UCLouvain EPL-INMA 16 June 2026	**Exam : LINMA2710** Scientific Computing	First name : Last name : NOMA :

Important instructions:

- The exam duration is 2 hours.
- Please start your answer just below each question (when possible) and continue on a new sheet of paper if necessary. Put your name on all sheets.
- As mentioned in the course outline, only personal handwritten notes and annotated printouts of the material posted on the course Moodle page are allowed. All the rest, including electronic devices (laptop, smartphone, etc.), is forbidden.
- Do not use red ink.
- The size of the answer box is chosen with a quite conservative margin so do not worry if your answer is shorter than the box.

---

1. Consider this C++ code, in which lines 14–17 have been removed:

```
1  #include <cmath>
2  #include <iostream>
3  using namespace std;
4
5  int operate (int a, int b)
6  {
7    return (a*b);
8  }
9
10 double operate (double a, double b)
11 {
12   return (a/b);
13 }
```

Solution:

```
14 double operate (double a, int exp)
15 {
16     return pow(a,exp);
17 }
```

```
19 int main ()
20 {
21   int m=5,n=2;
22   double y=5.0,z=2.0;
23   cout << operate (y,z) << '\n';
24   cout << operate (m,n) << '\n';
25   cout << operate (y,n) << '\n';
26   return 0;
27 }
```

(a) What output does line 23 produce? Briefly explain.

Solution: Line 23 produces 2.5. In line 23, in `operate (y,z)`, the arguments `y` and `z` have type `double`. Hence the function defined on lines 10–13 is called, and it returns 5.0/2.0, namely 2.5.

(b) What output does line 24 produce? Briefly explain. What programming features does this illustrate?

Solution: Line 24 produces 10. In line 24, in `operate (m,n)`, variables `m` and `n` have type `int`. Hence the function defined on lines 5–8 is called, and it returns 5*2, namely 10. This illustrates the programming feature termed *function overloading*.

(c) The prototype of function `pow`, that returns `base` raised to the power `exponent`, is
`double pow (double base, double exponent)`.
Fill out lines 14–17 in the box above such that line 25 prints 5.0 to the power 2, namely 25.0.

(d) Does the completed code perform narrowing cast or widening cast? Briefly explain.

Solution: In `pow(a,exp)`, function `pow` receives an `int` as second argument, whereas its prototype, `double pow (double base, double exponent)`, specifies that the type of the second parameter is `double`. As a consequence, the `int` variable `exp` is cast to type `double` in the scope of function `pow`. The destination `double` has a higher precision than the source `int`, hence this is a widening cast.

2. Consider Poisson's equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = 1$$

in a cube $\Omega = \{(x, y, z) \in \mathbb{R}^3 : 0 \leq (x, y, z) \leq L\}$. The boundary conditions are zero Dirichlet conditions. Consider an axis-aligned grid with steps $\Delta_x = L/J_x$, $\Delta_y = L/J_y$, $\Delta_z = L/J_z$.

(a) Write a finite volume scheme for this problem based on the given grid.

Solution: Let $U_{i,j,k}$ denote the approximate solution of the PDE at vertex $(i\Delta_x, j\Delta_y, k\Delta_z) \in \Omega$ for $i = 0, \ldots, J_x$, $j = 0, \ldots, J_y$, $k = 0, \ldots, J_z$. The scheme is

$$\begin{aligned}\frac{U_{i+1,j,k} - U_{i,j,k}}{\Delta_x}\Delta_y\Delta_z + \frac{U_{i-1,j,k} - U_{i,j,k}}{\Delta_x}\Delta_y\Delta_z \\ + \frac{U_{i,j+1,k} - U_{i,j,k}}{\Delta_y}\Delta_z\Delta_x + \frac{U_{i,j-1,k} - U_{i,j,k}}{\Delta_y}\Delta_z\Delta_x \\ + \frac{U_{i,j,k+1} - U_{i,j,k}}{\Delta_z}\Delta_x\Delta_y + \frac{U_{i-1,j,k-1} - U_{i,j,k}}{\Delta_z}\Delta_x\Delta_y \\ = \Delta_x\Delta_y\Delta_z\end{aligned}$$

for $i = 1, \ldots, J_x - 1$, $j = 1, \ldots, J_y - 1$, $k = 1, \ldots, J_z - 1$, with $U_{i,j,k} = 0$ whenever $i = 0$ or $i = J_x$ or $j = 0$ or $j = J_y$ or $k = 0$ or $k = J_z$.

Observe that the scheme simplifies to

$$\begin{aligned}\frac{U_{i+1,j,k} - 2U_{i,j,k} + U_{i-1,j,k}}{\Delta_x^2} + \frac{U_{i,j+1,k} - 2U_{i,j,k} + U_{i,j-1,k}}{\Delta_y^2} \\ + \frac{U_{i,j,k+1} - 2U_{i,j,k} + U_{i-1,j,k-1}}{\Delta_z^2} = 1.\end{aligned}$$

(b) Explain how you obtained the scheme.

Solution: A justification can be deduced from the part on finite volume methods in the course notes, considering the specified PDE, domain, boundary conditions, and the vertices $(i\Delta_x, j\Delta_y, k\Delta_z) \in \Omega$ for $i = 0, \ldots, J_x$, $j = 0, \ldots, J_y$, $k = 0, \ldots, J_z$. Observe that with these vertices, the Voronoi cells are axis-aligned rectangular parallelepipeds: $V_{i,j,k} = \{(x, y, z) \in \Omega : (i - \frac{1}{2})\Delta_x \leq x \leq (i+\frac{1}{2})\Delta_x, (j-\frac{1}{2})\Delta_y \leq y \leq (j+\frac{1}{2})\Delta_y, (k-\frac{1}{2})\Delta_z \leq z \leq (k+\frac{1}{2})\Delta_z\}$ for $i = 0, \ldots, J_x$, $j = 0, \ldots, J_y$, $k = 0, \ldots, J_z$. In other words, $V_{i,j,k}$ is the axis-aligned rectangular parallelepiped centered at $(i\Delta_x, j\Delta_y, k\Delta_z)$ of size $\Delta_x \times \Delta_y \times \Delta_z$ and restricted to Ω, this restriction only affecting the cells of the vertices that belong to the boundary $\partial\Omega$. Parallelepipeds have six faces, and each of the six terms on the left-hand side of the scheme equation given in answer 2(a) corresponds to the contribution of the boundary part between cell $V_{i,j,k}$ and one of the six neighboring cells.

3. You are tasked to implement a matrix-matrix-transpose product in the context of transformer inference. One key component of this model is the attention head where, given a matrix of queries Q and a matrix of keys K and you need to compute the dot products of all pairs of rows. This amounts to computing the entries (the so-called *attention scores*) of the following R matrix:

$$R = QK^\top.$$

You launch 3 AI coding agents to solve the same task of computing this product and they produce 3 different solutions:

IJK solution:

```
void ijk(const float *Q, const float *K, float *R, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            R[i * n + j] = 0.0f;
            for (int k = 0; k < n; k++)
                R[i * n + j] += Q[i * n + k] * K[j * n + k];
        }
    }
}
```

IKJ solution:

```
void ikj(const float *Q, const float *K, float *R, int n) {
    memset(R, 0, (size_t) (n * n * sizeof(float)));
    for (int i = 0; i < n; i++)
        for (int k = 0; k < n; k++) {
            for (int j = 0; j < n; j++)
                R[i * n + j] += Q[i * n + k] * K[j * n + k];
        }
}
```

Tiling solution[1]:

```
void block_tile(const float *Q, const float *K, float *R, int n, int b) {
    for (int bi = 0; bi < n; bi += b)
        for (int bj = 0; bj < n; bj += b) {
            for (int i = bi; i < bi + b; i++)
                for (int j = bj; j < bj + b; j++)
                    R[i * n + j] = 0.0f;
            for (int bk = 0; bk < n; bk += b)
                for (int i = bi; i < bi + b; i++)
                    for (int j = bj; j < bj + b; j++)
                        for (int k = bk; k < bk + b; k++)
                            R[i * n + j] += Q[i * n + k] * K[j * n + k];
        }
}
```

We will assume that

[1] To simplify, we assume that n is a multiple of b.

1. the three matrices are square $Q, K, R \in \mathbb{R}^{n \times n}$,
2. the `float` entries are represented as 32-bit floating point numbers,
3. the addresses of the first entries of Q, K and R are aligned with the start of a cache line,
4. the matrices are represented in a row-major format, which means that two entries of the same row and adjacent columns are adjacent in memory,
5. the dimension n is so large that even a single row of Q, K or R does not fit in the L1 cache.

(a) You first benchmark the IJK and IKJ solutions and notice that one is significantly faster than the other one. Which one do you think is faster[2] ? Explain why.

Solution: IJK is faster. Because we compute $R = QK^\top$, the inner sum is $R_{ij} = \sum_k Q_{ik} K_{jk}$, i.e. both matrices are walked along their *rows*.

- In **IJK** the innermost loop varies k with i, j fixed and accesses `Q[i*n+k]` and `K[j*n+k]`: both are unit-stride (sequential) reads down a row, perfectly cache- and prefetcher-friendly.
- In **IKJ** the innermost loop varies j with i, k fixed and accesses `K[j*n+k]`: j jumps by n floats between iterations, so we walk a *column* of K with stride n. Every read lands in a different cache line ⇒ essentially one cache miss per multiplication.

(Note: this is the opposite of the usual $C = AB$ rule, where IKJ wins. The transpose on K flips which order is row-sequential.)

(b) Given that the size of the L1 cache is 64 KiB (so 2^{16} bytes), what value of b should you take for the tiling solution ? Explain why.

Solution: We want the three working tiles (`BQ`, `BK`, `BR`) — each $b \times b$ of 4-byte floats — to live simultaneously in L1:

$$3\,b^2 \cdot 4 \;\le\; 2^{16} \iff b^2 \le \tfrac{2^{16}}{12} \approx 5461 \iff b \le 73.$$

A power-of-two choice that comfortably fits is $\boxed{b = 64}$ (uses $3 \cdot 64^2 \cdot 4 = 48\,\text{KiB}$ and aligns nicely with the 64-byte cache line, which holds 16 floats). Choosing b too large ⇒ tiles thrash L1; too small ⇒ tile overhead dominates and you get less reuse per byte loaded.

[2] Note that this is not the classical matrix-matrix product, since the left-hand side is multiplying by the **transpose** of the right-hand side.

(c) Given a cache line size of 64 bytes, how many cache lines will the L1 cache need to fetch[3] for each of the 3 solutions ?

Solution: A cache line holds $64/4 = 16$ floats. Recall that n is so large that even a single row does not fit in L1.

- **IJK.** For each (i, j) the inner k-loop reads row i of Q and row j of K sequentially ($n/16$ lines each); the row of R is written ($n/16$ lines per i). Since a single row no longer fits in L1, row i of Q is *evicted* before the next j iteration could reuse it, so it must be reloaded for every j (no reuse across j).

$$\underbrace{n^2 \cdot \tfrac{n}{16}}_{Q} + \underbrace{n^2 \cdot \tfrac{n}{16}}_{K} + \underbrace{n \cdot \tfrac{n}{16}}_{R} \;=\; \tfrac{n^3}{8} + \tfrac{n^2}{16} \;\approx\; \tfrac{n^3}{8}.$$

- **IKJ.** Q: $n^2/16$ lines ($Q[i{\cdot}n{+}k]$ is read sequentially in k, one line shared by 16 consecutive k). K: the inner j-loop strides by n, so *each* read of $K[j{\cdot}n{+}k]$ touches a fresh cache line: n lines per inner loop $\times\, n^2$ outer (i, k) pairs $= n^3$ lines. R: the inner j-loop sweeps the whole row i of R ($n/16$ lines); since a row no longer fits, it cannot stay cached across the k-loop and is reloaded for every (i, k) pair: $n^3/16$ lines.

$$\tfrac{n^2}{16} + n^3 + \tfrac{n^3}{16} \;\approx\; n^3 \quad (8\times \text{ worse than IJK}).$$

- **Tiling.** Each $b \times b$ tile is $b^2/16$ lines. With the loop order (b_i, b_j, b_k), the R-tile at (b_i, b_j) is reused across all n/b values of b_k; the Q- and K-tiles are reloaded for every (b_i, b_j, b_k).

$$\underbrace{2 \cdot \tfrac{n^3}{b^3} \cdot \tfrac{b^2}{16}}_{Q \text{ and } K} + \underbrace{\tfrac{n^2}{b^2} \cdot \tfrac{b^2}{16}}_{R} \;=\; \tfrac{n^3}{8b} + \tfrac{n^2}{16}.$$

  With $b = 64$: $\approx n^3/512$, i.e. $64\times$ better than IJK.

[3] from the L2/L3 or main memory. To simplify, you can ignore the L2 and L3 caches for this question.

(d) You now port the tiling solution to a GPU using OpenCL. Instead of tiling for the L1 cache, you tile for the on-chip *shared memory*: each work-group computes one $b \times b$ tile of R with $b \times b$ work-items that cooperatively stage one tile of Q and one tile of K into a buffer shared by the whole group, synchronize, and then accumulate the product out of that buffer. Fill in the missing lines in the following kernel.

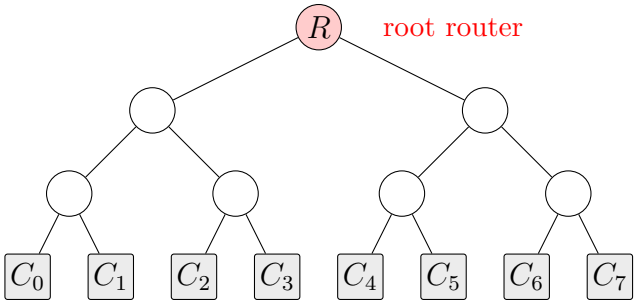
```
#define b 32

__kernel void tiled_qkt(__global const float *Q,
                        __global const float *K,
                        __global float *R,
                        const int n) {
    __local float BQ[b][b];
    __local float BK[b][b];

    const int li = get_local_id(1);
    const int lj = get_local_id(0);
    const int i  = get_global_id(1);
    const int j  = get_global_id(0);
    const int bj = get_group_id(0) * b;

    float acc = 0.0f;

    for (int bk = 0; bk < n; bk += b) {
        BQ[li][lj] = Q[i * n + (bk + lj)];
        BK[li][lj] = K[(bj + li) * n + (bk + lj)];
```

Solution:

```

        barrier(CLK_LOCAL_MEM_FENCE);

        for (int k = 0; k < b; k++) {
            acc += BQ[li][k] * BK[lj][k];
        }

        barrier(CLK_LOCAL_MEM_FENCE);
```

```
    }

    R[i * n + j] = acc;
}
```

4. You now train your transformer on a cluster. The cluster's interconnect is wired as a *binary tree*: the 8 **leaves** are the compute nodes $C_0, \ldots, C_7$ and every **internal node** is a router. Each router has at 4 available ports. Each non-root router uses one port towards its parent and one towards each of its two children, leaving one unused. We assume that all links and ports have the same bandwidth.



   (a) Computing the attention scores $R = QK^\top$ in a distributed way requires, at one point, a *transpose* of a matrix that is spread across the 8 compute nodes: each node initially holds a block of rows and, after the transpose, must hold a block of columns, so every node sends a piece of its data to every other node. You time this exchange and it is far slower than the individual link speeds would lead you to expect. Analyse how this collective behaves on the tree interconnect: what limits its running time, and how does that limit change if the cluster keeps growing as a binary tree? Justify your answer.

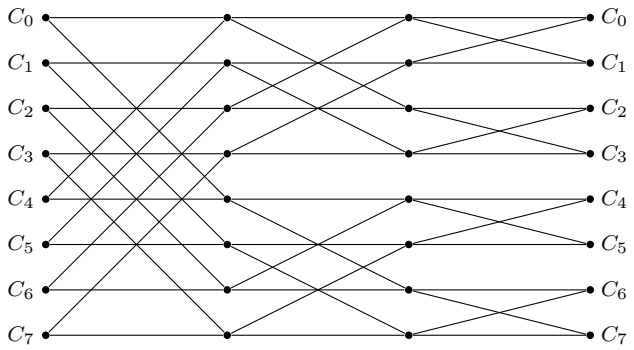
Solution: The root router R is the bottleneck. In a binary tree the only path between a left-half leaf and a right-half leaf goes up to the root and back down: every left↔right message must traverse R and its two links. So the whole all-to-all between the halves is funnelled through a single router.

The *bisection bandwidth* of the tree is just the capacity of those root links (a cut separating the left half from the right half severs only the root's two links), i.e. $O(1)$ regardless of N. The left/right exchange moves $\Theta(N^2)$ blocks but only $O(1)$ can cross per unit time, so the time grows with N. Adding compute nodes makes the problem *worse*: more leaves means more traffic squeezed through the same root, while the root's throughput stays fixed. The bottleneck is the topology, not the number of nodes.

(b) The company buys 5 **more routers** (same 4-port routers), so you now have $7 + 5 = 12$ routers in total. Explain how to use these 5 new routers to improve the running time of the transpose operation.

Solution: This is *exactly* the number needed to rewire the cluster as a *butterfly* network connecting the 8 compute nodes. A butterfly on $N = 2^3 = 8$ terminals has $\log_2 N = 3$ *stages*, each made of $N/2 = 4$ radix-2 switches (2 inputs + 2 outputs = 4 ports — exactly our routers). That is $3 \times 4 = 12$ switches, vs. the $N - 1 = 7$ routers of the tree, hence the $12 - 7 = 5$ extra ones.

Placement. Arrange the 12 routers as 3 columns of 4. Each compute node feeds into a first-stage switch; between consecutive stages, pair the rows that differ in one address bit — stride 4 at stage 1, stride 2 at stage 2, stride 1 at stage 3 (the perfect-shuffle / FFT wiring). Each pair forms one 2×2 switch, so every router uses all 4 ports.



Why it removes the bottleneck. The butterfly has bisection width $N/2 = 4$: a left/right cut now crosses 4 links instead of the tree's 2 at the single root, and there is no single shared router any path is forced through. The half-to-half all-to-all can use $N/2$ disjoint paths in parallel, so the bandwidth scales with the machine instead of being pinned to one root router (still with a uniform $\log_2 N = 3$ hops between any two nodes).

5. While inspecting the cluster, you discover that one of your colleagues Alice has been mining bitcoin on the compute nodes whenever no user job is running on them, i.e. he keeps the otherwise-idle nodes busy at 100% utilization.

   Bob complains that this is inflating the electricity bill. Alice replies:

   > *"The nodes have to stay powered on anyway as they must be ready the instant a user asks for compute, so we are not allowed to switch them off. A node that is on consumes its power regardless of what it does, so it draws the same whether it sits idle or mines bitcoin. I might as well put those idle cycles to use."*

   We grant Alice her premise: the nodes **must stay on and ready**, so the question is *not* whether they could be switched off. Taking that as fixed, is the rest of Alice reasoning "that an **on-but-idle** node draws the same power as an **on-and-fully-loaded** one" correct? Justify quantitatively.

Solution: No — her reasoning is wrong; "on" does not mean "drawing full power". The power of a CPU/GPU is not a constant; it splits into

$$P = \underbrace{P_{\text{static}}}_{\text{leakage}} + \underbrace{\alpha C V^2 f}_{\text{dynamic}},$$

where α is the *activity factor* (fraction of transistors actually switching). On an idle node $\alpha \approx 0$, so the dynamic term — which dominates the total under load — essentially vanishes. On top of that, an idle node is not held at its maximum operating point:

- **DVFS** lowers the clock f and voltage V when there is no work; since dynamic power $\propto V^2 f$, dropping both cuts it super-linearly.
- **Idle / sleep states** (CPU C-states, GPU power states) clock-gate and power-gate unused units, further shrinking P_{static}.

Concretely, a server CPU/GPU idles at a small fraction of its TDP and rises to (near) its full TDP only at 100% utilization — often a 3× to 10× swing between idle and a sustained mining load. Bitcoin mining deliberately pins every core at maximum, i.e. it drives α, f and V all to their peak, so it adds (close to) the *entire* dynamic power budget of every node it touches — plus extra cooling power to remove that heat.

So even though the nodes legitimately stay powered on, "idle" and "mining" are very far from the same electricity draw: the mining is genuinely adding to the bill, and the "it's the same anyway" argument is false.