CUDA Programming

Recap

- Check how the localities are in the following matrix multiplication programs (on CPU).

```
for (i = 0; i < M; ++i)
        for (j = 0; j < N; ++j)
                for (k = 0; k < P; ++k)
                        C[i][j] += A[i][k] * B[k][j];
```

```
for (i = 0; i < M; ++i)
        for (k = 0; k < P; ++k)
                for (j = 0; j < N; ++j)
                        C[i][j] += A[i][k] * B[k][j];
```

Times taken for (M, N, P) = (1024, 1024, 1024) are 9.5 seconds and 4.7 seconds.

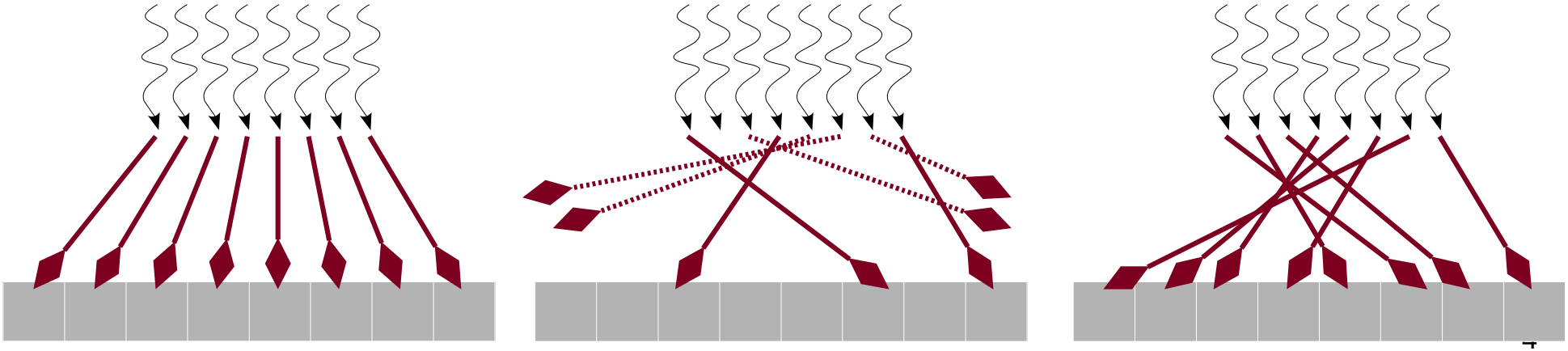
What happens on a GPU?

Memory Coalescing

- If *warp threads* access words from the same block of 32 words, their memory requests are clubbed into one.
  - That is, the memory requests are coalesced.
- **Without coalescing**, each load / store instruction would required one memory cycle.
  - A warp would require 32 memory cycles.
  - The throughput would significantly reduce.
  - GPU would be useful only for compute-heavy kernels.

Memory Coalescing

- If *warp threads* access words from the same block of 32 words, their memory requests are clubbed into one.
  - That is, the memory requests are coalesced.
- This can be effectively achieved for regular programs (such as dense matrix operations).



Coalesced **Uncoalesced** **Coalesced**

Memory Coalescing

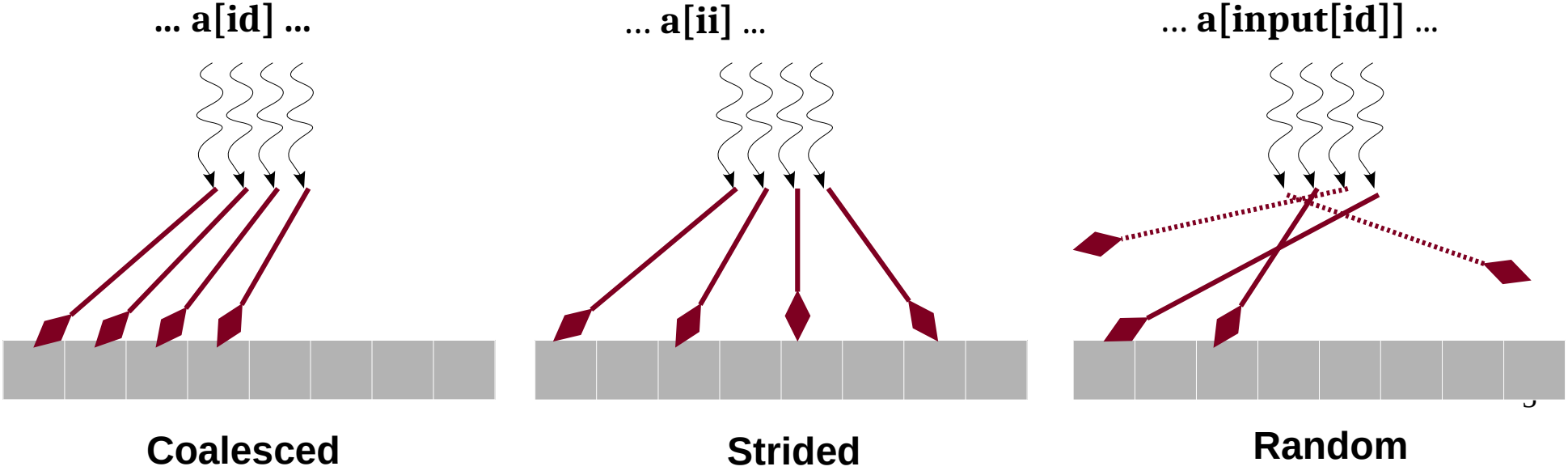
CPU

- Each thread should access consecutive elements of a chunk (strided).
- Array of Structures (AoS) has a better locality.

GPU

- A chunk should be accessed by consecutive threads (coalesced).
- Structure of Arrays (SoA) has a better performance.

```
start = id * chunksize;
end = start + chunksize;
for (ii = start; ii < end; ++ii)
```



AoS versus SoA

```
struct node {
    int a;
    double b;
    char c;
};
struct node allnodes[N];
```

```
struct node {
    int alla[N];
    double allb[N];
    char allc[N];
};
```

Expectation: When a thread accesses an attribute of a node, *it* also accesses *other attributes* of the *same node*.

Better locality (on CPU).

Expectation: When a thread accesses an attribute of a node, its *neighboring thread* accesses the *same attribute* of the *next node*.

Better coalescing (on GPU).

Classwork: Write code for the two types using cudaMemcpy.
(note that all arrays would be pointers)

AoS versus SoA

```
struct nodeAOS {
        int a;
        double b;
        char c;
} *allnodesAOS;
```

```
__global__ void dkernelaos(struct nodeAOS *allnodesAOS) {
        unsigned id = blockIdx.x * blockDim.x + threadIdx.x;
        allnodesAOS[id].a = id;
        allnodesAOS[id].b = 0.0;
        allnodesAOS[id].c = 'c';
}
```

```
struct nodeSOA {
        int *a;
        double *b;
        char *c;
} allnodesSOA;
```

```
__global__ void dkernelsoa(int *a, double *b, char *c) {
        unsigned id = blockIdx.x * blockDim.x + threadIdx.x;
        a[id] = id;
        b[id] = 0.0;
        c[id] = 'd';
}
```

AOS time = 61 units, SOA time = 22 units