# CME213/ME339
# Lecture 8

Eric Darve
Erich Elsen
Austin Gibbons

Department of Mechanical Engineering
Institute for Computational and Mathematical Engineering
Stanford University

Spring 2012

# Technical note on timing

- CPU timers or GPU timers?
- Recall that CUDA API functions are asynchronous, that is, they return control back to the calling CPU thread prior to completing their work.
- All kernel launches are asynchronous.
- Therefore, to accurately measure elapsed time, it is necessary to synchronize the CPU thread with the GPU by calling `cudaDeviceSynchronize()` **immediately before starting and stopping the CPU timer.**

# The better way

- Using GPU timers is often a better solution.
- It is independent of the operating system.
- The accuracy is sufficient for most applications.
- The synchronizations requirements are less stringent.

```
1  cudaEvent_t start, stop;
2  float time;
3
4  cudaEventCreate(&start);
5  cudaEventCreate(&stop);
6
7  cudaEventRecord( start, 0 );
8  kernel<<<grid,threads>>> ( ... );
9  cudaEventRecord( stop, 0 );
10
11 cudaEventSynchronize( stop );
12 cudaEventElapsedTime( &time, start, stop );
13
14 cudaEventDestroy( start );
15 cudaEventDestroy( stop );
```

# Understanding the timing code

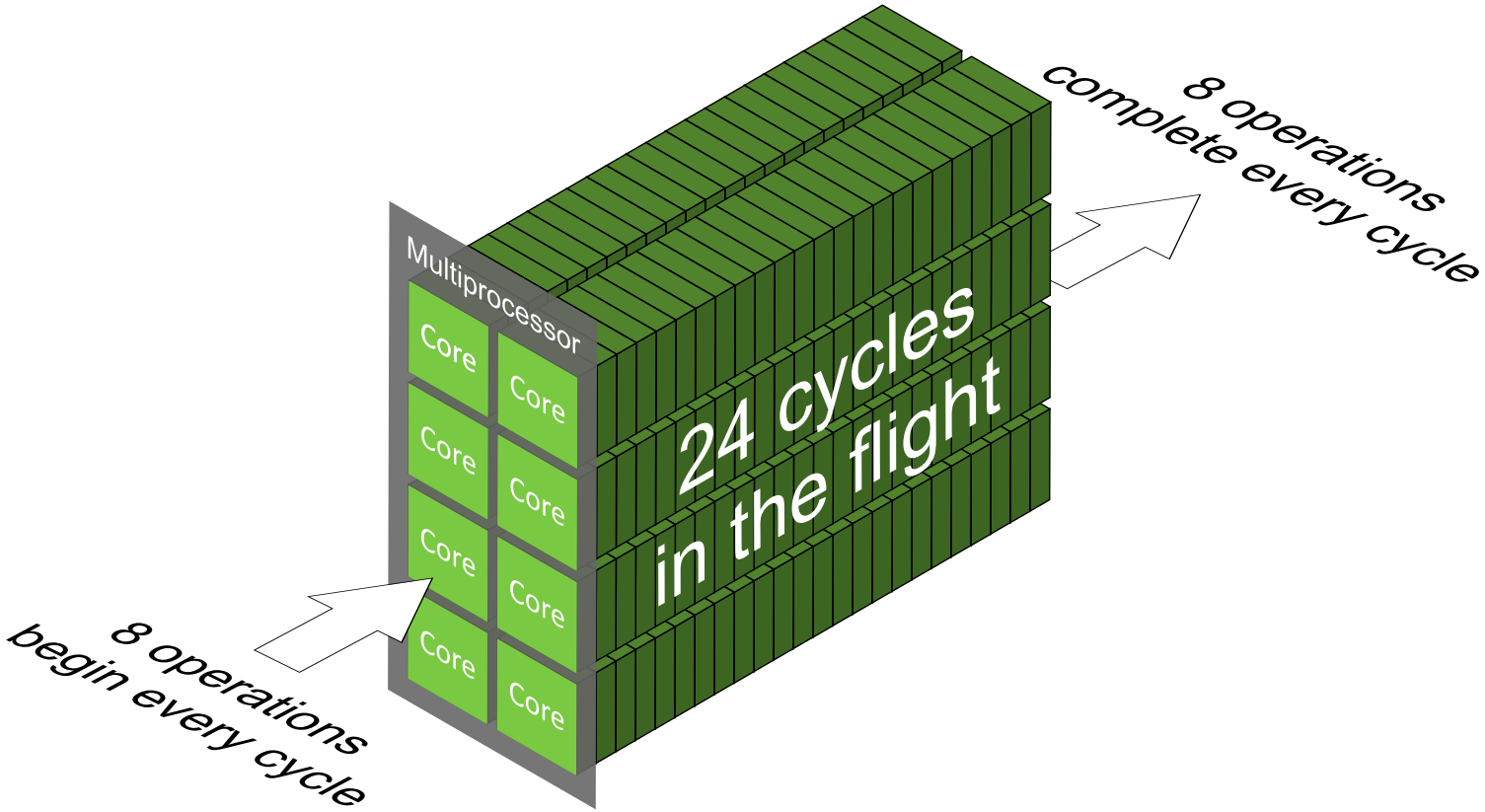
- `cudaEventRecord()` is used to place the `start` and `stop` events before and after the kernel call. Everything is still asynchronous. When this executes is unknown, but the order is fixed.
- The device records a timestamp for the event when it reaches that event during the execution.
- The `cudaEventElapsedTime()` function returns the time elapsed between the recording of the start and stop events. This requires a `cudaEventSynchronize()` since you cannot use this function until the two events have effectively been recorded.
- Value is expressed in milliseconds and has a resolution of approximately half a microsecond.

# Occupancy

Recall the fundamental assumptions to understand the execution of a CUDA program:

- Instructions are issued in order (this is specified by the compiled code)
- A thread stalls when one of the operands isn't ready
  - This happens typically when a data read from memory is not available.
  - Register dependency: this arises when an instruction uses a result stored in a register written by an instruction before it.
  - Note: a memory read instruction by itself doesn't stall execution; execution stalls only when the data is required by an instruction.
  - Latency is hidden by switching threads: the hardware moves on to the next warp if the current warp stalls.

# Little's law




Concurrency $\geq$ Latency $\times$ Throughput. You need to be able to fill up the pipeline! Concurrency = loosely defined as a set of operations that the hardware may execute simultaneously.

# Latency

| | Latency | Throughput | Parallelism |
|---|---|---|---|
| Arithmetic | 24 cycles | 32 ops/SM/cycle | 24 warps<br>$= 768$ ops/SM |
| Memory | 400–800 | 177 GB/sec | $< 7.14$ KB per SM |

7.14 KB is a lot. That corresponds to 58 warps requesting 128 bytes (4 bytes per thread) or 29 warps requesting 256 bytes (8 bytes per thread).

But the maximum number of warps per SM is 48! Thus you cannot achieve peak bandwidth when requesting only one 4 B word.

Memory latency can also be hidden by computation.

# Occupancy

- There is two basic ways to generate concurrency:
  1. Increase the number of threads
  2. Increase the parallel work generated by each thread
- Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps.
- Occupancy is determined by multiple factors, including the number of registers and shared memory required by each thread-block.