Exercise Sheet 3

CPU & GPU Architecture

1. Understanding Latency and Throughput

Consider the following loop that multiplies two vectors elementwise and accumulates the result using IEEE-754 floating-point numbers. Assume this code has been translated to efficient machine code (no interpreter overhead):

```
accum = 0.0  # ignore the time needed to initialize 'accum'
for i in range(n):
    accum += a[i] * b[i]
```

Assume it runs on a superscalar CPU with the characteristics below. As discussed in Lecture 5, a *superscalar* core issues multiple independent operations per cycle by dispatching them to parallel functional units, and modern out-of-order schedulers work to keep those units busy whenever independent work exists. The *latency* is the number of cycles between issuing an operation and the earliest cycle its result can be consumed. The *throughput* is the maximum number of independent operations of that kind the core can start in a single cycle.

Operation	Latency (cycles)	Throughput (operations/cycle)
Load (cache miss)	200	2.0
Floating-point multiply	5	1.0
Floating-point addition	4	2.0

Assume for simplicity that every load misses in cache and returns after 200 cycles. The memory subsystem can keep up to 50 outstanding misses in flight. For part (d), assume an ideal out-of-order processor: an instruction executes as soon as its operands are ready, and instruction loading/decoding is instant.

(a) Which of the statements below are true, and why?

- While waiting for the current multiply-and-accumulate to finish, the processor can issue the loads of `a[i+1]` and `b[i+1]` early.
- The 200-cycle cache miss latency is the main bottleneck, rather than the serial dependence of `accum` on previous iterations.
- Using a fused multiply-add removes the dependency between iterations.

(b) Using the data above, estimate the steady-state average number of cycles per processed element. State the bottleneck you assume dominates and how you combine latency/throughput terms to obtain your number.

(c) If the loop processes 1,000,000 elements, roughly how long does it take on a 4 GHz CPU according to your result in part (b)?

(d) If the loop runs for only two iterations, use the table above to give the precise number of cycles required.

Solution:

(a) **True**: The loads of `a[i+1]` and `b[i+1]` are independent of the current multiply-and-accumulate, so the processor can start them early while waiting for memory.
True: The 200-cycle cache miss latency is the main bottleneck, rather than the serial dependence of `accum` on previous iterations.
False: A fused multiply-add still accumulates into `accum`, so the dependency between iterations remains.

(b) Each iteration issues two 200-cycle loads. With up to 50 misses in flight, the memory system completes one load every 4 cycles ($50/200 = 0.25$ loads per cycle, or equivalently, one load every $1/0.25 = 4$ cycles). That is 4 cycles per load or 8 cycles per element. The multiply (1 per cycle) and addition (2 per cycle) overlap with that delay, so memory bandwidth dominates: about 8 cycles per element (2 ns at 4 GHz).

(c) Processing 1,000,000 elements at 8 cycles per element takes 8×10^6 cycles. At 4 GHz, this is $\frac{8\times10^6}{4\times10^9} = 0.002$ seconds $= 2$ ms.

(d) With an ideal out-of-order processor, we can trace the execution precisely:

- Cycle 0: Issue loads for `a[0]` and `b[0]`
- Cycle 1: Issue loads for `a[1]` and `b[1]` (2 loads/cycle throughput)
- Cycle 200: `a[0]` and `b[0]` ready; issue multiply `a[0] * b[0]`
- Cycle 201: `a[1]` and `b[1]` ready; issue multiply `a[1] * b[1]`
- Cycle 205: First multiply result ready; issue addition `accum + (a[0]*b[0])`
- Cycle 206: Second multiply result ready (waiting for `accum`)
- Cycle 209: First addition completes; `accum` updated; issue `accum + (a[1]*b[1])`
- Cycle 213: Second addition completes

Total: 213 cycles. The serial dependency through `accum` forces the second addition to wait for the first, adding 4 cycles beyond when its multiply operand is ready.

2. Memory Access Patterns

Consider the following two implementations that sum all elements of a matrix stored in row-major order (C/NumPy convention where rows are contiguous in memory):

```
# Version A: Row-major traversal
total_a = 0.0
for i in range(n):
    for j in range(m):
        total_a += A[i, j]

# Version B: Column-major traversal
total_b = 0.0
for j in range(m):
    for i in range(n):
        total_b += A[i, j]
```

(i) Which version will generally perform better on a modern CPU? Explain your reasoning in terms of cache behavior.

(ii) Assume the matrix is 1000×1000 with 8-byte floats, the cache line size is 64 bytes, and we have a 32 KB L1 cache. Estimate roughly how many cache lines must be fetched from main memory for each version. *Hint*: First compute how many matrix elements fit in one cache line, then reason about how many cache lines must be loaded for each access pattern.

Solution:

(i) **Version A performs better.** In row-major storage, consecutive elements in a row (varying `j`) are adjacent in memory. Version A accesses memory sequentially, exhibiting excellent spatial locality. When the CPU loads a cache line (typically 64 bytes), it gets multiple consecutive elements, and subsequent accesses hit in cache. In contrast, Version B still fetches a full 64-byte cache line for each row but uses only one of the eight doubles before moving on.

(ii) Each cache line holds $64/8 = 8$ doubles.

Version A: Accesses 10^6 elements sequentially. Approximately $10^6/8 = 125{,}000$ cache lines fetched (one miss per cache line, assuming cold start).

Version B: Accesses elements with stride 1000 (jumping between rows). Each cache line fetched contains eight consecutive doubles along a row, but the loop only uses the first one before jumping to the next row. The inner loop touches all 1000 rows before returning to the second element of the first row. Since $1000 \times 8 = 8{,}000$ bytes exceeds the 32 KB L1 cache capacity when considering other cache pollution, by the time the code returns to a row, that cache line has been evicted. The result: roughly 10^6 cache lines fetched (nearly one per element) and about **8× slower** performance.

3. CPU vs. GPU: Choosing the Right Architecture

For each of the following computational tasks, determine whether it is better suited for a CPU or a GPU, and briefly justify your answer (1-2 sentences). Assume modern consumer hardware: a multi-core CPU (e.g., 8-16 cores at 3-4 GHz) and a typical GPU (e.g., thousands of cores at 1-2 GHz with dedicated hardware for fast matrix multiplications).

(a) Multiplying two dense 5000×5000 matrices.

(b) Computing the Fibonacci sequence: $F_n = F_{n-1} + F_{n-2}$ for $n = 1, 2, \ldots, 10^6$.

(c) Traversing a large irregular graph (e.g., a social network with 100 million nodes) using depth-first search to find connected components.

(d) Applying the function $f(x) = \sin(x) \cdot e^{-x^2}$ independently to each of 100 million input values.

(e) Sorting an array of 10,000 integers using quicksort.

(f) Rendering 8 million pixels for a video game frame, where each pixel requires running a shader program with 200 arithmetic operations.

Solution:

(a) GPU: Matrix multiplication is highly parallel with excellent arithmetic intensity ($O(n^3)$ operations on $O(n^2)$ data). GPUs excel at this thanks to many parallel cores and dedicated hardware matrix-multiplier units that keep arithmetic busy while hiding latency through massive parallelism.

(b) CPU: This is fundamentally sequential—each term depends on the previous two when evaluated via the recurrence. GPUs cannot hide latency when there is no parallel work available. CPUs are optimized for low-latency sequential operations.

(c) CPU: Graph traversal has irregular, unpredictable memory access patterns (pointer chasing) and is inherently sequential (DFS). This causes poor cache behavior and branch mispredictions on GPUs. CPUs handle irregular workloads better with large caches and branch prediction.

(d) GPU: Embarrassingly parallel with 100M independent operations. Despite transcendental functions having high latency, the GPU hides latency by switching between millions of threads. The massive parallelism overwhelms any per-operation overhead.

(e) CPU: Quicksort on small data fits in CPU cache and has irregular branching patterns (partitioning depends on pivot comparisons). The data is too small to amortize GPU transfer overhead, and the algorithm's recursive, branchy nature suits CPUs better.

(f) GPU: This is exactly what GPUs were designed for. Millions of independent pixels, each requiring identical operations (same shader), with sufficient arithmetic intensity. Perfect for GPU's SIMD execution model and latency hiding.