CUDA Programming

Launch Configuration for Huge Data

```
#include <stdio.h>
#include <cuda.h>
__global__ void dkernel(unsigned *vector) {
        unsigned id = blockIdx.x * blockDim.x + threadIdx.x;
        vector[id] = id;
}
#define BLOCKSIZE       1024
int main(int nn, char *str[]) {
        unsigned N = atoi(str[1]);
        unsigned *vector, *hvector;
        cudaMalloc(&vector, N * sizeof(unsigned));
        hvector = (unsigned *)malloc(N * sizeof(unsigned));

        unsigned nblocks = ceil(N / BLOCKSIZE);
        printf("nblocks = %d\n", nblocks);

        dkernel<<<nblocks, BLOCKSIZE>>>(vector);
        cudaMemcpy(hvector, vector, N * sizeof(unsigned), cudaMemcpyDeviceToHost);
        for (unsigned ii = 0; ii < N; ++ii) {
                printf("%4d ", hvector[ii]);
        }
        return 0;
}
```

Access out-of-bounds

Find two issues with this code.

Needs floating-point division

Launch Configuration for Large Size

```
#include <stdio.h>
#include <cuda.h>
__global__ void dkernel(unsigned *vector, unsigned vectorsize) {
	unsigned id = blockIdx.x * blockDim.x + threadIdx.x;
	if (id < vectorsize) vector[id] = id;
}
#define BLOCKSIZE	1024
int main(int nn, char *str[]) {
	unsigned N = atoi(str[1]);
	unsigned *vector, *hvector;
	cudaMalloc(&vector, N * sizeof(unsigned));
	hvector = (unsigned *)malloc(N * sizeof(unsigned));

	unsigned nblocks = ceil((float)N / BLOCKSIZE);
	printf("nblocks = %d\n", nblocks);

	dkernel<<<nblocks, BLOCKSIZE>>>(vector, N);
	cudaMemcpy(hvector, vector, N * sizeof(unsigned), cudaMemcpyDeviceToHost);
	for (unsigned ii = 0; ii < N; ++ii) {
		printf("%4d ", hvector[ii]);
	}
	return 0;
}
```

Matrix Squaring

```
void squarecpu(unsigned *matrix, unsigned *result,
                    unsigned matrixsize /* = 64*/) {
  for (unsigned ii = 0; ii < matrixsize; ++ii) {
  for (unsigned jj = 0; jj < matrixsize; ++jj) {

    for (unsigned kk = 0; kk < matrixsize; ++kk) {
      result[ii * matrixsize + jj] +=
        matrix[ii * matrixsize + kk] * matrix[kk * matrixsize + jj];
    }
  }
  }
}
```

CPU time = 1.527 ms

Matrix Squaring (version 1)

```
square<<<1, N>>>(matrix, result, N); // N = 64
```

```
__global__ void square(unsigned *matrix,
                       unsigned *result,
                       unsigned matrixsize) {
    unsigned id = blockIdx.x * blockDim.x + threadIdx.x;
    for (unsigned jj = 0; jj < matrixsize; ++jj) {
        for (unsigned kk = 0; kk < matrixsize; ++kk) {
            result[id * matrixsize + jj] +=
                        matrix[id * matrixsize + kk] *
                        matrix[kk * matrixsize + jj];
}   }   }
```

CPU time = 1.527 ms, GPU v1 time = 6.391 ms

Matrix Squaring (version 2)

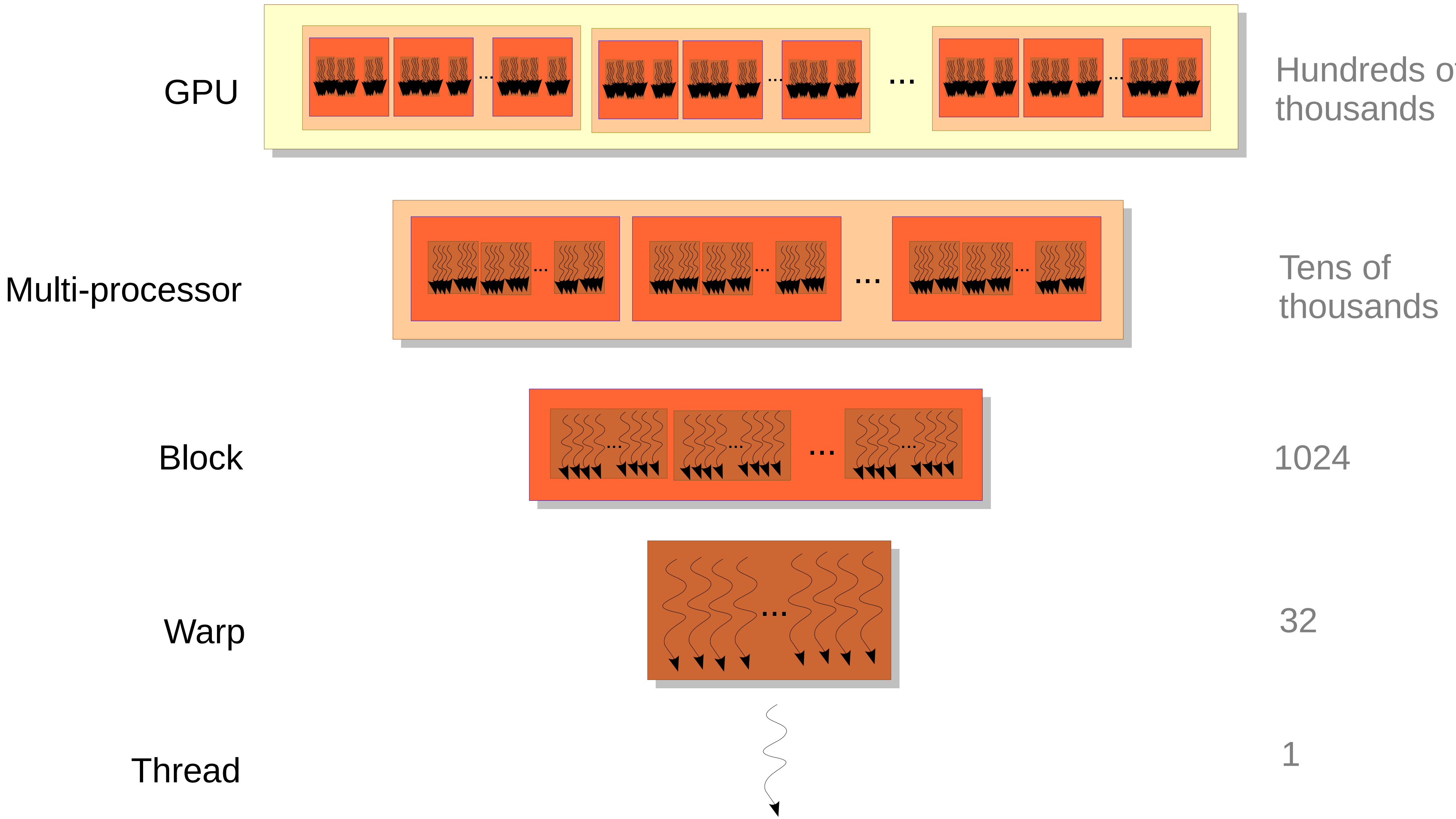
```
square<<<N, N>>>(matrix, result, N);    // N = 64
```

```
__global__ void square(unsigned *matrix,
                       unsigned *result,
                       unsigned matrixsize) {
    unsigned id = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned ii = id / matrixsize;
    unsigned jj = id % matrixsize;
    for (unsigned kk = 0; kk < matrixsize; ++kk) {
        result[ii * matrixsize + jj] += matrix[ii * matrixsize + kk] *
                                        matrix[kk * matrixsize + jj];
}   }
```

Homework: What if you interchange ii and jj?

CPU time = 1.527 ms, GPU v1 time = 6.391 ms,
GPU v2 time = 0.1 ms

GPU Computation Hierarchy



Warp

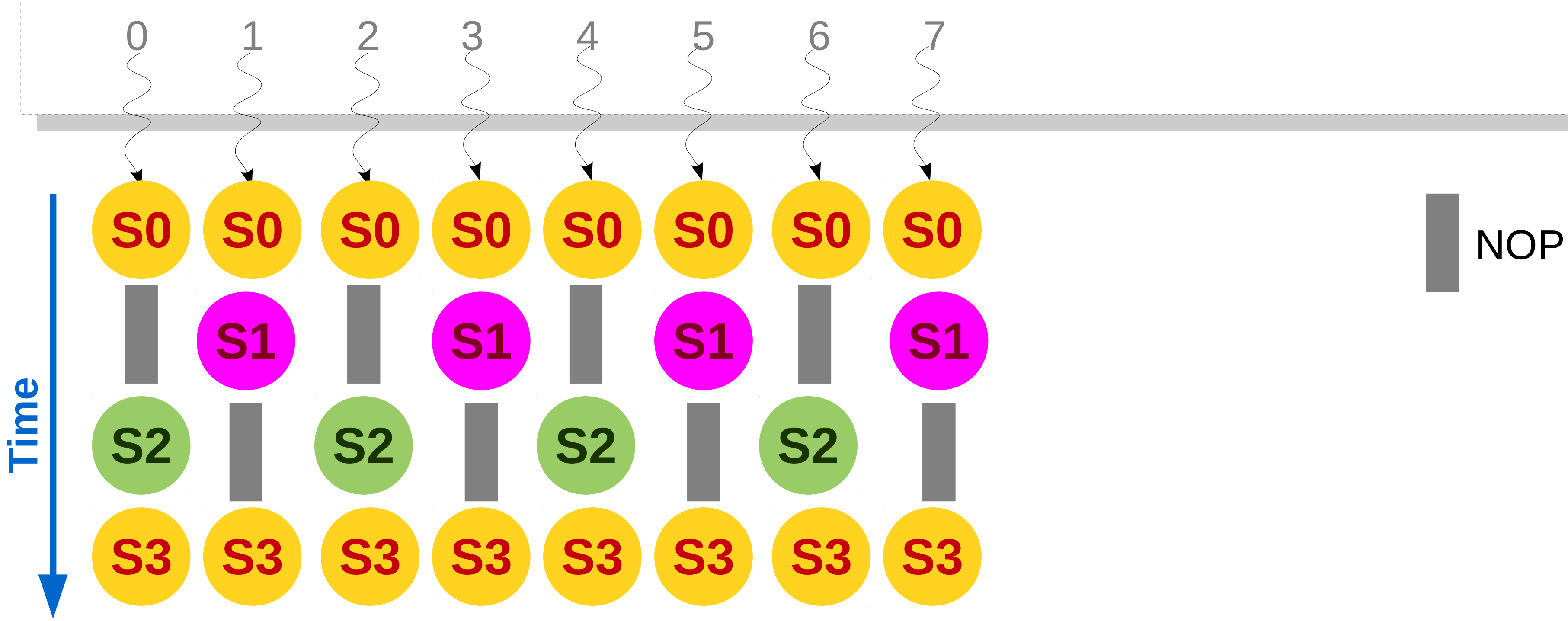
- A set of consecutive threads (currently 32) that execute in SIMD fashion.
- SIMD == Single Instruction Multiple Data
- Warp-threads are fully synchronized. There is an implicit barrier after each step / instruction.
- Memory coalescing is closely related to warps.

Takeaway

It is a misconception that all threads in a GPU execute in lock-step. Lock-step execution is true for threads only within a warp.

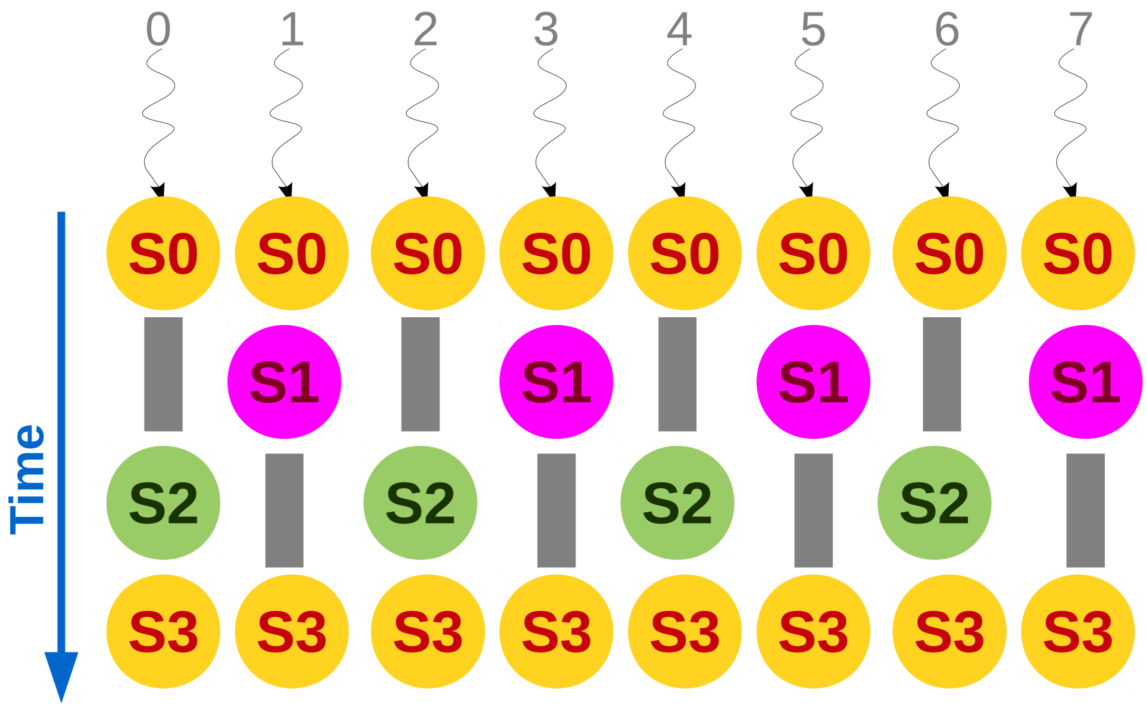
Warp with Conditions

```
__global__ void dkernel(unsigned *vector, unsigned vectorsize) {
    unsigned id = blockIdx.x * blockDim.x + threadIdx.x;   S0
    if (id % 2) vector[id] = id;                S1
    else vector[id] = vectorsize * vectorsize;      S2
     vector[id]++;   S3
}
```



Warp with Conditions

- When different warp-threads execute different instructions, threads are said to diverge.
- Hardware executes threads satisfying same condition together, ensuring that other threads execute a no-op.
- This adds sequentiality to the execution.
- This problem is termed as **thread-divergence**.



Classwork

```
__global__ void dkernel(unsigned *vector, unsigned vectorsize) {
    unsigned id = blockIdx.x * blockDim.x + threadIdx.x;
    for (unsigned ii = 0; ii < id; ++ii)
        vector[id] += ii;
}
```

Does this code diverge?

```
__global__ void dkernel(unsigned *vector, unsigned vectorsize) {
    unsigned id = blockIdx.x * blockDim.x + threadIdx.x;
    if (id % 2) vector[id] = id;
    else if (vector[id] % 2) vector[id] = id / 2;
    else vector[id] = id * 2;
}
```

vector is initialized to {0, 1, 2, 3, ...}.

Does this code diverge further?

Thread-Divergence

- Since thread-divergence makes execution sequential, conditions are evil in the kernel codes?

```
if (vectorsize < N) S1; else S2;
```

Condition but no divergence

- Then, conditions evaluating to different truth-values are evil?

```
if (id / 32) S1; else S2;
```

Different truth-values but no divergence

Takeaway

Conditions are not bad;
they evaluating to different truth-values is also not bad;
they evaluating to different truth-values for warp-threads is bad.

Classwork

- Rewrite the following program fragment to remove thread-divergence.

```
assert(x == y || x == z);
if (x == y) x = z;
else x = y;
```

```
assert(x == y || x == z);
x = y + z – x;
```