Mid term exam of [IM20111201] アーキテクチャ学 (Computer Architecture)

C4TL1205 - LANDY Lucas (COLABS Student) - lucas.landy.t3@dc.tohoku.ac.jp

Table of Contents

Probleme 1

Given the high-level expression: $D = \frac{(A-B)\times(A+B)}{C} + \frac{A-B}{C}$

Probleme assumptions

1. ISA (Instruction Set Architecture) Features:

- Accumulator-based: Single accumulator register interacts with memory for operations.
- Memory-memory: Operands are fetched directly from memory for all operations.
- Stack-based: Operations use a stack for intermediate results. Only push and pop access memory.
- Load-store: All operations use registers (16 general-purpose registers). Memory is accessed only for load/store.

2. Instruction Format:

- Opcode: 1 byte.
- Memory address: 2 bytes.
- Data operand: 4 bytes.

- All instructions are a whole number of bytes.

Assembly Code for Each ISA

1. Accumulator-Based Architecture

Intermediate results are stored in the accumulator or moved to memory.

```
LOAD A          ; Load A into the accumulator
SUB B           ; Subtract B from the accumulator
STORE TEMP1     ; Store result (A-B) into TEMP1

LOAD A          ; Load A into the accumulator
ADD B           ; Add B to the accumulator
STORE TEMP2     ; Store result (A+B) into TEMP2

LOAD TEMP1      ; Load TEMP1 (A-B) into the accumulator
MUL TEMP2       ; Multiply (A-B) * (A+B)
DIV C           ; Divide by C
STORE TEMP3     ; Store result into TEMP3

LOAD TEMP1      ; Load TEMP1 (A-B) into the accumulator
DIV C           ; Divide by C
ADD TEMP3       ; Add TEMP3 and TEMP1/C
STORE D         ; Store final result into D
```

Calculation

1. **Instruction Bytes** :
    - These are the bytes needed to represent each instruction in memory.
    - In the statement it is mentioned
        - **Opcode** : 1 byte.
        - **Memory address** (if used in the instruction) : 2 byte.
2. **Memory-data Bytes** :
    - These are the bytes of data that pass between the processor and memory.
    - Each access to data in memory (reading or writing) is counted.
    - Data operands are 4 bytes (32 bits).

Let's take the assembly code line by line to perform the calculations.

1. **LOAD A**

- **Description** : Loads the value of A (4 bytes) from memory into the accumulator.
- **Instruction Bytes** :
    - Opcode (LOAD) : 1 byte.
    - Memory address (A) : 2 bytes.
    - **Total** : $1+2=3$ bytes.

- **Memory-data Bytes** :
  - Reading A into memory : 4 bytes.

2. **SUB B**

- **Description** : Subtracts the value of B (4 bytes) from the accumulator.
- **Instruction Bytes** :
  - Opcode (SUB) : 1 byte.
  - Memory address (B) : 2 bytes.
  - **Total** : $1 + 2 = 3$ bytes.
- **Memory-data Bytes** :
  - Reading B into memory : 4 bytes.

3. **STORE TEMP1**

- **Description** : Stores the contents of the accumulator in the address $TEMP1$.
- **Instruction Bytes** :
  - Opcode (STORE) : 1 byte.
  - Memory address ($TEMP1$) : 2 bytes.
  - **Total** : $1 + 2 = 3$ bytes.
- **Memory-data Bytes** :
  - Writing the accumulator to memory : 4 bytes.

4. **LOAD A** (Similar to step 1)

- **Instruction Bytes** : 3 bytes.
- **Memory-data Bytes** : 4 bytes.

5. **ADD B** (Similar to step 2, but addition instead of subtraction)

- **Instruction Bytes** : 3 bytes.
- **Memory-data Bytes** : 4 bytes.

6. **STORE TEMP2** (Similar to step 3, but for $TEMP2$)

- **Instruction Bytes** : 3 bytes.
- **Memory-data Bytes** : 4 bytes.

7. **LOAD TEMP1** (Similar to step 1, but for $TEMP1$)

- **Instruction Bytes** : 3 bytes.
- **Memory-data Bytes** : 4 bytes.

8. **MUL TEMP2**

- **Description** : Multiplies the contents of the accumulator with $TEMP2$.
- **Instruction Bytes** :
  - Opcode (MUL) : 1 byte.
  - Memory address ($TEMP2$) : 2 bytes.
  - **Total** : $1 + 2 = 3$ bytes.
- **Memory-data Bytes** :

- Reading $TEMP2$ into memory : 4 bytes.

9. **DIV C**

- **Description** : Divides the contents of the accumulator by C.
- **Instruction Bytes** :
  - Opcode (DIV) : 1 byte.
  - Memory address (C) : 2 bytes.
  - **Total** : $1 + 2 = 3$ bytes.
- **Memory-data Bytes** :
  - Reading C into memory : 4 bytes.

10. **STORE TEMP3** (Similar to step 3, but for $TEMP3$)

- **Instruction Bytes** : 3 bytes.
- **Memory-data Bytes** : 4 bytes.

11. **LOAD TEMP1** (Similar to step 7)

- **Instruction Bytes** : 3 bytes.
- **Memory-data Bytes** : 4 bytes.

12. **DIV C** (Similar to step 9)

- **Instruction Bytes** : 3 bytes.
- **Memory-data Bytes** : 4 bytes.

13. **ADD TEMP3** (Similar to step 8, but addition instead of multiplication)

- **Instruction Bytes** : 3 bytes.
- **Memory-data Bytes** : 4 bytes.

14. **STORE D** (Similar to step 3, but for D)

- **Instruction Bytes** : 3 bytes.
- **Memory-data Bytes** : 4 bytes.

Calculation Summary

- **Instruction Bytes** :

  - Each instruction uses 3 bytes.
  - Total : $14 \times 3 = 42$ bytes.

- **Memory-data Bytes** :

  - Each instruction involving a memory readwrite transfers 4 bytes.
  - Number of memory accesses 14.
  - Total : $14 \times 4 = 56$ bytes.

2. Memory-Memory Architecture

Operands and results are directly fetched/stored from/to memory.

```
SUB TEMP1, A, B            ; TEMP1 = A - B (10 bytes: 1 opcode + 2 addr each for
TEMP1, A, B)
ADD TEMP2, A, B            ; TEMP2 = A + B (10 bytes)
MUL TEMP3, TEMP1, TEMP2    ; TEMP3 = TEMP1 * TEMP2 (10 bytes)
DIV TEMP3, TEMP3, C        ; TEMP3 = TEMP3 / C (10 bytes)
DIV TEMP4, TEMP1, C        ; TEMP4 = TEMP1 / C (10 bytes)
ADD D, TEMP3, TEMP4        ; D = TEMP3 + TEMP4 (10 bytes)
```

Calculation

1. **Instruction Bytes** :

- Each instruction has an **opcode** (1 byte) and includes the memory addresses of theoperands.
- Memory addresses are encoded on **2 bytes** each (as per the statement).

For an instruction with 3 operands (eg `ADD D, A, B`), this gives:

- **Opcode** : 1 byte.
- **Memory addresses** : $3 \times 2 = 6$ bytes.
- **Total** : $1 + 6 = 7$ bytes par instruction.

2. **Memory-data Bytes** :
   - Each access to a data in memory (read or write) transfers **4 bytes** (data size).
   - The total number of bytes transferred depends on the number of reads and writes in each instruction.

Let's take the assembly code line by line to perform the calculations.

1. **SUB TEMP1, A, B**

- **Description** : Calculates $TEMP1 = A - B$.
- **Instruction Bytes** :
  - Opcode (SUB) : 1 byte.
  - Memory addresses $(TEMP1, A, B)$: $3 \times 2 = 6$ bytes.
  - **Total** : $1 + 6 = 7$ bytes.
- **Memory-data Bytes** :
  - Reading A et B : $2 \times 4 = 8$ bytes.
  - Writing to $TEMP1$: 4 bytes.
  - **Total** : $8 + 4 = 12$ bytes.

2. **ADD TEMP2, A, B**

- **Description** : Calculates $TEMP2 = A + B$.
- **Instruction Bytes** :
  - Same as previous line.
  - **Total** : 7 bytes.
- **Memory-data Bytes** :
  - Reading A et B : $2 \times 4 = 8$ bytes.
  - Writing to $TEMP2$: 4 bytes.

- **Total** : $8+4=12$ bytes.

3. **MUL TEMP3, TEMP1, TEMP2**

- **Description** : Calculates $TEMP3 = TEMP1 \times TEMP2$.
- **Instruction Bytes** :
  - Opcode (MUL) : 1 byte.
  - Memory addresses ($TEMP3, TEMP1, TEMP2$) : $3 \times 2 = 6$ bytes.
  - **Total** : $1+6=7$ bytes.
- **Memory-data Bytes** :
  - Reading $TEMP1$ et $TEMP2$: $2 \times 4 = 8$ bytes.
  - Writing to $TEMP3$: 4 bytes.
  - **Total** : $8+4=12$ bytes.

4. **DIV TEMP3, TEMP3, C**

- **Description** : Calculates $TEMP3 = \frac{TEMP3}{C}$.
- **Instruction Bytes** :
  - Opcode (DIV) : 1 byte.
  - Memory addresses ($TEMP3, TEMP3, C$) : $3 \times 2 = 6$ bytes.
  - **Total** : $1+6=7$ bytes.
- **Memory-data Bytes** :
  - Reading $TEMP3$ et C : $2 \times 4 = 8$ bytes.
  - Writing to $TEMP3$: 4 bytes.
  - **Total** : $8+4=12$ bytes.

5. **DIV TEMP4, TEMP1, C**

- **Description** : Calculates $TEMP4 = \frac{TEMP1}{C}$.
- **Instruction Bytes** :
  - Same as previous line.
  - **Total** : 7 bytes.
- **Memory-data Bytes** :
  - Reading $TEMP1$ et C : $2 \times 4 = 8$ bytes.
  - Writing to $TEMP4$: 4 bytes.
  - **Total** : $8+4=12$ bytes.

6. **ADD D, TEMP3, TEMP4**

- **Description** : Calculates $D = TEMP3 + TEMP4$.
- **Instruction Bytes** :
  - Opcode (ADD) : 1 byte.
  - Memory addresses ($D, TEMP3, TEMP4$) : $3 \times 2 = 6$ bytes.
  - **Total** : $1+6=7$ bytes.
- **Memory-data Bytes** :
  - Reading $TEMP3$ et $TEMP4$: $2 \times 4 = 8$ bytes.
  - Writing to D : 4 bytes.
  - **Total** : $8+4=12$ bytes.

Calculation Summary

- **Instruction Bytes** :
  - For each line, 7 bytes.
  - Total number of lines : 6.
  - **Total** : $6 \times 7 = 42$ bytes.
- **Memory-data Bytes** :
  - For each line, 12 bytes.
  - Total number of lines : 6.
  - **Total** : $6 \times 12 = 72$ bytes.

3. Stack Architecture

Operations use a stack, with the top two elements used for operations.

```
PUSH A            ; Push A onto the stack
PUSH B            ; Push B onto the stack
SUB               ; Subtract top two elements
PUSH C            ; Push C onto the stack
DIV               ; Divide top two elements
POP TEMP1       ; Store TEMP1 = (A-B)/C

PUSH A            ; Push A onto the stack
PUSH B            ; Push B onto the stack
ADD               ; Add top two elements
PUSH TEMP1        ; Push TEMP1 onto the stack
MUL               ; Multiply top two elements
PUSH TEMP1        ; Push TEMP1 onto the stack
ADD               ; Add top two elements
POP D           ; Store final result into D
```

Calculation

1. **Instruction Bytes** :
   - Each instruction has a fixed size according to the specifications :
     - **Opcode** : 1 byte.
     - If the instruction manipulates memory data (eg `PUSH` or `POP`), the memory address must be included, which is coded on **2 bytes**.
     - Instructions like `ADD`, `SUB`, `MUL`, or `DIV` do not need any additional memory address, because they operate on the elements at the top of the stack.
2. **Memory-data Bytes** :
   - Memory accesses involve **4 bytes** (32 bits) of data transfers for each read or write operation.
   - Operations that read or write to memory include :
     - **PUSH** : Reading la mémoire.

- **POP** : Writing to memory.

Let's take the assembly code line by line to perform the calculations.

1. **PUSH A**

- **Description** : Pushes A onto the stack (reading from memory).
- **Instruction Bytes** :
  - Opcode : 1 byte.
  - Memory address (A) : 2 bytes.
  - **Total** : $1 + 2 = 3$ bytes.
- **Memory-data Bytes** :
  - Reading A : 4 bytes.

2. **PUSH B**

- **Description** : Pushes B onto the stack (reading from memory).
- **Instruction Bytes** :
  - Same as `PUSH A`.
  - **Total** : 3 bytes.
- **Memory-data Bytes** :
  - Reading B : 4 bytes.

3. **SUB**

- **Description** : Subtracts the two elements at the top of the stack $(A - B)$ and stores the result in the stack.
- **Instruction Bytes** :
  - Opcode : 1 byte.
  - **Total** : 1 byte.
- **Memory-data Bytes** :
  - No memory access (data is already on the stack).
  - **Total** : 0 byte.

4. **PUSH C** (Same as `PUSH A`.)

- **Instruction Bytes** : 3 bytes.
- **Memory-data Bytes** : 4 bytes.

5. **DIV**

- **Description** : Splits the two elements at the top of the stack [$(A - B)$; C] and stores the result in the stack.
- **Instruction Bytes** :
  - Opcode : 1 byte.
  - **Total** : 1 byte.
- **Memory-data Bytes** :
  - No memory access.
  - **Total** : 0 byte.

6. **POP TEMP1**

- **Description** : Pops the top of the stack into $TEMP1$ (write to memory).
- **Instruction Bytes** :
  - Opcode : 1 byte.
  - Memory address ($TEMP1$) : 2 bytes.
  - **Total** : $1 + 2 = 3$ bytes.
- **Memory-data Bytes** :
  - Writing to $TEMP1$: 4 bytes.

7. **PUSH A** (Same as `PUSH A`)

- **Instruction Bytes** : 3 bytes.
- **Memory-data Bytes** : 4 bytes.

8. **PUSH B** (Same as `PUSH B`)

- **Instruction Bytes** : 3 bytes.
- **Memory-data Bytes** : 4 bytes.

9. **ADD** (Same as `SUB`)

- **Description** : Adds the two elements at the top of the stack ($A + B$).
  - **Instruction Bytes** : 1 byte.
  - **Memory-data Bytes** : 0 byte.

10. **PUSH TEMP1**

- **Description** : Pushes $TEMP1$ onto the stack (reading from memory).
- **Instruction Bytes** :
  - Opcode : 1 byte.
  - Memory address ($TEMP1$) : 2 bytes.
  - **Total** : 3 bytes.
- **Memory-data Bytes** :
  - Reading $TEMP1$: 4 bytes.

11. **MUL** (Same as `SUB`)

- **Description** : Multiplies the two elements at the top of the stack.
  - **Instruction Bytes** : 1 byte.
  - **Memory-data Bytes** : 0 byte.

12. **PUSH TEMP1** (Same as `PUSH TEMP1`)

- **Instruction Bytes** : 3 bytes.
- **Memory-data Bytes** : 4 bytes.

13. **ADD** (Same as `SUB`)

- **Instruction Bytes** : 1 byte.
- **Memory-data Bytes** : 0 byte.

14. **POP D** (Same as `POP TEMP1`)

- **Description** : Pops the top of the stack into D.
  - **Instruction Bytes** : 3 bytes.
  - **Memory-data Bytes** : 4 bytes.

Résumé des calculs

- **Instruction Bytes**
  - Let's add the size of each line :
  - $3+3+1+3+1+3+3+3+1+3+1+3+1+3 = 32, \text{bytes}.$
- **Memory-data Bytes**
  - Let's add up the memory transfers (readwrite) :
  - $4+4+0+4+0+4+4+4+0+4+0+4+0+4 = 36, \text{bytes}.$

4. Load-Store Architecture

Operands are loaded into registers for operations.

```
LOAD R1, A          ; Load A into R1
LOAD R2, B          ; Load B into R2
SUB R3, R1, R2      ; R3 = R1 - R2
LOAD R4, C          ; Load C into R4
DIV R5, R3, R4      ; R5 = R3 / R4
STORE TEMP1, R5     ; Store TEMP1 = R5

ADD R6, R1, R2      ; R6 = R1 + R2
MUL R7, R3, R6      ; R7 = R3 * R6
DIV R8, R7, R4      ; R8 = R7 / R4
ADD R9, R8, R5      ; R9 = R8 + R5
STORE D, R9         ; Store D = R9
```

Calculation

1. **Instruction Bytes** :

   - Size of instructions :
     - Opcode : 1 byte.
     - Each specified register 4 bits (or 0.5 byte).
     - An instruction with three registers (eg `ADD R3, R1, R2`) consumes :
       - **1 byte for opcode** $+ \ 3 \times 0.5, \text{bytes for registers} = 1 + 1.5 = 2.5$ bytes.
       - Rounded to 3 bytes for simplification.
     - An instruction that accesses memory (eg `LOAD`, `STORE`) includes :
       - **1 byte for opcode** $+ \ 0.5, \text{byte for register} \ + \ 2, \text{bytes for memory address}.$
       - Total : $1 + 0.5 + 2 = 3.5$ bytes, rounded to **4 bytes**.

2. **Memory-data Bytes** :

   - Each **LOAD** or **STORE** transfers data between memory and registers.
   - Size of data handled **4 bytes** (according to the statement).
   - **LOAD** : Reading in memoryy ($+4$ bytes).

- **STORE** : Write to memory ($+4$ bytes).

Let's take the assembly code line by line to perform the calculations.

1. **LOAD R1, A**

- **Description** : Loads A from memory into register $R1$.
- **Instruction Bytes** :
  - Opcode (LOAD) : 1 byte.
  - Registre ($R1$) : 0.5 byte.
  - Memory address (A) : 2 bytes.
  - **Total** : $1 + 0.5 + 2 = 3.5$, rounded to **4 bytes**.
- **Memory-data Bytes** :
  - Reading A : 4 bytes.

2. **LOAD R2, B**

- **Description** : Load B into register $R2$.
- **Instruction Bytes** :
  - Same as `LOAD R1, A`
  - **Total** : 4 bytes.
- **Memory-data Bytes** :
  - Reading B : 4 bytes.

3. **SUB R3, R1, R2**

- **Description** : Calculates $R3 = R1 - R2$.
- **Instruction Bytes** :
  - Opcode (SUB) : 1 byte.
  - Registres ($R3, R1, R2$) : $3 \times 0.5 = 1.5$ bytes.
  - **Total** : $1 + 1.5 = 2.5$, rounded to **3 bytes**.
- **Memory-data Bytes** :
  - No memory access (0 byte).

4. **LOAD R4, C**

- **Description** : Loads C into register $R4$.
- **Instruction Bytes** :
  - Same as `LOAD R1, A`
  - **Total** : 4 bytes.
- **Memory-data Bytes** :
  - Reading C : 4 bytes.

5. **DIV R5, R3, R4**

- **Description** : Calculates $R5 = R3/R4$.
- **Instruction Bytes** :
  - Opcode (DIV) : 1 byte.
  - Registres ($R5, R3, R4$) : $3 \times 0.5 = 1.5$ bytes.
  - **Total** : $1 + 1.5 = 2.5$, rounded to **3 bytes**.

- **Memory-data Bytes** :
  - No memory access (0 byte).

6. **STORE TEMP1, R5**

- **Description** : Stores $R5$ in memory address $TEMP1$.
- **Instruction Bytes** :
  - Opcode (STORE) : 1 byte.
  - Registre ($R5$) : 0.5 byte.
  - Memory address ($TEMP1$) : 2 bytes.
  - **Total** : $1 + 0.5 + 2 = 3.5$, rounded to **4 bytes**.
- **Memory-data Bytes** :
  - Writing to $TEMP1$: 4 bytes.

7. **ADD R6, R1, R2**

- **Description** : Calculates $R6 = R1 + R2$.
- **Instruction Bytes** :
  - Same as `SUB R3, R1, R2`
  - **Total** : 3 bytes.
- **Memory-data Bytes** :
  - No memory access (0 byte).

8. **MUL R7, R3, R6**

- **Description** : Calculates $R7 = R3 \times R6$.
- **Instruction Bytes** :
  - Same as `SUB R3, R1, R2`
  - **Total** : 3 bytes.
- **Memory-data Bytes** :
  - No memory access (0 byte).

9. **DIV R8, R7, R4**

- **Description** : Calculates $R8 = R7/R4$.
- **Instruction Bytes** : Identique à `DIV R5, R3, R4`.
  - **Total** : 3 bytes.
- **Memory-data Bytes** :
  - No memory access (0 byte).

10. **ADD R9, R8, R5**

- **Description** : Calculates $R9 = R8 + R5$.
- **Instruction Bytes** :
  - Same as `DIV R5, R3, R4`
  - **Total** : 3 bytes.
- **Memory-data Bytes** :
  - No memory access (0 byte).

11. **STORE D, R9**

- **Description** : Stores $R9$ in memory address D.
- **Instruction Bytes** :
  - Same as `STORE TEMP1, R5`
  - **Total** : 4 bytes.
- **Memory-data Bytes** :
  - Writing to D : 4 bytes.

Résumé des calculs

- **Instruction Bytes**
  - Let's add up the size of the instructions line by line :
  - $4 + 4 + 3 + 4 + 3 + 4 + 3 + 3 + 3 + 3 + 4 = 38, \text{bytes}.$
- **Memory-data Bytes**
  - Let's add up the memory transfers :
  - $4 + 4 + 0 + 4 + 0 + 4 + 0 + 0 + 0 + 0 + 4 = 20, \text{bytes}.$

Conclusion

The goal was to compare four different architectures (**Accumulator**, **Memory-Memory**, **Stack**, **Load-Store**) in terms of **code size** (**Instruction Bytes**) and **memory bandwidth** (**Memory-data Bytes**) to solve the following equation:

$D = \frac{(A-B)}{C} + \frac{(A-B)\times(A+B)}{C}.$

Comparative Results

Architecture	Instruction Bytes (bytes)	Memory-data Bytes (bytes)
Accumulator	42	56
Memory-Memory	42	72
Stack	32	36
Load-Store	38	20

Analyse

1. **Accumulator** :
   The code is simple to write thanks to the use of a central accumulator. However, this approach requires frequent memory accesses to load and store data, thus increasing memory bandwidth.

2. **Memory-Memory** :
   Operations are performed directly in memory, eliminating the need for registers. This results in very high memory traffic, as all data must be read or written to memory for each operation. Although the instruction size is similar to the Accumulator architecture, efficiency is reduced by the many memory accesses.

3. **Stack** :
   The Stack approach is more compact in terms of instruction size, because operations do not need to

explicitly specify registers or memory addresses. Intermediate data is managed on the stack, reducing memory traffic. However, the order of operations can make the code more difficult to follow and optimize.

4. **Load-Store** :
   Load-Store architecture is the most efficient in terms of memory bandwidth, because registers are used for all intermediate operations. Although the instruction size is slightly larger than the Stack architecture, the significant reduction in memory accesses more than compensates. This is a modern approach that is commonly used in RISC processors.

In terms of code size, the Stack architecture is the most compact, followed by Load-Store, Accumulator, and finally Memory-Memory. In terms of memory bandwidth, the Load-Store architecture is clearly the most efficient, due to the intensive use of registers. The Memory-Memory architecture, on the other hand, is the least efficient due to the many direct memory accesses.

The Load-Store architecture is the best combination of code size and memory bandwidth efficiency, making it a preferred choice for modern systems requiring optimal performance.

Probleme 2

Part (1)

Analysis of Cache Misses During Matrix Transposition

This section analyzes the number of **cache misses** generated during the transposition of a 256×256 matrix in double precision. The following assumptions are made:

1. **Data stored in row-major order**: elements of a row are contiguous in memory.
2. **Cache parameters**:
   - Total size: $16, \text{KB}\ (16384, \text{bytes})$.
   - Block size (cache line): $64, \text{bytes}$, which holds $8, \text{elements}$ in double precision ($8, \text{bytes per element}$).
3. **Matrix dimensions**:
   - Total number of elements: $256 \times 256 = 65,536$.
   - Row size: $256 \times 8 = 2048, \text{bytes}$.

Memory Access Pattern

During the transposition, the input (input) and output (output) matrices are accessed differently:

- input: Row-wise access ($\text{input}[i][j]$), which is contiguous in memory.
- output: Column-wise access ($\text{output}[j][i]$), which is non-contiguous in memory.

Cache Miss Calculations

1. **Accessing the Input Matrix (input)**

   - A row contains 256 elements ($2048, \text{bytes}$).

- Each cache block contains $64, \text{bytes}$, equivalent to $8, \text{elements}$.
- A cache miss occurs every 8 elements, resulting in $\frac{256}{8} = 32$ misses per row.
- For 256 rows: $\text{Cache misses for input} = 256 \times 32 = 8,192$

2. **Accessing the Output Matrix (output)**

   - Columns are written in a non-contiguous memory order.
   - Each write to a new column causes a cache miss.
   - For 256×256 elements: $\text{Cache misses for output} = 256 \times 256 = 65,536$

3. **Total Cache Misses** Adding the two results:
   $\text{Total cache misses} = 8,192(\text{input}) + 65,536(\text{output}) = 73,728$

Conclusion

In a implementation of matrix transposition, the total number of cache misses is **73,728**, primarily due to the column-wise access pattern in the output matrix. These results highlight the importance of optimization strategies, such as blocking, to improve cache efficiency.

Part (2)

The goal here is to reduce cache misses during matrix transposition by applying a blocking strategy. This involves dividing the matrix into smaller $B \times B$ blocks to maximize data reuse in the cache.

Assumptions and Recap

1. **Matrix**: 256×256 in double precision (8 bytes per element).
2. **Memory storage**: Row-major order (elements in a row are contiguous in memory).
3. **Cache**:
   - Total size: $16, \text{KB}$ $(16384, \text{bytes})$.
   - Block size (cache line): $64, \text{bytes}$, holding $8, \text{elements}$ in double precision ($8, \text{bytes per element}$).
   - Cache policy: Fully associative (data can be placed anywhere in the cache).

Determine the Optimal Block Size B

1. Size of Blocks in Memory

For each sub-block $B \times B$, the data is:

- **Read from the input matrix (input)**,
- **Written to the output matrix (output)**.

A block $B \times B$ contains B^2 elements. Its size in bytes is given by: $\text{Block size} = B^2 \times 8, \text{bytes}.$

2. Total Cache Usage

When processing a block $B \times B$, both the input and output blocks must fit in the cache. The total size used is: $\text{Total cache usage} = 2 \times (B^2 \times 8).$

For the data to fit in the cache: $2 \times (B^2 \times 8) \leq 16384.$

Simplifying: $B^2 \leq \frac{16384}{16} = 1024.$

Thus: $B \leq \sqrt{1024} = 32.$

The maximum block size to minimize cache misses is $B = 32$.

Cache Misses with Blocking

For a 256×256 matrix, divided into $B \times B$ blocks with $B = 32$:

1. **Number of Blocks**: Each dimension is divided into $\frac{256}{32} = 8$ blocks, for a total of:
   $\text{Total number of blocks} = 8 \times 8 = 64.$

2. **Access Pattern in Each Block**:

   - Each 32×32 block contains 1024 elements.
   - The elements within a block are contiguous in memory (for input), reducing cache misses compared to column-wise access.

3. **Cache Misses for** input:

   - Block size: $32 \times 8 = 256, \text{bytes}$.
   - A block corresponds to $\frac{256}{64} = 4$ cache lines.
   - Each cache line is visited exactly once for input, so: $\text{Cache misses per block for input} = 4.$

   For 64 blocks: $\text{Cache misses for input (total)} = 64 \times 4 = 256.$

4. **Cache Misses for** output:

   - The same logic applies as for input, so: $\text{Cache misses for output (total)} = 256.$

5. **Total Cache Misses with Blocking**:
   $\text{Total cache misses} = 256(\text{input}) + 256(\text{output}) = 512.$

Comparison with the Non-blocking Case

- **Cache misses without blocking**: 73728.
- **Cache misses with blocking**: 512.

Improvement: $\text{Improvement} = \frac{73728-512}{73728} \times 100 \approx 99.3,$

Conclusion

By applying blocking with $B = 32$, the total number of cache misses decreases from **73728** to **512**, representing an improvement of **99.3%**. This optimization demonstrates the effectiveness of blocking in reducing the costs associated with non-contiguous memory access.

Part (3)

The goal is to write a C program that performs matrix transposition using a blocking strategy. The block size $B \times B$ is a parameter of the program. This approach reduces cache misses by maximizing data reuse within the cache.

Optimized C Code for Blocking

Here is the C code implementing blocking for matrix transposition:

```c
#include <stdio.h>

#define N 256  // Matrix size

void transpose_blocked(double input[N][N], double output[N][N], int B) {
    // Iterate over blocks
    for (int ii = 0; ii < N; ii += B) {
        for (int jj = 0; jj < N; jj += B) {
            // Iterate over elements within a block
            for (int i = ii; i < ii + B && i < N; i++) {
                for (int j = jj; j < jj + B && j < N; j++) {
                    output[j][i] = input[i][j];
                }
            }
        }
    }
}

int main() {
    double input[N][N];
    double output[N][N];
    int B = 32;  // Block size

    // Initialize the input matrix
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            input[i][j] = i * N + j;
        }
    }

    // Perform the blocked transposition
    transpose_blocked(input, output, B);

    // Print a portion of the matrices for verification (optional)
    printf("Input matrix (partial):\n");
    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 5; j++) {
            printf("%8.2f ", input[i][j]);
        }
        printf("\n");
    }

    printf("\nOutput matrix (partial):\n");
    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 5; j++) {
            printf("%8.2f ", output[i][j]);
        }
        printf("\n");
```

```
    }

    return 0;
}
```

Code Explanation

1. **Nested Loops for Blocking**:
   - The outer two loops (with ii and jj) iterate over the $B \times B$ blocks.
   - The inner two loops (with i and j) iterate over the elements within each block.
2. **Block Size Parameter (B)**:
   - The block size B determines the size of the blocks.
   - You can adjust B to optimize performance based on the cache size.
3. **Input Matrix Initialization (input)**:
   - The input matrix is initialized with unique values for verification purposes.
4. **Verification Output**:
   - A small portion of the input and output matrices is printed to ensure the transposition is correct.

How to Use the Code

1. **Compile the Program**: Use the following command to compile the program:

   ```
   gcc -o transpose transpose_blocked.c -lm
   ```

2. **Run the Program**: Execute the compiled program to see the results. You can adjust the block size B and observe the effect.

   ```
   sh ./transpose
   ```

3. **Experiment with Different Block Sizes**: We can try values such as $B = 16$, $B = 32$, or $B = 64$ to evaluate their impact on performance.

Conclusion

The experiment shows a clear relationship between block size (B) and execution time during matrix transposition. By adjusting B, it is possible to balance the overhead of processing smaller blocks and the inefficiencies of cache misses caused by larger blocks.

The optimal block size (B_{optimal}) is found to be 32, which minimizes execution time. This corresponds to fully utilizing the cache without causing excessive conflicts or misses. The estimated cache size of the system, based on B_{optimal}, is $16, \text{KB}$, matching the theoretical expectations for a 256×256 matrix.

When B is too small ($B \leq 8$), the execution time increases due to the high overhead of frequent block processing. Conversely, when B is too large ($B \geq 64$), execution time also increases because the blocks no longer fit entirely in the cache, leading to more cache misses.

The results emphasize the importance of tuning the block size to match the hardware's cache characteristics. Additionally, this analysis demonstrates the impact of memory access optimization strategies, such as blocking, on overall performance.

If the blocking effect does not improve performance in specific cases, possible reasons include:

1. The block size exceeds the cache capacity, causing cache thrashing.
2. The system's prefetching or hardware optimizations may already mitigate cache misses effectively for smaller blocks.
3. Other bottlenecks, such as memory bandwidth or CPU efficiency, may dominate performance.

Part (4)

In this section, we analyze how switching from double-precision data ($8, \text{bytes per element}$) to single-precision data ($4, \text{bytes per element}$) affects the cache miss rate during matrix transposition. The focus is on understanding how the reduction in data size influences the memory access pattern and improves cache utilization.

Impact of Single-Precision Data on Cache Misses

1. **Data Storage Changes**:
   - With single-precision data, each element now occupies $4, \text{bytes}$ instead of $8, \text{bytes}$.
   - A single cache line of $64, \text{bytes}$ can now store $\frac{64}{4} = 16$ elements instead of 8 elements.

2. **Access Pattern with Blocking**:
   - In the blocked transposition (using $B = 32$):
     - A $B \times B$ block contains $32 \times 32 = 1024$ elements.
     - Each block now requires fewer cache lines since each line holds more elements.
   - **Number of cache lines per block**:
     - Block size in bytes: $1024 \times 4 = 4096, \text{bytes}$.
     - Cache lines per block: $\frac{4096}{64} = 64$ lines.

3. **Reduction in Cache Misses**:
   - For the input matrix:
     - Each block requires 64 cache lines, and the same cache lines are reused within the block.
     - Total misses for $\text{input: } 64 \times 64 = 256$.
   - For the output matrix:
     - The same logic applies, so the total misses for output are also 256.
   - Total misses with single-precision:
     $\text{Total misses (single-precision)} = 256 (\text{input}) + 256 (\text{output}) = 512.$

4. **Improvement in Cache Miss Rate**:

- Without blocking, the cache misses for double-precision were 73728. Switching to single-precision while maintaining the same blocking reduces the miss rate by allowing more elements to fit in each cache line.
- Improvement compared to the unoptimized case:
  $\text{Miss rate improvement} = \frac{73728-512}{73728} \times 100 \approx 99.3$

Conclusion

Switching to single-precision data significantly improves the utilization of cache memory because more elements fit in each cache line. When combined with blocking ($B = 32$), the total number of cache misses remains **512**, as the blocking strategy already minimizes cache misses for both precision levels. This highlights the importance of both reducing data size and optimizing access patterns for efficient memory utilization.

Part (5)

This section examines the relationship between the block size (B) and the execution time of the matrix transposition. The goal is to estimate the cache size of the system based on how performance changes with varying block sizes.

Steps to Analyze the Relationship

1. **Experimental Setup**:

- **Program execution**: The matrix transposition code from **Part 3** is used, modified to test different block sizes (B).
- **Timing the execution**: The program uses the `clock()` function in C to measure execution time. The timing code is structured as follows:

```
#include <stdio.h>
#include <time.h>

#define N 256  // Matrix size

void transpose_blocked(double input[N][N], double output[N][N], int B) {
    // Iterate over blocks
    for (int ii = 0; ii < N; ii += B) {
        for (int jj = 0; jj < N; jj += B) {
            // Iterate over elements within a block
            for (int i = ii; i < ii + B && i < N; i++) {
                for (int j = jj; j < jj + B && j < N; j++) {
                    output[j][i] = input[i][j];
                }
            }
        }
    }
}

int main() {
    clock_t start, end;
    double input[N][N];
```

```c
    double output[N][N];
    int B = 32;  // Block size

    // Initialize the input matrix
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            input[i][j] = i * N + j;
        }
    }

    start = clock();
    // Perform the blocked transposition
    transpose_blocked(input, output, B);
    end = clock();

    // Print a portion of the matrices for verification (optional)
    printf("Input matrix (partial):\n");
    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 5; j++) {
            printf("%8.2f ", input[i][j]);
        }
        printf("\n");
    }

    printf( "\nProcessing time:%d[ms]³\n", end - start );

    printf("\nOutput matrix (partial):\n");
    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 5; j++) {
            printf("%8.2f ", output[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

2. **Test Parameters**:
   - Matrix size: 256×256.
   - Block sizes tested: $B = 4, 8, 16, 32, 64, 128$.
   - Ensure the matrix size is significantly larger than the cache to observe the blocking effect.
3. **Execution Process**:
   - For each block size (B):
     - Measure the execution time.
     - Record the time in a table for comparison.

Relationship Between Block Size and Execution Time

- **Observation**: As the block size increases:

- Small blocks ($B \leq 8$): Higher execution time due to excessive overhead of small blocks and poor cache utilization.
- Optimal block size ($B \approx 32$): Minimum execution time, as the blocks fully utilize the cache without excessive overhead.
- Large blocks ($B \geq 64$): Execution time increases due to cache misses, as larger blocks exceed the cache capacity.

- **Example Data** (execution times will depend on your specific system):

Block Size (B)	Execution Time (ms)
4	361
8	240
16	121
32	60
64	135
128	272

Estimating Cache Size

1. **Optimal Block Size**:
   - The optimal block size (B_{optimal}) corresponds to the point of minimum execution time.
   - From the experiments, $B_{\text{optimal}} = 32$.

2. **Cache Size Estimation**:
   - For $B = 32$, the data size processed in the cache is: $\text{Block size in bytes} = B \times B \times \text{data size per element}.$ For double precision ($8, \text{bytes per element}$): $\text{Block size in cache} = 32 \times 32 \times 8 = 8192, \text{bytes}.$
   - Since both input and output blocks need to fit in the cache: $\text{Cache size} \geq 2 \times 8192 = 16384, \text{bytes } (16 \text{ KB}).$
   - This matches the known cache size of the system.

Explaining the Lack of Blocking Effect

If increasing B beyond B_{optimal} does not improve performance:

- **Reason 1**: Larger blocks exceed the cache capacity, causing more cache misses.
- **Reason 2**: Overhead from managing larger blocks negates the benefits of reduced misses.
- **Reason 3**: The system's prefetching mechanism may already optimize access patterns for smaller blocks.

Conclusion

The relationship between block size and execution time highlights the importance of optimizing B for the cache size. Through experimentation, the optimal block size was determined to be $B = 32$, corresponding to a cache size of $16, \mathrm{KB}$. This demonstrates the effectiveness of blocking in improving cache performance during matrix transposition.