Administration

- Team/project proposals are due **today**

- **CPU and GPU servers for experimentation**
  - Use Euler for most of your experiments, be careful when experimenting on Leonhard!
  - Talk to us about larger-scale GPU tests first

- Questions?
Review of last lectures

- **Architecture case studies**
  - Memory performance is often the bottleneck
  - Parallelism grows with compute performance
  - Caching is important
  - Several issues to address for parallel systems

- **Cache Coherence**
  - Hardware support to aid programmers
  - Two guarantees:
    - *Write propagation* (updates are *eventually* visible to all readers)
    - *Write serialization* (writes to the same location are observed in global order)
  - Two major mechanisms:
    - *Snooping* (MESI, MOESI)
    - *Directory-based*
DPHPC Overview

**concepts & techniques**
- locality
  - caches
  - memory hierarchy
- parallelism
  - vector ISA
  - shared memory
  - distributed memory

**cache coherency**
- memory models
  - locks
  - lock free
  - wait free
  - linearizability
- distributed algorithms
- group communications

**models**
- Amdahl's and Gustafson's law
  - memory
    - $\alpha - \beta$
  - PRAM
  - LogP

**I/O complexity**
- balance principles I
- Little's Law
- balance principles II
- scheduling
Goals of this lecture

- Cache-coherence is not enough
  - Many more subtle issues for parallel programs

- Memory Models
  - Sequential consistency
  - Why threads cannot be implemented as a library
  - Relaxed consistency models

- GPU/SIMT Model

- Linearizability
  - More complex objects
Is Coherence Everything?

- Coherence is concerned with behavior of individual locations
- Consider the program (initial X,Y,Z = 0)

What value will Z on P2 have?
- Y=10 does not need to have completed before X=2 is visible to P2!
  - This allows P2 to exit the loop and read Y=0
  - This may not be the intent of the programmer!
  - This may be due to congestion (imagine X is pushed to a remote cache while Y misses to main memory) and or due to write buffering, or …
- Exercise: what happens when Y and X are on the same cache line (assume MESI and no write buffer)?
Memory Models

- Need to define what it means to “read a location” and “to write a location” and the respective ordering!
  - What values should be seen by a processor

- First thought: extend the abstractions seen by a sequential processor:
  - Compiler and hardware maintain data and control dependencies at all levels:

Two operations to the same location

- Y = 10
- ....
- T = 14
- Y = 15

One operation controls execution of others

- Y = 5
- X = 5
- T = 3
- Y = 3
- if (X == Y)
- Z = 5
- ....

Sequential Processor

- **Correctness condition:**
  - The result of the execution is the same as if the operations had been executed in the order specified by the program “*program order*”
  - A read returns the value last written to the same location “*last*” is determined by *program order*

- **Consider only memory operations (e.g., a trace)**

- **N Processors**
  - P1, P2, ..., PN

- **Operations**
  - Read, Write on shared variables (initial state: most often all 0)

- **Notation:**
  - P1: R(x):3 P1 reads x and observes the value 3
  - P2: W(x,5) P2 writes 5 to variable x
Program order
- Deals with a single processor
- Per-processor order of memory accesses, determined by program’s Control flow
- Often represented as trace

Visibility/memory order
- Deals with operations on all processors
- Order of memory accesses observed by one or more processors
- e.g., “every read of a memory location returns the value that was written last”
  Defined by memory model
Sequential Consistency

- Extension of sequential processor model

- The execution happens as if
  1. The operations of all processes were executed in some sequential order (atomicity requirement), and
  2. The operations of each individual processor appear in this sequence in the order specified by the program (program order requirement)

- Applies to all layers!
  - Disallows many compiler optimizations (e.g., reordering of any memory instruction)
  - Disallows many hardware optimizations (e.g., store buffers, nonblocking reads, invalidation buffers)
Illustration of Sequential Consistency

- Globally consistent view of memory operations (atomicity)
- Strict ordering in program order

Processors issue in program order

The “switch” selects arbitrary next operation

Program
A = B;

Memory

Read B  Write A

P₁
Write A

P₂
Write A

P₃
Write A

P₄
Write A
Original SC Definition

“The result of any execution is the same as if the operations of all the processes were executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its program”

(Lamport, 1979)
Alternative SC Definition

- Textbook: Hennessy/Patterson Computer Architecture

- A sequentially consistent system maintains three invariants:
  1. A load \( L \) from memory location \( A \) issued by processor \( P_i \) obtains the value of the previous store to \( A \) by \( P_i \), unless another processor stored a value to \( A \) in between
  2. A load \( L \) from memory location \( A \) obtains the value of a store \( S \) to \( A \) by another processor \( P_k \) if \( S \) and \( L \) are “sufficiently separated in time” and if no other store occurred between \( S \) and \( L \)
  3. Stores to the same location are serialized (defined as in (2))

- “Sufficiently separated in time” not precise
  - Works but is not formal (a formalization must include all possibilities)
Memory Models

- Contract at each level between programmer and processor

Programmer

High-level language (API/PL)

Compiler Frontend

Intermediate Language (IR)

Compiler Backend/JIT

Machine code (ISA)

Processor

- Optimizing transformations
- Reordering
- Operation overlap
  - OOO Execution
  - VLIW, Vector ISA
Operation Reordering

- Recap: “normal” sequential assumption:
  - Compiler and hardware can reorder instructions as long as control and data dependencies are met
- Examples:
  - Register allocation
  - Code motion
  - Common subexpression elimination
  - Loop transformations

Compiler

Hardware

- Pipelining
- Multiple issue (OOO)
- Write buffer bypassing
- Nonblocking reads
Simple compiler optimization

- Initially, all values are zero

- Assume P1 and P2 are compiled separately!
- What optimizations can a compiler perform for P1?
  
  \textit{Register allocation or even replace with constant, or}

  \textit{Switch statements}

- What happens?
  
  \textit{P2 may never terminate, or}

  \textit{Compute with wrong input}
Sequential Consistency Examples

- **Relying on program order: Dekker’s algorithm**
  - Initially, all zero

  P1
  
  ```
  a = 1
  if(b == 0)
      critical section
  a = 0
  ```

  P2
  
  ```
  b = 1
  if(a == 0)
      critical section
  b = 0
  ```

  - What can happen at compiler and hardware level?

Without SC, both writes may have gone to a write buffer, in which case both P1s would read 0 and enter the critical section together.

Nobody enters the critical section.
Sequential Consistency Examples

Relying on single sequential order (atomicity): three sharers

- P1:
  a = 5
  a = 1

- P2:
  if (a == 1)
  b = 1

- P3:
  if (b == 1)
  print(a)

What can be printed if visibility is not atomic?

P3 has not seen P1: W(a,1) yet!

What each P thinks the order is:

P1: W(a,5)
P1: W(a,1)
P2: R(a): 1
P2: W(b,1)
P3: R(b): 1
P3: R(a): 5
PRINT(5)
Optimizations violating program order

- **Analyzing P1 and P2 in isolation!**
  - Compiler can reorder

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
</table>
  | a = 1  
  if(b == 0)  
*critical section*  
a = 0 | b = 1  
if(a == 0)  
*critical section*  
b = 0 |

- Hardware can reorder, assume writes of a,b go to write buffer or speculation

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
</table>
  | a = 1  
if(b == 0)  
*critical section*  
a = 0 | b = 1  
if(a == 0)  
*critical section*  
b = 0 |

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
</table>
  | if(b == 0)  
*critical section*  
a = 0  
else  
a = 1 | if(a == 0)  
*critical section*  
b = 0  
else  
b = 1 |

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
</table>
  | if(b == 0)  
*critical section*  
a = 0 | if(a == 0)  
*critical section*  
b = 0  
else  
b = 1 |
Board
Considerations

- Define partial order on memory requests $A \rightarrow B$
  - If $P_i$ issues two requests $A$ and $B$ and $A$ is issued before $B$ in program order, then $A \rightarrow B$
  - $A$ and $B$ are issued to the same variable, and $A$ is issued first, then $A \rightarrow B$ (on all processors)

- These partial orders can be interleaved, define a total order
  - Many total orders are sequentially consistent!

- Example:
  - $P_1$: $W(a)$, $R(b)$, $W(c)$
  - $P_2$: $R(a)$, $W(a)$, $R(b)$
  - Are the following schedules (total orders) sequentially consistent?
    1. $P_1$: $W(a)$, $P_2$: $R(a)$, $P_2$: $W(a)$, $P_1$: $R(b)$, $P_2$: $R(b)$, $P_1$: $W(c)$
    2. $P_1$: $W(a)$, $P_2$: $R(a)$, $P_1$: $R(b)$, $P_2$: $R(b)$, $P_1$: $W(c)$, $P_2$: $W(a)$
    3. $P_2$: $R(a)$, $P_2$: $W(a)$, $P_1$: $R(b)$, $P_1$: $W(a)$, $P_1$: $W(c)$, $P_2$: $R(b)$
Write buffer example

- **Write buffer**
  - Absorbs writes faster than the next cache → prevents stalls
  - Aggregates writes to the same cache line → reduces cache traffic
Write buffer example

- Reads can bypass previous writes for faster completion
  - If read and write access different locations
  - No order between write and following read (W \rightarrow R)

P1
- $W(a, 1)$
- $R(b): 0$

P2
- $W(b, 1)$
- $R(a): 0$

*not seq. consistent*
Nonblocking read example

- $W \rightarrow W$: OK
- $R \rightarrow W, R \rightarrow R$: No order between read and following read/write

![Diagram showing nonblocking read example]

- $P_1$: Initial state:
  - $L_1$: $x = 0$, $y = 2$ (Cache miss!)
  - $L_2$: $x = 0$, $y = 2$

- $P_2$: Operations:
  - $R(y): 2$
  - $R(x): 0$
  - $W(x, 1)$
  - $W(y, 2)$

- Not seq. consistent

$y = 2$ in $L_1$ is not consistent with $y = 2$ in $L_2$. The state is not sequentially consistent.
Discussion

- **Programmer’s view:**
  - Prefer sequential consistency
  - Easiest to reason about

- **Compiler/hardware designer’s view:**
  - Sequential consistency disallows many optimizations!
  - Substantial speed difference
  - Most architectures and compilers don’t adhere to sequential consistency!

- **Solution: synchronized programming**
  - Access to shared data (aka. “racing accesses”) are ordered by synchronization operations
  - Synchronization operations guarantee memory ordering (aka. fence)
  - More later!
Cache Coherence vs. Memory Model

- Varying definitions!

- Cache coherence: a mechanism that propagates writes to other processors/caches if needed, recap:
  - Writes are eventually visible to all processors
  - Writes to the same location are observed in (one) order

- Memory models: define the bounds on when the value is propagated to other processors
  - E.g., sequential consistency requires all reads and writes to be ordered in program order

Good read: McKenney: “Memory Barriers: a Hardware View for Software Hackers”
The fun begins: Relaxed Memory Models

- **Sequential consistency**
  - $R \rightarrow R, R \rightarrow W, W \rightarrow R, W \rightarrow W$ (all orders guaranteed)

- **Relaxed consistency (varying terminology):**
  - Processor consistency (aka. Total Store Ordering)
    
    Relaxes $W \rightarrow R$
  - Partial write (store) order (aka. Partial Store Ordering)
    
    Relaxes $W \rightarrow R, W \rightarrow W$
  - Weak consistency and release consistency (aka. Relaxed Memory Order)
    
    Relaxes $R \rightarrow R, R \rightarrow W, W \rightarrow R, W \rightarrow W$
  - Other combinations/variants possible
    
    *There are even more types of orders (above is a simplification)*

Implemented (loosely) on the GPU, ARM
## Architectures

<table>
<thead>
<tr>
<th>Type</th>
<th>Alpha</th>
<th>ARMv7</th>
<th>PA-RISC</th>
<th>POWER</th>
<th>SPARC</th>
<th>x86</th>
<th>AMD64</th>
<th>IA-64</th>
<th>z/Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loads reordered after loads</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Loads reordered after stores</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Stores reordered after stores</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stores reordered after loads</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Atomic reordered with loads</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Atomic reordered with stores</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Dependent loads reordered</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Incoherent instruction cache pipeline</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Source: wikipedia
Case Study: Memory ordering on Intel (x86)

- Intel® 64 and IA-32 Architectures Software Developer's Manual
  - Volume 3A: System Programming Guide
  - Chapter 8.2 Memory Ordering

- Google Tech Talk: IA Memory Ordering
  - Richard L. Hudson
  - [http://www.youtube.com/watch?v=WUfvvFD5tAA](http://www.youtube.com/watch?v=WUfvvFD5tAA)
x86 Memory model: TLO + CC

- **Total lock order (TLO)**
  - Instructions with “lock” prefix enforce total order across all processors
  - Implicit locking: xchg (locked compare and exchange)

- **Causal consistency (CC)**
  - Write visibility is transitive

- **Eight principles**
  - After some revisions 😊
The Eight x86 Principles

1. “Reads are not reordered with other reads.” (R→R)
2. “Writes are not reordered with other writes.” (W→W)
3. “Writes are not reordered with older reads.” (R→W)
4. “Reads may be reordered with older writes to different locations but not with older writes to the same location.” (NO W→R!)
5. “In a multiprocessor system, memory ordering obeys causality.” (memory ordering respects transitive visibility)
6. “In a multiprocessor system, writes to the same location have a total order.” (implied by cache coherence)
7. “In a multiprocessor system, locked instructions have a total order.” (enables synchronized programming!)
8. “Reads and writes are not reordered with locked instructions.” (enables synchronized programming!)
Principle 1 and 2

Reads are not reordered with other reads. ($R \rightarrow R$)

Writes are not reordered with other writes. ($W \rightarrow W$)

All values zero initially. r1 and r2 are registers.

P1:
- $a = 1$
- $b = 2$

P2:
- $r1 = b$
- $r2 = a$

Question: is r1=0, r2=1 allowed?

Memory

Reads and writes observed in program order. Cannot be reordered!

If r1 == 2, then r2 must be 1!
Not allowed: r1 == 2, r2 == 0
Principle 3

Writes are not reordered with older reads. (R→W)

Question: is r1==1 and r2==1 allowed?

Question: is r1==0 and r2==0 allowed?

If r1 == 1, then P2:W(a) → P1:R(a), thus r2 must be 0!
If r2 == 1, then P1:W(b) → P1:R(b), thus r1 must be 0!

Memory

All values zero initially
Principle 4

Reads may be reordered with older writes to different locations but not with older writes to the same location. (NO W→R!)

Question: is r1=1, r2=0 allowed?

Allowed: r1=0, r2=0.
Sequential consistency can be enforced with mfence.
Attention: this rule may allow reads to move into critical sections!
Principle 5

In a multiprocessor system, memory ordering obeys causality (memory ordering respects transitive visibility).

Question: is $r1==1$, $r2==0$, $r3==1$ allowed?

If $r1 == 1$ and $r2==1$, then $r3$ must read 1. Not allowed: $r1 == 1$, $r2 == 1$, and $r3 == 0$. Provides some form of atomicity.
Principle 6

In a multiprocessor system, writes to the same location have a total order (implied by cache coherence).

All values zero initially

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=1</td>
<td>a=2</td>
<td>r1 = a</td>
<td>r3 = a</td>
<td></td>
</tr>
<tr>
<td>r2 = a</td>
<td>R(a)</td>
<td>R(a)</td>
<td>R(a)</td>
<td></td>
</tr>
</tbody>
</table>

Question: is r1 = 0, r2 = 2, r3 = 0, r4 = 1 allowed?

- Not allowed: r1 == 1, r2 == 2, r3 == 2, r4 == 1
- If P3 observes P1’s write before P2’s write, then P4 will also see P1’s write before P2’s write
- Provides some form of atomicity
In a multiprocessor system, locked instructions have a total order. (enables synchronized programming!)

**Principle 7**

All values zero initially, registers $r1=r2=1$

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>P2</td>
<td>P3</td>
<td>P4</td>
</tr>
</tbody>
</table>
| $\text{xchg}(a,r1)$ | $\text{xchg}(b,r2)$ | $r3 = a$ | $r5 = b$
| $r4 = b$ | $r6 = a$ |

All values zero initially, registers $r1==r2==1$

- **Not allowed:** $r3 == 1$, $r4 == 0$, $r5 == 0$, $r6 == 0$
- If P3 observes ordering $P1: \text{xchg} \rightarrow P2: \text{xchg}$, then P4 observes the same ordering
- (xchg has implicit lock)

Question: is $r3=1$, $r4=0$, $r5=0$, $r6=1$ allowed?
Principle 8

Reads and writes are not reordered with locked instructions. (enables synchronized programming!)

- Not allowed: \( r_2 = 0, r_4 = 0 \)
- Locked instructions have total order, so P1 and P2 agree on the same order
- If volatile variables use locked instructions → practical sequential consistency (more later)
An Alternative View: x86-TSO

- Sewell el al.: “x86-TSO: A Rigorous and Usable Programmer’s Model for x86 Multiprocessors”, CACM May 2010

“[...] real multiprocessors typically do not provide the sequentially consistent memory that is assumed by most work on semantics and verification. Instead, they have relaxed memory models, varying in subtle ways between processor families, in which different hardware threads may have only loosely consistent views of a shared memory. Second, the public vendor architectures, supposedly specifying what programmers can rely on, are often in ambiguous informal prose (a particularly poor medium for loose specifications), leading to widespread confusion. [...] We present a new x86-TSO programmer’s model that, to the best of our knowledge, suffers from none of these problems. It is mathematically precise (rigorously defined in HOL4) but can be presented as an intuitive abstract machine which should be widely accessible to working programmers. [...]”

Newer RMA systems: A. Dan, P. Lam, TH, A. Vechev: Modeling and Analysis of Remote Memory Access Programming, ACM OOPSLA’16
Meanwhile, on GPUs...

https://www.youtube.com/watch?v=VoggQoscJYvk

Published September 28, 2019
GPU Hardware Model

- GPUs devote more hardware to computations
- Not as general-purpose as a CPU
- Used as an accelerator to a *host system*
- Device is built from Streaming Multiprocessors (SMs)
- Scalable caching mechanism (not all coherent!)
- Communication with host system via PCIe bus
  - Or NVLINK

Figures and code courtesy of nvidia.com
GPU Hardware Model – The SM

- ALUs divided into specific functions
  - Can run concurrently
  - Hardware can fast-switch between active “threads” to hide latency

- Multiple levels of instruction and data caching
  - “L0” cache for instructions

- Scratch-pad (“shared”) memory controlled by the programmer

- Specialized units for specific operations
  - Example – Tensor Cores:
    
    \[
    D = \begin{pmatrix}
    A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\
    A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\
    & B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\
    & B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\
    & & B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \\
    & & & C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\
    & & & C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\
    & & & C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3}
    \end{pmatrix}
    \]

FP16 or FP32

FP16 or FP32

FP16

FP16

128KB L1 Data Cache / Shared Memory

Figures and code courtesy of nvidia.com
GPU Programming Model

- CUDA, OpenCL, ROCm
  - CUDA is for NVIDIA GPUs, ROCm for AMD, OpenCL for both (but slower)

- Highly documented in the CUDA Programming Guide
GPU Programming Model – Memory

- Memory explicitly allocated on the GPU
  - Managed memory exists, but slower

- Data copied from host ↔ GPU explicitly

- DMA if data is pinned on the host, or on another GPU

- Question: Are memory writes to pinned memory visible to the GPU?
  - Answer: Not necessarily without fences and the volatile keyword

```c
int main() {
    float *A = new float[N];
    float *gpuA;
    float *pinnedB;
    cudaMalloc(&gpuA, N * sizeof(float));
    cudaMallocHost(&pinnedB, N * sizeof(float));

    cudaMemcpyAsync(gpuA, A, N*sizeof(float), cudaMemcpyHostToDevice);

    // ...

    cudaMemcpy(A, gpuA, N*sizeof(float), cudaMemcpyDeviceToHost);

    cudaFreeHost(pinnedB);
    cudaFree(gpuA);
    delete[] A;
}
```
GPU Programming Model – Execution

- GPU-compiled functions written separately from host code
  - Both can coexist in .cu files, no GPU code in .c/.cpp files

- NVIDIA CUDA is (mostly) compatible with C++14

- Code is asynchronously executed
  - Streams act as “command queues” to the GPU
  - Events can be queued onto the stream for synchronization

- Logical division: threads and thread-blocks
  - threadIdx.{x,y,z}
  - blockIdx.{x,y,z}
  - blockDim.{x,y,z}

---

```c
__global__ void VecAdd(float* A, float* B, float* C) {
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main() {
    ...
    // Kernel invocation with M thread-blocks, each with N threads
    VecAdd<<<M, N>>>(A, B, C);
    ...
}
```

Figures and code courtesy of [nvidia.com](http://nvidia.com)
What does this kernel do?

```c
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C) {

    float Cvalue = 0;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    for (int e = 0; e < A.width; ++e)
        Cvalue += A.data[row * A.width + e] * B.data[e * B.width + col];

    C.data[row * C.width + col] = Cvalue;
}
```
Example GPU Execution Trace

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.98</td>
<td>cudaStreamSynchronize</td>
</tr>
<tr>
<td>11.99</td>
<td>cudaStreamSynchronize</td>
</tr>
<tr>
<td>12</td>
<td>cudaStreamSynchronize</td>
</tr>
<tr>
<td>12.01</td>
<td>cudaStreamSynchronize</td>
</tr>
<tr>
<td>12.02</td>
<td>cudaStreamSynchronize</td>
</tr>
<tr>
<td>12.03</td>
<td>cudaStreamSynchronize</td>
</tr>
<tr>
<td>12.04</td>
<td>cudaStreamSynchronize</td>
</tr>
<tr>
<td>12.05</td>
<td>cudaStreamSynchronize</td>
</tr>
<tr>
<td>12.06</td>
<td>cudaStreamSynchronize</td>
</tr>
<tr>
<td>12.07</td>
<td>cudaStreamSynchronize</td>
</tr>
</tbody>
</table>

Stream 23 waits for event queued on 17
Single Instruction, Multiple Threads (SIMT)

- A warp keeps a single PC and stack
  - Composed of 32 “threads”
- Thread-blocks are composed of multiple warps
  - Always lie in the same SM
- Warps execute the same instruction, but on different data
  - Same register name is mapped to a different actual register

- Warp divergence can cause 32x slowdown (and more):

```c
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```

```
IMAD.MOV.U32 R0, RZ, RZ, R6;
ISETP.GT.U32.AND P0, PT, R0, 0x7f, PT;
CS2R R6, SRZ;
MOV R8, 0x5;
SEL R8, R8, 0x1, P0;
@P0 BRA 0x2c00;
CS2R R6, SRZ;
@P0 BRA 0x2a10;
@!PT SHFL.IDX PT, RZ, RZ, RZ, RZ;
IMAD.SHL.U32 R2, R0, 0x8, RZ;
LDS.U.64 R16, [R2];
DSETP.GEU.AND P0, PT, |R16|, c[0x2][0x8], PT;
DADD R4, -RZ, |R16|;
```

Figures and code courtesy of nvidia.com
Memory Coalescing

- Memory accesses of aligned elements (e.g., LDG.<X>) can be shared across threads in a warp

<table>
<thead>
<tr>
<th>Compute capability:</th>
<th>2.0 and later</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory transactions:</td>
<td>Uncached</td>
</tr>
<tr>
<td></td>
<td>Cached</td>
</tr>
<tr>
<td>1x 32B at 128</td>
<td>1x 128B at 128</td>
</tr>
<tr>
<td>1x 32B at 160</td>
<td>1x 128B at 160</td>
</tr>
<tr>
<td>1x 32B at 192</td>
<td>1x 128B at 192</td>
</tr>
<tr>
<td>1x 32B at 224</td>
<td>1x 128B at 224</td>
</tr>
<tr>
<td>1x 32B at 256</td>
<td>1x 128B at 256</td>
</tr>
</tbody>
</table>

Figures and code courtesy of nvidia.com