause a miss in set 0.

(e) [10 points] What other changes could be made to the cache design to improve the performance of the single-core processor on this program?

We need the cache to have higher associativity
(we need more sets than a direct-mapped cache)

5. Genome Analysis [150 points]

During a process called read mapping in genome analysis, each genomic read (i.e., DNA sequence fragment) is mapped onto one or more possible locations in the reference genome based on the similarity between the read and the reference genome segment at that location. A read mapper applies the following 3-step hash table-based mapping method:

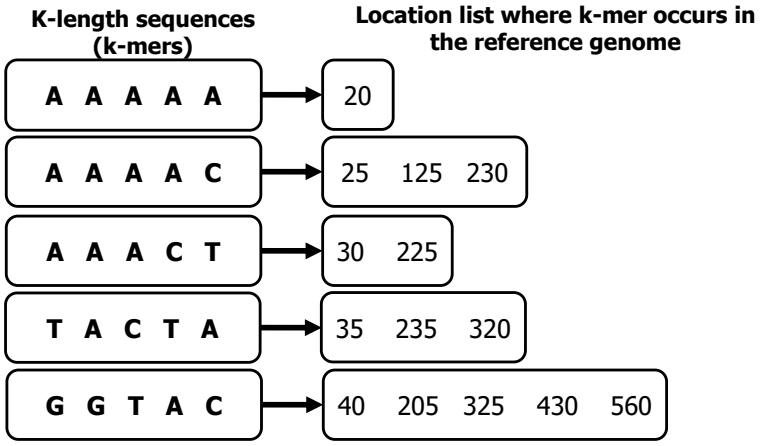
(1) The hash table-based read mapper first constructs a hash table that stores the list of locations in the reference genome that each possible short segment (i.e., k-mer, where k is the length of the segment) appears. Querying the hash table with a k-mer returns a list of locations for that k-mer.

(2) For each read, the mapper extracts 3 consecutive non-overlapping 5-mers and uses them to query the hash table.

(3) For each location of a k-mer, the mapper examines the differences between the entire read that includes the k-mer and the corresponding reference segment using the function: (*edit_distance()*). Allowable edit operations include: (1) *substitution*, (2) *insertion*, and (3) *deletion of a character*. For example, edit operations between the read sequence ATATTTATA and the reference sequence ATAAGAT are as follows:

```
ATATTTATA-      Read sequence
| | |    |  |   (3 matches, 3 insertions, 1 match, 1 substitution, 1 match, 1 deletion)
ATA---AGAT      Reference sequence
```

The hash table (constructed in Step 1) is provided below. It includes a list of 5-mers extracted from the human reference genome and their corresponding location lists (each number represents the starting location of that k-mer in the reference genome sequence). If a k-mer does *not* exist in the hash table, you can assume it does not appear in the reference genome. Answer the following questions based on this hash table whenever needed.



5.1. Edit Distance Computation

(a) Compute the *edit distance* for the following read and reference sequence pair and provide the complete list of the edit operations used for calculation. Show your work.
Read sequence: ATCCTTAAATCTAAAATT
Reference sequence: CCTTAGAAACTTAA

Read: A T C C T T A A A T C T A A A A T T
operation: I I : : : : : R I I I I : : : R : : D D
Ref: . . C C T T A G . . . . A A A C T T AA

2 insertions, 5 matches, 1 replace, 4 insertions, 3 matches, 1 replace, 2 match, 2 deletions

Edit_distance is equal to 2 + 1 + 4 + 1 + 2 = 10

(b) We would like to figure out (i.e., reverse engineer) the read sequence based on the following information available to us:

- The length of the read is 10.
- The *first* 5-mer of a read is found at the location 430 from the hash table.
- *edit_distance()* function returns edit distance value of 3 between the read sequence and the human reference sequence starting from location 430. At least one of these three edits is a **deletion**.
- The reference sequence segment used for the *edit_distance()* calculation is GGTACATAG.

Write down a read sequence that fits the criteria and show the complete list of edit operations. Note that more than one solution is possible.

Loc 430: GGTAC ATAG
R R D
Read is GGTAC AGA G T

edit_distance with 3 operations (one possible solution)

5.2. Read Mapping

Suppose that you would like to map the following reads to the human reference genome sequence. Each read is separated into smaller subsequences (k-mers) by underscores for readability.

read 1 = AAAAA_AAAAC_AAACT
read 2 = TACTA_GGTAC_AAACT
read 3 = GGTAC_AAACT_AAAAT
read 4 = AAAAC_TACTA_GGTAC

(a) How many times will the edit distance function, *edit_ distance()*, be invoked when following the mapping steps described at the beginning of the question?

Read 1: 1 + 3 + 2 = 6
" 2: 3 + 5 + 2 = 10
" 3: 5 + 2 + 0 = 7
" 4: 3 + 3 + 5 = 11

} Σ = 34 calls in total

(b) Suppose you want to change Step 3 to *additionally* include "*Adjacency Filtering*" (as discussed in lecture) in order to find *exact matches* before calling the *edit_ distance()* function. This means that, if we find an exact match via Adjacency Filtering, the *edit_ distance()* function is *not* invoked. Adjacency Filtering checks if first, second, and third k-mers extracted from the read exist in the reference genome at locations x, $x + k$, and $x + 2 \times k$, respectively.
At what locations in the reference genome does Adjacency Filtering find exact matches?

R1: 20, 25, 30 : 1 exact match
R2: no exact match
R3: no exact match
R4: no exact match