# CS 623: More CUDA Features

## Synchronization, Warp Primitives, and Streams

**Swarnendu Biswas**

Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur

Sem 2026-27-I

# Synchronization in CUDA

# Data Races in GPU Code

- Two accesses from two different threads **conflict** when they access the same shared variable where at least one access is a write
- Accesses are not ordered by synchronization operations are **concurrent** (i.e., can happen at the same time)

**Data race**
A pair of concurrent and conflicting accesses form a data race

A data race can occur between threads within a single warp, warps in the same block, threads in different blocks, threads in different kernels, threads in different GPUs, or between CPU and GPU threads

# Examples of Data Races

```
__global__ void intrawarp(int *d) {
  d[0] = threadIdx.x;
}
```

```
__global__ void interwarp(int *gc) {
  *gc = *gc + 1;
}
```

Data races are considered undefined behavior (UB) in CUDA

> The program may fail to compile, or it may execute incorrectly (either crashing or silently generating incorrect results), or it may fortuitously do exactly what the programmer intended.
>
> - John Regehr

John Regehr. A Guide to Undefined Behavior in C and C++.

# Data Races are Evil!

Transformations possible by a hypothetical optimizing compiler

```
__global__ void racyKernel(int *ret) {
  __shared__ int sum;
  sum = 0;
  __syncthreads();

  for (int i = 0; i < 4; i++)
    sum += 1; // atomicAdd(&sum, 1);
  }
  __syncthreads();

  // Expected sum == (4 * blockDim.x)
  if (threadIdx.x == 0)
    ret[blockIdx.x] = sum;
}
```

1

```
__global__ void optRacyKernel1(int *ret) {
  int sum = 0;
  for (int i = 0; i < 4; i++)
    sum += 1;
  if (threadIdx.x == 0)
    ret[blockIdx.x] = sum;
}
```

2

```
__global__ void optRacyKernel2(int *ret) {
  if (threadIdx.x == 0)
    ret[blockIdx.x] = 4;
}
```

M. Stephenson et al. SupeCollider: Scalable and Effective Data Race Detection for CUDA, PLDI 2026.

# Intra-Block Synchronization

- `__syncthreads()` waits until all threads in the thread block have reached this point
- All global and shared memory accesses made by the participating threads prior to `__syncthreads()` are visible to all threads in the block

- A `__syncthreads()` statement must be executed by **all** threads in a block
  - If `__syncthreads()` is in an `if` statement, then either all threads in the block execute the `then` path that includes the `__syncthreads()` or none of them does
  - If `__syncthreads()` statement is in each path of an `if-then-else` statement, then either all threads in a block execute the `__syncthreads()` on the `then` path or all of them execute the `else` path
    - The two `__syncthreads()` are different barrier synchronization points

syncthreads-deadlock1.cu
syncthreads-deadlock2.cu

# Data Races from PTX Perspective

**Data race**
Two conflicting memory operations that are not related in causality order and are not morally strong

- Morally strong memory operations have one of the following opcode qualifiers: .relaxed, .acquire, .release, .acq_rel, .volatile, or .mmio
  - Morally strong operations establish memory ordering
- Weak memory operations use the .weak qualifier and indicate a memory instruction with no synchronization
  - Do not impose ordering constraints relative to other threads
  - A weak store by one thread is not guaranteed to become visible to another thread without the help of additional synchronization or morally strong operations

# Atomic Operations

Atomic operations allow performing atomic read-modify-write (RMW) operations on one 32-bit, 64-bit, or 128-bit data residing in global or shared memory

- Prevent data races associated with multiple threads concurrently accessing a global or shared memory variable
- For example, `atomicAdd()`, `atomicSub()`, `atomicMin()`, `atomicMax()`, `atomicInc()`, `atomicDec()`, `atomicExch()`, `atomicCAS()`

Atomic functions do not act as memory fences and do not imply synchronization or ordering constraints for memory operations

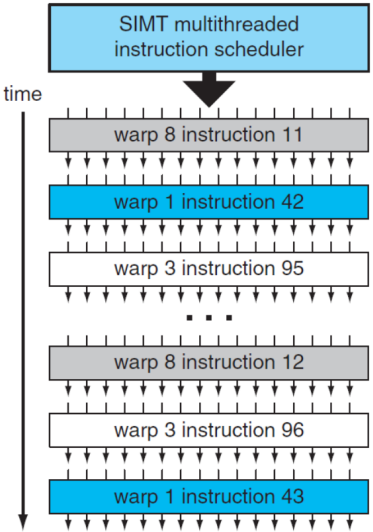
Possible lock implementation

```
*mutex=0;
while(!atomicCAS(mutex, 0, 1)) {}
// critical section
atomicExch(mutex, 0);
```

CUDA C++ Programming Guide: Atomic Functions

# Lockstep Execution

- All threads in a warp run in lockstep
- Warps share an instruction stream, i.e., same instruction is fetched for all threads in a warp during the instruction fetch cycle
  - Prior to Volta, warps used a single shared program counter
- In the execution phase, each thread will either execute the instruction or will execute nothing
- Individual threads in a warp have their own instruction address counter and register state
- Warp threads are fully synchronized, i.e., there is an implicit barrier after each step/instruction



# Thread Divergence

- If some threads take the `then` branch and other threads take the `else` branch, they cannot operate in lockstep
  - Some threads must wait for the others to execute, serializes execution at that point
- The programming model does not prevent thread divergence
- Divergence occurs only within a warp

```
if (threadIdx.x < 16)
  A[threadIdx.x] = 1;
else
  A[threadIdx.x] = 2;
```

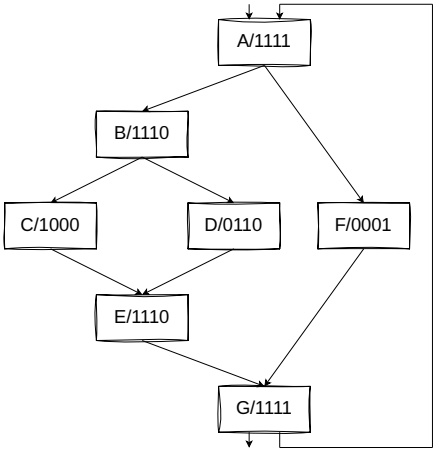
# Example Code and Corresponding PTX

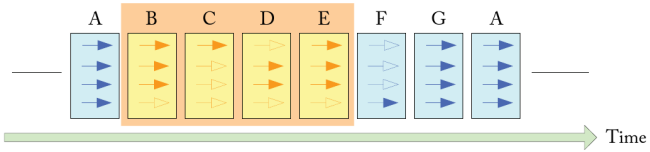
```
do {
  t1 = tid * N;       // A
  t2 = t1 + i;
  t3 = data1[t2];
  t4 = 0;
  if (t3 != t4) {
    t5 = data2[t2]; // B
    if (t5 != t4) {
      x += 1;         // C
    } else {
      y += 2;         // D
    }
  } else {
    z += 3;           // F
  }
  i++;                // G
} while (i < N);
```

```
A:    mul.lo.u32      t1, tid, N;
      add.u32         t2, t1, i;
      ld.global.u32   t3, [t2];
      mov.u32         t4, 0;
      setp.eq.u32     p1, t3, t4;
@p1   bra             F;
B:    ld.global.u32   t5, [t2];
      setp.eq.u32     p2, t5, t4;
@p2   bra             D;
C:    add.u32         x, x, 1;
      bra             E;
D:    add.u32         y, y, 2;
E:    bra             G;
F:    add.u32         z, z, 3;
G:    add.u32         i, i, 1;
      setp.le.u32     p3, i, N;
@p3   bra             A;
```

Jeesun Kim. Branch Divergence.

# SIMT Stack-based Execution



# SIMT Stack-based Execution

How does GPU hardware enable threads within a warp to follow different code paths?

- A per-warp stack is used to manage divergent control flow
- Only threads at the top of the stack are eligible for scheduling

| TOS | Reconv PC | Next PC | Active Mask |
|---|---|---|---|
| | – | G | 1111 |
| | G | B | 1110 |
| → | G | F | 0001 |

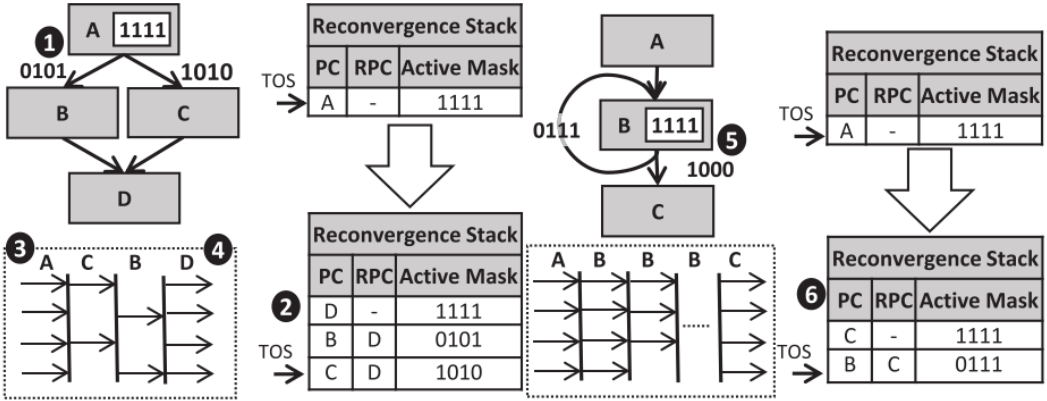
| TOS | Reconv PC | Next PC | Active Mask |
|---|---|---|---|
| | – | G | 1111 |
| | G | E | 1110 |
| | E | D | 0110 |
| → | E | C | 1000 |

| TOS | Reconv PC | Next PC | Active Mask |
|---|---|---|---|
| | – | G | 1111 |
| → | G | E | 1110 |

Threads that diverge can be forced to continue executing in lockstep from a reconvergence point

Is the order of insertion in the SIMT stack important?

# Stack-Based Reconvergence




**(a)** If-else Example **(b)** Loop Example

A. ElTantawy and T. M. Aamodt. MIMD Synchronization on SIMT Architectures, MICRO 2016.

# Reconvergence Scheduling Constraints

Stack-based SIMT execution constrains thread scheduling

**Serialization** If the threads in the taken branch execute first, then the threads in the not-taken branch block until the taken branch reaches the reconvergence point (or vice versa)

**Forced reconvergence** When the threads in the taken branch reach the reconvergence point, they block waiting for the threads in the not-taken branch to reach the reconvergence point (or vice versa)

A. ElTantawy and T. M. Aamodt. MIMD Synchronization on SIMT Architectures, MICRO 2016.

# SIMT Deadlock

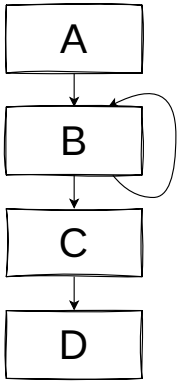
Stack-based implementation of SIMT execution can lead to a deadlock

- Happens because of a cyclic dependence between the scheduling constraints and a synchronization operation

```
A: *mutex=0;
B: while(!atomicCAS(mutex, 0, 1)) {}
C: // critical section
D: atomicExch(mutex, 0);
```

| TOS | Reconv PC | Next PC | Active Threads |
|---|---|---|---|
| | C | B | 1–31 |
| → | C | C | 0 |

| TOS | Reconv PC | Next PC | Active Threads |
|---|---|---|---|
| → | C | B | 1–31 |



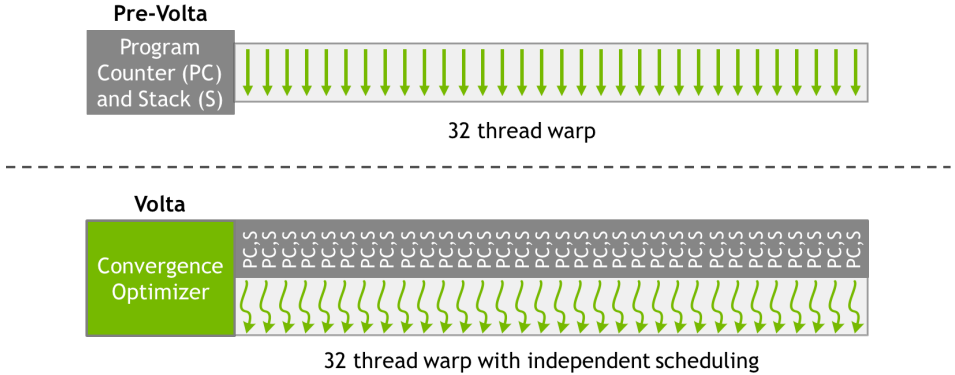
# Avoiding SIMT Deadlock

```
*mutex=0;
while(!atomicCAS(mutex, 0, 1)) {}
// critical section
atomicExch(mutex, 0);
```

```
int done = 0;
*mutex=0;
while (!done) {
  if (!atomicCAS(&mutex, 0, 1)) {
    // critical section
    atomicExch(mutex, 0);
    done = 1;
  }
}
```

# Volta SIMT Model

- Architectures older than Volta maintained a shared program counter (PC) and call stack per warp
- Volta onward, the execution state (i.e., PC and call stack) is maintained per thread




NVIDIA Tesla V100 GPU Architecture

# Independent Thread Scheduling (ITS)

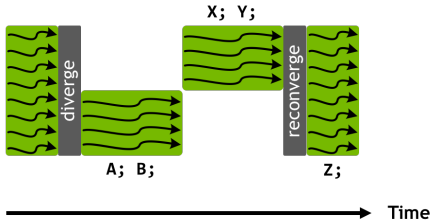
- Volta architecture introduces ITS among threads in a warp
- ITS allows intra-warp synchronization which was previously not possible
- Threads can now diverge and reconverge at sub-warp granularity
- ITS thus makes it easy to implement complex, fine-grained sharing on GPUs

- The SIMT stack is replaced with per-warp convergence barriers
  - The metadata includes barrier participation mask, barrier convergence state, and per-thread states like PC and active status

# SIMT Models across NVIDIA GPU Architectures

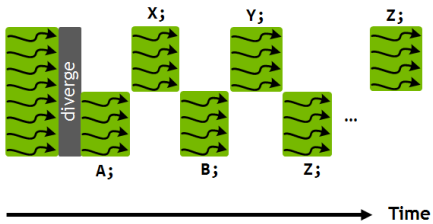
## Till Pascal

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




## Volta onward

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

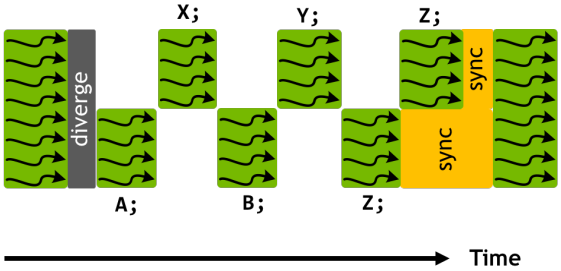



NVIDIA Tesla V100 GPU Architecture

# Intra-warp Synchronization

Use `__syncwarp()` to reconverge threads within a warp

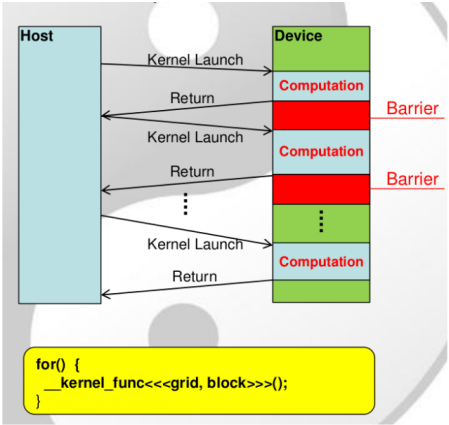
```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
__syncwarp()
```




NVIDIA Tesla V100 GPU Architecture

# Host-side Synchronization

- `cudaDeviceSynchronize()` synchronizes all threads in a grid
  - There are other cheaper variants
- For threads from different grids, writes from a kernel happen before reads from subsequent grid launches



# Scoped Synchronization

- CUDA provides scope qualifiers that limit the subset of the threads that are guaranteed to observe the synchronization
- CUDA exposes three scopes: block, device, and system
- For example, an atomic RMW operation with block scope (e.g., `atomicAdd_block( )`) is only visible to threads within the same thread block

# CUDA Memory Model

- A **memory consistency model** is a set of rules that govern how systems process memory operation requests from multiple processors
  - Determines the order in which memory operations appear to execute
  - Specifies the allowed behaviors of multithreaded programs executing with shared memory
  - Memory models are defined both for hardware and programming languages

- CUDA implements a weakly-ordered memory model
- The order in which a thread writes data is not necessarily the order in which the data is observed being written by another CUDA or host thread
- The behavior on concurrent and conflicting accesses is undefined

# Weakly-Ordered Memory Model in CUDA

```
__device__ int X = 1, Y = 2;
```

## Thread 1

```
__device__ void writeXY() {
  X = 10;
  Y = 20;
}
```

## Thread 2

```
__device__ void readXY() {
  int B = Y;
  int A = X;
}
```

What are the possible values of A and B?

A racy program has **undefined** behavior, and has no defined semantics

# Memory Fences

**Semantics of `__threadfence_block()`**

- All writes made by the calling thread before the call are observed by all threads in the block as occurring before all writes made by the calling thread after the call
- All reads from all memory made by the calling thread before the call are ordered before all reads from all memory made by the calling thread after the call

```
__device__ int X = 1, Y = 2;
```

## Thread 1

```
__device__ void writeXY() {
  X = 10;
  __threadfence();
  Y = 20;
}
```

## Thread 2

```
__device__ void readXY() {
  int B = Y;
  __threadfence();
  int A = X;
}
```

What are the possible values of A and B?

# What can we say about the usage of __threadfence( )?

```
__device__ int X = 1, Y = 2;
```

## Thread 1

```
__device__ void writeXY() {
  X = 10;
  __threadfence();
  Y = 20;
}
```

## Thread 2

```
__device__ void readXY() {
  int B = Y;
  __threadfence();
  int A = X;
}
```

Is using __threadfence( ) always (i) correct and (ii) required?

# `__syncthreads()` vs `__threadfence()`

## `__syncthreads()`

- Establishes **visibility** of memory operations across threads in a block
- Includes a barrier plus memory fence functionality (i.e., `__threadfence_block()`)

## `__threadfence()`

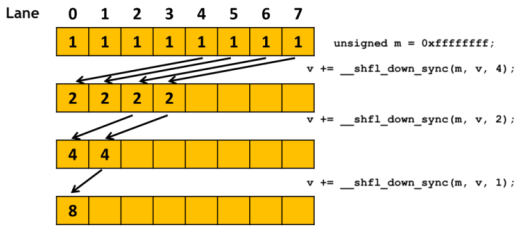
- Ensures **ordering** of memory operations by a thread
- `__threadfence_block()` does not imply a barrier

# Warp Primitives

Efficient intra-warp computation

# Warp-Level Primitives

- `_ballot_sync(mask, predicate)`
  - Evaluate `predicate` for all non-exited threads in `mask` and return an integer whose $N^{th}$ bit is set if `predicate` evaluates to non-zero for the $N^{th}$ thread
- `__activemask()`
  - Returns a 32-bit integer mask of all currently active threads in the calling warp




Copy a variable from a warp lane with higher ID relative to caller

- `__syncwarp(mask)`
  - Guarantees memory ordering among threads participating in the barrier
  - Threads within a warp that wish to communicate via memory can store to memory, execute `__syncwarp()`, and then safely read values stored by other threads

Using CUDA Warp-Level Primitives

# Concurrency and CUDA Streams

Overlap host and device computation with data transfers

# Classic Copy-Then-Execute Model

```
1 cudaMemcpy(d_a, h_a, numBytes, cudaMemcpyHostToDevice);
2 kernel<<<1,N>>>(d_a);
3 cudaMemcpy(h_res, d_a, numBytes, cudaMemcpyDeviceToHost);
```

- Data transfer on line 1 is blocking or synchronous
  - Host thread cannot launch the kernel until the copy is done
- Kernel launch on line 2 is asynchronous
- Data transfer on line 3 cannot begin until the kernel completes due to device-side ordering

How to Overlap Data Transfers in CUDA C/C++

GPU Pro Tip: CUDA 7 Streams Simplify Concurrency

# Overlap Host and Device Computation

```
1 cudaMemcpy(d_a, h_a, numBytes, cudaMemcpyHostToDevice);
2 kernel<<<1,N>>>(d_a);
3 cudaMemcpy(h_res, d_a, numBytes, cudaMemcpyDeviceToHost);
```

```
1 cudaMemcpy(d_a, h_a, numBytes, cudaMemcpyHostToDevice);
2 kernel<<<1,N>>>(d_a);
3 // Host gets work done
4 h_func(h_b);
5 cudaMemcpy(h_res, d_a, numBytes, cudaMemcpyDeviceToHost);
```

# Utilize GPU Hardware

- Overlap kernel execution with memory copy between host and device
- Overlap execution of multiple kernels if there are enough resources
- Recent GPUs support overlapped execution
  - Check the fields `asyncEngineCount`, `concurrentKernels`, and `deviceOverlap` from the `cudaDeviceProp` structure

# CUDA Streams

- A stream is a sequence of operations that execute on the device in the order in which they were issued by the host
  - Operations across streams can interleave and run concurrently
- All GPU device operations run in a stream
- The default "null" stream is used if no custom stream is specified, the default stream is synchronizing
  - No operation in the default stream will begin until all previously issued operations in any stream have completed
  - An operation in the default stream must complete before any other operation in any stream will begin

```
// Both launches are on the default stream
kernel1<<< blocks, threads, bytes >>>();    // default stream
kernel2<<< blocks, threads, bytes, 0 >>>(); // stream 0
```

How to Overlap Data Transfers in CUDA C/C++

# Using a Non-default Stream

Manipulate non-default streams from the host

```
cudaStream_t stream1;
cudaError_t result;
result = cudaStreamCreate(&stream1);
result = cudaStreamDestroy(stream1);
```

Issue a data transfer to a non-default stream

```
result = cudaMemcpyAsync(d_a, a, N, cudaMemcpyHostToDevice, stream1);
```

Specifying a stream during kernel launch is optional

```
kernel1<<<blocks, threads, bytes>>>(); // default/NULL stream
kernel2<<<blocks, threads, bytes, stream1>>>();
```

How to Overlap Data Transfers in CUDA C/C++

# Non-default Streams

- Operations in a non-default stream are non-blocking with the host
- Use `cudaDeviceSynchronize()`
  - Blocks host until all previously issued operations on the device have completed
- Cheaper alternatives
  - `cudaStreamSynchronize()`, `cudaEventSynchronize()`,...

```
cudaStream_t stream1;
cudaError_t res;
res = cudaStreamCreate(&stream1);
res = cudaMemcpyAsync(d_a, a, N, cudaMemcpyHostToDevice, stream1);
increment<<<1,N,0,stream1>>>(d_a);
// Blocks the host thread
cudaStreamSynchronize(stream1);
res = cudaStreamDestroy(&stream1);
```

# Overlapping Kernel Execution and Data Transfers

```
for (int i = 0; i < nStreams; ++i) {
  int offset = i * streamSize;
  cudaMemcpyAsync(&d_a[offset], &h_a[offset], streamBytes,
      cudaMemcpyHostToDevice, stream[i]);
  kernel<<<streamSize/blockSize, blockSize, 0, stream[i]>>>(d_a, offset);
  cudaMemcpyAsync(&h_a[offset], &d_a[offset], streamBytes,
      cudaMemcpyDeviceToHost, stream[i]);
}
```

How to Overlap Data Transfers in CUDA C/C++

# Overlapping Kernel Execution and Data Transfers

```
1  for (int i = 0; i < nStreams; ++i) {
2    int offset = i * streamSize;
3    cudaMemcpyAsync(&d_a[offset], &h_a[offset], streamBytes,
         cudaMemcpyHostToDevice, stream[i]);
4  }
5  for (int i = 0; i < nStreams; ++i) {
6    int offset = i * streamSize;
7    kernel<<<streamSize/blockSize, blockSize, 0, stream[i]>>>(d_a, offset);
8  }
9  for (int i = 0; i < nStreams; ++i) {
10   int offset = i * streamSize;
11   cudaMemcpyAsync(&h_a[offset], &d_a[offset], streamBytes,
         cudaMemcpyDeviceToHost, stream[i]);
12 }
```

# Streams and Concurrency in CUDA 7+

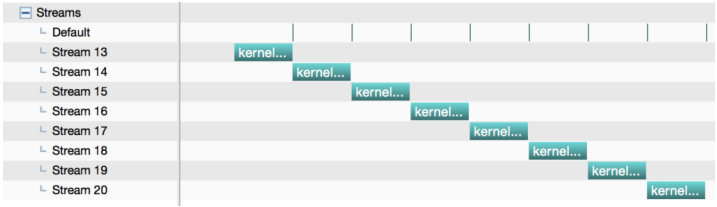
- Prior to CUDA 7, all host threads shared the default stream
  - The default stream implicitly synchronizes with all other streams on the device
  - Two commands from different streams cannot run concurrently if the host thread issues any CUDA command to the default stream between them

- CUDA 7+ provides an option to have a per-host-thread default stream
  - Commands issued to the default stream by different host threads can run concurrently
  - Commands in the default stream may run concurrently with commands in non-default streams
  - Pass the option `–default-stream per-thread` to nvcc to enable per-thread default streams in CUDA 7+

GPU Pro Tip: CUDA 7 Streams Simplify Concurrency

# Multi-Stream Example: Legacy Behavior

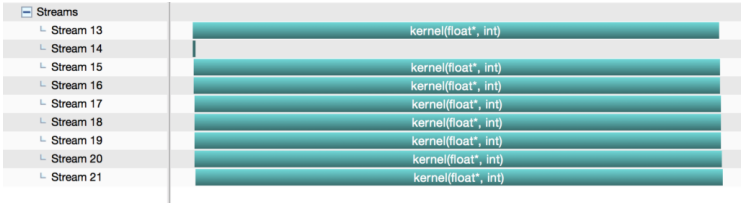
```
for (int i = 0; i < num_streams; i++) {
  cudaStreamCreate(&streams[i]);
  cudaMalloc(&data[i], N * sizeof(float));
  // launch one worker kernel per stream
  kernel<<<1, 64, 0, streams[i]>>>(data[i], N);
  // launch a dummy kernel on the default stream
  kernel<<<1, 1>>>(0, 0);
}
```




GPU Pro Tip: CUDA 7 Streams Simplify Concurrency

# Multi-Stream Example: Per-Thread Default Stream

```
for (int i = 0; i < num_streams; i++) {
  cudaStreamCreate(&streams[i]);
  cudaMalloc(&data[i], N * sizeof(float));
  // launch one worker kernel per stream
  kernel<<<1, 64, 0, streams[i]>>>(data[i], N);
  // launch a dummy kernel on the default stream
  kernel<<<1, 1>>>(0, 0);
}
```




GPU Pro Tip: CUDA 7 Streams Simplify Concurrency

# Achievable Concurrency

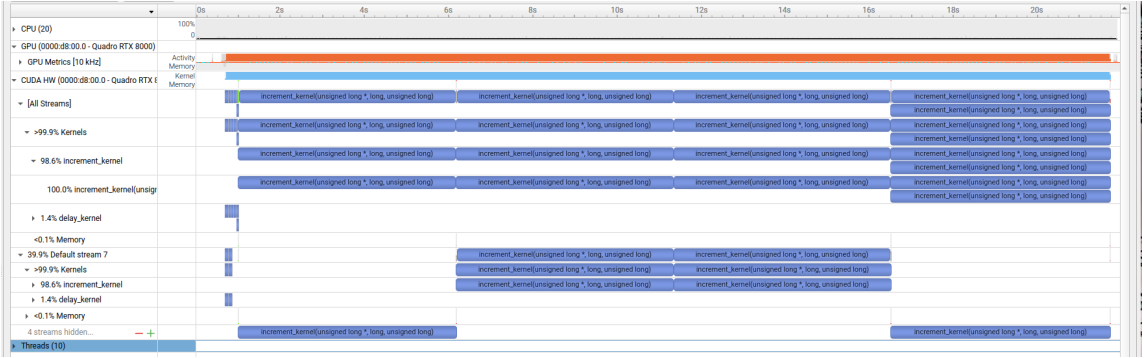
Is it possible to launch two kernels that do independent tasks concurrently?

```
1 // host and device initialization
2 .......
3 // launch kernel1
4 myMethod1 <<<.... >>> (params);
5 // launch kernel2
6 myMethod2 <<<.....>>> (params);
```

- We can launch concurrent kernels in different streams
- There must be resources available while one kernel is running to run concurrent kernels
- The maximum concurrency is 16 kernels on Fermi, 32 on Kepler, and 128 on Turing and Ampere

Compute Capabilities: Features and Technical Specifications

# Concurrent Kernels




concurrent-kernels.cu

# Achievable Concurrency

Can we run two CUDA kernels from two different applications concurrently?

- A kernel from one CUDA context (analogue of host processes for the device) cannot execute concurrently with a kernel from another CUDA context
- The GPU may time slice to provide forward progress to each context
- Must enable Multi-Process Service (MPS) to run kernels from multiple process simultaneously

# References



D. Kirk and W. M. Hwu. Programming Massively Parallel Processors. Chapters 1–5, 13, 20, 3rd edition, Morgan Kaufmann.


NVIDIA Corporation. CUDA C++ Programming Guide, Release 12.9.2.


NVIDIA Corporation. CUDA C++ Best Practices Guide, Release 12.9.