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.

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.

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.