

# CS 179: GPU Programming

---

LECTURE 5: SYNCHRONIZATION AND ILP

# Announcement

---

**Reminder:** Instead of emailing us the solution as some assignments may still ask for, please put a zip file in your home directory on Titan, in the format:

**lab[N]\_2021\_submission.zip**

Your submission should be a single archive file (.zip) with your README file and all code.

# Last time...

---

- GPU Memory System
  - Global Memory: the slowest and largest form of memory on the GPU, shared by all grids
    - Coalesced memory access minimizes the number of cache lines read
  - Shared Memory: very fast memory, located on the SM and shared by the block
    - Setup as 32 banks that can be accessed in parallel. Each successive 32-bit words are assigned to successive banks
    - A bank conflict occurs when 2 threads in a warp access different elements in the same bank
  - Registers: fastest memory possible, located on the SM, scope is the thread
  - Local Memory: located on the Global Memory, scope is the thread
  - L1/L2/L3 Cache
  - Texture and Constant Cache

# This time:

---

- Synchronization and Deadlock
- -- Digression on Floating Point calculations
- -- Digression on seeing Compiler optimizations (<https://godbolt.org/>)
- Atomic Operations
- Instruction Dependencies
- Instruction Level Parallelism (ILP)
- Warp Scheduler
- Occupancy

# Synchronization

---

Ideal case for parallelism:

- no resources shared between threads
- no communication needed between threads

However, many algorithms that require shared resources can still be accelerated by massive parallelism of the GPU.

- Need to avoid **Deadlock!** Processes can depend on each other and get stuck!
- Each member of a group can wait for another member, including itself, to take action.
  - <https://en.wikipedia.org/wiki/Deadlock>

# Synchronization

---

**Synchronization** is a process by which multiple threads must indirectly communicate with each other in order to make sure they do not clash with each other

Example of synchronization issue:

```
int x = 1;
```

```
Thread 1: x += 1;
```

```
Thread 2: x += 1;
```

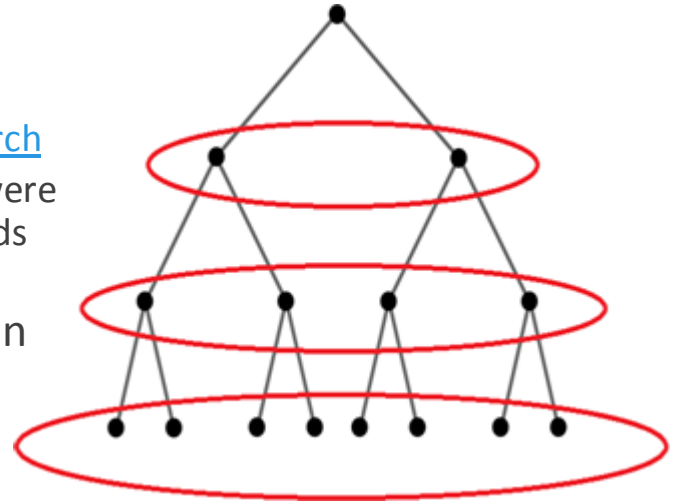
- Thread 1 reads in the value of x (which is 1) into a register
- Thread 2 reads in the value of x (which is still 1) into a register
- Both threads increment the values they read in but they both think the final value is 2
- They write x back out and the final result is 2

# Synchronization

---

Examples needing synchronization:

- Parallel BFS (Breadth First Search)
  - see [https://en.wikipedia.org/wiki/Parallel\\_breadth-first\\_search](https://en.wikipedia.org/wiki/Parallel_breadth-first_search)
  - (assuming thread touches elements in shared memory that were loaded by other threads. Can't start processing until all threads have finished loading data into shared memory.)
- Accurately summing a list of floating point numbers in parallel Many issues! See [http://ic.ease.upenn.edu/pdf/parallel\\_fpaccum\\_tc2016.pdf](http://ic.ease.upenn.edu/pdf/parallel_fpaccum_tc2016.pdf)
- Loading parallel data into a GPU's shared memory
- Dining philosophers problem –
  - [https://en.wikipedia.org/wiki/Dining\\_philosophers\\_problem](https://en.wikipedia.org/wiki/Dining_philosophers_problem)



# Digression: Accuracy Issues for Floating Point Calculations

---

Finite precision arithmetic! (Use double precision for numerical accuracy. Slower.)

Addition is NOT associative, so  $(a + b) + c \neq a + (b + c)$

For instance, there is a small enough positive number,  $\varepsilon$  such that  $1 + \varepsilon = 1$  .

Consider  $(\varepsilon + 1) - 1$  which is equal to  $(1) - 1$  which is zero.

Re-associate, and compare to  $\varepsilon + (1 - 1)$  which is  $\varepsilon + (0)$  which is  $\varepsilon$  .

The two different results are very different! Not enough bits to retain full accuracy!

Don't add large list of finite precision numbers in uncontrolled order. May as well be a random number generator! Add from smallest to largest, to preserve "bits."

Earliest C compilers assumed addition was associative, and ruined the calculations. Couldn't easily be used for numeric calculations!

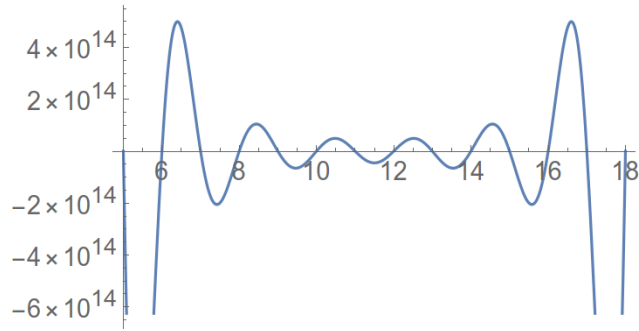


# How bad can floating point get?

```
f[x_] = Product[(x - i), {i, 1, 22}]
```

```
(-22 + x) (-21 + x) (-20 + x) (-19 + x) (-18 + x) (-17 + x) (-16 + x)  
(-15 + x) (-14 + x) (-13 + x) (-12 + x) (-11 + x) (-10 + x) (-9 + x)  
(-8 + x) (-7 + x) (-6 + x) (-5 + x) (-4 + x) (-3 + x) (-2 + x) (-1 + x)
```

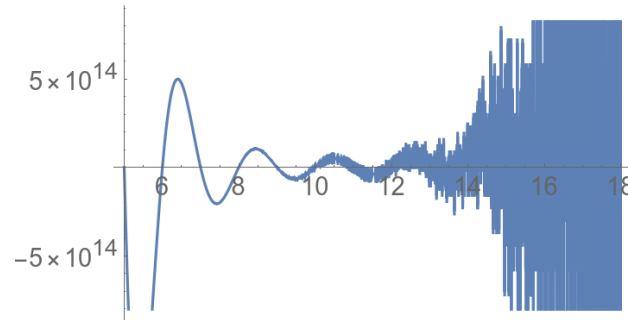
```
Plot[f[x], {x, 5, 18}]
```



Expanded Symbolically :

```
1 124 000 727 777 607 680 000 - 4 148 476 779 335 454 720 000 x +  
6 756 146 673 770 930 688 000 x2 - |... + 62 382 416 421 941 x14 -  
3 256 091 103 430 x15 + 136 717 357 942 x16 - 4 546 047 198 x17 +  
116 896 626 x18 - 2 240 315 x19 + 30 107 x20 - 253 x21 + x22
```

```
Plot[expandfExact[x], {x, 5, 18}]
```

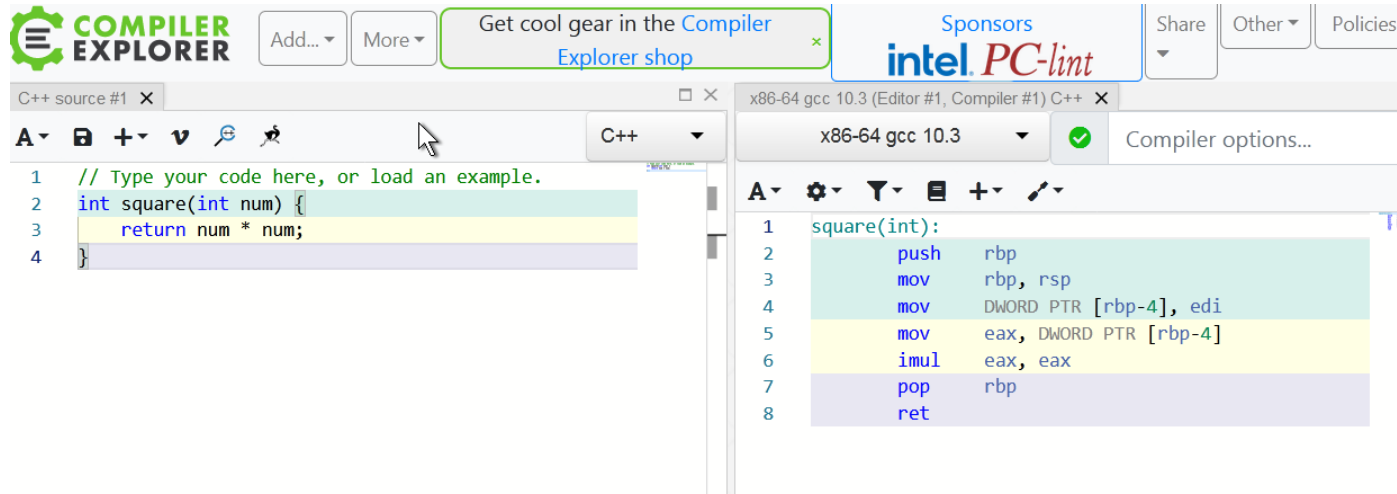


# Digression on code from compilers ...

See <https://godbolt.org/> for useful site to see result from different compilers

Sample source code is on the left

Resulting compiled code is on the right, where you can choose which compiler you're going to examine.



The screenshot displays the Compiler Explorer interface. At the top left is the 'COMPILER EXPLORER' logo. To its right are buttons for 'Add...', 'More', and a link to 'Get cool gear in the Compiler Explorer shop'. Further right are 'Sponsors' (including 'intel.PC-lint'), 'Share', 'Other', and 'Policies'.

The main area is split into two panes. The left pane, titled 'C++ source #1', contains the following C++ code:

```
1 // Type your code here, or load an example.
2 int square(int num) {
3     return num * num;
4 }
```

The right pane, titled 'x86-64 gcc 10.3 (Editor #1, Compiler #1) C++', shows the assembly output for the same code. The assembly is as follows:

```
1 square(int):
2     push    rbp
3     mov     rbp, rsp
4     mov     DWORD PTR [rbp-4], edi
5     mov     eax, DWORD PTR [rbp-4]
6     imul  eax, eax
7     pop     rbp
8     ret
```

# Synchronization

---

On a CPU, you can solve synchronization issues using Locks, Semaphores, Condition Variables, etc.

- Locks -- [https://en.wikipedia.org/wiki/Computer\\_lock](https://en.wikipedia.org/wiki/Computer_lock)
- Semaphores -- [https://en.wikipedia.org/wiki/Semaphore\\_\(programming\)](https://en.wikipedia.org/wiki/Semaphore_(programming))
- Condition Variables -- [https://en.wikipedia.org/wiki/Monitor\\_\(synchronization\)#Condition\\_variables\\_2](https://en.wikipedia.org/wiki/Monitor_(synchronization)#Condition_variables_2)

On a GPU, these solutions can frequently introduce too much memory and process overhead

- Simpler solutions available, many times better suited for parallel programs

# CUDA Synchronization

---

Usually use the `__syncthreads()` function to sync threads **within a block**

- Only works at the block level
  - SMs are separate from each other so can't do “better” than this
- Similar to `barrier()` function in C/C++  
[https://en.wikipedia.org/wiki/Barrier\\_\(computer\\_science\)](https://en.wikipedia.org/wiki/Barrier_(computer_science))
- This `__syncthreads()` call is very useful for kernels using **shared memory**.

# Atomic Operations

---

**Atomic Operations** are operations that **ONLY** happen **in sequence, independent of other processes**

- For example, if you want to add up N numbers by adding the numbers to a variable that starts in 0, you must add one number at a time
- Don't do this though. We'll talk about better ways to do this in the next lecture. Only use when you have no other options

“Atomic operations” in concurrent programming are program operations that run completely independently of any other processes.

See <https://www.techopedia.com/definition/3466/atomic-operation>

# Atomic Operations

---

CUDA provides built in atomic operations

- Use the functions: **atomic<op>(float \*address, float val);**
- Replace <op> with one of: Add, Sub, Exch, Min, Max, Inc, Dec, And, Or, Xor
  - e.g. atomicAdd(float \*address, float val) for atomic addition
- These functions are all implemented using a function called atomicCAS(int \*address, int compare, int val)
  - CAS stands for compare and swap. The function compares \*address to compare and swaps the value to val if the values are different
  - Double precision more accurate, but can be much slower!

# Instruction Dependencies

---

An **Instruction Dependency** is a requirement relationship between instructions that force a sequential execution

- In the example on the right, each summation call must happen in sequence because the value of `acc` depends on the previous summation as well

Can be caused by direct dependencies or requirements set by the execution order of code

- I.e. You can't start an instruction until all previous operations have been completed in a single thread

```
acc += x[0];  
acc += x[1];  
acc += x[2];  
acc += x[3];  
...
```

# Instruction Level Parallelism (ILP)

---

**Instruction Level Parallelism** is when you avoid performances losses caused by instruction dependencies

- Idea: we do not have to wait until instruction  $n$  has finished to start instruction  $n + 1$
- In CUDA, also removes performances losses caused by how certain operations are handled by the hardware



# ILP Example

```
z0 = x[0] + y[0];  
z1 = x[1] + y[1];
```

COMPILATION



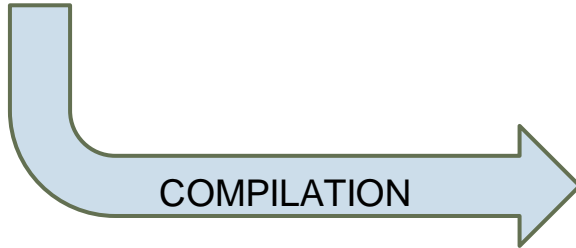
```
x0 = x[0];  
y0 = y[0];  
z0 = x0 + y0;
```

```
x1 = x[1];  
y1 = y[1];  
z1 = x1 + y1;
```

- The second half of the code can't start execution until the first half completes

# ILP Example

```
z0 = x[0] + y[0];  
z1 = x[1] + y[1];
```



```
x0 = x[0];  
y0 = y[0];  
x1 = x[1];  
y1 = y[1];  
z0 = x0 + y0;  
z1 = x1 + y1;
```

- Sequential nature of the code due to instruction dependency has been minimized.
- Additionally, this code minimizes the number of memory transactions required

# Warp Schedulers

---

Warp schedulers find a warp that is ready to execute its next instruction and available execution cores and then start execution

GPU has at least **one warp scheduler per SM**. Each scheduler has **2 dispatchers**.

The scheduler picks an eligible warp and executes all threads in the warp .

If any of the threads in the executing warp stalls (uncached memory read) the scheduler makes it inactive. If there are no eligible warps left then GPU **idles**

Context switch between warps is fast—About 1 or 2 cycles (1 nano-second on 1 GHz GPU)—The whole thread block has resources allocated on an SM by the compiler ahead of time

- GK110:
  - 4 warp schedulers in each SM and 2 dispatchers in each scheduler
  - Can start instructions in up to 4 warps each clock and up to 2 **subsequent, independent** instructions in each warp. Up to 80 warp instructions to hide latency of warp add (10 cycles)

# Occupancy

---

Idea: Need enough independent threads per SM to hide latencies

- Instruction latencies
- Memory access latencies

**Occupancy: number of concurrent threads per SM**

- Occupancy = active warps per SM / max warps per SM

Number of threads that fit per SM (max warps per SM) is determined by the hardware resources of the GPU.

Threads/block matters because (combined with the number of blocks) let's us know how many warps there are on the SM.

# Occupancy

---

The number of active warps per SM is determined by the limiting resources

- Registers per thread
  - SM registers are partitioned among the threads
- Shared memory per thread block
  - SM shared memory is partitioned among the blocks
- Threads per thread block
  - Threads are allocated at thread block granularity

Needed occupancy depends on the code

- More independent work per thread -> less occupancy is needed
- Memory-bound codes tend to need more occupancy Higher latency than for arithmetic, need more work to hide it

Don't need for 100% occupancy for maximum performance

# GK110 (Kepler) numbers

---

- max threads / SM = 2048 (64 warps)
- max threads / block = 1024 (32 warps)
- 32 bit registers / SM = 64k
- max shared memory / SM = 48KB

The number of blocks that run concurrently on a SM depends on the resource requirements of the block!

# GK110 Occupancy

---

## 100% occupancy

- 2 blocks of 1024 threads
- 32 registers/thread
- 24KB of shared memory / block

## 50% occupancy

- 1 block of 1024 threads
- 64 registers/thread
- 48KB of shared memory / block

# Review of Terms ...

---

- Synchronization
- Atomic Operations
- Instruction Dependencies
- Instruction Level Parallelism (ILP)
- Warp Scheduler
- Occupancy



# Other Resources

---

ILP

[https://www.nvidia.com/content/GTC-2010/pdfs/2238\\_GTC2010.pdf](https://www.nvidia.com/content/GTC-2010/pdfs/2238_GTC2010.pdf)

Warps and Occupancy

[http://on-demand.gputechconf.com/gtc-express/2011/presentations/cuda\\_webinars\\_WarpsAndOccupancy.pdf](http://on-demand.gputechconf.com/gtc-express/2011/presentations/cuda_webinars_WarpsAndOccupancy.pdf)

# Next time...

---

Set 2 Recitation on Friday (04/12)

GPU based algorithms (next week)