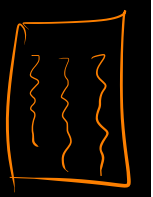
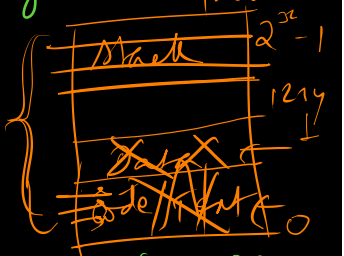
Sai

Alternate Approaches for hiding Memory Latency

I. Multithreading



Thread : Single Stream of control in the flow of a program




```
for( i = 0 ; i < N ; i++)
    for( j = 0 ; j < N ; j++)
        c[i][j] = dot_product(get_row(a, i),
                              get_col(b, j));
```

n^2 dot_product (..) → Each can be executed independently

↓

Can be assigned to different Compute abstractions (Threads)

Main thread can spawn n^2 or $n^2 - 1$ threads for each to take one iteration ←

↓

One process, multiple threads → Program Complete

Main thread to wait until all threads have finished

↓

What if there were n^2 cores?

n^2 speedup ←

Speedup? ← What if there was only one core?

```
for(i = 0; i < M; i++)
    for(j = 0; j < N; j++)
        c[i][j] = create_thread(dot_product(
                  get_row(a, i), get_col(b, j)));
```

(Assume only One core)

Thread 0 (on core): c[0][0] → requests for a[0][0] and b[0][0] (in the 0th cycle)

↓ Switches Context

Thread 1 (on core): c[1][1] → requests for a[1][0] and b[0][1] (in the 1st cycle)

⋮ ⋮

For memory latency of 100 ns, when will the data arrive?

After 100 ns, a[0][0] & b[0][0] arrive
→ Thread 0 can start computing

After 100 ns + 1 CPU cycle, a[0][1] & b[1][0] arrive
→ Thread 1 can start computing

(not considering bus width or bus cycle)

⇒ Hides Memory latency provided,

→ there is enough Concurrency to keep the processor busy

→ Program explicitly specifies it

→ Fast Context Switching of threads

II. Prefetching

```
for(i=0; i<N; i++)
    c[i] = a[i] + b[i];
```

```
load  R1, a[0]
add   R1, b[0]
store R1, c[0]
:
```

(Load just before use)

→

```
load  R1, a[0]
load  R2, b[0]
load  R3, a[1]
load  R4, b[1]
:
add   R1, R2
store R1, c[0]
:
```

(Load much ahead of use)

→ Even if there is a cache miss, data is likely to arrive by the time it is used.

→ Fresh load needed if a data item is overwritten between load and use (but no worse than the original)

→ Effectively, same as multithreading, in identifying no resource or data dependencies

→ Done by Compiler optimizations

Issues in Multithreading & Prefetching

I. 1 GHz CPU, 4-word Cache line, 1 cycle access to Cache, 100 ns latency to DRAM

Assume a Computation:

- CPU requests 1 word every cycle
- For 1 KB Cache, Cache Hit Ratio = 25%.
- For 32 KB Cache, Cache Hit Ratio = 90%.

Two cases: a) Single thread, 32 KB Cache

b) 32 threads, 1 KB Cache to each

Case a) 1 word every cycle and 90% Cache Hit Ratio

⇒ 1 in 10 requests hits memory

⇒ On an average, 1 word in 10 ns from DRAM

⇒ Memory bandwidth requirement = 400 MB/s

Case b) 1 word every cycle and 25% Cache Hit Ratio

⇒ 3 in 4 requests hits memory

⇒ On an average, 3 words in 4 ns from DRAM

⇒ Memory bandwidth requirement = 3 GB/s

// Bandwidth requirements of multithreaded program may increase significantly due to smaller cache residency of each thread.

// Tend to become bandwidth bound instead of latency bound.

// Prefetching does not worsen than the original, but if needed to fetch again, it means, fetching data item twice, resulting in doubling memory bandwidth requirement.

II. Additional hardware needed to effectively utilize multithreading or prefetching.
(Need large register files and caches)